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

Oskar Shiomi Jensen (oskarhs), Kevin Sun (krsun), Connor Wilkinson (wilkincr)

Problem and Motivation

An ever increasing number of applications use multiple LLM requests in their complex workflows. Map-reduce summary, chain summary, LLM-powered search, and multi-agent coding are 4 of the most popular LMM-based applications that require multiple LLM requests.

Currently, LLM services, such as OpenAI’s GPT, use an oversimplified, request-centric API design that loses crucial application-level information. Due to this design, 30-50% on average and 70% in the worst case of the latency of the API call happens outside of the LLM engine. This latency is caused by the network latency between services, the queuing delays of consecutive requests, and the unnecessary client-side executions in chatty interaction patterns with the API design. There are three main problems with the request-centric API design: excessive overhead of consecutive requests, misaligned scheduling objectives, and redundant computations.

Related Works

Deep Learning Serving Systems exist for more general deep learning models like Clipper and TensorFlow Serving, but do not have much consideration about the requirements of large language models. Orca employs continuous batching which is a fine-grained scheduling mechanism that batches multiple requests at the iteration level. Paged attention batches requests with different lengths using non-contiguous memory, increasing memory utilization. These techniques however do not consider the connections between requests at the application level and do not exploit the commonality of these requests.

LLM Orchestrator Frameworks simplify the process of prompt design, and orchestration of multiple LLM requests. For example LangChain allows developers to easily customize their own LLM applications with workflow templates like map-reduce or chaining. This is orthogonal to Parrot and can be integrated with Parrot’s API.

DAG-aware System Optimizations are used in tasks like scheduling and packing of parallel data analytic workloads. Systems are optimized by using DAGs to exploit dependencies between objects. Like these systems, Parrot exploits the dependency of LLM requests to optimize the end-to-end performance of applications.

Solution Overview

There are three parts to the Parrot system. The front-end, the Parrot Manager, and the Parrot LLM engine. The front-end makes LLM requests to the Parrot Manager using the Parrot API, which introduces Semantic Variables. Semantic Variables act as placeholders for both input and output in prompts. This creates a data pipeline when connecting multiple LLM requests. The Variables also enables data flow

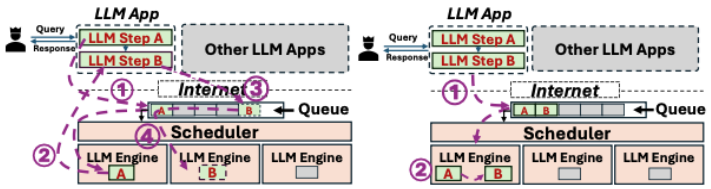
analysis to find correlations across multiple LLM requests. The asynchronous API design can also be annotated with performance criteria like latency.

```python
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)
```



(b) Current LLM Services (c) Our system: Parrot

The Parrot Manager does two inter-request analysis: DAG-based and prompt structure-based analysis. DAG-based Analysis treats requests and Semantic Variables as nodes and edges in a dependency graph. Primitives, like GetProducer and GetConsumer, are used to understand the relationships between requests.

Using these two analysis, there are four key optimizations that the Parrot Manager employs:

1. Using a graph-based executor, Parrot efficiently serves dependent requests. The executor polls the request queue to send dependent requests to corresponding engines to maximize batching opportunities.
2. Parrot manages performance objectives (i.e. throughput or latency) using the semantic variable annotation and the DAG of the LLM requests derived from the Parrot Primitives. The manager analyzes requests in reverse topological order, identifying task groups and propagating criteria, which determines the batch sizes, resource allocation, and balance of latency and throughput.
3. Parrot uses the Semantic Variables to automatically detect the commonality of prompts more efficiently. Parrot hashes at positions after each Semantic Variable. Parrot maintains a key-value store of the hashed prefix of tokens to a list of requests.
4. Application-centric scheduling matches requests to LLM engines at a cluster level. It tends to schedule requests belonging to the same application together to avoid the slowing down of interleaved scheduling. It tries to group requests with the same performance objective together. It also tries to assign requests with common prefixes to the same engine to reduce redundant computation and GPU memory transactions.

By taking advantage of these four optimizations, Parrot sees significant improvement in different workloads, achieving up to a 11.7x speedup for multi-agent applications, 2.37x speedup for map-reduce document processing, and 12x higher request rates for serving multiple GPT applications compared to baseline systems.

Limitations

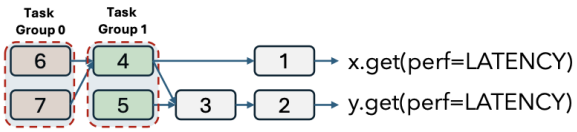
One limitation of this system is the added complexity to existing LLM serving infrastructures. Developers would need to add structure to the requests they call. Another limitation is the possibility of starvation due to unfair resource allocation with clustering.

Future Research Directions

Parrot only supports cloud-side orchestration of LLM requests without involving dynamic control flow and native functions. This means all conditional logic must run on the client side and runtime decisions about LLM calls are not possible. Parrot's API can be extended to include conditional connections and native code submission which can enable more optimizations such as speculative branching.

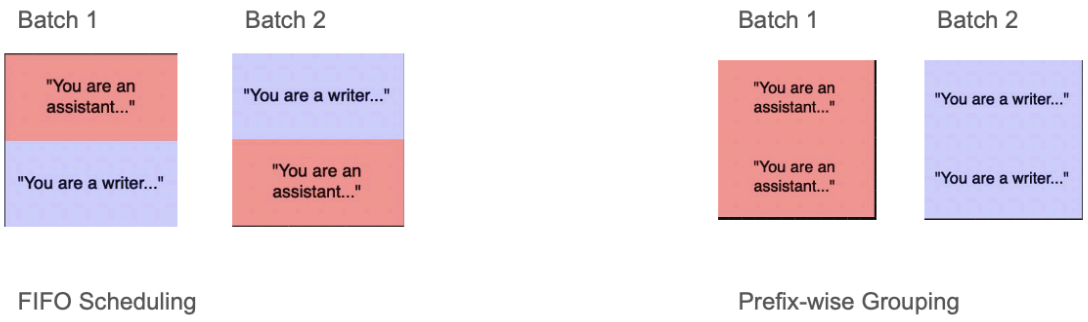
Scheduling features like handling outliers, job failures, delay scheduling, fairness, starvation, or heterogeneous clusters need to be studied in regard to the unique characteristics of LLM services and applications.

Summary of Class Discussion



Seems like they assume that each request takes the same amount of time/latency in this graph.
No this is only an image of the request dependencies, and does not represent the amount of time each request takes. This just shows the topological graph of the requests, which shows the dependency of the requests.

How do they actually decide the task group – the paper says the same stage goes to the same task group. What is the same stage? like a map reduce stage? Or just that they have to be on the same topological level
They sort by reverse topological level. They reverse the graph of dependencies and then conduct topological sort to see which tasks are in the same stage.



In this example are these 4 separate applications being scheduled or only 2 applications?

These are 4 separate applications but grouped into 2 by prefix similarity.

Why is Parrot with PagedAttention slower?
Page attention kernel loads the KV cache tile repeatedly whereas Parrot does it only once.

The coarse-grained, per-application scheduling seems to pose new challenges. There could be scalability challenges when you actually deploy to a production environment. What do you think?
The authors showed that latency was reduced overall when used to schedule on the overall application level.

Summary of “Automatic Root Cause Analysis via Large Language Models for Cloud Incidents”

Oskar Shiomi Jensen (oskarhs), Kevin Sun (krsun), Connor Wilkinson (wilkincr)

Problem and Motivation

Cloud computing infrastructure is a cornerstone of many applications, but maintaining the reliability of large and increasingly complicated cloud systems is a challenge. A typical incident life cycle follows detection, triaging, diagnosis and mitigation. To understand how and why the incident occurred, on-call engineers (OCEs) do root cause analysis (RCA). First, OCEs gather incident related data from sources like logs, traces and alerts. Then they analyze the data to identify patterns, anomalies, and correlations. Finally, OCEs use the analysis to hypothesize and verify a root cause. This manual collection of data creates a situation called the information spectrum, where OCEs either lack sufficient information for accurate diagnosis or are overwhelmed by excessive data. Troubleshooting guides (TSGs) exist to help OCEs, although many tend to fail to address new types of incidents and keep with system evolution.

In RCAs there are some key observations that motivate RCACopilot. First, determining the root cause based on a single data source can be challenging. Second, incidents stemming from similar or identical root causes often recur within a short period; 93.8% of recurring incidents reappear within a brief span of 20 days. Third, incidents with new root causes occur frequently and pose a greater challenge to analyze; 24.9% of incidents had a new root cause. With these insights, RCACopilot aims to efficiently infer, categorize and explain the root causes, reducing the time OCEs take to mitigate issues.

Related Works

Previous works of root cause analysis include using various data sources like logs and traces to conduct analysis techniques like extracting failure patterns, constructing service dependency graphs, or using traces to locate faulty services.

Large language models have already been used for things like automatic bug fixing and generation of assert statements. One previous paper fine-tuned GPT models to recommend root causes and mitigation steps to facilitate cloud incident management. Unlike this previous paper, RCACopilot uses LLMs to summarize diagnosis data and leverage chain-of-thoughts ability to predict and explain root causes.

Solution Overview

RCACopilot is an on-call system that automates RCA for cloud incidents. It has two stages after an incident is detected. The first stage is the diagnostic information collection stage, which uses incident handlers. An incident handler is an automated workflow designed to collect and process diagnostics information for a specific type of alert. It is similar to a decision tree where the root root node is the incident alert type and the branches are a series of connected actions. There are three important actions in this phase

1. Scope switching action: adjusts the data collection scope and tries to gather information relevant to the system components.
2. Query action: queries data from different sources and output results as key-value pair tables.
3. Mitigation action: suggests steps to resolve the incident and provide intermediate diagnostic information.

The second stage of RCACopilot is the root cause prediction stage. The system first uses an LLM to summarize the extensive diagnostic data into a concise summary of 120-140 words to improve readability and reduce noise. Then the system employs a FastText embedding model to map incidents into a semantic space and uses a similarity function that considers both semantic and temporal distances to find relevant historical incidents. With similar incidents, the system uses chain-of-thought prompting with the LLM to predict the incident's root cause category and generate explanations. This process allows RCACopilot to effectively handle both recurring and new types of incidents while providing clear explanations for its predictions. The system achieves RCA accuracy up to 0.76 and is used by 30+ teams at Microsoft.

Limitations

The quality of the predictions relies heavily on the accuracy of human-provided labels in the dataset and the system requires that monitors are put in place to detect incidents when they occur. This system requires significant initial effort to set up, especially when creating the incident handlers and continued effort to update the handlers to reflect the cloud systems continuous evolution.

Future Research Directions

There is future work to be done in possibly finding a more stable and consistent LLM model for this task rather than relying on OpenAI's GPT models. This system also needs to be validated on services other than Transport, which is Microsoft's global email service system. The incident handlers are another area of future work, in which handlers can be more automated and find ways to mitigate a system's lack of incident information.

Summary of Class Discussion

Are there non-AI/LLM based automation tools that are currently being used for root cause analysis?
There are data analysis workflows but the engineer has to combine the data streams. This can automate the data streams and reason with all the data.

Isn't the collection stage the area where LLM automation should be done? To get rid of engineers putting hours into it or making mistakes? Why don't they feed all of the diagnostic data to the LLM?
This is a major limitation of the paper. Hopefully they will implement this in the future. Besides less input tokens, it is actually more effective to feed summarized data than the full data to the LLM.

The summary example contains information about the total UDP sockets used by the process. The paper also says the summary should be 120-140 words. What if the summary doesn't contain important information such as the number of UDP sockets used?
This is a possibility. This was just a number that they thought was good at summarizing most incidents.

Is there a cost concern for this summary size?
I think the main purpose was to condense the content for the LLM to infer on. It did not talk about cost cutting.

Will the system answer the incident prompt with just one of the K nearest neighbors? Or does it expand on that and give more in its answer to the On Call engineer?
The system will only respond with the most similar incident response and does not expand on it.

Prof. Mosharaf comment: This seems better than I expected (throw everything into GPT4 and get a response). It has a better principle of building an actual application that approaches the problem step by step with AI as an assistant. However, the last step of giving GPT4 options ABCD and asking it to choose is still worrisome. If GPT4 hallucinates at that stage or any stage earlier (like summarization), it could still be harmful to on-call engineers trying to fix the problem.

The system is using two LLMs, one for summarization and prediction LLM? Why not just feed the unsummarized info into the prediction?
Empirically the summarization did marginally better than the unsummarized version. Also you will have to store everything for future incidents, so a summarized version is less costly.

Would it be better to determine which parts of the logs are more important and extract those rather than throw everything into the summary?
That would be a good approach to consider.

Prof. Mosharaf Comment: Engineers have tried a lot of different configurations and approaches when tackling a problem. Sometimes the dumb implementation works better. (Such as throwing everything into an LLM and getting a solution).

What does trained mean in this context? (slide 41)
Bottom 6 is fine-tuning, the rest are training from scratch. It may be training the embedding model rather than the GPT itself.

What do the authors do about situations where there is a brand new root cause that has not been seen before?
RCACopilot is not able to handle new root causes. However, it can handle new problems, which can still stem from the same root cause.

How do you add a new problem back into the system as historical data?
Add to vector DB after a successful diagnosis and then this situation can be used in the future?

Does the paper talk about evaluation? Such as whether the RCACopilot output was actually correct? Or are all outputs put back into the history pool to be used in the future.
There is a step where humans check the work.

How can RCA Copilot look for other incidents with the same root cause when the root cause is what you want to find?
K-nearest neighbors are looking for other incidents with similar diagnostic information and have occurred closely in time. These incidents' root causes are the most likely to occur and GPT4 will select the best option among them.

Prof. Mosharaf Comment: Maybe it would be better if RCA Copilot got rid of the last step of making the GPT choose the possible root cause options (ABCD etc.) It should just give a human the possible options and let the human choose.
Presenters response: depending on how RCA Copilot is built, it could be giving the users both a final decision for root cause as well as the other options it was deciding between.

The main evaluation metric is F1 score but F1 score is usually used for binary classification. Where is the binary classification being done here?
You see the different root causes as your choices and the binary classification is a yes or no on whether this is the root cause.