// kubecon + cloudnativecon north america 2026 · salt lake city



Dropping LLM inference cost by two-thirds with open weights and Kubernetes



Omisha Gupta

Solutions Architect, AWS

// three problems, three fixes

Proprietary LLM APIs are convenient.
And expensive at scale.

01

Every request is billed

Cost grows one-to-one with traffic.

02

You pay for unused intelligence

A general-purpose model doing one narrow job.

03

The model changes under you

New versions, new behaviour, new prices.

this talk → a smaller model that's better at the job, that you own, served elastically on Kubernetes

// token economics

Nobody stays on the cheap model.
Bills keep going up.

66%

of teams upgraded to their vendor's newest model

3–18×

per year: rising cost of running frontier models

3×

enterprise LLM spend in 12 months

93%

of enterprises over their AI budget

the newest model is never the cheap one

Menlo Ventures mid-2025 LLM market update (n=150) · Epoch AI, "The Price of Progress", 2025 · Menlo Ventures, Dec 2025 · McKinsey, May 2026

// token economics

Your bill has three dials.

In practice, all three go up.

bill = requests × tokens / request × price / token



more users, features, tenants, agents



avg prompt 1.5K → 6K tokens
agents ≈ 1,000× a chat



you upgrade to the newest model

more requests × longer prompts × pricier models

OpenRouter State of AI, Dec 2025 · McKinsey QuantumBlack, Jul 2026 · Menlo Ventures mid-2025

// my workload

A classification workload.

The API wasn't accurate enough.

01 · the task

Read the feedback, pick every label that applies

Often more than one.

“App keeps disconnecting, and my schedule reset.”

connection **schedule**

02 · with an API

A general-purpose model + a tuned prompt

Right only some of the time.

56%

exactly right, at best
newer version: worse

03 · what I did

Fine-tuned a small open model on our own data

Taught it the task.

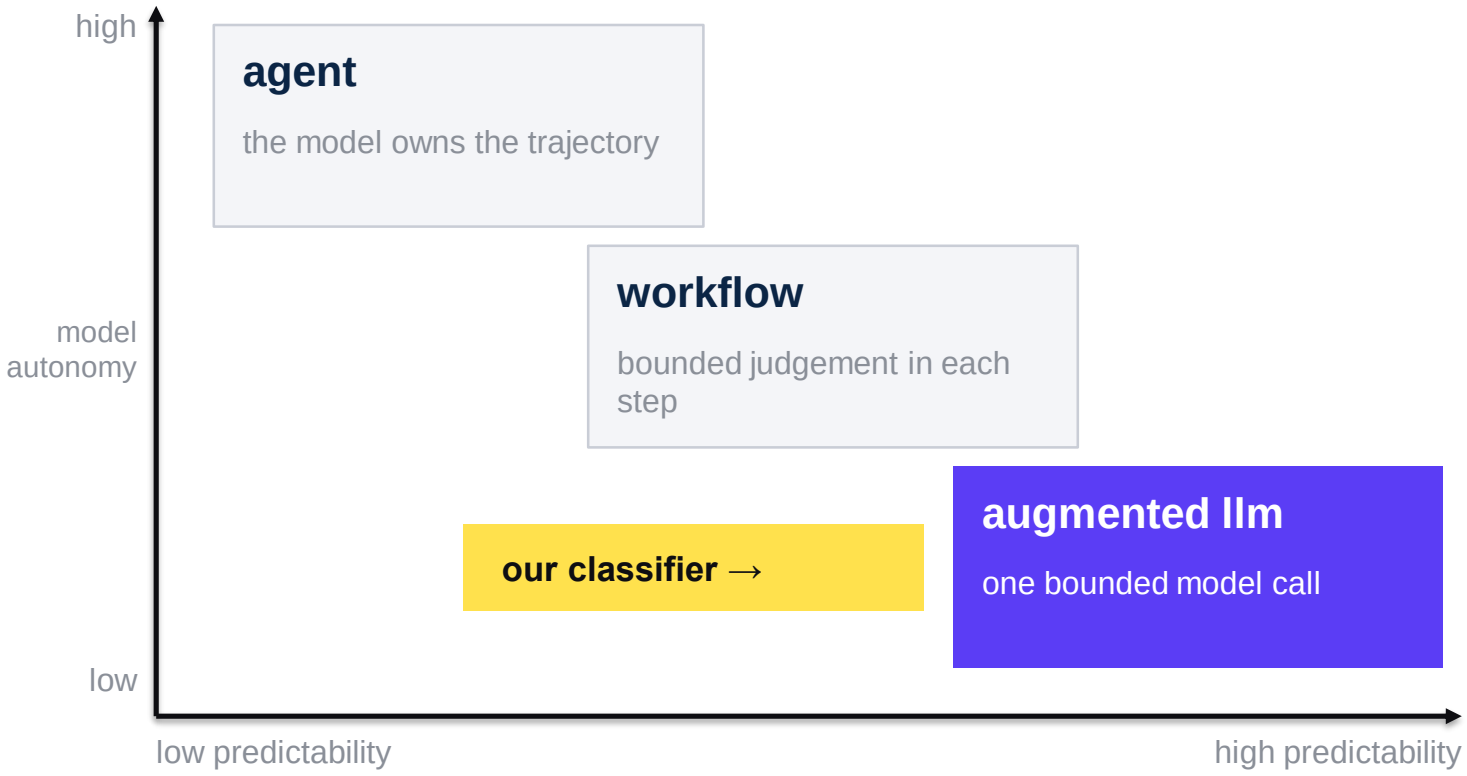
9.6K

labelled examples
8B model · one GPU

if the model keeps getting it wrong, teach it the task

// do you need that much intelligence?

Know where your use case sits before you pick a model.



Up and left

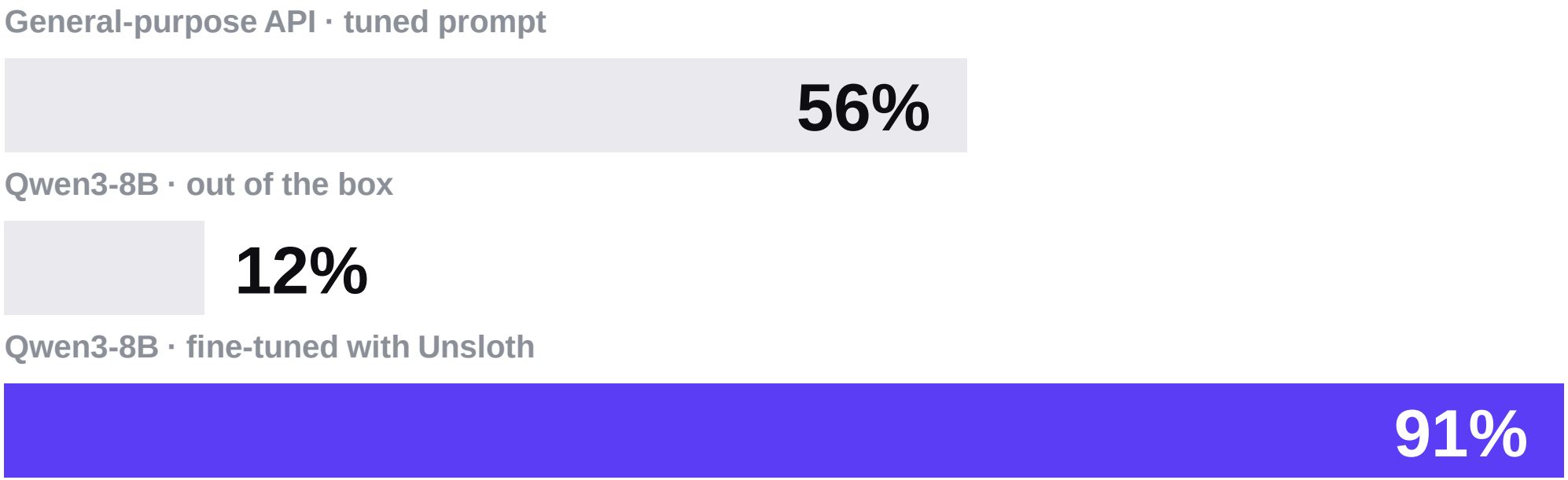
buys flexibility, spends predictability: harder to test, cap and price.

Down and right

a small model, a fixed prompt, testable output, a predictable bill.

// do you need that much intelligence?

You don't need a bigger model.
You need a trained one.



same model + training

exact match · n = 467 held-out records
untrained model: same prompt, answer accepted in any format

// when to fine-tune

Fine-tune when you can say yes to most of these.

fine-tune when

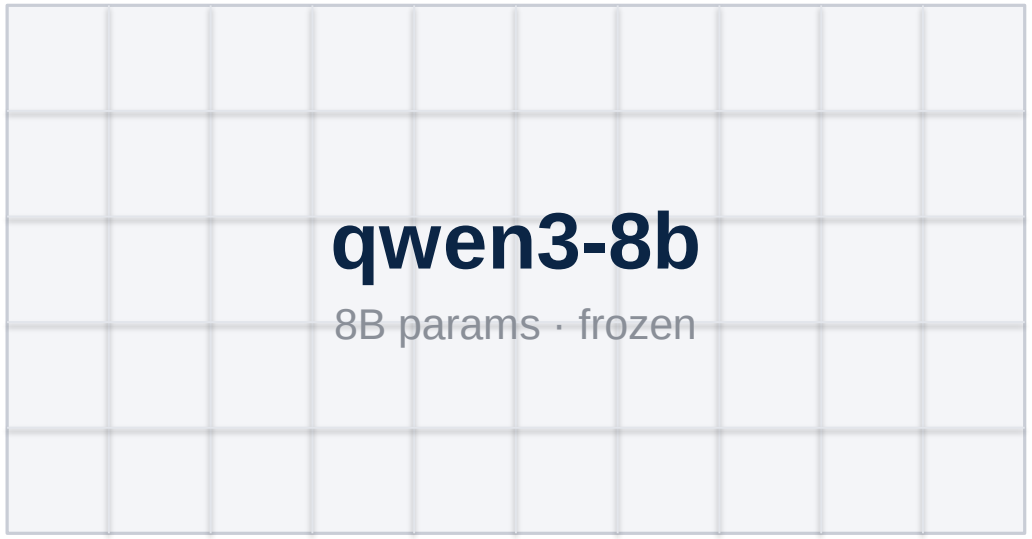
- ✓ **The task is narrow and repeats**
- ✓ **You have labelled examples (we had 9.6K)**
- ✓ **Volume is high enough that cost matters**
- ✓ **Prompting has plateaued (ours: 36–56%)**
- ✓ **The output must be predictable**

stay on a prompt when

- ✗ The task changes every week
- ✗ Volume is low
- ✗ It's open-ended reasoning
- ✗ You have no labelled data

// what's lora / qlora

Teach the model your task without retraining it.



trained

1%

add-on

334 MB

hardware

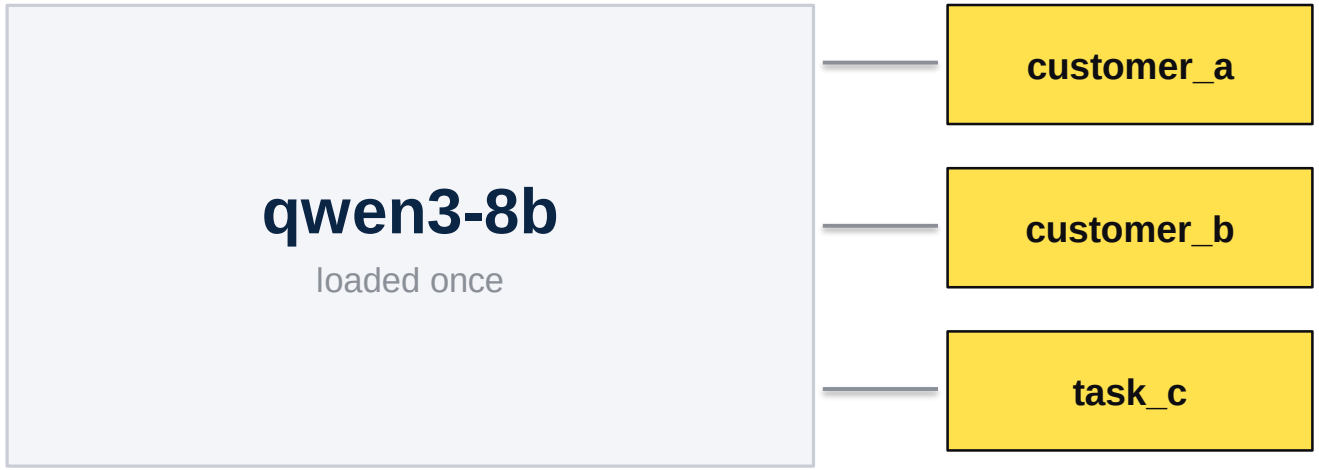
1 GPU · 12 h

like sticky notes on a textbook: the book stays the same, the notes teach it your task

QLoRA → keep the frozen part compressed while training, so it fits on one GPU

// how adapters work

One base model.
Many small add-ons.



hot-swap per request

base + adapter
merged into one model

simplest to serve · then quantize

one task → we merged, then quantized

// what is unsloth

Unsloth: fine-tune an 8B model on a single GPU.

open source

drop-in for Hugging Face TRL + PEFT

4-bit base

an 8B model trains in 24 GB

custom kernels

≈2× faster (Unsloth's claim)

```
from unsloth import FastLanguageModel

model, tok = FastLanguageModel.from_pretrained(
    "unsloth/Qwen3-8B",
    max_seq_length=4096, load_in_4bit=True)

model = FastLanguageModel.get_peft_model(model,
    r=32, lora_alpha=64,
    target_modules=["q_proj","k_proj","v_proj","o_proj",
                    "gate_proj","up_proj","down_proj"])
```

from our train.py, trimmed

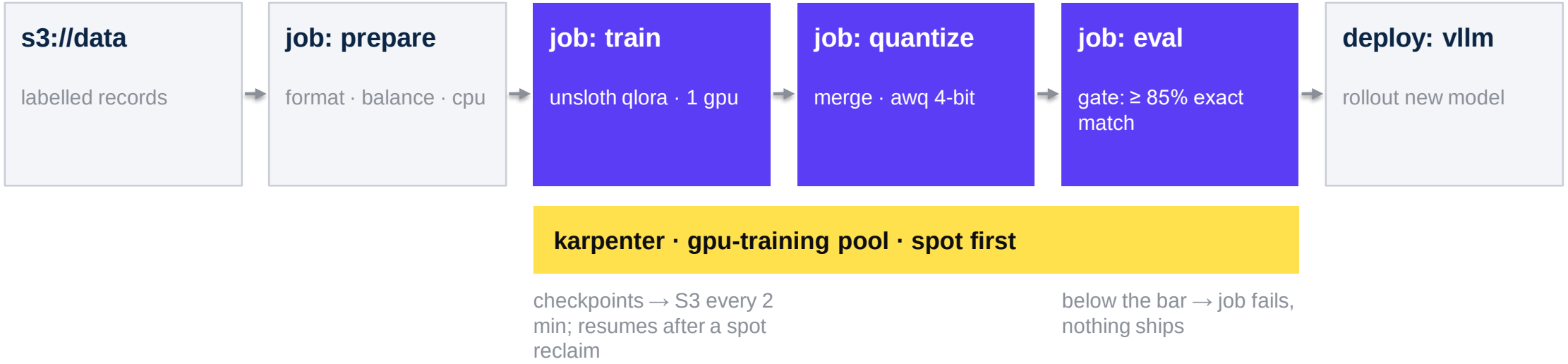
// how to fine-tune

The recipe:
five steps, one GPU.

01 · data	02 · format	03 · balance	04 · train	05 · evaluate
9.6K	**~840**	**×3**	**12 h**	**467**
labelled examples	tokens in · IDs out	multi-label examples	QLoRA r=32 · 1 GPU	held-out records

the eval set came before the model

The recipe, as Kubernetes Jobs.
GPUs show up for the job, then leave.



one cluster trains and serves

manifests: k8s/finetune/ · validated against the cluster, not yet run end to end

// what quantization is

Same model.
Shorter numbers.

a model is billions of numbers, called weights

before · 16-bit		after · 4-bit
0.7316284	→	0.73
-0.2874019	→	-0.29
0.0412653	→	0.04

illustrative values

round the numbers, keep the answers

¼ the space

each number stored in 4 bits
instead of 16

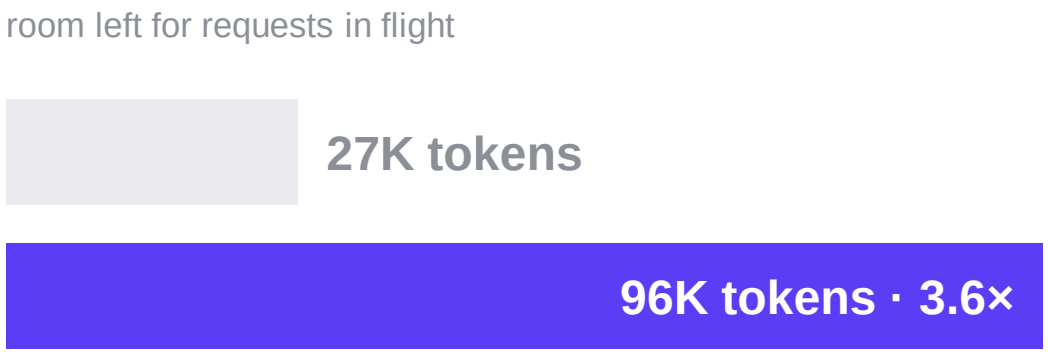
// why quantize

Shrink the model, keep the answers.

16 GB → 5.7 GB

16-bit weights → 4-bit weights

how: find the ~1% of weights that matter most, protect them, round the rest.



cost: **90.6%** → **89.3%** exact match

smaller model · bigger batches · same answers

// make it fit in a smaller instance

Everything on one 24 GB GPU.
About $1 an hour.



g5.xlarge · 1× NVIDIA A10G · 24 GB · $1.006/h on demand · measured by vLLM at startup

4-bit weights

16 GB → 5.7 GB

short prompts

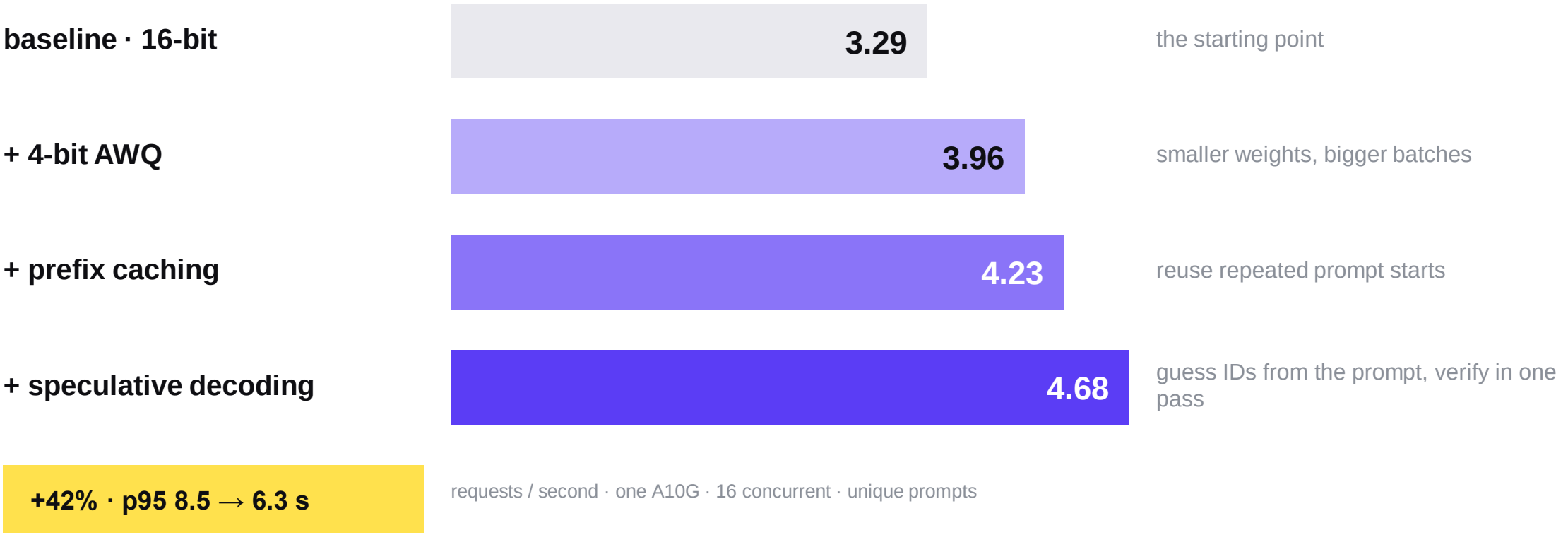
~3,800 → ~840 tokens

context cap

4,096 tokens · ~22 at once

// serving it with vllm

Same GPU.
42% more requests.



// kubernetes setup required

The stack, bottom to top.

prometheus + keda	scrape every 5 s · scale on requests in flight
vllm deployment	1 GPU per pod · model pulled from S3 (Pod Identity)
nvidia device plugin	the scheduler can hand out GPUs
karpenter	GPU NodePool · g5 · spot → on-demand · cap 12 GPUs
eks	Kubernetes 1.34 · 2× m5.xlarge for system pods

// how pods take this up

How pods pick up the load.

1 requests pile up

vLLM: running + waiting

2 prometheus scrapes

every 5 s

3 keda decides

in-flight ÷ 16 = replicas

4 pod is pending

no free GPU yet

5 karpenter adds a node

g5 ready in ~40 s

6 pod joins the service

model loads · traffic flows

scale on work in flight, not on GPU

// the config that does it

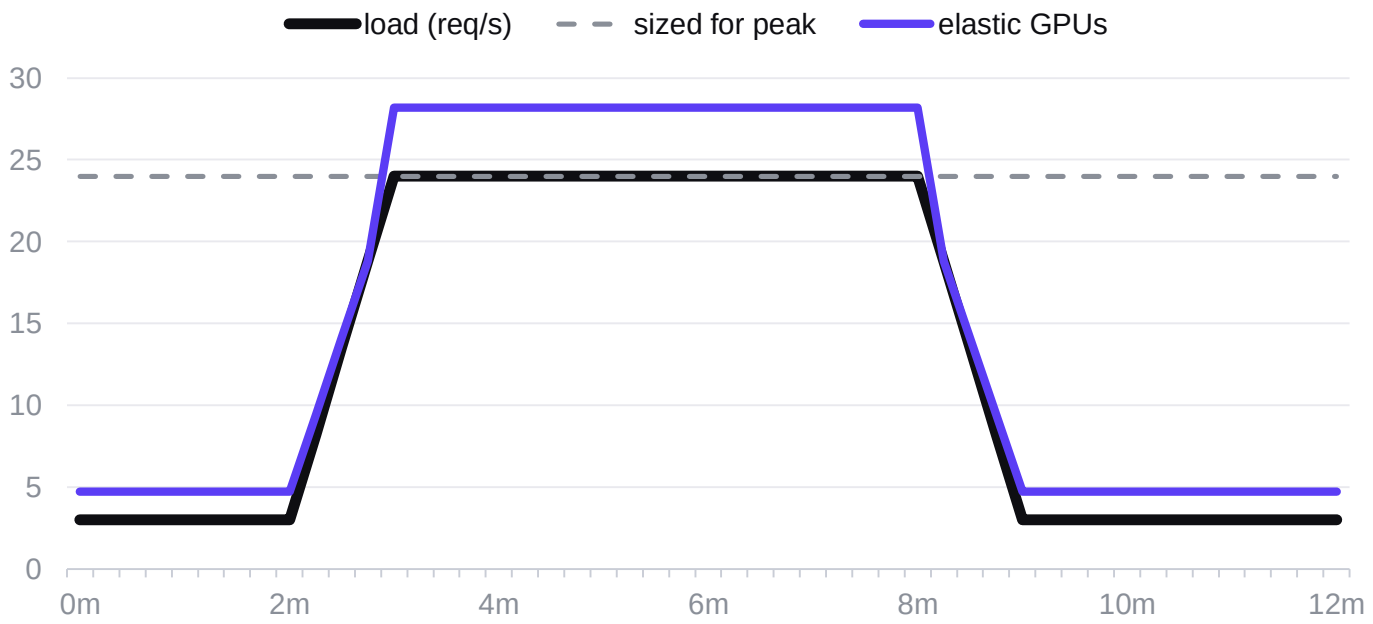
About 40 lines of YAML.

```yaml
kind: NodePool          # karpenter.sh/v1
spec:
  template:
    spec:
      requirements:
      - key: node.kubernetes.io/instance-type
        values: [g5.xlarge, g5.2xlarge, g6.xlarge]
      - key: karpenter.sh/capacity-type
        values: [spot, on-demand]
  limits:
    nvidia.com/gpu: 12      # spend guardrail
  disruption:
    consolidationPolicy: WhenEmpty
    consolidateAfter: 2m
```

```yaml
kind: ScaledObject      # keda.sh/v1alpha1
spec:
  scaleTargetRef: {name: vllm}
  minReplicaCount: 1        # warm floor
  maxReplicaCount: 10
  triggers:
  - type: prometheus
    metadata:
      query: >-
        sum(vllm:num_requests_running)
        + sum(vllm:num_requests_waiting)
      threshold: "16"     # per replica
```

// why kubernetes

Traffic doesn't wait for capacity.
Capacity should follow the traffic.



GPU-aware scheduling

NVIDIA device plugin · 1 GPU per pod

scale on any metric

KEDA · requests in flight

nodes on demand

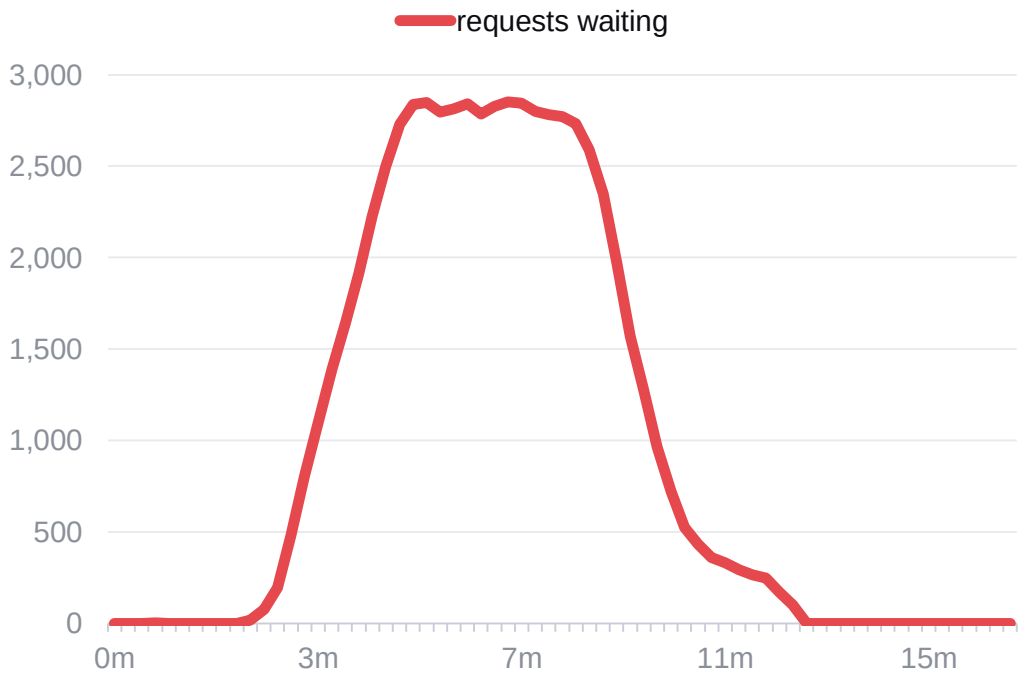
Karpenter · seconds to launch

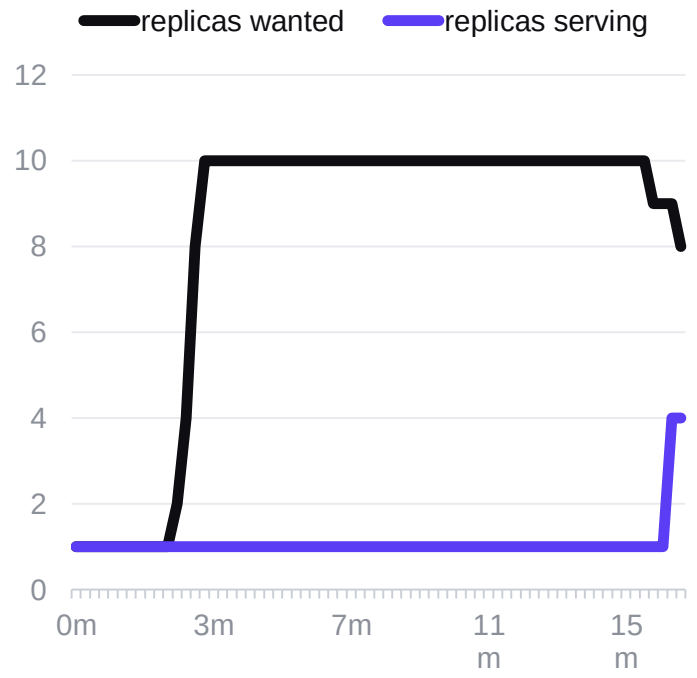
declarative + portable

same YAML, any cluster

// surge test · run 1 · measured

The autoscaler was right.
The capacity was late.



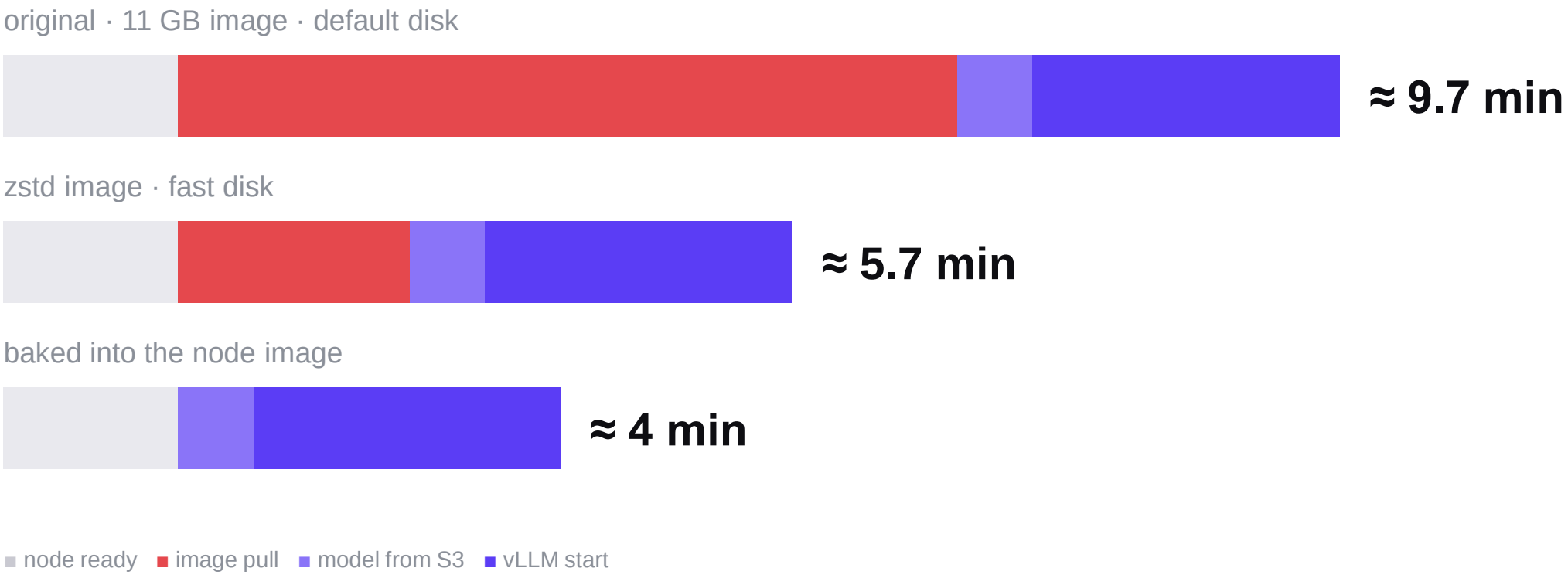


18%

of requests completed

// why a new GPU was slow

Shrinking the model didn't shrink the container image.



RUN 2 TO MEASURE

A new GPU is ~4 minutes from useful. Plan for that.

01	**smaller image**	zstd + fast disk · pull 5m40 → 1m41
02	**bake the node image**	vLLM pre-pulled · pull → 0 s
03	**keep headroom**	warm floor + placeholder pods
04	**flexible capacity**	several GPU types and zones
05	**watch platform automation**	patch reboots hit new nodes at ~4 min

keep headroom for the burst you can't wait 4 minutes for

// what we got

Better answers.
A fraction of the cost.

quality

56% ↑

91%

exact match

cost

100% ↓

~33%

estimate · same traffic, gpus kept warm

grey = before (general-purpose API) · purple = fine-tuned 8B on kubernetes

// take-home checklist

The blueprint.

01	**place the use case**	is it really one bounded call?
02	**measure**	tokens per request · traffic shape · an eval set
03	**fine-tune small**	QLoRA with Unsloth on your labelled data
04	**merge + quantize**	AWQ, then re-run the eval
05	**serve with vllm**	batching · prefix caching · speculative decoding
06	**scale on work in flight**	KEDA on running + waiting
07	**right-size with karpenter**	spot + on-demand · many GPU types · a spend cap
08	**attack cold start**	small image · baked node image · headroom

Thank you.

Feedback Please!



Omisha Gupta

Solutions Architect, AWS
Linkedin - https://www.linkedin.com/in/omishagupta/

Dropping LLM Inference Cost by Two-Thirds with Open Weights and Kubernetes