This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3619112

Efficient Function Orchestration for Large Language Models

Xiaoxia Liu, Peng Di, Cong Li, Jun Sun, and Jingyi Wang*

***Abstract*—Function calling is a fundamental capability of today's large language models, but sequential function calling posed efficiency problems. Recent studies have proposed to request function calls with parallelism support in order to alleviate this issue. However, they either delegate the concurrent function calls to users for execution which are conversely executed sequentially, or overlook the relations among various function calls, rending limited efficiency. This paper introduces `LLMOrch`, an advanced framework for automated, parallel function calling in large language models. The key principle behind `LLMOrch` is to identify an available processor to execute a function call while preventing any single processor from becoming overburdened. To this end, `LLMOrch` models the data relations (i.e., def-use) among different function calls and coordinates their executions by their control relations (i.e., mutual-exclusion) as well as the working status of the underlying processors. When comparing with state-of-the-art techniques, `LLMOrch` demonstrated comparable efficiency improvements in orchestrating I/O-intensive functions, while significantly outperforming (2×) them with compute-intensive functions. `LLMOrch`'s performance even showed a linear correlation to the number of allocated processors. We believe that these results highlight the potential of `LLMOrch` as an efficient solution for parallel function orchestration in the context of large language models.**

***Index Terms*—Large language models, function call, parallel function call, task scheduling.**

I. Introduction

Recent advancements in large language models (LLMs) [1], [2], [3] have led to the development of AI agents such as AutoGPT [4], SWE-agent [5], and Agentless [6]. Besides programming tasks, LLM-driven agents also led to improvements in intricate real-world challenges, including scientific computations [7], [8], software engineering [9], [10], protein engineering [11], [12], and cellular research [13].

LLM-driven agents extensively rely on external user functions (or tools) to expand their capabilities. To better align external functions with LLM's inherent capabilities, Yao et al. proposed the ReAct [14] framework for selecting functions and observing their outcomes in a sequential loop. Since when, function calling has been one of the defacto, fundamental LLM capabilities. For example, GPT-3.5 [15], ChatGLM3 [16], and LLaMA3 [17] support native function calling. LangChain [18] supplies a rich and convenient toolbox for a wide range of LLMs to call user functions. There are also datasets [19], [20] and tools [21] for augmenting existing LLMs with function calling capabilities. All these works significantly benefit LLM's capability towards function calling. Yet, their sequential nature poses a considerable problem towards efficiency [22], [23], [24], [25]. To alleviate the problem, recent studies focus on calling functions concurrently. OpenAI [26], [27], [28] and Google [29] released Parallel Function Calling (or ParallelFC) for empowering their own GPT-/Gemini-series LLMs to request multiple function calls in one LLM response. `LLM-Tool Compiler` [30] further accelerates the execution by fusing related operations into a single call, enabling token reuse and latency reduction. `LLMCompiler`, on the other hand, is able to concurrently execute function calls for *any* LLMs, even without ParallelFC capability [31]. These works have achieved modest efficiency improvements when addressing user queries.

Xiaoxia Liu, Cong Li and Jingyi Wang* are with Zhejiang University, Zhejiang 310007, China (e-mail: liuxiaoxia@zju.edu.cn; chifei.lc@antgroup.com; wangjyee@zju.edu.cn).

Peng Di and Cong Li are with Ant Group, Zhejiang 310020, China (e-mail: dipeng.dp@antgroup.com; chifei.lc@antgroup.com).

Jun Sun is with Singapore Management University, Singapore 188065, Singapore (e-mail: junsun@smu.edu.sg).

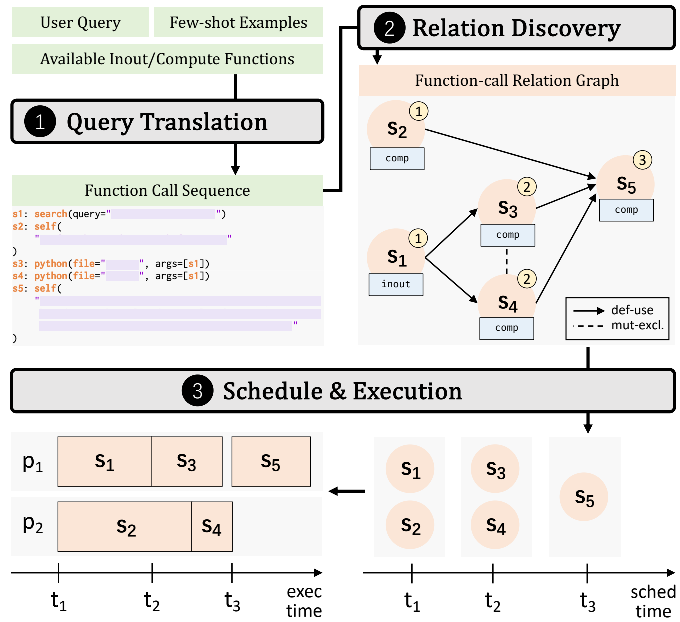

Fig. 1: Overview of `LLMOrch`. Each node in the Function-call Relation Graph is assigned a rank, represented by a number in the top-right yellow circle, which is computed based on the *def-use* (data) relations. The set of function calls with the same rank are scheduled concurrently for example s_1/s_2 and s_3/s_4, though their scheduling does not immediately trigger their execution. `LLMOrch` manages this coordination through their *mutual-exclusion* (control) relations and the current work status of the underlying processors. In this example, p_1 and p_2 represent two physical processors; s_3 and s_4 are coordinated to them respectively because they are mutual-exclusive function calls.

Despite their efforts, we identified two major limitations. (1) *Lack* of automatic parallel orchestration: Neither ParallelFC nor `LLM-Tool Compiler` supports automatic parallel orchestration, necessitating manual scheduling and execution by users. This limitation has resulted in sequential processing of function calls, even in those ParallelFC's official examples [28], [32]. (2) *Lack* of efficient parallel orchestration: While `LLMCompiler` attempts to improve parallelization performance by managing the schedule and execution of function calls, it often fails to consider the relations (either dependencies or exclusions) among function calls, rendering only limited efficiency improvements. These limitations present challenges for systems with resource constraints that depend on LLM-driven agents, for instance, AI-powered operating systems [33], [34] and robotic systems [35], [36], [37]. In these systems, the core LLM-drivens agent must organize and execute numerous (I/O- or compute-intensive) tasks such as image or text processing, data analysis, and robotic arm movement with limited computational resources. Improper function orchestration can lead to degraded performance.

`LLMOrch`. We present `LLMOrch` (Figure 1), an automatic and efficient function orchestrator with fine-grained parallelism support for parallel function calling for large language models. The key idea behind `LLMOrch` is to identify an available processor to execute a function call while preventing any single processor from becoming overburdened. To achieve this, we design `LLMOrch` around two key features: (1) Discover data and control relations from function calls and (2) Separate function call scheduling and execution to enable better parallelism.

In this paper, data relations (i.e., *def-use*) refer to the data flow dependencies among various function calls, while control relations (i.e., *mutual-exclusion*) model their execution (I/O- or compute-intensive) conditions. `LLMOrch`'s function orchestration follows a work-sharing style [38] in multithreaded computation, yet it differentiates between these two types of relations. In particular, for a user query and all available functions, `LLMOrch` builds a *Function-call Relation Graph* (FRG) to represent the relations, discovered from a function call sequence–a sequential, step-by-step plan for addressing the user query through function calls. `LLMOrch` then follows FRG to schedule function calls concurrently and coordinate their executions. Specifically, a function call is scheduled for execution immediately after all its data-dependent function calls are scheduled, this allowing for multiple function calls to be scheduled at a time. However in `LLMOrch`, the scheduling does not lead to their direct execution. Instead, `LLMOrch` avoids the concurrent execution of multiple mutual-exclusive function calls on one processor. `LLMOrch` queues the set of concurrent function calls and distributes them to execute on specific processors, if the processors are available and their mutual-exclusive calls are already distributed (either to a different processor or the same one after they complete).

When compared against ReAct [14], ParallelFC [28], and `LLMCompiler` [31], `LLMOrch` demonstrated comparable efficiency improvements in orchestrating I/O-intensive functions, while significantly outperforming them with functions involving considerable computations. Moreover, we observed that the improvements achieved by `LLMOrch` positively corelated to the number of allocated processors. In real-world scenarios "Purchase Intent Analysis" and "End-to-End Encryption", `LLMOrch` successfully addressed the given user queries with satisfactory answers similar to prior works. We believe that these results highlight `LLMOrch`'s usefulness and practicability, which we contribute to the discovery of data and control relations, as well as the separation of function call scheduling and execution. Furthermore, we hope that `LLMOrch` represents a small yet significant advancement in the field of function orchestration.

Contributions. Our main contributions are:

- We present a framework, `LLMOrch`, designed to orchestrate function calls in parallel by introducing the novel consideration of both data and control relations, and separating the processes of scheduling and execution.
- We evaluate `LLMOrch` against state-of-the-art techniques where `LLMOrch` achieved superior performance when orchestrating both I/O- and compute-intensive functions.
- We open-source `LLMOrch` to facilitate the community and future research: https://www.hostize.com/v/c3oLTBMUwn.

II. Illustrative Example

`LLMOrch` decomposes the orchestration of function calls for an LLM into three steps as illustrated in Figure 1: (1) Query Translation, (2) Relation Discovery, and (3) Schedule & Execution. This section walks through the process following an example with a (simplified) user query:

```
I want to do a search engine optimization (SEO)
audit. I'll firstly search the webpage
(https://openai.com/). I also need chatbot for a
SEO optimization example. Then I'll request the
local python file ("seo.py") for Technical SEO
Audit, and the local python file ("con.py") for
Content Analysis. Finally, I prefer a summary
report.
```

and available functions:

- `search(query)`: Search `query` through search engines and obtain the top-k results.
- `chatbot(prompt)`: Ask an LLM for an answer or a completion towards `prompt`.
- `python(file,args)`: Execute the python code in `file` with arguments `args`.

In this example, we will orchestrate these functions for GPT-4. We deployed LLaMA3-8B locally on CPUs (i.e., neither on GPUs nor remotely via a remote API call)[1] for `chatbot`.

Query Translation. Given the user query and all available functions, `LLMOrch` instructs LLMs to translate the query into a sequence of function calls. Unlike the defacto function call [44] and the ReAct [14] paradigm where LLMs request function calls progressively, we require LLMs to comprehend the user query and organize a function call sequence, following `LLMCompiler` [31] and AutoGPT [4]. We demand LLMs to attach each function call with a *unique ID*. For function calls requiring results of prior calls, the unique IDs (or IDs) serve as their results and can be referred to as arguments.

[1]This can be accomplished via like llama.cpp [39], MLC-LLM [40], PowerInfer [41], etc. The scenario is realistic and common for advanced users today and we believe it is the future for mobile operating systems [42], [43].

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3619112


```
s1: search(query="https://openai.com/")
s2: chatbot("Please give me a SEO optimization
          example.")
s3: python(file="seo.py", args=[s1])
s4: python(file="con.py", args=[s1])
s5: chatbot(
    "Generate a comprehensive SEO audit summary
          report based on technical audit results
     from {s3}, content analysis from {s4}, and
          keyword research {s2}."
)
```

Fig. 2: The function call sequence of our illustrative example after query translation. The grammar is similar to those used by ReAct and `LLMCompiler`. Each function call is given a unique ID (e.g., `s1`) which also serves as the result of the function call. Different function calls have explicit data-dependencies and implicit control exclusions; these decide their order for subsequent schedule and execution.

Figure 2 displays the function call sequence for our illustrative example, by GPT-4 [1]. In this example, GPT-4 first conducted search using `search`, then prompted LLaMA3 for an SEO example. It then sequentially executed `seo.py` and `con.py` using the results obtained in `s1` respectively for technical SEO and content analysis, and finally summarized a report by LLaMA3.

Relation Discovery. `LLMOrch` discovers the inter-relations among various function calls. In this paper, we consider two types of relations: data relations and control relations.

Data relations refer to the data flow relations among different function calls. Such relations are indicated by the definition of a call's results and all its uses in other calls' arguments. In the function call sequence, we explicitly capture these *def-use relations* through unique IDs and their references. The function calls, together with their *def-use relations*, construct a data dependency graph which is directed and acyclic. Accordingly, we assign each function call with a *rank* following their topological order in the graph. For instance, function calls `s3` (rank: 2) and `s4` (rank: 2) are data-dependent (in solid arrows) on the function call `s1` (rank: 1) in Figure 1. It is worthwhile mentioning that function calls with the same rank means they can be called concurrently.

Control relations model the behaviors of the functions. In this paper, we categorize available functions into two traditional groups: I/O-intensive functions (or *inout* functions) that primarily execute input-output operations without tying up processors, and compute-intensive functions (or *compute* functions) that exclusively occupy a processor until they pause or stop. As the first work that attempts to analyze control relations in parallel function orchestration, we conservatively consider that two function calls are *mutual exclusive* to each other if both of them are compute functions with the same rank. In our example, `search` is an inout function and `chatbot` and `python` are compute functions. In Figure 1, `s3` (rank: 2) are mutual exclusive (in dashed lines) to `s4` (rank: 2).

We incorporate all identified control relations into the data dependency graph by introducing "bidirectional, virtual edges". In this paper, we refer to the resulting directed acyclic graph as *Function-call Relation Graph* (FRG). The function call sequence is scheduled and subsequently executed according to FRG. The FRG for Figure 2 is presented in Figure 1.

TABLE I: Comparison with state-of-the-art works on function calling and orchestration. No works except `LLMOrch` consider control relations and support execution coordination. `LTCompiler` represents `LLM-Tool Compiler`, featured with the ability of function fusing.

Work	Sequential Calling	Parallel Calling	Data Relation	Control Relation	Execution Coordina.
ReAct [14]	✓	×	×	×	×
ParallelFC [28]	✓	✓	×	×	×
`LTCompiler` [30]	✓	✓	×	×	×
`LLMCompiler` [31]	✓	✓	✓	×	×
`LLMOrch`	✓	✓	✓	✓	✓

Schedule & Execution. `LLMOrch` schedules (or submits) function calls for execution based on their rank and data relations in FRG, starting with calls assigned a rank of 1. Note that scheduling a function call does not immediately result in its execution; `LLMOrch` coordinates its execution with other function calls to achieve commendable parallelism. As a completed function call becomes available, `LLMOrch` concurrently submits all function calls data-dependent on it and with an incremental (+1) rank. The scheduling process stops once all function calls have been scheduled.

As for the FRG of our illustrative example (Figure 1), the function calls `s1` and `s2` are initially scheduled concurrently. Upon completion of `s2`, `s5` will not be scheduled until `s3` and `s4`–which are scheduled after `s1`'s completion–also finish.

For each set of concurrent function calls scheduled for execution, `LLMOrch` queues them and then coordinates them according to their control relations in FRG and the work status of the underlying processors. The principle is to find a spare processor to execute a function call while preventing any processor from becoming overburdened (Algorithm 3). The coordinator thereby prioritizes the execution of inout function calls and separates the execution of mutual-exclusive calls; If there is an insufficient number of processors, `LLMOrch` sequentializes them.

Following the FRG in Figure 1, `s1` is executed before `s2` if there is only one processor available. `s3` and `s4` will be executed on separate processors if there are over two spare processors, otherwise they may be executed sequentially.

Discussions. We recognize that existing works ReAct [14], ParallelFC [28], and `LLMCompiler` [31] can accomplish the task with satisfactory results. However, they all struggle when it comes to fast function orchestration in resource-constrained settings. For instance, `LLMOrch` is able to provide responses to the user query in our illustrative example in 53.22 seconds when running on a device with two physical processors. However, the state-of-the-art `LLMCompiler` took 81.32 seconds to accomplish the same query, while ParallelFC and ReAct required 85.28 and 93.37 seconds, respectively. ReAct's unsatisfactory efficiency is attributed to its sequential nature. ParallelFC is always waiting for the completion of all parallel function calls before proceeding with the next

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3619112

call. `LLMCompiler` directly executes all concurrent function calls (like `s2`, `s3`, and `s4`) after each completed function call (like `s1`), without considering the control relations among them and without providing flexible execution coordination. In such cases, a system may become stuck if there are too many compute function call processes. In contrast, `LLMOrch` delays the execution of a compute function call unless there are processors available. This significantly sets our work apart from all the others. We summarize the key differences of these works in Table I.

III. The LLMOrch Framework

This section expands Section II with implementation details.

A. Query Translation

`LLMOrch` begins by translating the user query into a *Function Call Sequence*, following the context-free grammar provided below, which also serves as annotations for this section:

Sequence	σ	$\rightarrow$	κ^+
Func Call	κ	$\rightarrow$	i: $f\,(\alpha^*)$
Unique ID	i	$\rightarrow$	$\mathtt{s}_0$ | $\mathtt{s}_1$ | $\mathtt{s}_2$ | $\cdots$
Argument	α	$\rightarrow$	$n = e$
Expression	e	$\rightarrow$	i | v | e+e | e−e | $\cdots$
Arg Name	n	$\rightarrow$	name of a function argument
Arg Value	v	$\rightarrow$	number | string | array | $\cdots$
Function	f	$\rightarrow$	`self` | user-defined functions

This grammar was designed to be in line with ReAct [14] and `LLMCompiler` [31], with the intention of making the generated programs simple, easily learned by LLMs. In particular, a function call sequence (σ) is an array of function calls (κ), each with a unique ID (i) assigned to it. These unique IDs can be referenced to by other function calls as arguments (α). This establishes the def-use relations—the data relations that we consider in this paper—for scheduling them. The functions (f) that can be included in a function call sequence are either `self` or those defined by users to address the user query. It is important to note that the sequences generated are indifferent to inout and compute functions. However, `LLMOrch` is sensitive to them, and we ask users to explicitly label them as either inout or compute when passing them to `LLMOrch`.

To facilitate translation, we crafted prompts with few-shot examples, which we found to be more effective than directly prompting LLMs with the grammar. We conducted a small study involving 100 user queries from the HotpotQA [45] benchmark, of which over 80% failed when generating with the grammar. Conversely, the successful ratio with few-shot examples is over 85%.

B. Relation Discovery

`LLMOrch` then builds a *Function-call Relation Graph* (FRG) to represent the data and control relations among function calls.

Function-call Relation Graph. The FRG of a function call sequence $\sigma = [\kappa_1, \kappa_2, \cdots, \kappa_{|\sigma|}]$ is a directed acyclic graph $G = \langle K, R_1, R_2\rangle$:

Algorithm 1: Assigning ranks

```
1  function Assigner(FnCalls K, DataRelations R₁)
2      R ← R₁
3      Q ← orderedset {κ'' ∈ K | ∄κ' ∈ K. ⟨κ', κ''⟩ ∈ R}
4      for each κ ∈ Q do setrank (κ, 1)
5      while |Q| ≠ 0 do
6          κ' ← popfront (Q)
7          for each κ'' ∈ K. ⟨κ', κ''⟩ ∈ R do
8              R ← R / {⟨κ', κ''⟩}
9              if ∄κ''' ∈ K. ⟨κ''', κ''⟩ ∈ R then
10                 setrank (κ'', rank (κ') + 1)
11                 pushback (Q, κ'')
```

Algorithm 2: Scheduling function calls

```
function Scheduler(FnCalls K, DataRelations R₁)
    S ← set {}
    sendcoord ({κ ∈ K | rank (κ) = 1})
    while |S| ≠ |K| do
        C ← recvcoord ()
        while |C| ≠ 0 do
            κ' ← popfront (C)
            S' ← {κ'' ∈ K |
                  ⟨κ', κ''⟩ ∈ R₁ ∧ rank (κ'') = rank (κ') + 1}
            sendcoord (S')
            S ← S ∪ S'
```

- $K = \{\kappa \mid \kappa \in \sigma\}$ is the set of function calls in the sequence σ, representing G's nodes.
- $R_1 : K \times K$ is the set of def-use (data) relations among K, serving as G's directed edges:

$$\forall\langle\kappa', \kappa''\rangle \in R_1.\ \exists\alpha.\ \mathtt{uses}\,(\kappa''.\alpha.e,\ \kappa'.i)$$

  where $\mathtt{uses}\,(e, i)$ determines whether the expression e uses the definition i.
- $R_2 : K \times K$ is the set of mutual-exclusion (control) relations among K holding that for any $(\kappa', \kappa'') \in R_2$:

$$\mathtt{comp}\,(\kappa'.f) \wedge \mathtt{comp}\,(\kappa''.f) \wedge \mathtt{rank}\,(\kappa') = \mathtt{rank}\,(\kappa'')$$

  where $\mathtt{comp}\,(f)$ decides if the function f is a compute function; $\mathtt{rank}\,(\kappa)$ returns the *rank* of a function call κ.

Rank Assignment. While constructing FRG, specifically after creating K and R_1 and before creating R_2, `LLMOrch` assigns each function call $\kappa \in K$ with a *rank* based on their topological order as defined by K and R_1. In particular, the *Rank Assigner* (Algorithm 1) starts from nodes without incoming edges (i.e., function calls without data dependencies, Lines 3–4). We iteratively assign a function call κ'' with an incremental rank (Line 10) if it is data-dependent on κ' (Line 7) and all other data dependencies have been considered (Line 9). The process stops when all function calls have been assigned a rank (Line 5).

Discussion. Prior works [31] consider the def-use relations among different function calls while orchestrating them like ours. However, they did overlook their control relations, which were shown to be effective for efficient function orchestrations (Section IV-C). In this paper, we consider the mutual-exclusive control relations and combine them with the def-use data relations into FRG. It guides `LLMOrch`'s subsequent function orchestration.

Algorithm 3: Coordinating function call executions

```
1  [tb]
2  function Coordinator(FunctionCalls K, ControlRelations R₂)
3      S ← deque {} // Function calls that have been scheduled
4      C ← set {} // Function calls that have completed
5      thread
6          while |S| ≠ |K| do
7              // Wait for scheduler sending back newly scheduled function calls
8              pushback (S, recvsched ())
9      while |C| ≠ |K| do
10         P ← spareprocs () // Wait until there are spare processors
11         C′ ← the ordered set of calls ever completed on P
12         C ← C ∪ C′
13         // Notify scheduler of newly completed function calls
14         sendsched (C′)
15         S′ ← popfront (S) // Fetch the first set of concurrent calls for coordination and
             execution
16         M ← {κ′ ∈ S′ | ∃κ″ ∈ S′. (κ′, κ″) ∈ R₂}
17         I ← S′ / M
18         if |I| ≠ 0 then // Inout functions are given higher priorities
19             execute (I, {random (P)}) // Distribute all inout function calls to
                 processor p
20         if |M| ≠ 0 then // Execute them either exclusively or sequentially
21             execute (popfront (M, count=|P|) , P)
22             if |M| ≠ 0 then  pushfront (S, M) // Re-coordinate the
                 remaining function calls in the next iteration
```

C. Schedule & Execution

`LLMOrch` follows a work-sharing style [38] to schedule and execute the function call sequence σ based on its FRG: $G = \langle K, R_1, R_2 \rangle$. Unlike prior works [31], [28], `LLMOrch` separates the schedule of a function call from its execution to achieve a better parallelism. The basic idea is that: A function call will be scheduled once all its preceding data-dependent (as indicated by R_1) function calls are complete, while it may not be immediately executed; `LLMOrch` coordinates the execution of all submitted, concurrent function calls according to their mutual-exclusion relations (as implied by R_2) and the work status of the underlying processors.

`LLMOrch` thereby involves a *Call Scheduler* and an *Execution Coordinator*. They interact with each other through the following interfaces:

- `sendcoord` (S): The scheduler submits to the coordinator a set S comprising function calls intended to be executed concurrently.
- `sendsched` (C): The coordinator sends back to the scheduler the set of all completed calls C whenever new completions are available.
- `recvcoord` $()$: The scheduler waits until the coordinator returns all completed function calls.
- `recvsched` $()$: The coordinator waits until the scheduler provides a new set of function calls for execution.

Call Scheduling. The scheduler (Algorithm 2) submits function calls for coordination and execution based on their data relations (i.e., def-use relations as reflected by their ranks in this paper)–a common practice in task scheduling . The scheduler begins by submitting all function calls with rank 1 (Line 4) as they have no dependencies. The scheduler then waits until one of them completes, notified by the coordinator (Line 7). Upon the completion of a function call κ', the scheduler identifies the next set S' of concurrent calls and submits them for execution (Lines 9–12). Each function call $\kappa'' \in S'$ should be data-dependent on κ' and have an incremental rank (Line 10), meaning that it is data-dependent solely on κ' after κ' completes. The process continues until all function calls have been scheduled (Line 5).

Execution Coordination. Not all submitted function calls are executed immediately. The coordinator's principle is to locate an available processor to execute a function call while preventing any processor from becoming overburdened. To this end, the coordinator enforces constraints based on control relations (i.e., mutual-exclusive relations in this paper) and the current work status of the underlying processors:

1) Mutual-exclusive calls should not be coordinated to the same processor for execution as each of them occupies the processor until completion.
2) Inout functions are given higher priorities than computing functions as they will be tied up.
3) If there is an insufficient number of spare processors, `LLMOrch` can break the first constraint.

The detailed process is presented in Algorithm 3. The coordinator starts by initializing a thread to receive and queue newly scheduled function calls (Lines 4–7). When a set of available processors P (Line 9) is present, the coordinator notifies the scheduler of all the completed function calls (Lines 10–13). Subsequently, the coordinator coordinates the execution of the earliest set S' of scheduled, concurrent function calls (Line 14). In particular, it identifies all mutual-exclusive compute function calls (M, Line 15) and all the remaining inout function calls (I, Line 16). The coordinator prioritizes by assigning all inout calls to a single processor as each will be tied up after execution (Lines 17–18). As for mutual-exclusive calls, the coordinator separates their executions by fetching the first $|P|$ calls and distributing each to a single available processor (Lines 19–20). If there are any remaining calls yet to be coordinated, the coordinator manages them in the next iteration (Line 21) such that the processors are not overburdened.

Discussion. In addition to control relations, the separation of function call scheduling and execution as well as the coordination of function call executions distinguish our work from all previous ones [14], [28], [31]. Even though `LLMOrch` has employed a simple coordination policy (following work-sharing [38]) when realizing our instinctive principle, the coordinator was evaluated to be effective when providing an efficient function orchestration, especially for compute calls, as shown in Section IV-C. In contrast, `LLMCompiler` overlooks the availability of computational resources, directly delegating function call execution to the underlying runtime's scheduler, specifically Python's co-routine scheduler `asyncio` [46], with all function calls being executed in a single thread.

D. Errors & Recovery

`LLMOrch` is likely to encounter various errors. To address this, it incorporates a recovery mechanism to handle two types of errors: (1) Compile-time errors, which arise when LLMs fail to translate the user query into a valid function call sequence, resulting in such as invalid syntax and undefined IDs; (2)

Runtime errors, which occur when the execution of a function call does not normally exit.

Compile-time Recovery. `LLMOrch` employs a feedback-driven method—a standard treatment in the current state of program repair [47] and LLM-driven agents [48], [4]—to manage compile-time errors. When it fails to parse a function call sequence due to syntax or semantic violations, `LLMOrch` creates a repair prompt containing the error message, the erroneous function call sequence, and the few-shot examples used for user query translation. It then requests an LLM to repair the function call sequence to align with the provided examples. This repair process continues until no compile-time errors are detected or a maximum number of attempts is reached. If the repair fails, `LLMOrch` increases the temperature and calls the LLM to re-translate the user query from scratch, allowing for a limited number of additional attempts. If all attempts fail, `LLMOrch` is unable to handle the user query.

Runtime Recovery. One possible method is to restart `LLMOrch` from scratch, allowing it to generate a new function call sequence as in compile-time recovery. However, this would necessitate re-scheduling and re-executing all function calls, resulting in additional overhead. To mitigate this, `LLMOrch` identifies *recovery point*s in FRG and resume scheduling and execution from those points. In LLM-driven agents that execute LLM-generated, cascading function calls, we observed that the failure of a function call is often attributable to insufficient information provided by its data-dependent calls. Therefore, `LLMOrch` identifies recovery points by tracing back through the FRG; this treatment resembles to tracing back through stack trace to find the root cause of a bug. In particular, `LLMOrch` records the execution results of each executed function call. When there are *failed call*s identified at Line 10 (Algorithm 3)—potentially multiple due to the concurrent nature of function execution—`LLMOrch` first clears all scheduled but unexecuted function calls (i.e., $S - C$) from S. Based on the conservative assumption that errors occur in close proximity, it identifies their *closest* parents in FRG *sharing* a *smallest* rank as recovery points, and removes them and their children from S and C. After that, `LLMOrch` employs an LLM to repair all calls at recovery points (*recovery call*s for short) one by one, additionally providing the failed call (including its ID, function name, and arguments), any exceptions raised, and all available functions, in order to generate more accurate information for the failed call after repairing the recovery call. This may involve fixing arguments or substituting the function with a new one. Finally, it resumes scheduling all recovery calls. As in the compile-time recovery, `LLMOrch` allows runtime recovery only for a limited number of attempts; if these attempts fail, `LLMOrch` exits, complaining that it cannot resolve the user query.

Discussion. `LLMOrch` conservatively assumes that the root causes of errors occur in close proximity. Nevertheless, there is a performance-accuracy trade-off when determining the depth of tracing in FRG. `LLMOrch` opts to trace back only to the closest parents, as this sufficed in our experiments. For more complex user queries, determining the appropriate depth remains a challenge; we plan to address this in future work. Additionally, the LLM may replace an inout function with a compute function (or vice versa) during runtime recovery, which could alter FRG. However, we did not observe such instances in our evaluation, and the current implementation of `LLMOrch` prevents this from occurring. Moreover, `LLMOrch` does not support dynamic re-planning [31] as it is unable to obtain the correct answer for a practical user query.

IV. Experimental Evaluation

We structured our evaluation for `LLMOrch` around the following research questions:

RQ1 How does `LLMOrch` compare to state-of-the-art techniques when orchestrating inout functions? Can it achieve comparable performance? Since inout functions are the most frequently called in today's LLMs, this research question seeks to determine whether `LLMOrch` compromises performance on inout functions to support compute functions.

RQ2 When handling user queries with intensive compute function calls, how does `LLMOrch` fare? Can it deliver substantial efficiency improvements without compromising accuracy? The primary goal of `LLMOrch` is to support calling functions with heavy computation; this research question examines that objective.

RQ3 Does `LLMOrch`'s recovery mechanism effectively function in RQ1 and RQ2? Can it address real-world errors encountered while handling user queries? On average, how many recovery attempts are needed to address a raised error?

RQ4 How does the performance of `LLMOrch` scale as more processors are allocated?

RQ5 Can `LLMOrch` be effectively utilized in real-world scenarios?

A. Experimental Setup

Baselines. We selected ReAct [14], Parallel Function Calling (ParallelFC) [28], and `LLMCompiler` [31] as baselines[2] (Table I), and evaluated them on GPT-3.5-Turbo. To ensure a fair comparison, we reused—yet refined to the best of our efforts—`LLMCompiler`'s prompt and in-context examples to fit `LLMOrch` when translating user queries into function call sequences. We also adhered to `LLMCompiler`'s experimental setup when configuring ReAct and OpenAI's ParallelFC (or OpenAI PFC), including their prompts, in-context examples, and LLM parameters. However, instead of sequentially calling OpenAI PFC-generated parallel function calls (as seen in their official examples and `LLMCompiler`'s settings), we delegated the execution of function calls to Python's `asyncio`—the same runtime used by `LLMCompiler`—to enable concurrency.

Metrics. We evaluated the techniques based on their *accuracy*, *latency speedup*, and *token costs*, following `LLMCompiler`. Accuracy refers to the correctness of the answer obtained after addressing a user query using a specific technique.

[2]We did not include `LLM-Tool Compiler` in our evaluation as it is not yet open-source by the time of writing this paper.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3619112

Latency represents the end-to-end runtime (in seconds) for addressing a user query. Latency speedup measures the efficiency improvement achieved by a technique compared to ReAct, the sequential orchestration technique. Token costs are the collective input and output tokens consumed during the addressing of a user query.

Other Configurations. Our experiments were conducted on an Apple Macbook Pro with macOS 12.1, 32 GiB RAM, and a 10-core M1 Pro CPU. All results were obtained by averaging three runs. GPT-3.5-Turbo (1106) and some functions (e.g., `wiki`) are deployed remotely and we interacted with them through APIs. Therefore, the latency presented in our results may be influenced by internet conditions, but we ensured all the techniques were assessed in a same environment.

B. RQ1. Effectiveness on I/O-Intensive Tasks

Similar to `LLMCompiler`, we chose HotpotQA [45], Movie Recommendation (or MovieRec) [49], and ParallelQA [31] as our benchmark datasets. User queries of these datasets rely more on inout functions (`wiki`, `self`) than compute functions (`math`, `read`, `write`), but are with different parallelism complexity .

- HotpotQA comprises 1,500 user queries designed to compare two distinct entities based on specific aspects, for example, "Are both Duke Energy and Affiliated Managers Group based in Massachusetts?". The dataset represents the lowest complexity level, typically involving two concurrent `search`es and a final `self` for summarization.
- MovieRec, derived from Google's open-source BIG-bench project, consists of 500 user queries. These queries task to select the most similar movie from a group of four movies in comparison to another group of four movies. This dataset demonstrates moderate complexity, typically involving eight concurrent `search`es and a final `self` for summarization.
- ParallelQA is specially crafted by `LLMCompiler`, containing 113 user queries. The FRG constructed to address these queries typically follows three different, much more complex patterns, each involving 5–8 function calls and an average depth (i.e., the maximum rank) of four.

We enforced two processors for HotpotQA and eight processors for other datasets.

Table II's "RQ1: I/O-Intensive Evaluation" section showcases the results. Overall, `LLMOrch` achieved the highest latency speedup without compromising accuracy on all benchmark datasets, however, the average speedup of OpenAI PFC, `LLMCompiler`, and `LLMOrch` are all comparable with around 1.82×, 1.89×, and 2.18×, respectively. This situation is because these datasets rely more on inout function calls which are quickly suspended waiting for the I/O to finish, during when their processors switch to another function call for all techniques except ReAct. Among the three datasets, all techniques brought more speedups on the MovieRec benchmark possibly due to the higher number of concurrent function calls at a time for each user query. In this benchmark, `LLMOrch` accelerated the process by an average of 3.04×. It is noteworthy that, compared to the state-of-the-art `LLMCompiler`, the speedup of `LLMOrch` increased with the complexity of the dependencies (FRGs), even in such IO-intensive circumstances. Specifically, `LLMOrch` achieves the highest speedup (∼1.21×) on ParallelQA–the most complex benchmark—with 5–8 function calls and an average FRG depth of four—evaluated in this research question. In terms of token costs, `LLMOrch` achieved a reduction of up to 3.57×, 6.60× and 4.07× for HotpotQA, MovieRec and ParallelQA. This likely contributes to the query translation step in `LLMCompiler` and `LLMOrch`, which removes the need to iteratively interacting with LLMs and to generating a thought at each single step as in ReAct. OpenAI PFC is also the former case.

> **Answer to RQ1:** `LLMOrch` demonstrates comparable (actually superior) performance, particularly latency and token cost, to state-of-the-art techniques when orchestrating inout function calls, maintaining similar or even improved accuracy.

C. RQ2. Effectiveness on Compute-Intensive Tasks

As we did not find suitable datasets for this new problem, we specially adapted the widely used KITTI dataset [50] in autonomous driving and AGNews dataset [51] in text classification as our benchmark dataset. We named them KITTI* and AGNews*.

- KITTI*. We sampled 1,600 images and created 200 user queries accordingly to simulate real-world differential testing of two autonomous driving systems. For each query, we tasked it to calculate the difference of the steering angles of two groups of scenes, with each group containing four scenes from one system. This involves eight concurrent, compute-intensive `stereorcnn`es for steering angle detection, followed by two concurrent `self`s for averaging the angles per group and a final `self` to output their difference.
- AGNews*. The dataset comprises 120,000 samples from over 2,000 news sources. We created 120 user queries, each utilizing 1,000 samples to simulate a real-world data mining and analysis task, specifically for t-distributed Stochastic Neighbor Embedding (t-SNE) and Latent Dirichlet Allocation (LDA). Each query consists of five function calls: a `self` to generate guidelines for the task, a `read` to retrieve 1,000 news from the dataset, two concurrent compute-intensive calls `t-SNE` and `LDA` to perform data mining and visualization, and a final `write` to export the results.

We enforced four processors for KITTI* and two processors for AGNews*.

The "RQ2: Compute-Intensive Evaluation" section in Table II displays the results. In contrast to the previous experiment, the speedup introduced by `LLMOrch` were significantly higher (at least 1.32×) even than `LLMCompiler`. `LLMOrch`'s speedup over `LLMCompiler` reached even 1.94× on the KITTI* benchmark. We attribute this to the

TABLE II: Comparison results for different benchmarks. *Costs* indicates the token costs (dollars/1,000tokens). For techniques with recovery/replanning mechanisms, *#Recoveries* denotes the average number of recoveries required to address an error, or "∞" if the errors cannot be addressed out of two attempts; *#Errors* (x/y) shows the number of successfully addressed errors (x) out of all raised errors (y).

Benchmark	Technique	GPT-3.5-Turbo					
		Accuracy	*Latency*	*Speedup*	*Costs*	*#Errors*	*#Recoveries*
RQ1: I/O-Intensive Evaluation							
HotpotQA	ReAct	62.13%	7.61s	1.00×	5.00$	—	—
	OpenAI PFC	62.00%	5.39s	1.41×	1.57$	—	—
	`LLMCompiler`	62.00%	5.02s	1.52×	1.47$	0 / 3	∞
	`LLMOrch`	62.04%	4.80s	**1.59×**	1.401$	3 / 3	1
MovieRec	ReAct	77.20%	25.20s	1.00×	20.46$	—	—
	OpenAI PFC	77.40%	10.07s	2.50×	3.15$	—	—
	`LLMCompiler`	77.60%	9.79s	2.57×	3.04$	0 / 3	∞
	`LLMOrch`	77.40%	8.27s	**3.04×**	3.10$	3 / 3	1
ParallelQA	ReAct	89.30%	40.52s	1.00×	480$	—	—
	OpenAI PFC	88.50%	25.97s	1.56×	121$	—	—
	`LLMCompiler`	88.90%	25.44s	1.59×	103$	0 / 9	∞
	`LLMOrch`	89.44%	21.15s	**1.92×**	118$	9 / 9	1
RQ2: Compute-Intensive Evaluation							
KITTI*	ReAct	46.88%	303.45s	1.00×	15.60$	—	—
	OpenAI PFC	46.87%	277.67s	1.09×	2.51$	—	—
	`LLMCompiler`	46.87%	278.68s	1.09×	2.40$	0 / 0	0
	`LLMOrch`	46.87%	143.19s	**2.12×**	2.20$	0 / 0	0
AGNews*	ReAct	100.00%	58.72s	1.00×	4.25$	—	—
	OpenAI PFC	100.00%	39.53s	1.49×	2.33$	—	—
	`LLMCompiler`	100.00%	38.82s	1.51×	2.20$	0 / 0	0
	`LLMOrch`	100.00%	29.44s	**1.99×**	1.53$	0 / 0	0

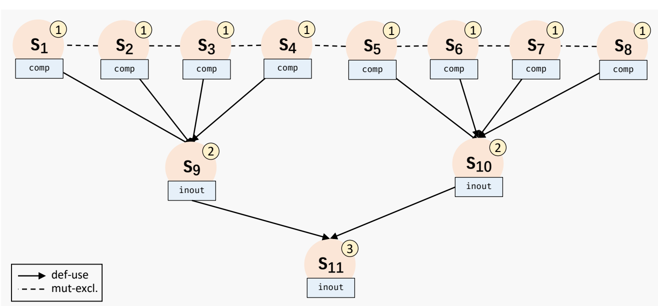

Fig. 3: FRG for The Example in RQ2.

discovered control relations and the separation of scheduling and execution on a resource-constrained machine (four available processors with eight concurrent compute function calls, or two available processors with five function calls). In `LLMCompiler`, all concurrent, computing function calls are executed simultaneously immediately after they are submitted, resulting in a $>2\times$ overburden of processors. Actually, the `asyncio`'s job scheduler schedules them sequentially in a single thread as they block the processor. In this case, the kernel's thread scheduler again schedules the single thread together with other threads periodically. In contrast, the four additional function calls are not seen by the kernel's thread scheduler until our coordinator coordinates them to a specific processor. As for AGNews*, the speedup of `LLMOrch` was less significant than that on KITTI* as the total number of concurrent compute calls is 2, aligning with the number of processors that we have allocated to it. As for token costs, `LLMOrch` achieved a reduction of around 7.09× on KITTI* benchmark and 2.78× on AGNews* benchmark, similar to OpenAI PFC and `LLMCompiler`.

KITTI* Example. The following displays a (simplified) example of the KITTI* benchmark:

```
We collect stereo images for both systems A and B,
named from 000001.png to 000008.png, with
odd-numbered images for system A and even-numbered
images for system B. Then, process all images
using the 3D object detection algorithm
stereorcnn.py to calculate the steering angles for
each system across different 4 perspectives.
Compute the average steering angles of the two
systems using a chatbot. Finally, calculate the
difference in average steering angles between
systems A and B to quantify the difference in
their stability.
```

Upon receiving the query, `LLMOrch` constructed its FRG as in Figure 3. The graph involved two sets of concurrent steering angle detection tasks, each set employing the compute function `stereorcnn` four times (s_1–s_4 and s_5–s_8), followed by a inout function call `self` (s_9 and s_{10}) to average all detected angles per group. The final difference of the two systems was output with an additional, inout function call `self` (s_{11}). When coordinating them to four processors, s_1–s_4 were executed initially on four independent processors. After their completion, s_5–s_8 were executed on the same 4 independent processors. Subsequently, s_9 and s_{10} were executed on an arbitrary processor as they were inout calls. Lastly, s_{11} was executed on an arbitrary processor.

AGNews* Example. Below is a (simplified) example for the AGNews* benchmark:

```
I want to visualize the collected news data. I
will read dataset details from readme.txt. Then
use the tsne.py script to visualize the
distribution of texts in the dataset, and use the
lda.py script to visualize the topic modeling of
```

```
the dataset. Meanwhile, I need a brief sample
procedure for t-SNE and LDA analysis on NLP tasks
from the chatbot. Finally, all outputs should be
saved in a folder called /result.
```

Although five function calls, the dependencies in each user query are not trivial. Indeed, the FRG and the overall scheduling procedure are consistent with Figure 1.

> **Answer to RQ2:** Compared to state-of-the-art techniques, `LLMOrch` demonstrated a significantly superior speedup in latency for compute-intensive function orchestrations without compromising the accuracy. The token costs of `LLMOrch` is comparable to them.

D. RQ3. Errors and Recovery

To evaluate whether `LLMOrch`'s recovery mechanism effectively addresses errors, we recorded instances of errors and recoveries during previous experiments. This investigation focused solely on `LLMCompiler` and `LLMOrch`, as ReAct and OpenAI's PFC do not support replanning or recovery. We granted `LLMCompiler` and `LLMOrch` two attempts.

The last two columns (*#Errors* and *#Recoveries*) in Table II summarize the results. Overall, we observed three (out of 1,500), three (out of 500), and nine (out of 113) runtime errors in HotpotQA, MovieRec, and ParallelQA, respectively. We did not find any compile-time errors, possibly contributing to the simple few-shot examples (rather than the complex grammar) we utilized for user query translation. No errors were raised in KITTI* and AGNews*, likely due to the straightforwardness of the user queries that we designed. These results also match the instinct that complex queries, characterized by numerous function calls and stricter input/output formats, e.g., invoking the "math" function in ParallelQA, are more susceptible to exceptions.

For all the runtime errors, `LLMOrch` succeeded on the first attempt, while `LLMCompiler` failed both times. After manual inspection, we realized that the failure of `LLMCompiler` was due to its (static) replanning strategy, which repeated the same operation without adjusting to runtime dynamic changes during execution (for example incorporating runtime feedback).

Example: ParallelQA #83. Below is an example that demonstrates `LLMOrch`'s recovery mechanism. In this instance, all three baseline techniques (ReAct, ParallelFC, and `LLMCompiler`) failed to handle the user query. ReAct and ParallelFC do not have replanning or recovery capabilities, while `LLMCompiler` attempted replanning but still failed. In contrast, `LLMOrch` was able to resolve the runtime errors within a single recovery attempt and successfully obtained the correct answer (488.19). The user query used in this example is as follows:

```
If Texas and Florida were to merge and become one
state, as well as California and Michigan, what
would be the largest population density among
these 2 new states and New Jersey? Answer in
people / square km.
```

All available functions are:

- `search(term,k=500)`: Search `term` in Wikipedia and obtain the first `k` words into a summary.
- `math(prompt)`: Ask an LLM-based calculator for mathematics results by generating a rigorous mathematical expression that conforms to Python syntax based on the `prompt`.

The function call sequences initially generated by `LLMCompiler` and `LLMOrch` are shown in Figure 5 and Figure 6, respectively.

Errors. When handling the query, `LLMOrch` and `LLMCompiler` may throw errors at varied `math` calls: `s3`/`5`, `s4`/`6`, `s12`/`14`, `s5`/`16` or `s10`/`17`. The errors occur because the preceding `search` (e.g., `s2`/`4`) did not retrieve enough information (500 words by default), causing the subsequent `math` to lack valid data in generating a rigorous mathematical expression, finally leading to execution failures.

Recovery. As it was unable to obtain the correct answer for the user query, `LLMCompiler`'s dynamic replanning capability is downgraded to static replanning, which requires LLMs to regenerate the overall function call sequence and reschedule all new function calls to run—similar to `LLMOrch`'s compile-time recovery. Despite this recovery, `LLMCompiler` still failed within two additional attempts even though with different function call sequences, raising the same errors at `math`. In contrast, `LLMOrch` captured the failed `math` calls `s3` and `s4` with the exception message "can't extract the population/area of Florida" and started tracing back through FRG. In this example, the recovery call tracked was `s2`. Subsequently, the failed calls `s3` and `s4`, along with the exception message and all available functions, were sent to GPT-3.5-Turbo for repair of `s2`. GPT-3.5-Turbo successfully identified the error and updated the `k` argument to `1000`, resulting in a call that provided the information for Florida's population and area. Finally, `LLMOrch` resumed scheduling from `s2`.

> **Answer to RQ3:** `LLMOrch`'s recovery mechanism effectively functioned in addressing the runtime errors during RQ1 and RQ2 within one recovery attempts.

E. RQ4. Performance on Different #Processors

This experiment concerns how `LLMOrch`'s performance scale in terms of the number of allocated processors. In this experiment, we reused the KITTI* benchmark. We accessed `LLMOrch`'s performance against ReAct's and `LLMCompiler`'s on processors ranging from one to eight–the maximum number of concurrent function calls allowed at a time in the benchmark. In this research question, all speedup are calculated based on ReAct's latency on the one processor setting.

Table III presents the results with Figure 4 plotting the trend of speedups. We found that the speedup of `LLMOrch` scales nearly *linearly* as the number of allocated processors increases continuously, except for a sudden latency reduction due to scheduling when allocating 4 processors. Particularly, the speedup reached up to 3.26× when allocating eight processors. In contrast, the state-of-the-art tool `LLMCompiler` achieved only up to 1.25× with speedups increasing seemingly

TABLE III: Statistics of latency speedups with allocated processors on the crafted KITTI benchmark. Columns "*Spd.*" represent speedups. The latency speedups are calculated between a respective technique and ReAct.

Technique	1 Processor		2 Processors		3 Processors		4 Processors	
	Latency	*Spd.*	*Latency*	*Spd.*	*Latency*	*Spd.*	*Latency*	*Spd.*
ReAct	328.08s	1.00×	316.66s	1.04×	312.12s	1.05×	303.45s	1.08×
OpenAI PFC	290.82s	1.13×	284.68s	1.15×	282.63s	1.16×	277.67s	1.18×
`LLMCompiler`	286.47s	1.15×	284.21s	1.15×	280.85s	1.17×	278.68s	1.18×
`LLMOrch`	280.36s	**1.17×**	218.52s	**1.50×**	208.19s	**1.58×**	143.19s	**2.29×**
Technique	5 Processors		6 Processors		7 Processors		8 Processors	
	Latency	*Spd.*	*Latency*	*Spd.*	*Latency*	*Spd.*	*Latency*	*Spd.*
ReAct	290.82s	1.13×	286.47s	1.15×	284.85s	1.15×	279.24s	1.17×
OpenAI PFC	275.54s	1.19×	270.24s	1.21×	265.24s	1.24×	264.47s	1.25×
`LLMCompiler`	272.04s	1.21×	269.81s	1.22×	267.11s	1.23×	262.24s	1.25×
`LLMOrch`	156.20s	**2.10×**	149.63s	**2.19×**	134.72s	**2.44×**	100.53s	**3.26×**

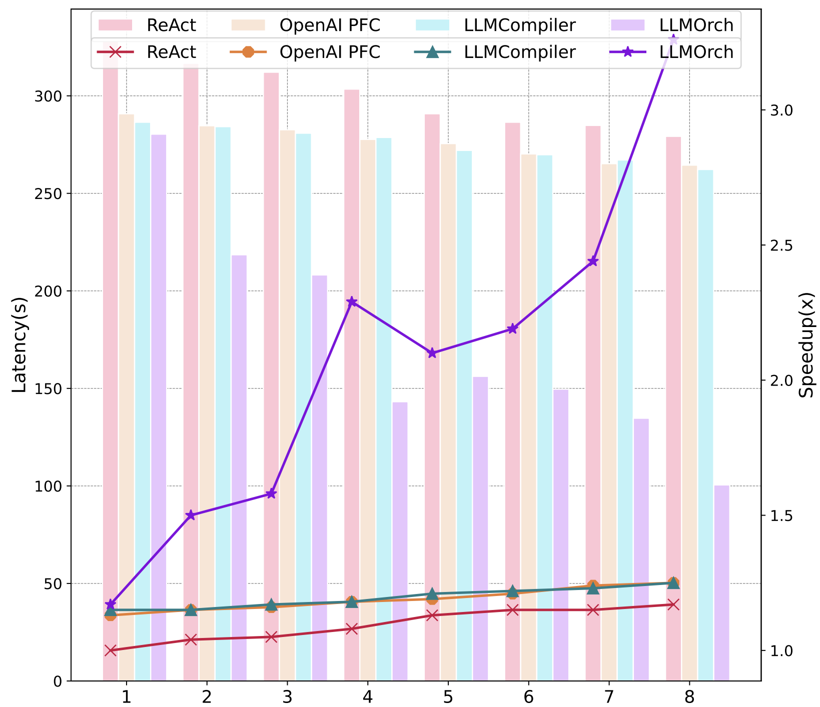

Fig. 4: Latency speedups with allocated processors on KITTI*.

logarithmically and OpenAI PFC was close to it. They execute all concurrent function calls in parallel immediately after they are scheduled. ReAct was hardly influenced by the number of processors as it is sequential. It is worthwhile mentioning that when the number of allocated processors reached 4, `LLMOrch` significantly (nearly 2×) outperformed the other two parallel techniques, thanks to our execution coordination, which alleviated the work burden of the underlying processors. It should be noted that, when allocating 5 to 7 processors, the speedup of `LLMOrch` appears to be similar to that of four processors. This is because the compute function calls (s_1–s_8 in Figure 3) in each query of the KITTI* dataset are homogeneous, taking a comparable amount of time to complete. As a result, one of the input function calls (e.g., s_{10}) must wait until all its predecessors finish. This issue is alleviated with eight processors, as they tend to stop nearly simultaneously.

> **Answer to RQ4:** The speedup achieved by `LLMOrch` demonstrates a linear relationship with the number of allocated processors, while the accuracy is maintained within a marginal range.

F. RQ5. Real-World Case Studies

Finally, we conducted two further real-world case studies to assess the practicability of `LLMOrch`, in addition to our illustrative example (Section II).

Purchase Intent Analysis. This is widely used in e-commerce and digital marketing. It summarizes end users' purchase behaviors by analyzing their behavioral data, interests, preferences, and purchase history, playing a significant role in these industries. In this case, we plan to treat LLMs as a recommendation system to analyze a user's purchasing behavior. Specifically, based on the purchase history of a user, we requested LLMs to predict whether the user would

```
1: search('Texas')
2: math('popul. of Texas in M?', ['$1'])
3: math('area of Texas in km^2?', ['$1'])
4: search('Florida')
5: math('popul. of Florida in M?', ['$4'])
6: math('area of Florida in km^2?', ['$4'])
7: search('California')
8: math('popul. of California in M?', ['$7'])
9: math('area of California in km^2?', ['$7'])
10: search('Michigan')
11: math('popul. of Michigan in M?', ['$10'])
12: math('area of Michigan in km^2?', ['$10'])
13: search('New Jersey')
14: math('popul. of N.J. in M?', ['$13'])
15: math('area of N.J. in km^2?', ['$13'])
16: math('($2 + $5) / ($3 + $6)')
17: math('($8 + $11) / ($9 + $12)')
18: math('$14 / $15')
19: math('max($16, $17, $18)')
20: join()
```

Fig. 5: `LLMCompiler`'s function call sequence. In the strings, "popul." is short for "population", "M" for "millions", and "N.J." for "New Jersey".

```
s1: search("Texas")
s2: search("Florida")
s3: math("Question: total population of Texas
        and Florida? {s1} {s2}")
s4: math("Question: total area of Texas and
        Florida in km^2? {s1} {s2}")
s5: math(" {s3} / {s4} ")
s6: search("California")
s7: search("Michigan")
s8: math("Question: total population of
        California and Michigan? {s6} {s7}")
s9: math("Question: total area of California
        and Michigan in km^2? {s6} {s7}")
s10: math(" {s8} / {s9} ")
s11: search("New Jersey")
s12: math("Question: population density of New
         Jersey in people / km^2? {s11}")
s13: math("max({s5}, {s10}, {s12})")
```

Fig. 6: `LLMOrch`'s function call sequence, sharing a similar grammar to that of `LLMCompiler`, both with each function call given a unique ID (e.g., `1`, `s1`). The ID serves as the result of the respective function call and can be referenced to by subsequent calls.

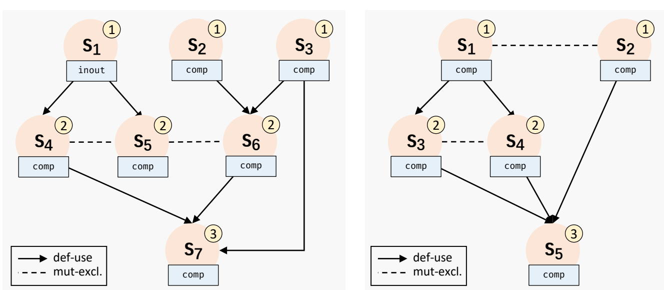

Fig. 7: The FRG for case studies "Purchase Intent Analysis" (left) and "End-to-End Encryption" (right).

be interested in purchasing Apple products. We provide the following (simplified) prompts:

```
Treat yourself as a personalized user
recommendation system to analyze users' purchase
intentions towards some products. First, search
Apple products via https://www.apple.com/. Use
chatbot to generate a tutorial for analyzing user
portraits. Then analyze the user's purchase
history (history.txt) by the script analyze.py.
Then use the content analysis script content.py
and the user experience analysis script user.py I
provided to analyze the product homepage. Let
chatbot build a user portrait based on the
historical analysis and the obtained tutorial.
Finally, judge the user's purchasing behavior
based on the product analysis, the user experience
analysis, and user portrait.
```

and available functions:

- `search(query)`: Search `query` through search engines and obtain the results.
- `chatbot(prompt)`: Ask an LLM for an answer or a completion towards `prompt`.
- `python(file,args)`: Execute the python code in `file` with arguments `args`.

As in our illustrative example, we deployed a local LLaMA3-8B as `chatbot`, rending it being a compute function as `python`. `search` is an inout function. It should be noted that we intentionally streamlined the user query so that our LLMs can parse and generate a correct function call sequence for this complex scenario, as `LLMOrch` does not have a re-planning component.

`LLMOrch` obtained the following function call sequence.

```
s1: search(query="https://www.apple.com/")
s2: chatbot("Generate a tutorial for analyzing user
          portraits.")
s3: python(file="analyze.py", args=["history.txt"])
s4: python(file="content.py", args=[s1])
s5: python(file="user.py", args=[s1])
s6: chatbot(
    "Build a user portrait based on the following
          historical record analysis and user
          portrait
     tutorial:
     ## Historical Record Analysis
     {s3}
     ## User Portrait Tutorial
     {s2}"
)
s7: chatbot(
    "Judge the user's purchasing behavior based on
          the following product analysis, historical
     record analysis and user portrait:
     ## Product Analysis
     {s4}
     ## Historical Record Analysis
     {s3}
     ## User Portrait
     {s6}"
)
```

It can be observed that the user query is processed into seven function calls, including one inout function call (`s1`) and six compute function calls (`s2`–`s7`). The left sub-figure of Figure 7 presents the FRG for them.

When running on three processors, `LLMOrch` was able to respond in around 77.10s, ∼1.71× (132.53s), ∼1.54× (119.00s), and ∼1.48× (114.62s) faster than ReAct, OpenAI PFC, and `LLMCompiler`, respectively, and all techniques output a similar, satisfactory response. It is observed when handling queries involving relatively complex data and control relations, `LLMCompiler` exhibits slight inefficiencies

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3619112

in scheduling, whereas `LLMOrch`, achieves a certain improvement in processing speed. However, we also find that it remains challenging for OpenAI PFC, `LLMCompiler`, and even `LLMOrch` to achieve an optimal acceleration even with increased computational resources when the topological relationships between function calls become complex. We discussed this limitation and possible future works in Section IV-G.

End-to-End Encryption. End-to-end encryption is commonly employed in practice, where a message sender encrypts a message and a message receiver decrypts it. In this example, we aim to simulate a real-world scenario where an end user without programming skills seeks to utilize LLMs to (1) encrypt his chat (specifically the video and text record) using some provided encryption scripts and (2) obtain the encryption statistics towards them.

We provided `LLMOrch` with below (simplified) user query:

```
I want to do end-to-end encryption for privacy
protection during a video call. I have a video
file myvideo.mp4 and a chat text file chat.txt. I
first need to generate a key for AES encryption
and also obtain an example to evaluate the
encryption effect (seems like the chatbot can do
both). Then use the ASE encryption tool to encrypt
the video and text. Finally, analyze the results
of encryption by learning from the example and
tell me.
```

In addition to `chatbot(prompt)`, we further offered `aes(action,file,key)` which returns the statistics (e.g., time, ratio) of encrypting/decrypting (specified by `action`) `file` with `key`. Both functions are compute functions.

`LLMOrch` obtained the following function call sequence:

```
s1: chatbot("Generate a key for aes encryption.")
s2: chatbot("Give me an example to analyze the
         encryption effect.")
s3: aes(action="encrypt", file="myvideo.mp4",
         key=s1)
s4: aes(action="encrypt", file="chat.txt", key=s1)
s5: chatbot(
    "Learn from the following example on how to
         analyze the encryption effect: {s2}
     Then summarize the results of the following
         two encryptions: {s3} and {s4}."
)
```

The FRG is displayed in the right sub-figure of Figure 7.

`LLMOrch` responded in approximately 53.82 seconds, while ReAct, OpenAI PFC, and `LLMCompiler` took approximately 109.11s ($\sim$2.03$\times$), 103.09s ($\sim$1.92$\times$), and 101.26s ($\sim$1.88$\times$), respectively, when running on two processors. This notable efficiency improvement can be attributed to `LLMOrch`'s coordination of the relationship between compute function calls and available processors. Specifically, in this case where no input functions are given, the advantages of `LLMCompiler` and OpenAI PFC over the sequential method ReAct become minimal. Furthermore, all techniques successfully encrypted both files.

> **Answer to RQ5:** `LLMOrch` is applicable to addressing real-world scenarios, providing satisfactory answers as state-of-the-art techniques while bringing superior improved efficiency.

G. Discussion

In summary, ParallelFC, `LLMCompiler`, and `LLMOrch` were all successful in reducing the latency for solving user queries and meanwhile in maintaining the accuracy within a marginal range in our evaluation, when compared with ReAct. As for `LLMOrch`, it demonstrated comparable speedups to the others when orchestrating inout function calls (Section IV-B), while significantly outperforming them for compute function calls (Section IV-C). It was also observed that the speedups achieved by `LLMOrch` had a linear relationship with the number of allocated processors (Section IV-E). When applying `LLMOrch` in real-world scenarios “Purchase Intent Analysis” and “End-to-End Encryption”, it successfully addressed the given user queries with a correct answer (Section IV-F). We believe that these results showcase the usefulness and practicability of `LLMOrch` in parallel function orchestration, specifically thanks to its discovery of data and control relations, as well as the separation of function call scheduling and execution.

Limitations. Despite the promising evaluation results `LLMOrch` has achieved, we realized that `LLMOrch` is primarily limited by its direct query translation, which translates a user query into a function call sequence without conditional (if/loop) structures. While this is effective for common scenarios, such as those in our and `LLMCompiler`'s evaluations, it falls short when the user query is complex and conditional, or if LLMs fail to produce a correct function call sequence even after our compile-time recovery. Another limitation is that the current `LLMOrch` only supports an unchanged FRG during runtime recovery. For complex user queries that require numerous reasoning steps, where accuracy is more critical than performance, limiting `LLMOrch` to trace back to the closest recovery point and disallowing replacing inout functions with compute functions (and vice versa) may restrict `LLMOrch`'s effectiveness.

Future Work. Beyond addressing the above limitations, there are several potential, exciting extensions for enhancing `LLMOrch` in the future: (1) Modeling the side effects of functions. In this work, we only consider the def-use (data) relations assuming that functions are pure. However, it is likely in practice that a function call alters its used arguments or even a global resource, essentially re-defining them. By modeling the side effects of functions, a widely used practice in compilers and language virtual machines, can help capture this behavior, leading to correct schedules for such cases. (2) Refining control relations to be more granular. In this paper, we only consider CPU-compute functions and consider two such function calls to be mutually exclusive if they are assigned the same rank. Yet, this strict criterion can lead to increased latency for GPU-compute functions, as two of such calls could still be coordinated for parallel execution if they do not occupy the full VRAM. We believe that making mutual-exclusive relations much finer-grained in the future could bring further speedups.

Threats to Validity. The first threat is the potential bias of the intentionally crafted (KITTI*) benchmarks, which may not fully represent practical real-world scenarios and could be influenced by human preferences. We tried our best to sample diverse and representative data and create queries simulating real-world scenarios. The second threat is that the benchmarks we employed in our evaluation might not be as complex as real-world scenarios. To mitigate this threat, we initially used the same benchmarks as `LLMCompiler`, the state-of-the-art baseline in parallel function orchestration. Additionally, we designed additional benchmarks (KITTI* and AGNews*), ensuring that each query included a minimum of eight intensive compute calls, each lasting for at least $\geq$65 seconds when executing on M1 Pro, along with several input calls. Furthermore, we provided case studies involving real-world scenarios to assess the practicality of `LLMOrch`. We believe that such experimental settings are sufficient to evaluate the performance of `LLMOrch`.

V. Related Work

Function Orchestration. Function calling has been a fundamental capability of LLMs since the advent of ReAct [14], to the best of our knowledge. ReAct orchestrates function calls by sequentially generating a thought, selecting a function, and observing the function's outcome for the next iteration unless the user query is resolved. Subsequently, frameworks like LangChain [18] followed ReAct to orchestrate functions sequentially and interact with LLMs. On June 13, 2023, OpenAI enabled native function calling for its GPT-series models [15], similar to ReAct. After that, datasets [19], [20], tools [21], and LLM variants [52], [20] were also created to augment existing open-source LLMs with native function calling capabilities. Berkeley created a leader board to benchmark LLM's function calling capabilities [53]. Today, function calling has been a defacto capability for LLMs like LLaMA3 [17], ChatGLM3 [16], and GLM4 [54].

As of November 6, 2023, OpenAI upgraded the function calling capability to Parallel Function Calling [28], [26], [27], enabling LLMs to produce multiple function call requests at a time. This enhancement aims to improve efficiency when orchestrating functions. However, such capabilities currently only exist in GPT-series models. Meanwhile, `LLMCompiler` proposed a framework to orchestrate function calls concurrently for any LLMs [31]. Like `LLMOrch`, `LLMCompiler` translates a user query into a sequence of function calls, analyzes their inter-dependencies, and executes all concurrent function calls directly. `LLM-Tool Compiler` optimizes function calling by analyzing and fusing multiple calls into a single one [30].

In contrast, `LLMOrch` concentrates on discovering and leveraging the data and control relations among function calls to improve the efficiency.

Prompt Engineering. Another series of work is to devise efficient approaches to interacting with LLMs and maximizing their inherent, inference capabilities [55]. These works can be categorized into three groups: (1) Prompting Approaches for example zero- and few-shot prompting, chain [56] or tree [57] or graph [58] of thoughts, and retrieval augmented generation [59]. (2) Prompting Languages for simplifying the interactions with LLMs, such as LMQL [60], SGLang [25]. 3) LLM Wrappers which integrates LLMs into conventional programming languages or tools, such as Semantic Kernel [61], LangChain [18], SuperAGI [62], and NeMo-Guardrails [63]. These works are orthogonal to ours and can be integrated into `LLMOrch`–for example query translation–for better controlling LLMs.

Low-Level Optimizations. Some works focus on accelerating LLM's training, fine-tuning, and inference [64]. Unlike us which put efforts in function calling, these works typically target token- [65], data- [66], model- [67], or system-level [23] optimizations or acceleration. They can be combined with us.

Multithreaded Computation. Efficiently scheduling concurrent tasks to maximize resource utilization and throughput is crucial. To our knowledge, Early List Scheduling [68] established the theoretical foundation for parallel scheduling. It makes static scheduling decisions based on estimated task times and dependencies, but lacks dynamic adjustment, limiting its effectiveness in load balancing and dependency management. Work Stealing [38], [69] addressed the load imbalance issue of List Scheduling, by allowing idle processors to steal tasks from busy ones. But it brought the task distribution and migration, resulting in complex task dependencies, and scalability issues. Classic strategies like Shortest Job First (SJF) [70], First In-First Out(FIFO) [71] and Earliest Deadline First (EDF) [72] prioritize tasks based on the shortest execution time and earliest deadline, respectively. These methods require prior knowledge or estimation of task execution times. Additionally, some approaches [73], [74] optimize parallel scheduling specifically for GPUs, focusing on dense linear algebra computations, while others [75] aim to be suitable for heterogeneous environments (GPU/CPU). `LLMOrch`'s approach follows a work-sharing [38] style, yet it differentiates between two types of relations.

VI. Conclusion

We present `LLMOrch`, an advanced framework for parallel function calling in large language models. The main idea behind `LLMOrch` is to identify an available processor to execute a function call while ensuring that no single processor is overwhelmed. To achieve this, `LLMOrch` models the data relations among function calls and coordinates their execution based on control relations and the work status of processors. When comparing against state-of-the-art techniques, `LLMOrch` showed similar efficiency improvements for inout function calls and significantly outperformed them for compute calls. `LLMOrch` also demonstrated potential in real-world scenarios.

References

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] Z. Du, Y. Qian, X. Liu, M. Ding, J. Qiu, Z. Yang, and J. Tang, "GLM: General language model pretraining with autoregressive blank infilling," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 320–335. [Online]. Available: https://aclanthology.org/2022.acl-long.26

[3] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[4] Significant-Gravitas, "AutoGPT," 2023. [Online]. Available: https://github.com/Significant-Gravitas/AutoGPT

[5] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," 2024.

[6] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," 2024. [Online]. Available: https://arxiv.org/abs/2407.01489

[7] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi *et al.*, "Mathematical discoveries from program search with large language models," *Nature*, vol. 625, no. 7995, pp. 468–475, 2024.

[8] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong, "Solving olympiad geometry without human demonstrations," *Nature*, vol. 625, no. 7995, pp. 476–482, 2024.

[9] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024.

[10] S. Fatima, H. Hemmati, and L. C. Briand, "Flakyfix: Using large language models for predicting flaky test fix categories and test code repair," *IEEE Transactions on Software Engineering*, vol. 50, no. 12, pp. 3146–3171, 2024.

[11] Y. Wang, H. Tang, L. Huang, L. Pan, L. Yang, H. Yang, F. Mu, and M. Yang, "Self-play reinforcement learning guides protein engineering," *Nature Machine Intelligence*, vol. 5, no. 8, pp. 845–860, 2023.

[12] J. T. Rapp, B. J. Bremer, and P. A. Romero, "Self-driving laboratories to autonomously navigate the protein fitness landscape," *Nature Chemical Engineering*, vol. 1, no. 1, pp. 97–107, 2024.

[13] M. Hao, J. Gong, X. Zeng, C. Liu, Y. Guo, X. Cheng, T. Wang, J. Ma, X. Zhang, and L. Song, "Large-scale foundation model on single-cell transcriptomics," *Nature Methods*, pp. 1–11, 2024.

[14] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[15] OpenAI, "Function calling and other api updates," 2023. [Online]. Available: https://openai.com/index/function-calling-and-other-api-updates/

[16] THUDM, "Chatglm3," 2023. [Online]. Available: https://github.com/THUDM/ChatGLM3

[17] Meta, "Llama3," 2024. [Online]. Available: https://llama.meta.com/

[18] H. Chase, "LangChain," Oct. 2022. [Online]. Available: https://github.com/hwchase17/langchain

[19] G. AI, "Glaive function calling," 2023. [Online]. Available: https://huggingface.co/datasets/glaiveai/glaive-function-calling

[20] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," 2023.

[21] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian *et al.*, "Toolllm: Facilitating large language models to master 16000+ real-world apis," *arXiv preprint arXiv:2307.16789*, 2023.

[22] T. Dao, "FlashAttention-2: Faster attention with better parallelism and work partitioning," in *International Conference on Learning Representations (ICLR)*, 2024.

[23] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[24] Y. Fu, P. Bailis, I. Stoica, and H. Zhang, "Break the sequential dependency of LLM inference using lookahead decoding," in *Forty-first International Conference on Machine Learning*, 2024. [Online]. Available: https://openreview.net/forum?id=eDjvSFOkXw

[25] L. Zheng, L. Yin, Z. Xie, J. Huang, C. Sun, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez *et al.*, "Efficiently programming large language models using sglang," *arXiv preprint arXiv:2312.07104*, 2023.

[26] OpenAI, "New models and developer products announced at devday," 2023. [Online]. Available: https://openai.com/index/new-models-and-developer-products-announced-at-devday/

[27] ——, "Pull request: Add parallel function calling," 2023. [Online]. Available: https://github.com/openai/openai-cookbook/pull/838

[28] ——, "Parallel function calling," 2024. [Online]. Available: https://platform.openai.com/docs/guides/function-calling/parallel-function-calling

[29] Gemini, "Introduction to function calling," 2024. [Online]. Available: https://cloud.google.com/vertex-ai/generative-ai/docs/multimodal/function-calling

[30] S. Singh, A. Karatzas, M. Fore, I. Anagnostopoulos, and D. Stamoulis, "An llm-tool compiler for fused parallel function calling," *arXiv preprint arXiv:2405.17438*, 2024.

[31] S. Kim, S. Moon, R. Tabrizi, N. Lee, M. W. Mahoney, K. Keutzer, and A. Gholami, "An LLM compiler for parallel function calling," in *Proceedings of the 41st International Conference on Machine Learning*, 2024. [Online]. Available: https://openreview.net/forum?id=uQ2FUoFjnF

[32] OpenAI, "Function calling," 2024. [Online]. Available: https://platform.openai.com/docs/assistants/tools/function-calling/quickstart

[33] K. Mei, Z. Li, S. Xu, R. Ye, Y. Ge, and Y. Zhang, "Aios: Llm agent operating system," 2024. [Online]. Available: https://arxiv.org/abs/2403.16971

[34] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji, "Executable code actions elicit better LLM agents," in *Proceedings of the 41st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, Eds., vol. 235. PMLR, 21–27 Jul 2024, pp. 50 208–50 232. [Online]. Available: https://proceedings.mlr.press/v235/wang24h.html

[35] C. Wang, S. Hasler, D. Tanneberg, F. Ocker, F. Joublin, A. Ceravola, J. Deigmoeller, and M. Gienger, "Lami: Large language models for multi-modal human-robot interaction," in *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, ser. CHI '24, vol. 2. ACM, May 2024, p. 1–10. [Online]. Available: http://dx.doi.org/10.1145/3613905.3651029

[36] A. Zeng, B. Ichter, F. Xia, T. Xiao, and V. Sindhwani, "Demonstrating large language models on robots." in *Robotics: Science and Systems*, 2023.

[37] Z. Gao, Y. Mu, J. Qu, M. Hu, L. Guo, P. Luo, and Y. Lu, "Dag-plan: Generating directed acyclic dependency graphs for dual-arm cooperative planning," 2024. [Online]. Available: https://arxiv.org/abs/2406.09953

[38] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, p. 720–748, sep 1999. [Online]. Available: https://doi.org/10.1145/324133.324234

[39] ggerganov, "llama.cpp." [Online]. Available: https://github.com/ggerganov/llama.cpp

[40] M. AI, "mlc-llm." [Online]. Available: https://github.com/mlc-ai/mlc-llm

[41] Y. Song, Z. Mi, H. Xie, and H. Chen, "Powerinfer: Fast large language model serving with a consumer-grade gpu," 2023.

[42] Z. Xue, Y. Song, Z. Mi, L. Chen, Y. Xia, and H. Chen, "Powerinfer-2: Fast large language model inference on a smartphone," 2024.

[43] K. Alizadeh, I. Mirzadeh, D. Belenko, K. Khatamifard, M. Cho, C. C. D. Mundo, M. Rastegari, and M. Farajtabar, "Llm in a flash: Efficient large language model inference with limited memory," 2024.

[44] B. D. Lund, T. Wang, N. R. Mannuru, B. Nie, S. Shimray, and Z. Wang, "Chatgpt and a new academic reality: Artificial intelligence-written research papers and the ethics of the large language models in scholarly publishing," *Journal of the Association for Information Science and Technology*, vol. 74, no. 5, pp. 570–581, 2023.

[45] Z. Yang, P. Qi, S. Zhang, Y. Bengio, W. W. Cohen, R. Salakhutdinov, and C. D. Manning, "Hotpotqa: A dataset for diverse, explainable multi-hop question answering," *arXiv preprint arXiv:1809.09600*, 2018.

[46] Python, "Developing with asyncio," 2024. [Online]. Available: https://docs.python.org/3/library/asyncio-dev.html

[47] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 819–831. [Online]. Available: https://doi.org/10.1145/3650212.3680323

[48] X. Ou, C. Li, Y. Jiang, and C. Xu, "The mutators reloaded: Fuzzing compilers with large language model generated mutation operators," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2024, San Diego, USA, 2024. [Online]. Available: https://connglli.github.io/pdfs/metamut_asplos24.pdf

[49] A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso *et al.*, "Beyond the imitation game: Quantifying and extrapolating the capabilities of language models," *arXiv preprint arXiv:2206.04615*, 2022.

[50] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[51] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Advances in Neural Information Processing Systems*, vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/250cf8b51c773f3f8dc8b4be867a9a02-Paper.pdf

[52] MeetKai, "functionary," 2023. [Online]. Available: https://github.com/MeetKai/functionary

[53] F. Yan, H. Mao, C. C.-J. Ji, T. Zhang, S. G. Patil, I. Stoica, and J. E. Gonzalez, "Berkeley function calling leaderboard," 2024.

[54] THUDM, "Glm-4," 2024. [Online]. Available: https://github.com/THUDM/GLM-4

[55] X. Liu, J. Wang, J. Sun, X. Yuan, G. Dong, P. Di, W. Wang, and D. Wang, "Prompting frameworks for large language models: A survey," *arXiv preprint arXiv:2311.12785*, 2023.

[56] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[57] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," 2023.

[58] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, and T. Hoefler, "Graph of thoughts: Solving elaborate problems with large language models," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 16, p. 17682–17690, Mar. 2024. [Online]. Available: http://dx.doi.org/10.1609/aaai.v38i16.29720

[59] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021.

[60] L. Beurer-Kellner, M. Fischer, and M. Vechev, "Prompting is programming: A query language for large language models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1946–1969, 2023.

[61] Microsoft, "Semantic-kernel," 2023. [Online]. Available: https://github.com/microsoft/semantic-kernel

[62] SuperAGI, "SuperAGI," 2023. [Online]. Available: https://superagi.com/

[63] NVIDIA, "NeMo-Guardrails," 2023. [Online]. Available: https://github.com/NVIDIA/NeMo-Guardrails

[64] Z. Zhou, X. Ning, K. Hong, T. Fu, J. Xu, S. Li, Y. Lou, L. Wang, Z. Yuan, X. Li, S. Yan, G. Dai, X.-P. Zhang, Y. Dong, and Y. Wang, "A survey on efficient inference for large language models," 2024.

[65] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis, "Efficient streaming language models with attention sinks," 2024.

[66] H. Jiang, Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu, "LLMLingua: Compressing prompts for accelerated inference of large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Dec. 2023, pp. 13 358–13 376. [Online]. Available: https://aclanthology.org/2023.emnlp-main.825

[67] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," 2024.

[68] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969. [Online]. Available: https://doi.org/10.1137/0117039

[69] J. Wang, B. Trach, M. Fu, D. Behrens, J. Schwender, Y. Liu, J. Lei, V. Vafeiadis, H. Härtig, and H. Chen, "BWoS: Formally verified block-based work stealing for parallel processing," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 833–850. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/wang-jiawei

[70] N. Bansal and M. Harchol-Balter, "Analysis of srpt scheduling: investigating unfairness," in *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 279–290. [Online]. Available: https://doi.org/10.1145/378420.378792

[71] D. Tuomenoksa and H. Siegel, "Task scheduling on the pasm parallel processing system," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 2, pp. 145–157, 1985.

[72] J. A. Stankovic, K. Ramamritham, and M. Spuri, *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. USA: Kluwer Academic Publishers, 1998.

[73] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–11.

[74] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. USA: USENIX Association, 2011, p. 2.

[75] H. Zhao, G. Qi, Q. Wang, J. Wang, P. Yang, and L. Qiao, "Energy-efficient task scheduling for heterogeneous cloud computing systems," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 952–959.