Svedberg Open
DISSEMINATION OF KNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/



Research Paper Open Access

A Case Study On Snowflake And AWS Integration: Building Scalable, Cloud-Native Data Platforms

Rambabu Bolineni

Independent Researcher.

Abstract

The convergence of Snowflake and Amazon Web Services has produced one of the most consequential architectural combinations in enterprise data engineering. Snowflake is not merely deployed on AWS — it is built on it, with Amazon S3 as its persistent storage substrate and EC2 as the compute layer behind its virtual warehouses. This structural co-location creates integration opportunities whose value is multiplicative rather than additive: the same S3 objects that Snowflake reads through external stages are accessible to AWS Glue, Athena, Lambda, and SageMaker without data movement, synchronization overhead, or access control gaps. This article examines the Snowflake-AWS integration through a case study lens, analyzing architectural patterns and engineering decisions across five domains: data ingestion from AWS services into Snowflake, transformation boundary decisions between Glue and Snowflake SQL, orchestration across Step Functions and Snowflake Tasks, security and governance continuity across the integration boundary, and ML pipeline integration via Snowpark and SageMaker. Three applied case studies—an advisor compensation pipeline in financial services, a HIPAA-regulated healthcare data integration, and an FDIC 370 regulatory compliance pipeline—ground the architectural analysis in the regulated environments where these decisions carry audit consequences. The central argument is that the Snowflake-AWS integration produces multiplicative rather than additive value, but only when integration points are engineered deliberately. S3 as a shared data substrate, PrivateLink for secure private connectivity, Snowpipe for event-driven ingestion, Snowpark for continuous ML feature engineering governance, and unified audit trails spanning CloudTrail and Snowflake ACCESS_HISTORY together produce a platform whose compliance and performance properties are structurally superior to either platform deployed in isolation.

Keywords: Snowflake, AWS integration, cloud-native data platform, S3 data substrate, Snowpipe, Snowpark, Step Functions, regulatory compliance, financial services data engineering.

1. Introduction

Snowflake and AWS are each dominant in their respective domains, and their co-existence in enterprise data architectures is common. What is less common is the deliberate exploitation of the integration opportunities their shared infrastructure creates. Many organizations deploy Snowflake on AWS and use each platform independently—treating Snowflake as a self-contained analytical system and AWS as a networking substrate—rather than as complementary components of a unified data platform [1][21]. The result is a platform that delivers the sum of its parts rather than the product of their integration.

The integration opportunity is structural, not incidental. Snowflake's persistent storage layer is Amazon S3 — not an abstraction over S3, but S3 itself [1][2]. The same S3 bucket that Snowflake reads through an external stage can be processed by AWS Glue, queried by Athena, enriched by Lambda, and used for SageMaker model training without copying data, without synchronization jobs, and without the access control gaps that accompany data duplication [22]. This shared accessibility is a consequence of how Snowflake is built, not a configuration option, and exploiting it requires understanding the integration at the architectural level rather than treating both platforms as black boxes.

The analytical position of this article is precise: the Snowflake-AWS integration produces multiplicative rather than additive value, but only when integration points are engineered deliberately. An organization that deploys Snowflake on AWS and uses each platform independently receives the sum of their individual capabilities. An organization that integrates them through S3 as a shared substrate, Snowpipe triggered by AWS S3 events, Step

Functions coordinating cross-service workflows, PrivateLink providing secure private connectivity, Snowpark maintaining governance continuity through ML feature engineering, and unified audit trails spanning CloudTrail and ACCESS_HISTORY receives capabilities that neither platform provides independently—and a compliance architecture that neither platform produces in isolation.

The scope of this case study spans five architectural domains. Section 2 establishes the structural foundations of Snowflake on AWS. Sections 3 through 5 examine the three primary engineering domains: ingestion, transformation, and orchestration. Section 6 addresses the governance and security implications of the integration boundary. Section 7 presents three applied case studies from regulated industries. Section 8 examines cost architecture. Section 9 synthesises the findings and identifies open questions.

2. Architectural Foundations: Snowflake on AWS

The integration patterns described in subsequent sections are only intelligible in light of how Snowflake is architecturally constructed on AWS infrastructure. This is not background—it is the engineering basis for every integration decision that follows [1][2].

2.1 Snowflake's AWS Infrastructure Layer

Snowflake's three-layer architecture maps onto AWS infrastructure components in ways that most practitioners do not fully internalize. The storage layer is Amazon S3: all table data, including micropartitions with their column-level metadata, is persisted in S3 buckets managed by Snowflake's service layer [1]. The compute layer runs on EC2 instances, provisioned and managed by Snowflake as virtual warehouses. The service layer—handling query parsing, optimization, metadata management, and access control—runs on AWS compute managed entirely by Snowflake [1][4].

The implication for platform architects is that Snowflake is not a separate cloud operating on top of AWS. It is a managed service that uses AWS as its physical substrate. The storage layer is not virtualized S3—it is S3. This means that the data Snowflake manages is physically accessible to AWS services with appropriate configuration, and that integration patterns based on S3 accessibility are exploitations of the platform's underlying architecture, not workarounds [1][2].

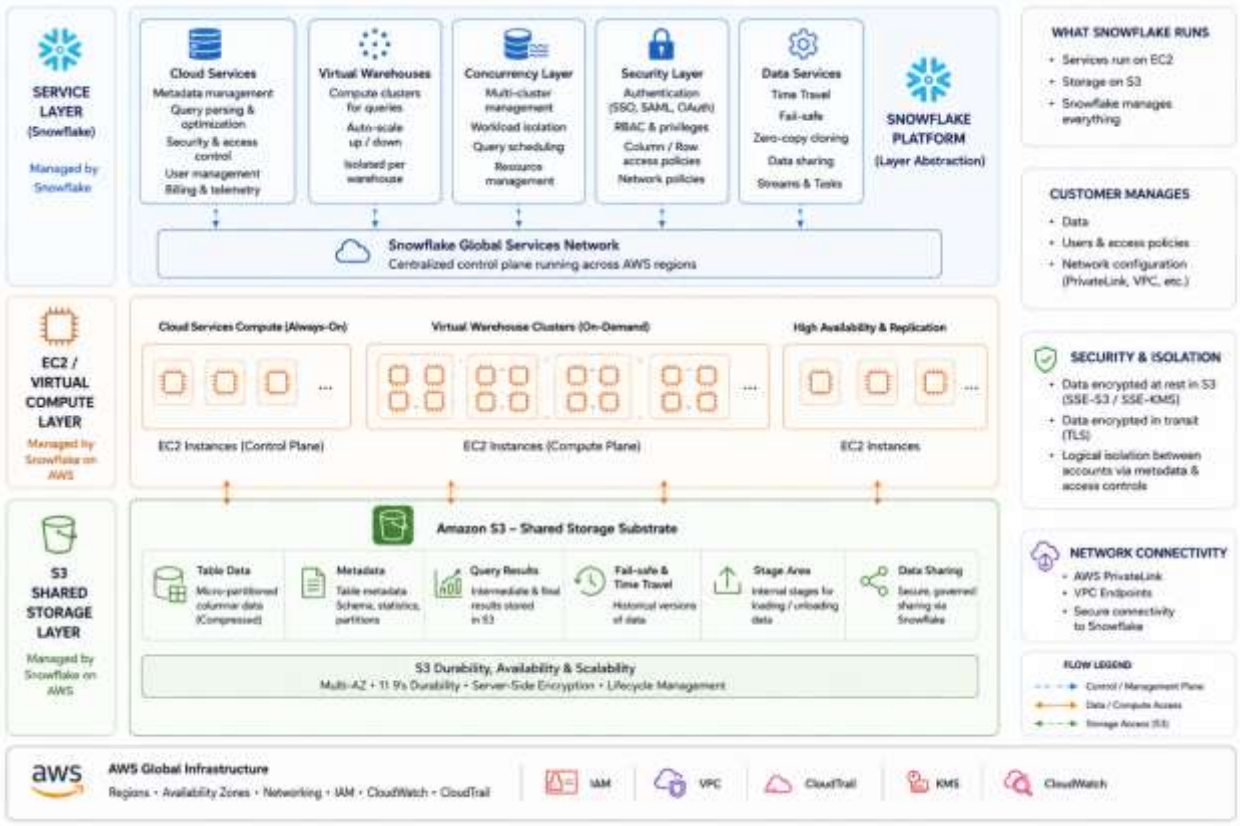


Fig. 1. Snowflake-AWS Three-Layer Infrastructure

2.2 S3 as Shared Data Substrate

The most consequential integration property that follows from Snowflake's AWS architecture is S3's role as a shared data substrate. The same S3 objects that Snowflake reads through an external stage can be read by Glue for Spark-based transformation; by Athena for SQL querying; by Lambda for event-driven processing; and by SageMaker for ML training data access—simultaneously, without data movement, and without the version drift and access control gaps that copied datasets introduce [2][22].

This shared accessibility enables a class of pipeline patterns that are structurally impossible in systems where the warehouse storage layer is opaque to external compute. A Glue job can preprocess raw data and write transformed output to the exact S3 path that Snowpipe monitors for auto-ingest. SageMaker can read training data from the same S3 external stage that Snowflake uses for analytics. A Lambda function can validate and enrich S3 objects before Snowflake reads them—none of these operations requiring data to pass through Snowflake's ingestion pipeline [1][2][22]. The engineering value is that data movement overhead—historically a significant fraction of complex pipeline latency and cost—can be eliminated by design rather than reduced through optimization.

2.3 AWS PrivateLink for Secure Connectivity

Snowflake's AWS PrivateLink integration provides private connectivity between a customer's VPC and their Snowflake account, routing all traffic through AWS's internal network rather than the public internet [3]. Lambda functions and Glue jobs executing within the VPC connect to Snowflake through the PrivateLink VPC endpoint without internet gateway or NAT gateway traversal. Combined with VPC endpoints for S3 and other AWS services, PrivateLink enables a fully private network architecture for the entire integration: ingestion, transformation, query execution, and API operations all occur within the private network boundary [3][17].

The implementation discipline that practitioners most frequently overlook is security group rule enforcement. PrivateLink is a network path, not a network control—Lambda and Glue security group rules must restrict outbound traffic to the PrivateLink endpoint and VPC endpoints only, preventing execution paths that route traffic through a NAT gateway to the public internet. Without this restriction, PrivateLink is a configuration that can be bypassed rather than a network isolation guarantee — a distinction that regulated industry network security reviews will identify [3][17].

3. Data Ingestion Architecture: AWS to Snowflake

Five ingestion patterns connect AWS data sources to Snowflake, each with distinct latency profiles, cost models, throughput characteristics, and governance implications. Selecting the right pattern requires matching these properties to the pipeline's actual SLA, data volume, and compliance requirements rather than defaulting to the most familiar mechanism [5][6][13].

Table 1. Five Ingestion Pattern Decision Matrix

Metric	Value	Unit/Qualifier
Recommended COPY INTO staged file size	100–250	MB compressed
Snowpipe data availability	Minutes	qualitative
Kinesis Firehose → S3 → Snowpipe latency	2–5	minutes (typical)
Step Functions Standard history retention	90	days
AES encryption at rest (Snowflake)	256	bit
External table: no micropartition pruning	N/A	qualitative
Glue cost components vs. Snowflake SQL	3 vs. 1	DPU-hours + S3 + Snowpipe vs. credits
Sculley et al. [20]: undeclared ML dependencies	Qualitative	systemic risk finding

3.1 S3 Staging and COPY INTO: Bulk Loading

COPY INTO from an S3 external stage is the highest-throughput ingestion pattern for large-volume batch loads. Data staged in S3—written by Glue transformation jobs, delivered by Firehose, or produced by upstream systems—is loaded into Snowflake in bulk, exploiting parallel loading across warehouse compute nodes [4][5]. File format selection directly affects load throughput: Parquet files load faster than CSV because Snowflake maps columnar file structure directly to micropartitions without row-by-row parsing overhead. Files sized between 100 MB and 250 MB compressed produce well-formed micropartitions; files outside this range degrade micropartition quality and load performance [4][5].

For financial services pipelines loading daily transaction extracts, weekly compensation files, or monthly regulatory datasets, COPY INTO delivers the highest throughput at the lowest warehouse credit cost. The engineering discipline is staging quality: file format, size, and compression before COPY INTO execution determine whether the load operation exploits Snowflake's bulk-loading optimizations or works against them.

3.2 Snowpipe Auto-Ingest: Event-Driven Continuous Loading

Snowpipe auto-ingest monitors S3 buckets for new objects using SQS event notifications, triggering automatic loading as files arrive [5]. The SQS integration creates a direct event-driven connection between the AWS S3 event system and Snowflake's ingestion pipeline — when a Glue job, Lambda function, or Firehose delivery writes a new file to S3, the SQS notification triggers Snowpipe without additional orchestration [5][6]. Data is available for querying within minutes of file arrival, using Snowflake-managed serverless compute billed per second rather than per warehouse-hour.

The dead letter queue configuration for the SQS notification queue is mandatory for regulated pipelines. S3 event notifications that fail Snowpipe processing must be captured and recoverable rather than discarded, because a lost ingestion event translates directly to missing data in the analytical system. For FINRA data integrity requirements and FDIC completeness obligations, the SQS DLQ is a compliance control, not an optional reliability feature [5][13].

3.3 Snowpipe Streaming via Kafka Connector

For streaming pipelines originating from Apache Kafka or Amazon MSK, Snowflake's Kafka connector enables direct streaming ingestion through the Snowpipe Streaming API, bypassing the file-based micro-batch model of standard Snowpipe [5][8]. The connector writes records in micro-batches using a streaming channel API that provides lower end-to-end latency than file-based Snowpipe at the cost of smaller micropartitions on recently ingested data. For real-time financial services use cases—intraday trade event processing and real-time payment monitoring—the latency benefit justifies the micropartition quality cost, with periodic automatic clustering configured to restore partition alignment after high-volume streaming periods [5, 8].

3.4 AWS Glue to Snowflake JDBC

Glue jobs can write directly to Snowflake via the Snowflake JDBC connector, eliminating the intermediate S3 staging step. This pattern is appropriate when the Glue job performs complex pre-load transformation—PII masking, schema normalization, and CDC application—and the output dataset is small enough that JDBC write throughput is acceptable [7][13]. For large output datasets, staging to S3 and using COPY INTO consistently outperforms direct JDBC writes because COPY INTO exploits Snowflake's parallel bulk loading architecture while JDBC writes are inherently row-oriented. The engineering decision is output dataset size: small outputs favor JDBC for pipeline simplicity; large outputs favor S3 staging for throughput [7].

3.5 Kinesis Firehose to S3 to Snowpipe: Managed Streaming Delivery

The Kinesis → Firehose → S3 → SQS → Snowpipe chain provides fully managed streaming ingestion from Kinesis sources into Snowflake without custom infrastructure [5][6]. Firehose's buffer configuration controls the file sizes delivered to S3 and therefore the micropartition quality of ingested data. End-to-end latency from event production to Snowflake availability is typically two to five minutes, determined primarily by Firehose's buffer interval [5][6]. For financial services streaming use cases where minute-scale latency is acceptable, this managed pattern eliminates the operational overhead of a Kafka cluster while providing comparable throughput for most transaction volumes.

4. Transformation Architecture: AWS Glue and Snowflake SQL

The transformation boundary decision — what to compute in Glue versus what to compute in Snowflake SQL — is the most consequential architectural choice in a Snowflake-AWS integration. It determines where computation costs accrue, where governance controls apply, where audit trails are produced, and how operational complexity is distributed across the platform boundary [7][9][21].

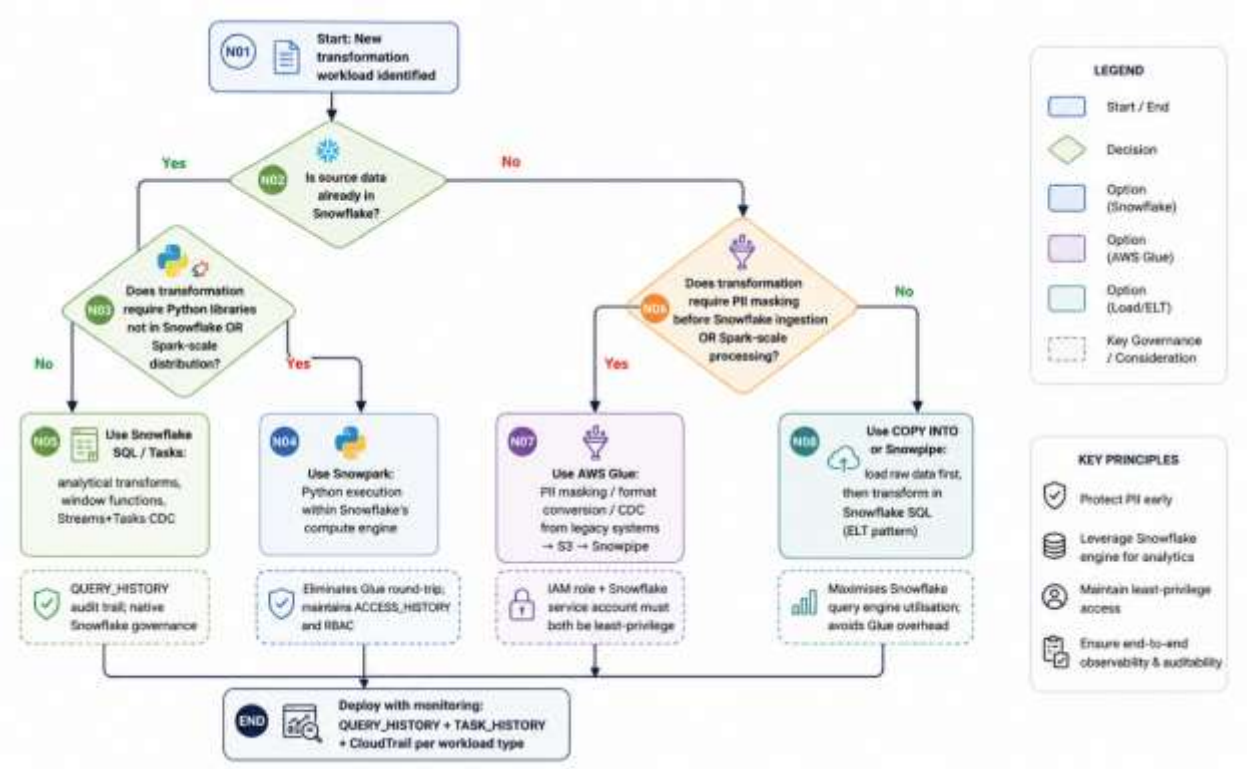


Fig. 2. Transformation Boundary Decision Framework

4.1 The ELT vs. ETL Split

Traditional ETL architectures transform data in a separate compute layer before loading into the target system. Cloud-native ELT architectures load raw data into the target system first and perform transformation using the target's compute engine, exploiting the target's scale and query optimization rather than requiring a separate transformation cluster [21]. In a Snowflake-AWS architecture, both models are available, and the optimal choice depends on specific transformation requirements rather than architectural preference.

Glue is appropriate for transformations that benefit from Spark's distributed processing model, require Python libraries not available in Snowflake, or must occur before data enters Snowflake's governance model — particularly PII masking, where raw sensitive values must never be visible to any Snowflake role [7][21]. Snowflake SQL is appropriate for analytical transformations on data already in Snowflake: complex multi-table joins, window functions, aggregations, and incremental CDC processing using Streams and Tasks, where Snowflake's query engine is optimized for the workload and the governance model applies consistently [4][11].

4.2 Glue for Pre-Load Transformation

Three categories of transformation are most appropriately performed in Glue before Snowflake ingestion. PII masking and data classification are the most compliance-critical: applying tokenization or masking to sensitive fields before any Snowflake role can access raw values satisfies a requirement that some regulated contexts impose categorically—raw PII values must never exist in the analytical environment, even transiently [7][17]. Schema normalization and format conversion—transforming the source CSV or XML to Parquet, resolving cross-source schema inconsistencies, and enforcing the target schema—are operationally simpler in Glue's Python environment than in Snowflake SQL. CDC applications from legacy source systems producing non-standard change log formats are frequently better handled in Glue, where Python's string manipulation capabilities address format idiosyncrasies that Snowflake SQL cannot [7][13].

4.3 Snowflake SQL for Post-Load Transformation

Post-load transformation in Snowflake SQL exploits the warehouse's execution engine for the workloads it handles best. Multi-table joins on large fact and dimension tables, window functions over time-series transaction data, complex aggregations for regulatory reporting views, and incremental CDC processing using Streams and Tasks are workloads where Snowflake SQL is faster, more cost-efficient, and better governed than an equivalent Glue Spark job [4][11]. Snowflake Tasks enable scheduled and stream-triggered SQL execution

within Snowflake, supporting intra-warehouse transformation DAGs without external orchestration [11]. Every SQL statement executed by a Task is recorded in QUERY_HISTORY with the executing role, warehouse, and timestamp, producing an audit trail that Glue job scripts do not generate natively [4][14].

4.4 Snowpark as the Integration Bridge

Snowpark collapses the Glue-Snowflake transformation boundary by enabling Python code to execute within Snowflake's compute engine, compiled to Snowflake SQL execution plans [9]. A transformation workload that previously required extracting data from Snowflake to a Glue cluster, processing it in Python, and loading results back executes entirely within Snowflake using Snowpark's DataFrame API — without the data movement overhead, the governance gap, or the lineage break that the Glue round-trip introduces.

The governance consequence is decisive for regulated pipelines. Snowpark transformations execute under Snowflake's RBAC model, produce ACCESS_HISTORY log entries, respect dynamic data masking policies, and maintain continuous data lineage within Snowflake's governance framework [9][20]. A glue-based extraction breaks this continuity: data operating outside Snowflake's governance model during the extraction creates a lineage gap that is difficult to close for FINRA, FDIC, and HIPAA audit purposes. For Python-based transformation workloads that can execute within Snowflake's compute constraints, Snowpark eliminates this gap structurally rather than through supplementary instrumentation [9][20].

5. Orchestration: Step Functions and Snowflake Tasks

Two orchestration layers coordinate execution in a Snowflake-AWS integration: Step Functions for cross-service workflows spanning AWS and Snowflake, and Snowflake Tasks for intra-warehouse transformation pipelines. The discipline of assigning the right operations to the right orchestration layer produces pipelines that are simpler, more observable, and more auditable than architectures that force one layer to handle both scopes [11][12].



Fig. 3. Orchestration Boundary Design

5.1 Step Functions for Cross-Service Coordination

Step Functions is the appropriate orchestration layer for workflows that cross AWS service boundaries and invoke Snowflake operations as part of a broader pipeline. Invoking Snowflake from Step Functions requires a Lambda intermediary: the state machine submits a Snowflake SQL statement or stored procedure call via a

Lambda function that connects through the Snowflake Python connector or JDBC, then polls for completion status before advancing to the next state [12][13]. This pattern enables Step Functions to coordinate Snowflake operations with the same retry logic, error handling, and execution history it provides for native AWS service invocations.

A pipeline that ingests data from Kinesis, transforms it in Glue, loads it via Snowpipe, executes a Snowflake task DAG, and extracts a reporting result to S3 is fully coordinated within a single Step Functions state machine—with a complete execution history that records every state transition, input, output, and timestamp across the entire cross-platform workflow [12]. For regulated pipelines where the operations team must demonstrate end-to-end pipeline execution integrity, this unified execution history is the primary audit evidence.

5.2 Snowflake Tasks for Intra-Warehouse Orchestration

Snowflake Tasks provide native scheduling and DAG execution for SQL-based pipelines running fully in Snowflake [11]. A root task (triggered by a CRON schedule or a stream detecting new data) initiates a DAG of dependent tasks, each performing a transformation phase and delivering its results to Snowflake tables. TASK_HISTORY records the execution status, duration, error messages, and SQL run by each task run. It provides intra-warehouse observability that complements CloudTrail's AWS-layer logging [11][14]. Tasks are less operationally complex than Step Functions for intra-warehouse pipelines Lambda-intermediary pattern: No cross-service networking, no Lambda function maintenance, no state machine definition for workloads that are completely in Snowflake.

5.3 Orchestration Boundary Design

Operations over the AWS-Snowflake service boundary should be owned by Step Functions. Snowflake Tasks should own operations that stay in Snowflake. The bridge is provided by the Lambda intermediate for the Step Functions workflow that is triggered by a Snowflake Task DAG and waits for its completion. Step Functions makes a request for task execution through Lambda, polls TASK_HISTORY for progress status, and progresses accordingly [11][12][13]. This boundary design results in two complementary audit trails: Step Functions execution history for the cross-platform orchestration layer, and Snowflake TASK_HISTORY and QUERY_HISTORY for the intra-warehouse execution layer . Each of these trails is correlated by propagating a Snowflake session parameter as a pipeline execution identifier [14][16] to achieve the integrated lineage record required by FINRA, FDIC, and HIPAA compliance reviews.

6. Security and Governance Across the Integration Boundary

The AWS-Snowflake integration boundary is a governance risk surface: data and control plane operations crossing from IAM-governed AWS resources to RBAC-governed Snowflake resources must maintain consistent access controls, encryption continuity, and unbroken audit trail coverage [14][15][17]. The risk is not that either platform is insecure—both provide mature security models—but that inconsistency at the boundary produces gaps that neither platform's security model closes independently.

6.1 IAM-to-RBAC Alignment Across the Boundary

Service accounts operating across the integration boundary require privilege scoping in both systems. A Glue job reads from S3 and writes to Snowflake. The Glue job has an IAM execution role for S3 access. The Glue job connects to Snowflake under a service account for Snowflake access. The IAM role should hold s3:GetObject on the specific source S3 prefix. The Snowflake service account should hold INSERT on the specific target Snowflake schema—not SYSADMIN or database-level CREATE privileges [14][15]. Least privilege must be applied consistently across both access control systems, not selectively. An organization that maintains narrow IAM roles but assigns Snowflake service accounts the SYSADMIN role has applied the principle on one side of the boundary and ignored it on the other—a finding that FINRA and FDIC examiners identify when reviewing service account privileges across both systems [15][18].

6.2 Network Security: PrivateLink and Enforcement

All AWS-to-Snowflake traffic in regulated pipelines should route through PrivateLink rather than public HTTPS endpoints [3][17]. Combining PrivateLink for Snowflake connectivity with VPC endpoints for S3 and other AWS services creates a fully private network architecture. The NIST Zero Trust Architecture framework [17] recommends treating every network path as a potential trust boundary rather than relying on perimeter controls. PrivateLink operationalizes this principle for the AWS-Snowflake connectivity path—but only when security group rules enforce that no alternative path exists. Without explicit outbound traffic restrictions, PrivateLink provides a preferred path, not an exclusive one, which is insufficient for regulated environments where network isolation is a hard control requirement [3][17].

6.3 Encryption Continuity Across the Boundary

Data flowing from S3 into Snowflake should maintain encryption key custody at both rest and transit boundaries. S3 objects encrypted with SSE-KMS are decrypted by Snowflake's service layer during COPY INTO or Snowpipe ingestion and re-encrypted in Snowflake's micropartition format using Snowflake's key hierarchy [4][14]. For organizations requiring continuous key custody across the full boundary, Snowflake's Tri-Secret Secure configuration ensures that Snowflake cannot decrypt data without the customer's AWS KMS key—providing end-to-end customer-controlled encryption from S3 storage through Snowflake's micro-partition layer [14].

6.4 Unified Audit Trail Construction

A complete pipeline audit trail involves correlation of log sources from both platforms: AWS CloudTrail for API-level events, Snowflake ACCESS_HISTORY for data access events, and Step Functions execution history for orchestration events [16][14][12]. A common pipeline execution identifier is generated at Step Functions invocation and propagated as a CloudTrail request context. This identifier is used as the Step Functions execution name and passed as a Snowflake session parameter so that ACCESS_HISTORY entries carry the identifier. This allows these three sources to be correlated. This unified trail answers the full scope of examiner inquiries for FINRA, FDIC, and HIPAA compliance reviews: what AWS API calls were made, what data did those operations access in Snowflake, and what was the entire execution context of the pipeline run [14][16][18].

7. Real-World Case Studies: Financial Services and Healthcare

Three applied case studies illustrate how the integration patterns described in Sections 3 through 6 compose production pipeline architectures in the regulated environments the author has operated in professionally. Each case study demonstrates that compliance properties emerge from integration architecture rather than from supplementary instrumentation.

7.1 Case Study A — Advisor Compensation Pipeline (FINRA)

An enterprise wealth management firm processing millions of daily trade and compensation records under FINRA data integrity requirements requires an architecture that offers near-real-time data availability, complex multi-level compensation calculation logic, and auditable regulatory reporting extracts. Trade events from internal trading systems are sent to Kinesis Data Streams. Firehose buffers and sends to the S3 bronze layer, which triggers Snowpipe to auto-ingest into Snowflake staging tables. A Glue job reads in compensation hierarchy data from a traditional HR system, normalizes it, applies schema standardization, and writes it to S3 for ingestion via Snowpipe into Snowflake hierarchy tables. Snowflake Tasks run the compensation calculation DAG (payout rules, fee schedules, and multi-level overrides) in Snowflake SQL on the imported trade and hierarchy data [5][6][11].

Step Functions orchestrates the regulatory reporting extraction cycle: a Lambda function triggers the reporting stored procedure in Snowflake, Step Functions waits for completion confirmation via TASK_HISTORY polling, and then a Lambda function extracts the reporting view to S3 for compliance delivery [12]. The audit trail is structural: CloudTrail records every S3 write and Lambda invocation; ACCESS_HISTORY records every table access in the compensation calculation DAG; TASK_HISTORY records every task execution with timing and status. Step Functions execution history records the full pipeline run, including retries. The unified trail satisfies FINRA Rule 4370 data integrity requirements without supplementary logging infrastructure, a direct consequence of the integration architecture [14][16][18].

7.2 Case Study B — Healthcare Data Integration (HIPAA)

A digital health platform integrating biometric, eligibility, and telehealth data requires HIPAA compliance at every pipeline stage: PII must never appear in the analytical environment in raw form, access to protected health information must enforce the minimum necessary standard, and all data access must be auditable without exception. A Glue job applies PII masking before any Snowflake role encounters raw values—tokenizing member identifiers, masking social security numbers, and generalizing geographic identifiers—writing the masked dataset to S3 for Snowpipe ingestion [7][17]. Snowflake dynamic masking policies enforce role-differentiated PHI access: a data analyst role sees masked values, a compliance officer role sees unmasked values, and an audit role logs all access via ACCESS_HISTORY [14].

Snowpark executes ML feature engineering directly within Snowflake—computing biometric risk scores, engagement indices, and clinical event features—without extracting data to an external compute environment. SageMaker retrieves governed feature datasets via the Snowflake Connector for Python, training models on data that has maintained governance continuity from raw ingestion through Glue masking, Snowflake RBAC, and Snowpark feature computation [9][19][20]. Prediction outputs are written back to Snowflake for operational consumption. The pipeline satisfies HIPAA's minimum necessary standard at every stage because

access constraints are structural properties of each component—not post-deployment overlays applied to an architecture that was not designed for them.

7.3 Case Study C — FDIC 370 Regulatory Compliance Pipeline

FDIC Part 370 requires banks to produce accurate depositor balance records with insurance classifications applied across ownership categories within defined timelines following a bank failure event. Data from multiple banking systems—core deposits, trust accounts, and retirement accounts—is ingested via S3 staging into Snowflake using Snowpipe. A Glue CDC job processes change deltas from legacy DataStage-based source systems, normalizing schemas and resolving depositor identity across sources before Snowflake ingestion [7][11].

Snowflake tasks execute the FDIC classification DAG—applying insurance rules across ownership categories, computing insured and uninsured balances, and flagging exception cases for manual review. Step Functions orchestrates the scheduled regulatory reporting extraction: triggering the Snowflake classification task DAG via Lambda, waiting for TASK_HISTORY completion confirmation, extracting the reporting view to S3, and notifying the compliance team via SNS [11][12]. The combined audit trail—Step Functions execution history, Snowflake TASK_HISTORY, Snowflake ACCESS_HISTORY, and CloudTrail—provides the data lineage record that FDIC examiners request when reviewing classification methodology and depositor data provenance [14][16].

8. Cost Architecture and Optimisation

A Snowflake-AWS integration operates under two simultaneous cost models: AWS per-usage billing for Lambda, Glue, Kinesis, and S3 and Snowflake credit-based billing for virtual warehouse compute and storage. Managing these two cost models requires visibility into both simultaneously, which neither AWS Cost Explorer nor Snowflake's cost monitoring tools provide in isolation [4][21].

8.1 Compute Placement and Cost Distribution

Performing a transformation in Glue that could be performed in Snowflake SQL incurs three cost components: Glue DPU-hours for computation, S3 write costs for staging the output, and Snowpipe serverless compute for ingestion. The same transformation executed in Snowflake SQL incurs one cost component: warehouse credit consumption [7][21]. For transformations on data already in Snowflake where Snowflake SQL is capable of performing the computation, Glue adds cost without architectural benefit. The converse also applies: performing large-scale data processing in Snowflake that is better suited to Glue's Spark distributed architecture forces virtual warehouses to perform workloads they are not optimized for, producing higher credit consumption than an equivalent Glue job would require [7][10]. Cost-optimal compute placement follows the capability boundary.

8.2 S3 Storage vs. Snowflake Storage

Data stored in Snowflake's managed micropartition format is billed at Snowflake's storage rate; data stored in S3 as external tables is billed at S3's substantially lower per-GB rate. For large historical datasets queried infrequently—cold regulatory archives, historical reference data, and long-tail transaction history—external table storage eliminates Snowflake storage costs while maintaining query accessibility [1][4]. The tradeoff is query performance: external table queries cannot exploit Snowflake's micropartition pruning or result caching. Cold data with infrequent access belongs in S3 external tables; hot data queried frequently belongs in native Snowflake tables, where the performance premium is justified by query frequency [1][4].

8.3 Unified Cost Attribution

AWS Cost Explorer filtered by pipeline-level resource tags provides the AWS cost component per pipeline. Snowflake's WAREHOUSE_METERING_HISTORY provides credit consumption per virtual warehouse attributable to the pipeline workloads running on each dedicated warehouse. Constructing a unified cost view requires a scheduled reporting job that queries both sources and aggregates results on a shared pipeline identifier—joining AWS cost data with Snowflake credit consumption by the same tag applied consistently at both resource provisioning stages [4][21]. For regulated financial services environments where business unit cost allocation is an internal audit requirement, this unified attribution transforms cost management from a multi-platform manual exercise into a reportable, automated deliverable.

9. Conclusion

The Snowflake-AWS integration examined in this article is not a product feature — it is an architectural relationship that emerges from Snowflake's physical construction on AWS infrastructure. S3 is not merely

Snowflake's storage backend; it is a shared data substrate that makes the data Snowflake manages simultaneously accessible to Glue, Athena, Lambda, SageMaker, and every other AWS service that reads from S3. That structural fact enables integration patterns whose value is multiplicative: event-driven ingestion where AWS S3 events trigger Snowpipe automatically; transformation boundary decisions that optimize computation cost and governance continuity simultaneously; orchestration that spans Step Functions and Snowflake Tasks with a unified audit trail; and ML feature engineering via Snowpark that maintains Snowflake's governance model through the full model lifecycle.

The three case studies—advisor compensation, HIPAA healthcare integration, and FDIC 370 compliance—demonstrate that these integration properties are not theoretical. In each case, the compliance audit trail, the access control architecture, and the pipeline performance characteristics emerge directly from the integration design rather than from supplementary instrumentation. FINRA's data integrity trail, HIPAA's minimum necessary access enforcement, and FDIC's lineage traceability are structural consequences of building the integration deliberately — not bolt-on compliance overlays applied after the architecture is designed.

Several open questions remain. As Snowpark matures, the optimal transformation boundary between Glue and Snowpark will shift toward Snowflake for more workload types—the current analysis reflects 2023 capabilities, and that boundary is not static. Cross-platform cost optimization tooling—unified dashboards aggregating AWS and Snowflake cost data without custom reporting pipelines—remains an underserved need for organizations managing both platforms at scale. Governance continuity in multi-cloud Snowflake deployments, where Snowflake instances operate across AWS and Azure and data moves between cloud providers, introduces cross-cloud IAM alignment challenges that the patterns described here do not fully address.

The closing argument is precise: platform integration engineering is a discipline distinct from platform expertise. Deep knowledge of Snowflake and deep knowledge of AWS do not automatically produce a well-integrated platform—they produce two platforms that co-exist. The integration emerges from deliberate architectural decisions at the five engineering boundaries examined in this article: ingestion pattern selection, transformation boundary placement, orchestration layer assignment, governance boundary alignment, and unified audit trail construction. For practitioners designing data platforms in regulated industries, those decisions determine not just how the platform performs but whether it is auditable, defensible, and compliant.

References

[1] B. Dageville et al., "The Snowflake Elastic Data Warehouse," Proc. ACM SIGMOD, pp. 215–226, 2016. https://www.cs.cmu.edu/~15721-f24/papers/Snowflake.pdf

[2] M. Vuppalapati et al., "Building an Elastic Query Engine on Disaggregated Storage," Proc. USENIX NSDI, 2020. https://www.usenix.org/system/files/nsdi20-paper-vuppalapati.pdf

[3] Snowflake Inc., Snowflake Documentation: AWS PrivateLink, docs.snowflake.com, 2023. https://docs.snowflake.com/en/user-guide/data-load-snowpipe-intro

[4] Snowflake Inc., Snowflake Documentation: Virtual Warehouses, docs.snowflake.com, 2023. https://docs.snowflake.com/en/user-guide/data-load-snowpipe-intro

[5] Snowflake Inc., Snowflake Documentation: Snowpipe, docs.snowflake.com, 2023. https://docs.snowflake.com/en/user-guide/data-load-snowpipe-intro

[6] Amazon Web Services, Amazon Kinesis Data Streams Developer Guide, docs.aws.amazon.com, 2023. https://docs.aws.amazon.com/pdfs/streams/latest/dev/kinesis-dg.pdf

[7] Amazon Web Services, AWS Glue Developer Guide, docs.aws.amazon.com, 2023. https://docs.aws.amazon.com/glue/

[8] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A Distributed Messaging System for Log Processing," Proc. NetDB Workshop, 2011. https://pages.cs.wisc.edu/~akella/CS744/F17/838-CloudPapers/Kafka.pdf

[9] Snowflake Inc., Snowflake Documentation: Snowpark Developer Guide, docs.snowflake.com, 2023. https://docs.snowflake.com/en/user-guide/data-load-snowpipe-intro

[10] M. Zaharia et al., "Apache Spark: A Unified Engine for Big Data Processing," Communications of the ACM, vol. 59, no. 11, pp. 56–65, 2016. https://dl.acm.org/doi/10.1145/2934664

[11] Snowflake Inc., Snowflake Documentation: Streams and Tasks, docs.snowflake.com, 2023. https://docs.snowflake.com/en/user-guide/data-pipelines-intro

[12] Amazon Web Services, AWS Step Functions Developer Guide, docs.aws.amazon.com, 2023. https://docs.aws.amazon.com/step-functions/

[13] M. Kleppmann, Designing Data-Intensive Applications. Sebastopol, CA: O'Reilly Media, 2017.

[14] Snowflake Inc., Snowflake Documentation: Access Control and Security, docs.snowflake.com, 2023. https://docs.snowflake.com/en/user-guide/security-access-control-overview
[15] Amazon Web Services, AWS Identity and Access Management User Guide, docs.aws.amazon.com, 2023. https://docs.aws.amazon.com/iam/
[16] Amazon Web Services, AWS CloudTrail User Guide, docs.aws.amazon.com, 2023. https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-user-guide.html
[17] National Institute of Standards and Technology, SP 800-207: Zero Trust Architecture, 2020. https://csrc.nist.gov/pubs/sp/800/207/final
[18] U.S. Securities and Exchange Commission, Regulation S-P: Privacy of Consumer Financial Information, 2023. https://www.sec.gov/rules-regulations/2024/06/s7-05-23
[19] M. Zaharia et al., "Accelerating the Machine Learning Lifecycle with MLflow," IEEE Data Engineering Bulletin, vol. 41, no. 4, pp. 39–45, 2018. https://people.eecs.berkeley.edu/~matei/papers/2018/ieee_mlflow.pdf
[20] D. Sculley et al., "Hidden Technical Debt in Machine Learning Systems," Proc. NeurIPS, pp. 2503–2511, 2015. https://proceedings.neurips.cc/paper_files/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf
[21] J. Reis and M. Housley, Fundamentals of Data Engineering. Sebastopol, CA: O'Reilly Media, 2022.
[22] M. Armbrust et al., "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics," Proc. CIDR, 2021. https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf