Making Apache Kafka Serverless: Lessons From Confluent Cloud

JUL 27, 2021 READ TIME: 15 MIN

Serverless offerings in the cloud are a favorite among software engineers—a prime example are object stores such as AWS S3. For the system designer, however, it is an engineering challenge to implement a distributed, stateful system such as Apache Kafka® as a true serverless offering. Not just Kafka without ZooKeeper, but Kafka without brokers! This raises a bunch of tough questions, such as: How can you elastically expand and contract a Kafka cluster automatically, without any service disruption and without the user having to do anything? How can you automatically keep the storage in a stateful system like Kafka balanced? This blog post describes Confluent Cloud's architecture and how Kubernetes and the Confluent Operator for Kafka are leveraged to achieve a serverless experience.

The experience is created with four keys components:

1. Confluent Cloud control plane
2. Confluent Kubernetes Operator
3. Self-Balancing Clusters
4. Infinite Storage

This article begins with a brief overview of Confluent Cloud's architecture, before diving deeper into each of the four components.

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:

WRITTEN BY

Prachetaa Raghavan

- Introduction: Design Considerations for Cloud-Native Data Systems
- Part 2: Speed, Scale, Storage: Our Journey from Apache Kafka to Performance in Confluent Cloud
- Part 3: From On-Prem to Cloud-Native: Multi-Tenancy in Confluent Cloud
- Part 4: Protecting Data Integrity in Confluent Cloud: Over 8 Trillion Messages Per Day

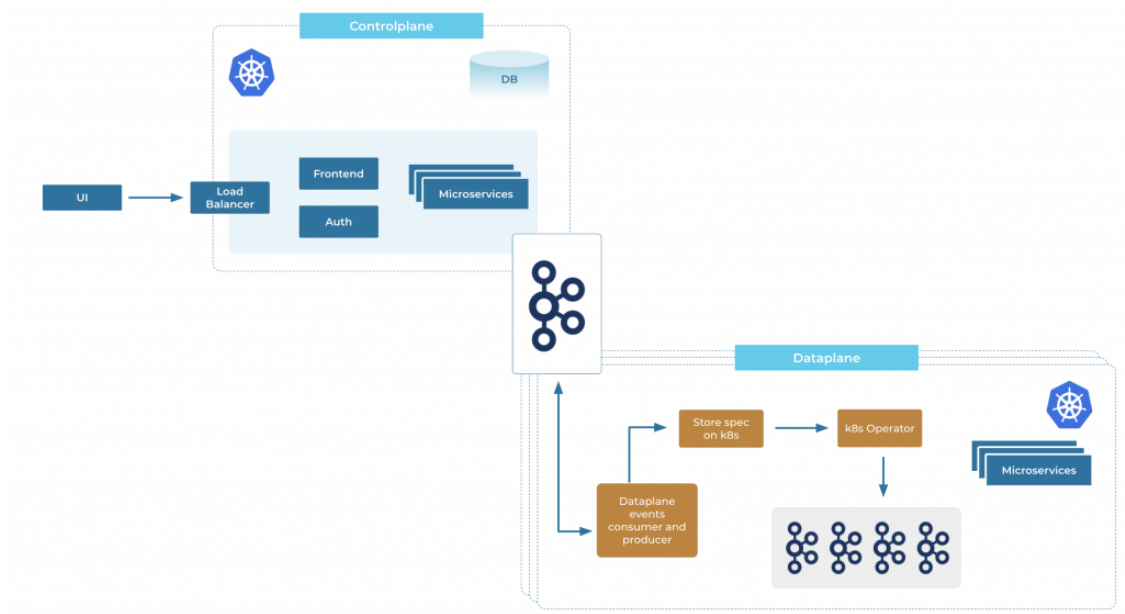
Overview of Confluent Cloud's architecture

At Confluent, we want to iterate very quickly for our customers and provide a great user experience with Confluent Cloud, taking in user feedback and building new features as well as improving on existing ones. To do so, our internal engineering teams need to move fast and be able to work independently. There are many ways to achieve this and we use an event-driven system, which allows different services to be decoupled, wherein each service can scale independently and have a higher resiliency when communicating between services. We use some well-known patterns to achieve these requirements.

We use a microservices architecture, which allows services to be independently deployable and have their own release cadence, and independent teams to maintain their own core logic and have a well-defined contract. Many of our services have their contracts defined via protocol buffers, and they communicate over gRPC.

Confluent Cloud also uses Kafka to communicate between two distinct decoupled pieces: the global/centralized **control plane** and the **data plane**. There are many instances of the data plane running in each region where Confluent Cloud is present. The communication between the control plane and various data planes is via Kafka (dubbed the mothership Kafka going forward), which sends control messages as well as receives status updates from different data planes back into the control plane.

Kubernetes is leveraged to run most of the services, both in the control plane as well as the data plane, including Kafka.



Here are some of the advantages of this architecture:

- **Decoupling failure points:** The data planes can independently function, even when the control plane has issues, and vice versa. Furthermore, if one data plane has an issue, the other data planes are not affected.
- **Auditability and observability:** We can easily figure out whether a particular data plane is slow to consume the events and look into it, and also have an event stream to know what changes have occurred over time across Confluent Cloud and replay events, if required.
- **Security and governance:** A central event stream makes it simpler to implement Confluent's security requirements—the Kafka instance needs to be protected, and consuming and producing to Kafka must be secure. This architecture also helps track and control where data comes from and who accesses it, thus providing governance.
- **Low latency and catch-up:** With an event-driven architecture, the control plane does not have to worry about the health of the data plane, as it is truly decoupled, and the data plane will consume and act on its events when it is healthy again. Within the control plane, there are some event-driven control flows, as certain operations can take a long time and we want to be asynchronous in our communication.

Beyond the high-level architecture, the following sections describe the four key components that deliver serverless elasticity for Kafka.

Control plane of Confluent Cloud

The control plane, along with all Kafka clusters and their associated connectors and ksqlDB apps, runs on Kubernetes. Confluent Cloud is present in over 50+ regions in total on Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure, and we run many Kubernetes clusters in each region, all of which are managed by the global control plane.

The control plane is responsible for:

- Provisioning, scaling, and decommissioning resources for Confluent services, such as managed Kafka, managed ksqlDB, and managed connectors.
- Managing access control across different managed services.
- Handling both internal as well as external customer-facing API requests, updating the desired state, and acting on the differences between the desired and the current state of the system. For example, for expanding a Kafka cluster, we record the intent by updating the desired state first, and then act on the desired state by resizing the Kafka cluster and sending the updated state to the data plane.
- Syncing API keys to the targeted Kafka cluster if new keys are created or keys are deactivated.
- Making placement decisions for internal as well as customer-facing resources (especially during the provision and expansion of Kafka clusters), which include placing the Kafka cluster on a particular Kubernetes cluster as well as in the right network so that it is accessible for the customer, placing the brokers and their associated volumes in different availability zones, and so on.
- And many more aspects, which are beyond the scope of this blog.

These functions are supported by many different microservices that interact with each other to make sure the customer request is satisfied.

To remove the customer burden of mapping capacity to cloud resources (instances, volumes, etc.), Confluent Cloud exposes a unit of capacity called Confluent Unit for Kafka (CKU). CKUs provide a pre-allocated amount of resources—specifying maximum capacity across various dimensions—which determines the capacity of your Kafka cluster.

When a customer expands their Kafka cluster or increases the CKUs on their Kafka cluster, the control plane translates the request into a Kubernetes custom resource, which we call a physical stateful cluster (PSC). First, we make sure that the user is authenticated and authorized to perform the operation. Once this is complete, the control plane checks whether there is enough capacity to satisfy the request. After such validations pass, the

intent of that request is persisted in a relational database. After persisting the intent, the control plane performs the translation of the request into the PSC for consumption by the data plane.

The PSC declaratively describes the desired state of the Kafka cluster (in fact, the same PSC is used to describe the desired state of ksqlDB and connectors as well), and is an input for the Confluent Kubernetes Operator explained below. The PSC is used to describe a single Kafka cluster, and handles any changes in the configuration of the Kafka cluster, such as enabling Infinite Storage (including information about the storage and how to access it), or resizing the Kafka cluster, as well as provisioning and de-provisioning the Kafka cluster. However, there are many more decisions and actions that the control plane takes at this stage when translating the request into the PSC.

When a resize request to increase the allocated resources is received, the control plane (based on the type of Kafka cluster) has to:

- Validate whether the expanded resource can fit in the available cloud resources based on current usage. We look at what the current consumption is based on different factors, such as Kubernetes cluster size, the account or project size usage, and so on, and compare it with the requested size.
- Figure out the placement of the brokers across availability zones along with their volumes.
- Recalculate the resource allocation of the Kafka cluster, such as the number of partitions, network ingress and egress, etc., based on the new size of the Kafka cluster, and sync the new CKU limits to the Kafka cluster as well as the updated storage information, etc.

Infinite Storage (enabled by Tiered Storage) is an important piece of the elasticity story, which is explained later in this blog. However, at this point, it is important to mention that tiering the data to a different storage layer also requires a certain level of orchestration in the control plane when provisioning a Kafka cluster, which is handled by an orchestration engine. The orchestration engine can be considered as a workflow engine that takes in a spec and makes sure that various dependencies on the spec are worked on in the topological order of dependencies.

The bucket provisioner service is responsible for creating a bucket for Infinite Storage based on which cloud the bucket is requested for—whether it is an S3 bucket, or a GCS bucket, or Azure Blob, etc. However, once a bucket is created, the right permissions need to be provided so that the Kafka cluster running in Kubernetes has access to read and write to the bucket. This means setting up the right credentials and access privileges, along with passing the configuration information along to Kafka so that Kafka is able to

tier to this storage layer. Furthermore, with the Bring Your Own Keys (BYOK) feature, if a key is provided by the customer during Kafka cluster provisioning, it is used to encrypt both the bucket data as well as the volumes associated with the brokers. In order to set these different pieces up during the provisioning of a Kafka cluster, the orchestration engine is responsible for working with different services wherein it understands the implicit dependencies in order to successfully provision a Kafka cluster with the right configuration.

The next section describes how the expansion intent gets propagated to the data plane.

Communication between control plane and data planes

Confluent systems use a change data capture (CDC) approach, where Debezium connectors capture the row-level changes committed to the Postgres database and send those changes to the mothership Kafka. These events are further consumed and the data is refined for consumption by the services running in the data plane. The use of a database and CDC in order for a microservice to manage its state in a relational store (with its ACID guarantees) and notify other services of any changes to the state is similar to the Outbox Pattern. For Confluent this strategy is key for reliable, recoverable, and auditable communication from the control plane services to the data plane.

The mothership Kafka has different streams intended for different types of actions. The sync-service (a service running in the data plane) listens for any events aimed at the Kubernetes cluster it is responsible for, and performs the required action based on the type of event. In an expansion, the recalculated PSC is marshaled into the protocol buffer representation and sent to the PSC topic on the mothership Kafka, and the sync-service on the data plane looks for messages that are intended for its Kubernetes cluster ID. Upon observing a message on the PSC topic, it unmarshals it and stores the PSC onto Kubernetes via the Kubernetes API.

Similarly, upon an expansion, the Kafka cluster being expanded with new configuration information needs to be updated. For example, the limits applied on the Kafka cluster need to be updated. This is very similar to syncing the PSC, wherein the updated Kafka cluster configuration is sent via a topic on the mothership Kafka and the sync-service observes the message on the topic and updates Kafka with the new configuration.

The data plane uses the same mothership Kafka to send information about the status of the operation, and the control plane services consume the status and store it in the

Postgres database, which then propagates to the customer when they view the state of the Kafka cluster via the UI or the CLI. For example, when an expansion is complete, the sync-service takes the status section on the PSC (explained a bit more in detail in the next section) and sends it to the control plane via the mothership Kafka, wherein the status information is then stored in the Postgres database. Upon an API request, the updated status information is propagated to the caller.

Many events, such as syncing new API keys or deactivating old API keys, provisioning/deprovisioning, etc., flow through the mothership Kafka, following the event-driven architecture.

Confluent Kubernetes Operator

Once consumed off of the mothership Kafka and the intent is stored onto etcd via the Kubernetes API, the Confluent Kubernetes Operator starts working on the PSC.

Confluent Kubernetes Operator—simply called operator going forward—works on the PSC (Physical Stateful Cluster) custom resource's desired state definition to implement the changes. The operator integrates closely with Kafka to perform a variety of operations.

When expanding a Kafka cluster, the operator brings up the Kubernetes external and internal network along and sets up persistent volumes for the new pods. The operator looks at the placement of the Kafka brokers in order to create persistent volume claims (PVCs) in the right zone in order to make sure the pods are also coming up per the placement requirement. It is only when the external DNS is resolvable that the operator brings up the additional pods in the Kafka cluster. This ensures that clients are able to resolve the DNS for the additional brokers that will be present in the Kafka cluster before partitions are placed on these brokers (which could happen as part of a new topic creation). As part of the pod creation, there is an init container that helps to dynamically figure out the zone in which the pod is running in order to pass the information to Kafka to ensure Kafka is aware of the zone it's running on to make placement decisions of its replicas.

```
  placement:
     us-west-2a:
       pods:
       - kafka-0
       - kafka-3
     us-west-2b:
       pods:
       - kafka-4
```

```
        - kafka-1
      us-west-2c:
        pods:
        - kafka-5
        - kafka-2
```

Note:

A subsection of the PSC that showcases the Kafka brokers/pods placement in a multi-zone Kubernetes cluster.

The operator works closely with Kafka and ensures that operations on the Kafka cluster are performed only if the operator perceives the Kafka cluster as healthy. For example, when a roll of the Kafka cluster comes along, the operator queries the metrics exposed by Kafka to make sure that there are no under-replicated partitions before restarting a broker. Pre- and post-roll validation is performed after every broker restart to make sure the Kafka cluster is in a stable state before moving on to roll or upgrade the next broker. These checks make sure that there is no loss of availability when Kafka upgrades are performed.

The Confluent Kubernetes Operator keeps an eye on the state of the resize so that the status is reported to the central control plane. It looks at the rebalancing state of Self-Balancing Clusters (SBCs), (which is covered in depth below) as well as the state of the Kubernetes cluster, looking at whether the disks are provisioned, the pods are up and running, and whether the Kafka cluster is reasonably balanced, and it updates the status section of the PSC accordingly.

```
status:
  conditions:
  - last_probe_time: 2021-07-04T13:23:52Z
    last_transition_time: 2020-07-12T15:48:06Z
    message: Cluster is not resizing, ignore=false
    reason: ClusterNotResizing
    status: "False"
    type: confluent.io/psc/cluster-resizing
 - last_probe_time: 2021-07-04T13:23:52Z
    last_transition_time: 2021-07-04T00:56:41Z
    message: Cluster is not rolling, ignore=false
    reason: ClusterNotRolling
    status: "False"
    type: confluent.io/psc/rolling
```

Note:

The status subsection of the PSC, following in the footsteps of the pod's status section in Kubernetes. It showcases different conditions, such as whether the Kafka cluster is resizing or is undergoing a rolling upgrade.

Self-Balancing Clusters (SBC's)

Kafka clusters in Confluent Cloud are self-balancing, notably with regard to their storage. If a Kafka cluster is balanced, it can distribute the load caused by Kafka clients, such as ksqlDB and Kafka Streams applications, evenly across the Kafka brokers, thus avoiding hotspots and maximizing the Kafka cluster's resource utilization.

Once the Confluent Kubernetes Operator expands the Kafka cluster and the newly added Kafka brokers start up, the Kafka cluster controller observes that the new brokers are not storing any data yet. This means the Kafka cluster's storage is now out of balance, since the workload is currently running only on the brokers that existed before the expansion. To remedy this, the Kafka cluster automatically balances the load onto the newly added brokers by migrating data (i.e., topic-partitions) onto the new brokers. With the Infinite Storage feature in Confluent Cloud, rebalancing data is very fast and efficient because only a subset of data has to be shuffled across different brokers across the network, more of which is explained in the next section.

Cluster balance is not just about storage, however. In reality, it is measured on several different dimensions, such as replica counts, leader counts, disk and network usage. Additionally, it is important to make sure that constraints such as the amount of available disk and network capacity are satisfied during any balancing decisions. Self-balancing algorithms and the many factors involved in operating a balanced Kafka cluster—not just when scaling the Kafka cluster, but also when workloads change for example—are described in more detail in the self-balancing announcement blog.

Elasticity with dual-tier architecture

Rebalancing the data in a Kafka cluster when the Kafka cluster is resized was briefly touched upon. However, no matter how amazing your self-balancing algorithms are, it is hard to create a smooth scaling experience when the process involves moving around many terabytes of data across brokers over the network—this consumes network bandwidth, disk IOPS, and CPU resources. Therefore, a critical component of the seamless elastic experience in Confluent Cloud is Infinite Storage: here, only the most immediately useful data is stored directly on Kafka brokers, and the remainder of the data

is in a separate storage tier (note: this tiering is fully transparent to clients, who simply read and write data to the Kafka cluster as usual).

With Infinite Storage, the volume of data that is present on the brokers is a small fraction of the total data in the Kafka cluster. It separates the concerns of data storage from data processing and allows each layer to scale independently. As a result, you only need to migrate a tiny subset of the data (with Infinite Storage, the time taken to add additional brokers and rebalance takes around 20-40 minutes, in comparison to without Infinite Storage, where it could take up to 1-2 days for large, busy clusters) when balancing the load in a Kafka cluster.

Confluent's dual-tier approach is different from other two-tier architectures. Notably, in other two-tier systems, the front tier is typically implemented as a read-cache over the backing tier (for long-term storage), and hence cannot provide durability or high availability of both reads and writes without the backing tier being online. In contrast, in Confluent's dual-tier architecture, the front tier operates independently from the backing tier, accepting and replicating writes as well as serving recent reads (the canonical messaging use case) *even when the backing tier is offline*. It also provides disk buffering, which allows the system to survive outages without affecting write availability. If the disks are filling up, you can expand the disks for additional buffer capacity, thereby leveraging the cloud provider's elasticity capabilities. The result is a truly elastic Kafka experience.

Figure: Confluent's dual-tier architecture.

Another important benefit of the dual-tier architecture is that data not immediately available in the front tier is read from the backing tier via a different network path. This completely separates the path that serves fresh data, often from the OS page cache and at very low latency, from the path that reads older data at high throughput without passing through the OS filesystem at all. This means that latency-sensitive apps are isolated from applications that read historical data, and both workloads can safely reside on the same Kafka cluster. In fact, because the local storage no longer needs to store all the data that your business must retain, the disks used can be optimized purely for performance—latency and bandwidth requirements—at much lower overall costs. You can find more information in our blog on infinite storage.

The future

An event streaming platform that is scalable, elastic and a seamless, serverless experience in Confluent Cloud for our customers involves:

- A control plane that translates user requests, recalculates the customer's Kafka cluster configuration, and communicates the new configuration to the data plane via the mothership Kafka.
- A decoupled control and data plane, which helps to separate points of failure and offers a very responsive experience for the customer.
- Confluent Kubernetes Operator for Kafka, which provides key functionality to make Kafka cloud-native, such as Infinite Storage and Self-Balancing Clusters. It coordinates with Kubernetes and the latter's own elastic features to help expand capacity quickly within minutes compared to hours or even days based on the load and the volume of data in the Kafka cluster.
- Leveraging public cloud providers and the elastic resources that they provide, such as spinning up new instances and provisioning new disks, etc., on demand quickly.

Increasingly, the gold standard for a cloud-native experience is "serverless." In the serverless model, services can scale down to zero, and resources are dynamically allocated based on workloads and policies. This vision—where users only need to think about their client application logic and Kafka resources magically arrive when needed and disappear when they are no longer required—is our North Star and the future we imagine for a cloud-native experience on Confluent.

If you missed Tim Berglund's awesome demo of scaling a Confluent Cloud Kafka cluster to 10 GB/sec, be sure to watch the video.

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 2: Speed, Scale, Storage: Our Journey from Apache Kafka to Performance in Confluent Cloud
- Part 3: From On-Prem to Cloud-Native: Multi-Tenancy in Confluent Cloud
- Part 4: Protecting Data Integrity in Confluent Cloud: Over 8 Trillion Messages Per Day

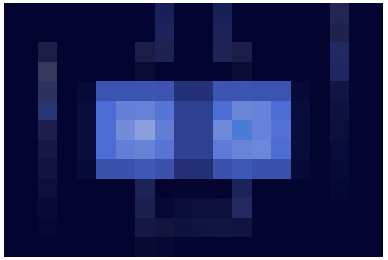
---

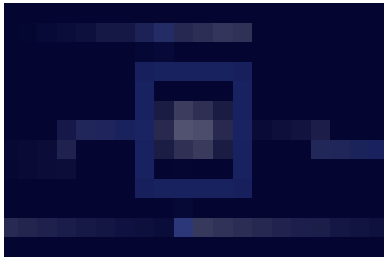
Prachetaa Raghavan is a software engineer at Confluent, focusing on different areas of the control plane. Prachetaa has worked on initiatives such as Kubernetes operator-based safe rolls of Kafka, self-serve cluster expansion, and building out the next generation unified and scalable platform of managing software releases in Confluent Cloud. Prachetaa now leads the elasticity, capacity, and operator initiatives in control plane.

Did you like this blog post? Share it now

Technology < Confluent

Subscribe to the Confluent blog





Subscribe

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...

OLIVIA GREENE
MATTHEW SEAL
ADI POLAK

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...

AHMED SAEF ZAMZAM
BRAEDEN QUIRANTE
JAI VIGNESH R

Mainframe Integration

SIEM Optimization

Hybrid and Multicloud

Internet of Things

Data Warehouse

Database