KV Cache Offloading mit NetApp

NetApp artificial intelligence solutions

NetApp
August 20, 2026

Inhalt

KV Cache Offloading mit NetApp

Die erste Welle von KI-Investitionen in Unternehmen konzentrierte sich auf das Modelltraining und die Feinabstimmung der Leistungsfähigkeit, nicht auf deren flächendeckende Bereitstellung. Mit dem Übergang von der Experimentierphase zur Produktion verlagert sich der Schwerpunkt auf Echtzeit-Inferenz: Suchassistenten, Chatbots, Copiloten und Agenten-Workflows, mit denen Nutzer kontinuierlich interagieren. In diesen Umgebungen steigt die GPU-Auslastung rasant an, nicht weil jede Anfrage einen vollständigen Trainingsdurchlauf erfordert, sondern weil jede Folgefrage, jede neue Gesprächswendung und jeder gleichzeitige Nutzer die Last auf den Inferenz-Stack erhöht.

Einführung

Es wird schnell ein Muster sichtbar: Die GPU führt überraschend viele Wiederholungsoperationen durch und berechnet die Aufmerksamkeit für bereits verarbeitete Token erneut. Diese Wiederholungen sind nicht allein ein Optimierungsproblem, sondern auch ein Problem der Speicherarchitektur. Die Reaktion der Branche ist eine mehrstufige Speicherstrategie, die auf dem KV cache basiert.

Was ist der KV Cache?

Vereinfacht ausgedrückt ist KV (Key-Value) Caching eine Technik, die zur Optimierung der Inferenz großer Sprachmodelle (LLM) verwendet wird, indem zuvor berechnete Werte in einem KV-Cache gespeichert werden, sodass diese Werte nicht für jedes neu generierte Token erneut berechnet werden müssen, was andernfalls notwendig wäre.

Hinweis: Für einen detaillierteren technischen Überblick siehe "Übersicht zu Hugging Face KV Caching".

Was ist KV Cache Offloading?

Die meisten Inferenz-Engines speichern den KV-Cache standardmäßig im GPU-Speicher. Das funktioniert gut, bis der Bedarf steigt. Die Größe des KV-Caches skaliert linear mit der Batchgröße (der Anzahl der gleichzeitig verarbeiteten Prompts) und der Sequenzlänge (der Länge jedes Gesprächs- oder Generierungsstroms). Wenn sich die Kontextfenster vergrößern, mehrstufige Chats häufiger werden und mehr Nutzer und Agenten Inferenzdienste nutzen, kann der Bedarf an KV-Cache schnell den verfügbaren GPU-Speicher übersteigen. In diesem Fall werden oft nützliche Cache-Einträge verdrängt und die Engine muss die Aufmerksamkeit für bereits verarbeitete Token erneut berechnen.

Die Auslagerung des KV-Caches behebt diese Einschränkung, indem der KV-Cache einer zuvor verarbeiteten Eingabeaufforderung in ein externes Ziel außerhalb des GPU- oder Beschleunigerspeichers, wie System-RAM, lokale Festplatte oder gemeinsam genutzten Speicher, verschoben wird. Ausgelagerte Einträge können so lange gespeichert werden, wie es die Kapazität zulässt, und bei ähnlichen Präfixen oder Folgeanfragen erneut referenziert werden, anstatt bei vollem GPU-Speicher verworfen zu werden. Diese Auslagerungsziele fungieren als gestaffelter Auslagerungsspeicher für den GPU-Speicher: langsamer als VRAM, aber größer und langlebiger, wodurch mehr Konversationskontext und gleichzeitige Arbeitslast vom System verarbeitet werden können, ohne dass wiederholtes Vorbefüllen erforderlich ist.

Bedeutung einer gemeinsamen Speicherebene

Die Auslagerung des KV-Caches ist bereits hilfreich, wenn der GPU-Speicher einer einzelnen Inferenz-Engine nicht ausreicht. Das Verschieben von Cache-Blöcken in den CPU-RAM erweitert die Kapazität auf einem Server. Das ist zwar nützlich, löst aber nur einen Teil des Produktionsproblems. Reale Inferenzplattformen laufen selten als einzelne Instanz einer Inferenz-Engine. Sie laufen hinter Load Balancern, skalieren horizontal und bedienen viele gleichzeitige Benutzer und Agenten. In dieser Umgebung ist es der gemeinsam genutzte Speicher, der den KV-Cache von einer lokalen Optimierung zu einer Plattformfähigkeit macht.

- **Gemeinsamer Speicher ermöglicht die Wiederverwendung über mehrere Serverinstanzen hinweg** Einer der Hauptvorteile von gemeinsamem Speicher ist die Möglichkeit, auf dieselben Dateien oder Objekte über mehrere Instanzen und Knoten hinweg zuzugreifen. Dies gilt insbesondere bei einer skalierten Bereitstellung von zwei oder mehr Serverinstanzen hinter einem Load Balancer, die jeweils so konfiguriert sind, dass sie KV-Cache-Blöcke in denselben gemeinsamen Bucket oder dasselbe NAS-Volume auslagern. Wenn beispielsweise eine Instanz ein Prompt-Präfix verarbeitet und die resultierenden Cache-Blöcke auslagert, kann eine andere Instanz diese Blöcke bei einem Cache-Treffer laden, anstatt die gleiche Vorbefüllung erneut zu berechnen. Dies ist wichtig, da der Produktionsdatenverkehr verteilt ist, die erste Frage eines Benutzers kann Instanz A erreichen, eine Folgefrage oder ein anderer Benutzer mit einem ähnlichen Systemprompt hingegen Instanz B. Ohne gemeinsamen Speicher verwaltet jede Instanz ihren eigenen privaten Cache.
- **Für Multi-Instanz-Bereitstellungen reicht der CPU-Speicher allein nicht aus.** Die CPU-Ebene ist zwar schnell, aber sie ist lokal auf jeden Serving-Engine-Prozess beschränkt. Eine Instanz kann nicht auf KV-Cache-Einträge zugreifen, die eine andere Instanz in ihren lokalen CPU-RAM ausgelagert hat, es sei denn, ein Peer-to-Peer-Kanal wird implementiert, was jedoch zusätzliche Latenz und operative Komplexität einführt. Gemeinsamer Speicher beseitigt diese Hürde. Der CPU-RAM bleibt als schnelle Zwischenspeicherebene auf jedem Knoten wertvoll, aber gemeinsam genutztes S3 oder NAS bietet die gemeinsame Speicherebene, auf die mehrere Engines lesen und schreiben können. GPUs werden nicht mehr isoliert skaliert, sondern mit gemeinsam genutzter Cache-Infrastruktur.
- **Ermöglichung einer dreistufigen Designstrategie** + Eine bevorzugte Architektur kombiniert:

  - **GPU-Speicher** – aktiver Cache für laufende Anfragen.
  - **CPU-Speicher** – schnelle Bereitstellung, wenn Eingabeaufforderungen in die Warteschlange gelangen.
  - **Gemeinsamer Speicher** – dauerhafter, großer, gemeinsam nutzbarer Backing Store.

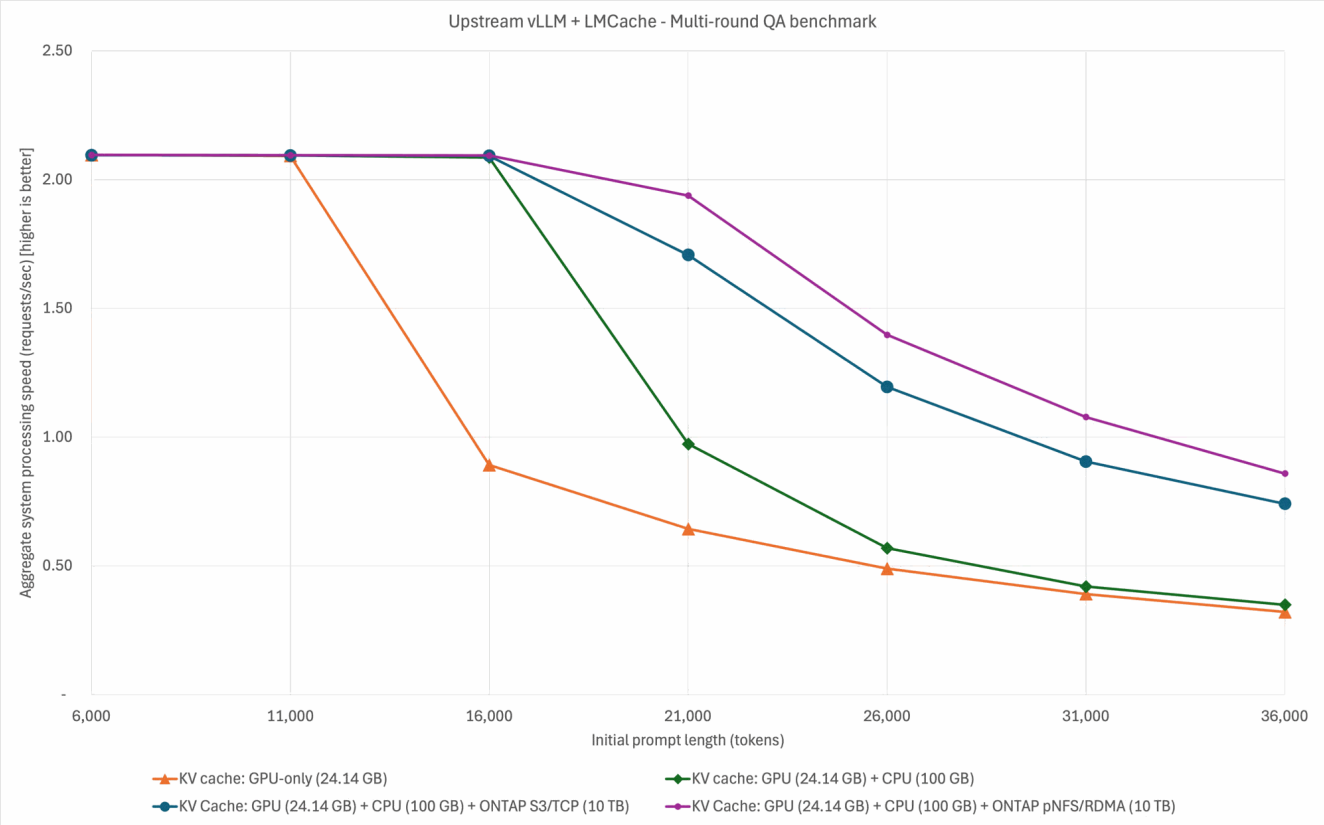
  Mit dieser Architektur können KV-Cache-Blöcke vom gemeinsam genutzten Tier in den CPU-Speicher verschoben werden, sobald neue Anfragen eintreffen, sodass die GPU auf aktive Arbeit fokussiert bleibt, während gleichzeitig ein dauerhafter Cache auf dem Storage erhalten bleibt.

- **Ökonomie der Inferenz, nicht nur Geschwindigkeit** Aus Produktionssicht ist die Bedeutung von gemeinsam genutztem Speicher nicht nur die Latenz. Es ist die Kontrolle über Speicher, Parallelität und Kosten. Serving-Engines wie vLLM verwalten den KV-Cache effizient innerhalb eines Knotens. KV-Cache-Offload-Frameworks wie LMCache machen den KV-Cache zu einer portablen, gemeinsam nutzbaren Ressource, die zwischen CPU-Speicher und Speicher verschoben werden kann. Plattformen für gemeinsam genutzten Speicher wie NetApp ONTAP und StorageGRID bieten die dauerhafte Ebene, die diese gemeinsame Nutzung über mehrere Instanzen hinweg praktisch macht. Mit LMCache, das an gemeinsam genutzten Speicher angebunden ist, können mehrere vLLM-Instanzen auf dieselben ausgelagerten KV-Cache-Einträge zugreifen, wodurch der Durchsatz verbessert und redundante Berechnungen bei wachsender Parallelität reduziert werden. So werden verschwendete GPU-Zyklen durch wiederholtes Vorbefüllen reduziert, was direkt relevant ist für Chatbots, Suchassistenten, Agenten und alle Workloads mit Multi-Turn-Threads, gemeinsam genutzten System-Prompts oder wiederkehrenden Kontextmustern.
- **Steigerung der Inferenzleistung und Optimierung der GPU-Investition** + Dieser Benchmark vergleicht

die Inferenzleistung bei zunehmender Konversationslänge und gleichzeitiger Last. Wird der KV-Cache ausschließlich im GPU-Speicher gehalten, ist der verfügbare Spielraum schnell erschöpft und der Durchsatz sinkt (orange Linie). Mit einem dreistufigen Design (GPU für aktive Verarbeitung, CPU für schnelles Staging und gemeinsam genutzter Speicher für einen dauerhaften, gemeinsam nutzbaren Cache) bleibt die Plattform in der Lage, mehr Sitzungen zu unterstützen und denselben starken Leistungsabfall zu vermeiden. In der Praxis ermöglicht die Stufenaufteilung, mehr Arbeitslast mit denselben GPUs zu bewältigen, da wiederholte Vorberechnungen reduziert werden.

  - **Einfach ausgedrückt:**
    - **GPU-Ebene** – verarbeitet aktive, laufende Aufgaben.
    - **CPU-Ebene** hält den zuletzt verwendeten Cache in der Nähe der Server-Engine.
    - **Gemeinsame Speicherebene** – speichert den Cache serverübergreifend und gibt GPU-/CPU-Speicher für neue Sitzungen frei.

Abbildung 1: Benchmark für mehrrundige Inferenz



Der Benchmark zeigt, dass die Hinzunahme von gemeinsam genutztem Speicher neben dem CPU-Speicher die Leistung nicht verringert, sondern den nutzbaren Betriebsbereich erweitert, bevor Durchsatzeinbußen auftreten. Tiering fügt effektiv Speicherebenen für die Inferenz hinzu und reduziert den Bedarf, bei steigender Anzahl von Sitzungen, Kontextlänge oder Parallelität GPUs hinzuzufügen.

- **Unternehmenstauglich: Standardprotokolle, kein proprietärer Client** + Gemeinsamer Speicher ist wichtig, aber nicht jeder gemeinsame Speicher ist gleich. NetApp unterstützt das KV-Cache-Offloading mit branchenüblichen Protokollen und Tools:
  - **StorageGRID S3**
  - **NFS / pNFS für ONTAP NAS**

Es ist kein proprietärer, kundenspezifischer Inferenzclient erforderlich. Betriebsteams können dieselben Speicherprotokolle verwenden, die sie bereits für andere Datendienste einsetzen, mit vertrautem Datenschutz, Sicherheit und Multicloud-Mobilität im Hintergrund.

KV-Cache-Offloading implementieren

Gemeinsam genutzter Speicher erweitert die KV-Cache-Kapazität und ermöglicht die Wiederverwendung über mehrere Serving-Instanzen hinweg. Für die Nutzung dieser Ebene in der Produktion werden eine Inferenz-Engine und ein Framework zur Auslagerung des KV-Caches konfiguriert. Die folgenden Abschnitte beschreiben die Implementierung dieses Stacks mit gängigen Tooling-Optionen.

vLLM

vLLM ist eine beliebte, leistungsstarke Open-Source-LLM-Inferenz-Engine. In dieser Lösung ist es die Engine, die Benutzeranfragen empfängt, Antworten generiert und den KV-Cache im GPU-Speicher während der Inferenz verwaltet. vLLM ist modular aufgebaut – es kann ausgewählt werden, welches Offloading-Framework verwendet werden soll. Die folgenden Abschnitte beschreiben, wie ein vLLM-basierter Serving Stack mit gängigen Offloading-Frameworks implementiert werden kann.

LMCache (In-Process-Modus)

LMCache ist ein weit verbreitetes Open-Source-Framework zur Auslagerung des KV-Cache. In diesem Abschnitt wird beschrieben, wie ein vLLM-basierter Serving-Stack implementiert wird, der LMCache für die KV-Cache-Auslagerung verwendet. In dieser Implementierung arbeitet LMCache mit vLLM zusammen, um den KV-Cache aus dem GPU-Speicher in größere Ebenen wie CPU-RAM, lokale Festplatte oder gemeinsam genutzten Speicher (wie StorageGRID S3 oder ein ONTAP NAS-Mount) zu verschieben.

Standardmäßig speichert vLLM den KV-Cache ausschließlich im GPU-Speicher. Mit zunehmender Systemlast wird dies zum Engpass. In dieser Lösung wird vLLM mit LMCache kombiniert, wobei vLLM das Modell bereitstellt, während LMCache den Cache auf CPU und gemeinsam genutztem NetApp Speicher auslagert und wiederverwendet. LMCache verbindet sich über einen Konnektor mit vLLM; die Aktivierung erfolgt beim Start mit dem vLLM-Flag --kv-transfer-config, sodass vLLM und LMCache den Cache im Speicher ablegen und ihn bei einem Cache-Treffer erneut laden.

Der In-Process-Modus ist die ursprüngliche Methode zum Ausführen von LMCache mit vLLM. Im In-Process-Modus läuft LMCache innerhalb des vLLM-Prozesses. Spätere LMCache-Versionen führten den Multiprocess-Modus ein, der aktuell für eine bessere Funktionsunterstützung und höhere Leistung empfohlen wird. Dieser Abschnitt behandelt den In-Process-Modus, der in der aktuellen LMCache-Dokumentation als veraltet gekennzeichnet ist. Für neue Installationen wird stattdessen die Verwendung des Multiprocess-Modus empfohlen.

Für diese Bereitstellung sind folgende Schritte vorgesehen:

- vLLM zusammen mit LMCache installieren
- vLLM mit aktiviertem LMCache Connector starten
- Ein dreistufiges Setup (GPU + CPU + gemeinsamer Speicher) mit zwei vLLM Instanzen (auf separaten GPUs) hinter einem vLLM Router, wobei beide dasselbe gemeinsame Speicher-Backend verwenden.

Voraussetzungen

- Linux GPU Server mit NVIDIA-Treibern (einschließlich CUDA) und mindestens einer GPU (zwei GPUs oder mehrere Server für das vollständige Zwei-Instanzen- und Router-Layout)

- Python und uv installiert
- Docker und NVIDIA Container Toolkit installiert (erforderlich für den vLLM Router in Schritt 4)
- Netzwerkzugriff zum Herunterladen des Modells (zum Beispiel Hugging Face) und zum Erreichen Ihres Speicherendpunkts

StorageGRID S3 Backend

- S3-Bucket für die Nutzung des KV-Cache erstellt
- Lese-/Schreibberechtigungen für den Bucket
- Netzwerkverbindung vom GPU-Server zum S3-Endpunkt
- Virtuelle S3-Anfragen im gehosteten Stil aktiviert

ONTAP pNFS/NFS Backend

- ONTAP Volume für den/die GPU-Server exportiert
- Mount-Pfad (zum Beispiel /mnt/kvcache) auf **jedem** Server, auf dem vLLM ausgeführt wird, erstellt. Beispiel für einen Mount-Befehl für hochperformantes pNFS über RDMA (RoCE):

```
mount -o
vers=4.1,proto=rdma,trunkdiscovery,write=eager,rsize=262144,wsize=262144
my_storage.my_company.net:/kvcache /mnt/kvcache
```

[[1-install-vllm-and-lmcache]]
=== 1. vLLM und LMCache installieren

Erstellen Sie eine virtuelle Umgebung und installieren Sie vLLM und LMCache. Weitere Informationen zur LMCache "Installationsdokumentation" finden Sie in der Dokumentation. Es ist entscheidend, dem korrekten Installationspfad für Ihre CUDA-Version zu folgen.

```
uv venv .venv
source .venv/bin/activate
uv pip install --upgrade lmcache vllm
```

An diesem Punkt verfügen Sie über die Inferenz-Engine (vLLM) und die Cache-Schicht (LMCache).

[[2-configure-lmcache]]
=== 2. LMCache konfigurieren

vLLM lagert den KV-Cache nicht automatisch aus. LMCache wird über den KV transfer connector von vLLM aktiviert. Eine YAML-Datei (lmcache-config.yaml) definiert die Konfiguration der CPU- und Shared-Storage-Ebene. LMCache liest diese YAML-Datei als Teil der vLLM-Inferenz-Startsequenz.

Beispielausschnitt: StorageGRID S3 gemeinsam genutzte Ebene

```
local_cpu: True
max_local_cpu_size: 100
```

```
save_decode_cache: True
remote_url: "s3://<bucket>.<storagegrid_s3_endpoint_fqdn>"
remote_serde: "naive"
extra_config:
  s3_num_io_threads: 320
  s3_prefer_http2: False
  s3_enable_s3express: False
  save_chunk_meta: False
  disable_tls: <True/False>
  aws_access_key_id: "<storagegrid_s3_access_key>"
  aws_secret_access_key: "<storagegrid_s3_secret_key>"
  s3_region: "us-east-1"
```

Hinweis: Ersetzen Sie Bucket-Name, Endpunkt, Anmeldeinformationen und `disable_tls` entsprechend Ihrer Umgebung.

Beispielausschnitt: ONTAP pNFS/NAS Shared Tier

```
local_cpu: true
max_local_cpu_size: 100
save_decode_cache: true
remote_storage_plugins: ["fs"]
extra_config:
  remote_storage_plugin.fs.base_path: /mnt/kvcache
  save_chunk_meta: false
```

Der ONTAP-Export ist vor Schritt 3 mithilfe des NFS/pNFS-Verfahrens Ihrer Website bereitzustellen.

[[3-run-two-vllm-instances-shared-cache-across-servers]]
=== 3. Zwei vLLM Instanzen ausführen (gemeinsamer Cache über mehrere Server hinweg)

Gemeinsame Umgebungsvariablen auf beiden Instanzen festlegen:

```
export LMCACHE_CONFIG_FILE=$(pwd)/lmcache-config.yaml
export PYTHONHASHSEED=0
export VLLM_API_KEY='<shared_api_key>'
```

Instanz 1 (GPU 0, Port 8001):

```
export CUDA_VISIBLE_DEVICES=0
vllm serve Qwen/Qwen3-8B \
  --tensor-parallel-size 1 \
  --kv-transfer-config
'{"kv_connector":"LMCacheConnectorV1","kv_role":"kv_both"}' \
```

```
  --port 8001
```

Instanz 2 (GPU 1, Port 8002, separates Terminal):

```
export CUDA_VISIBLE_DEVICES=1
vllm serve Qwen/Qwen3-8B \
  --tensor-parallel-size 1 \
  --kv-transfer-config
'{"kv_connector":"LMCacheConnectorV1","kv_role":"kv_both"}' \
  --port 8002
```

Verwenden Sie dieselbe Konfigurationsdatei und denselben Hash-Seed, damit beide Instanzen die Cache-Block-Identität kennen und die jeweils ausgelagerten Daten aus dem gemeinsamen Speicher lesen können. Setzen Sie `VLLM_API_KEY` auf beiden Instanzen; der Router übergibt diesen Wert als `OPENAI_API_KEY`, sodass weitergeleitete Anfragen von den Instanzen akzeptiert werden.

Hinweis: Eine einzelne GPU-Instanz kann auch zur Implementierung einer mehrstufigen Speicherarchitektur verwendet werden. In einem solchen Fall wird Schritt 4 übersprungen.

[[4-deploy-the-vllm-router-single-entry-point]]
=== 4. Bereitstellung des vLLM-Routers (einzelner Zugangspunkt)

Nachdem beide Instanzen fehlerfrei funktionieren, kann ein Router bereitgestellt werden, sodass die Clients eine OpenAI-kompatible URL verwenden.

```
docker run -d --rm --name vllm-router \
  --network host \
  -e OPENAI_API_KEY="${VLLM_API_KEY}" \
  ghcr.io/vllm-project/production-stack/router:latest \
  --port 8000 \
  --service-discovery static \
  --static-backends "http://localhost:8001,http://localhost:8002" \
  --static-models "Qwen/Qwen3-8B,Qwen/Qwen3-8B" \
  --routing-logic roundrobin
```

Senden Sie den Datenverkehr an http://localhost:8000/v1. Der Router verteilt die Anfragen; LMCache und gemeinsam genutzter Speicher ermöglichen die Wiederverwendung des Caches über mehrere Instanzen hinweg, nicht nur innerhalb einer einzelnen GPU.

Beispielsweise kann mit curl eine Anfrage an den Router gesendet werden, wie folgt (ersetzen Sie <shared_api_key> durch Ihren jeweiligen API-Schlüssel):

```
curl -s http://localhost:8000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H 'Authorization: Bearer <shared_api_key>' \
  -d '{
```

```
    "model": "Qwen/Qwen3-8B",
    "messages": [{"role": "user", "content": "What is 2+2?"}]
  }' | jq .
```

[[5-validate-that-tiering-is-working]]
=== 5. Es wird überprüft, ob das Tiering funktioniert.

- Beide vLLM Prozesse lauschen auf den Ports 8001 und 8002, der Router antwortet auf Port 8000.
- Eine Anfrage mit einem langen Präfix wird über den Router gesendet.
- Bestätigen Sie, dass Objekte oder Dateien im gemeinsam genutzten Speicher (S3-Bucket oder ls -ltr /mnt/kvcache) angezeigt werden.
- Eine ähnliche Anfrage erneut senden, die zweite Anfrage sollte in den Protokollen ein schnelleres Vorbefüllungs- oder Cache-Treffer-Verhalten zeigen.
- Wiederholen Sie dies über den Router, damit der Datenverkehr die andere Instanz erreichen kann.

Abschluss

Durch den Einsatz einer mehrstufigen KV-Cache-Architektur wird der KV-Cache vom GPU-Speicher in den CPU-Speicher und auf gemeinsam genutzten NetApp Speicher verlagert, sodass die Inferenz mit der Benutzerzahl und der Kontextlänge skalieren kann, ohne GPU-Zyklen durch wiederholtes Vorbefüllen zu verschwenden. Gemeinsam genutzter Speicher macht den Cache zu einer wiederverwendbaren Infrastruktur über verschiedene Serverinstanzen hinweg und ermöglicht es, mehr Wert aus der bestehenden GPU-Investition zu ziehen.

NetApp Storage ist eine ausgezeichnete Wahl für diese Ebene, da es über die Bruttokapazität hinaus genau das bietet, was für produktive Inferenz erforderlich ist: Standardzugriff über S3 und NFS/pNFS, einen gemeinsam genutzten Cache, den mehrere Serving-Engine-Instanzen gleichzeitig nutzen können, sowie Enterprise Data Services, auf die Sie bereits für Haltbarkeit, Schutz und Governance vertrauen. Ein proprietärer Client ist nicht erforderlich; KV-Cache-Offloading-Frameworks verbinden sich über vertraute Protokolle entweder mit StorageGRID S3 oder einem ONTAP NAS-Mount.

NetApp bietet Ihnen eine vertraute Speichergrundlage für das KV-Cache-Offloading, ohne dass sich daran etwas ändert, wie Inferenzprozesse ablaufen oder wie Ihre Teams Daten verwalten.