KubeCon CloudNativeCon North America 2026

#KubeCon #CloudNativeCon

INTERACTIVE INFERENCE 101

Minting Your Own Tokens

The server side of the token economy

Suraj Deshmukh
CoreWeave

Nilekh Chaudhari
Microsoft



KubeCon | CloudNativeCon
North America 2026



Nilekh Chaudhari

Senior Software Engineer

Microsoft

LinkedIn



Suraj Deshmukh

Senior Software Engineer

CoreWeave

linktr.ee/surajssd

LIVE

Watch the first token

A model server on one GPU in this cluster streams a reply.

Time to first token

How long until text starts

One token at a time

The reply grows token by token

Output rate

Tokens per second, including the wait

WHILE WE TALK

Sign in to your workspace

1. Open this link, or scan the QR code:

   `https://workspace.example.com`
2. Sign in with GitHub.
3. Open a terminal. It starts in the `tutorial/` folder.



Expected result

VS Code runs in your browser with the `tutorial` folder open.

The lessons are `01-vllm` to `04-prefill-disaggregation`.

WHAT WE WILL EXPLORE

What changes when you serve a model?

1

LLM servers cache aggressively

The KV cache stores every token a request has processed.

2

How much work can one server do?

Concurrency is how many requests are open at the same time. Run 1, then 4, and count how many finish in one minute.

3

GPUs are not fungible

Which GPUs a model gets, and how they connect, changes what runs.

We return to this list at the end.

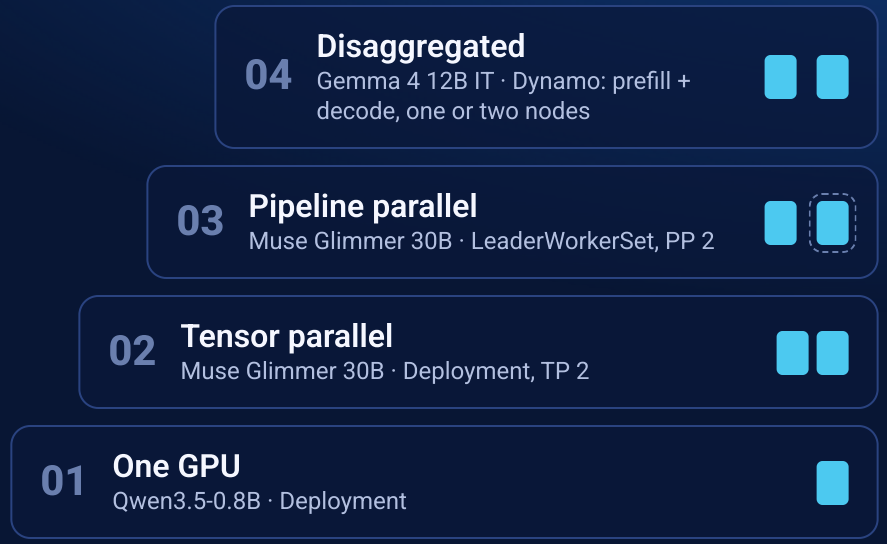
THE PLAN FOR TODAY

Climb the deployment ladder

Each lesson changes how the model is placed on GPUs. Lesson 04 also changes the model and the serving system.

- **01:** one model, one GPU
- **02:** one model split across two GPUs in one node
- **03:** one model split across two nodes
- **04:** reading the prompt and writing the reply on separate GPUs

Your GPU quota is **2**. Delete each lesson before you start the next one.



BEFORE YOU TYPE ANYTHING

How your workspace works

Where things run

- You get your own **workload namespace**. Your model runs there.
- `localhost` is your workspace, not your laptop.
- Models are read-only at `/models`. Nothing downloads.

How you work

- Keep **one terminal** for all lessons. The commands assume it.
- Copy every command from the lesson README.
- When a step fails, raise your hand.

HANDS-ON

Your turn

Open `tutorial/01-vllm/README.md`

Steps 1 and 2

1 minute

Raise your hand if a step fails. We will come to you.

Deploy the model now, then follow its logs

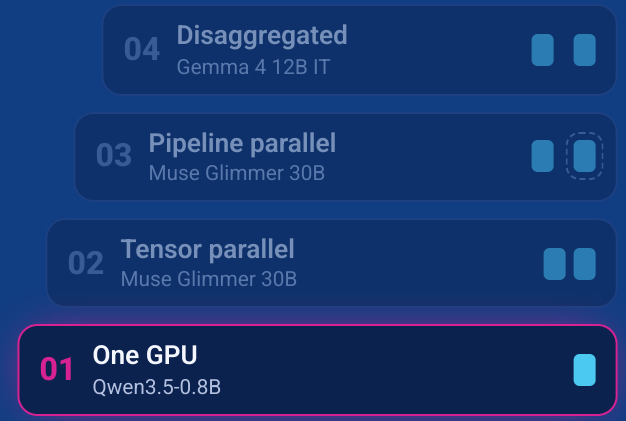
Leave the logs running. The model needs about two minutes to load. We explain what you see while it loads.

KubeCon | CloudNativeCon
North America 2026

LESSON 01

One model on one GPU

While Qwen3.5-0.8B loads: what a model server does with a request, and where the GPU memory goes.



INFERENCE

Tokens in, tokens out



Prefill makes the first token, then decode makes one token per step

Illustrative split. Each model has its own tokenizer.

- The server first reads the **whole prompt** in one pass. That pass also makes the first reply token. This is **prefill**.
- Then it writes the rest of the reply **one token per step**. This is **decode**.
- For a dense model, every decode step reads every weight from GPU memory. Serving many requests in one step shares that cost.

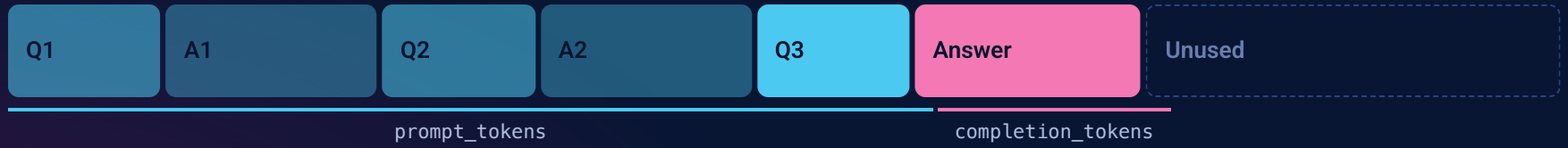
REQUEST ANATOMY

What one chat request contains

```
{
  "model": "Qwen/Qwen3.5-0.8B",
  "messages": [
    {"role": "user", "content": "Explain Kubernetes in one sentence."},
    {"role": "assistant", "content": "Kubernetes runs containers…"},
    {"role": "user", "content": "How does it scale?"},
    {"role": "assistant", "content": "It adds Pods…"},
    {"role": "user", "content": "And on GPUs?"}
  ],
  "max_tokens": 128,
  "temperature": 0
}
```

- The server turns `messages` into **prompt tokens**.
- A chat app resends the **whole history** every turn, so prompts grow.
- `max_tokens` caps the **output**. In lessons 02 and 03, reasoning counts as output too.
- Lesson 01 allows **2,048 tokens** for prompt and output together.

Turn 3: one request in a 2,048-token context window ILLUSTRATIVE SIZES



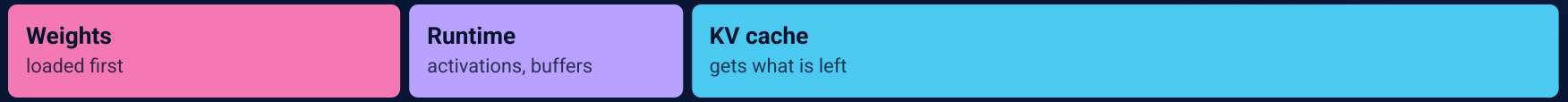
GPU MEMORY · LESSON 01

Where one GPU's memory goes

ONE NVIDIA L40: 48 GB



INSIDE THE VLLM BUDGET



- vLLM loads the weights, measures what a forward pass needs, and gives the rest of the budget to the **KV cache**.
- The KV cache size decides how many tokens the server can hold at once.

The split inside the budget is illustrative. Read the real numbers in your startup logs.

STARTUP LOGS

What to look for while the model loads

LOOK FOR	WHAT IT TELLS YOU
Model loading and the memory it took	The weights now sit in GPU memory
GPU KV cache size, in tokens	How many tokens all requests can hold together
Maximum concurrency for 2,048 tokens per request	How many full-length requests fit in that cache
`--enforce-eager` **in the arguments**	`torch.compile` and CUDA graph capture are skipped, so startup is faster
The API server starts on port 8000	The model is ready for requests

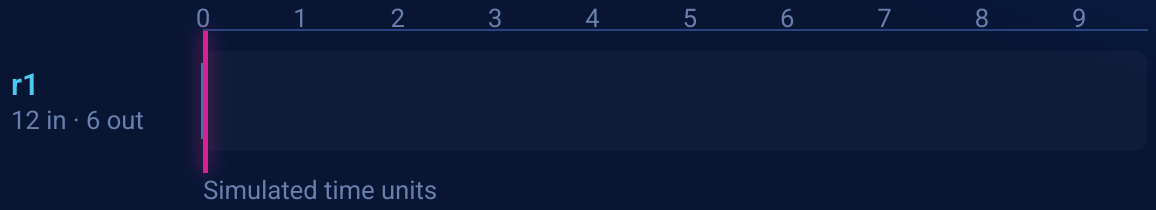
Later request logs show prompt throughput, generation throughput, and running and waiting requests.

SIMULATION B · PREFILL AND DECODE SIMULATED

Prefill, then one token per step

Prefill reads the prompt and emits the first token. Each decode step reads the last token and emits the next. Every step has a **token budget**; prompts can run whole or in chunks.

PREFILL Whole prompt | Chunks of 4 SPEED 0.5× | 1× | 2×



REQ	WAIT	TTFT	TPOT	E2E
r1	—	—	—	—

Prompt `<|im_start|>` `user` `Explain` `Kubernetes` `in` `one` `sentence` `.` `<|im_end|>` `<|im_start|>` `assistant` `\n`

Output

Press Step to run the first step.

t = 0.0 prefill decode output token

Step Play Reset

HANDS-ON

Your turn

Open `tutorial/01-vllm/README.md`

Steps 3 and 4

8 minutes

Raise your hand if a step fails. We will come to you.

Send requests, then load the server

Look for:

- The time to first token in the demo page
- How many requests finish in one minute at concurrency 1 and 4 (1 or 4 requests open at a time)

GRAFANA OVERVIEW · DURING THE TRAFFIC RUNS

What to watch

Running and waiting requests

How full the batch is, and how much demand is queued.

KV-cache use

How much of the token memory is in use.

Token throughput

Prompt and generation tokens per second.

Latency

Time to first token and end-to-end time.

GPU activity

Your one GPU. The data can lag by one minute.

Compare 1 and 4

Did four concurrent requests finish more work in the same minute?

DEBRIEF · LESSON 01

Four at once finished more work

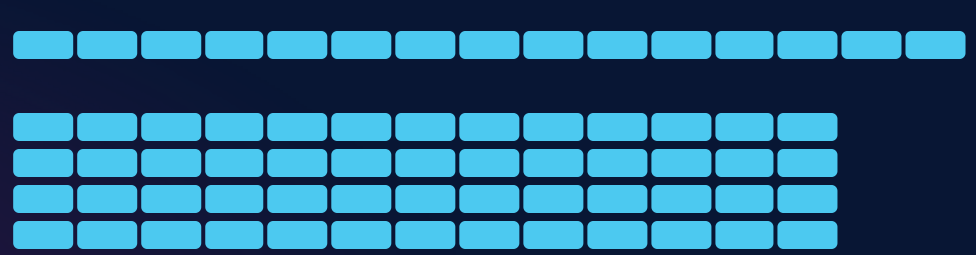
MEASURED ON THIS CLUSTER

15
requests in one minute at concurrency 1

52
requests in one minute at concurrency 4

`traffic.py`, Qwen3.5-0.8B, one fixed prompt with a 256-token output limit. The prompt can reuse cached prefixes. This is an observation, not a benchmark.

- Four requests did not take four times as long.
- Each decode step made one token for all four requests at once.
- Lesson 01 runs up to **8** sequences at once.
- vLLM's memory budget was **25% of GPU memory**. The next model's weights exceed one GPU's memory.



Each box is one finished request. The two runs were separate minutes. Within each run, the rows ran at the same time. `traffic.py` counts requests but does not time them, so the time per request is an estimate: one minute divided by the requests that one row finished.

NEXT PROBLEM · LESSON 02

Muse Glimmer 30B in BF16 does not fit on one L40

1. Stop the demo with `Ctrl+C` if it still runs.
2. Delete lesson 01:

```
kubectl delete -f vllm.yaml
```

3. Check your GPU quota. Expect `0/2` in use:

```
kubectl get resourcequota gpu-quota
```

4. Enter lesson 02 and apply it. The Pod needs a few minutes to load:

```
cd ../02-tensor-parallel
kubectl apply -f vllm.yaml
```



KubeCon | CloudNativeCon
North America 2026

LESSON 02

Batching, memory, and tensor parallelism

While Muse Glimmer loads: why concurrency 4 did more work, what limits it, and how one model spans two GPUs.



SIMULATION A · DECODE ONLY SIMULATED

Static vs continuous batching

Same requests, same GPU with **2 sequence slots**. Each step emits one token for every running request.

PRESET [Uneven lengths] [One long request] [Equal lengths]

SLOTS [1] [2] [3] [4]

SPEED [0.5×] [1×] [2×]

r1 2 r2 5 r3 2

NEW REQUEST TOKENS [1] [2] [4] [8] [Add request]

STEP 1 2 3 4 5 6 7

Static batching

3 requests waiting.

0/9 tokens 0 idle slot-steps

Continuous batching

3 requests waiting.

0/9 tokens 0 idle slot-steps

Step 0 of 7

[Step] [Play] [Reset]

OPERATIONAL GUIDANCE

Queues and latency reveal unmet demand

- **GPU utilization** reports the share of time that any kernel ran. One small request can keep it high.
- **Waiting requests** show demand that the server cannot serve yet.
- **Time to first token** shows how long users wait for output.
- Check both alongside running requests and KV-cache use.

Capacity signals

Queue growth, latency, and KV-cache use

GPU utilization alone is insufficient

Busy time does not show how much more work fits.

CLAIM 1 · MEMORY

What the KV cache stores

- Every layer computes a **key** and a **value** vector for each token it processes.
- The server keeps them, so a decode step reads the cache instead of recomputing the whole prompt.
- The cache grows by one token for every decode step.

KV bytes per cached token

```
2 × layers × KV heads × head dimension × bytes
2 × 32 × 8 × 128 × 2 = 128 KiB
```

One 2,048-token sequence: **256 MiB**

Illustrative full-attention layout with BF16 values. Use KV heads, not query heads.

> Memory decides how many **tokens** fit, not how many requests.

1 GiB of KV cache: 4 requests of 2,048 tokens



1 GiB of KV cache: 2 requests of 4,096 tokens



Illustrative: 128 KiB per token, from the layout above.

SIMULATION C · KV CACHE SIMULATED

KV cache blocks and reservations

A pool of **48 token slots** and **4 sequence slots**. Before a request starts, it reserves blocks for its whole prompt and output.

BLOCK SIZE 1 token | 4 tokens

SPEED 0.5× | 1× | 2×

0/48 blocks reserved **0** allocated **48** free to reserve

0 token slots wasted inside blocks



REQ	SIZE	CACHED	BLOCKS	STATE
r1	10+6	0	0/15	queued
r2	12+8	0	0/19	queued
r3	6+6	0	0/11	queued
r4	8+4	0	0/11	queued

Each request reserves ceil((prompt + output − 1) / 1) blocks before it starts.

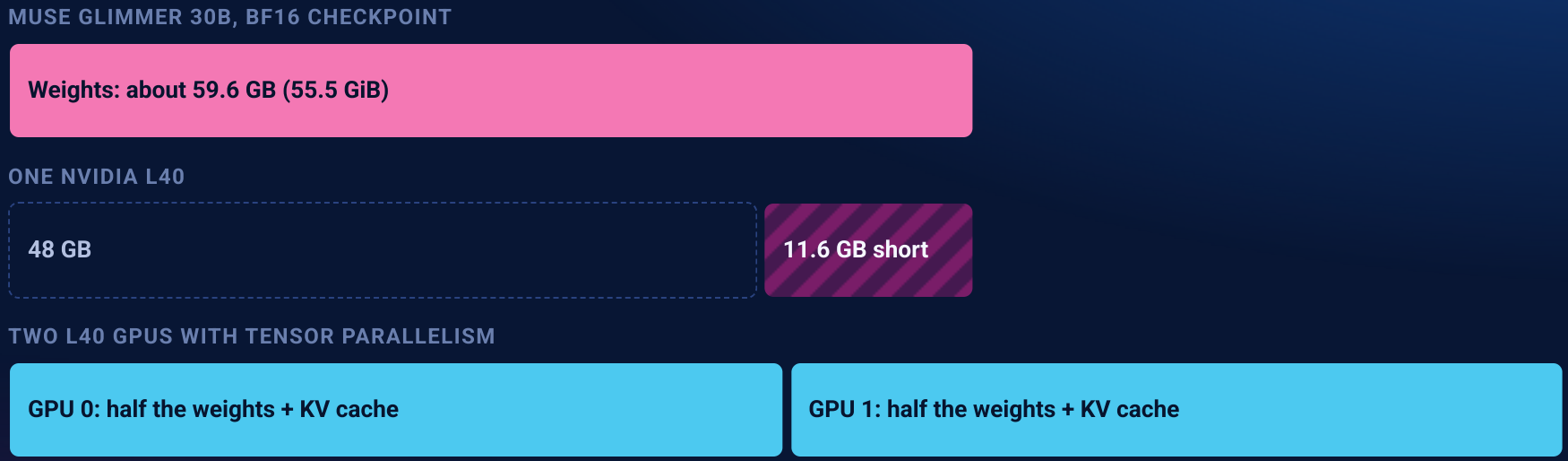
Conservative teaching policy: reserve the full output before admission. vLLM can admit more work and preempt requests when KV space runs out.

Step 0 cached token wasted slot reserved, not allocated

Step | Play | Reset

HARDWARE · LESSON 02

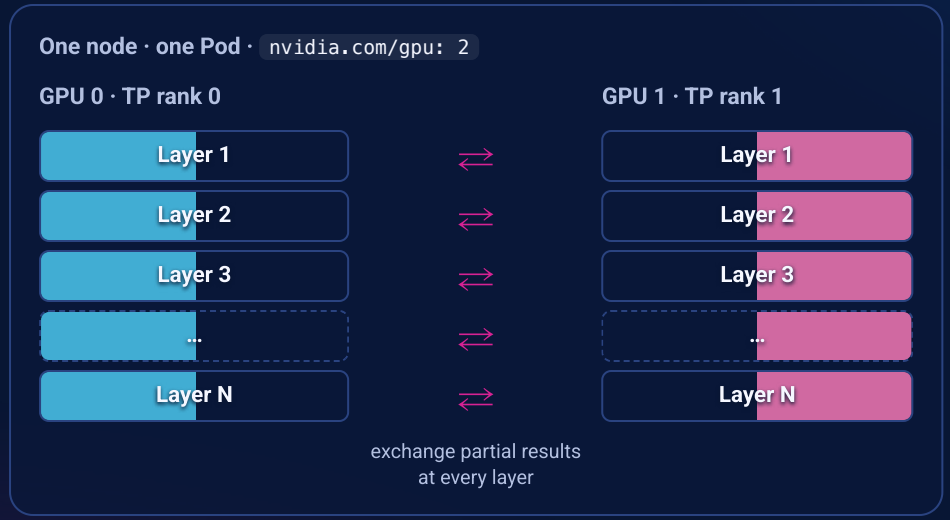
Why Muse Glimmer needs two GPUs



- A dense model needs all of its weights for every token. They must fit in GPU memory with room left for the KV cache.
- The vLLM logs show **26.11 GiB** of model memory on each GPU with TP 2.

TENSOR PARALLELISM · LESSON 02

Split every layer across two GPUs



Each GPU holds half of every weight matrix. Both GPUs work on every step of every request.

- vLLM: `--tensor-parallel-size 2`
- Kubernetes: **one Pod** that requests `nvidia.com/gpu: 2`
- The GPUs exchange data at every layer, so a fast link matters.
- This lesson's **single Pod** places both GPUs on **one node**.

> Two free GPUs on different nodes cannot run this Pod.

HANDS-ON

Your turn

Open `tutorial/02-tensor-parallel/README.md`

Steps 3 to 5

7 minutes

Raise your hand if a step fails. We will come to you.

Inspect both GPUs, then stream a reply

Look for:

- TP ranks 0 and 1 and the KV cache size in the logs
- Model memory in use on both GPUs
- The model reasons before it answers

DEBRIEF · LESSON 02

Lesson 02 placed both GPUs in one Pod

- Both GPUs held half of every layer.
- They exchanged data at **every layer**.
- The Pod requested both GPUs, so the scheduler needed **two free GPUs on the same node**.

In a busy cluster

Free GPUs are often spread across nodes: one here, one there. A TP 2 Pod waits even though two GPUs are free.

> Can one model use a GPU on each of two nodes?

NEXT PROBLEM · LESSON 03

Two free GPUs can be on different nodes

1. Stop the demo with `Ctrl+C`, then delete lesson 02:

```
kubectl delete -f vllm.yaml
```

2. Enter lesson 03. Expect the `leaderworkersets` resource:

```
cd ../03-pipeline-parallel
kubectl api-resources --api-group=leaderworkerset.x-k8s.io
```

3. Check your GPU quota. Expect `0/2` in use:

```
kubectl get resourcequota gpu-quota
```



KubeCon | CloudNativeCon
North America 2026

LESSON 03

Pipeline parallelism and hardware

While the LeaderWorkerSet group loads: what LWS creates, how to split by layers, and why the link between GPUs decides what you can run.

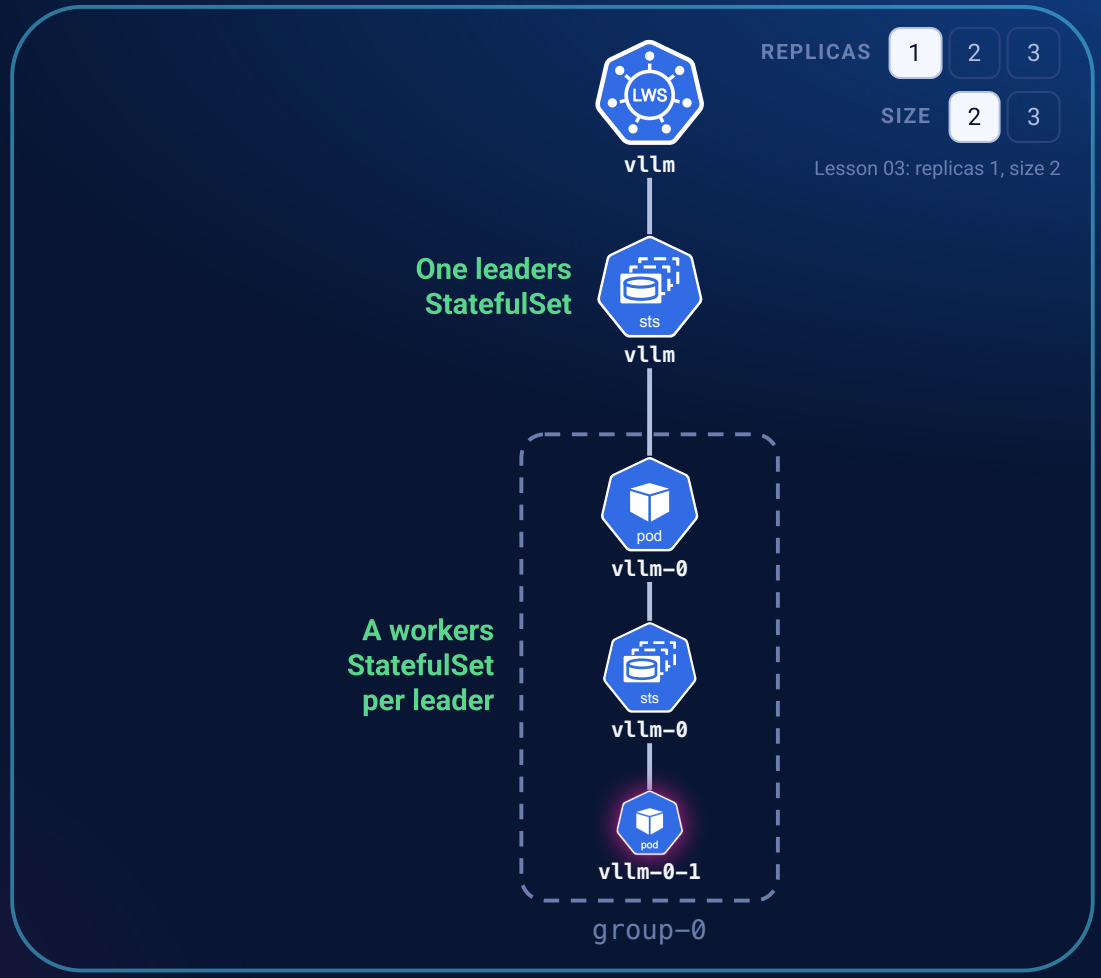


LEADERWORKERSET · LESSON 03

LeaderWorkerSet runs groups of Pods

WHAT LWS CREATES · STEP 5 OF 5

1. You apply LeaderWorkerSet `vllm` with `replicas: 1` and `size: 2`.
2. LWS creates one leaders StatefulSet, `vllm`, with one replica for each group.
3. The leaders StatefulSet creates one leader Pod for each group: `vllm-0`.
4. LWS creates a workers StatefulSet for each leader Pod: `vllm-0`. The leader Pod owns it.
5. Each workers StatefulSet creates size − 1 = 1 worker Pod: `vllm-0-1`.

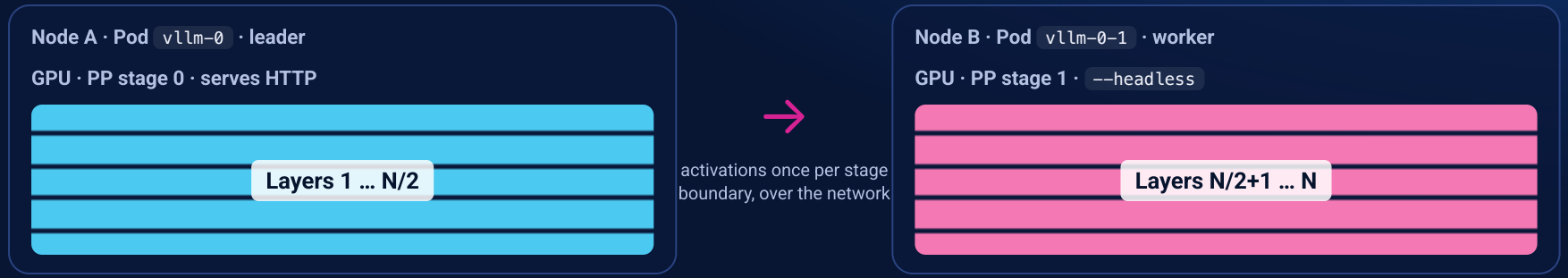


Adapted from the LeaderWorkerSet concept diagram, lws.sigs.k8s.io (CC BY 4.0). Icons: Kubernetes community icons and the LWS logo.

PIPELINE PARALLELISM · LESSON 03

Split the layers into stages

Service `vllm-leader` → leader only



LeaderWorkerSet keeps both Pods as one model replica. Required anti-affinity puts them on different nodes.

- vLLM: `--pipeline-parallel-size 2` across `--nnodes 2`
- A **stage** is a group of consecutive layers on one GPU. Here, stage 0 has the first half and stage 1 has the second half.
- Stage 0 sends activations to stage 1 once per step.
- One request uses one stage at a time, so PP alone does not make one request faster.
- PP needs less communication than TP, so it tolerates a slower link.

TENSOR OR PIPELINE

Same model, different split

	TENSOR PARALLEL · LESSON 02	PIPELINE PARALLEL · LESSON 03
What is split	Every layer, across GPUs	The layers, into stages
Communication	At every layer	Once per stage boundary
Communication cost	Frequent exchanges favor a fast link	Fewer exchanges tolerate a slower link
Kubernetes shape	One Pod with 2 GPUs	LeaderWorkerSet: 2 Pods, 1 GPU each
Placement in this lesson	Two free GPUs on one node	One free GPU on each of two nodes
Model memory per GPU, from the vLLM logs	26.11 GiB	28.56 GiB

LEADERWORKERSET

One model replica, two Pods

- **Leader** `vllm-0`: PP stage 0. It serves the HTTP API.
- **Worker** `vllm-0-1`: PP stage 1. It runs `--headless`.
- Service `vllm-leader` selects only the leader.
- LWS creates a headless Service named `vllm` for Pod discovery.

- LWS sets `LWS_GROUP_SIZE`, `LWS_WORKER_INDEX`, and `LWS_LEADER_ADDRESS`. vLLM uses them for the node count, rank, and rendezvous.
- `RecreateGroupOnPodRestart`: if one Pod restarts, the whole group restarts.
- `startupPolicy: LeaderCreated` starts both Pods early. The leader needs the worker before it is ready.

CLAIM 3 · HARDWARE

GPUs are not fungible: the link matters

How GPUs talk RELATIVE, NOT TO SCALE



NVLink	GPU to GPU inside a node. Rack-scale systems such as GB200 NVL72 extend one NVLink domain to 72 GPUs.
PCIe	GPU to GPU inside a node without NVLink. Lesson 02 TP traffic between the two L40 GPUs goes over PCIe.
RDMA	Between nodes over InfiniBand or RoCE. Data moves between machines without CPU copies.
TCP/IP	Between nodes over ordinary Ethernet, through the kernel network stack.

Workshop GPUs: NVIDIA L40

PCIe cards with 48 GB of GDDR6 memory and no NVLink.

What this means

TP wants the fastest link. PP can work over the network between nodes.

HANDS-ON

Your turn

Open `tutorial/03-pipeline-parallel/README.md`

Steps 3 to 5

6 minutes

Raise your hand if a step fails. We will come to you.

Inspect both stages, then stream a reply

Look for:

- PP rank 0 in the leader's logs
- One GPU in each Pod
- Engine metrics in Grafana for the leader only

DEBRIEF · LESSON 03

Split by layers: both GPUs serve every request

- Each GPU held half of the layers.
- Only the leader served HTTP.
- Every request went through both stages, for both prefill and decode.

Another way to split

Prefill and decode do different work. Prefill reads many prompt tokens at once. Decode reads one token per request per step.

What if each phase had its own GPU?

NEXT PROBLEM · LESSON 04

A request can be split by phase instead of by layer

1. Stop the demo with `Ctrl+C`, then delete lesson 03:

```
kubectl delete -f vllm.yaml
```

2. Enter lesson 04. Expect `dynamographdeployments`:

```
cd ../04-prefill-disaggregation
kubectl api-resources --api-group=nvidia.com
```

3. Check your GPU quota. Expect `0/2` in use:

```
kubectl get resourcequota gpu-quota
```



KubeCon | CloudNativeCon
North America 2026

LESSON 04

Disaggregated prefill and decode

While Dynamo loads: why prefill and decode get in each other's way, and what it costs to separate them.

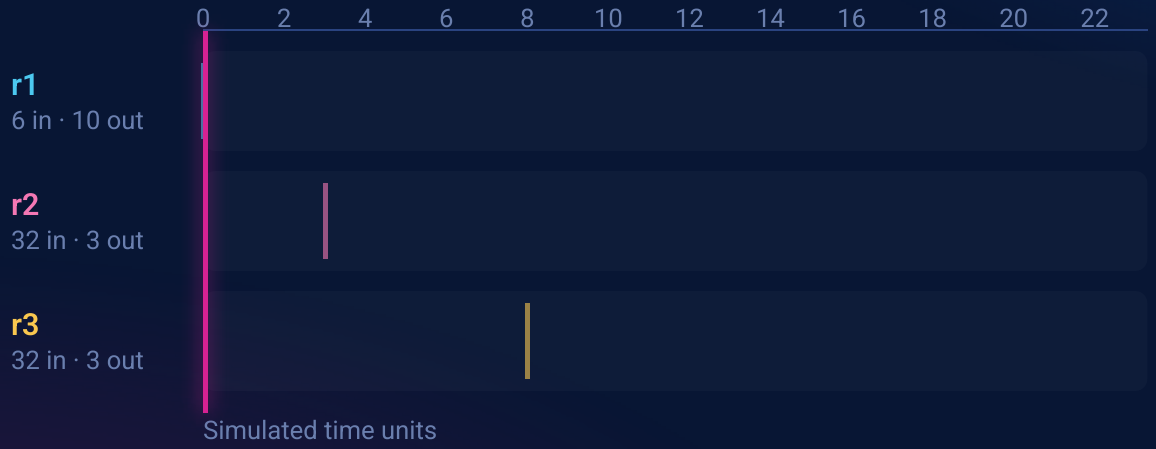


SIMULATION B · PREFILL AND DECODE SIMULATED ONE GPU · NO KV TRANSFER

Long prompts interrupt decoding

One GPU runs prefill and decode for all requests, so a long prompt's prefill shares each step with the running decodes. Every step has a **token budget**; prompts can run whole or in chunks.

PREFILL Whole prompt | Chunks of 8 SPEED 0.5× | 1× | 2×



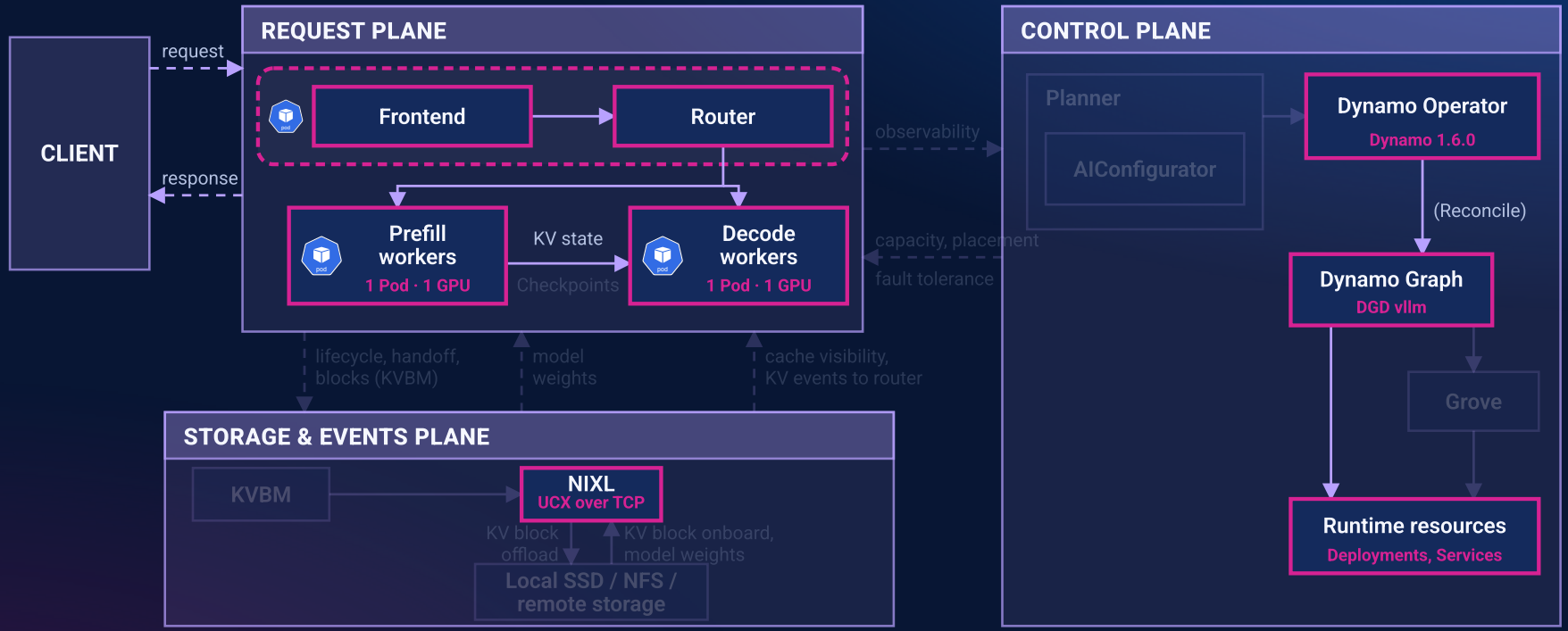
REQ	WAIT	TTFT	TPOT	E2E
r1	—	—	—	—
r2	—	—	—	—
r3	—	—	—	—

Press Step to run the first step.

t = 0.0 prefill decode output token

Step Play Reset

Dynamo splits serving into three planes

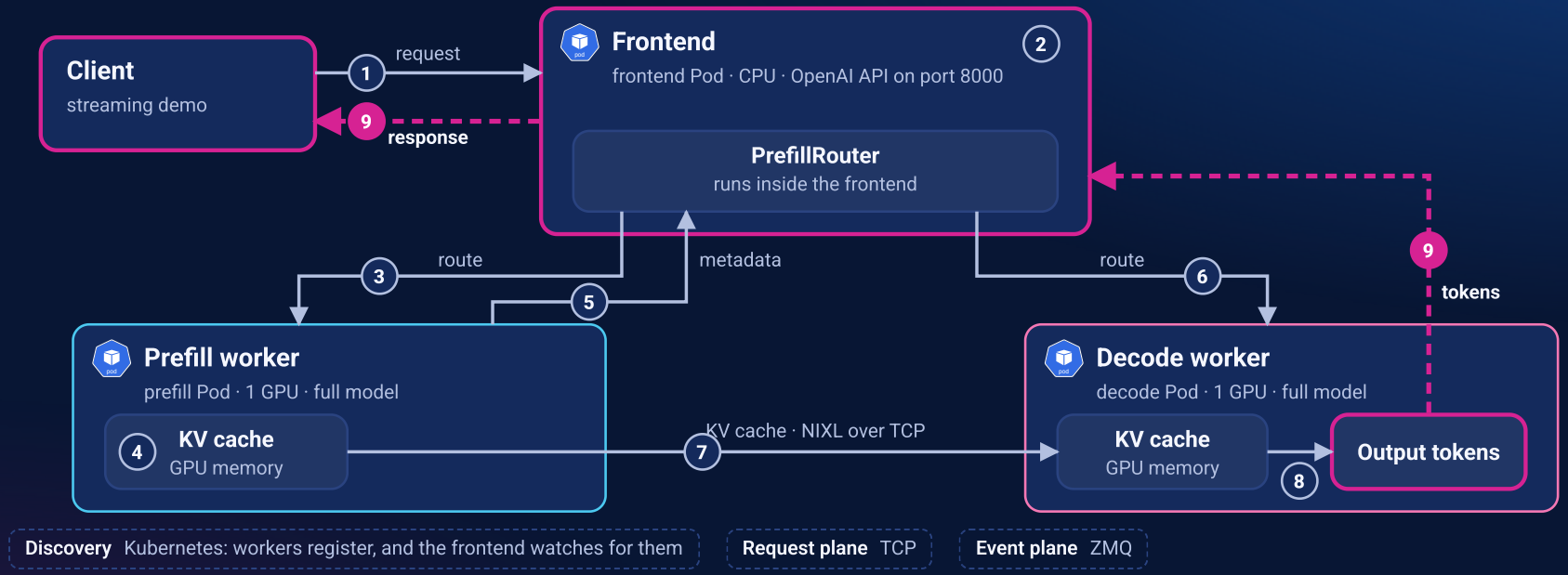


LESSON 04 Lesson 04 runs the highlighted parts: the operator and DynamoGraphDeployment vllm, one frontend Pod whose single container runs the frontend and a round-robin router, one prefill Pod, one decode Pod, and NIXL over TCP. It runs no Planner, Grove, or KVBM.

Adapted from the Dynamo architecture diagram in the Dynamo v1.6.0 documentation, docs.nvidia.com/dynamo (Apache-2.0). Pod icon: Kubernetes icon set.

NVIDIA DYNAMO · LESSON 04

One request through Dynamo

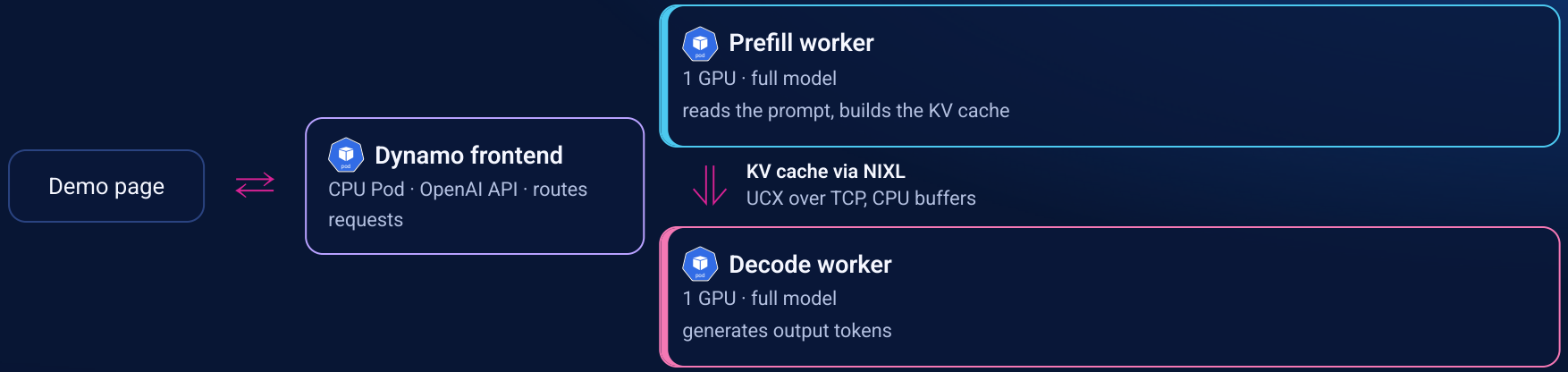


STEP 9 OF 9 **Response:** the tokens stream back through the frontend, which detokenizes them, to the demo.

Adapted from the request flow in the Dynamo v1.6.0 architecture documentation, docs.nvidia.com/dynamo (Apache-2.0). Pod icon: Kubernetes icon set.

DISAGGREGATED SERVING · LESSON 04

Give each phase its own GPU



Each worker pool can scale on its own. Both workers hold a complete copy of Gemma 4 12B IT.

- Gemma 4 12B IT in BF16 fits on one L40, so each worker holds a **full copy**.
- Muse Glimmer would need four GPUs here: two per copy.

- `--disaggregation-mode prefill` and `decode`
- `NixlConnector` roles: `kv_producer` and `kv_consumer`

THE COST

Moving the KV cache is not free

- Lesson 04 sends the KV cache over **TCP** with CPU buffers: `UCX_TLS=tcp,cuda_copy`. No RDMA, no CUDA IPC.
- At low concurrency on TCP, two split workers can be **slower** than one combined worker.
- Disaggregation pays off when prefill hurts decode latency and the link is fast.
- A NIXL startup line does not prove that a transfer worked. Check the streamed reply and both workers' `generate` counters.
- With CPU transfer buffers, hybrid KV allocation produces corrupt text in this runtime. Both workers keep `--disable-hybrid-kv-cache-manager`.

LESSON 04 · STARTUP LOGS

A sequence limit is not a memory guarantee

VLLM STARTUP OUTPUT

2 GiB KV cache per worker

≈ 6,240 token slots

≈ 3.05 sequences of 2,048 tokens

- The same workers allow `--max-num-seqs 8`.
- Only about three full-length sequences fit in the cache.
- Gemma 4 12B mixes sliding-window and full attention, so lesson 02's full-attention formula does not reproduce this number.

HANDS-ON

Your turn

Open `tutorial/04-prefill-disaggregation/README.md`

Steps 3 to 6

7 minutes

Raise your hand if a step fails. We will come to you.

Check both workers, then stream a reply

Look for:

- The prefill role in one worker's logs and the decode role in the other's
- Evidence that both workers handled your request
- The frontend row in Grafana

CLEAN UP LESSON 04

Release your GPUs

1. Stop the demo with `Ctrl+C`.
2. Delete the Dynamo deployment:

```
kubectl delete -f dynamo.yaml --cascade=foreground --timeout=180s
```

3. Check your GPU quota. Expect `0/2` in use:

```
kubectl get resourcequota gpu-quota
```

DEBRIEF · LESSON 04

Prefill and decode ran on separate GPUs

Both counters went up, and the reply streamed. KV transfer over TCP adds overhead.

Measure the benefit against a combined worker with the same model and workload.

So when does each serving shape pay off?

Latency and throughput

Which number to watch depends on who waits for the answer.

DEFINITIONS

Four numbers, four questions

METRIC	DEFINITION	THE USER FEELS IT AS
Time to first token (TTFT)	$t_{\text{first token}} - t_{\text{server receives}}$	The wait before text appears
Time per output token (TPOT)	$\frac{t_{\text{last token}} - t_{\text{first token}}}{N_{\text{output tokens}} - 1}$	Reading speed after the first token
End-to-end latency	$t_{\text{last token}} - t_{\text{server receives}}$	The wait for the whole answer
Output throughput	$\frac{N_{\text{output tokens, all requests}}}{t_{\text{last token, last request}} - t_{\text{server receives, first request}}}$	Tokens per second for the operator, which sets the cost per token

All times are on the server clock, from when the server receives the request until it sends the last token. The user also waits for the network.

- **Chat:** keep TTFT low and TPOT faster than people read.
- **Batch jobs:** latency matters less; throughput sets the cost.

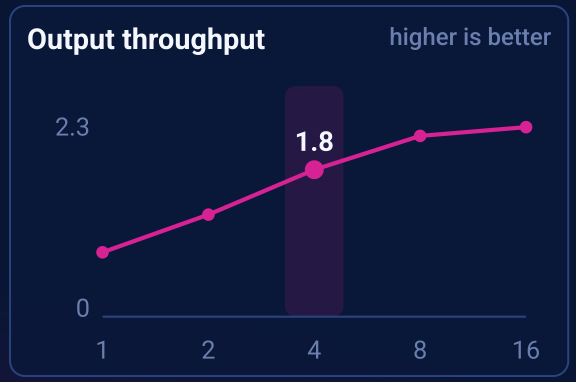
SIMULATION D · LATENCY AND THROUGHPUT SIMULATED

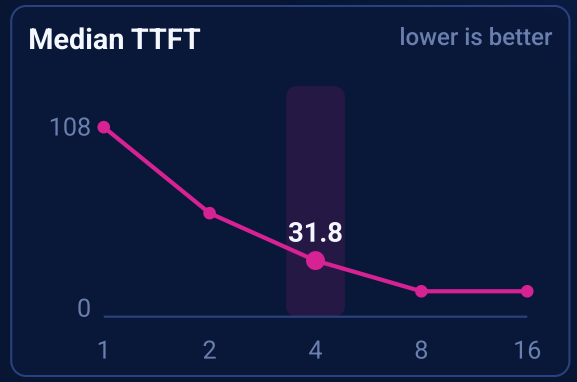
More concurrency is not free

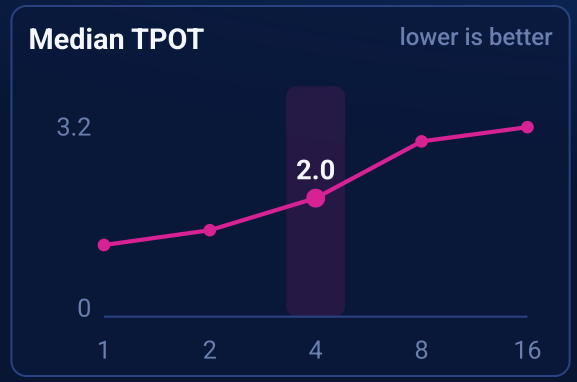
16 requests with mixed lengths arrive at a fixed interval. Every run uses the same arrivals; only the sequence limit changes. Bigger batches make each step slower.

OFFERED LOAD: Light (every 8) | Medium (every 3) | Heavy (every 1)

MAX SEQUENCES: 1 | 2 | 4 | 8 | 16





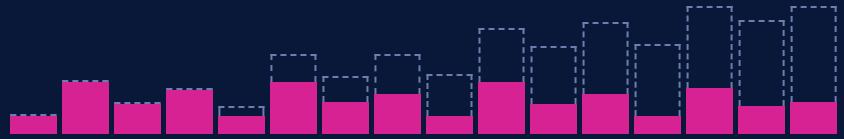


x axis: max sequences · simulated time units, not a GPU benchmark

MAX SEQUENCES 4

16/16 requests · window 0.0–106.8

Output throughput	Median TTFT	Median TPOT	Median E2E	Mean running
1.80	31.8	2.01	56.8	3.6



Per-request latency, r1 to r16: queue wait | prefill and decode

LIVE METRICS

Every source measures a different interval

SOURCE	WHAT IT MEASURES
Demo page	From browser submit to the first reasoning or answer text. Its rate includes that wait.
`traffic.py`	Completion tokens of completed requests, divided by the whole run. No TTFT or TPOT.
vLLM engine panels	TTFT, inter-token latency, and end-to-end time inside the engine. Percentiles are histogram estimates.
Dynamo frontend panels	Client-facing TTFT, latency, and request rate in lesson 04.

> Name the source before you compare two numbers.

RECAP

You climbed the ladder

- **01:** weights, runtime, and KV cache share one GPU budget. Batching serves many requests per step.
- **02:** tensor parallelism splits every layer. Our Pod requested two GPUs on one node.
- **03:** pipeline parallelism splits the layers into stages. LeaderWorkerSet runs it across nodes.
- **04:** disaggregation splits the request by phase. The KV cache moves between GPUs.



EVIDENCE AND OPERATIONAL GUIDANCE

What you saw for yourself

LLM servers cache aggressively

The KV cache holds keys and values for every processed token. Its capacity is in tokens, not requests: about 3 full sequences fit where 8 were allowed.

Batching increased completed work

Concurrency 4 finished more requests than 1. For capacity decisions, inspect queues and latency alongside GPU activity.

GPU placement changes what can run

Our TP Pod needed two GPUs on one node. Our PP group required two nodes. Fast links reduce communication overhead.

KubeCon CloudNativeCon North America 2026

THANK YOU

Questions?

Next topics

- Quantization (FP8, NVFP4)
- Expert parallelism
- Tool calling
- Goodput targets

Session feedback

Scan to share your feedback.

Minting Your Own Tokens: The Server Side of the Token Economy

KubeCon | CloudNativeCon
North America 2026



Nilekh Chaudhari

Senior Software Engineer

Microsoft

LinkedIn



Suraj Deshmukh

Senior Software Engineer

CoreWeave

linktr.ee/surajssd