CONFLUENT

Confluent Platform Reference Architecture

© 2020 Confluent, Inc.

Table of Contents

Choosing the right deployment model is critical for the success and scalability of the Confluent event streaming platform. We want to provide the right hardware (and cloud instances) for each use case to ensure that the system reliably provides high-throughput and low-latency data streams.

This white paper provides a reference for data architects and system administrators who are planning to deploy Apache Kafka® and Confluent Platform in production. We discuss important considerations for production deployments and provide guidelines for the selection of hardware and cloud instances. We also provide recommendations on how to deploy the Kafka Connect API in production, as well as components of Confluent Platform that integrate with Apache Kafka, such as the Confluent Schema Registry, Confluent REST Proxy, and Confluent Control Center.

This document is intended to provide guidelines for self-managing Confluent Platform, either on premises or in a public cloud. If you are interested in a fully-managed event streaming service, consider Confluent Cloud.

Confluent Platform Architecture

Apache Kafka is an open source event streaming platform. It provides the basic components necessary for managing streaming data: Storage (Kafka core), integration (Kafka Connect), and processing (Kafka Streams). Apache Kafka is proven technology, deployed in countless production environments and **powering some of the world's largest stream processing systems.**

Confluent Platform includes Apache Kafka, as well as selected software projects that are frequently used with Kafka. It is a one-stop shop for setting up a production-ready event streaming platform. Among the included projects are clients for the C, C++, Python, .NET, and Go programming languages; sink and source Connectors for over 100 data systems; Confluent Schema Registry, for managing Kafka topic metadata; Confluent REST Proxy, for integrating with web applications; and ksqlDB, for SQL-driven stream processing.

Taking this to the next level, Confluent Platform also addresses requirements of modern platform streaming applications. It includes:

- Confluent Control Center, for end-to-end monitoring and management
- Confluent Replicator, for managing multi-datacenter deployments
- Confluent Auto Data Balancer, for optimizing resource utilization and easy scalability
- Tiered Storage, for unlimited retention
- Multi-Region Clusters, for high availability

- Schema Validation, for data governance
- Kubernetes Operator, for containerized installation and operation
- Ansible Playbooks and Templates, for non-containerized installation and operation
- Role-Based Access Control (RBAC), Structured Audit Logs, and Secret Protection, for enterprise-grade security

We'll start by describing the architecture from the ground level up. For each component, we'll explain when it is needed and the best plan for deploying it in several scenarios. (We will discuss capacity and hardware recommendations in a later section.) You should also refer to the **installation instructions provided in the Confluent documentation** and the **Sizing Calculator for Apache Kafka and Confluent Platform**.

ZooKeeper

ZooKeeper is a centralized service for managing distributed processes. It is a mandatory component in every Apache Kafka cluster. While the Kafka community has been working to reduce Kafka clients' dependency on ZooKeeper, Kafka brokers still use ZooKeeper to manage cluster membership and to elect a cluster controller. ZooKeeper will eventually be fully removed from Apache Kafka with KIP-500.

To provide high availability, you will need at least three ZooKeeper nodes (allowing for one-node failure) or five nodes (allowing for two-node failures). All ZooKeeper nodes are equivalent, so they usually will run on identical nodes.

The number of ZooKeeper nodes MUST be odd.

Kafka Brokers

Apache Kafka uses messaging semantics, and Kafka brokers are Kafka's main storage and messaging components. The Kafka cluster maintains streams of messages, called topics; the topics are sharded into partitions (ordered, immutable logs of messages), and the partitions are replicated and distributed for high availability. The servers that run the Kafka cluster are called brokers.

You usually will want at least three Kafka brokers in a cluster, each running on a separate server. This enables you to replicate each Kafka partition at least three times and have a cluster that will survive a failure of two nodes without data loss.

If you have three Kafka brokers and any broker becomes unavailable, you will be unable to create new topics with three replicas until all brokers are available again. For this reason, if you have use cases that require you to frequently create new topics, we recommend running at least four brokers in a cluster.

If the Kafka cluster will not experience high load, you may run Kafka brokers on the same servers as the ZooKeeper nodes. In this case, we recommend that you allocate separate disks for ZooKeeper (as we will specify in the hardware recommendations below). For high-throughput use cases, we do recommend that you install Kafka brokers on separate nodes.

In Confluent Platform, a Kafka broker is referred to as Confluent Server.

Kafka Connect Workers

Kafka Connect is an Apache Kafka component that integrates with external systems, allowing you to pull data from source systems and push data to sink systems. It works as a pluggable interface: you can plug in a connector for each system with which you want to integrate. For example, to copy data from MySQL to Kafka and from Kafka to Elasticsearch, you can deploy Kafka Connect with JDBC and Elasticsearch connectors.

A wide ecosystem of connectors is available on Confluent Hub. Confluent Hub lists over 100 connectors and other integrations that are commercially supported by Confluent, supplied by partners and verified by Confluent, or provided by the community. Each connector has its own terms, but many can be installed directly from Confluent Hub via the Confluent Hub client Command Line Interface (CLI).

Kafka Connect can be deployed in either of two modes:

- **Standalone mode:** This is similar to how Logstash or Apache Flume are deployed. If you need to get logs from a specific machine to Kafka, you can run Kafka Connect with a file connector or spooling directory connector on the machine. It reads the files and sends events to Kafka.
- **Cluster mode:** This is the recommended deployment mode for Kafka Connect in production. You install Kafka Connect and its connectors on several machines. They discover each other, with Kafka brokers serving as the synchronization layer, and they automatically load-balance work and failover between them. You can connect to any Kafka Connect node (known as a "worker") and use a REST API to start, stop, pause, resume, or configure connectors anywhere on the cluster. Regardless of which node you use to start a connector, Kafka Connect determines the optimal level of parallelism for the connector and starts parallel tasks as needed, on the least-loaded available worker nodes, to pull or push data.

In standalone mode, you deploy Kafka Connect on the servers that contain files or applications with which you want to integrate. In cluster mode, Kafka Connect usually runs on a separate set of machines, especially if you plan to run multiple connectors simultaneously. Kafka Connect workers are stateless, so they also can be safely deployed in containers.

Kafka Clients

Apache Kafka clients are used in the applications that produce and consume events. The Apache Kafka's Java client JARs are included in the Confluent Platform Kafka packages and are installed alongside Kafka brokers, but they are typically deployed with the application. To import the JARs, you can add the client libraries as application dependencies using a build tool such as Apache Maven.

At the core of Confluent's other clients (C, C++, Python, .NET, and Go) is librdkafka, Confluent's C/C++ client for Apache Kafka. librdkafka is an open-source, well-proven, reliable, and high-performance client. By basing the Python (confluent-kafka-python), .NET (confluent-kafka-dotnet), and Go (confluent-kafka-go) clients on librdkafka, Confluent provides consistent APIs and semantics, high performance, and high-quality clients in various programming languages.

Confluent Platform includes the librdkafka packages, and these should be installed on servers where applications using the C, C++, Python, .NET, or Go clients will be installed.

Kafka Streams API

Kafka Streams, a component of open source Apache Kafka, is a powerful, easy-to-use library for building highly scalable, fault-tolerant, and stateful distributed stream processing applications on top of Kafka. It incorporates important **stream processing concepts**, such as properly distinguishing between event time and processing time, handling of late-arriving data, and efficient management of application state.

Kafka Streams is a library that is embedded in the application code (similarly to Eclipse Jetty, for instance). As such, Kafka Streams does not require allocated servers, but you do need to allocate servers for applications that use the Kafka Streams library (or at least allocate resources for these applications' containers). Kafka Streams uses parallel-running tasks for the different partitions and processing stages in the application, so it benefits from a higher core count. We recommend that you deploy multiple instances of the application on multiple servers. In this case, the Kafka Streams library will handle load balancing and failover automatically.

Kafka Streams uses an embedded RocksDB database to maintain its application state. We recommend that you use persistent SSD disks for the RocksDB storage.

© 2014-2020 Confluent, Inc.

Confluent ksqlDB Server

Confluent ksqlDB is a streaming SQL engine that implements continuous queries against Apache Kafka, using an embedded instance of Kafka Connect. It allows you to query, read, write, and process data with Apache Kafka in real time and at scale, using SQL-like semantics. You can use the ksqlDB CLI to write ksqlDB queries interactively. The CLI acts as a client and can run on any machine (server or laptop) that has access to the ksqlDB server. The ksqlDB server runs the engine that executes queries. Queries can operate on either streams or tables, and the streams or tables can be backed by pre-existing or derived Kafka topics or be populated by a specified Kafka connector.

ksqlDB is typically deployed on a set of servers that form a cluster. The number of servers in the cluster is determined by the processing capacity required. This includes the number of concurrent queries that will be executed on the cluster, as well as the complexity of the queries. ksqlDB benefits from higher CPU counts, good network throughput, and SSD storage for its RocksDB state store. A general starting point for a ksqlDB server is:

- 4 cores
- 32 GB RAM
- 100 GB SSD
- 1 Gbit network

We do not recommend multi-tenant use of ksqlDB. For example, if you run two ksqlDB applications on the same node, and one application is greedy, you will likely encounter resource issues related to multi-tenancy. We recommend that you use a single pool of ksqlDB Server instances per use case.

Confluent REST Proxy

The Confluent REST Proxy is a HTTP server that provides a RESTful interface to a Kafka cluster. You can use it to produce and consume messages, view the state of the cluster, and perform administrative actions without using the native Kafka protocol or clients. The REST Proxy is not a mandatory component of the platform. Deploy the REST Proxy if you wish to produce and consume messages to or from Kafka using a RESTful HTTP protocol. If your applications only use the native clients mentioned above, you can choose to not deploy the REST Proxy.

The REST Proxy is typically deployed on a separate set of machines. For additional throughput and high availability, we recommend that you deploy multiple REST Proxy servers behind a sticky load balancer. When using the high-level consumer API, it is important that all requests to the same consumer are directed to the same REST Proxy server. As such, you can use the sticky load balancer IP address or hostname when performing any REST call except for consuming messages. When consuming messages, use the IP address or hostname returned by the REST API call to create a consumer.

Since REST Proxy servers are stateless, they also can be safely deployed in containers.

Confluent Server contains its own embedded REST Proxy, which can be used for administrative operations but not for producing or consuming messages.

Confluent Schema Registry

Confluent Schema Registry is a serving layer for your metadata. It provides a RESTful interface for storing and retrieving Apache Avro schemas. It stores a versioned history of all schemas, provides multiple compatibility settings, and allows evolution of schemas according to the configured compatibility setting. The Confluent Schema Registry packages also include serializers, which can be plugged into Kafka clients and automatically handle schema storage and retrieval for Kafka messages sent in the Avro format.

Schema Registry is typically installed on its own servers, but in a smaller installation, you can safely install Schema Registry alongside Confluent REST Proxy and Kafka Connect workers. For high availability, install multiple Schema Registry servers.

When multiple servers are running, the Schema Registry uses a leader-follower architecture. In this configuration, only one Schema Registry instance is leader at any given moment. Only the leader can publish writes to the underlying Kafka log. All nodes can directly serve read requests. Follower nodes forward write requests to the current leader. Schema Registry stores all of its schemas in Kafka, so Schema Registry nodes do not require storage and can be deployed in containers.

Confluent Replicator

Confluent Replicator is a component added to Confluent Platform to help manage multi-cluster deployments of Confluent Platform and Apache Kafka. Replicator provides centralized configuration of cross-cluster replication. Unlike Apache Kafka's MirrorMaker, it replicates topic configuration in addition to topic messages.

Confluent Replicator is integrated with the Kafka Connect framework and should be installed on the Connect nodes in the destination cluster. If there are multiple Connect worker nodes, install Replicator on all of them. When installed on a larger number of nodes, Replicator can scale to replicate at higher throughput and will be highly available through its built-in failover mechanism.

Confluent Self-Balancing Clusters

Confluent Self-Balancing Clusters is a component of Confluent Server that optimizes resource utilization and helps to scale Kafka clusters. Self-Balancing Clusters evaluates information about the number of brokers, partitions, leaders, and partition sizes, decides on a balanced placement of partitions on brokers, and modifies the replicas assigned to each broker to achieve a balanced placement. For example, when a new broker is added to the cluster, Self-Balancing Clusters will move partitions to the new broker to balance the load between all brokers available in the cluster. To avoid impact on production workloads, Self-Balancing Clusters throttles the rebalancing traffic to a fraction of the available network capacity.

You can enable Self-Balancing Clusters via a configuration option in Confluent Server. When enabled, it runs continuously in the background.

Confluent Auto Data Balancer offers functionality similar to that of Self-Balancing Clusters. However, we recommend Self-Balancing Clusters over Auto Data Balancer. We do not recommend simultaneous use of both Auto Data Balancer and Self-Balancing Clusters.

Confluent Control Center

Confluent Control Center is Confluent's web-based tool for managing and monitoring Apache Kafka. It is part of Confluent Platform and provides three key types of functionality for building and monitoring production data pipelines and streaming applications:

- **Data stream monitoring and alerting:** You can use Control Center to monitor your data streams end to end, from producer to consumer. Use Control Center to verify that every message sent is received (and received only once), and to measure end-to-end system performance. Drill down to better understand cluster usage and identify any problems. Configure alerts to notify you when end-to-end performance does not match SLAs, and measure whether messages sent were received.
- **Multi-cluster monitoring and management:** A single Control Center node can monitor data flows in multiple clusters and can manage data replication between the clusters.
- **Kafka Connect configuration:** You can also use Control Center to manage and monitor Kafka Connect, the open source toolkit for connecting external systems to Kafka. You can easily add new sources to load data from external data systems and add new sinks to write data into external data systems. Additionally, you can use Confluent Control Center to manage, monitor, and configure connectors.

Confluent Control Center currently runs on a single machine. Due to the resources required, we recommend that you dedicate a separate machine for Control Center.

Tiered Storage

Tiered Storage achieves infinite retention and elastic scalability by offloading older Apache Kafka data to inexpensive object storage.

In addition to retention, operators now can also define a *hot set*, which controls the duration for which messages are stored in Kafka before being offloaded to object storage. When a consumer attempts to fetch a message that has been offloaded to object storage and is no longer stored in Kafka, the broker can fetch the message from object storage. As such, producers and consumers do not require any change in order to use infinite retention, and you not need to install any third-party archive and retrieval tooling.

You can enable Tiered Storage via configuration options in Confluent Server.

Multi-Region Clusters

Multi-Region Clusters run a single Kafka cluster across multiple datacenters and use automated client failover to protect against disasters.

Confluent Server is often run across availability zones or nearby datacenters. If the computer network between brokers across availability zones or nearby datacenters varies in reliability, latency, bandwidth, or cost, this can result in higher latency, lower throughput, and increased cost for producing and consuming messages. Multi-Region Clusters comprises three distinct pieces of functionality that have been added to Apache Kafka and Confluent Server in order to mitigate this effect:

- **Follower-Fetching:** Follower fetching allows consumers to consume directly from replica followers. This dramatically reduces the amount of cross-datacenter traffic between clients and brokers.
- **Observers:** The observer is a new type of partition replica. By default, observers will not join the in-sync replicas (ISR), but will try to keep up with the leader just like a follower. With follower-fetching, clients can also consume from observers. By not joining the ISR, observers give operators the ability to asynchronously replicate data.
- **Replica Placement:** Replica placement defines the strategy used to assign replicas to partitions in a topic. This feature relies on the `broker.rack` property configured for each broker.

You can enable Multi-Region Clusters via configuration options in Confluent Server.

Multi-Region Clusters are distinct from traditional replicator-based HA architecture, and migration from traditional replicator-based HA to Multi-Region Clusters should be done with care, because independent clusters may have overlap in metadata such as topic naming, broker IDs, etc.

Schema Validation

Schema Validation ensures data compatibility by validating and enforcing schemas in the Apache Kafka broker, with topic-level granularity.

Prior to Schema Validation, the Apache Kafka broker would accept whatever data an authorized client would send to it. Now, using Schema Validation, an operator can require that a message have a schema, and if the message does not have a schema, it will be rejected.

You can enable Schema Validation per topic using the Confluent CLI.

Role-Based Access Control (RBAC), Structured Audit Logs, and Secret Protection

These three components provide Confluent Platform users with enterprise-grade security controls.

Role-Based Access Control (RBAC) centrally manages access to critical resources across the entire platform, with fine-tuned authorization granularity.

RBAC is a method for controlling system access based on roles assigned to users within an organization. RBAC is defined around predefined roles and the privileges that are associated with those roles (also known as role bindings). Roles are a collection of permissions that you can bind to a resource; this binding allows the privileges associated with that role to be performed on that resource. You must grant the role to a principal at the time that you bind a resource to the role.

Using RBAC, you can manage the users that have access to specific Confluent Platform resources, and you can manage the actions that a user can perform within each resource. RBAC leverages the Confluent Platform Metadata Service to configure and manage your RBAC implementation from a centralized configuration context. This simplifies access management across Confluent Platform resources.

Structured Audit Logs trace user actions in a set of dedicated Kafka topics, to help operators detect abnormal behavior and security threats.

Structured Audit Logs provide a way to capture, protect, and preserve authorization activity into topics in Apache Kafka clusters on Confluent Platform, using the Confluent Server Authorizer. Specifically, audit logs record the runtime decisions of the permission checks that occur as users attempt to take actions which are protected by access-control lists (ACLs) and RBAC. Auditable events are recorded in chronological order (though it is possible for a consumer of audit log messages to fetch them out of order).

Secret Protection avoids exposure of secrets by encrypting sensitive information (e.g. passwords) that is stored in configuration files.

Compliance requirements often dictate that services should not store sensitive data as clear text in files. This sensitive data can include passwords---such as in the configuration parameters `ssl.key.password`, `ssl.keystore.password`, and `ssl.truststore.password`---or any other sensitive data in configuration files. You can administer secrets using the Confluent CLI's `confluent secret` commands.

When you run the `confluent secret` command, Confluent Platform modifies the configuration file, adding code that directs the configuration resolution system to pull the configuration from a secret provider. Confluent Platform also creates a second file (the secrets file) that contains the encrypted secrets.

Confluent Operator

Confluent Operator is ideal for use cases where you are using Kubernetes and want simplified Confluent Platform installation and operation. Confluent Operator delivers an enterprise-ready implementation of the Kubernetes Operator API to automate deployment, as well as key lifecycle operations such as rolling upgrades and recovery, on Kubernetes.

Confluent Platform Ansible Playbooks and Templates

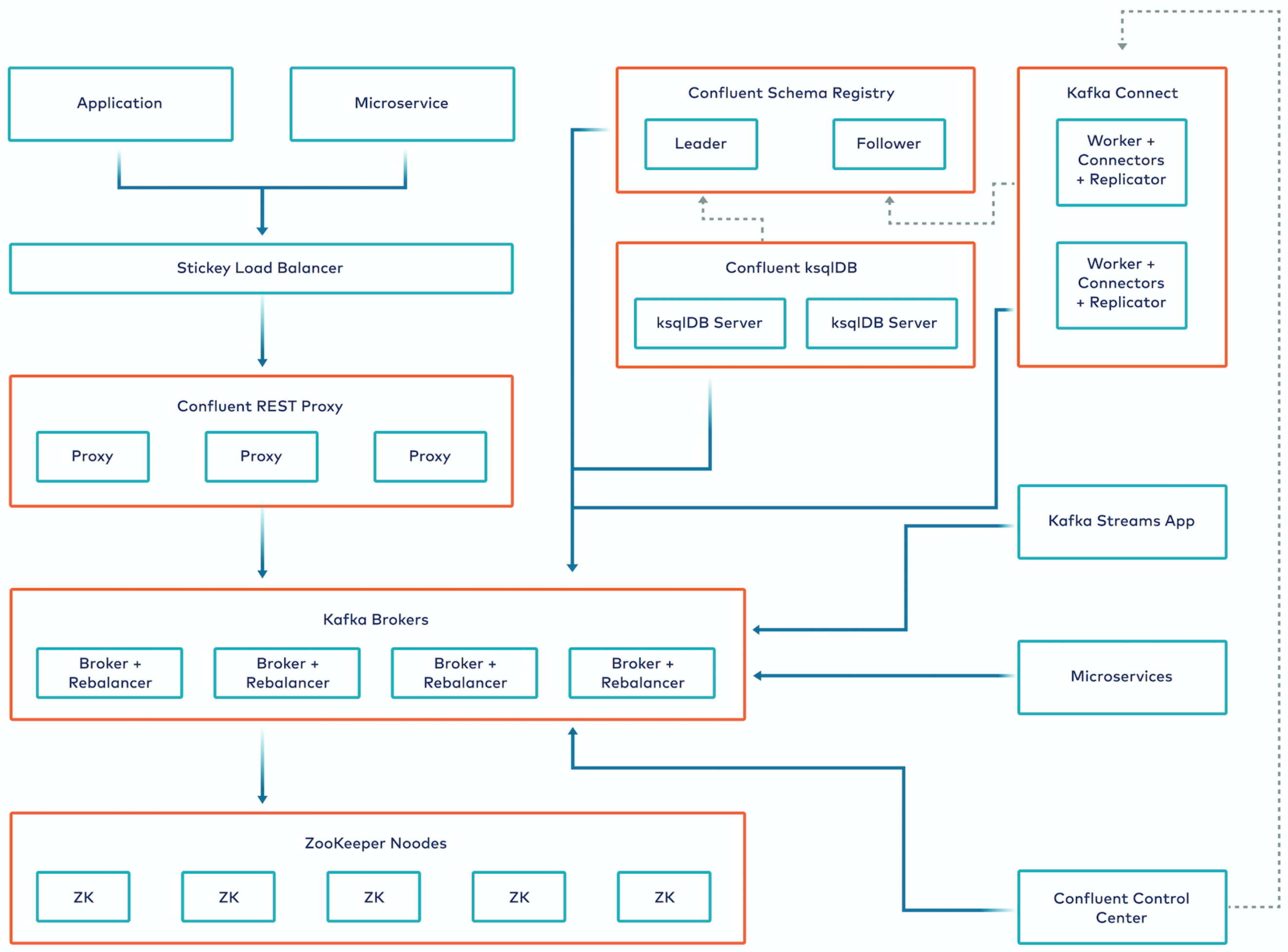
Confluent Platform Ansible playbooks and templates automate deployment of Confluent Platform in non-containerized environments, on bare metal or virtual machines.

These Ansible playbooks and templates are ideal for use cases where you are performing a non-containerized deployment and want simplified Confluent Platform installation and operation. They deliver tooling both for automating the platform's deployment, and for performing rolling upgrades as new versions of the software become available.

© 2014-2020 Confluent, Inc.

Large Cluster Reference Architecture

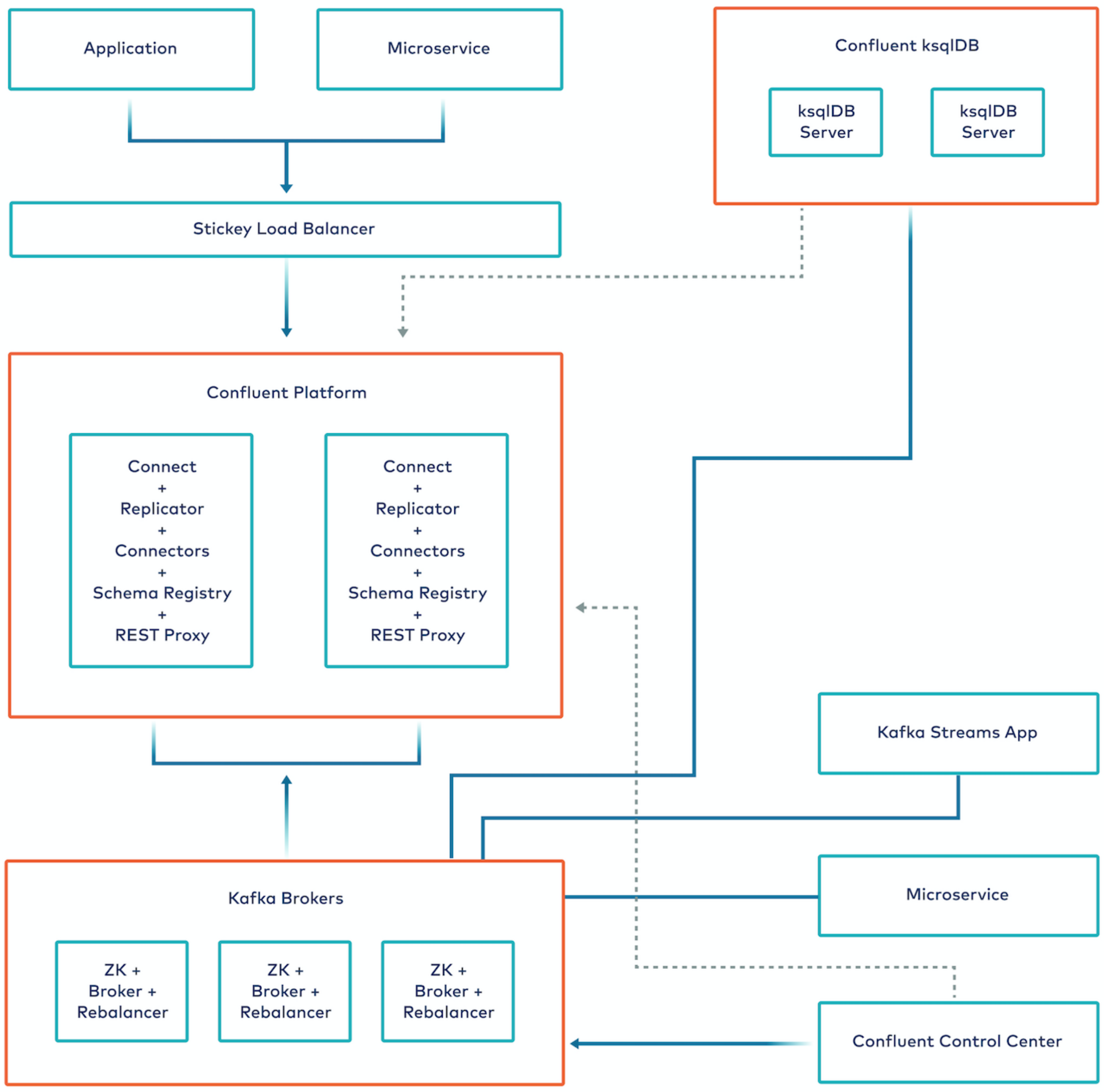
Taking all the recommendations above, a Confluent Platform cluster that is built for high-throughput long-term scalability will have an architecture similar to the following:



This architecture is designed to scale. Each component is given its own servers, and if any layer becomes overloaded, you can scale it independently by adding nodes to that specific layer. For example, when adding applications that use the Confluent REST Proxy, you may find that the REST Proxy no longer provides the required throughput, while the underlying Kafka brokers still have spare capacity. In this case, you can scale your entire platform simply by adding REST Proxy nodes.

Small Cluster Reference Architecture

Most organizations begin by adopting Confluent Platform for one use case with limited load. When this adoption proves successful, the organization will expand the cluster to accommodate additional applications and teams. We recommend this architecture for the early stages of Confluent Platform adoption, where investment in a full-scale deployment is usually not required for the success of the project. In these cases, it is best to start with fewer servers and install multiple components per server, but we recommend that you still provide dedicated servers for several resource-intensive components, such as Confluent Control Center and Confluent ksqlDB.



This architecture, although requiring far fewer servers, provides the high-availability of a full-scale cluster. As you expand your usage, you will notice bottlenecks that develop in the system. We recommend that you start by assigning bottleneck components to their own servers. When further growth is required, continue to scale by adding servers to each component. Following this approach, you can eventually evolve the recommended small cluster architecture to resemble the recommended large scale architecture.

Capacity Planning

When planning your architecture for Confluent Platform, you must provide sufficient resources for the planned workload. Storage, memory, CPU, and network resources can each be a potential bottleneck, and should be carefully considered.

Since every component is scalable, you can monitor each node's usage of storage, memory, and CPU, and add additional nodes when required. While most components do not store state, you can add nodes at any time and immediately take advantage of the added capacity. The main exception to this is the Kafka broker, which serves as the main storage component for the cluster. It is critical to closely monitor the Kafka brokers and to add capacity and rebalance before any broker is overloaded. As a general rule, we recommend that you do this whenever a resource reaches 60-70% of capacity. This is because the rebalancing operation itself will consume resources, and the more resources that are available for rebalancing, the less time the rebalancing operation will take and the sooner the cluster will resume operation with additional capacity.

You can monitor performance via Confluent Control Center or using any third-party JMX monitoring tool. To help you take action proactively, Confluent Control Center enables you to configure alerts for missed SLAs.

Storage

Storage is mostly a concern with ZooKeeper and Kafka brokers.

For **ZooKeeper**, your main concern is low latency writes to the transaction log. We recommend that you use dedicated disks to store the ZooKeeper transaction log, even in small scale deployments where ZooKeeper is installed alongside the Kafka brokers.

Kafka brokers are the main storage component in a Confluent Platform cluster, and generally require substantial storage capacity. Most deployments use 6-12 disks, usually with 1 TB of storage each. Obviously, the exact amount of storage you need will depend on the topic and partition counts, the rate at which applications write to each topic, and your configured retention policies.

You should also consider the type of storage. SSDs and spinning magnetic drives offer different performance characteristics. Depending on your use case, the performance benefits of SSDs may be worth their higher cost. While Kafka brokers write sequentially to each partition, most deployments store more than one partition per disk, and if your use case requires frequent disk access from Kafka brokers, minimizing seek times will increase throughput.

For filesystems, we recommend either ext4 or XFS. Both have been tested and used extensively in production Kafka clusters.

Although Confluent Platform supports shared-storage devices, we do not recommend their use. Confluent Platform is not tested with SAN / NAS. Achieving good performance and availability when using shared storage requires very careful configuration.

Confluent Platform's Tiered Storage supports uses cases where you need to store and access Apache Kafka data indefinitely. In addition to configuring retention policies, operators now can define a hot set, which controls how long messages are stored in Apache Kafka before being offloaded to object storage. When these messages are subsequently accessed, Tiered Storage will automatically fetch them from object storage if needed.

Confluent Control Center relies on local state stored in RocksDB. We recommend at least 300 GB of storage space, preferably in SSDs. All local data is stored in the directory specified by the `confluent.controlcenter.data.dir` configuration parameter.

Kafka Streams and **ksqlDB** are both stateful and use RocksDB as a local persistent state store. The exact use of storage depends on the specific Streams application; aggregation, windowed aggregation, and windowed joins all use RocksDB to store their state. The size required will depend on the number of partitions, number of unique keys in the stream (cardinality), size of keys and values, and the retention for windowed operations (specified in the DSL using the `until` operator).

Kafka Streams uses many few file descriptors for its RocksDB storage. Be sure to increase the number of file descriptors to 64 K or above.

Since calculating exact usage is complex, we typically allocate generous disk space to the Streams application so as to allow for sufficient local state. 100—300 GB is a good starting point for capacity planning. Confluent's ksqlDB performance tests use SSD storage for RocksDB instances, and we recommend this configuration.

Memory

Sufficient memory is essential for efficient use of almost all Confluent Platform components.

ZooKeeper uses the JVM heap, and 4 GB RAM is typically sufficient. Insufficient heap size will cause frequent garbage collection and result in high CPU usage, while an oversized heap may result in long garbage collection pauses and connectivity loss within the ZooKeeper cluster.

Kafka brokers use both the JVM heap and the operating system's page cache. Brokers use the JVM heap for replication of partitions between brokers and for log compaction. Replication requires 1 MB (the default value of the `replica.max.fetch.size` configuration property) for each partition on the broker. In Apache Kafka 0.10.1 (Confluent Platform 3.1), a new configuration property, `replica.fetch.response.max.bytes`, limits the total RAM used for replication to 10 MB. This avoids memory and garbage collection issues when there are a high number of partitions on a broker. For log compaction, calculating required memory is more complicated; if you are using this feature, refer to the Kafka documentation. For small-sized to medium-sized deployments, 4 GB heap size is usually sufficient.

In addition, we highly recommend that your consumers always read from memory, i.e. from data that has been written to Kafka and is still stored in the operating system's page cache. The amount of memory that this requires will depends on the rate at which data is written and the expected consumer lag. If, in a normal scenario, you write 20 GB of data per hour per broker and allow consumers to fall three hours behind, you should reserve 60 GB for the OS page cache. When consumers are forced to read from disk, performance will drop significantly.

Kafka Connect itself does not use much memory, but some connectors buffer data internally for efficiency. If you run multiple connectors that use buffering, increase the JVM heap size to 1 GB or higher.

The more memory given to **Confluent Control Center**, the better. We recommend at least 32 GB of RAM. The JVM heap size can be fairly small (the default is 3 GB), but the application requires additional memory for RocksDB in-memory indexes and caches and for the operating system page cache, which provides faster access to persistent data.

Kafka producer clients can achieve higher throughput given generous JVM heap sizes. In order to use the network more efficiently, Confluent's clients attempt to batch data as it is sent to brokers. They also store messages in memory until the messages are acknowledged successfully by the brokers. In the event of network or leader election issues, sufficient memory for the producer buffers will allow the producer to continue retrying to send messages to the broker, rather than blocking or throwing exceptions.

Kafka Streams and **ksqlDB** have several memory areas, and their total memory usage will depend on your specific Streams application and on the configuration. Apache Kafka 0.10.1 (Confluent Platform 3.1) adds a Streams buffer cache. By default, it is set to 10 MB. It is controlled through the `cache.max.bytes.buffering` configuration. Setting the buffer cache memory to a higher value will generally result in better performance for your Streams application.

In addition, Streams uses RocksDB memory stores for each partition involved in an aggregation, windowed aggregation, or windowed join. Kafka Streams exposes the RocksDB configuration, and we recommend using the **RocksDB tuning guide** to size these memory stores.

Kafka Streams also uses a Kafka consumer for each thread that you configure for your application. Each consumer allocates either 1 MB per partition or 50 MB per broker, whichever results in a lower total memory usage. Since calculation of all of these variables is complex, and since in general, allocating more memory increases performance of Streams applications, we typically allocate large amounts of memory, 32 GB and above.

Confluent REST Proxy buffers data for both producers and consumers. Each consumer uses at least 2 MB, and, in cases of receiving a large response from a broker (typical for bursty traffic), up to 64 MB. Producers have a buffer of 64 MB each. Begin by allocating 1 GB RAM, and add 64 MB for each producer and 16 MB for each consumer that you plan to support.

In all cases, we recommend using the JVM's G1 garbage collection, to minimize garbage collection overhead.

CPU

Most Confluent Platform components are not especially CPU-bound. Any high CPU usage is usually the result of a misconfiguration, insufficient allocated memory, or a bug. The exceptions are:

Compression: Kafka producers and consumers can be configured to compress and decompress data. We recommend that you enable compression, because it improves network and disk utilization. However, compression does require more client CPU usage. Before Apache Kafka 0.10.0, brokers decompressed and recompressed messages before storing them to disk, and this increased the brokers' CPU usage.

Encryption: Starting at version 0.9.0, Kafka clients can communicate with brokers using SSL. Encryption always brings a small performance overhead on both the client and the broker. Overhead is increased when the consumer connects over SSL, as the broker must encrypt messages before sending them to the consumer and cannot use the standard zero-copy optimization. This situation results in significantly higher CPU usage for the broker. Large scale deployments often go to some lengths to ensure that consumers are deployed within the same LAN as the brokers, which can omit the requirement of encryption.

High rate of client requests: If you have a large number of clients, or if consumers are configured with `max.fetch.wait=0`, they can send very frequent requests and effectively saturate the broker. In this situation, you can improve performance by configuring clients to batch requests.

Many Confluent Platform components are multi-threaded and benefit more from a large number of cores than from faster cores.

Network

Large-scale Kafka deployments using 1GbE typically become network-bound. If you are planning to scale the cluster to over 100MB/s, you must provision a higher-bandwidth network.

When provisioning for network capacity, you should take the replication traffic into account and leave some overhead for rebalancing operations and bursty clients. Network is one of the resources that are most difficult to provision; if you continue to add nodes, you will eventually run into switch limitations. We recommend that you consider enabling compression in order to achieve better throughput from existing network resources.

The Kafka producer will compress messages in batches, so configuring the producer to send larger batches will result in better compression ratio and improved network utilization.

Hardware Recommendations for On-Premises Deployment

Large Cluster

Component	Nodes	Storage	Memory	CPU
ZooKeeper	5, for fault tolerance	Transaction log: 512 GB SSD Storage: 2 x 1TB SATA, RAID 10	32 GB RAM	2-4 cores (typically not a bottleneck)
Kafka Broker	Minimum 3 (more for additional storage, RAM, network throughput)	12 x 1 TB disk. RAID 10 is optional. Separate OS disks from Kafka storage.	64 GB RAM+ (more is better)	Usually dual 12-core sockets
Kafka Connect	Minimum 2, for high availability	Only required for installation	0.5-4 GB heap size, depending on connectors used	Not typically CPU-bound; larger number of cores is more beneficial than faster cores
Confluent Schema Registry	Minimum 2, for high availability	Only required for installation	1 GB heap size	Not typically CPU-bound; larger number of cores is more beneficial than faster cores

Component	Nodes	Storage	Memory	CPU
Confluent REST Proxy	Minimum 2, for high availability (more for additional throughput)	Only required for installation	1 GB overhead + 64 MB per producer and 16 MB per consumer	Minimum 16 cores to handle parallel HTTP requests and background threads for consumers and producers
ksqlDB	Minimum 2, for high availability (more for additional throughput)	SSD; sizing depends on number of concurrent queries and on aggregation performed	Minimum 20 GB (Confluent tests use 30 GB)	Minimum 4 cores
Confluent Control Center	1	Minimum 300 GB (preferably SSD)	32 GB+	Minimum 8 cores; ideally more

Small Cluster

Component	Nodes	Storage	Memory	CPU
ZooKeeper + Kafka Broker	Minimum 3	12 x 1 TB disks 1 dedicated disk for ZooKeeper transaction log 1-2 dedicated disk(s) for OS Remaining disks dedicated to Kafka data	64 GB RAM+ (more is better)	Usually dual 12-core sockets
Kafka Connect + Confluent Schema Registry + Confluent Rest Proxy	Minimum 2	Only required for installation	1 GB for Kafka Connect 1 GB for Schema Registry 1 GB + 64 MB per producer + 16 MB per consumer for REST Proxy	Minimum 16 cores
ksqlDB	Minimum 2, for high availability (more for additional throughput)	SSD; sizing depends on number of concurrent queries and on aggregation performed	Minimum 20 GB (Confluent tests use 30 GB)	Minimum 4 cores
Confluent Control Center	1	Minimum 300 GB (preferably SSD)	32 GB+	Minimum 8 cores

© 2014-2020 Confluent, Inc.

Public Cloud Deployment

Today, many deployments run on public clouds, where node sizing is more flexible than ever before. The hardware recommendations discussed earlier are applicable when provisioning cloud instances. Special considerations to take into account are:

- **Cores:** Cloud providers use "virtual" cores when sizing machines. These are typically weaker than the modern cores you would use in your data center. You may need to scale up the number of cores when planning cloud deployments.
- **Network:** Most cloud providers only provide 10GbE on their highest-tier nodes. Ensure that after taking replication traffic into account, your cluster has sufficient nodes and network capacity to provide the required throughput.

Below are some examples of instance types that can be used in various cloud providers. Note that cloud offerings continuously evolve and that typically a variety of nodes share similar characteristics. This should not present a problem as long as you adhere to the hardware recommendations. The instance types below are only examples.

While the examples below show each component on a separate node, some operations teams prefer to standardize on a single instance type. This approach is easier for automation, but it also requires that you standardize on the largest required instance type. In this case, you may choose to co-locate some services, as long as each instance has sufficient resources for all of the co-located components on the instance.

You will need multiple instances of each node. Our previous recommendations regarding the number of Kafka brokers, Confluent REST Proxy servers, Kafka Connect workers, etc. still apply.

Amazon AWS EC2

Component	Node Type	Memory	CPU	Storage	Network
ZooKeeper	m5.large	8 GB	2 vCPU	1 x 32 GB SSD	Up to 10 Gbps
Kafka broker	r5.xlarge	30.5 GB	4 vCPU	SSD-based EBS storage Configure the instances as “EBS Optimized”	Up to 10 Gbps
Kafka Connect	c5.xlarge	8 GB	4 vCPU	Use EBS	Up to 10 Gbps

© 2014-2020 Confluent, Inc.

Confluent REST Proxy	c5.xlarge	8 GB	4 vCPU	Use EBS	Up to 10 Gbps
Confluent Schema Registry	m5.large	8 GB	2 vCPU	Use EBS	Up to 10 Gbps
Confluent KSQL	i3.xlarge or r5.xlarge	30.5 GB	4 vCPU	Use EBS (SSD, optimized)	Up to 10 Gbps
Confluent Control Center	m5.2xlarge	32 GB	8 vCPU	Use EBS (SSD, optimized)	Up to 10 Gbps

In past versions of this document, we also recommended use of "Storage Optimized" instances with local SSDs for Kafka brokers. At the time, we had concerns about EBS stability, latency, and throughput. Our experience in the past year has shown us that EBS is stable and can deliver the latency and throughput required by users of Confluent Platform. AWS gives you a choice of four tiers of EBS performance and "EBS Optimized Instances" with QoS guarantees. This is useful in cases where consistent storage performance is important.

Microsoft Azure

Component	Node Type	Memory	CPU	Storage	Network
ZooKeeper	DS3v2	14 GiB	4vCPU	P6 Premium Storage	high
Kafka broker	DS4v2	28 GiB	8 vCPU	S30 Managed disks	high
	DS5v2	56 GiB	16 vCPU	7 x P30 Premium Storage	high
Kafka Connect	A4v2	8 GiB	4 vCPU	None	high
Confluent Schema Registry	A2v2	4 GiB	2 vCPU	None	moderate
Confluent REST Proxy	A4v2	8 GiB	4 vCPU	None	high
Confluent Control Center	DS4v2	28 GiB	8 vCPU	None	high

Azure networking uses private IPs within the deployed VNet. Accessing the nodes from outside of Confluent node's region requires a network gateway. Azure provides gateways in three flavors:

- VPN Gateway https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-plan-design
- Public IP Gateway https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-public-ip-address
- Express Route Gateways https://docs.microsoft.com/en-us/azure/expressroute/expressroute-introduction

To ensure proper network connectivity, review the throughput limits of each type of gateway.

Google Cloud Compute Engine

Component	Node Type	Memory	CPU	Storage	Network
ZooKeeper	n1-standard-2	7.5 GB	2 vCPU	Zonal SSD persistent disks	
Kafka broker	n1-highmem-4	26 GB	4 vCPU	Zonal Standard persistent disks	
Kafka Connect	n1-standard-4	15 GB	4 vCPU	Zonal Standard persistent disks	
Confluent REST Proxy	n1-standard-4	15 GB	4 vCPU	Zonal Standard persistent disks	
Confluent Schema Registry	n1-standard-2	7.5 GB	2 vCPU	Zonal Standard persistent disks	
Confluent KSQL	n1-highmem-4	26 GB	4 vCPU	Zonal SSD persistent disks	
Confluent Control Center	n1-highmem-8	52 GB	8 vCPU	Zonal SSD persistent disks	

GCP persistent disks can be sized up to 64 TB, which is more than recommended by Confluent. For storage specifics, follow the recommendations for on-premises deployment.

Conclusion

This white paper is intended to share some of our best practices for deployments of Confluent Platform. Of course, each use case and workload is slightly different, and the best architectures are tailored to the specific requirements of the organization. When designing an architecture, considerations such as workload characteristics, access patterns, and SLAs are very important, but they are too specific to cover here. For help in choosing the right deployment strategy for your specific use cases, we recommend that you engage with Confluent Professional Services for architecture and operational review.

© 2014-2020 Confluent, Inc.