From On-Prem to Cloud-Native: Multi-Tenancy in Confluent Cloud

JUL 29, 2021 READ TIME: 18 MIN

Multi-tenancy brings cost-efficiency to infrastructure, and when done correctly, creates an economy of scale. Done incorrectly and you degrade the user experience and create maintenance nightmares for operators. This is true for Apache Kafka®, but more so for a cloud-native data storage system.

While on-prem multi-tenant solutions might get away with over-provisioning and manual capacity planning, cloud-native systems must take advantage of cloud elasticity to make them cost-effective, while at the same time supporting more stringent uptime and performance Service Level Objectives (SLOs).

In this blog post, we will discuss requirements for cloud-native multi-tenancy—within the scope of Apache Kafka, and beyond. We'll explain how cloud-native data systems are different from legacy databases. We'll then discuss in-depth how we used existing multi-tenancy capabilities in Apache Kafka to create cloud-native multi-tenancy in Confluent Cloud.

To learn more about how Kafka's cloud-native capabilities were enhanced for Confluent Cloud, check out the other posts in the Design Considerations for Cloud-Native Data Systems Series:

- Introduction: Design Considerations for Cloud-Native Data Systems
- Part 1: Making Apache Kafka Serverless: Lessons From Confluent Cloud
- Part 2: Speed, Scale, Storage: Our Journey from Apache Kafka to Performance in Confluent Cloud

WRITTEN BY

Anna Povzner Anastasia Vela

- Part 4: Protecting Data Integrity in Confluent Cloud: Over 8 Trillion Messages Per Day

Software Engineer

Background: Multi-tenancy in Apache Kafka

Apache Kafka is commonly used as a central nervous system, connecting a wide range of different data stores and applications from various teams and lines of business in real-time. To make it safe to colocate different applications and data stores in a single cluster, Kafka provides a set of features for setting up a functional multi-tenant environment. These include:

- Isolating data with authentication, authorization, and encryption
- Isolating user spaces (namespaces) by convention
- Isolating performance with quotas

These three types of isolation—data, namespaces, and performance—are key requirements for any multi-tenant system. The rest of this section discusses how Apache Kafka is designed to support all three of these requirements, and how you can use them to set up a multi-tenant Kafka cluster.

Security and data isolation

Securing connections between Kafka clients and brokers via authentication lets you control who can access the shared cluster. Kafka supports various authentication methods: SSL and SASL (PLAIN, SCRAM, Kerberos, and OAUTHBEARER).

A unit of data isolation is a *user principal*—an authenticated user or a grouping of unauthenticated users chosen by a broker using a configurable PrincipalBuilder. You can control which user principals have access to certain topics and consumer groups, and thus which user principals have access to data stored within a shared cluster. Managing user permissions is performed predominantly through the setting of access control lists (ACLs).

User space (namespace) isolation

To avoid conflicts on topic names created by different users, you can enforce namespaces by convention. The easiest way to achieve this is to set prefix ACLs (see KIP-290) to enforce a common prefix for all topic names created by a user. For example, an application identified by user principal A may only be permitted to create topics whose

name starts with “pageviews”. For more complex patterns and rules, you can define a custom CreateTopicPolicy (KIP-108).

Performance isolation

To avoid one rogue user or client negatively impacting performance for other users or clients, multi-tenant clusters must be configured with client quotas. Quotas can be set on <user-principal, client-id>, user-principal, or client-id. This architecture is flexible, as performance isolation is not tied to namespace or data isolation. For example, all clients from the same application can have permission to access all of the data created by the application, but subsets of clients within the application can set their own quotas.

It is important to configure each type of client quota in Apache Kafka, because the different quotas protect different types of resources:

- **Producer and consumer bandwidth quotas** set the limit on the permitted amount of traffic from a user or client to an individual broker, in bytes per second. When the broker detects that a user or client has exceeded its producer bandwidth quota, it starts to throttle producer requests. Similarly, brokers throttle fetch requests from clients when the user/client exceeds its consumer bandwidth quota.
- **Request quotas** limit a user’s impact on broker CPU usage by limiting the clock time that the broker spends on request handler (I/O) threads and network threads, processing requests and authenticating connections from the user or client (KIP-124). Setting request quotas protects not only request processing, but also the bandwidth of other tenants, because excessive CPU usage on a broker degrades the effective broker bandwidth.
- **Quotas on topic operations**—such as create, delete, and alter—prevent the Kafka cluster from being overwhelmed by highly concurrent topic operations by one user or client (KIP-599).

Apache Kafka supports additional quota types to ensure the stability of the cluster, the most notable being:

- **Broker/per-listener limits on connection creation rate** limit the amount of CPU spent by the broker/listener on creating new connections, so that there is enough CPU capacity for request processing (KIP-612).
- **Per-IP limits on connection creation rate** ensure that no clients take over most of the broker/per-listener quota (KIP-612).
- **Limit on the number of active connections** to each broker (KIP-402).

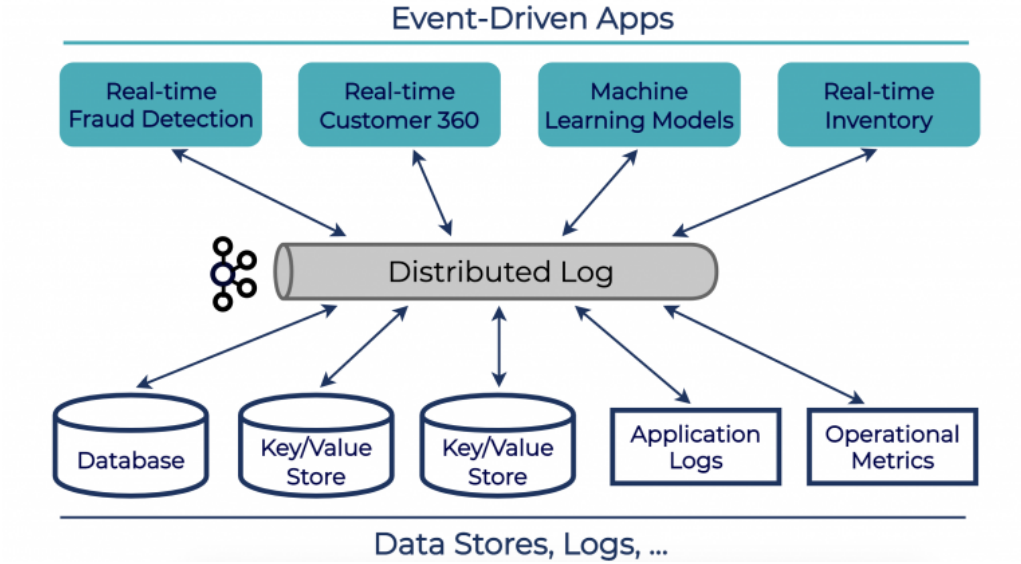
Capacity planning and monitoring

Client quotas are limits set on a per-broker basis. However, setting quotas does not guarantee that there is enough cluster capacity for every application to get the desired bandwidth. As with a dedicated cluster, setting up a multi-tenant environment requires capacity planning that accounts for all of the applications that will be deployed on the cluster, with some headroom for growth.

Application adoption could grow beyond the planned headroom, in which case clients could experience negative performance from throttling. Therefore, it is important to monitor whether your clients are being throttled and to consider increasing quotas and expanding the cluster to support increased usage.

Multi-tenancy the cloud-native way

Apache Kafka's multi-tenant capabilities, as described above, were designed and traditionally used within the context of an on-prem data center. A single company ran a multi-tenant Kafka cluster with all of its workloads, in order to avoid siloing data streams from different departments—the goal was to make the data available across the company, as shown in the example diagram below. Apache Kafka's multi-tenant features made this possible and allowed companies to run Kafka efficiently at company-scale.



Cloud-native systems have an additional set of requirements. These requirements call for a more comprehensive multi-tenant solution than what could be acceptable for on-premises:

- **Abstraction** requires not exposing low-level resources. For example, while Apache Kafka lets you configure quotas per broker, this adds too much operational complexity for customers of a cloud service.
- **A pay-as-you-go model** requires the system to use only the minimum required resources until more are needed. For example, any tenant can scale from 0 to 100 MBps (or your advertised bandwidth limit) and back to 0.
- **Uptime and performance SLOs** are much more stringent for a cloud service, which means that a multi-tenancy solution must have a minimal operational overhead, while supporting unknown and unpredictable workloads and protecting the cluster from DDoS attacks.
- **Cost-efficiency**: Solutions for all of the above need to be cost-efficient.

Cloud environments make it possible to get more networking bandwidth, computing power, and storage almost instantaneously. Users of cloud services expect elasticity, which means that cloud-native multi-tenant solutions must be able to keep up with this expectation and, at the same time, provide more stringent SLAs/SLOs for unpredictable workloads.

The pay-as-you-go model naturally leads to more workload unpredictability, as each tenant can scale their workload up from 0 to 100 MBps and back down at any time. Supporting a potentially unlimited number of tenants with unknown workloads makes any manual capacity planning or scaling impractical.

Thus, a well-architected cloud-native multi-tenant solution should take advantage of the cloud's infrastructure elasticity, automate capacity planning, and auto-scale to meet the above requirements.

Next, we will discuss in-depth how we leveraged existing multi-tenancy capabilities in Apache Kafka to create cloud-native multi-tenancy in Confluent Cloud.

Tenants in Confluent Cloud

When a customer creates a Basic or Standard cluster in Confluent Cloud, the control plane identifies an appropriate multi-tenant physical Kafka cluster based on the cloud provider and region specified by the customer. The metadata associated with the newly created logical cluster is transmitted to the same physical Kafka cluster. This metadata includes maximum ingress and egress and other limits per cluster advertised in our public documentation.

A tenant in Confluent Cloud is a logical Kafka cluster that represents a Confluent Cloud cluster. Each tenant is given an identifier, a Logical Cluster ID—a string like "lkc-abc123".

Customers of a cloud-native service expect to experience their cluster as their own independent cluster, with predictable performance and high availability, regardless of the workloads of other tenants sharing the same physical cluster. This requires supporting the three types of isolation that every multi-tenant system needs to provide: data isolation (security), metadata isolation (user spaces), and performance isolation (quotas). All three types of isolation are tied to the same tenant entity: a logical Kafka cluster.

In Confluent Cloud, we added support for the custom tenant entity that represents a unit of data, metadata, and performance isolation for a logical Kafka cluster by:

- **Extracting tenant ID on the broker during authentication** from the authenticated user. When the broker authenticates a client connection, it translates the authenticated user to a tenant ID from tenant metadata. This information is stored as part of the client session state, and from this point on, every client request will be associated with the tenant identifier.
- **Request/Response interceptor** that attaches a tenant ID to the request context, so that each request can only access resources owned by the tenant.
- **Customizing a quota entity** to represent all authenticated users of the same logical Kafka cluster (tenant) by implementing our own ClientQuotaCallback interface (KIP-257). Implementing a custom ClientQuotaCallback interface allowed us to use existing quota mechanisms from Apache Kafka to enforce limits on bandwidth, request processing, and partition operations per logical cluster on the broker, instead of per user or client ID.

The diagram below shows two logical clusters, lkc-bee71e and lkc-c0ffee, with clients connected to a Kafka broker in Confluent Cloud. From the client's perspective, resources such as topics are named just as in any other Kafka cluster. The interceptor attaches a tenant ID to the request, and from this point on, the broker ensures that each request can only access resources owned by the tenant. This way, two tenants can have a topic named users, but internally these will be represented as two different topics—each one associated with a different tenant.

Performance Isolation

We built upon the existing quotas framework in Apache Kafka by customizing quota allocations on the broker to represent all authenticated users from the logical Kafka clusters—tenants in Confluent Cloud. To achieve this, we implemented our own ClientQuotaCallback interface (KIP-257), which enabled brokers in Confluent Cloud to enforce produce and consume bandwidth quotas, request quotas, and quotas on topic operations per logical Kafka cluster.

However, requiring users of a cloud service to configure quotas per broker is too much detail. First, the underlying cluster, being cloud-native, will auto-scale. When it does, the number of brokers will change dynamically, and the per-broker quota will need to change to match the total quota given to a tenant. Second, the backing multi-tenant Kafka cluster may scale to a large number of brokers, and a tenant may end up on a subset of brokers. Third, there could be reasons for intentionally constraining each tenant to a subset of brokers, such as for fault isolation and for supporting uptime and performance SLOs. Thus, abstracting brokers is beneficial, both for simplifying operations and for allowing flexibility in how cluster resources are allocated to tenants to support SLOs.

We abstracted the brokers by adding the support for tenant-wide producer and consumer bandwidth quota configurations, which correspond to per-logical-cluster ingress and egress limits advertised in our documentation. Tenant-wide bandwidth quotas are enforced by each broker calculating and enforcing its own slice of tenant quota. We started with a simple approach of distributing tenant quota equally among all brokers that host at least one tenant partition leader, i.e., the brokers that receive produce/consume traffic. Each broker calculates its own slice of tenant quota based on tenant-wide quota configuration and cluster metadata, and updates it if needed when partition leadership changes occur.

Each broker serving requests from a tenant needs to have some CPU capacity allocated for processing tenant requests. Insufficient request processing capacity may lead to high latency and degraded bandwidth for a tenant. To ensure that no tenant takes over the broker CPU, we configure a request quota for each tenant on each broker, limiting each tenant to at most a portion of total broker processing capacity.

The following diagram illustrates bandwidth and request quota allocation on a Kafka cluster in Confluent Cloud, and highlights two logical clusters, lkc-bee71e and lkc-c0ffee, as an example. All brokers shown in the diagram host partition leaders of both tenants, and therefore are each assigned all three quotas. Brokers enforce each quota to ensure that tenants operate within their resource limits.

Capacity planning on the fly

We hold capacity needed for the current cluster load with headroom for short-term spikes in demand, and take advantage of cloud infrastructure elasticity to scale the cluster capacity when demand increases. The challenge here is to make sure that a sudden increase in demand, exceeding the headroom, does not overload the cluster before it is expanded. When demand reaches the current cluster capacity, a highly-utilized cluster

may cause high client latencies and degraded bandwidth. A demand exceeding the cluster capacity may cause client timeouts and even cluster unavailability.

Brokers in Apache Kafka are not aware of their current load, and do not have any mechanisms for automatically protecting themselves from overload. One workaround is for an operator to set bandwidth and request quotas on brokers, such that the sum of the quotas does not exceed available broker bandwidth and processing capacity. This ensures that even if each tenant spikes to their quota, there is still enough broker capacity available to them. This may work for on-prem, where it is common to over-provision clusters and proactively expand cluster capacity before deploying new applications. But this approach does not work with the pay-as-you-go model, because we are not holding the full capacity for which tenants are eligible—only the capacity that is currently needed.

We solved this in Confluent Cloud by introducing broker-wide limits (quotas) on broker resources, in addition to per-tenant quotas:

- Broker-wide producer and consumer bandwidth quotas
- Broker-wide request quota
- Broker-wide limit on connection attempt rate

When aggregate usage from all tenants exceeds a relevant broker-wide quota, the broker starts to throttle requests or connections. This throttling is temporary, until the cluster is either automatically rebalanced (if the high broker usage is caused by a hotspot) or automatically expanded to match the increase in demand.

The broker-wide producer/consumer bandwidth quota protects brokers from bandwidth overload. In our experience, the most likely situation in which this occurs is when producers are configured with acks != all, where the producer does not have to wait for full acknowledgement from all followers before pushing more bandwidth to the Kafka cluster. This may leave insufficient bandwidth capacity for replication, resulting in under-replicated partitions. The broker-wide producer/consumer bandwidth quota limits the aggregate producer/consumer bandwidth from all tenants on the broker, ensuring that enough bandwidth capacity is reserved for replication.

We set the broker-wide producer and consumer quotas to the effective broker producer and consumer bandwidth capacity. With a replication factor of 3 and a well-balanced cluster, the effective broker capacity for the producer bandwidth is ⅓ of the maximum sustained broker write capacity. The effective consumer bandwidth capacity per broker is the maximum broker read capacity excluding the bandwidth used for replication. Sustained write capacity is determined by disk bandwidth, while reads from real-time consumers are usually served from page cache. As a result, broker read capacity is

generally larger than broker write capacity. We determined maximum broker write and read capacity from hardware specs and confirmed with benchmark experiments.

The broker-wide request quota protects brokers from request overload by limiting the time that the broker spends on processing requests from tenants. Specifically, it does two things. First, it ensures that the broker keeps enough processing capacity for replication and other types of requests that are exempt from throttling. (Some requests in Kafka are exempt from throttling. Notably, follower fetch requests are exempt from throttling to ensure that followers do not fall out of in-sync replicas (ISR).) Second, the broker-wide request quota ensures that broker CPU and request queues do not get overloaded.

The broker-wide request quota limits the total percentage of time that tenants can spend on the request handler (I/O) threads and network threads within each quota window, the same as client request quotas in Apache Kafka (KIP-124) and tenant request quotas in Confluent Cloud. The quota is out of the total capacity of the network and I/O thread pools. The diagram below shows the default configuration in Apache Kafka, with three network threads (one listener) and 8 request handler threads.

Because the number of network and I/O threads in Kafka are typically configured based on the number of cores available on the broker host, the request quota helps to limit CPU usage by tenants. Tenants' most CPU-extensive operations happen on network and I/O threads: request processing, encryption, and authentication. However, blocking I/O can also happen on those threads, which would also account for request quota since the quota limits the clock time on the thread, not pure CPU usage.

Statically configuring broker-wide request quotas requires a trade-off between uptime / performance SLOs and cost-efficiency, i.e., choosing a larger headroom to account for a wider range of workloads vs. a smaller headroom for cost-efficiency. Auto-adjusting broker-wide request quotas, based on the current usage by throttling-exempt requests, is also not enough, because some of the total threads capacity may be unusable. For example, if enough threads block on I/O, a broker may have available CPU capacity that cannot be utilized due to a lack of threads to do the processing. If there are more threads than CPU cores, threads capacity does not always translate into CPU power available to make use of them.

We addressed this problem by using request queue size as an indicator of high broker load. In the common case, the broker dynamically calculates its broker-wide request quota as maximum network and I/O thread pool capacity, minus the current usage by

requests exempt from throttling. When the broker detects that requests start crossing the configured threshold for request queue size, the broker reduces the broker-wide request quotas further until it detects smaller request queue sizes. When queues become less loaded because the client load decreases, the broker raises the broker-wide request quota back to the common case level.

This approach is based on the fact that when the broker cannot keep up with the incoming client requests, the request queue builds up. Even though request queues are configured with maximum capacity via the queued.max.requests broker configuration (the default is 500), which helps to limit the number of requests at the broker, we observed that letting the queues max out slows the broker down significantly. When the request queue maxes out, network threads get blocked until one of the request handler threads picks up a request from the queue. There will also be requests on client connections waiting to be picked up by network threads. As the broker becomes slower due to threads being blocked, the requests start timing out, causing clients to retry and further increasing the load on the broker. In the worst-case scenario, the majority of requests time out before the broker sends the responses back.

When the request queue is not allowed to max out, the requests get throttled instead, which helps to avoid client timeouts.

The network listener limit on connection attempts per second limits the time that the broker spends on processing connections from tenants, as brokers in Confluent Cloud are configured with separate listeners for inter-broker communication and communications with tenants.

Request quotas do not control how many connections the broker accepts, even though processing of accepted new connections is done by the network threads. The diagram below shows how an increase in the connection attempt rate from clients may eventually lead to new connections using most of the broker-wide request quota, leaving no quota for request processing. Because the first request on a new connection is not limited by the request quota, the time the broker spends in processing new connections and requests from tenants may exceed the broker-wide request quota once most of the request quota is used by new connections.

Therefore, limiting the connection attempt rate is useful both for preventing connection storms and for limiting the rate of throttling-exempt requests on the broker, to minimize the chance of request quotas failing to protect the broker from request overload. To throttle connection attempts, the broker delays acceptance of a new connection by the amount of time that brings the attempt rate within the limit.

Auto-tuned tenant bandwidth and request quotas

So far, we've described how we configure brokers with broker-wide producer and consumer bandwidth quotas, and how brokers automatically calculate a broker-wide request quota. Brokers enforce broker-wide bandwidth and request quotas by auto-tuning corresponding tenant quotas, such that enforcing tuned tenant quotas also enforces the broker-wide quota.

Per-tenant, per-broker producer and consumer bandwidth quotas and the request quota represent the capacity that the tenant is eligible to use on the broker. Because the pay-as-you-go model encourages idle and low-utilization tenants, we can't predict which workloads will be added and how much capacity they will need. A spike in demand for bandwidth or request processing may exceed the available capacity while all tenants are operating below their quotas. If capacity is reached before the tenant quotas, tenants start competing with each other and degrading each other's experience in an unpredictable manner.

We address this by automatically tuning the tenants' quotas, in order to maintain a combined tenant usage below the broker-wide limit. Reduced tenant quotas due to insufficient capacity are always temporary, only in place while the cluster capacity is automatically expanding. We keep some excess capacity for workload spikes. When more workloads spike than accounted for in the headroom, we detect this and expand the cluster. While the cluster is expanding, tenant quotas' auto-tuning ensures that each tenant equally experiences insufficient cluster capacity.

On a configured time interval, the broker checks whether the combined tenant usage exceeds the broker-wide limit, and if so, the broker decreases the per-tenant dynamic quota of each active tenant. Since each cluster can have thousands of tenants, updating every tenant's quotas would be inefficient, so to make auto-tuning more efficient, we only account for the active tenants. Active tenants are tenants that have made a produce or consume request in the last few minutes. Once the tenant is no longer active, we increase their quota back to the original quota.

When demand exceeds the broker-wide quota, tenant quotas are auto-tuned such that each tenant gets a fair share of the resource, proportional to their original quota allocation. The algorithm initially calculates the fair quota for each active tenant as if each tenant tried to use their full quota. Then, the algorithm fairly re-allocates the spare quota from active tenants that use less than the fair quota to tenants that need more capacity. Each tenant's dynamic quota is then set to these new values.

The diagram below illustrates how this algorithm determines the dynamic quota for bandwidth quotas, where quotas of all tenants are equal. The darker rectangles show the

current bandwidth usage of each tenant, while the lighter rectangles show the bandwidth quota of each tenant. Tenants A and B already use below what would be the fair dynamic quota for them, and tenants C and D were assigned the dynamic quota to further limit their usage and to limit the combined tenant usage to within the broker-wide quota. Dynamic quotas for tenants A and B are set slightly above their current usage, allowing the broker to detect the increase in tenant demand in the next auto-tuning iteration.

Summary

We have demonstrated that Apache Kafka is multi-tenant, but only if used correctly, with manual capacity planning and over-provisioning resources to protect the cluster from unplanned increases in demand. Confluent Cloud builds upon the multi-tenancy in Apache Kafka to provide a complete cloud-native solution, including automated capacity planning and auto-scaling while supporting the pay-as-you-go model cost-efficiently.

If you'd like to get started with Confluent Cloud, sign up for a free trial and use the promo code CL60BLOG for an extra $60 of free Confluent Cloud usage.*

Get Started

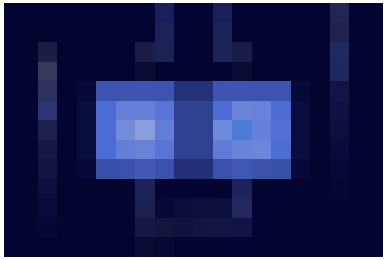
Other posts in this series

- Introduction: Design Considerations for Cloud-Native Data Systems
- Part 1: Making Apache Kafka Serverless: Lessons From Confluent Cloud
- Part 2: Speed, Scale, Storage: Our Journey from Apache Kafka to Performance in Confluent Cloud
- Part 4: Protecting Data Integrity in Confluent Cloud: Over 8 Trillion Messages Per Day

Anna Povzner is a software engineer on the Cloud-Native Kafka Team at Confluent, and she is a contributor to Apache Kafka. Her main area of expertise is in resource management and multi-tenancy in storage and distributed data systems. She received her doctorate from University of California, Santa Cruz, and was a researcher at IBM. Prior to Confluent, she was one of the early engineers in a storage startup where she helped build a scale-out, content-addressable storage system.

Anastasia Vela joined Confluent in August 2020 as a software engineer on the Kafka Cloud Native team. She earned her bachelor's degree in Data Science and Cognitive Science from the University of California, Berkeley.

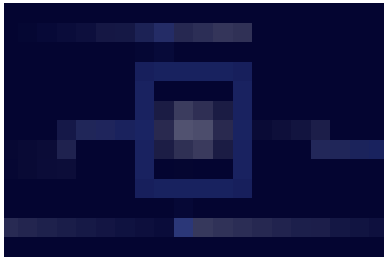
Did you like this blog post? Share it now



Build, Manage, and Monitor Data Streaming Applications, All Within Your Favorite IDE

SEP 17, 2024

We're excited to announce Early Access for Confluent for VS Code. This Visual Studio integration streamlines workflows, accelerates...



Unlock Real-Time Value from DynamoDB Data with Confluent's CDC Source Connector

AUG 27, 2024

62% of Confluent Cloud clusters run on AWS. Meanwhile, hundreds of thousands of customers are using DynamoDB. This blog...

OLIVIA GREENE
MATTHEW SEAL
ADI POLAK

AHMED SAEF ZAMZAM
BRAEDEN QUIRANTE
JAI VIGNESH R

Data Warehouse

Database