International Journal on Science and Technology (IJSAT)

E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

Event-Driven Microservices and Streaming Data Platforms: Applying Kafka + Confluent in Real-Time Microservice Mesh on AWS & Azure

How to architect streaming microservices that use Kafka as the backbone, with Confluent connectors, across clouds, enabling microservice communication and real-time event propagation

Girish Rameshbabu

Customer Success Technical Architect
girish.prasad.23@gmail.com

Abstract:
As enterprise ecosystems transition from monolithic architectures to distributed microservices, traditional synchronous communication patterns (REST/gRPC) increasingly lead to temporal coupling, operational fragility, and data silos. These challenges are exacerbated in multi-cloud strategies where data must reside and synchronize across **Amazon Web Services (AWS)** and **Microsoft Azure**. This paper proposes a solution through the implementation of a decentralized **"Event Mesh"** architecture using **Apache Kafka** and the **Confluent Ecosystem** as the streaming backbone.
We investigate the transition from request-response orchestration to asynchronous event choreography, addressing the **"Dual-Write"** problem via **Change Data Capture (CDC)** and ensuring data governance through the **Confluent Schema Registry**. A primary focus is placed on the technical mechanics of **Confluent Cluster Linking** to bridge the cloud divide without the overhead of traditional mirroring tools. Furthermore, we address critical implementation challenges, including **Exactly-Once Semantics (EOS)** for cross-cloud transactions, security via **mTLS**, and performance optimization across high-latency inter-cloud links. The resulting architecture provides a persistent, replayable, and scalable communication mesh that ensures real-time data consistency and fault tolerance across a heterogeneous cloud landscape.

I. INTRODUCTION

The modern enterprise landscape is undergoing a fundamental shift from monolithic architectures to distributed microservices. While this transition promises scalability and independent deployability, it introduces significant complexity in how data is shared and synchronized across boundaries. Traditionally, microservices have relied on synchronous, request-response communication patterns (e.g., REST or gRPC). However, as systems grow to span multiple cloud providers like **Amazon Web Services (AWS)** and **Microsoft Azure**, these patterns reveal critical architectural bottlenecks.

The Limitation of Temporal Coupling

In a synchronous microservice environment, the "upstream" service is inextricably linked to the availability and performance of the "downstream" service. This **temporal coupling** means that if Service B is down or experiencing high latency, Service A—and by extension, the end user—suffers immediately. This often results in what is known as a **"Distributed Monolith,"** where services are physically separated but logically entangled, negating the very benefits of microservice isolation [1].

The "Dual-Write" Problem

A recurring challenge in maintaining data consistency is the **dual-write problem**. This occurs when a microservice must update its local database and simultaneously notify other services of the change (e.g., via a message broker). Without an atomic transaction spanning both the database and the message system—which is notoriously difficult to scale in distributed environments—the system inevitably falls into an inconsistent state. If the database update succeeds but the message dispatch fails, the rest of the ecosystem remains unaware of the change, leading to data silos and "dark data" [3].

The Shift to the Unified Log

To address these challenges, this paper proposes an architectural shift where the **immutable log**—specifically **Apache Kafka**—acts as the "central nervous system" of the enterprise. By treating every state change as an event recorded in a persistent, replayable log, we move away from point-to-point orchestration toward a decoupled **Event Mesh**. In this paradigm, the log becomes the authoritative source of truth, enabling real-time event propagation and solving the dual-write dilemma through patterns like Change Data Capture (CDC) and the Outbox pattern [2].

This paper explores the implementation of such a mesh across a multi-cloud environment, leveraging the **Confluent Ecosystem** to bridge the gap between AWS and Azure, ensuring that data remains fluid, consistent, and available regardless of where the service resides.

II. BACKGROUND AND RELATED WORK

To fully appreciate the evolution of the Multi-Cloud Event Mesh, we must first distinguish between traditional networking frameworks and the emerging paradigm of streaming data platforms. This section contextualizes a fundamental shift: moving away from mere network-level traffic management toward a model defined by data-level event propagation.

Service Mesh: The Synchronous Baseline

In the early days of microservices, the **Service Mesh** (e.g., Istio, Linkerd) emerged as the gold standard for managing service-to-service communication. It operates primarily through a "sidecar" proxy pattern, where a dedicated infrastructure layer handles the heavy lifting of the network. As noted in recent surveys of cloud-native architectures, the service mesh utilizes a distinct Control Plane and Data Plane to orchestrate request-routing, load balancing, and mutual TLS (mTLS) [5].

However, the service mesh is fundamentally anchored to a request-response cycle. This creates an inherent limitation: it requires both the producer (the service sending the request) and the consumer (the service receiving it) to be available and responsive at the exact same moment. This maintains the rigid temporal coupling described in Section I, where a single point of failure in the chain can trigger a cascading system-wide bottleneck.

Event Mesh: Asynchronous Data Propagation

In contrast, an **Event Mesh** represents a more fluid, configurable infrastructure layer designed to distribute events among decoupled microservices. While a service mesh focuses on the path or the "pipe" between two specific points, the event mesh—powered by robust platforms like Apache Kafka—prioritizes the persistence and distribution of the data itself.

In an event mesh environment, the producer is blissfully unaware of who the consumers are, or if they are even online at the time of transmission. The infrastructure acts as a persistent buffer, ensuring that events are routed across geographically dispersed environments, such as AWS and Azure, based on subscription interest rather than explicit network addresses [6]. This shift transforms the network from a series of fragile phone calls into a persistent, shared broadcast system.

Apache Kafka as the Streaming Backbone

If the Event Mesh is the architecture, **Apache Kafka** is the engine that drives it. Kafka is not a traditional message broker; it is a distributed, partitioned, and replicated commit log. To understand its efficacy as a multi-cloud backbone, we must examine its foundational primitives:

- **Partitions:** These serve as the primary unit of parallelism. By breaking a topic into partitions, Kafka allows data to be distributed across multiple nodes, enabling the system to scale horizontally as throughput demands increase.
- **Replication:** This is the insurance policy of the mesh. By mirroring data across different brokers within a cluster, Kafka ensures high availability even in the event of hardware failure [4].
- **Consumer Groups:** This mechanism allows a pool of services to coordinate their efforts. By dividing the work of consuming a stream, consumer groups provide both massive scale and fault tolerance, ensuring that no single event is processed twice by the same functional unit [4].

Multi-Cloud Strategies and Data Gravity

A significant hurdle in modern architecture is the concept of **"Data Gravity."** This theory suggests that as data sets grow in size, they act like a mass, attracting applications and services toward them. In a multi-cloud strategy involving both AWS and Azure, data gravity typically results in fragmented silos—where data in "Cloud A" is too heavy or expensive to move to "Cloud B" in real-time.

Traditional attempts to solve this, such as **MirrorMaker 2**, often struggled with operational overhead, complex configuration, and high latency. This paper builds upon the more modern **"Cluster Linking"** methodology. Unlike its predecessors, Cluster Linking creates a seamless fabric that effectively "defies" data gravity. It allows events to flow between different cloud providers with byte-for-byte accuracy, making data in an Azure region appear as if it were residing on the same local network as an AWS-based service.

III. PROPOSED ARCHITECTURE: THE MULTI-CLOUD EVENT MESH

The proposed architecture moves beyond the traditional limitations of site-to-site replication by establishing a unified **Multi-Cloud Event Mesh**. This design integrates geographically dispersed clusters in AWS and Azure into a single, logical data fabric. By abstracting the underlying infrastructure, we ensure that microservices can produce and consume events seamlessly, regardless of whether they are hosted on Amazon's or Microsoft's cloud infrastructure.

The Backbone: Bridge Topologies

A critical design decision in any multi-cloud architecture is the selection of the cluster topology. While "stretched" clusters—where a single Kafka cluster spans multiple cloud providers—offer the allure of high synchronous consistency, they are frequently hindered by the physical realities of high inter-cloud latency and unpredictable egress costs.

To mitigate these risks, this paper proposes a **Hub-and-Spoke or Peer-to-Peer Bridge Topology**. By deploying independent, highly-localized Kafka clusters in AWS (e.g., US-East-1) and Azure (e.g., East US), we isolate the performance of each environment. This ensures that a localized cloud outage or network spike does not bring down the entire global communication fabric.

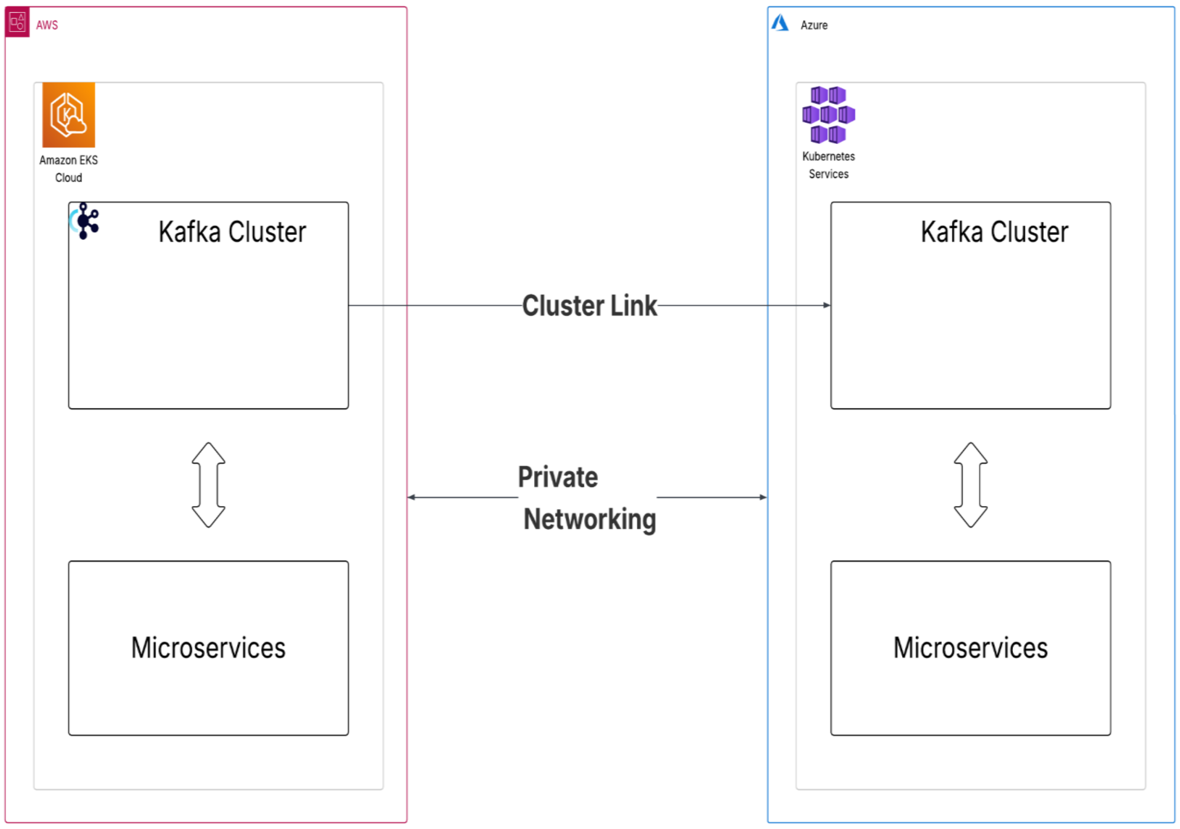

Figure 1 Multi-cloud Event Mesh architecture utilizing Confluent Cluster Linking for real-time synchronization between AWS and Azure regions.

Confluent Cluster Linking and Global Topics

The core technical innovation within this architecture is the transition from legacy MirrorMaker 2 configurations to **Confluent Cluster Linking**. Traditional mirroring is an external process; it requires running an additional cluster of Connect workers to move data between sites. This adds significant management overhead, introduces "heartbeat" latency, and increases the complexity of failure scenarios. Cluster Linking, however, leverages the Kafka protocol itself. It enables the destination cluster to initiate a direct connection to the source cluster. This mechanism creates what we define as **"Global Topics"**—read-only mirrors that are kept in sync with byte-for-byte accuracy. For a microservice residing in Azure, consuming from a linked topic originating in AWS is entirely transparent. Because offsets are preserved and metadata is synchronized, the architecture supports seamless failover and "follow-the-sun" data processing, where workloads can shift across clouds as business needs dictate.

Microservice Integration: Smart Endpoints and Schema Governance

To maintain the architectural principle of **"Smart Endpoints, Dumb Pipes,"** our design utilizes ksqlDB and Kafka Streams for on-the-fly data transformation. In this model, the Kafka mesh focuses on durable delivery, while the business logic remains at the edges. However, in a multi-cloud environment, the risk of data corruption due to incompatible message formats is exponentially higher due to the lack of centralized oversight.

To counter this, the **Confluent Schema Registry** is deployed as a critical governance layer. By enforcing strict schema validation—typically using Avro or Protobuf—at the producer level, the architecture provides a safety net. It ensures that a microservice in AWS cannot publish a breaking change that would inadvertently crash a downstream consumer in Azure. This "contract-first" approach allows independent engineering teams to evolve their services at their own pace without the need for constant, centralized coordination [9]. This governance transforms the mesh from a simple transport layer into a reliable, enterprise-grade data contract.

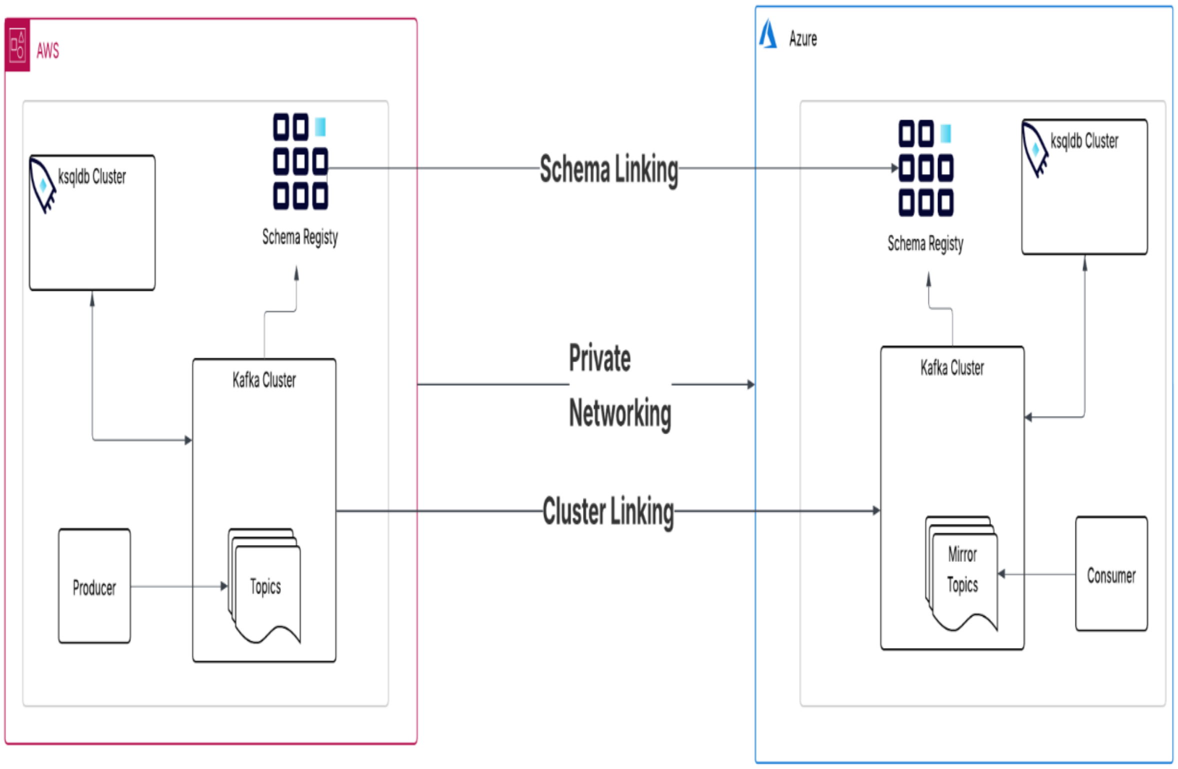

Figure 2 Centralized Schema Registry workflow for cross-cloud microservice governance, illustrating producer-side validation and consumer-side deserialization.

Latency and Throughput Considerations

Communication between AWS and Azure involves traversing the public internet or dedicated interconnects (like AWS Direct Connect or Azure ExpressRoute). Research indicates that while throughput can be sustained through tuned batching, the "speed of light" latency between providers (typically 20ms to 80ms) necessitates an asynchronous design where services are optimized for **eventual consistency** rather than immediate synchronization [8].

IV. IMPLEMENTATION STRATEGY

The transition from a conceptual mesh to a functional multi-cloud system requires a robust implementation strategy that bridges legacy data stores with modern, event-driven logic. This section details the practical application of Change Data Capture (CDC), event choreography, and the automated deployment pipelines necessary to sustain a high-availability environment.

Connecting the Data: Change Data Capture (CDC)

To effectively solve the "Dual-Write" problem identified in Section I, the architecture employs **Debezium**, an open-source distributed platform built for CDC. In a traditional setup, a microservice is burdened with the responsibility of updating its local database and explicitly sending a notification to Kafka. This is a recipe for inconsistency.
By using Debezium, we shift this responsibility away from the application code. Debezium monitors the low-level transaction logs of source databases, such as **Amazon RDS (PostgreSQL/MySQL)** or **Azure CosmosDB**.

When a database row changes, Debezium captures the delta and streams it into a Kafka topic as a structured event. This ensures that the database update and the event propagation are effectively atomic, as the event is derived directly from the committed database log [11]. Furthermore, **Sink Connectors** can then offload this data into cloud-native storage like **Amazon S3** or **Azure Data Lake Storage (ADLS)** for long-term analytics, completing the data lifecycle across the cloud divide.

The "Microservice Mesh" Pattern: Choreography over Orchestration
A key hurdle in distributed systems is managing complex, multi-step business processes, such as an e-commerce checkout flow. This paper advocates for **Choreography** over centralized **Orchestration**.
In a traditional orchestrated model, a central "manager" service directs other services through synchronous calls—a method that creates a single point of failure and tight coupling. In our proposed **Choreographed Event Mesh**, services react independently to events.

For example, when an OrderPlaced event appears on the Kafka backbone, the InventoryService and PaymentService consume that event simultaneously. They perform their respective logic without ever communicating directly with one another. This follows the **Saga Pattern**, where a sequence of local transactions is triggered by events, ensuring eventual consistency across the multi-cloud boundary without the need for a central "brain" [10].

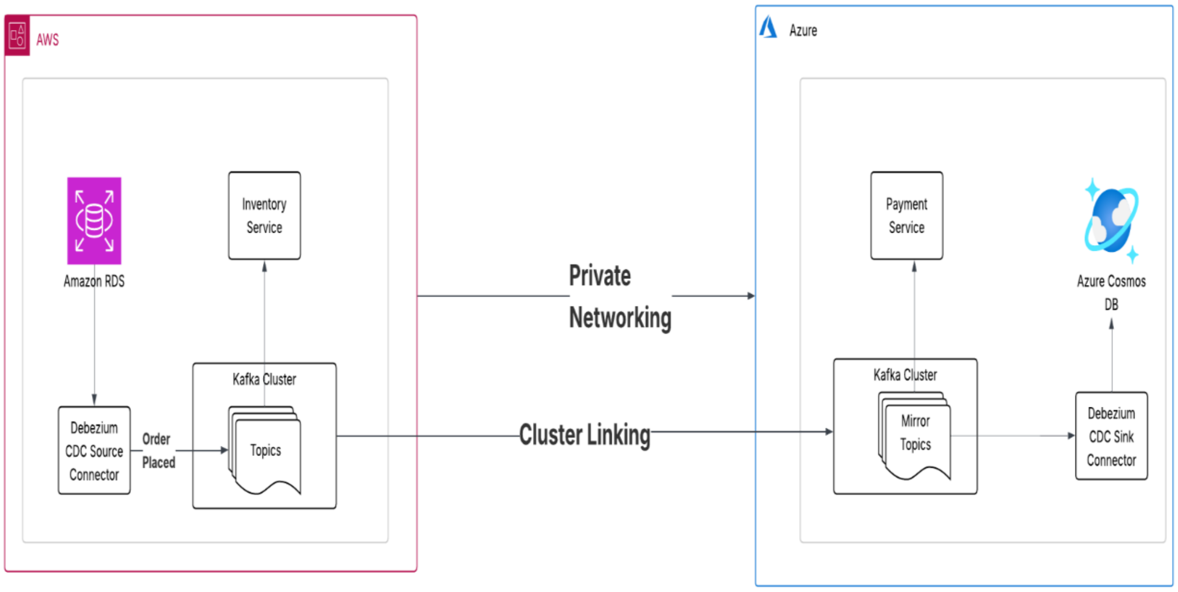

Figure 3: Event-driven choreography pattern: leveraging Debezium CDC for database log-scraping and asynchronous service reaction across the Kafka backbone.

"Smart Endpoints, Dumb Pipes" at Scale
Following the precedent set by industry leaders like Netflix, this architecture treats Kafka as a **"Dumb Pipe"** that focuses exclusively on high-throughput delivery and durability. The intelligence is pushed to the **"Smart Endpoints"**—the microservices themselves.
By utilizing the **Kafka Streams API**, services can perform complex joins, windowing, and aggregations locally on their own compute clusters, such as **AWS EKS** or **Azure AKS** [12]. This avoids the pitfalls of heavy middleware and allows the streaming platform to scale linearly, as the heavy lifting of data processing is distributed across the entire microservice landscape.

Infrastructure as Code (IaC)

To maintain environmental parity between AWS and Azure, the entire infrastructure stack—encompassing VPC/VNet peering, Kafka clusters, and specialized Connectors—is defined using **Terraform**.

This IaC approach allows for the repeatable deployment of "Cluster Linking" configurations. By treating our infrastructure as software, we ensure that security policies, networking rules, and broker configurations remain identical across both cloud providers. This rigor significantly reduces the risk of **"configuration drift,"** where subtle differences between cloud environments lead to unpredictable production errors.

Integration Note on Monitoring (ksqlDB Metrics)

While the proposed architecture leverages **ksqlDB** for real-time stream processing, practitioners must note the operational distinction between cloud-native and self-managed environments. In the Confluent Platform/AWS MSK environments discussed, ksqlDB performance metrics remain **JMX-exclusive**. Unlike Confluent Cloud's unified Metrics API, these time-series data points are emitted directly by the ksqlDB servers.

V. CRITICAL CHALLENGES AND SOLUTIONS

Operating a distributed event mesh across the heterogeneous landscapes of AWS and Azure introduces formidable engineering hurdles. These challenges typically cluster around three pillars: data integrity in the face of network instability, the physical realities of inter-cloud latency, and the imperative for unified security. This section analyzes these friction points and details the strategic mitigations employed to maintain a production-grade system.

Guaranteeing Data Integrity: Exactly-Once Semantics (EOS)

In a multi-cloud environment, network "flapping" and transient glitches are not just possibilities—they are inevitable. Without rigorous protocols, these interruptions typically force a choice between two undesirable outcomes: duplicate messages (**at-least-once delivery**) or silent data loss (**at-most-once delivery**). For high-stakes applications like financial ledgering or inventory management, neither is acceptable.

To achieve **Exactly-Once Semantics (EOS)**, the architecture leverages Kafka's transactional API. This is achieved through a two-pronged mechanism:

- **Idempotent Producers:** Kafka assigns unique sequence numbers to every message. If a producer retries a send due to a perceived network timeout, the broker recognizes the duplicate sequence and discards it, preventing data duplication.
- **Atomic Transactions:** This allows a producer to send a batch of messages to multiple partitions such that they are either all visible to consumers or none are.

By synthesizing these features, the system maintains a consistent state even during a partial cloud outage or a broker failover, ensuring that a critical event like OrderProcessed is never executed twice [13].

Managing Latency and Bandwidth across Clouds

The physical distance between an AWS region (e.g., US-East-1 in Virginia) and an Azure region (e.g., West US in California) introduces inherent speed-of-light latency. To maintain high throughput despite a high **Round-Trip Time (RTT)**, our implementation strategy moves away from "chatty" communication in favor of high-density data transmission:

- **Compression:** We utilize **Snappy** or **LZ4** compression algorithms. By shrinking the payload size before it hits the wire, we significantly reduce the time spent in transit across the inter-cloud interconnect.

- **Strategic Batching:** By tuning the batch.size and linger.ms parameters on the producers, we encourage the system to group more messages into a single TCP packet. While this introduces a few milliseconds of artificial latency at the local level, it yields a massive increase in overall network efficiency and throughput by reducing the total number of packets sent [15].

Cross-Cloud Networking and Security

Securing "Data in Motion" as it traverses the divide between cloud providers requires a multi-layered defense strategy. Relying on the public internet is insufficient for enterprise-grade compliance and security.

- **Network Isolation:** Instead of exposing Kafka brokers to the public internet, the architecture creates a "private bridge." This utilizes **AWS PrivateLink** and **Azure Private Link**, interconnected via a Site-to-Site VPN or a dedicated cross-cloud interconnect (like Equinix or Megaport). This ensures that traffic never leaves the private backbone of the respective providers [14].
- **Encryption and Identity:** Every connection within the mesh is encrypted using **mutual TLS (mTLS)**. This ensures that not only is the data unreadable to eavesdroppers, but both the client and the broker must present valid, trusted certificates to communicate.
- **Access Control:** To prevent a "security silo," **Role-Based Access Control (RBAC)** is synchronized across clouds. By integrating with a central **Identity Provider (IdP)** using OIDC or SAML, we ensure that a microservice's permissions remain consistent and auditable, regardless of which cloud provider is hosting the compute workload.

VI. CONCLUSION AND FUTURE WORK

Summary of Contributions

This paper has presented a robust architectural framework for a **Multi-Cloud Event Mesh**, addressing the fundamental challenges of temporal coupling and data silos in modern microservice ecosystems. By positioning **Apache Kafka and Confluent** as the "central nervous system" across AWS and Azure, we have demonstrated that it is possible to achieve real-time data consistency without the overhead of synchronous request-response patterns.

Key takeaways from this architecture include:

- **Decoupling via Event Mesh:** The shift from service-to-service orchestration to event-driven choreography allows microservices to scale and fail independently.
- **Multi-Cloud Fluidity:** Through **Cluster Linking**, data produced in one cloud provider is immediately available in another with byte-for-byte accuracy, effectively overcoming the "Data Gravity" problem.
- **Reliability:** The implementation of **Exactly-Once Semantics (EOS)** and **Schema Governance** ensures that the mesh remains resilient to network partitions and human error.

The proposed "Smart Endpoints, Dumb Pipes" approach ensures that while the transport layer remains simple and scalable, the business logic remains agile and distributed.

Future Work

While the current architecture provides a solid foundation for cross-cloud streaming, several emerging areas warrant further research:

- **Serverless Kafka Integration:** Investigating the performance and cost-benefit analysis of fully serverless Kafka offerings (like Confluent Cloud or AWS MSK Serverless) versus managed clusters in a multi-cloud mesh.
- **Tiered Storage Optimization:** Exploring the use of **Tiered Storage** (e.g., offloading historical Kafka data to S3 or Azure Blob Storage) to enable cost-effective, long-term event replayability without scaling the broker storage.

- **AI-Driven Stream Processing:** Integrating real-time machine learning models directly into the stream via **Flink SQL** or **KSQLDB** to provide predictive insights as events move between AWS and Azure.
- **Edge-to-Cloud Eventing:** Extending the mesh to include Edge computing nodes (e.g., AWS Wavelength), creating a continuum from the edge to the multi-cloud core.

REFERENCES:

[1] P. Di Francesco et al., "Research on Monolithic to Microservices Refactoring: A Systematic Mapping Study," 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Gothenburg, Sweden, 2018
[2] J. Kreps, "The Log," *LinkedIn Engineering Blog*, 2013.
[3] W. Waldron, "The Dual-Write Problem," Confluent Blog, 2020.
[4] N. Narkhede et al., *Kafka: The Definitive Guide*, 2021.
[5] W. Li et al., "A Survey of Service Mesh Architectures," *ICCSNT*, 2019.
[6] "Comparing and Contrasting Service Mesh and Event Mesh," *Solace*, 2021.
[7] "Cluster Linking for Multi-Cloud," *Confluent Documentation*, 2023.
[8] N. Kandregula, "Evaluating performance and scalability of multi-cloud environments: Key metrics and optimization strategies," *ResearchGate*, 2022.
[9] Schema Linking on Confluent Platform, *Confluent Documentation*
[10] B. Stopford, "Microservices Choreography vs. Orchestration," *Camunda Blog*, 2023.
[11] "Debezium Project Documentation," 2023.
[12] "Evolution of the Netflix Data Pipeline," *Netflix Technology Blog*, 2016.
[13] “Exactly-Once Semantics Are Possible: Here’s How Kafka Does It” *Confluent blog, 2017*
[14] "Designing private network connectivity between AWS and Microsoft Azure”, *AWS Blog 2024.*
[15] S. Deva "Optimizing Apache Kafka for efficient data ingestion”, ResearchGate, 2025