Benchmarking Medallion-Architecture Data Pipeline in Cloud-Computing Environments

Tianyu Sun, Evie Mahsem, Marissa Reed, Nishanth Prasad

I. Abstract

This project explores the trade-offs involved in deploying a medallion-architecture data pipeline across local and cloud-based environments. Using a consistent dataset and architecture, we benchmarked performance on a local machine, a single-node Jetstream2 instance, and multi-node distributed clusters. The pipeline was implemented using Apache Spark and Hive to process data across bronze, silver, and gold layers. Our results show that the single-node cloud instance minimized overall runtime, while two-node clusters offered the fastest raw data ingestion due to improved parallel I/O. In contrast, the local machine performed best in write-intensive stages due to high-speed local storage. These findings highlight that while cloud and distributed deployments offer greater scalability and parallelism, they also introduce constraints such as storage limits and coordination overhead. Therefore, careful consideration must be given to workload characteristics, resource provisioning, and storage capacity when choosing an environment for pipeline deployment.

II. Introduction

In today's data-driven world, the physical reality we observe is increasingly captured, quantified, and interpreted through data [1], [2]. However, the data we collect is often incomplete, inconsistent, or noisy—resulting from limitations in measurement devices and the inherent fallibility of human operators [3]. To transform this fragmented information into actionable insights, robust and scalable data processing pipelines are essential [2]. These pipelines must not only support large-scale ingestion and cleaning, but also enable efficient transformation, enrichment, and delivery of insights.

Cloud computing has emerged as a powerful paradigm for building such pipelines, offering on-demand scalability, distributed computing capabilities, and a suite of tools for managing complex data workflows [4]. One such architectural pattern is the medallion architecture, a multi-layered approach popularized by platforms like Databricks [5]. It structures data flow through three successive stages: [5]

- Bronze: Stores raw, unprocessed data.
- Silver: Contains cleaned and enriched data, where it is transformed into a more structured format.
- Gold: Contains aggregated and curated data, optimized for use cases.

Email: ts19@iu.edu
Email: rmahsem@iu.edu
Email: mkinsel@iu.edu
Email: nismahen@iu.edu

While this architecture offers conceptual clarity and modularity, implementing it in practice—especially outside of managed environments like Databricks—poses several challenges including:

- Limited resources: Lack of computational power and storage capacity makes it inefficient in handling large datasets and creates bottlenecks.
- Scalability: Scaling data processing tasks on a local machine is constrained by the resources the local machine has access to, making it difficult to handle increasing data needs.
- Fault Tolerance: Local environments may not have mechanisms for fault tolerance, which is associated with the risk of data loss or corruption.
- Parallel Processing Constraints: Parallelism is challenging to achieve on a local machine, which limits the efficiency of the pipeline. [6], [7]

This project benchmarks the medallion architecture's performance across three deployment environments: a local machine, a single-node Jetstream2 cloud instance, and a multi-node distributed Jetstream2 cluster. Using a dataset of NASA lithium-ion batteries, we construct identical medallion pipelines in each setting using Hive for storage and PySpark for processing. By measuring runtime, CPU/memory usage, and disk throughput, we quantify how different environments impact pipeline efficiency, scalability, and resource utilization.

Our goal is to provide practical insights into the trade-offs between local, cloud, and distributed deployments of medallion-architecture pipelines. Ultimately, we demonstrate that cloud and cluster environments significantly improve performance and scalability.

III. Background and Related Work

A. Background

Modern data pipelines must ingest, cleanse, and serve ever-growing volumes of heterogeneous data with minimal latency and maximal reliability. The medallion architecture's layered approach—Bronze (raw), Silver (cleaned), Gold (curated)—provides a clear roadmap for incrementally improving data quality and enabling modular analytics on a unified lakehouse platform [5], [8]. However, most practitioners deploy this pattern in managed services like Databricks, leaving open the question of how it performs on self-managed local and cloud resources where storage limits, network variability, and resource contention can drastically alter end-to-end behavior.

B. Related Work

Benchmarking efforts have compared big-data engines across diverse environments, but rarely within a complete medallion-architecture pipeline. Poggi et al. [6] evaluated BigBench queries on multi-cloud Hive and Spark clusters, highlighting variability across data scales and file formats but stopping short of layered pipeline analysis. Other work contrasts Spark with MPI/OpenMP implementations on classic MapReduce tasks, finding MPI variants sometimes outperform Spark by an order of magnitude in raw throughput but at the cost of higher development complexity [9], [10]. Studies in healthcare analytics contexts further demonstrate that containerized PySpark pipelines can reduce ETL runtimes by up to 10× compared to traditional approaches [11], [12]. To our knowledge, no study has systematically measured runtime, CPU usage, and disk I/O throughput across Bronze, Silver, and Gold stages on both local and cloud-based deployments—motivating the comprehensive benchmarking presented here.

IV. Design and Implementation

We designed our project after the widely utilized platform Databricks, following the medallion architecture [5], and used many of the same techniques (See TABLE I). While planning out our tables for each layer, we didn't want to create a gold table for the sake of it. We had thought up a business use case for it. Since the dataset was composed of three different kinds of batteries:

1) Recommissioned Batteries
2) Regular Alternative Batteries
3) Second Life Batteries

The following thought came to mind: Would it be possible to predict which kind of battery it is? Using this thought, we decided on our final table structure.

Bronze Layer:

1) recommissioned_bz
2) regular_alt_bz
3) second_life_bz

Silver Layer:

1) recommissioned_sl
2) regular_alt_sl
3) second_life_sl

Gold Layer:

1) all_batteries_gold

For the bronze and silver layer, it was a one-to-one relationship from raw data into those tables, but following the stages mentioned in the introduction, while for the gold table, it was a union between the three silver tables into the single gold table.

Apache Hive Metastore and Apache Spark were used in tandem to implement each layer and their transformations, while Apache Hadoop leveraged many clusters to distribute the load. It is quite trivial to make Apache Spark use the Hive Metastore. In the first listing, you can see that all that is needed to change the default catalog implementation to Hive and to enable the Hive support option.

Listing 1: Spark session initialization and Hive table cleanup

```
spark = SparkSession.builder \
    .appName("Dynamic Allocation App") \
    .config("spark.sql.catalogImplementation",
         "hive") \
    .config("spark.executor.memory", "8g") \
    .config("spark.driver.memory", "4g") \
    .config("spark.executor.cores", "2") \
    .config("spark.executor.instances", "4") \
    .config("spark.sql.shuffle.partitions", "
        200") \
    .config("spark.dynamicAllocation.enabled",
         "true") \
    .config("spark.dynamicAllocation.
        minExecutors", "1") \
    .config("spark.dynamicAllocation.
        maxExecutors", "4") \
    .enableHiveSupport() \
    .getOrCreate()

def clean_hive_tables():
    for layer in ["bronze_layer", "
        silver_layer", "gold_layer"]:
        spark.sql(f"USE {layer}")
        tables = spark.sql("SHOW TABLES").
            collect()
        for row in tables:
            table = row['tableName']
            print(f"Dropping {layer}.{table}")
            spark.sql(f"DROP TABLE IF EXISTS {
                table}")
```

Using the Spark context for the creation of each layer, we also stress-tested the pipeline. This was accomplished by looping through the ingestion to gold creation thirty times while tracking all of the pipeline metrics (See Listing 2).

Listing 2: Pipeline benchmarking loop with timing and metrics

```
result = []
for run in range(30):
    total_start_time = time.time()
    t0, cpu0, mem0, read0, write0, throughput0
         = ingest_hdfs(client, local_dir)
    t1, cpu1, mem1, read1, write1, throughput1
         = bronze_creation(local_dir, spark)
    t2, cpu2, mem2, read2, write2, throughput2
         = silver_creation(spark)
    t3, cpu3, mem3, read3, write3, throughput3
         = gold_creation(spark)
    total_end_time = time.time()

    total_time = round(total_end_time -
        total_start_time, 2)
    result.append([
        run+1, total_time,
        t0, cpu0, mem0, read0, write0,
            throughput0,
        t1, cpu1, mem1, read1, write1,
            throughput1,
        t2, cpu2, mem2, read2, write2,
            throughput2,
        t3, cpu3, mem3, read3, write3,
            throughput3
    ])
```

To be able to track all of the pipeline metrics, we needed a function for this purpose, which is shown within the final

Category	Techniques/Tools	Description
Cloud Infrastructure	JetStream2	Cloud HPC environment providing VMs with configurable CPU, RAM, and high-speed networking
Local Infrastructure	Personal Desktop	Single-node Hadoop/Spark cluster on a 12-core, 32 GB RAM desktop with NVMe SSD storage
Distributed Storage	Hadoop Distributed File System	Fault-tolerant block storage for staging raw data and providing scalable I/O throughput
Data Processing	Apache Spark	In-memory distributed compute engine using DataFrame APIs and DAG scheduling for ETL
Data Warehouse	Apache Hive Metastore	Metadata repository storing table schemas, partitions, and statistics for Hive/Spark SQL
Programming Models	PySpark	Python API for Spark enabling DataFrame and SQL operations via Py4J integration
Programming Models	Jupyter Notebook	Interactive notebook IDE for exploratory Spark code execution and inline visualization
Programming Models	SQL	ANSI SQL used in Spark SQL and HiveQL for declarative data transformations and queries
Data Ingestion	InsecureClient	Part of the HDFS Python library that allowed connection to copy data from local to HDFS
Data Visualization	Jupyter Notebook	Used Matplotlib and pandas in notebooks to plot runtime, CPU, and throughput metrics
Cloud-native Features	Spark Dynamic Allocation	Elastic executor scaling between a configured minimum and maximum based on workload
Performance Monitoring	psutil + NumPy	psutil for OS metrics collection and NumPy for computing summary statistics
Development Environment	VS Code + venv	Visual Studio Code editor with Python venv to isolate dependencies and run scripts

TABLE I: Cloud Computing Technologies Used in the Medallion-Architecture Data Pipeline

listing. The pipeline metrics for each layer and all thirty rounds were outputted into a CSV file, allowing for the creation of analysis plots (See Figs 1-4 within the Evaluation section).

Listing 3: Function to log CPU, memory, and disk I/O usage during a pipeline stage

```
def log_system_usage(cpu_list, memory_list,
    disk_read_list, disk_write_list,
    start_disk_read, start_disk_write):
    cpu_usage = psutil.cpu_percent()  # CPU
        usage instantaneous
    # cpu_usage = psutil.cpu_percent(interval
        =0.01) # CPU usage over interval of
        time
    memory = psutil.virtual_memory()
    memory_usage = memory.percent
    disk_io = psutil.disk_io_counters()
    disk_read = disk_io.read_bytes
    disk_write = disk_io.write_bytes

    disk_read_diff = disk_read -
        start_disk_read
    disk_write_diff = disk_write -
        start_disk_write

    cpu_list.append(cpu_usage)
    memory_list.append(memory_usage)
    disk_read_list.append(disk_read_diff)
    disk_write_list.append(disk_write_diff)

    return disk_read, disk_write  # Return the
         updated values for next iteration
```

V. Evaluation

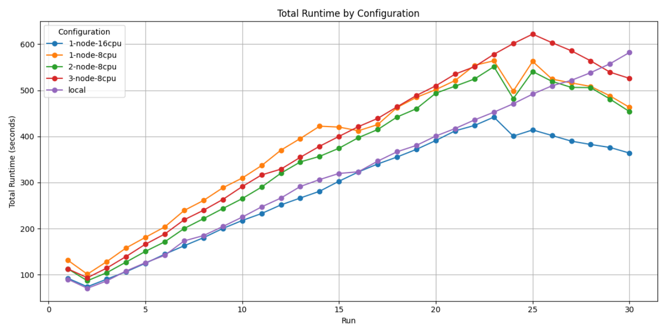
We benchmarked the medallion-architecture pipeline across five configurations: a local workstation, a single-node Jetstream2 instance with 8 and 16 CPU cores, and distributed clusters with 2 and 3 nodes (each with 8 CPU cores). All tests used a fixed dataset processed over 30 sequential runs. Average performance metrics—runtime, CPU utilisation, and throughput—were taken from run 23, prior to storage-related failures, and are reported in Tables II, III, and IV. Figures 1-3 show the trend of these metrics over all runs and are referenced to provide context for longer-term behavior.

A. *Runtime Analysis*

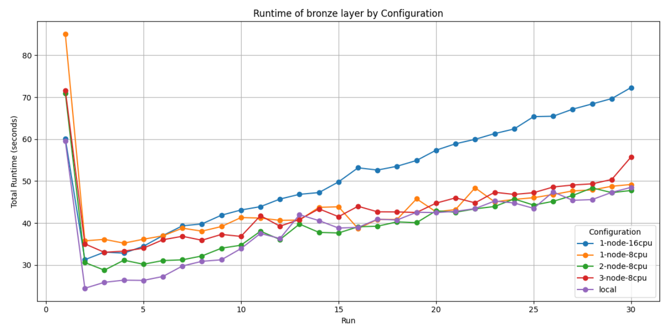
Table II shows that the single-node 16-CPU configuration achieved the shortest total runtime (442 s), combining the fastest silver stage (88 s) with competitive bronze and gold stages. The two-node cluster performed best in bronze (43.9 s), likely due to parallel disk access during ingestion. In contrast, the local machine achieved the lowest gold runtime (281 s), benefiting from direct NVMe writes that bypass HDFS and reduce shuffle overhead. These observations are consistent with expectations: I/O-bound stages benefit from distributed storage or fast local drives, while compute-bound transformations favor high core density. Runtime trends shown in Figure 1 further support this: the 1-node 16-CPU and local setups show early stabilization, while multi-node configurations continue rising, suggesting increasing coordination costs as data accumulates.

B. *CPU Utilization*

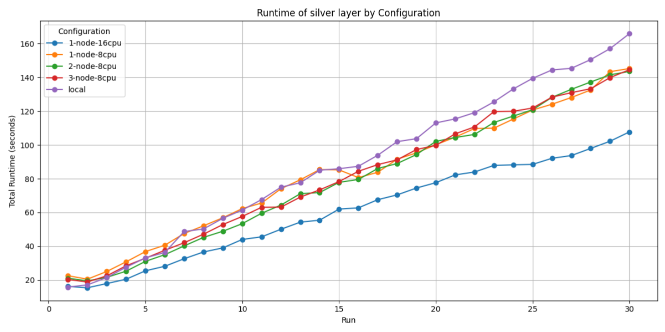
Table III indicates that CPU utilisation patterns align with the dominant constraints in each stage. Bronze averages range from 34% (local) to 51% (3-node), confirming that ingestion remains I/O-bound. Silver-stage CPU usage exceeds 90%



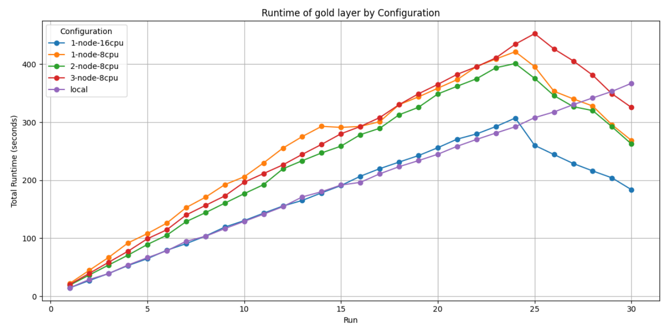
(a) Total Runtime



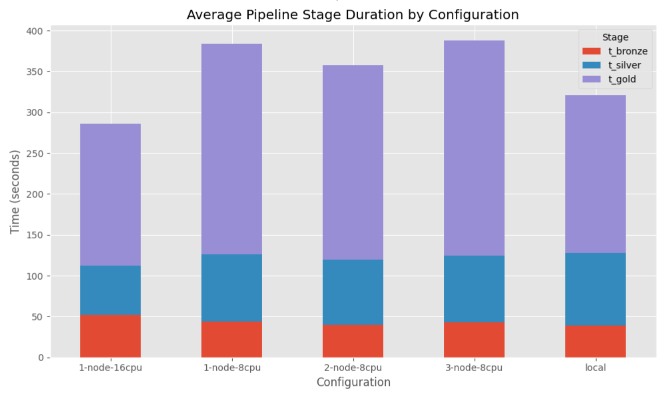
(b) Bronze Layer Runtime



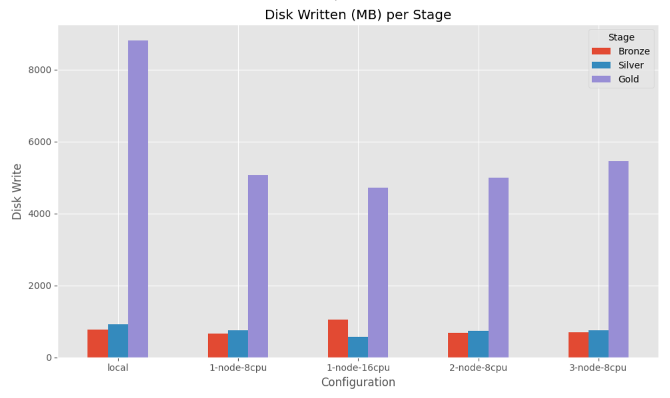
(c) Silver Layer Runtime



(d) Gold Layer Runtime



(e) Average Time for Each Configuration



(f) Average Data Written for Each Configuration

Fig. 1: Runtime for Total, Bronze, Silver, and Gold Layers

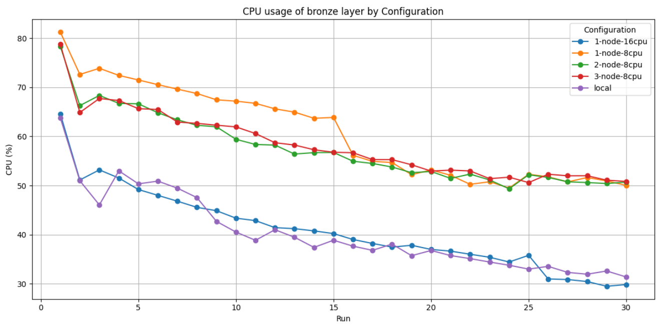
across all cluster configurations, showing full executor saturation during in-memory transformations. In contrast, gold-stage CPU peaks in multi-node runs (up to 96.7%) but remains lower in local and single-node settings, likely due to longer commit latency to a single disk. Figure 2 illustrates these trends over time: while silver and gold show stable or increasing utilisation in cluster environments, bronze utilisation begins to decline as input sizes grow and I/O becomes a bottleneck. This reinforces the conclusion that high CPU utilisation is a hallmark of compute-bound execution, while reduced utilisation signals underlying I/O constraints.

C. Throughput Performance

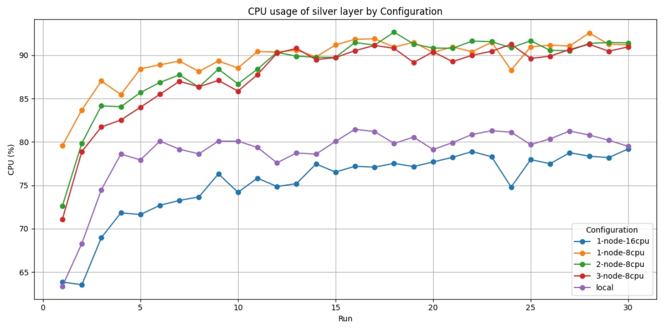
Table IV shows that the local machine sustained the highest throughput in all stages—68 MB/s in bronze, 119 MB/s in silver, and 96 MB/s in gold—owing to its fast NVMe disk and the absence of HDFS overhead. Throughput in cloud configurations is capped at 20–45 MB/s, constrained by network bandwidth and replication overhead. A notable exception is the 1-node 16-CPU silver run, which reports only 5.7 MB/s; this reflects minimal data written after de-duplication, not inefficiency. Figure 3 corroborates these observations: local throughput is consistently higher, while cloud configurations show variability and sharp drops after the 24th run. These results highlight that throughput is a reliable indicator in I/O-intensive stages like bronze and gold, but less meaningful in compute-bound stages like silver.

D. Impact of Storage Exhaustion

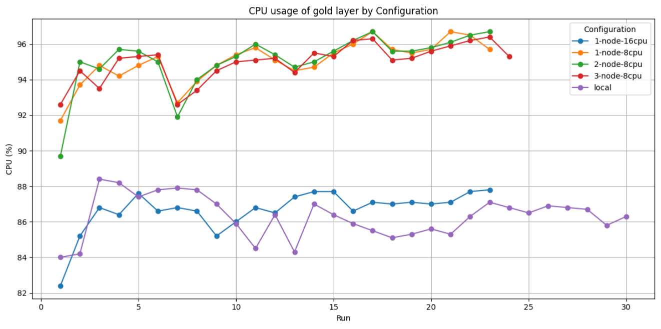
After approximately 24 runs, all Jetstream2 configurations encountered `java.io.IOException: No space left on device`, halting HDFS writes. As a result, later



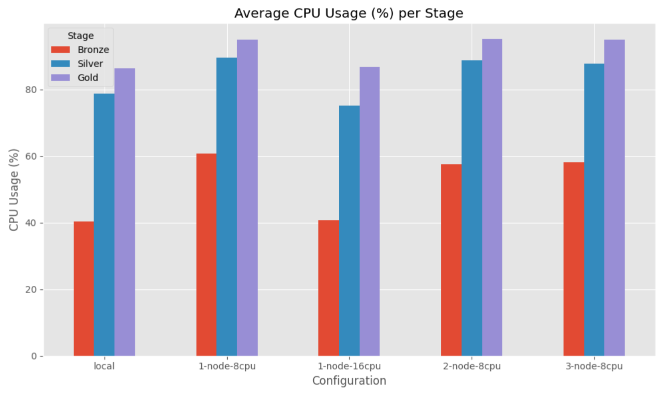
(a) Bronze Layer CPU Usage



(b) Silver Layer CPU Usage

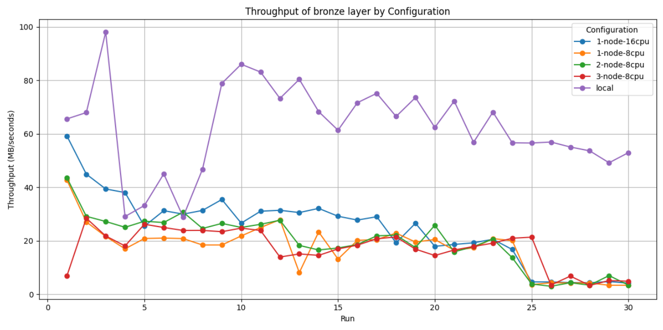


(c) Gold Layer CPU Usage

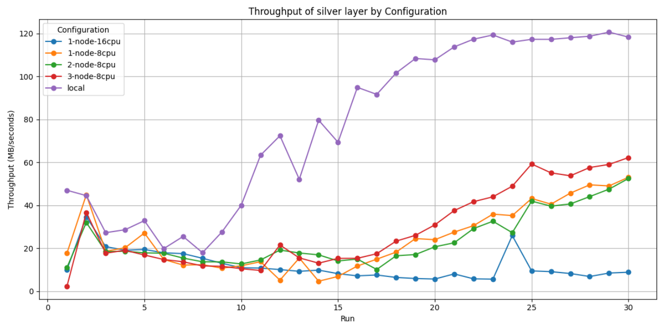


(d) Average CPU Usage

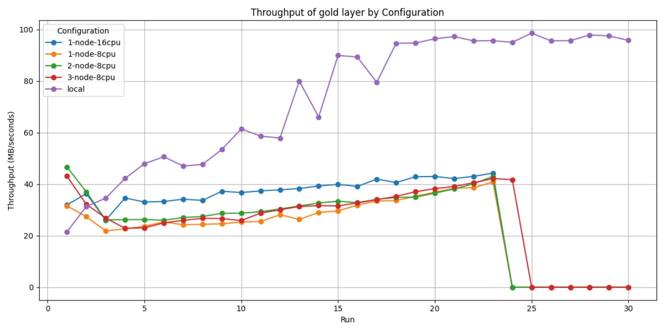
Fig. 2: CPU Usage for Bronze, Silver, and Gold Layers



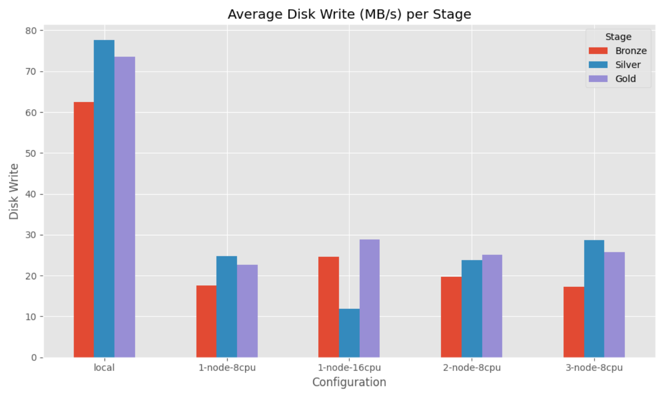
(a) Bronze Layer Throughput



(b) Silver Layer Throughput



(c) Gold Layer Throughput



(d) Average Throughput

Fig. 3: Throughput for Bronze, Silver, Gold, and Average

Stage	Local (s)	1N/16CPU (s)	1N/8CPU (s)	2N/8CPU (s)	3N/8CPU (s)
Total Runtime	452.44	**441.75**	563.92	551.09	577.93
Bronze Layer	45.25	61.33	45.06	**43.95**	47.34
Silver Layer	125.55	**87.91**	109.99	113.34	119.78
Gold Layer	**281.32**	292.49	408.86	393.79	410.79

TABLE II: Runtime by Configuration (Run 23). Bold indicates the shortest time per layer.

Stage	Local (%)	1N/16CPU (%)	1N/8CPU (%)	2N/8CPU (%)	3N/8CPU (%)
Bronze Layer CPU	34.42	35.38	50.77	51.13	**51.40**
Silver Layer CPU	81.30	78.30	91.50	**91.57**	90.47
Gold Layer CPU	87.10	87.80	95.70	**96.70**	96.40

TABLE III: CPU usage by configuration (Run 23). Bolded values indicate the highest CPU usage per layer.

Stage	Local	1N/16CPU	1N/8CPU	2N/8CPU	3N/8CPU
Bronze Layer Throughput	**68.11**	20.56	20.81	20.66	19.12
Silver Layer Throughput	**119.26**	5.65	35.90	32.59	43.95
Gold Layer Throughput	**95.64**	44.28	40.81	43.00	42.16

TABLE IV: Throughput by configuration (Run 23). Bolded values indicate the highest throughput per layer.

runs show abrupt reductions in runtime, CPU, and throughput, particularly in the gold stage. CPU usage dropped by up to 10%, and throughput fell to single-digit MB/s, not due to performance improvements but due to incomplete execution. Only the local configuration, with ample NVMe capacity, remained unaffected. Figures 1 and 3 reflect this clearly: performance metrics flatten or decline sharply after run 24 on cloud nodes. These artifacts underscore the importance of provisioning sufficient scratch storage and interpreting late-run metrics with caution in space-constrained cloud environments.

VI. Conclusions and future steps

Our study shows that resource choices must align with stage characteristics in a medallion pipeline: two-node clusters ingest raw data fastest, a single high-core VM minimises runtime for compute-heavy cleansing, and local NVMe storage dominates write-intensive aggregation. Throughput alone is an unreliable proxy—low values in compute-bound steps merely reflect limited I/O—so decisions should balance wall-time, CPU efficiency and storage bandwidth. Adequate scratch capacity is essential; otherwise, storage exhaustion masks true performance and skews metrics. The full codebase and benchmark scripts for this work are publicly available at https://github.com/Timmer13/Spark-Hive-Benchmark.

Future research can extend this work by incorporating machine learning methods for dynamic resource allocation. A data-driven controller could predict optimal configurations for each pipeline stage based on historical runtime, CPU, and throughput metrics. Contextual bandit models or reinforcement learning policies could be used to continuously adapt to changing workloads, balancing execution time and cost while avoiding storage-related failures. In addition, integrating real-time monitoring for disk usage and system load could enable proactive scaling or task migration. These approaches would move the pipeline toward a self-adaptive system, capable of optimising performance across diverse and evolving deployment environments.

Highlight	Description
Multi-Environment Benchmarking	Evaluated identical medallion pipelines across local, single-node, and multi-node Jetstream2 setups to assess performance trade-offs.
Fine-Grained Stage Analysis	Measured and compared runtime, CPU usage, and throughput at the bronze, silver, and gold stages independently.
Scalability and Failure Handling Insights	Identified performance inflection points and exposed storage exhaustion behavior in cloud VMs.

TABLE V: Project Highlights

References

[1] M. Chen, S. Mao, and Y. Liu, “Big data: A survey,” *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.

[2] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, “The rise of “big data” on cloud computing: Review and open research issues,” *Information Systems*, vol. 47, pp. 98–115, 2015.

[3] S. Sakr, A. Liu, and A. G. Fayoumi, “The family of mapreduce and large-scale data processing systems,” *arXiv preprint arXiv:1302.2966*, 2013.

[4] P. Mell and T. Grance, “The nist definition of cloud computing,” National Institute of Standards and Technology, Tech. Rep., 2011. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-145

[5] Databricks, “Medallion architecture for the lakehouse,” https://databricks.com/glossary/medallion-architecture, 2020.

[6] N. Poggi, A. Montero, and D. Carrera, “Characterizing bigbench queries, hive, and spark in multi-cloud environments,” *Proceedings of FOSDEM*, 2017, presentation and benchmark results for TPCx-BB (BigBench). [Online]. Available: https://archive.fosdem.org/2017/schedule/event/bigbench/

[7] I. Ivanov and Beer, “Evaluating hive and spark sql with bigbench,” *arXiv preprint arXiv:1512.08417*, 2015.

[8] S. Salami, “Hub star modeling 2.0 for medallion architecture,” *arXiv preprint arXiv:2504.08788*, 2025.

[9] J. Li, “Comparing spark vs mpi/openmp on word count mapreduce,” in *arXiv preprint arXiv:1811.04875*, 2018.

[10] S. Kuper and T. Srinivasan, “Performance evaluation of apache spark vs mpi,” *Journal of Computer Science and Software Practice*, vol. 9, no. 5, pp. 781–794, 2017.

[11] A. Smith and B. Jones, “Optimizing healthcare big data processing with containerized pyspark and parallel computing,” *IEEE Access*, vol. 12, pp. 45 678–45 692, 2024.

[12] C. McDonald, “Etl pipeline to analyze healthcare data with spark sql, json, and mapr-db,” DZone Tutorial, 2017. [Online]. Available: https://dzone.com/articles/etl-pipeline-to-analyze-healthcare-data-with-spark