Kora: A Cloud-Native Event Streaming Platform For Kafka

Anna Povzner, Prince Mahajan, Jason Gustafson, Jun Rao, Ismael Juma, Feng Min, Shriram Sridharan, Nikhil Bhatia, Gopi Attaluri, Adithya Chandra, Stanislav Kozlovski, Rajini Sivaram, Lucas Bradstreet, Bob Barrett, Dhruvil Shah, David Jacot, David Arthur, Manveer Chawla, Ron Dagostino, Colin McCabe, Manikumar Reddy Obili, Kowshik Prakasam, Jose Garcia Sancio, Vikas Singh, Alok Nikhil, Kamal Gupta
Confluent Inc.
kora-paper@confluent.io

ABSTRACT

Event streaming is an increasingly critical infrastructure service used in many industries and there is growing demand for cloud-native solutions. Confluent Cloud provides a massive scale event streaming platform built on top of Apache Kafka with tens of thousands of clusters running in 70+ regions across AWS, Google Cloud, and Azure. This paper introduces *Kora*, the cloud-native platform for Apache Kafka at the core of Confluent Cloud. We describe Kora's design that enables it to meet its cloud-native goals, such as reliability, elasticity, and cost efficiency. We discuss Kora's abstractions which allow users to think in terms of their workload requirements and not the underlying infrastructure, and we discuss how Kora is designed to provide consistent, predictable performance across cloud environments with diverse capabilities.

PVLDB Reference Format:
Anna Povzner, Prince Mahajan, Jason Gustafson, Jun Rao, Ismael Juma, Feng Min, Shriram Sridharan, Nikhil Bhatia, Gopi Attaluri, Adithya Chandra, Stanislav Kozlovski, Rajini Sivaram, Lucas Bradstreet, Bob Barrett, Dhruvil Shah, David Jacot, David Arthur, Manveer Chawla, Ron Dagostino, Colin McCabe, Manikumar Reddy Obili, Kowshik Prakasam, Jose Garcia Sancio, Vikas Singh, Alok Nikhil, Kamal Gupta . Kora: A Cloud-Native Event Streaming Platform For Kafka. PVLDB, 16(12): 3822 - 3834, 2023.
doi:10.14778/3611540.3611567

1 INTRODUCTION

Event streaming has become an immensely popular paradigm in the past decade. It gives businesses the ability to respond in real-time to changes in market conditions. Microservices generate huge amounts of event data from numerous sources and these events must be delivered to consumer applications as quickly and efficiently as possible. Event streaming systems provide the glue to connect event producers and consumers. Like most other critical infrastructure these days, there is increasing demand for cloud-native event streaming systems with ever higher expectations for reliability, elasticity, and cost efficiency.

Apache Kafka [11] is the open-source leader in the event streaming space. Kafka is a fault-tolerant, durable, scalable, high-throughput, distributed real-time event streaming system. It is used by 80% of Fortune 500 businesses [7]. Kafka's API has become the de facto standard for event streaming as even competing systems [3, 10, 13] provide compatibility with Kafka's Produce and Consume APIs. However, Kafka was built before cloud systems dominated and its architecture reflects assumptions of a much more static environment. For example, a single-tiered storage layer made the system slow to adapt to changes in workloads since it required massive movement of data in order to rebalance load evenly in a cluster.

Confluent Cloud provides a fully-managed, cloud-native event streaming platform based on Apache Kafka. Our platform, called *Kora*, is highly available, scalable, elastic, secure, and globally interconnected. As a true cloud-native platform, it abstracts low-level resources such as Kafka brokers and hides operational complexities such as system upgrades. It supports a pay-as-you-go model: users can start small and scale their workloads to GBs/sec and back when needed while only paying for resources they use. Users can choose between a cost-effective multi-tenant configuration as well as dedicated solutions if stronger isolation is required.

In this paper, we describe our experience building Kora as a true cloud-native event streaming platform. The challenge we faced was to provide a highly available service with consistent performance at low cost across three clouds with heterogeneous infrastructure. Consistent performance covers many dimensions, such as latency and throughput, which depend on the requirements and scale of user workloads that we do not control. Heterogeneous infrastructure implies different categories and frequencies of failures that all must be handled robustly. We discuss how Kora rethinks Kafka and supporting systems' architecture for this dynamic multi-tenant cloud environment and provide production data to illustrate the effectiveness of our design choices.

Confluent Cloud has grown in the past six years to support tens of thousands of clusters across three clouds (AWS, GCP, Azure) and 73 regions. It powers a wide range of industries from retail to financial services and healthcare, including mission-critical workloads. Kora, the engine that fueled this growth, is the result of continuous evolution through feedback from the field and our efforts to improve operations while reducing costs. Rather than being based on a single key idea, Kora builds on a number of well-established ideas and principles from the literature and our experience and learning from operating our infrastructure at scale. We believe that this paper makes the following contributions for the research community.

- We present the architecture of Kora, a cloud-native event streaming platform that synthesizes well-known techniques

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611567

from literature to deliver the promise of cloud: high availability, durability, scalability, elasticity, cost efficiency, performance, multi-tenancy, and multi-cloud support. For example, our architecture decouples its storage and compute tiers to facilitate elasticity, performance, and cost efficiency.
- We describe Kora's abstractions that have been in production use for several years and enabled the service to grow to massive scale. These abstractions have helped our customers distance themselves from the underlying hardware and think in terms of application requirements, while allowing us to experiment with newer hardware types and configurations in our quest for optimum price-performance ratio.
- Finally, we give insights into key customer pain points and the challenges we faced to address them. For example, while the promise of newer, faster hardware is tempting to meet increasing customer performance demands, the reality is that evaluating new hardware for an optimum price-performance ratio takes significant investment given large and diverse workload requirements.

In the next section, we provide some background and terminology about event streaming systems and Apache Kafka specifically. We then introduce Kora's architecture and dive into several specific areas for deeper analysis.

2 BACKGROUND

Event streaming systems solve the problem of routing events from applications producing them (producers) to the downstream applications processing them (consumers). Figure 1 shows a high-level view of an event streaming application with Kafka in the middle.

Events in Kafka are organized into *topics*, each of which is partitioned for higher throughput. A *topic partition* is structured as a persistent, replicated log of events: each copy of the log is known as a *replica*. Events written to the log are interchangably referred to as either *records* or *messages*. Each record in a partition is given a unique *offset*, which is incremented sequentially after every write. *Producers* write to the end of the log while *consumers* can read from any offset. Kafka relies on file system caches for efficient access to partitions. It is optimized for consumers reading at the end of the log, which is the most common access pattern.

As consumers make progress, they record the offset of the last processed record in a separate internal topic so that they can resume processing from the same point after failures. *Consumer groups* provide a way to distribute the partitions of a topic in order to enable parallel processing. This simple design allows Kafka to support writing huge volumes of events with a high degree of read fan-out.

Application workloads are typically characterized by throughput, either in terms of the rate of events or bytes. Ingress and egress throughput are often distinguished because of the fan-out to consumer applications. It is common to have multiple consumer applications consuming the same event stream.

Latency is a critical measure of the performance of event streaming systems since applications often operate with real-time expectations. For example, a ride-sharing service must be able to respond immediately to changes in demand or road conditions. We measure

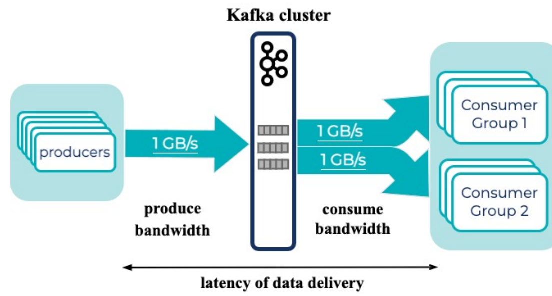

Figure 1: Event streaming model.

end-to-end latency as shown in Figure 1 as the elapsed time between event creation by a producer and delivery to the consumer.

3 OVERVIEW

This section covers Kora's design goals and high-level architecture. We identify the main components and their role in the overall service. In subsequent sections, we will discuss specific features in more detail. First, we review the goals behind the system.

3.1 Design goals

The design and architecture of Kora is motivated by the following key objectives:

Availability and Durability. Our customers use our service for business-critical services and for storing critical data. Lapses in durability or availability lead to direct revenue loss and are completely unacceptable. We offer an uptime SLA [4] of 99.95% for single zone clusters and 99.99% for multi-zone clusters

Scalability. Scalability is crucial for most customers as changing infrastructure backends is very risky and expensive, especially in terms of engineering resources. Therefore, customers want to use a backend service that they know will continue to scale as their business grows year-over-year.

Elasticity. Customers can expand and shrink their clusters as their workloads scales. Additionally, Kora adapts to changes in workload patterns to provide optimal performance for a given cluster size.

Performance. Low latency at high throughput is the hallmark of event streaming platforms. We have made the conscious choice of directly passing all performance wins to the users. Therefore, over time, users may see the performance of their applications improve.

Low cost. Our customers want all the benefits of the cloud but at the cheapest possible net cost. Our design, therefore, puts a lot of emphasis on optimizing the cost for our users and we often lean towards choices that yield better price-performance ratio.

Multitenancy. Multitenancy is one of the key enablers of a low-price and highly elastic cloud experience in the form of a pay-as-you-go model. Our design features several key mechanisms required by a truly multi-tenant Event Streaming platform work.

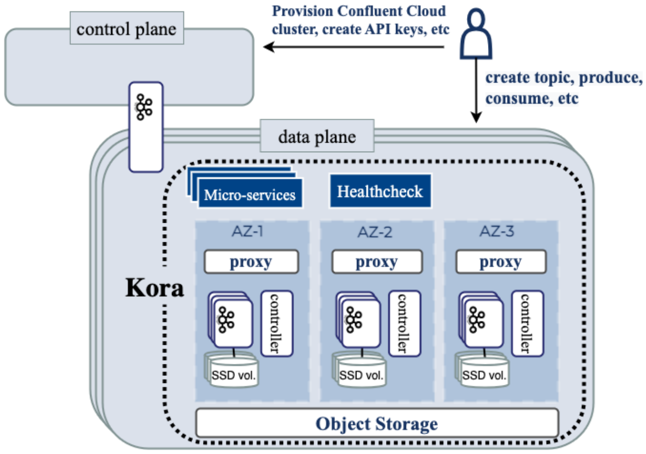

Figure 2: Control plane and data planes in Confluent Cloud. Data plane is built on Kora platform and comprises of network, storage, compute, and management microservices.

Multi-cloud support. Kora runs on AWS, GCP, and Azure. Many of our design choices are driven by our desire to provide a unified experience to our users while minimizing the operational burden due to differences between clouds.

3.2 Architecture

Confluent Cloud has two distinct decoupled pieces as shown in Figure 2: a centralized control plane and a decentralized data plane. The control plane is responsible for provisioning resources for the data plane, which consists of independent *physical Kafka clusters (or PKCs)* running on the Kora platform. Each Kora instance hosts a single PKC in our deployment and comprises of network, storage, compute, and management microservices.

The user-visible unit of provisioning, however, is a *logical Kafka cluster (or LKC)*. A PKC may host one or more LKCs depending on the needs of the application. Applications with strict requirements for isolation may use a dedicated PKC, while others may prefer the cost savings of a multi-tenant cluster. The LKC provides namespace isolation to abstract away the underlying cluster resource. For an end user, the client APIs remain the same.

Users interact with the control plane through an HTTP API to specify cluster requirements. The control plane handles the physical allocation of resources (compute, storage, network, etc) and their placement across availability zones using Kubernetes. Clusters will typically be spread evenly across availability zones in order to provide high availability, but Confluent Cloud also supports a less expensive single-zone option for use cases with reduced availability and durability requirements. The control plane is also responsible for initializing cluster configurations, such as resource quotas and user API keys. These are propagated to the PKC using the Kafka protocol and are stored in internal Kafka topics.

Each PKC in the data plane looks much like a normal Kafka cluster and users interact with it using standard Kafka clients. Within the PKC are a set of brokers which are responsible for hosting topic partition data and a set of controllers which are responsible for managing cluster metadata, such as replica assignments.

The proxy layer in each PKC is responsible for routing to individual brokers using SNI (Server Name Identification). The proxy is stateless and scales separately from brokers. It can support large clusters without experiencing bottlenecks like port exhaustion. Network access rules and connection limits are enforced at the proxy layer prior to authentication on the Kafka broker.

Every component in the system exposes telemetry for its key performance indicators. We also deploy a health check monitor which sits outside the internal network to track client-observed performance. Health check monitors continuously probe the brokers to detect any lapses in availability or performance. Probing the brokers from outside our cluster is critical to catch any issues in the network stack (e.g. DNS resolution, anything in the proxy layer) which might not be caught by the controller.

Kora deploys two significant departures from Kafka architecture as it has been known for the past ten years. First, metadata has been pulled out of Zookeeper and into an internal topic. Second, the storage layer now has two tiers: local volumes on the brokers and an object store. We review below the motivation behind these changes and how they contribute to the cloud-native architecture.

3.3 Metadata Management

Kafka's architecture relies on a centralized *controller* to manage cluster-wide metadata such as replica assignments and topic configurations. The controller is also responsible for tracking the liveness of brokers and for electing topic partition *leaders*. Brokers register with the controller on startup and maintain a session through heartbeats. If no heartbeats are received before a session timeout expires, the broker is considered offline and new leaders are elected as needed.

The centralized perspective of the controller is ideal for ensuring that the load in the cluster is balanced. However, traditionally, there was little that it could do outside of balancing replicas and leaders based on raw counts. It had no insight into the ingress/egress load on each topic, which meant that the overall load could become extremely skewed. In Kora architecture, we built an additional component in the controller, which is able to leverage a more accurate model of cluster load based on the telemetry reported by brokers. This is discussed in more detail in subsection 4.3.

Traditionally, the controller was co-located with brokers. Any broker was capable of becoming the controller through an election process facilitated by Zookeeper. On larger clusters, the co-location could be problematic since the controller's work can be substantial. For example, in the case of a broker failure, the controller must elect new leaders for thousands of topic partitions. This could cause a noticeable performance degradation for the broker. Furthermore, when rolling a cluster for an upgrade, the controller would often have to change several times. Loading a new controller from Zookeeper was not cheap, so these controller changes made the cluster unstable during rolls.

These problems were one of the central motivations behind the Kafka Raft (or KRaft) architecture [19]. Rather than using Zookeeper, metadata in KRaft is stored in an internal topic partition which uses a consensus protocol based on Raft [14]. The centralized controller is elected as the leader of this partition, and replicas follow the log and build the metadata state so that they can be ready to immediately take over leadership responsibilities after a failure.

Additionally, the role of the controller was split from the broker into a separate process. This means we can allocate resources to it independently from the brokers and its workload can be isolated more effectively. With Kubernetes, we can still pack the controller process onto the same instances that the brokers are deployed on in order to save costs on smaller clusters where the controller is not so busy. But we also have the option in larger clusters to use a dedicated instance. It also means we can roll all of the broker processes in the cluster while retaining a stable controller.

3.4 Data Storage

Traditionally, Kafka leveraged only local volumes with broker affinity in its storage layer. Each replica of a topic partition maintained its own complete copy of the entire log of events. Replication of the log was done through Kafka's own custom protocol. For a cloud service, this presents two major challenges.

First is the trade-off between cost and performance. For better performance, we need to use more expensive disk types, but since their cost is proportional to the size of the volume, it quickly becomes prohibitively expensive as the amount of data increases.

The second challenge is ensuring predictable performance. It is crucial to be able to balance replica assignments in the cluster to adapt to changes in user workloads. For Kafka, changing a replica assignment implies copying the full log of events in the topic partition to the new replicas. The time it takes to copy that data obviously increases as the size of the data increases. More data makes the system slower to react, which also means a higher risk that the workload will have changed again after the reassignment finishes. Furthermore, copying data is not only expensive from a cost perspective, but it also takes system resources away from the user workload itself.

To address these issues, we built a tiered storage layer. New event data from producers is first written to local disks and replicated through Kafka's own protocol as before. Most consumer applications continue to read event data from this tier as soon as it is written. However, as the data in the system ages, it is moved to a second tier, a much cheaper object store (such as AWS S3). After doing so, the event data can be removed from each replica. This means local volumes can be much smaller since they only need to retain the active log data. This not only gives us much more flexibility to choose disk types with better performance, but it also solves the rebalancing problem. There is no need to move archived data: we only need to move the smaller *active set* on the local volume. Even further, there is no longer a practical limit on the amount of data that a topic partition can retain. While the previous architecture was limited by the maximum size of a single local disk volume, now we are only limited by the object store.

The tiered storage layer is a big win for price and performance, but it does come with complexity. The system must maintain additional metadata about the topic partition log data which has been archived in the object store. We use an internal topic to maintain this metadata. As new segments of log data are uploaded to the object store, we publish the respective metadata to this internal topic. Each replica in the cluster watches this topic for changes so that they know when local data can be removed and to build a reference table in order to serve reads. If a consumer requests log data outside the local volume, the replica can load the correct segment from the object store.

4 CLOUD-NATIVE BUILDING BLOCKS

In this section, we discuss how we designed Kora to support cloud-native properties for dedicated Kafka clusters. The building blocks described in this section also form the basis for our multi-tenant service.

4.1 Abstractions for a cloud native cluster

Abstractions are important for delivering a true cloud-native experience, where users don't have to reason about low-level details of the cloud such as the amount of memory or CPU type, network bandwidth, IOPS/throughput/storage bandwidth, etc. By expressing our contract for performance, availability, and isolation in form of high-level constructs such as ingress and egress bandwidth, we free users from the burden of thinking about low-level details and ensuring that these low-level resources are adequate for their high-level tasks. Conversely, these high-level constructs also allow us to modify the low-level implementation details, such as the instance type or storage class, when it benefits our customers in key dimensions such as performance or cost as discussed in Section 4.2.

We abstract clusters by exposing a unit of capacity called a *Confluent Kafka Units (aka CKU)*. It represents the minimum cluster size that can be provisioned and similarly the minimum unit for expanding or shrinking the cluster. We ensure that clusters with equivalent CKUs perform comparably for the same workload across three clouds [16]. A CKU specification exposes several dimensions to users such as maximum ingress and egress bandwidth, request rate, and connection count and rate.

A CKU exposes maximum capacity on each of the dimensions to avoid artificially limiting the types of workloads it can support. However, to hit the maximum on one of the dimensions usually requires using less of other resources. For example, fully utilizing bandwidth requires good batching, fewer requests, and fewer connections. As a result, because it is a fixed-size entity, a cluster can run out of capacity before hitting any CKU limits.

We address this issue by exposing a *cluster's load* that provides visibility into the utilization of the backing physical Kafka cluster, which we approximate as utilization of the most loaded broker. This approach works well in approximating the performance that the customers can expect because well-balanced clusters have similar utilization per broker, whereas, for imbalanced clusters, the workloads experience the impact of most loaded broker in their p99 latency. We model broker utilization using a traditional definition of server utilization: the proportion of time a server is busy [8]. The advantage of this approach is that workloads experience an increase in server utilization with an increase in latency, and latency grows exponentially when the system approaches saturation. This helps users reason about their expected cluster performance based on the utilization metric.

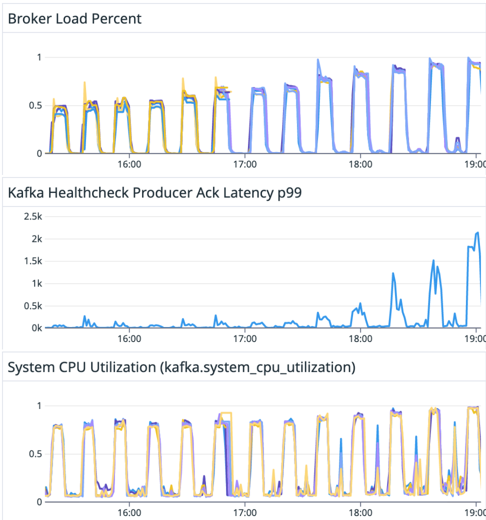

Figure 3: Impact of an increasing load from a benchmark on broker load, latency, and CPU. Each spike is a separate benchmark run with more load.

The CKU abstraction enables users to get a reasonable first-order approximation of the cluster size that they need to provision and the associated performance expectation and costs without having to run any benchmarks. Cluster load guides them when their cluster is running hot and would benefit from expansion. We are also building an auto-scaling framework that will enable them to automatically shrink/expand their cluster based on their cluster's load.

4.1.1 Broker Load (Utilization). The broker load (utilization) metric is the basis for a *cluster's load*. The key challenge is that the direct use of CPU, IOPS/disk-throughput, or network bandwidth falls short when the workload is stressing other dimensions. The direct measure of server load is not straightforward because it requires accurately measuring request service time and excluding any time waiting in the internal disk queues or other underlying resources we do not control.

Our solution was to use queuing theory laws on the relationship between latency and server utilization [8, 9]: under heavy load, latency grows exponentially with the utilization. Latency, or queueing delay, has been shown [5] to be a more robust signal that a workload needs more resources rather than utilization of specific resources such as IOPS or CPU. However, exposing latency to users as a measure of load is harder to reason about because of the exponential relationship – it is much easier to reason about utilization that grows linearly with load rather than latency growing exponentially with the load.

To measure broker utilization, we modeled the broker as a single-server queuing system with arbitrary inter-arrival and service time distribution (G/G/1). In this model, a job is either a network request or a connection creation request. The wait time W is the time a request or connection waits in various queues in the broker or underlying infrastructure, excluding waiting for replication or waiting on clients to send the response.

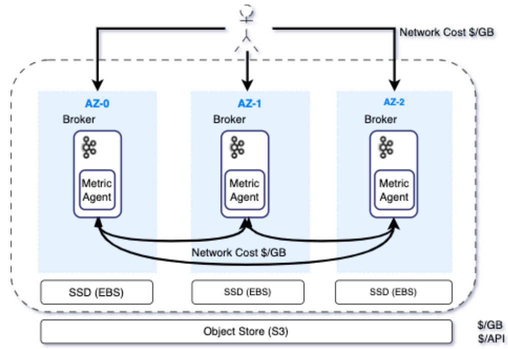

Figure 4: Cost drivers of a sample deployment in AWS.

We used Kingman's approximation of wait time under heavy load [9] to calculate the broker utilization during high load. For Kingman's formula, we calculate $E[W]$ as an exponential decaying 1-minute moving average of measured wait time W using the Unix load averaging approach [20]. We empirically found the coefficients for the formula via a set of benchmarks that covered a range of workloads. For low utilization cases, we approximate broker utilization by using utilization of network and request threads in the broker.

We illustrate the effectiveness of our broker load metric using a simple experiment with a CPU intensive workload as shown in Figure 3. We keep on increasing the load on the cluster by increasing the number of partitions and we observe that in this CPU intensive workload, broker load tracks CPU usage increase. In contrast, latency increases much more dramatically when the cluster gets overloaded showing why it is inappropriate to use latency directly as a measure of load. We have observed similar predictable trends with IO and network intensive workloads as well.

4.2 Cluster organization for cost efficiency

In this section, we describe how various services are laid out on physical resources (instance type, storage class) and how we go about choosing those physical resources with the intent of optimizing cost. Two notable design choices allow us the flexibility of adjusting our hardware and layout without impacting user-experience. First, our service expectations are described in higher level constructs as discussed in Section 4.1. These constructs allow us to change the hardware without violating the contracts for performance. In contrast, many cloud services operate in Bring-Your-Own-Account (BYOA) model, thereby punting the complexity of hardware selection and its associated tradeoffs to the users. Second, our design relies on a decoupled, persistent block storage layer rather than ephemeral instance storage. This gives us the flexibility to choose the optimum VM instance type and block storage volume independently while retaining strong guarantees of durability.

Next, we discuss three key aspects that influence cost: core Kafka compute and storage, networking, and supporting microservices. Figure 4 shows the major interactions and components from an AWS-hosted cluster for reference.

4.2.1 Core Kafka costs. Costs incurred by our core Kafka cluster are a significant part of our net cost. For low-throughput use cases on dedicated clusters, it is the dominant factor. Our goal is to leverage *right-sizing* to keep this cost at the minimum possible level needed to sustain our performance requirements.

The complexity of optimizing cost comes from the significant heterogeneity in hardware offerings and the associated cost structures, both within a cloud provider and across multiple clouds. Cloud providers offer many options that vary, even within a single cloud, along critical dimensions such as performance (which itself has multiple dimensions), fleet-wide availability, discounts, and capacity considerations. For example, for provisioned block storage, one cloud provider may support much higher base IOPS than others, whereas another provider may lack the ability to scale IOPS/throughput independently of the underlying storage capacity. Even within the same cloud provider, one block storage option comes with fixed throughput and IOPS but can burst to a higher value to support transient spikes, whereas other block storage options provide configurable IOPS/throughput/storage. Even on the machine types, availability varies. The same class of VMs may sometime include different architecture generations that perform variably [6, 18]. Lastly, newer architectures, such as ARM may not be readily available in enough capacity for all regions across all cloud providers. This heterogeneity makes it challenging to deliver a consistent cost and performance to our customers. Moreover, this challenge needs to be tackled on a continuous basis as pricing, availability of newer architecture, and storage options evolve over time.

Our goal is to optimize the performance per dollar for the guarantees we want to provide. We use the following process for evaluating new instance types. We have a baseline of resources we need to support our guarantees. For example, there is a lower bound on storage bandwidth and network bandwidth that we need to support our target ingress/egress throughput. This lower bound helps us rule out many instances and volume types. Next, we have a set of performance tests that help us assess the expected performance from the new setup. If the results look promising, we will proceed with the fleet-wide rollout in a staged manner.

Notably, this process might require further tuning to get the ideal performance. For example, we recently migrated from AWS GP2 volumes, which come with 256 MB/s of provisioned throughput and 750 IOPS (burstable to 3000) to AWS GP3 volumes, which start with 125 MB/s of provisioned throughput and 3000 IOPS. The GP3 volumes were cheaper but to get better performance from GP3, we had to change how we flushed data to disk so that we don't have a big backlog of page-cache changes built up. This change required extensive probing and analysis using a diverse range of workloads and low-level kernel knobs to extract both the cost and latency wins.

Similarly, we recently switched from a memory-optimized instance to a CPU-optimized instance with half the memory after extensive analysis. Changes like these yield significant cost savings while still improving fleet-wide performance but are very hard to do right. This is the key value proposition of using a cloud-native platform such as ours.

As a sample data point, by virtue of our continuous right-sizing efforts, we have improved the fleet-wide P99 latency by a factor of 3 over the last few months.

4.2.2 Network costs. The most significant network cost in a Kafka cluster is cross-AZ replication which is incurred by our Multi-AZ clusters. This cost is especially pronounced for throughput-dominated workloads. While these multi-AZ clusters offer superior durability and availability guarantees in the face of a zonal outage, some use cases are okay with weaker guarantees. We offer a cheaper single-AZ deployment option for these use cases that eliminates the overhead of cross-AZ replication by hosting all brokers in the same AZ.

Customers also incur network costs for client-server traffic if it leads to cross-AZ data transfer. We support a fetch-from-follower model that allows client fetch requests to be served from a follower replica in the same AZ, if there is one available and sufficiently caught-up.

4.2.3 Microservice costs. Kafka clusters require various microservices to enable observability, auditing, billing, etc. Rather than provisioning dedicated resources for these services, we *bin-pack* them alongside the Kafka brokers, while reserving about 80% of the VM's resources for use by the Kafka broker. Bin-packing works well in practice as storage and network are typically the key bottlenecks in an IO-intensive system such as Kafka. Though there are indeed workloads where this limits broker performance, it provides an attractive tradeoff as customers can scale up their clusters if they desire more performance. An alternative approach—to provision additional VMs for other non-Kafka services components— would force the customers to pay the cost for these additional nodes *all* the time whether or not their use cases benefit from it.

4.3 Elasticity

This section describes the infrastructure that supports elasticity—viz expansion, shrink and balancing. The ability to control system resources elastically is critical in the cloud as workloads are continuously changing. Our abstractions give us a clear signal on when an action should be taken—the presence of clear resource limits per CKU (e.g., number of partitions, connections, network bandwidth) and the aforementioned cluster load metric help both us and users make straightforward decisions on whether to scale up or down.

A key challenge in ensuring elasticity is that Kafka is a stateful system. A specific produce/consume request must be served by a specific broker which has the state to satisfy it. While the use of tiered storage (Section 3.4) helps immensely, we still need a mechanism to move replicas around when clusters expand or shrink or when the load pattern changes. We describe how we balance load skews first. The same infrastructure facilitates re-balancing during shrink and expand.

4.3.1 Load Balancing. In the same way that the platform on aggregate can be overloaded and needs to be expanded, a single node in the platform can be overloaded due to the current workload disproportionately affecting it. Because customer workloads are inherently unpredictable - a cloud-native system that is responsive and accommodating to changing load needs to have continuous monitoring and automatic mitigation for such scenarios.

Two key questions arise: What metrics to balance on and how reactive should the mitigation be? We chose to balance on a blend of metrics such as ingress bytes, egress bytes, disk usage and *broker load* metric described earlier in subsubsection 4.1.1. However, a load-balancing reassignment can be disruptive to client workloads as it changes metadata, forcing clients to refresh it and re-establish connections to other nodes. Therefore, a delicate balance needs to be struck between reactivity and stability. Too frequent balancing can be disruptive to clients and induce wasted work whereas too infrequent balancing can leave the brokers imbalanced leading to degraded performance.

Balancing in our cloud is managed by a component called *Self-Balancing Clusters (SBC)*. SBC is an internal component built inside the Kafka Controller that has the responsibility of collecting metrics about every Kafka broker in the cluster, creating an internal model of the cluster, and optimizing said model by reassigning replicas based on simple heuristics. Recall that in Kafka's data model, a topic has a set of partitions and each partition has a set of replicas. SBC is based on Cruise Control [12] and it is configured to act on a prioritized list of *goals*. Each goal attempts to balance a particular metric by generating a set of potential replica movements that need to be blessed by all higher-priority goals; thus, higher-priority goals have a higher chance of getting balanced. Furthermore, Cruise Control distinguishes between triggering goals, that trigger a rebalance round vs balancing goals, that are just executed in a best-effort manner. Thus, critical metrics, such as disk usage or network imbalance, need to be classified as triggering goals.

One challenge that we experienced was that it was hard to attribute some resources, e.g., broker load, to replicas, which is the unit of work reassignment in our infrastructure. We leveraged a heuristic to distribute a broker's overall resource usage between all the replicas hosted on that broker based on a weighted combination of a set of other representative metrics such as ingress bandwidth, egress bandwidth, request rate, etc. Another challenge was that some large clusters could have hundreds of thousands of replicas. To scale for such large clusters, we abstain from collecting replica-level metrics; instead, we fall back to topic or broker-level metric collections and use heuristics to attribute metrics to replicas.

The benefit of our elasticity infrastructure is best explained visually. Figure 5 is an example of a production system getting rebalanced using our infrastructure. The results show a previously-large skew in latency very quickly converging to a well-rounded balance amidst the nodes. The impact of the consolidation of skew can be seen in our health check's latency, which sees immediate improvements for its outliers.

4.3.2 Shrink and Expand. The flow of expanding a cluster, at a high level, begins from the UI. The customer initiates a scale-up from there aided by real-time information depicting the current usage of the cluster. Once the request reaches the data plane, new VMs are provisioned. SBC is notified about the presence of new brokers in the cluster and automatically initiates a broker addition operation that begins reassigning replicas to them. The addition operation is deemed complete when every newly-added broker has a fair share of load moved to it.

A key requirement of scaling up is speed. While scaling down is usually done when there isn't much pressure on the system, scaling

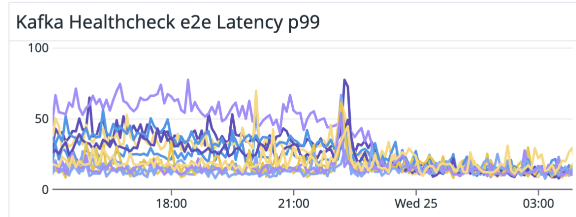

Figure 5: Latency improvements resulting from effective rebalancing. Each line represents a different broker in a cluster.

up is in stark contrast. Cluster expansion needs to complete as fast as possible before the system risks becoming overloaded, while also taking minimal system resources away from the user workload. Our tiered storage architecture is key in allowing us to achieve this since it reduces the size of data that must be moved.

In order to complete expansions as quickly as possible, SBC chooses replicas based on their contribution to the overall load, which follows a power law distribution. A minority of all replicas cause a majority of the load. From the user perspective, the expansion is complete once the new replicas are handling a fair share of the load.

4.4 Observability

Comprehensive observability is crucial to operating a large system. We focus on two types of metrics: (a) operational metrics to understand the health of a cluster and to drive the feedback loop for automated and manual mitigations, and (b) characterize fleet-wide trends that then enable us to prioritize investments for the long-term health of the platform. We have extensive instrumentation and metrics throughout our various services. We discuss some interesting challenges and insights below.

4.4.1 Client-centric end-to-end metrics. For cloud services like ours, it is imperative that we focus on client-observed metrics, especially for key performance metrics such as latency and availability. Server-side metrics omit a bunch of hops (viz *load-balancers* and *proxy*) that client requests experience. As a result, any overload in these intermediate services or network connectivity issues will be completely ignored by a server-side measurement. To bridge this gap, Kora includes a *healthcheck agent (HC)* in each cluster (Section 3.2). The HC agent sits outside our internal network and continuously probes the brokers, via produce and consume requests, to detect any lapses in availability or performance. These periodic HC requests traverse the same path—via the load-balancers and proxy servers—as the client requests, and thus are more accurate in capturing a client's perspective. The latency and success rate of these probes are recorded and fed into our dashboards and alerting system, automated mitigation systems (Section 4.5), and SLO computations (Section 4.4.2). The HC embeds a producer and consumer so that it can measure end-to-end latency exactly as a user would see it.

4.4.2 Fleet-wide SLO. While cluster-level metrics are useful for understanding how a specific cluster is performing, we need metrics to evaluate how our overall fleet is performing over a longer

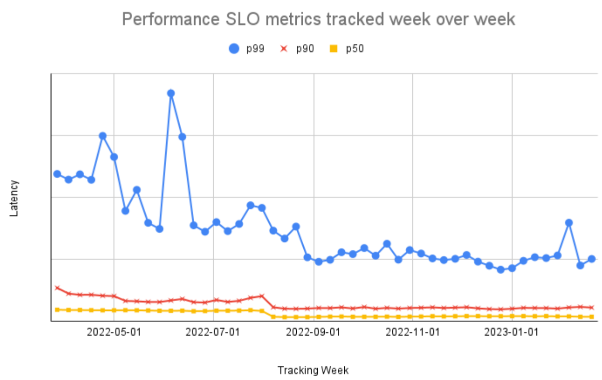

Figure 6: Fleet-wide Performance SLO tracking week over week.

period of time. Looking at a specific cluster is too noisy as workloads vary significantly across the clusters. For this reason, we created fleet-wide metrics with the following objective: (a) abstract the performance of the entire fleet in a small set of metrics (e.g., latency, availability), (b) allow us to observe trends and prioritize improvements when necessary. We used the following methodology to compute fleet-wide aggregate for a given metric (say e2e latency):

- Our HC agent sends 100 produce and 100 consume probes every 1 minute for each broker. We have special internal partitions whose leadership and assignment are sticky with each broker to ensure that we are accurately capturing the latency for that broker.
- The P99 e2e latency for that minute for that broker is computed over the set of successful probes. Similarly, for a metric like availability, we compute the number of successful requests for that minute.
- For the latency SLO computation, we take the worst e2e latency across all the brokers as the metric for that minute.
- We compute the weekly-latency-SLO metric for that cluster as P99 over all the data points for that week.
- We compute the fleet-wide-latency-SLO metrics as median, P90, and P99 of the weekly-latency-SLO metrics for all clusters.

Tracking these metrics and investigating the underlying trends in a principled manner has allowed us to identify the most widespread issues that impact our SLOs and improve our fleet-wide SLOs by several multiples over the last year. Figure 6 shows our latency SLO graph over the last 8 months. We follow an analogous methodology for tracking availability metrics as well.

4.5 Automated Mitigation

Confluent Cloud, backed by Kora, offers an uptime SLA of 99.99% for multi-zone clusters. This can be challenging to uphold when the underlying cloud providers do not all offer the same guarantees. Indeed, a majority of our availability lapses have been caused by malfunctioning cloud infrastructure. The issues we have seen roughly fall into two categories - outright unavailability of the network or storage infrastructure or severely degraded infrastructure that can persist for days and contribute to high latency. To get a sense of the problems, we will go over two examples that we have aimed to mitigate with our solution:

4.5.1 Network Unavailability. As mentioned earlier, server-side metrics can miss unavailability in any external components (Section 4.4.1). For such cases, a feedback loop between an external component and the internal cluster is necessary to convey information about the incident.

4.5.2 Storage Degradation. In-sync-replicas (aka ISR) represent the set of replicas that are actively replicating data for a single partition. Produce requests are usually configured to wait for acknowledgement from all replicas to be deemed successful. Latency is therefore largely determined by the slowest broker in the ISR. Client requests typically batch data for many partitions, which means just one slow broker out of a large set can degrade latency significantly across every partition in the batch. We have frequently seen cases where the underlying cloud SSD volume begins to exhibit chronically high latency for days unless a mitigating action to replace it is taken.

4.5.3 Mitigation. Given the widespread nature of the problem, we chose to build a principled generic solution that handles all such cases of infrastructure degradation. Specifically, we have built a feedback loop consisting of a *degradation detector* component that collects metrics from the cluster and uses them to decide if any component is malfunctioning and if any action needs to be taken.

There are a number of mitigation loops that we have implemented to address the varying problems. When a problem is detected, it is marked with a distinct broker health state each of which is treated with its respective mitigation strategy.

The health check agent probe and external client traffic are monitored in each broker by a network health manager thread. The thread monitors the requests received by both workloads—if it stops receiving requests from both sources for an extended period of time, the thread concludes that the broker has lost its external network connectivity and marks it as such. The mitigation strategy we employ during such unavailability is to migrate partition leadership away from a troubled broker. In the Kafka protocol, traffic is usually served from the leader replica of a partition. Switching leadership is a fast and effective strategy because it requires no data movement. We have abstracted the action of removing all leadership from a broker into an operation called *broker demotion*, which is executed by the controller. Upon being notified of an unhealthy broker, the controller's first step is to demote the faulty broker, resulting in the partition leadership moving onto a healthy broker that continues to serve traffic to the clients.

Similarly, each broker runs a storage health manager thread that monitors the progress of storage operations on the broker. When they do not show progress for a certain period, the thread concludes that the broker is unhealthy and restarts it. The restart has the natural effect of migrating leadership through the Kafka protocol and fencing the broker, as it will not be able to start up and join back the ISR until its storage issue is resolved.

Performance degradation of any particular node is detected based on a comparison with the cluster's global state. The mitigation

Table 1: Durability incidents that we observed in our test and production environments.

Storage corruption	Storage corruption at the leader can cause it to trim the prefix of its log. The trimming forces its followers to also trim their log leading to data loss even though Kafka's internal data replication is working correctly.
Metadata divergence	In our test environment, we observed data loss caused by the divergence in tiered storage metadata between leaders and followers. The divergence was triggered by failure to persist an update to storage.
Configuration update bug	A bug in applying Kafka's dynamic configuration settings caused spurious changes in the retention time for some topics.
Race condition in updating *log-start-offset*	*log-start-offset* tracks the starting point of non-garbage-collected log. We found a race condition in updating it that was causing Kafka to prematurely delete records.

strategy for such cases is to move the broker out of the ISR for its partitions but allow it to continue replicating data. This automatically both migrates leadership away from the broker but also ensures that it is not in the critical path of requests so that its high latency does not impact clients.

As an additional fail-safe, in case the automatic mitigation does not work, the system will notify a human operator to take manual action. We have further implemented tooling to assist them in such situations.

We have thoroughly analyzed a couple of zonal outages involving storage unavailability and observed that automated mitigation worked as designed and minimized unavailability. Similarly, during a 30 day interval, our degradation detection mechanism identified and automatically handled 12 cases of transient hardware degradation across 3 major cloud providers. Thanks to these improvements, we were able to improve our uptime SLA from 99.95% to 99.99% for multi-zone clusters and likewise, have a significant impact on our internal performance SLA.

4.6 Ensuring data durability

Our customers trust us to keep their business-critical data safe. While techniques such as replication, data scrubbing, and usage of a high-durability object store go a long way in ensuring durability, they fall short of fulfilling the guarantee users demand: that their data will be safe despite regional outages, cloud-provider outages, software bugs, disk corruption, memory corruption, misconfigurations, and even operator errors. Indeed, at the scale at which we operate, we observe these issues on a regular cadence. Table 1 shows a set of incidents that we observed in our test and production environments over the last few years. Notably, this summary doesn't include operator errors where the customer accidentally deleted their data.

Unsurprisingly, it is challenging to ensure durability in face of this broad category of potential issues. Kora provides three key protections to protect customer data. To safeguard against cloud and regional failures, Kora offers Confluent's customers the ability to set up seamless replication between distinct Kafka clusters, allowing them to failover to the backup cluster if the primary cluster suffers an outage. To guard against operator errors, misconfigurations, or bugs wiping out data, Kora implements *backup and restoration* infrastructure leveraging highly durable object storage. Finally, to protect against arbitrary bugs, Kora performs continual *durability audits* to validate our data and metadata state against a suite of data integrity invariants implemented by an *audit engine*.

4.6.1 Global replication using Cluster Linking. Kora's *cluster linking* allows replicating all data and *all* metadata state in a seamless manner across two independent Kafka clusters. These source and destination clusters can be in different regions for tolerance against regional failures, in different continents for major disaster recovery plans, or even in different cloud providers. Furthermore, because all relevant metadata is replicated between the source and destination clusters, the failover can be executed by simply pointing the clients to the new cluster endpoint. All their API keys, offsets, and partition states are preserved during the failover. For example, consumers can continue reading from the last committed offset upon failing over to the new cluster.

The key enabler for achieving this seamless experience is that we have leveraged the native Kafka replication protocol to replicate data between source and destination clusters. Because a lot of metadata is already organized as internal topics, this naturally facilitates the replication of metadata as well.

4.6.2 Backup and restore. Our backup and restore infrastructure keeps a backup of all tiered data and associated metadata for a configurable number of days. Thus, if a user discovers that they have accidentally deleted their business-critical data, they can contact us and we can recover a prefix of the log. Notably, since the only knob we expose to the users is *retention time*, users can only delete a prefix of the log. It is also noteworthy that we can only recover a prefix of the log via this mechanism; if a suffix including a non-tiered log is lost, we cannot recover it yet as the metadata state for a non-tiered log is more complex to recover.

4.6.3 Audit infrastructure. The goal of our audit infrastructure is to (a) catch bugs before they hit production and (b) discover data loss incidents in a timely manner so that their impact can be mitigated. It works as shown in Figure 7. All operations that change the consistency-related metadata state (e.g., log-start-offset, which tracks the starting offset of a log) will be logged as an *audit event* in a *durability audit database*. Periodically, typically at a daily cadence, a batch job goes through all collected audit events and validates them for consistency. For example, the validation might ensure that log-start-offset increments are aligned with the user's retention policy of X days. If the increment is larger, an alert is issued that prompts our engineers to take mitigating actions and escalate to customers. Because Kafka replicates data, in many cases, a timely alert can actually help us save data by either demoting a corrupt leader, letting the follower take over, or through other manual operations that reset the state of the corrupted broker.

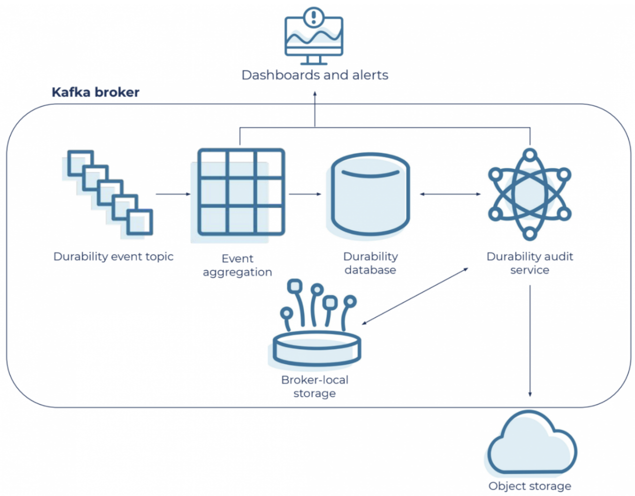

Figure 7: Durability Audit Infrastructure.

The intuition behind our auditing approach is simple: the Kafka broker is fairly complex and is constantly being evolved to add more features and improvements. In contrast, the audit engine is a very simple state machine that runs through a set of relatively static rules and policies. We use the static and robust audit state machine to catch invariant and policy violations in the Kafka code. Auditing has helped us identify critical bugs in our staging infrastructure, saving customers critical data. Additionally, the use of our audit infrastructure in production has enabled us to catch durability lapses before they could cause damage.

4.7 Upgrades

Upgrading software is critical for continuous improvement and innovation in our core service. The key challenge is how to do it safely. Indeed, prior to our investment in this area, upgrades were a major source of customer escalations due to associated high latency and transient unavailability. Notably, though we have improved the situation significantly, work is still ongoing to make the underlying Kafka protocol more robust to software upgrades.

For the production deployment, we follow the industry's best practices such as rolling upgrades, limiting the number of active versions in the fleet, and staging the upgrades. Specifically, we group brokers based on their availability zone (AZ). Our *roll* process, which restarts the brokers with an upgraded software, is facilitated by a *platform manager* and works as follows. We roll brokers in a zonal order; thus, brokers from two different zones are never rolled together as that can cause the unavailability of partitions whose replicas are hosted on those brokers. Within a zone, we can roll multiple brokers together as our placement logic guarantees that no partition will have multiple replicas in the same AZ. However, for capacity reasons, we still limit the number of brokers that are rolled simultaneously. For smaller clusters, we typically roll just one broker at a time. For larger clusters, we roll a few brokers in parallel to ensure that the end-to-end roll time for a cluster, when the cluster is in elevated load mode, stays bounded.

We have added a lot of instrumentation and monitoring to ensure that each rolled broker is fully online and functional (from a replication perspective) before we proceed to roll the next set of brokers. This ensures that our desired number of *offline brokers* is not compromised. Given that the risk of unavailability increases with the time taken to upgrade a cluster, we have also invested a lot of effort in the optimization of bottlenecks (e.g. log recovery) in the critical path of restarting a broker.

The use of a semi-automated platform manager encoding the above approach has enabled frequent fleet upgrades, allowing for faster innovation and rapid patching of security vulnerabilities and performance regressions. In contrast, upgrading a core service such as Kafka is so challenging and disruptive that many large users of self-hosted Kafka clusters operate the same version of code for several months to a few years before upgrading.

5 MULTI-TENANCY

The economies of scale that multi-tenancy offers also enables a true cloud-native experience: higher levels of abstraction, elasticity, and a pay-as-you-go model. It is cost-efficient to keep extra capacity to accommodate spikes in demand because the cost is amortized among many tenants sharing the same physical cluster. This elasticity enables event streaming workloads to scale their ingress and egress bandwidth without provisioning additional capacity and pay only for what they use.

When a user creates a cluster with a multi-tenant configuration, it shares the same physical resources with other tenants. As described in section 3.2, we use a logical unit of provisioning known as Logical Kafka Clusters (or LKCs). Each LKC is restricted by limits on the number of partitions, ingress/egress bandwidth, CPU usage, and connection rate. The underlying Physical Kafka (or PKCs) are also protected by aggregate limits to prevent resource exhaustion. Tenant workloads can scale to the maximum bandwidth as long as they stay within other limits, and capacity can be added to the PKC as needed.

5.1 Logical Cluster as a Unit of Isolation

The LKC is both an abstraction and a unit of data and performance isolation. This design unifies both dedicated and multi-tenant services in a consistent user experience. A dedicated cluster is a multi-tenant cluster with just one tenant. It also brings the benefit of isolation to internal services. We use a separate LKC for our health check agent to limit its resource usage and to isolate its state. Similarly, internal state used within Kafka itself, such as consumer offset storage, is protected through LKC isolation.

Data isolation for an LKC is achieved with authentication (via API keys), authorization, and encryption. Namespace isolation is not natively supported in Kafka. Kora supports namespacing by annotating each cluster resource (topics, consumer groups, ACLs, etc.) with the respective *logical cluster ID*, a unique identifier for each LKC. To make this namespacing transparent to the clients accessing a logical cluster, we implemented an interceptor in the broker to annotate requests dynamically using the logical cluster ID, which gets associated with a client connection during authentication.

Figure 8 illustrates this with an example of two logical clusters, *lkc-bee71e* and *lkc-c0ffee*, with clients connected to a Kafka broker

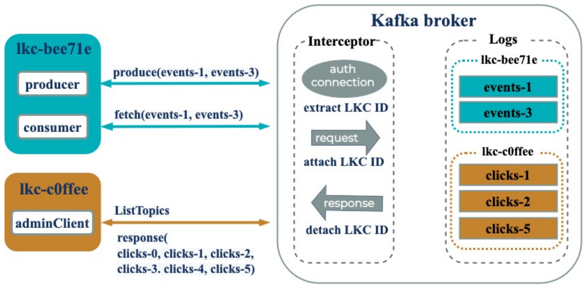

Figure 8: Logical cluster namespace isolation.

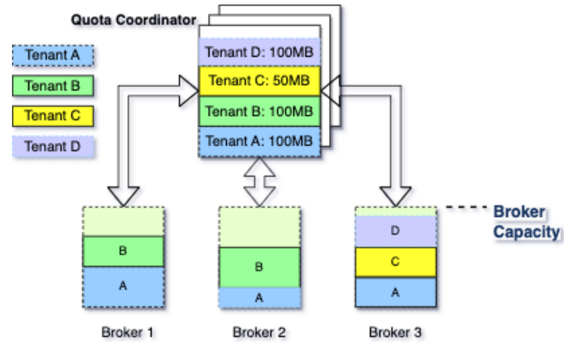

Figure 9: Quota system using bandwidth as an example.

in Kora. From the client's perspective, resources such as topics are named just as in any other Kafka cluster. The interceptor attaches a logical cluster ID to the request and ensures that each request can only access the resources owned by that tenant.

5.2 Performance Isolation

Even though a logical cluster (tenant) can share the physical cluster with other tenants, a cloud-native service must provide the experience of a dedicated cluster in terms of availability and performance. This is especially challenging in cloud-native systems with a pay-as-you-go model and a high density of tenants, needed to achieve good hardware utilization. Kora's multi-tenant clusters host thousands of tenants and any tenant's workload can experience transient spikes or permanent scale-up at any time.

We achieve performance isolation by enforcing tenant-level quotas on various resources: ingress and egress bandwidth, CPU usage, number of connections and connection attempt rates, quotas on workload behaviors impacting memory usage, and quota on partition creations/deletions to avoid overloading the Kafka controller. CPU usage by a tenant is approximated as the time the broker spends processing requests from that tenant.

The tenant-level quota is distributed among the brokers hosting the tenant. Each broker enforces their portion of the tenant quota independently. For example, tenant *A* in Figure 9 has a total quota of 100MB/sec which is distributed among brokers 1, 2, and 3. In practice, for a quota enforcement mechanism to effectively isolate tenants, we need to address two issues: 1) oversubscribed tenants may cause brokers to be overloaded; and 2) the workloads may shift usage over time between different brokers.

5.2.1 Back Pressure and Auto-Tuning. With the pay-as-you-go model, multi-tenant physical clusters are oversubscribed as most of the tenants use much less than the maximum bandwidth capacity of their logical cluster. However, any tenant's usage may spike at any time. Each broker is shared among multiple tenants, and if the demand spikes, it is possible that tenants in aggregate would need more capacity than is available on the broker. Overloaded brokers may cause high latency, timeouts, or unavailability for all tenants.

We address this issue by setting safe broker-wide limits on specific resources such as ingress and egress bandwidth, CPU, and connection rate. Once a specific limit is reached, the broker starts backpressuring requests or connections, depending on the limit reached, for all the tenants. This state is temporary: the high resource usage on a broker normally triggers a rebalance operation to even out load, or, in rare occasions, when the whole cluster is close to capacity, the multi-tenant cluster gets expanded to support the higher overall demand.

Backpressure is achieved via auto-tuning tenant quotas on the broker such that the combined tenant usage remains below the broker-wide limit [17]. The tenant quotas are auto-tuned proportionally to their total quota allocation on the broker. This mechanism ensures fair sharing of resources among tenants during temporary overload and re-uses the quota enforcement mechanism for backpressure.

The broker-wide limits are generally defined by benchmarking brokers across clouds. The CPU-related limit is unique because there is no easy way to measure and attribute CPU usage to a tenant. Instead, the quota is defined as the clock time the broker spends processing requests and connections, and the safe limit is variable. So to protect CPU, request backpressure is triggered when request queues reache a certain threshold.

5.2.2 Dynamic Quota Management. A straightforward method for distributing tenant-level quotas among the brokers hosting the tenant is to *statically* divide the quota evenly across the brokers. This static approach, which was deployed initially, worked reasonably well on lower subscribed clusters. Naturally, as clusters scaled, the static approach proved increasingly ineffective. This was especially true for imbalanced workloads with hot partitions which sometimes shift over time. Each broker's quota share may end up being relatively small, so imbalanced workloads may cause some brokers to throttle excessively, even when overall cluster usage is below the tenant-wide quota.

Kora addresses this issue by using a dynamic quota mechanism that adjusts bandwidth distribution based on a tenant's bandwidth consumption. This is achieved through the use of a shared quota service to manage quota distribution, a design similar to that used by other storage systems [15]. As shown in Figure 9, this is accomplished by periodically publishing per-tenant and per-broker bandwidth consumption data and associated throttling information to the quota coordinator. The quota coordinator then aggregates this data, recalculates the quota for each broker-tenant pair, and distributes it to the relevant brokers at configurable intervals. The calculated bandwidth is subject to the existing auto-tuning mechanism at the broker level, which may adjust the bandwidth downward to

Figure 10: Cell Architecture.

avoid overloading the broker. To accommodate large multi-tenant environments, multiple quota coordinators are deployed, and each quota entity is mapped to a coordinator via deterministic hashing.

A potential drawback of the feedback loop described above is its sensitivity to workload fluctuations. If the throughput on a partition varies significantly and unpredictably within a short time span, the dynamic quota algorithm may cause frequent and brief throttling events, which can degrade the tail latency of the system. To mitigate this issue, we implemented lazy throttling, a technique that postpones the throttling decision for a tenant until its cluster-wide usage exceeds a certain threshold relative to its tenant quota.

When we switched from a static to a dynamic quota distribution, the percentage of tenants that could achieve the 99.95% bandwidth service level objective (i.e., 5 minutes of total throttled time per week) increased from 99% to over 99.9%.

5.3 Isolation at scale

Kafka distributes the replicas of a topic across all brokers in a cluster to maximize topic throughput. In a cloud-native system, thousands of tenants share the same multi-tenant cluster and each tenant can be relatively small in comparison to the total cluster capacity. Hence, this basic topic distribution approach results in an unnecessary collocation of most of the tenants on every broker leading to several issues: (a) huge blast radius during failures, (b) manageability issues as failures are more common during cluster upgrades, and (c) less efficient use of cluster resources since spreading tenants thinly across the cluster results in more connections and requests.

Our solution is to restrict each tenant to a subset of brokers known as a *cell*. The brokers in each cell are evenly distributed across availability zones for high availability as shown in Figure 10. Each tenant is assigned to a particular cell, though this assignment can change over time. When a tenant creates a topic, its partitions are distributed across all the brokers within the cell. The cell size is selected to ensure that it can at least support the maximum bandwidth and other requirements of a single logical cluster.

When the demand from the tenants in a cell grows, causing the cell to reach its capacity, some tenants are moved to a less loaded cell. If no such cell is available, the cluster is expanded to create a new cell. We track the load of a cell using a *cell load metric* computed as the maximum of the average broker load, replica count utilization, and bandwidth utilization across the brokers in the cell.

When a new tenant is placed on the cluster, our tenant placement mechanism chooses two available cells at random and assigns the tenant to the cell with the lower load. The advantages of this algorithm have been extensively studied[1, 2]. Given that we have no knowledge of tenant load during tenant creation, this mechanism allows us to favor cells with a low load but avoid placing all tenants in a single least loaded cell which might create hotspots.

With our cellular design, we are able to scale Kafka clusters modularly, starting with a small cluster and adding more capacity as customers need it. Clusters with thousands of tenants are more costly to provision and benchmark; in contrast, cells are smaller, thus cheaper to provision and benchmark on a continuous basis. Because inter-broker replication traffic is limited to brokers within the same cell, we are able to scale almost linearly as we add more cells and tenants.

In addition to reducing the blast radius during failures and improving manageability, cells help workloads use cluster resources more efficiently. Cells help reduce the number of connections and help improve batching on the client side because partitions are distributed among a smaller number of brokers. To illustrate this point, we set up an experimental 24 broker cluster with 6-broker cells. We deployed 4 tenants, each with 2 topics of 24 partitions and 2 topics of 240 partitions. Each topic had one producer, generating 50k messages per second, and one consumer. Without cells, typically, each broker would have at least one partition from every tenant and clients would have to connect to every broker. With cells, clients connect only to brokers in the cell hosting that tenant's partitions. When we ran the benchmark, the cluster load with cells was 53% compared to 73% without cells.

6 CONCLUSION

Event streaming systems are an increasingly critical infrastructure and are undergoing a shift to cloud-based services which offer more elasticity, scalability, manageability, and cost efficiency. In this paper, we presented Kora, the cloud-native event streaming system platform based on Kafka that powers Confluent Cloud. Kora is built on the following foundations: a tiered storage layer to improve cost and performance, elasticity and consistent performance through incremental load balancing, cost effective multi-tenancy with dynamic quota management and cell-based isolation, continuous monitoring of both system health and data integrity, and clean abstraction with standard Kafka protocols and CKUs to hide underlying resources.

7 ACKNOWLEDGMENTS

Kora was the result of effort from numerous individuals, especially our current and former teammates, who worked tirelessly over the years to realize the vision of a true cloud-native event streaming platform. It is impossible to name them all but we thank them for their contributions to Kora. We are grateful to the leadership team at Confluent: Jay Kreps, Neha Narkhede, Chad Verbowski, Ganesh Srinivasan, Addison Huddy, Gwen Shapira, and Sriram Subramanian for helping formulate the strategic vision behind Kora and supporting us in building it. We also thank the Apache Kafka community for driving the growth and adoption of Kafka as the de facto event streaming platform, which led to creation of Kora.

REFERENCES

[1] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. 1995. Parallel randomized load balancing. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. 238–247.

[2] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. 1994. Balanced allocations. In *Proceedings of the twenty-sixth annual ACM symposium on theory of computing*. 593–602.

[3] Praseed Balakrishnan. 2022. *Redpanda Cloud brings the fastest Kafka® API to the cloud.* Retrieved March 1, 2023 from https://redpanda.com/blog/introducing-redpanda-cloud-for-kafka

[4] Confluent. 2022. Confluent Cloud Service Level Agreement. Retrieved March 2, 2023 from https://assets.confluent.io/m/7225ea051b0a7b42/original/20220809-Confluent_Cloud_Service_Level_Agreement-August-2022.pdf

[5] Sudipto Das, Feng Li, Vivek R Narasayya, and Arnd Christian König. 2016. Automated demand-driven resource scaling in relational database-as-a-service. In *Proceedings of the 2016 International Conference on Management of Data*. 1923–1934.

[6] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D Bowers, and Michael M Swift. 2012. More for your money: exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*. 1–14.

[7] Apache Software Foundation. 2023. *Apache Kafka.* Retrieved March 1, 2023 from https://kafka.apache.org/

[8] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press.

[9] John FC Kingman. 1962. On queues in heavy traffic. *Journal of the Royal Statistical Society: Series B (Methodological)* 24, 2 (1962), 383–392.

[10] David Kjerrumgaard. 2021. *Apache Pulsar in Action.* Simon and Schuster.

[11] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. Athens, Greece, 1–7.

[12] LinkedIn. 2023. *Cruise Control wiki.* Retrieved February 21, 2023 from https://github.com/linkedin/cruise-control/wiki

[13] Microsoft. 2023. Azure Event Hubs. Retrieved February 26, 2023 from https://azure.microsoft.com/en-us/products/event-hubs

[14] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[15] Kristal T. Pollack, Darrell D.E. Long, Richard A. Golding, Ralph A. Becker-Szendy, and Benjamin Reed. 2007. Quota enforcement for high-performance distributed storage systems. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*. 72–86. https://doi.org/10.1109/MSST.2007.4367965

[16] Anna Povzner and Scott Hendricks. 2020. Benchmark Your Dedicated Apache Kafka® Cluster on Confluent Cloud. Retrieved February 28, 2023 from https://assets.confluent.io/m/2d7c883a8aa6a71d/original/20200501-WP-Benchmark_Your_Dedicated_Apache_Kafka_Cluster_on_Confluent_Cloud.pdf

[17] Anna Povzner and Anastasia Vela. 2021 [Online]. From On-Prem to Cloud-Native: Multi-Tenancy in Confluent Cloud. Confluent, blog. https://www.confluent.io/blog/cloud-native-multi-tenant-kafka-with-confluent-cloud/

[18] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 460–471.

[19] Ben Stopford and Ismael Juma. 2021 [Online]. Apache Kafka Made Simple: A First Glimpse of a Kafka Without ZooKeeper. Confluent, blog. https://www.confluent.io/blog/kafka-without-zookeeper-a-sneak-peek/

[20] Ray Walker. 2006. *Examining Load Average.* Retrieved February 28, 2023 from https://www.linuxjournal.com/article/9001