Orca: A Distributed Serving System for Transformer Based Generative Models

Pratheek D Souza Rebello Yichen Zheng Guanyi Xu Kexun Zhang

In class Presentation on March 20th, 2024

Outline

- Introduction & Related Work
- Challenges & Solutions
- Orca Design
- Evaluation
- Summary & Future work

Serving LLMs is expensive

A GPT-3 175B instance requires **2 VMs**, each with 8 NVIDIA A100 GPUs on Azure.

Each VM costs **$27.197/hour**.

2VMs cost **$476491.44/year**.

If we need to host 400 instances, **$190.6M / year**.

Price in 2022, GPT-3 models are now deprecated on Azure.

Orca Improves Throughput by 36.9x

Compared to NVIDIA FasterTransformer, Orca improves throughput by **36.9x** at the same level of latency on GPT-3 175B.

Inference of Autoregressive LMs

Multi-iteration characteristic

Unlike BERT or ResNet, they generate one token at a time.

Initiation phase (1st iteration)

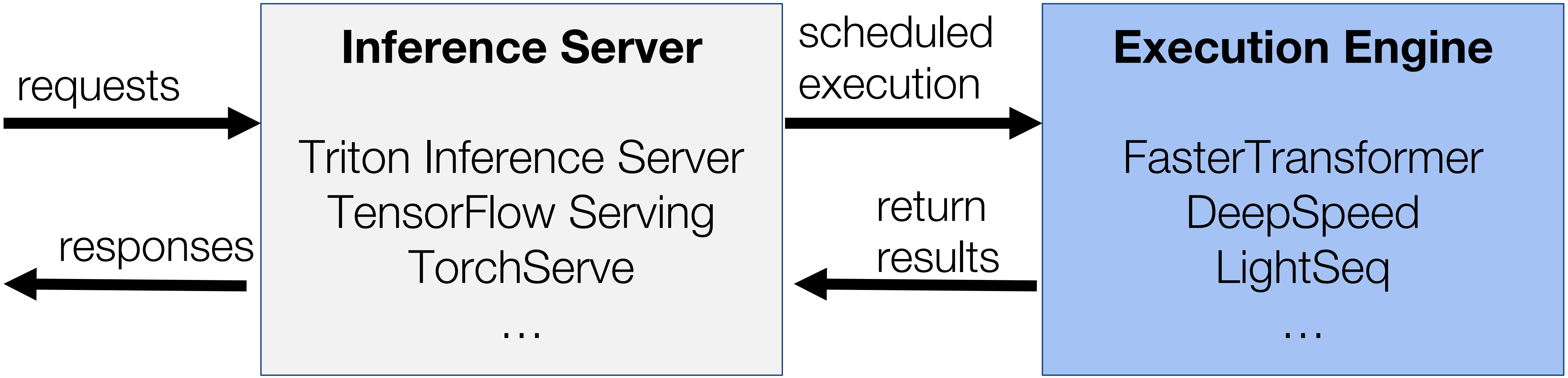
Process all input tokens (prefix/prompt) at once.

Increment phase (2nt to last iteration)

Process a single token generated from the prev. iteration.

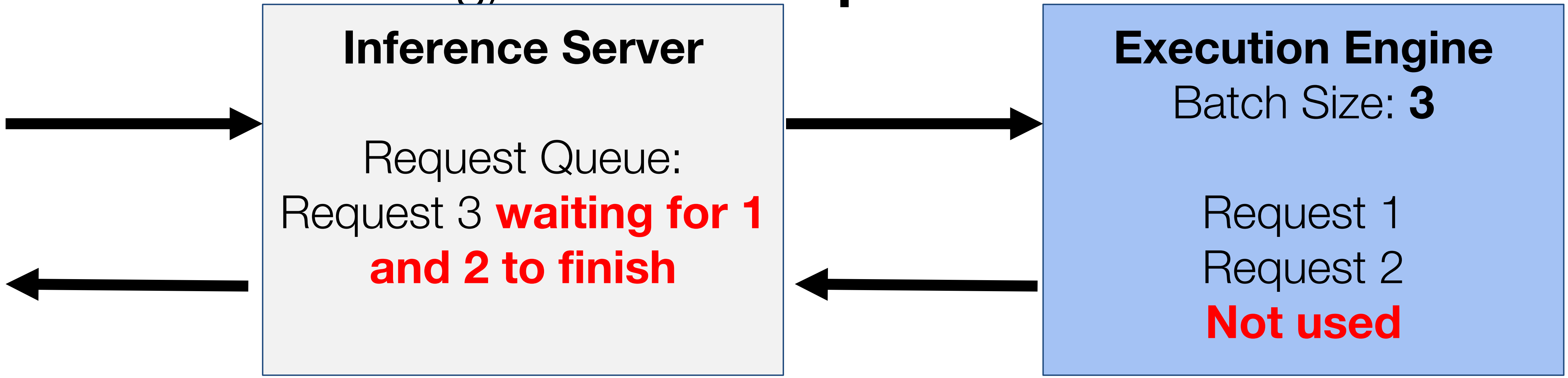
Use attention keys and values of all prev. tokens

Serving of Autoregressive LMs



Current Systems are Request-Level

Existing execution engines (FasterTransformer, LightSeq, etc.) assumes **request-level scheduling**.
Existing inference servers (Triton Inference Server, TensorFlow Serving) assumes **request-level execution**.



Orca Allows Iteration-Level Scheduling

Orca maintains a request pool.

Each time we take as many requests as possible from the pool and run them for **one iteration**.

Newly arrived requests wait for **only one iteration** when batch size allows.

This makes batching harder.

Iteration-Level Scheduling is Harder to Batch

Batching is only applicable when

requests are in the **same phase** (initiation or increment)

requests have the **same length**

X1: I think

X2: I love

X3: A person

X1: [2, H]

X2: [2, H]

X3: [2, H]

Coalesce

Batch: [3, 2, H]

Iteration-Level Scheduling is Harder to Batch

Batching is only applicable when

- requests are in the **same phase** (initiation or increment)
- requests have the **same length**

X1: I think this

X2: I love you

X3: A person is

X1: [1, H]

X2: [1, H]

X3: [**3**, H]

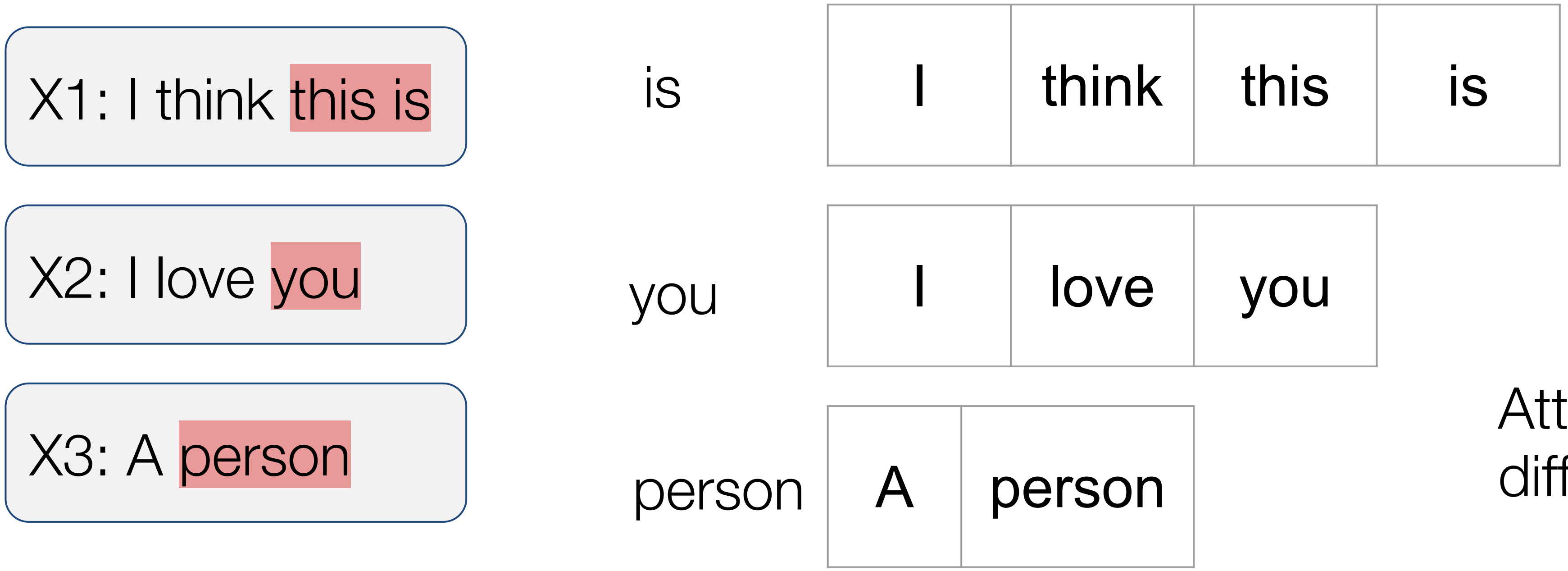
Can't Coalesce →

Iteration-Level Scheduling is Harder to Batch

Batching is only applicable when

requests are in the **same phase** (initiation or increment)

requests have the **same length**



Attention matrices have different shapes.

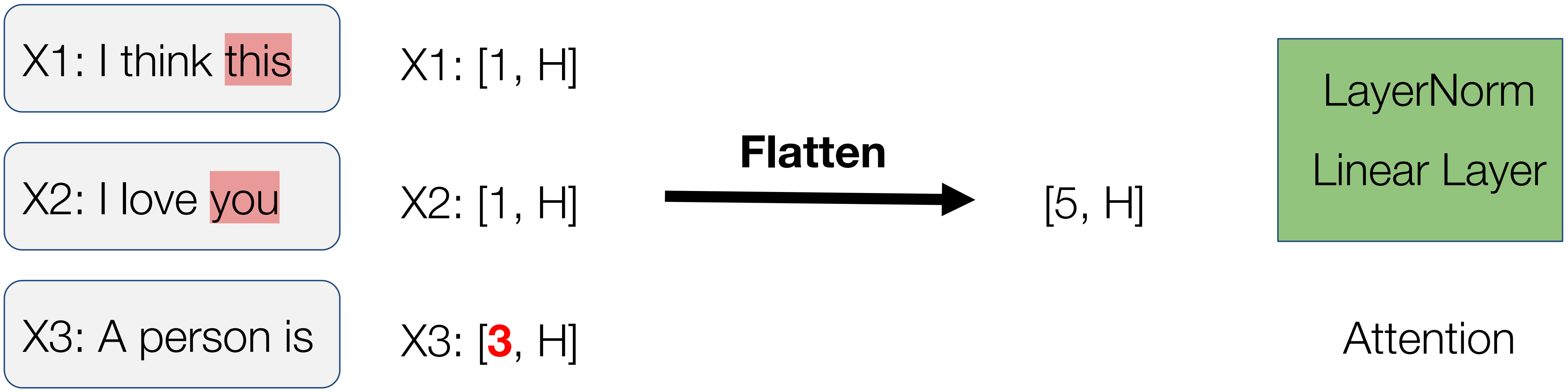
Orca is Inspired by BatchMaker

BatchMaker is a serving system for RNNs that perform scheduling and batching at cell-level.

Batching transformers at iteration level is harder because different requests have different number of keys and values, which isn't the case for RNNs.

Orca Uses Selective Batching

Not all operations are incompatible with irregularly shaped tensors.
Matrix multiplication and layer normalization can be batched, because they do not distinguish different requests.



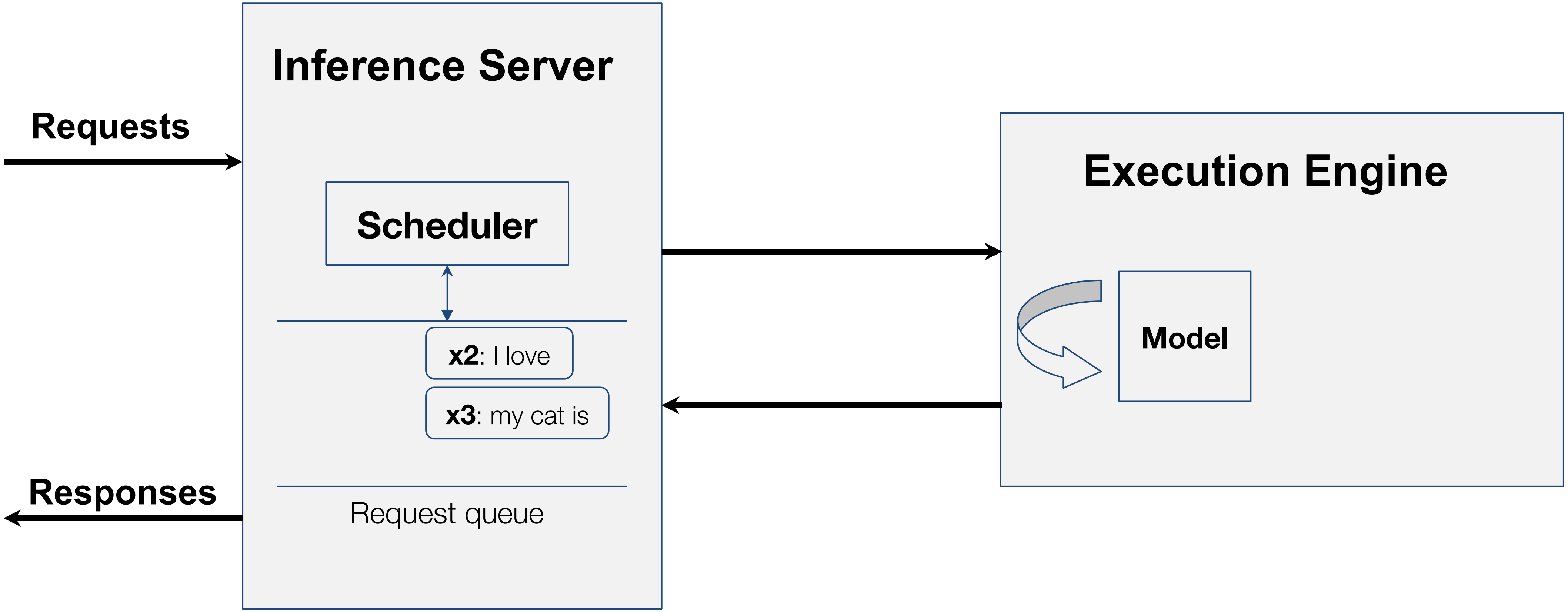
Outline

- Introduction & Related Work
- Challenges & Solutions
- Orca Design
- Evaluation
- Summary & Future work

Example

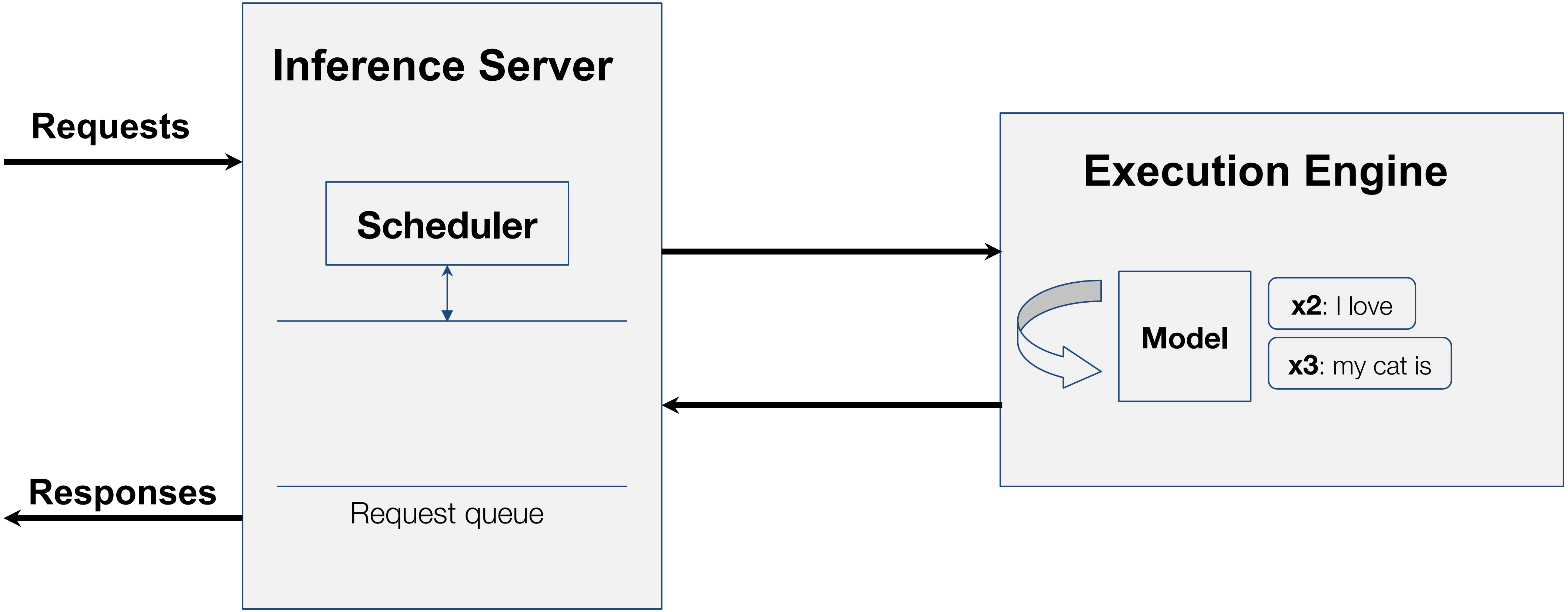
Given input	iter:1	iter:2	iter:3	iter:4	iter:5
x1: I think	x1: I think this	x1: I think this idea	x1: I think this idea is	x1: I think this idea is bad	x1: I think this idea is bad <eos>
x2: I love	x2: I love the	x2: I love the asian	x2: I love the asian food	x2: I love the asian food <eos>	
x3: my cat is	x3: my cat is so	x3: my cat is so cute	x3: my cat is so cute <eos>		
x4: what is	x4: what is your	x4: what is your opinion	x4: what is your opinion <eos>		

C1: Request Scheduling



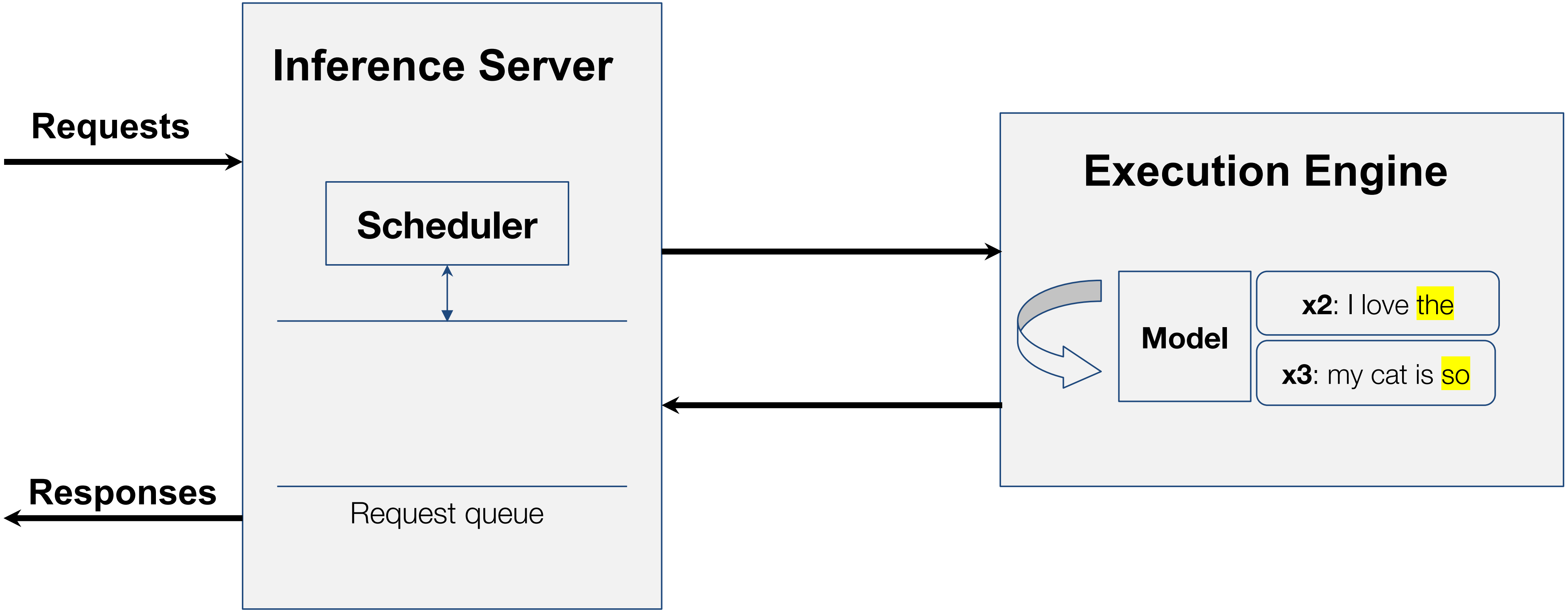
* Assume we set the max_batch_size = 3

Timestep 1



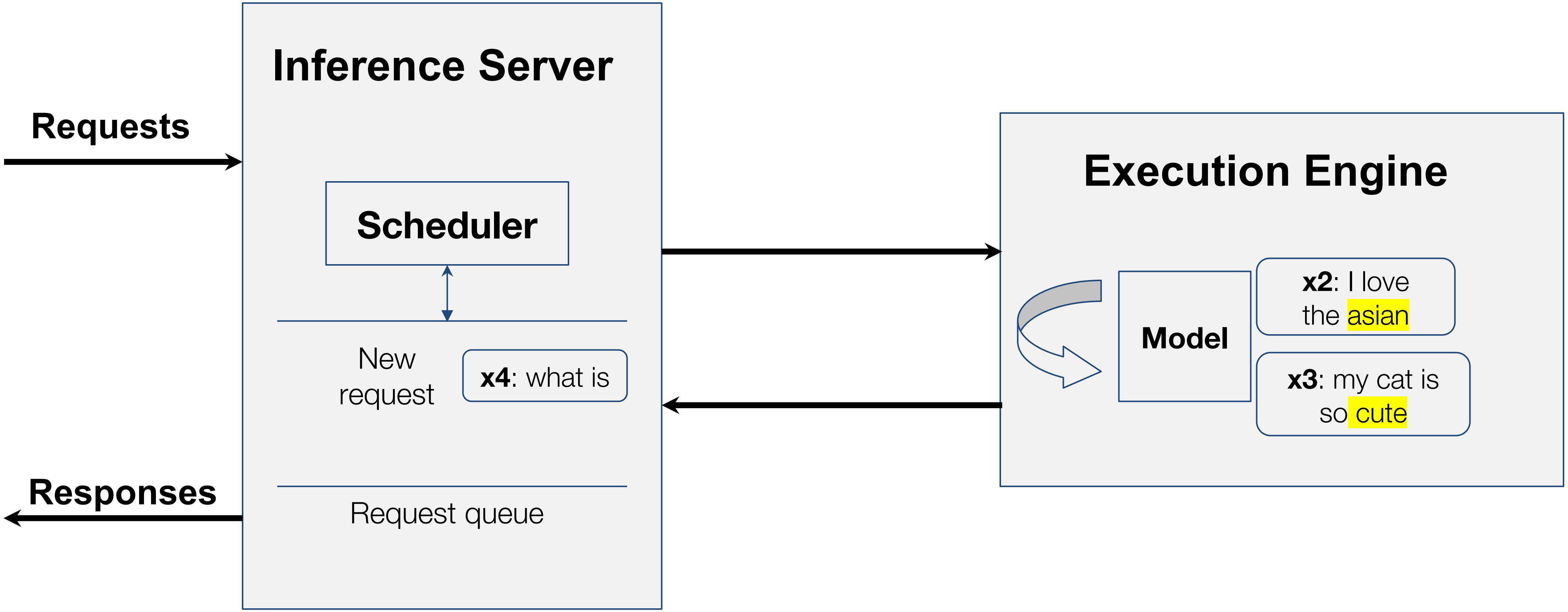
*** Assume we set the max_batch_size = 3**

Timestep 2



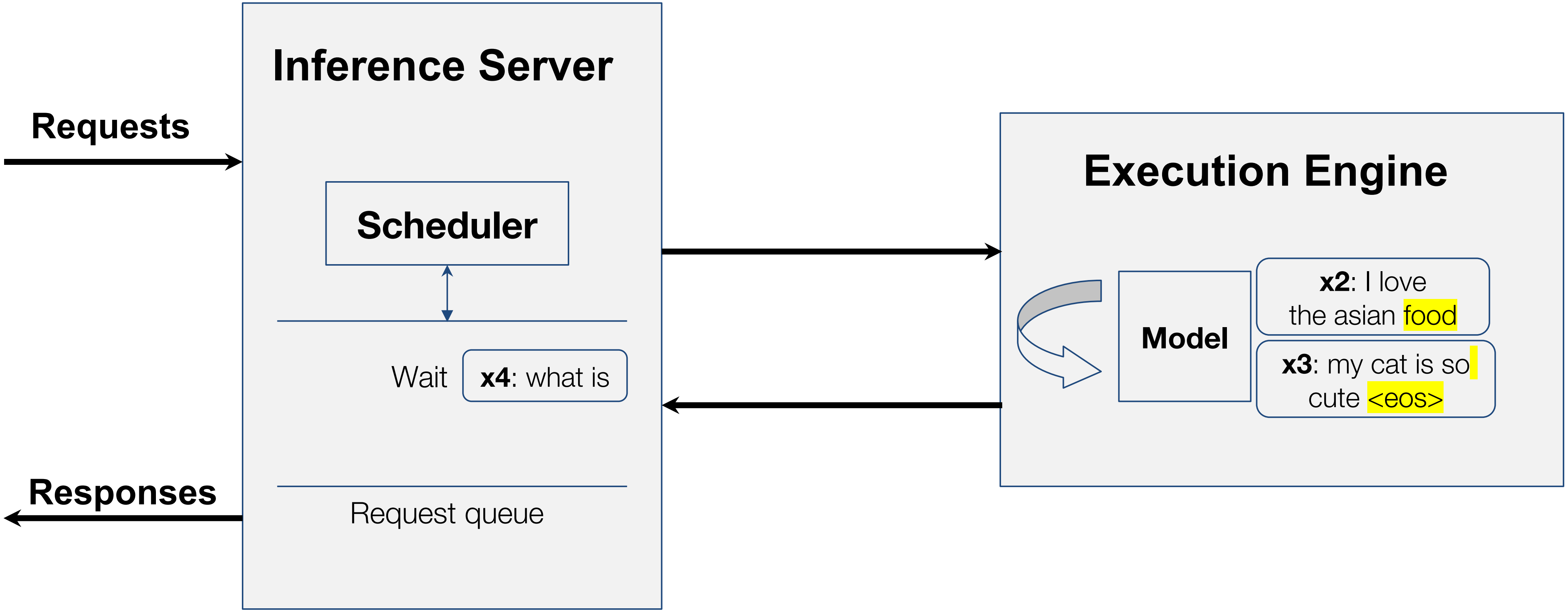
* **Assume we set the max_batch_size = 3**

Timestep 3



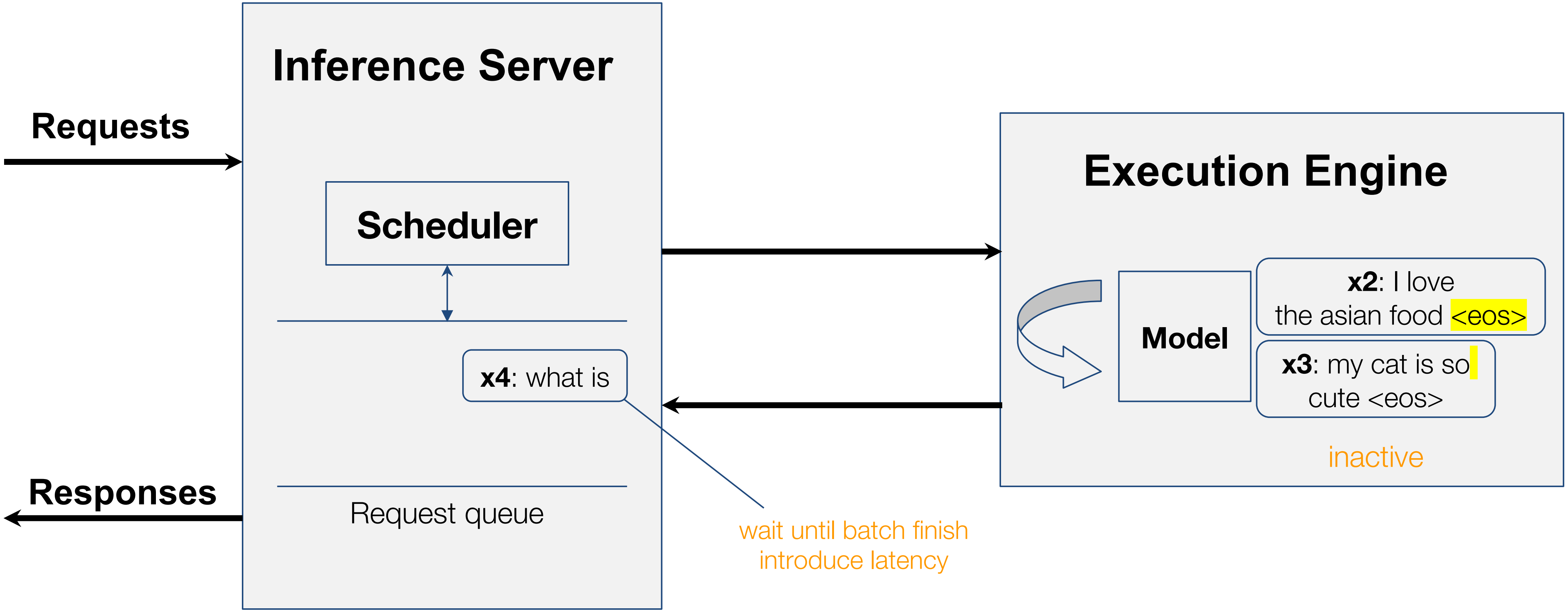
*** Assume we set the max_batch_size = 3**

Timestep 4



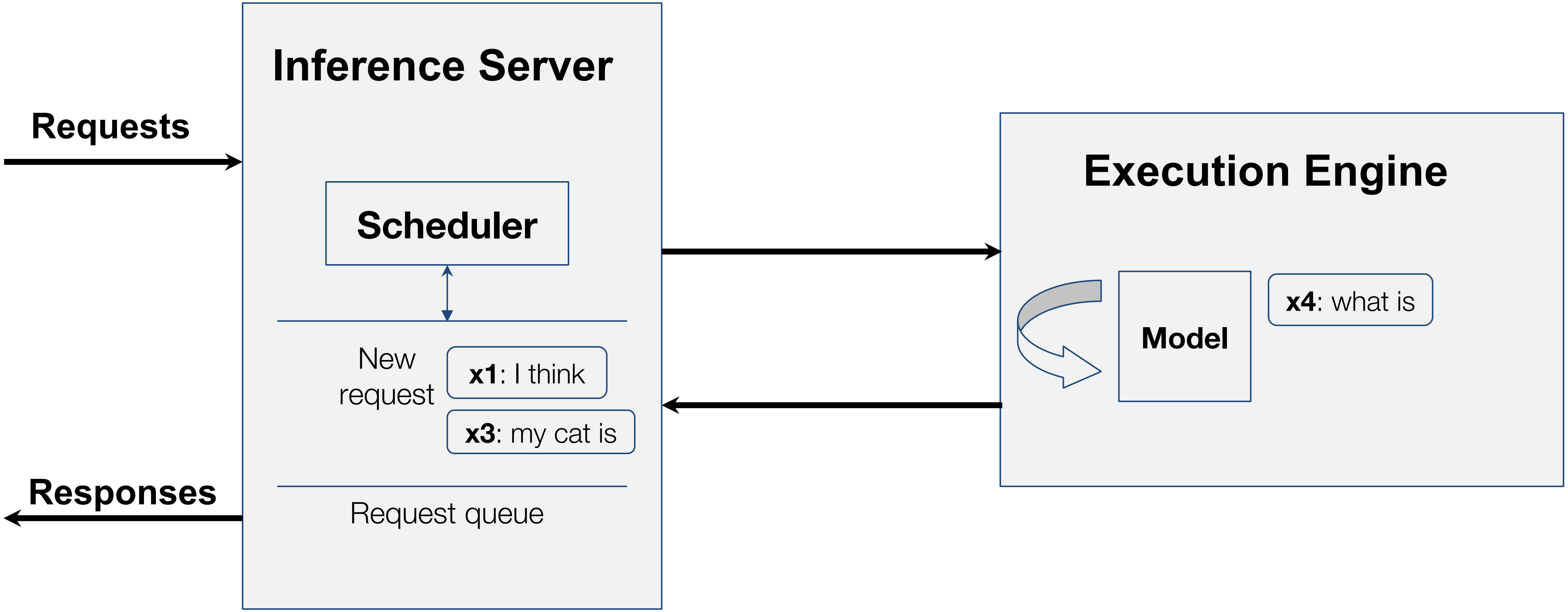
* **Assume we set the max_batch_size = 3**

Timestep 5



*** Assume we set the max_batch_size = 3**

Timestep 6



* Assume we set the max_batch_size = 3

Drawback: High Latency

1. Requests in same batch may have extra computation due to other “active” requests.

2. Newly arrived requests wait until all requests in the current batch have finished.

S1: Iteration-Level Scheduling

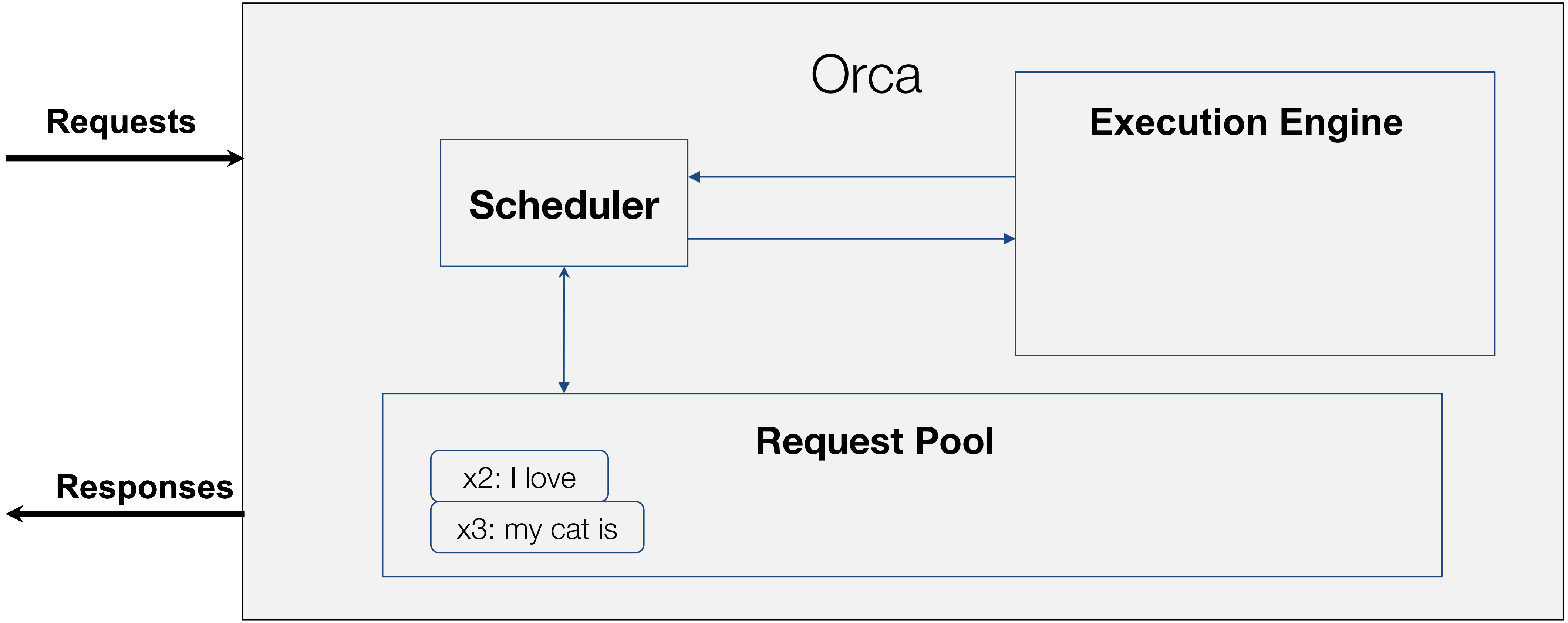
Requests

Responses

Schedule Procedure:

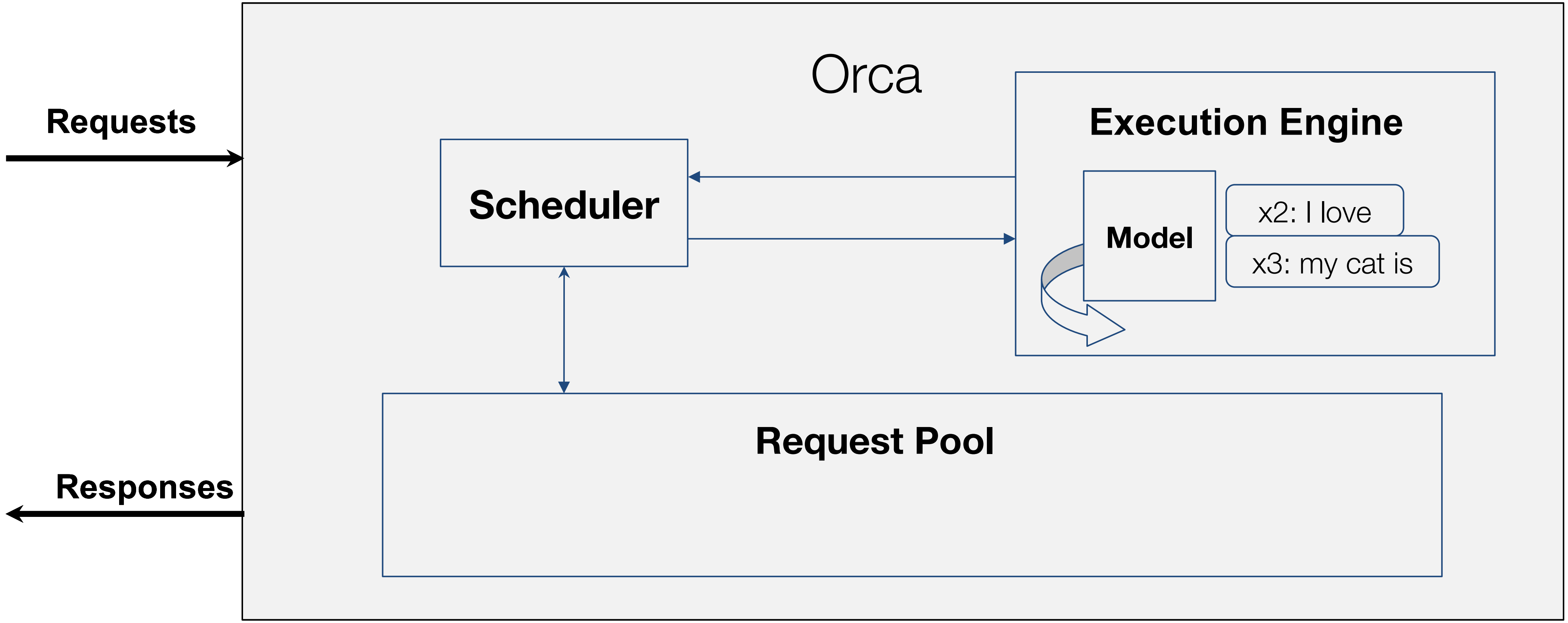
1. Select requests
2. Invoke engine to execute one iteration
3. Receive result
4. Make the decision

Timestep 0



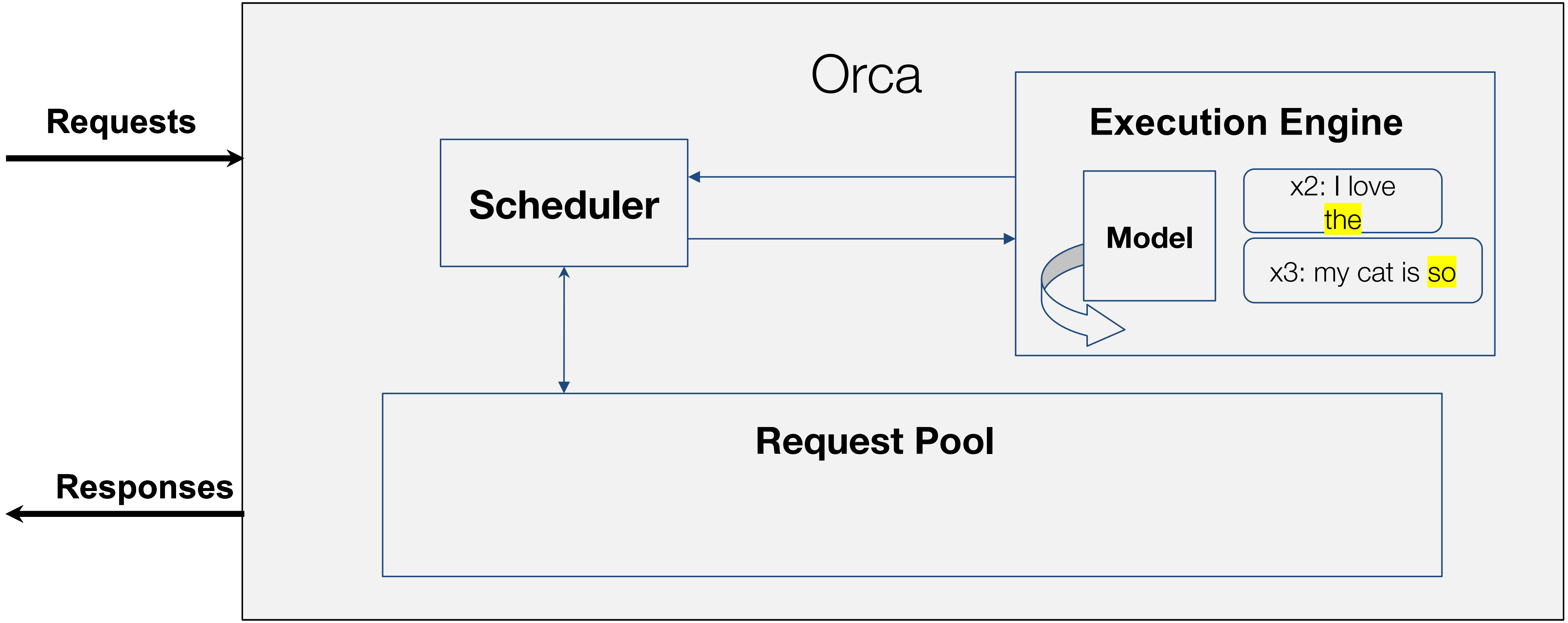
*** Assume we set the max_batch_size = 3**

Timestep 1



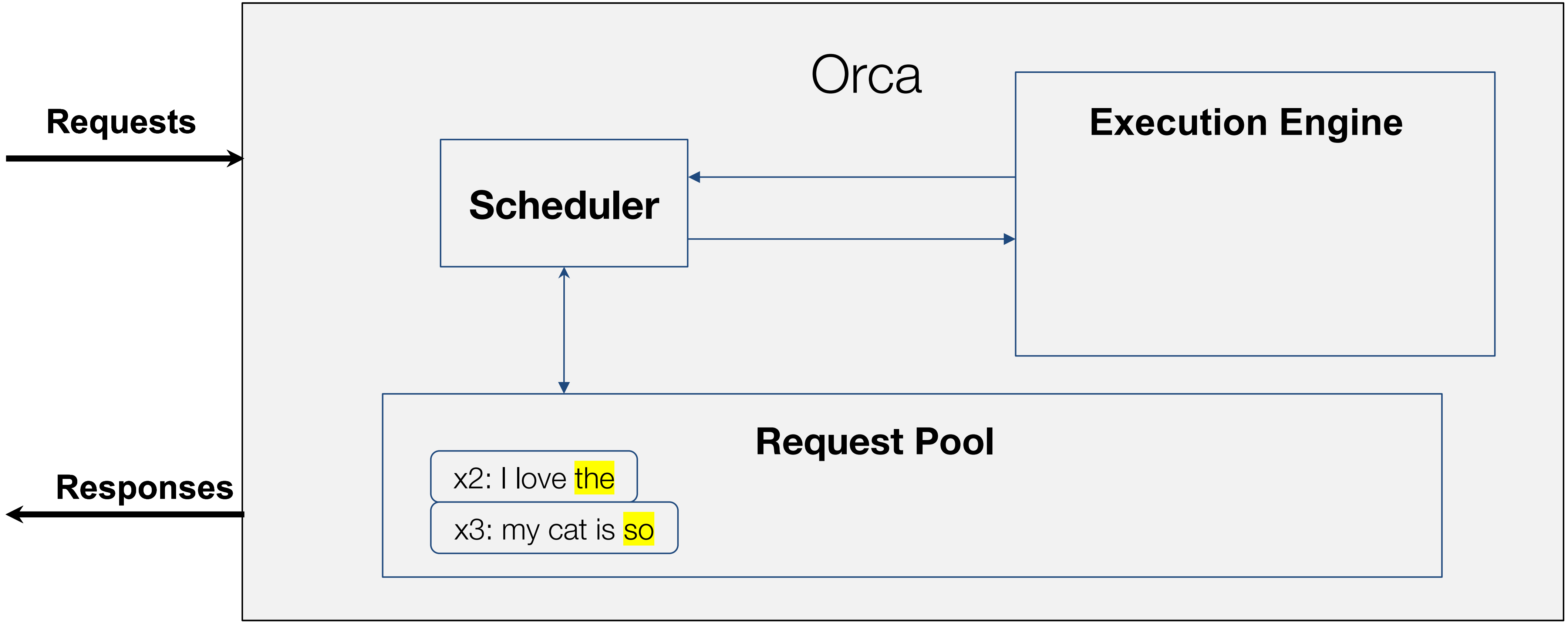
*** Assume we set the max_batch_size = 3**

Timestep 2



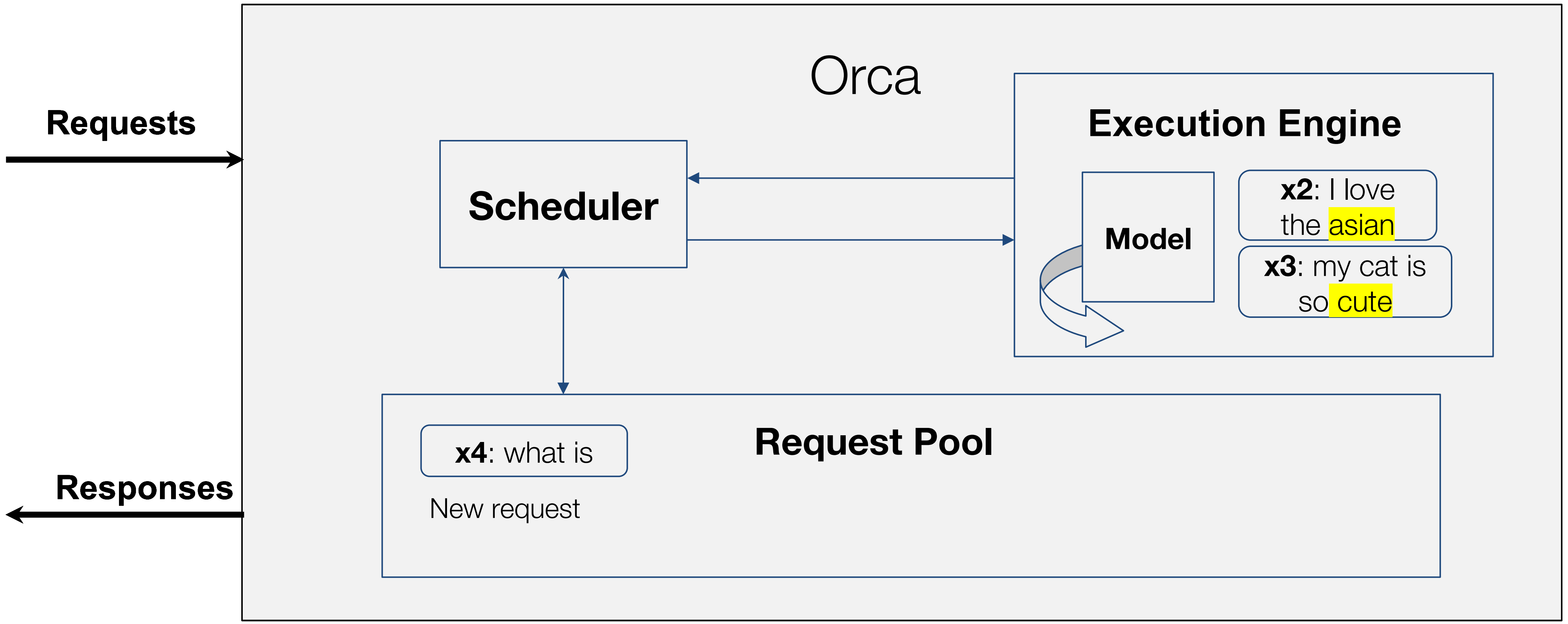
* Assume we set the max_batch_size = 3

Timestep 2



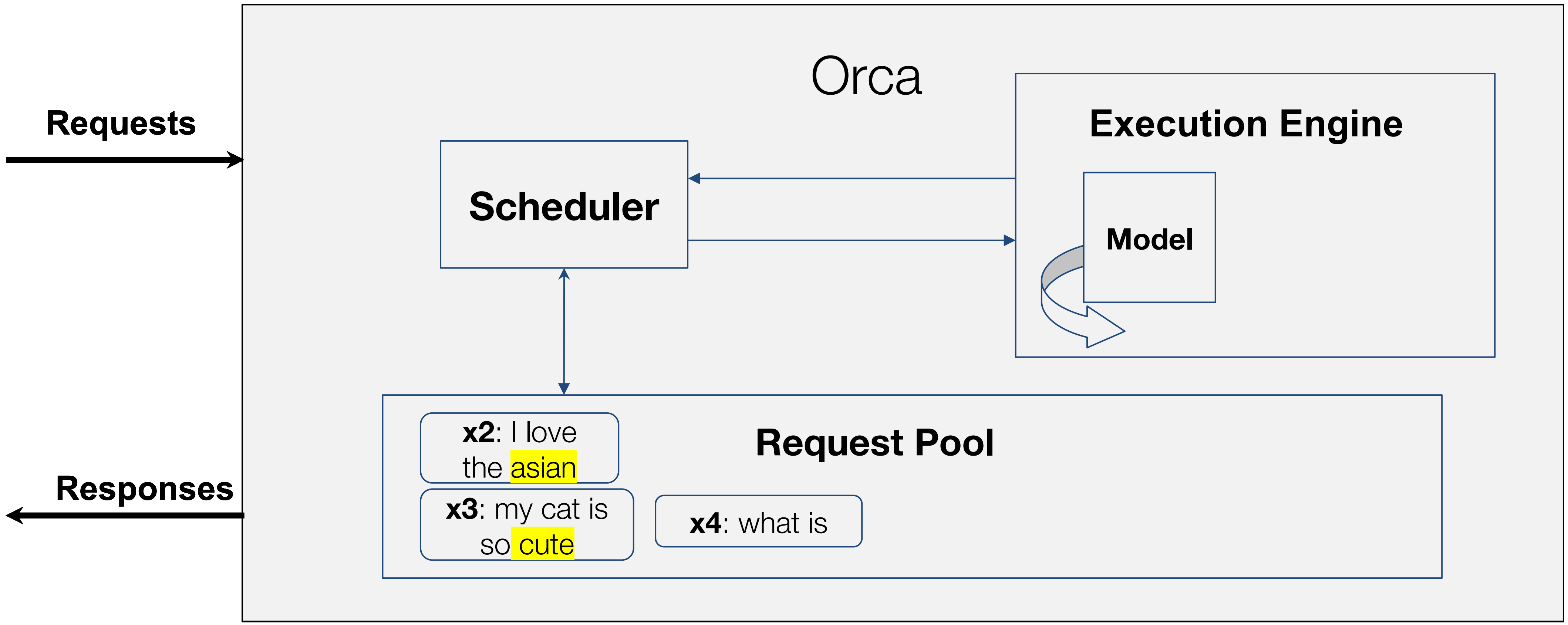
*** Assume we set the max_batch_size = 3**

Timestep 3



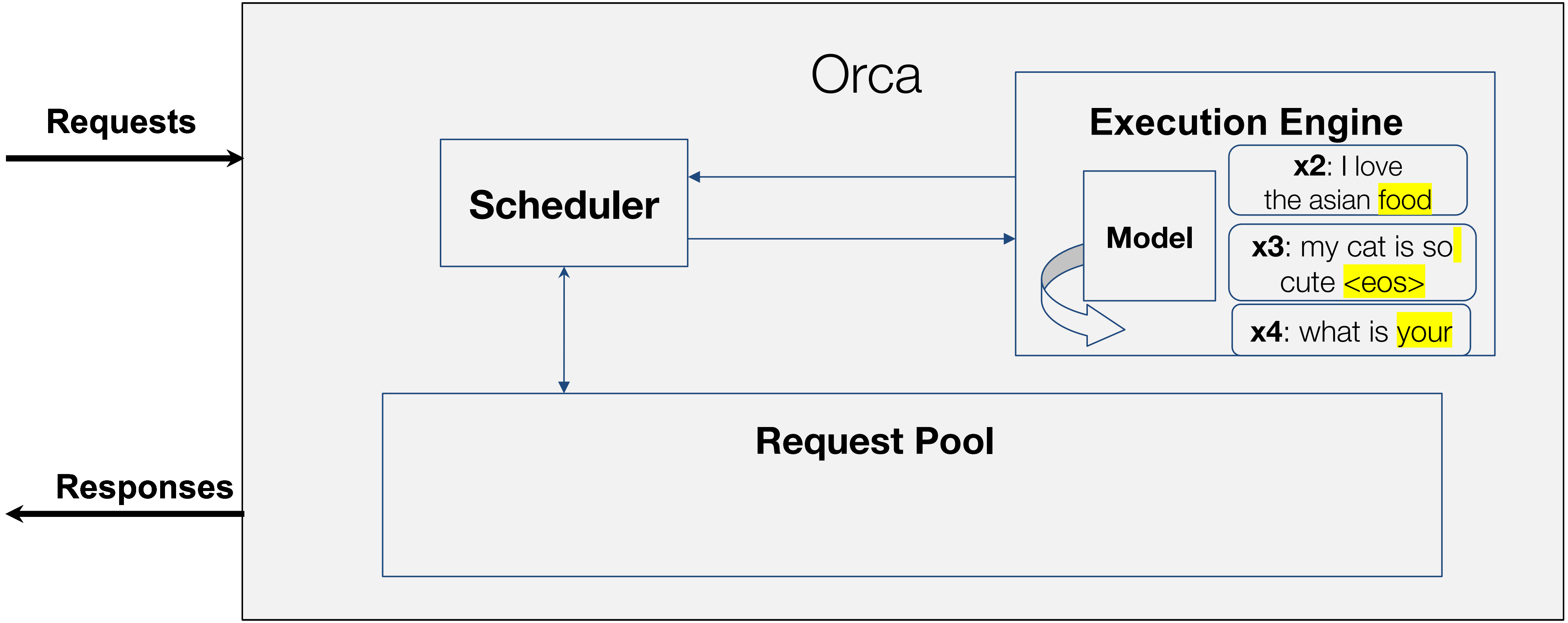
*** Assume we set the max_batch_size = 3**

Timestep 3



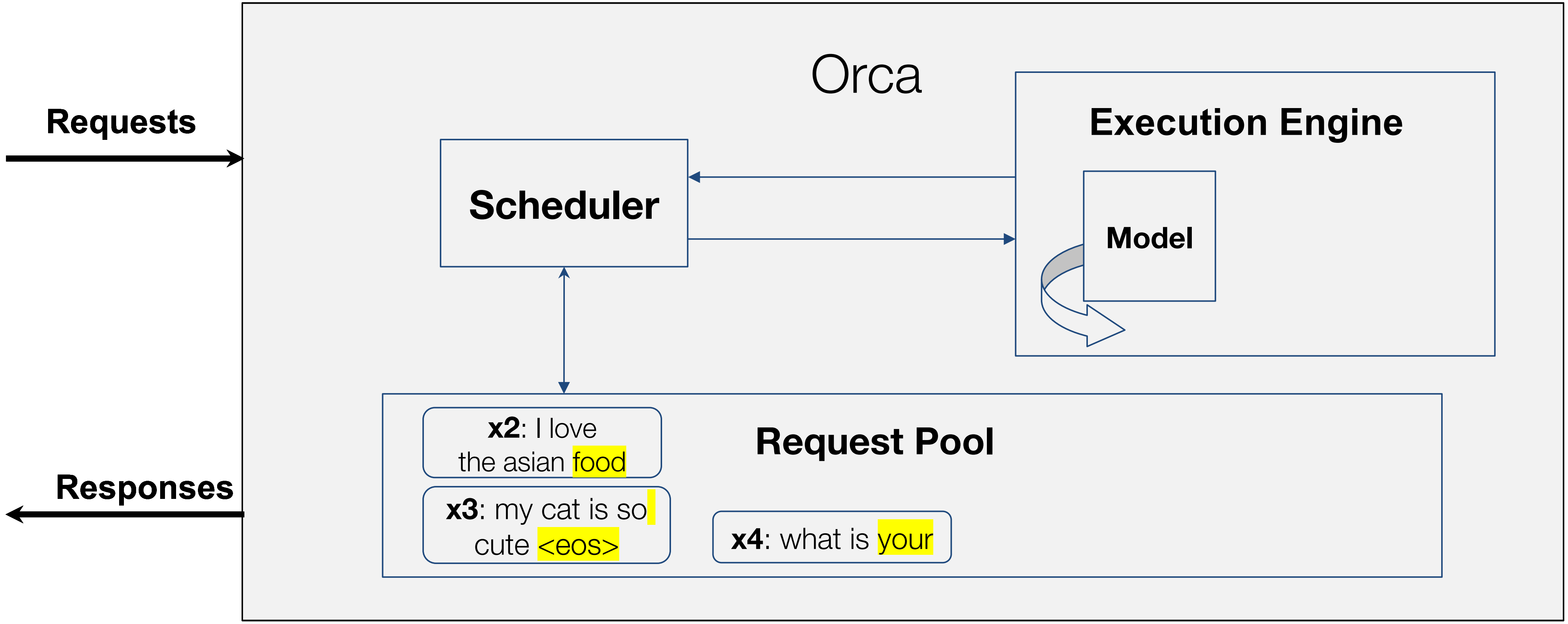
* Assume we set the max_batch_size = 3

Timestep 4



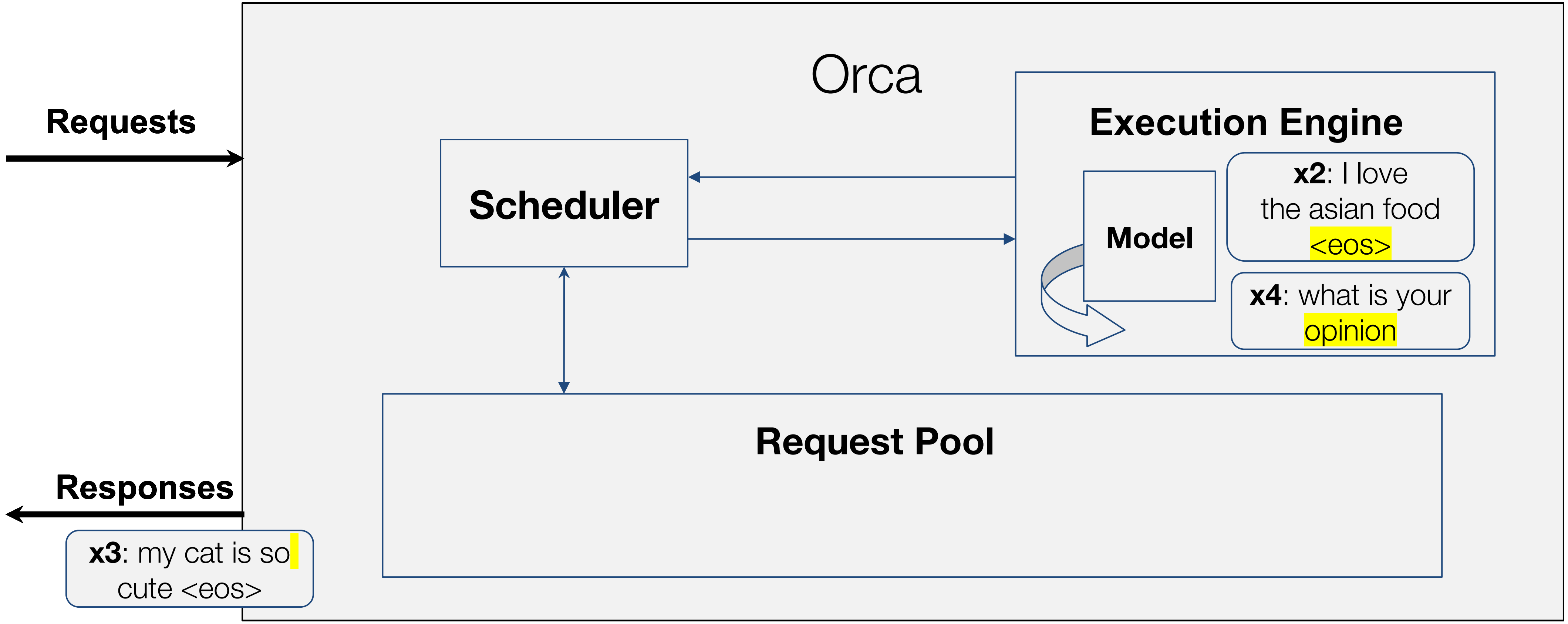
*** Assume we set the max_batch_size = 3**

Timestep 4



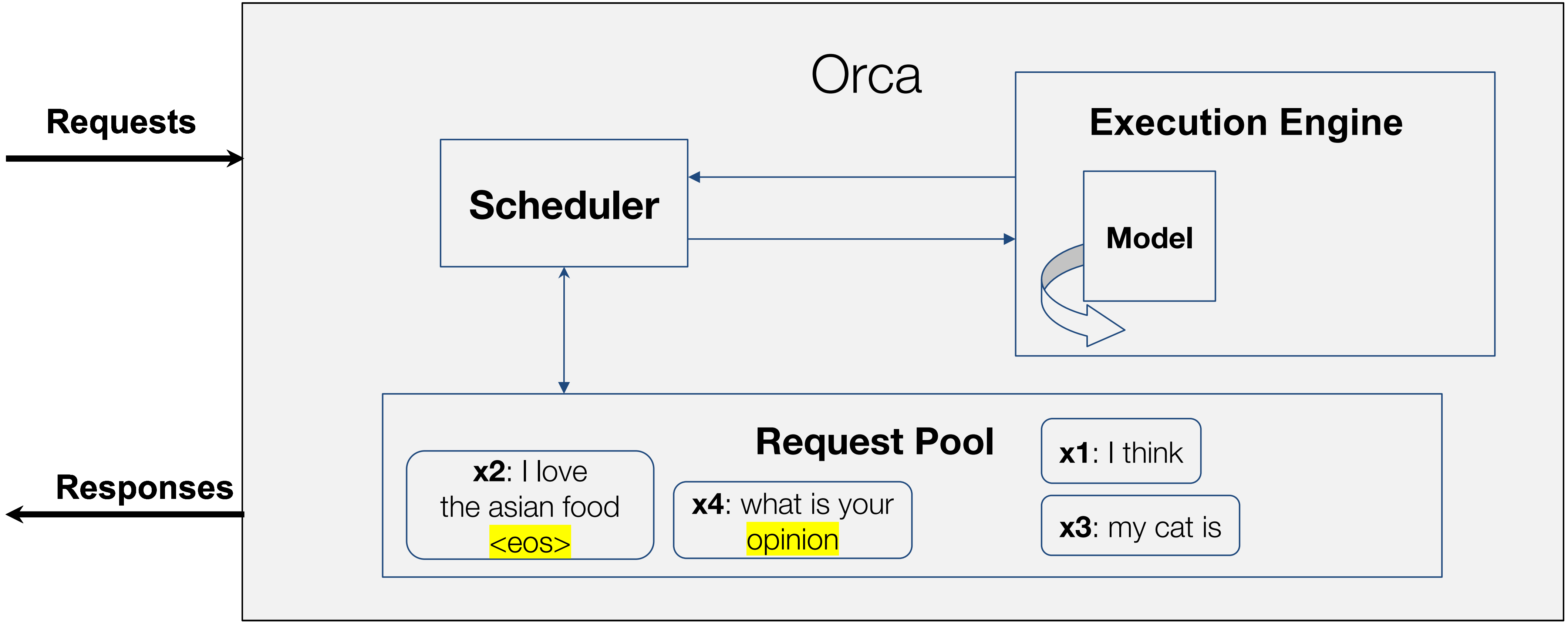
* Assume we set the max_batch_size = 3

Timestep 5



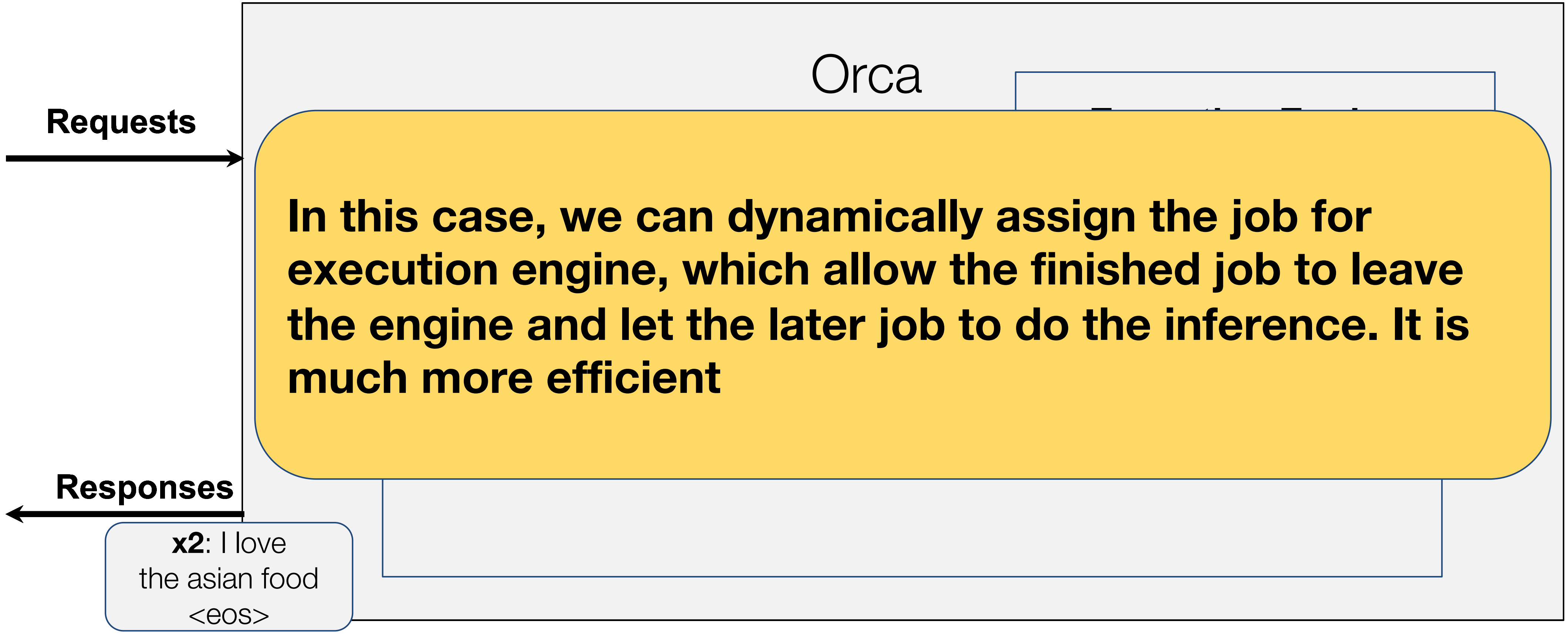
* **Assume we set the max_batch_size = 3**

Timestep 5



* Assume we set the max_batch_size = 3

Timestep 6



* **Assume we set the max_batch_size = 3**

C2: Batching

Batching is only applicable when
requests are in the **same phase** (initiation or increment)
requests have the **same length**

X1: I think this is

X2: I love you

X3: A person

Three cases cannot batch normally:

1. both requests are in the initiation phase and each has different number of input tokens
2. both are in the increment phase and each is processing a token at different index from each other
3. each request is in the different phase: initiation or increment

S2: Selective Batching

Each iteration:

Normally: input tensor shape: [L,H] $\longrightarrow$ [B,L, H] by concatenating

Selective: input tensor shape: [L,H] $\longrightarrow$ [∑L, H]

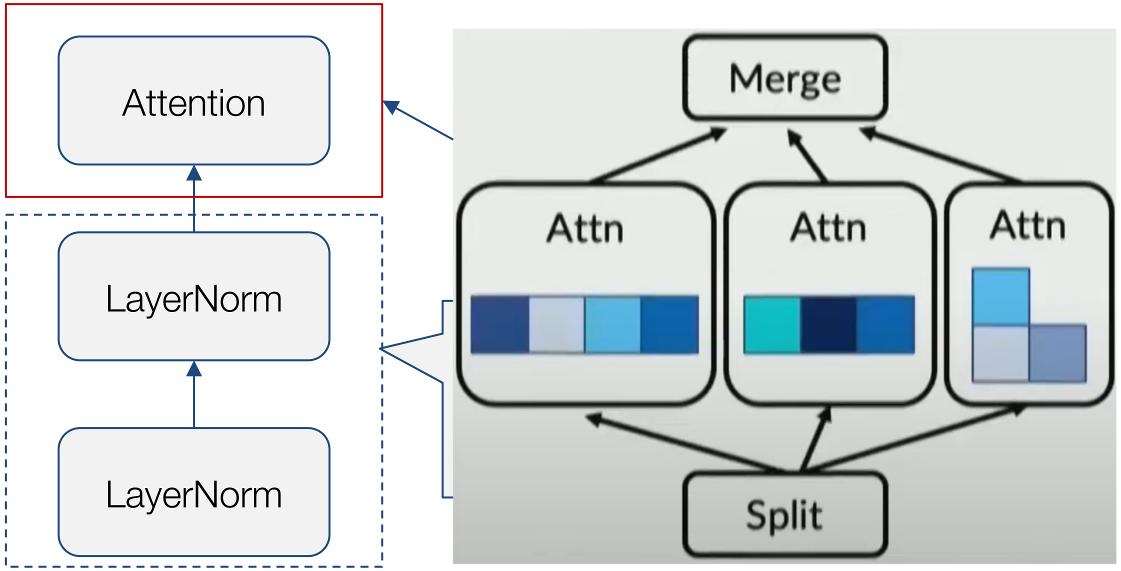
x1: I think

x2: I love the

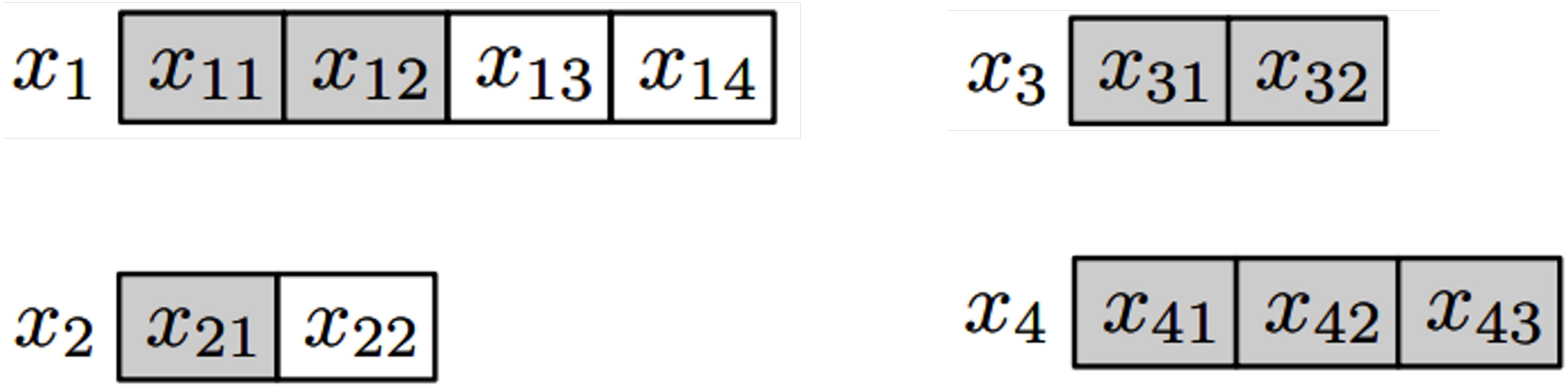
x1 tensor: [2, H] (init)

x2 tensor: [1, H] (incre)

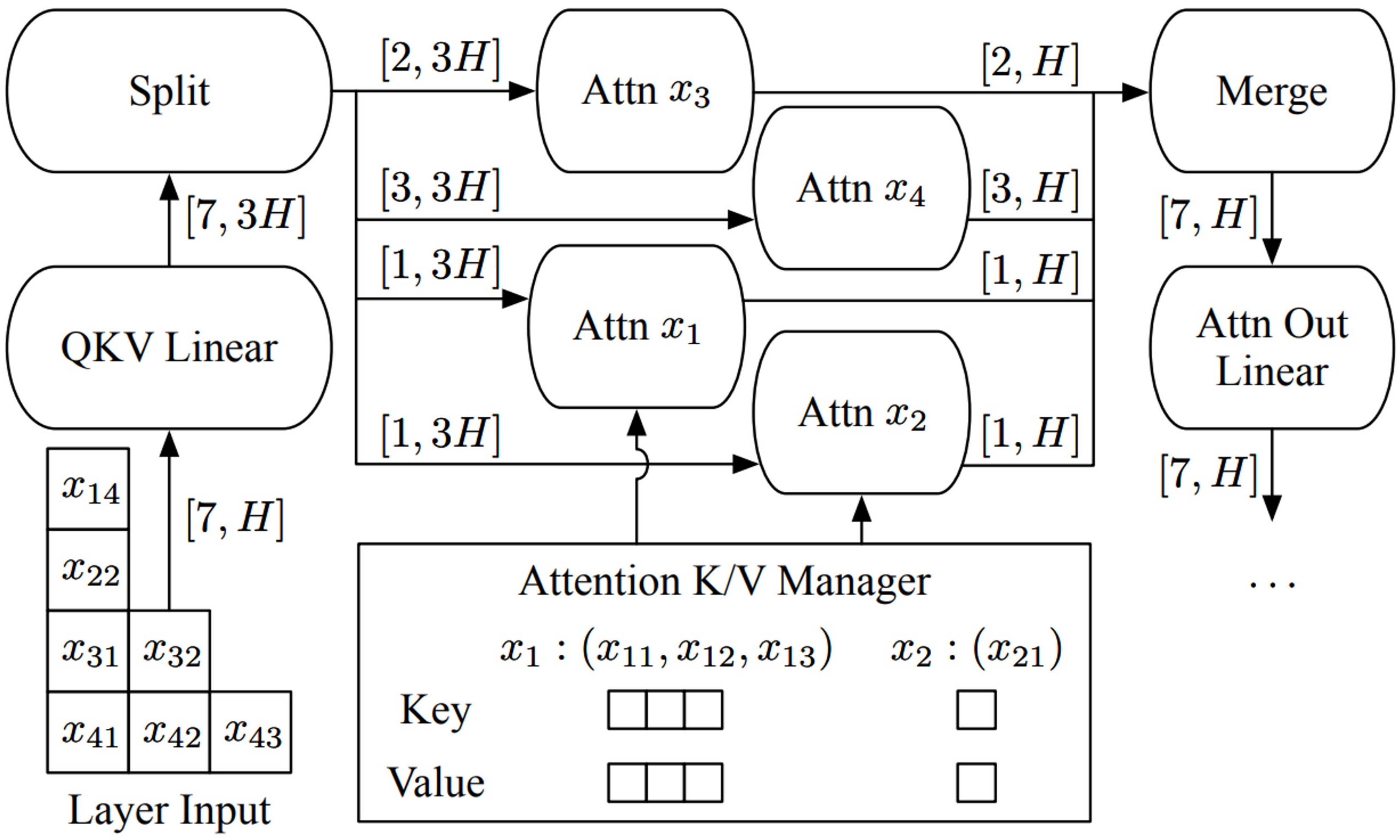
select_batch: **[3, H]** $\longrightarrow$



Example



Example

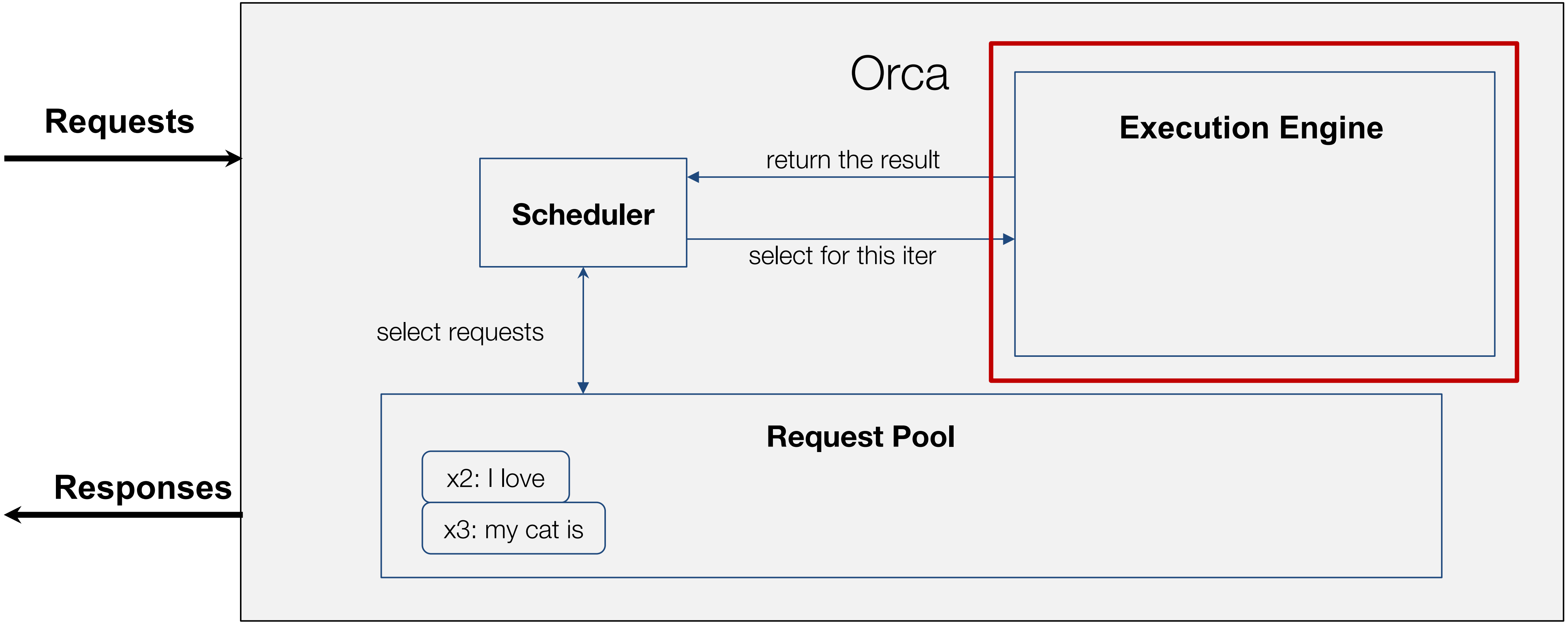


Split
[2, 3H]
Attn x_3
[2, H]
Merge
[7, 3H]
[3, 3H]
Attn x_4
[3, H]
[7, H]
[1, 3H]
[1, H]
QKV Linear
Attn x_1
Attn Out Linear
Attn x_2
[1, 3H]
[1, H]
x_{14}
x_{22}
[7, H]
[7, H]
x_{31} x_{32}
x_{41} x_{42} x_{43}
Layer Input
Attention K/V Manager
$x_1 : (x_{11}, x_{12}, x_{13})$
$x_2 : (x_{21})$
Key
Value
...

Outline

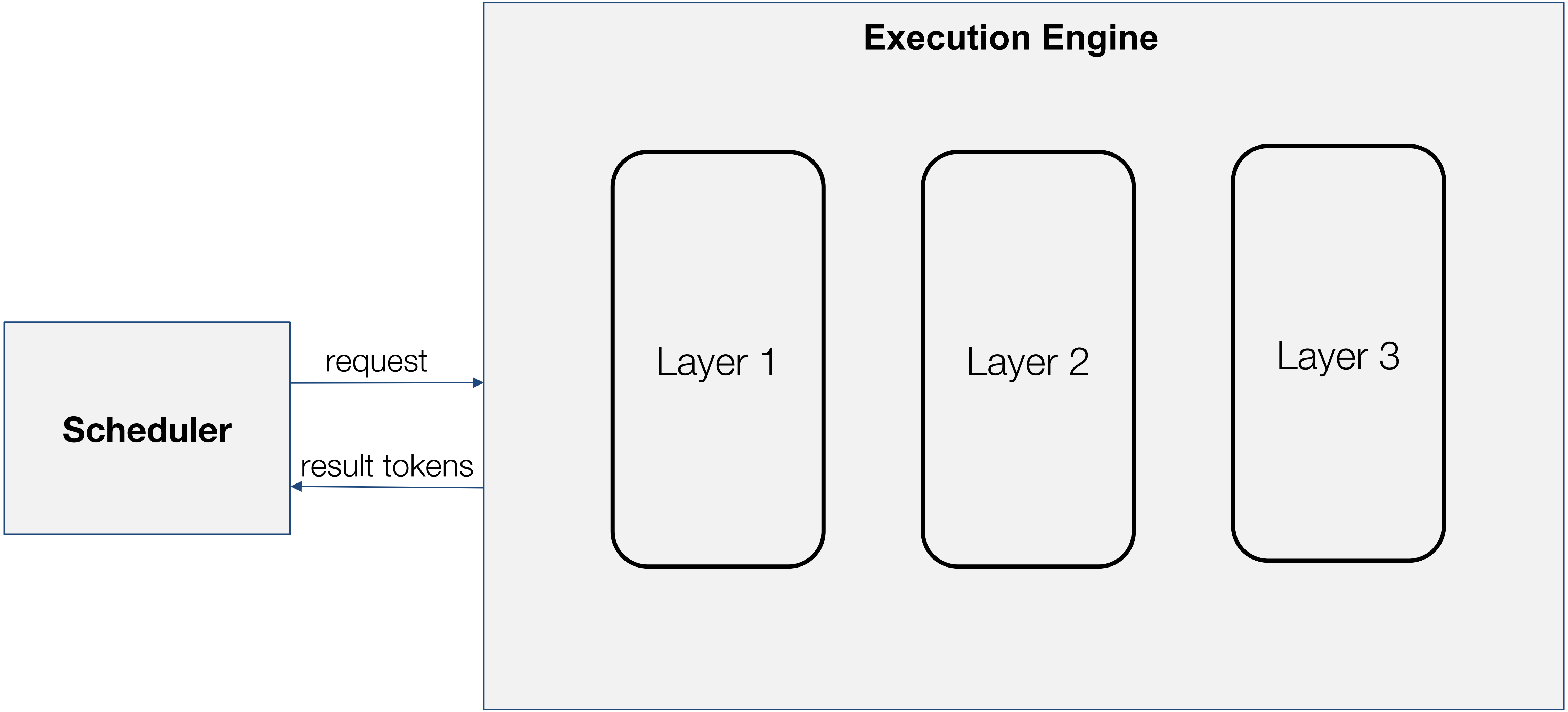
- Introduction & Related Work
- Challenges & Solutions
- Orca Design
- Evaluation
- Summary & Future work

1. Distributed Architecture



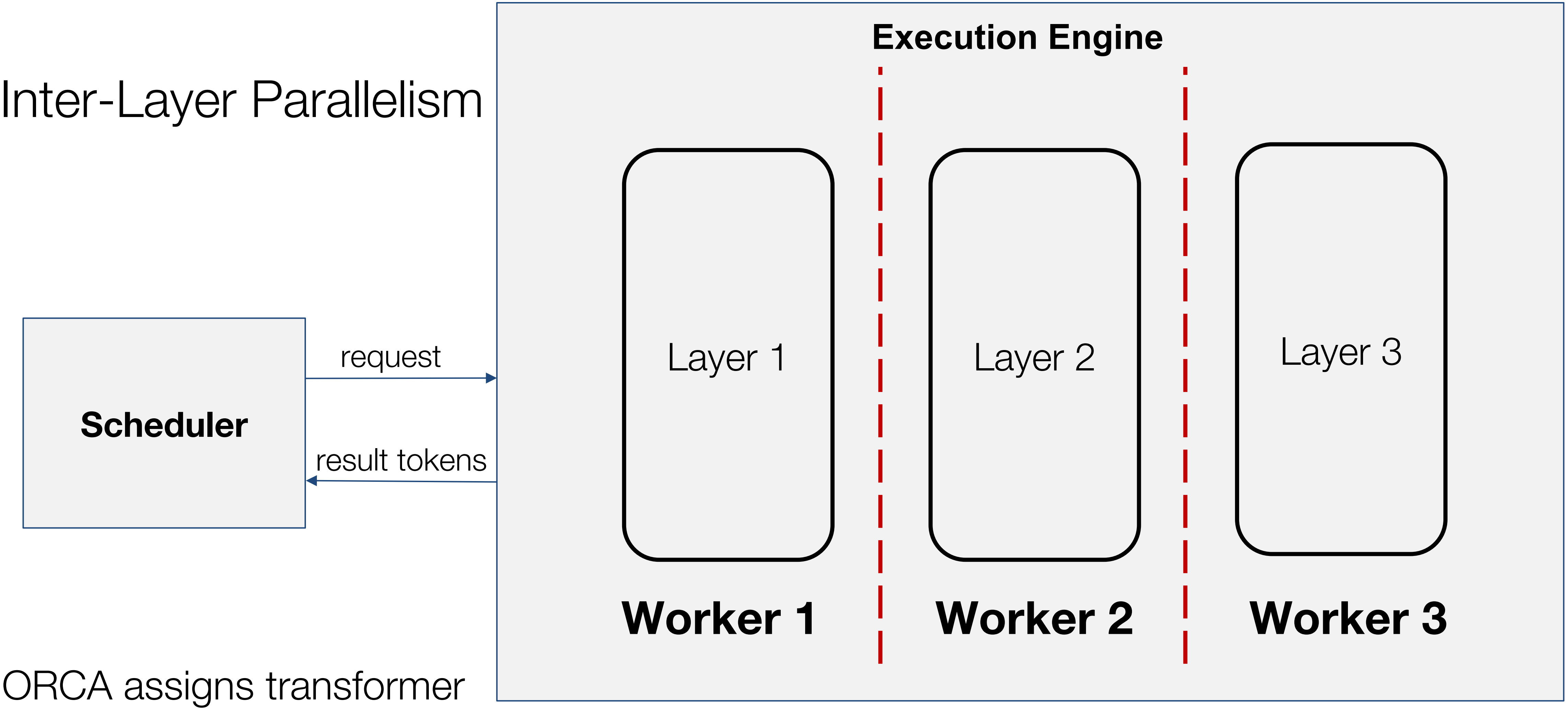
We first focus on the execution engine design

1. Distributed Architecture



1. Distributed Architecture

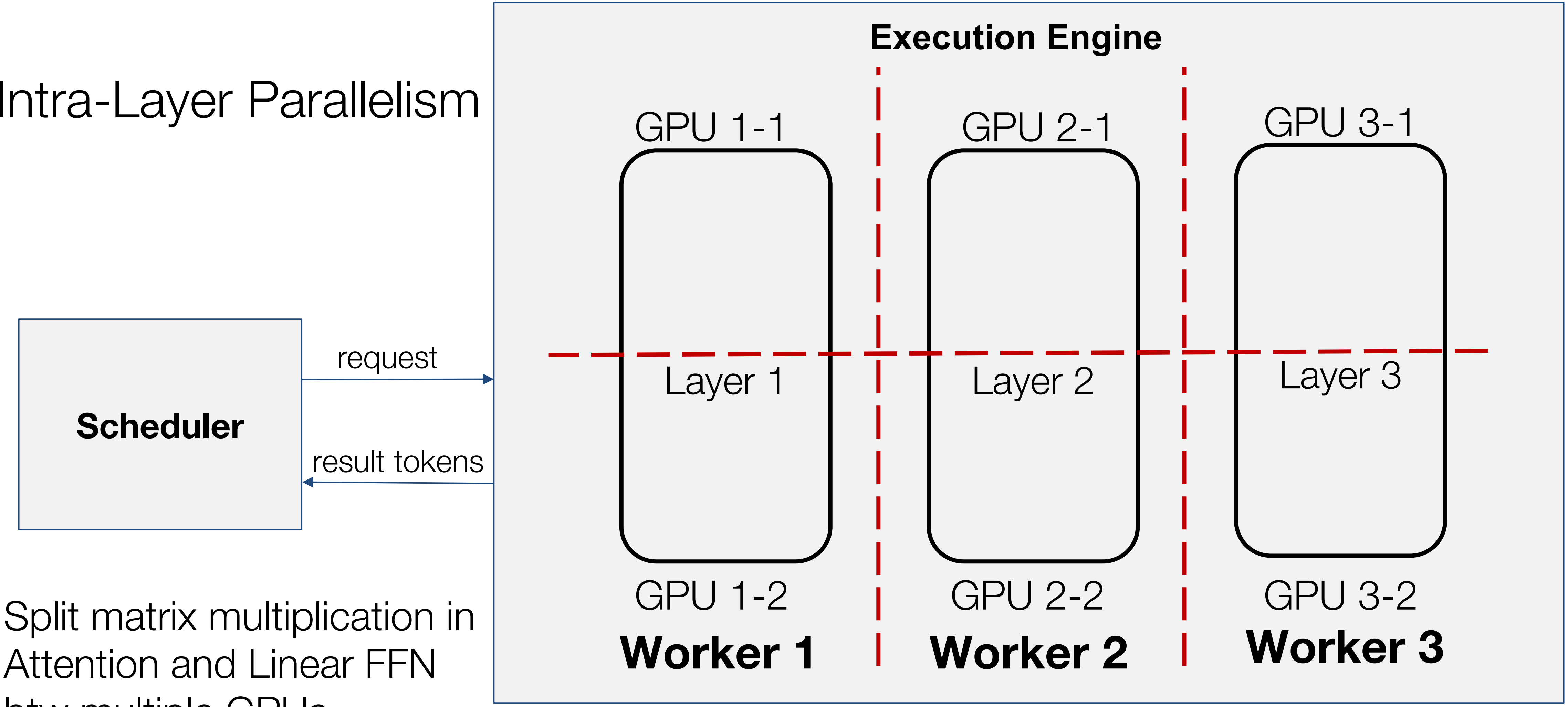
Inter-Layer Parallelism



ORCA assigns transformer layers to each worker/partition

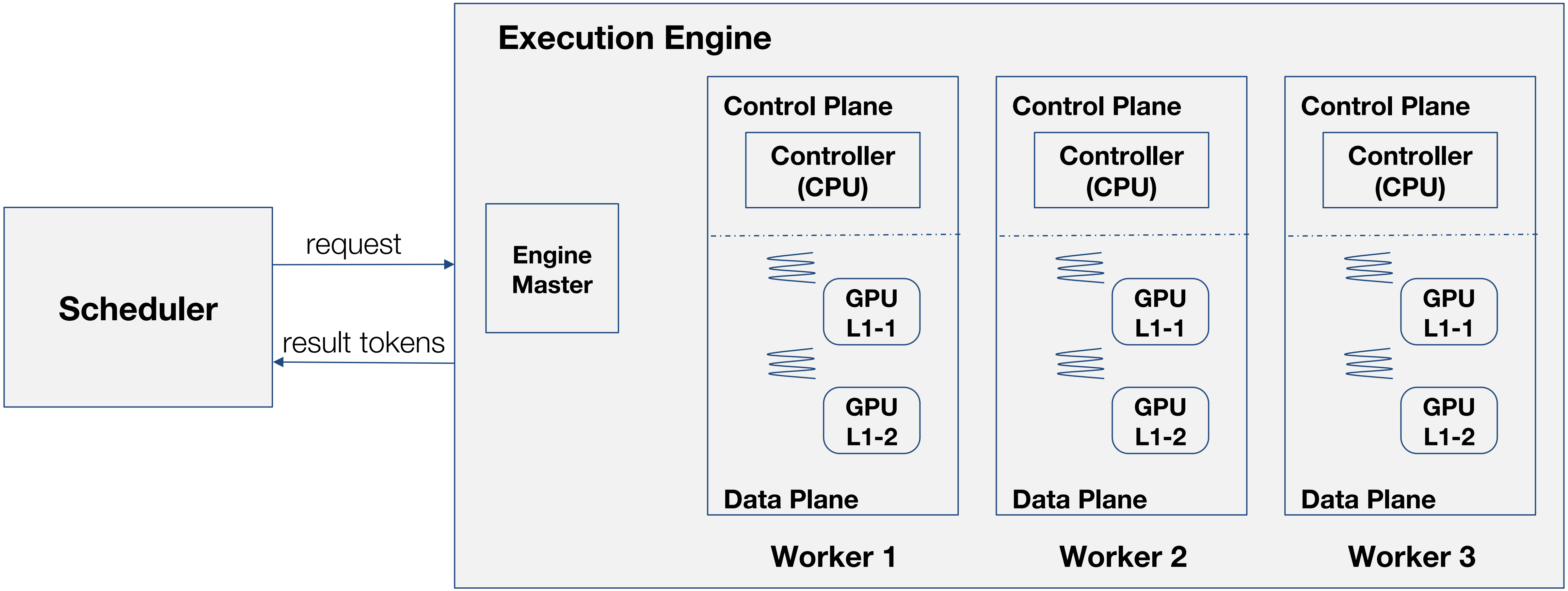
1. Distributed Architecture

Intra-Layer Parallelism

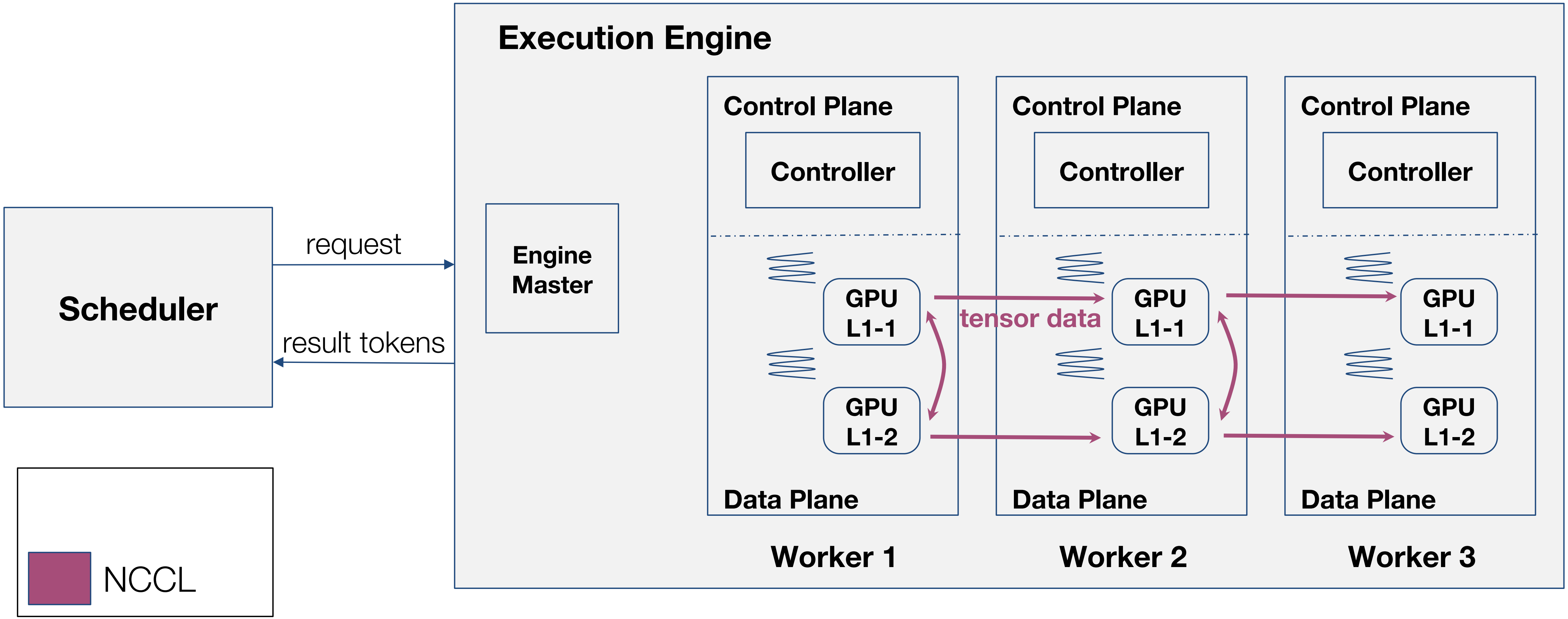


Split matrix multiplication in Attention and Linear FFN btw multiple GPUs

1. Distributed Architecture



Data Flow


Execution Engine
Scheduler
request
result tokens
Engine Master
Control Plane
Controller
GPU L1-1
GPU L1-2
tensor data
Data Plane
Worker 1
Control Plane
Controller
GPU L1-1
GPU L1-2
Data Plane
Worker 2
Control Plane
Controller
GPU L1-1
GPU L1-2
Data Plane
Worker 3
NCCL

Data Flow v/s Control Flow

Only intermediate **tensor data** is exchanged via NCCL (GPU-GPU) communication

Control messages btw engine master and worker controllers are exchanged by a separate communication channel (not involving GPUs) ex: gRPC - Remote Procedure Calls

Minimizes synch overheads btw CPU and GPU

Control Flow

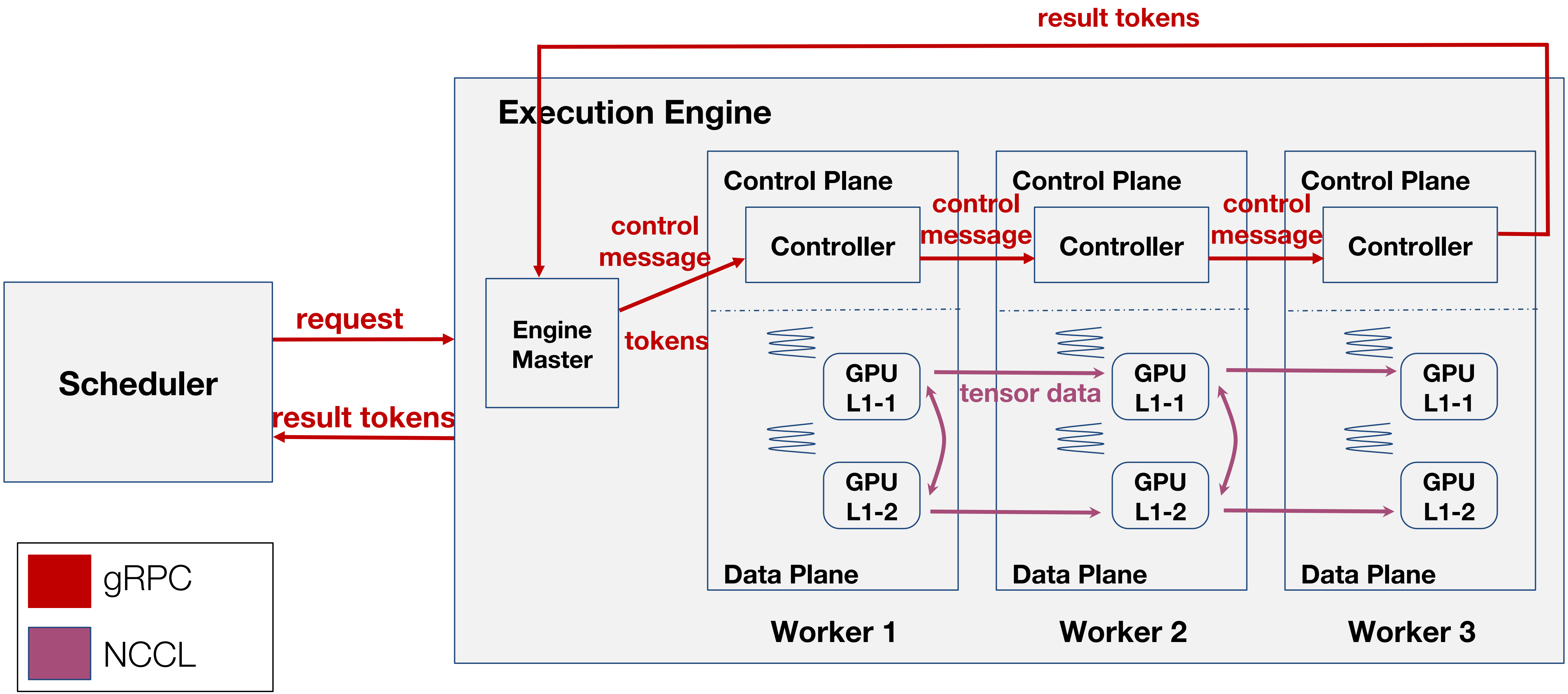
Control messages contain information about requests like *id*, *phase, token index* (for requests in increment phase), *number of input tokens* (for requests in initiation phase)

Engine Master sends the control message to **Worker 1 Controller**.

Controller passes the message to **GPU**s which start issuing kernels for computation ex: querying the key-value memory from attention manager

Meanwhile, **Worker 1 Controller** forwards control message to **Worker 2 Controller**

Control Flow



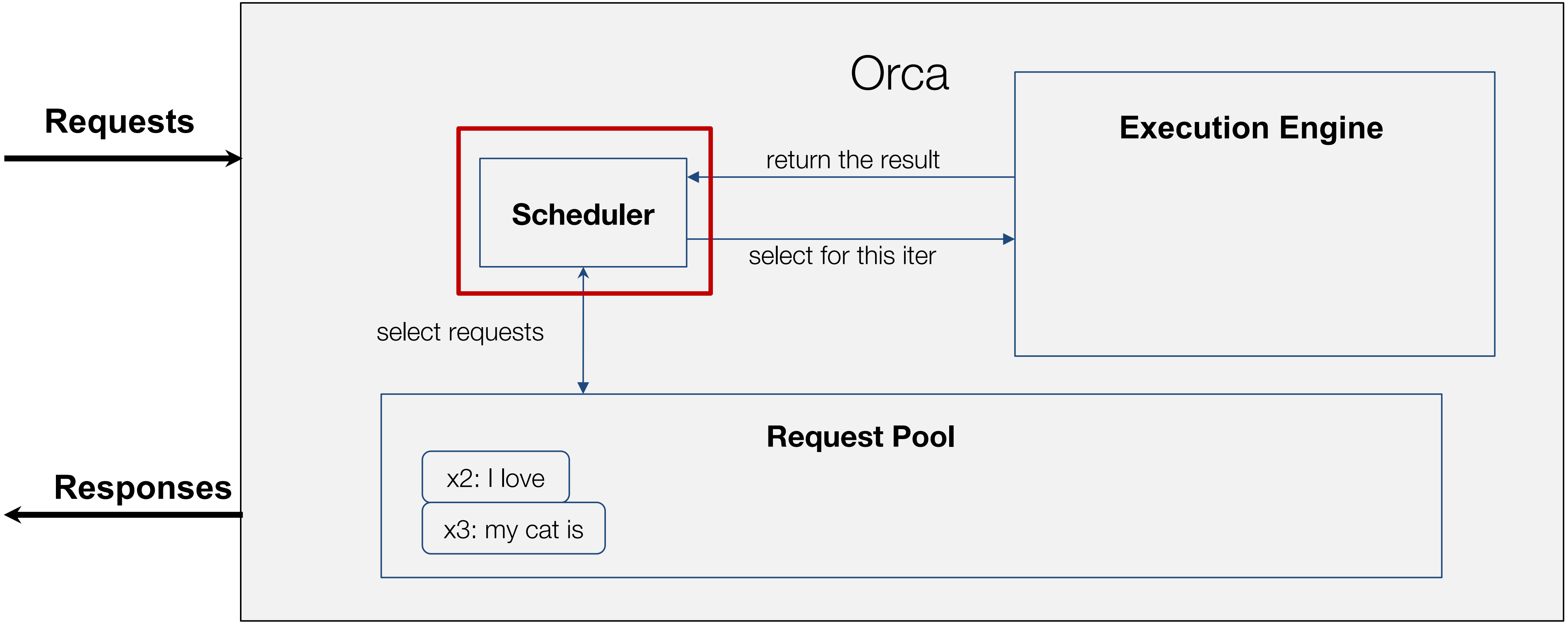
Control Flow

Control messages are sent from worker to worker without waiting for synchronization from GPUs

Only the **last worker** must wait for the GPUs to finish and then it can collect the output tokens and return to master

FasterTransformer, Megatron-LM exchange control messages via NCCL which requires CPU-GPU synch **at every step** imposing a non-negligible communication overhead

2. Scheduling Algorithm



Next we look at the scheduler design

2. Scheduling Algorithm

At each iteration which requests should ORCA process?

First Come First Serve (FCFS)

The algorithm maintains the following invariant:

Given a pair of requests (**a**, **b**), If **a** arrived before **b**, **a** should have run same or more iterations than **b**

This is achieved by **sorting requests by arrival time**

Note: It is still possible that **b** returns to client before **a** only if **b** requires fewer number of iterations to complete

2. Scheduling Algorithm

GPU Memory Constraint

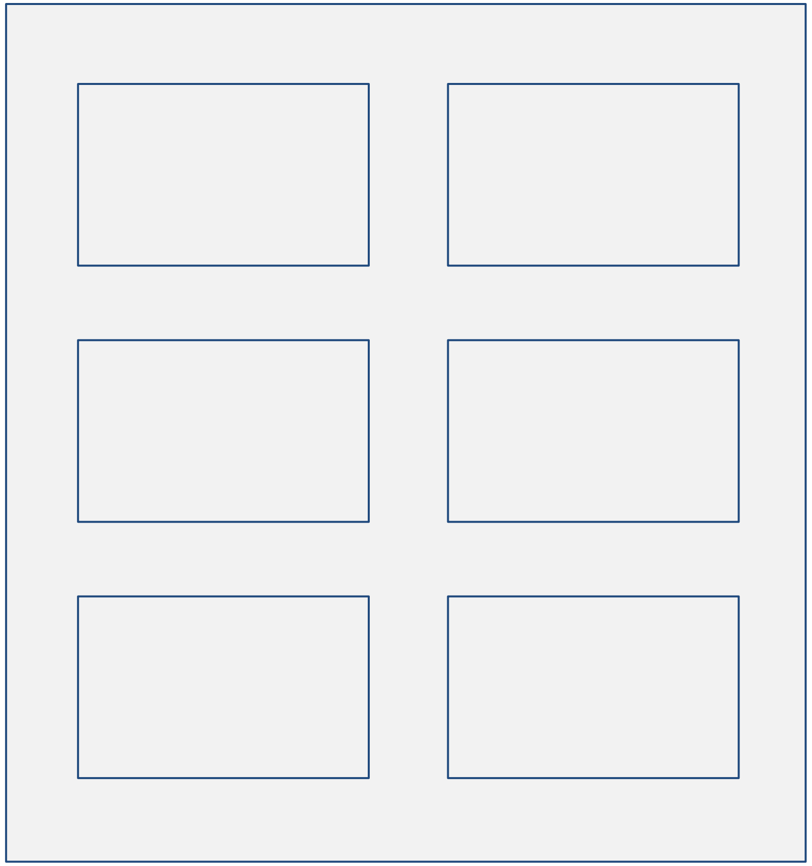
Each token needs a buffer to store attention key-values

Buffers for a request cannot be freed till the request has completed **all** iterations

Naively allocating buffers can lead to **deadlock**

2. Scheduling Algorithm

Attention K/V Manager

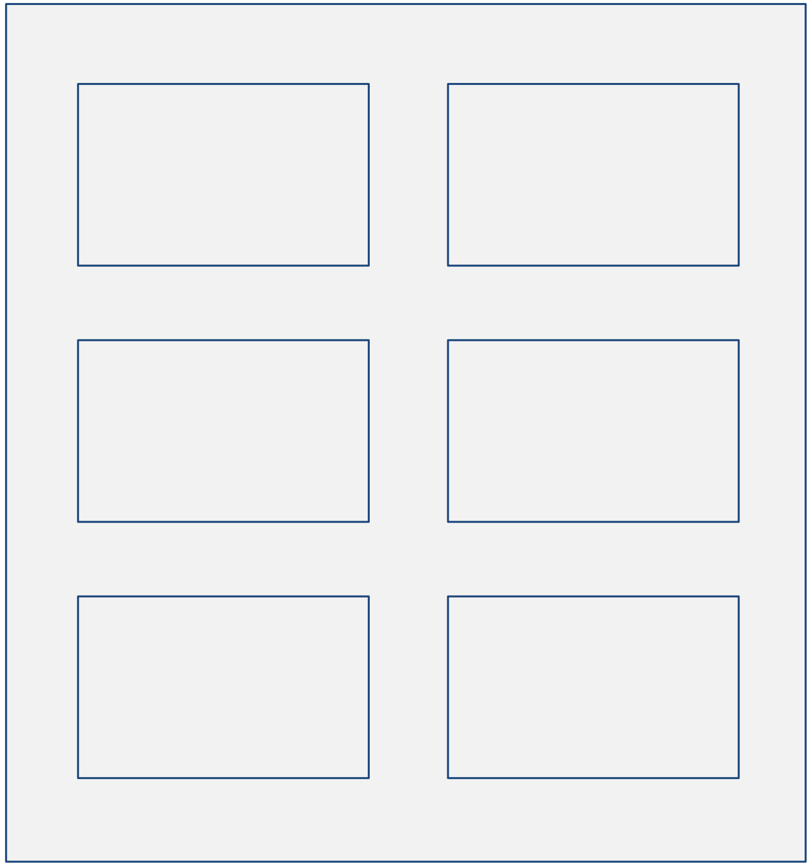


2. Scheduling Algorithm

Iteration 1 …

x1: my cat is

Attention K/V Manager



2. Scheduling Algorithm

Iteration 1 …

x1: my cat is

Attention K/V Manager

2. Scheduling Algorithm

Iteration 1 …

x1: my cat is

Iteration 2 …

x1: I

Attention K/V Manager

2. Scheduling Algorithm

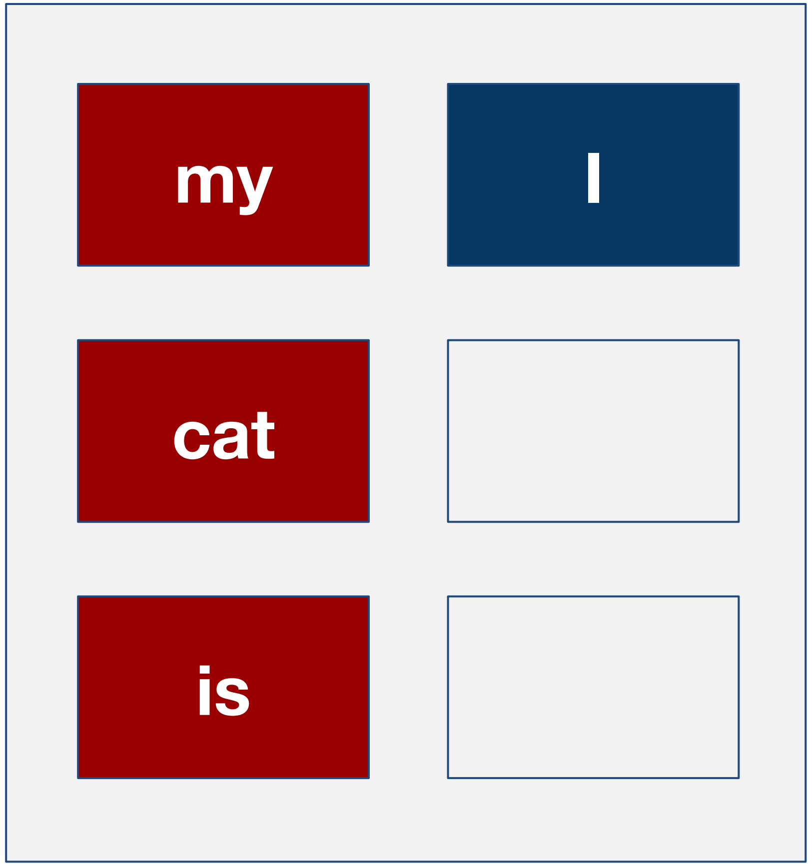
Iteration 1 …

x1: my cat is

Iteration 2 …

x1: I

Attention K/V Manager



2. Scheduling Algorithm

Iteration 1

x1: my cat is so

Iteration 2 …

x1: I

Attention K/V Manager

2. Scheduling Algorithm

Iteration 1

x1: my cat is so

Iteration 2

x1: I like

Attention K/V Manager

my	**I**
cat	**so**
is	**like**

2. Scheduling Algorithm

Iteration 1

x1: my cat is so

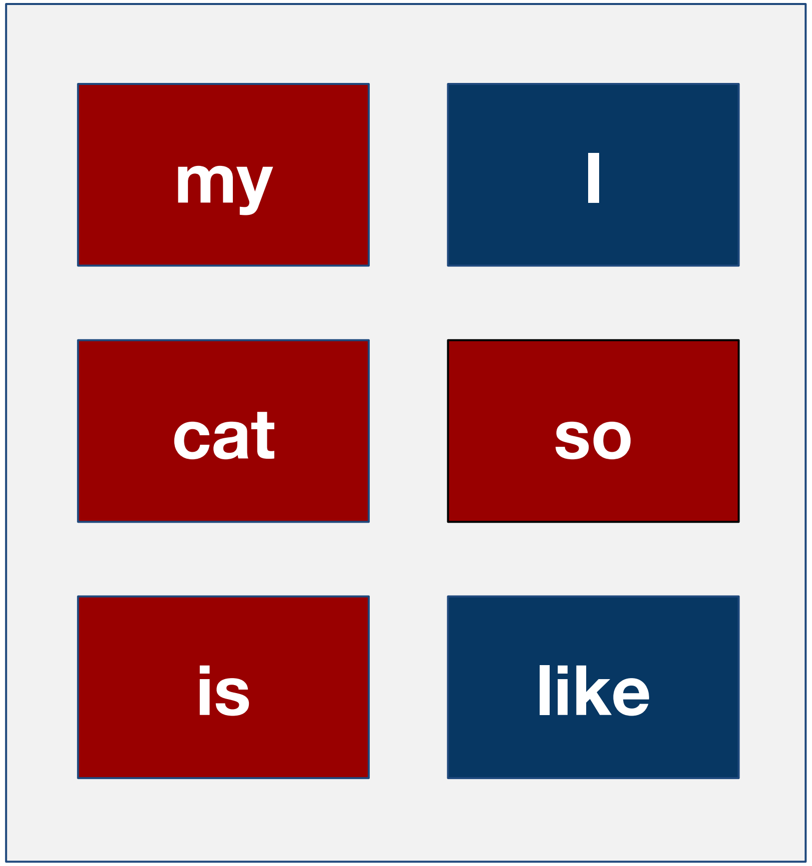
Iteration 2

x1: I like

Iteration 3

Deadlock!!

Attention K/V Manager



2. Scheduling Algorithm

Solution: **Memory Aware Allocation** - reserve enough GPU space for all iterations at the 1st iteration itself

Predetermine ‘max_tokens’ any request can generate

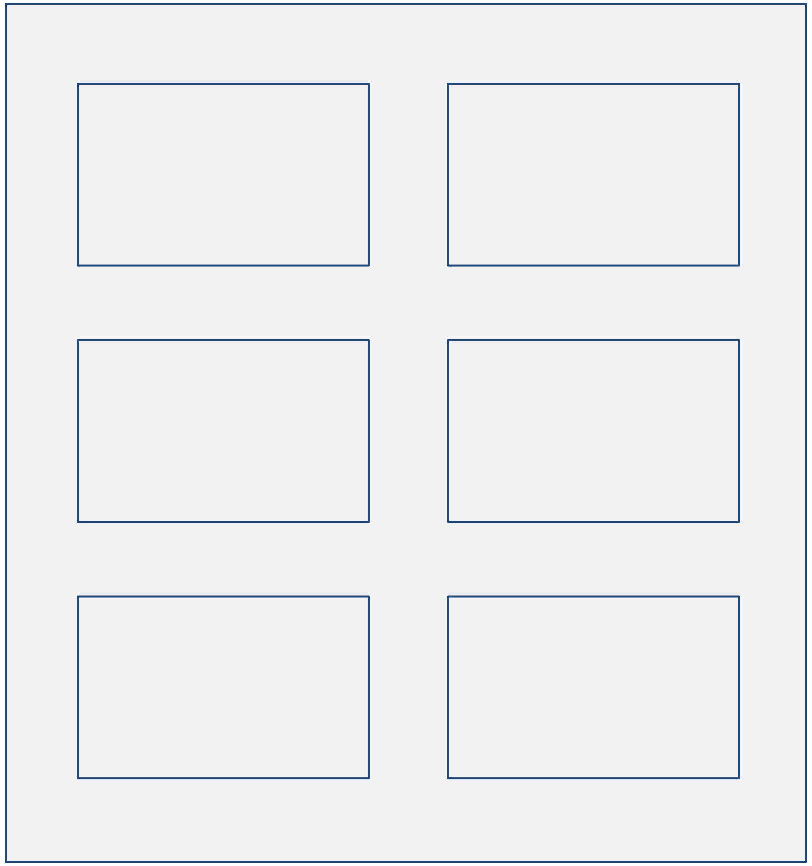
2. Scheduling Algorithm

Iteration 1 …

x1: my cat is

eg: max_tokens = 5

Attention K/V Manager



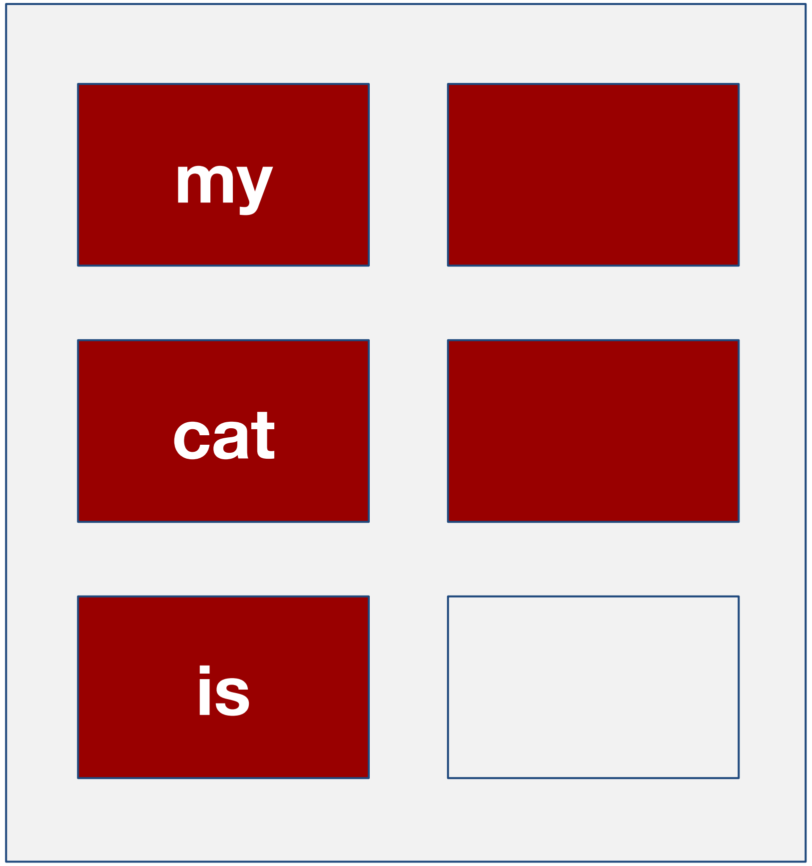
2. Scheduling Algorithm

Iteration 1 …

x1: my cat is

eg: max_tokens = 5

Attention K/V Manager



Scheduler Pseudocode Walkthrough

[n_scheduled = number of busy workers]

Params: *n_workers*: number of workers, *max_bs*: max batch size, *n_slots*: number of K/V slots

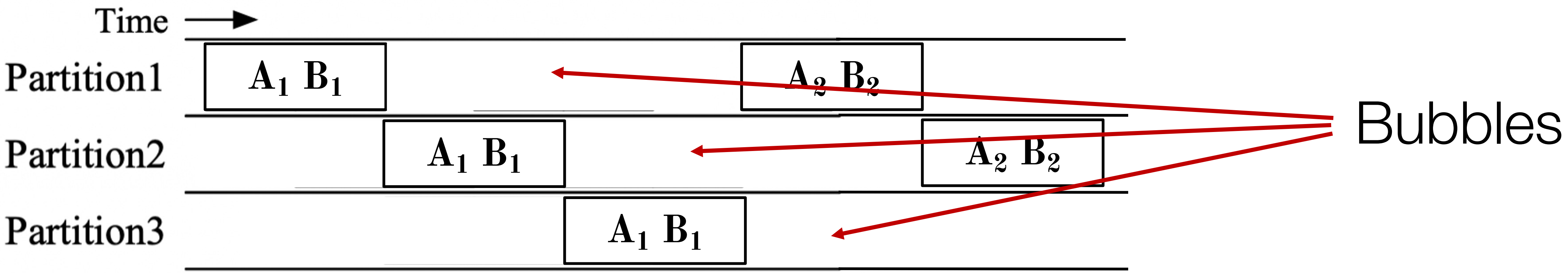
```
n_scheduled ← 0
n_rsrv ← 0
while true do
    batch, n_rsrv ← Select(request_pool, n_rsrv)
    schedule engine to run one iteration of
    the model for the batch
    foreach req in batch do
        req.state ← RUNNING
    n_scheduled ← n_scheduled + 1
    if n_scheduled = n_workers then
        wait for return of a scheduled batch
        foreach req in the returned batch do
            req.state ← INCREMENT
            if finished(req) then
                n_rsrv ← n_rsrv − req.max_tokens      [GPU Memory is Freed]
        n_scheduled ← n_scheduled − 1
```

[n_rsrv = number of slots already reserved]

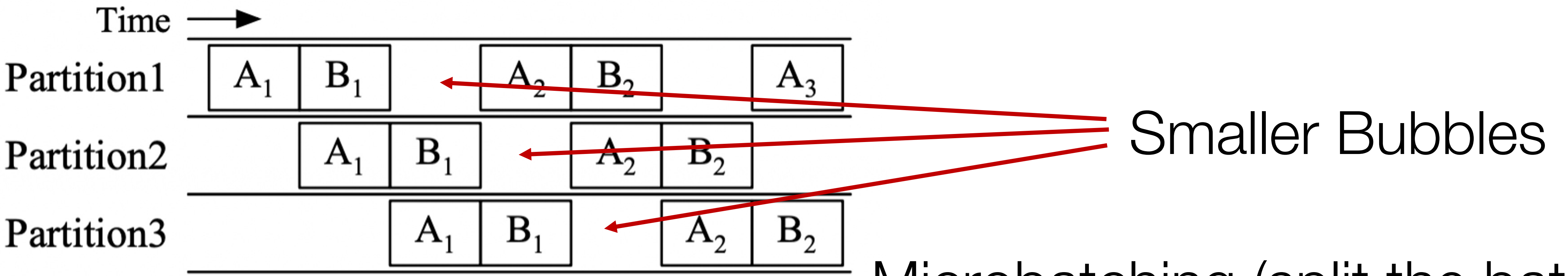
```
def Select(pool, n_rsrv):
    batch ← {}
    pool ← {req ∈ pool | req.state ≠ RUNNING}
    SortByArrivalTime(pool)   [FCFS]
    foreach req in pool do
        if batch.size() = max_bs then break
        if req.state = INITIATION then
            new_n_rsrv ← n_rsrv + req.max_tokens      [GPU Memory Constraint]
            if new_n_rsrv > n_slots then break
            n_rsrv ← new_n_rsrv
        batch ← batch ∪ {req}
    return batch, n_rsrv
```

There are concurrent threads running which insert newly arrived requests into the pool and remove finished requests

3. Pipeline Parallelism



(a) Vanilla execution pipeline



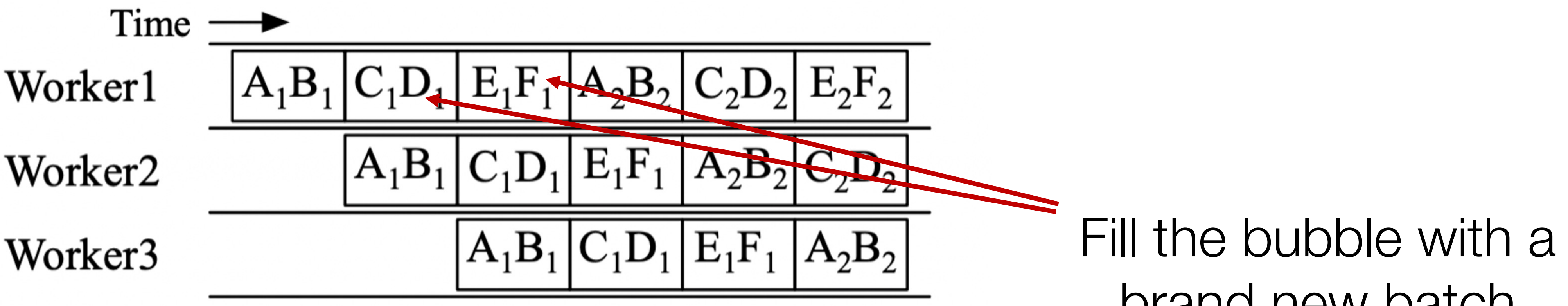
(b) FasterTransformer execution pipeline.

Microbatching (split the batch)

3. Pipeline Parallelism

Key reason for bubbles: **Request Level Scheduling** - must wait till batch completes **all iterations** before starting a new batch

ORCA has **Iteration Level Scheduling**!



(c) ORCA execution pipeline

3. Pipeline Parallelism

ORCA will schedule a batch whenever there is a free worker:

If num scheduled batches < num workers
Assign batch to free worker
Else wait

No worker is ever idle if there are requests

Pipelining achieved without microbatching!

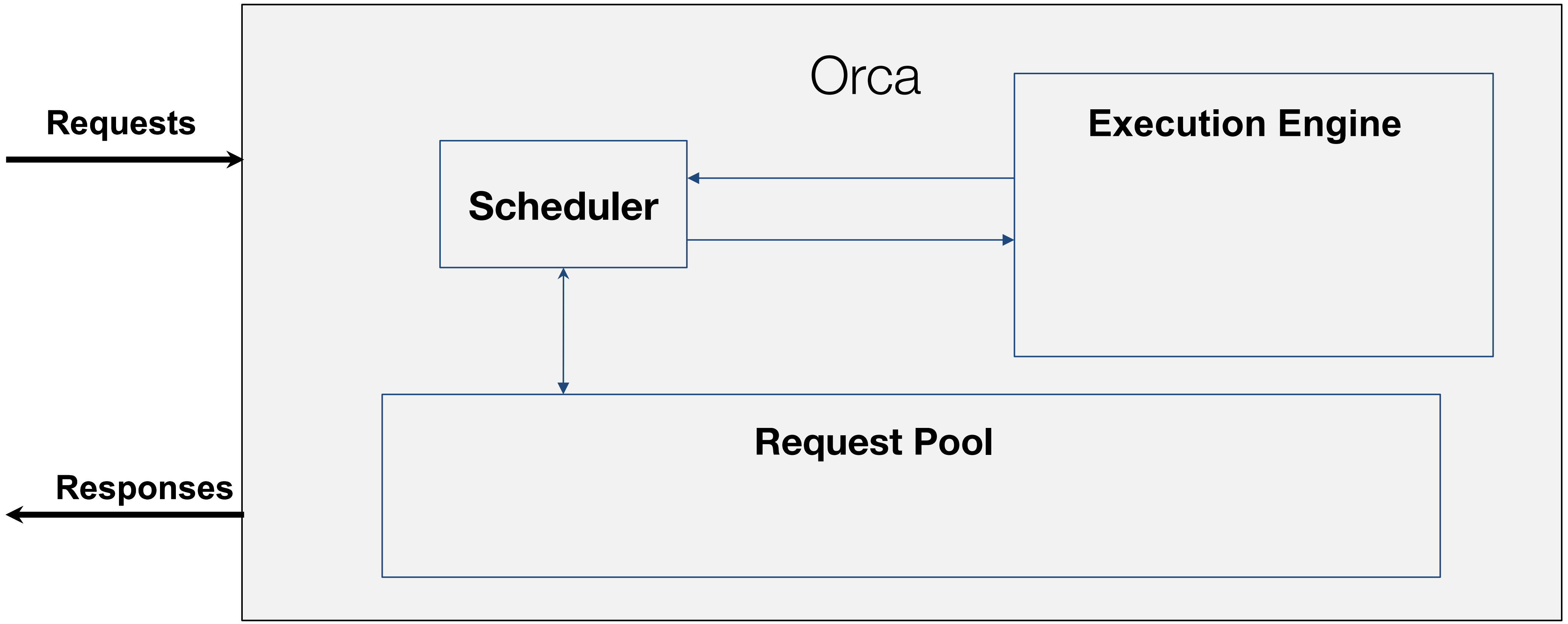
Outline

- Introduction & Related Work
- Challenges & Solutions
- Orca Design
- Evaluation
- Summary & Future work

Evaluation setup

- Model
  - GPT-3 models up to 341B parameters
- Hardware
  - Azure VMs with 8 A100 40G GPUs
- Baseline
  - Execution engine: NVIDIA FasterTransformer
  - Inference server: custom scheduler that mimics the batching scheduler of the NVIDIA Triton

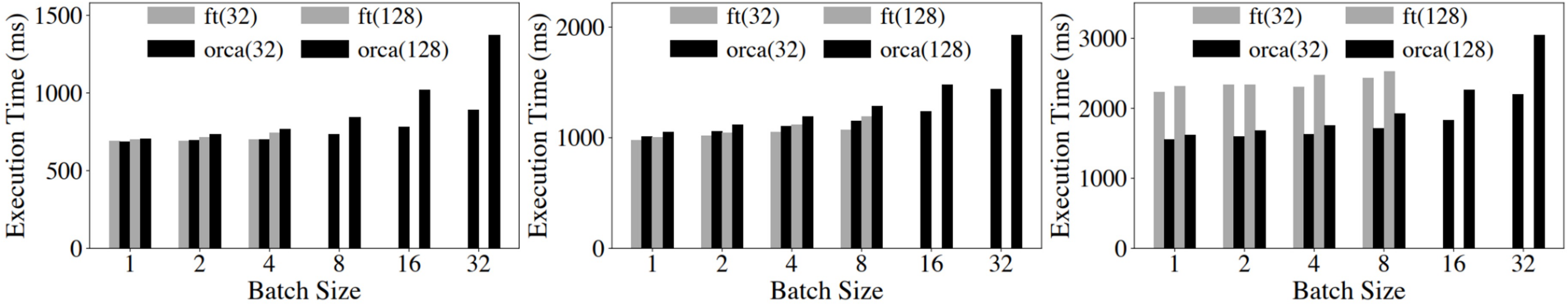
ORCA System Design(recap)



Scenario 1(Engine efficiency)

- Disabled iteration-level scheduler
- Aims to test the overall performance of ORCA engine given the experimental scenario that batch of same-length input keeps arriving

Results (Scenario 1)



(a) 13B model, 1 GPU. (b) 101B model, 8 GPUs. (c) 175B model, 16 GPUs.

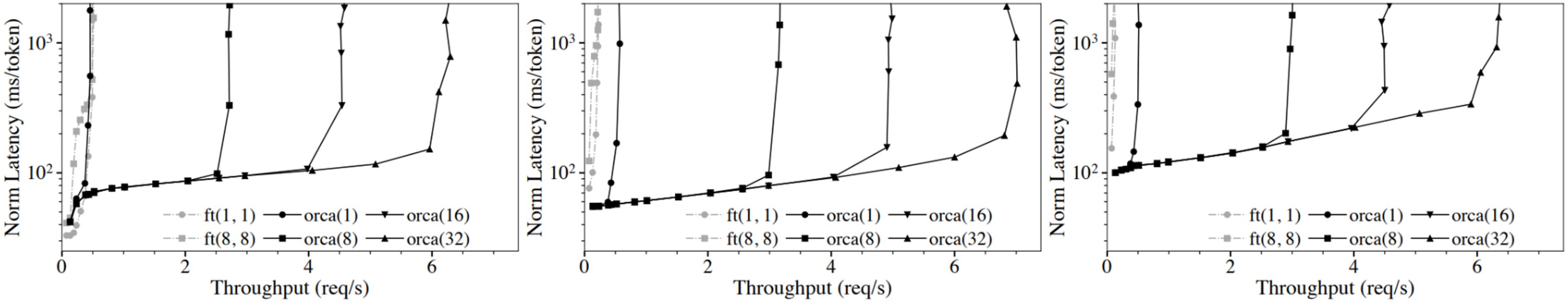
- No attention operation batching
- Control-data plane separation for better performance on 175B and 341B models

Scenario 2 (End-to-end)

- Workload
  - Synthesized the trace of client requests
  - Request arrival time: Poisson process with varying request rate
  - Input length: Uniform(32, 512)
  - Output length: Uniform(1, 128)

- Measure: Latency-throughput
  - Normalized latency by output length since processing time is approximately proportional to output length

Results(Scenario 2)



(a) 101B model, 8 GPU.

(b) 175B model, 16 GPUs.

(c) 341B model, 32 GPUs.

- No significant speedup for small number of requests
- 36.9× speedup for large number of requests

Outline

- Introduction & Related Work
- Challenges & Solutions
- Orca Design
- Evaluation
- Summary & Future work

Summary

- **Challenges that ORCA Address**
  - Request Scheduling
  - Batching
- **Solutions**
  - Iteration-level scheduling
  - Selective batching
- **Design**
  - Distributed Architecture
  - Iteration-level scheduler
  - Pipeline Parallelism

Pros & Cons

- **Pros**
  - First serving system for Transformer-based models that employs iteration-level scheduling and selective batching
  - Improves throughput by 36.9x for GPT-3 175B model
- **Cons**
  - Lack of broader evaluations
  - No open-source codebase for replication
  - Such sophisticated scheduling system makes memory management challenging

Future directions

- Optimizing memory management for LLM serving
  - Optimizing KV-cache
  - PagedAttention!