

ASYNCHRONOUS LLM FUNCTION CALLING

In Gim¹ Seung-seob Lee¹ Lin Zhong¹

ABSTRACT

Large language models (LLMs) use function calls to interface with external tools and data source. However, the current approach to LLM function calling is inherently synchronous, where each call blocks LLM inference, limiting LLM operation and concurrent function execution. In this work, we propose AsyncLM, a system for asynchronous LLM function calling. AsyncLM improves LLM’s operational efficiency by enabling LLMs to generate and execute function calls concurrently. Instead of waiting for each call’s completion, AsyncLM introduces an interrupt mechanism to asynchronously notify the LLM in-flight when function calls return. We design an in-context protocol for function calls and interrupts, provide fine-tuning strategy to adapt LLMs to the interrupt semantics, and implement these mechanisms efficiently on LLM inference process. We demonstrate that AsyncLM can reduce end-to-end task completion latency from $1.6\times$ - $5.4\times$ compared to synchronous function calling on a set of benchmark tasks in the Berkeley function calling leaderboard (BFCL). Furthermore, we discuss how interrupt mechanisms can be extended to enable novel human-LLM or LLM-LLM interactions.

1 INTRODUCTION

Function-calling capabilities enable large language models (LLMs) to access external data sources and tools, such as weather forecasts and calculators. Both commercial and open-source LLMs have integrated this feature (Schick et al., 2024; Patil et al., 2024), unlocking new possibilities for diverse applications, from autonomous AI agents operating in dynamic environments (Wang et al., 2024) to neurosymbolic systems combining symbolic reasoning with LLMs to solve complex problems (Trinh et al., 2024).

LLM function calls are synchronous, with the LLM and the function call executor taking turns generating and executing calls. Although simple to implement, this approach is neither resource-efficient nor responsive. Each function call blocks LLM inference—one of the most resource-intensive processes—until the function returns. From the executor’s perspective, this limits concurrency since all function calls must finish in the order they are initiated by the LLM. These inefficiencies worsen as the number of functions increases with the complexity of the task (Zaharia et al., 2024).

Several studies have tried to address these challenges, including using compilers to parallelize function calls (Kim et al., 2023), fusing sequential calls to reduce overhead (Singh et al., 2024a), designing concise call syntax (Chen et al., 2023), and optimizing LLM serving systems for function

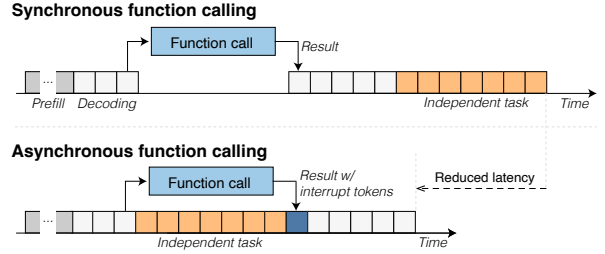


Figure 1. **Synchronous vs. asynchronous function calling.** Asynchronous function calling improves LLM’s operational efficiency, by enabling the LLM to continue generating tokens for independent tasks while the function call executes in the background.

calling (Abhyankar et al., 2024; Gao et al., 2024; Xu et al., 2024). While these methods help reduce function execution time or the number of function calls (§2), they are fundamentally limited by the synchronous nature of function call, e.g., the LLM waiting for the function call executor to finish.

We propose AsyncLM, a system that enables *asynchronous* interactions between LLMs and function call executors to overcome these limitations. In AsyncLM, the LLM and the executor operate independently without blocking each other, drawing inspiration from asynchronous programming paradigms where events, e.g., function call completions, occur independently of the main program flow, e.g., LLM token generation stream. Figure 1 illustrates the concept.

The key mechanism of AsyncLM is *interruptible* LLM decoding. In AsyncLM, function calls can be non-blocking; when a function call returns, the executor asynchronously notifies the LLM by injecting *interrupt tokens* into the

¹Department of Computer Science, Yale University, New Haven, United States. Correspondence to: In Gim <in.gim@yale.edu>.

LLM’s token generation stream. Although the concept is simple, AsyncLM must address two challenges in coordinating the LLM and the executor. (i) The interrupt token must be carefully timed to avoid interfering with other tasks or function calls the LLM is generating. For example, injecting an interrupt halfway through an argument for another function call could disrupt both the ongoing function call generation and the returned function call. (ii) The LLM must correctly handle the injected interrupt, even though such interrupts do not follow the conversational patterns on which the model was trained.

To address these challenges, we co-design the interface for asynchronous function calls and the LLM fine-tuning strategy. (i) We develop a token-efficient domain-specific language called CML to represent function calls and interrupts (§3). AsyncLM detects CML syntax during token generation to identify the start and end of function calls, deferring interrupts accordingly. CML also helps the LLM resume processing after an interrupt by embedding the necessary context within the interrupt syntax. (ii) We overcome the second challenge by fine-tuning the LLM to teach it how to generate asynchronous function calls, how to resume from handling an interrupt, and how to notify the LLM serving system when it must pause and wait for function call returns (§4). All three cases use CML as the interface. Additionally, when the LLM chooses to wait, AsyncLM determines the most efficient strategy to manage the blocking state, such as dropping, swapping, or retaining the KV cache.

The advantages of AsyncLM over synchronous function calling are twofold. First, AsyncLM reduces the execution time by overlapping the generation and execution of functions. This property ensures that AsyncLM is at least as fast as synchronous function calling with parallel execution, even in the worst case (§6.3). Second, AsyncLM enables automatic parallelism without requiring prior knowledge of the dependency graph for future function calls. For example, given the task, “Tell me if Leonhard Euler is one of my academic ancestors using the math genealogy API,” and an API definition, `find_advisors(str) → list[str]`, AsyncLM can recursively and in parallel invoke `find_advisors`, similar to performing a parallel depth-first search. In contrast, efficiently handling such tasks in a synchronous function calling scheme is challenging, as it requires determining the list of parallelizable function calls in advance (§2).

We implement AsyncLM on Llama 3 models (Dubey et al., 2024) and emulate fine-tuning on GPT-4o using in-context prompting—where the model is guided through CML examples and instructions in the input prompt (§5). We evaluate both models on a suite of function calling tasks from the Berkeley Function Calling Leaderboard (BFCL) (Yan et al., 2024) (§6). Benchmark comparisons show that AsyncLM accelerates end-to-end task completion by up to $1.6\times$ – $5.4\times$

over synchronous sequential function calling and achieves a speedup of up to $2.1\times$ over synchronous parallel function calling. We also assess the impact of AsyncLM on function calling accuracy, demonstrating that AsyncLM maintains the same level of accuracy as synchronous function calling in fine-tuned Llama models. Notably, GPT-4o can handle asynchronous function calling without explicit fine-tuning. Finally, we discuss the potential of the introduced interrupt mechanism for use in novel AI applications involving human-LLM or LLM-LLM interactions, such as interruptible LLM assistants.

2 BACKGROUND AND RELATED WORK

The concept of augmenting LLMs with code execution (Milon et al., 2023) has been extensively explored, notably in contexts like retrieval-augmented generation (Khattab et al., 2022; Yao et al., 2023), autonomous agents (Huang et al., 2024; Wang et al., 2024), and neurosymbolic problem solving (Pan et al., 2023; Trinh et al., 2024).

Learning to generate function calls. The most common method for LLM interaction with external systems is tool or function calling (Schick et al., 2024; Shen et al., 2024), where the model autonomously generates calls to external executors (e.g., API servers, code interpreters). This ability is refined either through fine-tuning on diverse task datasets (Patil et al., 2024) or using in-context instructions (Liang et al., 2024). Function descriptions, including arguments, are provided in a structured prompt format, often in JSON. Our approach builds on this paradigm by enabling asynchronous function calling, requiring LLMs to (1) consider execution time in call generation and (2) use interrupt semantics to decide subsequent calls.

Efficient LLM function calling. A major challenge in function calling is optimizing efficiency to enhance resource utilization and reduce latency. Various studies have explored different optimization strategies to improve the function calling process. For instance, parallel function calling approaches (Kim et al., 2023; OpenAI, 2023) instruct the LLM to bundle calls that can be executed simultaneously, enabling external compilers to optimize these batches for parallel execution. Sequential function call optimizations include methods like function call fusion (Singh et al., 2024a), caching (Singh et al., 2024b), compact syntax for call representation (Chen et al., 2023), and partial execution of function calls (Xu et al., 2024), which allow overlapping of generation and execution of a single code block.

Limitations of a synchronous interaction. Currently, LLMs perform synchronous function calls. This interleaved nature of generation and execution introduces extra overheads in LLM inference, due to the stateless nature of LLMs,

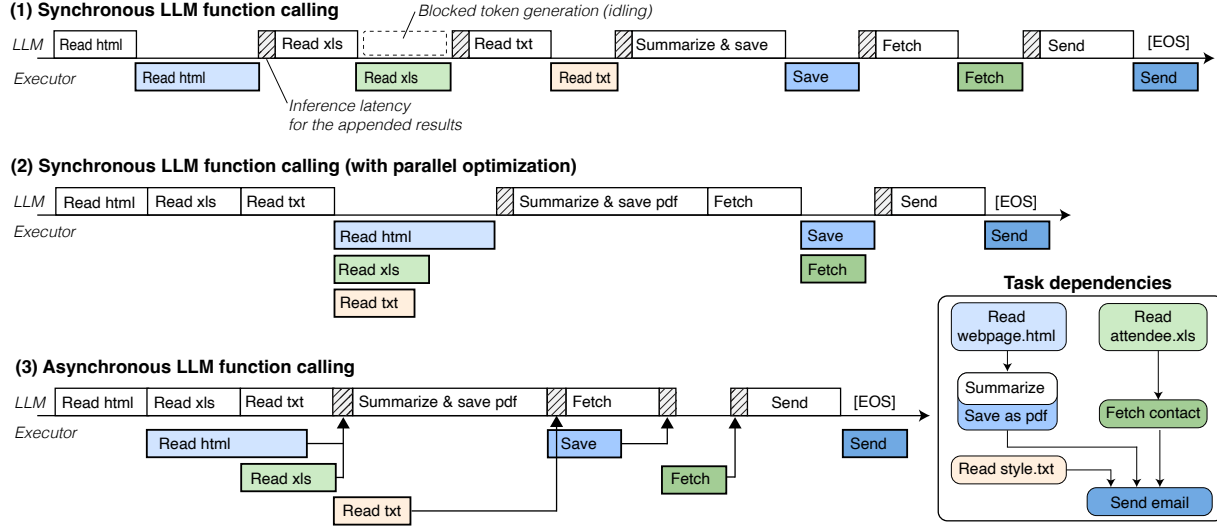


Figure 2. **Comparison of LLM-executor interactions.** Parallel function calling reduces end-to-end execution time by generating function calls upfront and executing them in parallel, while still requiring the LLM to wait for all calls to complete. Asynchronous function calling overcomes this limitation by allowing the LLM to generate tokens while function calls execute in the background. **The task:** “Summarize webpage.html into event.pdf, email it (styled per style.txt) to people in attendee.xls, and CC the dept chair from the directory if not listed.”

which require a new session per function call. This has motivated research on efficient management of KV cache usage during waiting periods (Abhyankar et al., 2024; Yu & Li, 2023; Gao et al., 2024). More importantly, synchronous function calling fundamentally limits resource utilization, since either only token generation or function execution can happen at the same time. ReWOO (Xu et al., 2023) addresses this limitation by prompting LLMs to decouple reasoning from observation to consolidate the execution of multiple function calls into one. Our work is conceptually aligned with ReWOO but remains agnostic to the LLM’s reasoning strategy, making it more broadly applicable.

Our approach. AsyncLM addresses the limitations of synchronous function calling by enabling asynchronous execution. Figure 2 illustrates the benefits of asynchronous function calling over synchronous schemes. While function parallelization optimizations (Kim et al., 2023; OpenAI, 2023) reduce end-to-end execution time, the LLM must still wait for function calls to complete, leaving resources idle. Asynchronous function calling eliminates this bottleneck by allowing the LLM to continue generating tokens for independent tasks alongside ongoing function calls. To support asynchronous calls, AsyncLM introduces mechanisms to asynchronously notify LLMs in-flight when function call returns, and fine-tunes the LLM to understand interrupts and leverage concurrent execution.

AsyncLM is built around two key innovations. First, it uses a domain-specific language (CML) to describe asynchronous function calls and interrupts, embedding the necessary con-

text for seamless integration (§3). Second, its fine-tuning strategy trains the LLM to respond to asynchronous events, generate concurrent function calls, and notify the serving system when it needs to pause and wait for previous calls to complete (§4) to ensure correct function call dependencies. We implement these components in AsyncLM’s LLM serving system, which monitors token generation, schedules function calls on the executor, and manages the token stream by injecting interrupts or pausing generation as needed (§5).

3 REPRESENTING ASYNCHRONOUS INTERACTION WITH CML

We define a simple domain-specific language, called *Context Markup Language* (CML), to represent asynchronous function calls and interrupts. CML acts as the interface between the LLM and the executor, with its syntax ensuring that each component provides the necessary context when interacting through this interface. For example, the LLM can generate a function call in CML to notify the executor, while the executor signals completion by inserting interrupt tokens in CML, as demonstrated in Figure 3.

CML uses a minimal set of specialized tokens: [CALL], [INTR], [TRAP], [END], and [HEAD]. The [CALL], [INTR], and [TRAP] tokens initiate a *control block*, which represents a function call (§3.1), an interrupt (§3.2), or a trap (a special interrupt), respectively. The [END] token marks the end of the control block, and [HEAD] separates optionally provided metadata, such as a function call identifier, and the body of function call. The rest of this

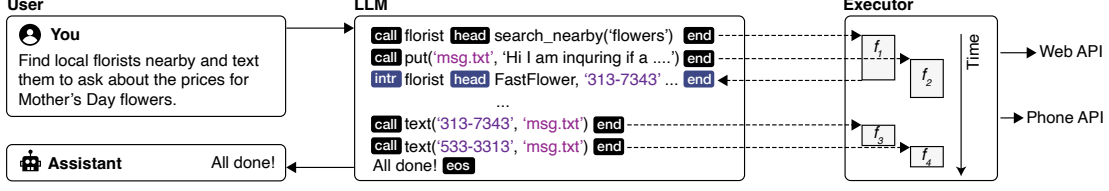


Figure 3. **Asynchronous function execution workflow.** The LLM generates function calls in CML format, which are monitored real-time and sent to the code executor in the background. When a function call completes, the executor asynchronously notifies the LLM by inserting an interrupt along with the execution result.

section defines the semantics of CML.

3.1 Initiating Function Calls

In AsyncLM, the LLM initiates function calls using the following format: `[CALL] function call [END]`. The function call can be written in any valid executable language, such as Python code or a JSON abstract syntax tree (AST), supported by the executor. However, only one language should be used consistently within each function call block. Independent function calls should be placed in separate blocks to allow for parallel execution.

The generated function calls initiate execution without blocking the token stream, enabling implicit parallelism. For example, as shown in Figure 3, if the LLM generates two function calls (`search_nearby` and `put`), the executor processes each in a separate worker, allowing pipelined generation and execution of function calls. This overlap reduces latency, as the execution of `search_nearby` can occur simultaneously with the generation of writing text messages. Function calls are generated according to their dependency order, ensuring that dependent calls are executed in the correct sequence. For instance, the LLM can generate a function call, `text`, only after searching for florists and composing the message are done.

Assigning identifiers for interrupts. If the LLM will need to refer to the function result for subsequent calls or reasoning, it can include an identifier in the function header (e.g., `[CALL] job1 [HEAD] function call [END]`), where `job1` acts as a unique identifier. To avoid conflicts, the LLM generates identifiers following Python variable naming conventions. Identifiers must remain unique throughout the session. Although our prototype does not implement this, a uniqueness check could be included as part of syntax validation (§5.1).

3.2 Triggerring Interrupts

When the executor completes a function call with a registered identifier, it asynchronously notifies the LLM by inserting an interrupt block at the end of the token stream (i.e., the LLM context). Token generation resumes after

the interrupt block is added. The format of an interrupt block is: `[INTR] id [HEAD] value [END]`, where `id` matches the identifier from the corresponding `[CALL]` block (e.g., `job1` from the earlier example). The `value` contains the executor’s result, such as the function’s output or an error message.

Critical sections. Since interrupts modify the LLM’s token generation flow, it is essential to ensure that the context following an interrupt conforms to CML syntax. For example, if an interrupt block is inserted during the generation of a function call, the resulting tokens could become logically incorrect, violating CML syntax. To prevent this, AsyncLM temporarily disables interrupts during specific token generation periods, inspired by how operating systems defer lower-priority interrupts while handling others. A flag called *critical section*, implemented within the interrupt manager (§5.3), determines whether the executor can insert interrupts. This flag is set to `false` while a function call block is being generated and resets to `true` once the LLM exits the block. Any interrupts that occur during a critical section are queued and inserted when the flag is set to `true`.

Waiting for interrupts. In synchronous function calling, the LLM stops generating tokens after making a function call, waiting for the call to complete. In contrast, asynchronous function calling lets the LLM continue generating tokens for other tasks without waiting. However, the LLM must sometimes pause to wait for function results when inter-task dependencies exist. The standard `[EOS]` token is not sufficient for this purpose because it cannot indicate whether token generation is complete and resources can be released, or if the LLM is temporarily pausing until the function result arrives.

To resolve this ambiguity, AsyncLM introduces self-initiated interrupts, called *traps*, which notify the LLM serving system (§5.4) when to pause token generation. Each trap follows the simple structure `[TRAP] [END]`. Traps create explicit boundaries for asynchronous function calls, helping generate training samples to fine-tune LLMs for asynchronous function handling (§4). They also enable optimization opportunities for the serving system (§5).

4 LEARNING TO HANDLE INTERRUPTS

We propose a fine-tuning scheme to train LLMs to (i) generate asynchronous function calls and traps using CML and (ii) handle interrupts that deliver the results of previous function calls. The core idea is to construct a dataset with simulated function calls and interrupts that model ideal interactions between the LLM and the executor. We extract task descriptions and function definitions from publicly available function-calling datasets, add estimated completion times, and present them to the LLM in JSON format as part of the prompt (OpenAI, 2023; Patil et al., 2024; Liang et al., 2024). For simplicity, we assume that each task can be completed with a finite number of function calls.

4.1 Training objectives

The primary objective of fine-tuning in AsyncLM is to train the LLM to minimize the total task completion time (i.e., the makespan) by effectively utilizing asynchronous function calling, while respecting task dependencies and considering estimated function execution times. Specifically, the LLM must make decisions in the following areas.

Deciding the next function call. The LLM selects the next function to call based on the context.¹ It identifies which functions are available to be called immediately—those without any pending dependencies—and chooses the most suitable one. AsyncLM uses a Longest-Processing-Time-first (LPT) strategy, where the LLM prioritizes calling the function with the longest estimated execution time among those that are ready. This approach reduces idle time by maximizing the overlap between function call generation and execution, making it particularly effective when multiple independent functions can be called. LPT is optimal for scenarios with parallel function calling (§6).

Handling interrupts. When an interrupt appears in the context, the LLM must decide whether to continue generating tokens for the current task or shift to addressing the interrupted task. For example, the LLM might ignore the interrupt if the completed task has lower priority than the current task or if it was the final task in a sequence. AsyncLM handles interrupts using the same priority principles as for deciding the next function call. An interrupt may introduce new functions that are ready to be called; if a newly available function has the longest estimated processing time among the options, the LLM is trained to call it next. For instance, in the task scheduling scenario shown in Figure 2, when “Read html” and “Read xls” complete, the LLM chooses to do “Summarize & save pdf” first over “Fetch contact,” prioritizing the function with the longer

¹We use the *context* to refer to the sequence of tokens currently visible to the LLM.

estimated processing time.

Generating Traps. When no functions are available to call due to dependencies on future interrupts, and there are no other tokens to generate, the LLM must generate a trap to pause token generation.

4.2 Generating Training Samples

To create training samples, we simulate ideal interactions between the LLM and executor using existing function-calling traces from LLM benchmarks with multi-turn scenarios (Lu et al., 2024; Yan et al., 2024; Yao et al., 2024).

DAG generation. We developed a Python program to extract directed acyclic graphs (DAGs) of function calls from each sample in the dataset. In sequential scenarios, the DAG is linear, with nodes representing function calls and edges indicating dependencies. In parallel scenarios, the DAG branches to represent independent function calls. Multi-turn scenarios consist of multiple DAGs.

Simulating interrupts. We simulate interactions between the LLM and the executor on these DAGs using the LPT strategy. In each simulation run, we assign random estimated execution times to function calls, ranging from 1 ms to 1 s, to prevent the model from overfitting to specific function names. These estimates are provided to the LLM as part of the input prompt. To determine interrupt timing, we randomly set a time-per-output-token (TPOT) between 5 ms and 30 ms per simulation and track elapsed time by counting generated tokens. When a function call completes, we insert an interrupt block containing the execution result. If all functions in the DAG are waiting on dependencies, we insert a trap block and inject the next interrupt.

5 IMPLEMENTATION

We implement AsyncLM by intercepting the LLM’s autoregressive token generation, as illustrated in Figure 4. The system is built using the Python transformers library (Wolf et al., 2020). AsyncLM consists of four main components: a token monitor, an executor, an interrupt manager, and a trap handler. In this section, we describe the implementation of each component and the modifications made to the LLM inference. While some aspects of these components can be implemented on cloud-based LLM APIs without modifying the serving system (§5.5), such implementations are practical only in certain scenarios (§6.2).

5.1 Token Monitor

The token monitor audits and regulates the LLM’s token generation process. Its primary functions are (i) to notify the executor or trap handler immediately when a function

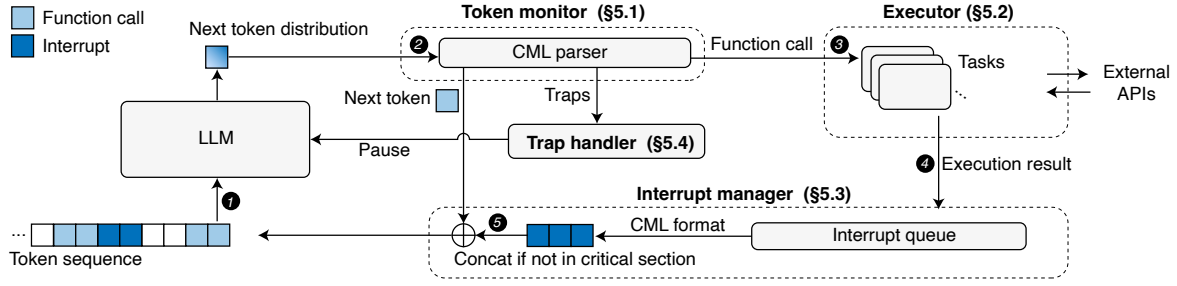


Figure 4. **Overview of AsyncLM’s LLM inference process.** LLMs generate tokens autoregressively. AsyncLM implements asynchronous function calling by augmenting this process with additional steps to monitor the token generation process for CML recognition and manage asynchronous function calls, interrupts, and traps based on recognized CML syntax. The numbered steps, ❶ to ❺, show the logical flow from generating a function call to executing it and returning the result.

call or trap is detected in the token stream, and (ii) to enforce CML syntax compliance. Implemented as a hook into the LLM’s next-token sampling process, the token monitor constrains the next-token distribution to ensure generated tokens follow valid CML syntax (Geng et al., 2023). It uses a finite-state machine (FSM) to enforce CML rules and blocks the LLM from producing `[INTR]` tokens, thereby preventing “hallucinated” interrupts, as interrupts must be injected by the system rather than generated by the LLM. After sampling each token, the token monitor checks for a function call or trap and, if detected, immediately notifies the executor (§5.2) or trap handler (§5.4). Additionally, after determining whether the LLM is in an interruptible state, the token monitor sets a critical section flag accordingly. It sends both the generated token and the critical section flag to the interrupt manager (§5.3).

5.2 Executor

The executor manages the execution of function calls generated by the LLM. It receives each function call from the token monitor (formatted in Python syntax in our prototype) along with an optional identifier used to manage future interrupts. Each function call runs on a dedicated worker, allowing multiple calls to execute concurrently if resources permit. These workers interact directly with external systems, such as API servers or code interpreters. Once a function completes, the executor sends the result and identifier to the interrupt manager (§5.3).

5.3 Interrupt Manager

The interrupt manager has three main functions: (i) managing an interrupt queue, (ii) tracking when the token generation process is interruptible based on a critical section flag, and (iii) inserting CML-formatted interrupt blocks into the token stream. When the executor completes a function call, it adds the result to the interrupt queue. During each decoding step, the interrupt manager receives newly generated tokens and the critical section flag from the token monitor

(§5.1). If the process is interruptible, i.e., the critical section flag is not set, it formats all queued interrupts in CML, tokenizes them, and appends them to the generated tokens. These tokens will then be processed by the LLM in the next decoding step.

5.4 Trap Handler

The trap handler’s goal is to minimize idle KV cache usage in GPU memory without adding latency to task completion. When the token monitor detects a trap—indicating that token generation needs to be paused—it notifies the trap handler with the current context. Based on this notification, the trap handler determines the best strategy for managing the KV cache during the pause by considering the number of tokens in the current context and the estimated time until the next interrupt completes. Prior work has shown that recomputing the KV cache scales quadratically with the number of tokens, while swapping it to host memory scales linearly (Abhyankar et al., 2024; Gim et al., 2024). Given the scaling characteristics of these costs, the trap handler keeps the KV cache in GPU memory if both recompute and swap times exceed the estimated wait time. Otherwise, it opts to recompute if recompute latency is lower, or to swap if swap latency is lower. We provide a real-world example of an ideal trap handling strategy in §6.2.

5.5 Implementation on Chat Completion APIs

To demonstrate AsyncLM’s adaptability, we also implement it using OpenAI’s (streaming) chat completion API without modifying the serving system. The executor implementation can be reused as is. For the token monitor, we implement only the CML parser without constrained decoding, as the API already streams sampled tokens. To emulate the interrupt insertion mechanism in the interrupt manager, we start a new request to the API server whenever an interrupt is triggered, discarding the previous session. The trap handler is unnecessary for cloud APIs, which are stateless and recompute the entire KV cache from scratch

for each new session when an interrupt is inserted.² We note that this implementation is practical only when the time-to-first-token (TTFT) latency is low, which is generally not the case for most cloud services (details in §6.2).

6 EVALUATION

Our evaluation answers two questions: (i) *latency* (§6.1–§6.4), i.e., how much does asynchronous function calling reduce task completion latency compared to synchronous methods, and (ii) *correctness* (§6.5), i.e., how does the asynchronous mechanism affect the correctness of generated function calls.

Workloads. To evaluate task completion latency and function calling accuracy, we use the Berkeley function calling leaderboard (BFCL) (Yan et al., 2024), which captures real-world function calls across eight domains, such as vehicle control, travel booking, file system operations, and the Twitter API. BFCL includes 84 unique functions. We utilize three datasets from BFCL to cover different function calling scenarios: `v1-parallel`, `v2-parallel-live`, and `v3-base-multi-turn`. Specifically, `v1-parallel` and `v2-parallel-live` provide 400 parallel function calling scenarios, while `v3-base-multi-turn` offers 200 multi-step function calling scenarios. To simulate more complex multi-step parallel function calling, we created a new dataset `v3-multi-step-parallel`. This dataset consists of 200 scenarios formed by randomly combining three distinct multi-step samples from the first round in `v3-base-multi-turn`. From the total of 800 samples, we use 200 for fine-tuning and the remaining 600 for evaluation. The measured execution time of each function ranges from 30 ms to 500 ms, with an average of 110 ms.

AsyncLM setup. We consider two LLM deployment settings: local and cloud. In the local deployment, LLM inference and function execution run on the same machine equipped with an NVIDIA RTX 4090 GPU. This deployment uses Llama-3.2 models (Dubey et al., 2024) with 3B and 1B parameters. We fine-tune the Llama models following default configurations in LlamaFactory (Zheng et al., 2024). Local models are served with Text-Generation-Interface (Hugging-Face, 2023). In the cloud deployment, only function execution is local. This deployment uses OpenAI’s GPT-4o and GPT-4o-mini. We adapt them using few-shot prompting; we select one example from each dataset and provide detailed instructions on their interpretation.

Baselines. To compare against synchronous LLM function calling, we employ two synchronous baselines:

- *Sync*: Sequential function calling where LLM code gener-

ation and execution are interleaved.

- *Sync-Parallel*: Parallel function calling (OpenAI, 2023; Kim et al., 2023), where the LLM bundles independent function calls, and the executor runs them in parallel.

We also compare two implementations of AsyncLM:

- *Async-Naive*: Based on the chat completion API without modifying the underlying LLM serving system.
- *Async*: AsyncLM with the co-designed inference process.

For *Async* latency measurements on cloud (GPT-4o), we report emulated results based on average token generation latency statistics (5 ms per output token) from the OpenAI API. This offers a rough estimation of the latency benefits of implementing AsyncLM on cloud-based LLMs.

Note that the officially reported accuracy of LLMs in generating function calls in BFCL is 62% for GPT-4o and 43% for Llama-3.2-3B (Yan et al., 2024). This can lead to biases in the latency evaluation, since incorrect function calls can miss some essential steps or have redundant steps. To mitigate this interference in latency evaluations, we include a “cheat sheet” with ground truth answers in the prompt. This ensures consistency in the generated function calls across measurements.

6.1 Parallel Function Calling

First we employ a simplistic setup where no function calls depend on each other and report results in Figure 5. Parallel function calling is common (OpenAI, 2023). For example, when a user asks “Which city has a higher chance of rain tomorrow, Seattle or Vancouver?”, the LLM can generate two function calls: one for weather data in Seattle and another for Vancouver, which can be executed in parallel. We use `v1-parallel` and `v2-parallel-live` from BFCL for this evaluation.

Results. *Async* improves latency by up to $2.1\times$ over *Sync*. We measure the end-to-end task completion latency as the time between the first and last token generation. Our results show that *Async* completes tasks faster than *Sync* by $1.6\times$ for the local deployment and $2.1\times$ for cloud. In comparison, *Sync-Parallel* is $1.3\times$ faster than *Sync* for local and $1.7\times$ faster for cloud. Although *Async-Naive* is slower than *Async*, it is still $1.2\times$ faster than *Sync-Parallel* for local.

6.2 Multi-Step Parallel Function Calling

For evaluations on more complex setup under function call dependencies, we evaluate AsyncLM using the `v3-multi-step-parallel` dataset, which consists of three independent tasks requiring 1–5 sequential function calls each. The results are presented in Figure 6. In these function calling scenarios, AsyncLM must respect task order dependencies and manage interrupt identifiers for each function call. For example, the task “make pasta” can involve two independent

²While some LLM services support caching prompts for reuse, these caches are typically designed for frequently accessed information, such as documents, rather than for runtime use.

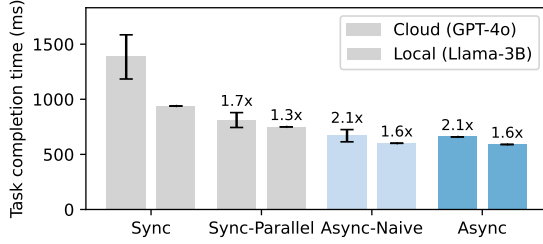


Figure 5. **Parallel function calling latencies.** End-to-end task completion latency across the combined BFCL parallel dataset, with error bars representing the 10th and 90th percentile ranges. Bar annotations indicate speedups relative to *Sync*.

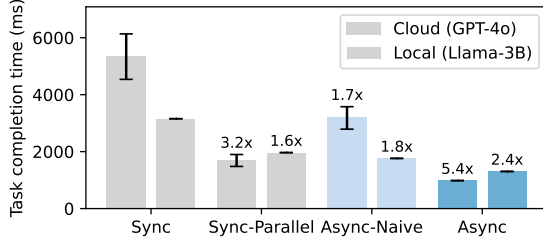


Figure 6. **Multi-step parallel function calling latencies.** End-to-end task completion latency for simultaneously handling three parallel tasks in the BFCL multi-step dataset.

sequences: (i) `boil_water()` followed by `put_pasta_noodles`, and (ii) `chop_vegetables()` followed by `stir_fry()`, and culminating in `mix_everything()`.

Results. AsyncLM reduces latency by up to $5.4\times$ over *Sync* by parallelizing function calls in independent sequences; while *Sync-Parallel* reduces latency by $3.2\times$ compared to *Sync*. As illustrated in Figure 2, unlike in *Sync-Parallel*, where the LLM needs to wait for all bundled functions to complete, *Async* uses the LPT strategy to schedule individual function calls. This enables the LLM to optimize token generation cycles more effectively, similar to out-of-order execution in CPU instruction scheduling.

6.3 Latency Analysis

To understand how asynchronous function calling reduces latency, we conduct theoretical analyses under parallel function calling (§6.1). All proofs are available in Appendix A.

Overlapping generation and execution. Asynchronous function calling is at least as fast as synchronous function calling. The total latency for *Sync* in simple parallel function calling can be modeled as:

$$L_{\text{Sync}}(F) = \sum_{f \in F} G(f) + \sum_{f \in F} E(f),$$

given F as a set of functions that do not depend on each other for execution, where $G(f)$ is the token generation latency for $f \in F$, and $E(f)$ is the execution time. For

Sync-Parallel, the total latency is:

$$L_{\text{Sync-Parallel}}(F) = \sum_{f \in F} G(f) + \max_{f \in F} E(f),$$

assuming negligible overhead for parallelizing them. For *Async*, using the LPT heuristic, the total latency is:

$$L_{\text{Async}}(F) = \max_{f \in F} \left(E(f) + \sum_{g \in \text{pred}(f, F)} G(g) \right),$$

where $\text{pred}(f, F) = \{g \in F \mid E(f) \leq E(g)\}$. Intuitively, this formation indicates that each function call f initiates after generating all function calls that has the same or higher priorities, i.e., sum of generation times in $\text{pred}(f, F)$. We prove that *Async* is at least as fast as *Sync-Parallel* in the worst case, assuming the number of generated tokens is the same and there are no additional overheads in *Sync-Parallel* or *Async* beyond $G(f)$ s and $E(f)$ s.

Theorem 6.1 *For any set of independent functions F , $L_{\text{Async}}(F) \leq L_{\text{Sync-Parallel}}(F) < L_{\text{Sync}}(F)$.*

Characterizing speedups with average function execution times. Assuming that the $E(\cdot)$ follows a normal distribution, we estimate the expected speedup of *Async* over *Sync* using \bar{E} and \bar{G} , where \bar{E} represents the average execution time, and \bar{G} denotes the average generation time of functions in F .

Theorem 6.2 *The ratio of speedup, $\frac{L_{\text{Sync}}}{L_{\text{Async}}}$ is approximately $1 + \frac{\bar{E}}{\bar{G}}$ with an error of $\mathcal{O}((\frac{\bar{E}}{\bar{G}})^2)$ when $|F|$ is large.*

This result indicates that tasks involving long average execution times (e.g., expensive I/Os) leads to a better speedup, whereas tasks with short execution times (e.g., simple arithmetic operations) may not benefit as much from AsyncLM.

Rationale on using LPT heuristic. LPT is optimal for parallel function calling. We prove that the LPT heuristic minimizes total latency in asynchronous parallel function calling. Intuitively, starting the execution of longer functions earlier maximizes overlap with token generation, reducing the overall makespan. To put this argument formally,

Theorem 6.3 *Continuing from Theorem 6.1, Any deviation from the LPT order cannot result in a lower total latency.*

These analyses roughly demonstrate the advantage of asynchronous function calling, by overlapping generation and execution, as well as the rationale of LPT heuristic, especially when functions are independent of each other.

6.4 System Overheads

Inference overhead of AsyncLM. AsyncLM introduces two potential sources of LLM inference overhead compared to *Sync*: (i) syntactic overhead from using CML and function call identifiers, and (ii) overhead from interrupts, caused

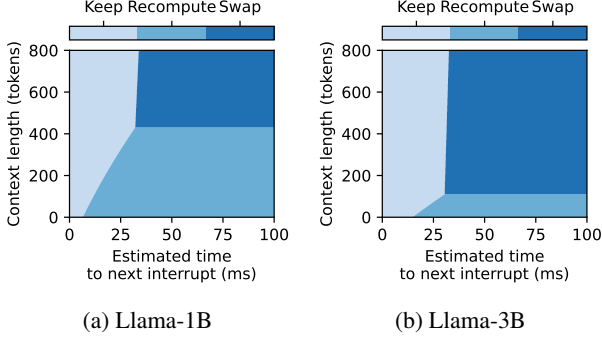


Figure 7. The diagrams of the trap handling strategy that minimizes idle KV cache in memory without sacrificing latency. The optimal strategy depends on the estimated time to the next interrupt and the number of tokens in the context.

by injecting function execution results and their identifiers into the LLM context. The syntactic overhead is small since *Sync* and *Sync-Parallel* also require specific formats for function calls, e.g., wrapping function calls with markdown code syntax. AsyncLM adds two or three control tokens per function call, which is negligible relative to the total tokens generated. Similarly, the overhead from interrupts is mostly coming from in-context interrupt identifiers compared to *Sync*, since both methods receive function return values within the LLM context. *Async* incurs an average of 20 additional tokens compared to *Sync* on the `v3-multi-step-parallel` dataset, adding about 90 ms of latency and 27.5 MB of additional KV cache in GPU memory on 3B model—less than 5% of total memory usage.

Performance without co-optimized the serving system. Implementing AsyncLM on the OpenAI API (*Async-Naive*) is impractical for GPT-4o due to high time-to-first-token (TTFT) latency. On GPT-4o, *Async-Naive* suffers from $1.5\times$ higher latency than *Sync-Parallel* (Figure 6). The increased latency stems from the overhead of initiating a new API call for each interrupt, resulting in an average TTFT of 310 ms. Interestingly, *Async-Naive* on the local Llama-3B does not experience such high TTFT, with an average TTFT of only 59 ms, making it $1.1\times$ faster than *Sync-Parallel*. Techniques like prompt caching or stateful inference may mitigate the TTFT overhead in *Async-Naive*, potentially improving performance.

LPT heuristic under task dependencies. Empirical experiments show that LPT is an effective heuristic. In our local *Async* evaluation in Figure 6, LPT was 8% faster (on average) than randomly choosing the next function. However, we note that LPT is not guaranteed to be optimal when functions have dependencies. Consider functions a , b , and c , where c depends on a , and their execution times satisfy $E(a) < E(b) < E(c)$. LPT would schedule them as $b - a - c$, since it does not consider future dependencies, while the optimal order is $a - c - b$. This challenge falls under

Table 1. Average function composition accuracy (AST matching) of *Sync* and *Async* in multi-step parallel function calling. (FT) denotes fine-tuned models; (ICL) denotes few-shot prompting.

MODEL	SYNC	SYNC (IN CML)	ASYNC
GPT-4o (ICL)	57.84%	57.80%	59.61%
4o-MINI (ICL)	55.08%	53.41%	41.44%
LLAMA-3B (ICL)	16.98%	17.33%	5.46%
LLAMA-1B (ICL)	0.49%	0.39%	1.45%
LLAMA-3B (FT)	57.33%	66.07%	65.97%
LLAMA-1B (FT)	6.33%	12.15%	12.06%

resource-constrained project scheduling, an NP-hard problem (Brucker et al., 1999; Hartmann & Briskorn, 2022). LPT is a greedy heuristic that considers only currently available information without predicting future dependencies.

Trap handling strategy. Trap handler can reduce idle KV cache memory usage by estimating the expected latency of swapping and recomputation based on the context length. Figure 7 analyzes the optimal trap handling strategy for AsyncLM on Llama-1B and Llama-3B models, using latency models of KV cache swap (linear) and recomputation (quadratic). For example, if the next interrupt is expected in 100 ms and the context has 300 tokens, the trap handler can drop the KV cache and recompute in 1B model, but should swap it out in 3B due to higher recomputation overhead.

6.5 Function Calling Accuracy

To assess how AsyncLM affects LLMs’ ability to generate accurate function calls, i.e., correct signature, arguments, and calling order, we examine function call traces of each baseline using multi-step parallel dataset. We ensure all necessary function definitions are provided in the prompt. Function calling accuracy is measured by exact AST matching for each function call against the ground truth call, and their execution order.

We compare two LLM adaptation strategies: fine-tuning and few-shot prompting. Llama models are adapted using both methods, while GPT-4o is adapted with few-shot prompting only. To understand the effect of CML syntax, we also test a scenario where we format function calls in *Sync* in CML. Table 1 presents the results for GPT-4o and Llama models under these adaptation strategies.

Impact of CML on accuracy. Formatting *Sync* responses using CML syntax has minimal effect on accuracy. In GPT-4o, accuracy remains virtually unchanged, while Llama-3B shows a slight improvement. Fine-tuned models exhibit a modest increase in accuracy when using CML, likely because the fine-tuning samples were formatted with CML.

Impact of Asynchronous Function Calling on Accuracy. Both GPT-4o (with few-shot prompting) and fine-tuned

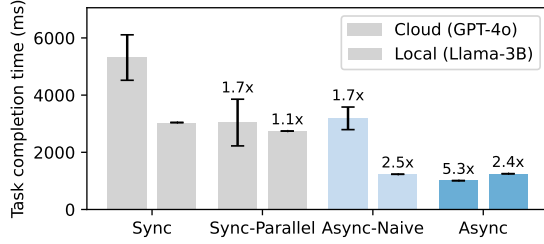


Figure 8. **Evaluation on user-triggered interrupts.** This repeats the experiment in Figure 6, while each task is represented by an interrupt instead of a prompt, arriving at 0, 200, and 400 ms.

Llama models maintain similar accuracy across *Sync* and *Async*, suggesting that asynchronous function calling does not degrade the models’ function calling accuracy. However, Llama models adapted with few-shot prompting experience a significant drop in accuracy when switching from *Sync* to *Async*. We attribute this to the limited in-context learning capability of small models to adapt to CML syntax.

Fine-tuning vs. Few-shot Prompting for LLM Adaptation. While GPT-4o adapts well to asynchronous function calling with few-shot prompting, smaller models like GPT-4o-mini experience a slight reduction in accuracy in *Async*. Few-shot prompted Llama models show low overall accuracy, whereas fine-tuning significantly improves their performance. Notably, fine-tuned Llama models achieve higher accuracy in *Async* compared to GPT-4o in *Sync*, highlighting the importance of fine-tuning for smaller LLMs.

6.6 Discussion – Human-triggered Interrupts

The interrupt mechanism proposed in AsyncLM not only allows asynchronous function calling, but also enables new types of human-LLM and LLM-LLM interactions. We first discuss flexibility of this mechanism, then illustrate its potential with real-world examples.

Generality and flexibility of interrupt. AsyncLM extends interrupt functionality beyond executor notifications, allowing interrupts to signal external events, such as user inputs or system triggers. This capability enables real-time interactions, where users can interrupt an ongoing LLM inference to add or adjust tasks without waiting for the current task to finish. Although not detailed in this paper, we incorporated user-triggered interrupts into our fine-tuning dataset by randomly injecting new tasks from multi-turn conversation datasets. While this dataset may not perfectly represent real human-LLM interactions, we explored its effectiveness in the following scenarios.

Interruptible AI assistants. LLM chatbots currently operate synchronously, requiring users to wait for complete responses. This approach increases perceived latency and limits real-time task handling. With AsyncLM, users can issue new requests immediately, even during ongoing LLM

responses. For instance, users often make follow-up requests without waiting for the initial response. A user might first ask to “find a hotel in Seattle” and then quickly add “near Space Needle.” Similarly, a user can correct a previous request, such as saying “Actually, make it Thursday” after initially scheduling a meeting for Wednesday.

To examine this scenario, we repeated the multi-step parallel function calling experiment, presenting tasks as a series of user-triggered interrupts rather than prompts. We inserted these interrupts every 200 ms and measured the end-to-end task completion latency of *Async*, as shown in Figure 8. For comparison, we ran *Sync* and *Sync-Parallel* with the same tasks, each processing them sequentially. *Async* reduced latency by $2.4\times$ compared to *Sync*, whereas *Sync-Parallel* achieved only a $1.1\times$ reduction. The limited improvement of *Sync-Parallel* is because it requires generating the entire function calls at once, which is unsuitable for this scenario.

Multi-communicating LLM agents. AsyncLM also enables multiple autonomous LLM agents to communicate simultaneously. Typically, agent-to-agent communication uses synchronous message exchanges (Wu et al., 2023; Chan et al., 2024), where agents take turns. This limits their ability to multitask and confines communication to a round-robin style. With AsyncLM, one-to-many or many-to-many agent communication can be implemented using interrupt mechanisms. Each agent can send messages to others as interrupts, enabling more dynamic interactions, which could enable more realistic LLM-based social simulations (Zhou et al., 2024; Park et al., 2023). This new communication pattern may introduce more complexity and require LLMs to learn when to chime in just like human conversation.

7 CONCLUSION

AsyncLM is a system that enables asynchronous LLM function calling by allowing the LLM and the function executor to operate independently. Our core innovation is making LLM inference interruptible through (1) CML, an in-context interface that facilitates asynchronous interaction, (2) adaptation of LLMs to leverage asynchronous semantics for optimized task completion latency, and (3) efficient implementation of an interrupt mechanism on the LLM inference pipeline. Empirical evaluations on the BFCL benchmark demonstrate that AsyncLM reduces latency by $1.6\times$ – $5.4\times$ compared to synchronous methods. AsyncLM opens new possibilities for improving the operational efficiency of LLMs that interact with external tools, data sources, humans, and other LLMs.

ACKNOWLEDGMENTS

This work is supported in part by NSF Athena AI Institute (Award #2112562) and Yale University.

REFERENCES

- Abhyankar, R., He, Z., Srivatsa, V., Zhang, H., and Zhang, Y. Infercept: Efficient intercept support for augmented large language model inference. In *Proc. ICML*, 2024.
- Brucker, P., Drexler, A., Möhring, R., Neumann, K., and Pesch, E. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 1999.
- Chan, C., Chen, W., Su, Y., Yu, J., Xue, W., Zhang, S., Fu, J., and Liu, Z. Chateval: Towards better llm-based evaluators through multi-agent debate. In *Proc. ICLR*, 2024.
- Chen, G., Yu, X., and Zhong, L. Typefly: Flying drones with large language model. *CoRR*, abs/2312.14950, 2023.
- Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *CoRR*, abs/2407.21783, 2024.
- Gao, S., Chen, Y., and Shu, J. Fast state restoration in llm serving with hcache. *CoRR*, abs/2410.05004, 2024.
- Geng, S., Josifoski, M., Peyrard, M., and West, R. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proc. EMNLP*, 2023.
- Gim, I., Chen, G., Lee, S.-s., Sarda, N., Khandelwal, A., and Zhong, L. Prompt Cache: Modular attention reuse for low-latency inference. In *Proc. MLSys*, 2024.
- Hartmann, S. and Briskorn, D. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 2022.
- Huang, X., Liu, W., Chen, X., Wang, X., Wang, H., Lian, D., Wang, Y., Tang, R., and Chen, E. Understanding the planning of LLM agents: A survey. *CoRR*, abs/2402.02716, 2024.
- Hugging-Face. Text generation inference. <https://github.com/huggingface/text-generation-inference>, 2023. Large Language Model Text Generation Inference Server in Rust and Python.
- Khattab, O., Santhanam, K., Li, X. L., Hall, D., Liang, P., Potts, C., and Zaharia, M. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive NLP. *CoRR*, abs/2212.14024, 2022.
- Kim, S., Moon, S., Tabrizi, R., Lee, N., Mahoney, M. W., Keutzer, K., and Gholami, A. An llm compiler for parallel function calling. In *Proc. ICML*, 2023.
- Liang, Y., Wu, C., Song, T., Wu, W., Xia, Y., Liu, Y., Ou, Y., Lu, S., Ji, L., Mao, S., et al. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. *Intelligent Computing*, 2024.
- Lu, J., Holleis, T., Zhang, Y., Aumayer, B., Nan, F., Bai, F., Ma, S., Ma, S., Li, M., Yin, G., Wang, Z., and Pang, R. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. *CoRR*, abs/2408.04682, 2024.
- Mialon, G., Dessì, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., Rozière, B., Schick, T., Dwivedi-Yu, J., Celikyilmaz, A., Grave, E., LeCun, Y., and Scialom, T. Augmented language models: a survey. *Trans. Mach. Learn. Res.*, 2023.
- OpenAI. Parallel function calling, 2023. URL <https://platform.openai.com/docs/guides/function-calling>. Accessed: 2024-10-26.
- Pan, L., Albalak, A., Wang, X., and Wang, W. Y. Logiclm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Proc. EMNLP*, 2023.
- Park, J. S., O’Brien, J., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. Generative agents: Interactive simulacra of human behavior. In *Proc. ACM UIST*, 2023.
- Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. Gorilla: Large language model connected with massive apis. In *Proc. NeurIPS*, 2024.
- Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. In *Proc. NeurIPS*, 2024.
- Shen, Y., Song, K., Tan, X., Li, D., Lu, W., and Zhuang, Y. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. In *Proc. NeurIPS*, 2024.
- Singh, S., Fore, M., Karatzas, A., Lee, C., Jian, Y., Shang-guan, L., Yu, F., Anagnostopoulos, I., and Stamoulis, D. Llm-dcache: Improving tool-augmented llms with gpt-driven localized data caching. *CoRR*, abs/2406.06799, 2024a.
- Singh, S., Karatzas, A., Fore, M., Anagnostopoulos, I., and Stamoulis, D. An llm-tool compiler for fused parallel function calling. *CoRR*, abs/2405.17438, 2024b.
- Trinh, T. H., Wu, Y., Le, Q. V., He, H., and Luong, T. Solving olympiad geometry without human demonstrations. *Nature*, 2024.

- Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 2024.
- Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Transformers: State-of-the-art natural language processing. In *Proc. NeurIPS*, 2020.
- Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., and Wang, C. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *CoRR*, abs/2308.08155, 2023.
- Xu, B., Peng, Z., Lei, B., Mukherjee, S., Liu, Y., and Xu, D. Rewoo: Decoupling reasoning from observations for efficient augmented language models. *CoRR*, abs/2305.18323, 2023.
- Xu, Y., Kong, X., Chen, T., and Zhuo, D. Conveyor: Efficient tool-aware LLM serving with tool partial execution. *CoRR*, abs/2406.00059, 2024.
- Yan, F., Mao, H., Ji, C. C.-J., Zhang, T., Patil, S. G., Stoica, I., and Gonzalez, J. E. Berkeley function calling leaderboard. <https://gorilla.cs.berkeley.edu>, 2024.
- Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *Proc. ICLR*, 2023.
- Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. τ -bench: A benchmark for tool-agent-user interaction in real-world domains. *CoRR*, abs/2406.12045, 2024.
- Yu, L. and Li, J. Stateful large language model serving with pensieve. *CoRR*, abs/2312.05516, 2023.
- Zaharia, M., Khattab, O., Chen, L., Davis, J. Q., Miller, H., Potts, C., Zou, J., Carbin, M., Frankle, J., Rao, N., and Ghodsi, A. The shift from models to compound ai systems. <https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/>, 2024.
- Zheng, Y., Zhang, R., Zhang, J., Ye, Y., Luo, Z., Feng, Z., and Ma, Y. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proc. ACL*, 2024.
- Zhou, X., Zhu, H., Mathur, L., Zhang, R., Yu, H., Qi, Z., Morency, L., Bisk, Y., Fried, D., Neubig, G., and Sap, M. SOTOPIA: interactive evaluation for social intelligence in language agents. In *Proc. ICLR*, 2024.

APPENDIX

A LATENCY ANALYSIS

In this section, we analyze the total latency of different execution models for a set of independent functions and provide proof sketches for the theorems that demonstrate the advantages of asynchronous function calling.

A.1 Context and Definitions

Let $F = \{f_1, f_2, \dots, f_n\}$ be a set of n independent functions. Each function $f \in F$ has a token generation latency $G(f) > 0$ and an execution time $E(f) > 0$. We define the average generation latency $\bar{G} = \frac{1}{n} \sum_{f \in F} G(f)$ and the average execution time $\bar{E} = \frac{1}{n} \sum_{f \in F} E(f)$.

The total latencies under different execution models are as follows.

- *Synchronous execution (Sync)*: Tokens are generated and functions are executed sequentially:

$$L_{\text{Sync}}(F) = \sum_{f \in F} G(f) + \sum_{f \in F} E(f) = n(\bar{G} + \bar{E}).$$

- *Synchronous with concurrent execution (Sync-Parallel)*: Tokens are generated sequentially, functions are executed concurrently:

$$L_{\text{Sync-Parallel}}(F) = \sum_{f \in F} G(f) + \max_{f \in F} E(f).$$

- *Asynchronous execution (Async)* using the *Longest Processing Time (LPT)* heuristic: Tokens are generated sequentially, functions are scheduled in decreasing order of $E(f)$:

$$L_{\text{Async}}(F) = \max_{f \in F} \left(E(f) + \sum_{g \in \text{pred}(f, F)} G(g) \right),$$

where $\text{pred}(f, F) = \{g \in F \mid E(f) \leq E(g)\}$.

A.2 Proof Sketch of Theorem 6.1

Theorem 6.1 *For any set of independent functions F ,*

$$L_{\text{Async}}(F) \leq L_{\text{Sync-Parallel}}(F) < L_{\text{Sync}}(F).$$

1. Establishing $L_{\text{Sync-Parallel}}(F) < L_{\text{Sync}}(F)$:

Since the maximum execution time is less than the sum of all execution times,

$$\max_{f \in F} E(f) < \sum_{f \in F} E(f),$$

it follows that

$$L_{\text{Sync-Parallel}}(F) < \sum_{f \in F} G(f) + \sum_{f \in F} E(f) = L_{\text{Sync}}(F).$$

2. Establishing $L_{\text{Async}}(F) \leq L_{\text{Sync-Parallel}}(F)$:

Under the LPT heuristic, functions are ordered such that $E(f_1) \geq E(f_2) \geq \dots \geq E(f_n)$. The completion time of function f_i is

$$C(f_i) = \left(\sum_{j=1}^i G(f_j) \right) + E(f_i).$$

The total latency is $L_{\text{Async}}(F) = \max_{1 \leq i \leq n} C(f_i)$. Since for all i ,

$$\sum_{j=1}^i G(f_j) \leq \sum_{f \in F} G(f), \quad E(f_i) \leq E(f_1) = \max_{f \in F} E(f),$$

we have

$$C(f_i) \leq \sum_{f \in F} G(f) + \max_{f \in F} E(f) = L_{\text{Sync-Parallel}}(F),$$

and thus $L_{\text{Async}}(F) \leq L_{\text{Sync-Parallel}}(F)$.

A.3 Proof Sketch of Theorem 6.2

Theorem 6.2 *The ratio $\frac{L_{\text{Sync}}(F)}{L_{\text{Async}}(F)}$ is approximately $1 + \frac{\bar{E}}{\bar{G}}$ with an error order of $\left(\frac{\bar{E}}{\bar{G}}\right)^2$ when $n = |F|$ is large.*

When n is large, we can approximate the cumulative token generation latency and the maximum execution time using order statistics, assuming $E(f)$ follows a normal distribution:

$$\sum_{f \in F} G(f) \approx n\bar{G}, \quad \max_{f \in F} E(f) \approx \bar{E} + \sigma\sqrt{2 \ln n}.$$

Using the inequality $L_{\text{Async}}(F) \leq L_{\text{Sync-Parallel}}(F)$, we approximate the total latency as

$$L_{\text{Async}}(F) \approx n\bar{G} + \bar{E} + \sigma\sqrt{2 \ln n}.$$

The ratio of total latencies is then

$$\frac{L_{\text{Sync}}(F)}{L_{\text{Async}}(F)} \approx \frac{n\bar{G} + n\bar{E}}{n\bar{G} + \bar{E} + \sigma\sqrt{2 \ln n}}.$$

Since $\varepsilon = \frac{\bar{E} + \sigma\sqrt{2 \ln n}}{n\bar{G}} \ll 1$ for large n , we can approximate $\frac{1}{1 + \varepsilon} \approx 1 - \varepsilon$. Applying this approximation,

$$\frac{L_{\text{Sync}}(F)}{L_{\text{Async}}(F)} \approx \left(1 + \frac{\bar{E}}{\bar{G}}\right) \left(1 - \frac{\bar{E} + \sigma\sqrt{2 \ln n}}{n\bar{G}}\right).$$

Neglecting terms of order $O(1/n)$, we obtain

$$\frac{L_{\text{Sync}}(F)}{L_{\text{Async}}(F)} \approx 1 + \frac{\bar{E}}{\bar{G}}.$$

The error in this approximation is of order $\left(\frac{\bar{E}}{\bar{G}}\right)^2$.

A.4 Proof Sketch of Theorem 6.3

Theorem 6.3 *Any deviation from the LPT order cannot result in a lower total latency.*

Assume, for contradiction, that there exists a schedule σ' that deviates from the LPT order and yields a lower total latency than the LPT schedule σ^* . This implies that in σ' , there exist functions f_i and f_j such that $E(f_i) < E(f_j)$ but f_j is scheduled after f_i .

Let the cumulative token generation latency before f_i and f_j in σ' be S_i and S_j , respectively, with $S_j = S_i + G(f_i)$. The completion time is:

$$C_{\sigma'}(f_j) = S_j + G(f_j) + E(f_j) = S_i + G(f_i) + G(f_j) + E(f_j).$$

Consider swapping f_i and f_j to obtain schedule σ'' . The sum of token generation latencies become $S_j'' = S_i$ for f_j , and $S_i'' = S_j'' + G(f_j)$ for f_i . The new completion time is

$$C_{\sigma''}(f_i) = S_i'' + G(f_i) + E(f_i) = S_i + G(f_j) + G(f_i) + E(f_i).$$

Comparing the completion times before and after the swap:

$$C_{\sigma''}(f_i) - C_{\sigma'}(f_i) = E(f_i) - E(f_j) < 0.$$

Thus, swapping f_i and f_j always reduces the maximum completion time. By repeatedly applying such swaps, any schedule can be transformed into the LPT schedule without increasing the total latency. This contradicts the assumption that σ' yields a lower total latency than σ^* . Therefore, any deviation from the LPT order cannot result in a lower total latency.