PyTorch CONFERENCE EUROPE 2026

Bringing Google's Colossus to PyTorch

Rapid storage via fsspec to keep GPUs busy

Speakers Intro

Ankita Luthra

Senior Software Engineer, Google

Ankita Luthra is a Software Developer at Google, focused on AI/ML infrastructure and scalable data pipelines. Her work with open-source tools like fsspec (gcsfs) and gcsfuse improves how frameworks such as PyTorch/JAX efficiently access data from Google Cloud Storage.

Trinadh Kotturu

Senior Product Manager, Google

Trinadh Kotturu is a Senior Product Manager specializing in AI/ML and analytics client strategy at Google. An alumnus of IIM Bangalore with 12 years of experience, he has a proven track record of shipping v1 products and scaling them into robust platform services. His expertise spans large-scale distributed storage systems, autonomous driving, and system resiliency.

The Storage Tax: Don't Let I/O Starve Your GPUs

Slow data/model loading, checkpointing wastes expensive accelerator time

Llama 4 Scout (17Bx16E)

Training Data & Modality

40T tokens

Compared to 15T tokens for Llama 3.3 (70B)

Checkpoint Size

1.4TB

With 12 bytes per parameter. Faster checkpoints for resiliency.

Context Length

10M token

Loading 128B params: 250GB (BF16) or ~64GB (INT4)

Why Storage Architecture Matters

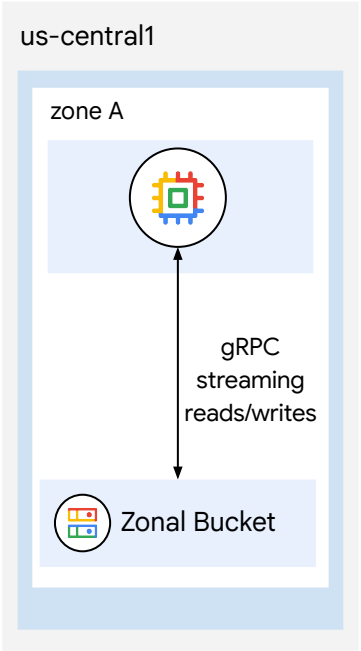
The Throughput Gap: 8-GPU nodes can hit 20 GB/s NIC bandwidth. Multi-node clusters reach TB/s requirements.

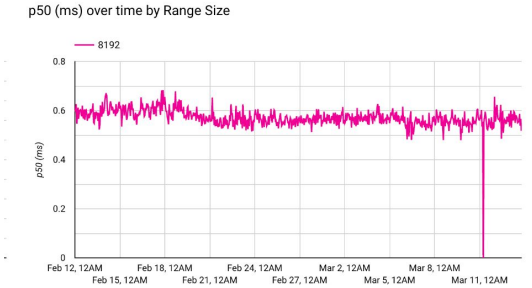
Latency & Agentic AI: KV data outgrows HBM, forcing storage offloads (the "Memory Wall"). Retrieval directly impacts TimeToFirstToken (TTFT).

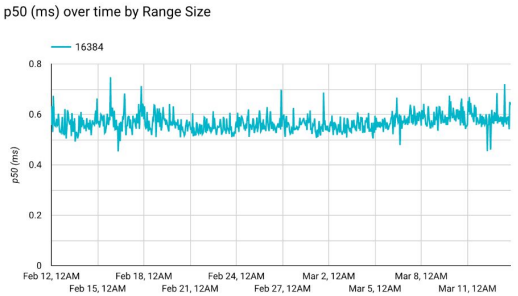
Rapid Storage

High performance object storage in a dedicated zonal bucket built on top of Google's colossus

- Colocated zonal bucket provides up to **15 TiB/s throughput** and **20M QPS**
- **<1 ms latency** for random reads and append writes.
- **Optimized for AIML workloads** with appendable objects + native Hierarchical Namespace





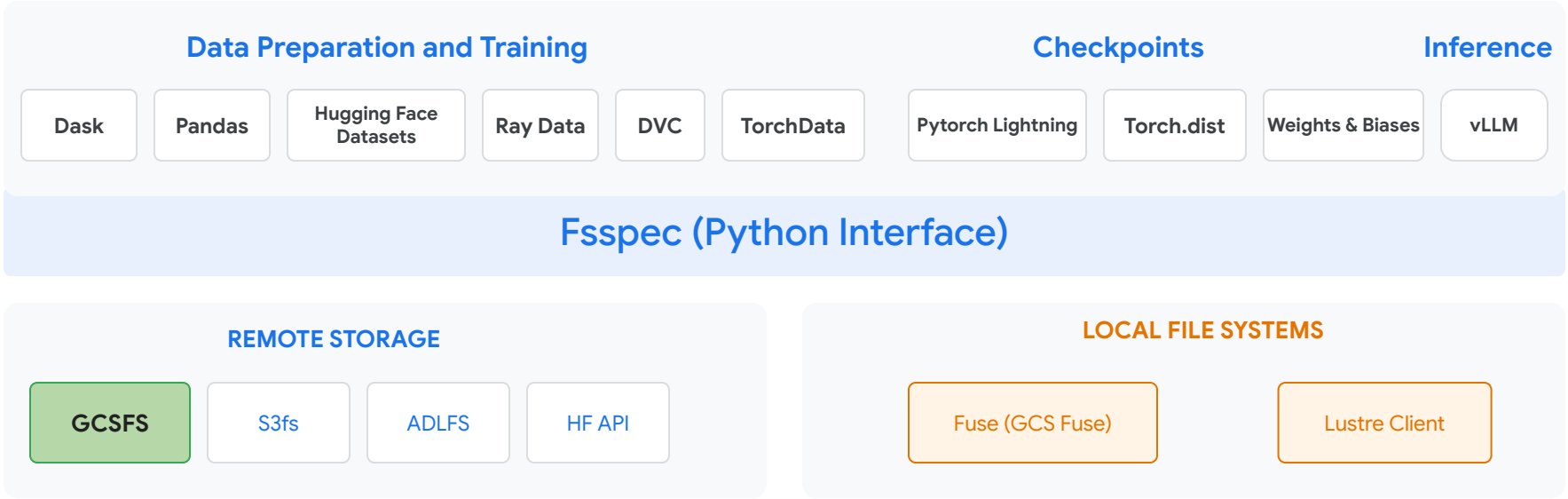


Ultra low latency for zonally colocated workloads

sub-ms 8k random reads & sub-ms 16k append writes @ p50

Fsspec is pervasive in PyTorch

Industry standard for Pythonic file interface



fsspec.open() Lifecycle

1. "Dispatch" filesystem by looking at registry (parse prefix)
2. Instantiate the filesystem (e.g., gcsfs)
3. Instantiate the file handle

Integrating Rapid Storage with GCSFS

Bypassing REST APIs with persistent gRPC streams for high-performance PyTorch workloads

4.8x
faster reads
(sequential and random)

2.8x
faster writes

Microbenchmark results

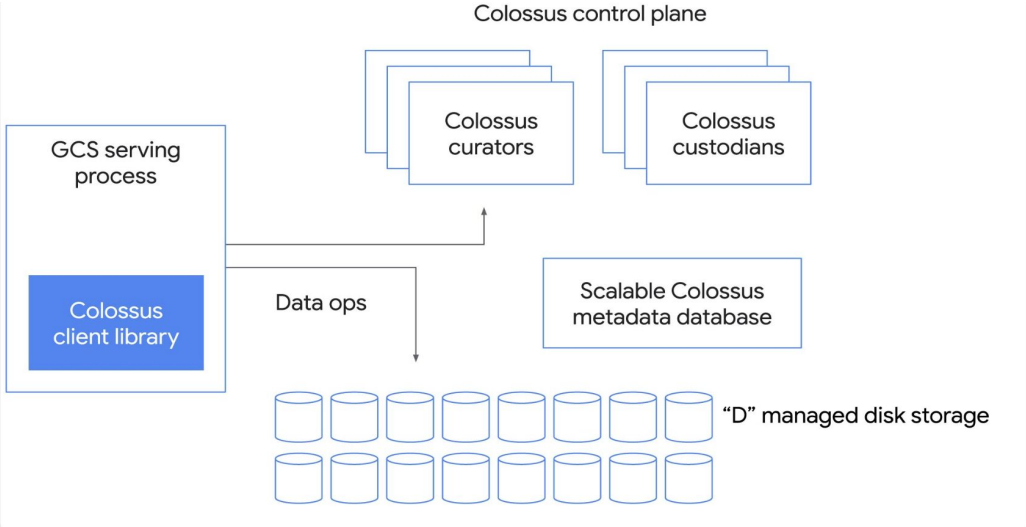
Try it yourself

Behind the scenes: Google Colossus

Google's cluster-level file system and the next-generation successor to the original Google File System (GFS).

Colossus

- Massive Scale & Impact
- Distributed Metadata
- Intelligent Storage
- Availability
- Durability



Behind the scenes: Standard Buckets

High Level Internals of Standard Storage

Simple + Efficient

01 Stateless Http connections

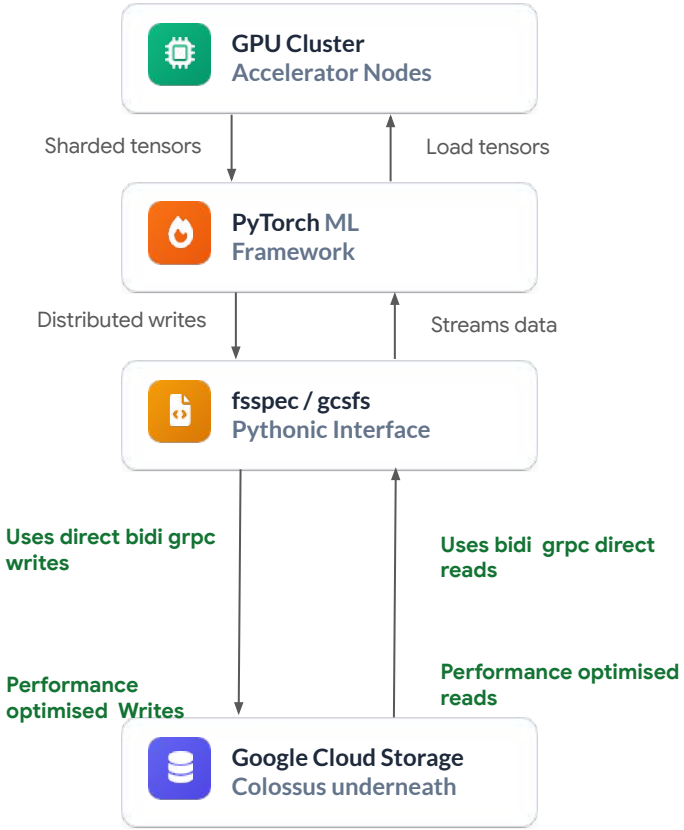
02 Efficient Storage

03 Sharded format

04 Multiple Network Hops

Behind the scenes: Rapid Buckets

High Level Internals of Rapid Storage



Performance Optimized for GPU intensive workloads

01 Direct Connectivity

02 Stateful grpc based streaming APIs

03 Zonal Co-location

04 Optimised On-disk format

05 Hierarchical Support

Rapid Buckets integration with fsspec/gcsfs

fsspec/gcsfs >=2026.3.0

New Features Added in GCSFS

- Google supported sdk for Rapid & HNS
- Stateful Read Handles
- Bucket Type auto detection
- Folder apis Support
- Atomic Renames
- GCS Native Appends

75	+30K	-6K
Total Commits	Lines Added	Lines Deleted

*Shoutout to fsspec maintainer **Martin Durant** from **Anaconda Inc** for collaborating with us in integrating Rapid Storage into fsspec*

Microbenchmarking results are added on gcsfs github: https://github.com/fsspec/gcsfs/blob/main/docs/source/rapid_storage_support.rst

Rapid Buckets With Pytorch

How to use Rapid Buckets in Pytorch workloads

Standard Buckets

```
# 1. Load Data (pseudo code)
ds = datasets.load_dataset("parquet",
   data_files=f"gs://my-standard-bucket/*.parquet",
   split..., streaming=True)
train_loader = DataLoader(dataset,
   batch_size..., collate_fn...,
   workers..., keep_workers_alive=True)


# 2. Configure Trainer
trainer = pl.Trainer(**trainer_config)


# 3. Train Model
trainer.fit(LlamaLitModel(model), train_loader,
   ckpt_path="gs://my-standard-bucket/ckpts")
```

Rapid Buckets

No-op Migration

```
# 1. Load Data (pseudo code)
ds = datasets.load_dataset("parquet",
   data_files=f"gs://my-rapid-bucket/*.parquet",
   split..., streaming=True)
train_loader = DataLoader(dataset,
   batch_size..., collate_fn...,
   workers..., keep_workers_alive=True)


# 2. Configure Trainer
trainer = pl.Trainer(**trainer_config)


# 3. Train Model
trainer.fit(LlamaLitModel(model), train_loader,
   ckpt_path="gs://my-rapid-bucket/checkpoints")
```

Try GCSFS optimised with Rapid Bucket for performance boost

https://docs.cloud.google.com/storage/docs/rapid/rapid-bucket
https://github.com/fsspec/gcsfs