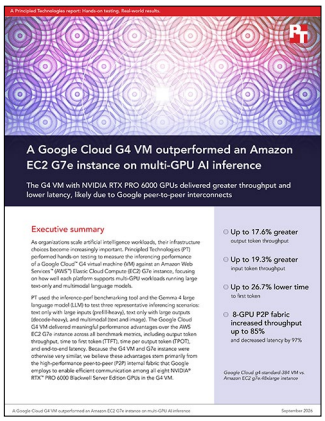


The science behind the report:

A Google Cloud G4 VM outperformed an Amazon EC2 G7e instance on multi-GPU AI inference

This document describes what we tested, how we tested, and what we found. To learn how these facts translate into real-world benefits, read the report A Google Cloud G4 VM outperformed an Amazon EC2 G7e instance on multi-GPU AI inference.

We concluded our hands-on testing on August 14, 2026. During testing, we determined the appropriate hardware and software configurations and applied updates as they became available. The results in this report reflect configurations that we finalized on August 14, 2026 or earlier. Unavoidably, these configurations may not represent the latest versions available when this report appears.

Our results

To learn more about how we have calculated the wins in this report, go to https://facts.pt/calculating-and-highlighting-wins. Unless we state otherwise, we have followed the rules and principles we outline in that document.

Table 1: Performance results of our testing. Higher is better. Source: PT.

	Google Cloud g4-standard-384	AWS EC2 g7e.48xlarge	Absolute difference	Relative difference
Input token rate (TPS)				
Prefill-heavy (text)	39,076.0	35,699.7	3,376.3	9.4%
Decode-heavy (text)	93.6	83.2	10.4	12.5%
Multimodal: Text and image	2,775.2	2,324.0	451.2	19.4%
Output token rate (TPS)				
Prefill-heavy (text)	823.5	822.3	1.2	0.1%
Decode-heavy (text)	2,442.7	2,222.3	220.4	9.9%
Multimodal: Text and image	284.3	241.8	42.5	17.5%
Image processing rate (images per second)				
Multimodal: Text and image	2,775.2	2,324.0	451.2	19.4%

Table 2: Latency results of our testing. Lower is better. Source: PT.

	Google Cloud g4-standard-384	AWS EC2 g7e.48xlarge	Absolute difference	Relative difference
Median TTFT (ms)				
Prefill-heavy (text)	950.2	927.8	-22.4	-2.4%
Decode-heavy (text)	77.3	105.6	28.3	26.8%
Multimodal: Text and image	1,006.5	1,267.1	260.6	20.5%
Median TPOT (ms)				
Prefill-heavy (text)	18.2	18.2	0.0	0.0%
Decode-heavy (text)	13.7	15.2	1.5	9.8%
Multimodal: Text and image	62.4	90.4	28.0	30.9%
Median request latency (ms)				
Prefill-heavy (text)	11,746.3	11,391.0	-355.3	-3.1%
Decode-heavy (text)	76,150.3	82,195.1	6,044.8	7.3%
Multimodal: Text and image	3,045.0	4,233.8	1,188.8	28.0%
Input token length				
Prefill-heavy (text)	14,945.0	14,685.0	260.0	1.7%
Decode-heavy (text)	204.5	192.5	12.0	5.8%
Multimodal: Text and image	282.0	274.5	7.5	2.6%

System configuration information

Table 3: Detailed information on the systems we tested.

General information	Google solution	Amazon solution
Date testing ended	8/14/2026	8/13/2026
Cloud service provider (CSP)	Google Cloud	Amazon Web Services
Region	us-west4	eu-central-1
Kubernetes version	v1.36.2-gke.2064000	v 1.36.2
vLLM image	vllm/vllm-openai:v0.21.1	vllm/vllm-openai:v0.21.1
Inference Workload 1		
Purpose	Investigate NCCL P2P fabric configuration	
Model	Meta Llama-3.1-70B-Instruct	
Workload client and version	vllm bench server v0.22.1	
Cluster environment	Docker	
P2P Fabric level	SYS / PHB / PXB / LOC	SYS / PHB / PXB / LOC / Default
Iterations and result choice	3 test runs, median reported	
Inference Workload 2		
Purpose	Prefill-heavy: large text input and modest text output	
Model	Google Gemma-4-26B-A4B	
Workload client and version	inference-perf v0.6.0 with transformers v5.13.0	
Cluster environment	Google Kubernetes Engine	Elastic Kubernetes Service
NVIDIA P2P fabric level	SYS	Default
Iterations and result choice	3 test runs, median reported	
Inference Workload 3		
Purpose	Decode-heavy: modest text input and large text output	
Model	Google Gemma-4-26B-A4B	
Workload client and version	inference-perf v0.6.0 with transformers v5.13.0	
Cluster environment	Google Kubernetes Engine	Elastic Kubernetes Service
P2P Fabric level	SYS	Default
Iterations and result choice	3 test runs, median reported	
Inference Workload 4		
Purpose	Multimodal: text + one image input and modest text output	
Model	Google Gemma-4-26B-A4B	
Workload client and version	inference-perf v0.6.0 with transformers v5.13.0	
Cluster environment	Google Kubernetes Engine	Elastic Kubernetes Service
P2P Fabric level	SYS	Default
Iterations and result choice	3 test runs, median reported	

General information	Google solution	Amazon solution
Cloud VMs for accelerator nodes		
Number of VMs	1	1
VM or instance	g4-standard-384	g7e.48xlarge
VM image	cos-125-19216-532-62	ami-003353cbbba8e1e5b
vCPU	384	192
Memory (GB)	1,440	2,048
Underlying processor model	5th generation AMD® EPYC® (Turin)	5th generation Intel® Xeon® Scalable (Emerald Rapids)
Max. network bandwidth (Gbps)	400	1,600
Additional storage (GB)	1,280	1,300
Accelerators	NVIDIA® RTX PRO 6000	NVIDIA RTX PRO 6000
Number of accelerators	8	8
High bandwidth memory (GB)	96	96
NVIDIA driver version	580.173.02	580.159.03
Cloud VMs for inference client nodes		
Number of VMs	1	1
VM machine / instance	n2-standard-16	m6i.4xlarge
VM image	cos-125-19216-532-62	ami-0b507eaedb1fd10ec
vCPUs	16	16
Memory (GB)	64	64
Underlying processor model	Intel Ice Lake or Cascade Lake	Intel Xeon 8375C
Max. network bandwidth (Gbps)	32	12.5
Additional storage	N/A	N/A
Cloud VMs for worker nodes		
Number of VMs	3	3
VM machine / instance	n2-standard-8	m6i.xlarge
VM image	cos-125-19216-532-62	ami-0b507eaedb1fd10ec
vCPU	8	4
Memory (GB)	32	16
Underlying processor model	Intel Ice Lake or Cascade Lake	Intel Xeon 8375C
Max. network bandwidth (Gbps)	16	12.5
Additional storage	N/A	N/A

How we tested

Summary

In this section, we describe our method for two classes of testing the eight-GPU G4 and G7e instances. For the first class of testing, we tested the inference performance for three use cases for the Gemma-4 model with one vLLM inference engine. For the second, we used three multi-GPU microbenchmarks (no inferencing) to determine any performance differences.

Testing inference with Gemma-4

For each cloud environment, create and configure one Kubernetes cluster with an eight-GPU VM/instance with the NVIDIA RTX PRO 6000 GPUs.

1. On the local control environment, install and configure CLI tools for each cloud environment
2. Create the base Kubernetes cluster
3. Create a client VM in the cluster for inference-perf
4. Create a storage bucket to contain the test results, and create roles and permissions so Kubernetes jobs could access the bucket storage.
5. Install inference-perf on the local control environment.
6. For each use case and inference-engine configuration, run inference-perf as follows:
    a. Create a vLLM pod.
    b. When vLLM is ready to accept requests, run inference-perf as a Kubernetes job. Its results are on bucket storage.
    c. When the job ends, retrieve the inference-perf results from storage as well as the job and vLLM logs.
    d. Create CSV files from the inference-perf JSON results.

Creating the test environment for GCP

1. Use the following parameters to define the GCP region, VM types, and networking.

```
export HF_TOKEN="hf_XXXXX"
export USER_EMAIL="XXXX&YYY"
export PROJECT_ID="XXXX"
export PROJECT_NUMBER="XXXXX"
export REGION="us-west4"
export SUB_REGION="a"
export CONTROL_PLANE_LOCATION="$REGION"
export PROXY_SUBNET_RANGE="10.120.0.0/23"
export VPC_NETWORK_NAME="default"
export PROXY_SUBNET_NAME="proxy-only-subnet-${REGION}"
export CLUSTER_NAME="vllm-inference-cluster"
export WORKER_MACH="n2-standard-8"
export CLIENT_MACH="n2-standard-16"
export ACCELERATOR_POOL_NAME="rtx6kpro-pool"
export ACCELERATOR_ZONES="${CONTROL_PLANE_LOCATION}-${SUB_REGION}"
export GPU_COUNT=8
export GPU_TYPE="nvidia-rtx-pro-6000"
export ACCEL_MACH="g4-standard-384"
export IP_STORAGE_TYPE=google_cloud_storage}
export IP_IMAGE="$REGION-docker.pkg.dev/$PROJECT_ID/ai-benchmark/inference-perf:v0.6.0-mod"
```

2. Create the base Kubernetes cluster, G4 instance, and client VM:

```
gcloud compute networks subnets create "${PROXY_SUBNET_NAME}" \
    --purpose=REGIONAL_MANAGED_PROXY \
    --role=ACTIVE \
    --region="${REGION}" \
    --network="${VPC_NETWORK_NAME}" \
    --range="${PROXY_SUBNET_RANGE}" \
    --project="${PROJECT_ID}"
gcloud container clusters create "${CLUSTER_NAME}" \
    --project="${PROJECT_ID}" \
    --location="${CONTROL_PLANE_LOCATION}" \
    --workload-pool="${PROJECT_ID}".svc.id.goog \v
```

```
    --release-channel=rapid \
    --machine-type="${WORKER_MACH}" \
    --num-nodes=1 \
    --enable-managed-prometheus \
    --monitoring=SYSTEM,DEPLOYMENT,HPA,POD,DCGM \
    --auto-monitoring-scope=ALL \
    --gateway-api=standard
gcloud container node-pools create "${ACCELERATOR_POOL_NAME}" \
    --cluster="${CLUSTER_NAME}" \
    --project="${PROJECT_ID}" \
    --workload-metadata=GKE_METADATA \
    --node-locations="${ACCELERATOR_ZONES}" \
    --location="${CONTROL_PLANE_LOCATION}" \
    --machine-type="${ACCEL_MACH}" \
    --accelerator type="${GPU_TYPE}",count="${GPU_COUNT}",gpu-driver-version=latest \
    --disk-size=1280Gi \
    --num-nodes=1
gcloud container node-pools create client-pool \
    --cluster="${CLUSTER_NAME}" \
    --project="${PROJECT_ID}" \
    --workload-metadata=GKE_METADATA \
    --node-locations="${ACCELERATOR_ZONES}" \
    --location="${CONTROL_PLANE_LOCATION}" \
    --machine-type="${CLIENT_MACH}" \
    --num-nodes=1
```

3. Create the storage bucket and service account, apply IAM roles and policies, and add HF token to a Kubernetes secret:

```
gcloud storage buckets create gs://campos-olp --location=US --uniform-bucket-level-access
kubectl create serviceaccount inf-perf
gcloud iam service-accounts create iam-inf-perf --project=${PROJECT_ID}
for role in roles/storage.{objectAdmin,buckets.{get,list},objects.list}; do
  gcloud storage buckets add-iam-policy-binding gs://campos-olp \
    --member="serviceAccount:iam-inf-perf@${PROJECT_ID}.iam.gserviceaccount.com" \
    --role="$role"
done
gcloud iam service-accounts add-iam-policy-binding \
  iam-inf-perf@${PROJECT_ID}.iam.gserviceaccount.com \
  --role="roles/iam.workloadIdentityUser" \
  --member="serviceAccount:${PROJECT_ID}.svc.id.goog[default/inf-perf]"
kubectl annotate sa inf-perf \
  iam.gke.io/gcp-service-account=iam-inf-perf@${PROJECT_ID}.iam.gserviceaccount.com
kubectl create secret generic hf-token --from-literal=token="${HF_TOKEN}" \
 --dry-run=client -o yaml | kubectl apply -f -
```

4. Create a Docker image of inference-perf v0.6.0 that uses python `transformers` module v5.13.0 instead of v4.57.6. This change is needed for Gemma-4 tokenization in inference-perf. Use the same image you created for AWS. See step 4 in the next section for details.
5. Create a GCP repository for this image, and push it there:

```
gcloud artifacts repositories create ai-benchmark --location="$REGION" \
  --repository-format=docker
gcloud auth print-access-token | docker login -u oauth2accesstoken \
  --password-stdin https://"$REGION"-docker.pkg.dev
docker tag $IP_IMAGE "$REGION-docker.pkg.dev/$PROJECT_ID/ai-benchmark/$IP_IMAGE"
docker push "$REGION-docker.pkg.dev/$PROJECT_ID/ai-benchmark/$IP_IMAGE"
```

Creating the test environment for AWS

1. Use the following parameters to define the AWS region, VM types, and networking:

```
export K8S_VERSION=1.35
export AWS_REGION="eu-central-1"
export CLUSTER=vllm-cluster
export NVIDIA_AMI="$(aws ssm get-parameter --name /aws/service/bottlerocket/aws-k8s-${K8S_VERSION}-
nvidia/x86_64/latest/image_id --query Parameter.Value --output text)"
export STANDARD_AMI="$(aws ssm get-parameter --name /aws/service/bottlerocket/aws-k8s-${K8S_
VERSION}/x86_64/latest/image_id --query Parameter.Value --output text)"
export ACCOUNT_ID="$(aws sts get-caller-identity --query Account --output text)"
export AWS_ACCOUNT_ID="$ACCOUNT_ID"
export WORKER_MACH=m6i.2xlarge
export CLIENT_MACH=m6i.4xlarge
export ACCEL_MACH=g7e.48xlarge
export IP_STORAGE_TYPE="simple_storage_service"
export IP_IMAGE="$REGION-docker.pkg.dev/$PROJECT_ID/ai-benchmark/inference-perf:v0.6.0-mod"
```

2. Create the base Kubernetes cluster, G7e instance, and client VM:

```
# the files cluster.yaml, acccel.yaml and client.yaml are in the Appendix
envsubst < cluster.yaml | eksctl create cluster -f -
envsubst < accel.yaml | eksctl create  nodegroup -f -
envsubst < client.yaml | eksctl create  nodegroup -f -
```

3. Create the storage bucket and service account, apply IAM roles and policies, and add our HF token to a Kubernetes secret:

```
aws s3api create-bucket --bucket campos-olp --region "$AWS_REGION"
eksctl utils associate-iam-oidc-provider --cluster "$CLUSTER" --approve
# The file s3-access-policy.json is in the appendix.
aws iam create-policy --policy-name s3-policy --policy-document file://s3-access-policy.json
kubectl apply -f - << EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: inf-perf
EOF
oidc_provider=$(aws eks describe-cluster --name "$CLUSTER" --region "$AWS_REGION" --query "cluster.
identity.oidc.issuer" --output text | sed -e "s/^https:\/\///")
service_account=inf-perf
namespace=default
cat >s3-trust-policy.json <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated": "arn:aws:iam::$ACCOUNT_ID:oidc-provider/$oidc_provider"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "$oidc_provider:aud": "sts.amazonaws.com",
          "$oidc_provider:sub": "system:serviceaccount:$namespace:$service_account"
        }
      }
    }
  ]
}
EOF
aws iam create-role --role-name s3-role --assume-role-policy-document \
  file://s3-trust-policy.json --description "access s3 from EKS"
aws iam attach-role-policy --role-name s3-role \
  --policy-arn="arn:aws:iam::$ACCOUNT_ID:policy/s3-policy"
```

```
kubectl annotate serviceaccount -n "$namespace" "$service_account" \
  eks.amazonaws.com/role-arn="arn:aws:iam::$ACCOUNT_ID:role/s3-role"
kubectl create secret generic hf-token --from-literal=token="${HF_TOKEN}" \
 --dry-run=client -o yaml | kubectl apply -f -
```

4. For Gemma-4 tokenization in inference-perf, you need an inference-perf image that uses python `transformers` module v5.13.0 instead of v4.57.6. Use the following script to create a Docker image of inference-perf v0.6.0, create an AWS registry, and push the inference-perf image to it:

```
#!/usr/bin/env bash
set -euo pipefail
AWS_REGION="us-east-1"
ECR_REPO_NAME="inference-perf"
IMAGE_TAG="v0.6.0-mod"
DOCKERFILE_PATH="."
EKS_CLUSTER_NAME="vllm-cluster"
AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)
ECR_REGISTRY="${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com"
ECR_URI="${ECR_REGISTRY}/${ECR_REPO_NAME}"
aws ecr describe-repositories --repository-names "${ECR_REPO_NAME}" \
  --region "${AWS_REGION}" >/dev/null 2>&1 || \
    aws ecr create-repository \
    --repository-name "${ECR_REPO_NAME}" \
    --region "${AWS_REGION}" \
    --image-scanning-configuration scanOnPush=true \
    --image-tag-mutability IMMUTABLE
aws ecr get-login-password --region "${AWS_REGION}" | \
  docker login --username AWS --password-stdin "${ECR_REGISTRY}"
docker build -t "${ECR_REPO_NAME}:${IMAGE_TAG}" "${DOCKERFILE_PATH}"
docker tag "${ECR_REPO_NAME}:${IMAGE_TAG}" "${ECR_URI}:${IMAGE_TAG}"
docker push "${ECR_URI}:${IMAGE_TAG}"
```

Testing procedure

For each use case, create one vLLM pod with the necessary vLLM arguments. Once the pod is ready, deploy the inference-perf job with the use case's configuration.

Note that we varied the parallelization strategy for each platform and for each use case to see which strategy produced the best performance. That is, we looked at four combinations of (tensor parallel size/data parallel size): (8/1), size): (8/1), (4/2), (2/4), and (1/8), keeping the other vLLM parameters the same. The vLLM parameters below are for the (8/1) parallelization strategy.

We ran each configuration three times and report the median.

Prefill-heavy (text) use case

1. Additional vLLM arguments:

```
            - "--tensor-parallel-size=8"
            - "--data-parallel-size=1"
            - "--tool-call-parser=gemma4"
            - "--reasoning-parser=gemma4"
            - "--enable-auto-tool-choice"
            - "--max-model-len=10000
            - "--max-num-seqs=32"
            - "--long-prefill-token-threshold=8192"
            - "--gpu-memory-utilization=0.90"
            - "--trust-remote-code"
```

2. Inference-perf configuration file:

```
CONFIG_YML=inference-perf-prefill.yaml
```

Decode-heavy (text) use case

1. Additional vLLM arguments:

```
            - "--tensor-parallel-size=8"
            - "--data-parallel-size=1"
            - "--max-model-len=8704"
            - "--max-num-seqs=200"
            - "--max-num-batched-tokens=2560"
            - "--gpu-memory-utilization=0.90"
            - "--async-scheduling"
            - '--compilation-config={"cudagraph_mode":"FULL_DECODE_ONLY"}'
            - '--limit-mm-per-prompt={"image":0,"audio":0}'
```

2. Inference-perf configuration file:

```
CONFIG_YML=inference-perf-decode.yaml
```

Multimodal (text and image) use case

1. Additional vLLM arguments:

```
            - "--tensor-parallel-size=8"
            - "--data-parallel-size=1"
            - "--max-model-len=4096"
            - "--max-num-seqs=128"
            - "--max-num-batched-tokens=2560"
            - "--gpu-memory-utilization=0.90"
            - "--api-server-count=4"
            - "--async-scheduling"
            - '--limit-mm-per-prompt={"image":1,"audio":0}'
            - '--mm-processor-kwargs={"max_soft_tokens":560}'
```

2. Inference-perf configuration file:

```
CONFIG_YML=inference-perf-multimodal.yaml
```

Performing one test run

1. Start the vLLM pod, using the file vllm-gemma4.yaml with the correct vLLM arguments added:

```
envsubst < vllm-gemma4.yaml | kubectl apply -f -
```

2. When the pod is ready for inference requests, run inference-perf with the correct config file for the use case:

```
kubectl delete cm inference-perf-config
rm -f config.yml
envsubst < ${CONFIG_YML} > config.yml
kubectl create configmap inference-perf-config --from-file=config.yml
```

3. Run the inference-perf job:

```
envsubst < ip_manifest.yaml | kubectl apply -f -
```

4. Once the job finishes, collect the logs:

```
kubectl logs job/inference-perf > client_logs.txt
kubectl logs deployment/vllm-gemma4 > VLLM_logs.txt
```

Using microbenchmarking to measure the multi-GPU peer-to-peer performance

Provisioning the test instances

PT ran three microbenchmarks on each instance: `p2pBandwidthLatencyTest`, `nvbandwidth`, and NCCL Tests. For brevity, this document presents only the `p2pBandwidthLatencyTest` results; the software installation steps below are the steps PT took and include the packages the other two tools require.

This section lists prerequisites for the microbenchmarks. Provision one instance on each cloud with the following attributes:

- **Google Cloud:** one `g4-standard-384` instance (eight NVIDIA RTX PRO 6000 Blackwell Server Edition GPUs)
- **AWS:** one `g7e.48xlarge` instance (eight NVIDIA RTX PRO 6000 Blackwell Server Edition GPUs)
- **On both:** use the cloud provider's public Ubuntu 24.04 LTS image, not a preconfigured deep-learning image, so that the driver, CUDA, and NCCL versions below install cleanly without removing a preinstalled stack
- **On both:** a 200GB root disk, an SSH login with sudo, and outbound internet access to `developer.download.nvidia.com` and `github.com`
- **On both:** all eight GPUs visible to the guest, which `lspci | grep -i nvidia | wc -l` confirms by returning 8

The driver version this methodology pins requires a Linux 6.x kernel. The Google Cloud Ubuntu 24.04 image PT used booted a 6.17 kernel. Our AWS image booted a 7.0 kernel, against which the pinned driver did not compile, so the AWS instance required us to pin the kernel version to 6.17. Every other step is identical on both clouds.

Installing the pinned driver, CUDA, and NCCL stack

This section installs the following software stack on both instances: NVIDIA open kernel module driver 580.126.09, CUDA Toolkit 13.0 (package 13.0.3-1, nvcc 13.0.88), and NCCL 2.28.3.

1. Install the build tools the benchmarks need:

```
sudo apt update && sudo apt install -y build-essential cmake git wget libboost-program-options-dev
```

2. Add the NVIDIA CUDA apt repository for Ubuntu 24.04:

```
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2404/x86_64/cuda-
keyring_1.1-1_all.deb
sudo dpkg -i cuda-keyring_1.1-1_all.deb && sudo apt update
```

3. Confirm the repository still offers the pinned versions:

```
apt-cache madison nvidia-open-580 | grep 580.126.09
apt-cache madison libnccl2 | grep 2.28.3
```

4. Install the driver with every sub-package pinned to the same build:

```
V=580.126.09-1ubuntu1
sudo apt install -y --no-install-recommends \
  nvidia-open-580=$V nvidia-driver-580-open=$V nvidia-dkms-580-open=$V \
  nvidia-kernel-source-580-open=$V nvidia-kernel-common-580=$V nvidia-firmware-580=$V \
  libnvidia-common-580=$V libnvidia-extra-580=$V libnvidia-cfg1-580=$V \
  libnvidia-compute-580=$V libnvidia-decode-580=$V libnvidia-encode-580=$V \
  libnvidia-fbc1-580=$V libnvidia-gl-580=$V libnvidia-gpucomp-580=$V \
  xserver-xorg-video-nvidia-580=$V
```

5. Install the CUDA toolkit and NCCL:

```
sudo apt install -y cuda-toolkit-13-0
sudo apt install -y libnccl2=2.28.3-1+cuda13.0 libnccl-dev=2.28.3-1+cuda13.0
```

6. Hold every pinned package:

```
sudo apt-mark hold nvidia-open-580 nvidia-driver-580-open nvidia-dkms-580-open \
  nvidia-kernel-source-580-open nvidia-kernel-common-580 nvidia-firmware-580 \
  libnvidia-common-580 libnvidia-extra-580 libnvidia-cfg1-580 libnvidia-compute-580 \
  libnvidia-decode-580 libnvidia-encode-580 libnvidia-fbc1-580 libnvidia-gl-580 \
  libnvidia-gpucomp-580 xserver-xorg-video-nvidia-580 libnccl2 libnccl-dev
```

7. Add the CUDA binaries and libraries to the shell environment:

```
cat >> ~/.bashrc <<'EOF'
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64${LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH}
EOF
```

8. Reboot the instance, and log back in.

Recording the software versions

Capture the installed versions to a file before running any benchmark, so that the results carry their own provenance. Held packages show a status of `hi` rather than `ii`; that is expected.

1. Create the results directory, and write the version record:

```
mkdir -p ~/results/nccl
{
  echo "=== Driver ==="
  head -1 /proc/driver/nvidia/version
  dpkg-query -W -f='${Package} ${Version} [${db:Status-Abbrev}]\n' \
    nvidia-open-580 nvidia-driver-580-open nvidia-dkms-580-open libnvidia-compute-580
  echo "=== CUDA toolkit ==="
  nvcc --version | grep release
  dpkg-query -W -f='${Package} ${Version} [${db:Status-Abbrev}]\n' cuda-toolkit-13-0
  echo "=== NCCL ==="
  dpkg-query -W -f='${Package} ${Version} [${db:Status-Abbrev}]\n' libnccl2 libnccl-dev
  grep -E '^#define NCCL_(MAJOR|MINOR|PATCH)' /usr/include/nccl.h | awk '{printf "%s=%s ", $2, $3}
END {print ""}'
  echo "=== OS / kernel ==="
  lsb_release -ds; uname -r
  echo "=== GPUs ==="
  nvidia-smi --query-gpu=index,name,driver_version --format=csv,noheader
  echo "Captured: $(date -u +%Y-%m-%dT%H:%M:%SZ) on $(hostname)"
} | tee ~/results/versions.txt
```

2. Confirm the record shows driver 580.126.09, CUDA release 13.0, `NCCL_MAJOR=2 NCCL_MINOR=28 NCCL_PATCH=3`, a 6.17 kernel, and eight GPUs before continuing.

Inspecting the GPU topology

The three benchmarks measure the effect of two things the hypervisor controls: how the eight GPUs attach to the PCIe fabric, and whether the platform allows high-speed, low-latency peer-to-peer (P2P) memory access between any given pair. The `nvidia-smi topo -m` matrix reports the guest-visible link type between each pair (PIX for a shared PCIe switch, NODE for a shared CPU socket, SYS for a cross-socket path); `nvidia-smi topo -p2p` reports whether the platform allows peer reads and writes for each pair.

1. Record the topology matrix, the P2P read and write capability matrices, and the PCIe link parameters, then confirm that the last command reports PCIe generation 5 and link width 16 for every GPU:

```
nvidia-smi topo -m | tee ~/results/topo_matrix.txt
nvidia-smi topo -p2p r | tee ~/results/topo_p2p_read.txt
nvidia-smi topo -p2p w | tee ~/results/topo_p2p_write.txt
nvidia-smi --query-gpu=index,pci.bus_id,pcie.link.gen.max,pcie.link.width.max --format=csv | tee ~/
results/pcie_links.txt
```

In our testing, both instances presented the same `topo -m` matrix: four PIX pairs (GPUs 0 and 1, 2 and 3, 4 and 5, 6 and 7), NODE between pairs on the same socket, and SYS across sockets. The `topo -p2p` matrices differed. The Google Cloud instance reported OK for all 56 directional GPU pairs, and the AWS instance reported OK for the eight directional cells of the four PIX pairs and NS (not supported) for the remaining 48.

GCP and AWS — GPU interconnect matrix

Table 4: Guest-visible GPU interconnect matrix from nvidia-smi topo -m on the Google Cloud g4-standard-384 instance. The AWS g7e.48xlarge instance presented an identical link-type matrix; only its CPU affinity ranges differed (0-47,96-143 for NUMA node 0 and 48-95,144-191 for node 1). PIX = same PCIe switch; NODE = same CPU socket, different switch; SYS = other CPU socket.

GPU	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	CPU affinity	NUMA
GPU0	—	PIX	NODE	NODE	SYS	SYS	SYS	SYS	0-95,192-287	0
GPU1	PIX	—	NODE	NODE	SYS	SYS	SYS	SYS	0-95,192-287	0
GPU2	NODE	NODE	—	PIX	SYS	SYS	SYS	SYS	0-95,192-287	0
GPU3	NODE	NODE	PIX	—	SYS	SYS	SYS	SYS	0-95,192-287	0
GPU4	SYS	SYS	SYS	SYS	—	PIX	NODE	NODE	96-191,288-383	1
GPU5	SYS	SYS	SYS	SYS	PIX	—	NODE	NODE	96-191,288-383	1
GPU6	SYS	SYS	SYS	SYS	NODE	NODE	—	PIX	96-191,288-383	1
GPU7	SYS	SYS	SYS	SYS	NODE	NODE	PIX	—	96-191,288-383	1

GCP — P2P read capability (nvidia-smi topo -p2p r)

Table 5: P2P read capability from nvidia-smi topo -p2p r, Google Cloud g4-standard-384. OK = direct peer access for the pair. The write matrix was identical.

GPU	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
GPU0	—	OK	OK	OK	OK	OK	OK	OK
GPU1	OK	—	OK	OK	OK	OK	OK	OK
GPU2	OK	OK	—	OK	OK	OK	OK	OK
GPU3	OK	OK	OK	—	OK	OK	OK	OK
GPU4	OK	OK	OK	OK	—	OK	OK	OK
GPU5	OK	OK	OK	OK	OK	—	OK	OK
GPU6	OK	OK	OK	OK	OK	OK	—	OK
GPU7	OK	OK	OK	OK	OK	OK	OK	—

AWS — P2P read capability (nvidia-smi topo -p2p r)

Table 6: P2P read capability from nvidia-smi topo -p2p r, AWS g7e.48xlarge. NS = not supported. The write matrix was identical.

GPU	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
GPU0	—	OK	NS	NS	NS	NS	NS	NS
GPU1	OK	—	NS	NS	NS	NS	NS	NS
GPU2	NS	NS	—	OK	NS	NS	NS	NS
GPU3	NS	NS	OK	—	NS	NS	NS	NS
GPU4	NS	NS	NS	NS	—	OK	NS	NS
GPU5	NS	NS	NS	NS	OK	—	NS	NS
GPU6	NS	NS	NS	NS	NS	NS	—	OK
GPU7	NS	NS	NS	NS	NS	NS	OK	—

Building and running p2pBandwidthLatencyTest

`p2pBandwidthLatencyTest` from NVIDIA CUDA samples measures copy bandwidth and latency between every pair of GPUs. It runs each test twice, once with CUDA P2P access disabled and once with it enabled, so a single run yields data for unidirectional bandwidth, bidirectional bandwidth, and latency, each with P2P off and on. When P2P access is unavailable, the tool falls back to a host-staged copy in the "enabled" pass, so on the AWS instance, the non-adjacent cells of the P2P-enabled matrices measure that fallback rather than direct P2P. The default mode uses peer writes; a separate flag measures peer reads. Run the default mode three times to characterize run-to-run variation for the sub-microsecond latency tests, and run the read mode once.

The tables below contain data from only the default-mode passes. Each table cell is the median of the three default-mode passes for that GPU pair and direction. We omit the diagonal cells, which measure local copies within one GPU's own memory.

In the summary table, the adjacent-GPU results are the median of the eight cells for the four GPU pairs that share a PCIe switch (GPUs 0 and 1, 2 and 3, 4 and 5, and 6 and 7, counting each direction separately). The non-adjacent-GPU results are the median of the remaining 48 pairings.

1. Clone the CUDA samples at the v13.0 tag, and record the tag:

```
cd ~ && git clone --branch v13.0 --depth 1 https://github.com/NVIDIA/cuda-samples.git
cd cuda-samples && git describe --tags | tee ~/results/cuda_samples_version.txt
```

2. Build the sample:

```
cd Samples/5_Domain_Specific/p2pBandwidthLatencyTest
mkdir -p build && cd build && cmake .. && make -j
```

3. Run the default (peer-write) test three times, saving each pass:

```
for i in 1 2 3; do ./p2pBandwidthLatencyTest | tee ~/results/p2p_bwlat_run${i}.txt; done
```

4. Run the peer-read test once:

```
./p2pBandwidthLatencyTest -p2p_read | tee ~/results/p2p_bwlat_read.txt
```

Summary - Adjacent-GPU pairs (same PCIe switch) vs non-adjacent-GPU pairs medians

Table 7: p2pBandwidthLatencyTest summary, medians over the eight directional cells of the four PIX pairs (adjacent GPUs) and over the remaining 48 cells (non-adjacent GPUs), from three default-mode runs per cloud. Bandwidth in GB/s, higher is better; latency in microseconds, lower is better.

Metric	GCP adjacent GPUs	GCP non-adjacent GPUs	AWS adjacent GPUs	AWS non-adjacent GPUs
Uni BW, P2P on (GB/s)	54.21	52.72	54.84	40.72
Bi BW, P2P on (GB/s)	102.14	102.10	105.05	55.15
Latency, P2P on (µs)	0.45	0.45	0.49	20.27
Latency, P2P off (µs)	16.44	16.96	19.44	20.25

GCP — Unidirectional bandwidth, P2P enabled (GB/s)

Table 8: p2pBandwidthLatencyTest unidirectional bandwidth with P2P enabled, GPU (row) to GPU (column), Google Cloud g4-standard-384, in GB/s, median of three runs. Higher is better. Shaded cells are GPU pairs with direct P2P.

GPU	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
GPU0	—	53.33	52.46	51.93	52.62	52.00	53.17	53.29
GPU1	54.17	—	52.21	52.39	52.96	52.40	52.68	53.38
GPU2	53.29	52.94	—	54.04	52.52	52.86	52.94	52.95
GPU3	53.49	52.96	53.71	—	52.57	52.73	52.68	52.12
GPU4	53.12	52.68	52.91	52.72	—	54.28	52.59	53.29
GPU5	53.14	52.86	52.49	52.95	54.25	—	53.23	52.64
GPU6	52.66	52.63	52.65	53.75	52.16	52.81	—	54.78
GPU7	53.17	52.67	52.07	52.96	52.31	52.71	54.70	—

AWS — Unidirectional bandwidth, P2P enabled (GB/s)

Table 9: p2pBandwidthLatencyTest unidirectional bandwidth with P2P enabled, GPU (row) to GPU (column), AWS g7e.48xlarge, in GB/s, median of three runs. Higher is better. Non-adjacent GPU cells measure the tool's host-staged fallback, not direct P2P. Shaded cells are GPU pairs with direct P2P.

GPU	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
GPU0	—	55.04	40.90	40.83	40.63	40.73	40.60	40.68
GPU1	55.18	—	40.72	40.75	40.65	40.77	40.65	40.67
GPU2	40.61	40.80	—	54.66	40.64	40.53	40.64	40.66
GPU3	40.56	40.77	54.82	—	40.71	40.61	40.68	40.66
GPU4	41.05	40.72	40.78	40.96	—	54.86	40.65	40.83
GPU5	40.62	41.17	40.69	40.78	55.46	—	40.72	40.90
GPU6	40.85	40.65	40.62	41.45	40.38	40.67	—	54.58
GPU7	40.80	40.86	41.02	41.40	40.51	40.79	54.76	—

GCP — Bidirectional bandwidth, P2P enabled (GB/s)

Table 10: p2pBandwidthLatencyTest bidirectional bandwidth with P2P enabled, GPU (row) and GPU (column), Google Cloud g4-standard-384, in GB/s, median of three runs. Higher is better. Shaded cells are GPU pairs with direct P2P.

GPU	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
GPU0	—	101.99	101.44	102.00	101.73	101.60	102.19	102.16
GPU1	102.07	—	102.43	101.88	102.08	101.68	102.34	102.36
GPU2	101.62	102.51	—	102.07	102.19	102.15	101.61	101.76
GPU3	101.96	101.82	102.22	—	102.13	102.20	101.52	101.80
GPU4	102.20	102.26	102.12	101.83	—	102.02	102.20	102.21
GPU5	101.56	101.84	102.17	102.18	102.77	—	102.03	102.00
GPU6	102.12	102.45	101.45	102.27	102.21	102.66	—	102.40
GPU7	101.96	102.36	101.72	101.78	102.59	102.06	102.63	—

AWS — Bidirectional bandwidth, P2P enabled (GB/s)

Table 11: p2pBandwidthLatencyTest bidirectional bandwidth with P2P enabled, GPU (row) and GPU (column), AWS g7e.48xlarge, in GB/s, median of three runs. Higher is better. Non-adjacent GPU cells measure the tool's host-staged fallback, not direct P2P. Shaded cells are GPU pairs with direct P2P.

GPU	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
GPU0	—	105.11	55.01	54.89	55.34	55.41	55.72	54.95
GPU1	105.16	—	54.90	54.95	55.45	56.01	55.60	55.96
GPU2	55.16	55.06	—	105.66	55.00	54.97	55.47	55.29
GPU3	55.01	54.88	104.84	—	55.77	55.66	55.18	55.39
GPU4	55.00	54.62	55.20	55.19	—	105.29	54.83	55.17
GPU5	54.93	55.49	55.08	55.26	104.95	—	55.37	55.22
GPU6	54.84	55.06	55.13	54.89	55.21	55.21	—	104.98
GPU7	54.76	54.82	54.87	55.35	55.02	54.35	104.84	—

GCP — Latency, P2P enabled (µs) — lower is better

Table 12: p2pBandwidthLatencyTest latency with P2P enabled, GPU (row) to GPU (column), Google Cloud g4-standard-384, in microseconds, median of three runs. Lower is better. Shaded cells are GPU pairs with direct P2P.

GPU	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
GPU0	—	0.45	0.45	0.52	0.44	0.51	0.52	0.45
GPU1	0.52	—	0.46	0.45	0.51	0.52	0.45	0.45
GPU2	0.45	0.50	—	0.44	0.45	0.52	0.51	0.44
GPU3	0.45	0.45	0.44	—	0.52	0.46	0.45	0.46
GPU4	0.52	0.45	0.45	0.46	—	0.46	0.46	0.51
GPU5	0.45	0.44	0.44	0.44	0.45	—	0.46	0.44
GPU6	0.45	0.46	0.46	0.45	0.45	0.45	—	0.45
GPU7	0.51	0.52	0.52	0.45	0.45	0.45	0.45	—

AWS — Latency, P2P enabled (µs) — lower is better

Table 13: p2pBandwidthLatencyTest latency with P2P enabled, GPU (row) to GPU (column), AWS g7e.48xlarge, in microseconds, median of three runs. Lower is better. Non-adjacent GPU cells measure the tool's host-staged fallback. Shaded cells are GPU pairs with direct P2P.

GPU	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
GPU0	—	0.47	19.62	19.29	20.28	19.29	20.29	20.19
GPU1	0.45	—	19.44	19.27	20.27	20.28	20.28	20.28
GPU2	19.81	19.38	—	0.52	20.29	20.29	20.29	20.29
GPU3	19.33	18.58	0.47	—	20.28	20.29	20.29	20.27
GPU4	20.28	20.28	20.20	20.30	—	0.47	19.97	20.28
GPU5	20.30	20.29	20.29	20.27	0.52	—	20.06	20.18
GPU6	20.29	20.30	20.30	20.28	20.22	20.26	—	0.52
GPU7	20.28	20.27	20.20	20.08	20.26	20.25	0.52	—

Appendix: Auxiliary files

```
### File cluster.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: $CLUSTER
  region: $AWS_REGION
  version: "$K8S_VERSION"
autoModeConfig:
  enabled: false
addons:
  - name: vpc-cni
    version: latest
  - name: coredns
    version: latest
  - name: kube-proxy
    version: latest
managedNodeGroups:
  - name: vllm-cluster-nodes
    instanceType: $WORKER_MACH
    ami: $STANDARD_AMI
    amiFamily: Bottlerocket
    desiredCapacity: 3
    labels:
      role: cpu-worker
    tags:
      nodegroup-role: cpu-worker
### File accel.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: $CLUSTER
  region: $AWS_REGION
  version: "$K8S_VERSION"
autoModeConfig:
  enabled: false
managedNodeGroups:
  - name: vllm-g7e-nodes
    spot: true
    instanceType: $ACCEL_MACH
    desiredCapacity: 1
    volumeSize: 1300
    ami: $NVIDIA_AMI
    amiFamily: Bottlerocket
    privateNetworking: true
    labels:
      role: large-model-worker
      nvidia.com/gpu: "true"
      k8s.amazonaws.com/accelerator: nvidia-gpu
    tags:
      nodegroup-role: large-model-worker
### File client.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: $CLUSTER
  region: $AWS_REGION
  version: "$K8S_VERSION"
managedNodeGroups:
  - name: client-pool
    instanceType: $CLIENT_MACH
    ami: $STANDARD_AMI
    amiFamily: Bottlerocket
    desiredCapacity: 1
    labels:
      role: client
    tags:
      nodegroup-role: client
    iam:
```

```
      withAddonPolicies:
        cloudWatch: true
### File ip_manifest.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: inference-perf
  labels: {"app": "inference-perf"}
spec:
  template:
    metadata:
      labels: {"app": "inference-perf"}
    spec:
      serviceAccountName: inf-perf
      nodeSelector:
        cloud.google.com/gke-nodepool: client-pool
      containers:
        - name: inference-perf
          image: "$IP_IMAGE"
          imagePullPolicy: Always
          command: ["inference-perf"]
          args: ["--config_file", "/etc/config/config.yml", "--log-level", "INFO"]
          env:
            - name: HF_TOKEN
              valueFrom:
                secretKeyRef:
                  name: hf-token
                  key: token
          volumeMounts:
            - {name: config-volume, mountPath: /etc/config, readOnly: true}
      restartPolicy: Never
      volumes:
        - name: config-volume
          configMap: {"name": "inference-perf-config"}
### File vllm-gemma4.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vllm-gemma4
spec:
  replicas: 1
  selector:
    matchLabels:
      app: vllm-gemma4
  template:
    metadata:
      labels:
        app: vllm-gemma4
    spec:
      nodeSelector:
        ### node.kubernetes.io/instance-type:
        node.kubernetes.io/instance-type: $ACCEL_MACH
      volumes:
        - name: cache
          emptyDir: {}
        - name: shm
          emptyDir:
            medium: Memory
            sizeLimit: 32Gi
      containers:
        - name: vllm
          image: vllm/vllm-openai:v0.22.1
          args:
            - "--model=google/gemma-4-26B-A4B"
            - "--dtype=bfloat16"
            - "--host=0.0.0.0"
            - "--port=8000"
# NOTE -- insert the vLLM arguments for the use case here:
####
####
          env:
```

```
            - name: HF_TOKEN
              valueFrom:
                secretKeyRef:
                  name: hf-token
                  key: token
            - {name: NCCL_P2P_LEVEL, value: "SYS"}
            - {name: NCCL_DEBUG, value: "WARN"}
            - {name: VLLM_HTTP_TIMEOUT_KEEP_ALIVE, value: "30"}
          ports:
            - containerPort: 8000
          resources:
            limits:
              nvidia.com/gpu: 8
          volumeMounts:
            - name: cache
              mountPath: /root/.cache/huggingface
            - name: shm
              mountPath: /dev/shm
          readinessProbe:
            httpGet:
              path: /health
              port: 8000
            initialDelaySeconds: 300
            periodSeconds: 60
          startupProbe:
            httpGet:
              path: /health
              port: 8000
            failureThreshold: 60
            periodSeconds: 30
---
apiVersion: v1
kind: Service
metadata:
  name: vllm-gemma4
spec:
  selector:
    app: vllm-gemma4
  ports:
    - port: 80
      targetPort: 8000
  type: ClusterIP
# File inference-perf-prefill.yaml
load:
  type: concurrent
  stages:
  - {num_requests: 450, concurrency_level:  6}
  - {num_requests: 450, concurrency_level: 12}
  - {num_requests: 450, concurrency_level: 18}
  - {num_requests: 450, concurrency_level: 24}
  - {num_requests: 450, concurrency_level: 30}
  - {num_requests: 450, concurrency_level: 36}
  - {num_requests: 450, concurrency_level: 42}
  - {num_requests: 450, concurrency_level: 48}
  - {num_requests: 450, concurrency_level: 54}
  - {num_requests: 450, concurrency_level: 60}
  num_workers: 1
api:
  streaming: true
  type: completion
server:
  type: vllm
  model_name: google/gemma-4-26B-A4B-it
  base_url: http://vllm-gemma4.default.svc.cluster.local
tokenizer:
  pretrained_model_name_or_path: google/gemma-4-26B-A4B
metrics:
  type: default
report:
  request_lifecycle: {summary: true, per_stage: true, per_request: false}
storage:
```

```
  ${IP_STORAGE_TYPE}: {bucket_name: "campos-olp"}
data:
  type: conversation_replay
  conversation_replay:
    num_conversations: 30
    seed: 42
    shared_system_prompt_len: 500
    turns_per_conversation: {type: fixed, mean: 1}
    input_tokens_per_turn: {type: normal, min: 1000, max: 80000, mean: 15000, std_dev: 22000}
    output_tokens_per_turn: {type: normal, min: 50, max: 2000, mean: 500, std_dev: 300}
### File: inference-perf-multimodal.yaml
load:
  type: concurrent
  stages:
    - {num_requests: 400, concurrency_level:   4}
    - {num_requests: 400, concurrency_level:   8}
    - {num_requests: 400, concurrency_level:  20}
    - {num_requests: 400, concurrency_level:  50}
    - {num_requests: 400, concurrency_level: 125}
    - {num_requests: 400, concurrency_level: 312}
    - {num_requests: 400, concurrency_level: 781}
  num_workers: 8
  worker_max_concurrency: 100
  worker_max_tcp_connections: 2500
api:
  type: chat
  streaming: true
server:
  type: vllm
  model_name: google/gemma-4-26B-A4B
  base_url: http://vllm-gemma4.default.svc.cluster.local
  ignore_eos: false
tokenizer:
  pretrained_model_name_or_path: google/gemma-4-26B-A4B
  trust_remote_code: true
metrics:
  type: default
report:
  request_lifecycle: {summary: true, per_stage: true, per_request: false, use_server_
output_tokens: true}
storage:
  ${IP_STORAGE_TYPE}: {bucket_name: "campos-olp"}
data:
  type: synthetic
  input_distribution: {min: 40, max: 600, mean: 240, std_dev: 120, total_count: 2000}
  output_distribution: {min: 100, max: 1024, mean: 400, std_dev: 180, total_count: 2000}
  multimodal:
    image:
      count: {type: fixed, min: 1, max: 1, mean: 1}
      insertion_point: 0.0
      resolutions: [{resolution: "1080p", weight: 1.0}]
# File inference-perf-decode.yaml
load:
  type: concurrent
  stages:
    - {num_requests: 200, concurrency_level:   8}
    - {num_requests: 200, concurrency_level:  20}
    - {num_requests: 200, concurrency_level:  50}
    - {num_requests: 200, concurrency_level: 125}
    - {num_requests: 200, concurrency_level: 312}
    - {num_requests: 200, concurrency_level: 781}
  num_workers: 8
  worker_max_concurrency: 100
  worker_max_tcp_connections: 2500
api:
  type: completion
  streaming: true
server:
  type: vllm
  model_name: google/gemma-4-26B-A4B
  base_url: http://vllm-gemma4.default.svc.cluster.local
```

```
  ignore_eos: true
tokenizer:
  pretrained_model_name_or_path: google/gemma-4-26B-A4B
  trust_remote_code: true
metrics:
  type: default
report:
  request_lifecycle: {summary: true, per_stage: true, per_request: false}
storage:
  ${IP_STORAGE_TYPE}: {bucket_name: "campos-olp"}
data:
  type: random
  input_distribution: {type: normal, min: 100, max: 300, mean: 200, std_dev: 35, total_count: 2000}
  output_distribution: {type: normal, min: 2048, max: 8196, mean: 5120, std_dev: 1024,
total_count: 2000}
### File s3-access-policy.json
{
  "Version": "2012-10-17",
  "Statement": [
    {"Effect": "Allow", "Action": ["s3:ListAllMyBuckets"], "Resource": "*"},
    {"Effect": "Allow", "Action": ["s3:ListBucket"], "Resource": ["arn:aws:s3:::campos-olp"]},
    {"Effect": "Allow", "Action": ["s3:PutObject", "s3:GetObject", "s3:DeleteObject"], "Resource":
["arn:aws:s3:::campos-olp/*"]}
  ]
}
```

This project was commissioned by Google.

Read the report ▶

Primary contributors

Tech: Dan Sullivan

Writing: Laura W.

Design: Jared White

PM: Scott Luchene

How we created this report

A PT team, which includes the contributors we've listed and others, created this report and performed the technical work behind it. We used AI to draft sections of this report.

Principled Technologies®

Facts matter.®

Principled Technologies is a registered trademark of Principled Technologies, Inc. All other product names are the trademarks of their respective owners. For additional information, review the science behind this report.

DISCLAIMER OF WARRANTIES; LIMITATION OF LIABILITY:

Principled Technologies, Inc. has made reasonable efforts to ensure the accuracy and validity of its testing, however, Principled Technologies, Inc. specifically disclaims any warranty, expressed or implied, relating to the test results and analysis, their accuracy, completeness or quality, including any implied warranty of fitness for any particular purpose. All persons or entities relying on the results of any testing do so at their own risk, and agree that Principled Technologies, Inc., its employees and its subcontractors shall have no liability whatsoever from any claim of loss or damage on account of any alleged error or defect in any testing procedure or result.

In no event shall Principled Technologies, Inc. be liable for indirect, special, incidental, or consequential damages in connection with its testing, even if advised of the possibility of such damages. In no event shall Principled Technologies, Inc.'s liability, including for direct damages, exceed the amounts paid in connection with Principled Technologies, Inc.'s testing. Customer's sole and exclusive remedies are as set forth herein.