DLIT71339: The Speed of Thought: Navigate LLM Inference Autoscaling for a Gen AI Application Toward Production

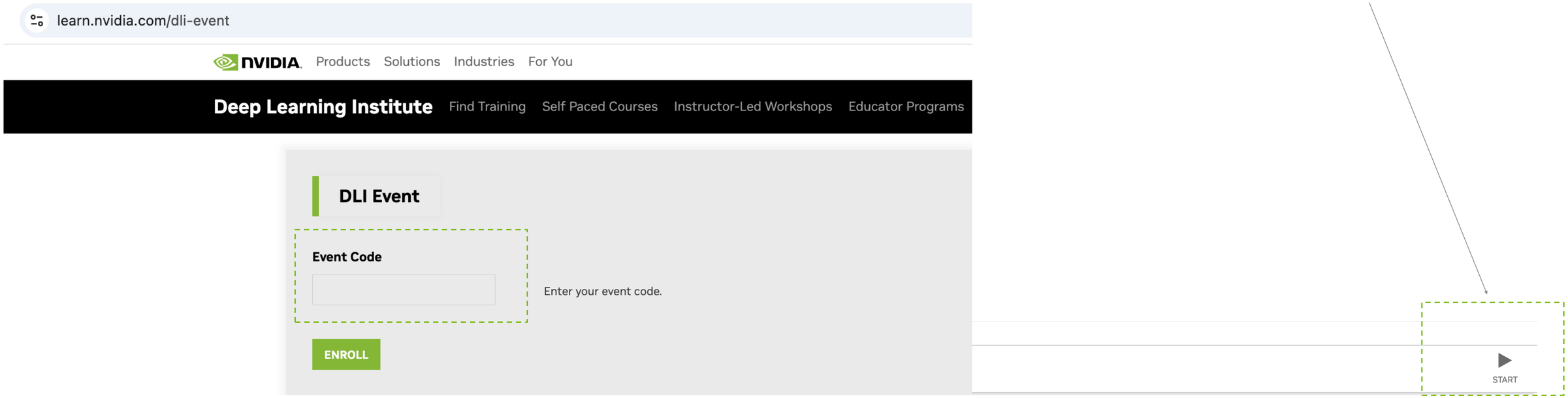
Mohak Chadha, Dmitry Mironov, Sergio Perez | GTC 2025

Launching the Lab

Take a picture of this slide

- Go to https://learn.nvidia.com/dli-event
- Enter event code:
- You may need to login/create account.

Click on this



Agenda

- Challenges of LLM Inference in Production
- NIMs
- Notebook 1: NIM Operator for K8s
- Notebook 2: Benchmarking NIM
- Notebook 3: Observability
- Notebook 4: Autoscaling with Custom Metrics

Challenges of LLM Inference in Production

Inference Latency (SLAs)

Tokens

- Units of text that the model processes
- One token generally corresponds to ~4 characters of text for common English text (~ ¾ of a word)
- Input sequence length of tokens (**ISL**) are fed into a model. LLM generates of output sequence length (**OSL**) of tokens one at a time

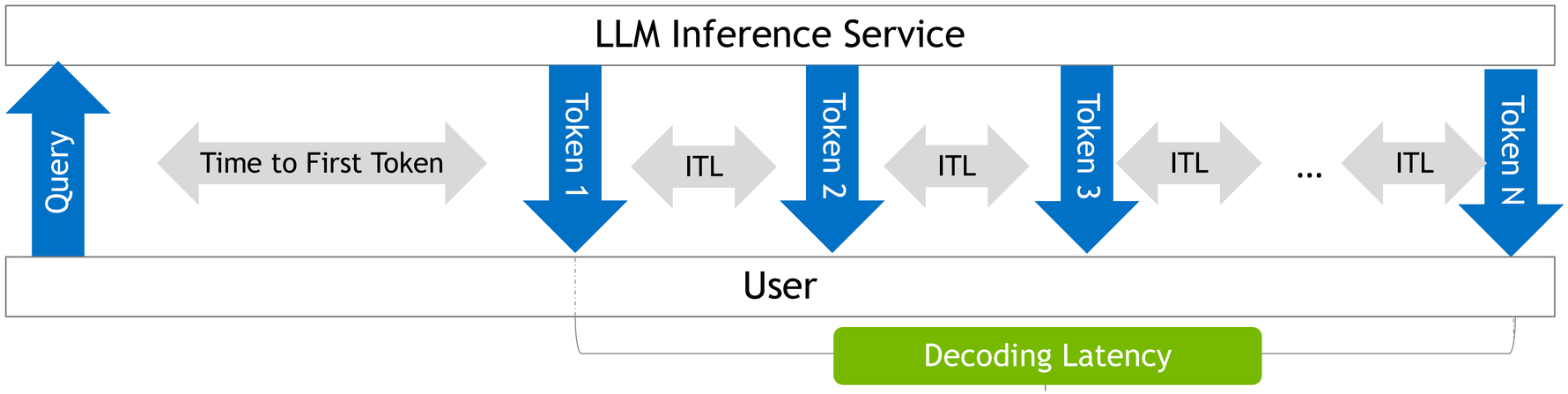
```
Tokenization breaks a word into
tokens of about 4 characters
```

Time to First Token (TTFT)

- Time it takes for the model to generate the first token after receiving a request
- This part of execution is called **Prefill**
- Critical for chat-like applications

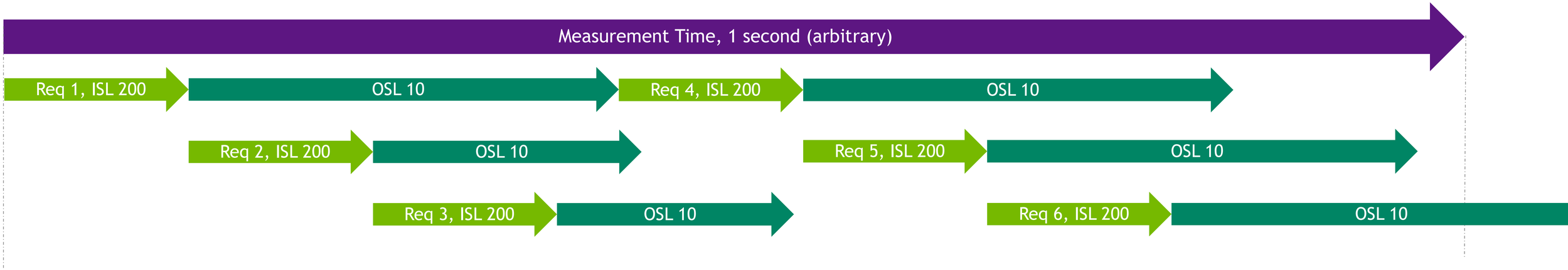
End-to-end Latency (E2E)

- Time it takes for a model to generate a complete response to a user's prompt.
- E2E Latency = TTFT + **Decoding** Latency
- Inter-token Latency (**ITL**) is an average time between output tokens
- Critical, when full response has to be processed further: guardrails, tool calling



Inference Throughput Example (Costs)

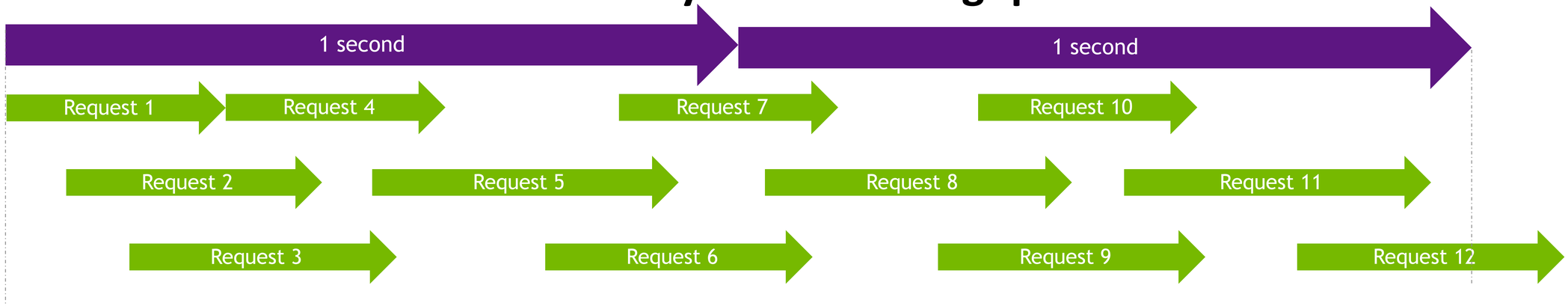
Assuming 1 GPUs per model, 1 instance



Legend: Prefill Decoding

- Requests per second per deployments (**RPS**) = 5 started and completed requests / 1 second = 5 (200 tokens in, 10 tokens out)
- Output tokens per second per deployments (**TPS**) = 57 output tokens / 1 second = 57 (200 tokens in, 10 tokens out)
- **Requests per second per GPU** = RPS / #GPU = 5 / 1 = 5 (200 tokens in, 10 tokens out)
  - If 2X RPS needed keeping latency the same, need 2X servers.
  - **Best definition of throughput. Direct relationship to costs. Always try to calculate it.**

Concurrency is NOT throughput



- Top system: **concurrency** 3 (serving 3 concurrent requests), average E2E Latency 250 ms, RPS = 5.5
- Bottom system: **concurrency** 6, average E2E Latency 2000 ms (slower!), RPS = 3 (more GPUs needed!)
- Application developers may call it 20 concurrent users: some are typing, some are listening to the generated response.

Optimizing for Latency or Throughput

Significant impact of inference strategy

Online — live generation

- Complexity: it matters to people how quickly they will get their response
- Imposing latency requirement significantly decreases available throughput. Need to balance between throughput and latency

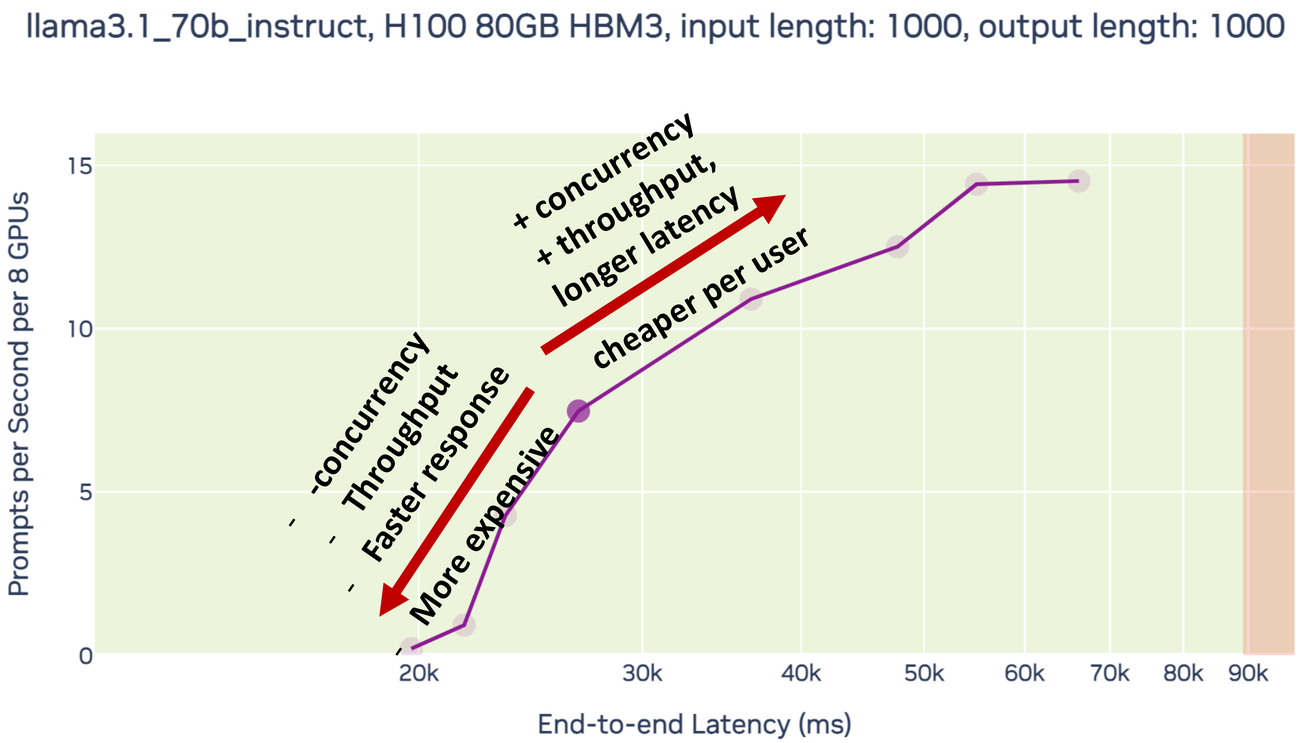
Offline — postponed computation

- Simplest execution model
- Throughput, throughput, throughput: maximum GPU utilization, maximum batch size

Fun fact: Fast human reading speed is 90 ms/token (=500 words/minute at 0.75 tokens/word) (avg is 200 ms/token)

Plot of latency versus throughput

Optimizing inference means selecting a dot in this plot

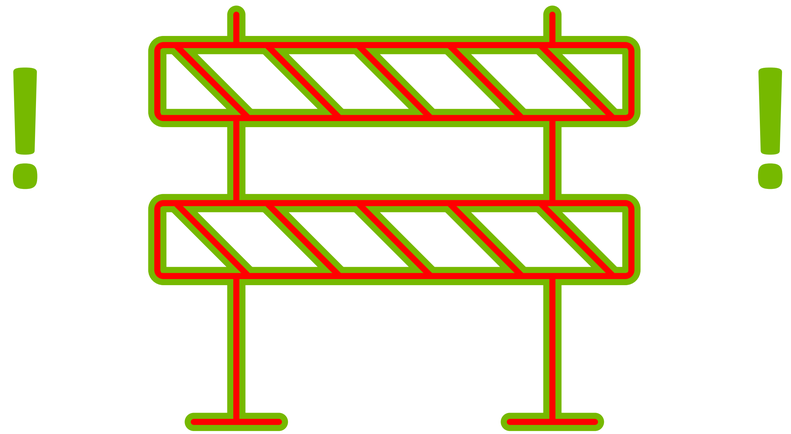


Concurrency is related to batch size:

- GPUs are better utilized with high batch sizes
- High batch sizes means better throughput
  - ✅ More prompts per second
- But high batch size means that the user has to wait for longer to receive the response
  - ❌ Higher latency

Variables impacting the throughput-latency curve

1. ✅ What model are you planning to use?
2. ✅ What is the average number of tokens in the prompt to your LLM (Length of input)?
   - For English one token is approximately 0.75 of a word.
   - Make sure to include system prompt.
3. ✅ What is the average number of tokens in your LLM output?
4. ✅ How many requests per second should your system process at its peak?
5. ✅ What is your latency limit? First-token? Last-token?
6. ✅ What GPUs are you considering?



Autoscaling Example

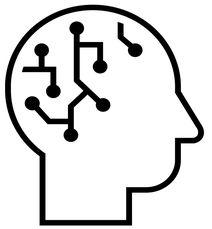
Customers expect SLA of latency and throughput

Customers

Queries

Responses

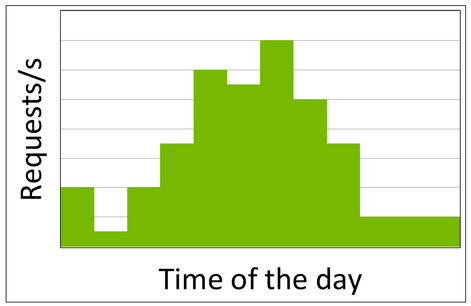
LLM Chatbot



Real sizing example:
Llama 3.1 70B
1000 in / 1000 out

8 H100 can serve
(at concurrency 200)

- Throughput: 7.5 requests/s
- Latency: 26s



How can we keep a similar SLA of Latency when the requests/s (Throughput) increase?

Variable distribution of request/s during the day

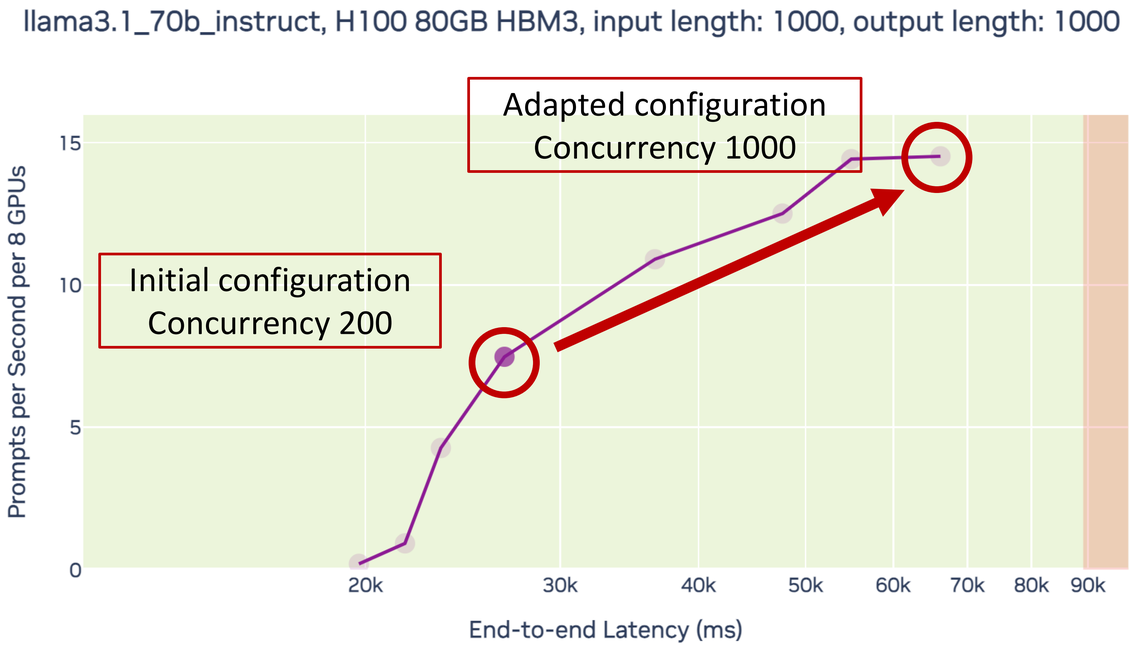
Options when requests/s increase:

1. Keep same number of 8 GPUs but increase concurrency
   - Throughput increases
   - Latency is penalized

2. Keep concurrency but autoscale to more GPUs
   - Allows to keep same latency for the users
   - Autoscaled pods scale throughput

Scenario 1: Keep same number of 8 GPUs

Number of requests per second doubles from 7.5 to 15



Initial configuration:

- Throughput: 7.5 requests/s
- Latency: 26s

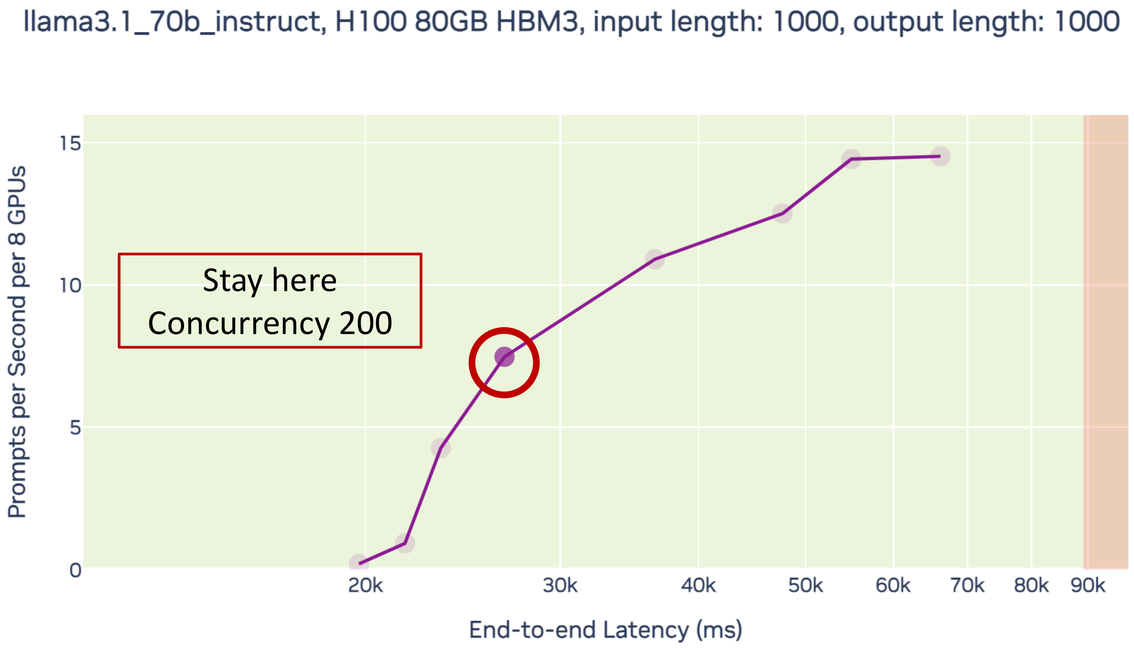
Adapted configuration:

- Throughput: 15 requests/s
- Latency: **66s**

With the same 8 GPUs, we can serve 15 requests/s instead of 7.5
— but the latency for each users goes up from 26s to 66s!

Scenario 2: Autoscale from 8 to 16 GPUs

Number of requests per second doubles from 7.5 to 15

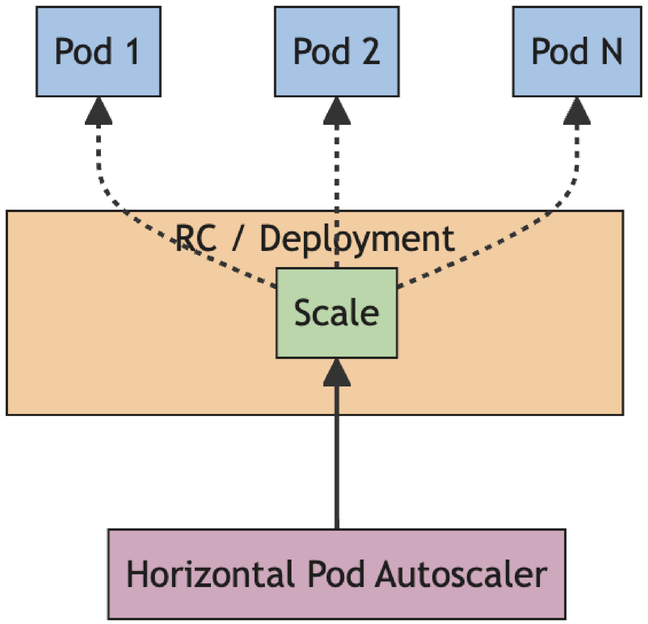


8 GPUs
- Throughput: 7.5 requests/s
- Latency: 26s

16 GPUs
- Throughput: **15 requests/s**
- Latency: 26s

Doubling the GPUs allows to keep the same latency
— but we can serve double the requests/s

Autoscaling to maintain latency SLA



When demands increases, it's important to autoscale efficiently

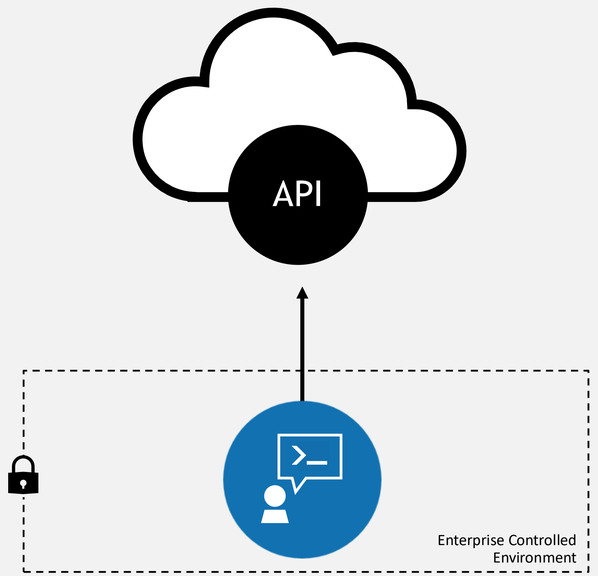
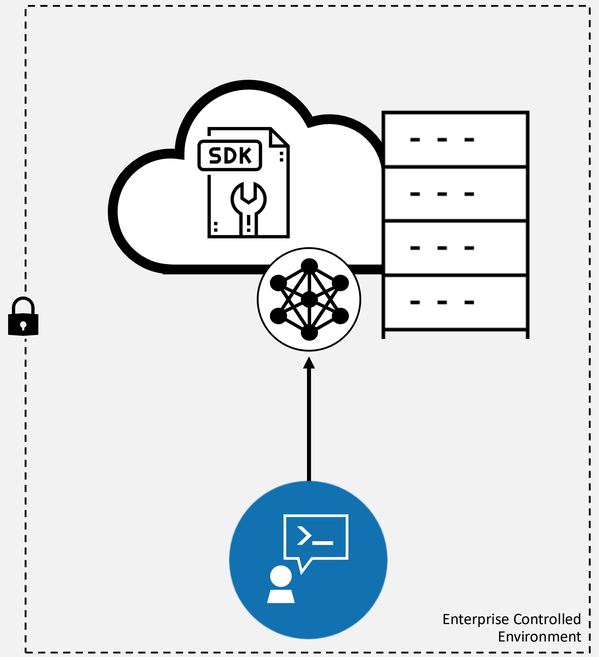
- If SLA can be met with same number of GPUs, no need to autoscale
- If latency increases too much, autoscaling is needed

In this tutorial you'll learn:

1. Reason about throughput-latency to optimize inference
2. Autoscale in K8s when a variable like request/s is high
3. Easily deploy LLMs in K8s with NIM Operator

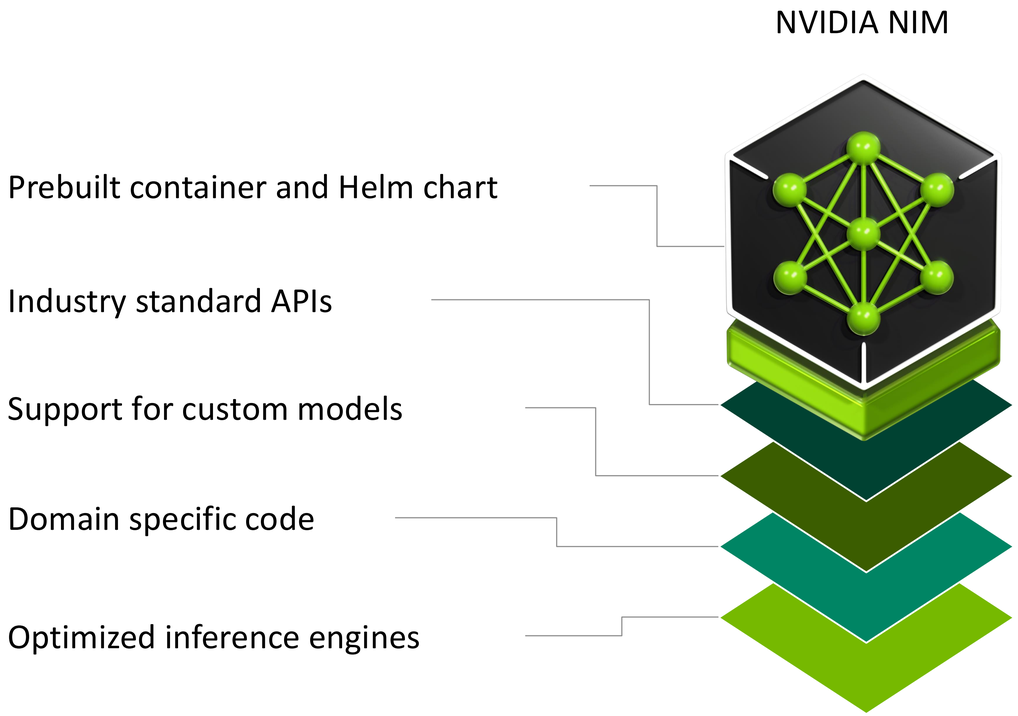
NIMs

Enterprises Face Challenges Experimenting with Generative AI

	Managed Generative AI Services	Do-It-Yourself Deployment	
Easy to use APIs for development			**Run anywhere** across data center and cloud
Fast path to getting started with AI			**Securely manage** data in self hosted environment
Infrastructure limited to managed environment			**Optimization required** for specific infrastructure
Data and prompts are **shared externally**			**Custom code** for APIs and fine-tuned models
Limited control for overall generative AI strategy			**Ongoing maintenance** and updates

NVIDIA NIM Optimized Inference Microservices

Accelerated runtime for generative AI



Portable Deploy anywhere with security and control of AI applications and data

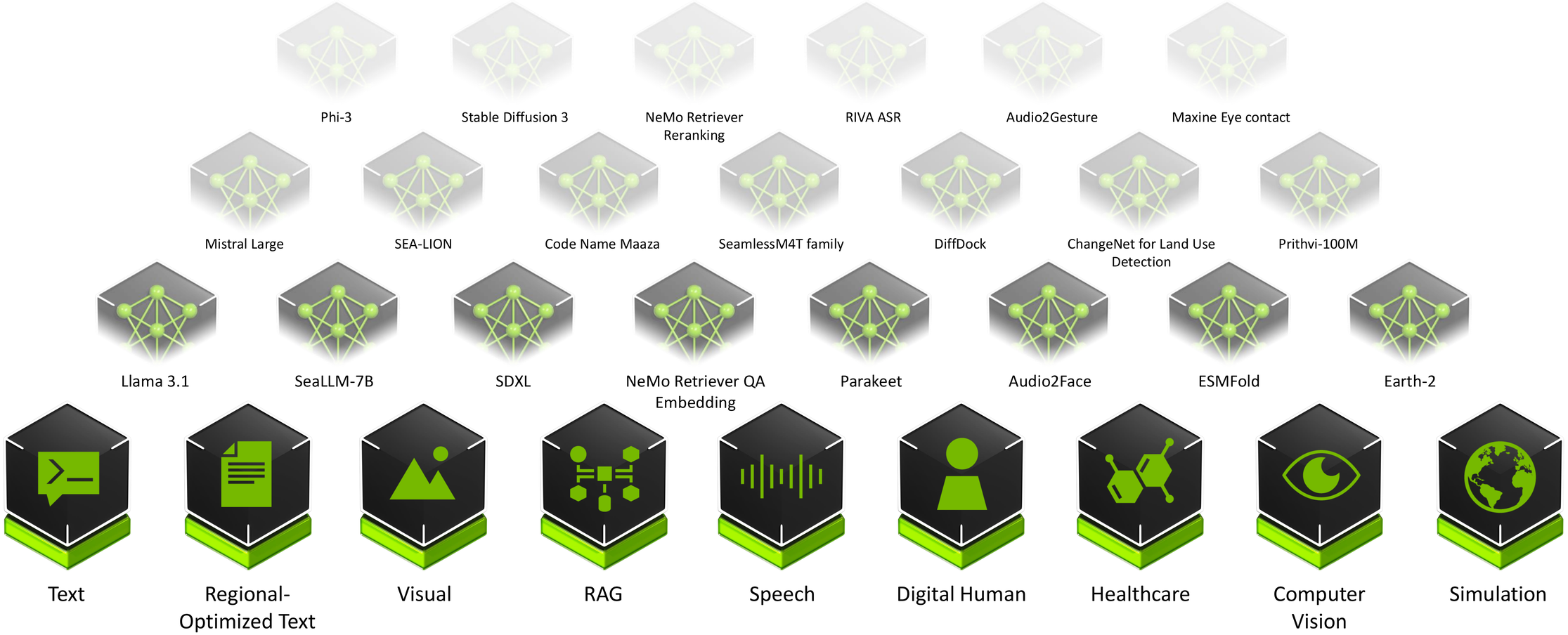
Easy to Use Speed time to market with prebuilt, continuously maintained microservices

Enterprise Supported Gain confidence with API stability, security patching, quality assurance and support

Performance Optimize throughput to maximize token generation for lower TCO and ROI

Microsoft Azure · aws · Google Cloud · ORACLE · DGX & DGX Cloud · DELL Technologies · Hewlett Packard Enterprise · Lenovo · SUPERMICRO

NVIDIA NIM For Every Domain



NVIDIA NIM

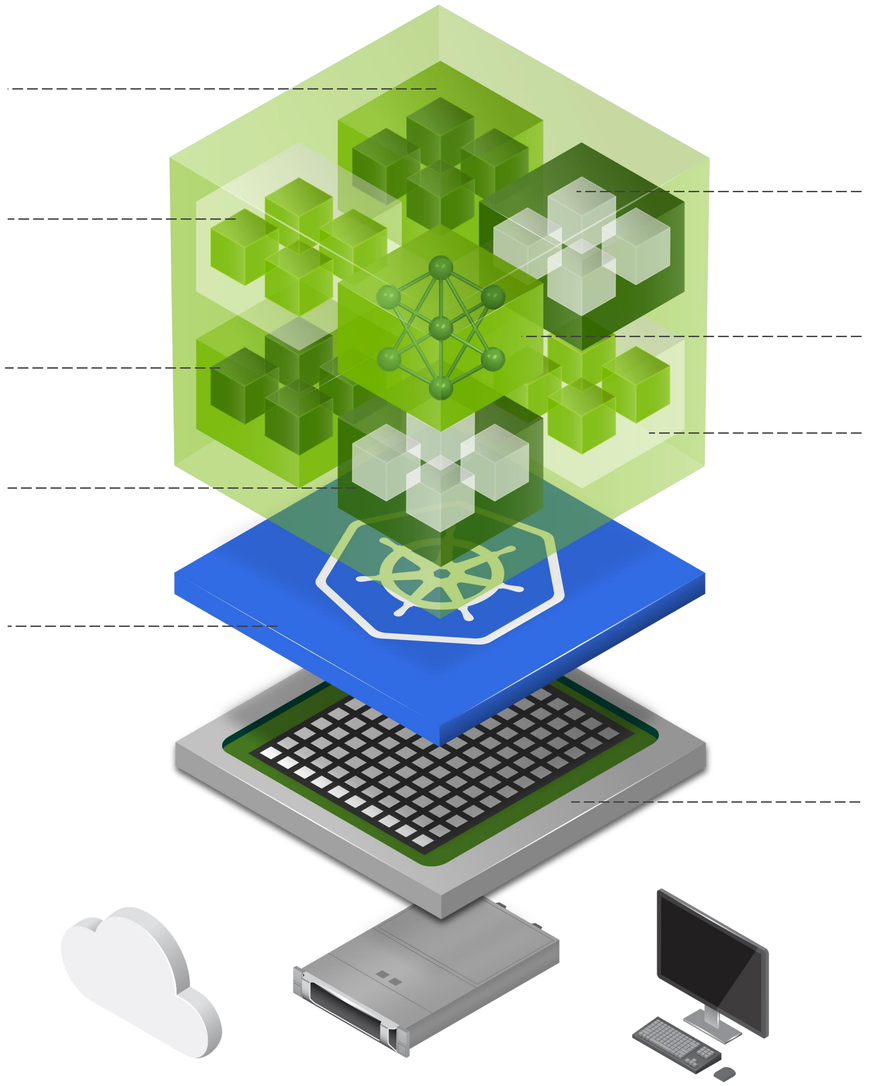
Standard APIs
Text, Speech, Image,
Video, 3D, Biology

NVIDIA Triton Inference Server
cuDF, CV-CUDA, DALI, NCCL,
Postprocessing Decoder

Cloud-Native Stack
GPU Operator, Network Operator

Enterprise Management
Health Check, Identity, Metrics,
Monitoring, Secrets Management

Kubernetes



NVIDIA TensorRT and TensorRT-LLM
cuBLAS, cuDNN, In-Flight Batching,
Memory Optimization, FP8 Quantization

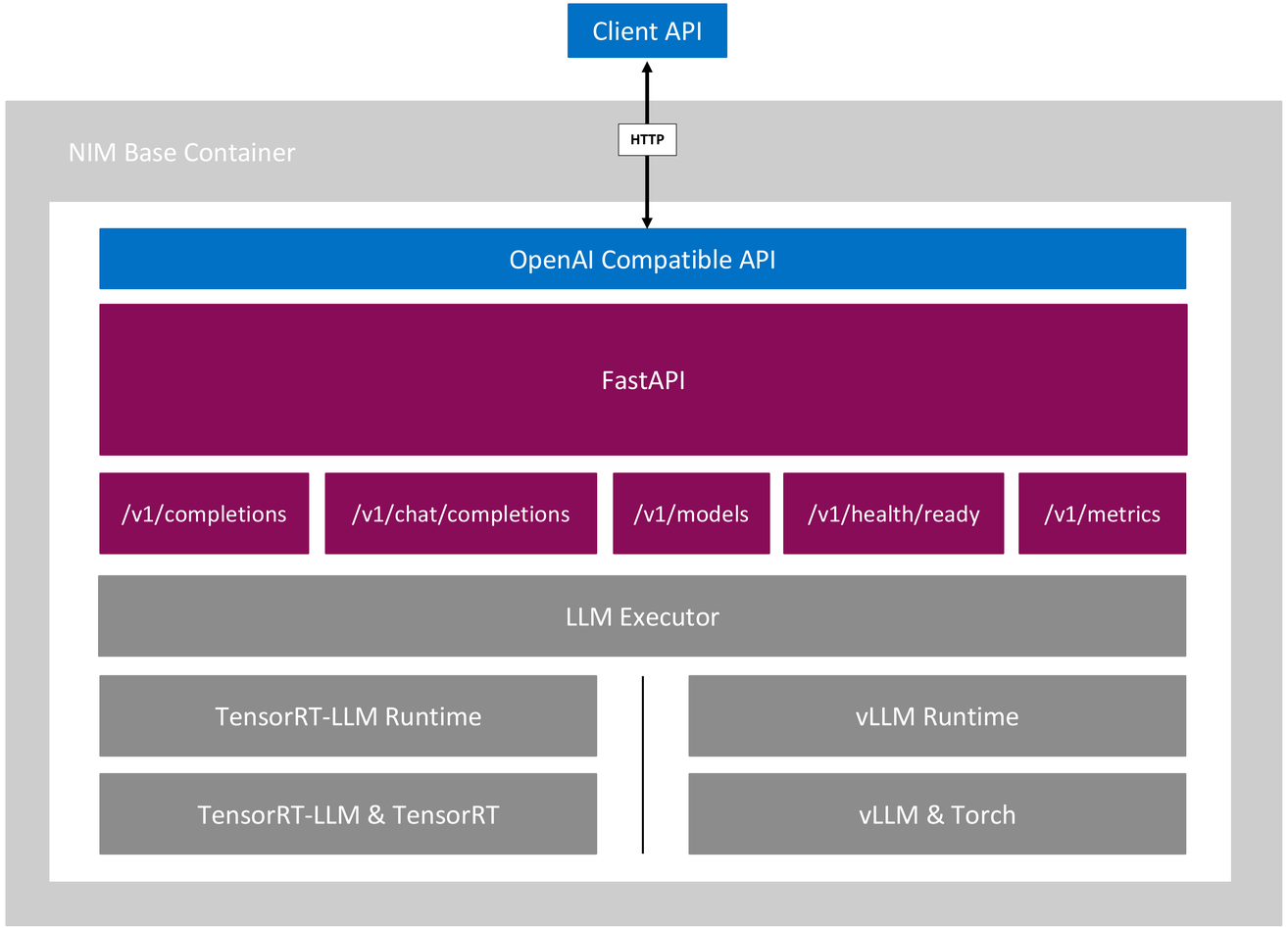
Optimized Model
Single GPU, Multi-GPU, Multi-Node

Customization Cache
P-Tuning, LoRA, Model Weights

CUDA

NVIDIA NIM for LLM Architecture

- HTTP REST API conforms to OpenAI specification for easy developer integration
- Liveness, health check and metrics endpoints for monitoring and enterprise management
- NVIDIA NIM includes multiple LLM runtimes
  - TensorRT-LLM and vLLM
  - Runtime is selected based on detected hardware and available optimized engines, with preference given to optimized engines



NIM Deployment

Easily manage cost-efficient autoscaling across NVIDIA accelerated cloud infrastructure

NIM on baremetal, VMI, AKS, Azure ML and Azure AI Studio

NIM Deploy - Azure

NIM on baremetal, AMI, EKS and Amazon SageMaker

NIM Deploy - AWS

NIM on baremetal, VMI, GKE and Google Cloud Vertex AI

NIM Deploy – Google Cloud

NIM on baremetal, CMI, OKE, and OCI Data Science Service

How-to Blog

NIM on KServe Inference Platform on Kubernetes

NIM Deploy - KServe

Simplified NIM Deployment on Kubernetes

NIM Deploy - Helm

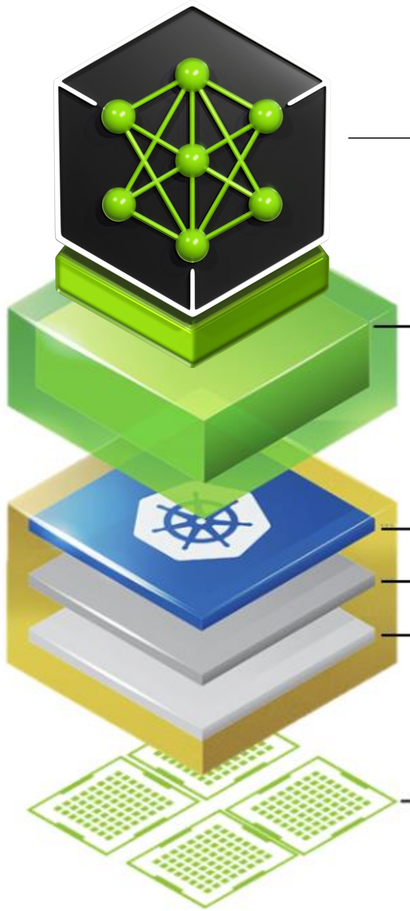
NIM Deployment Lifecycle Management on Kubernetes

NIM Operator

Notebook 1: NIM Operator for K8s

Production Environment Overview

The Lab Environment



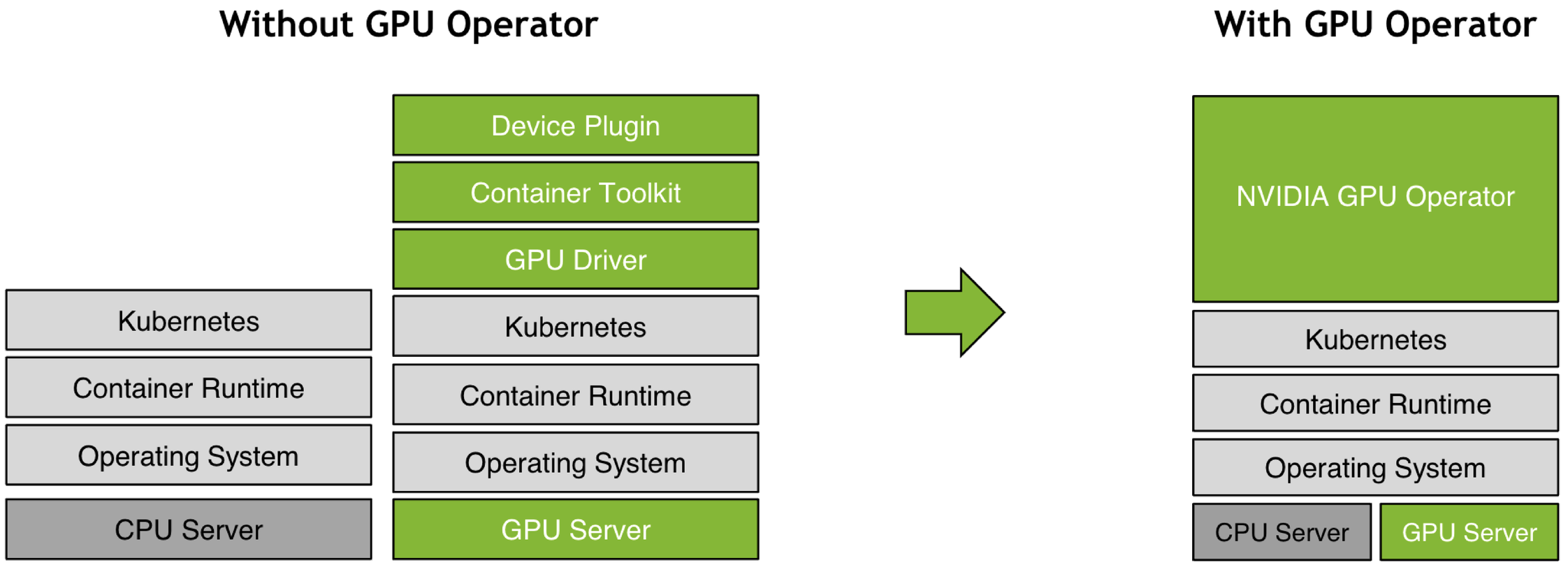
Easily deploy GenAI pipelines

Automate software components for NVIDIA HW in K8s

Opensource container orchestration

Docker containers

GPU Operator



GPU Operator

NIM Deployment Paths

In addition to NIM Container

Benefits

- Easy out-of-the-box inference
- Easy for partners to integrate

Limitations

- Requires KServe stack
- Focus on Inference, does not support microservices

Core Users

- Partner Platforms Using KServe
- OSS KServe Users

Availability

- Reference implementation available in nim-deploy repo github.com/NVIDIA/nim-deploy

Benefits

- Deploy NIM in Kubernetes
- Full NVIDIA control of app

Limitations

- Difficult to chain together NIM microservices
- Difficult to lifecycle full applications

Core Users

- Kubernetes Platforms

Availability

- Available on NGC
- Reference code available in nim-deploy repo github.com/NVIDIA/nim-deploy

NIM Operator

Benefits

- Manage NIM deployments in Kubernetes
- Full NVIDIA control of app & lifecycle

Limitations

- Complex development lifecycle

Core Users

- CSP Platforms
- Kubernetes Platforms

Availability:

- MVP GA Sept 30

Easily Deploy Generative AI Pipelines with NVIDIA NIM Operator

NIMCache, NIMService, NIMPipelines

NIMCache

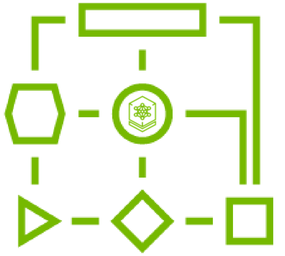
Improved performance with intelligent model pre-caching

Faster deployment of NIM microservices during inference requests and auto-scaling

NIMService

Lifecycle management of NVIDIA NIMs

Simplified deploy. upgrades, rollout for NIM microservices, and LoRA adapters



NIMPipelines

Automated AI pipelines deployment

Single click/command to deploy and lifecycle manage all generative AI application NIM microservices

Objectives of this notebook

1. Deploy the NIM Operator

2. Apply the manifests to deploy NIMCache and NIMService

3. Generate responses calling the NIM endpoint

Notebook 2: Benchmarking NIM

Stress-testing a client-server system

Simulate many client LLM requests

1. Simulate LLM requests

2. Measure time to get response
to calculate latency and throughput

Client

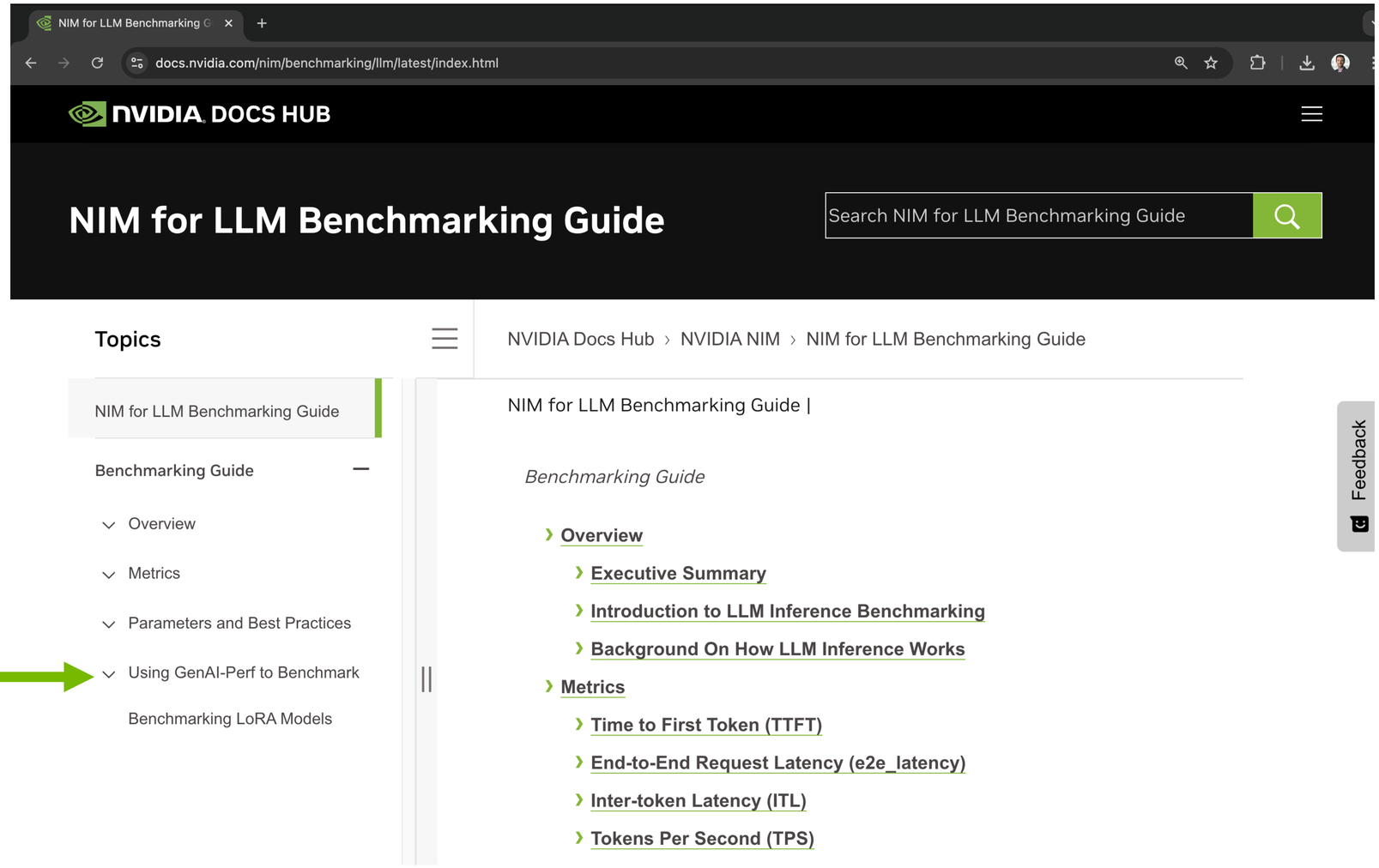
GenAI-Perf

Locust

Server

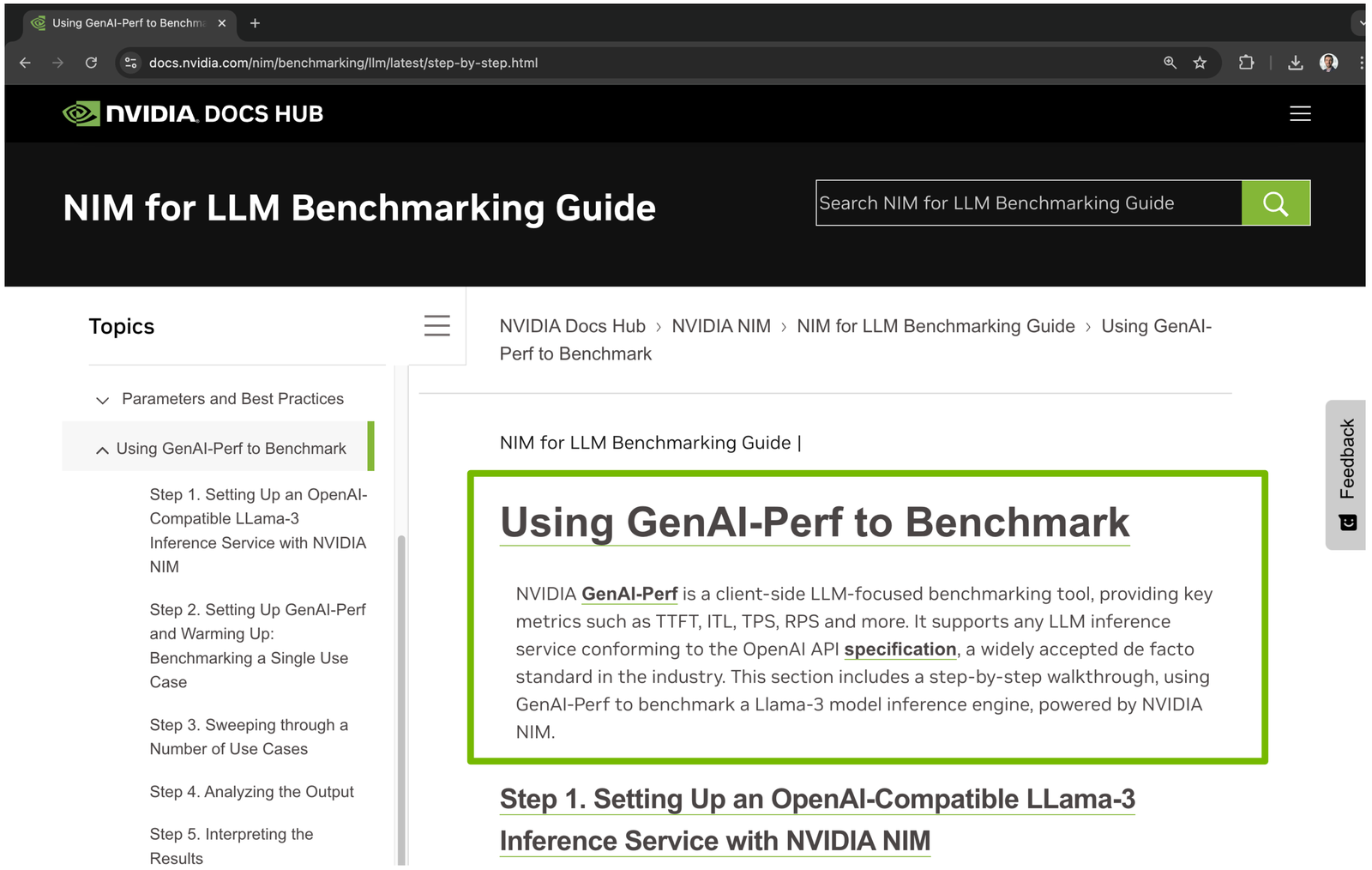
NIM for LLM Benchmarking Guide

https://docs.nvidia.com/nim/benchmarking/llm/latest/index.html



GenAI-Perf to benchmark LLMs

GenAI-Perf to Benchmark



GenAI-Perf command

Sample output generated by GenAI-Perf

```
export INPUT_SEQUENCE_LENGTH=200
export INPUT_SEQUENCE_STD=10
export OUTPUT_SEQUENCE_LENGTH=200
export CONCURRENCY=10
export MODEL=meta/llama3-8b-instruct

genai-perf profile \
    -m $MODEL \
    --endpoint-type chat \
    --service-kind openai \
    --streaming \
    -u "http://meta-llama3-1-8b-instruct:8000" \
    --synthetic-input-tokens-mean $INPUT_SEQUENCE_LENGTH \
    --synthetic-input-tokens-stddev 0 \
    --concurrency $CONCURRENCY \
    --output-tokens-mean $OUTPUT_SEQUENCE_LENGTH \
    --extra-inputs max_tokens:$OUTPUT_SEQUENCE_LENGTH \
    --extra-inputs min_tokens:$OUTPUT_SEQUENCE_LENGTH \
    --extra-inputs ignore_eos:true \
    --measurement-interval 5000 \
    --artifact-dir /genai-perf-results \
    --profile-export-file ${INPUT_SEQUENCE_LENGTH}_${OUTPUT_SEQUENCE_LENGTH}_${CONCURRENCY}.json
    -- \
    -v \
    --request-count $((10 * CONCURRENCY)) \
    --max-threads=256
```

LLM Metrics

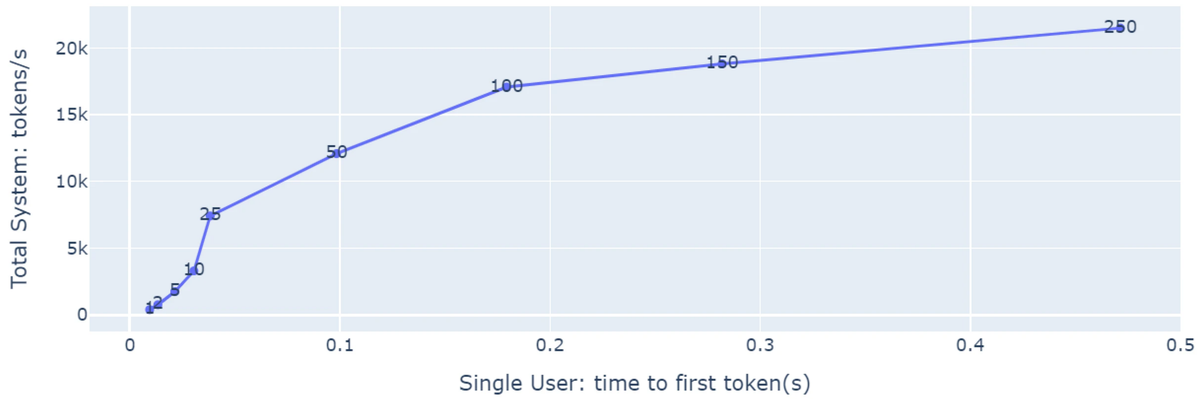
Statistic	avg	min	max	p99
Time to first token (ns)	85,485,242	27,402,273	152,621,817	130,194,943
Inter token latency (ns)	8,847,758	2,113,030	74,794,303	9,477,464
Request latency (ns)	1,848,822,497	1,844,511,394	1,924,017,143	1,905,132,459
Num output token	184	177	190	189
Num input token	200	198	201	200

Output token throughput (per sec): 995.61
Request throughput (per sec): 5.41

Sweeping across concurrencies

Running multiple GenAI-Perf calls

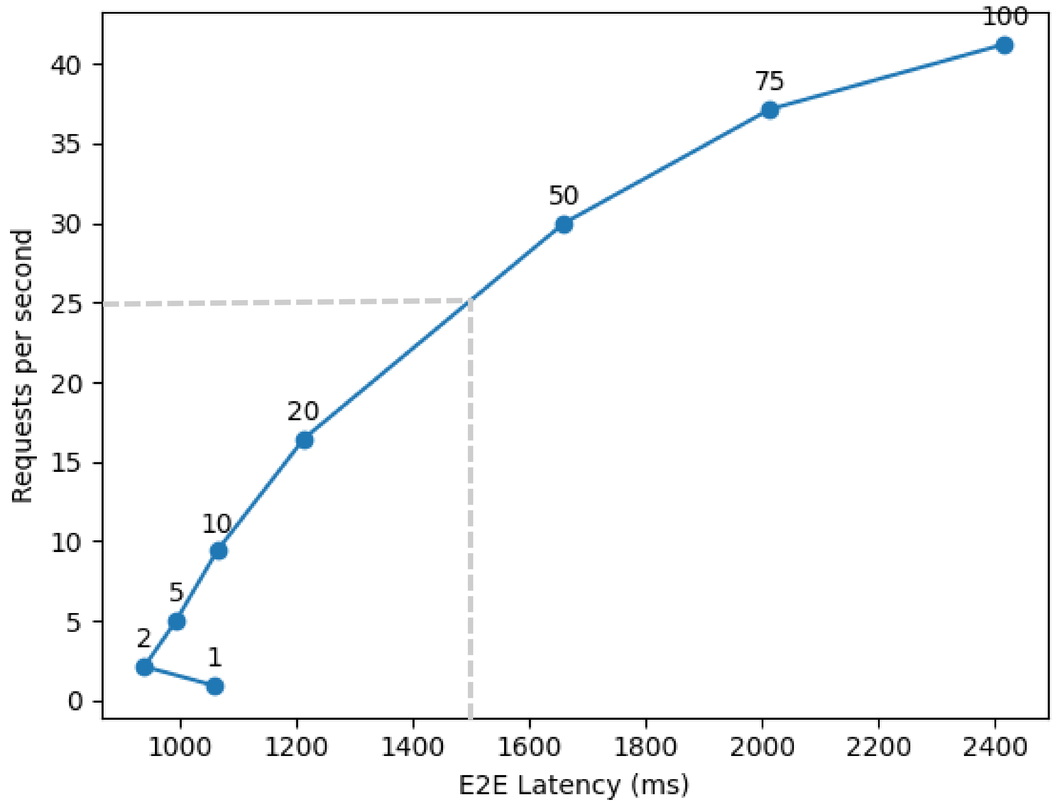
```
for concurrency in 1 2 5 10 50 100 250; do

    local INPUT_SEQUENCE_LENGTH=$inputLength
    local INPUT_SEQUENCE_STD=0
    local OUTPUT_SEQUENCE_LENGTH=$outputLength
    local CONCURRENCY=$concurrency
    local MODEL=meta/llama3-8b-instruct

    genai-perf \
        -m $MODEL \
        --endpoint-type chat \
        --service-kind openai \
        --streaming \
        -u localhost:8000 \
        --synthetic-input-tokens-mean $INPUT_SEQUENCE_LENGTH \
        --synthetic-input-tokens-stddev $INPUT_SEQUENCE_STD \
        --concurrency $CONCURRENCY \
        --output-tokens-mean $OUTPUT_SEQUENCE_LENGTH \
        --extra-inputs max_tokens:$OUTPUT_SEQUENCE_LENGTH \
        --extra-inputs min_tokens:$OUTPUT_SEQUENCE_LENGTH \
        --extra-inputs ignore_eos:true \
        --tokenizer meta-llama/Meta-Llama-3-8B-Instruct \
        --measurement-interval 10000 \
```



Throughput vs E2E Latency Measured

The Use Case in the Notebook

- Maximum RPS per instance around 51 requests/second
  - Measured with a very high concurrency
- With E2E Latency of 1.5 seconds, can sustain around 25 RPS per instance
- If 1.5 second is our target latency limit, this is the per instance throughput we should expect



Objectives of this notebook

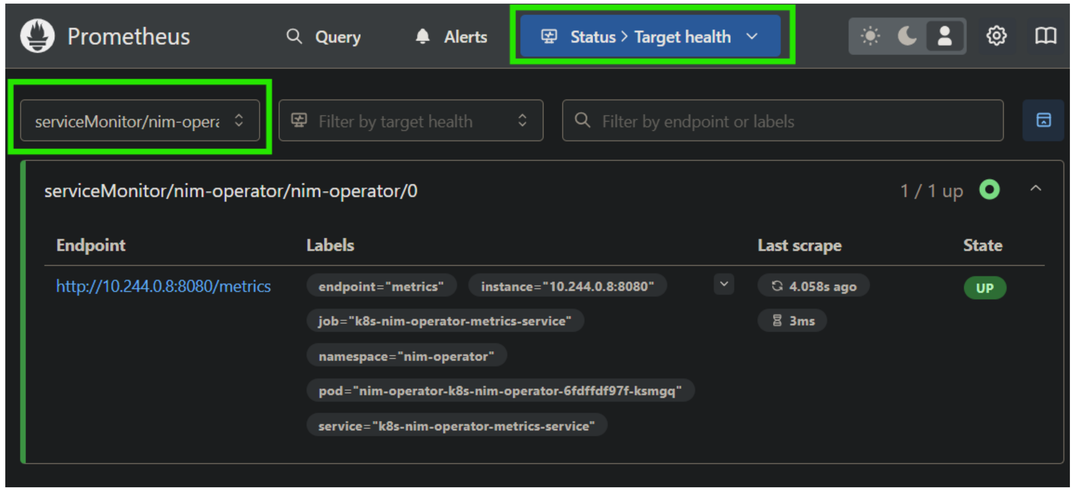
1. Read about latency and throughput metrics
2. Run GenAI-Perf in K8s and understand results
3. Run a sweep of concurrencies
4. Plot latency-throughput curves

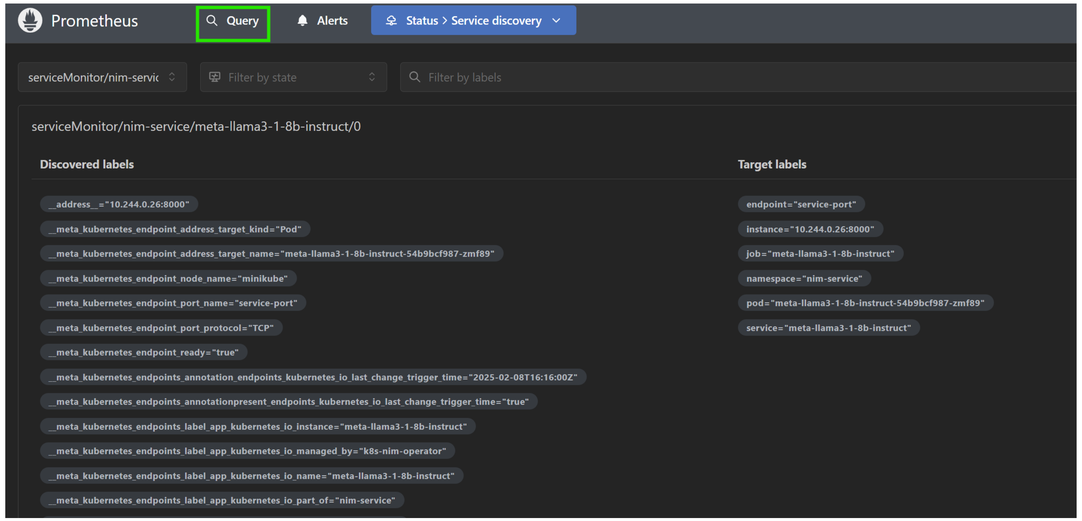
Notebook 3: Observability

Prometheus

An Open-Source Tool for Monitoring and Alerting Systems

- Prometheus periodically requests metrics from targets using a pull-based model
- Stores them in a time-series database
- A collection of microservices, installed as one Helm Chart
- Defines various types of the metrics





Gauge Metrics

```
gpu_cache_usage_perc{model_name="meta/llama-3.1-8b-instruct"} 0.75
```

- # HELP gpu_cache_usage_perc GPU KV-cache usage. 1 means 100 percent usage.
- # TYPE gpu_cache_usage_perc gauge
- gpu_cache_usage_perc{model_name="meta/llama-3.1-8b-instruct"} 0.005150951495206753
- 1. A help message explaining what the metric means
- 2. A type declaration specifying this is a `gauge` metric - meaning it shows a current value that can go up or down over time, there are no other assumptions about its behaviour.
- 3. The actual measurement data
- Label set: Appears in curly braces, containing `model_name="meta/llama-3.1-8b-instruct"`
- - Numeric value: 0.75, representing 75% GPU KV-cache usage for this model

- When multiple models are loaded, each gets its own metric line with a unique label set. This allows Prometheus to:
- 1. Track GPU cache usage separately for each model
- 2. Store historical usage data
- 3. Enable alert creation for high usage thresholds
- 4. Power dashboard visualizations showing cache usage trends

Counter Metrics

- # HELP request_success_total Count of successful requests.
- # TYPE request_success_total counter
- request_success_total{model_name="meta/llama-3.1-8b-instruct"} 501.0
- It only increases over time (never decreases)
- It tracks cumulative totals
- It can reset to zero if the process restarts
- It counts how many times a specific event has occurred

- This tells us:
- 1. The model "meta/llama-3.1-8b-instruct" has successfully handled 517 requests
- 2. The model name is specified in curly braces as a label
- 3. Each model gets its own counter, just like with the `gpu_cache_usage_perc` metric

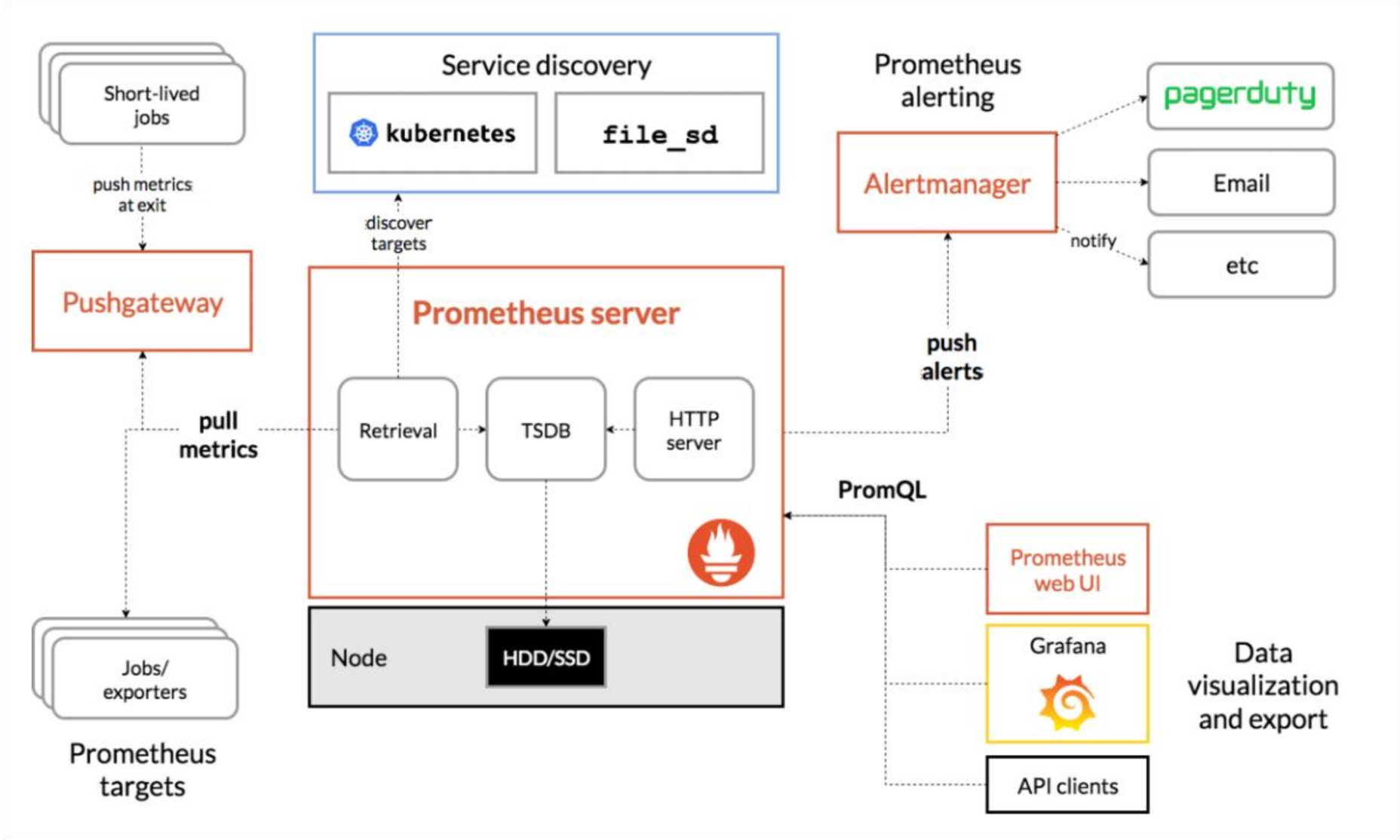
Histogram Metric

- # HELP e2e_request_latency_seconds Histogram of end to end request latency in seconds.
- # TYPE e2e_request_latency_seconds histogram
- e2e_request_latency_seconds_bucket{le="1.0",model_name="meta/llama-3.1-8b-instruct"} 1.0
- e2e_request_latency_seconds_bucket{le="2.5",model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_bucket{le="5.0",model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_bucket{le="10.0",model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_bucket{le="15.0",model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_bucket{le="20.0",model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_bucket{le="30.0",model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_bucket{le="40.0",model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_bucket{le="50.0",model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_bucket{le="60.0",model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_bucket{le="+Inf",model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_count{model_name="meta/llama-3.1-8b-instruct"} 501.0
- e2e_request_latency_seconds_sum{model_name="meta/llama-3.1-8b-instruct"} 525.8991451263428

- A histogram metric divides data into "buckets" that count observations falling within specific ranges. Each bucket has a label `le` (less than or equal to) that defines its upper boundary.
- Prometheus uses cumulative buckets, meaning each bucket includes the count from all lower buckets. The final bucket `le="+Inf"` shows the total number of requests.

Monitoring the NIM Containers

Prometheus

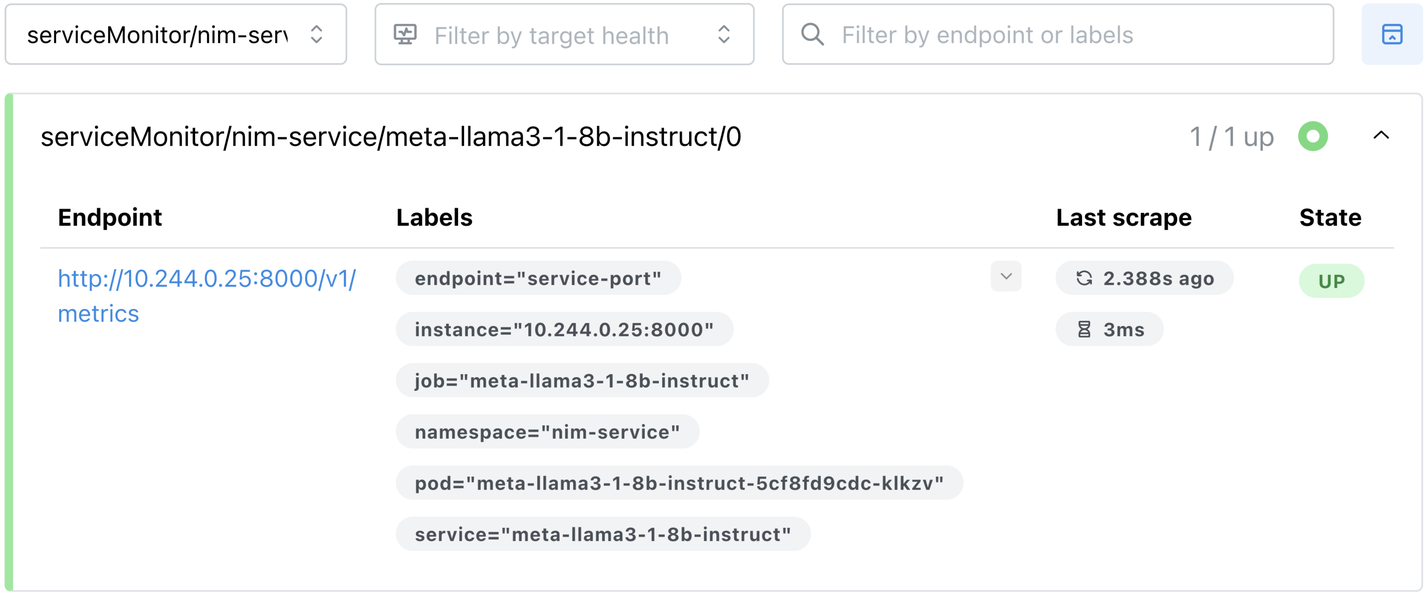


Prometheus

Obtaining NIM Metrics

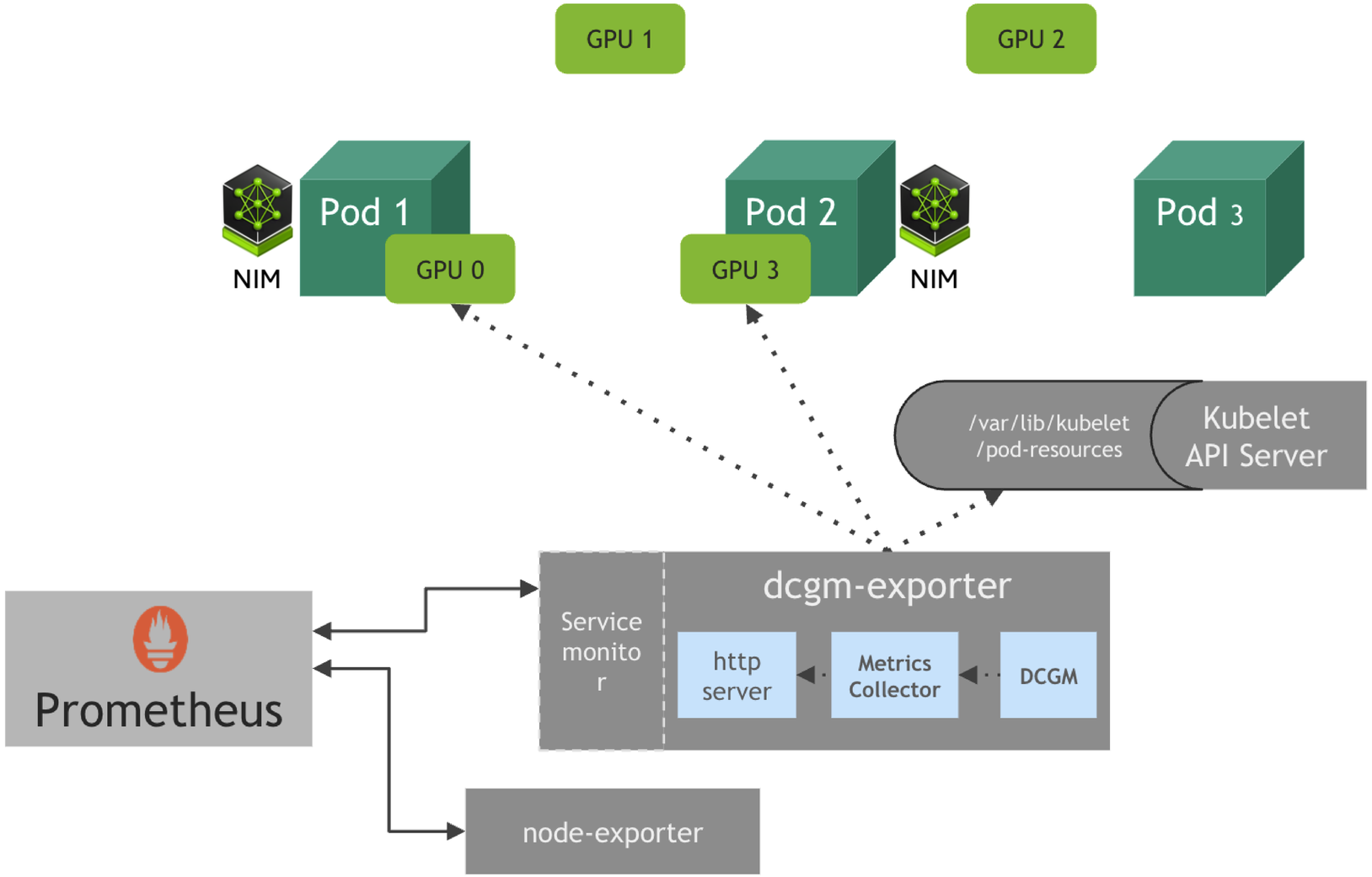
Service Monitor

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: meta-llama3-1-8b-instruct
  namespace: nim-service
  labels:
    release: prometheus
    app.kubernetes.io/instance: meta-llama3-1-8b-instruct
    app.kubernetes.io/managed-by: k8s-nim-operator
    app.kubernetes.io/name: meta-llama3-1-8b-instruct
    app.kubernetes.io/part-of: nim-service
spec:
  endpoints:
  - port: service-port
    interval: 30s
    path: /v1/metrics
    scheme: http
  namespaceSelector:
    matchNames:
    - nim-service
  selector:
    matchLabels:
      app.kubernetes.io/instance: meta-llama3-1-8b-instruct
      app.kubernetes.io/managed-by: k8s-nim-operator
      app.kubernetes.io/name: meta-llama3-1-8b-instruct
      app.kubernetes.io/part-of: nim-service
```



Obtaining GPU Metrics

DCGM Exporter



NVIDIA DCGM

Visualizing Metrics with Grafana

Grafana Dashboard

Objectives of this notebook

1. Understanding NIM Metrics

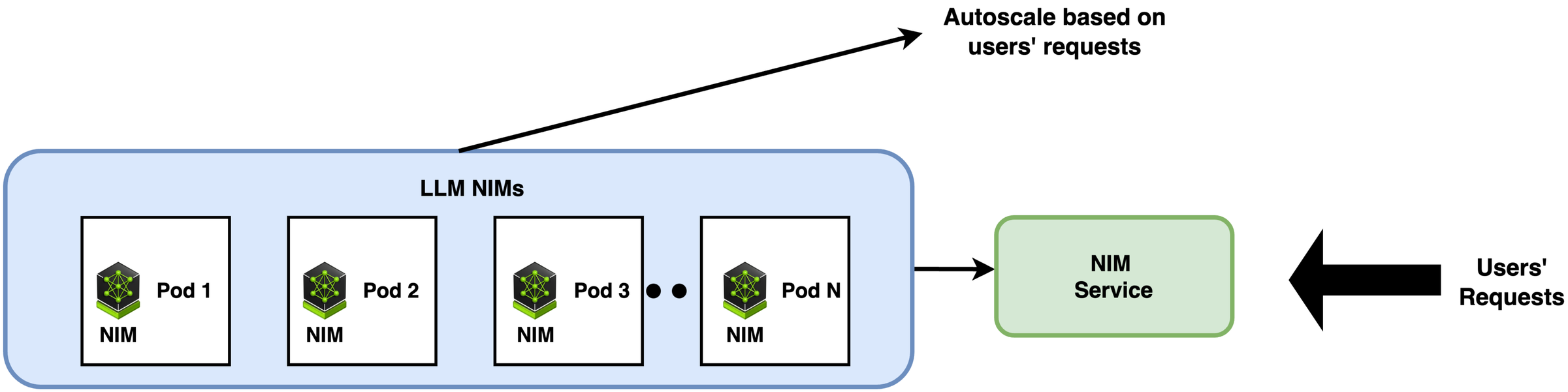
2. Enabling NIM Metrics in Prometheus

3. Visualizing the Collected Metrics in Prometheus and Grafana

Notebook 4: Autoscaling with custom Metrics

Autoscaling NIMs

Autoscaling the NIMs based on the User Workload



Autoscaling NIMs

LLM NIM Metrics

Category	Metric	Metric Name	Description	Granularity	Frequency
KV Cache	GPU Cache Usage	`gpu_cache_usage_perc`	GPU KV-cache usage. 1 means 100 percent usage	Per model	Per iteration
Count	Running Count	`num_requests_running`	Number of requests currently running on GPU	Per model	Per iteration
	Waiting Count	`num_requests_waiting`	Number of requests waiting to be processed	Per model	Per iteration
	Max Request Count	`num_request_max`	Max number of concurrently running requests	Per model	Per iteration
	Total Prompt Token Count	`prompt_tokens_total`	Number of prefill tokens processed	Per model	Per iteration
	Total Generation Token Count	`generation_tokens_total`	Number of generation tokens processed	Per model	Per iteration
Latency	Time to First Token	`time_to_first_token_seconds`	Histogram of time to first token in seconds	Per model	Per request
	Time per Output Token	`time_per_output_token_seconds`	Histogram of time per output token in seconds	Per model	Per request
	End to End	`e2e_request_latency_seconds`	Histogram of end to end request latency in seconds	Per model	Per request
Count	Prompt Token Count	`request_prompt_tokens`	Histogram of number of prefill tokens processed	Per model	Per request
	Generation Token Count	`request_generation_tokens`	Histogram of number of generation tokens processed	Per model	Per request
	Finished Request Count	`request_success_total`	Number of finished requests, with label indicating finish reason	Per model	Per request

Autoscaling NIMs

Custom Metrics

custom-metrics-autoscaling /etc/prometheus/rules/prometheus-prometheus-kube-prometheus-prometheus-rulefiles-0/prometheus-nim-operator-as-rules-8558d9c3-74bd-4f96-8d7c-a202ace59724.yaml

last run 3.233s ago | took 1ms | every 30s

nemollm_gpu_util_avg — 3.333s ago | 0ms | OK

```
avg by (kubernetes_node, pod, namespace, gpu) (DCGM_FI_DEV_GPU_UTIL{pod=~"meta-llama3-.*"})
```

nemollm_gpu_power_avg — 3.333s ago | 0ms | OK

```
avg by (kubernetes_node, pod, namespace, gpu) (DCGM_FI_DEV_POWER_USAGE{pod=~"meta-llama3-.*"})
```

average_latency_per_request — 3.333s ago | 0ms | OK

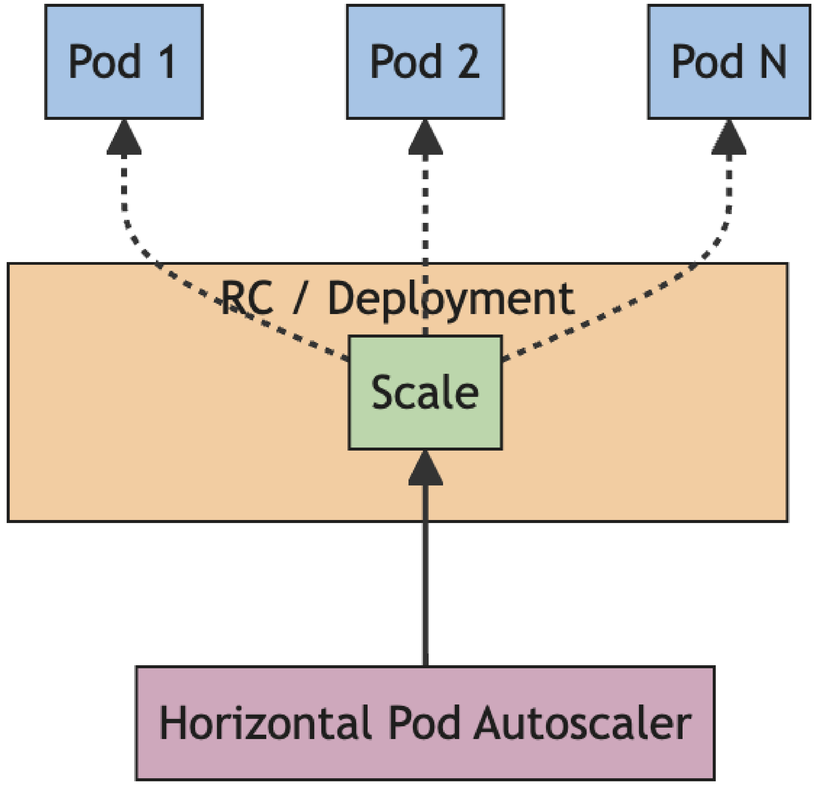
```
sum by (kubernetes_node, pod, namespace) (increase(e2e_request_latency_seconds_sum{pod=~"meta-llama3-.*"}[1m])) / sum by (kubernetes_node, pod, namespace)
(increase(request_finish_total{finished_reason=~"stop|length",pod=~"meta-llama3-.*"}[1m]))
```

inter_token_latency_p95 — 3.332s ago | 0ms | OK

```
histogram_quantile(0.95, sum by (kubernetes_node, pod, namespace, le) (rate(time_per_output_token_seconds_bucket{pod=~"meta-llama3-.*"}[1m])))
```

Autoscaling NIMs

What is Horizontal Pod Autoscaler (HPA)



Horizontal Pod Autoscaling | Kubernetes

Autoscaling NIMs

HorizontalPodAutoscaler (HPA) - Custom Prometheus Rules for NIMs

Scaling Specification in NIMService

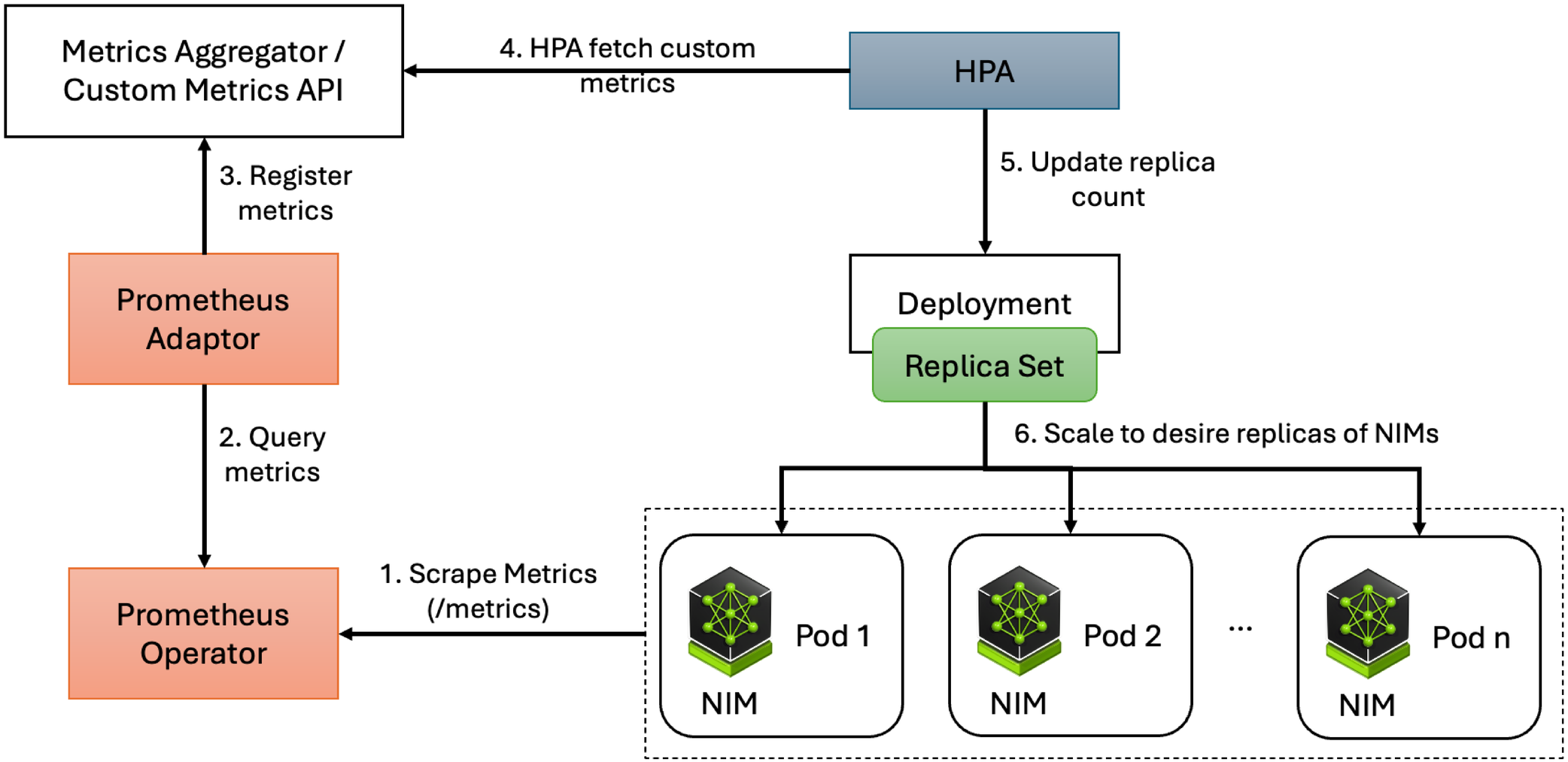
```yaml
scale:
  enabled: true
  hpa:
    maxReplicas: 2 # Increase to scale Pods up to a higher limit
    minReplicas: 1
    metrics:
      - type: Pods
        pods:
          metric:
            name: nemollm_gpu_util_avg # Custom metric for GPU usage
          target:
            type: AverageValue
            averageValue: 50
    behavior:
      scaleDown:
        stabilizationWindowSeconds: 300
        policies:
          - type: Pods
            value: 1
            periodSeconds: 300
```

Metrics	Expressions
nemollm_gpu_util_avg	*avg by (kubernetes_node, pod, namespace, gpu) (DCGM_FI_DEV_GPU_UTIL{pod=~"meta-llama3-.*"})*
nemollm_gpu_power_avg	*avg by (kubernetes_node, pod, namespace, gpu) (DCGM_FI_DEV_POWER_USAGE{pod=~"meta-llama3-.*"})*

Horizontal Pod Autoscaling | Kubernetes

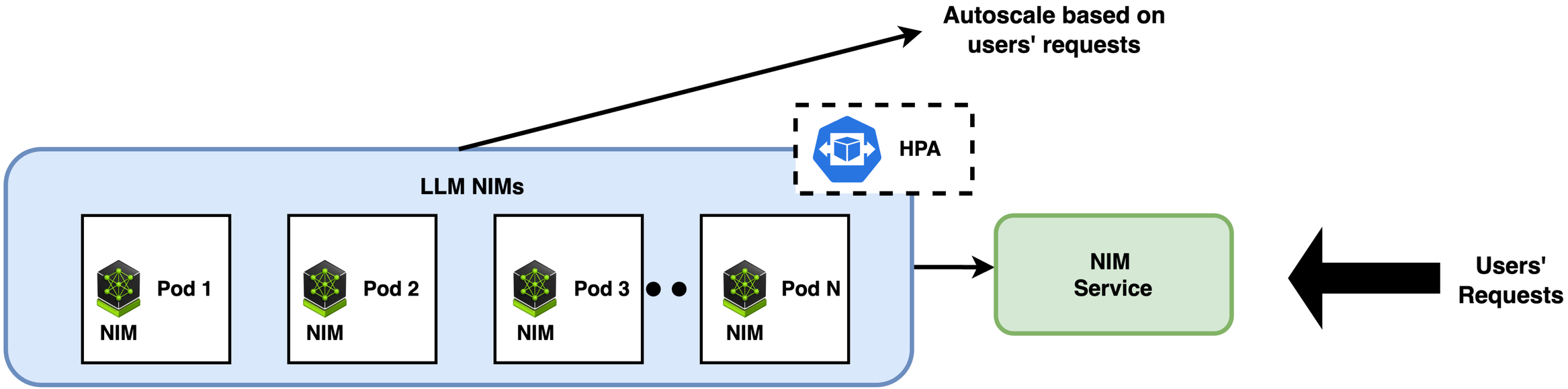
Autoscaling NIMs

How HPA works with custom metrics?



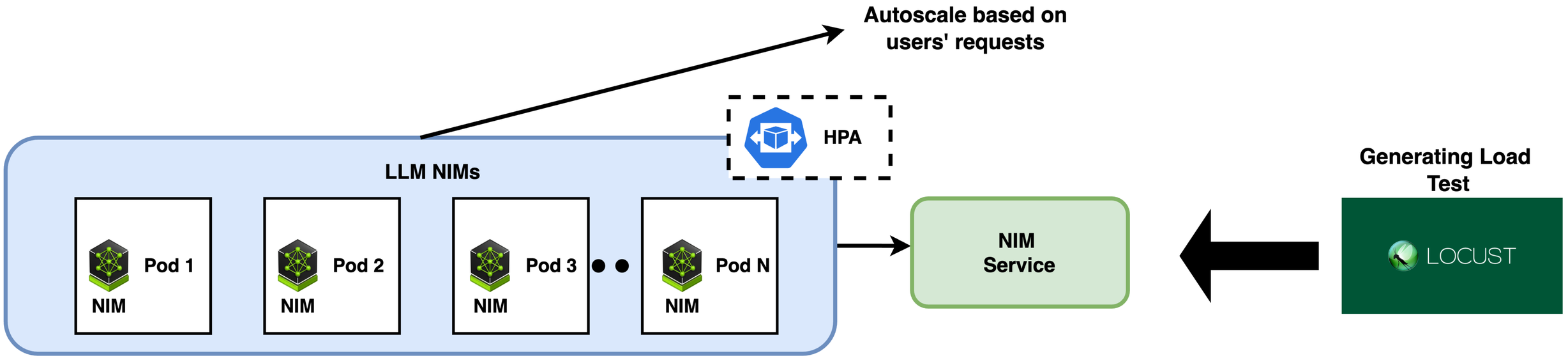
Autoscaling NIMs

Adding HPA to the Deployment



Autoscaling NIMs

Workload generation using Locust



locust

Configuring Locust

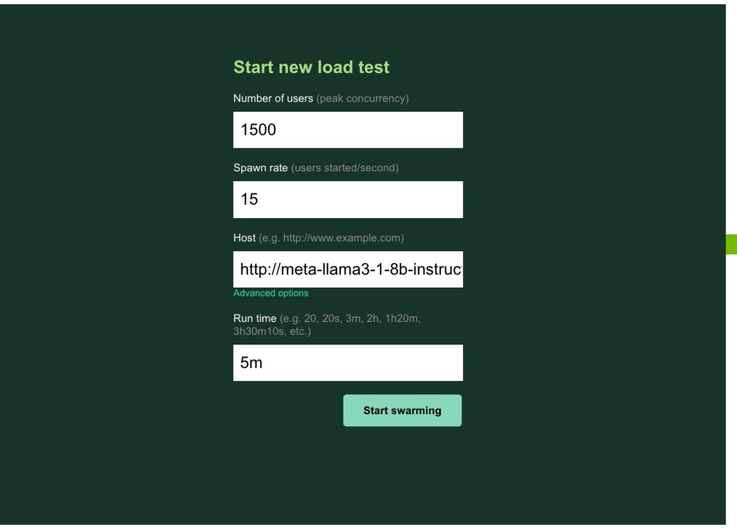
Workload generation using Locust

```
from locust import HttpUser, TaskSet, task, constant_throughput

class UserTasks(TaskSet):
    @task
    def index(self):
        data = {
            "messages": [
                {
                    "content": "You are a polite, respectful, and professional AI assistant dedicated to helping p
traveler, family, group of friends, or on a business trip, you tailor your suggestions to their unique preferences
a courteous tone in every interaction, and deliver information that is both relevant and accurate. Remember that y
interests in mind. ",
                    "role": "system",
                },
                {
                    "content": "What should I do for a 4 day vacation at Cape Hatteras National Seashore?",
                    "role": "user"
                },
            ],
            "model": "meta/llama-3.1-8b-instruct",
            "top_p": 1,
            "n": 1,
            "max_tokens": 50,
            "stream": False,
            "frequency_penalty": 0.0,
            "stop": ["STOP"]
        }
        headers = {
            "Accept": "application/json",
            "Content-Type": "application/json"
        }
        response = self.client.post("/v1/chat/completions", json=data, headers=headers)
        print(response.text)

class WebsiteUser(HttpUser):
    tasks = [UserTasks]
    # we assume that our latency is lower than 100 seconds
    # constant_throughput(0.01) makes sure that each user submits a throughput of 0.01 requests per second
    # to do this it starts a task every 1/0.01 = 100 seconds
    wait_time = constant_throughput(0.01)
```

locust

Autoscaling NIMs

Workload generation using Locust



RPS = **#users*0.01== Spawn Rate**

locust

Objectives of this notebook

1. Create custom Prometheus recording rules
2. Deploy Prometheus-adapter
3. Configure HPA
4. Load Testing