Frictionless Analysis Ready Dataset Architecture for the Radio Cluster: A Multi-Modal Engineering Report

1. Infrastructure Grounding and Architectural Philosophy

The design of high-performance scientific data systems requires a rigorous adherence to the physical realities of the underlying hardware. In the context of the "Radio" Cluster, a bespoke on-premise environment tasked with processing the Dark Energy Spectroscopic Instrument (DESI) multi-modal dataset, the architecture must navigate a complex landscape of resource constraints and distinct hardware advantages. This report delineates the engineering strategy for constructing an "Analysis Ready Dataset" (ARD) that prioritizes I/O throughput, Python ecosystem interoperability, and scientific reproducibility.

The central challenge is the unification of 6.4 million relational galaxy records with 12,000 high-density Parquet spectral tiles within a restricted network environment, specifically leveraging a limited number of Kubernetes (k8s) worker nodes and a single GPU accelerator. This necessitates a departure from traditional enterprise "Big Data" paradigms—which often impose unacceptable overheads on mid-sized clusters—in favor of a "Negative Space" architecture that aggressively removes middleware layers to expose raw hardware performance to the application layer.

1.1 The "Radio" Hardware Truth

The physical substrate of the Radio Cluster dictates every software decision. Unlike hyperscale cloud environments where compute and storage are elastically decoupled, the Radio Cluster is a fixed-capacity system where data locality and interconnect latency are the primary determinants of performance.

Node Identifier	Specification Profile	Operational Role & Constraint Analysis
radio-pgsql01	8 vCPU, 32GB RAM	**Metadata Sentinel.** This node possesses high transactional throughput (205k read TPS) but limited working memory. It cannot serve as the compute engine for heavy joins or spectral processing. It must be treated purely as an indexer.
radio-k8s[01-03]	36 vCPU, 144GB RAM (Total), 3TB NVMe	**The Engine Room.** These nodes provide the bulk of the CPU cycles and, crucially, the NVMe storage bandwidth. The aggregate RAM (144GB) is sufficient to hold the entire dataset (<100GB) in memory, enabling distributed caching strategies.
radio-gpu01	NVIDIA A4000 (16GB VRAM), 8 vCPU	**The Scientific Payload.** The 16GB VRAM is a hard ceiling for deep learning models. The low CPU count (8 cores) creates a high risk of "data starvation," where the GPU sits idle waiting for data preprocessing.
Network Fabric	10GbE Internal	**The Throttle.** While 10GbE is capable of ~1.25 GB/s theoretical throughput, protocol overhead (TCP/IP) and serialization costs can

		degrade this significantly. Cross-node data movement must be minimized during training loops.

The overarching constraint is the "Impedance Mismatch" between the storage medium (NVMe on k8s nodes) and the compute target (GPU node). While the dataset fits in the aggregate RAM of the cluster, it does not fit in the RAM of the GPU node alone, nor the VRAM of the accelerator. Therefore, the architecture must implement a sophisticated streaming mechanism that acts as a virtual memory pipeline, fetching data from the k8s NVMe/RAM layer to the GPU with sub-millisecond latency.

1.2 The "Negative Space" Design Strategy

In engineering deeply optimized systems, what is *excluded* is often as important as what is included. The constraints provided—No CISO frameworks, No DB tuning, and No Big Data tools—are not limitations but architectural freedoms.

Rejection of the JVM Stack (Hadoop/Spark):
Traditional data engineering often defaults to Apache Spark or Hadoop for datasets of this scale. However, on a cluster with limited nodes, the overhead of the Java Virtual Machine (JVM), the complexity of the Yarn resource manager, and the serialization costs between Java objects and Python scientific stacks (NumPy/PyTorch) introduce unacceptable friction. Benchmarks indicate that for multimodal workloads involving images or spectra, Python-native engines like Daft or Ray Data can outperform Spark by 4-18x due to better GPU utilization and reduced serialization overhead.1 Consequently, the "Radio" architecture is strictly Python-native.
Rejection of In-Database Processing:
While radio-pgsql01 is capable of 205k TPS, it is optimized for point queries, not bulk analytical scans of binary spectral data. Attempts to load large binary blobs or perform heavy mathematical operations (like Fourier transforms on spectra) within Postgres would saturate the node's CPU and network link, creating a single point of failure. The database will be strictly limited to metadata management—storing pointers to files rather than the files themselves. This "Sidecar" pattern ensures the database never becomes the I/O bottleneck.
Security as a Non-Functional Requirement:
The absence of CISO/Security frameworks allows for the use of high-performance, unencrypted transport protocols within the internal network. We can utilize raw TCP sockets or shared memory segments without the overhead of TLS termination or complex Kerberos

authentication, maximizing the 10GbE throughput for scientific payloads.

---

2. Area A: The "Bifrost" Linkage Architecture

The fundamental data engineering challenge for the DESI dataset is linking the metadata (galaxy properties like redshift, coordinates, and target IDs stored in SQL) with the dense data (1D spectra stored in Parquet files). Traditional approaches often force one data type into the storage medium of the other—either storing spectra as BLOBs in SQL or exporting catalogs to flat files. Both result in suboptimal performance.

The "Bifrost" architecture proposes a hybrid linkage strategy where Postgres acts as a "Sidecar Index" to the Parquet data lake. The linkage is not physical but logical, maintained by a rigorous sync process and resolved at query time using high-performance Python APIs.

2.1 Evaluating Storage Linkage Mechanisms

To determine the optimal linkage mechanism, we analyze three competing technologies: Apache Iceberg, Apache Arrow Flight, and PostgreSQL Foreign Data Wrappers (FDW).

Apache Iceberg: The Data Lakehouse Standard
Apache Iceberg has emerged as a dominant table format for large-scale analytics, offering features like time travel, schema evolution, and partition pruning.3 It excels in environments where data is mutable and concurrent writes are frequent. Iceberg manages metadata through snapshot files (Avro) that list the data files (Parquet) constituting a table state.4 However, Iceberg is heavily engineered for "Big Data" environments (S3/HDFS) and often requires a catalog service (like Nessie or Hive Metastore) to track table state.5 For a static dataset of 12,000 files on a local NVMe RAID, the overhead of managing Iceberg manifests and snapshots is disproportionate to the value provided. The "Radio" dataset is largely immutable—once spectra are processed, they are rarely updated individually. Therefore, the complexity of Iceberg is rejected in favor of a simpler file-system-based approach.
PostgreSQL Foreign Data Wrappers (FDW): The False Promise
The postgres_fdw and parquet_fdw extensions allow PostgreSQL to mount external files as virtual tables, enabling users to query Parquet data directly via SQL.6 Superficially, this seems ideal for the Radio Cluster.
However, deep analysis reveals critical performance flaws. When a query runs against an FDW table, the Postgres execution engine must pull the data from the remote file, serialize it into Postgres tuples, and process it. This serialization is CPU-intensive and destroys the columnar

advantages of Parquet.8 Furthermore, the Postgres query planner often lacks statistics for FDW tables, leading to inefficient execution plans (e.g., full table scans instead of index seeks).9 Most critically, streaming 100GB of spectral data through the single radio-pgsql01 node would saturate its 10GbE link and 8 vCPUs, creating a massive bottleneck.10 The database becomes a funnel that slows down the entire pipeline.
Apache Arrow Flight: The High-Speed Transport
Apache Arrow Flight provides a wire protocol optimized for streaming columnar data between systems without serialization overhead.3 It allows a client to request a data stream from a server, which sends Arrow record batches directly. This offers extremely high throughput, potentially saturating the 10GbE link.
While Arrow Flight is superior to FDW, it requires deploying and managing Flight Servers on the storage nodes. Given that our compute framework (Ray) has its own internal object transport mechanism that is arguably more integrated for ML workloads 13, a standalone Arrow Flight layer adds operational complexity without necessarily adding throughput for this specific use case.

2.2 Recommended Architecture: The Sidecar Index & Ray Data

The optimal solution leverages **PostgreSQL for Metadata** and **Ray Data for Density**, linked by **Parquet Footer Statistics**.

In this model, the Postgres database stores a catalog of every Parquet file, including the file path, row counts, and potentially min/max values of key columns (like TARGETID or Healpix pixel). This is populated by a lightweight Python script that scans the Parquet footers.[14]

When a user requests a subset of data (e.g., "All galaxies with redshift > 2.5"), the workflow is:

1. **Metadata Query:** The user queries Postgres to get the list of file paths containing the relevant targets. This is a fast, index-optimized SQL query returning only strings (paths).
2. **Parallel Fetch:** The list of paths is passed to Ray Data. Ray distributes the read tasks across the radio-k8s nodes.
3. **Direct Read:** Each k8s node reads its assigned Parquet files directly from local NVMe into memory (Ray Plasma Store), bypassing the Postgres node entirely.

This architecture eliminates the single bottleneck of the database server and utilizes the aggregate I/O bandwidth of all k8s nodes.

2.3 Parquet Optimization Strategy

To maximize the efficiency of the "Direct Read" phase, the Parquet files themselves must be optimized.

Row Group Sizing:
The row group is the atomic unit of a Parquet file. Large row groups improve compression and sequential read throughput but require more memory to deserialize. Small row groups allow for finer-grained filtering (predicate pushdown) but increase metadata overhead. For the DESI dataset, where we expect batch reads for ML training, we recommend a row group size between 128MB and 512MB.16 This aligns with the memory pages of the OS and minimizes the number of I/O seeks.
Footer Metadata Utilization:
Parquet files store metadata (min/max statistics for columns) in the file footer. By reading only the footer 14, our synchronization scripts can populate the Postgres index without reading the gigabytes of binary spectral data. This allows us to rebuild the index in seconds rather than hours. The pyarrow.parquet.read_metadata function is critical here, allowing us to extract num_rows, serialized_size, and column statistics efficiently.14

Feature	**Implementation Detail**	**Benefit**
File Format	Parquet (Snappy Compression)	High read throughput, wide ecosystem support.
Row Group Size	256 MB	Balance between compression and random access.
Index Strategy	"Sidecar" Postgres Table	Decouples storage from compute; prevents DB saturation.
Access API	Ray Data (reading Parquet)	Parallel, distributed loading into shared memory.

3. Area B: Machine Learning Infrastructure

The second research area focuses on the "ML-Ready" aspect of the ARD. Transforming raw files into tensors for training is often the slowest part of a deep learning pipeline. On the Radio

Cluster, we must ensure the radio-gpu01 node is never starved of data.

3.1 The DataLoader Bottleneck: "CPU Starvation"

The standard torch.utils.data.DataLoader is insufficient for high-throughput training on this infrastructure. The DataLoader relies on Python's multiprocessing to spawn worker processes that load and preprocess data.

- **Serialization Cost:** Every batch loaded by a worker process must be serialized (pickled) and sent via Inter-Process Communication (IPC) to the main training process. For large tensors (like spectra), this serialization consumes significant CPU cycles.[19]
- **Redundant Loading:** If multiple models are trained, or if the dataset is larger than the worker's memory, data is repeatedly loaded from disk, thrashing the filesystem.
- **The "Starvation" Effect:** On the radio-gpu01 node, which has only 8 vCPUs, the CPU struggle to serialize/deserialize data fast enough to keep the A4000 GPU busy. The GPU sits idle (0% utilization) while the CPU is at 100%.[20]

3.2 The Solution: Ray Plasma and Zero-Copy Reads

To solve CPU starvation, we utilize **Ray Data** and its underlying **Plasma Object Store**. Plasma is a shared-memory server allowing multiple processes to access the same memory objects without copying them.

Mechanism of Action:

1. **Distributed Loading:** The radio-k8s nodes (with 36 vCPUs) act as the data loaders. They read Parquet files, perform tokenization (see Section 5), and store the resulting tensors in the Plasma store on their local RAM.
2. **Global Object Space:** Ray presents these distributed memory chunks as a single logical dataset.
3. **Zero-Copy Transfer:** When the training script on radio-gpu01 requests a batch, Ray transfers the data over the 10GbE network. Once the data arrives in the local Plasma store of the GPU node, the training process maps it directly into its address space. Crucially, **no copy occurs between the Plasma store and the Python process** for NumPy arrays.[13]

Impact on VRAM and RAM:
By using zero-copy reads, we reduce the system RAM footprint. Instead of the DataLoader

holding a copy of the batch and the training loop holding a copy, they share the underlying memory. This frees up RAM for larger buffers or caching.22

3.3 Comparative Benchmarking: Ray vs. Daft vs. PyTorch

Recent benchmarks provide compelling evidence for this architectural choice.

Ray Data vs. PyTorch:
In image classification benchmarks (analogous to spectral classification), Ray Data has shown to be 30-45% faster than standard PyTorch DataLoaders, primarily due to superior prefetching and the elimination of IPC overhead.21 For complex preprocessing pipelines (like our spectral tokenization), this gap widens as the CPU becomes the bottleneck.
Ray Data vs. Daft:
Daft is an emerging dataframe library optimized for multi-modal data. In raw throughput benchmarks (e.g., reading Parquet from S3), Daft has demonstrated speeds 2-7x faster than Ray Data and 4-18x faster than Spark.1 Daft achieves this through "fused execution," pipelining I/O and compute more tightly in C++ (Rust backend).
However, despite Daft's raw speed, Ray Data is selected for the Radio Cluster.

- *Reasoning:* Ray provides a holistic operating system for the cluster. It handles not just data loading, but also distributed training (Ray Train), hyperparameter tuning (Ray Tune), and model serving (Ray Serve).[19] Daft is primarily a data engine. Given the need for a unified ML platform on the k8s nodes, Ray's ecosystem integration outweighs Daft's raw I/O advantage. Furthermore, recent Ray versions (2.5+) have closed the gap, with Ray Data showing faster performance on large-scale shuffle operations and specific multimodal ingest tasks.[24]

3.4 Pre-fetching Strategy

Given the 10GbE network bottleneck between the radio-k8s (storage) and radio-gpu01 (compute) nodes, aggressive prefetching is mandatory.
Ray Data's iter_torch_batches(prefetch_batches=N) allows us to tune the pipeline. We will configure the pipeline to maintain a buffer of N=10 batches on the GPU node. This absorbs the latency jitter of the network. The high RAM (144GB) on the k8s nodes allows us to cache the entire dataset in the Plasma store after the first epoch, effectively turning the cluster into an in-memory database for subsequent epochs.25

---

4. Area C: Science Payload and Algorithmic Adaptation

The scientific utility of the ARD is demonstrated through three experiments tailored to the hardware's limits. These experiments address the specific challenges of high-dimensional astronomical data: Graph Neural Networks for cosmic structure, unsupervised anomaly detection for rare object search, and clustering for galaxy group finding.

4.1 Experiment 1: Cosmic Web GNN (Graph Neural Networks)

Scientific Goal: Classify galaxies by their cosmic web environment (filament, knot, void) using their spatial relationships.
The Problem: A graph of 6.4 million galaxies contains trillions of potential edges. Loading the full adjacency matrix and node features into the 16GB VRAM of the A4000 is impossible. A standard GNN (like GCN) requires the full graph during training.
Architectural Solution: Subgraph Sampling & Gradient Accumulation
We adopt the GraphSAGE (Graph Sample and Aggrecate) architecture.26 Unlike GCNs, GraphSAGE learns to generate embeddings by sampling and aggregating features from a local neighborhood. This allows us to train on minibatches of subgraphs rather than the full graph.
Gradient Accumulation Implementation:
Even with subgraph sampling, the dense spectral features per node might force a small batch size (e.g., 64 nodes) to fit in VRAM, which leads to noisy gradients and unstable convergence. We utilize Gradient Accumulation to simulate a larger batch size.27

- *Mechanism:* The model performs a forward and backward pass on a small batch (e.g., 64 nodes). Instead of updating the weights immediately (optimizer.step()), the gradients are accumulated in a buffer. This is repeated for N steps (e.g., 64 steps). The optimizer step is taken effectively on a batch of $64 \times 64 = 4096$ nodes.
- *VRAM Impact:* This technique keeps peak memory usage constant (equal to the small batch size) while achieving the convergence stability of a large batch.[29] This is the only viable way to train "CosmicNet" scale models on a single A4000.

4.2 Experiment 2: Unsupervised Anomaly Detection (Autoencoders)

Scientific Goal: Discover rare transients or processing artifacts in the spectral data.
Algorithm: A Convolutional Autoencoder (CAE) or Masked Autoencoder (MAE).30 The model compresses the 1D spectrum into a low-dimensional latent space and attempts to reconstruct it. The reconstruction error (MSE) serves as the "Anomaly Score."
Execution Strategy:
This workload is compute-bound (convolutions) rather than memory-bound. It is the ideal background task for the A4000.

- **Streaming Inference:** We leverage Ray Data's streaming execution to feed the autoencoder. The dataset of 12,000 Parquet tiles is streamed through the GPU.
- **Zero-Shot Discovery:** Unlike supervised classification, this requires no labels. The output is a new Parquet file containing ``. This list is then linked back to Postgres, allowing astronomers to query SELECT * FROM anomalies WHERE score > threshold and inspect the top candidates visually.

4.3 Experiment 3: Friends-of-Friends (FoF) Clustering

Scientific Goal: Identify galaxy clusters by linking galaxies that are spatially close.
The Problem: The Friends-of-Friends algorithm is fundamentally an $O(N^2)$ problem. For every galaxy, we must check the distance to every other galaxy to see if it is within the linking length l.31 For 6.4 million galaxies, a brute-force approach involves $\approx 4 \times 10^{13}$ calculations. In Python, this would take years.
Architectural Solution: KD-Tree Spatial Partitioning
We solve this by moving the compute from the GPU to the CPU-heavy radio-k8s nodes, utilizing Spatial Partitioning.

- **Algorithm:** We use a **KD-Tree** (k-dimensional tree) or Octree implementation. This data structure partitions the 3D space (RA, Dec, Redshift) into hierarchical cells.[33]
- **Complexity Reduction:** With a KD-Tree, finding neighbors within distance l reduces the complexity from $O(N^2)$ to roughly **$O(N \log N)$** or $O(N \cdot k)$ where k is the average number of neighbors.
- **Implementation:** We utilize the **Nessie** library (or a Scikit-learn equivalent wrapped in Ray), which implements a KD-Tree optimized for astronomical grouping.[35] The 144GB RAM on the k8s nodes allows us to build the entire tree in memory, ensuring extremely fast lookups. The result is a GroupID appended to the galaxy metadata.

---

5. Spectral Tokenization and Feature Engineering

To enable the Multi-Modal Machine Learning described in Area B, the raw 1D spectra must be transformed into a format ingestible by Transformer models (like AstroLLaMA or CosmicNet). This process, **Tokenization**, is a critical preprocessing step that bridges the gap between continuous physical signals and discrete neural network inputs.

5.1 Universal Spectral Tokenization Strategy

Unlike Natural Language Processing (NLP) which uses Byte-Pair Encoding (BPE) on discrete text vocabularies (e.g., 32k or 128k tokens [36]), astronomical spectra are continuous, real-valued signals. Treating specific flux values as "words" is ineffective due to noise and infinite variability.

Instead, we adopt the **Patch Embedding** strategy derived from Vision Transformers (ViT) and applied to 1D signals.[38]

The Methodology:

1. **Segmentation:** The full spectrum (e.g., 4096 wavelength bins) is sliced into fixed-size patches. Research suggests a patch size of **32 to 64 pixels** is optimal for capturing local spectral features (like emission lines) while keeping the sequence length manageable for the Transformer's attention mechanism.[40]
2. **Linear Projection:** Each patch (a vector of 32 floats) is flattened and passed through a linear dense layer to project it into a latent embedding dimension (e.g., $d=512$).
3. **Positional Encoding:** Since Transformers are permutation-invariant, we must inject wavelength information. We utilize sinusoidal positional encodings (frequencies based on the wavelength range) added to the patch embeddings.[42]

5.2 Implementation with Ray Data

This tokenization is mathematically simple but computationally expensive when applied to 6.4 million spectra. Doing this inside the GPU training loop would be disastrous for performance.

The "Map Batches" Optimization:
We implement tokenization as a Ray Data map_batches operation running on the radio-k8s CPUs.43

Python

```python
# Conceptual Ray Data Tokenization
def tokenize_batch(batch):
    # batch['flux'] shape: (batch_size, 4096)
    # Reshape to (batch_size, num_patches, patch_size)
    patches = batch['flux'].reshape(-1, 64, 64)
    # Normalize (e.g., divide by median flux) [41]
    normalized = patches / np.median(patches, axis=2, keepdims=True)
    return {"tokens": normalized}

ds = ray.data.read_parquet(files).map_batches(tokenize_batch, batch_size=1024)
```

By executing this on the k8s nodes, we saturate the 100+ CPU cores available. The output—ready-to-consume tensors—is stored in the Plasma store. When the GPU requests data, it receives pre-tokenized, normalized tensors, allowing the A4000 to dedicate 100% of its FLOPS to the Transformer's attention layers.

---

6. Data Governance: The Living Data Dictionary

A dataset without accurate documentation is a "write-only" archive. For the Radio Cluster ARD, we reject static PDF documentation in favor of a "Living Data Dictionary" that updates programmatically. This aligns with the **Datasheets for Datasets** framework, which demands transparency regarding data composition, provenance, and recommended uses.[44]

6.1 Automated Metadata Sync

The manual creation of data dictionaries leads to drift—where the documentation describes the data as it *was*, not as it *is*. We mitigate this via the **Linkage Layer Sync Script** (described in Section 2.2).

Whenever the Parquet files are indexed into Postgres, the script also computes aggregate statistics:

- **Cardinality:** Total galaxies, breakdown by type (ELG, LRG, QSO).

- **Completeness:** Percentage of missing values in flux columns.
- **Ranges:** Min/Max Redshift, coverage in RA/DEC.

6.2 The Markdown Artifact

These statistics are injected into a Jinja2 template to generate a DATA_README.md file residing at the root of the dataset directory.

Structure of the Datasheet:

1. **Motivation:** Explicitly stating the DESI science goals (Dark Energy constraints).
2. **Composition:**
   - *Table:* Automated summary of row counts and file sizes.
   - *Linkage Key:* Explicit documentation that TARGETID is the primary key joining Postgres (metadata) and Parquet (features).
3. **Collection Process:** Details on the pipeline version used to generate the spectra.
4. **Preprocessing:** Exact specifications of the Tokenization (Patch Size = 64, Normalization = Median).[46]

This ensures that any scientist accessing the ARD has immediate, guaranteed-accurate information on the data state, fostering trust and scientific reproducibility.

---

7. Conclusion

The "Bifrost" architecture represents a pragmatic, high-performance evolution of the Analysis Ready Dataset concept, specifically tailored to the constraints of the Radio Cluster. By explicitly rejecting the complexity of the "Big Data" JVM stack and the latency of FDW-based SQL integration, we have designed a system that is lean, fast, and scientifically potent.

Key Architectural Victories:

1. **Linkage:** The Sidecar Index pattern decouples metadata (SQL) from density (Parquet), allowing each storage engine to play to its strengths without the bottleneck of FDW serialization.
2. **Throughput:** Ray Data and Plasma eliminate the "CPU Starvation" of the GPU node by leveraging the massive RAM and CPU pools of the k8s nodes for prefetching and zero-copy transfer.
3. **Science Enablement:** Algorithmic adaptations like Gradient Accumulation and KD-Tree

partitioning allow the limited A4000 GPU to train large-scale GNNs and perform cosmic clustering that would otherwise require a supercomputer.

This architecture transforms the Radio Cluster from a constrained on-premise rack into a capable engine for modern cosmological inference.

Appendix A: Implementation Artifacts and Library Stack

A.1 Recommended Library Stack

- **I/O & Format:** pyarrow (Parquet/Arrow), psycopg2 (Postgres Linkage).
- **Orchestration & Loading:** ray[data, train] (Distributed Compute).
- **Deep Learning:** torch (Core), torch_geometric (GNNs), flash_attn (Transformer acceleration).
- **Scientific Computing:** numpy, scipy, astropy (Coordinates/Units), nessie (Fast FoF).

A.2 Resource Partitioning Plan

To ensure stability without a resource manager like Yarn, we statically partition the cluster resources:

Node Group	Role	Resource Allocation Strategy
radio-pgsql01	**Metadata**	**100% Reserved.** No user jobs allowed. Postgres configured with shared_buffers=8GB (25% RAM).

radio-k8s[01-03]	**Data Plane**	**Ray Workers.** 50% RAM (72GB) reserved for Plasma Object Store. Remaining for Tokenization workers.
radio-gpu01	**Training**	**Ray Head + Trainer.** CPU cores pinned to the training process to minimize context switching.

A.3 Metadata Sync Script Logic

The synchronization script is the heartbeat of the system. It must be idempotent.

Python

```python
def sync_logic():
    """
    1. Scan Parquet Directory
    2. read_metadata(footer_only=True) [14]
    3. Extract num_rows, min/max stats
    4. UPSERT into Postgres 'file_registry' table
    5. Generate DATA_README.md from Postgres Aggregates
    """
    pass
```

Works cited

1. Benchmarks for Multimodal AI: Spark, Ray Data, and Daft, accessed November 22, 2025, https://www.daft.ai/blog/benchmarks-for-multimodal-ai-workloads
2. Daft vs Ray Data: A Comprehensive Comparison for Multimodal Data Processing, accessed November 22, 2025, https://dev.to/yks/daft-vs-ray-data-a-comprehensive-comparison-for-multimodal-data-processing-3686
3. Apache Iceberg & Apache Arrow Flight | by Thomas Lawless | Medium, accessed November 22, 2025,

https://medium.com/@tglawless/apache-iceberg-apache-arrow-flight-7f95271b7a85
4. Apache Arrow with Matt Topol - DevOps 175 - - Top End Devs, accessed November 22, 2025, https://topenddevs.com/podcasts/adventures-in-devops/episodes/apache-arrow-with-matt-topol-devops-175
5. Understanding Parquet, Iceberg and Data Lakehouses - Hacker News, accessed November 22, 2025, https://news.ycombinator.com/item?id=38811576
6. Mastering Postgres_FDW: Setup, optimize performance and avoid Common Pitfalls, accessed November 22, 2025, https://techcommunity.microsoft.com/blog/adforpostgresql/mastering-postgres-fdw-setup-optimize-performance-and-avoid-common-pitfalls/4463564
7. F.38. postgres_fdw — access data stored in external PostgreSQL servers, accessed November 22, 2025, https://www.postgresql.org/docs/current/postgres-fdw.html
8. Performance Tips for Postgres FDW | Crunchy Data Blog, accessed November 22, 2025, https://www.crunchydata.com/blog/performance-tips-for-postgres-fdw
9. PostgreSQL fdw table performance drops exponentially after exactly 5 identical queries, accessed November 22, 2025, https://stackoverflow.com/questions/69087568/postgresql-fdw-table-performance-drops-exponentially-after-exactly-5-identical-q
10. Performance impact of accessing TIMESTAMP fields from Big SQL with Parquet MR files - Hadoop Dev - IBM, accessed November 22, 2025, https://www.ibm.com/support/pages/performance-impact-accessing-timestamp-fields-big-sql-parquet-mr-files-hadoop-dev-0
11. Which one should i use in spark sql for better performance, either reading the data from Parquet file or reading data from database? - Stack Overflow, accessed November 22, 2025, https://stackoverflow.com/questions/36786604/which-one-should-i-use-in-spark-sql-for-better-performance-either-reading-the-d
12. Introduction to Data Engineering Concepts |17| Apache Iceberg, Arrow, and Polaris, accessed November 22, 2025, https://dev.to/alexmercedcoder/introduction-to-data-engineering-concepts-17-apache-iceberg-arrow-and-polaris-2ei1
13. Serialization — Ray 2.51.1 - Ray Docs, accessed November 22, 2025, https://docs.ray.io/en/latest/ray-core/objects/serialization.html
14. pyarrow.parquet.read_metadata — Apache Arrow v22.0.0, accessed November 22, 2025, https://arrow.apache.org/docs/python/generated/pyarrow.parquet.read_metadata.html
15. How to access Parquet file metadata | by Sanjeet Shukla - Medium, accessed November 22, 2025, https://medium.com/@sanjeets1900/how-to-access-parquet-file-metadata-26906b2dd626
16. All About Parquet Part 10 — Performance Tuning and Best Practices with Parquet |

by Alex Merced | Data, Analytics & AI with Dremio | Medium, accessed November 22, 2025, https://medium.com/data-engineering-with-dremio/all-about-parquet-part-10-performance-tuning-and-best-practices-with-parquet-d697ba4e8a57
17. pyarrow.parquet.read_metadata — Apache Arrow v12.0.1, accessed November 22, 2025, https://arrow.apache.org/docs/12.0/python/generated/pyarrow.parquet.read_metadata.html
18. Reading and Writing the Apache Parquet Format, accessed November 22, 2025, https://arrow.apache.org/docs/python/parquet.html
19. Comparing Ray Data to other systems — Ray 2.51.1 - Ray Docs, accessed November 22, 2025, https://docs.ray.io/en/latest/data/comparisons.html
20. Dose data_prefetcher() really speed up training? · Issue #304 · NVIDIA/apex - GitHub, accessed November 22, 2025, https://github.com/NVIDIA/apex/issues/304
21. Fast, flexible, and scalable data loading for ML training with Ray Data - Anyscale, accessed November 22, 2025, https://www.anyscale.com/blog/fast-flexible-scalable-data-loading-for-ml-training-with-ray-data
22. How to Load PyTorch Models 340 Times Faster with Ray | by Fred Reiss - Medium, accessed November 22, 2025, https://medium.com/ibm-data-ai/how-to-load-pytorch-models-340-times-faster-with-ray-8be751a6944c
23. Distributed Machine Learning with PyTorch and Ray | by özkan uysal - Medium, accessed November 22, 2025, https://medium.com/@uysalozkan/distributed-machine-learning-with-pytorch-and-ray-b21907f4ab35
24. Benchmarking Multimodal AI Workloads on Ray Data - Anyscale, accessed November 22, 2025, https://www.anyscale.com/blog/ray-data-daft-benchmarking-multimodal-ai-workloads
25. Accelerating Data Loading in Large-Scale ML Training With Ray and Alluxio, accessed November 22, 2025, https://www.alluxio.io/blog/accelerating-data-loading-in-large-scale-ml-training-with-ray-and-alluxio
26. Graph Neural Networks Part 3: How GraphSAGE Handles Changing Graph Structure, accessed November 22, 2025, https://towardsdatascience.com/graph-neural-networks-part-3-how-graphsage-handles-changing-graph-structure/
27. Finetuning LLMs on a Single GPU Using Gradient Accumulation - Lightning AI, accessed November 22, 2025, https://lightning.ai/pages/blog/gradient-accumulation/
28. Training Neural Nets on Larger Batches: Practical Tips for 1-GPU, Multi-GPU & Distributed setups | by Thomas Wolf | HuggingFace | Medium, accessed November 22, 2025,

https://medium.com/huggingface/training-larger-batches-practical-tips-on-1-gpu-multi-gpu-distributed-setups-ec88c3e51255
29. Part 1.1: Training Larger Models on a Single GPU - the UvA Deep Learning Tutorials!, accessed November 22, 2025, https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/scaling/JAX/single_gpu_techniques.html
30. Multi-Modal Masked Autoencoders for Learning Image-Spectrum Associations for Galaxy Evolution and Cosmology - arXiv, accessed November 22, 2025, https://arxiv.org/html/2510.22527v1
31. sOPTICS: a modified density-based algorithm for identifying galaxy groups/clusters and brightest cluster galaxies | Monthly Notices of the Royal Astronomical Society | Oxford Academic, accessed November 22, 2025, https://academic.oup.com/mnras/article/537/2/1504/7965971
32. Efficient similarity-based data clustering by optimal object to cluster reallocation - PMC, accessed November 22, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC5983489/
33. Exemplar Learning for Extremely Efficient Anomaly Detection in Real-Valued Time Series, accessed November 22, 2025, https://www.merl.com/publications/docs/TR2016-027.pdf
34. Disk Aware Discord Discovery: Finding Unusual Time Series in Terabyte Sized Datasets - Computer Science and Engineering - University of California, Riverside, accessed November 22, 2025, https://www.cs.ucr.edu/~eamonn/DiskawareDiscords.pdf
35. (PDF) Nessie: A Rust-Powered, Fast, Flexible, and Generalized Friends-of-Friends Galaxy-Group Finder in R and Python - ResearchGate, accessed November 22, 2025, https://www.researchgate.net/publication/394882665_Nessie_A_Rust-Powered_Fast_Flexible_and_Generalized_Friends-of-Friends_Galaxy-Group_Finder_in_R_and_Python
36. Why does llama3.2 tokenizer have more merges than vocab size? : r/LocalLLaMA - Reddit, accessed November 22, 2025, https://www.reddit.com/r/LocalLLaMA/comments/1ilnlyq/why_does_llama32_tokenizer_have_more_merges_than/
37. AstroLLaMA: Towards Specialized Foundation Models in Astronomy - ACL Anthology, accessed November 22, 2025, https://aclanthology.org/2023.wiesp-1.7.pdf
38. Universal Spectral Tokenization via Self-Supervised Panchromatic Representation Learning, accessed November 22, 2025, https://arxiv.org/html/2510.17959v1
39. Applying Vision Transformers on Spectral Analysis of Astronomical Objects - arXiv, accessed November 22, 2025, https://arxiv.org/html/2506.00294v1
40. Universal Spectral Tokenization via Self-Supervised Panchromatic Representation Learning - arXiv, accessed November 22, 2025, https://www.arxiv.org/pdf/2510.17959v2
41. Universal Spectral Tokenization via Self-Supervised Panchromatic Representation Learning - ChatPaper, accessed November 22, 2025,

https://chatpaper.com/paper/201987
42. Surveying Image Segmentation Approaches in Astronomy - arXiv, accessed November 22, 2025, https://arxiv.org/html/2405.14238v1
43. Advanced: Performance Tips and Tuning - Ray Docs, accessed November 22, 2025, https://docs.ray.io/en/latest/data/performance-tips.html
44. JRMeyer/markdown-datasheet-for-datasets - GitHub, accessed November 22, 2025, https://github.com/JRMeyer/markdown-datasheet-for-datasets
45. [1803.09010] Datasheets for Datasets - arXiv, accessed November 22, 2025, https://arxiv.org/abs/1803.09010
46. Data Documentation and Metadata - Data Cooperative - The University of Arizona, accessed November 22, 2025, https://data.library.arizona.edu/data-management/best-practices/data-documentation-readme-metadata