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

Geralyn Chong

March 17, 2025

Contents

1 Challenges of LLM Inference in Production

Tokens are units of text that the model processes which generally correspond to 4 characters of text. Time to First Token (TTFT) is the time is takes for the model to generate the first token after recieving a requiest. End-to-end Latency.

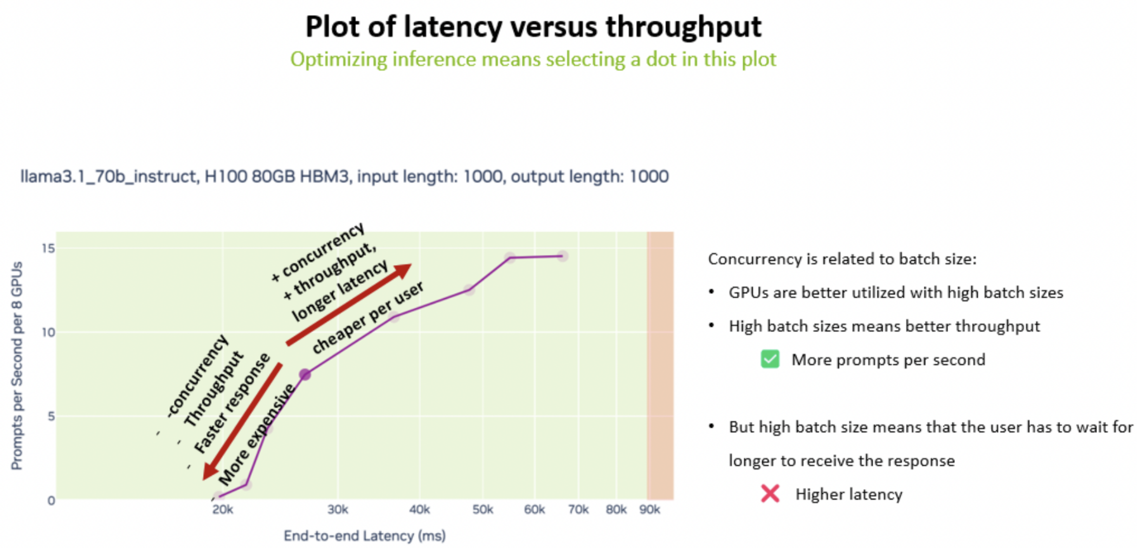
Inference Throughput: concept of a batch where there are multiple sequences that come from different clients. Number of tokens per unit of time.

- Reuqest per second per deployment (RPS)
- Tokens per second per deployment (TPS)
- Requests per second per GPU

Concurrency is NOT throughput: How many tokens are being processed at the same time.

- Low Concurrency:Results in lower latency due to smaller batch sizes for GPU processing. However, leads to suboptimal throughput and higher cost per token. GPU resources are underutilized
- High Concurrency: Improves overall throughput and reduces cost per token. Makes better use of GPU capacity through increased parallel processing. May increase latency due to larger batch sizes

Optimizing for Latency or Throughput: Online - live generation where you interact with them **live**. Here latency is really important becuase it matters how quickly they will get hteir repsonse. Offline - postponed computation refering to the simplest execution of the model where we want to maximize GPU utilization.

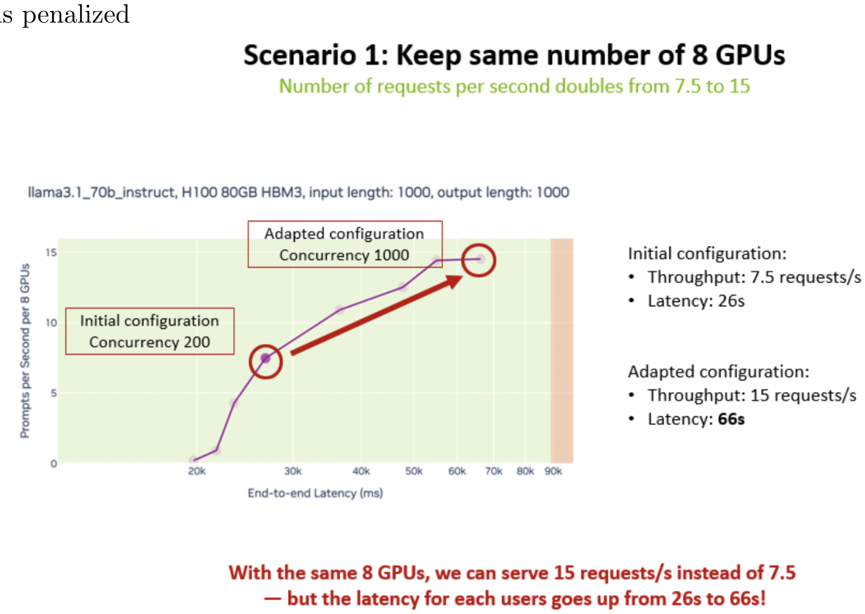


1.1 Variables impacting latency vs throughput plot

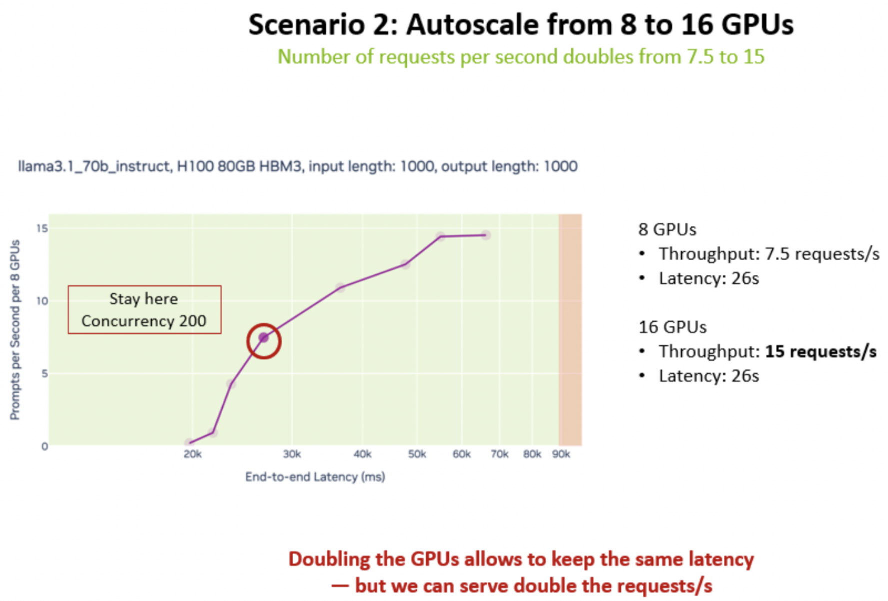
- Context window (Input length)
- Size of the model used
- Latency requirement that bounds performance
- Size of tokens needed to be produced

Autoscaling Example: What is autoscaling? When there are peaks in the number of requests to our server, how can we manipulate our resources to best support these requests? "How can we keep a similar SLA of latency when the request/s increase?

1. Keep same number of 8 GPUs but increase the concurrency: Throughout increases and latency is penalized



2. Keep concurrency but autoscale to more GPUs: allow to keep same latency for users but autoscaled pods scale throughput



2 NIMs

Ease of deployment of LLMs for production and have full control over your own model. Optimized Inference Microservices that have accelerated runtime for gen AI.

- Portable
- Easy to use
- Enterprise Supported
- Performance

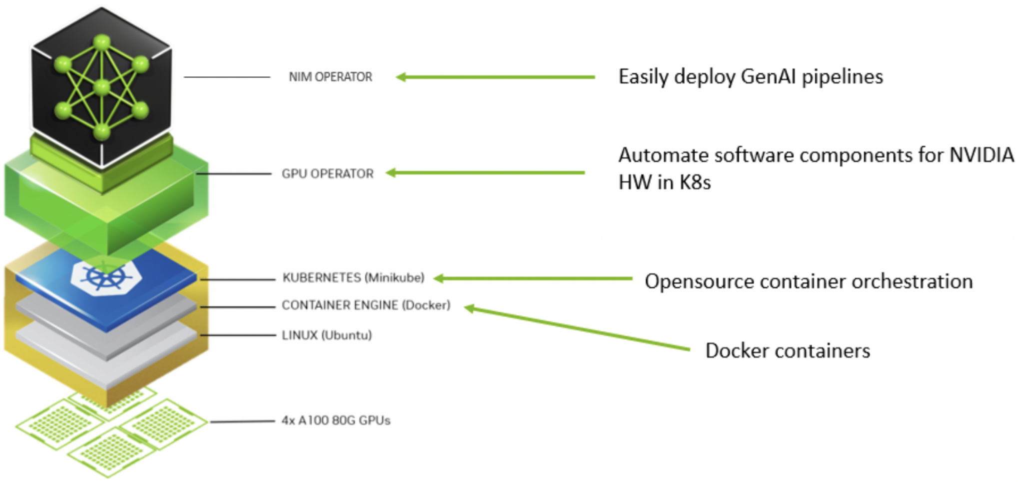
Applications to many domains.

Internals of NIM:

1. Allows for multiple types of backends
2. LLM Executor
3. FastAPI

3 Lab 1: NIM Operator for K8s

Lab environment:



text

The NIM operator introduces two essential custom resources that work together to deploy AI models:

1. NIM Cache: Downloads models from NVIDIA NGC - GPU Cloud and storing them persistently on network storage for future use
2. NIMService: his resource takes a model from an existing NIMCache and deploys it as a microservice, making it available for inference requests. *What are inference requests?* Running live data that the model has not seen before in order to produce predictions or complete tasks.

4 Benchmarking NIM

Simulate a certain load of client requests and measure from the client-side the time to get response to calculate latency and trhoughput. Focusing on GenAI-Perf to compute metrics of latency, throughput, and concurrency with ease. Locust is also another tool for measuring these stats.

This tool allows us to compare these measurement of statistics across different LLM providers. Docs.

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

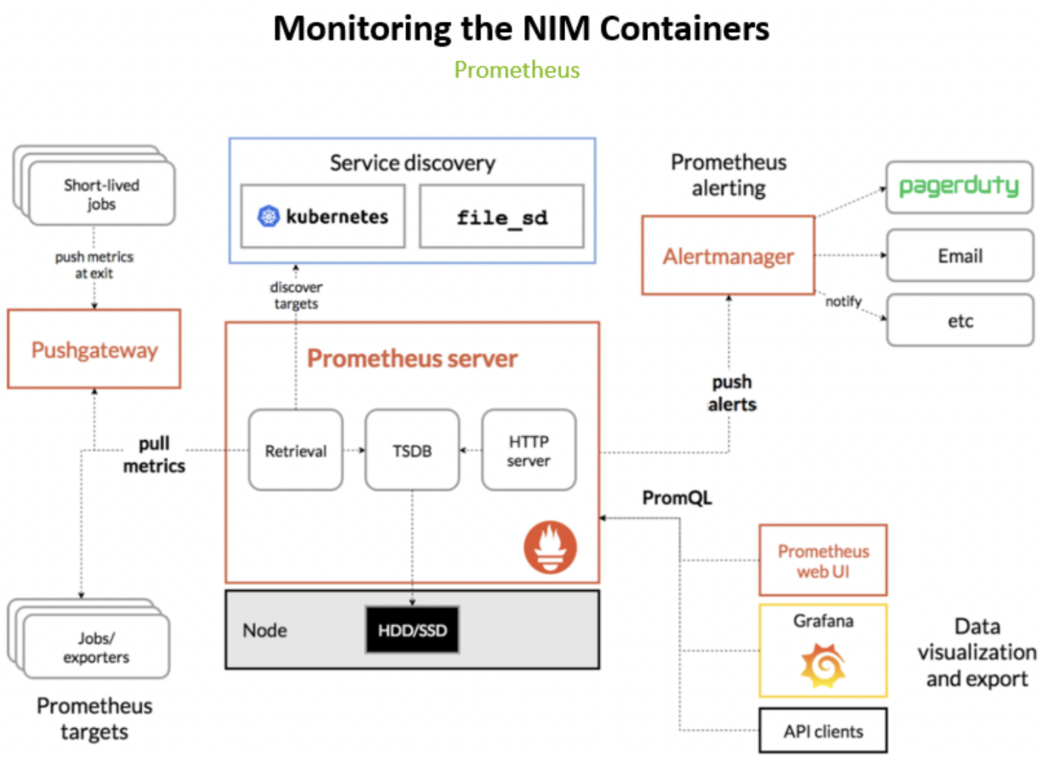
Output token throughput (per sec): 995.61
Request throughput (per sec): 5.41

5 Observability

Promethus is an open-source tool that allows us to monitor and alert on the resources used of our system. Gauge Metrics represent the current value of the fraction of the GPU memory devoted for the KV-cahce that is being utilized by our model. Counter Metrics cannot decrease. Histogram Metrics divide data into "buckets" that count observation falling within specific ranges. Each bucket has le that defines its upper bound.

- le: less or equal to the value
- le='+inf': shows the total number of requests because all latency will be less than ∞.

Monitoring the NIM containers:



Obtaining GPU Metrics via DCGM Exporter can be done as well.

NVIDIA NIM provides monitoring of NIM Service level metrics and NIM Operator metrics. Service level metrics are taken from service pods focusing of the model's performance and resource utilization. Operator metrics are collected from Operator pods and track the number of instances in various states. The following code snippets are taken from the lab:

- Grabbing the Gauge metrics ($1 = 100\% of GPU$ used)

  ```
  !curl -Ns -X "GET" "{NIM_ENDPOINT}/v1/metrics" | grep "gpu_cache_usage_perc"
  ```

  1. A help message: explains what the message means
  2. Type declaration of the metric: `gauge`
  3. The actual measurement data

  When multiple models are loaded each model gets its own metric with unique label sets.

6 Autoscaling with custom metrics

Autoscaling based on the user workload.

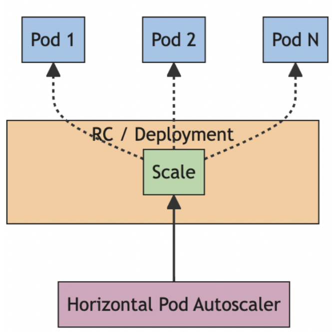
Autoscaling NIMs

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

Types of Autoscaling in Kubernetes: Focusing on HPA (Horizontal Pod Autoscaler) which increased replica count based on provided metrics. Scaling specification in NIMService: need to specifyy the custom metric for GPU usage as well as the average value across the pods used.

Get HPA controller which is responsible for Autoscaler in each container. Here we then use the generating load test using Locust. Using a constant throughput function, we ensure that users send a reuqest every x seconds. In locust, we ensure that each user generates as many request as possible but to some configured extent. $RPS = \text{num of users} * 0.01 = \text{Spawn rate}$.



There are other methods of autoscaling as well such as the Vertical Pod Autoscaler (VPA) adjusting the resource requests and limits of individual pods to match actual usage. VPA is beneficial for applications with predictable and stable workloads, where resource requirements may vary over time. Cluster Autoscalers automatically adjust the size of the node pool in a cluster.

For example, when there are insufficient nodes, CA provides more nodes and underutilized nodes are removed. By working with cloud provider APIs, it is able to scale the infrastructure.