ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Parrot: Efficient Serving of LLM-based Applications with Semantic Variable

Chaofan Lin[1]*, Zhenhua Han[2], Chengruidong Zhang[2], Yuqing Yang[2]
Fan Yang[2], Chen Chen[1]*, Lili Qiu[2]
[1]*Shanghai Jiao Tong University,* [2]*Microsoft Research*

Abstract

The rise of large language models (LLMs) has enabled LLM-based applications (a.k.a. AI agents or co-pilots), a new software paradigm that combines the strength of LLM and conventional software. Diverse LLM applications from different tenants could design complex workflows using multiple LLM requests to accomplish one task. However, they have to use the over-simplified request-level API provided by today's public LLM services, losing essential application-level information. Public LLM services have to blindly optimize individual LLM requests, leading to sub-optimal end-to-end performance of LLM applications.

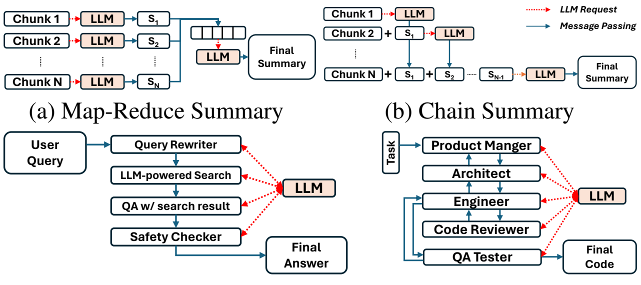
This paper introduces Parrot, an LLM service system that focuses on the end-to-end experience of LLM-based applications. Parrot proposes *Semantic Variable*, a unified abstraction to expose application-level knowledge to public LLM services. A Semantic Variable annotates an input/output variable in the prompt of a request, and creates the data pipeline when connecting multiple LLM requests, providing a natural way to program LLM applications. Exposing Semantic Variables to the public LLM service allows it to perform conventional data flow analysis to uncover the correlation across multiple LLM requests. This correlation opens a brand-new optimization space for the end-to-end performance of LLM-based applications. Extensive evaluations demonstrate that Parrot can achieve up to an order-of-magnitude improvement for popular and practical use cases of LLM applications.

1 Introduction

Large language models (LLMs) have demonstrated a remarkable language understanding capability [7,41]. This enables a paradigm shift in application development. In this new paradigm, one or multiple application entities, known as AI agents or co-pilots, communicate with LLMs via natural language, known as "prompts", to accomplish a task collaboratively. For example, Meeting applications like Microsoft Teams or Google Meet can summarize meeting discussions through LLMs [33]. Search engines like Google and Bing can be enhanced with Chat ability through LLMs [14,34]. It is believed such LLM-based applications will become the mainstream applications in the near future [13].

To accomplish a task, LLM-based applications typically require multiple rounds of conversation. The conversation, implemented through multiple API calls to LLM, demonstrates complex workflow patterns. Figure 1 illustrates several popular conversation patterns. For example, a meeting summary application [8,33] often divides a lengthy document into multiple shorter sections, each satisfying the length constraint of the LLM conversation and thus can be summarized and combined into the final summary through the Map-Reduce or chaining summary patterns. Chat-based applications, e.g., Bing Copilot [34], call LLM APIs multiple times to generate answers based on user queries. Multiple agents, each representing a different role played by different LLM calls, can collaborate to achieve a task [22,47,54].

Public LLM service providers have to face diverse tenants and applications, each with different workflows and performance preference. However, existing API design for LLM service provision is still request-centric. Public LLM services only observe tons of individual requests, without knowing any



(a) Map-Reduce Summary (b) Chain Summary

(c) LLM-Powered Search (d) Multi-agent Coding

Figure 1: The workflow of popular LLM-based applications. The final result requires multiple LLM requests.

*This work is partially done while Chaofan Lin's internship and Dr. Chen Chen's visting scholar in Microsoft Research.

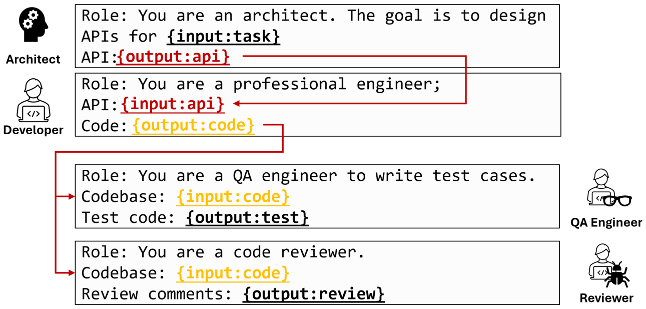
application-level information, e.g., which requests belong to the same application, how different requests are connected, or whether there are any similarities. The lost application-level information makes public LLM service blindly optimize the performance of individual requests, leading to sub-optimal *end-to-end* performance of LLM applications. In this paper, we observe there exist significant opportunities to improve the *end-to-end* experience of LLM applications by exploiting the application-level information, especially the *correlation* of multiple LLM requests.

First, multiple consecutive LLM requests may be dependent: the result of one request could be the direct input of the next request. Therefore, it is desirable to colocate those requests together and execute them consecutively on the LLM service side. However, unaware of their dependencies, these requests have to be executed interactively between the client side of LLM-based applications and the public LLM services. These clients, often located on the other end of the Internet, can only issue the second request after they receive the result of the first request. This unnecessarily incurs *extra overhead of consecutive requests* on network latency as well as losing the opportunity of co-scheduling these consecutive requests (§3).

Second, LLM requests may have *diverse scheduling preference*, even within a single application. For example, in Figure 1a, to reduce the end-to-end latency, the requests representing multiple Map tasks should be batched more aggressively to increase the throughput of the Map tasks; while the Reduce task, due to its scarcity, should be optimized for latency. Unfortunately, public LLM services cannot discriminate the difference between the two types of tasks. As a result, the current practice is to blindly optimize the latency for individual requests, which might not be desirable for the end-to-end experience.

Third, there exists a high degree of *commonality* across LLM requests. Popular LLM applications (e.g., Bing Copilot [32], GPTs [42]) use a long system prompt, including task definitions, examples, and safety rules, to guide the behavior of LLM applications. The long system prompt is usually static and common for all users. As existing public LLM services treat each request individually, these common prefix prompts are provided repeatedly in each request, leading to a great waste of storage, computation, and memory bandwidth. Our analysis of a production LLM-based search engine shows that over 94% of tokens in the requests are repeated across different users.

Although we have seen some emerging engine-level techniques [25,56,63] proposed to optimize the above three cases, they all work based on certain application-level knowledge, which is lost in nowadays public LLM services. In a nutshell, due to the lack of understanding of the correlations of LLM requests, existing LLM services cannot leverage the three opportunities, leading to high end-to-end service latency and reduced throughput. Based on the above facts and insights, we introduce Parrot, an LLM service system that treats LLM applications as first-class citizens. Parrot retains most of application-level information by a simple abstraction Semantic Variable, achieving a perfect balance between increasing system complexity and bringing new information for optimization. A Semantic Variable is a text region in the prompt with a specific semantic purpose, such as a task instruction, a list of few-shot examples, an input, or an output. A Semantic Variable can also work as the data pipeline that connects multiple LLM requests. Semantic Variable naturally exposes the information of prompt structures and correlations of requests to LLM services. By inspecting Semantic Variable at runtime, Parrot can perform conventional data flow analysis to derive the data dependency between LLM requests just-in-time.



```
Role: You are an architect. The goal is to design
APIs for {input:task}
API:{output:api}
```

```
Role: You are a professional engineer;
API:{input:api}
Code:{output:code}
```

```
Role: You are a QA engineer to write test cases.
Codebase: {input:code}
Test code: {output:test}
```

```
Role: You are a code reviewer.
Codebase: {input:code}
Review comments: {output:review}
```

Figure 2: The communication of consecutive LLM requests in multi-agent applications.

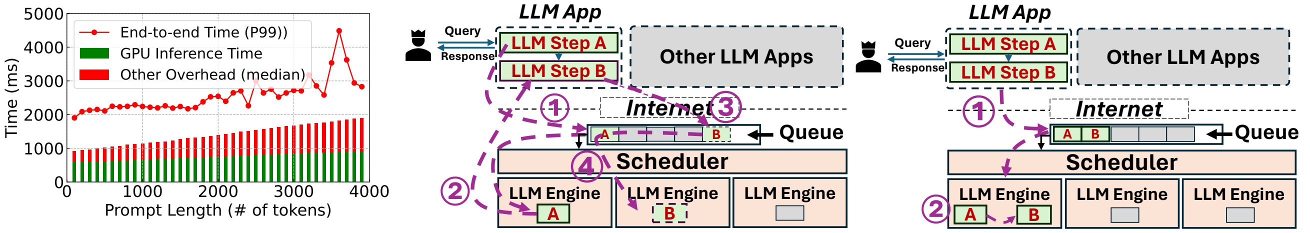
By analyzing the application-level information, Parrot's unified abstraction naturally enables joint optimizations, which bring better global optimality. The same data pipeline built by Semantic Variables can enable multiple optimizations simultaneously, including hiding data pipeline's latency, objective deduction for a better scheduling and commonality analysis to perform de-duplication. Parrot's scheduling also takes different opportunities into accounts under the unified abstraction. Our extensive evaluation of Parrot on popular LLM-based applications, including the production and open-source projects, shows Parrot achieves up to 11.7× speedup or 12× higher throughput compared with the state-of-the-art solutions. Parrot is open-sourced at https://github.com/microsoft/ParrotServe, including the code for artifact evaluations to reproduce our experiment results.

2 Background

LLM Service. Most LLM services are provisioned as a conditional generation service via a text completion API.

$$\mathit{Completion}(\text{prompt} : \text{str}) \rightarrow \text{generated\_text} : \text{str}.$$

The application client provides a text prompt, and the LLM service responds with the generated text. Behind the API, an LLM service provider runs one or multiple clusters of LLM inference engines. A request scheduler dispatches LLM



(a) Latency Breakdown

(b) Current LLM Services

(c) Our system: Parrot

Figure 3: The end-to-end latency breakdown of current LLM services. The source of the overhead comes from network and queuing due to chatty interaction between LLM application and LLM services, which is eliminated in our system Parrot.

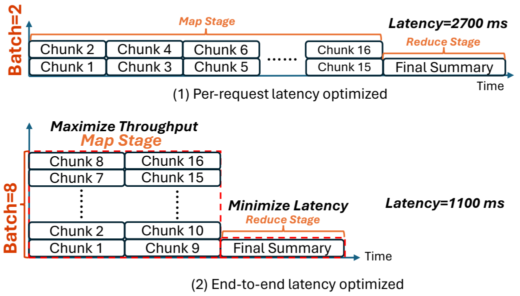
requests from a queue to an LLM inference engine, which uses a set of GPUs to conduct the LLM inference.

LLM-based Applications. Figure 1 highlights the representative workflows of how LLM is used in the applications. Due to the limited context window of LLMs (e.g., 4,096 for GPT-3.5-Turbo [40]), data analytics on long documents follow a *map-reduce style* (Figure 1a) or *chain style* (Figure 1b) workflow to generate the final results. It splits the long transcript into chunks, uses multiple requests to generate partial results for each chunk (the Map task), and combines them altogether (a Reduce task) or incrementally (the chain style) to generate the final result. Chat-based search engine in Figure 1c may use consecutive LLM requests to discern query intention, enrich the query with supplementary information, retrieve related data, undergo a safety check, and finally generate the response. Multi-agent in Figure 1d and Figure 2 is another type of workflow using multiple LLM requests, each with a designated role. Different roles work collaboratively on the same task, e.g., AutoGen [54] and MetaGPT [22] use the roles like product manager, architect, engineer, and QA tester. They communicate with each other on a software project. Each role is supported by one or multiple LLM requests to act as the designed role to generate their responses.

3 Problems of Serving LLM Applications

Although LLM's text completion API provides a flexible way of building LLM applications, it loses the application-level information to public LLM services, leading to the following challenges.

Excessive Overhead of Consecutive Requests. As demonstrated in Figure 1, LLM applications frequently make multiple LLM calls to complete a single task. Due to the request-centric design of existing public LLM services, which generate responses for each request individually, developers have to parse the output of an LLM request and compose the prompts for subsequent LLM requests on the client side. Figure 3a shows our empirical study of the latency breakdown of the LLM calls from a popular LLM application in our production, which uses a chain-style workflow. The prompt lengths range from 150 to 4000 tokens and the output length is around 50 tokens. We find there is a significant portion of the latency of LLM API call originates outside the LLM engine (30 ∼ 50% on average and over 70% in the worst cases). The overhead increases with the growing length of prompts. The high latency can sometimes result in API timeouts and resubmissions.



(1) Per-request latency optimized

(2) End-to-end latency optimized

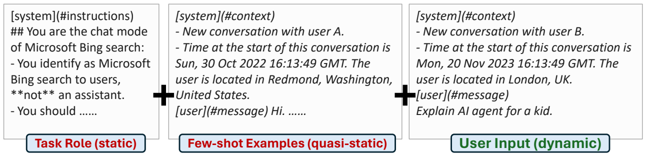
Figure 4: Request-centric scheduling v.s. application-centric scheduling for the map-reduce style document summary task.

Such overhead is due to the chatty interaction between LLM services and clients. Figure 3b illustrates the overhead of a simple two-step LLM application (e.g., *chain-style* summary of two text chunks). Existing LLM services are unaware of the dependency among such requests, where the output of the previous request may be the direct input of the next one. For such consecutive and dependent requests, the client has to wait for the arrival of the response to the first LLM request (②) before submitting the next LLM request (③). This unnecessarily incurs heavy network latency because clients and LLM services are typically in different data centers. Moreover, the next LLM request has to suffer extra queuing delays (④), because requests from other applications may arrive between the consecutive LLM requests.

LLM-based App.	# Calls	Tokens	Repeated (%)*
Long Doc. Analytics	2 ∼ 40	3.5k ∼ 80k	3%
Chat Search	2 ∼ 10	5k	94%
MetaGPT [22]	14	17k	72%
AutoGen [54]	17	57k	99%

*We count a paragraph as repeated if it appears in at least two LLM requests.

Table 1: Statistics of LLM calls of LLM applications.



```
[system](#instructions)
## You are the chat mode
of Microsoft Bing search:
- You identify as Microsoft
Bing search to users,
**not** an assistant.
- You should ......
```

```
[system](#context)
- New conversation with user A.
- Time at the start of this conversation is
Sun, 30 Oct 2022 16:13:49 GMT. The
user is located in Redmond, Washington,
United States.
[user](#message) Hi. ......
```

```
[system](#context)
- New conversation with user B.
- Time at the start of this conversation is
Mon, 20 Nov 2023 16:13:49 GMT. The
user is located in London, UK.
[user](#message)
Explain AI agent for a kid.
```

Figure 5: The prompt structure of search copilot shows a long prompt reused by different user queries.

In Table 1, we evaluated four popular LLM applications. The first two are from our production, and the last two are popular open-source projects. They all require tens of LLM calls to complete a single task, which results in high user-perceived latency. Our evaluation in §8.2 shows LLM services that treat requests individually could slow down the end-to-end latency by over 2×. An LLM service can eliminate the overhead if it can handle consecutive requests in a batch. Parrot adopts such an approach. As shown in Figure 3c, the two steps of the same application are scheduled together, thus allowing the output of Step A to be fed directly into Step B—with the network and queuing overhead bypassed.

Misaligned Scheduling Objectives. Due to the lost application information (workflow and application performance objective), existing public LLM services have to blindly use a universal treatment for all requests, e.g., optimizing per-request latency [44]. However, LLM-based applications are more concerned about the end-to-end experience, rather than individual requests. This misaligned optimization objectives may negatively impact end-to-end performance. Considering the map-reduce document summary in Figure 1a, the system should minimize the end-to-end time it takes to receive the final summary, rather than the latency of individual requests. The LLM services optimized for individual requests are not optimal for end-to-end latency.

As depicted in Figure 4, current LLM services must limit the number of concurrent requests running on each LLM engine to control the latency of individual requests. However, there is a trade-off between latency and throughput in LLM inference. Increasing the batch size can bring up to 8.2× higher throughput but lead to 95% higher latency [9]. Yet, if we understand the application-level performance objective, which in this case is the end-to-end latency, we can determine that the ideal scheduling strategy should maximize the throughput (using higher batch sizes) during the map stage and minimize request latency during the reduce stage. This strategy reduces end-to-end latency by 2.4×. Moreover, it uncovers the potential to enhance cluster throughput without compromising the end-to-end latency of LLM applications. This insight is essential for addressing the conflict between rising demand and limited hardware resources. It underscores the necessity of scheduling LLM requests from the perspective of LLM applications, but it also presents the challenge of managing diverse LLM requests with varying performance objectives.

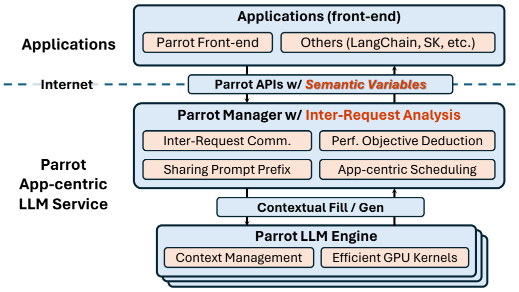


Figure 6: Parrot system overview.

Redundant Computations. Currently, most LLM-based applications exhibit a high degree of redundancy in the prompts of their requests. For instance, Bing Chat [32] has handled more than 1 billion chat prompts. These prompts share the same system prompts that defines the functionality of Bing Chat. OpenAI introduces GPTs [42] to let users customize a ChatGPT for a specific purpose whose prompt template is the same across users. The commonality in prompts is crucial as it delineates the functionality and restrictions of LLM-based applications. The prompt structure in Figure 5 [52] includes a role definition, several examples to enhance the precision of LLM's behaviors and user query details. While the user input is dynamic, the task role is always fixed, and the few-shot examples could be quasi-static in that the same type of tasks use the same examples. This is why more than 94% of prefix tokens could be repetitively used across LLM requests for various users (Table 1). Such commonality also exists in multi-agent applications. For example, MetaGPT [22] and AutoGen [54] recurrently incorporate conversation history into the prompt over several rounds of LLM requests, leading to 72% and 99% redundancy respectively. These redundant sections excessively utilize GPU memory bandwidth and are computed for multiple times. Earlier results have proposed optimizations in LLM engines to avoid redundant GPU memory of shared prompt [25]. However, it is hard for public LLM services to swiftly detect and co-locate the prompt-sharing requests, which be dynamically generated, from tons of diverse requests from diverse applications. Without knowledge about the prompt structure, extensive token-by-token matching for every LLM request is expensive at the cluster level. Hence, if the cluster scheduler of public LLM service cannot dispatch prompt-sharing requests to the same engine, the engine-level redundancy avoidance optimizations would be hard to take effect.

4 Parrot Design

Figure 6 depicts the overview of Parrot's design. Parrot provides a natural way of programming LLM applications with Semantic Variable annotations (§4.1), which is compatible of existing LLM orchestration frameworks, e.g., LangChain [8]. Centering on this abstraction, Parrot Manager is designed

```
import Parrot as P
from Parrot.PerformanceCriteria import LATENCY

@P.SemanticFunction
def WritePythonCode(task: P.SemanticVariable):
""" You are an expert software engineer.
    Write python code of {{input:task}}.
    Code: {{output:code}}
"""

@P.SemanticFunction
def WriteTestCode(
    task: P.SemanticVariable,
    code: P.SemanticVariable):
""" You are an experienced QA engineer.
    You write test code for {{input:task}}.
    Code: {{input:code}}.
    Your test code: {{output:test}}
"""

def WriteSnakeGame():
  task = P.SemanticVariable("a snake game")
  code = WritePythonCode(task)
  test = WriteTestCode(task, code)
  return code.get(perf=LATENCY), test.get(perf=LATENCY)
```

Figure 7: Example: a multi-agent application in Parrot.

to schedule LLM requests at a cluster-level, by deriving the application-level knowledge (§4.2) and optimizing end-to-end performance of application (§5). The manager will schedule the LLM requests to LLM `Engine`, which is formed by a GPU server (or a group of servers) in the cluster that can serve LLM requests independently.

4.1 Semantic Variable

Parrot treats an LLM request as a semantic function[1] implemented using natural language and executed by LLMs. A Semantic Variable is defined as a input or output variable of a semantic function, which is referred as a placeholder in the prompt. Figure 7 shows a simplified example of multi-agent application like MetaGPT [22]. It contains two `SemanticFunctions`, one for the software engineer to write code and one for the QA engineer to write test code. It has three Semantic Variables: `task`, `code`, and `test`, for task description, the code to be developed by the software engineer, and the test code to be developed by the QA engineer, respectively. Although existing LLM orchestration frameworks (e.g., LangChain [8]) also allow placeholders in a prompt, however, the placeholders are rendered with real data before the submission, hence public LLM services cannot detect such a structure. Instead, Parrot relies on Semantic Variables to preserve the prompt structure for further inter-request analysis in public LLM services side.

In addition to the semantic functions, LLM application developers can further define orchestration functions that connect multiple semantic functions (e.g., `WriteSnakeGame` in Figure 7). The Semantic Variables connecting multiple semantic functions form the data pipeline of multiple LLM requests in the public LLM service. A simple data flow analysis of the semantic functions can be done to reveals the connections of multiple LLM requests. E.g., in Figure 7, the `code` variable connects the two LLM requests originating from `WritePythonCode` and `WriteTestCode`, showing their sequential dependency. Different from traditional completion API, Parrot splits a completion request to `submit` operation and `get` operation (§7). A function calling of `SemanticFunction` will trigger the `submit` API to submit a LLM request with its prompt and input Semantic Variables. The execution of a `SemanticFunction` is asynchronous thus it returns the futures of the output Semantic Variables. Through the `get` API, applications can fetch the value of an output Semantic Variable from the public LLM service in an on-demand manner. This asynchronous design allows Parrot-powered LLM service to receive all LLM requests not blocked by native functions and analyze their relationships just-in-time.

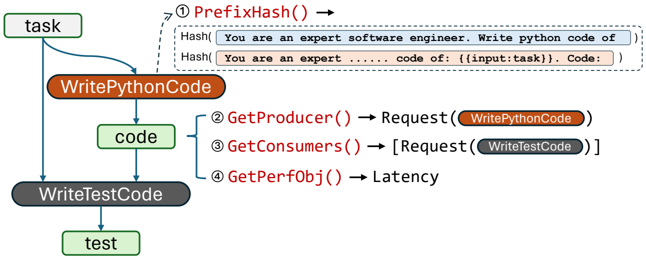

task
WritePythonCode
code
WriteTestCode
test
① PrefixHash() →
Hash(You are an expert software engineer. Write python code of)
Hash(You are an expert ...... code of: {{input:task}}. Code:)
② GetProducer() → Request(WritePythonCode)
③ GetConsumers() → [Request(WriteTestCode)]
④ GetPerfObj() → Latency

Figure 8: Primitives (selected) for Inter-Request Analysis.

The `get` operation supports annotation of performance criteria, showing the end-to-end performance requirement of an application, which can be end-to-end latency or throughput (extensible to more criteria like per-token latency when streaming, and time-to-first-token). For example, the final outputs, `code` and `test` in Figure 7, are fetched using `get` with an objective of end-to-end latency. Criteria of middle variables will be automatically deduced and propagated from final outputs (§5.2). After propagation, each variable is attached to a criterion, which finally works by serving as a hint to Parrot's scheduler (§5.4).

4.2 Primitives of Inter-Request Analysis

In general, Parrot perform inter-request analysis mainly by two types of application-level information deduced from Semantic Variable: DAG of requests and prompt structure. Figure 8 illustrates the DAG workflow of the example shown in Figure 7 and the primitives used for inter-request analysis and optimizations.

DAG-based analysis. As requests, or `SemanticFunctions`, are submitted beforehand, Parrot can receive them all at once and analyze their correlations just-in-time on the service side. Parrot maintains a DAG-like data structure in each user's

[1] The term *semantic function* is borrowed from Semantic Kernel [36].

registered session. Each node is either a request or a Semantic Variable that connects different requests. When a request comes, Parrot inserts it to DAG by linking edges with Semantic Variables it refers through placeholders in the prompts. Parrot can perform conventional dataflow analysis [1, 38] using the primitives to get the producer and consumers of Semantic Variables (i.e., `GetProducer` and `GetConsumers`) to recover dependency of LLM requests. Using the request DAG and the annotated performance criteria (via `GetPerfObj`) of final output Semantic Variables, Parrot can deduct the request-level scheduling preference by analyzing the DAG and the performance objective of final outputs (§5.2).

Prompt structure-based analysis. Based on the prompt structure declared by Semantic Variables, Parrot supports extracting the hash values of an LLM request at multiple positions split by Semantic Variables (i.e., `PrefixHash`). For example, the prompt of `WritePythonCode` has two potential sharing prefix: the text before `{{input:task}}` and the text before `{{output:code}}`, thus there will be two prefix hash values generated. The prefix hashes of LLM requests will be used by swift detection of commonality across multiple requests, supporting both static and dynamically generated contents, as well as within the same type of application or even across applications (§5.3).

5 Optimizations with Semantic Variable

5.1 Serving Dependent Requests

To avoid the unnecessary client-side execution, it requires the dependency of requests at the application level, which is lost in today's public LLM services. With the DAG and primitives illustrated in §4.2, Parrot serves dependent requests efficiently through a graph-based executor. The executor polls constantly and sends it to corresponding engine once ready (i.e. producer requests are all finished), which allows instant execution and maximizes batching opportunities. For consecutive execution of dependent requests, materialized value is transmitted through a message queue allocated for corresponding Semantic Variable, avoiding unnecessary chatty communication between clients and LLM services.

The value of a Semantic Variable in a request may require transformation before being exchanged, e.g., the value of a Semantic Variable is extracted from the JSON-formatted output of an LLM request, which is then fed into consecutive LLM requests. Similar to existing message queue systems that support message transformation (e.g., Kafka [5]), Parrot also supports string transformation to manipulate Semantic Variables during value exchanging among LLM requests. Parrot supports most output parsing methods of LangChain [8], which covers most use cases of LLM applications.

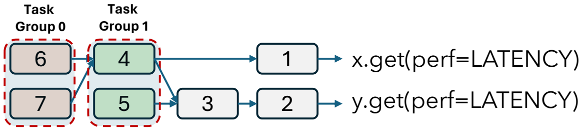


Figure 9: Performance deduction for an LLM-based application generating two latency-sensitive Semantic Variable.

5.2 Performance Objective Deduction

To optimize the end-to-end performance of applications, we need to know the application-level performance criteria. To help deriving the request-level scheduling preference from the end-to-end application's performance requirement, we need to understand the workflow of the LLM application, which is the DAG of LLM requests derived by Parrot's primitives.

When an application annotates a Semantic Variable to prefer higher throughput, all requests generating this Semantic Variable (both directly or indirectly) will be marked as throughput-preferred when scheduling. This scheduling preference is usually beneficial for offline data processing, such as bulk document analysis.

Handling latency-sensitive applications is more intricate. As demonstrated in Figure 4, achieving low end-to-end latency may sometimes require prioritizing throughput at the Mapping stage. The latency of individual requests can sacrificed so as to reduce the completion time of the entire DAG of requests. Parrot analyzes LLM requests in reverse topological order, beginning with those linked to latency-critical Semantic Variable, as depicted in Figure 9. With the extracted DAG, LLM requests that directly result in latency-critical Semantic Variables are labeled as latency-sensitive (Request 1 and 2), as are their immediate predecessors (Request 3). Parallel LLM requests at the same stage are grouped into a *task group* (Task Groups 0 and 1). The scheduler should minimize the latency of the entire task group, often leading to a higher batch capacity for higher throughput of token generation.

5.3 Sharing Prompt Prefix

When an LLM request is scheduled to an LLM engine, a context on the engine is created to store the state of the model execution for this request (mainly KV cache). Existing works have proposed to share the KV cache of common prefix of prompts in LLM engines to save the GPU memory. However, as we have explained in §3, today's public LLM service face diverse applications and requests, which is hard to identify the commonality at the cluster level. Token-by-token comparison is impractical due to high time complexity, especially for very long context with massive requests. In Parrot, by exposing Semantic Variables to LLM service, we can understand the prompt structure to automatically detect the commonality more efficiently at the granularity of Semantic Variables. Using Parrot's primitive of `PrefixHash`, Parrot only needs to check the hash value at positions after each Semantic Variable in a request's prompt. Parrot maintains a key-value store, where each entry maps a (hashed) prefix of tokens to a list of requests, thus the scheduler can quickly check the opportunity in an online manner, supporting both static and dynamically-generated prompt within one application or even across different applications.

Furthermore, we propose better GPU kernel for the attention computation of the requests with a common prefix. We first leverage vLLM's paged memory management [25] to save the redundent GPU memory. But vLLM's kernel still suffers from redundant computation and memory loading of the shared tokens. Therefore, we design a new Attention decoding algorithm by combining FlashAttenation [12] and PagedAttention [25] that treat the shared and non-shared token separately. This significantly accelerates the attention of shared contexts (implementation details in §7).

5.4 Application-Centric Scheduling

Parrot's scheduling is a problem that matches LLM requests to LLM engines, i.e. the cluster-level scheduling, while the engine-level scheduling will be covered in the implementation details of the engine in §7. To fix the problem of existing public LLM service that blindly optimize diverse individual requests, Parrot's scheduling policy leverages the application-level knowledge to optimize the end-to-end performance. Specifically, the primary goal of Parrot's scheduler is to meet the varied performance goals of LLM applications while optimizing GPU cluster utilization. As explained in §3, a conflict arises when combining throughput and latency oriented requests: large batch sizes increase throughput and GPU efficiency but degrade latency, and vice versa. Transformer-based LLM inference is largely memory-bound, with latency influenced by the count of concurrent tokens within the engine. To meet performance targets of LLM applications, particularly latency, an LLM engine must regulate the token count below a specified threshold, which is determined by the LLM request with the most strict latency constraint. Therefore, Parrot's scheduling principles are twofold: (1) group LLM requests with similar performance requirements to circumvent the conflict, and (2) maximize opportunities for sharing across requests.

Algorithm 1 outlines the scheduling process of Parrot. With the extracted DAG, the system arranges the LLM requests according to their topological order (line 1). Parrot tends to schedule requests belonging to the same application together to avoid the slowing down of interleaved scheduling (§8.2). For requests identified as part of a task group through Parrot's performance objective deduction, the scheduler attempts to allocate the entire task group together (line 4-line 5). Additionally, if Parrot detects other queued requests or running contexts with a common prefix, it tries to assign them to the same LLM engine (line 3, line 6-line 9), to utilize Parrot's context fork to reduce the redundant computation and GPU memory transactions. For an LLM request without the above opportunity, Parrot schedules the request independently (line 10-line 11). Due to limited space, we omit the details of how Parrot chooses LLM engines (i.e., `FindEngine`). Briefly, Parrot finds the engine that satisfies the scheduling preference of a request while minimizing the negative impacts. For instance, if a latency-sensitive request is scheduled to an LLM engine that can run up to 64,000 tokens of throughput-driven requests, its capacity will be significantly reduced to 2,000 to satisfy its strict latency requirement. But, if it is scheduled to an engine that has already been running a latency-sensitive request, the capacity reduction is negligible.

Algorithm 1: Parrot's Request Scheduling.

```
   Data: Q: the request queue
1  Q.sort() ; /* Topological order */
2  for $r \in Q$ do
3      SharedReqsInQueue, CtxInEngine =
         FindSharedPrefix($r$);
4      if $r.TaskGroup \neq \varnothing$ then
5          $r^*$ = FindEngine(r.TaskGroup);
6      else if $SharedReqsInQueue \neq \varnothing$ then
7          $r^*$ = FindEngine(SharedReqsInQueue);
8      else if $CtxInEngine \neq \varnothing$ then
9          $r^*$ = FindEngine(r, filter=CtxInEngine);
10     if $r^* = \varnothing$ then
11         $r^*$ = FindEngine($r$);
12     Q.remove($r^*$);
```

6 Discussion

Dynamic Applications and Function Calling. Currently, Parrot only supports cloud-side orchestration of LLM requests without involving dynamic control flow and native functions (e.g., Python Code). They still require client-side execution. We intentionally disable the offloading of these functions to public LLM services to minimize the security risks of malicious injection. For private LLM services whose LLM applications are trusted or there is a trusted zone to execute these functions, Parrot's APIs can be easily extended with conditional connections and native code submission. Moreover, these extensions further enable new optimizations, e.g., we can speculatively pre-launch high-probability branches in dynamic applications based on past profiles. This also proves the potential of Parrot's design when facing new types of applications. We leave these extensions as future works.

Other Applications of Inter-Request Analysis. The inter-request analysis in Parrot enables a new optimization space not limited to the ones we introduced in §5. A large-scale service has more scheduling features to consider, including handling outliers [3], job failures [58], delay scheduling [57], fairness [15, 61], starvation [17], or supporting heterogeneous clusters [24, 37], which have been widely studied in other systems. Parrot provides a new view from the perspective of LLM-based applications: we need to understand the interconnection and commonality of LLM requests to optimize applications' end-to-end performance. These features can be revisited in the LLM service system by considering the new characteristics of LLM applications. In this paper, we focus on Parrot's mechanisms and a few use cases, leaving other optimizations as promising future works.

Parrot with LLM Orchestration Frameworks. There have been several frameworks for developers to build LLM-based applications, e.g., LangChain [8], SemanticKernel [36], and PromptFlow [35]. The key function of these frameworks is to "glue" different LLM calls to accomplish a complex task (aka. LLM orchestration). Parrot can be integrated with these frameworks by extending their calling of LLM service APIs with Semantic Variables. Most of these frameworks have already used a template-based approach in which developers can design a template with placeholders, and render the placeholders at runtime. These placeholders naturally have the same concept as Parrot's Semantic Variable. However, because these frameworks will render the template prompt before the submission, LLM services lose the information on the prompt structure. To make these frameworks compatible with Parrot, both the template itself and the variables to render the template (using Semantic Variable in Parrot) need to be wrapped as a `SemanticFunction` so the necessary information is exposed to Parrot's LLM service.

7 Implementation

Parrot is an end-to-end LLM service for LLM applications, implemented on Python with about 14,000 lines of code. Its front-end provides the abstraction of Semantic Variable, and `SemanticFunction`, which is transformed into Parrot's APIs (implemented with FastAPI [48]) to be submitted as LLM requests. A centralized Parrot manager handles the management of LLM requests, including Semantic Variables, communication, and scheduling. We also build an LLM engine based on efficient kernels from vLLM [25], xFormers [26], and ourselves. The engine supports advanced features for LLM serving, including paged memory management [25] and continues batching [56]. Parrot's front-end and manager are implemented in 1,600 and 3,200 lines of Python, respectively. Parrot's LLM engine is implemented in 5,400 lines of Python and 1,600 lines of CUDA. We have implemented OPT [60] and LLaMA [51] with PyTorch [45] and Transformers [53].

APIs. Applications programmed by `SemanticFunctions` or other frontends are finally lowered to requests to universal APIs through different adapters. Parrot provides OpenAI-like APIs with the extension of Semantic Variables. The request body of two operations mentioned in §4.1 is shown as follows:

```
(submit) {"prompt": str, "placeholders": [{"name":
↪  str, "in_out": bool, "semantic_var_id": str,
↪  "transforms": str}, ...], "session_id": str}

(get) {"semantic_var_id": str, "criteria": str,
↪  "session_id": str}
```

In addition to the static string prompt, Parrot preserves the input and output placeholders. A placeholder is associated with a semantic variable either for rendering the input or parsing the output. As introduced in §5.1. Parrot supports transformations before the input or after the output. Parrot also supports other APIs for setting and fetching the value of Semantic Variables. The error message will be returned when fetching an Semantic Variable, whose intermediate steps fail (including engine, communication, and string transformation).

Kernel Optimization. vLLM's GPU kernel, while capable of reusing results cached in GPU memory for shared prefix tokens in a prompt, sometimes excessively reloads these tokens from global to shared memory, impeding attention score computations. Using OpenAI Triton [43] and CUDA, we have developed a novel GPU kernel, integrating concepts from PagedAttention [25] and FlashAttention [11, 12], to accelerate attention decoding computation involving shared prefixes. This kernel retains PagedAttention's approach of storing the key-value (KV) cache in disparate memory segments and utilizes a page table per request to monitor block status and placement. Furthermore, employing FlashAttention principles, the kernel maximizes data reuse within shared memory. Unlike reloading tiles repeatedly in the PagedAttention's implementation, it loads KV cache tiles for the shared prefix to shared memory only once, diminishing memory transactions between the L2 Cache and Shared Memory. The kernel initially calculates interim attention metrics (including attention scores, `qk_max`, `exp_sum`) for the shared prefix using the loaded tiles and records these back to HBM. Subsequently, it processes the new tokens' partial attention beyond the prefix, amalgamating this with the prefix's interim results to derive the ultimate attention output.

Universal Engine Abstraction. Parrot's cluster manager controls multiple engines running various models, tokenizers, KV cache layouts, etc. To enable Parrot's optimizations, LLM engines need to support (1) stateful generation (e.g., guidance [18]) and (2) sharing KV cache states across different requests. Hence we propose a universal abstraction to describe the minimal capability required to LLM engines to be integrated into Parrot.

```
def Fill(token_ids: List[int], context_id: int,
 ↪  parent_context_id: int)
def Generate(sampling_configs: Dict, context_id:
 ↪  int, parent_context_id: int)
def FreeContext(context_id: int)
```

These three methods not only cover the basic completion functionality of LLM inference engine, but also provide a flexible context management interface. The `Fill` method processes the initial prompt tokens, calculates and fills the KV cache into corresponding context. The `Generate` method produces tokens via generative decoding that produces one token per iteration until it reaches the length limit, user-defined termination character or EOS (end-of-sequence) token, under certain sampling configurations (e.g. temperature). `Fills` and `Generates` are scheduled and batched by engine's scheduler per iteration using continuous batching [56]. Creating and forking contexts can also be realized with these two methods by setting `context_id` and `parent_context_id`, respectively. The `FreeContext` method explicitly frees a context (i.e. free its KV cache in GPU memory). Separating `Fill` and `Generate` not only fits Semantic Variable naturally: constant text and input values are processed by `Fill`; the output values are generated by `Generate`, but also breaks the request-level dependency into a finer granularity, enabling more parallel execution opportunities [2, 21, 46, 64].

8 Evaluation

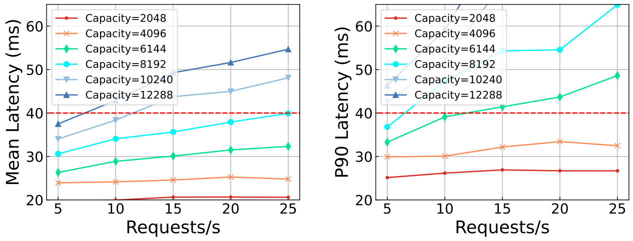
8.1 Experimental Setup

Testbed. We evaluate Parrot with two separate setups for single-GPU and multi-GPU experiments. The single-GPU evaluations use a server with a 24-core AMD-EPYC-7V13 CPUs equipped with one NVIDIA A100 (80GB) GPU. The multi-GPU evaluations use a server with 64-core EPYC AMD CPU and four NVIDIA A6000 (48GB) GPUs. Both servers run CUDA 12.1 and cuDNN 8.9.2.

Workloads. Our evaluations are performed to run four representative LLM applications. Each LLM engine uses one GPU and runs a LLaMA 13B or LLaMA 7B model [51] . For LLM-based data analytics on long documents, we use the Arxiv dataset [27], executing chain and map-reduce summarizations on an extensive collection of academic papers. To investigate the sharing opportunities of LLM-based applications with many users, we run the prompts from Bing Copilot and GPTs [42] with synthesized user queries. For multi-agent applications, we build a multi-agent programming application using MetaGPT [22], which contains a system architect to design APIs, multiple programmers to write code for different files, reviewers to share review comments. The programmers will also revise the code based on comments. For chat service workloads, we derived scenarios from the ShareGPT dataset [50], which mirrors real LLM chat conversations. According to the distribution of our measurement, we introduced a random delay of 200 $\sim$ 300 ms to LLM requests to emulate typical network overhead seen over the Internet. To create realistic workloads, we documented the LLM responses using GPT-4 [41], ensuring the LLaMA models generated text of similar length for system performance analysis. Table 2 presents the workloads and their optimizations in Parrot.

Workload	Serving Dependent Requests.	Perf. Obj. Deduction	Sharing Prompt	App-centric Scheduling
Data Analytics	✓	✓		✓
Serving Popular LLM Applications			✓	✓
Multi-agent App.	✓	✓	✓	✓
Mixed Workloads	✓	✓		✓

Table 2: The workloads and the optimizations taking effect.

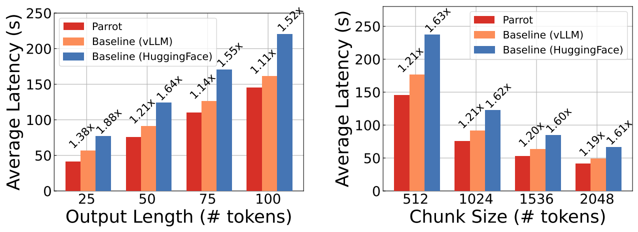


(a) Mean Latency

(b) P90 Latency

Figure 10: Latency (per output token) of vLLM with varying token capacities and request rates. Requests are sampled from ShareGPT [50] and their arrival time follows Poisson distributions.

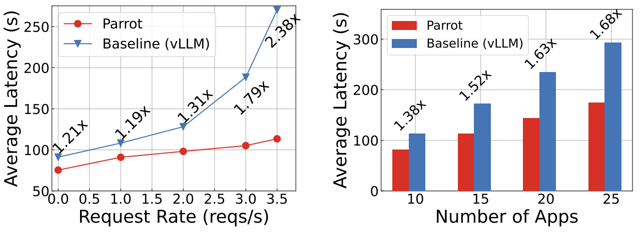
Baseline. We benchmark Parrot against sate-of-the-art solutions for building LLM applications and serving LLM requests. The majority of LLM applications used in our baseline comparisons are developed using LangChain [8], which is the predominant framework for LLM application development. The LLM applications in baselines leverage OpenAI-style chat completion APIs as provided by FastChat [62]. FastChat is a widely recognized open-source LLM serving system with over 30,000 stars on its repository. Incoming requests to FastChat are allocated to LLM engines that run either HuggingFace's Transformers library [53] or vLLM [25], both of which incorporate cutting-edge enhancements for LLM execution, such as FlashAttention [12], PagedAttention [25], and continuous batching techniques [56]. The default scheduling strategy employed by FastChat assigns incoming requests to the LLM engine with the smallest current queue. Since existing LLM services typically expose their functionality through "chat" completion APIs, baseline assessments treat all requests as independent and assume a high sensitivity to latency. To manage token generation response times, each LLM engine is subject to a capacity threshold, which is the



(a) Output lengths

(b) Chunk sizes

Figure 11: Average E2E latency of chain summarization with varying output lengths and chunk sizes.



(a) With background requests

(b) Multiple summary apps.

Figure 12: Average E2E latency of chain-summary with background requests or other chain-summary applications.

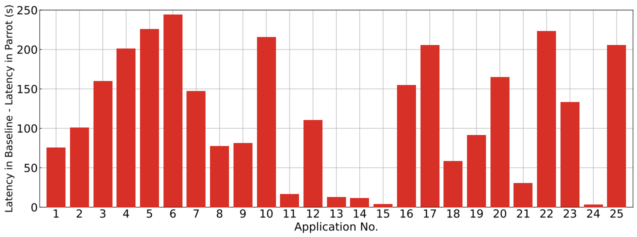


Figure 13: The difference in E2E latency of the 25 chain-summary application between Baseline and Parrot. All applications finish earlier in Parrot.

aggregate token count from all active requests on the engine.

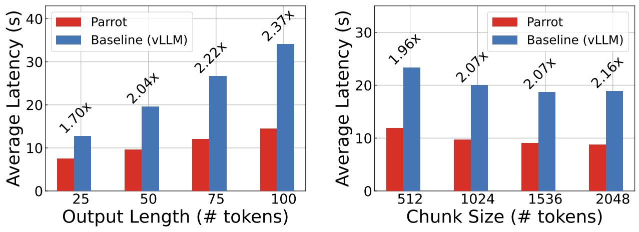
Since existing LLM token generation is usually bound by memory bandwidth, the per-token generation latency of an engine is mainly affected by the number of running tokens in a batch. As depicted in Figure 10, our experiments indicate that the latency per output token, i.e. TPOT (Time-per-output-token) for vLLM, with continuous batching enabled, experiences a notable uptick when the engine's workload using a batch capacity beyond 6144. In our evaluation, we use the setting that an LLM engine can keep its generation latency under 40 ms/s for latency-sensitive requests, consistent with our experience of OpenAI's LLM services. When all LLM engines hit their maximum capacity, any additional LLM requests are queued in a FIFO (First In, First Out) manner, awaiting the completion and release of resources by ongoing tasks. Serving longer context (e.g., 32k or even 1M tokens) within a satisfactory latency require either more GPUs using tensor-parallel [49] or sequence-parallel [6] approaches, or approximate attention (e.g., StreamingLLM [55]), which is beyond the scope of this paper.

8.2 Data Analytics on Long Documents

Our experimental analysis within data analytics randomly picks ten long documents from the Arxiv-March dataset [27], using chain-summary and map-reduce summary. Each document has over 20,000 tokens. The results measures the mean end-to-end latency across all documents.

Chain-style Applications. Our evaluation demonstrates how Parrot enhances chain summarization by mitigating the excessive communication overhead stemming from client interactions. Figure 11 presents the average end-to-end latency for summarizing a single document using one LLM engine (A100, LLaMA 13B) . We adjust the chunk size (the count of tokens per chunk) and the output length, with results shown in Figure 11a and Figure 11b, respectively. Parrot achieves a reduction in end-to-end latency by as much as 1.38× and 1.88× compared to the baselines employing vLLM and HuggingFace, respectively. The efficiency of Parrot primarily stems from the decreased network latency, which is a consequence of reduced client interaction. As the output length increases, the time spent on generation becomes more significant, leading to a diminishing advantage for Parrot over the baseline. By increasing the chunk size, we decrease the number of chunks, yet the extent of the speedup is contingent upon the network latency savings for each chunk. Given that token generation is substantially more time-consuming than prompt processing, we observe a consistent speedup with variable chunk sizes and a fixed output length (1.2× and 1.66× relative to vLLM and HuggingFace, respectively). This indicates that Parrot's optimization for dependent LLM requests is particularly beneficial for shorter outputs, which are prevalent in various LLM applications such as summarization, short answer generation, scoring, and choice provision. Due to HuggingFace's slower performance relative to vLLM, subsequent evaluations focus solely on the comparison between Parrot and vLLM.

Figure 12a extends the evaluation by introducing background LLM requests at varying rates to examine the capability of Parrot in mitigating additional queuing delays for dependent requests. Parrot slashes the end-to-end latency by a factor of 2.38× in comparison to the baseline (vLLM). With Parrot, as soon as the summary for the first chunk is completed, the subsequent chunk is processed immediately by incorporating the summaries of previous chunks into the prompt, which aids in generating the summary for the next chunk. In contrast, the baseline treats all LLM requests individually. As a result, in addition to the network latency from client interactions, subsequent requests must re-enter the queue, leading



(a) Output lengths

(b) Chunk sizes

Figure 14: Average E2E latency of Map-Reduce document summary with varying output lengths and chunk sizes.

to added queuing delays. Figure 12b further illustrates the end-to-end latency when multiple chain-summary applications are submitted concurrently, with each application tasked with generating a summary for a separate document. Parrot manages to reduce the average end-to-end latency for all applications by 1.68× without slowing down any applications compared to the baseline according to Figure 13. The baseline, by interleaving the execution of different applications, exacerbates the slowdown of the end-to-end latency for all applications. These experiments validate that recognizing the interconnections of LLM requests can significantly enhance end-to-end performance, as opposed to processing requests in isolation.

Map-Reduce Applications. An alternative implementation of the document summarization application follows the map-reduce paradigm as depicted in Figure 1a. This approach consists of multiple parallel mapping LLM requests, where each request summarizes a distinct segment of the document, followed by a reducing LLM request that aggregates these individual summaries into a final summary. As shown in Figure 14, Parrot realizes a 2.37× acceleration over the baseline with one LLM engine (A100, LLaMA 13B). Since the mapping LLM requests are independent, they are dispatched concurrently by both Parrot and the baseline. The primary advantage of Parrot stems from its deduction of a performance objective that identifies the mapping tasks as a task group. By recognizing this relationship, Parrot is capable of optimizing the latency of the entire task group through larger batch sizes, which in turn enhances throughput. In contrast, the baseline processes each LLM request in isolation, operating under the presumption that they are all sensitive to latency. This constrains the baseline to utilize a limited token capacity (4096 tokens) on the LLM engine to achieve optimal latency for individual tasks, which is detrimental to the end-to-end performance of applications. It underscores the necessity for LLM services to distinguish LLM requests to optimize the end-to-end performance of varied LLM applications.

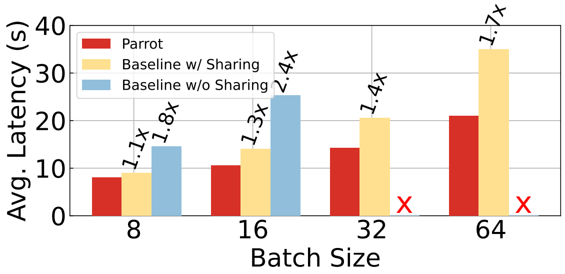
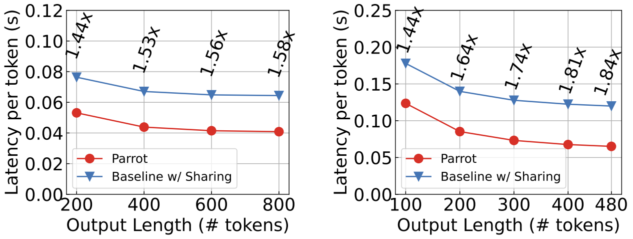


Figure 15: Latency of Bing Copilot with varying batch sizes.

8.3 Serving Popular LLM Applications

Production applications need to face massive users. As explained in Figure 5, developers often need to use a very long system prompt to define the behavior of LLMs. Therefore, users of the same LLM application often use the shared prompt, which can benefit from Parrot's context fork mechanism and Parrot's scheduling policy that co-locates LLM requests sharing a long prompt prefix. Because we do not have access to the intermediate steps of Bing Copilot, we only evaluate the final request generating the response to users. We synthesized 64 requests from the length distribution we measured using Bing Copilot. The system prompt length is about 6000 tokens. The output lengths ranges from 180 to 800 tokens. Figure 15 shows the average request latency of Bing Copilot of Parrot and the baselines. Because the LLM service in the baseline system does not know the prompt structure, it is hard to infer the shared prompt from massive LLM requests. Compared to the baseline without sharing prompt, Parrot achieves 1.8× $\sim$ 2.4× speedup for batch sizes of 8 and 16. Further increasing the batch size leads to out-of-memory due to the massive KV cache of shared system prompt. We also build an advanced baseline using vLLM's paged attention to support sharing the prompt with a static prefix. Both Parrot and vLLM use the paged memory management [25], thus both systems can hold the same number of tokens in an LLM engine (A100, LLaMA 7B). Parrot further achieves 1.1× $\sim$ 1.7× speedup over vLLM because of the better GPU kernel. Although vLLM can save extra memory usage of the shared prompt, its GPU kernel still has to reload the tokens repeatedly. Given that the token generation of LLMs is bound by memory bandwidth, such redundant memory loading slows down the end-to-end inference. By combining FlashAttention and PagedAttention, Parrot only needs to load the tokens of the shared prompt once, when computing the attention from the diverged tokens of different users. Parrot's speedup of shared prompt mainly comes from the token generation, thus the longer output length leads to higher improvement. Figure 16 shows Parrot achieves 1.58× and 1.84× speedup compared to vLLM using paged attention, showing 40 *ms* per-output-token latency at a batch size of 32.

In Figure 17, we further evaluated the serving of multiple GPTs applications [42], each of which has multiple users, in



(a) Batch Size = 32

(b) Batch Size = 64

Figure 16: Latency per output token of Bing Copilot.

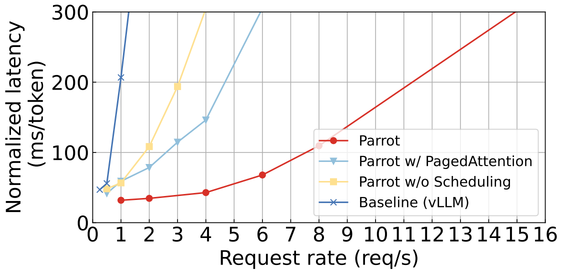
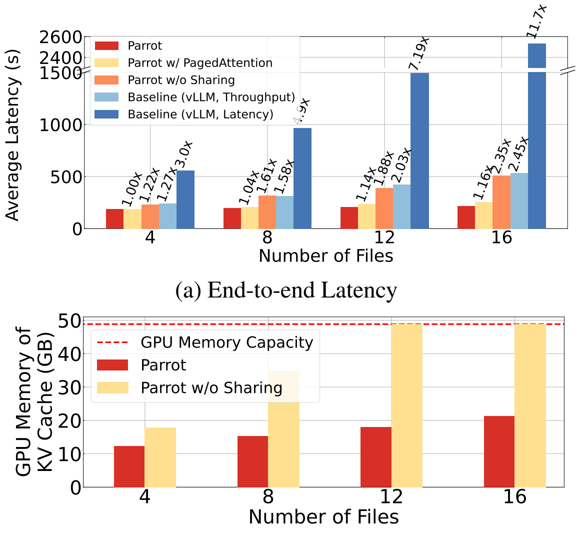


Figure 17: Serving multiple GPTs applications.

a multi-GPU cluster. Four A6000 (48GB) GPUs are deployed with four LLM engines (LLaMA 7B). We select four GPTs applications in four popular categories including productivity, programming, image generation, and data analysis. The LLM requests are randomly generated from the four categories with equal probability. LLM requests arrive at fixed rates following Poisson distribution. Parrot can sustain 12× higher request rates compared to the baseline without sharing. Because the baseline's scheduling policy is not aware of the shared prompt within each LLM application, the requests are mixed in all LLM engines making it impossible to reuse the common prompt prefix. Parrot's scheduling policy co-locates LLM requests of the same applications to maximize the sharing opportunity, achieving both lower inference latency and higher cluster throughput. After turning off such affinity scheduling policy, Parrot only exhibits 3× higher request rates compared to the baseline, because the requests with shared prefix are often dispatched to different engines thus reduced the sharing opportunities. Moreover, Parrot's attention kernel helps Parrot to achieve 2.4× higher rate compared to Parrot using vLLM's PagedAttention, by avoiding the redundant memory loading for attention of shared prompts.

8.4 Multi-agent Applications

We assess the performance of multi-agent systems utilizing MetaGPT [22] within Parrot. A workflow is constructed with three distinct roles. Initially, the Architect outlines the project's file structures and specifies APIs within each file for a given task. Subsequently, multiple Coders undertake the



(a) End-to-end Latency

(b) GPU Memory of KV Cache

Figure 18: The latency and memory usage for multi-agent programming, with varying number of files to program.

project implementation, with each focusing on a specific file. Following the integration of the code from all files, several Reviewers engage in the process, each examining and commenting on a single file. The Coders then revise their code based on these comments. This review-and-revision cycle is iterated three times to produce the final code. Figure 18 illustrates the latency and memory consumption of Parrot compared to baseline systems on one A100 running LLaMA 13B. Parrot achieves a speedup of up to 11.7× compared with the latency-centric baseline. The primary improvement is attributed to Parrot's capability to deduct the performance objectives for LLM requests based on the end-to-end performance criteria. For this specific multi-agent scenario, the goal is to minimize the time taken to deliver the final code. Parrot identifies multiple task groups within the parallel processes of coding, reviewing, and revising, facilitating larger batch sizes to enhance throughput and reduce the completion time of task groups. We also contrast Parrot with an throughput-centric baseline that uses larger batch on purpose to optimize cluster throughput, which also shows higher concurrency and better completion time than the latency-centric baseline.

Even when compared to the throughput-centric baseline, Parrot demonstrates superiority, being faster by up to 2.45×. This enhancement mainly stems from Parrot's ability to decrease redundancy through its prompt structure analysis, which contributes a 2.35× acceleration. Given the interactive nature of the roles in MetaGPT, there is considerable overlap in the context among different roles, which Parrot capitalizes on by sharing this common context as a prompt prefix. The static prefix sharing mechanism from vLLM does not work in this dynamic scenario. Without a grasp of the prompt's structure, it cannot identify dynamically generated Semantic

Variables that could also be shared during runtime. As depicted in Figure 18b, Parrot without this sharing capability would hit the GPU memory ceiling. Additionally, Parrot's specialized GPU kernel for processing the shared prefix achieves a further 1.2× speedup when there are 16 files, compared to using vLLM's PagedAttention, due to the reduced memory transactions.

8.5 Scheduling of Mixed Workloads

To assess the performance of Parrot on a multi-GPU setup, we configure a cluster with four A6000 (48GB) GPUs, each hosting a separate LLM engine (LLaMA 7B), resulting in a total of four LLM engines. We emulate a real-world scenario where LLM services encounter a variety of demands by injecting a mix of requests from chat applications at a rate of 1 req/s and from data analytic tasks (i.e., map-reduce applications) previously analyzed in §8.2. Requests from the chat applications are characterized by their need for low latency, whereas the map-reduce applications prioritize high throughput, creating a challenge when they are concurrently processed by the same LLM engine. We benchmark Parrot against two reference implementations: one tailored for latency, limiting engine capacity to reduce decoding time, and another for throughput, utilizing full engine capacity to maximize GPU utilization.

The results depicted in Figure 19 demonstrate that Parrot attains a 5.5× and 1.23× improvement in normalized latency (measured as request latency per number of output tokens) [25, 56] for chat applications in comparison to the latency-focused and throughput-focused baselines, respectively. In terms of token generation speed for chat applications, Parrot delivers performance on par with the latency-centric baseline and outperforms the throughput-centric baseline by 1.72×. For map-reduce applications, Parrot reaches a 3.7× speedup over the latency-centric baseline and is 1.05× more efficient than the throughput-centric baseline. Parrot excels by providing both low latency for chat applications and high throughput for map-reduce applications. It mitigates the contention between chat and map-reduce workloads by intelligently scheduling them on separate engines. These findings underscore the significance of specialized handling for diverse requests to enhance the overall performance of LLM services.

9 Related Works

Deep Learning Serving Systems. The field of model serving has seen a surge of research activity in recent years, with many systems developed to address the different challenges of deep learning model deployment. The systems include Clipper [10], TensorFlow Serving [39], Clockwork [19], REEF [20], AlpaServe [28], which have explored many aspects including batching, caching, placement, scheduling, model parallelism for the serving of single or multiple models.

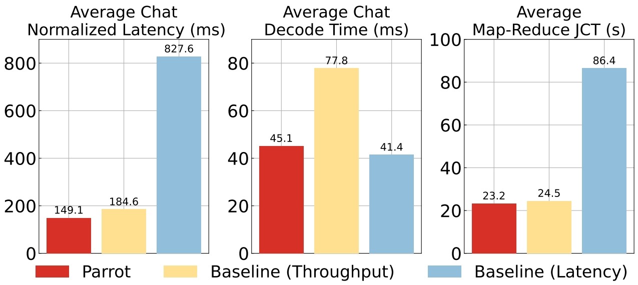


Figure 19: The mixture of chat and map-reduce applications.

These systems were proposed for serving general deep learning models, which have less consideration about the unique requirements of large language models, e.g., autoregressive decoding. Orca [56] proposed a fine-grained scheduling mechanism that can batch multiple LLM requests at the iteration level, which is also known as continuous batching. vLLM proposes PagedAttention [25] allows the batching of LLM requests with different lengths using non-contiguous memory, increasing memory utilization. These systems for LLM serving still treat LLM requests separately, missing the opportunities to understand the interconnections within an application and exploit the commonality of different requests. Parrot is orthogonal to them. With more application-level knowledge exposed by Semantic Variables, Parrot can do data flow analysis on LLM requests, which enables a brand new optimization space with the final goal of optimizing the end-to-end performance of applications, rather than individual requests.

LLM Orchestrator Frameworks. LLM orchestration frameworks help developers create and manage applications powered by LLMs. They simplify the process of prompt design, and orchestration of multiple LLM requests, which enable developers to interact with LLMs easily. LangChain [8] is a Python framework that provides many workflow patterns, e.g., chain, map-reduce so that developers can easily customize their own LLM applications. Semantic Kernel [36] introduces Planners are semantic agents that can automatically generate plans based on the needs of the users. PromptFlow [35] supports chains of native and semantic functions and visualizes them as a graph. LlamaIndex [29] allows developers to use natural language queries to retrieve relevant documents. Parrot is orthogonal to these frameworks and can be easily integrated with these frameworks to support Parrot's APIs with Semantic Variable abstraction, as discussed in §6.

DAG-aware System Optimizations. Dependency graphs or DAGs (Directed Acyclic Graphs) widely exist in many kinds of systems, and many optimizations have been proposed to optimize the systems by exploiting the DAG information. Tez [4], Dryad [23], and Graphene [16] use the task dependency to optimize the scheduling and packing of parallel data analytic workloads. SONIC [30], Caerus [59], and Orion [31] optimize serverless functions from the aspects of communication, latency, and cost. Parrot learns from the previous system works and realizes the importance of correlations of LLM requests to optimize the end-to-end performance of LLM applications. This motivates Parrot to build APIs for exposing such dependency information. Moreover, it is unique to LLM applications to understand the prompt structure in addition to request-level dependency, which is necessary for communication and identifying commonality across LLM requests. This motivates us to propose the Semantic Variable abstraction, instead of just using a DAG of requests.

10 Conclusion

This paper proposes Parrot that treats LLM applications as first-class citizens and targets to optimize the end-to-end performance of LLM applications, instead of only optimizing individual LLM requests. We propose Semantic Variable as the key abstraction that exposes the dependency and commonality of LLM requests, enabling a new optimization space. Our evaluation shows Parrot can optimize LLM-based applications by up to 11.7×. We envision this new angle of efficiency improvement of LLM applications brings a broad future direction to study other scheduling features like the fairness of *end-to-end* performance of LLM applications.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their constructive feedback and suggestions. Zhenhua Han, Yuqing Yang and Chen Chen are the corresponding authors.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310*, 2024.

[3] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in Map-Reduce clusters using mantri. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[4] Apache. Tez. https://tez.apache.org/, November 2019.

[5] Apache. Kafka. https://kafka.apache.org/, October 2023.

[6] Zhengda Bian, Hongxin Liu, Boxiang Wang, Haichen Huang, Yongbin Li, Chuanrui Wang, Fan Cui, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. *CoRR*, abs/2110.14883, 2021.

[7] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023.

[8] Harrison Chase. LangChain. https://github.com/langchain-ai/langchain, October 2022.

[9] Lequn Chen. Dissecting batching effects in gpt inference. https://le.qun.ch/en/blog/2023/05/13/transformer-batching/, May 2023.

[10] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[11] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[12] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022.

[13] Bill Gates. Ai is about to completely change how you use computers and upend the software industry. https://www.gatesnotes.com/AI-agents, Nov 2023.

[14] Google. Google bard. https://bard.google.com/, Nov 2023.

[15] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in Multi-Resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, November 2016. USENIX Association.

[16] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and Dependency-Aware scheduling for Data-Parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, Savannah, GA, November 2016. USENIX Association.

[17] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

[18] guidance ai. Guidance. https://github.com/guidance-ai/guidance, November 2023.

[19] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[20] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[21] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[22] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[23] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 59–72, New York, NY, USA, 2007. Association for Computing Machinery.

[24] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 642–657, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, and Daniel Haziza. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

[27] Yucheng Li. Unlocking context constraints of llms: Enhancing context efficiency of llms with self-information-based content filtering, 2023.

[28] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[29] Jerry Liu. LlamaIndex, November 2022.

[30] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[31] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 22), pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[32] Microsoft. Bing chat. https://www.bing.com/chat, Nov 2023.

[33] Microsoft. Meeting recap in microsoft teams. https://www.microsoft.com/en-us/microsoft-teams/premium, May 2023.

[34] Microsoft. Microsoft 365 copilot. https://www.microsoft.com/en-us/microsoft-365/enterprise/microsoft-365-copilot, Mar 2023.

[35] Microsoft. PromptFlow. https://github.com/microsoft/promptflow, November 2023.

[36] Microsoft. Semantic Kernel. https://github.com/microsoft/semantic-kernel, November 2023.

[37] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[38] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[39] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[40] OpenAI. Chatgpt. https://chat.openai.com/, Nov 2023.

[41] OpenAI. Gpt-4 technical report, 2023.

[42] OpenAI. Introducing gpts. https://openai.com/blog/introducing-gpts, Nov 2023.

[43] OpenAI. OpenAI Triton. https://github.com/openai/triton, November 2023.

[44] OpenAI. Production best practices - openai api. https://platform.openai.com/docs/guides/production-best-practices/improving-latencies, Nov 2023.

[45] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[46] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

[47] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

[48] Sebastián Ramírez. FastAPI. https://github.com/tiangolo/fastapi.

[49] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[50] ShareGPT Team. Sharegpt dataset. https://sharegpt.com/, Nov 2023.

[51] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[52] Unknown. Prompt of bing chat. https://www.make-safe-ai.com/is-bing-chat-safe/Prompts_Conversations.txt, Nov 2023.

[53] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. pages 38–45. Association for Computational Linguistics, October 2020.

[54] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

[55] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv*, 2023.

[56] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[57] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 265–278, New York, NY, USA, 2010. Association for Computing Machinery.

[58] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, 2008.

[59] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 653–669. USENIX Association, April 2021.

[60] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[61] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532. USENIX Association, November 2020.

[62] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[63] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang, 2023.

[64] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.