An LLM Compiler for Parallel Function Calling

Sehoon Kim [*1] **Suhong Moon** [*1] **Ryan Tabrizi** [1] **Nicholas Lee** [1] **Michael W. Mahoney** [1 2 3]
Kurt Keutzer [1] **Amir Gholami** [1 2]

Abstract

The reasoning capabilities of the recent LLMs enable them to execute external function calls to overcome their inherent limitations, such as knowledge cutoffs, poor arithmetic skills, or lack of access to private data. This development has allowed LLMs to select and coordinate multiple functions based on the context to tackle more complex problems. However, current methods for function calling often require sequential reasoning and acting for each function which can result in high latency, cost, and sometimes inaccurate behavior. To address this, we introduce `LLMCompiler`, which executes functions in parallel to efficiently orchestrate multiple function calls. Drawing inspiration from the principles of classical compilers, `LLMCompiler` enables parallel function calling with three components: (i) a Function Calling Planner, formulating execution plans for function calling; (ii) a Task Fetching Unit, dispatching function calling tasks; and (iii) an Executor, executing these tasks in parallel. `LLMCompiler` automatically generates an optimized orchestration for the function calls and can be used with both open-source and closed-source models. We have benchmarked `LLMCompiler` on a range of tasks with different patterns of function calling. We observe consistent latency speedup of up to $3.7\times$, cost savings of up to $6.7\times$, and accuracy improvement of up to $\sim 9\%$ compared to ReAct. Our code is available at https://github.com/SqueezeAILab/LLMCompiler.

1. Introduction

Recent advances in the reasoning capability of Large Language Models (LLMs) have expanded the applicability of LLMs beyond content generation to solving complex problems (Besta et al., 2023; Chen et al., 2023b; Gao et al., 2022; Kojima et al., 2023; Wang et al., 2023b; Wei et al., 2022; Yang et al., 2022; Yao et al., 2023b; Zhou et al., 2023b); and recent works have also shown how this reasoning capability can be helpful in improving accuracy for solving complex and logical tasks. The reasoning capability has also allowed function (i.e., tool) calling capability, where LLMs can invoke provided functions and use the function outputs to help complete their tasks. These functions range from a simple calculator that can invoke arithmetic operations to more complex LLM-based functions.

The ability of LLMs to integrate various tools and function calls could enable a fundamental shift in how we develop LLM-based software. However, this brings up an important challenge: *what is the most effective approach to incorporate multiple function calls?* A notable approach has been introduced in ReAct (Yao et al., 2022), where the LLM calls a function, analyzes the outcomes, and then reasons about the next action, which involves a subsequent function call. For a simple example illustrated in Fig. 1 (Left), where the LLM is asked if Scott Derrickson and Ed Wood have the same nationality, ReAct initially analyzes the query and decides to use a search tool to search for Scott Derrickson. The result of this search (i.e., observation) is then concatenated back to the original prompt for the LLM to reason about the next action, which invokes another search tool to gather information about Ed Wood.

ReAct has been a pioneering work in enabling function calling, and it has been integrated into several frameworks (Langchain; Liu, 2022). However, scaling this approach for more complex applications requires considerable optimizations. This is due to the sequential nature of ReAct, where it executes function calls and reasons about their observations one after the other. This approach, along with the agent systems that extend ReAct (Khot et al., 2023; Qin et al., 2023; Ruan et al., 2023b; Sumers et al., 2023; Yao et al., 2023b), may lead to inefficiencies in latency and cost, due to the sequential function calling and repetitive LLM invocations for each reasoning and action step. Furthermore, while dynamic reasoning about the observations has benefits in certain cases, concatenating the outcomes of intermediate function calls could disrupt the LLM's execution flow, potentially reducing accuracy (Xu et al., 2023). Common failure cases include repetitive invocation of the same func-

[*]Equal contribution [1]UC Berkeley [2]ICSI [3]LBNL. Correspondence to: Amir Gholami <amirgh@berkeley.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

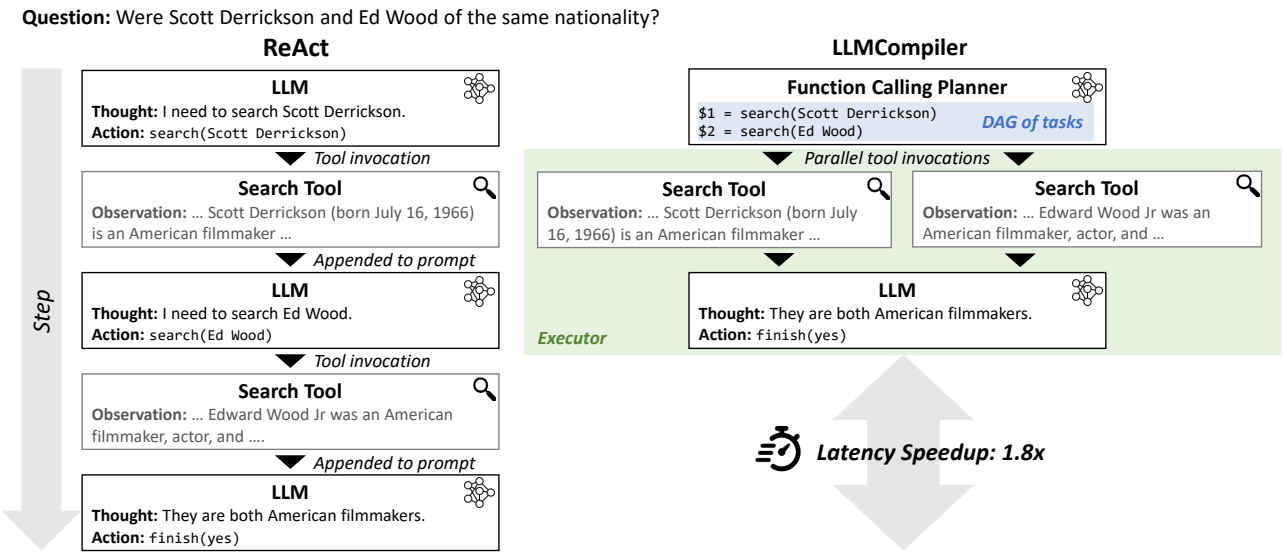

Figure 1. An illustration of the runtime dynamics of `LLMCompiler`, in comparison with ReAct (Yao et al., 2022), given a sample question from the HotpotQA benchmark (Yang et al., 2018). In `LLMCompiler` (Right), the Planner first decomposes the query into several tasks with inter-dependencies. The Executor then executes multiple tasks in parallel, respecting their dependencies. Finally, `LLMCompiler` joins all observations from the tool executions to produce the final response. In contrast, sequential tool execution of the existing frameworks like ReAct (Left) leads to longer execution latency. In this example, `LLMCompiler` attains a latency speedup of $1.8\times$ on the HotpotQA benchmark. While a 2-way parallelizable question from HotpotQA is presented here for the sake of simple visual illustration, `LLMCompiler` is capable of managing tasks with more complex dependency patterns (Fig. 2 and Sec. 5).

tion, which is also highlighted in the original paper (Yao et al., 2022), and early stopping based on the partial intermediate results, as will be further discussed in Sec. 5.1 and Appendix A.

To address this challenge, we draw inspiration from classical compilers, where optimizing instruction executions in traditional programming languages has been extensively explored. A key optimization technique in compilers involves identifying instructions that can be executed in parallel and effectively managing their dependencies. Similarly, one can envision a compiler, tailored for LLM function calling, which can efficiently orchestrate various function calls and their dependencies. This shares a similar philosophy with the recent studies that align LLMs with computer systems (Karpathy, 2023; Packer et al., 2023). To this end, we introduce `LLMCompiler`, a novel framework that enables parallel multi-tool execution of LLMs across different models and workloads. To the best of our knowledge, `LLMCompiler` is the first framework to optimize the orchestration of LLM function calling that can not only improve latency and cost, but also accuracy, by minimizing interference from the outputs of intermediate function calls. In more detail, we make the following contributions:

- We introduce `LLMCompiler`, an LLM compiler that optimizes the parallel function calling performance of LLMs. At a high level, this is achieved by introducing three key components: (i) a Function Calling Planner (Sec. 3.1) that identifies an execution flow; (ii) a Task Fetching Unit (Sec. 3.2) that dispatches the function calls in parallel; (iii) an Executor (Sec. 3.3) that executes the dispatched tasks using the associated functions.
- We evaluate `LLMCompiler` on *embarrassingly parallel* patterns using HotpotQA (Yang et al., 2018) and Movie Recommendation (Srivastava et al., 2022), where we observe $1.80\times$/$3.74\times$ speedup and $3.37\times$/$6.73\times$ cost reduction compared to ReAct (Sec. 5.1).
- To test the performance on more complex patterns, we introduce a new benchmark called ParallelQA which includes various non-trival function calling patterns. We show up to $2.27\times$ speedup, $4.65\times$ cost reduction, and 9% improved accuracy compared to ReAct (Sec. 5.2).
- We evaluate `LLMCompiler`'s capability in dynamic replanning, which is achieved through a feedback loop from the Executor back to our Function Calling Planner. For the Game of 24 (Yao et al., 2023b), which requires repeated replanning based on the intermediate results, `LLMCompiler` demonstrates a $2\times$ speedup compared to Tree-of-Thoughts (Sec. 5.3).
- We show that `LLMCompiler` can explore the interactive decision-making environment effectively and efficiently. On WebShop, `LLMCompiler` achieves up to $101.7\times$ speedup and 25.7% improved success rate compared to the baselines (Sec. 5.4).

2. Related Work

2.1. Latency Optimization in LLMs

Various studies have focused on optimizing model design (Chen et al., 2023a; Dettmers et al., 2023; Frantar & Alistarh, 2023; Frantar et al., 2022; Kim et al., 2023; 2024; Kwon et al., 2022; Leviathan et al., 2023; Lin et al., 2023) and systems (tgi; trt; Kwon et al., 2023; Yu et al., 2022) for efficient LLM inference. Optimizations at the application level, however, are less explored. This is critical from a practical point of view for situations involving black-box LLM models and services where modifications to the models and the underlying inference pipeline are highly restricted.

Skeleton-of-Thought (Ning et al., 2023) recently proposed to reduce latency through application-level parallel decoding. This method involves a two-step process of an initial skeleton generation phase, followed by parallel execution of skeleton items. However, it is primarily designed for embarrassingly parallel workloads and does not support problems that have inherently interdependent tasks, as it assumes no dependencies between skeleton tasks. This limits its applicability in complex scenarios such as coding (Austin et al., 2021; Chen et al., 2021; Hendrycks et al., 2021a; Madaan et al., 2023) or math (Hendrycks et al., 2021b;c) problems, as also stated in the paper (Ning et al., 2023). `LLMCompiler` addresses this by translating an input query into a series of tasks with inter-dependencies, thereby expanding the spectrum of problems it can handle.

Concurrently to our work, OpenAI has recently introduced a parallel function calling feature in their 1106 release, enhancing user query processing through the simultaneous generation of multiple function calls (OpenAI, 2023). Despite its potential for reducing LLM execution time, this feature has certain limitations, as it is exclusively available for OpenAI's proprietary models. However, there is a growing demand for using open-source models driven by the increasing number of open-source LLMs as well as parameter-efficient training techniques (Houlsby et al., 2019; Hu et al., 2022; Lester et al., 2021) for finetuning and customization. `LLMCompiler` enables efficient parallel function calling for open-source models, and also, as we will show later in Sec. 5, it can potentially achieve better latency and cost.

2.2. Plan and Solve Strategy

Several studies (Hao et al., 2023; Patel et al., 2022; Press et al., 2023; Wolfson et al., 2020; Zhou et al., 2023b) have explored prompting methods of breaking down complex queries into various levels of detail to solve them, thereby improving LLM's performance in reasoning tasks. Specifically, Decomposed Prompting (Khot et al., 2023) tackles complex tasks by decomposing them into simpler sub-tasks, each optimized through LLMs with dedicated prompts. Step-Back Prompting (Zheng et al., 2023) enables LLMs to abstract high-level concepts from details to enhance reasoning abilities across various tasks. Plan-and-Solve Prompting (Wang et al., 2023a) segments multi-step reasoning tasks into subtasks to minimize errors and improve task accuracy without manual prompting. However, these methods primarily focus on improving the accuracy of reasoning benchmarks. In contrast, `LLMCompiler` uses a planner to identify parallelizable patterns within queries, aiming to reduce latency while maintaining accuracy.

In addition to the aforementioned works, TPTU (Ruan et al., 2023a), HuggingGPT (Shen et al., 2023), and ViperGPT (Surís et al., 2023) have introduced end-to-end plan-and-solve frameworks. `LLMCompiler` sets itself apart by providing a general framework that enables efficient and accurate function calling in a broader range of problems. This stems from `LLMCompiler`'s capabilities in (i) planning and replanning; (ii) parallel execution; and (iii) addressing a wider range of problem domains, which will be discussed in more detail in Appendix F.

Another notable work is ReWOO (Xu et al., 2023) which employs a planner to separate the reasoning process from the execution and observation phases to decrease token usage and cost as compared to ReAct. Our approach is different from ReWOO in multiple aspects. First, `LLMCompiler` allows parallel function calling which can reduce latency as well as cost. Second, `LLMCompiler` supports dynamic replanning which is important for problems whose execution flow cannot be determined statically in the beginning (Sec. 5.3).

2.3. Tool-Augmented LLMs

The enhanced reasoning capability of LLMs has enabled them to invoke user-provided functions and use their outputs to effectively complete tasks. Detailed exploration of this subject is provided in the Appendix C.1.

3. Methodology

To illustrate the components of `LLMCompiler`, we use a simple 2-way parallel example in Fig. 2. To answer "How much does Microsoft's market cap need to increase to exceed Apple's market cap?," the LLM first needs to conduct web searches for both companies' market caps, followed by a division operation. While the existing frameworks, including ReAct, perform these tasks sequentially, it is evident that they can be executed in parallel. The key question is how to automatically determine which tasks are parallelizable and which are interdependent, so we can orchestrate the execution of the different tasks accordingly. `LLMCompiler` accomplishes this through a system that consists of the follow-

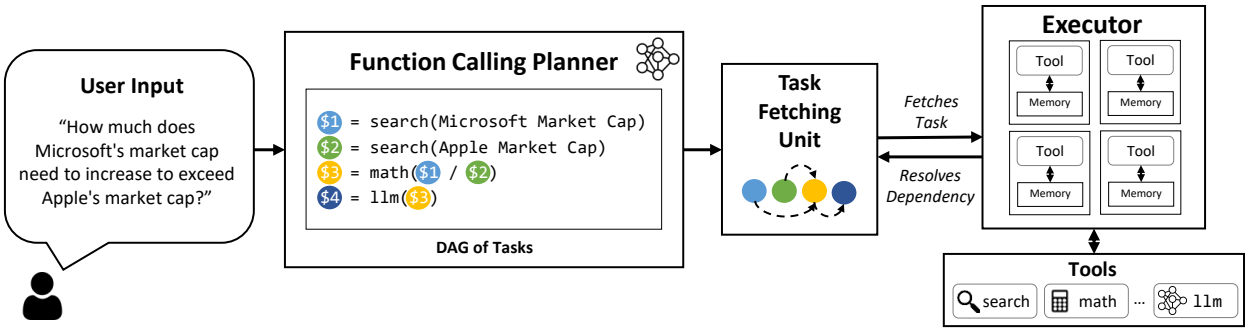

Figure 2. Overview of the `LLMCompiler` framework. The Function Calling Planner generates a DAG of tasks with their inter-dependencies. These tasks are then dispatched by the Task Fetching Unit to the Executor in parallel based on their dependencies. In this example, Task `$1` and `$2` are fetched together for parallel execution of two independent search tasks. After each task is performed, the results are forwarded back to the Task Fetching Unit to unblock the dependent tasks after replacing their placeholder variables (e.g., the variable `$1` and `$2` in Task `$3`) with actual values. Once all tasks have been executed, the final answer is delivered to the user.

ing three components: a Function Calling Planner (Sec. 3.1) that generates a sequence of tasks and their dependencies; a Task Fetching Unit (Sec. 3.2) that replaces arguments based on intermediate results and fetches the tasks; and an Executor (Sec. 3.3) that executes the tasks with associated tools. To use `LLMCompiler`, users are only required to provide tool definitions, and optional in-context examples for the Planner, as will be further discussed in Sec. 4.1.

3.1. Function Calling Planner

The Function Calling Planner is responsible for generating a sequence of tasks to be executed along with any dependency among them. For instance, Tasks `$1` and `$2` in Fig. 2 are two independent searches that can be performed in parallel. However, Task `$3` has a dependency on the outcomes of the first and second searches. Therefore, the Planner's role is to automatically identify the necessary tasks, their input arguments, as well as their inter-dependencies using the sophisticated reasoning capability of LLMs, essentially forming a directed acyclic graph of task dependencies. If a task is dependent on a preceding task, it incorporates a placeholder variable, such as `$1` in Task 3 of Fig. 2, which will later be substituted with the actual output from the preceding task (Sec. 3.2).

The Planner in `LLMCompiler` leverages LLMs' reasoning capability to decompose tasks from natural language inputs. To achieve this, the Planner LLM incorporates a pre-defined prompt that guides it on how to create dependency graphs and to ensure correct syntax (see Appendix H for details). Besides this, users also need to supply tool definitions and optional in-context examples for the Planner. These examples provide detailed demonstrations of task decomposition specific to a problem, helping the Planner to better understand the rules. Further details on user-supplied information for `LLMCompiler` are elaborated in Sec. 4.1. In Sec. 4.2, we introduce an additional optimization for the Planner that streams tasks as soon as they are created, instead of waiting to complete the entire planning process.

3.2. Task Fetching Unit

The Task Fetching Unit, inspired by the instruction fetching units in modern computer architectures, fetches tasks to the Executor as soon as they are ready for (parallel) execution based on a greedy policy. Another key functionality is to replace variables with the actual outputs from preceding tasks, which were initially set as placeholders by the Planner. For the example in Fig. 2, the variable `$1` and `$2` in Task `$3` would be replaced with the actual market cap of Microsoft and Apple. This can be implemented with a simple fetching and queuing mechanism without a dedicated LLM.

3.3. Executor

The Executor asynchronously executes tasks fetched from the Task Fetching Unit. As the Task Fetching Unit guarantees that all the tasks dispatched to the Executor are independent, it can simply execute them concurrently. The Executor is equipped with user-provided tools, and it delegates the task to the associated tool. These tools can be simple functions like a calculator, Wikipedia search, or API calls, or they can even be LLM agents that are tailored for a specific task. As depicted in the Executor block of Fig. 2, each task has dedicated memory to store its intermediate outcomes, similar to what typical sequential frameworks do when aggregating observations as a single prompt (Yao et al., 2022). Upon completion of the task, the final results are forwarded as input to the tasks dependent on them.

3.4. Dynamic Replanning

In various applications, the execution graph may need to adapt based on intermediate results that are a priori unknown. An analogy in programming is branching, where the path

of execution is determined only during runtime, depending on which branch conditions are satisfied. Such dynamic execution patterns can also appear with LLM function calling. For simple branching (e.g., if-else statements) one could statically compile the execution flow and choose the right dynamically based on the intermediate results. However, for more complex branching it may be better to do a recompilation or replanning based on the intermediate results.

When replanning, the intermediate results are sent back from the Executor to the Function Calling Planner which then generates a new set of tasks with their associated dependencies. These tasks are then sent to the Task Fetching Unit and subsequently to the Executor. This cycle continues until the desired final result is achieved and can be delivered to the user. We show an example use case of this in Sec. 5.3 for solving the Game of 24 using the Tree-of-Thoughts approach.

4. `LLMCompiler` Details

4.1. User-Supplied Information

`LLMCompiler` requires two inputs from the user:

1. **Tool Definitions**: Users need to specify the tools that LLMs can use, including their descriptions and argument specifications. This is essentially the same requirement as other frameworks like ReAct and OpenAI function calling.
2. **In-context Examples for the Planner**: Optionally, users can provide `LLMCompiler` with examples of how the Planner should behave. For instance, in the case of Fig. 2, users may provide examples illustrating expected inter-task dependencies for certain queries. These examples can assist the Planner LLM understand how to use various tools and generate the appropriate dependency graph for incoming inputs in the correct format. In Appendix G, we include the examples that we used in our evaluations.

4.2. Streamed Planner

The Planner may incur a non-trivial overhead for user queries that involve a lot of tasks as it blocks the Task Fetching Unit and the Executor, which must wait for the Planner output before initiating their processes. However, analogous to instruction pipelining in modern computer systems, this can be mitigated by enabling the Planner to asynchronously stream the dependency graph, thereby allowing each task to be immediately processed by the Executor as soon as its dependencies are all resolved. In Table C.1, we present a latency comparison of `LLMCompiler` with and without the streaming mechanism across different benchmarks. The results demonstrate consistent latency improvements with streaming. Particularly, in the ParallelQA benchmark, the streaming feature leads to a latency gain of up to 1.3$\times$. This is attributed to the math tool's longer execution time for ParallelQA, which can effectively hide the Planner's latency in generating subsequent tasks, unlike the shorter execution times of the `search` tool used in HotpotQA and Movie Recommendation.

5. Results

In this section, we evaluate `LLMCompiler` using a variety of models and problem types. We use both the proprietary GPT models and the open-source LLaMA-2 model, with the latter demonstrating `LLMCompiler`'s capability in enabling parallel function calling in open-source models. Furthermore, there are various types of parallel function calling patterns that can be addressed with LLMs. This ranges from embarrassingly parallel patterns, where all tasks can be executed in parallel without any dependencies between them, to more complex dependency patterns, as illustrated in Fig. 3. Importantly, we also assess `LLMCompiler` on the Game of 24 benchmark, which involves dynamic replanning based on intermediate results, highlighting its adaptability to dynamic dependency graphs. Finally, we apply `LLMCompiler` to the WebShop benchmark to showcase its potential in decision-making tasks. Overall, we start presenting results for simple execution patterns, and then we move to more complex ones.

5.1. Embarrassingly Parallel Function Calling

The simplest scenario involves an LLM using a tool repeatedly for independent tasks such as conducting parallel searches or analyses to gather information on different topics, like the pattern depicted in Fig. 3 (a). While these tasks are independent of each other and can be executed in parallel, ReAct, along with other LLM solutions as they stand, would need to run sequentially. This leads to increased latency and token consumption due to its frequent LLM invocations for each tool usage, as also illustrated in Fig. 1. In this section, we demonstrate how `LLMCompiler` can identify parallelizable patterns and execute independent tasks concurrently to resolve this issue. To do so, we use the following two benchmarks:

- **HotpotQA:** A dataset that evaluates multi-hop reasoning (Yang et al., 2018). We only use the comparison dev set. This contains 1.5k questions comparing two different entities, thus exhibiting a 2-way embarrassingly parallel execution pattern. An example question is shown in Fig. 1.

- **Movie Recommendation:** A dataset with 500 examples that asks to identify the most similar movie out of four options to another set of four movies, exhibiting an 8-way embarrassingly parallel pattern (Srivastava et al., 2022).

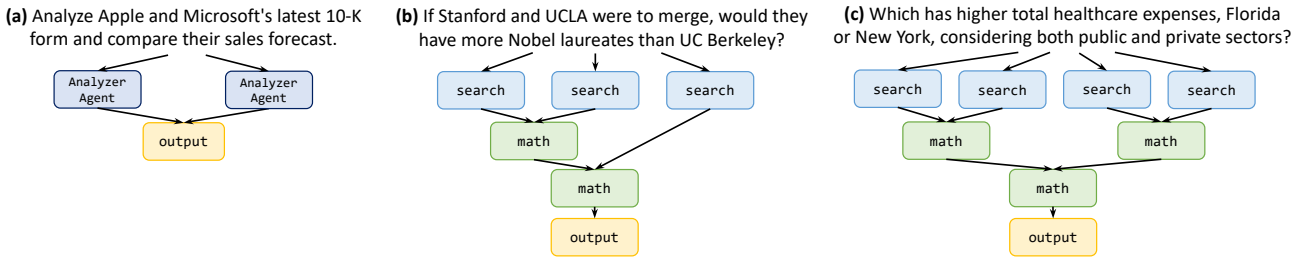

Figure 3. Examples of questions with different function calling patterns and their dependency graphs. HotpotQA and Movie Recommendation datasets exhibit pattern (a), and ParallelQA dataset exhibits patterns (b) and (c), among other patterns. In (a), we need to analyze each company's latest 10-K. In (b), we need three searches for each school, followed by one addition and one comparison operation. In (c), we need to search for each state's annual healthcare spending in each sector, sum each state's spending, and then perform a comparison.

Table 1. Accuracy and latency comparison of `LLMCompiler` compared to the baseline on different benchmarks, including HotpotQA, Movie Recommendation, our custom dataset named ParallelQA, and the Game of 24. For HotpotQA and Movie Recommendation, we frequently observe looping and early stopping (Sec. 5.1). To minimize these behaviors as much as possible, we incorporated ReAct-specific prompting which we denote as ReAct†. ReAct (without †) indicates the original results without this prompting. We do not include the latency for the original ReAct since looping and early stopping make precise latency measurement difficult.

Benchmark	Method	GPT (Closed-source)			LLaMA-2 70B (Open-source)		
		Accuracy (%)	Latency (s)	Speedup	Accuracy (%)	Latency (s)	Speedup
HotpotQA	ReAct	61.52	-	-	54.74	-	-
	ReAct†	62.47	7.12	1.00×	54.40	13.44	1.00×
	OAI Parallel Function	62.05	4.42	1.61×	-	-	-
	`LLMCompiler`	62.00	**3.95**	**1.80×**	57.83	**9.58**	**1.40×**
Movie Rec.	ReAct	68.60	-	-	70.00	-	-
	ReAct†	72.47	20.47	1.00×	70.60	33.37	1.00×
	OAI Parallel Function	77.00	7.42	2.76×	-	-	-
	`LLMCompiler`	77.13	**5.47**	**3.74×**	77.80	**11.83**	**2.82×**
ParallelQA	ReAct	89.09	35.90	1.00×	59.59	15.47	1.00×
	OAI Parallel Function	87.32	19.29	1.86×	-	-	-
	`LLMCompiler`	89.38	**16.69**	**2.15×**	68.14	**26.20**	**2.27×**
Game of 24	Tree-of-Thoughts	74.00	241.2	1.00×	30.00	952.06	1.00×
	`LLMCompiler`	75.33	**83.6**	**2.89×**	32.00	**456.02**	**2.09×**

Table 2. Input and output token consumption as well as the estimated cost on HotpotQA, Movie Recommendation, and our custom dataset named ParallelQA. The cost is computed based on the pricing table of the GPT models used for each benchmark.

Benchmark	Method	Tokens In.	Tokens Out.	Cost ($/1k)	Cost Red.
HotpotQA	ReAct	2900	120	5.00	1.00×
	OAI Para. Func.	2500	63	2.66	1.87×
	`LLMCompiler`	1300	80	**1.47**	**3.37×**
Movie Rec.	ReAct	20000	230	20.46	1.00×
	OAI Para. Func.	5800	160	6.14	3.33×
	`LLMCompiler`	2800	115	**3.04**	**6.73×**
ParallelQA	ReAct	46000	470	480	1.00×
	OAI Para. Func.	25000	370	260	1.81×
	`LLMCompiler`	9200	340	**103**	**4.65×**

Experimental Setups. As a baseline method, we compare `LLMCompiler` with ReAct. We follow the ReAct setup (Yao et al., 2022) using the same Wikipedia search tool that LLMs can use to search for information. We did not include the lookup tool since it is not relevant to our problem setting. We have optimized the prompt and in-context examples for both ReAct and `LLMCompiler` to the best of our abilities. For all experiments across these datasets, we use gpt-3.5-turbo (1106 release). For the experiments using GPT, we additionally report the results using OpenAI's parallel function calling capability, which was announced concurrently with our work. We also show how `LLMCompiler` can be effectively combined with the open-source LLaMA-2 70B model to provide the model with parallel function calling capabilities. For all experiments, we have measured accuracy, end-to-end latency, as well as input and output token usage. See Appendix D for details on experimental setups.

Accuracy and Latency. We report the accuracy, end-to-end latency, and relative speed-up of `LLMCompiler` compared to ReAct in Tab. 1. First, we observe that ReAct consistently achieves lower accuracy compared to OpenAI parallel function calling and `LLMCompiler`. We identify two main failure modes in ReAct: (1) the tendency for redundant generation of prior function calls, a point also noted in the original ReAct paper (Yao et al., 2022); and (2) premature early stopping based on the incomplete intermediate results. In Appendix A, we offer a detailed analysis demonstrating how these two prevalent failure cases significantly

hurt ReAct's accuracy, and how they can be resolved with LLMCompiler, leading to an accuracy enhancement of up to 7 – 8%. Furthermore, we have conducted interventional experiments in which we incorporated ReAct-specific prompts to avoid repetitive function calls and early stopping. ReAct† in Tab. 1 refers to ReAct *with* this ReAct-specific prompt. The ReAct-specific prompt yields a general accuracy improvement with ReAct† as compared to the original ReAct. Nevertheless, LLMCompiler still demonstrates on-par and better accuracy than ReAct†, as such prompting does not serve as a perfect solution to completely avoiding the erroneous behavior of ReAct.

Additionally, when compared to ReAct†, LLMCompiler demonstrates a noticeable speedup of $1.80\times$ and $1.40\times$ on HotpotQA with GPT and LLaMA, respectively. Similarly, LLMCompiler demonstrates $3.74\times$ and $2.82\times$ speedup on Movie Recommendation with each model. Note that we benchmark the latency of LLMCompiler against that of ReAct† since the repeating and early stopping behavior of the original ReAct as discussed above makes its latency unpredictable and unsuitable for a fair comparison. LLMCompiler demonstrates a speedup of up to 35% compared to OpenAI parallel function calling whose latency gain over ReAct is $1.61\times$ and $2.76\times$ on each benchmark.[1]

Costs. Another important consideration of using LLMs is cost, which depends on the input and output token usage. The costs for GPT experiments are provided in Tab. 2. LLMCompiler is more cost-efficient than ReAct for cost, as it involves less frequent LLM invocations. Interestingly, LLMCompiler also outperforms the recent OpenAI parallel function calling in cost efficiency. This is because LLMCompiler's planning phase is more prompt length efficient than that of OpenAI parallel function calling since our Planner's in-context examples are rather short and only include plans, not observations (see Appendix H).

5.2. Parallel Function Calling with Dependencies

The cases considered above are rather simple, as only one tool is used and all tasks can be executed independently of one another. However, similar to code execution in traditional code blocks, we may encounter function calling scenarios that involve more complex dependencies. To systematically evaluate the capability to plan out function calling in scenarios that involve complex task dependencies, we have designed a custom benchmark called ParallelQA. This benchmark is designed to incorporate non-trivial function calling patterns, including three different types of patterns in Fig. 3 (b) and (c). Inspired by the IfQA benchmark (Yu et al., 2023), ParallelQA contains 113 examples that involve mathematical questions on factual attributes of various entities. In particular, completing the task requires using two tools (i.e., search and math tools), with the second tool's argument depending on the result of the first tool's output. We have meticulously included questions that are answerable only with information from Wikipedia's first paragraph, effectively factoring out the failure cases due to unsuccessful searches. See Appendix I for more details in ParallelQA.

Experimental Setups. Similar to Sec. 5.1, we use ReAct (Yao et al., 2022) as the main baseline. Here, both ReAct and LLMCompiler are equipped with two tools: (1) the search tool, identical to the one mentioned in Sec.5.1; and (2) the math tool, which solves mathematical problems. The math tool is inspired by the Langchain (Langchain)'s LLMMathChain, which uses an LLM as an agent that interprets input queries and invokes the numexpr function with the appropriate formula. This enables the math chain to address a broad spectrum of math problems that are written both in mathematical and verbal form. See Appendix D for more details on experimental setups.

Accuracy and Latency. As shown in the ParallelQA row of Tab. 1, LLMCompiler arrives at the final answer with an average speedup of $2.15\times$ with gpt-4-turbo and $2.27\times$ with LLaMA-2 70B, by avoiding sequential execution of the dependency graphs. Beyond the latency speedup, we observe higher accuracy of LLMCompiler with the LLaMA-2 model as compared to that of ReAct, due to the reasons discussed in Sec. 5.1. Particularly in the LLaMA-2 experiment, where LLMCompiler achieves around a 9% increase in accuracy, we note that $\sim$20% of the examples experienced repetitive function calls with ReAct, aligning with our observations from the accuracy analysis detailed in Appendix A. Additionally, a comprehensive analysis of LLMCompiler's failure cases is provided in Appendix B, where we note minimal Planner failures, highlighting LLMCompiler's effectiveness in breaking down problems into complex multi-task dependencies.

Cost. Similar to Sec. 5.1, LLMCompiler demonstrates substantial cost reductions of $4.65\times$ and $2.57\times$ compared to ReAct and OpenAI's parallel function calling, respectively, as indicated in Tab. 2. This efficiency stems from LLMCompiler's reduced frequency of LLM invocations, which is also the case with OpenAI's parallel function calling, which is limited to planning out immediate parallelizable tasks, not the entire dependency graph. For example, in Fig. 3 (c), OpenAI's method would necessitate three distinct LLM calls for initial search tasks, following math tasks, and the final math task. In contrast, LLMCompiler achieves this with a single LLM call, planning all tasks concurrently.

[1] Unfortunately, we are unable to conclude why this is the case, as OpenAI has not publicly disclosed any details about their function calling mechanism. One speculation is that there might be additional overheads to validate the function and argument names and to convert them into a system prompt. Nevertheless, we have seen a consistent trend with multiple runs over several days.

5.3. Parallel Function Calling with Replanning

In the previous sections, we have discussed cases in which dependency graphs can be determined statically. However, there are cases where dependency graphs need to be constructed dynamically depending on intermediate observations. Here, we consider one such dynamic approach in the context of the Game of 24 with the Tree-of-Thoughts (ToT) strategy proposed in (Yao et al., 2023b). The Game of 24 is a task to generate 24 using a set of four numbers and basic arithmetic operations. For example, from the numbers 2, 4, 4, and 7, a solution could be $4 \times (7 - 4) \times 2 = 24$. ToT approaches this task through two iterative LLM processes: (i) the thought proposer generates candidate partial solutions by selecting two numbers and applying an operation (e.g. 2, 3, 7 from 2, 4, 4, 7 by calculating 7 - 4); (ii) the state evaluator assesses the potential of each candidate. Only the promising candidates are then processed in subsequent iterations of the thought proposer and state evaluator until 24 is reached. Details about the Game of 24 benchmark and the ToT strategy can be found in Appendix J.

While ToT achieves significant improvement at solving the Game of 24, its sequential, breadth-first search approach through the state tree can be time-consuming. `LLMCompiler` offers a faster alternative by enabling parallel execution of the thought proposer and the subsequent feasibility evaluator, akin to a parallel beam search method.

Experimental Setups. Although `LLMCompiler` offers latency advantages, solving this problem with a single static graph is not feasible, as the Planner cannot plan out the thought proposing stage before identifying the selected candidates from the state evaluator of the previous iteration. Consequently, the Planner is limited to planning only within one iteration at a time. To address this, we resort to `LLMCompiler`'s replanning capability. In particular, `LLMCompiler` is equipped with three tools: `thought_proposer` and `state_evaluator`, which are both LLMs adapted from the original ToT framework, and `top_k_select`, which chooses the top k candidates from the `thought_proposer` based on the `state_evaluator`'s assessment. After all these tools are executed, `LLMCompiler` can decide to "replan" if no proposal reaches 24, triggering the Planner to devise new plans using the shortlisted states from `top_k_select` of the previous iteration. In this way, `LLMCompiler` can dynamically regenerate plans of each iteration, being able to tackle highly complex tasks that require iterative replanning based on the outcomes of previous plans.

To evaluate `LLMCompiler`'s performance on the Game of 24, we use 100 different instances of the game. For each problem, we consider the output as successful if its operations are valid and yield 24 while also using the provided numbers exactly once each. Further details on experiment setups are outlined in Appendix D.

Success Rate and Latency. In the last two rows of Tab. 1, we explore the latency and success rate of `LLMCompiler` in comparison to the baseline described in (Yao et al., 2023b) on the Game of 24 benchmark. With the gpt-4 model, `LLMCompiler` demonstrates a $2.89\times$ enhancement in latency while slightly improving the success rate compared to the baseline. Similarly, when applied with the LLaMA-2 model, `LLMCompiler` shows a $2.01\times$ improvement in latency, again without compromising on success rate. These results demonstrate not only a significant latency reduction without quality degradation, but also the replanning capability of `LLMCompiler` for solving complex problems.

5.4. Application: `LLMCompiler` in Interactive Decision Making Tasks

In this section, we demonstrate that `LLMCompiler` can explore language-based interactive environments effectively by benchmarking `LLMCompiler` on WebShop (Yao et al., 2023a). As highlighted in (Shinn et al., 2023; Yao et al., 2022; 2023a), WebShop exhibits considerable diversity, which requires extensive exploration to purchase the most appropriate item. While recent work feature advanced exploration strategies and show promising results (Ma et al., 2023; Zhou et al., 2023a), their approaches are largely based on a sequential and extensive tree search that incurs significant latency penalties. Here, `LLMCompiler` showcases an exploration strategy that is both effective and efficient with the use of parallel function calling. Our method enables broader exploration of items in the environment, which improves success rate compared to ReAct. At the same time, this exploration can be parallelized, yielding up to $101.7\times$ speedup against baselines that perform sequential exploration.

Experimental Setups. We evaluate `LLMCompiler` against three baselines on this benchmark, ReAct (Yao et al., 2022), LATS (Zhou et al., 2023a), and LASER (Ma et al., 2023), using 500 WebShop instructions. The evaluation metrics are success rate, average score, and latency. More details of the WebShop environment and the baseline methods are provided in Appendix K. For this experiment, `LLMCompiler` is equipped with two tools: `search` and `explore`. The `search` function triggers the model to generate and dispatch a query that returns a list of typically ten items from the Webshop environment. The `explore` function then clicks through links for each of the found items and retrieves information about options, prices, attributes, and features that are available. Finally, based on the gathered information, `LLMCompiler` decides on the item that best matches the input instruction for purchasing. Further details on experiments can be found in Appendix D.

Table 3. Performance and Latency Analysis for WebShop. We evaluate `LLMCompiler` with two models: gpt-4 and gpt-3.5-turbo and compare `LLMCompiler` against three baselines: ReAct, LATS, and LASER. We report success rate and average score in percentage. We reproduce the success rate and average score for ReAct, while those for LATS and LASER are from their papers. N denotes the number of examples used for evaluation.

Model	Method	Succ. Rate	Score	Latency (s)	N
gpt-3.5-turbo	ReAct	19.8	54.2	5.98	500
	LATS	38.0	75.9	1066	50
	`LLMCompiler`	44.0	72.8	10.72	50
	`LLMCompiler`	48.2	74.2	10.48	500
gpt-4-0613	ReAct	35.2	58.8	19.90	500
	LASER	50.0	75.6	72.16	500
	`LLMCompiler`	55.6	77.1	26.73	500

Performance and Latency. Our approach significantly outperforms all baseline models as shown in Table 3. When using gpt-3.5-turbo, `LLMCompiler` achieves a 28.4% and 6% improvement in success rate against ReAct and LATS; with gpt-4, our method improves upon ReAct and LASER by 20.4% and 5.6%, respectively. In terms of latency, `LLMCompiler` exhibits a $101.7\times$ and $2.69\times$ speedup against LATS and LASER. While we note that `LLMCompiler` execution is slightly slower than ReAct on this benchmark, mainly due to the Planner overhead, we also highlight that the gains in success rate far outweigh the minor latency penalty.

We further delve into why `LLMCompiler` attains such an improved success rate and score compared to ReAct. Based on our observations, we discover that the ReAct agent tends to commit to a decision with imperfect information, a scenario that can arise when the agent has not gathered sufficient details about the features and options available for items. This observation was also noted in (Shinn et al., 2023) – without exploring more items in the environment, the agent struggles to differentiate between seemingly similar choices, ultimately failing to make the correct decision. In contrast, `LLMCompiler` undergoes further exploration by visiting all ten items found by `search` and retrieving relevant information about each item. We find that employing an effective search strategy is critical to decision-making tasks such as the WebShop benchmark.

The relatively high performance of LATS can also be explained in terms of its exploration scheme. In this framework, the agent executes a brute-force search through the state and action space of Webshop, exploring as many as 30 trajectories before making the final purchase. While this approach provides richer information for decision-making, the end-to-end execution becomes prohibitively slow.

We report that our method, `LLMCompiler`, outperforms LASER by an average score of 1.5. When compared to LATS, this score is within the standard deviation range of our method. The average score for `LLMCompiler`, along with its standard deviation, is 72.8 ± 4.01 for gpt-3.5-turbo. Further note that while the performance differences are marginal, our method exhibits significant execution speedup, $101.7\times$ over LATS and $2.69\times$ over LASER.

6. Conclusions

Existing methods for invoking multiple functions with LLMs resort to sequential and dynamic reasoning. As a result, they suffer from inefficiencies in latency, cost, and accuracy. As a solution, we introduced `LLMCompiler`, a compiler-inspired framework that enables efficient parallel function calling across various LLMs, including open-source models like LLaMA-2 and OpenAI's GPT series. By decomposing user inputs into tasks with defined inter-dependencies and executing these tasks concurrently through its Planner, Task Fetching Unit, and Executor components, `LLMCompiler` demonstrates substantial improvements in latency (up to $3.7\times$), cost efficiency (up to $6.7\times$), and accuracy (up to $\sim$9%), even outperforming OpenAI's parallel function calling feature in latency gains. We look forward to future work building upon `LLMCompiler` that will improve both the capabilities and efficiencies of LLMs in executing complex, large-scale tasks, thus transforming the future development of LLM-based applications.

Impact Statement

This paper presents research towards advancing the field of Machine Learning. While there are many potential societal consequences of our work, we do not find any one to be particularly noteworthy.

Acknowledgements

We appreciate the valuable feedback from Minwoo Kang. We acknowledge gracious support from Furiosa team. We also appreciate the support from Microsoft through their Accelerating Foundation Model Research, including great support from Sean Kuno. Furthermore, we appreciate support from Google Cloud, the Google TRC team, and specifically Jonathan Caton, and Prof. David Patterson. Prof. Keutzer's lab is sponsored by the Intel corporation, Intel One-API, Intel VLAB team, the Intel One-API center of excellence, as well as funding through BDD and BAIR. We also appreciate support from Samsung including Dongkyun Kim, and David Thorsley. We appreciate the great support from Ellick Chan, Saurabh Tangri, Andres Rodriguez, and Kittur Ganesh. Sehoon Kim and Suhong Moon would like to acknowledge the support from the Korea Foundation for Advanced Studies. Amir Gholami was supported through funding from Samsung SAIT. Michael W. Mahoney would also like to acknowledge a J. P. Morgan Chase Faculty Research Award as well as the DOE, NSF, and IARPA. Our conclusions do not necessarily reflect the position or the

policy of our sponsors, and no official endorsement should be inferred.

References

https://huggingface.co/text-generation-inference.

https://github.com/nvidia/tensorrt-llm.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models, 2021.

Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Gianinazzi, L., Gajda, J., Lehmann, T., Podstawski, M., Niewiadomski, H., Nyczyk, P., and Hoefler, T. Graph of thoughts: Solving elaborate problems with large language models, 2023.

Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., and Jumper, J. Accelerating large language model decoding with speculative sampling, 2023a.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. 2021.

Chen, W., Ma, X., Wang, X., and Cohen, W. W. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2023b.

Dettmers, T., Svirschevski, R., Egiazarian, V., Kuznedelev, D., Frantar, E., Ashkboos, S., Borzunov, A., Hoefler, T., and Alistarh, D. Spqr: A sparse-quantized representation for near-lossless llm weight compression, 2023.

Frantar, E. and Alistarh, D. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. GPTQ: Accurate post-training compression for generative pretrained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

Hao, S., Gu, Y., Ma, H., Hong, J. J., Wang, Z., Wang, D. Z., and Hu, Z. Reasoning with language model is planning with world model, 2023.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps. *NeurIPS*, 2021a.

Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021b.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021c.

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pp. 2790–2799. PMLR, 2019.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Karpathy, A. Intro to large language models, 2023.

Khot, T., Trivedi, H., Finlayson, M., Fu, Y., Richardson, K., Clark, P., and Sabharwal, A. Decomposed prompting: A modular approach for solving complex tasks. In *The Eleventh International Conference on Learning Representations*, 2023.

Kim, S., Hooper, C., Gholami, A., Dong, Z., Li, X., Shen, S., Mahoney, M. W., and Keutzer, K. Squeezellm: Dense-and-sparse quantization, 2023.

Kim, S., Mangalam, K., Moon, S., Malik, J., Mahoney, M. W., Gholami, A., and Keutzer, K. Speculative decoding with big little decoder, 2024.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners, 2023.

Kwon, W., Kim, S., Mahoney, M. W., Hassoun, J., Keutzer, K., and Gholami, A. A fast post-training pruning framework for transformers, 2022.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Langchain. https://github.com/langchain-ai/langchain.

Lester, B., Al-Rfou, R., and Constant, N. The power of scale for parameter-efficient prompt tuning, 2021.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding, 2023.

Liang, Y., Wu, C., Song, T., Wu, W., Xia, Y., Liu, Y., Ou, Y., Lu, S., Ji, L., Mao, S., Wang, Y., Shou, L., Gong, M., and Duan, N. Taskmatrix.ai: Completing tasks by connecting foundation models with millions of apis, 2023.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., Gan, C., and Han, S. Awq: Activation-aware weight quantization for llm compression and acceleration, 2023.

Liu, J. LlamaIndex, 11 2022. URL https://github.com/jerryjliu/llama_index.

Ma, K., Zhang, H., Wang, H., Pan, X., and Yu, D. Laser: Llm agent with state-space exploration for web navigation, 2023.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback, 2023.

Ning, X., Lin, Z., Zhou, Z., Wang, Z., Yang, H., and Wang, Y. Skeleton-of-thought: Large language models can do parallel decoding, 2023.

OpenAI. Gpt-4 technical report, 2023.

OpenAI. New models and developer products announced at devday, 2023.

Packer, C., Fang, V., Patil, S. G., Lin, K., Wooders, S., and Gonzalez, J. E. Memgpt: Towards llms as operating systems, 2023.

Patel, P., Mishra, S., Parmar, M., and Baral, C. Is a question decomposition unit all we need? 2022.

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

Press, O., Zhang, M., Min, S., Schmidt, L., Smith, N. A., and Lewis, M. Measuring and narrowing the compositionality gap in language models, 2023.

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

Ruan, J., Chen, Y., Zhang, B., Xu, Z., Bao, T., Du, G., Shi, S., Mao, H., Zeng, X., and Zhao, R. Tptu: Task planning and tool usage of large language model-based ai agents. *arXiv preprint arXiv:2308.03427*, 2023a.

Ruan, Y., Dong, H., Wang, A., Pitis, S., Zhou, Y., Ba, J., Dubois, Y., Maddison, C. J., and Hashimoto, T. Identifying the risks of lm agents with an lm-emulated sandbox. *arXiv preprint arXiv:2309.15817*, 2023b.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

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, 2023.

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning, 2023.

Song, Y., Xiong, W., Zhu, D., Wu, W., Qian, H., Song, M., Huang, H., Li, C., Wang, K., Yao, R., Tian, Y., and Li, S. Restgpt: Connecting large language models with real-world restful apis, 2023.

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

Sumers, T. R., Yao, S., Narasimhan, K., and Griffiths, T. L. Cognitive architectures for language agents, 2023.

Surís, D., Menon, S., and Vondrick, C. Vipergpt: Visual inference via python execution for reasoning, 2023.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur,

M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023.

Wang, L., Xu, W., Lan, Y., Hu, Z., Lan, Y., Lee, R. K.-W., and Lim, E.-P. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023a.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models, 2023b.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. volume 35, pp. 24824–24837, 2022.

Wolfson, T., Geva, M., Gupta, A., Gardner, M., Goldberg, Y., Deutch, D., and Berant, J. Break it down: A question understanding benchmark. *Transactions of the Association for Computational Linguistics*, 2020.

Xu, B., Peng, Z., Lei, B., Mukherjee, S., Liu, Y., and Xu, D. Rewoo: Decoupling reasoning from observations for efficient augmented language models, 2023.

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

Yang, Z., Dong, L., Du, X., Cheng, H., Cambria, E., Liu, X., Gao, J., and Wei, F. Language models as inductive reasoners, 2022.

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

Yao, S., Chen, H., Yang, J., and Narasimhan, K. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023a.

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

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Yu, W., Jiang, M., Clark, P., and Sabharwal, A. Ifqa: A dataset for open-domain question answering under counterfactual presuppositions, 2023.

Zheng, H. S., Mishra, S., Chen, X., Cheng, H.-T., Chi, E. H., Le, Q. V., and Zhou, D. Take a step back: Evoking reasoning via abstraction in large language models, 2023.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., and Wang, Y.-X. Language agent tree search unifies reasoning acting and planning in language models, 2023a.

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q. V., and Chi, E. H. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*, 2023b.

A. Accuracy Analysis: ReAct vs. `LLMCompiler`

In this section, we delve into a detailed analysis that compares the accuracy of both ReAct and `LLMCompiler`, highlighting two failure cases that are prevalent in ReAct: (i) premature early stopping; and (ii) repetitive function calls. Furthermore, we demonstrate that while those failure cases negatively impact the ReAct accuracy, they can be effectively addressed by `LLMCompiler`, thereby yielding the improved accuracy of our framework. We analyze two specific scenarios: the Movie Recommendation evaluation with GPT, where ReAct often prematurely stops, leading to significantly lower accuracy compared to `LLMCompiler` (68.60 vs. 77.13 in Tab. 1); and the HotpotQA evaluation with LLaMA-2 70B, where ReAct's repetitive function calls result in a notable accuracy degradation compared to `LLMCompiler` (70.00 vs. 77.80 in Tab. 1).

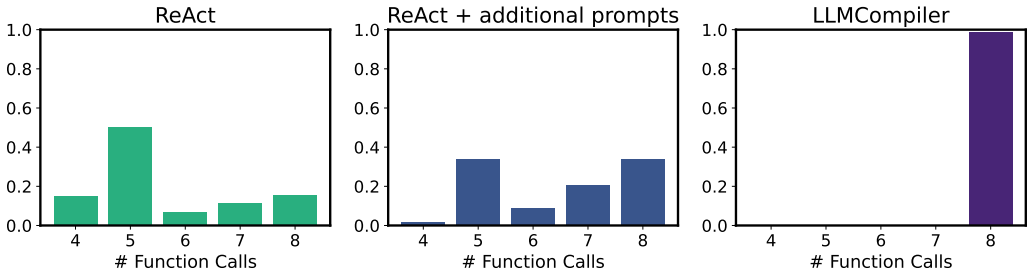

Figure A.1. Distributions of the number of function calls when running the Movie Recommendation benchmark on ReAct (Left), ReAct with specific prompts to avoid early stopping (Middle, corresponding to ReAct† in Tab. 1), and `LLMCompiler` (Right). `LLMCompiler` (Right) consistently completes the search for all 8 movies, whereas ReAct (Left) often exit early, demonstrated by about 85% of examples stopping early. Although the custom prompts shift ReAct's histogram to higher function calls (Middle), they still fall short of ensuring comprehensive searches for all movies. gpt-3.5-turbo is used for the experiment.

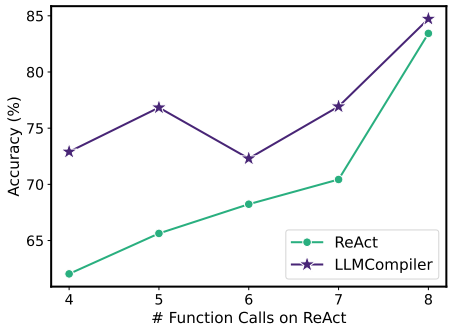

Figure A.2. The Movie Recommendation accuracy of the examples that are categorized by the number of function calls on ReAct, measured both on ReAct and `LLMCompiler`. The plot indicates that in ReAct, a decrease in the number of function calls correlates with lower accuracy, indicating that premature exits lead to reduced accuracy. In contrast, when the same examples are evaluated using `LLMCompiler`, which ensures complete searches for all eight movies before reaching a decision, they achieve higher and more consistant accuracy than those processed by ReAct. gpt-3.5-turbo is used for the experiment, and the results are averaged over 3 different runs.

Premature Early Stopping of ReAct. ReAct frequently suffers from premature early stopping, ceasing function calls too early, and making decisions based on incomplete information. A clear example of this is observed in the Movie Recommendation benchmark, where ReAct often searches for fewer than the required 8 movies before delivering its final answer. In Fig. A.1 (Left), we illustrate the distribution of the number of function calls within ReAct (using GPT) across thhe Movie Recommendation benchmark. Here, we observe around 85% of the examples exhibit early stopping, making decisions without completing all 8 movie searches. This contrasts with `LLMCompiler` (Right), where almost all examples (99%) complete the full search of 8 movies. Although adding specific prompts to ReAct to prevent early stopping shifts the distribution towards more function calls (Fig. A.1, Middle), resulting in an accuracy improvement from 68.60 to 72.47 (ReAct† in Tab. 1), it is nevertheless an imperfect solution.

To further assess how early stopping negatively impacts accuracy, we categorize Movie Recommendation benchmark examples by their number of function calls in ReAct. We then evaluated these groups using `LLMCompiler`, ensuring complete search results for all 8 movies. Fig. A.2 reveals that fewer function calls in ReAct correlate with lower average

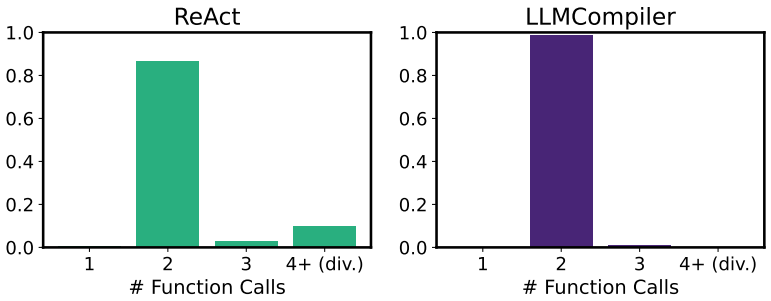

Figure A.3. Distributions of the number of function calls when running the HotpotQA benchmark on ReAct (Left) and `LLMCompiler` (Right). While `LLMCompiler` (Right) consistently completes the task within 2 function calls, which is expected as HotpotQA exhibits a 2-way parallelizable pattern, ReAct (Left) shows that around 10% of the examples undergo repetitive (>4) function calls, resulting in a diverging behavior of the framework. LLaMA-2 70B is used for the experiment.

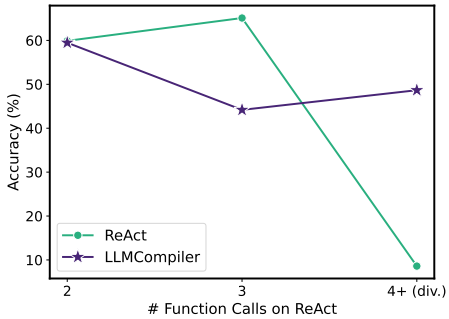

Figure A.4. The HotpotQA accuracy of the examples that are categorized by the number of function calls on ReAct, measured both on ReAct and `LLMCompiler`. The plot indicates that in ReAct, repetitive function calls of more than or equal to four times can result in a significant accuracy degradation due to its infinite looping and diverging behavior. On the other hand, when the same examples are evaluated using `LLMCompiler`, which ensures only two searches per example, they achieve a higher of around 50%. LLaMA-2 70B is used for the experiment.

accuracy (green line). Conversely, if these examples were processed through `LLMCompiler`, with complete searches for all eight movies, they consistently attained higher accuracy (purple line). This not only indicates that ReAct struggles with premature exits (which is not fully addressed by prompting), but the earlier it stops, the greater the decline in accuracy, contributing to the overall accuracy drop observed in Tab. 1. In contrast, `LLMCompiler` effectively addresses this issue.

Repetitive Function Calls of ReAct. Another common failure case of ReAct is its tendency for repetitive function calls, often leading to infinite loops or exceeding the context length limit. This problem is particularly noticeable in the HotpotQA benchmark where ReAct repeatedly calls the same function if the Wikipedia search returns insufficient information about the searched entity. Although HotpotQA is inherently 2-way parallelizable, as illustrated in Fig. A.3, we observe that about 10% of its examples require more than four function calls in ReAct, usually resulting in an infinite loop or a divergent behavior. In contrast, `LLMCompiler` executes only two function calls for most examples.

To show how the repetitive function calls impact the overall accuracy, we conduct an accuracy analysis similar to the previous case. In Fig. A.4, we categorize HotpotQA benchmark examples by the number of function calls in ReAct, and then we compare their accuracy on both ReAct and `LLMCompiler`. The analysis reveals that examples that launch two function calls in ReAct maintain the same accuracy in `LLMCompiler`. However, cases with more than four function calls in ReAct, which often lead to divergent behavior, show less than 10% accuracy in ReAct. On the other hand, when these examples are processed with `LLMCompiler`, they achieve around 50% accuracy by circumventing repetitive calls. It is worth noting that there are instances with three function calls in ReAct, where an extra search can lead to improved accuracy by retrying with an alternate entity name when the initial search fails, yielding a better accuracy than `LLMCompiler`. While this shows a potential adaptability advantage of ReAct, such instances represent less than 3% of cases.

Table C.1. A latency comparison between using and not using streaming in the Planner. Streaming yields consistent latency improvement across different benchmarks, as it enables the Task Fetching Unit to start task execution immediately as each task is produced by the Planner. The impact of streaming is especially notable in the ParallelQA benchmark, where tool execution times are long enough to effectively hide the Planner's execution time.

Benchmark	w/o streaming (s)	w/ streaming (s)	Latency speedup
HotpotQA	4.00	**3.95**	1.01×
Movie Rec.	5.64	**5.47**	1.03×
ParallelQA	21.72	**16.69**	1.30×

B. Failure Case Analysis of `LLMCompiler`

This section delves into a qualitative analysis of `LLMCompiler`'s failure cases on the ParallelQA benchmark, which can be broadly attributed to failures in the Planner, Executor, or the final output process. Failures in the final output process refer to cases when LLMs are unable to use the observations collected from tool execution (which are incorporated into the context) to deliver the correct answer to the user. Among the 10.6% (36 examples) of `LLMCompiler`'s total failures reported in Tab. 1, we have noted that the Planner, Executor, and final output process contributed to 8%, 64%, and 28% of the failures, respectively. The Planner's 8% failure rate is exclusive to `LLMCompiler`. For instance, the Planner would incorrectly map inputs and outputs by assigning a wrong identifier as an input to a subsequent task, thereby forming an incorrect DAG. However, with adequate tool definitions and in-context examples, Planner errors are significantly reduced (only 3 instances in total throughout our evaluation), highlighting the LLM's capability to decompose problems into complex multi-task dependencies.

The remaining 92% of the total failures are attributed to the Executor and the final output process. The Executor accounts for most of these failures (64%), with common issues like the `math` tool choosing wrong attributes or mishandling unit conversions. For the final output process (28% of failures), errors include incorrect conclusions from the gathered observations, such as failing to pick the smallest attribute from the collected data. It's worth noting that these problems are not exclusive to `LLMCompiler`, but they also occur in ReAct. Nevertheless, `LLMCompiler` tends to have slightly fewer failures in these areas than ReAct, as it provides only relevant contexts to each tool, aiding in more accurate information extraction. We believe that optimizing the structure of the agent scratchpad, rather than simply appending observations, could further reduce failures in the final output process.

C. Related Work

Here, we continue with related work, which we started in Sec. 2.

C.1. Tool-Augmented LLMs

A notable work is Toolformer (Schick et al., 2023), which produces a custom LLM output to let the LLM decide what the inputs for calling the functions should be and where to insert the result. This approach has inspired various tool calling frameworks (Liang et al., 2023; Shen et al., 2023). ReAct (Yao et al., 2022) proposed to have LLMs interact with external environments through reasoning and action generation for improved performance. Gorilla (Patil et al., 2023) introduced a finetuned LLM designed for function calling, and ToolLLM (Qin et al., 2023) and RestGPT (Song et al., 2023) have extended LLMs to support real-world APIs. Moreover, OpenAI (OpenAI, 2023) released their own function calling capabilities, allowing their LLMs to return formatted JSON for execution.

D. Experimental Details

Our experiments evaluate two different common scenarios: (1) using API-based closed-source models; and (2) using open-source models with an in-house serving framework. We use OpenAI's GPT models as closed-source models, in particular, gpt-3.5-turbo (1106 release) for HotpotQA and Movie Recommendation, gpt-4-turbo (1106 release) for ParallelQA, and gpt-4 (0613 release) for Game of 24. Experiments on HotpotQA, Movie Recommendation, and ParallelQA were all conducted in November 2023 after the 1106 release. The Game of 24 experiments were conducted over a two-month period from September to October 2023. For an open-source model, we use LLaMA-2 (Touvron et al., 2023), which was hosted on 2 A100-80GB GPUs using the vLLM (Kwon et al., 2023) framework. All the runs have been carried out with

zero temperature, except for `thought_proposer` and `state_evaluator` for the Game of 24 evaluation, where the temperature is set to 0.7. Since OpenAI has randomness in outputs even with temperature 0, we have conducted 3 runs, and we reported the average accuracy. Across ReAct, OpenAI parallel function calling, and `LLMCompiler`, we perform 3, 1, and 5-shot learning for HotpotQA, Movie Recommendation, and ParallelQA, respectively; the same examples across different methods were used to ensure a fair comparison. For the Game of 24, we use 2 in-context examples for the Planner. We use the same instruction prompts across different methods for a fair comparison, except for ReAct† in Sec. 5.1 with additional ReAct-specific prompts. For WebShop experiment, we use gpt-4-0613 with 8k context window and gpt-3.5-turbo model with 16k context window.

E. Analysis

E.1. Parallel Speedup Modeling

While `LLMCompiler` shows noticeable latency gain in various workloads, it is not achieving the $N\times$ latency speedup for N-way parallel workloads. This is mostly due to the overhead associated with `LLMCompiler`'s Planner and final answering process that cannot be parallelized. In our Movie Recommendation experiment, `LLMCompiler`'s Planner and the answering process have an overhead of 1.88 and 1.62 seconds on average, respectively, whose combined overhead already comprises more than half of `LLMCompiler`'s overall latency in Tab 1. Another source of overhead is the straggler effect among the parallel tasks when they need to join together. We observe the average latency of the slowest `search` to be 1.13 seconds, which is nearly $2\times$ the average latency of all tasks, which is 0.61 seconds. Below, we provide an analytical latency modeling of ReAct, `LLMCompiler`, and `LLMCompiler` with streaming, and we provide an analysis of achievable latency speedup.

In this section, our focus is on *embarrassingly parallelizable* workload (pattern Fig. 3(a)), as this allows for a clearer understanding of the impact of each component on potential latency gains. For the precise latency analysis, we consider three key components: the Planner, the Task Fetching Unit, and the Executor, in Fig. 2. Assume that the Planner generates N different tasks to be done. We define P_i as the Planner's output corresponding to the i-th atomic task. Each P_i is a blueprint for a specific atomic task, which we refer to as E_i. The execution of E_i involves a specific function call using the appropriate tool. The latency function of each unit in the system is defined to quantify the time taken for specific operations. For the Planner, the latency is denoted as $T_P(P_i)$, representing the time taken by the Planner to generate the plan P_i. Similarly, for the Executor, the latency, $T_E(E_i)$, corresponds to the time required to complete the task E_i. We ignore the latency of Task Formulation Unit, as it is negligible in this section. Our focus here is on comparing the latency models of ReAct (Yao et al., 2022), and `LLMCompiler`.

To begin our analysis of ReAct's latency, we express its total latency as:

$$T^R = \sum_{i=1}^{N} \left(T_P^R(P_i) + T_E(E_i)\right). \tag{1}$$

Here, the superscript R refers to ReAct. In the ReAct agent system, the process typically involves initial thought generation, followed by action generation and the acquisition of observations through function calls associated with the tool. The creation of both thought and action are collectively considered as part of generating P_i. It is important to note that while the Planner's latency is denoted with a superscript (indicating ReAct), the Executor's latency does not have such a superscript. This is because the function calling and the tools execution remain the same between ReAct and `LLMCompiler`.

For `LLMCompiler`, where all parallelizable tasks are processed concurrently, the total latency is determined by the slowest task among these tasks. Hence, the latency model for `LLMCompiler` can be represented as:

$$T^C = \sum_{i=1}^{N} T_P^C(P_i) + \max_{k\in 1,\ldots,N} T_E(E_k). \tag{2}$$

This expression captures the sum of all planning times plus the execution time of the longest task, reflecting the system's focus on parallel execution.

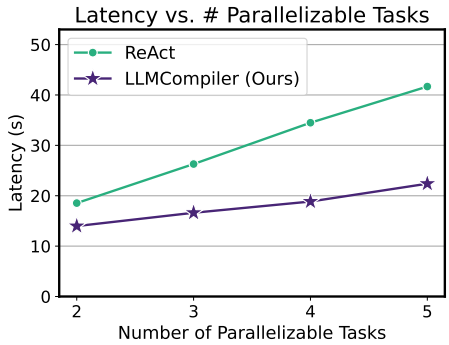

Figure E.5. Latency on the ParallelQA benchmark grouped by the number of maximum parallelizable tasks.

Further, if the Planner employs streaming of the dependency graph, the latency model undergoes a modification and can be expressed as:

$$T^{SC} = \sum_{i=1}^{N} T_P^C(P_i) + T_E(E_N). \tag{3}$$

It is important to note that $T^{SC} \leq T^C$. This implies that the streaming mechanism allows for a more efficient handling of task dependencies, potentially reducing overall latency.

In evaluating the potential speedup achievable with the `LLMCompiler` framework compared to ReAct, the speedup metric, denoted as γ, is defined as follows:

$$\gamma = \frac{T^R}{T^C} = \frac{\sum_{i=1}^{N} \left(T_P^R(P_i) + T_E(E_i)\right)}{\sum_{i=1}^{N} T_P^C(P_i) + \max_{k \in 1,\ldots,N} T_E(E_k)}. \tag{4}$$

This ratio represents the comparative efficiency of `LLMCompiler` over ReAct, considering both planning and execution latencies.

To estimate the upper bound of this speedup, γ_{max}, we assume that the executor latency $T_E(E_i)$ is dominant over the planning latency $T_P(P_i)$ and all the latencies of executing tasks remain the same. Under this assumption, the upper bound is calculated as:

$$\gamma_{\text{max}} \approx \frac{\sum_{i=1}^{N} T_E(E_i)}{\max_{k \in 1,\ldots,N} T_E(E_k)} = N, \tag{5}$$

indicating the theoretical maximum speedup, γ_{max}, is equal to the number of tasks, N.

On the other hand, the lower bound of the speedup, γ, is observed when the planning latency is the predominant factor. Given that the planning latencies of both ReAct and `LLMCompiler` are generally similar, the minimum speedup is approximated as:

$$\gamma_{\text{min}} \approx \frac{\sum_{i=1}^{N} T_P^R(P_i)}{\sum_{i=1}^{N} T_P^C(P_i)} \approx 1. \tag{6}$$

From these observations, we can conclude that to achieve significant latency gains with `LLMCompiler`, it is crucial to (i) reduce the planner overhead and (ii) minimize the occurrence of stragglers.

E.2. Latency versus Number of Parallelizable Tasks

In Fig. E.5, we also report a more detailed latency breakdown on ParallelQA where we show the end-to-end latency as a function of the number of parallel tasks. This is often referred to as weak-scaling in high-performance computing, where the ideal behavior is to have a constant latency as the number of tasks is increased. We can see that ReAct's latency increases proportionally to the number of tasks, which is expected as it executes the tasks sequentially. In contrast, the latency of `LLMCompiler` increases at a much smaller rate, as it can perform multiple function calls in parallel when possible. The reason the end-to-end latency increases slightly with `LLMCompiler` is due to the overhead of the Planner, which needs to generate plans initially, and which cannot be parallelized. We provide a further analysis of this in Appendix E.1.

Table E.2. Accuracy and latency comparison of `LLMCompiler` compared to ReAct on the HotpotQA bridge benchmark. ReAct† denotes ReAct with additional prompting that minimizes looping and early stopping, similar to Tab. 1.

Method	Accuracy (%)	Latency (s)
ReAct	22.7	7.07
ReAct†	23.1	6.42
`LLMCompiler`	**26.3**	**4.70**

Table E.3. Qualitative comparison between `LLMCompiler` and other frameworks including ReAct (Yao et al., 2022), TPTU (SA for Sequential Agent and OA for One-step Agent) (Ruan et al., 2023a), ViperGPT (Surís et al., 2023) and HuggingGPT (Shen et al., 2023).

Method	Planning	Replanning	Parallel Execution	Domain
ReAct	X	-	X	All
TPTU-SA	X	-	X	All
TPTU-OA	O	X	X	All
ViperGPT	O	X	X	Limited
HuggingGPT	O	X	O	Limited
`LLMCompiler`	O	O	O	All

E.3. Additional Experiments on the HotpotQA Bridge Benchmark

In our main experiments in Sec. 5.1, we used the comparison benchmark in HotpotQA to demonstrate the capability of `LLMCompiler` in efficiently executing 2-way parallelizable workloads. The other part of the benchmark, called 'bridge,' involves sequential tasks such as "What government position was held by the woman who portrayed Corliss Archer in the film Kiss and Tell?" `LLMCompiler` is not limited to the comparison benchmark, but it can also be applied to the bridge benchmark due to its replanning capability: initially, it searches for the woman who played Corliss Archer in the film Kiss and Tell, and then, through replanning, searches the government position held by this woman for the example above.

Similar to our experiments with the comparison benchmark, Tab. E.2 compares `LLMCompiler` against ReAct and ReAct with the additional prompt that avoids repetitive function calling and early stopping (ReAct†) on the bridge benchmark. We observe 4 and 3% accuracy improvement, respectively, which is attributed to ReAct's repetitive function invocation – even with the additional prompt (ReAct†), we have still observed 5% of the examples failing with this issue. Furthermore, such repetitive function call also accounts for the slightly higher latency of ReAct compared to ours. This experiment demonstrates that `LLMCompiler` allows for efficient and accurate function calling for both parallel and sequential workloads.

F. Additional Discussions about Related Works

TPTU (Ruan et al., 2023a), HuggingGPT (Shen et al., 2023), and ViperGPT (Surís et al., 2023) have introduced end-to-end plan-and-solve frameworks. In this section, we discuss how `LLMCompiler` distinguishes itself from other frameworks from various angles, including the capabilities in (i) planning and replanning; (ii) parallel execution; and (iii) addressing a wider range of problem domains. Refer to Tab. E.3 for the summary.

Parallel Execution: Parallel execution is a critical feature in the `LLMCompiler` framework that allows for efficient function calling and job completion. While the One-step Agent in TPTU (i.e., TPTU-OA) incorporates planning, it does not enable parallel function calling, as it only decomposes a user input into a sequence of functions and the associated arguments without their inter-dependencies. ViperGPT generates Python codes. However, ViperGPT, by itself, does not support parallel execution without a dedicated parallel processing engine since the standard Python interpreter lacks support for parallel execution. While HuggingGPT enables parallel execution, it strictly targets models in HuggingFace, making it hard to apply in a wide range of problems and domains that LLMCompiler supports.

Planning and Replanning: The TPTU's Sequential Agent (i.e., TPTU-SA) is an iterative framework like ReAct (Yao et al., 2022) that executes one action per iteration. While TPTU-OA, HuggingGPT, and ViperGPT are all planning-based frameworks that plan out multiple actions prior to execution, they lack replanning capabilities. `LLMCompiler`, in contrast, incorporates the replanning mechanism to generate a new set of tasks when the previous plans are not sufficient enough to deliver the response back to the user. This enables `LLMCompiler` to adapt plans based on intermediate results that are a priori unknown, without the need for introducing complex branching logic, thereby extending the scope of problems that it can address.

Table F.4. Accuracy and latency speedup comparison of `LLMCompiler` compared to ReAct and TPTU (SA for Sequential Agent and OA for One-step Agent) on the HotpotQA comparison benchmark using gpt-3.5-turbo. ReAct† and TPTU-SA† denote ReAct and TPTU-SA with additional prompting that minimizes looping and early stopping, respectively, similar to Tab. 1.

Method	Accuracy (%)	Speedup
ReAct	61.52	-
ReAct†	62.47	1×
TPTU-SA	34.16	-
TPTU-SA†	44.59	1.09×
TPTU-OA	57.50	1.35×
`LLMCompiler`	**62.00**	**1.51**×

Problem Domains: ViperGPT and HuggingGPT aim for vision tasks via Python code generation and models in HuggingFace, respectively, showing significant promise in these specific areas. In contrast, `LLMCompiler` targets a general framework that enables efficient and accurate function calling in a wide range of problem domains, rather than restricting itself to specific fields.

F.1. Quantitative Comparison between `LLMCompiler` and TPTU

Additionally, in Tab. F.4, we additionally provide accuracy and latency speedup of `LLMCompiler` against TPTU-SA and TPTU-OA. Since the official implementation of TPTU is not available, we implemented TPTU-SA and TPTU-OA based on the prompts provided in the original paper. As can be seen in the table, the results clearly demonstrate `LLMCompiler`'s latency and accuracy benefit over both TPTU-SA and TPTU-OA. Compared with TPTU-SA, `LLMCompiler` exhibits a significant accuracy improvement due to TPTU's prevalent issue with repetitive function calls. Note that this issue is not fully mitigated even with better prompting (TPTU-SA†), leading to $\sim$15% of examples failing with repetitive function calls. Compared with both TPTU-SA and TPTU-OA, `LLMCompiler` also benefits from reduced latency through parallel task execution. Overall, the results are consistent with the main experiments and analysis against other baseline methods (i.e., ReAct and OpenAI's parallel function calling).

G. User-Supplied Examples for `LLMCompiler` Configuration

`LLMCompiler` provides a simple interface that allows for tailoring the framework to different use cases by providing tool definitions as well as optional in-context examples for the Planner. Below, we provide the Planner example prompts that are used to set up the framework for the Movie Recommendation and Game of 24 benchmarks with only a few lines of prompts.

G.1. Movie Recommendation Example Prompts

```
Question: Find a movie similar to Mission Impossible, The Silence of the
Lambs, American Beauty, Star Wars Episode IV - A New Hope
Options:
Austin Powers International Man of Mystery
Alesha Popvich and Tugarin the Dragon
In Cold Blood
Rosetta

1. search("Mission Impossible")
2. search("The Silence of the Lambs")
3. search("American Beauty")
4. search("Star Wars Episode IV - A New Hope")
5. search("Austin Powers International Man of Mystery")
6. search("Alesha Popvich and Tugarin the Dragon")
7. search("In Cold Blood")
8. search("Rosetta")
Thought: I can answer the question now.
```

```
9.  finish()
###
```

G.2. Game of 24 Example Prompts

```
Question:  "1 2 3 4", state_list:  [""]
$1 = thought_proposer("1 2 3 4", "")
$2 = state_evaluator("1 2 3 4", "$1")
$3 = top_k_select("1 2 3 4", ["$1"], ["$2"])
$4 = finish()
###
Question:  "1 2 3 4", state_list:  ["1+2=3(left:3 3 4)","2-1=1(left:1 3
4)","3-1=2(left:2 2 4)","4-1=3(left:2 3 3)","2*1=2(left:2 3 4)"]
$1 = thought_proposer("1 2 3 4", "1+2=3(left:3 3 4)")
$2 = thought_proposer("1 2 3 4", "2-1=1(left:1 3 4)")
$3 = thought_proposer("1 2 3 4", "3-1=2(left:2 2 4)")
$4 = thought_proposer("1 2 3 4", "4-1=3(left:2 3 3)")
$5 = thought_proposer("1 2 3 4", "2*1=2(left:2 3 4)")
$6 = state_evaluator("1 2 3 4", "$1")
$7 = state_evaluator("1 2 3 4", "$2")
$8 = state_evaluator("1 2 3 4", "$3")
$9 = state_evaluator("1 2 3 4", "$4")
$10 = state_evaluator("1 2 3 4", "$5")
$11 = top_k_select("1 2 3 4", ["$1", "$2", "$3", "$4", "$5"], ["$6", "$7",
"$8", "$9", "$10"])
$12 = finish()
###
```

H. Pre-defined `LLMCompiler` Planner Prompts

The pre-defined `LLMCompiler` Planner prompt provides it with specific instructions on how to break down tasks and generate dependency graphs while ensuring that the associated syntax is formatted correctly. This prompt contains specific rules such as assigning each task to a new line, beginning each task with a numerical identifier, and using the `$` sign to denote intermediate variables.

```
- Each action described above contains input/output types and descriptions.
- You must strictly adhere to the input and output types for each action.
- The action descriptions contain the guidelines.  You MUST strictly follow
 those guidelines when you use the actions.
- Each action in the plan should strictly be one of the above types.  Follow
 the Python conventions for each action.
- Each action MUST have a unique ID, which is strictly increasing.
- Inputs for actions can either be constants or outputs from preceding
 actions.  In the latter case, use the format $id to denote the ID of the
 previous action whose output will be the input.
- Ensure the plan maximizes parallelizability.
- Only use the provided action types.  If a query cannot be addressed using
 these, invoke the finish action for the next steps.
- Never explain the plan with comments (e.g.  #).
- Never introduce new actions other than the ones provided.
```

In addition to user-provided functions, the Planner includes a special, hard-coded `finish` function. The Planner uses this function either when the plan is sufficient to address the user query or when it can no longer proceed with planning before executing the current plan, i.e., when it deems replanning necessary. When the Planner outputs the `finish` function, its plan generation stops. Refer to Appendix G for examples of the Planner's usage of the `finish` function in planning. The definition of the `finish` function is as below and is included as a prompt to the Planner along with the definitions of other user-provided functions.

```
finish():
 - Collects and combines results from prior actions.
 - A LLM agent is called upon invoking join to either finalize the user
query or wait until the plans are executed.
 - join should always be the last action in the plan, and will be called in
two scenarios:
 (a) if the answer can be determined by gathering the outputs from tasks to
generate the final response.
 (b) if the answer cannot be determined in the planning phase before you
execute the plans.
```

I. ParallelQA Benchmark Generation

Inspired by the IfQA benchmark (Yu et al., 2023), our custom benchmark ParallelQA contains 113 examples that are designed to use mathematical questions on factual details of different entities to answer questions, thus requiring a mix of search and mathematical operations that are interdependent in various ways. For instance, the benchmark includes examples like "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?" requires four parallel search tasks, followed by math tasks dependent on the search outcomes, that can be executed in parallel.

The main objective of the benchmark is to quantify the framework's ability to decompose an input into multiple tasks to derive an answer. Therefore, we have meticulously selected 56 distinct entities across various domains whose attributes can be accessible from Wikipedia search. By minimizing tool execution (i.e., Wikipedia search) failures, we have aimed our benchmark to effectively assess the frameworks' abilities to decompose questions into multiple tasks, plan them out, and derive final answers based on observations. Furthermore, to incorporate diverse execution patterns, we crafted various dependency patterns that perform unary and binary math operations after searching for additional information about entities in a given question. We have also curated different questions that accommodate different numbers of maximally parallelizable tasks, ranging from 2 to 5, and we have included varying numbers of joins between parallel function calls as well to increase problem complexity. For instance, we have 2 and 3 joins in Fig. 3 (b) and (c), respectively. The benchmark contains 113 different examples, that were populated by GPT-4 based on the aforementioned criteria and labeled by humans afterward.

J. Details of the Game of 24 and the Tree-of-Thoughts Approach

The Game of 24 is a mathematical reasoning game that challenges players to manipulate a given set of four numbers, using the basic arithmetic operations of addition, subtraction, multiplication, and division, to arrive at the number 24. The rule of this game is that the given numbers must be used only once. For instance, given the numbers 2, 4, 4, and 7, one possible solution is $4 \times (7 - 4) \times 2 = 24$. This is a non-trivial reasoning benchmark for LLMs, highlighted by the fact that even advanced models like GPT-4 exhibit only a 4% success rate, even when using chain-of-thought prompting (Yao et al., 2023b).

In ToT, the problem is solved in several steps. At each step, the LLM, referred to as the thought proposer, generates thoughts. Each thought is a partial solution that consists of two numbers and an arithmetic operation between them. Then, these thoughts are fed into the state evaluator which assigns a label for each of them. These labels are 'sure,' 'likely,' and 'impossible,' which are given to thoughts to denote how likely they could produce 24 with additional arithmetic operations between the result and the remaining numbers. Only the thoughts that are likely to produce 24 continue onto the next step. This process is illustrated in Figure J.6.

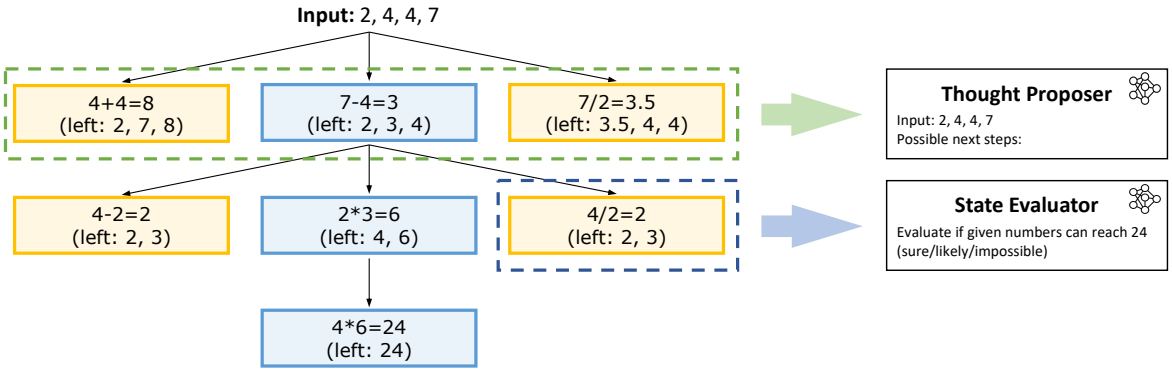

Figure J.6. Visualization of the Tree of Thoughts (ToT) in the Game of 24. Each node represents a distinct proposal, beginning with the root node and branching out through the application of single operations by the thought proposer. Subsequent states are evaluated by the state evaluator for their potential to reach the target number 24. The ToT retains the top-5 states according to their values.

K. Details of WebShop Experiments

K.1. WebShop Environment

The WebShop environment simulates an online shopping platform. Tasks are designed for the agent to find the item that best matches the given instruction. For instance, if the instruction specifies, "I am looking for a queen-sized bed that is black, and priced lower than 140.00 dollars," the agent's task is to pinpoint the bed that precisely fits these criteria: "queen-sized," "black," and "priced under 140.00 dollars." For each item, there is an associated reward measuring how well this item matches the instruction based on price, item options, and other details contained in the item page. The evaluation metrics are the success rate—the proportion of episodes where the selected product satisfies all requirements—and the average score—the mean reward across episodes.

K.2. Baseline Methods

In addition to ReAct, we use LASER (Ma et al., 2023) and LATS (Zhou et al., 2023a) as baseline methods to compare against `LLMCompiler`. LASER (Ma et al., 2023) solves tasks through a state-exploration approach. In the context of WebShop, the possible environment pages are encoded as different states (e.g., search page, item page, and item detail subpage). Actions are used to transition between these states, such as executing a search query, selecting an item, checking the item detail, navigating the next search page and so on. The Webshop exploration is therefore reduced to a search problem on the given state-space graph.

Using a variant of Monte Carlo Tree Search, LATS (Zhou et al., 2023a) plans its actions by constructing a decision tree, evaluating potential moves based on their likelihood of success, and selecting actions through a balance of exploration and exploitation. The agent then adapts its strategy based on feedback from the environment, learning from both successes and failures to refine its decision-making process. This iterative approach allows LATS to navigate complex online shopping tasks, albeit much more slowly due to its exhaustive tree search.