JUMPCODER: Go Beyond Autoregressive Coder via Online Modification

Mouxiang Chen, Hao Tian, Zhongxin Liu[*], Xiaoxue Ren, Jianling Sun
The State Key Laboratory of Blockchain and Data Security,
Zhejiang University
`{chenmx,icouldbe,liu_zx,xxren,sunjl}@zju.edu.cn`

Abstract

While existing code large language models (code LLMs) exhibit impressive capabilities in code generation, their autoregressive sequential generation inherently lacks reversibility. This limitation hinders them from timely correcting previous missing statements during coding as humans do, often leading to error propagation and suboptimal performance. We introduce JUMPCODER, a novel model-agnostic framework that enables human-like online modification and non-sequential generation to augment code LLMs. The key idea behind JUMPCODER is to insert new code into the currently generated code when necessary during generation, which is achieved through an auxiliary infilling model that works in tandem with the code LLM. Since identifying the best infill position beforehand is intractable, we adopt an *infill-first, judge-later* strategy, which experiments with filling at the k most critical positions following the generation of each line, and uses an Abstract Syntax Tree (AST) parser alongside the Generation Model Scoring to effectively judge the validity of each potential infill. Extensive experiments using six state-of-the-art code LLMs across multiple and multilingual benchmarks consistently indicate significant improvements over all baselines. Our code is public at `https://github.com/Keytoyze/JumpCoder`.

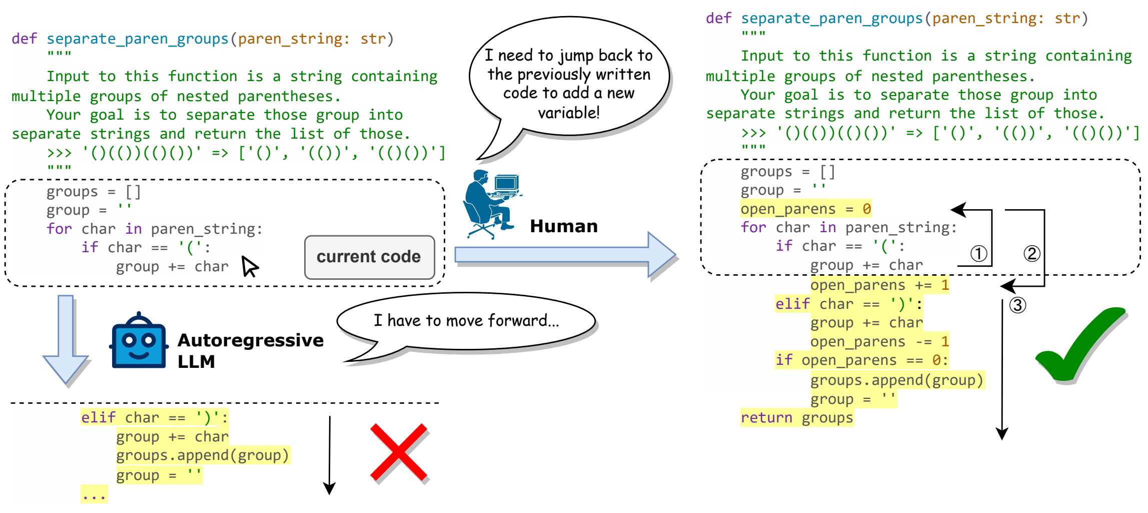
1 Introduction

Recently, Large Language Models (LLMs) (Zhao et al., 2023) have attracted considerable interest and achieved notable successes. As a prominent application of LLMs, numerous code LLMs (Fried et al., 2023; Li et al., 2023; Luo et al., 2023; Rozière et al., 2024; Wang et al., 2023a; Wei et al., 2023; Zheng et al., 2023) are frequently released and exhibit exceptional efficacy in code generation tasks. Despite their remarkable success, current LLMs work in an autoregressive style, generating text segments sequentially from left to right. This leads to an inherent limitation of ***irreversibility*** — they are incapable of revising previously generated text. Notably, even skilled human programmers often struggle to write code in a linear style, since real-world code commonly undergoes iterative editing and refinement (Fried et al., 2023). For example, to accommodate the developing code, programmers need to retrospectively introduce key declarations, such as new variables, references, and functions in the previously written code.

During the early stage of generation, code LLMs often miss key declarations due to the high generation uncertainty (Zheng et al., 2024). Because of the irreversibility limitation, code LLMs, unlike humans, cannot add missing declarations when they need these declarations later. This often results in producing code with undefined identifiers, known as `NameError` in Python. Notably, for the widely-used CODELLAMA (Rozière et al., 2024), `NameErrors` accounted for up to 6.0% to 18.6% of the total error cases on HumanEval in our experiments. Although these `NameErrors` are relatively easy to address by syntax checking or model constraints (Dong et al., 2023), the lack of necessary declarations or statements in the generated code segments often hinders LLMs from continuing to generate semantically correct code, causes the error to accumulate continuously along the generation, and limits the performance of LLMs. Fig. 1 presents an intuitive example, demonstrating that in the absence of necessary variables, LLMs are prone to generating syntactically correct but semantically incorrect code, which is more difficult to detect and rectify.

To address this limitation, the crux is to enable the model to *jump back* to previously generated code for inserting new code like a human whenever a missing declaration or statement is required.

[*]Corresponding author.



```
def separate_paren_groups(paren_string: str)
    """
    Input to this function is a string containing
multiple groups of nested parentheses.
    Your goal is to separate those group into
separate strings and return the list of those.
    >>> '()(())(()())' => ['()', '(())', '(()())']
    """
    groups = []
    group = ''
    for char in paren_string:
        if char == '(':
            group += char
```

```
        elif char == ')':
            group += char
            groups.append(group)
            group = ''
    ...
```

```
def separate_paren_groups(paren_string: str)
    """
    Input to this function is a string containing
multiple groups of nested parentheses.
    Your goal is to separate those group into
separate strings and return the list of those.
    >>> '()(())(()())' => ['()', '(())', '(()())']
    """
    groups = []
    group = ''
    open_parens = 0
    for char in paren_string:
        if char == '(':
            group += char
            open_parens += 1
        elif char == ')':
            group += char
            open_parens -= 1
        if open_parens == 0:
            groups.append(group)
            group = ''
    return groups
```

I need to jump back to the previously written code to add a new variable!
Human
current code
Autoregressive LLM
I have to move forward...
①
②
③

Figure 1: An illustrative example demonstrating the difference between humans and LLMs. When a new variable is required, humans can jump back to the front section to define it, but LLMs, constrained by their autoregressive nature, can only continue generation and lead to error propagation.

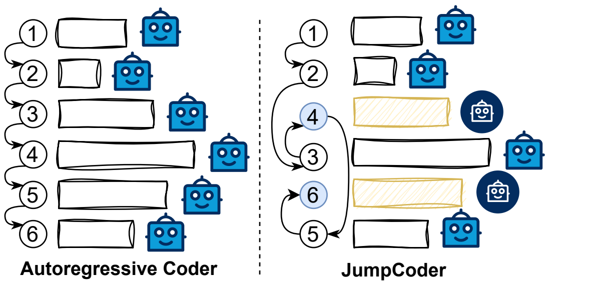


Figure 2: Schematic illustrations of traditional autoregressive coder and the proposed JUMPCODER. Code lines are generated by generation model (🤖) and infilling model (🤖).

Based on it, we propose **JUMPCODER**, a model-agnostic framework to enhance code generation performance. We introduce an innovative hybrid code generation scheme combining two models: the *generation model* drafts the subsequent line of code, while the *infilling model* (Fried et al., 2023) fills a line within the generated code when necessary. The infilling model can retrospectively declare new variables, add new functions, insert additional calculations, and so on, enabling JUMPCODER to achieve online modification and non-sequential generation. Both the generation model and the infilling model are pre-trained and do not require further fine-tuning. Fig. 2 illustrates the comparison between traditional autoregressive coder and our proposed JUMPCODER.

A major challenge to instantiate this idea is deciding whether (and where) to infill, or continue generation from the current code. To overcome this, we propose an *infill-first, judge-later* paradigm: after generating a line, let the infilling model experiment with filling at the start of the k most critical lines that have been generated, and subsequently judge their contributions to the current generation. This paradigm benefits from *parallel generation* and *speculative infilling* optimizations, which avoids a significant reduction in efficiency. The next challenge is how to implement the judgment method. We combine an **Abstract Syntax Tree (AST) parser** and the **Generation Model Scoring** to achieve this. AST parser is employed for the code using undefined identifiers, which accepts the infill that correctly adds the missing declaration. For other scenarios, the generation model scores the code following each infill position by comparing their mean token logit improvements. If an infill enhances the score of subsequent lines, we assume it indicates an improvement in overall generation quality. In cases outside these two scenarios, we infer that infilling is unnecessary, and opt to generate the next line.

Extensive experiments on six variants of CODELLAMA (Rozière et al., 2024) and WIZARDCODER (Luo et al., 2023) on five code generation benchmarks indicated a consistent and significant enhancement across all baselines. Notably, JUMPCODER is adaptable to various programming languages, and helps the code LLMs achieve a pass rate increase of 4.8% - 8.2% in four multilingual HumanEval benchmarks at most, while substantially reducing undefined identifier errors.

To the best of our knowledge, our approach enables online modification for code generation for the first time. We hope our work can inspire further research into the irreversibility limitation of LLMs in the future. Our contributions are threefold:

- We investigate the irreversibility, a significant yet underexplored limitation, of code LLMs.
- We introduce JUMPCODER, a model-agnostic code generation framework for augmenting code LLMs without retraining.
- We perform an extensive evaluation of JUMPCODER on various program languages and code LLMs, demonstrating widespread improvements over current baseline models.

2 Preliminaries

Code generation Code generation, an important task of software engineering, focuses on automatically producing source code from software requirements (Liu et al., 2022). The emergence of LLMs has propelled this field, demonstrating remarkable capabilities in code generation (Rozière et al., 2024; Wei et al., 2023; Luo et al., 2023), especially when fine-tuned on domain-specific code datasets.

Code infilling Conventional language models typically train on left-to-right next token prediction. Although LLMs excel in code generation, they face limitations in code editing tasks like code infilling, where handling bidirectional context is crucial. To overcome this, the **Fill-In-the-Middle** (FIM) training method (Bavarian et al., 2022) has been proposed. FIM divides input code into three parts (`prefix`, `infix`, `suffix`), then reorders them into a sequence `<PRE>` $\oplus$ `prefix` $\oplus$ `<SUF>` $\oplus$ `suffix` $\oplus$ `<MID>` $\oplus$ `infix` $\oplus$ `<EOT>` for training. This trains the model on the conditional distribution $P($`infix` $\mid$ `<PRE>` $\oplus$ `prefix` $\oplus$ `<SUF>` $\oplus$ `suffix`$)$, enhancing its infilling capabilities. However, this approach doesn't directly translate to code generation, where subsequent contexts are absent.

3 JumpCoder

This section elaborates on JUMPCODER, which is designed to address the irreversibility limitation of autoregressive generation. Such limitation often leads to the generation of syntactically or semantically incorrect code. To address it, the key is to facilitate the model's ability to achieve *online modification*, *e.g.*, jump back to the previously generated code to insert a new line like a human, which enables code LLMs to add missing statements. Based on it, we introduce an innovative hybrid generation approach that enables the retrospective insertion of new lines by assisting the generation model with an infilling model, as shown in Fig. 2.

However, identifying the optimal locations for infilling within the current code is a significant challenge. We discovered that while pinpointing the best infill location beforehand is intractable, assessing the suitability of an infill post-insertion is comparatively simpler. Therefore, we propose an *infill-first, judge-later* strategy: after generating a line, the infilling model experiments with inserting a line before each of the k most critical lines that have been generated, followed by judging their impact on the current generation. This judging process is crucial as we found that unnecessary infills tend to degrade code quality (see Appendix C for concise examples).

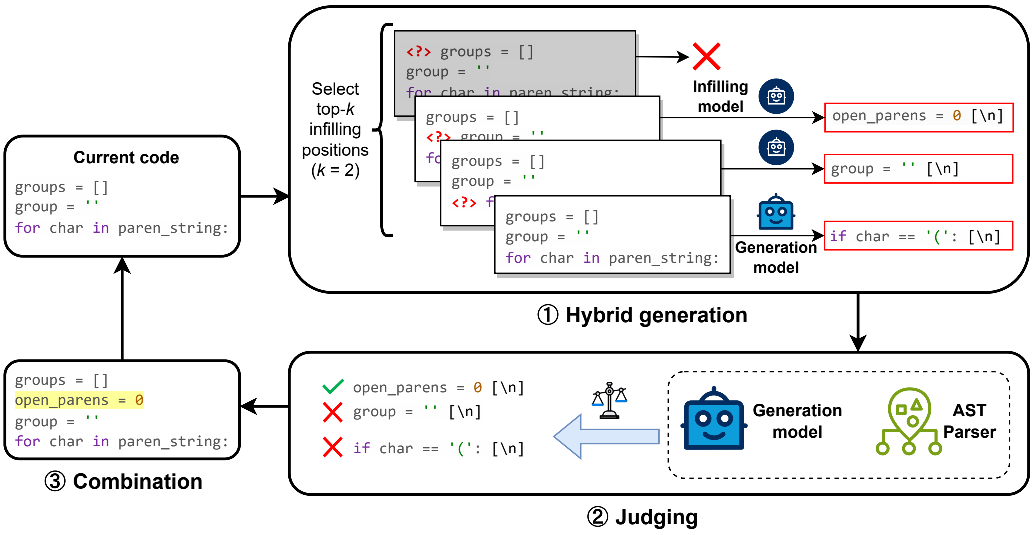
Specifically, JUMPCODER iteratively updates code line by line. Each iteration adds a new line to the current code, which encompasses three steps: **Hybrid generation** (§ 3.1), involving simultaneous line generation by both models; **Judging** (§ 3.2), assessing the infilling model's output to determine line selection; and **Combination** (§ 3.3), merging this selection into the current code before continuing proceeding. The complete framework is outlined in Fig. 3, and the algorithm is summarized in Appendix A.

3.1 Hybrid Generation

In this section, we describe the overview of hybrid generation in § 3.1.1, and introduce optimization techniques for speeding in § 3.1.2.

3.1.1 Overview

We consider the n-th iteration when we have n lines of generated code. Let $(L_1, L_2, \cdots, L_n)$ denote current code, where $L_i = (x^i_1, x^i_2, \cdots, x^i_{|L_i|})$ denotes the i-th line and x^i_j is its j-th token. We generate a new line $\mathbb{L}'_{n+1}$ using a generation model $\mathcal{M}_G$ and infill a line $\mathbb{L}'_i$ before each L_i using an infilling model $\mathcal{M}_I$. It is important to note that a higher value of n significantly increases computational and memory demands for infilling. Consequently, we limit our focus to the top-k positions of *uncertainty* for infilling. This uncertainty is quantified by the logit of the first non-indent token in



```
groups = []
group = ''
for char in paren_string:
```

```
<?> groups = []
group = ''
for char in paren_string:
```

```
groups = []
<?> group = ''
for char in paren_string:
```

```
groups = []
group = ''
<?> for char in paren_string:
```

```
groups = []
group = ''
for char in paren_string:
```

```
open_parens = 0 [\n]
group = '' [\n]
if char == '(': [\n]
```

```
✓ open_parens = 0 [\n]
✗ group = '' [\n]
✗ if char == '(': [\n]
```

```
groups = []
open_parens = 0
group = ''
for char in paren_string:
```

Figure 3: JUMPCODER Framework. The iterative code update process comprises three important stages: Hybrid generation, Judging and Combination. Each iteration inserts a new line of code.

L_i given by the generation model $\mathcal{M}_G$. The underlying principle is that a lower logit for the initial non-indent token suggests a higher potential for varied generations at that position, thereby making it a prime candidate for infilling a new line. We exclude the initial indentation tokens because they are largely predetermined by syntactic rules or coding standards, which lead to elevated logit values and do not effectively reflect uncertainty for these lines. The overall process is formulated as:

$$\mathcal{I} = \arg\underset{i\in\{1,\cdots,n\}}{\text{Top-}k}\, S(x_0^i; \mathcal{M}_G),$$
$$\mathbb{L}_i' \sim P_{\mathcal{M}_I}(\cdot \mid \texttt{<PRE>} \oplus L_{[1:i-1]} \oplus \texttt{<SUF>} \oplus L_{[i:n]}), \quad i \in \mathcal{I},$$
$$\mathbb{L}_{n+1}' \sim P_{\mathcal{M}_G}(\cdot \mid L_{[1:n]}),$$

where x_0^i denotes the first non-indent token in line L_i, and $S(\cdot; \mathcal{M}_G)$ denotes the logit determined by $\mathcal{M}_G$. $P_{\mathcal{M}_I}(\cdot)$ and $P_{\mathcal{M}_G}(\cdot)$ signify sampling a line through infilling and generation respectively.

3.1.2 Efficiency Optimization

Compared to the autoregressive method, our approach introduces an additional infilling model and extra computations, which may potentially increase overhead. Fortunately, these computations can be substantially optimized in practice, as follows:

Parallel Generation Hybrid generation can do in parallel, reducing time complexity from $O(N(k+1))$ to $O(N)$ when parallelization is available.

Speculative Infilling Despite parallel processing, each line's processing time still depends on the slowest of the $k+1$ lines, typically the infill line. We found that JUMPCODER frequently infills the same positions repeatedly on different iterations, often leading to identical infilling results. Based on it, we cache the infill for each position after the generation. When infilling the same position again, we employ the speculative decoding (Chen et al., 2023b; Leviathan et al., 2023) to parallelly verify the cached infill's correctness and resort to standard infilling only if necessary. This strategy improves speed by around 30%.

Despite these optimizations, one limitation is that the hybrid generation imposes extra arithmetic operations and increases memory consumption, particularly when the infilling and generation models differ. Fortunately, the bottleneck in practice often lies in memory bandwidth and communication, not arithmetic operations (Leviathan et al., 2023), enabling the use of extra resources for hybrid generation.

3.2 Judging

Upon acquiring k infilling lines and a generation line, the next step is to judge which one from these $k+1$ lines is the most appropriate for updating the existing n lines. This judging process is accomplished by integrating an **Abstract Syntax Tree (AST) Parser** with the **Generation Model Scoring**.

AST Parser We observed that for the current code using undefined identifiers (*e.g.*, `NameError` in Python), the infilling model typically succeeds in generating the corresponding missing statements. In such cases, we can employ the AST parser to *deterministically* assess the correctness of the infill. The parser is able to analyze the AST of the current code, and identify any usage of undefined identifiers. If an infill $\mathbb{L}'_i$ correctly supplements this missing identifier (such as referencing an external library or defining a new function), we can ascertain the accuracy of $\mathbb{L}'_i$. In this case, we record the score of this infill as infinity, *i.e.*, $V_i = +\infty$, forcing it to be utilized in the combination stage.

Generation Model Scoring While the AST parser can deterministically judge correct infills, it falls short in the cases where infills have latent utility but do not directly address undefined identifiers. For example, as illustrated in Fig. 1, the AST parser cannot discern the infill (`open_parens = 0`) due to the absence of subsequent references to this variable. To address this, a pivotal insight is that a trained LLM is adept at assessing the overall generation quality by computing token scores. We observed that *correct infills usually lead LLMs to assign higher scores to subsequent tokens*, since they improve the overall code quality. Based on this finding, we introduce a scoring-based approach to determine the potential impact of infills.

Specifically, for the infilling line $\mathbb{L}'_i$ preceding the i-th line, we calculate the average token logit improvement of subsequent lines before and after combining $\mathbb{L}'_i$, namely:

$$\Delta_k = \frac{1}{|L_k|}\sum_{j=1}^{|L_k|} \mathbb{S}'(x_j^k; \mathcal{M}_G) - S(x_j^k; \mathcal{M}_G),$$

where $k \in \{i, \cdots, n\}$. $\mathbb{S}'(\cdot)$ and $S(\cdot)$ denotes the logit score determined by $\mathcal{M}_G$. $S(\cdot)$ is computed based on the initial code $[L_1 \oplus \cdots \oplus L_n]$, and $\mathbb{S}'(\cdot)$ is computed based on the combined code $[L_1 \oplus \cdots \oplus L_{i-1} \oplus \mathbb{L}'_i \oplus L_i \oplus \cdots \oplus L_n]$. Δ_k represents the improvement of line L_k after combining $\mathbb{L}'_i$.

Next, we identify all consecutive lines after $\mathbb{L}'_i$ which exhibit notable improvement. We determine a t (where $i \leq t \leq n$) such that the improvement for subsequent lines $\Delta_i, \Delta_{i+1}, \cdots, \Delta_t$ exceeds a pre-defined threshold τ, and the improvement Δ_{t+1} for L_{t+1} (if $n > t$) falls below τ. If these improved lines are not more than half of all subsequent lines, *i.e.*, $t-i+1 \leq {}^{(n-i+1)}/_2$, the infill is not considered beneficial for most of the subsequent lines and is thus disregarded. Otherwise, the total improvement V_i from L_i to L_t is recorded as the score of this infill, formulated as $V_i = \sum_{k=i}^{t} \Delta_k$. Additionally, if $n > t$, we opt to *remove* the extra lines after line t when $\mathbb{L}'_i$ is combined, as they signify diminished improvement and necessitate revision.

Discussion Compared to AST Parser, Generation Model Scoring has a broader application scope. However, its judgments are not infallible and may occasionally result in errors. Tuning the improvement threshold τ can significantly reduce such misjudgments and increase the precision. We present examples of good and bad infills judged by JUMPCODER in Appendix C to clearly illustrate the role of Judging.

3.3 Combination

Following the Judging stage, we filter out infills without significant improvement and record the score V_i of each infill $\mathbb{L}'_i$. During the Combination stage, if all infills are filtered out, it indicates that no additional infill is necessary at the moment, and thus we adopt the standard generation line $\mathbb{L}'_{n+1}$. Otherwise, we adopt the infill with the highest score.

4 Experiments

4.1 Setup

Models and Benchmarks We tested two open-source code LLMs: CODELLAMA (Rozière et al., 2024) and WIZARDCODER-PYTHON (Luo et al., 2023), with the latter being one of the SOTA models. CODELLAMA was tested across 7B and 13B versions using the PYTHON and INSTRUCT variants, due to their superior performance in code generation compared to the base model. For WIZARDCODER-PYTHON, we utilized the 13B and 34B versions. The infilling model used in our JUMPCODER is CODELLAMA-INSTRUCT-7B by default, which has been pre-trained with the FIM objective (Bavarian et al., 2022).

We evaluated our approach across a range of widely-used code generation benchmarks, including HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). To showcase the adaptability of our method across different programming languages, we also evaluate it on MultiPL-E (Cassano et al., 2022). For HumanEval and MBPP, we employed the prompts provided by Ben Allal et al. (2022) for CODELLAMA's evaluation and modified them to fit the instruction format of WIZARDCODER (Luo et al., 2023). For MultiPL-E,

Table 1: Results of Pass@1 (%) on HumanEval and MBPP using greedy generation. Following Ben Allal et al. (2022), we use 0-shot for HumanEval and 1-shot for MBPP in our reproduced experiments. JC = JUMPCODER. V = Vanilla. F = Filtered. O = Oracle. †: Results are taken from Rozière et al. (2024).

Generation model	Method	HumanEval	MBPP
GPT-3.5†		48.1	52.2
GPT-4†		67.0	-
STARCODER† (15B)		33.6	52.7
CODELLAMA† (7B)		33.5	41.4
CODELLAMA† (13B)		36.0	47.0
CODELLAMA† (34B)		48.8	55.0
CODELLAMA -INSTRUCT (7B)	-	36.0	42.4
	+ JC (V)	37.8 (+1.8)	44.8 (+2.4)
	+ JC (F)	**39.6 (+3.6)**	**45.2 (+2.8)**
	+ JC (O)	*39.6 (+3.6)*	*45.2 (+2.8)*
CODELLAMA -PYTHON (7B)	-	38.4	43.2
	+ JC (V)	40.2 (+1.8)	45.4 (+2.2)
	+ JC (F)	**41.5 (+3.1)**	**45.6 (+2.4)**
	+ JC (O)	*41.5 (+3.1)*	*46.8 (+3.6)*
CODELLAMA -INSTRUCT (13B)	-	40.9	45.8
	+ JC (V)	**44.5 (+3.6)**	**46.8 (+1.0)**
	+ JC (F)	43.9 (+3.0)	46.6 (+0.8)
	+ JC (O)	*45.7 (+4.8)*	*48.0 (+2.2)*
CODELLAMA -PYTHON (13B)	-	43.9	50.0
	+ JC (V)	**45.7 (+1.8)**	**51.0 (+1.0)**
	+ JC (F)	**45.7 (+1.8)**	50.8 (+0.8)
	+ JC (O)	*47.0 (+3.1)*	*53.2 (+3.2)*
WIZARDCODER -PYTHON (13B)	-	64.0	56.8
	+ JC (V)	64.6 (+0.6)	**57.2 (+0.4)**
	+ JC (F)	**65.2 (+1.2)**	**57.2 (+0.4)**
	+ JC (O)	*65.9 (+1.9)*	*57.2 (+0.4)*
WIZARDCODER -PYTHON (34B)	-	73.8	59.2
	+ JC (V)	**74.4 (+0.6)**	59.2 (+0.0)
	+ JC (F)	**74.4 (+0.6)**	**59.6 (+0.4)**
	+ JC (O)	*75.0 (+1.2)*	*60.0 (+0.8)*

following Wei et al. (2023) we directly used the provided completion formats.

Implementation Details In alignment with previous studies (Chen et al., 2023c; Wei et al., 2023), we applied greedy decoding for both generation and infilling in each task, and reported the Pass@1 metric. Besides, we found that JUMPCODER may degenerate the code in a few cases when the autoregressive generation is already correct. To further investigate these degenerated cases, we design two strategies for selecting one code from the codes generated by JUMPCODER (denoted as A) and traditional autoregressive coder (denoted as B): (1) **JUMPCODER (filtered)**, in which if all infills are judged by AST Parser instead of Generation Model Scoring, A is chosen, otherwise the code with lower perplexity (PPL) prevails; and (2) **JUMPCODER (oracle)**, comparing A and B with the test cases used for evaluation and selecting the better one, serving as the performance upper bound. We provide further implementation details in Appendix B.1.

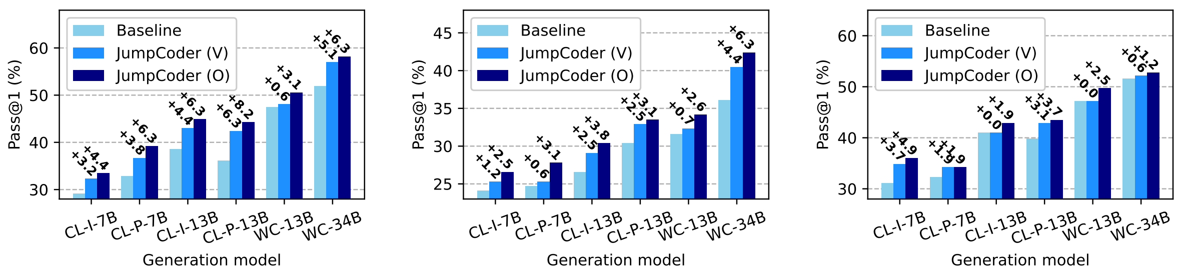
4.2 Results on HumanEval and MBPP

Table 1 summarizes results of Pass@1 on HumanEval and MBPP across various generation models. We can observe that JUMPCODER consistently enhances performance across both benchmarks compared to the base models. Specifically,

- JUMPCODER (Vanilla) shows a notable improvement over the baselines. It indicates that JUMPCODER can effectively improve existing code LLMs by addressing the irreversibility limitation.
- JUMPCODER's improvements are more pronounced with weaker baselines (CODELLAMA) than with the stronger ones (WIZARDCODER), likely due to the former's challenges in establishing necessary foundations, *e.g.*, predefined variables and references, for subsequent code (see Appendix C for concise examples). However, as discussed in the following section, WIZARDCODER's capability for addressing these challenges does not stand out in specific languages.
- JUMPCODER (Vanilla) does not achieve the optimal potential, as indicated by JUMPCODER (Oracle) which is the performance upper bound, suggesting the presence of certain suboptimal infills. Such cases result in a decline from the original autoregressive coder. This issue is partially addressed by JUMPCODER (Filtered), which consistently outperforms JUMPCODER (Vanilla) and even reaches the performance of JUMPCODER (Oracle) in several scenarios.

4.3 Results on MultiPL-E

Fig. 4 illustrates the improvements achieved by JUMPCODER in Java, C# and C++ using the MultiPL-E (Cassano et al., 2022) benchmark. The findings reveal consistent enhancements across these programming languages. On average, JUMPCODER passes an additional 2.9% (Python), 5.8% (Java), 3.6% (C#) and 2.7% (C++) problems. Remarkably, while JUMPCODER's advancements over WIZARDCODER are less marked in Python, considerable gains are evident in Java and C#. This implies the impact of irreversibility limitation may be language-dependent: certain languages pose more challenges for models not only in preemptively generating essential declarations or state-



(a) Java (b) C# (c) C++

Figure 4: Results on MultiPL-E. CL-I/P = CODELLAMA-INSTRUCT/PYTHON. WC = WIZARDCODER-PYTHON.

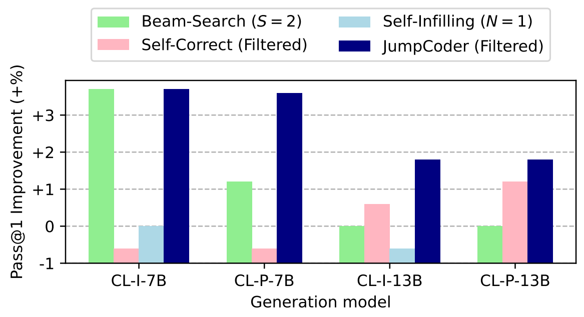


Figure 5: Performance improvements of various baselines over greedy decoding on HumanEval.

ments but also in producing suitable subsequent codes to compensate for previous omissions.

4.4 Results Compared with Other Baselines

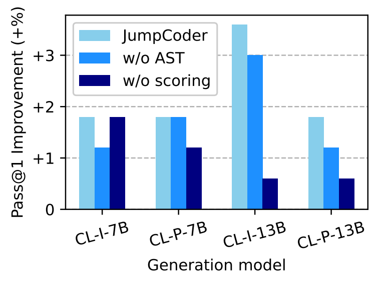
In addition, we compared JUMPCODER with three similar baselines: SELF-CORRECT (Huang et al., 2023; Madaan et al., 2023; Ganguli et al., 2023), BEAM-SEARCH (Wiseman and Rush, 2016) and SELF-INFILLING (Zheng et al., 2024). Details on the baselines and the evaluation protocol are provided in Appendix B.2. Fig. 5 presents the evaluation results, which reveals JUMPCODER's consistent superiority. Notably, BEAM-SEARCH outperforms GREEDY-DECODING only with the 7B models, suggesting the 13B model's ability to generate correct code on the first try. In contrast, SELF-CORRECT underperforms GREEDY-DECODING with the 7B model, likely due to the limited capability of the small-size model in comprehending complicated instructions effectively. We also observed that SELF-INFILLING did not perform notably well, likely due to two factors: (1) The infilling capability of the instruct models may have been weakened due to the instruction fine-tuning (Zheng et al., 2024); (2) The effectiveness of SELF-INFILLING largely depends on rectifying empty code outputs (*e.g.*, outputting placeholder statements like "pass"), an effect that is less evident in our setting where all the generated code has substantial content (refer to Appendix B.2).

5 Analysis

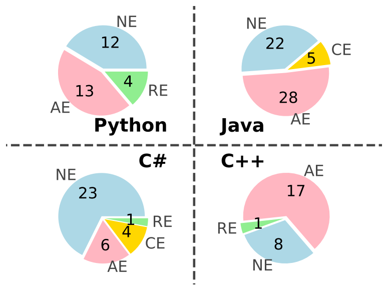
Ablation Studies we evaluated JUMPCODER (Vanilla) by excluding the AST Parser (labeled as w/o AST) and Generation Model Scoring (labeled as w/o scoring). Fig. 6(a) illustrates the performance improvements of these variants compared to the complete JUMPCODER on various generation models. The results indicate both variants were less effective than the full JUMPCODER. This is attributed to the complementary roles of the AST Parser and Generation Model Scoring: the AST Parser deterministically accepts the infills that resolve undefined identifier errors, which can be probabilistically rejected by Generation Model Scoring. Conversely, Generation Model Scoring is able to accept more good infills. This synergy allows more effective infills, highlighting the enhanced performance achieved by integrating both Judging components in JUMPCODER.

Different Types of Errors Addressed Fig. 6(b) illustrates different error types addressed by JUMPCODER. Notably, the enhancements brought by JUMPCODER stem not only from supplementing missing declarations in the generated code (*i.e.*, address undefined identifier errors), but also from inserting essential statements that facilitate the correctness (*i.e.*, address other errors). We provide detailed examples illustrating how JUMPCODER tackles different error types in Appendix C.

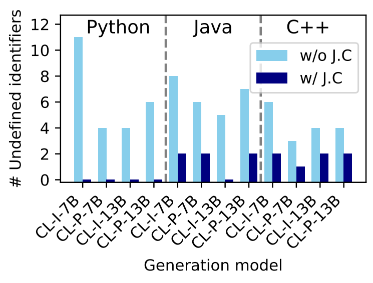
Addressing Undefined Identifier Errors Fig. 6(c) demonstrates the reduction in undefined identifier errors across various languages with JUMPCODER's application. Notably, it eradicates



(a) Ablation study

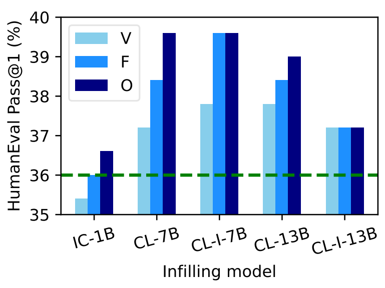


(b) Types of errors addressed

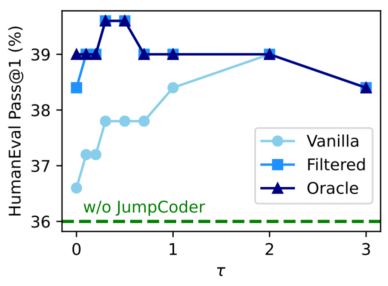


(c) Number of undefined identifiers

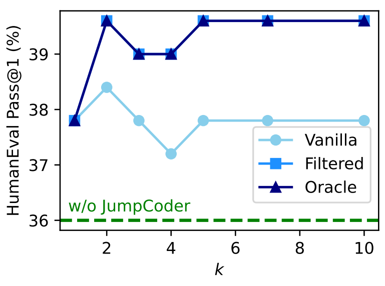
Figure 6: **(a)** Ablation study of removing AST Parser and Generation Model Scoring during the judging stage. **(b)** Different types of errors addressed by JUMPCODER. NE = Name Error (*i.e.*, undefined identifier error). AE = Assertion Error (*i.e.*, failed at test cases). RE = Runtime Error. CE = Compile Error. **(c)** The number of undefined identifier errors for different languages. CL-I/P = CODELLAMA-INSTRUCT/PYTHON.



(a) Infilling model



(b) Improvement threshold τ



(c) Infill at top-k positions

Figure 7: Impact of infilling models and hyperparameters on three variants of JUMPCODER on HumanEval. The generation model is CODELLAMA-INSTRUCT-7B, whose performance is marked in the green dashed line. IC = INCODER. CL = CODELLAMA. CL-I = CODELLAMA-INSTRUCT.

such errors in Python and significantly decreases them in C++ and Java. This demonstrates that allowing the model to jump back and re-encode previously generated parts enables undefined identifiers to be defined, confirming JUMPCODER's effectiveness. We also noticed that C++ and Java errors are not completely resolved due to the infilling model's occasional inability to correctly fill in missing library applications and functions. Addressing this issue further is left for future work.

Infilling Model, Parameter and Efficiency Analysis We adjusted various infilling models and two hyperparameters (improvement threshold τ and number of infills k) to examine their impact on the model performance. We first evaluated INCODER (Fried et al., 2023), CODELLAMA (Rozière et al., 2024) and their instruction variants as the infilling models, all of which pre-trained with the FIM objective (Bavarian et al., 2022). Results in Fig. 7(a) show that CODELLAMA-7B outperforms even the larger 13B model, suggesting smaller models can effectively power JUMPCODER. We also found that INCODER underperformed due to inadequate function and library completion capabilities. Additionally, we observed that the CODELLAMA-13B-INSTRUCT frequently generated statements with incorrect indentation. One possible explanation is that the instruct fine-tuning diminishes its infilling proficiency.

Fig. 7(b) shows the results when tuning the improvement threshold τ. It indicates that a low hyperparameter τ (< 0.3) encourages JUMPCODER to adopt *aggressive* infilling, which leads to the integration of numerous inferior infills, reducing overall performance. When τ is high (> 2), JUMPCODER's *conservative* infill selection results in the exclusion of potentially beneficial infills, marginally diminishing performance. An intermediary τ enables the model to achieve an optimal trade-off between good and bad infills.

Fig. 7(c) shows the results on adjusting the maximum infill positions k. It suggests that a small k limits JUMPCODER's infill options, inadvertently filtering out advantageous infills and thereby decreasing performance. As k increases, JUMP-

CODER's performance reaches a plateau. Interestingly, JUMPCODER (Vanilla) exhibits an initial performance dip with an increase in k, likely due to the model selecting more suboptimal infills. Fortunately, JUMPCODER (Filtered) can effectively filter out these less desirable infills.

Additionally, we provide an in-depth discussion on generation speed and memory analysis in Appendix B.3, which reveals that JUMPCODER's generation speed is $0.7\times$ that of autoregressive generation, maintaining a comparable order of magnitude.

6 Related Work

In this section, we review two lines of related work: code generation and code infilling.

Code Generation Training large language models with an extensive corpus of code is a common approach for code generation tasks (Zan et al., 2023). There are several large language models trained on massive code data, such as Codex (Chen et al., 2021), CodeGen (Nijkamp et al., 2023b), AlphaCode (Li et al., 2022), CodeGeeX (Zheng et al., 2023), StarCoder (Li et al., 2023) and Code Llama (Rozière et al., 2024). These code generation models work in an autoregressive manner, which results in their inability to do online modification. Some studies focus on improving the work of autoregressive code generation models. In self-debugging (Chen et al., 2023c), the model generates multiple iterations of code based on the explanation and execution results of the initially generated code. In contrast, our approach enables the model for online modification during a single round. Another concurrent study, self-infilling (Zheng et al., 2024), explores enhancing autoregressive code generation through infilling capabilities. Our approach differs from self-infilling in three key aspects: Firstly, self-infilling requires the generation model to possess infilling abilities and mandates specific suffix formats for different languages, which limits its applicability. Secondly, our method allows for online modifications, aligning more closely with human coding practices, whereas self-infilling merely alters the order of generation. Lastly, the two methods are somewhat complementary: certain self-infilling features, such as loops, can be integrated into our method, and our capabilities for online modification could also enhance their infilling or generation processes.

Several research targets improving LLM-based code generation (Chen et al., 2023a; Dong et al., 2023; Le et al., 2022; Fried et al., 2023; Shen et al., 2023). CodeT (Chen et al., 2023a) utilizes one language model to generate both code snippets and test cases. Subsequently, it ranks the code snippets by assessing the consistency between the code snippets and the associated test cases. CODEP (Dong et al., 2023) enhances the existing code generation framework by incorporating an automatic pushdown automaton (PDA) module to ensure the syntactic correctness of generated code. Recently, many code generation approaches finetuned code LLMs with high-quality instruction finetuning datasets (Luo et al., 2023; Wei et al., 2023; Gunasekar et al., 2023; Muennighoff et al., 2023). Our approach also harmonizes with these models and can be used to enhance their performance.

Code Infilling Various code generation models possess the capability of code infilling. While excelling in code generation, decoder-only architecture code generation models face limitations in understanding context due to their unidirectional attention mechanism. In response to this architectural limitation, these models employ methodologies akin to the FIM (Bavarian et al., 2022) concept. The model learns to fill these regions in a standard left-to-right autoregressive manner, acquiring infilling capabilities. This approach has been used to pre-train on various code generation models, including Incoder (Fried et al., 2023), StarCoder (Li et al., 2023), CodeGen 2 (Nijkamp et al., 2023a) and CodeLlama (Rozière et al., 2024), and has boosted downstream tasks such as code completion and document generation. In this work, we leverage these code-infilling capabilities to enable online modification and enhance code generation.

7 Conclusion

This research analyzes the inherent irreversibility limitation of the autoregressive sequential generation, a critical yet underexplored issue. We introduce JUMPCODER, a framework that enables human-like online code modification for existing LLMs through an infilling model without retraining. This framework is readily adaptable to diverse programming languages. Through extensive experimentation, we demonstrate that our proposed JUMPCODER significantly enhances code generation quality across various code LLMs and benchmarks. We hope our work can motivate more future investigations into this limitation of LLMs.

Limitations

There are some worthwhile directions for future research to address the limitations in this paper, which we list below:

- Further optimizing the time/memory efficiency of JUMPCODER. In our framework, we employ an *infill-first, judge-later* strategy. While parallel generation and speculative infilling are utilized to expedite the process, it remains comparatively slower than traditional generation methods and increases memory consumption. Future research could focus on exploring alternative paradigms for online modification during generation, such as devising a component to preemptively identify the most effective infill positions.

- Refine the judging process for infills. While the current judging approach proves effective in our experiments, it depends on heuristic rules and may sometimes endorse suboptimal infills. Developing a more generalized judging framework to enhance the synergy between generation and infilling models presents a compelling research avenue.

- Generalize JUMPCODER's capabilities and scope. One limitation of JUMPCODER is that it can only add new code within the current code. Broadening the online modification to encompass online deletions or changes offers a compelling expansion path. Additionally, the challenge of irreversibility is intrinsic to autoregressive generation across various applications, not exclusively to code generation. Despite being tailored for code generation, the promising results from our framework hint at the broader utility for tasks such as mathematical problem-solving and natural language generation.

Ethics Statement

Our work complies with the ACL Ethics Policy. All datasets and models are publicly accessible. We have not identified any significant ethical considerations associated with our work. We believe our findings can inspire further research into the irreversibility limitation of LLMs.

Acknowledgments

This research is supported by the National Natural Science Foundation of China (No. 62202420) and the Software Engineering Application Technology Lab at Huawei under the Contract TC20231108060. Zhongxin Liu gratefully acknowledges the support of Zhejiang University Education Foundation Qizhen Scholar Foundation.

References

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

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *CoRR*, abs/2207.14255.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. 2022. A framework for the evaluation of code generation models. `https://github.com/bigcode-project/bigcode-evaluation-harness`.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023a. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023b. Accelerating large language model decoding with speculative sampling.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde 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, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023c. Teaching large language models to self-debug.

Yihong Dong, Ge Li, and Zhi Jin. 2023. Codep: Grammatical seq2seq model for general-purpose code generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 188–198, New York, NY, USA. Association for Computing Machinery.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Deep Ganguli, Amanda Askell, Nicholas Schiefer, Thomas I. Liao, Kamilė Lukošiūtė, Anna Chen, Anna Goldie, Azalia Mirhoseini, Catherine Olsson, Danny Hernandez, Dawn Drain, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jackson Kernion, Jamie Kerr, Jared Mueller, Joshua Landau, Kamal Ndousse, Karina Nguyen, Liane Lovitt, Michael Sellitto, Nelson Elhage, Noemi Mercado, Nova DasSarma, Oliver Rausch, Robert Lasenby, Robin Larson, Sam Ringer, Sandipan Kundu, Saurav Kadavath, Scott Johnston, Shauna Kravec, Sheer El Showk, Tamera Lanham, Timothy Telleen-Lawton, Tom Henighan, Tristan Hume, Yuntao Bai, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, Christopher Olah, Jack Clark, Samuel R. Bowman, and Jared Kaplan. 2023. The capacity for moral self-correction in large language models.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks are all you need.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large language models cannot self-correct reasoning yet.

Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. 2019. Ctrl: A conditional transformer language model for controllable generation.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pages 21314–21328. Curran Associates, Inc.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 19274–19286. PMLR.

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, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you!

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. 2022. Deep learning based program generation from requirements text: Are we there yet? *IEEE Transactions on Software Engineering*, 48(4):1268–1289.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam

Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. Codegen2: Lessons for training llms on programming and natural languages.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023b. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

John W Ratcliff, David Metzener, et al. 1988. Pattern matching: The gestalt approach. *Dr. Dobb's Journal*, 13(7):46.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023a. Codet5+: Open code large language models for code understanding and generation.

Zejun Wang, Jia Li, Ge Li, and Zhi Jin. 2023b. Chatcoder: Chat-based refine requirement improves llms' code generation.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need.

Sam Wiseman and Alexander M. Rush. 2016. Sequence-to-sequence learning as beam-search optimization. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1296–1306, Austin, Texas. Association for Computational Linguistics.

Jin Xu, Xiaojiang Liu, Jianhao Yan, Deng Cai, Huayang Li, and Jian Li. 2022. Learning to break the loop: Analyzing and mitigating repetitions for neural text generation. In *Advances in Neural Information Processing Systems*, volume 35, pages 3082–3095. Curran Associates, Inc.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, Toronto, Canada. Association for Computational Linguistics.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models.

Lin Zheng, Jianbo Yuan, Zhi Zhang, Hongxia Yang, and Lingpeng Kong. 2024. Self-infilling code generation.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models.

Appendix

A Algorithm

In this section, we present pseudocode diagrams of our method to facilitate a clearer understanding of the operational workflow. Algorithm 1 involves generating the next line of code or filling in the Top-k critical lines in the current code based on the available code. Algorithm 2 assesses the validity of infills. Algorithm 3 selects the best infill based on the scores. Finally, Algorithm 4 outlines the workflow of our entire framework.

Algorithm 1: Hybrid generation Algorithm of JumpCoder

```
Input  : Lines of current code $L$, generation model $\mathcal{M}_G$, infilling model $\mathcal{M}_I$,
         hyperparameters $k$.
Output : Generated / infilled lines in each position $\mathbb{L}'$, infill positions $\mathcal{I}$.
1  $n \leftarrow |L|$;
2  $\mathbb{L}' = \varnothing$;
   ▷ Evaluate the first non-indent token score for each line.
3  for $i \leftarrow 1, \cdots, n$ parallel do
4      $S_i \leftarrow S(x_0^i; \mathcal{M}_G)$;
5  end
   ▷ Get the critical positions based on top-$k$ lowest $S_i$.
6  $\mathcal{I} \leftarrow \arg \text{Top-}k\ S_i$;
   ▷ Infilling.
7  for $i \in \mathcal{I}$ parallel do
8      $\mathbb{L}'_i \sim P_{\mathcal{M}_I}(\cdot \mid$ <PRE> $\oplus L_{[1:i-1]} \oplus$ <SUF> $\oplus L_{[i:n]})$;
9      $\mathbb{L}' \leftarrow \mathbb{L}' \cup \{\mathbb{L}'_i\}$;
10 end
   ▷ Generation.
11 $\mathbb{L}'_{n+1} \sim P_{\mathcal{M}_G}(\cdot \mid L_{[1:n]})$;
12 $\mathbb{L}' \leftarrow \mathbb{L}' \cup \{\mathbb{L}'_{n+1}\}$;
13 return $\mathbb{L}', \mathcal{I}$
```

Algorithm 2: Judging Algorithm of JumpCoder

```
Input  : Lines of current code $L$, generated / infilled lines in each position $\mathbb{L}'$,
         generation model $\mathcal{M}_G$, hyperparameters $\tau$, infill positions $\mathcal{I}$.
Output : Set of infill scores $\mathbb{V}$ and the corresponding updated codes $\mathbb{L}^{\text{next}}$.
1  $n \leftarrow |L|$;
2  $\mathbb{L}^{\text{next}} = \varnothing$;
3  $\mathbb{V} = \varnothing$;
4  for $i \in \mathcal{I}$ do
       ▷ Judge $\mathbb{L}'_i$ by AST Parser.
5      if undefined identifier error is resolved then
           ▷ We directly assign the highest score to this infill to force it to be accepted.
6          $V_i \leftarrow +\infty$;
7          $L_i^{\text{next}} \leftarrow L_1 \oplus L_2 \oplus \cdots \oplus L_{i-1} \oplus \mathbb{L}'_i \oplus L_i \oplus \cdots \oplus L_n$;
8          $\mathbb{V} \leftarrow \mathbb{V} \cup \{V_i\}$;
9          $\mathbb{L}^{\text{next}} \leftarrow \mathbb{L}^{\text{next}} \cup \{L_i^{\text{next}}\}$;
10         continue;
11     end
       ▷ Judge $\mathbb{L}'_i$ by Generation Model Scoring.
       ▷ Find the last line ($t$) that have significant improvements.
12     $k \leftarrow i - 1, \quad t \leftarrow i - 1$;
13     repeat
14         $k \leftarrow k + 1$;
15         $\Delta_k \leftarrow \frac{1}{|L_k|} \sum_{j=1}^{|L_k|} \mathbb{S}'(x_j^k; \mathcal{M}_G) - S(x_j^k; \mathcal{M}_G)$;
16         if $\Delta_k > \tau$ then
17             $t \leftarrow k$;
18         end
19     until $k + 1 > n$ or $\Delta_k \leq \tau$;
20     if $t - i + 1 \leq {}^{(n-i+1)}/_{2}$ then
           ▷ $\mathbb{L}'_i$ is not beneficial for more than half subsequent lines.
           ▷ Ignore it.
21         continue;
22     end
       ▷ Accept $\mathbb{L}'_i$ and record the average improvement.
23     $V_i \leftarrow \sum_{k=i}^{t} \Delta_k$;
       ▷ Combine $\mathbb{L}'_i$ and remove the lines without improvement.
24     $L_i^{\text{next}} \leftarrow L_1 \oplus L_2 \oplus \cdots \oplus L_{i-1} \oplus \mathbb{L}'_i \oplus L_i \oplus \cdots \oplus L_t$;
25     $\mathbb{V} \leftarrow \mathbb{V} \cup \{V_i\}$;
26     $\mathbb{L}^{\text{next}} \leftarrow \mathbb{L}^{\text{next}} \cup \{L_i^{\text{next}}\}$;
27 end
28 return $\mathbb{V}, \mathbb{L}^{next}$
```

B Further Experimental Details and Results

B.1 Implementation Details of JumpCoder

Computation Sources We conducted all the experiments in eight NVIDIA A800 GPUs for around 100 GPU hours. All models are loaded in the `float16` format.

Hyperparameters Regarding our method's hyperparameters k and τ, by default we set $k = 5$ for Python and $k = 10$ for C++ and Java tasks. τ is set at 0.8 for C++, and 0.3 for other languages.

AST Parsers The AST Parser is employed during the Judging stage to detect undefined identifiers in code snippets. For Python, we utilize the built-in `ast` module to analyze the AST of the current code. For C#, we leverage `tree-sitter`[1] to parse the AST. For Java and C++, we invoke their respective compilers to compile the code and analyze error logs to determine the presence of undefined identi-

[1] https://github.com/tree-sitter/tree-sitter-c-sharp

Algorithm 3: Combination Algorithm of JUMPCODER

```
Input  : Lines of current code $L$, set of infill scores $\mathbb{V}$ and the corresponding updated codes $\mathbb{L}^{\text{next}}$, generated line $\mathbb{L}'_{n+1}$.
Output : Next iteration of code.
1 $n \leftarrow |L|$;
2 if $\mathbb{V} = \varnothing$ then
      ▷ Good infill doesn't exist. We accept the line generated by generation model $\mathcal{M}_G$
3     return $L_1 \oplus L_2 \oplus \cdots \oplus L_n \oplus \mathbb{L}'_{n+1}$;
4 else
      ▷ Find the infill with the highest value.
5     $i^* \leftarrow \arg\max \mathbb{V}$;
6     return $\mathbb{L}^{\textit{next}}_{i^*}$;
7 end
```

Algorithm 4: JUMPCODER

```
Input  : Input prompt $\mathcal{P}$, generation model $\mathcal{M}_G$, infilling model $\mathcal{M}_I$, hyperparameters $\tau$ and $k$.
Output : Lines of generated code $L$.
1 $L \leftarrow \mathcal{P}$;
2 while the stopping criteria is not met do
      ▷ Stage 1. Hybrid generation
3     Invoke hybrid generation (Algorithm 1) with input $L, \mathcal{M}_G, \mathcal{M}_I, k$;
4     Obtain generated / infilled lines in each position $\mathbb{L}'$ and infill positions $\mathcal{I}$;
      ▷ Stage 2. Judging
5     Invoke Judging (Algorithm 2) with input $L, \mathbb{L}', \mathcal{M}_G, \tau, \mathcal{I}$;
6     Obtain set of infill scores $\mathbb{V}$ and the corresponding updated codes $\mathbb{L}^{\text{next}}$;
      ▷ Stage 3. Combination
7     Invoke Combination (Algorithm 3) with input $\mathbb{V}, \mathbb{L}^{\text{next}}, \mathbb{L}'$ and update $L$;
8 end
9 return $L$
```

fiers. Specifically, we identify undefined identifiers by checking if the compilation error logs contain `"cannot find symbol"` (for Java) and `"was not declared in this scope"` (for C++). To prevent syntax errors such as mismatched braces during incomplete generation, we ensured the balancing of braces first before invoking the compiler.

Tokenization Tokenization in infilling models is challenging, as infilling at certain positions can disrupt token boundaries, leading to irregular tokens and degraded performance (Zheng et al., 2024). JUMPCODER's infilling, strategically placed at the beginning of lines, aligns with the tokenization approaches of the infilling models we tested (CODELLAMA (Rozière et al., 2024) and INCODER (Fried et al., 2023)), which treat newline characters as distinct tokens (`[\n]`). This placement minimizes the incidence of irregular tokens. However, it's noteworthy that some tokenizers, like StarCoder (Li et al., 2023), combine `[\n]` with subsequent indentation into a single token. It renders our current infilling strategy inappropriate for such tokenizers since this tokenizer strategy breaks the token. Addressing this compatibility issue comprehensively is earmarked for future research.

Multi-line Function Infills Given JUMPCODER's line-level operation, our approach to infilling is limited to single-line infilling. A straightforward method is employing the newline token `[\n]` as a stop token in the infilling stage, substantially boosting the process's efficiency. However, we also recognized the need for multi-line continuity when filling the missing functions. Consequently, we implemented an adaptive stop criterion: the newline token serves as the default stop token. If the generation of a function signature is detected, this stop token is eliminated, facilitating the function's completion. This strategy effectively harmonizes the efficiency of line-level and function-level infilling, maintaining simplicity and clarity.

Filter Repetition of Infills It is known that language models tend to repeat previous sentences (Xu et al., 2022; Radford et al., 2019; Keskar et al., 2019), leading infilling models to occasionally produce contextually redundant statements. These poorly suited infills, frequently receiving higher scores from generation models, are often accepted during Judging. To address this, we utilized the gestalt pattern matching algorithm (Ratcliff et al., 1988) to exclude the infills that are overly similar to the existing context, applying a similarity threshold of 0.85.

Filter Invalid Multi-line Block Infills As JUMPCODER is currently restricted to single-line infills (except for function-level infilling), generating initial lines of code blocks like `if/for/while/try` often results in syntactic inconsistencies with the subsequent code. Therefore, we exclude infills that begin with keywords such as `if/for/while/try`, deferring the infilling of entire code blocks to future work for enhanced coherence and syntactic compatibility.

B.2 Implementation Details of Baselines and Evaluation Protocol

In this section, we introduce several similar baselines and compare them with JUMPCODER. We selected the following baseline:

Beam-Search and Rank BEAM-SEARCH (Wiseman and Rush, 2016) manages distinct search beams and makes a better selection during the decoding. Given that JUMPCODER requires generating up to $k = 5$ infills at each iteration, we demonstrate the benefits of this variant by comparing it with autoregressive generation five times, specifically using BEAM-SEARCH with a beam size of 5. We also compare RANK (Chen et al., 2021), which generates $k = 5$ codes and ranks them with the mean log probability. We used nucleus sampling with $p = 0.95$ and a temperature of 0.8, following the setting in Chen et al. (2021).

Self-Infilling Similar to our use of an infilling model for online modification, SELF-INFILLING (Zheng et al., 2024) employs infilling to augment generation processes. However, its application necessitates inherent infilling capabilities within the model, restricting its use to CODELLAMA-INSTRUCT models in our studies. We tested the recommended $N = 2$ setting where N denotes the number of the loop times within SELF-INFILLING. Since it was verified in their paper that the looping mechanism strategy is important, we report their performance using both two looping mechanism strategies, *Vanilla Split*, and *Extended Split*.

Self-Correct Self-Correct (Huang et al., 2023; Madaan et al., 2023; Ganguli et al., 2023) allows the model to generate two rounds and the second round aims to correct the code generated in the first round. It is akin to our approach of online modification, although its correction is offline. We used the prompt similar to Huang et al. (2023):

```
{Problem Prompt}
{Code generated in the first round}

"""
Review the above code and find
    problems (e.g., NameError) with
    the code. Based on the problems
    you found, improve your answer.
    Below is the refined code:
"""

{Problem Prompt}
{Code generated in the second round}
```

Similar to the variants used in our JUMPCODER, we refer to the code generated in the second round as **Vanilla**, name the code from the two rounds with the lower PPL as **Filtered**, and label the code from the two rounds that performs better in the evaluation stage as **Oracle**.

We also noticed that some recent research has created enhanced prompts to incorporate the compiler feedback after running test cases (Chen et al., 2023c; Zhang et al., 2023; Zhou et al., 2023) or human feedback (Wang et al., 2023b) for improved reasoning and coding. However, these approaches significantly alter the prompt and introduce considerable extra information to enhance LLMs, which is out of the scope of this paper. We choose the basic SELF-CORRECT as our focus is fairly comparing our online modification approach with the offline self-correct approach. Besides, such research concerning advanced prompts is complementary to our approach. For instance, our method can be applied in the second (or subsequent) round of generation with these baselines. Exploring the integration of powerful prompt techniques with our method is an avenue for future research.

Evaluation Protocol We found that existing code LLMs often perform degenerate behaviors resulting in empty program outputs, like mere `pass` or `"# TODO: implement me"` statements. This significantly obstructs the generation of valid programs and hurts the performance. To ensure a focused comparison of the performance of valid programs, we restricted the model from generating tokens `#` and `pass`. This strategy eliminates such invalid placeholder outputs, resulting in valid programs for all problems and a notable improvement in the performance of both the base models and our method.

Table 2 summarizes the full results, which shows that our JUMPCODER outperforms the baselines across different models and settings.

B.3 Efficiency Comparison

Speed comparison Fig. 8 shows the evaluation results on the efficiency comparison of JUMPCODER and traditional autoregressive coder. The average generation speed of JUMPCODER is about 0.7× that of autoregressive generation. This demonstrates that JUMPCODER's generation speed remains within the same order of magnitude as autoregressive generation. Furthermore, it indicates that JUMPCODER avoids significant performance

Table 2: (Full results of Fig. 5) The comparative analysis of JUMPCODER against various baselines.

Generation model	Size	Method	Setting		
			Vanilla	Filtered	Oracle
CODELLAMA-INSTRUCT	7B	GREEDY-DECODING	37.8	-	-
		BEAM-SEARCH ($S = 5$)	37.2	-	-
		SELF-INFILLING (*Vanilla split*, $N = 2$)	36.0	-	-
		SELF-INFILLING (*Extended split*, $N = 2$)	29.3	-	-
		RANK ($k = 5$)	39.0	-	-
		SELF-CORRECT	37.2	37.2	39.0
		JUMPCODER	**40.2**	**41.5**	**41.5**
CODELLAMA-PYTHON	7B	GREEDY-DECODING	42.7	-	-
		BEAM-SEARCH ($S = 5$)	41.4	-	-
		RANK ($k = 5$)	43.9	-	-
		SELF-CORRECT	42.1	42.1	42.7
		JUMPCODER	**46.3**	**46.3**	**47.6**
CODELLAMA-INSTRUCT	13B	GREEDY-DECODING	43.9	-	-
		BEAM-SEARCH ($S = 5$)	43.9	-	-
		SELF-INFILLING (*Vanilla split*, $N = 2$)	37.2	-	-
		SELF-INFILLING (*Extended split*, $N = 2$)	36.0	-	-
		RANK ($k = 5$)	42.7	-	-
		SELF-CORRECT	44.5	44.5	44.5
		JUMPCODER	**45.7**	**45.7**	**48.2**
CODELLAMA-PYTHON	13B	GREEDY-DECODING	48.8	-	-
		BEAM-SEARCH ($S = 5$)	49.3	-	-
		RANK ($k = 5$)	50.0	-	-
		SELF-CORRECT	50.0	50.0	50.6
		JUMPCODER	**50.6**	**50.6**	**51.2**

degradation across different problems.

Memory comparison Our JUMPCODER can utilize the same model for both generation and infilling tasks, or employ two separate models. To compare the runtime overhead between JUMPCODER and autoregressive coder, we use a single 7B model as both models, which avoids the inclusion of additional static model parameter storage in the comparison. Fig. 9 displays the runtime memory overhead for JUMPCODER. We find that JUMPCODER introduces up to nearly double the usage overhead, due to it potentially generating up to $k = 5$ infills simultaneously at maximum. Fortunately, with our speculative infilling optimization, many problems do not require generating so many infills simultaneously, which saves a part of runtime GPU memory.

C Examples of JUMPCODER

In this section, we first present several infills that were accepted (or rejected) by JUMPCODER during the Judging phase in Tables 3 and 4, to more intuitively demonstrate the workings of Judging. Then, we compare JUMPCODER with the traditional Autoregressive Coder through a few examples. Specifically, Figs. 10 to 13 show some Python samples that are solved effectively by our framework, and Figs. 14 to 16 show some Java, C++ and C# samples by our framework. Furthermore, Fig. 17 shows a sample that can be correctly solved by the autoregressive coder but is degenerated by our framework. We mark the infills in **bold orange**.

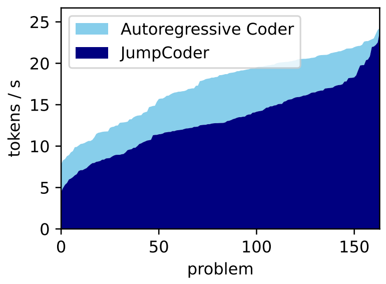


Figure 8: Number of tokens generated per second of the JUMPCODER on HumanEval for each problem. The generation model utilized is CODELLAMA-INSTRUCT-7B, with CODELLAMA-7B being the infilling model. The number of infill positions (k) is set to 5. Problems are sorted according to their generation speed.

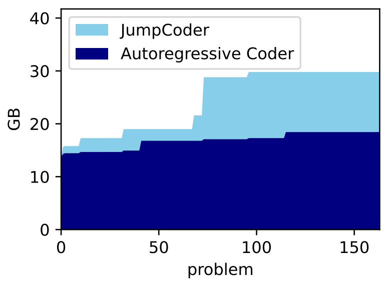


Figure 9: Memory usage of JUMPCODER on HumanEval when both the generation model and infilling model are the same CODELLAMA-INSTRUCT-7B for each problem. The number of infill positions (k) is set to 5. Problems are sorted according to the memory usage.

Table 3: Good infills *accepted* by JumpCoder during the Judging phase, along with the corresponding scores V. They can be broadly categorized into four types: Function, Reference, Variable, and Calculation.

Type	Example (infills are marked in **bold orange**)	Infill score V

Function	**`def is_prime(n):`**
	`    if n == 1:`
	`        return False`
	`    ...`
	`return '␣'.join(word for word in sentence.split() if`
	`    is_prime(len(word)))`
		$+\infty$

Reference	**`from functools import reduce`**
	`return sum(numbers), reduce(lambda x, y: x * y, numbers, 1)`
		$+\infty$

Variable	**`curr_sum = 0`**
	`for j in range(i, len(nums)):`
		0.75

	`unique_chars = {}`
	`max_length = 0`
		1.23

Calculation	**`lst = [x for x in lst if len(x) % 2 == 0]`**
	`lst = sorted(lst, key=lambda x: (len(x), x))`
		1.38

	`s = s.strip()` `# second infill`
	`s = s.replace(",", " ")` `# first infill`
	`return s.split()`
		first infill: 2.63
		second infill: 1.19

Table 4: Bad infills *rejected* by JumpCoder during the Judging phase, with the corresponding scores V. The first two examples violate syntax or coding standards, the third introduces incorrect calculations, and the fourth breaches semantics (`count += 1` should follow the `if` statement).

Example (infills are marked in **bold orange**)	Infill score V

`if not test:`
`    return {}`
`test = test.split()`
`else:`
	-6.15

`if number + remaining >= need:`
`    return [number + remaining - need, need]`
`else:`
`    return [number + remaining, need - (number + remaining)]`
`    return [number + remaining, 0]`
	-2.94

`if not arr:`
`    return True`
`arr.sort()`
`if len(arr) == 1:`
	-0.65

`count = 0`
`for sentence in S.split('.'):`
`    count += 1`
`    if sentence.startswith('I'):`
	0.19

```python
def minSubArraySum(nums):
    """
    Given an array of integers nums, find the minimum sum of any non-empty sub-array
    of nums.
    Example
    minSubArraySum([2, 3, 4, 1, 2, 4]) == 1
    minSubArraySum([-1, -2, -3]) == -6
    """
```

```python
    if not nums:
        return 0
    min_sum = sum(nums)
    for i in range(len(nums)):
        for j in range(i, len(nums)):
            if sum(nums[i:j]) < min_sum:
                min_sum = sum(nums[i:j])
    return min_sum
```

Autoregressive Coder (×)

```python
    if not nums:
        return 0
    min_sum = sum(nums)
    for i in range(len(nums)):
        curr_sum = 0
        for j in range(i, len(nums)):
            curr_sum += nums[j]
            if curr_sum < min_sum:
                min_sum = curr_sum
    return min_sum
```

JUMPCODER (✓)

Figure 10: An example program: In this example, our approach prompts the model to fill in a variable for tracking the current `sum(subarray)`, while the autoregressive coder uses the `sum` function for summation. This helps avoid errors arising from the use of list slicing operations.

```python
from typing import List


def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple groups of
    nested parentheses. Your goal is to
    separate those group into separate strings and return the list of those.
    Separate groups are balanced (each open brace is properly closed) and
    not nested within each other
    Ignore any spaces in the input string.
    >>> separate_paren_groups('( ) (( )) (( )( ))')
    ['()', '(())', '(()())']
    """
```

```python
    groups = []
    group = ''
    for char in paren_string:
        if char == '(':
            group += char
        elif char == ')':
            group += char
            groups.append(group)
            group = ''
        else:
            group += char
    return groups
```

Autoregressive Coder (×)

```python
    paren_string = paren_string.replace(' ',
     '')
    open_parens = 0
    group = ''
    groups = []
    for char in paren_string:
        if char == '(':
            open_parens += 1
            group += char
        elif char == ')':
            open_parens -= 1
            group += char
        if open_parens == 0:
            groups.append(group)
            group = ''
            open_parens = 0
    return groups
```

JUMPCODER (✓)

Figure 11: An example program: In this example, the presence of spaces in the string requires preprocessing. Our approach successfully empowers the model to make modifications in filling the preprocessing code and introducing a necessary variable (`open_parens`) for matching parentheses.

```python
from typing import List, Tuple


def sum_product(numbers: List[int]) -> Tuple[int, int]:
    """ For a given list of integers, return a tuple consisting of a sum
    and a product of all the integers in a list.
    Empty sum should be equal to 0 and empty product should be equal to 1.
    >>> sum_product([])
    (0, 1)
    >>> sum_product([1, 2, 3, 4])
    (10, 24)
    """
```

```python
from typing import List, Tuple


def sum_product(numbers: List[int]) -> Tuple[
     int, int]:

    return sum(numbers), reduce(lambda x
    , y: x * y, numbers, 1)
```

Autoregressive Coder (×)

```python
from functools import reduce
from typing import List, Tuple


def sum_product(numbers: List[int]) -> Tuple[
     int, int]:

    return sum(numbers), reduce(lambda x
    , y: x * y, numbers, 1)
```

JUMPCODER (✓)

Figure 12: An example program: In this example, the code generated by the original model utilizes the `reduce` function from the `functools` library without importing the relevant function. Our approach ensures the model successfully imports the `reduce` function.

```python
def words_in_sentence(sentence):
    """
    You are given a string representing a sentence,
    the sentence contains some words separated by a space,
    and you have to return a string that contains the words from the original sentence,
    whose lengths are prime numbers,
    the order of the words in the new string should be the same as the original one.

    Example 1:
        Input: sentence = "This is a test"
        Output: "is"

    Example 2:
        Input: sentence = "lets go for swimming"
        Output: "go for"

    Constraints:
        * 1 <= len(sentence) <= 100
        * sentence contains only letters
    """
```

```python
    return '␣'.join(word for word in sentence.
        split() if is_prime(
         len(word)))
```

Autoregressive Coder (×)

```python
    def is_prime(n):
        if n == 1:
            return False
        for i in range(2,int(n0.5)+1):
            if n % i == 0:
                return False
        return True

    return '␣'.join(word for word in sentence.
        split() if is_prime(len(word)))
```

JUMPCODER (✓)

Figure 13: An example program: In this example, the code generated by the original model uses the undefined function `is_prime(n)`. Our method enables the model to retroactively supplement the `is_prime(n)` function.

```java
import java.util.*;
import java.lang.reflect.*;
import org.javatuples.*;
import java.security.*;
import java.math.*;
import java.io.*;
import java.util.stream.*;
class Problem {
    // Input to this function is a string representing musical notes in a special ASCII format.
    // Your task is to parse this string and return array list of integers corresponding to how many
         beats does each not last.
    // Here is a legend:
    // 'o' - whole note, lasts four beats
    // 'o|' - half note, lasts two beats
    // '.|' - quater note, lasts one beat
    // >>> parseMusic(("o o| .| o| o| .| .| .| .| o o"))
    // (new ArrayList<Long>(Arrays.asList((long)4l, (long)2l, (long)1l, (long)2l, (long)2l, (long)1l, (
         long)1l, (long)1l, (long)1l, (long)4l, (long)4l)))
    public static ArrayList<Long> parseMusic(String music_string) {
```

```java
ArrayList<Long> result = new ArrayList<Long>();
for (int i = 0; i < music_string.length(); i++) {
    if (music_string.charAt(i) == 'o') {
        result.add((long)4);
    } else if (music_string.charAt(i) == 'o' &&
         music_string.charAt(i + 1) == '|') {
        result.add((long)2);
        i++;
    } else if (music_string.charAt(i) == '.' &&
         music_string.charAt(i + 1) == '|') {
        result.add((long)1);
        i++;
    }
}
return result;
}
```

Autoregressive Coder (×)

```java
String[] notes = music_string.split(" ");
ArrayList<Long> result = new ArrayList<Long>();
for (String note : notes) {
    if (note.equals("o")) {
        result.add((long)4);
    } else if (note.equals("o|")) {
        result.add((long)2);
    } else if (note.equals(".|")) {
        result.add((long)1);
    }
}
return result;
}
```

JUMPCODER (✓)

Figure 14: A Java example program: In this example, the presence of spaces in the string requires preprocessing. Our method successfully fills in the preprocessing code to split the input string into a string array.

```csharp
using System;
using System.Numerics;
using System.Diagnostics;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Security.Cryptography;
class Problem {
    // Return a string containing space-delimited numbers starting from 0 upto n inclusive.
    // >>> StringSequence((0L))
    // ("0")
    // >>> StringSequence((5L))
    // ("0 1 2 3 4 5")
    public static string StringSequence(long n) {
```

```csharp
        var sb = new StringBuilder();
        for (long i = 0; i <= n; i++) {
            sb.Append(i);
            sb.Append("␣");
        }
        return sb.ToString();
}
```

Autoregressive Coder (×)

```csharp
        var sb = new StringBuilder();
        for (long i = 0; i <= n; i++) {
            sb.Append(i);
            sb.Append("␣");
        }
        sb.Length--;
        return sb.ToString();
}
```

JUMPCODER (✓)

Figure 15: A C# example program: In this example, our approach incorporates an additional step to remove the space at the end of the string.

```cpp
#include<assert.h>
#include<bits/stdc++.h>
// You are given a string s.
// if s[i] is a letter, reverse its case from lower to upper or vise versa,
// otherwise keep it as it is.
// If the string contains no letters, reverse the string.
// The function should return the resulted string.
// Examples
// >>> solve(("1234"))
// ("4321")
// >>> solve(("ab"))
// ("AB")
// >>> solve(("#a@C"))
// ("#A@c")
std::string solve(std::string s) {
```

```cpp
    std::string result;
    for (char c : s) {
        if (c >= 'a' && c <= 'z') {
            result += c - 'a' + 'A';
        } else if (c >= 'A' && c <= 'Z') {
            result += c - 'A' + 'a';
        } else {
            result += c;
        }
    }
    return result;
}
```

Autoregressive Coder (×)

```cpp
    bool has_letter = false;
    std::string result;
    for (char c : s) {
        if (c >= 'a' && c <= 'z') {
            result += c - 'a' + 'A';
            has_letter = true;
        } else if (c >= 'A' && c <= 'Z') {
            result += c - 'A' + 'a';
            has_letter = true;
        } else {
            result += c;
        }
    }
    if (!has_letter) {
        std::reverse(result.begin(), result.
            end());
    }
    return result;
}
```

JUMPCODER (✓)

Figure 16: A C++ example program: In this example, our approach retrospectively defines a necessary variable (`has_letter`) which records the presence of a letter.

```python
def digitSum(s):
    """Task
    Write a function that takes a string as input and returns the sum of the upper characters only'
    ASCII codes.

    Examples:
        digitSum("") => 0
        digitSum("abAB") => 131
        digitSum("abcCd") => 67
        digitSum("helloE") => 69
        digitSum("woArBld") => 131
        digitSum("aAaaaXa") => 153
    """
```

```python
    sum = 0
    for c in s:
        if c.isupper():
            sum += ord(c)
    return sum
```

Autoregressive Coder (✓)

```python
    s = s.upper()
    sum = 0
    for c in s:
        if c.isalpha():
            sum += ord(c)
    return sum
```

JUMPCODER (×)

Figure 17: An example program: In this example, we illustrate that the infilling model may exhibit a certain level of misunderstanding, filling in possibly erroneous code.