Executing as You Generate: Hiding Execution Latency in LLM Code Generation

Zhensu Sun[*]
zssun@smu.edu.sg
Singapore Management University
Singapore

Zhihao Lin[*]
mathieulin@buaa.edu.cn
Beihang University
China

Zhi Chen
zhi.chen.2023@smu.edu.sg
Singapore Management University
Singapore

Chengran Yang
cryang@smu.edu.sg
Singapore Management University
Singapore

Mingyi Zhou
zhoumingyi@buaa.edu.cn
Beihang University
China

Li Li
lilicoding@ieee.org
Beihang University
China

David Lo
davidlo@smu.edu.sg
Singapore Management University
Singapore

arXiv:submit/7413660 [cs.SE] 1 Apr 2026

ABSTRACT

Current LLM-based coding agents follow a serial execution paradigm: the model first generates the complete code, then invokes an interpreter to execute it. This sequential workflow leaves the executor idle during generation and the generator idle during execution, resulting in unnecessary end-to-end latency. We observe that, unlike human developers, LLMs produce code tokens sequentially without revision, making it possible to execute code as it is being generated. We formalize this **parallel execution** paradigm, modeling it as a three-stage pipeline of generation, detection, and execution, and derive closed-form latency bounds that characterize its speedup potential and operating regimes. We then present Eager, a concrete implementation featuring AST-based chunking, dynamic batching with gated execution, and early error interruption. We evaluate Eager across four benchmarks, seven LLMs, and three execution environments. Results show that Eager reduces the non-overlapped execution latency by up to 99.9% and the end-to-end latency by up to 55% across seven LLMs and four benchmarks.

ACM Reference Format:
Zhensu Sun, Zhihao Lin, Zhi Chen, Chengran Yang, Mingyi Zhou, Li Li, and David Lo. 2026. Executing as You Generate: Hiding Execution Latency in LLM Code Generation. In . ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

[*]Both authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

For human developers, the process of writing source code is inherently non-linear: they constantly navigate across different sections of a code file to revise, restructure, and refine their logic. Naturally, the code is executed only after it is deemed ready. In this paper, we refer to this write-then-execute paradigm as **serial execution**. This paradigm has been largely inherited by current LLMs for dealing with their tasks, such as file manipulation [18, 29], data processing [12], and code testing [27]. Specifically, a language model first generates a code block, invokes the code interpreter for execution, pauses to wait for the execution result, and then resumes generation conditioned on that result.

However, unlike human developers, modern token-based language models do not revise tokens once they are generated—each token is committed the moment it is produced. Inspired by this property, we propose a new code execution paradigm for LLMs, i.e., **parallel execution**, in which every code statement can be dispatched to the interpreter the moment it is produced, rather than waiting for the full block to be complete. This paradigm is naturally supported by interpreted programming languages such as Python and JavaScript, where code does not need to be compiled as a whole. Its benefit is intuitive: since generation and execution now overlap in time, the end-to-end latency is no longer the sum of the two phases. As illustrated in Figure 1, serial execution incurs a total wall-clock time of $T_{\text{gen}} + T_{\text{exec}}$, whereas parallel execution reduces this to approximately $T_{\text{gen}} + T_{\text{tail}}$, where T_{gen}, T_{exec}, and T_{tail} denote the generation time, the total execution time, and the execution time of only the final chunk, respectively.

To the best of our knowledge, the parallel paradigm between the execution and generation of LLM-produced code has not yet been explored in the research community. Prior work has proposed incremental execution strategies [18, 28, 31], where a model generates a few lines, executes them, and conditions subsequent generation on the observed output. For example, Open Interpreter [18] executes code in Jupyter-style cells and feeds the output back to the model, and EG-CFG [16] integrates real-time execution signals into the code generation process. These approaches use execution feedback

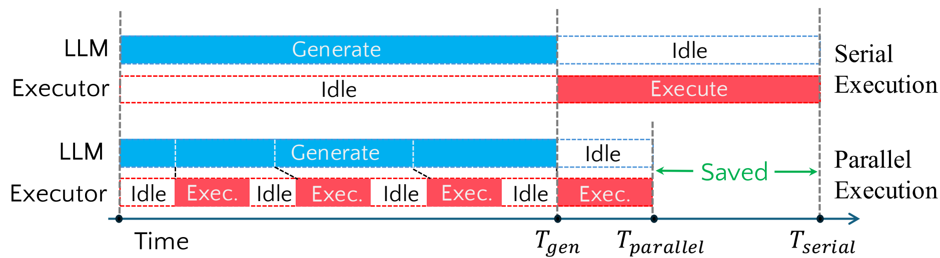

Figure 1: An illustrative example comparing Serial Execution and Parallel Execution. For a code snippet with four chunks, Parallel Execution overlaps the first three chunks with the generation process, saving the corresponding waiting time.

to improve code quality by steering subsequent generation. However, they are still fundamentally serial with respect to latency: the model must pause generation while awaiting each execution result, and the total wall-clock time includes both generation and execution in full. In contrast, the parallel execution paradigm we propose targets a complementary goal: reducing user-perceived latency by overlapping generation and execution, without altering the generated code. As a result, the following question remains open: Is parallel execution of LLM-generated code practically viable, and what are its benefits and costs?

Guided by this analysis, we present EAGER (**E**xecuting **A**s you **GE**ne**R**ate), a code execution framework that realizes this paradigm. We demonstrate this paradigm for Python, the dominant language in LLM code generation, though the principle applies to other interpreted languages. Its core design follows a producer–consumer pipeline, where the LLM acts as the producer and the executor as the consumer. As the language model generates tokens, an AST-based chunker incrementally identifies complete Python statements from the token stream and enqueues each chunk into a buffer. Concurrently, an executor dequeues and runs available chunks within a persistent interpreter session, preserving variable bindings across successive batches. The executor employs a dynamic batching strategy: when multiple chunks have accumulated in the queue, it merges them into a single batch, thereby amortizing the per-invocation setup overhead. Additionally, EAGER features an early error interruption mechanism: since chunks are executed as they are generated, a runtime error in any chunk is detected immediately, at which point EAGER terminates the LLM generation and returns the error along with the partially generated code. This avoids wasting generation time on code that depends on already-failed state.

We evaluate EAGER extensively across four code generation benchmarks, seven LLMs, and three execution environments. First, since the generation speed of real LLMs is difficult to control systematically, we use simulated token streams replayed at fixed rates (20 to 200 tokens per second) across three execution environments (local, Docker, and Open Interpreter [18]) to isolate the effect of generation speed and deployment setting on overlap efficiency. Under this controlled setting, EAGER hides 83–100% of execution time behind generation across all configurations, with end-to-end latency savings of up to 35% and consistent results across environments. Second, using real-time LLM outputs, EAGER reduces the non-overlapped execution latency to near zero across most settings, hiding over 95% of execution time behind generation. The end-to-end latency is reduced by up to 37% for error-free executions and up to 55% for error-encountered executions, where the early interruption mechanism avoids wasted generation on already-failed code. Third, we find that the early error interruption provided by EAGER not only reduces latency but also improves subsequent code repair success rates by up to 44 percentage points on data-centric benchmarks. When EAGER interrupts generation upon an error, the LLM receives only the code up to the failure point, rather than the complete but flawed program. This prevents the model from being anchored by the incorrect code generated after the error, giving it more freedom to produce a corrected solution.

In summary, this paper makes the following contributions:

- We formalize the parallel execution paradigm for LLM code generation, providing a theoretical framework that characterizes its latency bounds, speedup potential, and operating regimes.
- We present EAGER, a concrete implementation of parallel execution featuring AST-based chunking, dynamic batching with gated execution, and an early error interruption mechanism.
- We conduct a comprehensive empirical evaluation demonstrating that parallel execution consistently reduces latency across diverse benchmarks, models, and environments, while additionally benefiting code repair through earlier error feedback.

2 PARALLEL EXECUTION

In this section, we introduce the general workflow of parallel execution and theoretically analyze its latency.

2.1 Workflow

Current LLMs follow a strictly sequential paradigm for code execution: the model first generates the entire program and only then invokes the execution environment. This results in a clear temporal separation between generation and execution, leading to significant idle time on both sides.

We instead formulate the process as a *streaming pipeline*. The LLM acts as a *producer* that autoregressively emits tokens. A detection module continuously processes the token stream to identify executable chunks, defined as minimal syntactically complete and semantically executable units. Once a chunk is detected, it is immediately dispatched to an execution engine, which maintains a persistent session to preserve program state. This design enables

temporal overlap across all stages: while the LLM generates tokens for later chunks, earlier chunks can already be detected and executed.

2.2 Theoretical Modeling

We model the system as a three-stage pipeline consisting of generation, detection, and execution. The total latency is determined by the critical path through these stages.

Notation. We define:

- L: total number of tokens,
- v_{gen}: generation speed (tokens per second),
- T_{FT}: time-to-first-token,
- N: number of executable chunks,
- l_i: length of chunk i, with $\sum_{i=1}^{N} l_i = L$,
- δ_i: residual detection delay for chunk i (the portion of detection cost not hidden behind generation),
- T_{setup}: per-chunk execution overhead,
- $T_{exe,i}$: execution time of chunk i.

2.2.1 *Serial Execution.*

In the serial paradigm, the model first generates the complete program, after which the interpreter executes it as a monolithic block. The total latency is therefore:

$$T_{serial} = T_{FT} + \frac{L}{v_{gen}} + T_{setup}^{(\text{full})} + T_{exe}^{(\text{full})}, \tag{1}$$

where $T_{setup}^{(\text{full})}$ denotes the one-time execution setup cost for the complete program, and $T_{exe}^{(\text{full})}$ denotes the execution time of the full program.

For consistency with the chunked formulation, one may approximate:

$$T_{exe}^{(\text{full})} \approx \sum_{i=1}^{N} T_{exe,i}, \tag{2}$$

but importantly, the serial baseline does not incur repeated per-chunk setup overhead or streaming detection overhead.

2.2.2 *Parallel Execution.*

In the parallel paradigm, generation, detection, and execution are overlapped. Let $t_{g,i}$ denote the time when chunk i has been fully generated, and let $t_{e,i}$ denote the time when execution of chunk i completes.

Generation. The time at which chunk i has been fully generated depends on the cumulative token length of all preceding chunks:

$$t_{g,i} = T_{FT} + \frac{\sum_{j=1}^{i} l_j}{v_{gen}} \tag{3}$$

Detection. The detector processes the token stream online. We write the chunk-ready time as:

$$t_{d,i} = t_{g,i} + \delta_i \tag{4}$$

where δ_i is the residual detection delay, i.e., the portion of detection cost that is not hidden behind generation.

Execution. Chunk i can only start executing once it has been detected from the generated code and the previous chunk has finished executing:

$$t_{e,i} = \max(t_{d,i},\ t_{e,i-1}) + T_{setup} + T_{exe,i} \tag{5}$$

The overall parallel latency is the completion time of the final chunk:

$$T_{parallel} = t_{e,N} \tag{6}$$

Closed-form characterization. Unrolling the recurrence gives

$$T_{parallel} = \max_{1 \le i \le N} \left[t_{g,i} + \delta_i + \sum_{j=i}^{N} (T_{setup} + T_{exe,j}) \right] \tag{7}$$

2.2.3 *Latency Bounds.*

The closed-form expression of the parallel execution allows us to derive upper and lower bounds on the latency.

Upper bound. For any $i \in \{1, \ldots, N\}$, the generation prefix satisfies $\sum_{j=1}^{i} l_j \le L$, the detection residual satisfies $\delta_i \le \bar{\delta} \triangleq \max_{1 \le k \le N} \delta_k$, and the execution tail satisfies $\sum_{j=i}^{N}(T_{setup} + T_{exe,j}) \le \sum_{j=1}^{N}(T_{setup} + T_{exe,j})$. Applying these to every term inside the outer maximum yields

$$T_{parallel} \le T_{FT} + \frac{L}{v_{gen}} + \bar{\delta} + N\,T_{setup} + \sum_{j=1}^{N} T_{exe,j} \tag{8}$$

This upper bound corresponds to a *zero-overlap* execution in which every stage waits for its predecessor to complete entirely. Compared with the serial baseline, the additional cost is at most

$$T_{parallel} - T_{serial} \le \bar{\delta} + N\,T_{setup} - T_{setup}^{(\text{full})} \tag{9}$$

, which captures the overhead from (i) streaming detection and (ii) repeated per-chunk setup. In practice $\bar{\delta}$ is on the order of milliseconds because the detector operates on a lightweight grammar, so the overhead is dominated by the cumulative setup cost $N\,T_{setup} - T_{setup}^{(\text{full})}$. When both $\bar{\delta}$ and $N\,T_{setup} - T_{setup}^{(\text{full})}$ are negligible, the upper bound reduces to T_{serial}, showing that the parallel scheme introduces no regression under these conditions.

Lower bound. Two structural constraints yield complementary lower bounds.

(1) **Generation constraint** (setting $i = N$). The system must generate all tokens before the last chunk can complete:

$$T_{parallel} \ge T_{FT} + \frac{L}{v_{gen}} + \delta_N + T_{setup} + T_{exe,N} \tag{10}$$

(2) **Execution constraint** (setting $i = 1$). The system must execute all chunks in order after the first chunk becomes available:

$$T_{parallel} \ge T_{FT} + \frac{l_1}{v_{gen}} + \delta_1 + N\,T_{setup} + \sum_{j=1}^{N} T_{exe,j} \tag{11}$$

Combining both gives the composite lower bound:

$$T_{parallel} \ge \max \left\{ \begin{array}{l} T_{FT} + \frac{L}{v_{gen}} + \delta_N + T_{setup} + T_{exe,N}, \\ T_{FT} + \frac{l_1}{v_{gen}} + \delta_1 + N\,T_{setup} + \sum_{j=1}^{N} T_{exe,j} \end{array} \right\} \tag{12}$$

The first term dominates when generation is the bottleneck (*generation-dominated regime*); the second dominates when execution is the bottleneck (*execution-dominated regime*).

2.2.4 Speedup Bounds. Define the speedup $S = T_{serial}/T_{parallel}$. From the lower bound (10) on $T_{parallel}$, the speedup is at most:

$$S \leq \frac{T_{FT} + \frac{L}{v_{gen}} + T_{setup}^{(\text{full})} + T_{exe}^{(\text{full})}}{T_{FT} + \frac{L}{v_{gen}} + \delta_N + T_{setup} + T_{exe,N}} \tag{13}$$

when δ_N and $T_{exe,N}$ are small relative to L/v_{gen}, which simplifies to $S \lesssim 1 + (T_{setup}^{(\text{full})} + T_{exe}^{(\text{full})})/(T_{FT} + L/v_{gen})$. This is achieved when all execution is hidden behind generation. From the upper bound (8), $S \geq 1$ whenever $T_{setup}^{(\text{full})} \geq \bar{\delta} + N\,T_{setup}$, i.e., the one-time serial setup cost exceeds the cumulative chunk overhead. Whether this condition holds depends on the execution environment. In our experimental setup, where chunks are dispatched to a persistent REPL session with per-call overhead on the order of 1 ms, T_{setup} and $\bar{\delta}$ are both small enough for the condition to be satisfied comfortably. However, in environments with heavier per-chunk orchestration costs (e.g., container cold starts or cross-process IPC), the cumulative term $N\,T_{setup}$ may become non-negligible, and the condition should be verified empirically.

2.2.5 Regime Analysis under Uniform Chunks. To obtain sharper insight, we analyze the special case of *uniform chunks*: $l_i = L/N$, $T_{exe,i} = \tau_e$, and $\delta_i = \delta$. Define the per-chunk generation time $\alpha \triangleq L/(N\,v_{gen})$ and per-chunk execution time $\beta \triangleq T_{setup} + \tau_e$. The inner term of the closed-form expression becomes affine in i:

$$f(i) = \underbrace{T_{FT} + \delta + (N+1)\beta}_{\text{constant}} + i\,(\alpha - \beta) \tag{14}$$

yielding the following three regimes:

R1: Generation-dominated ($\alpha > \beta$). The maximum is at $i = N$:

$$T_{parallel} = T_{FT} + \frac{L}{v_{gen}} + \delta + \beta \tag{15}$$

Execution of every chunk except the last is entirely hidden behind generation.

R2: Execution-dominated ($\alpha < \beta$). The maximum is at $i = 1$:

$$T_{parallel} = T_{FT} + \alpha + \delta + N\beta \tag{16}$$

Generation of every chunk except the first is hidden behind execution.

R3: Balanced ($\alpha = \beta$). The pipeline is perfectly paced. Setting $\alpha = \beta$, i.e., $L/(N\,v_{gen}) = T_{setup} + T_{exe}^{(\text{full})}/N$, and solving for N gives the critical chunk count:

$$N^* = \frac{L/v_{gen} - T_{exe}^{(\text{full})}}{T_{setup}} \tag{17}$$

which is positive whenever total generation time exceeds total execution time, the common case for LLM code generation. For $N \leq N^*$ additional chunks improve overlap; beyond N^* the cumulative setup overhead $N\,T_{setup}$ dominates and latency degrades.

3 IMPLEMENTATION

Building on the theoretical framework in Section 2.2, we present Eager, a concrete implementation of parallel execution for LLM code generation. Eager instantiates the pipeline with design choices aimed at minimizing the detection and per-chunk setup overhead identified in Section 2.2, without affecting execution outcomes.

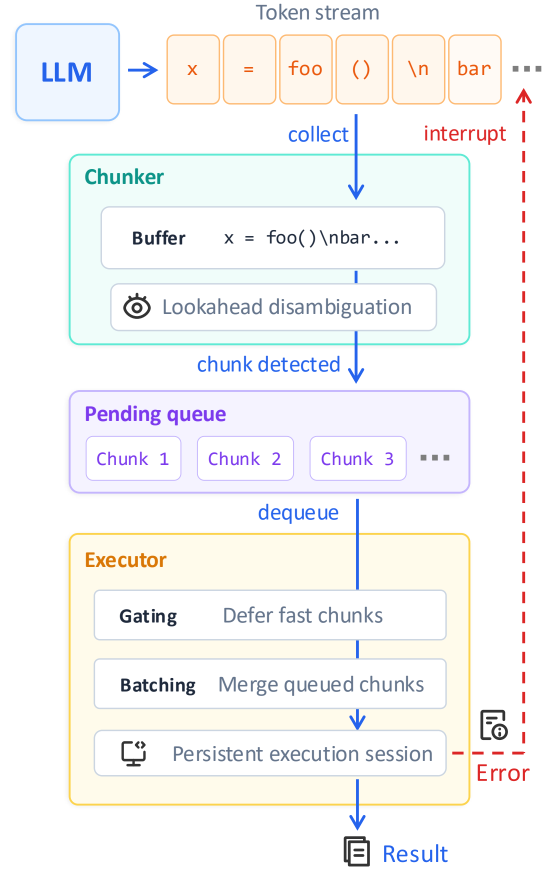

Figure 2: Architecture of Eager.

As illustrated in Figure 2, Eager consists of a chunker and an executor. *Chunker* accumulates the streaming tokens generated by LLMs in a buffer and identifies complete Python statements via AST parsing. Detected chunks are dispatched to a *pending queue*, from which an *executor* dequeues, applies gating and batching optimizations, and delegates execution to a persistent session. If a runtime error is encountered, the executor sends an interrupt signal back to terminate the LLM generation immediately (the dashed path in Figure 2). We describe each component in detail below.

3.1 Producer: AST-Based Chunker

The chunker operates on the streaming token output from an LLM and identifies executable chunks incrementally. We implement the chunker based on AST statement boundaries. As each token arrives, it is appended to a code buffer, and the chunker attempts to parse the buffer into a Python AST. A chunk boundary is recognized when the buffer forms a complete top-level statement (e.g., assignments, expressions, loops, or function calls). A statement is considered *complete* when it can be unambiguously determined that no further tokens will be generated as part of it—that is, the statement has no remaining portions yet to be produced by the LLM. Once a chunk is

confirmed as complete, the corresponding statements are removed from the buffer and dispatched to the executor, while any remaining tokens stay in the buffer for subsequent detection.

In many cases, syntactic completeness directly implies completeness. For example, upon receiving `print("hello")` followed by a newline, the chunker can immediately confirm this as a standalone statement and dispatch it. However, in other cases, a syntactically valid statement may still have remaining portions to be generated. Consider the following scenario where tokens arrive incrementally:

```
def foo():
    print(1)
```

After receiving `print(1)` and a newline, the function body parses as syntactically valid. Yet the next line could continue the function body with additional statements, in which case the definition is not yet complete. To handle such ambiguities, Eager employs a *lookahead* strategy: when a statement parses successfully but its completeness cannot be definitively confirmed from syntax alone, the chunker waits for one additional token before committing. If the next token rules out the possibility of the current statement continuing (e.g., by beginning a new top-level statement or indicating a dedent), the chunk is finalized and dispatched; otherwise, the buffer continues to accumulate. This mechanism ensures that dispatched chunks are both syntactically complete and semantically independent.

3.2 Consumer: Gated Executor with Dynamic Batching

The executor acts as the consumer of the pipeline, receiving chunks dispatched by the chunker. It interacts with a persistent execution session, whether a local Python subprocess, a sandboxed environment, or a Docker container, that preserves all imports, variable bindings, and function definitions across successive chunk executions. This execution session is therefore responsible for the execution of the chunks.

Confirmed chunks are placed into a pending queue. When the executor becomes available, it merges all currently pending chunks into a single big chunk, rather than executing them one by one. This *dynamic batching* naturally adapts to the pace difference between the chunker and the executor: when execution is slower than detection, more chunks accumulate and are batched together, reducing the effective number of executor invocations and thus the cumulative setup cost $N\,T_{setup}$ in Equation 8. This also makes the speedup condition $S \geq 1$ easier to satisfy, since the overhead term $N\,T_{setup} - T_{setup}^{(\text{full})}$ in Equation 9 shrinks accordingly.

On top of batching, the executor applies a *gating* policy to skip chunks whose execution time is negligible compared to the per-invocation setup overhead. Typical examples are chunks consisting of function or class declarations, which produce no observable computation when executed in isolation. For such chunks, the setup cost T_{setup} dominates the actual execution time $T_{exe,i} \approx 0$, making individual execution wasteful, where each invocation adds overhead without contributing to useful overlap. The executor therefore defers these chunks: they remain in the pending queue and are merged with the next non-deferred chunk, at which point the declarations become available to support subsequent code that depends on them.

3.3 Error Handling in the Pipeline

The preceding subsections describe the normal flow of the pipeline: the chunker detects and dispatches chunks, and the executor batches and executes them. We now describe how the pipeline handles runtime errors.

In serial execution, runtime errors are only observed after the entire program has been generated and executed as a whole. In Eager, since chunks are executed incrementally alongside generation, a runtime error is caught as soon as the failing chunk finishes execution. At that point, Eager immediately terminates the LLM generation process. The error message, together with the code generated up to the point of failure, is returned to the caller. No further tokens are generated or executed.

This early interruption provides two benefits. First, it avoids spending generation time on code that depends on already-failed state, effectively reducing both the token count L and chunk count N on failure paths and thus lowering end-to-end latency. Second, it delivers error feedback at the earliest possible point, enabling a tighter repair loop—as we empirically validate in RQ3 (Section 5.4), this earlier feedback leads to higher repair success rates in most scenarios.

Notably, this early interruption is an optional feature. Eager can also be configured to continue generation after an error is detected and defer the error report until the full program has been produced. Under this configuration, Eager still provides latency savings through overlapped execution of the error-free prefix, while preserving the same complete-code error semantics as serial execution.

4 EXPERIMENT SETUP

Using Eager as a demonstration, we experimentally assess the performance of parallel execution. In this section, we will introduce the settings of the experiments, including the benchmarks, LLMs, execution environments, and implementation details. The rationale behind our setup is driven by answering the following three research questions:

- **RQ1:** How much latency can Eager save compared to serial execution across different token generation speeds and execution environments?
- **RQ2:** Do the latency savings generalize from simulated to real LLM code generation?
- **RQ3:** Does the earlier error feedback provided by Eager help or hinder LLMs' subsequent code repair?

4.1 Benchmarks

We use four benchmarks where the LLMs are required to produce executable Python scripts. These benchmarks cover various task scenarios including **DSBench** [15] and **DABench** [14] for data analysis, **PandasPlotBench** [10] for data visualization, and **GitChameleon** [19] for version-specific code generation:

- **DSBench** and **DABench** are both benchmarks that evaluate LLMs on data analysis tasks, where the model is given tabular data files along with natural language questions and must generate executable Python scripts to derive the answers. DSBench is sourced from Modeloff financial analysis competitions and Kaggle challenges, featuring 442 data analysis tasks with realistic

settings such as long task descriptions, multimodal backgrounds, and multi-table structures. DABench provides 257 data analysis questions derived from real-world CSV files crawled from GitHub.
- **PandasPlotBench** is a human-curated benchmark for evaluating LLMs' ability to generate data visualization code. It contains 175 tasks, each requiring the model to produce plotting code given a natural language description and a Pandas DataFrame specification. The tasks are derived from the Matplotlib gallery and cover a range of chart types.
- **GitChameleon** is a manually curated benchmark that evaluates LLMs' ability to generate version-specific Python code. It contains 116 code completion problems, each conditioned on a particular library version and accompanied by executable unit tests. Unlike conventional code generation benchmarks, GitChameleon specifically tests whether models can produce code compatible with a specified version of a library, reflecting real-world scenarios where developers are constrained to specific dependency versions.

4.2 Large Language Models

We evaluate Eager across seven LLMs spanning both open-weight and proprietary models, covering a range of model scales, speeds, and architectures.

For open-weight models, we use DeepSeek-V3.2 [6], a 685B-parameter MoE model with sparse attention; MiMo-V2-Flash [26], Xiaomi's 309B MoE model (15B active) optimized for fast inference via hybrid sliding-window attention and multi-token prediction; Qwen3-Coder [25], Alibaba's 480B MoE coding model (35B active) trained with long-horizon reinforcement learning for agentic coding; and DeepSeek-Reasoner [5], a reasoning-focused model that employs extended chain-of-thought during inference.

For proprietary models, we use GPT-4o-mini [21], OpenAI's lightweight multimodal model; GPT-5.1-Codex-Mini [22], a variant of GPT-5.1-Codex further optimized for agentic coding tasks; and Gemini-3.1-Flash-Lite [4], Google's model in the Gemini 3 series designed for high-volume, low-latency workloads.

4.3 Execution Environments

As the LLM-generated code could be executed in various environments in practice, we experiment with three mainstream options:
- **Local execution** runs the generated Python scripts directly on the host machine, offering the lowest overhead and fastest startup time. This represents the simplest setup commonly used in lightweight scripting and prototyping workflows.
- **Docker-based execution** runs the scripts inside an isolated Docker container [8]. This provides a reproducible and sandboxed environment that prevents unintended side effects on the host system, and is widely adopted in production-grade agent frameworks for dependency management and security.
- **Open Interpreter sandbox** executes code through the Open Interpreter [3] framework, which provides a managed sandboxed environment with built-in support for LLM-driven code execution. It represents a higher-level abstraction where the execution runtime is integrated into an end-to-end agent pipeline.

These three environments exhibit different levels of isolation and startup overhead, allowing us to evaluate whether the latency savings of Eager generalize across practical deployment settings. All environments are configured with 2 CPU cores to reflect the resource-constrained settings typical of cloud-hosted code execution sandboxes.

4.4 Evaluation Metrics

We use two metrics to quantify the latency impact of Eager:
- **Non-overlapped Execution Latency (NEL)** measures the portion of code execution time that falls outside the LLM generation phase. In serial execution, the entire execution time is non-overlapped, as execution begins only after generation completes. In parallel execution with Eager, part of the execution overlaps with the ongoing token generation, and this metric captures only the remaining portion that still contributes to user-perceived delay. A lower non-overlapped execution time indicates that more execution has been effectively hidden behind generation.
- **End-to-End Latency (E2EL)** measures the wall-clock time from the start of the LLM call to the completion of code execution, i.e., the total time the user must wait. It equals the sum of the LLM generation time and the non-overlapped execution time. This metric directly reflects the delay perceived by the user.

4.5 Implementation Details

All experiments are conducted on a server running Ubuntu 22.04 with an Intel Xeon Platinum 8352V processor. To ensure reliable and reproducible timing measurements, we apply single-CPU isolation via `taskset` for local and Open Interpreter runs, and CPU pinning (`--cpuset-cpus`) for Docker-based runs. For code execution, the local and Docker environments use a persistent Python REPL subprocess that preserves imports, variables, and definitions across chunks, with per-call overhead of approximately 1 ms. The Open Interpreter environment uses its native Jupyter-kernel backend as the executor. All experiments use isolated, task-local executors to prevent cross-task interference. The LLMs are accessed through OpenRouter APIs [23] via streaming mode. All model names used throughout the paper correspond to their official API identifiers as of March 2026.

5 RESULTS

In this section, we report our experimental results and answer the three research questions.

5.1 Preliminary: Chunk Reconstruction Fidelity

Before measuring the latency benefits of Eager, we first validate that its chunking mechanism preserves program integrity. We collect the code generated by all seven LLMs across the four benchmarks (DABench, DSBench, PandasPlotBench, and GitChameleon) and run each program through Eager's chunking pipeline under the Docker environment. For every program, we concatenate the emitted chunks and compare the result against the original generated code. Across all programs and all seven models, the reassembled code is character-level identical to the original in every case. This confirms that the AST-based chunker correctly identifies statement

Table 1: Mock-token latency savings (%) of EAGER over serial execution across four benchmarks under Docker. NEL (Non-overlapped Execution Latency) saving measures the reduction in execution time that does not overlap with generation; E2EL (End-to-End Latency) saving measures the reduction in total wall-clock time.

		DABench		DSBench		PdPlotBench		GitCham.	
Env.	TPS	NEL	E2EL	NEL	E2EL	NEL	E2EL	NEL	E2EL
Docker	20	94.3	2.1	96.7	1.1	88.9	6.1	100	4.2
Docker	50	94.3	5.1	96.1	2.7	88.4	13.7	100	8.4
Docker	100	94.4	9.6	95.0	5.1	88.0	23.4	97.3	12.9
Docker	200	93.9	17.4	91.7	9.3	83.4	34.9	87.9	20.7
Local	20	93.8	2.2	96.2	1.1	88.9	6.3	100	5.9
Local	50	93.9	5.3	95.5	2.8	88.2	14.2	99.3	10.6
Local	100	94.0	10.3	94.5	5.4	87.8	24.5	93.1	17.5
Local	200	91.6	18.0	90.2	10.2	81.1	35.3	77.2	25.0
OI	20	94.3	3.2	95.9	1.9	85.3	6.5	100	12.2
OI	50	94.4	7.8	95.3	4.4	82.8	14.5	97.4	23.6
OI	100	94.3	14.6	94.7	8.3	81.6	24.3	92.3	36.2
OI	200	90.6	23.9	92.0	15.0	73.3	34.0	86.8	49.1

boundaries without dropping, duplicating, or reordering any tokens, regardless of the generating model.

Since EAGER executes chunks in sequence within a single persistent session, carrying forward all imports, variable bindings, and function definitions across chunk boundaries without any concurrency or re-initialization, lossless reconstruction directly implies execution equivalence for deterministic programs, which constitute the all the LLM-generated code on our benchmarks.

5.2 RQ1: Latency Savings Across Generation Speeds and Environments

As the generation speed of real LLMs are hard to systematically control, we choose a simulated setting. Instead of using real LLMs to produce the token stream for EAGER to execute, we use the already generated code solutions from all four benchmarks and replay them as a mock token stream at a fixed Token-Per-Second (TPS) rate. Specifically, we use the gold (reference) solutions from PandasPlotBench and GitChameleon and the runnable solutions generated by DeepSeek-V3.2 for DABench and DSBench. We sweep across four representative TPS rates—20, 50, 100, and 200—covering the range from slower open-weight model deployments to fast proprietary APIs. We run all these code scripts across the three execution environments (local, Docker, and Open Interpreter) and compare EAGER against serial execution in terms of non-overlapped execution time and end-to-end latency.

As shown in Table 1, EAGER achieves consistently high NEL savings across all benchmarks, environments, and TPS rates, with reductions typically in the range of 83–100%. This indicates that the vast majority of execution time is successfully hidden behind generation regardless of the specific configuration. For example, on DABench and DSBench, the NEL savings remain above 90% even at 200 TPS across all three environments, meaning that less than 10% of the original execution time is left exposed after overlapping. GitChameleon achieves 100% NEL savings at 20 TPS under Docker, indicating complete overlap between generation and execution at lower generation speeds.

The E2EL savings, while more modest, follow a clear trend: they increase as generation speed decreases. At 20 TPS, end-to-end savings are relatively small (1–6% across benchmarks) because generation time dominates and the absolute execution time being hidden is small relative to the total. At 200 TPS, the savings grow substantially (up to 35% on PandasPlotBench under Docker), as faster generation compresses the generation window and makes execution a larger fraction of the total latency. This is consistent with the theoretical prediction that the E2EL benefit of parallel execution scales with the ratio of execution time to generation time.

Across the three environments, the savings are broadly consistent, confirming that EAGER generalizes across deployment settings. Docker and Local show similar results, with Docker having a slight edge due to more stable timing from CPU pinning. Open Interpreter exhibits marginally lower savings, particularly on PandasPlotBench and GitChameleon at higher TPS rates (e.g., 73.3% NEL on PandasPlotBench at 200 TPS vs. 83.4% under Docker), which we attribute to the additional overhead of the Jupyter-kernel execution backend.

Answer to RQ1: EAGER hides 83–100% of execution time behind generation across all tested configurations. The NEL savings are robust to changes in generation speed and execution environment, while the E2EL savings increase as generation speed grows, reaching up to 35% at 200 TPS.

5.3 RQ2: Generalization to Real LLM Code Generation

To validate whether the savings observed in RQ1 generalize to real-world settings, we use each of the seven LLMs to generate code for the four benchmark tasks via streaming. The generated code is executed with EAGER in the Docker environment, and the resulting latency is compared against the serial execution baseline. Notably, when a runtime error occurs during execution, EAGER interrupts the code generation process immediately. We therefore report results separately for error-free and error-encountered executions.

As shown in Table 2, EAGER consistently reduces latency across nearly all model–benchmark combinations under both conditions. For error-free executions, the non-overlapped execution latency (NEL) drops to near zero in many cases (e.g., 2 ms for DeepSeek-V3.2 on DABench, 1 ms for GPT-4o-mini on DABench), indicating that execution is almost entirely hidden behind generation for these settings. The end-to-end latency (E2EL) improves correspondingly, with reductions ranging from modest savings on benchmarks where execution is already fast (e.g., DSBench with GPT-5.1-Codex, where the baseline NEL is only 62 ms) to substantial reductions where execution constitutes a larger fraction of total latency (e.g., Gemini-3.1 on PandasPlotBench, from 1440 ms to 903 ms, a 37.3% reduction). Figure 3 illustrates this effect on a concrete DABench task generated by DeepSeek-V3.2: most of the execution chunks (green bars) fall within the generation window, reducing the end-to-end latency from 8909 ms to 8561 ms. The savings are most pronounced for models with slower generation speeds, where the longer generation window provides more room to overlap execution—consistent

Table 2: Latency comparison (ms) between serial execution (Baseline) and EAGER (EAGER) across seven LLMs on Docker. NEL (Non-overlapped Execution Latency) measures time spent on execution that does not overlap with generation; E2EL (End-to-End Latency) measures total wall-clock time. Results are grouped by whether the generated code produces a runtime error. For GitChameleon error cases, EAGER NEL is marked "—" because EAGER terminates generation upon detecting the error, leaving no post-generation execution phase.

		Error-free (ms)								Error-encountered (ms)							
		DABench		DSBench		PdPlotBench		GitCham.		DABench		DSBench		PdPlotBench		GitCham.	
Model	Exec.	NEL	E2EL	NEL	E2EL	NEL	E2EL	NEL	E2EL	NEL	E2EL	NEL	E2EL	NEL	E2EL	NEL	E2EL
DeepSeek-V3.2	Baseline	590	8467	256	28069	912	11182	357	14480	638	9799	669	30362	678	15297	290	10161
	EAGER	2	7879	16	27816	199	10470	16	14140	0	3957	362	16547	0	7257	—	8432
GPT-4o-mini	Baseline	607	3999	286	4887	759	5558	321	3411	660	4991	1775	8264	811	6646	362	5766
	EAGER	1	3393	78	4679	54	4853	14	3104	0	1829	1447	4159	33	2377	—	4936
MiMo-V2-Flash	Baseline	581	4082	272	35090	768	5004	370	3014	658	4457	1625	14859	513	5983	424	4801
	EAGER	12	3513	29	34845	94	4330	45	2689	5	2565	1316	6492	6	2610	—	3809
Qwen3-Coder	Baseline	628	2858	156	7458	774	5347	358	4819	696	4317	675	13163	675	5169	686	5247
	EAGER	80	2310	3	7296	81	4642	39	4500	85	1938	342	3289	0	2225	—	3930
GPT-5.1-Codex	Baseline	594	1737	62	479	791	2244	344	1868	631	1786	1259	6890	737	2912	429	2076
	EAGER	88	1231	34	451	190	1643	82	1607	127	832	979	6384	28	1515	—	1107
Gemini-3.1	Baseline	583	1325	135	1381	753	1440	361	1614	624	1546	259	1670	719	1558	378	1743
	EAGER	167	909	9	1255	215	903	65	1317	163	758	18	783	103	786	—	1035
DeepSeek-R	Baseline	589	6073	153	15324	763	8467	416	7352	661	7661	997	23680	681	10326	467	9994
	EAGER	1	5476	17	15174	43	7742	32	6967	0	2934	699	10488	1	4619	—	5243

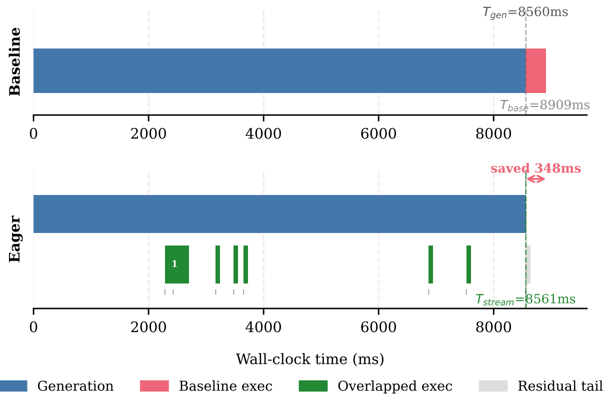

Figure 3: A real-world example of EAGER on a DABench task (id: dabench_19) generated by DeepSeek-V3.2. The baseline (serial execution) completes in 8909 ms (T_{gen} = 8560 ms + execution = 349 ms), while EAGER overlaps most of execution chunks with generation, finishing in 8561 ms and saving 348 ms. The green bars indicate individual chunk executions running concurrently with token generation.

with the generation-dominated regime identified in the theoretical analysis (Section 2.2).

For error-encountered executions, EAGER provides even larger latency reductions, as the early interruption mechanism eliminates both the remaining generation time and the execution of subsequent code that would inevitably depend on already-failed state. This is most visible in the E2EL: DeepSeek-V3.2 on DSBench drops from 30362 ms to 16547 ms (a 45.5% reduction), and DeepSeek-R on PandasPlotBench drops from 10326 ms to 4619 ms (a 55.3% reduction). The NEL for error cases is often 0 ms on benchmarks like DABench and PandasPlotBench, meaning the error was caught entirely within the generation window. For GitChameleon error cases, the NEL under EAGER is undefined because generation is terminated at the point of error, so only E2EL is reported. The E2EL nonetheless shows clear improvements across all models (e.g., DeepSeek-R from 9994 ms to 5243 ms), confirming that early interruption saves substantial waiting time even when the post-generation tail cannot be measured.

> **Answer to RQ2:** The latency savings observed under simulated conditions in RQ1 generalize to real LLM outputs across all seven models and four benchmarks. For error-free executions, EAGER reduces the NEL to near zero in most settings, with E2EL reductions of up to 37.3%. For error-encountered executions, the early interruption mechanism provides even larger savings, reducing E2EL by up to 55.3%.

5.4 RQ3: Effect of Earlier Error Feedback on Code Repair

As observed in RQ2, EAGER catches errors during generation rather than after it completes, providing earlier feedback to the LLM. This early interruption also terminates the code generation process, leaving the remaining code ungenerated. In this RQ, we investigate whether the partial code resulting from early error interruption affects the model's ability to resolve the detected error in subsequent repair attempts. Specifically, for each error-encountered sample in RQ2, we append the error message to the already-generated code

Table 3: Error resolution rates (%) for repairing from partially generated code (Partial) versus complete code (Full) across four benchmarks. Δ denotes the percentage-point difference.

	DABench			DSBench			PandasPlotBench			GitChameleon		
Model	Full	Partial	Δ	Full	Partial	Δ	Full	Partial	Δ	Full	Partial	Δ
DeepSeek-V3.2	48.4	64.5	+16.1	71.3	74.4	+3.0	61.5	69.2	+7.7	66.7	64.9	-1.8
GPT-4o-mini	67.1	74.7	+7.6	32.3	76.6	**+44.3**	64.8	74.1	+9.3	34.2	35.4	+1.3
MiMo-V2-Flash	57.9	71.1	+13.2	51.8	62.8	+10.9	52.6	68.4	+15.8	46.5	43.7	-2.8
Qwen3-Coder	37.1	69.4	**+32.3**	60.3	78.7	+18.4	58.3	83.3	**+25.0**	51.0	49.0	-2.0
GPT-5.1-Codex	78.7	80.9	+2.1	31.0	36.1	+5.1	81.8	86.4	+4.5	82.5	67.5	-15.0
Gemini-3.1	43.4	75.5	**+32.1**	70.0	89.1	+19.1	78.4	86.5	+8.1	58.3	58.3	+0.0
DeepSeek-R	59.0	77.0	+18.0	53.5	68.6	+15.1	73.7	94.7	**+21.1**	67.9	58.5	-9.4

and let the same model continue generating a fix. We compare two conditions (**Full** and **Partial**): under serial execution, the model continues with the complete code followed by the error message; under Eager, the model continues with the partially generated code followed by the error message. We then compare the error resolution rates between the two conditions—that is, whether the repaired code executes without reproducing the original runtime error. As reported in Table 3, across the three data-centric benchmarks (DABench, DSBench, and PandasPlotBench), repairing from partial code consistently achieves higher error resolution rates than repairing from full code, with improvements ranging from +2.1 to +44.3 percentage points. The gains are particularly pronounced for models such as Qwen3-Coder and Gemini-3.1, which achieve over 30 percentage points of improvement on DABench and GPT-4o-mini, which gains +44.3 percentage points on DSBench. This result may seem counterintuitive at first, as one might expect that having the complete code would provide the model with more context for repair. However, we hypothesize that the full program, which has already executed to completion and failed, may anchor the model toward preserving its original (flawed) logic. In contrast, the partial code from Eager leaves the remainder ungenerated, giving the model more freedom to regenerate a corrected solution from the point of failure. Moreover, we observe that on the three data-centric benchmarks, errors tend to occur early in the generated code, at a median position of around 35% of the total code length. This means that under serial execution, on average 65% of the code is generated *after* the error has already occurred, without any awareness of the runtime failure. This post-error suffix often depends on state that no longer holds (e.g., referencing a DataFrame column that caused a `KeyError`), providing the model with misleading context during repair. By contrast, Eager interrupts generation at the point of failure, avoiding this misleading suffix entirely. The exception is GitChameleon, where repairing from partial code slightly underperforms in most cases (up to −15.0 percentage points for GPT-5.1-Codex). We attribute this to the nature of GitChameleon tasks: they are version-specific code completion problems where the error typically stems from using an incorrect API for the target library version. In these tasks, the code after the error point often encodes the *intended* API usage, which is precisely the information needed to diagnose a version mismatch. To investigate further, we analyzed the 27 cases (across all models) where full-code repair resolved the error but partial-code repair did not: 63% of these cases had their code truncated by early interruption (tokens saved > 0%), confirming that the lost suffix contained error-relevant context. The remaining 37% showed identical input code, where the difference is attributable to LLM repair stochasticity.

> **Answer to RQ3:** The early interruption behavior of Eager is not only beneficial for reducing latency (as shown in RQ1 and RQ2) but also advantageous for subsequent code repair. On the three data-centric benchmarks, repairing from partial code improves error resolution rates by 2.1 to 44.3 percentage points over repairing from the complete program. The exception is GitChameleon (−3.7 pp on average), where the truncated suffix contains version-specific API context needed for repair.

6 THREATS TO VALIDITY

Language generalizability. Our implementation and evaluation focus on Python, the dominant language for LLM code generation tasks. The core idea of parallel execution, overlapping generation with incremental execution, is language-agnostic and poses no theoretical barrier to adoption in other interpreted languages (e.g., JavaScript, R) that support similar REPL-based execution. Extending to statically typed or compiled languages may require additional consideration for compilation overhead, but the general pipeline design remains applicable.

Benchmark representativeness. Our evaluation covers four benchmarks spanning data analysis, data visualization, and version-specific code generation, which represent common use cases of LLM code interpreters in practice. These benchmarks primarily involve short-to-medium-length scripts. Workloads with substantially different characteristics, such as long-running computational programs or multi-file projects, may exhibit different overlap dynamics and are not covered in our current evaluation.

Docker cold-start outliers. In the Docker environment, the tasks in GitChameleon require creating isolated virtual environments for version-specific library testing. A small number of tasks (46 out of the total) exhibited abnormally high execution times due to virtual environment cold-start overhead. We exclude these outliers from our reported results. Removing this filter changes the aggregate results by less than 1 percentage point, confirming that our conclusions are robust to this decision.

7 RELATED WORK

7.1 Code Generation with Execution in the Loop

Recent LLM work has used code execution as an important mechanism for reasoning and interaction. For example, program-aided reasoning methods such as PAL [11] and Program of Thoughts [1] usually let the model first generate a complete program and only then execute it, using code mainly as a reliable substrate for arithmetic or symbolic computation rather than as a source of fine-grained feedback during decoding . Related language-to-code approaches in the same vein also largely follow this generate-then-execute pattern [24]. Follow-up work preserved this basic structure while making the outer loop more execution-aware. LEVER uses execution outcomes to verify and rerank candidate programs [20], while Chain of Code augments executable code with selective emulation when some steps cannot be directly executed [17]. In parallel, self-correction systems such as Self-Debugging [2], CYCLE [7], and runtime-verification-based debugging methods [34] treat execution errors, failed tests, or traces as signals for iterative repair after an initial solution has already been produced. More recently, EG-CFG [16] test the code line-by-line as it is being written, using real-time execution feedback to steer the model toward functionally correct and executable solutions. Our work is orthogonal to these efforts: rather than improving code quality, we reduce user-perceived latency by overlapping generation and execution, a technique that composes naturally with any existing code generation or repair strategy.

7.2 Execution Environments for LLMs

Recent LLM systems have also developed increasingly capable execution substrates, including interactive interpreters, notebook-style runtimes, and containerized sandboxes that make generated code executable, stateful, and reproducible across turns. OpenCodeInterpreter integrates code generation with execution and iterative refinement in a code-interpreter-style workflow [33], while InterCode exposes execution as part of the task environment through self-contained Docker sandboxes and standardized feedback channels [30]. Concurrently, infrastructure-oriented work such as MPLSandbox emphasizes practical multi-language isolation and unified compiler or runtime feedback for LLM-based coding systems [9], and notebook-centric agents and benchmarks further show the importance of maintaining persistent execution state in data analysis settings [13, 32]. These systems make execution available to the model, but their main goal is to provide a reliable runtime for interaction, evaluation, or iterative repair, rather than to minimize end-to-end response time. In contrast, our focus is not on building a richer sandbox or stronger debugging loop, but on a systems-level scheduling question: how to overlap generation and execution so that runtime feedback can be exploited without forcing the user to wait for a strictly sequential generate-then-run pipeline.

8 DISCUSSION

Implications for programming language design. Existing programming languages were designed under the assumption that source code is written in its entirety before execution. However, LLM-based code generation fundamentally changes this assumption: code is produced token by token as a stream, with each prefix potentially forming a meaningful partial program. This mismatch forces systems like EAGER to reconstruct executability from a stream that the language was never designed to support incrementally—relying on AST parsing heuristics, lookahead strategies, and gating policies to determine when a partial program is safe and worthy to execute. As LLM-generated code becomes an increasingly prevalent mode of program creation, we see an opportunity for future programming languages to treat *streamability* as a first-class design goal: for instance, through explicit statement delimiters that eliminate boundary ambiguity, or through language-level primitives that allow the runtime to consume and execute code incrementally as it is produced. Such designs would reduce the complexity of systems like EAGER and more broadly benefit the emerging ecosystem of AI-assisted programming.

When does parallel execution help most? The theoretical analysis in Section 2.2 identifies two key factors that determine the benefit of parallel execution: the ratio between generation time and execution time, and the per-chunk setup overhead. In practice, the largest latency savings occur in the *generation-dominated regime*, where the LLM generates code slowly relative to execution speed, leaving large room to hide execution behind generation. This is the common case for today's LLM serving: most code generated by LLMs involves lightweight data processing or API calls that execute in milliseconds, while generation itself takes seconds. Conversely, the benefit diminishes for workloads dominated by heavy computation (e.g., large-scale numerical simulations), where execution time far exceeds generation time and little can be overlapped. Even in such cases, parallel execution does not regress beyond serial execution, as the overhead is bounded by Equation 9, which is small in our experiments.

Rethinking execution in agent frameworks. Many existing LLM agent frameworks adopt a file-based execution pattern: the agent first writes the complete generated code into a file, then invokes an interpreter to execute the file as a whole. This design is inherently unfriendly to parallel execution, as the code must be fully materialized before execution can begin. Our work demonstrates that significant latency savings are available by simply restructuring this interface to accept code incrementally. We encourage future agent frameworks to adopt parallel-aware execution interfaces that allow the executor to consume code incrementally as it is generated, rather than waiting for full completion. This is a lightweight modification that integrates naturally with existing generation, repair, and planning strategies.

Notably, this applies equally to multi-file projects. In typical agent workflows, dependent modules are generated and written to disk before the entry-point script is produced. Parallel execution then applies to the entry-point script in the same way as to any single-file program: cross-file dependencies are resolved at runtime through `import` statements, each of which is an ordinary executable chunk that loads an already-existing module. Even when a script dynamically generates auxiliary files and imports them, the sequential chunk execution in EAGER naturally respects the write-then-import ordering, as the file-writing statement is dispatched and executed before the subsequent `import`.

9 CONCLUSION AND FUTURE WORK

In this paper, we introduced parallel execution, a paradigm that dispatches LLM-generated code statements to the interpreter as they are produced, overlapping generation and execution to reduce end-to-end latency. We formalized this paradigm theoretically and realized it in EAGER, a framework featuring AST-based chunking, dynamic batching, and early error interruption. Experiments across four benchmarks, seven LLMs, and three execution environments showed that EAGER consistently hides the majority of execution time behind generation, achieving latency reductions of up to 35%, while the early error interruption mechanism improved error resolution rates by up to 44 percentage points on data-centric tasks.

Several directions remain for future work. First, while our current implementation targets Python, extending the chunker and executor to other interpreted languages such as JavaScript and R would broaden the applicability of parallel execution. Second, current LLMs generate code without awareness that it will be executed incrementally. Training or prompting models to produce more "streamable" code could further improve overlap efficiency. Third, integrating parallel execution into multi-turn agent loops, where the LLM iterates between generation, execution, and planning over multiple rounds, presents an opportunity to compound the latency savings across an entire agent trajectory.

DATA AVAILABILITY STATEMENT

We open-source our artifacts at https://doi.org/10.6084/m9.figshare.31869469.

REFERENCES

[1] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. *Trans. Mach. Learn. Res.* 2023 (nov 2022).

[2] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *ArXiv* abs/2304.05128 (apr 2023).

[3] OpenInterpreter Contributors. 2026. openinterpreter/open-interpreter: A natural language interface for computers. GitHub repository. https://github.com/openinterpreter/open-interpreter Accessed: 2026-03-26.

[4] Google DeepMind. 2025. Gemini 3.1 Flash Lite: Our Most Cost-Effective AI Model Yet. https://deepmind.google/models/gemini/flash-lite/.

[5] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948* (2025).

[6] DeepSeek-AI. 2025. DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models. *arXiv preprint arXiv:2512.02556* (2025).

[7] Yangruibo Ding, Marcus J. Min, Gail E. Kaiser, and Baishakhi Ray. 2024. CYCLE: Learning to Self-Refine the Code Generation. *Proceedings of the ACM on Programming Languages* 8 (mar 2024), 392 – 418.

[8] Inc. Docker. 2024. Docker: Accelerated Container Application Development. https://www.docker.com/.

[9] Shihan Dou, Jiazheng Zhang, Jianxiang Zang, Yunbo Tao, Haoxiang Jia, Shichun Liu, Yuming Yang, Shenxi Wu, Shaoqing Zhang, Muling Wu, Changze Lv, Limao Xiong, Wenyu Zhan, Lin Zhang, Rongxiang Weng, Jingang Wang, Xunliang Cai, Yuemin Wu, Ming-bo Wen, Rui Zheng, Tao Ji, Yixin Cao, Tao Gui, Xipeng Qiu, Qi Zhang, and Xuanjing Huang. 2024. Multi-Programming Language Sandbox for LLMs. *ArXiv* abs/2410.23074 (oct 2024).

[10] Timur Galimzyanov, Sergey Titov, Yaroslav Golubev, and Egor Bogomolov. 2024. Drawing Pandas: A Benchmark for LLMs in Generating Plotting Code. *arXiv preprint arXiv:2412.02764* (2024).

[11] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. PAL: Program-aided Language Models. *ArXiv* abs/2211.10435 (nov 2022).

[12] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International conference on machine learning*. PMLR, 10764–10799.

[13] Sirui Hong, Yizhang Lin, Bangbang Liu, Binhao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Lingyao Zhang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Wenyi Wang, Xiangru Tang, Xiang Lu, Xinbing Liang, Yaying Fei, Yuheng Cheng, Zhibin Gou, Zongze Xu, Chenglin Wu, Li Zhang, Min Yang, and Xiawu Zheng. 2024. Data Interpreter: An LLM Agent For Data Science. *Annual Meeting of the Association for Computational Linguistics* (feb 2024), 19796–19821.

[14] Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, Yao Cheng, Jianbo Yuan, Jiwei Li, Kun Kuang, Yang Yang, Hongxia Yang, and Fei Wu. 2024. InfiAgent-DABench: Evaluating Agents on Data Analysis Tasks. In *Proceedings of the 41st International Conference on Machine Learning*.

[15] Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2024. DSBench: How Far Are Data Science Agents from Becoming Data Science Experts? *arXiv preprint arXiv:2409.07703* (2024).

[16] Boaz Lavon, Shahar Katz, and Lior Wolf. 2025. Execution guided line-by-line code generation. *arXiv preprint arXiv:2506.10948* (2025).

[17] Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Fei-Fei Li, Fei Xia, and Brian Ichter. 2023. Chain of Code: Reasoning with a Language Model-Augmented Code Emulator. *ArXiv* abs/2312.04474 (dec 2023).

[18] Killian Luca and Contributors. 2024. Open Interpreter. https://github.com/OpenInterpreter/open-interpreter.

[19] Diganta Misra, Nizar Islah, Victor May, Brice Rauby, Zihan Wang, Justine Gehring, Antonio Orvieto, Muawiz Chaudhary, Eilif Benjamin Muller, Irina Rish, Samira Ebrahimi Kahou, and Massimo Caccia. 2025. GitChameleon 2.0: Evaluating AI Code Generation Against Python Library Version Incompatibilities. *arXiv preprint arXiv:2507.12367* (2025).

[20] Ansong Ni, Srini Iyer, Dragomir Radev, et al. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *ICML*.

[21] OpenAI. 2024. GPT-4o mini: Advancing Cost-Efficient Intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

[22] OpenAI. 2025. GPT-5.1-Codex-Mini Model. https://developers.openai.com/api/docs/models/gpt-5.1-codex-mini.

[23] OpenRouter. 2025. OpenRouter: A Unified API for LLMs. https://openrouter.ai/.

[24] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural Language to Code Translation with Execution. *ArXiv* abs/2204.11454 (apr 2022).

[25] Qwen Team. 2025. Qwen3-Coder: Agentic Coding with Flow. https://qwenlm.github.io/blog/qwen3-coder/.

[26] Xiaomi MiMo Team. 2026. MiMo-V2-Flash Technical Report. *arXiv preprint arXiv:2601.02780* (2026).

[27] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936.

[28] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. *arXiv preprint arXiv:2402.01030* (2024).

[29] Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. 2024. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456* (2024).

[30] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. InterCode: Standardizing and Benchmarking Interactive Coding with Execution Feedback. *ArXiv* abs/2306.14898 (jun 2023).

[31] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2024. InterCode: Standardizing and Benchmarking Interactive Coding with Execution Feedback. In *NeurIPS*.

[32] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, A. Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, H. Michalewski, Oleksandr Polozov, and Charles Sutton. 2022. Natural Language to Code Generation in Interactive Data Science Notebooks. *ArXiv* abs/2212.09248 (dec 2022).

[33] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. *Annual Meeting of the Association for Computational Linguistics* (feb 2024), 12834–12859.

[34] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug Like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step. *Annual Meeting of the Association for Computational Linguistics* (feb 2024), 851–870.