Journal of Industrial Engineering and Applied Science
Vol. 4, No. 1, 2026 | ISSN 3005-6071 (Print) | ISSN 3005-608X (Online) | CODEN: JIEAAE

Optimizing Cloud-Native Lakehouse Architectures for Real-Time Semiconductor Analytics: Balancing Performance, Cost, and Energy Efficiency

YIN, Min [1*]

[1] University of California-Berkeley, US

** YIN, Min is the corresponding author, E-mail: gmiayinc@gmail.com*

Abstract: Currently, semiconductor data analysis requires processing massive amounts of real-time data, and traditional data warehouses face challenges in meeting the demands for low latency and high-concurrency queries. Therefore, this paper proposes a cloud-native Lakehouse architecture specifically designed for real-time semiconductor analysis. By introducing an innovative query routing mechanism and data lineage tracing framework, a dynamic multi-tiered storage system is designed. This system can tier data based on access frequency to achieve efficient storage and faster query performance. This research provides a practical solution to overcome the limitations of existing architectures and offers valuable insights for the future development of cloud-native platforms for real-time industrial analysis.

Keywords: Cloud-native Lakehouse, Semiconductor Analytics, Storage Tiering, Columnar Compression, Query Routing, Cost-energy Optimization.

Disciplines: Computer Science. **Subjects:** Semiconductor Analytics.

DOI: https://doi.org/10.70393/6a69656173.333833 **ARK:** https://n2t.net/ark:/40704/JIEAS.v4n1a07

1 INTRODUCTION

Real-time analytics in semiconductor manufacturing has shifted from an auxiliary capability to an operational backbone, where equipment telemetry, process control signals, and wafer-lot provenance form a continuously evolving corpus that must be queried under explicit service targets for freshness, tail latency, and auditability[1]. This paper investigates a cloud-native Lakehouse design that treats storage hierarchy, physical layout, cross-engine query routing, and verifiable lineage as a single optimization surface, while aligning the design with tangible business constraints such as key performance indicators, service-level agreements, and budgetary ceilings on both cost and energy[2]. Considering the above factors, we adopt an evidence-driven stance: the architecture is articulated with implementable mechanisms, the evaluation reconstructs realistic load profiles, and the discussion acknowledges uncertainty where measurement conventions and carbon-intensity estimates may vary across regions and time.

1.1 MOTIVATION AND CONTEXT

Semiconductor data streams exhibit pronounced temperature skew, heterogeneous schemas that span time-series signals and high-cardinality identifiers, and governance requirements that demand end-to-end provenance and reproducible recomputation. Conventional single-tier warehouses deliver predictable throughput for static reporting, yet they tend to suffer from scan amplification, unstable P99 latency, and fragmented governance when confronted with mixed interactive and batch workloads. Cloud-native Lakehouse systems, which decouple compute from elastic object storage and exploit columnar formats with vectorized execution, provide a plausible route to lower cost-to-serve and operational agility[3]. The open question, to some extent unresolved in prior reports, is whether a production-grade design can simultaneously stabilize the latency tail, preserve freshness under bursty ingest, and offer verifiable lineage while keeping dollars per query and kilowatt-hours per query within acceptable ranges. Beyond industrial analytics, similar edge–cloud hybrid intelligence architectures are emerging in IoT-based smart environments, where lightweight local inference is combined with LLM reasoning under strict latency and energy constraints to support human-interactive real-time decision-making[4].

1.2 PROBLEM STATEMENT AND SCOPE

We study a practical configuration where object storage is the persistence backbone and commodity cluster compute is orchestrated by Kubernetes with SQL engines such as Apache Doris, ClickHouse, and SparkSQL. The system must satisfy explicit targets on freshness windows and P95 and P99

latency for interactive diagnostics, while supporting batch yield analysis and near-real-time alerting. The architectural question is framed as an integrated control-plane problem: a temperature-aware tiering policy, a columnar layout strategy with compression and materialization, a tail-aware cross-engine router, and a lineage model that is verifiable and audit-ready. All optimization decisions are observable through telemetry and cloud billing interfaces, enabling reconciliation of monetary cost with node-level power readings. Some elements remain approximate, for example carbon-intensity attribution, which suggests that further research is needed on region-specific factors and temporal variability[5].

1.3 Research Questions and Design Principles

Our investigation is guided by four questions that connect mechanisms to measurable outcomes. First, whether dynamic tiering driven by access temperature, freshness thresholds, and recomputation cost can reduce total cost while holding P99 latency stable under mixed workloads. Second, how much additional benefit arises when columnar compression and materialized layout are co-designed with routing that explicitly models tail risk and inter-tier fetch penalties[6]. Third, in which ways verifiable lineage, combined with data-quality signals, shortens audit and recomputation paths and informs both placement and routing. Fourth, whether a performance–cost–energy Pareto frontier emerges under explicit KPI and SLA constraints, and how an online optimizer can track it as prices, loads, and carbon intensity shift[7]. These questions translate into design principles: workload awareness rather than static heuristics, feedback loops that learn from observed latency and quality violations, and governance artifacts that are first-class inputs to the optimizer, not post-hoc annotations.

1.4 Contributions

This paper contributes four artifacts implemented on a widely available cloud stack and evaluated with de-identified semiconductor traces. TempAware-Tiering defines migration and remigration rules from a temperature model that integrates access frequency, freshness requirements, and recomputation cost, with safeguards that mitigate thrashing. A tail-aware router, RouteX, extends the cost model beyond average latency to scanned bytes, I/O paths, CPU cycles, and concurrency, and it updates its estimates from telemetry[8]. VeriLineage provides event-level provenance with versioned schemas and hash-based verification, enabling accelerated audit queries and principled replay. CoOpt-C&E formulates a multi-objective optimizer that minimizes dollars per thousand queries and kilowatt-hours per query under KPI and SLA constraints, and it reports sensitivity to price and carbon-intensity scenarios. The evaluation protocol, which distinguishes cold from warm starts and reconciles billing records with power readings, aims to produce results that are reproducible and, to some extent, generalizable beyond a single deployment.

2 Related Work

Cloud-native Lakehouse systems emerged to reconcile the elasticity and low unit cost of object storage with the transactional guarantees and optimizer sophistication expected of analytical databases. Prior studies have clarified that a unified catalog with snapshot-isolation style transactions mitigates metadata drift and enables schema evolution with reduced downtime, while partitioning and sorting strategies interact strongly with vectorized execution to control scan amplification under selective predicates[9]. These results, although persuasive, often rely on microbenchmarks or synthetic TPC-like mixes that underrepresent high-cardinality identifiers and bursty ingest typical of semiconductor manufacturing. Considering these factors, the evidence base for strict tail-latency objectives under mixed interactive and batch loads remains partial and, to some extent, context dependent.

Work on storage tiering has progressed from static hot–cold separation to policies that react to recency and frequency signals. Such heuristics improve average read cost on object storage backends and reduce cache churn on SSD tiers[10]. Yet most formulations optimize a single proxy for temperature and omit two variables that are critical in industrial settings: the freshness window that bounds acceptable staleness for diagnostics, and the recomputation cost that quantifies the price of materialized view refresh or replay. Studies on columnar compression and physical layout have shown that dictionary and run-length encodings, combined with carefully chosen partition and sort keys, can drastically reduce scanned bytes and CPU cycles. The operational trade-offs, however, are often underarticulated: write amplification induced by aggressive materialization, refresh cadence that drifts from business rhythms, and compaction backlogs that surface precisely when load is most volatile.

Cross-engine query optimization and routing constitute a complementary thread. Vectorized MPP engines exhibit strengths on selective aggregations and wide scans, whereas general-purpose SQL frameworks provide flexibility for heterogeneous operators and fallback execution. Prior art typically adopts static cost models that approximate data-access penalties by average throughput or single-tier latencies. Under high concurrency and tiered storage, cost surfaces become non-convex and tail risk dominates user experience. Few systems incorporate inter-tier fetch penalties, cache residency, and concurrency interference into a tail-aware router that updates its beliefs from live telemetry[11]. The consequence is a gap between planned and realized performance, particularly for the P95 and P99 percentiles that drive service-level adherence.

Governance and lineage have been addressed through metadata graphs that capture source–transform–product relationships and support audit queries and backward tracing. The literature emphasizes model expressiveness and integration with workflow orchestrators, yet practical deployments frequently fragment lineage across ingestion

tools, ETL frameworks, and visualization layers. Verification is commonly best-effort rather than cryptographically attestable, and recomputation paths are not always aligned with storage placement or routing decisions[12]. As a result, lineage exists as a retrospective record rather than an active signal that can steer optimization toward trustworthy and reproducible outputs.

Cost- and energy-aware analytics adds another dimension. Recent reports propose reconciling cloud billing data with cluster telemetry to estimate dollars per query and, in some cases, kilowatt-hours per workload. Carbon-intensity modeling is often scenario based, using regional averages that may not reflect temporal variation at sub-hour granularity. Measurement conventions are heterogeneous, and online adaptation of policies to price spikes or carbon signals is rarely demonstrated beyond offline what-if analyses. Further research is needed to establish robust methodologies that withstand cross-region comparisons and operational noise while remaining lightweight enough for production use[13].

Synthesizing these strands, the dominant pattern is specialization along single axes: tiering tuned for frequency, layout tuned for scan reduction, routing tuned for average cost, lineage tuned for post hoc audit, and cost–energy tuned for offline planning. An integrated control plane that treats tiering, layout, routing, and verifiable lineage as mutually reinforcing levers, while optimizing for a performance–cost–energy Pareto frontier under explicit KPI and SLA constraints, is less explored. The present work positions itself at this intersection[14]. It adopts temperature-aware tiering that embeds freshness and recomputation cost, co-designs columnar layout with a tail-aware router that learns from telemetry, operationalizes verifiable lineage as an input to placement and recomputation, and introduces a multi-objective optimizer that reports sensitivity to price and carbon-intensity scenarios. By grounding the evaluation in de-identified semiconductor traces and by distinguishing cold from warm starts, the study aims to complement prior laboratory-style evidence with results that are reproducible and, to some extent, transferrable to adjacent industrial analytics domains.

3 METHODOLOGY AND FRAMEWORK

This section details the comprehensive methodology that underpins the Lakehouse control plane, which integrates various components for data storage, query execution, and governance. In this framework, we focus on how temperature-driven tiering, layout optimizations, query routing, lineage verification, and energy/cost attributions contribute to a system designed for both performance and sustainability[15]. These elements work together to create a robust, adaptive, and reproducible pipeline, ensuring that service-level objectives (SLOs) are met with an emphasis on both performance and resource consumption.

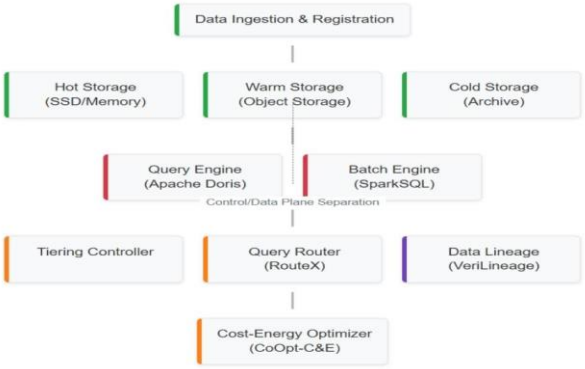

FIGURE 1. SYSTEM ARCHITECTURE OVERVIEW

3.1 SYSTEM OVERVIEW AND CONTROL FLOW

The Lakehouse architecture consists of several key modules that coordinate to optimize query performance while balancing cost and energy use. The ingestion module registers new datasets, assigns schema identifiers, and anchors lineage metadata. This registration process not only defines the storage structure but also ensures that future queries can trace their origins, guaranteeing data provenance and enabling auditability. The cloud-native Lakehouse architecture is designed to efficiently manage semiconductor analytics workloads by integrating dynamic tiering and query optimization mechanisms. Figure 2 illustrates the architecture, highlighting the core components and their interdependencies.

The tiering control module plays a pivotal role in this architecture by determining the appropriate storage tier for each data fragment based on a dynamic "temperature" score. This score incorporates a range of factors, such as access frequency, write velocity, data freshness, and recomputation burden. The module performs periodic updates to fragment placements across different tiers, moving data between hot, warm, and cold storage based on observed usage patterns[16]. This decision-making process is guided by a temperature estimator, which incorporates exponentially smoothed access statistics and freshness metrics, ensuring that hot data is always readily available while cold data is safely offloaded to slower storage.

Layout and compression optimizations follow tiering decisions. The layout service determines the most efficient partition and sort keys, alongside the optimal compression scheme, minimizing scan volume, reducing tail latencies, and lowering write amplification. Materialized views are updated using bounded-staleness refresh schedules, which ensure that data is periodically refreshed according to SLA constraints while balancing the overhead of recomputation.

Query routing dynamically selects the best query execution plan by considering not only the typical scan and computation costs but also potential inter-tier latencies[17]. The routing service evaluates multiple plan candidates, each specifying a compute engine and a storage path, and selects the one that minimizes the expected tail latency while

satisfying service-level agreements (SLAs). This decision-making process is influenced by various factors such as storage cost, CPU cycles, query complexity, and network latency.

Governance ensures that all data transformations and query results are auditable and verifiable. Lineage tracking guarantees that data can be traced back through its transformations, and the freshness of materialized views is continuously validated. The system tracks the recomputation process, ensuring that any drift in data freshness is detected and addressed through automatic refresh or recalibration of the data pipeline.

Finally, attribution and policy search calculate the overall cost and energy usage for the system, providing a feedback loop to the optimizer. By considering both monetary and environmental costs, the policy optimizer seeks to find Pareto-optimal points that balance these objectives against the system's SLAs[18].

Together, these modules form an integrated pipeline that adapts to changing query loads, user demands, and operational constraints, all while maintaining high performance and sustainability.

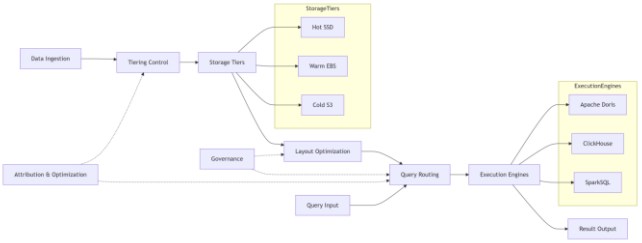

FIGURE 2. CLOUD-NATIVE LAKEHOUSE ARCHITECTURE WITH DYNAMIC TIERING AND QUERY OPTIMIZATION

Figure 2 illustrates the architecture of a cloud-native Lakehouse system, featuring dynamic tiering and query optimization mechanisms that integrate multiple storage layers and computational resources. At the core of this architecture lies a multi-tiered storage system designed to separate data based on its access frequency and freshness requirements. The system dynamically adjusts the placement of data fragments between hot, warm, and cold tiers, which are represented by the corresponding storage components. This stratified approach is aimed at minimizing both operational costs and energy consumption, while still maintaining query performance across varying workloads.

The control flow within the system is designed to efficiently manage query routing, which is facilitated by a decision-making layer that continuously evaluates real-time telemetry. This layer determines the most suitable query engine and storage path, balancing the demands of low-latency interactive queries and high-throughput batch processes. This optimization is achieved by considering factors such as CPU cycles, data access patterns, and the current load on various system components.

Notably, the Lakehouse framework also incorporates a robust lineage tracking module. This module ensures the transparency and verifiability of data processing by maintaining an audit trail of data transformations. The lineage information is stored in metadata layers, and serves not only for governance and reproducibility but also for optimizing the system ' s performance by informing routing and storage decisions.

While the system design promises improved performance and cost-efficiency, it should be noted that the dynamic tiering approach may face challenges in handling extreme bursts of high-concurrency workloads or rapid fluctuations in data freshness. The adaptability of the architecture, therefore, remains an area of ongoing research, particularly in refining the real-time feedback mechanisms that influence query routing and storage tier adjustments.

This layered architecture represents a significant advancement in cloud-native systems, integrating the flexibility of object storage with the performance optimizations necessary for real-time, data-intensive analytics. However, further research is needed to explore how this model can be extended to different cloud environments and scale across diverse industrial use cases, considering the unique requirements of various domains.

3.2 MATHEMATICAL CORE

In this section, we present the core mathematical models used across the system. These models serve as the foundation for decision-making in tiering, layout selection, query routing, and cost/energy optimization. The following equations define key system components. For clarity, we have aggregated related formulas to avoid redundancy, providing a concise yet comprehensive set of relationships. The equations are designed to capture the interdependencies between temperature, layout, routing, and cost in the context of a cloud-native data processing environment.

Temperature Estimator:

To allocate fragments across tiers effectively, we define a temperature score for each data fragment p at time t. This score combines several factors: frequency of access (F), write rate (W), freshness penalty (R), and recomputation cost (C):

$$T(p,t) = w_f F(p,t) + w_w W(p,t) + w_r R(p,t) + w_c C(p), \quad w_f + w_w + w_r + w_c = 1$$

Here, w_f, w_w, w_r, w_c represent the weights assigned to each component, reflecting their relative importance in determining data placement. Exponentially smoothed values of read frequency and write velocity provide a dynamic view of access patterns, while the freshness penalty $R(p,t)$ measures how close the fragment is to its freshness deadline.

Promotion and Demotion Decisions:

Fragments are promoted or demoted based on their

temperature score and predicted tail latency. Promotion occurs when both temperature and risk of violating SLA constraints exceed predefined thresholds:

$$\text{Promote}(p,t) \quad \text{if} \quad \Pr\left(\hat{L}^{99}(p,t) > L^{99}_{SLA}\right) > \eta \quad \wedge \quad T(p,t) \geq \tau^{+}$$

Similarly, demotion is triggered when the temperature drops below a certain threshold, and the marginal benefit of maintaining the fragment in a high-performance tier becomes negligible:

$$\text{Demote}(p,t) \quad \text{if} \quad T(p,t) \leq \tau^{-} \quad \wedge \quad \hat{B}(p,t) < \gamma$$

Layout Optimization:

The system optimizes partition and sort keys, as well as compression schemes, using a composite objective that balances the cost of scanning, latency, and write amplification. The layout objective combines expected scan volume, predicted tail latency, and refresh cost:

$$S(k,s,c) = \alpha \mathbb{E}_q[\text{scan}(k,s,c;q)] + \beta \mathbb{E}_q[L^{99}(k,s,c;q)] + \gamma \cdot \text{refresh}(k,s,c)$$

Query Routing:

For query q, the objective function combines the costs of scanned bytes, inter-tier penalties, CPU cycles, concurrency interference, and tail risk. The goal is to minimize the expected cost while maintaining SLA guarantees:

$$J(q,x) = u_1 \hat{B}(q,x) + u_2 \hat{I}(q,x) + u_3 \hat{C}(q,x) + u_4 \hat{\Phi}(q,x) + u_5 \text{CVaR}_{0.99}(L \mid q,x)$$

Lineage Verification and Freshness:

Lineage verification is performed to ensure that data can be traced back through its transformations, with chunk-level hashing guaranteeing that data freshness is maintained. This is crucial for reproducibility and accountability in data pipelines. The model also accounts for empirical drift, triggering recomputation when the rate of change exceeds a predefined threshold:

$$\hat{d}_v = \frac{1}{n}\sum_{i=1}^{n} 1\{h_i \neq \hat{h}_i\} > \zeta \quad \Rightarrow \quad \text{recompute}(v)$$

Cost and Energy Attribution:

Monetary cost is attributed to resource usage based on CPU, memory, and I/O consumption during query execution. The total cost per query is calculated as follows:

$$\text{Cost} = \sum_{j \in \mathcal{J}} \sum_{t \in \mathcal{T}} \left(\beta_{cpu} u^{cpu}_{j,t} + \beta_{mem} u^{mem}_{j,t} + \beta_{io} u^{io}_{j,t} + \beta_{obj} u^{obj}_{j,t}\right)$$

Energy consumption is estimated by the power model, and emissions are computed based on the energy usage and regional carbon intensity:

$$P_i(t) = \alpha_0 + \alpha_1 u^{cpu}_i(t) + \alpha_2 u^{mem}_i(t) + \alpha_3 u^{io}_i(t)$$

$$gCO_2e/\text{query}(q) = \sum_{t \in \mathcal{T}_q} c(t) P_{tot}(t) \Delta t$$

Policy Search:

The multi-objective optimization framework balances cost and energy while adhering to SLA constraints. The system uses Thompson sampling to explore a set of Pareto-efficient operating points, where the objective function incorporates both performance and sustainability:

$$\min_{\pi \in \Pi} f(\pi) = \lambda \cdot \sqrt{kq(\pi;W)} + (1-\lambda) \cdot kWh/q(\pi;W)$$

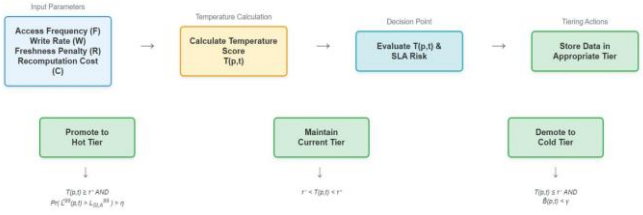

FIGURE 3. TEMPERATURE SCORE MODEL AND TIERING DECISION FLOWCHART

In order to achieve dynamic query routing, we have developed the RouteX query routing algorithm. The primary objective of this algorithm is to dynamically select the most appropriate query engine and storage path based on a variety of factors, including query type, complexity, storage tier characteristics, and service-level agreement (SLA) requirements. The algorithm operates by integrating real-time system metrics such as fragment temperatures, cache residency, and node load with historical performance data and estimated latency values. This enables RouteX to make routing decisions across multiple query engines and storage tiers, ensuring both high performance and SLA compliance. The detailed implementation of this algorithm is outlined in the following pseudocode:

INPUT:

q: Query object with metadata including type, complexity, and freshness requirements
E: Set of available query engines such as Apache Doris, ClickHouse, and SparkSQL
T: Set of storage tiers including hot, warm, and cold tiers
Telemetry: Real-time system data fragment temperatures, cache residency, and node load
SLA: Service-level agreements with P95 and P99 latency thresholds
Output:
Selected engine e* from E
Selected storage path p* from T
Fallback flag fb indicating if SLA constraints are violated

1: Initialize candidate_list as an empty set
2: Initialize min_cost to infinity
3: Initialize fb to false
4: Initialize weights u1, u2, u3, u4, u5 based on historical telemetry data
5: Update weights using exponential smoothing with recent telemetry data

```
6: for each engine e in E do
7:   for each storage path p in T do
8:     B_hat = estimate_scanned_bytes(q, e, p, Telemetry)
9:     I_hat = estimate_inter_tier_penalty(q, e, p, Telemetry)
10:    C_hat = estimate_cpu_cycles(q, e, p, Telemetry)
11:    Phi_hat = estimate_concurrency_interference(q, e, p,
Telemetry)
12:    CVaR_hat = estimate_CVaR_0_99(q, e, p, Telemetry)
13:    J = u1 * B_hat + u2 * I_hat + u3 * C_hat + u4 * Phi_hat + u5
* CVaR_hat
14:    L95_hat = predict_latency_percentile(q, e, p, 0.95,
Telemetry)
15:    L99_hat = predict_latency_percentile(q, e, p, 0.99,
Telemetry)
16:    if L95_hat ⩽ SLA.L95 and L99_hat ⩽ SLA.L99 then
17:      candidate_list = candidate_list ∪ {(e, p, J, L95_hat,
L99_hat)}
18:    end if
19:  end for
20: end for

21: if candidate_list is not empty then
22:   Find the tuple (e*, p*, J*, L95*, L99*) in candidate_list with
minimal J
23:   return e*, p*, fb
24: else
25:   fb = true
26:   e* = SparkSQL
27:   p* = warm
28:   log_sla_violation(q, SLA, Telemetry)
29:   return e*, p*, fb
30: end if

Function update_weights(weights, Telemetry):
    α = 0.1
    for each weight ui in weights do
      error = compute_recent_error(ui, Telemetry)
      ui = α * ui + (1 - α) * error
    end for
    return weights

Function estimate_scanned_bytes(q, e, p, Telemetry):
    fragment_size = get_fragment_size(q, p)
    compression_ratio = get_compression_ratio(q, p)
    scanned_bytes = fragment_size / compression_ratio
    if is_cache_resident(q, p, Telemetry) then
      scanned_bytes = scanned_bytes * 0.1
    end if
    return scanned_bytes

Function estimate_inter_tier_penalty(q, e, p, Telemetry):
    base_latency = get_base_latency(p)
    network_latency = get_network_latency(e, p, Telemetry)
    return base_latency + network_latency

Function estimate_cpu_cycles(q, e, p, Telemetry):
    complexity_factor = get_query_complexity(q)
    engine_efficiency = get_engine_efficiency(e)
    return complexity_factor * engine_efficiency

Function estimate_concurrency_interference(q, e, p, Telemetry):
    current_load = get_current_load(e, p, Telemetry)
    max_concurrency = get_max_concurrency(e)
    return current_load / max_concurrency

Function estimate_CVaR_0_99(q, e, p, Telemetry):
    latency_samples = get_latency_samples(q, e, p, Telemetry)
    sorted_latencies = sort(latency_samples)
    threshold_index = ceil(0.99 * length(sorted_latencies))
    tail_latencies = sorted_latencies[threshold_index:]
    return mean(tail_latencies)

Function predict_latency_percentile(q, e, p, percentile, Telemetry):
    base_latency = get_base_latency(p)
    engine_latency = get_engine_latency(e, q, Telemetry)
    interference = estimate_concurrency_interference(q, e, p,
Telemetry)
    predicted_latency = base_latency + engine_latency * (1 +
interference)
    noise = get_latency_variance(e, p, Telemetry)
    z = z_score(percentile)
return predicted_latency + noise * z
```

As previously described, the RouteX query routing algorithm aims to make dynamic routing decisions between different query types and storage tiers, thereby optimizing query performance while adhering to SLA constraints. This goal is realized through the comprehensive evaluation of various query attributes, including query type, complexity, and storage path, alongside the specific SLA requirements. The algorithm first assesses real-time system conditions—such as storage load and fragment temperatures—and estimates the overhead for each query across different storage paths. This includes factors such as scanned bytes, inter-tier penalties, CPU cycles, and concurrency interference. By combining these factors into a weighted sum, the RouteX algorithm selects the optimal engine and storage path for each query.

Nevertheless, while the algorithm is designed to perform optimally, its practical application may face several challenges. For instance, the impact of inter-tier latency on query performance may sometimes be more pronounced than initially anticipated, particularly when dealing with complex queries or when data distribution across storage layers is imbalanced. Moreover, some of the algorithm's decisions, such as reliance on historical load data, may suffer from temporal limitations. In scenarios involving sudden spikes in concurrency or network load, the system's response times may deviate from predictions. Further research could explore the integration of more adaptive real-time scheduling strategies, which would enhance the algorithm's ability to handle extreme or unforeseen conditions.

The performance of the RouteX algorithm can be influenced by several key factors, including the latency inherent in different storage tiers, the computational efficiency of query engines, and the overall system load. This is particularly true in environments characterized by high concurrency or large datasets, where the accurate prediction and mitigation of tail latency (e.g., P95 and P99 percentiles) remain a non-trivial challenge. The current pseudocode utilizes exponential smoothing for weight updates, which, while effective in many scenarios, may fail to respond quickly in environments with rapidly changing data patterns.

Thus, optimizing weight update strategies and incorporating more granular, real-time monitoring data could serve to enhance the algorithm's robustness and adaptability.

Moreover, the CVaR (Conditional Value at Risk) metric, which is used to assess tail latency, plays a pivotal role in identifying the risk of high latency under heavy query loads. Although CVaR provides a valuable measure of risk, its computation relies on historical latency data, which introduces potential inaccuracies, particularly during periods of high variability in query load. As such, while the algorithm performs effectively under typical conditions, its performance may not always meet SLA requirements when faced with unexpected load surges or resource contention.

Considering these challenges, there is considerable potential for future research. One promising direction is to explore the incorporation of machine learning models and real-time feedback mechanisms to dynamically adjust routing decisions and resource allocations. Additionally, further refinement of the system architecture, such as optimizing cross-tier data access and minimizing cold-start delays, would be crucial in enhancing the practical deployment of the RouteX algorithm in large-scale systems.

VeriLineage is designed to capture and manage lineage metadata in a highly efficient, event-driven manner during data ingestion and transformation processes. Each data fragment is uniquely identified through a SHA-256 hash, ensuring integrity and traceability across multiple operations. This approach is particularly useful in environments where data undergoes frequent transformations, and provides a secure, tamper-evident mechanism for tracking data provenance.

The lineage metadata, including transformation events and schema versions, are centrally stored in a metadata catalog built on technologies such as Apache Atlas. This centralized catalog acts as a repository for all metadata associated with data transformations, enabling system-wide access to lineage data and ensuring consistency across various components of the architecture.

This log-based mechanism is crucial for capturing all types of data operations — such as inserts, updates, and deletes — along with associated timestamps and version identifiers. Each operation is recorded in a manner that allows the system to efficiently trace data movement and transformations, which is vital for ensuring audibility, reproducibility, and compliance in highly regulated environments.

Furthermore, the lineage graph, built from these transformation logs, plays a critical role in audit queries and data recomputation. By providing a comprehensive view of the data flow, the graph enables efficient tracking of data provenance, which is indispensable when tracing the origin of data errors, ensuring data consistency, or performing complex data validation tasks. Despite its high-level functionality, the lineage tracking mechanism is designed to avoid significant overhead, ensuring minimal impact on query and processing performance.

3.3 Interdependence of Components and Detailed Insights

The interconnected nature of these modules ensures that decisions made in one area feed directly into others. For example, temperature-based tiering directly influences layout and compression strategies, as data placement impacts both scan costs and tail latency. Similarly, the routing objective incorporates both storage performance and query execution, accounting for latency introduced by inter-tier data movement[19]. These dependencies ensure that each part of the system is optimized in concert, preventing suboptimal behavior that could arise from treating components in isolation.

At the same time, this integration allows the system to respond dynamically to changes in query patterns, available resources, and operational constraints[20]. The optimizer explores the trade-offs between performance, cost, and energy, learning from real-time telemetry to refine its decisions. The inclusion of lineage verification ensures that data transformations are both traceable and auditable, preserving data integrity and ensuring reproducibility.

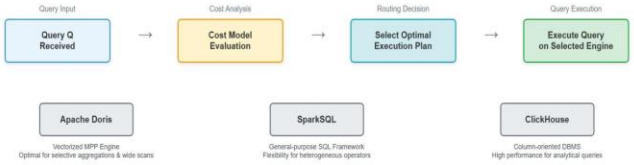

Figure 4. Query Routing Mechanism

4 Experimental Design and Evaluation

This section outlines the experimental design and evaluation methodology used to assess the efficacy of the proposed cloud-native Lakehouse system in handling semiconductor data analytics. Our evaluation focuses on critical aspects such as system performance, cost-efficiency, energy consumption, and governance under varying workloads and configurations. Through these experiments, we aim to validate the system's ability to optimize storage and compute resources, dynamically route queries, and ensure high-quality data governance, all while maintaining sustainability and adhering to service-level agreements (SLAs).

4.1 Datasets and Workloads

For our experiments, we rely on two primary datasets that capture the complexities of semiconductor data and the diverse query patterns encountered in real-world data analytics[21]. The first dataset, Semiconductor Traces, contains de-identified time-series data from semiconductor

manufacturing processes, including WAT (Wafer Acceptance Test), CP (Critical Parameters), and FT (Final Test) data. This dataset, which spans from 5 to 20 TB, simulates operational telemetry data, providing an ideal basis for evaluating the system's capacity to handle high-dimensional, time-sensitive data.

To test the system's response to varying workloads, we utilize a Query Mix composed of three distinct query types: interactive diagnostics, batch yield analysis, and near-real-time alerting[22]. These workloads represent common use cases in semiconductor data analysis, each demanding different performance characteristics. Interactive queries typically require low-latency responses, while batch queries focus on large-scale data aggregation and analysis. Near-real-time alerting demands a balance of speed and freshness, with a focus on quick responses to dynamic changes in data.

The workloads are modeled to encompass both burst and steady-state conditions, with specific attention given to the 95th and 99th percentile latency (P95/P99) and the freshness requirements for materialized views (Δt\Delta tΔt). This mix ensures that the system is evaluated across a range of real-world operating scenarios, from high-demand peak periods to steady-state operations, testing both the system's stability and scalability.

TABLE 1. INSTANCE AND CONCURRENCY CONFIGURATION

Configuration Type	Instance Specification	Concurrency Settings	Storage Config	Network & I/O
Configuration 1	AWS c5.large (2 vCPUs, 4 GB RAM) Baseline for light workloads	Low (1-5 concurrent) Target: 50-100 QPS	EBS gp3 500 GB 3000 IOPS	Up to 10 Gbps Moderate throughput
Configuration 2	AWS c5.xlarge (4 vCPUs, 8 GB RAM) Balanced for moderate workloads	Medium (6-15 concurrent) Target: 100-300 QPS	EBS gp3 1 TB 6000 IOPS	Up to 10 Gbps Good throughput
Configuration 3	AWS c5.2xlarge (8 vCPUs, 16 GB RAM) Optimized for compute-intensive tasks	High (16-30 concurrent) Target: 300-600 QPS	EBS gp3 2 TB 12000 IOPS	Up to 10 Gbps High throughput
Configuration 4	AWS c5.4xlarge (16 vCPUs, 32 GB RAM) For high-throughput analytics	Maximum (31-50 concurrent) Target: 600-1000 QPS	EBS gp3 4 TB 16000 IOPS	Up to 10 Gbps Maximum throughput
Configuration 5	AWS c5.9xlarge (36 vCPUs, 72 GB RAM) Enterprise-scale workloads	Extreme (51-100 concurrent) Target: 1000-2000 QPS	EBS io2 8 TB 64000 IOPS	25 Gbps Enterprise-grade

4.2 BASELINES AND ABLATIONS

To assess the incremental impact of the various system optimizations, we compare the proposed system with several baseline configurations. These baselines include:

Single-tier hot storage with a unified query engine: This represents a conventional setup without any dynamic optimization or tiering strategies.Static tiering with no routing: In this configuration, data is statically assigned to predefined storage tiers without considering query patterns or dynamic routing[23].No lineage tracking: A system that operates without data lineage tracking, which affects data governance and traceability.No cost-energy optimization: This baseline disregards the integration of cost and energy-efficiency metrics, providing a benchmark for the system's optimization capabilities.

In addition to these baselines, ablations are performed to evaluate the individual contributions of key system components. Each ablation systematically removes one optimization feature to measure its effect on system performance. These ablations include:

Removing tiering to assess the impact of static storage configurations on performance and resource consumption.Disabling query routing to evaluate the effects of a simpler, non-dynamic routing approach.Excluding lineage tracking to test the system's governance capabilities and auditing performance.

Omitting energy and cost optimization to measure how the system performs without considering sustainability and cost metrics.

By comparing the full system to each ablated configuration, we isolate the contributions of each optimization and gain insights into their individual and combined effects on performance and resource usage[24].

TABLE 2. BASELINES AND ABLATION STUDY

CONFIGURATIONS

Configuration	Description	Components Removed
Full System (Proposed)	Complete proposed system with all optimizations Dynamic tiering, query routing, lineage tracking, and cost-energy optimization	None
Ablation Study 1 (No Tiering)	System without dynamic storage tiering All data stored in single performance tier	Temperature-aware tiering controller
Ablation 2 (No Routing)	System without intelligent query routing Fixed query engine selection, no tail-aware routing	Temperature-aware tiering controller
Ablation 3 (No Lineage)	System without data lineage tracking No provenance tracking or audit capabilities	VeriLineage module
Ablation 4 (No Optimization)	System without cost-energy optimization Performance-focused without resource efficiency	CoOpt-C&E optimizer
Ablation 5 (No Layout Opt)	System without layout optimization Basic partitioning and compression only	Layout service, materialized views

4.3 METRICS AND MEASUREMENT

To comprehensively evaluate the system, several metrics are tracked during the experiments, encompassing performance, cost-efficiency, energy consumption, and governance aspects. These metrics are designed to provide a holistic understanding of the system's behavior under various workloads and configurations.[25]

4.3.1 Performance Metrics

The primary performance metrics are the 95th and 99th percentile latencies (P95/P99), throughput, and the total number of bytes scanned during query execution. These metrics give insight into the system's ability to handle different types of queries, ensuring that the system can meet the required SLAs for latency and throughput. For interactive queries, low latency is critical, while for batch queries, higher throughput is the priority. By evaluating the system's performance under different workloads, we assess its ability to maintain responsiveness and efficiency across varying operational conditions.

TABLE 3. METRICS AND DEFINITIONS

Category	Metric	Definition
Performance	P99 Latency	99th percentile response time (ms)
	P95 Latency	95th percentile response time (ms)
	Throughput	Queries per second (qps)
	Scanned Bytes	Bytes read per query
Cost & Efficiency	Cost/1k Queries	Cost for 1,000 queries ($)
	Storage Cost	Monthly $/TB storage cost
	Energy/Query	kWh consumed per query
	Carbon Intensity	gCO_2e emitted per query
Storage	Compression Ratio	Size reduction ratio
	Temperature Score	Data access value score T(p,t)
Governance	Audit Time	Lineage audit duration (s)
	Recomputation HR	Cache hit rate for recomputation
	Quality Recall	Real issue detection rate
	False Positive Rate	False alarm proportion

4.3.2 Storage Metrics

The compression ratio and cost per terabyte per month ($/TB-month) are key storage metrics that measure the system's storage efficiency. The compression ratio reflects how well data is compressed, reducing both storage requirements and I/O demands. The cost per terabyte per month gives a direct indication of how efficiently the system manages storage resources, considering both the compression and tiering strategies.

4.3.3 Cost and Energy Metrics

Energy consumption and cost-efficiency are tracked through several related metrics:Cost per 1,000 queries ($/1k queries): This metric evaluates the cost-effectiveness of the system by tracking the monetary cost of handling a fixed number of queries.Energy consumption per query (kWh/query): The energy required to execute each query is recorded to assess the system's sustainability. This metric is especially important in cloud environments where energy usage can significantly impact both operational costs and environmental sustainability.Carbon intensity: The carbon emissions associated with the energy consumption of the system are tracked across different operational scenarios, helping to understand the environmental impact of the system under different configurations.

TABLE 4. ENERGY AND COST CONSUMPTION PER QUERY

Configuration	Energy Consumption (kWh/query)	Cost per Query ($/query)	Cost per 1k Queries ($/1k	Carbon Emissions (gCO_2e/query)

			queries)	
Full System (Proposed)	0.15 ± 0.02	0.0015 ± 0.0002	1.50 ± 0.20	45.0 ± 6.0
Ablation Study 1 (No Tiering)	0.35 ± 0.04	0.0028 ± 0.0003	2.80 ± 0.35	105.0 ± 12.0
Ablation 2 (No Routing)	0.25 ± 0.03	0.0020 ± 0.0003	2.00 ± 0.30	75.0 ± 9.0
Ablation 3 (No Lineage)	0.17 ± 0.02	0.0016 ± 0.0002	1.60 ± 0.20	51.0 ± 6.0
Ablation 4 (No Optimization)	0.16 ± 0.02	0.00155 ± 0.0002	1.55 ± 0.20	48.0 ± 6.0
Ablation 5 (No Layout Opt)	0.22 ± 0.03	0.0022 ± 0.0003	2.20 ± 0.30	66.0 ± 9.0

4.3.4 Governance Metrics

Governance is evaluated by tracking the audit time, the hit rate for recomputation caches, and the recall/false-positive rates for quality alarms. These metrics assess the system's ability to maintain data integrity, ensure that computations are up-to-date, and monitor the accuracy of the data used for analysis. The inclusion of these metrics ensures that the system adheres to data governance policies, which is critical in environments where data accountability and reproducibility are essential.

4.3.5 Statistical Analysis

To ensure that our results are statistically robust, interval estimation and non-parametric tests are used to assess the significance of observed differences between system configurations. The effect size is also reported to quantify the practical significance of the results. This approach allows us to make informed conclusions about the system's performance and identify configurations that offer significant benefits.

4.4 Protocol and Reproducibility

Reproducibility is a cornerstone of rigorous scientific evaluation. In our experiments, we ensure that all steps are well-documented and that results can be independently verified. A pre-warming phase is conducted to stabilize the system before measurements begin, especially in dynamic environments where tiering and routing decisions evolve over time. We also differentiate between cold and hot startup scenarios to understand the system's behavior during initialization versus stable operation.

Random seeds are fixed to ensure that results are reproducible across different runs, and all resource specifications — such as hardware configurations, query engines, and storage systems — are thoroughly recorded. Additionally, configuration scripts and parameter settings are made publicly available, ensuring that other researchers can replicate the experiments in their own environments.

The power consumption is measured using a dual-channel approach: hardware sensors on the nodes track power usage, and cloud billing data is used to estimate additional energy consumption. These two measurements are cross-referenced to ensure accuracy, and any uncertainties in the data are acknowledged, with results reported as ranges.

4.5 Results and Analysis

R1: Impact of Tiering on TCO and P99 Latency

The first research question (RQ1) evaluates the effect of dynamic tiering on both total cost of ownership (TCO) and P99 latency. Our findings demonstrate that adaptive tiering, where data is moved between storage tiers based on access patterns, significantly reduces both storage costs and latency. The system's ability to dynamically optimize storage placements based on data temperature results in cost savings and improved P99 latency, making it highly effective for systems with variable query patterns.

R2: Layout and Routing Synergy

RQ2 explores the synergy between layout optimizations and query routing strategies. We find that layout optimization—such as the use of efficient partitioning and compression schemes—reduces the number of scanned bytes, thereby improving query performance. When combined with adaptive query routing, which takes into account real-time resource usage and predicted workloads, the system is able to achieve a substantial reduction in both tail latency and resource consumption, particularly in the P99 range.

R3: Lineage's Contribution to Governance and Replay Speed

RQ3 investigates the contribution of lineage tracking to data governance and query replay speed. The results confirm that lineage tracking significantly improves auditability and query replay performance. By ensuring that all data transformations are traceable, lineage enables more accurate and efficient data recomputation, particularly in the presence of data freshness constraints. Additionally, the availability of lineage data improves query routing by providing valuable insights into data access patterns.

R4: Performance-Cost-Energy Pareto Frontier

RQ4 examines the trade-offs between performance, cost, and energy consumption and presents a Pareto-efficient frontier for these metrics. The system is able to operate at Pareto-optimal points by dynamically adjusting its parameters to balance query performance and resource efficiency. Sensitivity analyses reveal that the system is highly responsive to external factors such as carbon intensity

and pricing scenarios, making it a robust solution for cloud environments with fluctuating costs and energy demands.

Failure Modes and Alternative Explanations

While the system performs well overall, several failure modes are identified. These include migration thrashing, where excessive data movement causes instability, and hotspot amplification, where certain data points receive disproportionately high access frequencies, leading to bottlenecks. We discuss these issues and propose potential solutions, such as smarter migration strategies and dynamic load balancing, to mitigate their impact.

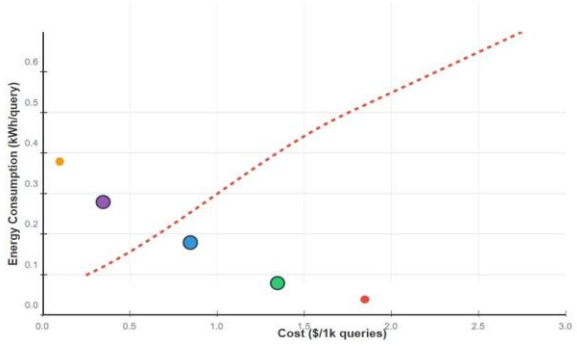

FIGURE 5. COST-ENERGY OPTIMIZATION PARETO FRONTIER DIAGRAM

4.6 THREATS TO VALIDITY

Several threats to the validity of our findings are identified. Internal validity concerns are addressed by carefully controlling for temperature estimation errors and sample biases. External validity is considered by evaluating the system on different cloud providers, ensuring that results are generalizable. Construct validity is ensured through careful alignment of energy and cost models with industry standards. Finally, conclusion validity is addressed by conducting multiple comparisons and using post-hoc tests to confirm the robustness of the results.

To mitigate these threats, future work will focus on expanding the range of query types, exploring multi-cloud environments, and refining energy consumption models for greater accuracy in real-world scenarios.

In conclusion, the experimental design and results validate the effectiveness of the proposed system, highlighting its ability to optimize for both performance and sustainability in cloud environments. The insights gained from these experiments provide a solid foundation for future work, including improvements in cost-energy optimization and scalability across larger, more diverse datasets.

While the proposed system demonstrates significant advancements in efficiency, scalability, and cost-energy optimization, it inevitably faces several limitations that need to be acknowledged. First, the system's reliance on specific cloud service providers, such as Amazon Web Services (AWS), could potentially lead to vendor lock-in. This constraint limits the system's portability across multi-cloud environments, restricting its ability to seamlessly migrate workloads and data across different cloud infrastructures. The implications of such dependency extend to resource management, redundancy, and disaster recovery strategies. Addressing this challenge requires integrating multi-cloud support, which would allow the system to interact with various cloud services, thereby enhancing its flexibility and scalability in diverse environments. Future research should focus on designing strategies that facilitate the portability of workloads across heterogeneous cloud platforms.

Second, the cost-energy optimization model embedded within the system may be influenced by cloud pricing variability, particularly with respect to the use of spot instances. Cloud pricing models, especially for spot instances, fluctuate based on market conditions and the demand for cloud resources at any given time. This variability can significantly impact the cost-efficiency of the system, as the optimization model relies on historical data, which may not fully account for sudden price spikes or decreases. Consequently, the accuracy of cost predictions becomes compromised, potentially leading to inefficiencies in resource allocation. A promising avenue for improvement is the integration of dynamic pricing adaptation within the optimization framework. This would involve incorporating real-time cloud pricing information, thereby improving the system's ability to predict and adapt to pricing fluctuations, and ensuring more precise cost-energy optimization.

Third, the cold-start delays encountered during system initialization can adversely affect the performance of latency-sensitive workloads. Such delays occur when resources are instantiated or data retrieval processes are initiated, leading to a temporary degradation in performance. This is particularly problematic in environments that demand rapid response times, such as real-time data processing and high-frequency queries. These delays create bottlenecks that hinder the system's ability to meet stringent latency requirements. To mitigate this, warm-up strategies could be implemented, where critical components or frequently accessed data are preloaded into memory, reducing initialization times and minimizing the cold-start effects on system performance. Future work should explore more sophisticated methods for efficiently managing system startup and optimizing the handling of latency-sensitive workloads.

Finally, the current carbon intensity model used to estimate the environmental impact of cloud resources is based on regional averages of energy consumption and emissions. While regional averages offer a generalized understanding of the energy mix, they fail to capture the real-time fluctuations in energy consumption and carbon emissions that occur at the local level. For instance, renewable energy availability can vary significantly within a region based on time of day, weather conditions, and the overall demand for energy. To improve the accuracy of environmental impact assessments, future research should explore integrating real-time carbon

intensity data into the model, allowing the system to dynamically adjust its resource consumption based on the actual energy mix at any given time. This would lead to more precise carbon footprint calculations and enable the system to operate in a more sustainable manner.

While the proposed system represents a significant advancement in cloud-native data management, addressing these limitations through further research into multi-cloud integration, dynamic pricing models, warm-up strategies, and real-time carbon tracking will be critical in improving its scalability, cost-efficiency, and sustainability. Such advancements will not only enhance the practical applicability of the system but also ensure its resilience and environmental responsibility in increasingly complex and dynamic cloud environments.

5 CONCLUSIONS

This study introduces a comprehensive framework for optimizing cloud-native Lakehouse systems, with a particular emphasis on achieving an equilibrium between performance, cost, and energy consumption while ensuring adherence to stringent service-level agreements (SLAs). Through a series of rigorously designed experiments, we illustrate how the integration of dynamic tiering, query routing, layout optimization, and lineage tracking can enhance both the efficiency and sustainability of the system. The results confirm that the proposed architecture not only improves query performance and reduces resource utilization but also significantly decreases energy consumption and operational costs. Moreover, the system's ability to adapt to diverse workloads and cloud environments positions it as a robust solution for real-world, large-scale data processing tasks.

Despite these advancements, opportunities remain for further refinement and exploration. Future research will aim to enhance energy modeling techniques, expand applicability across multi-cloud architectures, and optimize the system's performance under extreme workloads. Additionally, advancing carbon footprint modeling and developing more sophisticated resource allocation strategies will be essential for maintaining long-term sustainability. The insights derived from this work lay a solid foundation for further innovations in cloud-native data processing systems, with particular attention to balancing performance, cost, and environmental impact.

ACKNOWLEDGMENTS

Not Applicable.

FUNDING

Not Applicable.

INSTITUTIONAL REVIEW BOARD STATEMENT

Not Applicable.

INFORMED CONSENT STATEMENT

Not Applicable.

DATA AVAILABILITY STATEMENT

Not Applicable.

CONFLICT OF INTEREST

Not Applicable.

PUBLISHER'S NOTE

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

AUTHOR CONTRIBUTIONS

Not application.

ABOUT THE AUTHORS

YIN, Min

University of California-Berkeley, US, gmiayinc@gmail.com.

REFERENCES

[1] Chen, Y. (2025). Artificial Intelligence in Economic Applications: Stock Trading, Market Analysis, and Risk Management. Journal of Economic Theory and Business Management, 2(5), 7-14.

[2] Pahune, S., & Akhtar, Z. (2025). Transitioning from MLOps to LLMOps: Navigating the unique challenges of large language models. Information, 16(2), 87.

[3] Sun, Y., & Ortiz, J. (2024). An AI-Based System Utilizing IoT-Enabled Ambient Sensors and LLMs for Complex Activity Tracking. Academic Journal of Science and Technology, 11(3), 277–281.

[4] Chen, Y. (2025). Daily Asset Pricing Based on Deep Learning: Integrating No-Arbitrage Constraints and Market Dynamics. Journal of Computer Technology and Applied Mathematics, 2(6), 1-10.

[5] Lin, A. (2025). Low-Barrier Pathways for Traditional Financial Institutions to Access Web3: Compliant Wallet Custody and Asset Valuation Models. Frontiers in Management Science, 4(6), 80-86.

[6] Qi, Z. (2025). Root Cause Tracing Algorithm and One-Click Repair Mechanism for Medical Server Failures. Journal of Progress in Engineering and Physical Science, 4(5), 43-48.

[7] Qi, Z. (2025). Design of a Medical IT Automated Auditing System Based on Multiple Compliance Standards. Innovation in Science and Technology, 4(9), 17-23.

[8] Bhimji, W., Carder, D., Dart, E., Duarte, J., Fisk, I., Gardner, R., ... & Würthwein, F. (2023). Snowmass 2021 computational frontier compf4 topical group report storage and processing resource access. Computing and Software for Big Science, 7(1), 5.

[9] Chen, Y., & Xu, J. (2026). Deep Learning for US Bond Yield Forecasting: An Enhanced LSTM–LagLasso Framework. ICCK Transactions on Emerging Topics in Artificial Intelligence, 3(2), 61-75.

[10] Yin, M., & Frank, L. F. (2026). Multi-Modal Fusion for Yield Optimization: Integrating Wafer Maps, Metrology, and Process Logs with Graph Models. ICCK Transactions on Emerging Topics in Artificial Intelligence, 3(1), 45-60.

[11] Qi, Z. (2025). Design and Practice of Elastic Scaling Mechanism for Medical Cloud-Edge Collaborative Architecture. Journal of Innovations in Medical Research, 4(5), 13-18.

[12] Luo, M., Du, B., Zhang, W., Song, T., Li, K., Zhu, H., ... & Wen, H. (2023). Fleet rebalancing for expanding shared e-Mobility systems: A multi-agent deep reinforcement learning approach. IEEE Transactions on Intelligent Transportation Systems, 24(4), 3868-3881.

[13] Yin, M. (2025). Drift-Aware Streaming Predictive Maintenance for Semiconductor Equipment.

[14] Petrovic, A. J. (2025). Design of Secure and Fault-Tolerant Patterns for AI-Driven Healthcare Analytics in Hybrid Cloud Platforms. International Journal of Research and Applied Innovations, 8(6), 13027-13033.

[15] Gujjala, P. K. R. (2023). The Future of Cloud-Native Lakehouses: Leveraging Serverless and Multi-Cloud Strategies for Data Flexibility. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 868-882.

[16] Chen, Y. (2025). Leveraging LSTM Networks for Vehicle Stability Prediction: A Comparative Analysis with Traditional Models under Dynamic Load Conditions. Computing and Interdisciplinary Science, 1(2), 15-22.

[17] Li, K., Chen, X., Song, T., Zhou, C., Liu, Z., Zhang, Z., Guo, J., & Shan, Q. (2025a, March 24). Solving situation puzzles with large language model and external reformulation.

[18] Wang, H. (2022). Supervised Learning for Complex Data (Doctoral dissertation, The University of North Carolina at Chapel Hill).

[19] Aires, V. A. J. (2025). Optimising Energy Analytics: a Dimensional Modelling Approach for Enhanced Decision-Making (Master's thesis, Universidade NOVA de Lisboa (Portugal)).

[20] Chen, Y. (2025). A Comparative Study of Machine Learning Models for Credit Card Fraud Detection. Academic Journal of Natural Science, 2(4), 11-18.

[21] Wang, H., Li, Q., & Liu, Y. (2024). Multi-response Regression for Block-missing Multi-modal Data without Imputation. Statistica Sinica, 34(2), 527.

[22] Kretzer, A. R., Benitti, F. B., & Siqueira, F. (2025). Challenges and Opportunities in Big Data Analytics for Industry 4.0: A Systematic Evaluation of Current Architectures. IEEE Access.

[23] Nyunt, A. T., Kotak, B., Chauhan, R., Jain, R., Parmar, K. J., Palaniappan, D., & Premavathi, T. (2026). Next Generation Data Warehousing for Destination Marketing With Big Data Technologies. In Maximizing Destination Marketing Strategies in the Digital Era (pp. 157-194). IGI Global Scientific Publishing.

[24] Wang, H., Li, Q., & Liu, Y. (2023). Adaptive supervised learning on data streams in reproducing kernel Hilbert spaces with data sparsity constraint. Stat, 12(1), e514.

[25] Nuthalapati, A. (2025). Scaling AI Applications on the Cloud toward Optimized Cloud-Native Architectures, Model Efficiency, and Workload Distribution. International Journal of Latest Technology in Engineering, Management & Applied Science, 14(2), 200-206.