# ENHANCING CODE GENERATION WITH GRAMMAR-AUGMENTED REPEATED SAMPLING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Large Language Models (LLMs) have demonstrated notable skill in generating code, yet their outputs often suffer from syntactic errors and limited coverage when restricted to single-attempt inference. In this work, we investigate integrating grammar-augmented decoding via SynCode's deterministic finite automaton (DFA) constraints, and repeated sampling techniques to enhance code generation quality. We evaluate three open-source LLMs on a subset of the HumanEval benchmark and find that, while grammar enforcement reduces syntactic errors, it does not consistently improve pass@k performance. In contrast, scaling inference compute through repeated sampling significantly boosts coverage and accuracy, narrowing the performance gap between smaller, more cost-effective models and their larger counterparts.

## 1 INTRODUCTION

Large Language Models (LLMs) have shown remarkable growth in their capabilities to generate code and perform mathematical reasoning tasks. However, despite steady improvements, these models often produce outputs with syntactical and semantic errors, limiting their applicability in real-world programming and computational problem-solving contexts. Traditional approaches have focused on increasing computing resources, improving model architectures, training methods, and prompt engineering, yet code generated by LLMs still frequently contains syntax errors, incomplete logic, or fails to adhere to the strict grammatical rules of programming languages.

In this work, we propose an integrated framework that addresses the dual challenges of ensuring syntactical correctness and increasing the reliability of generated solutions. We take advantage of the grammar-augmented decoding capabilities of SynCode, an approach that enforces adherence to a context-free grammar (CFG) through a deterministic finite automaton (DFA) mask store, to filter out syntactically invalid tokens during the generation process. To further enhance model performance, we incorporate repeated sampling strategies inspired by the "Large Language Monkeys" framework. By generating multiple candidate solutions and selecting the best ones through majority voting, our approach explores a broader solution space, improving coverage, and reducing the likelihood of syntactic or logical failure modes.

## 2 BACKGROUND

Our paper incorporates `SyncodeLogitsProcessor` from the paper "SynCode: LLM Generation with Grammar Augmentation(2)." By applying SynCode as a logit processor with Python, we aim to improve the syntactical accuracy of code generated by models. Recognizing the limitations of single-attempt inference, our methodology incorporates repeated sampling strategies from the paper "Large Language Monkeys: Scaling Inference Compute with Repeated Sampling(1)" to further increase coverage and reliability. To facilitate this integration, we modified the SynCode logit processor to support batch processing, enabling more efficient handling of multiple samples simultaneously.

# 3 METHODOLOGY

To carry out our experiments our methodology incorporates the following general pipeline: (1) model selection and setup, (2) application of grammar-enforced decoding via SynCode, (3) sampling multiple candidate solutions per prompt, and (4) evaluation on a standard code generation benchmark using established metrics. For consistency, the same procedure is applied to multiple LLMs, each in a similar experimental setup.

## 3.1 MODELS

We consider multiple open-source LLMs specialized for code-related tasks: stabilityai/stable-code-3b, codellama/CodeLlama-7b-Instruct-hf, and m-a-p/OpenCodeInterpreter-DS-6.7B. Each model is loaded with the Hugging Face Transformers library in a GPU environment to enable efficient batched inference. Tokenization is performed using the model-specific tokenizer, ensuring proper handling of padding and end-of-sequence tokens. Whenever necessary, special tokens are introduced to align the model configuration with the tokenizer's requirements.

## 3.2 GRAMMAR-AUGMENTED DECODING WITH SYNCODE

We implement SynCode as a `LogitsProcessor` within the Transformers inference pipeline. Before generating tokens, the processor computes a mask that zeroes out invalid token probabilities, thereby guiding the model to produce syntactically correct completions. Additionally, we introduce batch-processing modifications to the SynCode logit processor, allowing simultaneous grammar-enforced decoding for multiple parallel prompts and samples.

## 3.3 REPEATED SAMPLING AND PROMPTING STRATEGIES

To improve coverage and reliability, we employ repeated sampling at inference time for 50 samples. Rather than relying on a single model completion per problem, we generate multiple candidate solutions by sampling at a positive temperature. This repeated sampling approach increases the likelihood that at least one candidate is correct, effectively scaling the model's inference "compute." We experiment with varying numbers of samples per test case, balancing performance gains against computational cost.

For each problem in the benchmark dataset, we test two inference configurations:

- **With Grammar Enforcement:** Using SynCode to ensure syntactically valid completions.
- **Without Grammar Enforcement:** Allowing the model to produce completions without syntactical constraints, serving as a baseline for comparison.

Our zero-shot prompting directly feeds the problem prompt into the model without in-context examples measuring the model's innate code generation ability.

## 3.4 HYPERPARAMETERS

For each given problem prompt, we sample up to 50 candidate solutions at inference time. All samples are generated with a fixed temperature (`temperature = 0.7`) and nucleus sampling (`top_p = 0.95`) to maintain diversity in outputs without sacrificing coherence. These hyperparameters are informed by prior work and minor empirical tuning based on a small subset of tasks.

We use a maximum generation length of `max_length = 512` tokens for each sample, ensuring that the model can produce a reasonably complete solution without excessive truncation. We duplicate each prompt for the number of samples desired and prompt the model.

## 3.5 DATASETS AND EVALUATION

We use the HumanEval dataset to assess the correctness and robustness of generated code solutions. HumanEval provides a suite of Python coding problems paired with test cases, enabling a direct

measure of functional correctness. We randomly sample a subset of problems from the test set for efficiency while still capturing a diverse range of coding tasks.

To evaluate performance, we utilize the `code_eval` metric, which runs the generated code solutions against the provided test cases. The primary reported metric is `pass@k`, representing the fraction of problems for which at least one of the `k` generated samples passes all test cases. By varying `k`, we examine how scaling the number of sampled completions affects overall success rates.

### 3.6 Computational Analysis

We further analyze the computational aspects of scaling inference by measuring FLOPs, MACs, and parameter counts using the `calflops`(3) library. This provides insights into the trade-offs between the computational cost of repeated sampling and the achieved improvements in coverage and correctness. Such cost analyses inform the practical considerations of deploying grammar-augmented repeated sampling in real-world scenarios.

## 4 Implementation Details

In order to integrate SynCode's grammar-constrained decoding into a workflow that processes multiple samples simultaneously, modifications were made to the original `SyncodeLogitsProcessor` class. These changes enable batch processing support, allowing multiple problem prompts (or repeated samples for the same prompt) to be handled in parallel. Below, we highlight the key alterations introduced in the new `SyncodeLogitsProcessor2` class compared to the original implementation:

### 4.1 Batch-Aware Data Structures

The original `SyncodeLogitsProcessor` maintained a single parser instance and a single set of parser states. In `SyncodeLogitsProcessor2`, we introduce arrays to store and manage state information for each sample in the batch. Key attributes that were previously scalars (e.g., `last_valid_state`, `function_ends`, `start_from`, and `inc_parser`) are now lists indexed by the batch dimension. For instance, `self.inc_parsers` becomes a list of `IncrementalParser` instances, one per sample in the batch.

### 4.2 Multiple Parser Instances

In the original version, a single `IncrementalParser` was reset at the start of each prompt. The new code creates and maintains one parser per sample. During `reset()`, `SyncodeLogitsProcessor2` expects a list of prompts (equal to the number of samples) and initializes a corresponding parser instance for each prompt. This ensures that each generated sequence is parsed independently, preserving correct syntactic constraints for every sample.

### 4.3 Adjustments to `reset()` Method

The `reset()` method in the original code assumed a single prompt. In the updated implementation, `reset()` accepts a list of prompts and sets up the parsing environment accordingly. It updates all state arrays (e.g., `self.last_valid_state`, `self.function_ends`, `self.start_from`) and reinitializes each parser. This change allows multiple sequences to start decoding from distinct prompt contexts without interfering with one another.

### 4.4 Per-Sample Parsing and Validity Checks

The logic for determining which tokens are valid now operates on a per-sample basis. Functions like `is_valid()` and the main `__call__()` method iterate over each batch element (`idx`) and work with `self.inc_parsers[idx]`, extracting the correct partial code segments (`partial_code = self._get_partial_codes(...)`). This ensures that grammar-based filtering is applied independently to each sequence, allowing each sample to follow its own syntactical path.

### 4.5 HANDLING LOGITS AND MASKS FOR EACH SAMPLE

In the scoring step (`__call__()`), where the grammar-driven masks are applied to the model's output logits, the code now loops over the batch dimension. After computing the grammar mask (`accept_mask`) for each sample, it applies the mask to the corresponding slice of the logits, ensuring that each sample only receives syntactically valid tokens based on its own incremental parsing state.

### 4.6 NO CHANGES TO CORE GRAMMAR LOGIC

The underlying grammar parsing, DFA mask generation, and CFG-based logic remain the same. The principal modification is the introduction of a batch dimension so that these operations are replicated in parallel for each sample. The handling of partial codes and accept sequences remains conceptually identical—just repeated `N` times for `N` samples in the batch. For a detailed view of the actual code changes refer to Appendix A.

## 5 RESULTS

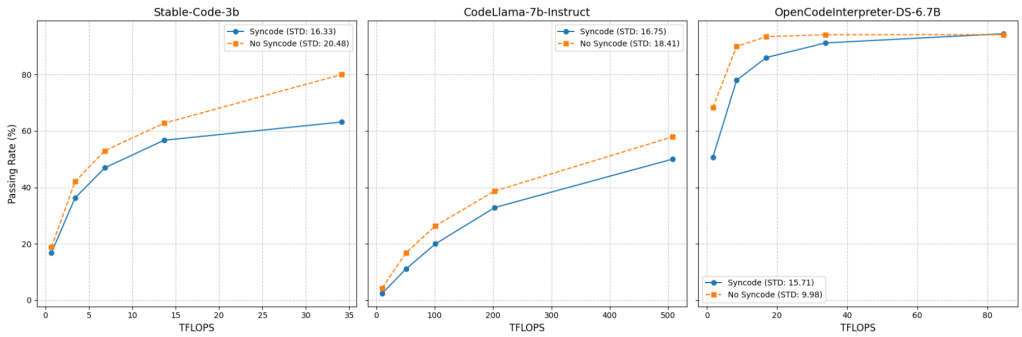
### 5.1 MODEL HUMANEVAL RESULTS

Table 1: Pass@k Comparison with and without Syncode

| Model | Syncode | No Syncode | Pass@1 | Pass@5 | Pass@10 | Pass@50 |
|---|---|---|---|---|---|---|
| Stable-Code-3b | ✓ | – | 16.84% | 36.29% | 46.92% | 63.16% |
| Stable-Code-3b | – | ✓ | 18.80% | 42.09% | 52.98% | 80.00% |
| CodeLlama-7b-Instruct | ✓ | – | 2.40% | 11.05% | 19.98% | 50.00% |
| CodeLlama-7b-Instruct | – | ✓ | 4.32% | 16.74% | 26.39% | 57.89% |
| OpenCodeInterpreter-DS-6.7B | ✓ | – | 50.67% | 77.93% | 86.00% | 94.44% |
| OpenCodeInterpreter-DS-6.7B | – | ✓ | 68.24% | 89.92% | 93.40% | 94.12% |

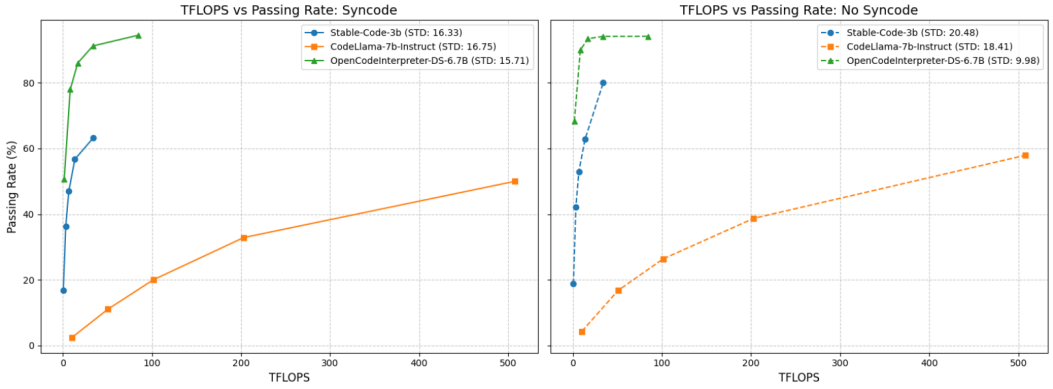
Table 1 summarizes the pass@k results ($k \in \{1, 5, 10, 50\}$) across three models—Stable-Code-3b, CodeLlama-7b-Instruct, and OpenCodeInterpreter-DS-6.7B—comparing performance with and without SynCode. Across all tested $k$-values, every model performs better without SynCode: pass@1, pass@5, pass@10, and pass@50 scores are consistently higher for the baseline decoding scenario. The largest improvements are observed at higher sampling counts, indicating that repeated sampling is more effective than syntax-enforcing mechanisms. Notably, while Stable-Code-3b and CodeLlama-7b-Instruct show significant gains without SynCode, OpenCodeInterpreter-DS-6.7B ultimately achieves nearly the same high-level performance ($\geq 94\%$) at pass@50 regardless of whether SynCode is used or not. The standard deviations (STD) of these metrics are also worth mentioning: in general, STD is higher without SynCode, suggesting greater variability in the unconstrained outputs, except for OpenCodeInterpreter-DS-6.7B, where the variability does not follow this trend as strongly.

### 5.2 PER-MODEL FLOPS COMPARISONS




The second chart compares the performance of each model using Syncode and not using Syncode. Each of the three graphs plots the HumanEval passing rate versus TFLOPs for a single model, comparing SynCode-enabled decoding and decoding without SynCode. TFLOPs here correspond to the approximate compute used, factoring in model TFLOPs per prompt multiplied by the number of samples. We see that for Stable-Code-3b and CodeLlama-7b-Instruct, the lines representing "no SynCode" decoding consistently outperform the "SynCode" lines. Even as we scale the number of samples (and hence TFLOPs), the unconstrained decoding strategy scales better, leading to higher pass rates at comparable computational costs. For OpenCodeInterpreter-DS-6.7B, a similar pattern is observed initially, but as the number of samples grows large, the difference narrows considerably. Beyond a certain point, both lines approach a converging performance level around $90\%$–$95\%$ pass rate, indicating that repeated sampling can eventually overcome the syntactic constraints imposed by SynCode for this particular model. The relative improvement in performance with increasing samples also highlights that while SynCode ensures syntactical correctness, it may not be necessary to achieve top-tier accuracy given enough computational budget and repeated sampling attempts.

### 5.3 AGGREGATE SYNCODE VS. NO SYNCODE




In the third chart, the results are presented in a more aggregated fashion, grouping all three models on a single plot for the SynCode scenario and another plot for the no-SynCode scenario. By placing all models together, these graphs highlight the differences in how each model scales with TFLOPs under each decoding regime. Both graphs show the stark differenes in TFLOPs for each model, both using Syncode and not using Syncode. On the no-SynCode graph, each model's passing rate grows more steeply, than the Syncode graph, and reaches higher overall levels, confirming that removing syntactical constraints allows more freedom for the models to stumble upon correct solutions, especially as the number of samples is increased. Interestingly, while both Stable-Code-3b and CodeLlama-7b-Instruct show not only higher means but also higher variability (STD) without SynCode, OpenCodeInterpreter-DS-6.7B appears less impacted by syntax enforcement in terms of variability. The combined view demonstrates that SynCode, while beneficial for syntactical correctness, tends to cap solution diversity and final accuracy levels, except for the largest sampling scenarios, where OpenCodeInterpreter-DS-6.7B attains near-equal top-tier performance whether SynCode is employed or not.

## 6 Conclusion and Future Work

In this work, we investigated the impact of integrating syntactical decoding constraints (via SynCode) and repeated sampling strategies on the code generation capabilities of several smaller large language models. Our experiments, conducted on a subset of the HumanEval benchmark, indicate that while grammar-augmented decoding ensures syntactically valid outputs, it does not consistently lead to improved pass@k scores. In fact, for the models considered, Stable-Code-3b, CodeLlama-7b-Instruct, and OpenCodeInterpreter-DS-6.7B, removing syntactical constraints generally yielded higher passing rates. Repeated sampling significantly enhanced performance across all models, allowing even smaller models to approach the solution quality of larger, more resource-intensive systems. Notably, OpenCodeInterpreter-DS-6.7B nearly matched its no SynCode performance at high sample counts, suggesting that with sufficient inference attempts, syntactic constraints become less crucial to achieving high accuracy.

Despite these positive results, our study has several limitations. High computational costs restricted the scope of our experiments. We did not utilize the full HumanEval dataset due to the prohibitive time and resources required, and our evaluation was confined to a subset of tasks and a small selection of models. Incorporating additional benchmarks, languages, or domains would provide a more comprehensive understanding of when and where grammar augmentation truly shines. Additionally, while we measured the FLOPs and TFLOPs per inference for our chosen models, ranging from approximately 682.71 GFLOPs for Stable-Code-3b to 10.15 TFLOPs for CodeLlama-7b-Instruct, and an estimated 1.69 TFLOPs for OpenCodeInterpreter-DS-6.7B, we did not exhaustively explore techniques to reduce these computational demands. Larger models, such as GPT-4 with an estimated compute cost in the trillions of FLOPs, remain out of reach for these methods under current constraints.

These limitations inform several avenues for future work. First, exploring more efficient sampling methods or parallelization strategies could mitigate the resource intensiveness of repeated inference. Second, evaluating the effectiveness of SynCode and similar constraints on a broader range of tasks, beyond Python and beyond pure code generation, may uncover scenarios where syntactical guarantees prove more valuable. Third, investigating methods to automatically verify correctness or reduce the computational overhead could make it feasible to test more models and more benchmarks, enabling a deeper understanding of the interplay between syntax enforcement, repeated sampling, and model scale.

In conclusion, while our results show that SynCode alone does not improve final accuracy for standard Python tasks, repeated sampling can compensate for both syntactic shortcomings and model size limitations. By scaling inference compute, smaller, more cost-effective models can approach the performance levels of larger models. Future research should focus on reducing the resource barriers to broader experimentation, expanding the set of domains and tasks, and refining methods to seamlessly integrate syntactical constraints with efficient, large-scale sampling strategies.

## References

[1] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024.

[2] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Syncode: Llm generation with grammar augmentation, 2024.

[3] xiaoju ye. calflops: a flops and params calculate tool for neural networks in pytorch framework, 2023.

## A Appendix

Our updated SyncodeLogitProcessor2 code as well as the rest of our code can be found here: `https://github.com/marzbana/GARS.git`.