KVCache Cache (KV$$) in the Wild:

Characterizing and *Optimizing* KVCache Cache at a Large Cloud Provider

Jiahao Wang*[1], Jinbo Han*[1], Xingda Wei[1], Sijie Shen[2], **Dingyan Zhang[1]**, Chenguang Fang[2], Rong Chen[1], Wenyuan Yu[2], and Haibo Chen[1]

(* Equal contribution)

IPADS, Shanghai Jiao Tong University[1] && Alibaba Group[2]





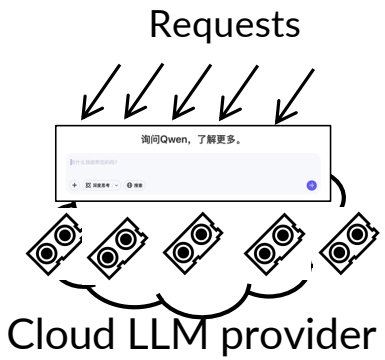


Background: Caching KV$ in LLM serving

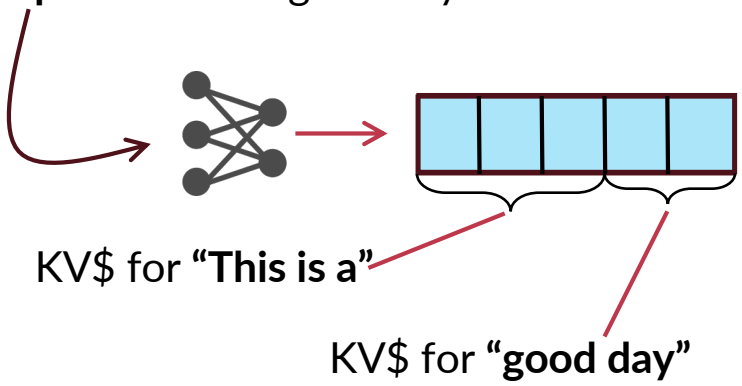
KVCache (KV$) cache and reuse intermediate result among generation steps

KV$ cache (KV$$) cache the KV$ of processed requests

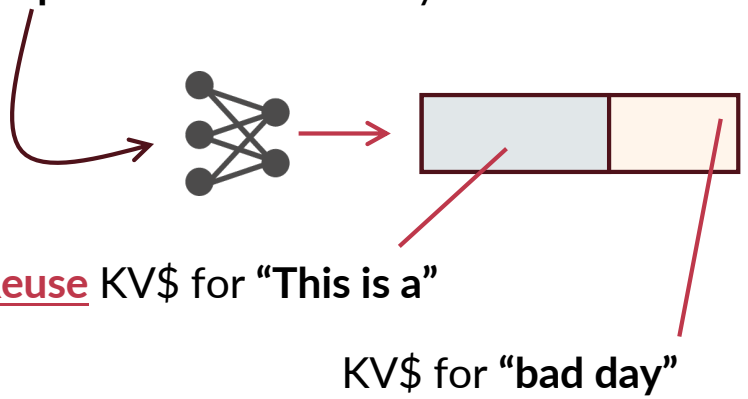
- Avoid re-computation when cached & hit => Save GPU resources !



Req#1: "This is a good day"



Req#2: "This is a bad day"

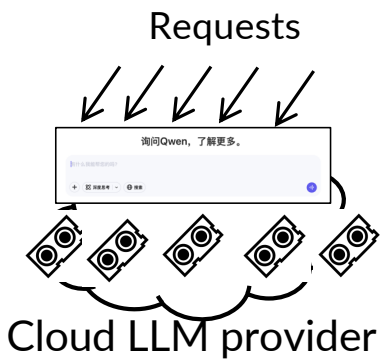


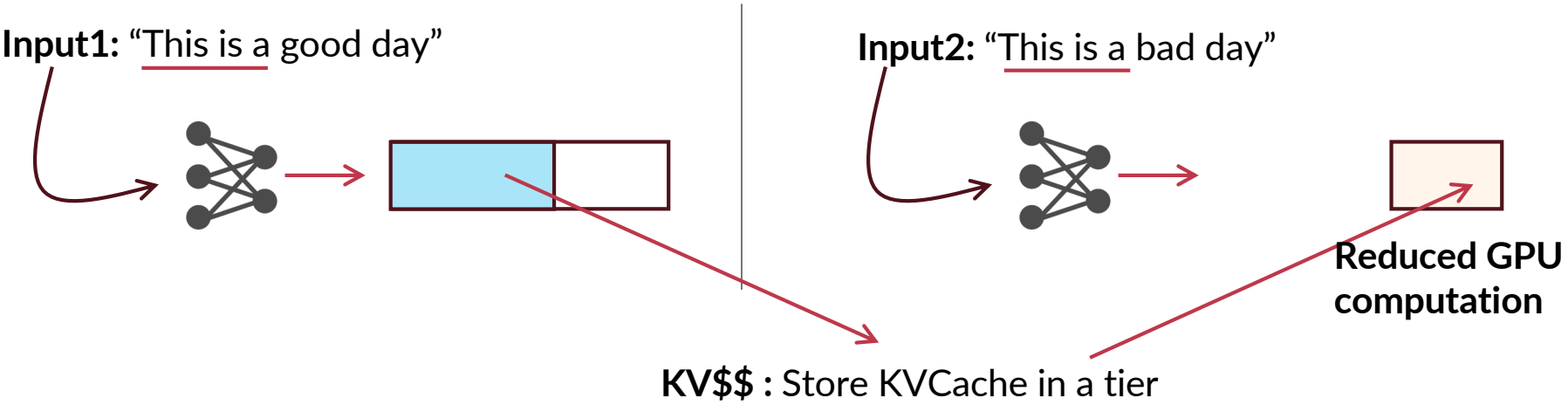
Background: Caching KV$ in LLM serving

KVCache (KV$) cache and reuse intermediate result among generation steps

KV$ cache (KV$$) cache the KV$ of processed requests

- Avoid re-computation when cached & hit => Save GPU resources !





***Design goal*: improve cache efficiency**



Efficiency = High cache hit rate + Low cache space cost

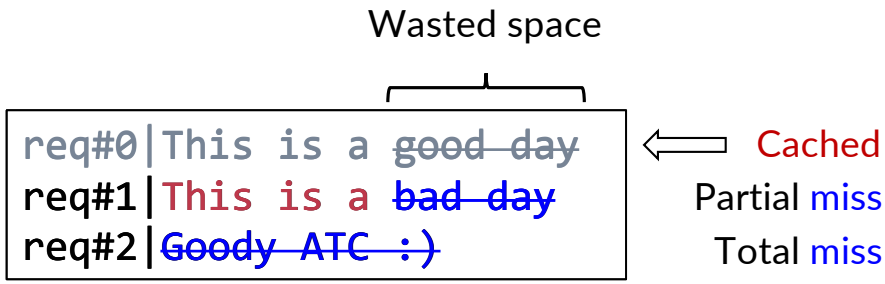
Design goal: improve cache efficiency



Efficiency = High cache hit rate + Low cache space cost

Which requests to cache & which to evict?

- Caching the KV$ that won't get hit reduces hit rate



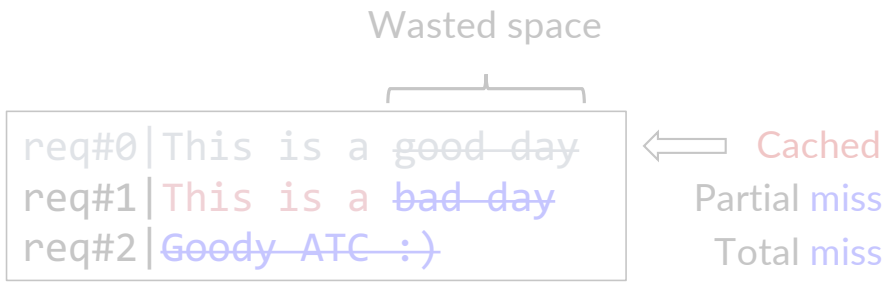
Design goal: improve cache efficiency



Efficiency = High cache hit rate + Low cache space cost

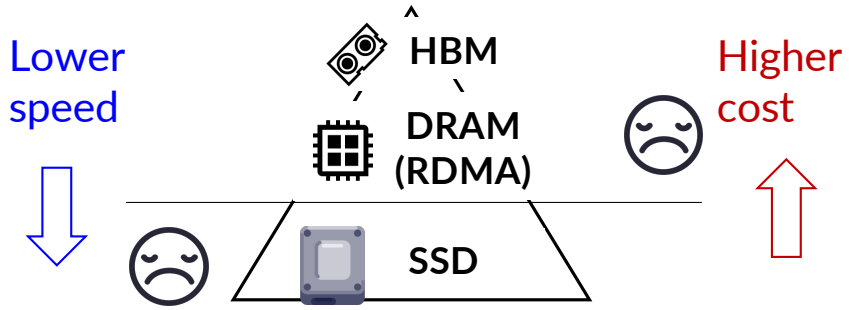
Which requests to cache & which to evict?

- Caching the KV$ that won't get hit <u>reduces hit rate</u>



How large cache space is required for KV$?

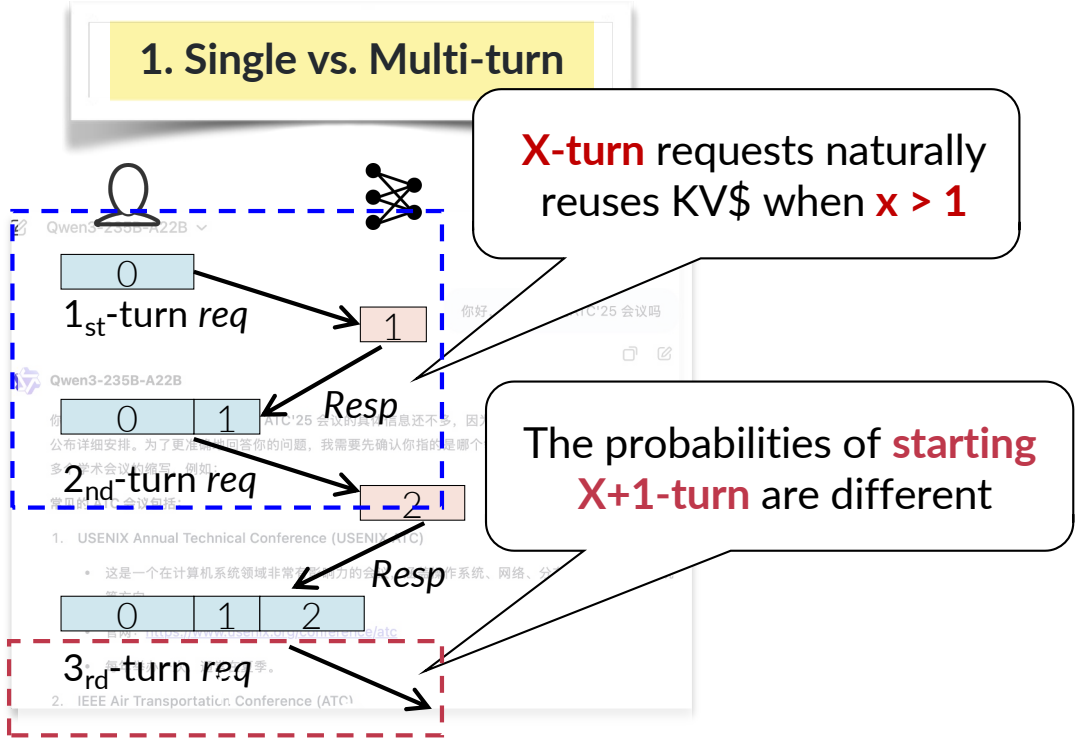
- Storing excessive KV$ for performance <u>increases storage cost</u>



A classic in memory hierarchy speed, capacity, cost

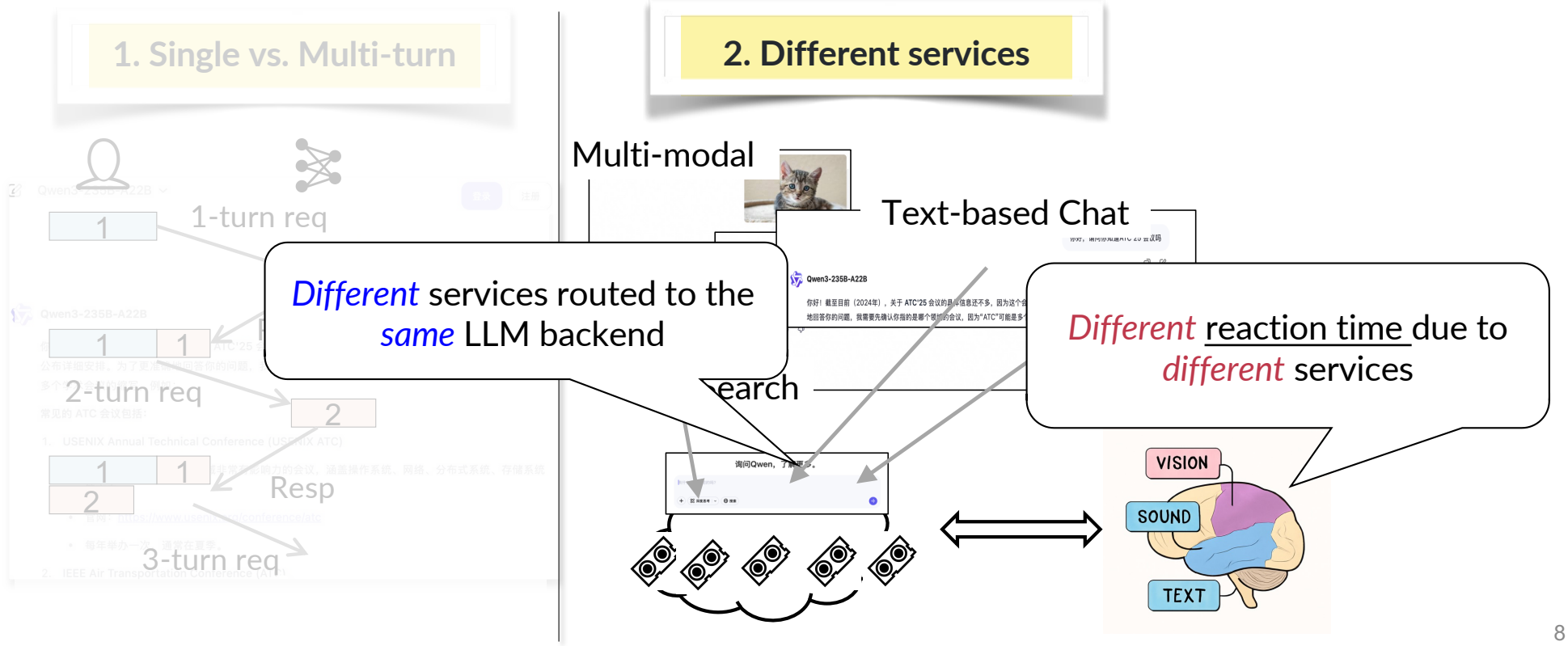
Design challenge: high variabilities of LLM serving workloads

The variabilities are in three factors:



Design challenge: high variabilities of LLM serving workloads

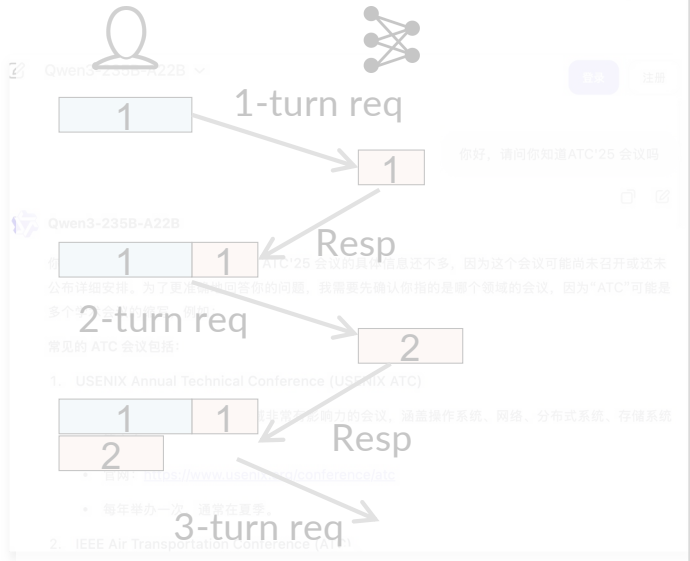
The variabilities are in three factors:



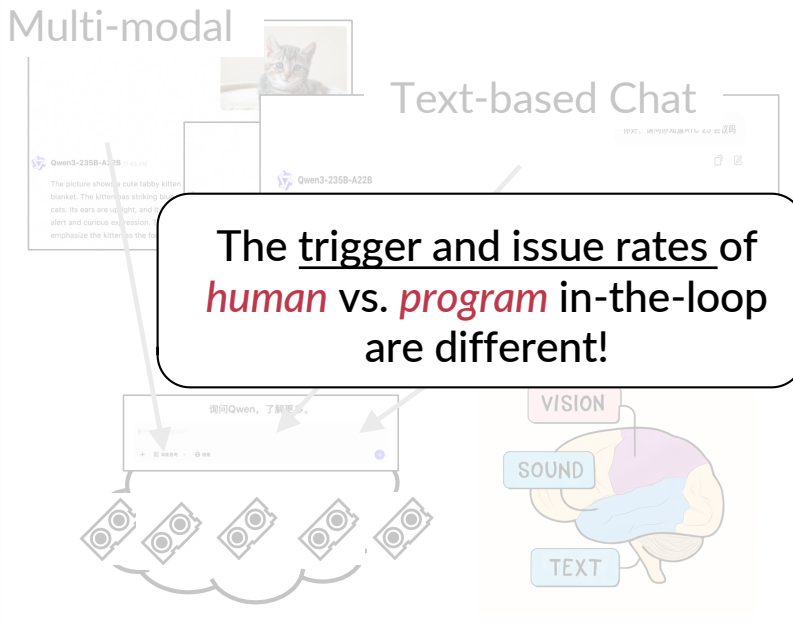
Design challenge: high variabilities of LLM serving workloads

The variabilities are in three factors:

1. Single vs. Multi-turn



2. Different purposes



3. Human vs. Program in-the-loop

Browser/Phone/etc.



Human in the loop

vs.

Program in the loop



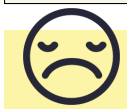
The trigger and issue rates of *human* vs. *program* in-the-loop are different!

Problem: workload-agnostic approaches lead to suboptimal designs

Which requests to cache & which to evict?

- Caching the KV$ that won't get hit reduces hit rate

Existing systems, like CachedAttention[1], MoonCake[2] and Pensieve[3], adopt workload-agnostic designs like **LRU**, **LFU**...



Workload-agnostic designs may inadvertently evict KV$ w/ high value, **degrading up to 23.9% hit rates**

[1] Cost-Efficient Large Language Model Serving for Multi-turn Conversations with **CachedAttention**, **ATC'24**
[2] **Mooncake**: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot, **FAST'25**
[3] Stateful Large Language Model Serving with **Pensieve**, **EuroSys'25**

Problem: workload-agnostic approaches lead to suboptimal designs

Which requests to cache & which to evict?

- Caching the KV$ that won't get hit reduces hit rate

Existing systems, like CachedAttention[1], MoonCake[2] and Pensieve[3], adopt workload-agnostic designs like LRU, LFU...

Workload-agnostic designs may inadvertently evict KV$ w/ high value, **degrading up to 23.9% hit rates**

How large cache space is required for KV$?

- Storing excessive KV$ for performance increases storage cost

Existing systems, like MoonCake[2], leverage all available DRAM in the cluster to store KV$



Workload-agnostic design may pollute cache space w/ no reuse KV$, leads to **up to 40% DRAM wastes**

[1] Cost-Efficient Large Language Model Serving for Multi-turn Conversations with **CachedAttention**, **ATC'24**
[2] **Mooncake**: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot, **FAST'25**
[3] Stateful Large Language Model Serving with **Pensieve**, **EuroSys'25**

Our work: workload-driven approach for design optimization

Which requests to cache & which to evict?

How large cache space is required for KV$?

Our work: workload-driven approach for design optimization

Which requests to cache & which to evict?

How large cache space is required for KV$?

Characterizing KV$ usage with differentiated workloads



Findings

Insights

Optimizing

Cost + Perf.





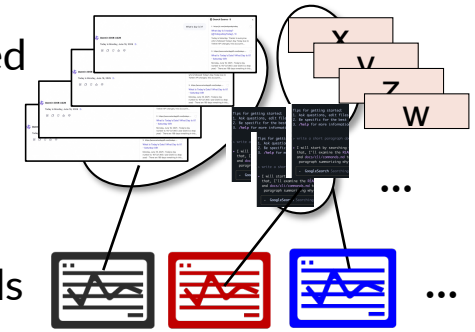
Our work: workload-driven approach for design optimization

Which requests to cache & which to evict?

How large cache space is required for KV$?

Characterizing methodology

1. Categorize requests based on workloads
2. Analyze the KV$ reuse pattern of each workloads



Characterizing KV$ usage with differentiated workloads



Findings

Insights

Optimizing

Cost + Perf.

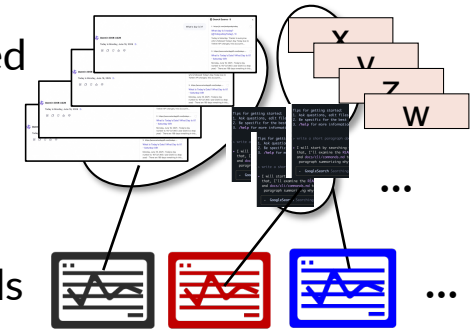
Our work: workload-driven approach for design optimization

Which requests to cache & which to evict?

How large cache space is required for KV$?

Characterizing methodology

1. Categorize requests based on workloads
2. Analyze the KV$ reuse pattern of each workloads



Workload-aware KV$$ eviction policy

Workload-aware KV$$ space estimator

Characterizing KV$ usage with differentiated workloads



Findings
Insights

Optimizing

Cost + Perf.

Challenge: incomplete real-world LLM serving traces

Current open-sourced traces lack sufficient information

Characterizing methodology

1. **Categorize requests** based on workloads
2. Analyze the **reuse pattern** of each workload

	Type	Turn	Time	Content
ShareGPT [1]	✗	✓	✗	✓
Mooncake [2]	✗	✗	✓	✓

[1] ShareGPT4V: Improving Large Multi-Modal Models with Better Captions, arXiv'23

[2] Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. FAST'25

Our work: the first versatile KV$ trace + workload-aware KV$$

Traces: One-week anonymous trace from *Alibaba Bailian Platform*



- Trace A: Chat-Bot Application (ToC)
- Trace B: API-call (ToB)
- All requests are sent to the same ***Qwen*** model, the leading open-sourced model

Open-sourced at: https://github.com/alibaba-edu/qwen-bailian-usagetraces-anon

	Type	Turn	Time	Content
ShareGPT [1]	✗	✓	✗	✓
Mooncake [2]	✗	✗	✓	✓
Our trace	✓	✓	✓	✓

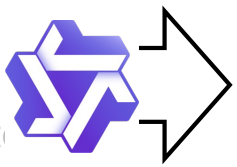
[1] ShareGPT4V: Improving Large Multi-Modal Models with Better Captions, arXiv'23

[2] Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. FAST'25

Outline of the talk

Which requests to cache & which to evict?

1. Group requests based on workloads
2. Analyze the reuse patt
of each workload



Key insights & takeaways for KV$$ systems

Workload-aware KV$$ eviction policy

Workload-aware KV$$ space estimator

How large cache space is required for KV$?

Characterizing KV$ usage with differentiated workloads



Findings
Insights

Optimizing

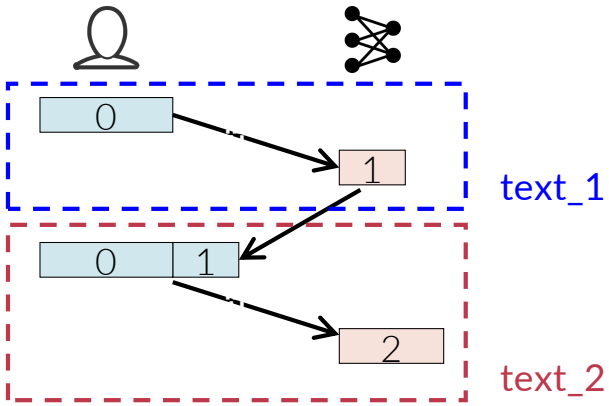
Cost + Perf.



Insight#1: The reuse pattern vary among workloads

We categorize workload by [`type` X `turn`]

- e.g., text_1 is a set of requests in chatbot services with exactly 1 request issued
- text_1 & text_2 are **different** workloads!



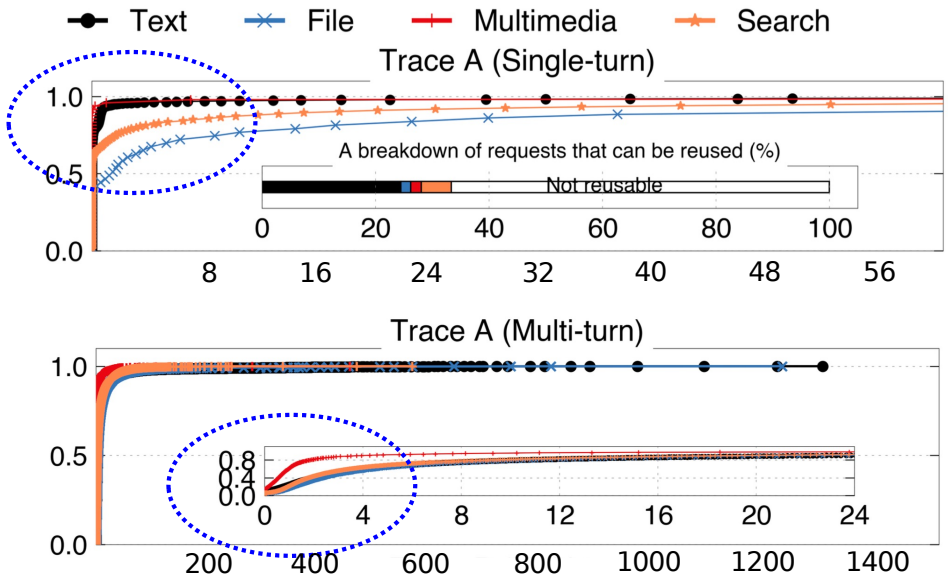
Insight#1: The reuse pattern vary among workloads

We categorize workload by [`type` X `turn`]

- e.g., text_1 is a set of requests in chatbot services with exactly 1 request issued
- text_1 & text_2 are different workloads!

Reuse time distributions are different

- 30% single-turn requests can be reused!
- Reuse of text_1 is unnecessarily text_2

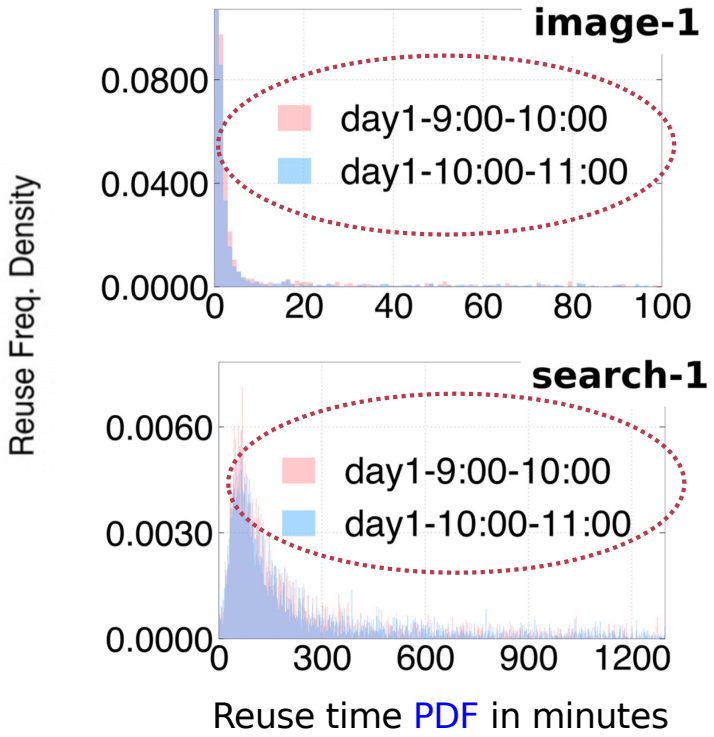


Takeaway#1: Caching and eviction should consider workload categories including single-turn

Insight#2: The reuse probability of a category is predictable

The reuse time distribution is stationary across the adjacent hours

- The density functions are in the same shape
- i.e., an exponential distribution



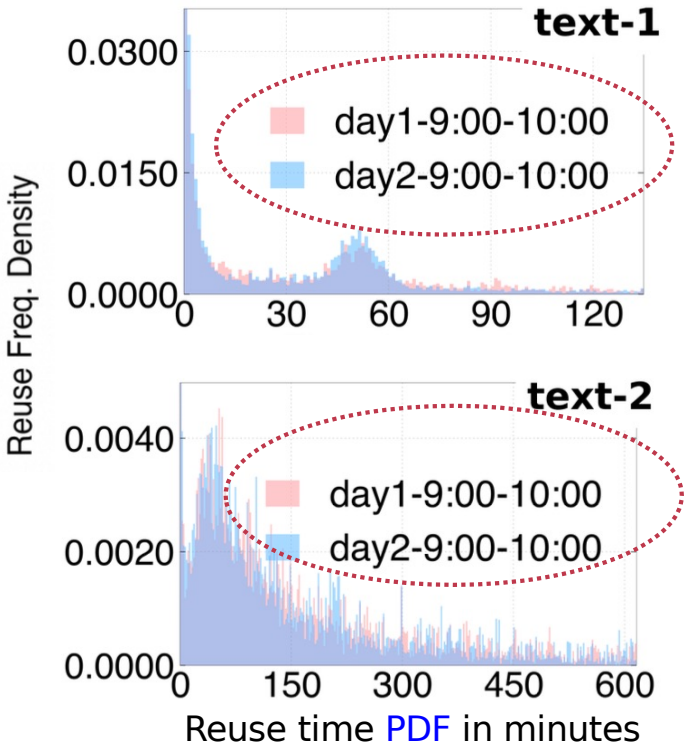
Insight#2: The reuse probability of a category is predictable

The reuse time distribution is stationary across the adjacent hours

- The density functions are in the same shape

The reuse time distribution is stationary in the same period across days

- The density functions are the same either



Takeaway#2: Utilize workload history to optimize the cache policies

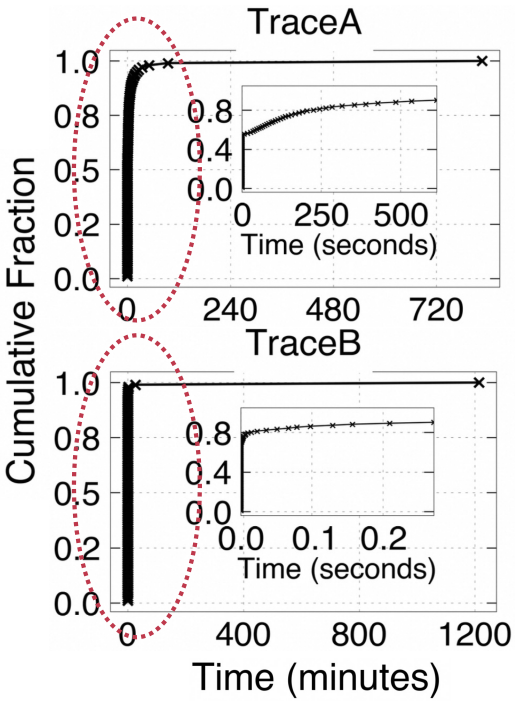
Insight#3: The KV$ lifespan is ephemeral & predictable

The lifespan of KV$ is from generation to the last reuse time

- i.e., the ideal caching interval

Most KV$ have short lifespans, thus are ephemeral

- p90 lifespan of KVCache are 10min in Trace A
- p90 lifespan of KVCache are even 0.3s in Trace B



KV$ lifespan distribution

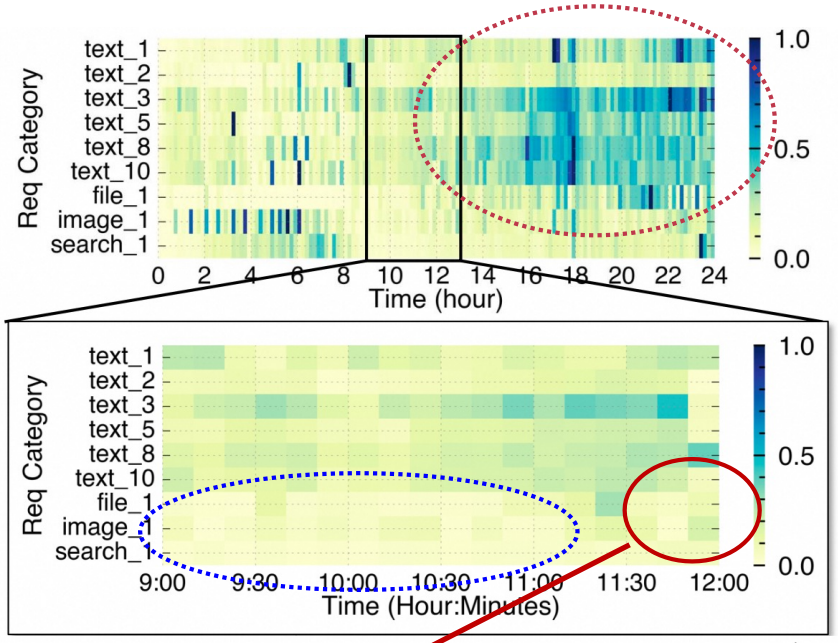
Insight#3: The KV$ lifespan is ephemeral & predictable

The lifespan of KV$ is from generation to the last reuse time

Most KV$ have short lifespans, thus are ephemeral

KV$ lifespan is also predictable

- Though lifespan is changing during a day
- Lifespan remains stable within an hour



Intensity: The normalized mean lifespan

Takeaway #3: Avoid caching KV$ for a long time + Use the predication to reduce cache space

Outline of the talk

Which requests to cache & which to evict?

How large cache space is required for KV$?

1. Group requests based on workloads
2. Analyze the reuse patte… of each workload

Insight #3

Key insights & takeaways for KV$$ systems

Workload-aware KV$$ eviction policy

Workload-aware KV$$ …tor



For more **findings + takeaways**, please check our paper!

Characterizing KV$ usage with diff…ed workloads



Findings

Insights

Optimizing

Cost + Perf.



Outline of the talk

Which requests to cache & which to evict?

How large cache space is required for KV$?

1. Group requests based on workloads
2. Analyze the reuse patte… of each workload

Key insights & takeaways for KV$$ systems

Workload-aware KV$$ eviction policy

Workload-aware KV$$ space estimator

Characterizing KV$ usage with differentiated workloads



Findings
Insights

Optimizing

Cost + Perf.

Design: a workload-aware cache policy

KV$ policy: how to decide whether a request's KV$ should be stored in the KV$?

Essentially, we need to calculate a priority for each KV$ block

- If the cache space is sufficient, we cache it
- If the cache space is insufficient, we select a victim

Key question

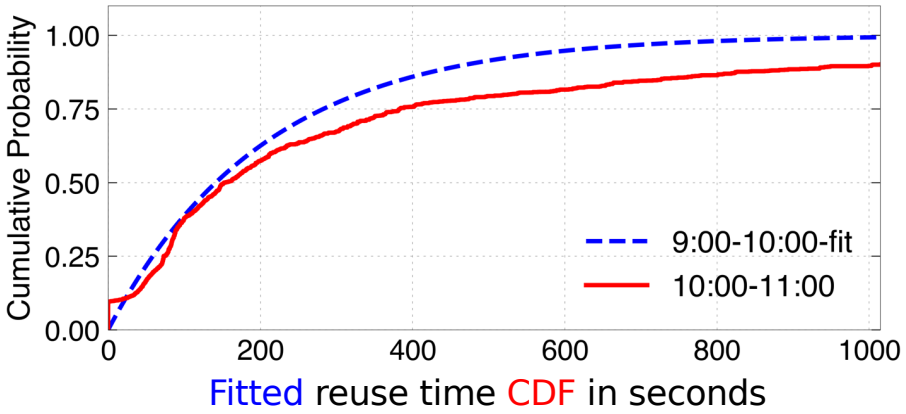
- How to define the the priority?

Let's start from the well-known LRU

Least

$$priority = -(t - t_{last})$$

Recently Used



Design: a workload-aware cache policy

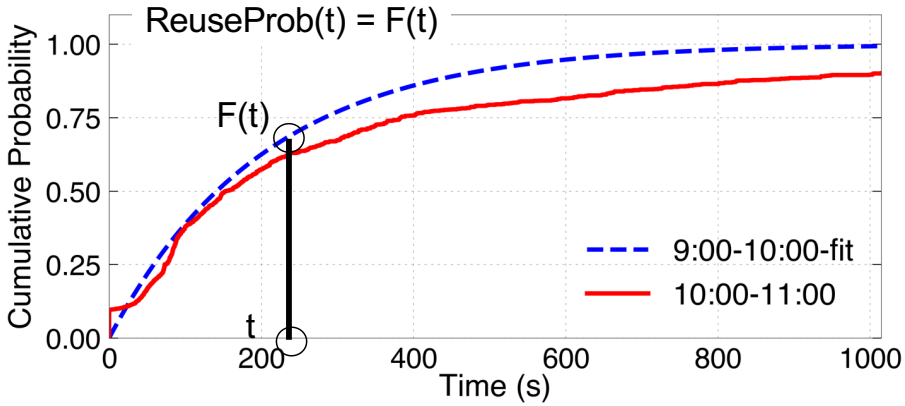
KV$ policy: how to decide whether a request's KV$ should be stored in the KV$?

Essentially, we need to calculate a priority for each KV$ block

- If the cache space is sufficient, we cache it
- If the cache space is insufficient, we select a

Key question

- How to define the the priority?



$$priority = -\mathrm{Prob}_{reuse}(t - t_{last})$$

+ Insight#1+2: The reuse pattern of workloads diff & can be predicted

Design: a workload-aware cache policy

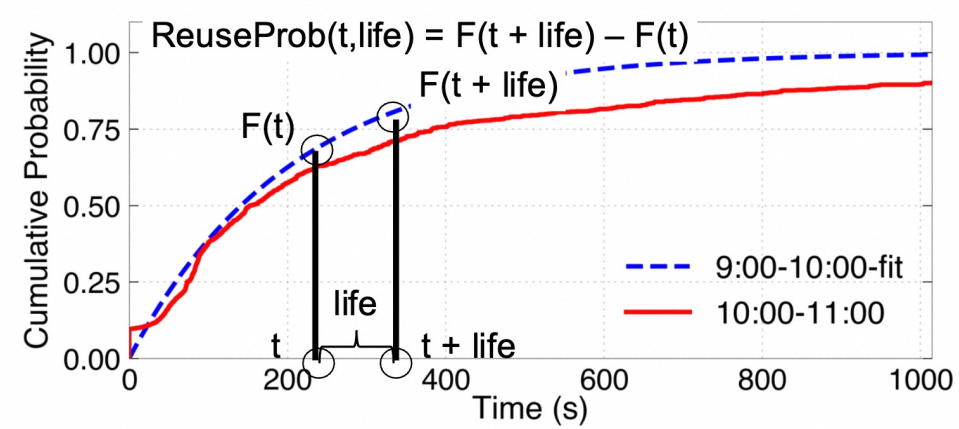
KV$ policy: how to decide whether a request's KV$ should be stored in the KV$?

Essentially, we need to calculate a priority for each KV$ block

- If the cache space is sufficient, we cache it
- If the cache space is insufficient, we select a

Key question

- How to define the the priority?



$$priority = -\text{Prob}_{reuse}(t - t_{last}, life)$$

+ Insight#3: The KV$ lifespan is ephemeral & predictable

Evaluation: workload-aware cache policy (WA)

Testbed

- 8 Nvidia A800 GPUs

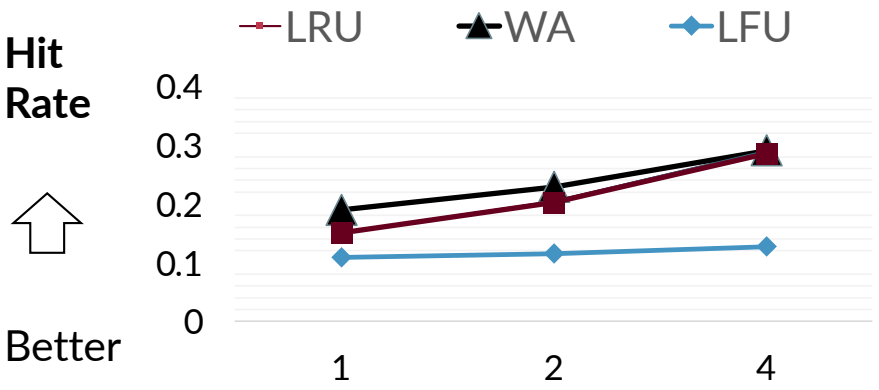
Model

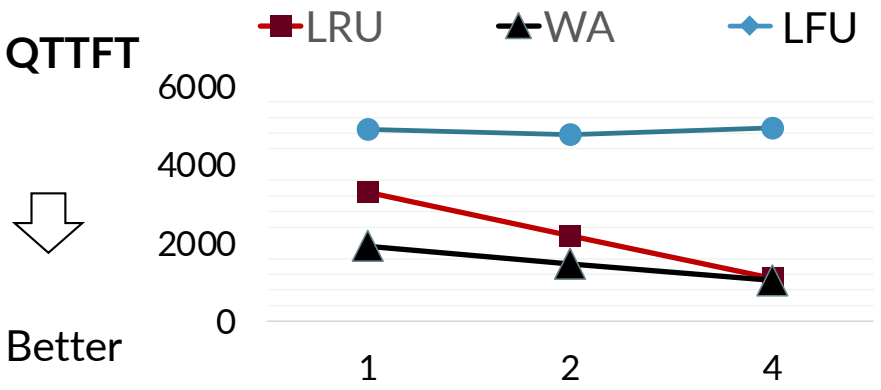
- Qwen-2

Evaluated trace

- Our open-sourced traces

1 – 24% KVCache hit increases & **29 – 42%** queued TTFT reductions





More insights + design details in our paper!

Detailed workload-aware policy

- e.g., optimizations to make it effective in practice

More evaluations

- Detailed albadation study + comparison w/ more known policies

More insights

- KV$ reuses are skewed
- KV$ reuses across users are rare
- ...



KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider

Jiahao Wang[†1], Jinbo Han[†1], Xingda Wei[✉1], Sijie Shen[2], Dingyan Zhang[1], Chenguang Fang[2], Rong Chen[1], Wenyuan Yu[2], and Haibo Chen[1]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University [2]Alibaba Group

Abstract

Serving large language models (LLMs) is important for cloud providers, and caching intermediate results (KV$) after processing each request substantially improves serving throughput and latency. However, there is limited understanding of how LLM serving benefits from KV$ caching, where system design decisions like cache eviction policies are highly workload-dependent.

In this paper, we present the first systematic characterization of the KV$ workload patterns from one of the leading LLM service providers. We draw observations that were not covered by previous studies focusing on synthetic workloads, including: KV$ reuses are skewed across requests, where reuses between single-turn requests are equally important as multi-turn requests; the reuse time and probability are diverse considering all requests, but for a specific request category, the pattern tends to be predictable; and the overall cache size required for an ideal cache hit ratio is moderate. Based on the characterization, we further propose a workload-aware cache eviction policy that improves the serving performance under real-world traces, especially with limited cache capacity.

1 Introduction

Large language models (LLMs) have become an important cloud service thanks to their emerging capabilities in various tasks. Both customer-facing services (to-C) like conversation-based chatbots [43] and business services (to-B) through LLM API calls have been widely deployed on the cloud [42, 21, 4]. As online services, serving LLMs with low latency and high throughput is critical, but achieving this in a cost-efficient way is challenging due to the huge computational requirements to serve each LLM request.

Caching intermediate results of served requests (KV$) has become a common methodology to improve serving performance [45, 5]: when two requests share the same input prefix, their KV$ are identical; so if we cache the KV$ of the first (KV$ cache), we don't need to compute the KV$ for the next. Hence, the serving latency is reduced with the throughput improved thanks to the reduced computation under hits.

Like traditional data caching systems [60, 7, 40], the effectiveness of KV$ cache system components like cache eviction policy should be tailored to the KV$ reuse features of workloads. However, LLM serving workloads are complex due to their diverse request types as well as the unique features of each request type. For example, a production LLM service handles both to-C workloads (like chatting) and to-B workloads (like task automation with API calling). These requests can further be processed either as single-turn or multi-turn interactions (§2). Since the KV$ can be reused across different single-turn requests and multi-turn requests, both the single-turn request types and the multi-turn patterns can possibly affect the design choices of KV$ cache.

	Time	Type	UID	Turn	Content
ShareGPT [2]	✗	✗	✗	✓	✓
Mooncake [1]	✓	✗	✗	✗	✓
Traces in this work	✓	✓	✓	✓	✓

Table 1: *A comparison of currently available LLM serving traces. **Time** is the invocation time of each request. **Type** is the request type, e.g., whether it is an API call. **UID** is the user ID that invokes the request. **Turn** is the multi-turn information of the request. **Content** is the (anonymous) input and output tokens of each request.*

There remains a significant gap in the understanding of common LLM serving workloads for KV$ cache due to the lack of in-depth analysis of real-world traces. First, it is unclear whether production workloads have many reuses and which request type contributes most to the reuses. Second, the distributions of the reuse time or reuse probability of KV$ remain unknown, which is critical to cache policy designs. Finally, there is a lack of characterization of the lifespan of KV$, which is essential in helping us estimate the required cache capacity for serving LLMs in the wild.

Characterizing production traces. To bridge this gap, we collected and characterized two representative production serving workloads from one of the world's largest cloud providers (ALIYUN): a to-C workload where users submit requests to the cloud-hosted LLM service via browser-based chatbots or mobile applications, and a to-B workload where developers send OpenAI-compatible requests to the LLM service through API calls in their programs. We collected as much detailed information as possible within the company's strict privacy policy, including request submission times, request types (single-turn, multi-turn, API requests, or others),

[†]The two authors contributed equally to this work.
[✉]Xingda Wei is the corresponding author (wxdwfc@sjtu.edu.cn).

Conclusion

Problem: efficiently deploying KV$$ at LLM service provider is non-trivial

Approach

- Characterize the KV$ usage in the wild then optimize
- Derive policies based on different LLM workloads
- Leveraging the predictable features of KV$ usage for optimization

Our work:

- The first systematic study of the KV$$ in the wild
- The first workload-aware KV$$ optimization system, which gives up to 23.9% more KV$ hit given the same KV$$ space

Thanks & QA