Preprints.org

Article

Not peer-reviewed version

Grammar-Guided Incremental Method for Efficient LLM-Generated Code Execution

Anton Svystunov * and Yaroslav Tereshchenko

Posted Date: 2 April 2026

doi: 10.20944/preprints202604.0147.v1

Keywords: Large Language Models; code generation; context-free grammar augmentation; incremental parsing; real-time code execution; early error detection

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2026 doi:10.20944/preprints202604.0147.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Grammar-Guided Incremental Method for Efficient LLM-Generated Code Execution

Anton Svystunov and **Yaroslav Tereshchenko**

Taras Shevchenko National University of Kyiv, Kyiv, Ukraine
* Correspondence: antonsvystunov@knu.ua

Abstract

Rapid advancements in large language models with code generation abilities have enabled new paradigms in automated software development, positioning AI both as a coding assistant and an active actor within complex software ecosystems. Traditional code generation pipelines, mostly relying on tool calling via ReAct approach, require a complete code snippet to be generated and followed by validation and correction, often leading to significant latency and resource overhead due to sequential inference and execution processes. This research introduces a novel asynchronous inference algorithm that integrates context-free grammar parsing with real-time REPL-based execution, enabling early detection of syntax, semantic, and runtime errors without completing entire code snippets. We formally define the suitability criteria for LLMs in a target programming language, establish parse-tree-based identification of top-level statements, and present an incremental buffer-parsing mechanism that triggers execution upon recognition of complete statements. Implemented for Python 3 using the Lark parser and evaluated on a modified MBPP split (N=113 tasks; dataset and prompts in the Appendix) across six models—CodeAct–Mistral, GPT-OSS 20B, Gemma 3, Llama 3.2, Phi 4, and Qwen3-Coder 30B—our method is compared to a synchronous baseline using paired Wilcoxon tests with Bonferroni correction. Empirical results show significantly faster time-to-first-output for every model, large reductions in total latency where top-level script execution dominates (up to roughly an order of magnitude for CodeAct–Mistral), and no material change in pass or correctness rates, indicating that incremental execution improves responsiveness without altering task outcomes. With special prompting or finetuning, the method shows up to 4x reduction in latency for valid code generation. The benchmark results confirm that synchronous inference constraints can be alleviated through grammar-guided incremental execution, allowing more efficient and responsive agent-driven code execution workflows. Future research will explore predictive parsing techniques, deeper integration with agentic system architectures, security constraints, and formulating runtime requirements for scalable deployment of LLM-generated code execution environments.

Keywords: Large Language Models; code generation; context-free grammar augmentation; incremental parsing; real-time code execution; early error detection

1. Introduction

Recent advances in Large Language Models (LLMs) have made agentic control over software systems increasingly feasible. In the area of code generation, the field has evolved from one-pass synthesis toward multi-stage and agentic workflows. Code-oriented pretrained models such as StarCoder demonstrated strong general-purpose program synthesis capabilities [1], while later systems such as MapCoder introduced explicit planning, generation, and debugging stages into the inference pipeline [2]. Execution-aware approaches, including LDB, further extended this paradigm by using runtime information to support step-by-step debugging and correction [3]. More broadly, these developments reflect a shift from viewing LLMs merely as code autocompletion tools toward treating them as active components of larger software systems. This perspective is aligned with the ideas of AI

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2026 doi:10.20944/preprints202604.0147.v1

OS [4] and Software 3.0 [5], where AI models are expected to participate directly in software execution and control loops.

Despite this progress, most existing approaches still preserve a sequential *generate–then–validate* workflow. Even when planning, debugging, or self-correction mechanisms are added, code is typically analyzed only after a complete or nearly complete fragment has already been produced. As a result, syntax, semantic, and runtime errors are often detected late, which increases latency and may require multiple correction iterations. This limitation is especially important in agentic execution settings, where delayed feedback from the environment directly reduces system responsiveness and efficiency. Related studies also show that LLM-based code generation remains vulnerable to ambiguous task descriptions, repetition, and broader reliability issues [6,7], which makes earlier and more fine-grained validation increasingly important.

Another line of research addresses code correctness directly during decoding. For example, SynCode incorporates context-free grammar constraints into the generation process to ensure syntactic validity of partial outputs [8]. Such approaches reduce parser-level errors and improve structural consistency, but syntax-level constraints alone do not resolve semantic inconsistencies, contextual errors, or runtime failures. These types of failures can only be exposed through execution or through deeper incremental analysis of the generated program. Therefore, although constrained decoding improves one important aspect of code generation, it does not remove the need for mechanisms that can validate and execute code earlier in the inference process.

In this paper, we focus on the agentic use case in which an AI system generates and executes programs in an external control environment. Unlike conventional pipelines that wait for full program generation before validation and execution, we study an inference-time algorithm based on incremental parsing and early execution. The main idea is to identify executable top-level statements as soon as they are formed and to run them immediately, thereby obtaining earlier feedback about syntax, semantic, and runtime errors. In this way, the proposed approach addresses a limitation shared by many existing methods: although they improve the quality of generated code, they still largely postpone executable validation until after substantial generation has already occurred.

The object of the research is LLM inference algorithms. The aim and objectives of the research are to describe an effective algorithm for LLM-based code generation that will:

- detect syntax errors without the need to complete the code generation;
- execute generated code as soon as possible;
- detect semantic and runtime errors without the need to complete the code generation.

2. Materials and Methods

The main ideas on which the proposed algorithm is built are:

- Context-free grammar inference performed alongside LLM inference can detect syntax errors early [8].
- REPL can be used to execute code line-by-line as it is generated.
- LLM inference should be stopped once an error occurs.

To describe a theoretical background for the algorithm, we introduce some definitions and constraints.

First, we need to narrow the list of applicable LLMs and programming languages.

Definition 1. *Let*

- *V_L be the vocabulary of tokens produced by the LLM,*
- *$V_L^* = \bigcup_{n=0}^{\infty} V_L^n$ be the set of all finite sequences of tokens ("words"),*
- *Lex_p^* be the set of all finite sequences of lexemes of the lexer of programming language p.*

We say that the LLM is suitable for code generation *in the target programming language p if*

$$\forall \ell \in Lex_p^* \; \exists s \in V_L^* : \; \ell_1 \cdot \ell_2 \cdot \ldots \ell_n = s_1 \cdot s_2 \cdot \ldots s_m.$$

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2026 doi:10.20944/preprints202604.0147.v1

Second, the target programming language should have a REPL interface, which can execute top-level statements in a target programming language. Formally, we define a top-level statement as follows.

Definition 2. *Let $G = (N, T, P, S)$ be a context-free grammar with start symbol S. A* parse tree *T for G is a finite, ordered, labelled tree satisfying:*

- $\mathrm{root}(T)$ *is labelled S.*
- *If a node v is labelled $A \in N$ and has children $v_1, \ldots, v_k$, then there is a production*

$$A \;\rightarrow\; X_1\, X_2 \cdots X_k \quad \in P \quad \text{with} \quad \mathrm{label}(v_i) = X_i.$$

- *The leaves of T are labelled by tokens in $T \cup \{\varepsilon\}$. Reading the leaf labels from left to right gives the* yield *of T.*

Define $\mathrm{PT}_G = \{\, T \mid T \text{ is a parse tree of } G \}$, $\mathrm{yield}(T) =$ *concatenation of the leaf labels of T. Suppose* `Stmt` $\in N$ *is the non-terminal for a single statement. Then a token string $w \in T^*$ is a* top-level statement *of G exactly when*

$$\exists\, T \in \mathrm{PT}_G : \; \mathrm{root}(T) = S \;\wedge\; \exists\, v \in T \big[\, \mathrm{label}(v) = \texttt{Stmt} \;\wedge\; \mathrm{yield}(T_v) = w \,\big],$$

where T_v denotes the subtree of T rooted at node v.

Third, we need to identify when top-level statements from the token stream can be executed. As we work with a partial sequence of lexemes on each step of inference and not the complete program, we should execute the top-level statement only once we have an assurance that the statement is complete and a new top-level statement has been started.

The initial version of the inference algorithm is described in Algorithm 1.

We refer to this grammar-guided incremental execution strategy as *JitGen* (just-in-time generation) throughout the remainder of this paper.

3. Results

In this paper, we evaluate a simplified idea based on using only the parser without a parallel lexer executing a DFA. Thus, on each token generation, we perform buffer parsing instead of storing a previous state of grammar inference.

We implemented a script that targets the Python 3 programming language.

Hypothesis. We compare two algorithms.

- The first (so-called "sync") executes the code only after the LLM has finished the generation of the code snippet.
- The second (so-called "async") parses and executes the code on each line of inference.

In both cases, the code snippet is generated and executed only once. Execution (and LLM inference) is halted once we get a syntax or runtime error.

By performing the benchmark, we check two hypotheses:

- If the generated code contains errors, the async algorithm will halt on error earlier than the sync algorithm.
- If the generated code is correct, the async algorithm will execute the code earlier (first-output time).

Evaluation Setup. We use the modified A.1 MBPP [9] dataset as a source of coding problems in the Python language and run both sync and async code execution approaches on CodeAct–Mistral, GPT-OSS 20B, Gemma 3, Llama 3.2, Phi 4, and Qwen3-Coder 30B. Models were executed on the same set of prompts A.4 with the same hyperparameters (seed, temperature, max tokens, etc.) for both

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2026 doi:10.20944/preprints202604.0147.v1

Algorithm 1 RunCodeFromStream

```
1: procedure RunCodeFromStream(inputStream)
2:     interpreter ← CreateCodeExecutor()
3:     codeBuffer ← ""
4:     for each fragment in inputStream do
5:         codeBuffer ← codeBuffer + fragment
6:         tempCode ← codeBuffer
7:         try
8:         tree ← ParseCode(tempCode)                    ▷ May be substituted with stateful parser
9:         catch IncompleteInputError
10:        continue                                       ▷ Wait for more fragments
11:        catch SyntaxError e
12:        if e indicates incomplete input then
13:            continue
14:        else
15:            raise Error
16:        end if
17:        end try
18:        if tree has multiple complete statements then
19:            executedUpto ← 0
20:            for each complete statement except last do
21:                start ← statement.startPosition
22:                end ← statement.endPosition
23:                code ← codeBuffer[start : end]
24:                if code is not empty then
25:                    output ← interpreter.Execute(code)
26:                    yield output
27:                end if
28:                executedUpto ← end
29:            end for
30:            codeBuffer ← codeBuffer[executedUpto :]
31:        end if
32:    end for
33:    if codeBuffer is not empty then                   ▷ If there is any code left unprocessed
34:        try
35:        tree ← ParseCode(codeBuffer)
36:        for each statement in tree do
37:            start ← statement.startPosition
38:            end ← statement.endPosition
39:            code ← codeBuffer[start : end]
40:            if code is not empty then
41:                output ← interpreter.Execute(code)
42:                yield output
43:            end if
44:        end for
45:        catch Error e
46:        raise Error
47:        end try
48:    end if
49: end procedure
```

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2026 doi:10.20944/preprints202604.0147.v1

algorithms. Execution time has been measured on an AMD Ryzen 7 7700X 8-core CPU with NVIDIA GeForce RTX 4090. As a parser implementation, we used the Python 3 grammar from the Lark package.

To assess whether the differences in execution time and time to first output between the async (JitGen) and sync (sequential) strategies were statistically significant, we employed the Wilcoxon signed-rank test [10], a non-parametric paired-difference test that does not assume normality of the underlying distributions. For each model and metric (total execution time and time to first output), we computed the paired differences $\Delta = \text{Async} - \text{Sync}$ across all matched task instances. Zero differences were excluded prior to testing, as the Wilcoxon signed-rank test is undefined for tied pairs at zero [11]. Tests were conducted as two-sided, and pairs with fewer than 10 non-zero differences were excluded from analysis to ensure sufficient statistical power. To control the family-wise error rate across the k hypothesis tests performed (two metrics $\times$ number of models), we applied the Bonferroni correction [12], adjusting each p-value as $p_{\text{corrected}} = \min(p \cdot k,\ 1)$, with a significance threshold of $\alpha = 0.05$. As a measure of effect size, we computed the rank-biserial correlation [13], defined as $r = 1 - \frac{2W}{n(n+1)/2}$, where W is the Wilcoxon test statistic and n is the number of non-zero paired differences. This effect size indicates the proportion of favorable versus unfavorable pair rankings, with values near -1 or $+1$ indicating strong directional effects.

Supplementary Materials provide figures not shown in the main text (model-wise overview heatmap, ECDF and paired-difference views of time to first output, speedup ratios, correctness, and error-type counts) and additional tables: Wilcoxon tests (Table A1), pass and correctness rates (Table A2), and per-category outcome counts (Table A3); see Figures A1–A6.

The benchmark results in Figures 1 and 2, with descriptive medians and means in Table 1 and supplementary tests in Table A1, support the following:

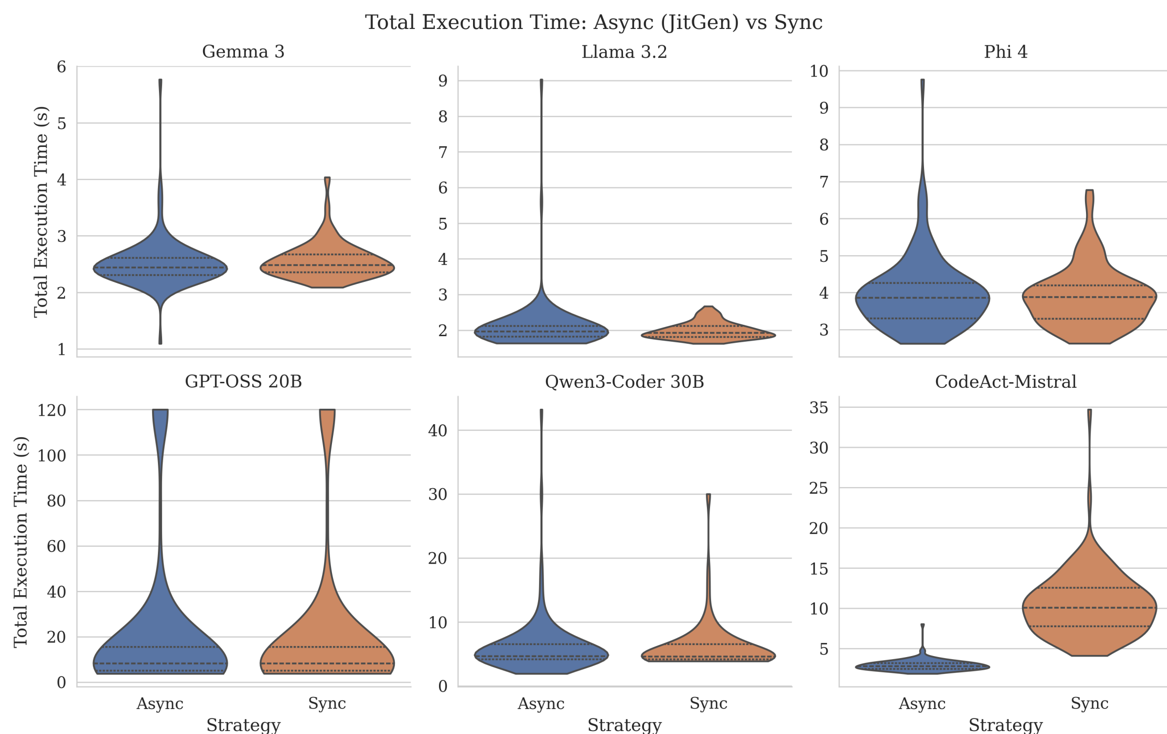

Figure 1. Distribution of total execution time (seconds) by model and algorithm (N=113 tasks per series; same generated programs).

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

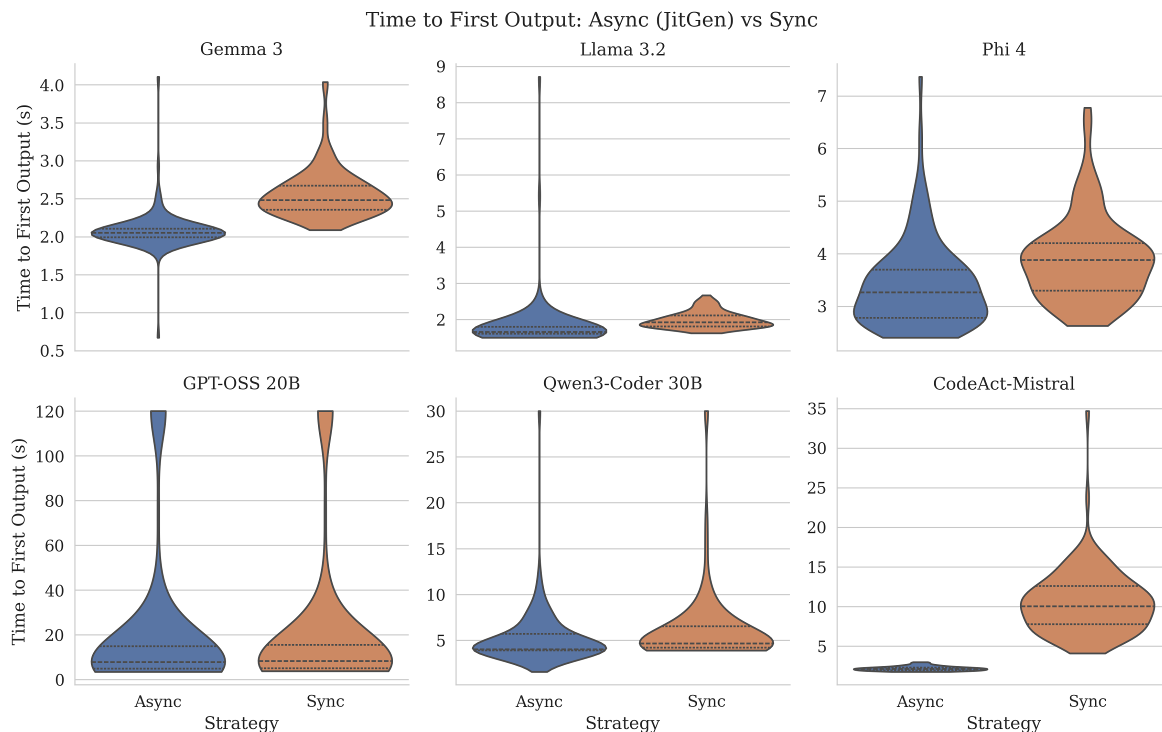

Figure 2. Distribution of time to first output (seconds) by model and algorithm (N=113 tasks per series).

Table 1. Descriptive statistics for execution time and time to first output (seconds).

Model	Algo.	N	Exec. time			First output		
			Mean	Med.	Std	Mean	Med.	Std
CodeAct-Mistral	Async	113	2.926	2.802	0.719	2.163	2.117	0.286
CodeAct-Mistral	Sync	113	10.490	10.074	4.084	10.466	10.027	4.085
GPT-OSS 20B	Async	113	20.523	8.295	32.635	20.179	7.958	32.730
GPT-OSS 20B	Sync	113	20.464	8.301	32.636	20.464	8.300	32.636
Gemma 3	Async	113	2.516	2.442	0.444	2.081	2.055	0.273
Gemma 3	Sync	113	2.565	2.486	0.327	2.565	2.485	0.327
Llama 3.2	Async	113	2.101	1.969	0.778	1.821	1.659	0.765
Llama 3.2	Sync	113	1.992	1.930	0.231	1.992	1.930	0.231
Phi 4	Async	113	3.969	3.864	0.983	3.395	3.263	0.823
Phi 4	Sync	113	3.886	3.884	0.805	3.886	3.884	0.805
Qwen3-Coder 30B	Async	113	6.232	4.696	4.852	5.232	4.027	2.967
Qwen3-Coder 30B	Sync	113	6.100	4.642	4.019	6.100	4.642	4.019

1. Time to first output is substantially lower under async (JitGen) for every model; paired Wilcoxon tests remain significant after Bonferroni correction (Table A1).
2. Total execution time shows the largest async advantage for CodeAct–Mistral (roughly an order of magnitude at the median), consistent with more top-level, line-by-line executable script structure [14]. For GPT-OSS 20B and Qwen3-Coder 30B, median total times are nearly unchanged between strategies, and differences are not significant after correction—whereas first-output gains remain clear, matching a setting where much of the stream is shared reasoning before runnable code appears [15,16].
3. Gemma 3 and Phi 4 exhibit significant paired differences for total execution time in our tests (Table A1); Llama 3.2 does not for total time after correction, despite faster first output under async.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2026
doi:10.20944/preprints202604.0147.v1

4. Correctness and timeout rates match between async and sync for each model; pass rates match for five of six models, with Llama 3.2 differing by less than one percentage point (85.84% vs 86.73%; Table A2).

Overall, async (JitGen) execution reduces time-to-first-output everywhere in our benchmark, reduces total latency most clearly when models emit runnable top-level code early (e.g., CodeAct–Mistral), and leaves correctness and timeouts unchanged, with only a small pass-rate discrepancy on Llama 3.2.

4. Conclusions

In this paper, we proposed a novel LLM inference algorithm for code generation and analyzed the benchmarks for a partial implementation. The benchmark results show that the proposed algorithm allows a significant decrease in the error detection time in LLM-generated code and allows code snippets to be executed in parallel with LLM inference.

Further studies will be dedicated to algorithm performance improvements, application of predictive parsing, agentic systems integration, and formulation of the requirements for a runtime for LLM-generated code execution.

Author Contributions: Conceptualization, A.S; methodology, A.S.; analysis of sources A.S. and Y.T.; algorithm, A.S.; benchmark script implementation, A.S.; writing—original draft preparation, A.S.; writing—review and editing, Y.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The datasets used in this study, including MBPP, are publicly available. The implementation of the benchmark script is available from the corresponding author on reasonable request.

Acknowledgments: All authors contributed to the conception, design, implementation, and writing of this work. All materials and resources used in this study were prepared by the authors.All authors thanks Taras Shevchenko National University of Kyiv for support.

Conflicts of Interest: The authors declares no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DFA	Deterministic Finite Automaton
LLM	Large Language Model
REPL	Read-Eval-Print Loop

Appendix A.

This supplement provides a model-wise overview heatmap, plots complementary to the main-text latency figures (ECDF and paired differences in time to first output, speedup ratios, correctness, and error-type counts), and tabulated outcome rates, error breakdowns, and paired Wilcoxon tests for the benchmark in the Results section (descriptive time statistics appear as Table 1 in the main text). The MBPP variant used for evaluation is public; implementation details are available from the corresponding author on reasonable request.

Appendix A.1. Dataset

Modified MBPP dataset which is used for evaluation is available at https://huggingface.co/datasets/AntonSvystunov/mbpp-jitgen-validation.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Appendix A.2. Extended Benchmark Figures

Summary: Pass Rate (%), Correctness (%), Median Times (s)

	PassRate (Async)	PassRate (Sync)	Correctness (Async)	Correctness (Sync)	MdnExecTime (Async)	MdnExecTime (Sync)	MdnFirstOutput (Async)	MdnFirstOutput (Sync)
Gemma 3	98.23	98.23	54.87	54.87	2.44	2.49	2.05	2.49
Llama 3.2	85.84	86.73	30.97	30.97	1.97	1.93	1.66	1.93
Phi 4	99.12	99.12	71.68	71.68	3.86	3.88	3.26	3.88
GPT-OSS 20B	100.00	100.00	58.41	58.41	8.29	8.30	7.96	8.30
Qwen3-Coder 30B	99.12	99.12	77.88	77.88	4.70	4.64	4.03	4.64
CodeAct-Mistral	86.73	86.73	41.59	41.59	2.80	10.07	2.12	10.03

Figure A1. Overview heatmap of benchmark metrics across models and algorithms (supplementary material).

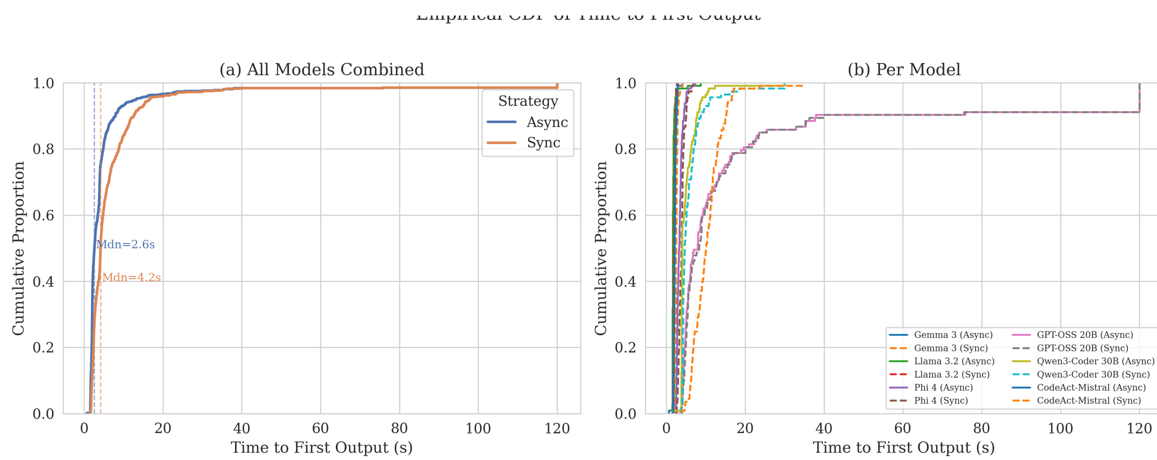

Figure A2. Empirical cumulative distribution of time to first output across tasks.

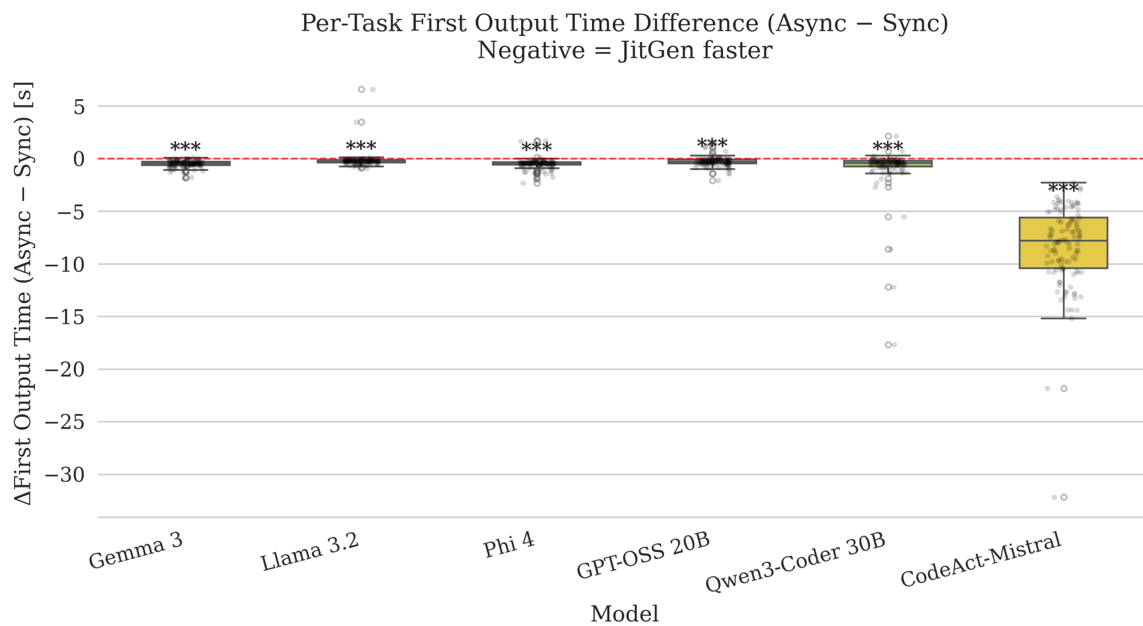

Figure A3. Paired differences in time to first output (Δ = Async − Sync, seconds) per task.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2026
doi:10.20944/preprints202604.0147.v1

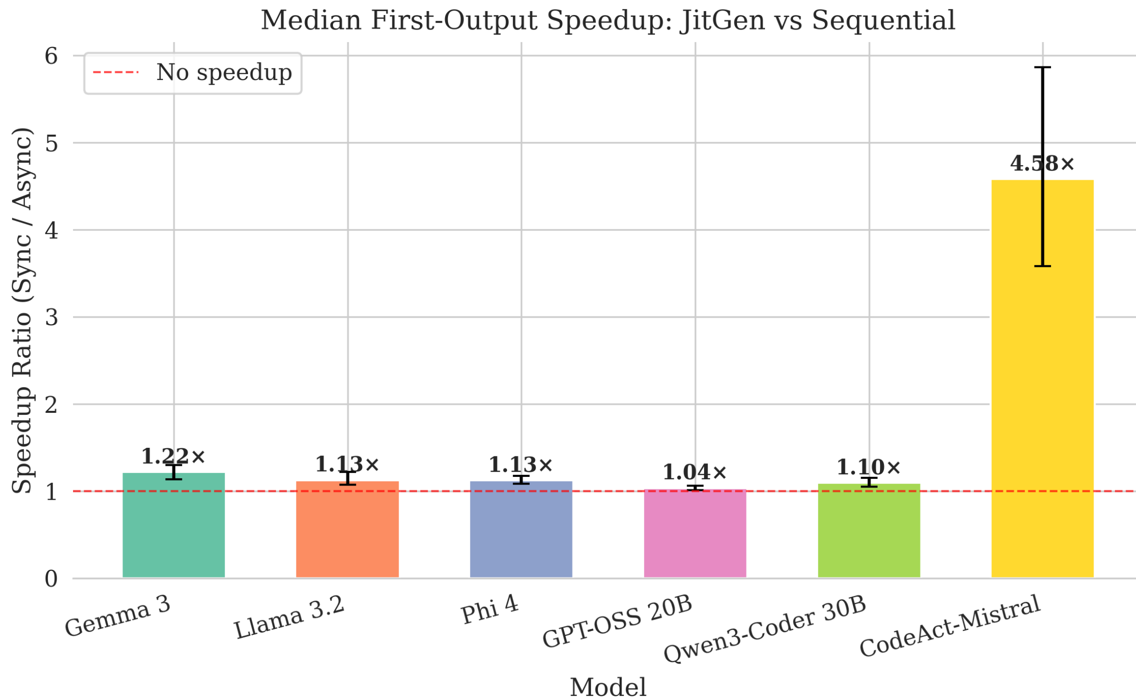

Figure A4. Speedup ratio comparing async (JitGen) and synchronous execution across models.

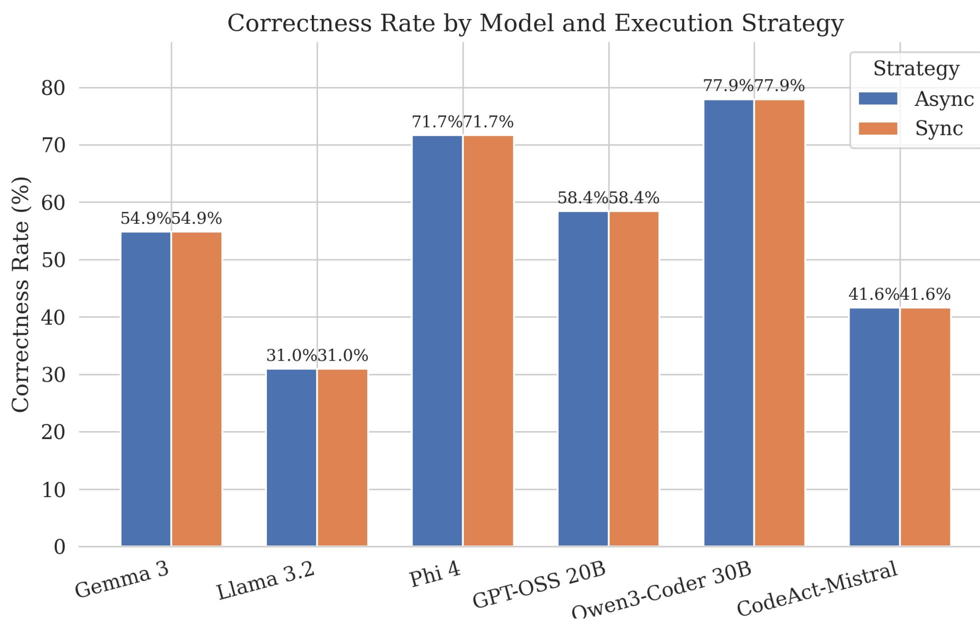

Figure A5. Pass and correctness rates by model and algorithm.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Figure A6. Counts of run outcomes by error category (same generated code; execution strategy differs only in timing).

Appendix A.3. Supplementary Tables

All paired tests use Δ = Async − Sync on matched tasks; negative mean or median Δ for time metrics indicates faster async. Significance uses two-sided Wilcoxon signed-rank tests with Bonferroni correction across tests as in the main text.

Table A1. Paired Wilcoxon signed-rank tests: execution time and time to first output.

Model	Metric	N	Mean Δ	Med. Δ	W	p	$p_{\text{corr.}}$	r	Sig.
Gemma 3	Exec. time	113	−0.0488	−0.0522	803	$< 10^{-11}$	$< 10^{-10}$	0.751	Yes
Gemma 3	First output	113	−0.484	−0.4455	8	$< 10^{-19}$	$< 10^{-18}$	0.9975	Yes
Llama 3.2	Exec. time	113	0.1086	0.0087	2222	0.00423	0.0507	0.310	No
Llama 3.2	First output	113	−0.1713	−0.2139	248	$< 10^{-16}$	$< 10^{-15}$	0.923	Yes
Phi 4	Exec. time	113	0.0834	0.0153	1715	1.61×10^{-5}	1.93×10^{-4}	0.468	Yes
Phi 4	First output	113	−0.4903	−0.447	221	$< 10^{-17}$	$< 10^{-16}$	0.931	Yes
GPT-OSS 20B	Exec. time	113	0.0587	0.0015	2763	0.190	1.000	0.142	No
GPT-OSS 20B	First output	113	−0.2851	−0.2887	653	$< 10^{-12}$	$< 10^{-11}$	0.797	Yes
Qwen3-Coder 30B	Exec. time	113	0.1321	0.0037	2792	0.220	1.000	0.133	No
Qwen3-Coder 30B	First output	113	−0.868	−0.4104	225	$< 10^{-17}$	$< 10^{-16}$	0.930	Yes
CodeAct-Mistral	Exec. time	113	−7.5635	−7.334	0	$< 10^{-19}$	$< 10^{-18}$	1.000	Yes
CodeAct-Mistral	First output	113	−8.3037	−7.8275	0	$< 10^{-19}$	$< 10^{-18}$	1.000	Yes

Table A2. Pass rate, correctness rate, and timeout rate (%) by model and algorithm.

Model	Algo.	N	Pass	Correct	Timeout
CodeAct-Mistral	Async	113	86.73	41.59	0.0
CodeAct-Mistral	Sync	113	86.73	41.59	0.0
GPT-OSS 20B	Async	113	100.00	58.41	0.0
GPT-OSS 20B	Sync	113	100.00	58.41	0.0
Gemma 3	Async	113	98.23	54.87	0.0
Gemma 3	Sync	113	98.23	54.87	0.0
Llama 3.2	Async	113	85.84	30.97	0.0
Llama 3.2	Sync	113	86.73	30.97	0.0
Phi 4	Async	113	99.12	71.68	0.0
Phi 4	Sync	113	99.12	71.68	0.0
Qwen3-Coder 30B	Async	113	99.12	77.88	0.0
Qwen3-Coder 30B	Sync	113	99.12	77.88	0.0

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2026
doi:10.20944/preprints202604.0147.v1

Table A3. Task counts by outcome category (totals over $N = 113$ tasks per model and algorithm).

Model	Algo.	No input()	Index err.	Other err.	Success	Type err.
CodeAct-Mistral	Async	0	0	14	98	1
CodeAct-Mistral	Sync	0	0	14	98	1
GPT-OSS 20B	Async	0	0	0	113	0
GPT-OSS 20B	Sync	0	0	0	113	0
Gemma 3	Async	0	0	2	111	0
Gemma 3	Sync	0	0	2	111	0
Llama 3.2	Async	5	2	6	97	3
Llama 3.2	Sync	5	2	5	98	3
Phi 4	Async	0	0	1	112	0
Phi 4	Sync	0	0	1	112	0
Qwen3-Coder 30B	Async	0	0	1	112	0
Qwen3-Coder 30B	Sync	0	0	1	112	0

Appendix A.4. Benchmark Prompts

System Prompt

```
You generate Python scripts to be executed line-by-line.

IMPORTANT OUTPUT RULE:
- Your final answer must be ONLY Python code wrapped in a ‘‘‘python code block.
- Format: ‘‘‘python
<code>
‘‘‘

Your job:
- Given a problem statement and one test case, write a script that computes the answer
for that test case.
- Infer variables from the test case and define them as local variables near the top.
- Print the result using print(...) exactly as required.

Non-negotiable constraints:
- DO NOT use input() (or any interactive blocking call).
- Do not read stdin.
- No imports / no external libraries.
- No eval/exec/compile.
- No infinite loops; every loop must have a clear termination condition.
- Script must run top-to-bottom without errors.
- Define functions only if absolutely necessary, but the main logic should be in the
global scope.

*Structure* your code in following order:
1. Helper functions (if needed)
2. Variable definitions inferred from the test input
3. Main logic to solve the task
4. Final print statement with the answer

Separate your code into logical section with comments to match the above structure.
```

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

doi:10.20944/preprints202604.0147.v1


```
Internal self-check (do this silently before finalizing):
- [ ] All required values are defined as local variables from the test input
- [ ] No input() usage
- [ ] No imports
- [ ] Output matches the required format exactly
- [ ] Script runs top-to-bottom
- [ ] All utility code is defined before it's used
- [ ] Code structure follows the specified order

If ambiguous, choose the simplest interpretation consistent with the test case and
required output format.
If impossible, print a clear error message.
```

Human Prompt

```
TASK:
Write a Python code to find the sum of maximum increasing subsequence of the
given array.

TASK INPUT (infer local variables from this):
[3, 4, 5, 10], 4

EXPECTED OUTPUT FORMAT EXAMPLE:
106

Write a Python script that:
1) Defines local variables from the test input
2) Solves the task
3) Prints the result in the specified format

Remember: no input(), no imports, stdout only via print(...).

CRITICAL: Wrap your entire code in a ‘‘‘python code block.
```

References

1. Li, R.; Allal, L.B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. StarCoder: may the source be with you!, 2023, [arXiv:cs.CL/2305.06161].
2. Islam, M.A.; Ali, M.E.; Parvez, M.R. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving, 2024, [arXiv:cs.CL/2405.11403].
3. Zhong, L.; Wang, Z.; Shang, J. Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step-by-step, 2024, [arXiv:cs.SE/2402.16906].
4. Xu, S.; Li, Z.; Mei, K.; Zhang, Y. AIOS Compiler: LLM as Interpreter for Natural Language Programming and Flow Programming of AI Agents, 2024, [arXiv:cs.CL/2405.06907].
5. Hassan, A.E.; Oliva, G.A.; Lin, D.; Chen, B.; Jiang, Z.M. Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap, 2024, [arXiv:cs.SE/2410.06107].
6. Larbi, M.; Akli, A.; Papadakis, M.; Bouyousfi, R.; Cordy, M.; Sarro, F.; Traon, Y.L. When Prompts Go Wrong: Evaluating Code Model Robustness to Ambiguous, Contradictory, and Incomplete Task Descriptions, 2025, [arXiv:cs.SE/2507.20439].
7. Dong, Y.; Liu, Y.; Jiang, X.; Gu, B.; Jin, Z.; Li, G. Rethinking Repetition Problems of LLMs in Code Generation. In Proceedings of the Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers); Che, W.; Nabende, J.; Shutova, E.; Pilehvar, M.T., Eds., Vienna, Austria, 2025; pp. 965–985. https://doi.org/10.18653/v1/2025.acl-long.48.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2026
doi:10.20944/preprints202604.0147.v1

8. Ugare, S.; Suresh, T.; Kang, H.; Misailovic, S.; Singh, G. SynCode: LLM Generation with Grammar Augmentation, 2024, [arXiv:cs.LG/2403.01632].
9. Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* **2021**.
10. Wilcoxon, F. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* **1945**, *1*, 80–83. https://doi.org/10.2307/3001968.
11. Pratt, J.W. Remarks on Zeros and Ties in the Wilcoxon Signed Rank Procedures. *Journal of the American Statistical Association* **1959**, *54*, 655–667. https://doi.org/10.1080/01621459.1959.10501526.
12. Bonferroni, C. Teoria statistica delle classi e calcolo delle probabilità. *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commerciali di Firenze* **1936**, *8*, 3–62.
13. Kerby, D.S. The Simple Difference Formula: An Approach to Teaching Nonparametric Correlation. *Comprehensive Psychology* **2014**, *3*, 11.IT.3.1. https://doi.org/10.2466/11.IT.3.1.
14. Wang, X.; Chen, Y.; Yuan, L.; Zhang, Y.; Li, Y.; Peng, H.; Ji, H. Executable Code Actions Elicit Better LLM Agents, 2024, [arXiv:cs.CL/2402.01030].
15. OpenAI.; :.; Agarwal, S.; Ahmad, L.; Ai, J.; Altman, S.; Applebaum, A.; Arbus, E.; Arora, R.K.; Bai, Y.; et al. gpt-oss-120b & gpt-oss-20b Model Card, 2025, [arXiv:cs.CL/2508.10925].
16. Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Gao, C.; Huang, C.; Lv, C.; et al. Qwen3 Technical Report, 2025, [arXiv:cs.CL/2505.09388].

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.