DATA ON KUBERNETES - DATA ANALYTICS AND AI/ML WORKLOADS

Version 1
July 2026

Authors

Alexa Griffith
Chunxu Tang
Joe Huang
Lu Qiu
Raghu Shankar
Robert Hodges
Shawn Sun
Victor Lu
Wannes Rosiers
Xing Yang

TAG INFRASTRUCTURE

OVERVIEW

The Cloud Native AI whitepaper[1] delves into the interaction of Cloud Native (CN) and Artificial Intelligence (AI) technologies, discusses the current state, the challenges, the opportunities, and the potential solutions in this area.

In this paper, we aim at highlighting the characteristics of data analytics and AI/ML workloads and the patterns and trends in data storage to meet the new challenges.

There are different stages of AI/ML with different utilization patterns. The batch, training/fine-tuning workloads have sustained utilization of resources and are long-running jobs. On the other hand, the inference workloads have spiky utilization of resources and need immediate access[2].

Different stages of AI/ML workloads have different requirements for data storage.

Data analytics and AI/ML workloads usually contain a large amount of structured and unstructured data. These workloads need to be highly scalable, highly performant with low latency. They typically need read-only access to data. These characteristics have implications for data storage requirements.

Data warehouses and data lakes are typically used to store large data for analytics workloads. Data warehouses are optimized for Online Analytical Processing (OLAP) as compared to traditional relational databases which are used for Online Transaction Processing (OLTP). Column-oriented databases can be used in data warehouses for efficient query search[3]. Data warehouses are normally utilized to store structured data. Data lakes, on the other hand, can handle both structured and unstructured data.

Data caching plays an important role in data analytics and AI/ML workloads. It can help improve performance dramatically by placing data close to where it needs to be accessed and processed. It can also help to avoid redundant computation.

In recent years, vector databases have gained popularity due to their capability to do similarity search rather than the exact match used in traditional database searches. This is very important for AI/ML workloads, especially generative AI workloads.

Block, file, and object storage can be used as the underlying storage systems to store data for data analytics and AI/ML workloads. Object storage's ability to allow data to be shared with multiple workloads simultaneously, optimized throughput for parallelised workloads, and highly scalable capacities make it a popular choice for AI/ML workloads as well as data analytics.

Data warehouses and data lakes are centralized repositories. There are trends to decentralize them using the data mesh architecture for building AI/ML models. Another data paradigm is data fabric, an architectural approach and technology framework that addresses data lake challenges by providing a unified and integrated view of data across various sources[4].

Modern Data Architecture Principles emphasize on data quality, data governance, consistency, data as a shareable asset, and data security and privacy.

Patterns and Trends in Data Storage

- Data Warehouses, Data Lakes, and Data Lake Houses
- Data Cache and Data Locality
- Databases
- Block, File, and Object Storage
- Network Management
- Data Pipelines
- Data Mesh and Data Fabric
- Hardware/Software Co-design

Storage in the AI Lifecycle

Different phases of AI workloads have distinct storage requirements and usage patterns. Training workloads are throughput-oriented with sustained data movement, inference workloads are latency-sensitive with spiky traffic patterns, and AI agents require complex state management across iterative reasoning loops.

- Training and its Storage Usage Patterns
- Inference and its Storage Usage Patterns
- AI Agent and its Storage Usage Patterns

References

1. https://www.cncf.io/wp-content/uploads/2024/03/cloud_native_ai24_031424a-2.pdf
2. https://sched.co/1YhIO
3. Designing Data Intensive Applications, Martin Kleppmann, O'Reilly Media, 2 May 2017
4. From data warehouse to data fabric: the evolution of data architecture, CNCF Blog, 21 July 2023

DATA WAREHOUSES, DATA LAKES, AND DATA LAKE HOUSES

Data warehouses typically store structured data from transaction processing systems and other business applications. In most cases, the data is cleansed and curated before going into a data warehouse. For data warehouses, data is traditionally stored in relational databases using conventional disk storage, often using a columnar format that stores data as compressed arrays. Modern data warehouses generally use shared object storage to handle large datasets.

Data lakes can handle a combination of structured, semi-structured and unstructured data, which is often stored in its native format to make the full sets of raw data available for analysis. Data lakes are primarily used for data science applications that involve machine learning, predictive modeling and other advanced analytics techniques. Data is typically stored in NoSQL databases such as Cassandra, object storage, or distributed file systems such as Hadoop, etc[1].

Data lakes and data warehouses are merging for many users. A data lake house is a hybrid model of data warehouses and data lakes[2]. It can support both structured and unstructured data and has the characteristics of both data warehouses and data lakes. For example, it is common to store information in open file formats like Apache Parquet[3] that offer columnar format, compression, and indexes to facilitate fast query. Data lake houses also utilize open table formats like Apache Iceberg[4] that organize files into tabular representation complete with features like schema management and transactions. Data lakes enable safe, shared access to data from many types of analytic applications–principally data warehouses, AI/ML, and data science.

References

1. Data lake vs. data warehouse: Key differences explained
2. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics, CIDR 2021
3. Apache Parquet
4. Apache Iceberg

DATA CACHE AND DATA LOCALITY

Data Cache

With the increasing popularity of data lakes and compute-storage disaggregation, the compute tier is now decoupled from the storage tier in both data analytics and AI/ML applications. Data caching plays a crucial role in accelerating data loading, minimizing data transfer between compute and storage layers, and reducing API calls, an often overlooked cost in object storage access. Data access patterns differ slightly between data analytics and AI/ML workloads.

Modern query optimization strategies, such as dynamic filtering on columnar data files, typically result in small, disparate reads rather than sequentially reading large chunks of data. Conversely, AI/ML workloads lean towards hybrid data access patterns, where random and sequential reads coexist based on the specific AI/ML scenario.

To fit the needs of varying workloads, the data cache needs to be scalable and elastic. A hierarchical caching design, with a local cache on each compute node and a distributed cache between compute and storage layers, can be effective for handling high volumes of data in data and AI applications. Integrating such a unified data cache into different components of the machine learning life cycle, including data preprocessing, feature engineering, model training, and model serving, can potentially improve the overall system performance[1].

Data Locality

In the realm of data processing and machine learning, data locality is a crucial factor that significantly impacts performance and cost. The cost considerations include not only data transfer expenses but also the hidden costs associated with low CPU/GPU utilization rates.

Several strategies can be used to achieve data locality, with each offering advantages and challenges:

READING DATA DIRECTLY FROM REMOTE STORAGE

- **Benefits:** This approach requires minimal setup effort.
- **Challenges:** Every training epoch requires re-reading all data from remote storage. Since multiple epochs are often necessary for better accuracy, this method can lead to significant time spent on data loading rather than training.

COPYING DATA TO LOCAL STORAGE BEFORE TRAINING

- **Benefits:** This method ensures that all data is local, thus gaining all the benefits of data locality.
- **Challenges:** Management can be difficult. Users must manually delete training data after use, and cache space is limited. For very large datasets, the benefits can be limited by local storage capacity.

LOCAL CACHE LAYER FOR DATA REUSE

- **Examples:** Tools like S3FS-FUSE with built-in local cache and Alluxio FUSE (Filesystem in Userspace interface).

- **Benefits:** Reused data remains local, and the cache layer handles data management, eliminating the need for manual supervision.
- **Challenges:** Cache space remains limited, thus for large datasets, the benefits might still be constrained.

DISTRIBUTED CACHE LAYER

- **Benefits:** A distributed cache system ensures that data is either local or adjacent, offers centralized data management, and provides scalable cache space.
- **Challenges:** Building and maintaining a distributed caching system can be complex and resource-intensive.

Data locality offers significant performance gains and cost savings, particularly for data-intensive applications and machine learning workloads. However, as discussed above, each strategy comes with its own set of trade-offs. Users must carefully consider their specific needs and constraints when selecting an approach to maximize the benefits of data locality.

A notable solution for addressing data locality issues is Fluid[2], a CNCF project. Fluid offers significant performance gains and cost savings, particularly for machine learning workloads. It enables connecting to remote storage and supports local and/or distributed caching using Kubernetes-native approaches, significantly simplifying data locality management and accelerating AI workloads. Many of these benefits are enabled by Fluid runtimes, such as Alluxio, an open-source data orchestration and distributed caching system.

References

1. https://www.alluxio.io/blog/data-caching-strategies-for-data-analytics-and-ai-dataai-summit-2023-session-recap/
2. https://fluid-cloudnative.github.io/

DATABASES

Column-Oriented Databases

Column-oriented databases are optimized for reads in the form of analytic queries. Columnar databases store their data by columns, rather than by rows[1]. This makes reads very fast, because column storage supports high levels of compression and parallelization, as well as reduced I/O from skipping unused columns. Most columnar databases now use the relational model and support SQL as a query language. As mentioned earlier, these databases are often utilized for data warehouses.

A popular form of column-oriented database is optimized for highly analytical, complex-query tasks. Snowflake is an example: it stores data in an automatically optimized columnar format within the storage layer, organized into databases as specified by the user with SQL query language. Other column-oriented databases include Apache Cassandra, Apache HBase, ClickHouse, etc.

Real-time Analytic Databases

Real-time analytic databases are a subset of column-oriented databases that focus on use cases that demand fixed, low-latency response, for example P95 sub-second response on slicing and dicing queries. They support very high ingest rates from event streams and highly parallelized scans with constant response latency as well as the ability to scale capacity linearly by adding more hardware. Examples of such databases include Apache Druid, ClickHouse, and Starrocks. They generally favor high performance over features like completeness of SQL implementation or data consistency across nodes.

Federated Query Engines

Federated query engines offer the ability to run analytic queries across diverse data sources ranging from SQL databases to HDFS and object storage. Other than caching, they typically do not have storage but instead depend on remote databases or object storage to hold source data. Presto is an open source distributed query engine for running interactive analytic queries using a standard SQL dialect and batch workload against data sources of all sizes ranging from gigabytes to petabytes[2]. These capabilities also exist to a greater or lesser extent in many other data warehouses. For example, BigQuery and ClickHouse also offer federated query.

Vector Databases

A Vector Database's ability to make unstructured data discoverable as well as locate similar data points among potentially billions make it ideal for helping train generative AI models[3].

A vector database is a database that can store vectors along with other data items. Vector databases can be searched to retrieve the semantically closest matching database records. Vectors are mathematical representations of data in a high-dimensional space. A vector's position in this space represents its characteristics. Words, phrases, or entire documents, and images, audio, video, and other types of data can all be vectorized[4].

Vector databases can be used for similarity search, multi-modal search, recommendation engines, large language models (LLMs), etc. Vector databases are also used to implement Retrieval-Augmented Generation (RAG), a method to improve domain-specific responses of large language models[5].

For example, Milvus[6] is an open-source Cloud Native vector database that is highly flexible, reliable, and fast. Milvus was created to store, index, and manage massive embedding vectors[7] generated by deep neural networks and other machine learning (ML) models. As a cloud-native vector database, Milvus separates storage and computation by design. Storage is the backbone of the system, and is responsible for data persistence. It comprises meta storage, log broker, and object storage. Milvus uses the key-value store etcd to store metadata, uses a messaging/streaming platform such as Kafka or Pulsar to store messages, and uses object storage to store data.

In addition to new databases purposely designed and built to support vector search, this capability is also being added to support some existing database systems.

References

1. NoSQL database types explained: Column-oriented databases
2. https://prestodb.io
3. Vector search now a critical component of GenAI development
4. https://en.wikipedia.org/wiki/Vector_database
5. Retrieval-augmented generation
6. https://milvus.io/
7. Embedding vectors

BLOCK, FILE, AND OBJECT STORAGE

As described in the CNCF Storage Landscape Whitepaper, block storage is best suited for workloads that require availability, low latency performance, and good throughput performance for individual workloads, but it is less suited for workloads that require capacity scaling and sharing data with multiple workloads simultaneously. Block storage can be used accordingly for AI/ML workloads depending on the requirements.

File system based storage is best suited for use cases that need to share data with multiple workloads simultaneously, and need optimized throughput for aggregated workloads. So file system based storage is also a good choice for AI/ML workloads.

The Container Storage Interface (CSI) is a standard for exposing arbitrary block and file storage systems to containerized workloads on Container Orchestration Systems (COs) like Kubernetes[1].

Object storage is best suited for workloads that require availability, large capacities (PB scale), durability, sharing data with multiple workloads simultaneously, and optimized throughput for parallelised workloads[2]. This makes object storage an excellent choice for AI/ML workloads. It is also increasingly favored for analytic databases, where it enables separation of compute and storage necessary to support diverse workloads on very large datasets.

Object storage is least suited for workloads that require low latency performance. A cache layer can be used together with object storage to alleviate the latency constraint as well as reduce costs due to excessive API calls. Object storage I/O occurs through API calls, which means that object storage reads do not benefit from the OS page cache. For this reason, workloads like data warehouses that perform large numbers of repeated reads from object storage almost always include local and/or distributed caches.

Container Object Storage Interface (COSI) is a Kubernetes storage project that introduces standard APIs for the provisioning and consuming of object storage in Kubernetes[3].

FUSE CSI Driver

FUSE (Filesystem in Userspace) is an interface that allows non-privileged users to expose their filesystem to Unix and Unix-like operating systems. In other words, by implementing the interface, a storage system can be translated into a virtual file system. Users are able to access the storage system as if accessing the local filesystem. The FUSE process can be a long-running process inside its own pod and the filesystem can be mounted onto the host machine or Virtual Machine (VM). Whenever an application pod gets scheduled on the host, it uses the hostPath volume to access the storage system.

However, the FUSE process may not always be favored because different workloads may cause under-utilization of resources such as CPU and memory. This is where CSI drivers play their role. Only when the application pods get scheduled, the CSI drivers will get triggered to start the FUSE process, either in a separate pod or in a sidecar mode, that is, injecting another container into the application pod using webhook. Whether the CSI drivers support pod or sidecar depends on the individual implementations.

Major cloud providers have been supporting FUSE CSI Drivers for their object storage services. For example, Azure Blob CSI Driver was released at the end of 2022, Google Cloud Service (GCS) FUSE CSI Driver was released in 2023, and Amazon Web Service (AWS) mountpoint S3 CSI Driver was released in 2023.

References

1. https://github.com/container-storage-interface/spec/blob/master/spec.md
2. CNCF Storage Landscape Whitepaper
3. https://kubernetes.io/blog/2022/09/02/cosi-kubernetes-object-storage-management/

NETWORK MANAGEMENT

Networking is an increasingly important consideration for object storage as well as caches and introduces networking issues that cross layers of the cloud native stack. These manifest in different ways depending on the workload. They may require specialized configuration to set up connectivity within Kubernetes. They also imply ample monitoring to diagnose faults and to assess capacity.

For example, object storage services often apply aggressive rate limiting policies, which lead to either performance throttling or failures at high load. Read and write requests to object storage consume application network bandwidth, hence may produce contention between applications running on the same Kubernetes worker node.

Many design patterns for communication depend on the ability to connect between regions and availability zones using private networking services. These services include AWS PrivateLink[1], Azure Private Link[2], and Google Cloud Private Service Connect[3]. These design patterns include routing traffic between databases running in separate availability zones. They also include design patterns to apply compute resources to object storage using dedicated network connections, such as AWS Direct Connect, Azure ExpressRoute, or Google Cloud Interconnect, which can significantly reduce transit costs and improve performance when accessing object storage.

References

1. AWS PrivateLink
2. Azure Private Link
3. Google Cloud Private Service Connect

DATA PIPELINES - FROM TRANSACTIONS TO ANALYTICS IN THE AI ERA

For decades the data storage architecture for transaction processing and analytics/AI processing has been evolving to meet the demands of that era. Lately, industry drivers for AI and near-real-time insights have made the batch model a bottleneck. Simultaneously, cloud-native paradigms push systems towards asynchronous, loosely-coupled interactions and an architecture where data is emitted as a stream of events[1].

In addition to traditional real-time transactions accessing databases and data stores, there are now asynchronous events, AI agents, and applications data (e.g. logs) accessing and updating these data stores that need to be factored. Many data stores (e.g. PostgreSQL) have evolved to multi-modal data stores required for AI use cases, that include, relational, vector, time series, geo-spatial, and graph[2]. Many of these infrastructures can be set up as cloud native deployments with native Kubernetes support or via operators providing all the benefits of activation, scaling, and monitoring.

These drivers have given rise to many options for batch or streaming Extract, transform and load for analytics and agentic AI. There are many patterns to fit diverse use case needs. This section expands on open-source and CNCF infrastructures that support data analytics in 2026 and beyond.

There are 3 major patterns that an organization can adopt as a roadmap:

- Batch processing data from operational databases to warehouse style databases. Lower complexity and resources.
- Micro-batching from operational databases to Lakehouse engines for richer and higher performance analytics as well as MLops for training AI models - balance capabilities, performance, and resources.
- For near real-time analytics: Real-time streaming from operational data stores to analytics databases or lakehouses typically using change data capture and Kafka.

Infrastructures that support one or more of the above patterns are:

- **Data flow orchestration:** Platforms designed to create, schedule and monitor workflows for many enterprise data centric use cases - business ops, ETL/ELT, MLOps and Infrastructure management. They include directed acyclic graphs (DAGs), built-in retries, and idempotency (examples Apache Airflow, Apache NiFi, Prefect, Dagster, and Knative).
- **Change data capture:** Captures database change events (insert/update/delete) and publishes them to a stream, allowing near real-time replication of operational data into analytical systems (examples Debezium, GoldenGate, StreamSets, and native CDC in Postgres).
- **Event streaming platforms:** High-throughput, fault-tolerant event bus that decouples data producers from consumers. Enables real-time distribution of changes from databases, sensors, logs, and other data sources to multiple subscribers with minimal latency. Notable ones are Apache Kafka, Apache Flink, and Strimzi.

- **Data Quality:** Critical criteria for analytics and training data (examples Great Expectations, Deequ, and Soda)
- **Apache Iceberg Data Lakehouse:** Legacy data lake with ACID compliance. It's a vendor neutral standard with open table format and REST-based API catalog interface. The table format handles schema evolution, partition evolution, and time travel, and partitioning without sacrificing openness.
- **Query Engines:** Required for data processing in both batch and stream modes. Typical capabilities are distributed, multi-language, handle large datasets, high throughput, low latency (examples Apache Spark, Trino, DuckDB, and PySpark)

Three architecture patterns for tackling data movement and transformation with pipelines are documented with proof of concept (POC) in this GitHub site[3].

References

1. "Building a Modern Data Platform based on Data Lakehouse architecture and Cloud-Native Ecosystem", Ahmed AbouZaid, Sep 2024
2. "Just use Postgres for Everything", Stephan Schmidt, updated Dec 2025
3. Modern ETL Infrastructure

DATA MESH AND DATA FABRIC

Data Mesh

Data mesh is an emerging set of principles, gaining traction through the acknowledgment that data is being created and utilized throughout the entire organization. It federates the responsibility to provide data as a product, enables data workers through a self-service data platform, and introduces governance-by-design.

Data warehouses and data lakes are centralized. There are trends to decentralize them towards data mesh. AI/ML models can be built following the data mesh paradigm, a decentralized approach that enables domain teams to perform cross-domain data analysis on their own. At its core is the domain with its responsible team and its operational and analytical data. The domain team ingests operational data and builds analytical data models as data products to perform their own analysis. It may also choose to publish data products with data contracts to serve other domains' data needs[1]. Domain teams can also utilize data products from other domain teams, to enrich their own data products. As AI/ML applications are consumer-oriented data products, this is often the case for AI/ML workloads.

Data mesh is based on four fundamental principles:

- **The domain ownership principle** mandates the domain teams to take responsibility for their data.
- **The data as a product principle** projects a product thinking philosophy onto analytical data.
- **The idea behind the self-serve data infrastructure platform** is to adopt platform thinking to data infrastructure.
- **The federated governance principle** achieves interoperability of all data products through standardization, which is promoted through the whole data mesh by the governance group.

The data mesh paradigm follows a domain-driven design and product thinking to overcome data challenges.

Data Fabric

Data fabric is created to aid data gathering, governance, and distribution.

Data fabric is an architecture that facilitates the end-to-end integration of various data pipelines and cloud environments through intelligent and automated systems. At the center of the data fabric is rich metadata that enables automation, which is designed to automate data integration, engineering, and governance between data providers and consumers[2].

Alongside automation, the data fabric is tasked with aggregating data from various sources, managing the lifecycle of the data and ensuring privacy and data compliance with regulations, and exposing the data to different data consumers through other enterprise search catalogs.

Data fabric relies on automation for discovering, governing, suggesting, and delivering data to data consumers.

Modern Data Architecture Principles emphasize on data quality, data governance, consistency, data as a shareable asset, and data security and privacy.[3]

References

1. https://www.datamesh-architecture.com/
2. https://mia-platform.eu/blog/data-mesh-vs-data-fabric/
3. https://www.cncf.io/blog/2023/07/21/from-data-warehouse-to-data-fabric-the-evolution-of-data-architecture

HARDWARE/SOFTWARE CO-DESIGN

In the past, faster processors and an increasing number of parallel processes were sufficient to meet most requirements from AI and ML workloads. But "Dennard scaling is gone, Amdahl's Law is reaching its limit, and Moore's Law is becoming difficult and expensive to follow, particularly as power and performance benefits diminish"[1].

There is a growing need for domain-specific architecture that customizes hardware for particular AI/ML workloads, optimizing algorithms to leverage hardware parallelism, memory hierarchy, and data movement efficiently. This approach also aims to create systems that are both scalable and energy-efficient. Many tools are being developed to help AI/ML developers and system administrators capture workload details necessary for choosing hardware configurations and making optimization changes. These tools enable a deeper understanding of AI/ML software and algorithms, which is essential for leveraging cutting edge hardware technology innovations to meet these requirements.

There has also been a surge in newer memory technologies, interconnects, and hierarchies, promising enhanced capabilities such as increased bandwidth, capacity, throughput, data sharing, and scalability. This facilitates dynamic and unified memory management across these diverse components. One such hardware technology is CXL - Compute Express Link, an open standard for high-speed, high-capacity connections between central processing units (CPUs) and devices or memory, designed for high-performance data center computing.

References

1. https://semiengineering.com/chip-design-shifts-as-fundamental-laws-run-out-of-steam/

TRAINING AND ITS STORAGE USAGE PATTERNS

Unlike inference workloads, which are predominantly request–response and latency-sensitive, model training is a long-running, distributed, throughput-oriented workload. The primary objective during training is to maintain high utilization of accelerator resources (GPUs/TPUs). From a storage perspective, this translates into sustained, large-scale data movement with predictable read pressure and periodic write amplification events.

Training workloads in cloud-native environments impose distinct requirements on storage systems, particularly in Kubernetes-based deployments where storage is abstracted through CSI drivers and must operate across nodes, zones, and failure domains.

Key Storage Usage Patterns in Training

- **Read-Intensive Throughput:** Training involves "shuffling" and streaming petabyte-scale datasets (images, text tokens, or sensor logs) across hundreds of distributed nodes. Storage must support high-bandwidth, parallel access to prevent the "I/O Wait" bottleneck, where GPUs sit idle waiting for the next batch of data.
- **Checkpointing and Resumability (Write-Burstiness):** Distributed training is prone to hardware failures or preemption (e.g., using Spot instances). To mitigate loss, the system periodically saves the entire model state (checkpoints) to persistent storage. This creates massive, synchronized write bursts across the cluster, requiring storage backends with high aggregate write performance and metadata stability.
- **Randomized Data Access and Shuffling:** To improve model generalization, training pipelines typically shuffle data and perform pseudo-random access across datasets. This disrupts sequential read optimizations and produces high fan-out random read patterns. Storage systems must tolerate non-sequential access, high IOPS across distributed workers, and concurrent metadata lookups. Cloud-native storage solutions often use distributed file systems (parallel FS) or specialized CSI drivers with local NVMe caching to provide the low-latency random seeks required for diverse data loaders.
- **Metadata Overhead:** Training datasets often consist of millions of small files (e.g., individual JPEGs or JSON snippets). This puts immense pressure on the storage metadata server. Architectures often move toward "containerized" data formats like TFRecord, WebDataset, or Apache Parquet to aggregate small files into larger, more manageable chunks, reducing metadata operations.
- **Data Versioning and Lineage:** Reproducibility is a core tenet of MLOps. Storage must support immutable snapshots or integration with versioning tools to ensure that the exact dataset used for a specific training run can be audited or retrained in the future.
- **Scratch Space and Intermediate State:** During the training process, temporary data such as shuffled buffers or gradients are often stored in "scratch" space. This requires high-speed, ephemeral storage, often HostPath or Local Persistent Volumes (PVs), to minimize the latency of intermediate calculations.

Practical Considerations for Organizations

When implementing training storage infrastructure, organizations need to balance performance, cost, and operational complexity. Common patterns include:

- **Hybrid Storage Architecture:** Use object storage (S3-compatible, GCS, Azure Blob) for dataset storage and checkpoints, combined with local NVMe or SSD for active training data and scratch space.
- **Data Pipeline Optimization:** Convert small-file datasets to chunked formats (TFRecord, Parquet, WebDataset) early in the pipeline to reduce metadata overhead and improve throughput.
- **Checkpoint Strategy:** Implement incremental checkpointing and use storage classes with high write performance for checkpoint bursts. Consider checkpoint compression and deduplication for cost optimization.

Common Pitfalls: Over-provisioning expensive high-IOPS storage for the entire dataset when only active training data needs low latency; insufficient scratch space leading to training failures; lack of data versioning causing reproducibility issues; and inadequate checkpoint strategies resulting in long recovery times after failures.

References

1. Apache Parquet - Columnar Storage Format
2. TensorFlow TFRecord Format and Examples
3. WebDataset - Efficient Data Loading for ML
4. PyTorch Data Loading and Processing
5. TensorFlow Data Performance Guide
6. DeepSpeed Training Optimization
7. Horovod Distributed Deep Learning Framework
8. DVC - Data Version Control for ML
9. Kubeflow Trainer
10. Ray Data - Distributed Data Processing

INFERENCE AND ITS STORAGE USAGE PATTERNS

While training focuses on the high-throughput ingestion of massive datasets to create a model, inference is the "production" phase where the model is used to make predictions on new data. In a Kubernetes environment, inference workloads are often deployed as microservices (e.g., using KServe, vLLM, Seldon Core, and more). These workloads are characterized by sensitivity to latency, spiky traffic patterns, and the need for rapid model loading during scaling events.

Key Storage Usage Patterns in Inference

1. **Model Repository Access (Read-Heavy):** The primary storage interaction is loading model weights (often several GBs to TBs for LLMs) from a central repository or object store (S3, GCS) into the GPU/CPU memory.

   Model Caching is a common implementation to store the model locally and reduce service startup time by loading the model directly from the cache.

2. **Model Versioning and Rollouts:** Inference requires strict consistency between the model version and the application code. Storage must support atomic switches between model versions to enable "Blue-Green" or "Canary" deployments without downtime.
3. **Model loading and Artifact access patterns:** Inference workloads use optimized loading techniques to balance storage I/O, memory usage, and startup latency. Model artifacts are commonly memory-mapped to enable on-demand paging rather than fully loading weights at startup, and may be stored as sharded checkpoints to reduce per-node memory requirements at the cost of additional coordination during loading. For very large models, lazy loading further limits initial I/O, while quantized and storage-efficient formats reduce storage footprint, memory consumption, and data transfer overhead during inference.
4. **Feature Stores and Real-time Lookups:** For many inference workloads, such as recommendation or fraud detection, model inputs include features retrieved dynamically from external systems rather than solely from the request payload. These feature stores must support low-latency, high-throughput access, often through in-memory caching or colocated replicas. Storage reliability and consistency are critical to ensure feature values remain aligned with model expectations at serving time.
5. **KV Caching (Key-Value Cache for LLMs):** In generative AI inference, the "KV Cache" stores the mathematical representations of previous tokens in a conversation to avoid redundant calculations. vLLM provides an open source implementation for this.

   Prefix Caching (KV Cache): Store pre-computed prompt states, enabling faster responses for repetitive or long-context queries.

6. **Cold Starts, Scaling, and Storage Locality:** Inference workloads exhibit highly variable traffic, often triggering rapid scale-out events. Without local model caching, new replicas may simultaneously fetch large model artifacts from remote storage, increasing startup latency and storage contention. Production systems mitigate this through node pre-warming, warm replica pools, and init containers or sidecars that stage models on local storage. Ensuring storage locality between model data and compute resources is critical for predictable inference latency during scale-out.

Practical Considerations for Organizations

Organizations implementing inference services need to select storage solutions that support heavy read workloads to meet performance requirements while implementing solid model versioning and rollout strategies. Key considerations include:

- **Storage Performance:** Choose storage backends that provide high read throughput and low latency for model loading. Consider local NVMe caching, distributed file systems, or object storage with Content Delivery Network (CDN) capabilities.
- **Model Management:** Implement robust model versioning with atomic deployments, A/B testing capabilities, and rollback mechanisms. Use model registries that integrate with your CI/CD pipeline.
- **Scaling Strategy:** Design for predictable cold start performance through pre-warming, model caching, and storage locality. Plan for traffic spikes with appropriate auto-scaling policies.
- **Feature Store Integration:** Ensure low-latency access to feature data with appropriate caching strategies and data consistency guarantees.

Common Pitfalls: Underestimating model loading time impact on scaling; insufficient bandwidth for concurrent model downloads during scale-out; lack of proper model versioning causing deployment issues; and inadequate KV cache sizing for LLM workloads leading to performance degradation.

References

1. NVIDIA Disaggregated Serving Architecture
2. BentoML LLM Inference Guide
3. KServe Model Serving Storage
4. vLLM Docker Deployment Guide
5. LLM Deployment and Optimization Patterns
6. Seldon Core MLOps Platform
7. NVIDIA TensorRT SDK

AI AGENT AND ITS STORAGE USAGE PATTERNS

The typical operational process of an AI Agent is a state-driven, closed-loop iterative architecture, centered on utilizing the model as the brain to dynamically integrate multi-dimensional contexts within every execution cycle, and produce various storage usage patterns. The typical processing workflow is as follows[1,2]:

1. **Session Initialization:** upon receiving the input (text, multi-modal data or streaming media), the AI Agent first performs intent recognition and then goal decomposition by synthesizing the input, the initial session state and long-term memory (such as historical preferences, relevant information from past interactions and external knowledge sources). Multi-modal data in the input may include one or several media types like text, image, document, audio clip, video clip etc. Streaming media is continuous live voice or live video and usually in full duplex transmission, not like traditional request-response mode, providing real-time interaction is critical for this kind of AI agent. Input should be persisted with the connection to the session.
2. **Iterative Reasoning Loop:** During each iteration, the model makes decisions through a deeply integrated context window that aggregates the following elements in real-time:

   Instructions and Constraints: The predefined or dynamic synthesized AI Agent prompt and specific requirements for the current step. Instructions and constraints may be divided into one or several text parts, ranging from few bytes to kilobytes.

   Real-time State in Short-term Memory: State is used for holding temporary data relevant only to the current active session, including current variable values, to-do list (or directed acyclic graph (DAG)) and execution status etc. State is mutable, and the contents of the state are expected to be altered as the session evolves in every step. To ensure the AI Agent can complete long-running steps despite various potential failures including node failure, state can be persisted into files or database fields, depending on AI agent developer's decision.

   Events History in Short-term Memory: The complete "Thought-Action-Observation" trajectory occurring within the current session, in chronological sequence including the input, interaction messages, generated responses, tool calls and results, AI agent calls and results. Events history is immutable. Persisting and reloading the entire events history at each step is highly time-consuming. When feeding the events history into the model, it should optimally leverage inference acceleration techniques such as KV caching. Therefore, the events history storage must support append-only writes. Since the model's context window is highly limited, as iteration steps increase, repetitive accumulated content not only affects inference accuracy, latency, and cost but also increases computational overhead. Thus, events history requires good deduplication implementation. Event history must also account for node-level failures, necessitating cross-node accessibility.

   Artifacts as intermediate results: physical objects generated or processed in previous steps, such as code files, images, video clips, audio segments, or documents. Artifacts could be part of the next step inputs as intermediate results for cross steps information sharing, or final deliverables. For example, if the session is to generate source code for a web site, some files may be updated in each step, some files will be reused as inputs for next steps, while others may be generated and kept unchanged in later steps. The session execution generates a large volume of artifacts. These artifacts

require deduplication, cache to improve persistence and loading speeds, while also supporting cross-AI agent and cross-node access. For live streaming media, the stream must be segmented into media blobs and persisted. This serves to mitigate node failures, enables context sharing across AI agents to reduce storage overhead and avoid redundant user input requests. Additionally, occurring events, transcribed text from streaming media and media blobs must be synchronized with their corresponding media blobs on a timeline.

Long-term Memory: search to recall cross-session relevant historical data and external domain knowledge base. The completed session will be ingested and consolidated into persisted long-term memory.

3. **Tool/AI Agent Orchestration and Execution:** Based on the integrated context, the model determines the next action. This may involve invoking tools (e.g., executing Python code in sandbox, calling MCP servers, function calling provided by the AI agent etc.), reading/writing artifacts (e.g., loading content of browsed page, analyzing an image, generating a video, uploading file to remote S3 storage), or dispatching requests to other AI Agents. For multi-AI agents' collaboration scenarios, the data used in the Iterative Reasoning Loop may be shared among AI agents.
4. **Observation and State Update:** The results returned from each action (such as code execution failure, API payloads, newly generated images, or feedback messages from collaborating AI agents) are captured as new observations or artifacts. These are appended to the short-term memory events history or saved to the artifact repository, updating variable values in short-term memory state. To-do list (or directed acyclic graph (DAG)) execution status will be updated by the Model in the next cycle according to the action result. Some AI agent implementation may prefer to save the state to file, this turns to another action.
5. **Delivery and Memory Consolidation:** The AI agent continuously evaluates whether the feedback satisfies the termination criteria. Upon completion, the critical decision paths, conclusion and produced artifacts are not only used to finalize the goal as deliverables, but may also be indexed into long-term memory, allowing them to be reactivated and referenced in future independent sessions.

The aforementioned persisted contents vary drastically in several aspects in AI Agent workflow, must meet both AI agents' providers and end-users' purposes, for example:

- An AI Agent processing streaming media might temporarily store interactive video/audio segments for replay purposes.
- Intermediate generated files may reside in the sandboxed execution environment with limited retention for context reuse.
- Final conclusions and delivered artifacts to be published on a website.

References

1. OpenClaw - Your own personal AI assistant
2. Agent Development Kit (ADK) - An open-source, code-first Python framework for building AI agents


TAG
INFRASTRUCTURE