AMD EPYC

USER GUIDE
AMD EPYC 9004

LLM-In-a-Box Solutions for Nutanix™ AOS/AHV Solutions Build Guide

Publication	58736
Revision	1.0
Issue Date	August, 2024

Date	Version	Changes
August, 2024	1.0	Initial public release

Table of Contents

This page intentionally left blank.

Chapter 1 Introduction & Requirements

This document guides you through the procedures involved in configuring and setting up a Nutanix cluster with a focus on deploying LLama2 models and performance benchmarking of multi-user requests that specifically evaluate Large Language Models (LLMs), such as llama-2-7b-int8 and its variant models. It contains detailed, step-by-step instructions for establishing an NKE multi-node cluster environment, deploying LLama2, and conducting performance benchmarking. It also addresses the deployment process on both Nutanix VMs and cluster and specifies the minimum hardware and software configurations necessary for this setup.

LLM-in-a-Box is a turnkey AI solution for organizations wanting to implement GPT capabilities while maintaining control of their data and applications. It includes everything needed to build AI-ready infrastructure, including:

- Nutanix™ Cloud Platform infrastructure.
- Nutanix Files and Object storage for running and fine-tuning GPT models.
- Open-source software to deploy and run AI workloads, including PyTorch, and Kubeflow.
- Support for a curated set of LLMs, including Llama2, Falcon, and MPT.

1.1 Technologies

4th Gen AMD EPYC™ processors are ideal for AI workloads because of their high core counts and strong performance. Nutanix Kubernetes Engine (NKE) simplifies deploying and managing machine learning pipelines, and Kserve optimizes model serving for fast predictions. The Llama2 LLM brings advanced natural language processing capabilities, and Nutanix AOS allows centralized management and monitoring of the entire AI infrastructure. This combination unlocks transformative AI experiences on your existing infrastructure.

1.2 AMD EPYC 9004 Series Processors

AMD EPYC 9004 Series Processors continue to redefine the standards for modern datacenters. 4th Gen AMD EPYC processors are built on the innovative x86 architecture and "Zen 4" core. 4th Gen AMD EPYC processors deliver efficient, optimized performance by combining high frequencies, the largest-available L3 cache, up to 128 (1P) or up to 160 (2P) lanes of PCIe® Gen 5 I/O, synchronized fabric and memory clock speeds, and support for up to 6 TB of DDR5-4800 memory. Built-in security features, such as AMD Infinity Fabric™ technology, Secure Memory Encryption (SME), and Secure Encrypted Virtualization (SEVSNP) help protect data while it is in use. AMD Infinity Guard features vary by EPYC™ Processor generations and/or series. (Infinity Guard security features must be enabled by server OEMs and/or Cloud Service Providers to operate. Check with your OEM or provider to confirm support of these features. Learn more about Infinity Guard at http://www.amd.com/en/products/processors/server/epyc/infinity-guard.html. GD-183A.)

1.3 Important Reading

Please be sure to read the following guides (available from the AMD Documentation Hub), which contain important foundational information about 5th Gen AMD EPYC processors:

- *AMD EPYC™ 9005 Processor Architecture Overview*
- *BIOS & Workload Tuning Guide for AMD EPYC™ 9005 Series Processors*

1.4 Requirements

The following requirements must be met in order to deploy LLM-in-a-Box solutions on Nutanix AOS/AHV:

- Nutanix cluster with AOS/AHV.
- A Prism Central installed on the same cluster (required for Nutanix Kubernetes Engine [NKE]).
- A Kubernetes cluster initialized using NKE, where each worker node is configured with 76 vCPUs, 128 GB memory, and 1 vNUMA node.

1.5 Caveats

Obtaining maximum performance requires fine tuning the software packages described in this user guide. Some of the procedures described herein may not be standard for a typical production environment. This specifically applies to the following operations that use the foundational Linux supports instead of Nutanix AOS management facilities:

- The Kuberenets worker VMs must be manually configured to use host CPU passthrough.
- The Kubernetes worker VMs must be manually pinned to the desired CPU cores.
- The Kubernetes worker VMs must be manually pinned to the desired NUMA node.

Chapter 2

Host BIOS Settings

Table 2-1 lists the recommended host BIOS settings for LLM-in-a-Box on a Nutanix cluster. You must power cycle the host after modifying BIOS settings.

Name	Recommended Setting	Description
Global C-State Control	Auto	• **Enabled/Auto:** Controls IO based C-state generation and DF C-states, including core processor C-States • **Disabled:** AMD strongly recommends not disabling this option because this also disables core processor C-States.
DF C-States	Disabled	Controls DF C-states. • **Disabled:** Prevents the AMD Infinity Fabric from entering a low-power state. • **Enabled/Auto:** Allows the AMD Infinity Fabric to enter a low-power state.
Power Profile Selection	Auto	• **Auto/0:** High-performance mode • **1:** Efficiency mode • **2:** Maximum I/O performance mode
Core Performance Boost	Auto	• **Enabled/Auto:** Enables Core Performance Boost. • **Disabled:** Disables Core Performance Boost.
Determinism Control	Manual	• **Auto:** Use default performance determinism settings. • **Manual:** Specify custom performance determinism settings.
Determinism Enable	Auto	• **Auto/0:** Power. • **1:** Performance.

Table 2-1: Recommended host BIOS settings for deploying LLM-in-a-Box on a Nutanix cluster

Nodes Per Socket (NPS)	2	**Memory Interleaving:** The **NPS** setting always determines the memory interleaving regardless of whether **LLC as NUMA** is **Enabled** or **Disabled**. # of NUMA nodes (if **LLC as NUMA Domain** is **Disabled**): • **NPS1:** One NUMA node per socket (Most cloud providers use this as it provides consistent average memory latency to all the accesses within a socket). • **NPS2:** Two NUMA nodes per socket. • **NPS4:** Four NUMA nodes per socket • **NPS0 (not recommended):** Only applicable for dual-socket systems. A single NUMA node is created for the whole two-socket platform. AMD recommends either NPS1 or NPS4 depending on your use case. **Windows systems:** Make sure that the number of logical processors per NUMA node is <=64. You can do this by using NPS2 or NPS4 instead of the default NPS1.
TSME	Auto	• **Auto/Disabled:** Disables transparent secure memory encryption. • **Enabled:** Enables transparent secure memory encryption.
SEV Control	Disabled	In a multi-tenant environment (such as a cloud), Secure Encrypted Virtualization (SEV) mode isolates virtual machines from each other and from the hypervisor. • **Disabled:** SEV is disabled. • **Enabled:** SEV is enabled. If you disable and then reenable SEV, then you will need to power cycle your system after changing this setting back to **Enabled**.
SEV-ES	Disabled	Secure Encrypted Virtualization-Encrypted State (SEV-ES) mode extends SEV protection to the contents of the CPU registers by encrypting them when a virtual machine stops running. Combining SEV and SEV-ES can reduce the attack surface of a VM by helping protect the confidentiality of data in memory. • **Disabled:** SEV-ES is disabled. • **Enabled:** SEV-ES is enabled.

Table 2-1: Recommended host BIOS settings for deploying LLM-in-a-Box on a Nutanix cluster (Continued)

SEV-SNP Support	Disabled	Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) mode builds on SEV and SEV-ES by adding strong memory integrity protection to create an isolated execution environment that helps prevent malicious hypervisor-based attacks such as data replay and memory re-mapping. SEV-SNP also introduces several additional optional security enhancements that support additional VM use models, offer stronger protection around interrupt behavior, and increase protection against recently-disclosed side channel attacks. • **Disabled:** SEV-SNP is disabled. • **Enabled:** SEV-SNP is enabled.

Table 2-1: Recommended host BIOS settings for deploying LLM-in-a-Box on a Nutanix cluster (Continued)

This page intentionally left blank.

Chapter 3 Performance Optimization

The GPT stack is deployed in a pod running within a K8s node. NKE runs each K8s node in a VM. Obtaining maximum performance requires configuring the K8s VM node to utilize the AMD EPYC processor AVX features and to align with AMD EPYC processor NUMA topology. This chapter refers to both single and multiple K8s node VM(s) as `VMk8s-node`.

3.1 AVX Configuration

As mentioned above, all `VMk8s-node` must be configured to use the AMD EPYC AVX2 and AVX512 instruction sets. By default, the Nutanix AHV does not support 4th Gen AMD EPYC processor, but you can work around this by configuring the hypervisor to passthrough the host CPU type to `VMk8s-node` via the `cpu_passthrough` AHV VM attribute.

You should also make sure that `VMk8s-node` is not running two AVX threads on the same CPU core simultaneously because this will cause AVX resource contention on the CPU. You can avoid this by configuring `VMk8s-node` to run with a single thread using the `num_threads_per_core` AHV VM attribute.

To enable `cpu_passthrough` and `num_threads_per_core` for a `VMk8s-node`:

1. Login to any of the CVMs in the cluster as the `nutanix` user.
2. Obtain the list of all VMs running on the cluster.
   ```
    acli vm.list
   ```
3. Power off the `VMk8s-node`.
   ```
   acli vm.off <vm name>
   ```
4. Set the VM attributes,
   ```
   acli vm.update <vm_name> cpu_passthrough=”true” num_threads_per_core=”1”
   ```
5. Power on the `VMk8s-node`
   ```
   acli vm.on <vm name>
   ```
6. Repeat Steps 3-5 for all `VMk8s-node` running the GPT stack.

3.2 EPYC NUMA Alignment

The `VMk8s-node` must be configured to run within an AMD EPYC processor NUMA node. To do this:

1. On Prism Central, migrate the NKE related VMs that are not `VMk8s-node` to one of the AHV hosts in the cluster.
2. Login to one of the CVMs, then power off all the `VMk8s-node` that run the GPT stack.
   a. Obtain the list of all VMs running on the cluster.
   ```
    acli vm.list
   ```
   b. Power off the `VMk8s-node`.
   ```
   acli vm.off <vm name>
   ```
3. Execute the following commands to set `VMk8s-node` to physical NUMA node affinity and to set the VM to host affinity.

```
acli vm.update <vm name> extra_flags=”numa_pinning=1”
acli vm.affinity_set <vm name> host_list=<host id>
```

Note: Each `VMk8s-node` *should have a distinct host affinity. Avoid sharing an AHV host with more than one* `VMk8s-node`.

4. Execute the following command to power up each `VMk8s-node`.
   acli vm.on <vm name>

Follow Steps a-b on each AHV host that runs a `VMk8s-node`.

   a. Execute the following command to ensure the `VMk8s-node` is running on NUMA node #1. If not, then power off the VM and then power it on again; repeat this as necessary until the VM is running on NUMA node #1.

Note: The `numa_pinning` *setting does not set the NUMA affinity 100% of the time.*

```
numastat -p qemu-kvm
```

The output appears as shown below. Execute the ps command to find out which PID corresponds to `VMk8s-node`. The memory usage should appear in the `Node 1` column.

```
Per-node process memory usage (in MBs)
PID                           Node 0          Node 1           Total
------------------   --------------- --------------- ---------------
14143 (qemu-kvm)            49197.79            4.78        49202.57
1467714 (qemu-kvm)             37.60        49165.80        49203.40
------------------   --------------- --------------- ---------------
Total                       49235.39        49170.58        98405.96
```

   b. Execute the `acli` command on the CVM to find out the UUID of the `VMk8s-node`.
   ```
   ssh nutanix@<cvm-ip> /usr/local/nutanix/bin/acli vm.list | fgrep worker
   ```

For example:

```
# ssh nutanix@192.168.5.254 /usr/local/nutanix/bin/acli vm.list | fgrep worker-0
Nutanix Controller VM
karbon-nke-titanite-26e2b2-worker-0 4cc166c0-5f24-42c7-8c62-b0ae291506d2
```

c. Execute the following command to pin the vCPUs of `VMk8s-node` to a fix set of cores.

```
virsh vcpupin <vm uuid> <vcpu #> <core #>
```

The following example assume the `VMk8s-node` has 76 vCPUs, and the host CPU has 96 cores. The cores on the NUMA node #1 correspond to vCPU cores 96 to 191 on a 96-core 4th Gen AMD EPYC processor. It is good practice to pin the `VMk8s-node` starting at core number=<high core number>-<number vCPUs>+1. This example uses have 191–76+1=116.

```
virsh vcpupin 4cc166c0-5f24-42c7-8c62-b0ae291506d2 0 116
virsh vcpupin 4cc166c0-5f24-42c7-8c62-b0ae291506d2 1 117
virsh vcpupin 4cc166c0-5f24-42c7-8c62-b0ae291506d2 2 118
. . .
virsh vcpupin 4cc166c0-5f24-42c7-8c62-b0ae291506d2 74 190
virsh vcpupin 4cc166c0-5f24-42c7-8c62-b0ae291506d2 75 191
```

This page intentionally left blank.

Chapter 4 Deploying Quantized Llama2 on Nutanix Cluster (TorchServe kind)

This chapter describes how to deploy the Hugging Face Llama2-chat-hf int8 model on NKE.

4.1 Build a Llama2 TorchServe Model Archive

4.1.1 Step 1: Setup PVC and Create a Temporary Pod

1. Create a PVC by executing the following commands in the terminal:

```
$ kubectl apply -f pvc.yaml

## pvc.yaml ##

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: model-store-claim
spec:
  resources:
    requests:
      storage: 80Gi
  accessModes:
    - ReadWriteOnce
```

2. Execute the following commands to mount the PVC named model-store-pod on temp pod:

```
$ kubectl apply -f temp_pod.yaml
## temp_pod.yaml ##
apiVersion: v1
kind: Pod
metadata:
  name: model-store-pod
spec:
  volumes:
    - name: model-store
      persistentVolumeClaim:
        claimName: model-store-claim
  containers:
    - name: model-store
      image: ubuntu
      command: [ "sleep" ]
      args: [ "infinity" ]
      volumeMounts:
        - mountPath: "/mnt"
          name: model-store
      resources:
        limits:
```

```
        cpu: 2000m
        memory: 5Gi
```

4.1.2 Step 2: Install the tools and model

1. Install dependencies and make a symbolic link inside the temp pod.
   ```
   $ apt-get update && apt-get install git python3 pip
   $ ln -s /usr/bin/python3 /usr/bin/python
   ```

2. Clone the Torchserve* repository.

3. Execute the following commands:
   ```
   $ cd serve
   $ python3 ./ts_scripts/install_dependencies.py
   $ pip install torchserve torch-model-archiver $ torch-workflow-archiver
   ```

4. Download the model.
   ```
   $ cd examples/large_models/Huggingface_accelerate
   $ apt-get install libopenmpi-dev
   ```

5. Modify requirements.txt by adding the following lines:
   ```
   $ llama-cpp-python==0.2.78
   $ pip install -r requirements.txt
   $ pip install -U "huggingface_hub[cli]"
   $ huggingface-cli login
   ```

6. Execute the following command to download the desired model to the `model` folder:
   ```
   $ python3 Download_model.py --model_path <path> --model_name <huggingface-repo>
   ```

   To download a quantized Llama2-7b-chat model:
   ```
   $ python3 Download_model.py --model_path model --model_name arunimad/Llama2-7b-chat-
   int8
   ```

4.1.3 Step 3: Modify Model Configuration File and Create the Model Archive

1. Modify model-config.yaml as follows:

   - Edit `model_name` as desired. For example:
     `llama-cpp-7b`

   - Edit the `model_path` variable to point to the quantized model. For example:
     ```
     model/models--arunimad--Llama2-7b-chat-int8/snapshots/
     4525405b7837e6d0a5fc98e295e9646942d33b4f/ggml-model-q8.gguf
     ```

2. Modify llama_cpp_handler.py by replacing the inference function with the following function:

```
def inference(self, data):
      prompt_template = '''[INST] <<SYS>>You are a helpful chatbot that will answer to the
prompt as needed.<</SYS>>{prompt}[/INST]'''
       result = self.model(prompt_template.format(prompt = data["prompt"]),
max_tokens=data["max_tokens"], top_p=data["top_p"],temperature=data["temperature"],
echo=False)
       tokens = self.model.tokenize(bytes(data["prompt"], "utf-8"))
       return result
```

3. Create the model archive using `torch-model-archiver`:

```
torch-model-archiver --model-name llamacpp-7b --version 1.0 --handler llama_cpp_handler.py
--config-file model-config.yaml --archive-format no-archive --requirements-file
requirements.txt
```

4. Create a `model_store` folder.

5. Copy the archive folder into `model_store`.

6. Copy `model/folder` with the downloaded model into the `archive` folder.

4.1.4 Step 4: Copy Required Files to the model-store-pod

1. Execute into the pod.

2. Create `model-store` and `config` folders by executing the following commands:
   ```
   $ kubectl exec -it model-store-pod – bash
   $ cd mnt
   $ mkdir model-store
   $ mkdir config
   ```

3. Copy the model files into the PVC pod.
   ```
   $ kc cp model_store/llama-cpp-7b model-store-pod:/mnt/model-store/
   ```

4. Add `config.properties` to the `config` folder.

```
$ kc cp config.properties model-store-pod:/mnt/config/

inference_address=http://0.0.0.0:8080
management_address=http://0.0.0.0:8081
metrics_address=http://0.0.0.0:8082
enable_envvars_config=true
install_py_dep_per_model=true
load_models=all
model_store=/home/model-server/model_store
model_snapshot={"name":"startup.cfg","modelCount":1,"models":{"llama-cpp-
7b":{"1.0":{"defaultVersion":true,"marName":"llama-cpp-
7b","minWorkers":1,"maxWorkers":1,"batchSize":1,"maxBatchDelay":200,"responseTimeout":1200
}}}}
```

4.1.5 Step 5: Copy model_store Contents

Copy the `model_store` contents from `serve/examples/LLM/llama/chat_app/model_store` to `/mnt/model_store`.

4.2 Deployment

1. Create `deploy_int8.yaml`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ts-def
  labels:
    app: ts-def
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ts-def
```

```
  template:
    metadata:
      labels:
        app: ts-def
    spec:
      volumes:
        - name: model-store
          persistentVolumeClaim:
            claimName: model-store-claim-int8
      containers:
        - name: torchserve
          image: pytorch/torchserve:latest-cpu
          command: ["/bin/sh", "-c"]
          args: ["torchserve --start --ts-config /home/model-server/config/
config.properties; sleep infinity"]
          env:
            - name: LOG_LOCATION
              value: "/home/model-server/logs"
            - name: METRICS_LOCATION
              value: "/home/model-server/logs"
            - name: TEMP
              value: "/home/model-server/tmp"
            - name: OPENBLAS_NUM_THREADS
              value: "1"
          ports:
            - containerPort: 8080
            - containerPort: 8081
            - containerPort: 8082
          volumeMounts:
            - name: model-store
              mountPath: /home/model-server
          resources:
            limits:
              cpu: 48
              memory: 48Gi
            requests:
              cpu: 48
              memory: 48Gi
          securityContext:
            allowPrivilegeEscalation: false
            runAsUser: 0
```

2. Execute the following command to deploy the Torchserve server:
   ```
   $ kubectl apply -f deploy_int8.yaml
   ```

3. Check the pod logs in the deployment. If there are any errors, then proceed to Step 4, else skip to Step 6.
   ```
   $ kubectl logs pod-name
   ```

4. Create a tmp directory in the torchserve pod.
   ```
   $ kubectl exec -it pod-name - bash
   $ cd /home/model-server
   $ mkdir tmp
   $ Exit pod and delete pod :
   $ kubectl delete pod pod-name
   ```

5. Check the new pod logs to verify that the model loads.

6. Execute the following commands to create a service:

```
$ kubectl apply -f svc.yaml

apiVersion: v1
kind: Service
metadata:
  name: svc
  namespace: int8
spec:
  type: LoadBalancer
  selector:
    app: ts-def
  ports:
  - name: inference
    port: 8080
    targetPort: 8080
  - name: inference2
    port: 8085
    targetPort: 8085
  - name: mgmt
    port: 8081
    targetPort: 8081
  - name: metrics
    port: 8082
    targetPort: 8082
  - name: streamlit01
    port: 8501
    targetPort: 8501
```

You can now send requests using cURL commands. Use the following templates, as appropriate:

- **LLM Endpoint:**
  `http://<routename>/predictions/<model-name>`
- **Sample Endpoint:**

```
curl -v "http://localhost:8080/predictions/llamacpp-7b" -H 'Content-Type: application/
json' -d "@sample.json"
```

- **Sample JSON:**

```
{
  "prompt": "1+1",
  "max_tokens": 128,
  "top_p": 1.0,
  "temperature": 1.0
}
```

This page intentionally left blank.

Chapter 5

Benchmarking Llama2 Inference Serving on AMD EPYC Systems

The latency, throughput, and scaling testing described in this user guide uses EchoSwift*.

5.1 Explaining Benchmarking

The objective of the LLM-Inference-Bench tools measures the latency of each request in milliseconds per token, Time Taken for the First Token (TTFT), and Throughput measured in the number of tokens per second of request sent to a served LLM. These metrics are captured using varying input tokens (query length), output tokens (response length), and simulated parallel users.

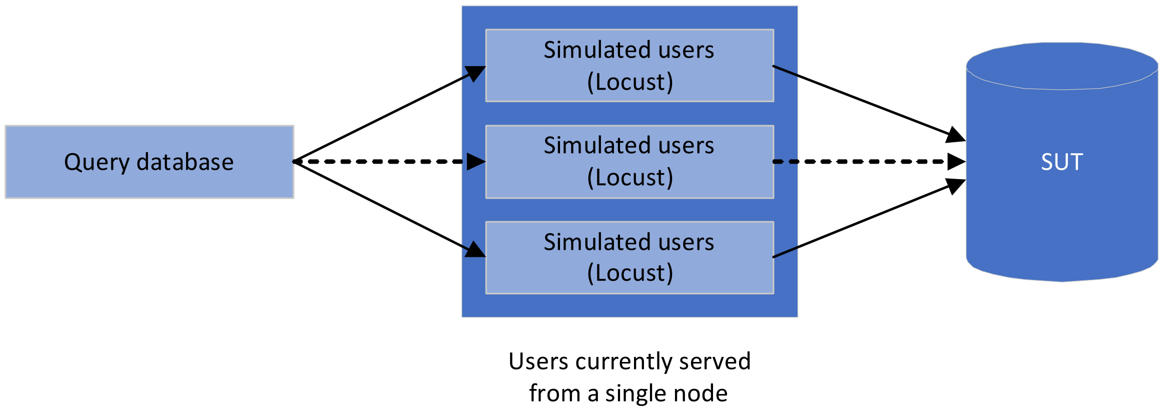

Figure 5-1: EchoSwift architecture

You can use this LLM inference benchmark to:

- **Measure performance:** Evaluate model latency, throughput, and resource (CPU-Core/Memory) utilization under varying workloads and configurations.
- **Identify bottlenecks:** Analyze performance metrics to pinpoint potential bottlenecks in different deployment scenarios impacting model efficiency and scalability.
- **Optimize deployment:** Utilize benchmarking data to fine-tune model deployment for improved resource utilization and cost-effectiveness.
- **Validate scalability:** Assess the model’s ability to handle increasing workloads and maintain performance with a growing number of concurrent requests.

Metrics captured:

- Number of input tokens
- Number of output tokens
- Throughput (tokens/second)
- End-to-end latency (ms)
- Token latency (ms/tokens)
- TTFT for streaming
- CPU and memory utilization while running the load test.

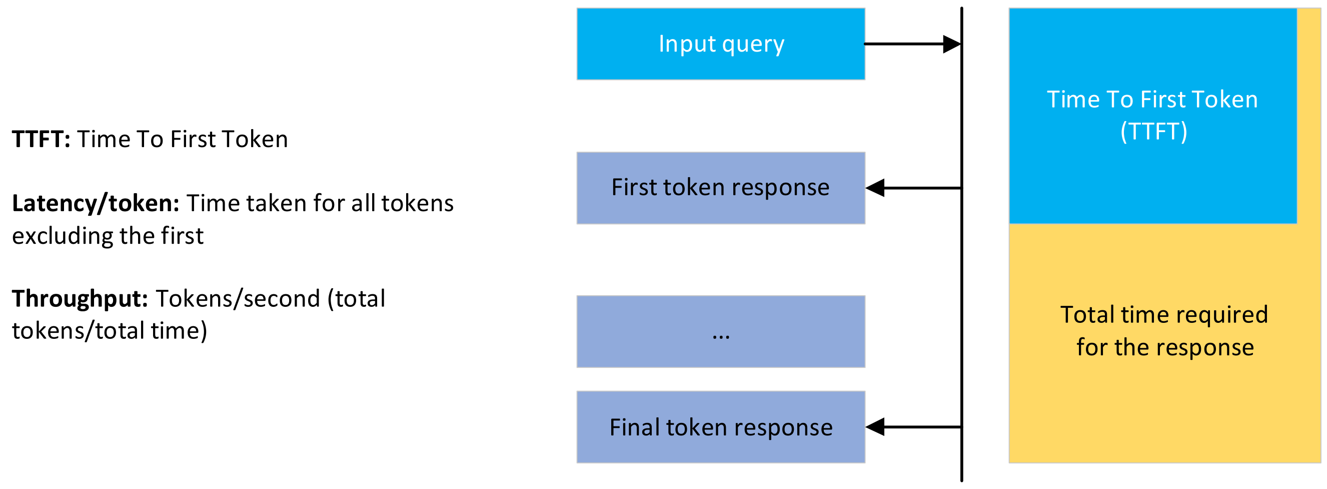
TTFT: Time To First Token
Latency/token: Time taken for all tokens excluding the first
Throughput: Tokens/second (total tokens/total time)
Input query
First token response
...
Final token response
Time To First Token (TTFT)
Total time required for the response

Figure 5-2: Latency, throughput, and TTFT calculations

Performance metrics calculations:

- **Throughput:** Measures the average number of tokens that can be generated per second:
  Throughput (tokens/Second) = (Output Token Length) / Total Time (s)
- **Token latency:** Measures the average time it takes to generate one token:
  Latency (ms/token) = Total Time (s) * 1000 / Output Token Length
- **TTFT:** Indicates the responsiveness of the model. Time from start of request to generation of first token.

5.2 Locust

Locust is an open-source tool that simulates multiuser requests on systems under test. It is built in python and excels at user behavior simulation. It is both flexible and easy of use, and its distributed testing capabilities allow simulating thousands of concurrent users. These features enable comprehensive system performance evaluation under realistic scenarios. The following prerequisites must be met in order to run the benchmark:

Item	Configuration/Version
# of cores	16
RAM	20 GB
datasets	2.13.1
locust	2.18.4
streamlit	1.29.0
docker	5.0.3
kserve	0.10.0
torch	1.13.0
transformers	4.31.0
huggingface_hub	0.20.0

Table 5-1: Locust hardware and software requirements

5.2.1 Benchmark Client

The benchmark script takes the required variables and starts sending requests to the previously-deployed model endpoint. Locust creates virtual users and simulates production-level load testing. The benchmark will run against the above generation endpoint, which can be any inference server endpoint, such as a TGI endpoint hosting a model (`http://localhost:8080/generate_stream`).

Define the configurations inside the `locust.sh` shell script before starting the load test. Here are some sample configurations:

- # of parallel users [1, 3, 10].
- Varying input tokens [32, 64, 128]
- Varying output tokens [32, 64, 128]

```
$   ./llm_inference_benchmark.sh<output_dir <generation_endpoint>
```

This page intentionally left blank.

Chapter 6

Linear Llama2 7B "CPU-Only" Scaling

Deploying LLMs for real-time inference tasks often requires efficient resource management to handle varying workloads. This chapter discusses linear scaling of CPU-only LLM serving on Llama2 7B utilizing an infrastructure built on Red Hat OpenShift. The primary objective is to showcase how the CPU resources allocated to Llama2 7B serving pods scale linearly with increasing inference requests.

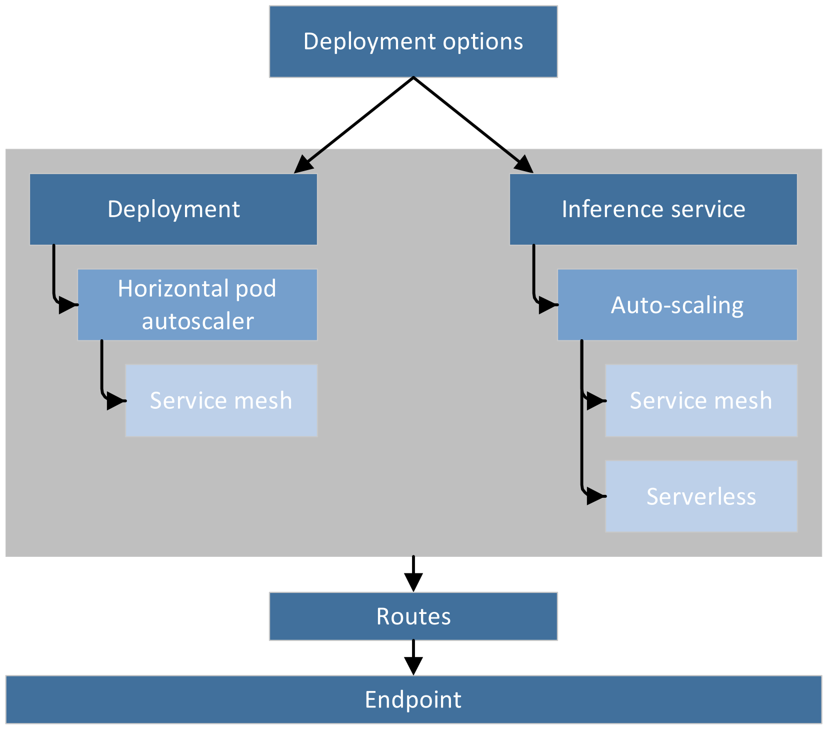

Figure 6-1: Deployment options

Figure 6-1 shows a Nutanix LLM deployment using Kind. The Deployment path offers a Horizontal Pod Autoscaler for scaling container instances and a Service Mesh for managing microservices. Autoscaling allows the number of serving replicas to adjust dynamically based on CPU utilization, thereby ensuring optimal performance and resource utilization.However, the Nutanix cluster only supports Persistent volume with the access mode Read-Write-Once; scaling multiple replicas is thus only possible if the node has enough resources for all of the replicas, to avoid pod mount/volume issues. Please see Persistent Volume Claims* for more information.

This page intentionally left blank.