Harvesting, Processing, Storing data from HPCM systems

Ben Lenard,Eric Pershey,Brian Toonen,Peter Upton,Doug Waldron,Micheal Zhang,Lisa Childers, Bryan Brickman
Argonne National Laboratory
Lemont, IL, USA
{blenard,pershey,toonen,pupton,dwaldron,mzhang,childers,bbrickman}@anl.gov

Abstract

With the Argonne Leadership Computing Facility (ALCF) acquiring more HPE Supercomputers, each equipped with its own HPCM stack, alongside other operational programs, we devised a strategy to centralize monitoring data from these systems. This centralized system aggregates data from various sources and securely distributes it to different consumers, including various teams and platforms within the ALCF.

Initially developed for the deployment of the Polaris supercomputer, this strategy was expanded to include the Aurora supercomputer. We now feed data into a Data Lake, enabling multi-stage processing and the application of code that utilizes bitmaps to quickly identify issues within the Aurora system.

The contributions made by this paper are as follows:

- Discuss the centralization of data from HPCM's Kafka bus to a secure and centralized location for multi-consumer access and the harvesting of log data from ephemeral nodes
- Highlight and show the value of a data warehouse using row and columnar based storage
- Highlight the value of a data (Delta) lake for data processing as well as demonstrate the use of bitmaps and Apache Spark to efficiently search data generated by a supercomputer at Aurora's scale

While some of the concepts discussed are specific to HPCM systems, the principles of centralized data brokering using Kafka, and the use of data lakes and bitmaps, may be generalized to other platforms.

ACM Reference Format:
Ben Lenard,Eric Pershey,Brian Toonen,Peter Upton,Doug Waldron,Micheal Zhang,Lisa Childers, Bryan Brickman. 2025. Harvesting, Processing, Storing data from HPCM systems. In *Proceedings of Cray User Group (CUG '25).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The Argonne Leadership Computing Facility (ALCF) has hosted a large number of supercomputers over the years, frequently operating multiple systems concurrently. To monitor their health and fulfill Department of Energy (DoE) reporting requirements, the ALCF collects metrics and log data from these systems.

Given the multiple data consumers and diverse network segments, a centralized Kafka cluster was deployed to provide a secure, centralized source for access to monitoring data. While HPE Performance Cluster Manager (HPCM) publishes various metrics and log data to its own internal Kafka bus, other programs generate logs on ephemeral nodes (e.g., PBS Pro logs, XALT, and Prolog scripts), gathering these data presents a challenge. While these programs could be reconfigured or rewritten to write to journald (`systemd-journald`) or Kafka directly, this is not always a practical solution. As these programs often write directly to log files, ALCF uses FileBeat, a log transport agent, to transport log data in near real-time (NRT) to the Kafka cluster.

Once centralized in the Kafka cluster, data are transformed into usable information. This centralized approach avoids many-to-many integration problems. While a single database may seem ideal, different database software offer unique benefits; some are more appropriate for text searching, while others are optimized for time series data.

We provide insights into our centralizing data from HPCM's Kafka bus to a secure location for multi-consumer access and harvesting log data from ephemeral nodes as well as utilizing a Delta Lake and data warehouse for debugging systems at the Exascale and report generation respectively.

Additionally, we introduce our deployment of a data lake (Delta lake) to store and clean data from Aurora. A data lake is a centralized repository that stores, processes, and secures large amounts of data in its original form. In our case, the original form may be encoded using Apache Avro, thus requiring decoding and storage. We opted for a Delta lake for its features like Atomic, Consistent, Isolated, and Durable (ACID) transactions and schema enforcement.

A single supercomputer may not necessitate an extensive messaging bus for replication, log gathering, and data processing. However, the ALCF's multiple supercomputers and systems require an extensive messaging bus for data acquisition and processing. Different systems are likely to have different software versions and reside on other network segments. Our goal was to eliminate many-to-many integration complexity by providing a centralized messaging bus, allowing all data consumers a single endpoint for connection, regardless of the systems' data. Each application only requires access to the appropriate Kafka topic, greatly simplifying the integration process.

To centralize data from each system's HPCM Kafka bus, we use Kafka MirrorMaker to clone Kafka topic data from the HPCM system into the ALCF centralized system.

While direct connection to each system's HPCM Kafka bus is possible, managing user access logistics and tracking increases

CUG '25, New York, NY

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

with each additional system. This paper covers the deployment of these technologies and the design decisions made during our environment's implementation.

2 Terminology

Event: For the purposes of this paper, an event occurs when a supercomputer control system changes the state of a given node. This state change may be the result of a failure, either with the given node or the infrastructure, such as network, power, or cooling that supports the particular node.

RAS Events: A Reliability, Availability, and Serviceability (RAS) event happens when a component experiences a failure on the hardware or software of the system. The IBM Blue Gene [14] as well as the Cray XC [16] systems provide logging for capturing instances when something has failed.

MTTI: Mean Time to Interrupt is defined as the average outage time (scheduled or unscheduled). It is also known as Mean Time Between Interrupt:

$$\frac{TimePeriod-(ScheduledOutages+UnscheduledOutages)}{Count_{ScheduledOutages}+Count_{UnscheduledOutages}+1} \quad (1)$$

MTTF Mean Time to Failure is defined as the time, on average, to an unscheduled outage on the system or system component:

$$\frac{TimeInPeriod-UnscheduledOutages}{Count_{UnscheduledOutages}+1} \quad (2)$$

Overall Availability: The overall availability is the percentage of time a system is available to users without interruption. Outage time reflects both scheduled and unscheduled outages:

$$\frac{TimePeriod-UnavailableDueToOutages}{TimePeriod} * 100 \quad (3)$$

Scheduled Availability: Scheduled availability is the percentage of time a designated level of resource is available to users, excluding scheduled downtime for maintenance and upgrades:

$$\frac{TimePeriod-UnavailableDueToOutages}{TimePeriod-UnavailableDueToScheduledOutages} * 100 \quad (4)$$

Utilization: Utilization is the amount of time the system was used divided by of the total available time:

$$\frac{Core\ hours\ used\ in\ period}{Core\ hours\ available\ in\ period} * 100 \quad (5)$$

Bitmap and Bitmask: Bitmasks are used to represent machine node states, allowing the use of simple operations such as 'xor', 'and', and 'or' when comparing two bitmasks against each other. Using these simple operations reduces the amount of computational resources required. The bitmask shows which components are dependent on a given event. The involved nodes are represented by setting their bits in the bitmask. An example of bitmask usage is combining two bitmasks with an 'and' operation. If all of the bits in the result are zero, there is no node intersection between the events.

The order in which the bits refer to underlying hardware is encoded in the bitmap. It is beneficial to keep the index of the node close to the index of the related network links, because the node and its corresponding network links are almost always used simultaneously. Closeness is given by bits that are set near each other, such as 8 bits set in a row [24-31]. An example of bits set further apart would be: 8 bits set [24-24], [28-28], [32-32], [75-75], [81-83], [275-275]. Closeness enables us to compress the masks by representing them as intervals rather than a set of single bits.

Location to Mask: Location to Mask (LTM)[15] maps a node in the machine to the bitmask that represents the machine as in a supercomputer. For example, 4 nodes can be represented by the bitmask of '0000' with one bit per node; the first node maps to the first bit (the most significant bit, or MSB), the second node to the second bit, and so on. A value of "1" at the index of a specific node, indicates that that node is involved in the event described by the bitmask. For example, bitmask '1010' represents an event that involves node #0 and #2. The bit positions are also mapped to the node names, for example `nid00000` and `nid00002` in the case of the above example event. In other words, each compute node can be mapped to a bit in the mask.

ETL: In data warehousing, Extract Transform and Load (ETL) commonly refers to the process of extracting target data, transforming data into the structure required by the organization, and loading the data into the database. In the context of this paper, we parse the logs from the supercomputer control systems looking for error events, transform the data, and load the data into our system.

Time and Space correlation: On any given ALCF system, a job can consume the whole system or part of the machine (e.g. not all of the compute nodes are dedicated to the job.) Time and Space Correlation helps identify which jobs are impacted by an interrupt, since an interrupt can affect only a portion of the machine. For example, if a key component that interconnected several nodes failed and was replaced, this allows us to understand the impact of that component's failure during the relevant given time frame.

Data Lake and Delta Lake: A Data Lake [11] is a centralized repository designed to store large-scale structured, semi-structured, and unstructured data. It allows for the storage of vast amounts of raw data in its original form, eliminating the need for restructuring. This approach enables the data to be reprocessed multiple times as requirements evolve. The primary advantage of storing disparate information in a data lake is centralization of the data.

Data lakes have faced challenges related to inconsistent data, data reliability, and poor performance. Armbrust et al. [13] introduce the concept of a Delta Lake, which offers significant advantages over traditional data lakes. Delta Lakes address several limitations of traditional data lakes, including data consistency, reliability, and scalability. They provide ACID (Atomicity, Consistency, Isolation, Durability) guarantees, ensuring that data within a Delta Lake is consistent and reliable. Additionally, they provide scalable metadata handling by using a transaction log. Delta Lakes also support schema enforcement and data versioning. Because Delta Lakes use S3-style object storage, they benefit from a widely adopted storage standard, making them compatible with many applications.

Apache Parquet: Apache Parquet [4] is a columnar storage file format optimized for data processing and analytics. Since it is a columnar storage format, the whole row of data does not need to be read, only the values for the desired column. This greatly reduces the amount of I/O operations required as only the needed columns are accessed. Parquet files also support different forms of compression, as well as complex data types such as nested structures, arrays, and maps.

Apache Spark: Apache Spark [5] is a distributed computing framework designed for big data processing and analytics. Since

Spark is designed around in-memory data processing, it is faster then traditional disk-based Big Data frameworks like Apache Hadoop.

Apache Kafka and MirrorMaker: Apache Kafka [6] is an open-source distributed event streaming platform; it is designed to handle high-throughput, fault-tolerant, and scalable messaging. Kafka enables applications to publish, process, store, and subscribe to event streams asynchronously. Kafka has the ability to use Secure Sockets Layer (SSL) as well as Role Based Access Controls (RBAC) to implement Access Control Lists.

Kafka contains various components:

- Broker: A Broker is a server in a Kafka cluster that stores incoming data and serves client requests. A Kafka cluster typically consists of multiple brokers for performance as well as fault tolerance.
- Topic: A topic is a feed name where messages are published, akin to a database table.
- Partitions: Each topic has one or more partitions. Using more than one partition enables parallelism at the cost of ordered messages. The topic is spilt into subtopics and spread across multiple brokers.
- KRAFT and RAFT: Raft is a consensus algorithm designed to manage a replicated log across distributed systems in a simple and understandable way. It ensures that multiple nodes in a cluster agree on a consistent state, even during network failures. Raft is commonly used in distributed databases, storage systems, and coordination services. KRaft is the Kafka implementation of RAFT. ZooKeeper is another RAFT service provider previously used by Kafka. When using KRaft, each partition is one of three states: Leader, Follower or Candidate. If a broker fails, its partitions that were in the leader state get assigned to other brokers with up-to-date replicas.
- Replicas and In-Sync Replicas: In a production environment each topic's partition should have at least two replicas assigned to a different broker should one broker fail. The primary replica where writes are occurring is called a leader and the non-leader replica is a follower. Should the broker containing a leader fail, a follower becomes the new leader after an election is held by the RAFT protocol. Another advantage to having multiple replicas is that consumers can also read from the followers.
- Producers: They are responsible for publishing messages to one or more Kafka topics.
- Consumers: They subscribe to topics and consume messages from that given topic.
- Apache AVRO: Apache Avro [7] is a data serialization system that provides efficient, compact, and schema-based serialization of data. It is widely used in big data and distributed systems for data exchange between applications. It is not a part of Kafka but is the data format used for our messages.
- Apache MirrorMaker: Apache MirrorMaker is a tool used to replicate Kafka topics from one Kafka cluster to another.

IBM Db2 and BLU: IBM Db2 is a relational database management system (RDBMS) that has historically stored data in a row format; BLU [17] provides the ability to store the data in a columnar format. Depending on the workload, columnar format can provide superior performance.

Filebeat: Filebeat[8] is a lightweight log shipper developed by Elastic, and is used to collect, parse, and forward log data from various sources to destinations like Elasticsearch, Logstash, or Kafka.

Polaris: Polaris is a Hewlett Packard Enterprise (HPE) supercomputer system at the ALCF. It features 560 AMD EPYC Milan processors and 2,240 NVIDIA A100 Tensor Core GPUs, delivering a peak performance of around 44 petaflops. Polaris has 560 compute nodes with Lustre storage.

Sirius: Sirius is an ALCF-internal machine used to stage and test changes intended for the production Polaris deployment.

Sunspot: Sunspot is an ALCF-internal machine used to stage and test changes intended for the production Aurora deployment.

Crux: Crux is a petascale class supercomputer composed of 256 nodes, each node having 2 AMD EPYC Milan processors with HPE Slingshot 11 fabric endpoints. Crux enables users to execute pre- and post-processing tasks to gain further insights into their computational results.

Aurora: Aurora [2] is an exascale class supercomputer composed of 10,624 compute nodes and each compute node having 2 Intel Xeon CPU Max Series processors: 64GB HBM on each, 512GB DDR5 each; 6 Intel Data Center GPU Max Series, 128GB HBM on each, RAMBO cache on each; Unified Memory Architecture; 8 SlingShot 11 fabric. It ranked number 3 on the Top 500 list in the Fall of 2024 [3].

XALT: XALT [9] is an open source High Performance Computing (HPC) application build and execution tracking framework. Highly configurable, XALT produces Javascript Object Notation (JSON) records detailing the libraries and other runtime information that enable users to build and execute their HPC applications.

PBS Pro: PBS Pro[1] is a distributed workload manager that schedules and monitors the execution of applications. Computing resources, such as time and nodes, are allocated to applications based on a job descriptions submitted by the users and scheduling policies defined by the facility.

- Accounting Logs: Well-defined log of the usage of resources by jobs on the system.
- Server (pbs_server): the resource manager and handler of requests from all other systems, such as pbs_mom, pbs_sched, and pbs_comm.
- Server Logs (pbs_server): The log of the PBS server which contains the requests from all the other systems, resource state changes and overall status of the system.
- Scheduler (pbs_sched) Logs: Contains a log of jobs that it is attempting to schedule on the system.
- Mom (pbs_mom): The component in PBS environment that executes the launching of jobs, monitors their status, and updates the server on the resource and job status.
- Mom Logs (pbs_mom): The log file tracks the activity of PBS mom, detailing job executions, lifetime and monitoring events.
- Comm (pbs_comm): The component that sits between the pbs_server and pbs_mom that aggregates communication on a large system.
- Comm logs (pbs_comm): The log of the comm tree and all the pbs_mom components that are registered with it.

3 Related Work

In Adamson et al.[12], the Oak Ridge Leadership Computing Facility (OLCF) describes using a centralized Kafka cluster for telemetry data collection and exporting to backend storage. The ALCF solution is similar but additionally incorporates data from non-HPCM components and has a more defined data ingestion process.

The Argonne Leadership Computing Facility (ALCF) developed the Operational Data Processing System (ODPS)[15] to facilitate the storage and retrieval of large-scale machine operational data, specifically for monitoring supercomputers and their system metrics. ODPS employs a bitmask-based storage method, where each event is stored as a single database row, with affected nodes represented using a bitmask. This approach enables efficient machine state tracking and node monitoring while optimizing reporting calculations and correlating impacted jobs during failures. The bitmask representation also enhances computational efficiency by leveraging bitwise operations such as XOR, AND, and OR. Furthermore, ODPS identifies intersections between jobs and component failures using bitwise AND operations.

In ODPS, an event is defined as any occurrence that impacts a currently running job on the system. Since most jobs do not utilize the entire system, multiple independent jobs may run simultaneously. This event-driven model minimizes data redundancy and improves query performance, making it easier to analyze operational patterns efficiently while enhancing scalability in handling vast amounts of machine-generated data.

Written in Python 3, ODPS is platform-independent and integrates with relational database management systems (RDBMS) such as Db2 and MySQL. It offers a REST API for data consumers and a Low-Level API for administrators, providing seamless data access without requiring knowledge of the underlying data structures.

The ALCF also outlined the method for harvesting data from HPC systems using ETL processes. The ETL workflow involves parsing text files via Python applications and loading the extracted data into a traditional row-based RDBMS. In production, the system loads data into a Db2 database, though the framework is compatible with both Db2 and MySQL.

Bitmasks play a crucial role within both the database and the ODPS API. Since Theta had a relatively small bitmask of 4392 bits, compressed in the database down to 1098 bytes, converting the bitmasks for queries was relatively fast using bitwise operations. More work would have been needed to use build-in bitwise operations of Db2 and MySQL, but was deemed unnecessary with such a small mask.

Querying a row loads an entire database page; however, because ODPS operates on a small subset of data, it eliminates the need to transfer large volumes of data across the network. Once events are loaded, Python processes the bitmasks by converting them into extended NumPy arrays with an unsigned 8-bit integer dtype and overflow protection. NumPy's built-in bitwise operations further optimize computational performance.

At its core, ODPS relies on an abstraction layer built around a Python class called Location to Map (LTM). LTM provides a standardized interface for processing machine operational data using bitmask representation, even as system configurations evolve.

The LTM layer consists of two key tables: Location To Mask Map (LTMM) and Location to Map Base (LTMB). As the system changes over time, a tracking mechanism is necessary. The LTMB table records machine metadata over specific time periods, including node types and the smallest machine unit mapped to a bitmask. Each machine has a corresponding LTMB record for a given timeframe, capturing metadata about the overall system. Meanwhile, the LTMM table stores details about individual nodes relative to their corresponding LTMB entry.

Whenever a machine undergoes a size change, its nodes are logged in the LTMM table. Essentially, LTMB provides a contextual snapshot (or incarnation) of the machine, while LTMM tracks node-level metadata. For example, on Theta, LTMM stores attributes such as Node ID (NID), NID name, IP address, and component name (CName). These are maintained in machine-independent structures, with actual name mappings (e.g., NID) recorded in the LTMB.

To ensure metadata remains up to date, the LTMB is refreshed every two minutes on Theta via an ETL process, updating only the end timestamp while preserving all other fields. LTMM entries remain immutable to maintain historical data integrity and preserve bitmask mappings. If system attributes change beyond what LTMB can accommodate, a new LTMB is created along with a fresh set of LTMM entries.

Although maintaining a live copy with a change journal in another table is possible, ODPS is primarily designed for historical and range-based queries. This enables seamless analysis across system size changes and structural modifications over time.

4 The ALCF Environment and Motivation

The ALCF has multiple HPE HPC systems that run the HPCM management stack. Each of these systems is siloed, in the sense that monitoring and operational data only exists within its own system. There are several limitations to the siloed approach, including local storage limitations for metric and operational data within each system, the need for aggregate reporting across the facility, the need for other facilities to use a given system's data, and the need to manage the integrations of multiple consumers of these data from an application and security standpoint.

While some data are operational in terms of real-time monitoring, a subset of the data is included in the Operational Assessment Report (OAR), an annual facility report containing various operational metrics. OAR reporting metrics include utilization as well as availability of systems within the ALCF.

Within the ALCF, we currently have four HPE systems that have the HPCM management stack deployed, as well as other systems that do not utilize HPCM; of the four HPCM systems, two are reportable systems, Aurora and Polaris. Reportable in the sense that their metrics are within the OAR. These four systems also may run different releases of the HPCM software which means that the monitoring stack could differ. In addition to this, when HPE changes the monitoring portion of HPCM, the metrics may be reset depending on the changes within HPCM, for example moving from one database backend to another.

In addition to the metrics and log data provided by HPCM, there are other logs required. These data include the log data from our scheduler, PBS Pro, and XALT, and other software that we have

written for our systems. In terms of PBS Pro, there is a central scheduling daemon for PBS Pro and individual logs on each compute node, namely the MoM logs, along with the logs from XALT. Each of our compute nodes does not have persistent storage. At the scale of Aurora, a NFS share and parsing of 10,000 log files in near real-time would be impractical.

Due to these differences in systems' monitoring, we needed a method to aggregate metrics and log data in a uniform manner in order to preserve the data over time. We also needed a way to store the harvest and store the data so that our business intelligence team can generate the metrics for annual reporting, as well as provide internal analytics that span HPC systems.

Within the ALCF, we have a Business Intelligence (BI) team that has a data warehouse that utilizes Db2 11.5 as a relational database backend as well as Pentaho, Grafana, and other proprietary reporting software for the report generation. A feature of Db2 is the BLU engine which is a columnar store; columnar stores can be significantly faster then traditional row based storage depending on the workload. In terms of metrics, most analytics for that type of workload are aggregations, so a columnar store is faster since its only reading the metrics required for the process; thus is a significant reduction in IO. The ALCF has used previously use Db2 BLU to analyze the metrics of our General Parallel File System (GPFS) filesystems.

While the BI team provides the required metrics for annual reporting, the data is also required for debugging and troubleshooting; while dashboards provided by the HPCM monitoring stack can be helpful, they do not reflect the state of the facility as a whole. The need for processing logs and assembling a timeline of events is key to understand the health of an HPC system regardless of its management stack as there are many interconnected components that must be monitored to ensure the stable operation of systems within the facility.

The flow of the data can be visualized by figure 1 where you can see the different compute systems and the flow from their Kafka clusters to the centralized cluster via MirrorMaker. Once it is on our cluster, the two different consumers of the data, Business Intelligence and ODPS 2.0. Note that the picture shows the Test and Development (T&D) systems connected similarly to production, however sometimes these systems are reconfigured in a different manor for testing.

5 Deployment within our Environment

Within the ALCF, we needed a solution to aggregate all metrics and log data from Sirius, Crux, Polaris, Aurora, as well as non-HPCM systems, into a centralized message bus so various consumers of the data can access the data in near real time; we also needed a way to enable the use of Access Control Lists (ACLs) for per topic access, as well as SSL to ensure network security. We needed a way to decouple the HPCM systems from each other in terms of metric and log gathering. The consumers of the data include software used by the BI team and other software developed by the ALCF, such as ODPS 2.0.

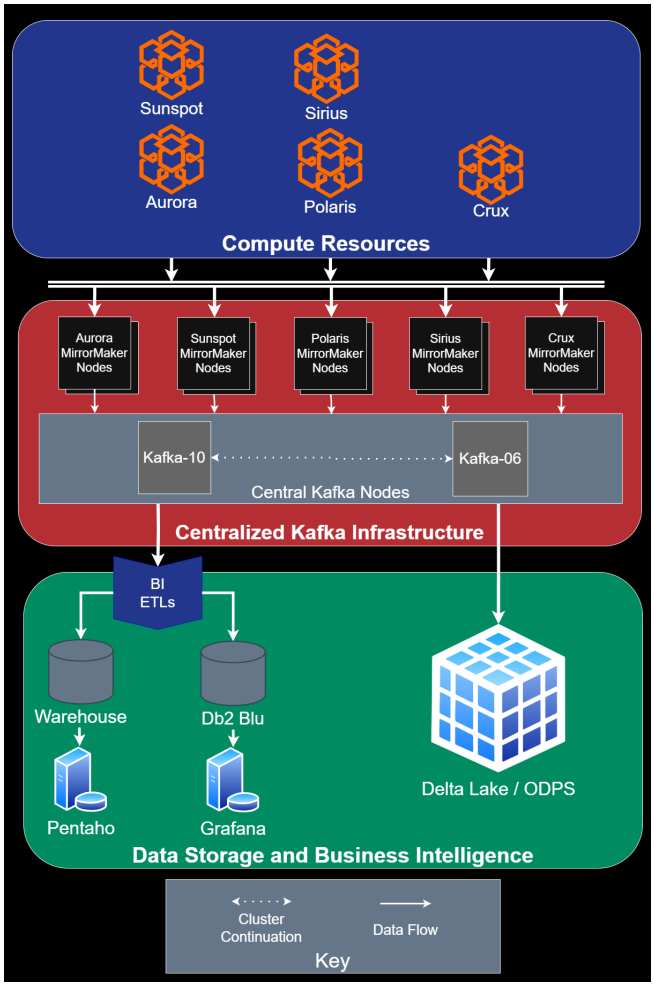

Figure 1: Data Flow within the ALCF

5.1 Harvesting the Data

Each HPCM system has its own Confluent Kafka cluster (HPCM includes a proprietary version of Kafka) in which HPCM components publish metrics and data so that other HPCM components can store them into OpenSearch, Victoria Metrics, or TimeScale DB. Each of these Kafka clusters is independent of one another. HPCM scripts for installation and management are tightly coupled with the software.

Starting with the acceptance testing of Polaris, the first HPCM system within the ALCF, we deployed a centralized five node Apache Kafka (version 3.3) Cluster within our VMWare environment; we utilized Apache Kafka MirrorMaker to mirror the topics from the Polaris HPCM Kafka Cluster into our centralized cluster. In this way multiple consumers had access to the data within the various topics without requiring direct access to the Polaris HPCM Kafka Cluster. Direct access to the HPCM Kafka Clusters would increase the load on the HPCM managed Kafka nodes and would destabilize the system.

HPCM is not involved in processing log data generated by XALT and PBS Pro that the ALCF needs for reporting and troubleshooting purposes, and is thus must be ingested separately. To do this, we

use Filebeat. We configured it on each of compute node in order to ingest the log data generated by these two applications and publish data into topics we created on the HPCM Kafka Cluster. From the this cluster, the topics are then mirrored to the centralized ALCF Kafka.

Polaris was the first ALCF supercomputer to be GPU based rather than CPU based. It also was the first ALCF system to use HPCM. While Polaris is only 560 compute nodes, it aided with development of processes to curate the new data, as well as evaluate the data being generated by HPCM's components. Aurora's architecture is different to Polaris, but was similar enough to gave the ALCF an idea of what to expect prior to Aurora.

As we moved forward from Polaris and acquired, or repurposed, HPCM systems (i.e. Crux, Sirius, and Aurora), we came to understand that we needed to have almost continuous availability for the centralized Kafka cluster as well as the MirrorMaker services. This key so that near real-time data acquisition can continue even if there is failure or system maintenance. In other words, there are very few times when all of ALCF's HPCM systems are down at the same time.

As we began to bring Aurora on-line, and added the other HPCM systems' data to the centralized cluster, it was clear that our existing VMWare environment was unable to handle the volume of data generated by these systems. Thus, we replaced our existing Kafka cluster with physical hosts. In addition to this, since Kafka is moving away from ZooKeeper for quorum and metadata management, we implemented Kraft (Kafka Raft). As part of this change, we increased the number of nodes to ensure that we had more then the minium nodes in the quorum in order to provide redundancy. We have a five node Kafka cluster, where all five nodes are voting member in the quorum. In this way two nodes can be down and the cluster still has the majority of the quorum members available. This is in addition to the existing settings, such as enabling SSL, access control lists (ACLs), and manual topic creation for the topics that we are replicating. We choose manual topic creation since the HPCM systems' Kafka clusters do not impose any type of ACLs, if a topic is changed, added, or deleted, on the HPCM Kafka cluster it would propagate to the centralized cluster which would have negative effects.

When we manually create topics on the centralized Kafka cluster, we make sure that our topic and the HPCM topic match in terms of partitions. The partition counts must be identical so that MirrorMaker can clone the topic. MirrorMaker spawns one task per topic partition in order to act as a consumer on the HPCM side or a producer on the centralized cluster side. While we need to match the partition count per topic, we need not match the replica count for that topic. Replica count a function of how many servers are within the cluster and design objectives. In the centralized Kafka cluster, we have five servers that each vote within the quorum. Because of this, in our centralized cluster, we need a replication factor of three so that in the case of two nodes failing the topic and partition are still be available for use. As stated earlier, one of our design objectives is maximum availability to support all systems that require the Kafka cluster.

MirrorMaker, has been rolled into Kafka Connect. Kafka Connect's architecture allows the deployment of MirrorMaker to be highly available by running multiple copies. At the ALCF, we deploy a pair of MirrorMaker instances per HPCM system so that we can patch one of the two instances at a time. These deployments range from VMs to physical systems depending on the size of the HPCM system; for example, for Aurora we have two physical servers where they move approximately 1 Gbps from the Aurora Kafka cluster to the centralized cluster.

This deployment method has allowed the ALCF to perform patching of the OS and firmware of the physical systems, if applicable, without having an outage or performance degradation.

5.2 Processing and Storing the Data

The processing and storage of data is the second major component of this paper. At the ALCF, there are two different requirements of the data being generated by these systems, a historic view for reporting and an operational view, ether for debugging, or for Acceptance testing and failure analysis. Both priorities are equally important.

5.3 Business Intelligence

The ALCF Business Intelligence (BI) team has used a Db2 for its reporting for the OAR and other information requested by the Department of Energy or ALCF management. In the past, the BI team would mine the data via reading log files, and transform the data to storing it in a Db2 database in row format. The ETL process is written in Python and utilizes Jenkins for periodic execution. Starting a few years ago, the BI team started using Db2 columnar engine for storing metrics since this format provided better query performance for running this aggregation; thus this database became a hybrid workload in the sense that it used row and columnar row storage for the different pieces of data from the HPC systems. This data, managed by the BI team, is in a data warehouse that spans over a decade and several terabytes.

As we saw the value of the increased performance of the Db2 BLU engine, the BI team began utilizing the BLU database heavily to store different metrics about Polaris and Aurora. Over time, the BLU database grew from a few TB in size to over 62TB in size. The BLU database is still part of the BI team, however the instance is optimized for columnar data storage in order to provide the best performance.

These databases utilize 32Gb Fiber Channel protocol to the storage system, which is a pair of Netapp AFF400's.

5.4 ODPS 2.0

While the ALCF developed ODPS 1.0, there were various limitations to ODPS since it was built around a RDBMS. As the ALCF prepared for the acceptance testing of Aurora, the ALCF used ODPS 1.0 to help find scaling issues with PBS Pro. As a result, it was deemed that a solution was needed that was faster, and more scalable, than the current implementation of ODPS. The other significant issue was high memory usage by the ODPS software; the node has 512GB of RAM and the process would consistently be killed by the OS for exceeding the maximum amount of memory. In addition to this, we also needed to support a bitmask of 87671 bits and larger to support the scale of Aurora.

5.4.1 Migration of Logfiles to a Message Bus. In ODPS, the ETL process initially relied on reading log files, processing them, and inserting the extracted data into a database. With ODPS 2.0, this workflow has transitioned to a centralized message bus powered by Apache Kafka. The Kafka cluster retains copies of messages from the system's message bus, while log parsers on individual nodes transmit log data in near real-time. Each Kafka topic is partitioned, enabling producers and consumers to leverage parallelism efficiently.

5.4.2 Migration of Python to Spark. The original ODPS design has a fundamental limitation that prevents it from efficiently supporting exascale-level processing. While ODPS successfully handled job-event intersections for Theta, scaling up to Aurora requires processing data for 10,624 nodes and their associated network links. When system logic is applied, the bitmask expands to 84,992 bits. More external hardware was included which expanded it again to 87,671 entries, significantly increasing processing time. Additionally, as an exascale machine, Aurora generates a much larger volume of data due to the higher number of jobs being executed and number of nodes that exist.

ODPS was built using Python and Pandas, which constrained its ability to parallelize computations. Given the continuous growth of HPC systems and the stagnation of CPU clock speeds, alternative methods were explored to enhance data processing and parallelization. Apache Spark emerged as the ideal solution due to its seamless integration with Kafka, offering built-in checkpointing, error handling, and schema management. Furthermore, since Kafka topics are partitioned, scaling up the number of Spark workers enables concurrent processing of partitions, significantly improving efficiency.

5.4.3 Migration of the RDBMS to a Delta Lake. In terms of data consistency, RDBMS platforms offer ACID compliance and schema validation to ensure data integrity before insertion. These systems operate with pages, and some platforms, such as Oracle and Db2, support the creation of tablespaces with configurable page sizes, ranging from 4k to 32k. For rows that exceed the standard page size, a larger page can be used to prevent the row from being split, a process known as row chaining [10]. In contrast, Delta Lake in ODPS 2.0 allows the storage of columns or values larger than the 32k page size limit imposed by many RDBMS platforms. While data types like LOBs (Large OBjects) and CLOBs (Character Large Objects) could be utilized for this purpose, they are often inefficient for querying.

In the original ODPS design, the bitmask was stored in the RDBMS as a hexadecimal value, requiring serialization and deserialization with each read or write operation. Now, by storing the bitmask directly as a structure within Delta Lake as a column, we eliminate the need for serialization and deserialization, making the data immediately usable by the application. For ODPS 2.0 and the Delta Lake, it utilizes S3 protocal over traditional 100Gb Ethernet to the storage system, which is a pair of Netapp AFF400's.

The ALCF Delta Lake has 3 Levels that divide the data into levels.

- Level 0: Contains the streamed data from Kafka with the raw data that was produced using Filebeat or direct entry/creation. The data is encoded in either JSON or AVRO format. Metadata from Kafka's timestamp column is used to create a column called min15 which is that timestamp rounded down to the nearest 15 minute. Each topic resides in their own table and all are partitioned by the min15 column.
- Level 1: Contains the streamed data from Level 0 with the encoding, JSON or AVRO decoded. The Kafka metadata columns within the Level 0 message is renamed/prefixed with "kafka-". The Filebeat metadata within the JSON is renamed/prefixed with "filebeat-". Both the Kafka and Filebeats metadata structures are flattened by moving all subkeys of the column structure into the highest level of the structure. Remaining columns belong to the data itself and some are renamed to enhance their meaning. A log_timestamp column is added that is either the parsed log timestamp from the log message or using the Filebeat timestamp if no other time is found. Columns log_month(month of log_timestamp), log_hour(hour of log_timestamp) and error_flag_mask are added. The column error_flag_mask is based on the delta of the Filebeat timestamp against the log timestamp. If they are more than 4 days in the past, they are marked with a 2 and if they are more than 15 minutes in the future they are marked with 1. The error_flag_mask is useful in finding machines where the clock is off. UTC is used for all times, so the times should all be the same. The data is then partitioned by log_month, log_hour, and error_flag_mask and each topic is written into their own table with their own schema. There are also 2 other tables, zone and graph. The zone table contains all DNS entries, hourly if changed. The graph table contains the supercomputer machine structure as a graph with all node, cabinet, rack, and much more that is used to create bitmasks for Level 2.
- Level 2: Contains the streamed data from Level 1 into a single "merged" table that use one schema. This is a custom table that contains columns such as bitmask, jobid, taskid and other specific data that is needed by our searches. There are different translations based on each topic from Level 1 because all the data must end up in the same schema in Level 2. Since the data all is in the same table and same schema, it allows queries across all sources of data. A column source_name is added and is based on the source topic from Level 1. Each log line from all sources goes through IP and hostname resolution as well as bitmask creation and added to each row. The extra data, such as jobid, is extracted from the line, if it exists. A column called error_flags is added that is a mask that is used to mark the line if specific words are seen in that line. The data are then written and partitioned by the log_month, log_hour and source_name columns.
- Other Tables: Other tables exist such as "Job Run" that are used as search starting points. Job Run comes from Level 2 and is updated every hour by an ETL and contains all PBS job executions.

Since each of the levels stream from one to another, there is an average of 5 minute delay from when a log is written on a host system to being seen on in Level 2. This can be adjusted and is based on the cores available to move the data.

6 Reporting

6.1 BI

There are various examples of reporting that the BI team produces, from project usage to usage by science domain, and other reportable metrics. This reporting feeds a business intelligence tool called Pentaho which generates various reports. Another example is power consumption by Aurora. Power consumption by Aurora is stored within the Db2 BLU database and the timeseries data can be visualized within Grafana.

6.2 ODPS

The ODPS 2.0 system and the data it stores can allow one to correlate and find the intersection of various data points, or timelines within the system. For example, when there are issues with a job of the system (i.e. failure, would not start, hardware issues, etc), one can utilitze the system for job failure analyse. On a similar note, one can utilize ODPS 2.0 to find events tied to a hardware failure, or even a specific error across many days across all nodes. All of these examples of uses of ODPS 2.0 aide in the troubleshooting of the HPCM systems.

7 Conclusion

We have shown the usefulness of a centralized Kafka message bus to allow various applications to consume data from the varied HPC systems. We have shown the reporting that we store within a traditional data warehouse in row and columnar format. We have also shown how a Delta Lake can be utilized for the ODPS 2.0 application to aid in the troubleshooting of HPCM systems.

8 Future Work

In the future, we plan on optimizing ODPS 2.0 to improve upon its performance as we expect larger HPC systems within the ALCF. In addition to this, we are investigating new uses for the ODPS 2.0 system for these large systems. We also investigating other database platforms (such as Victoria Metrics) for use in by the BI team in generating reports.

Acknowledgments

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH1135.

References

[1] 2022. Altair PBS Professional 2022.1. https://2022.help.altair.com/2022.1.0/PBS%20Professional/PBSInstallGuide2022.1.pdf
[2] 2024. https://www.alcf.anl.gov/aurora
[3] 2024. https://top500.org/lists/top500/2024/11/
[4] 2025. https://parquet.apache.org/docs/overview/
[5] 2025. https://spark.apache.org/docs/latest/
[6] 2025. https://kafka.apache.org/
[7] 2025. https://avro.apache.org/
[8] 2025. https://www.elastic.co/beats/filebeat
[9] 2025. https://github.com/xalt/xalt
[10] 2025. https://www.orafaq.com/wiki/Chained_row
[11] 2025. What is a Data Lake. https://aws.amazon.com/what-is/data-lake/
[12] Ryan Adamson. 2023. Stream: A scalable federated HPC telemetry platform. https://www.ornl.gov/publication/stream-scalable-federated-hpc-telemetry-platform
[13] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3411–3424. doi:10.14778/3415478.3415560
[14] Gary Lakner, Brant Knudson, et al. 2013. *IBM system Blue Gene solution: Blue Gene/Q system administration.* IBM Redbooks.
[15] Ben Lenard, Eric Pershey, Zachary Nault, and Alexander Rasin. 2023. An Approach for Efficient Processing of Machine Operational Data. In *Database and Expert Systems Applications*, Christine Strauss, Toshiyuki Amagasa, Gabriele Kotsis, A. Min Tjoa, and Ismail Khalil (Eds.). Springer Nature Switzerland, Cham, 129–146.
[16] Greg Pautsch, Duncan Roweth, and Scott Schroeder. 2013. *The Cray® XC™ Supercomputer Series: Energy-Efficient Computing.* Technical Report. Technical Report.
[17] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.