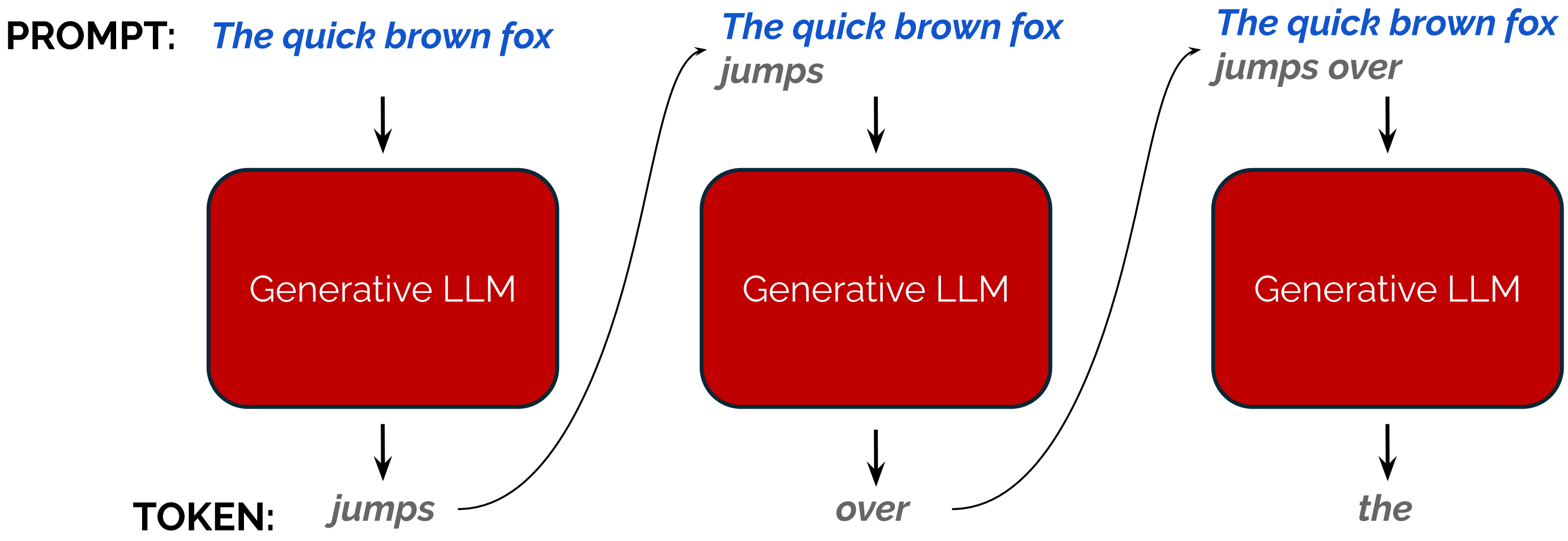
DéjàVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving

Foteini Strati[1,3], Sara McAllister[2,3], Amar Phanishayee[3], Jakub Tarnawski[3], Ana Klimovic[1]
[1]ETH Zurich, [2]Carnegie Mellon University, [3]Microsoft Research

ICML
International Conference
On Machine Learning

Generative LLM Serving



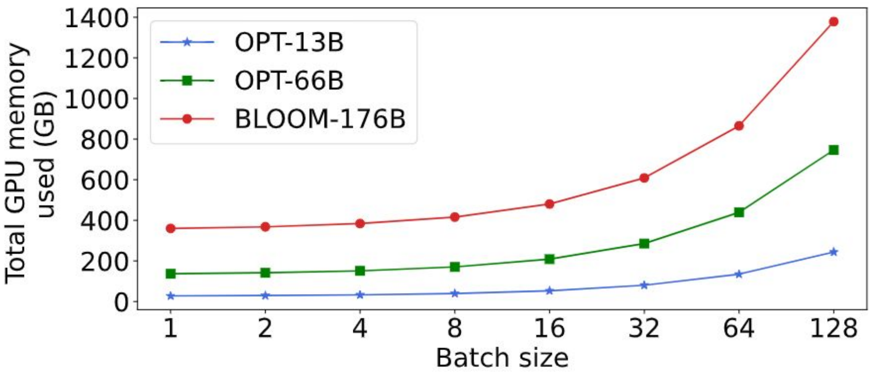
Generative LLM Serving characteristics

Generative LLM Serving characteristics

1. **Is stateful:**
- Generation at position *i* depends on tokens at positions *[0, i-1]*
- Results saved at a ***Key-Value cache*** to speed up computation

Generative LLM Serving characteristics

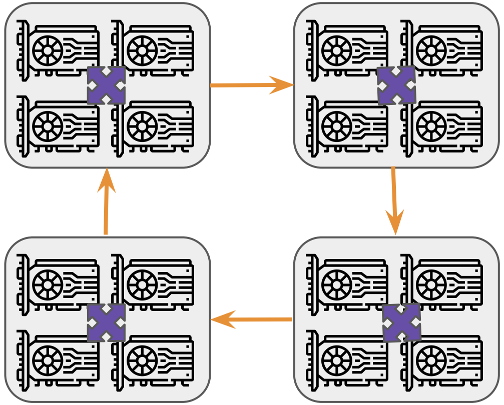
1. Is stateful
2. **Has large memory footprint:**
    - Model parameters
    - KV cache and other intermediate states



Generative LLM Serving characteristics

1. Is stateful
2. **Has large memory footprint:**
   - Model parameters
   - KV cache and other intermediate states

Inference is distributed across multiple GPUs with tensor and pipeline parallelism



Challenges of distributed LLM serving

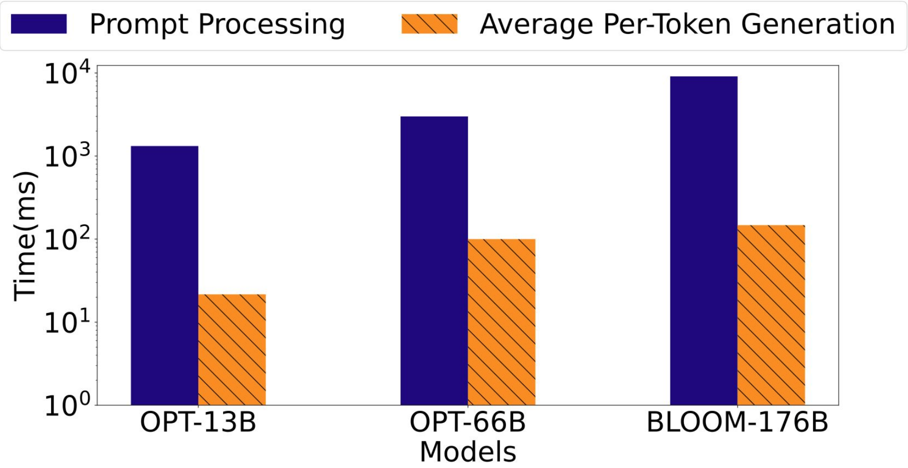
1. Difference in prompt and per-token generation latency leads to pipeline bubbles
2. Inefficient usage of GPU memory
3. No failure handling

Challenges of distributed LLM serving

1. **Difference in prompt and per-token generation latency leads to pipeline bubbles**
2. Inefficient usage of GPU memory
3. No failure handling

Prompt processing vs per-token generation latency

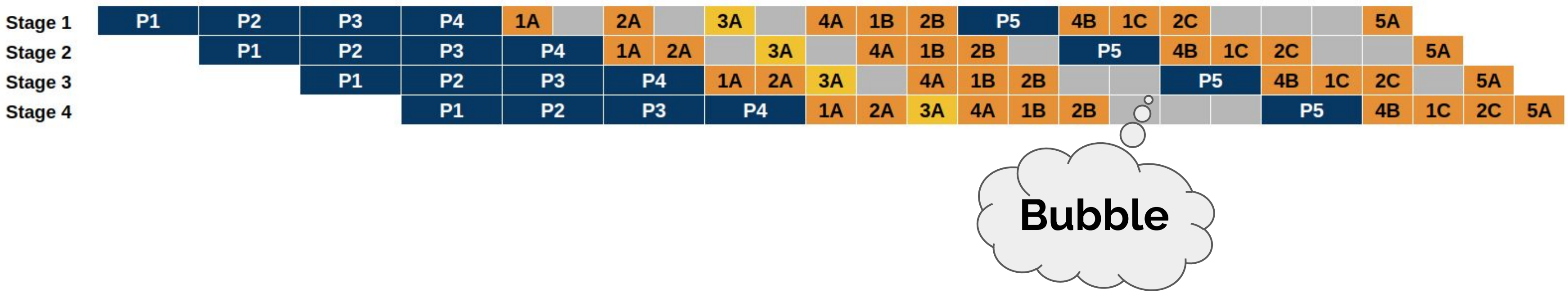
Due to the usage of KV cache, the time for prompt processing can be much higher than per-token generation time



Prompt processing vs per-token generation latency

Due to the usage of KV cache, the time for prompt processing can be much higher than per-token generation time

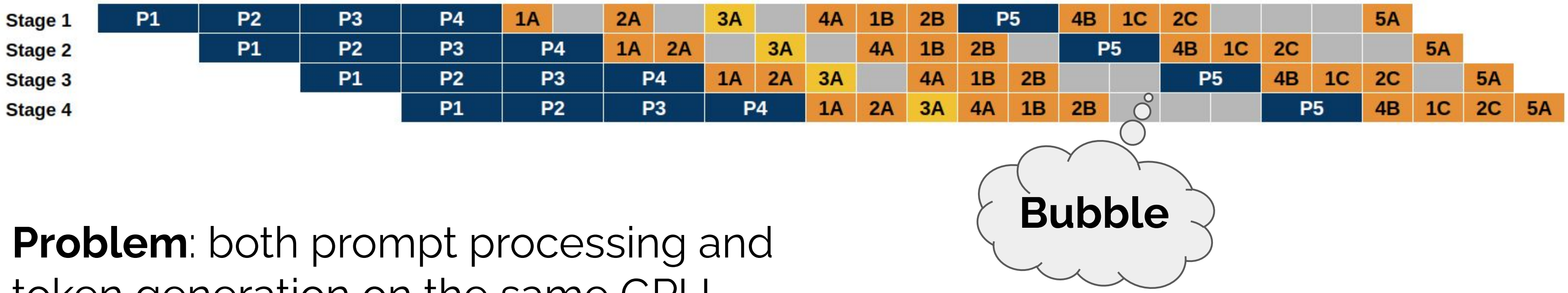
Bubbles and resource underutilization in pipeline parallel setups



Prompt processing vs per-token generation latency

Due to the usage of KV cache, the time for prompt processing can be much higher than per-token generation time

Bubbles and resource underutilization in pipeline parallel setups

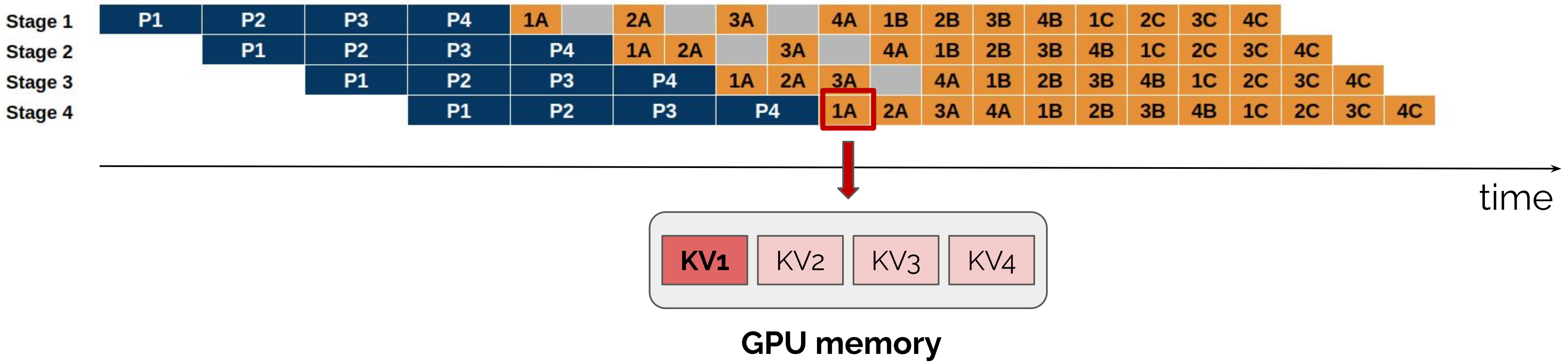


Problem: both prompt processing and token generation on the same GPU

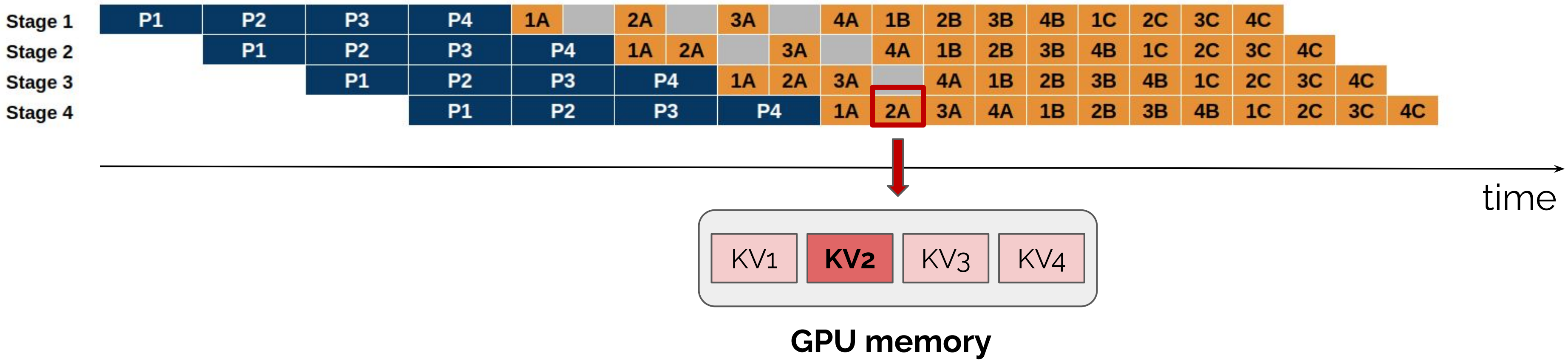
Challenges of distributed LLM serving

1. Difference in prompt and per-token generation latency leads to pipeline bubbles
2. **Inefficient usage of GPU memory**
3. No failure handling

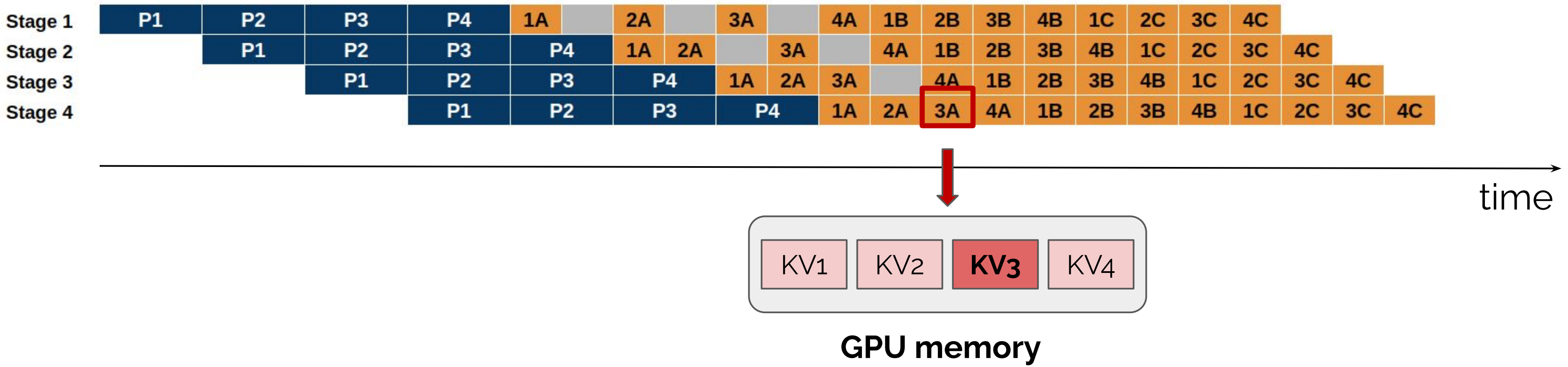
Inefficient usage of GPU memory



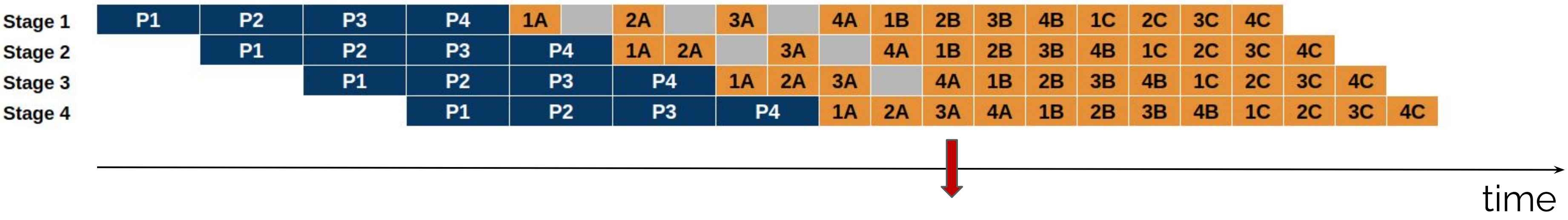
Inefficient usage of GPU memory



Inefficient usage of GPU memory



Inefficient usage of GPU memory



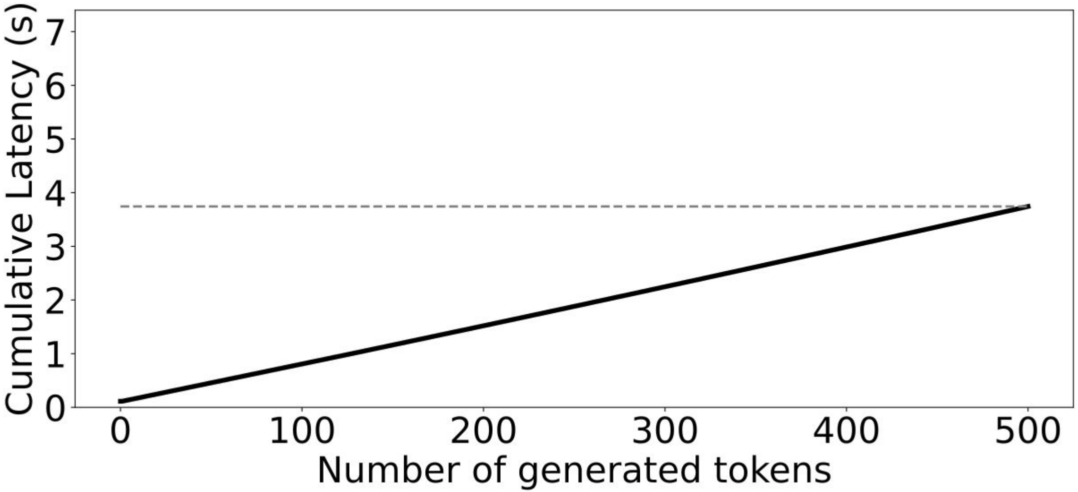
One microbatch is processed at a time.

However, the KV cache of *all* microbatches is kept in GPU memory!

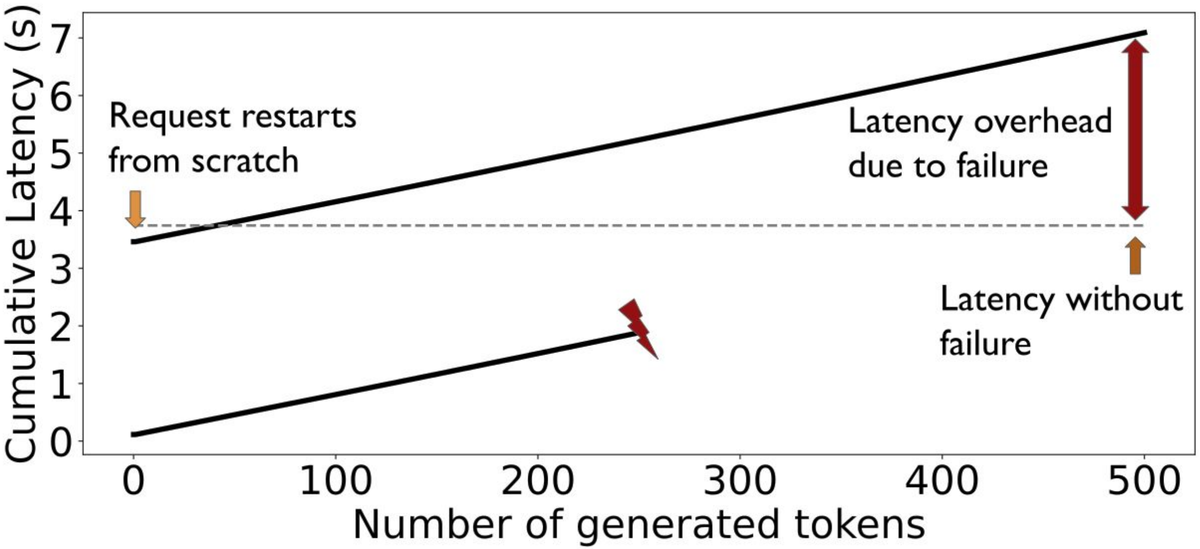
Challenges of distributed LLM serving

1. Difference in prompt and per-token generation latency leads to pipeline bubbles
2. Inefficient usage of GPU memory
3. **No failure handling**

Latency without failures



Latency with a failure



Challenges of distributed LLM serving

1. Difference in prompt and per-token generation latency leads to pipeline bubbles
2. Inefficient usage of GPU memory
3. No failure handling

Challenges and our proposed solutions

1. Difference in prompt and per-token generation latency leads to pipeline bubbles

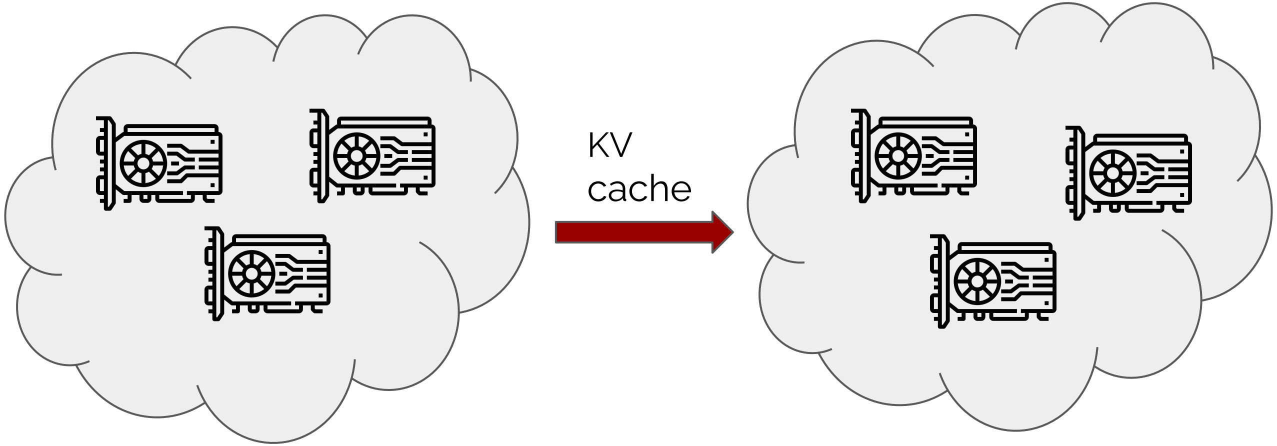
   Prompt-token disaggregation

2. Inefficient usage of GPU memory
3. No failure handling

DéjàVu Key Idea 1: Prompt-Token disaggregation

Dedicated GPUs for
Prompt processing

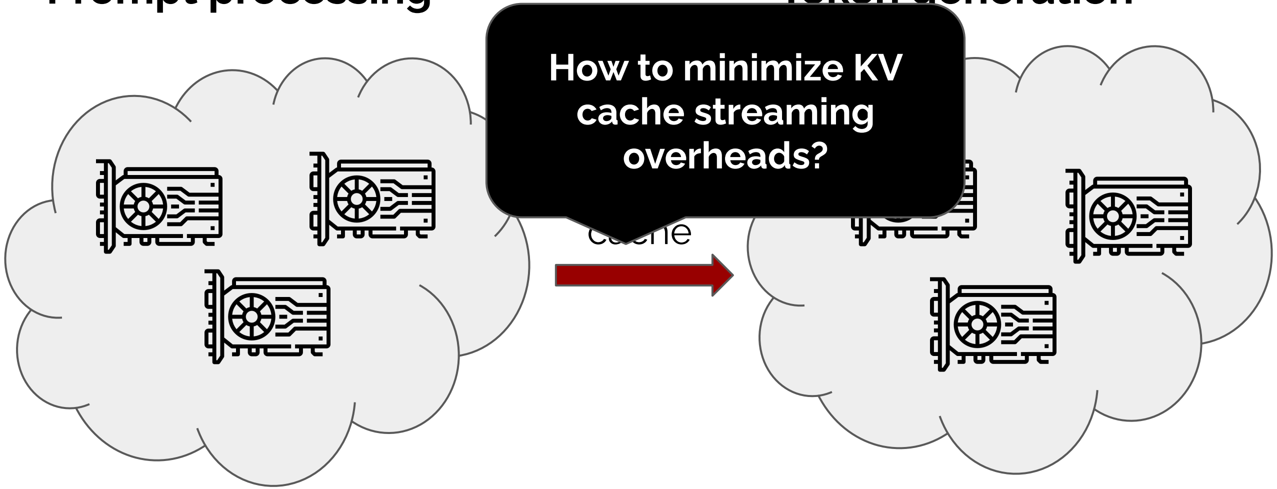
Dedicated GPUs for
Token generation



DéjàVu Key Idea 1: Prompt-Token disaggregation

Dedicated GPUs for
Prompt processing

Dedicated GPUs for
Token generation



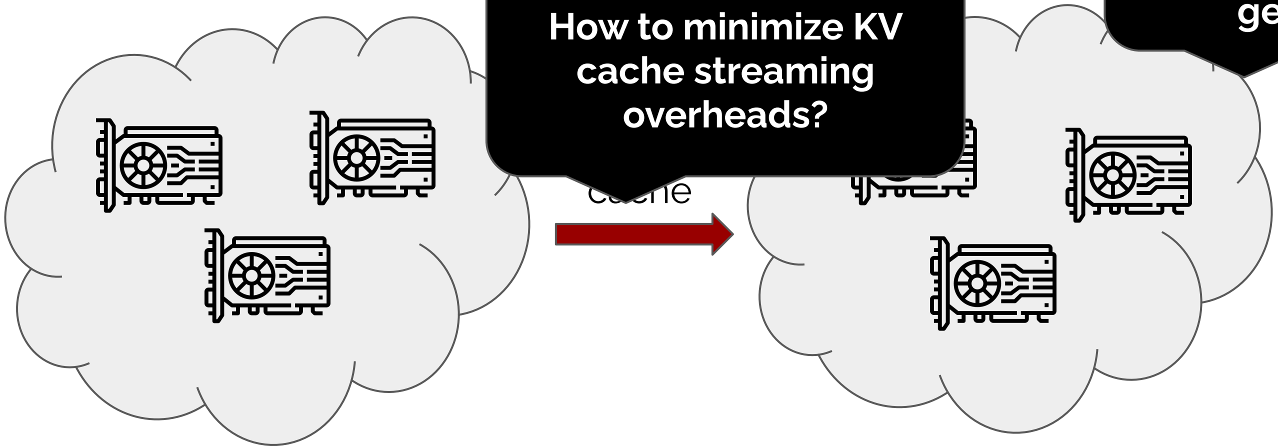
DéjàVu Key Idea 1: Prompt-Token disaggregation

Dedicated GPUs for
Prompt processing

Dedicated GPUs
Token generatio

How to minimize KV cache streaming overheads?

How to allocate resources for prompt processing and token generation?



Challenges and our proposed solutions

1. Difference in prompt and per-token generation latency leads to pipeline bubbles

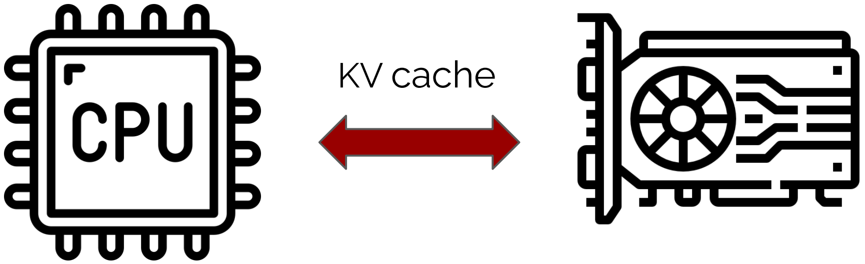
   Prompt-token disaggregation

2. Inefficient usage of GPU memory

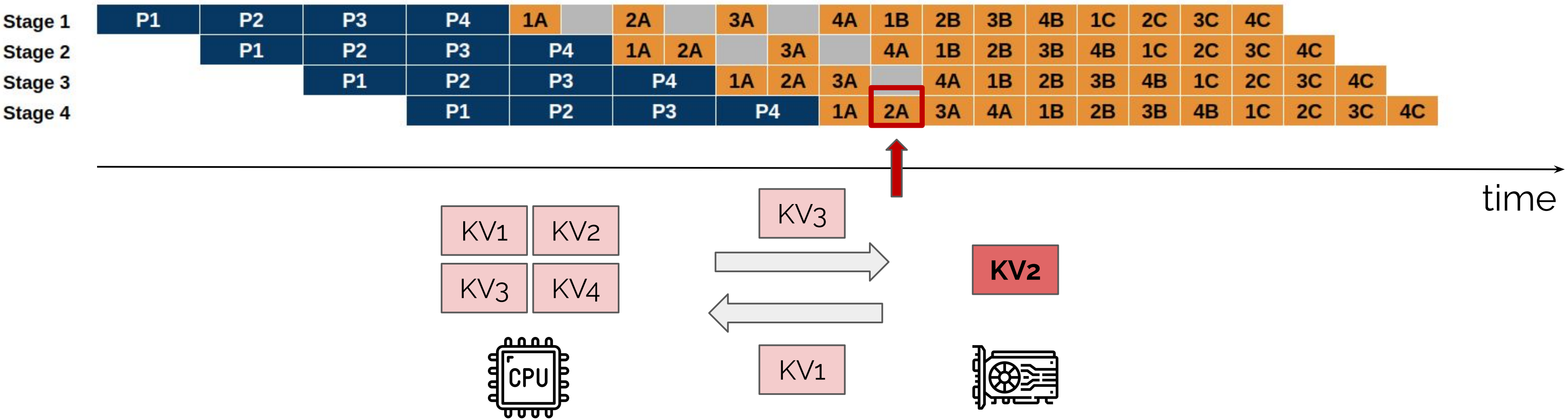
   Microbatch swapping

3. No failure handling

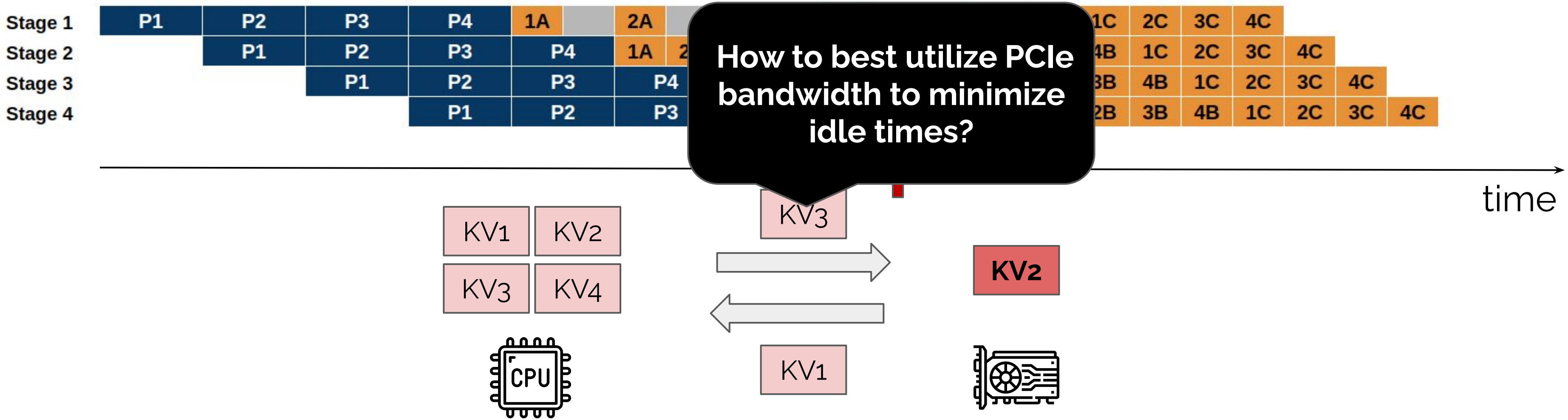
DéjàVu Key Idea 2: Microbatch swapping



DéjàVu Key Idea 2: Microbatch swapping



DéjàVu Key Idea 2: Microbatch swapping



Challenges and our proposed solutions

1. Difference in prompt and per-token generation latency leads to pipeline bubbles

   Prompt-token disaggregation

2. Inefficient usage of GPU memory

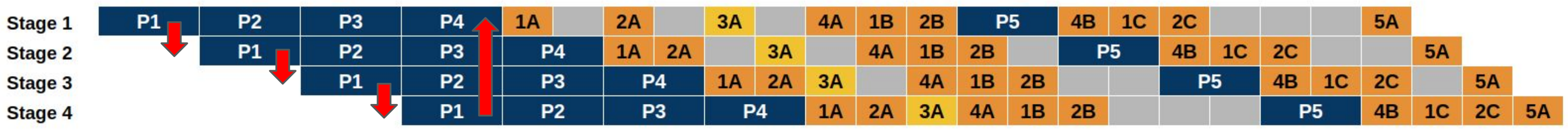
   Microbatch swapping

3. No failure handling

   KV cache replication and fault-handling mechanism

DéjàVu Key Idea 3: KV cache replication

KV cache replication to CPU memory



- efficient fault detection and recovery

Proposed solutions

3. Statefulness and inefficient failure handling

Idea: Replicate the KV cache to persistent storage or remote CPU memory

How to minimize KV cache streaming overheads?

How to efficiently detect failures and recover?

Common requirement

1. Prompt-token disaggregation
2. Microbatch swapping
3. KV cache replication and fault-handling mechanism

They all require a fast and versatile KV cache streaming mechanism

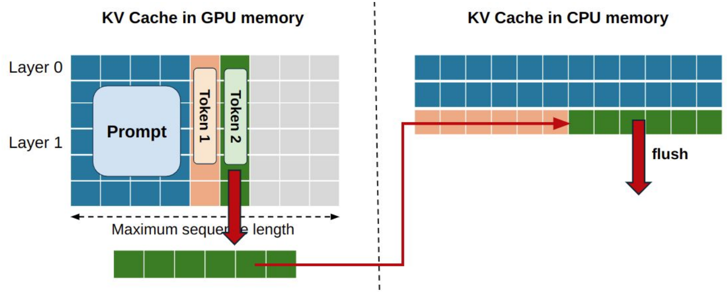
DéjàVuLib

- An **efficient** and **modular** KV cache streaming library, with **optimizations for fast streaming**

DéjàVuLib

- An **efficient** and **modular** KV cache streaming library, with **optimizations for fast streaming**

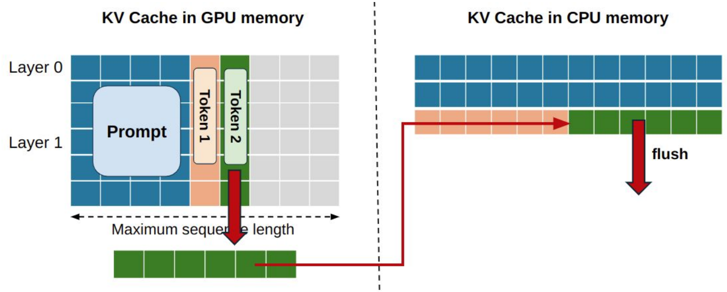
1. **Buffered Transfers**



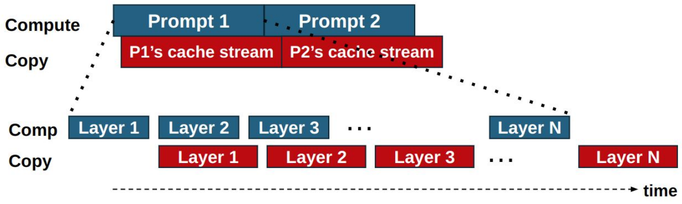
DéjàVuLib

- An **efficient** and **modular** KV cache streaming library, with **optimizations for fast streaming**

1. **Buffered Transfers**



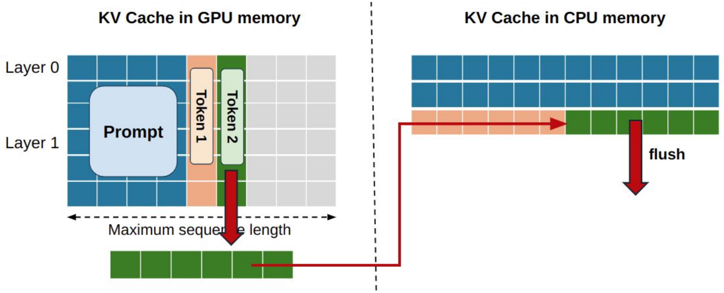
2. **Layer-by-layer prompt cache streaming**



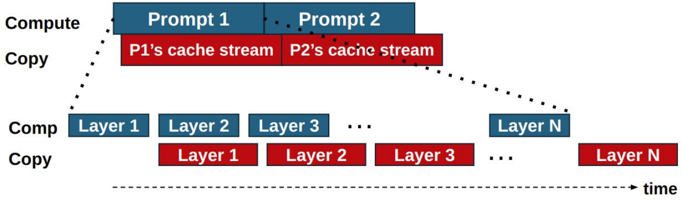
DéjàVuLib

- An **efficient** and **modular** KV cache streaming library, with **optimizations for fast streaming**

1. **Buffered Transfers**



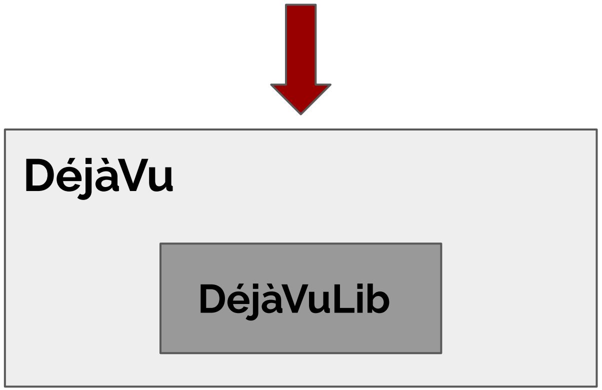
2. **Layer-by-layer prompt cache streaming**



3. **Token computation and streaming overlap**

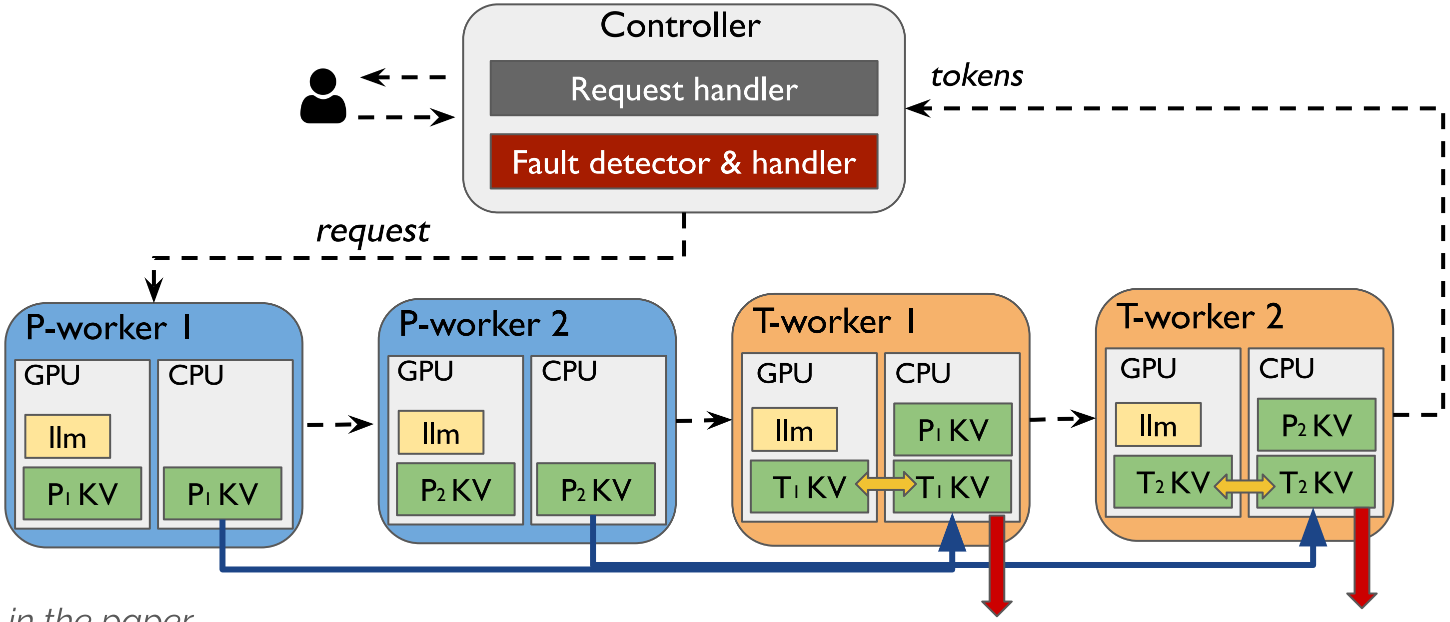
Common requirement

1. Prompt-token disaggregation
2. Microbatch swapping
3. KV cache replication and fault-handling mechanism



The DéjàVu system

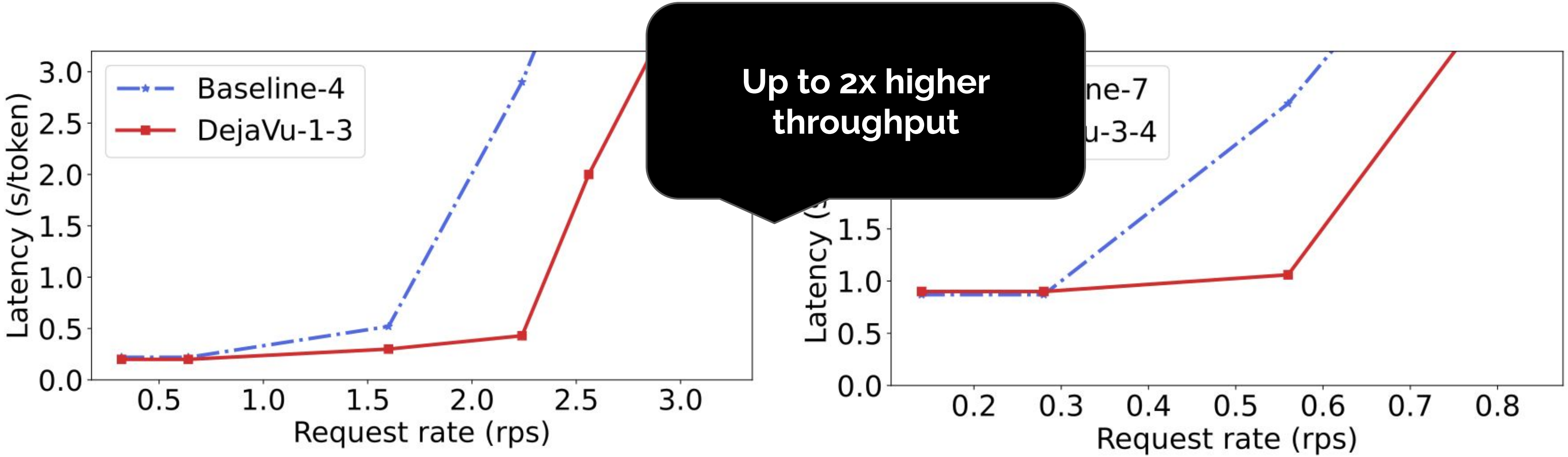
Supports: **disaggregation, KV cache swapping, fault-tolerance**



More details in the paper

Evaluation

DéjàVu disaggregation



OPT-66B **BLOOM-176B**

- Baseline-X: FasterTransformer with X machines
- DejaVu-X-Y: Disaggregation with X machines for prompt, Y for token

lmsys dataset

Conclusion

Challenges	DéjàVu Solutions
Bimodal prompt vs per-token generation latency	**Prompt-token disaggregation**
Inefficient usage of GPU memory	**Microbatch swapping**
Statefulness and inefficient failure handling	**KV cache replication and fault-handling mechanism**

Conclusion

Challenges		DéjàVu Solutions
Bimodal prompt vs per-token generation latency	→	**Prompt-token disaggregation**
Inefficient usage of GPU memory	→	**Microbatch swapping**
Statefulness and inefficient failure handling	→	**KV cache replication and fault-handling mechanism**

DéjàVu github repo: