Confluent Enterprise Reference Architecture

Gwen Shapira

ABSTRACT

Choosing the right deployment model is critical for the success and scalability of the streaming data 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 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 hardware selection and the selection of instances for cloud providers. We also provide recommendations on how to deploy Kafka Connect in production, as well as components of Confluent Platform that integrate with Apache Kafka, such as the Schema Registry, REST Proxy, and Confluent Control Center.

Table of Contents

Confluent Enterprise Architecture

Apache Kafka is an open source streaming platform designed to provide 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 to power 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, which makes Confluent Platform a one-stop shop for setting up a production-ready streams platform. These projects include clients for C, C++, Python, and Go programming languages; connectors for JDBC, Elastic Search, and HDFS; Schema Registry for managing metadata for Kafka topics; and REST Proxy for integrating with web applications.

Confluent Enterprise takes this to the next level by addressing requirements of modern enterprise streaming applications. It includes Confluent Control Center for end-to-end monitoring of data streams, Replicator for managing multi-datacenter deployments, and Automated Data Rebalancer for optimizing resource utilization and easy scalability.

We'll start describing the architecture from ground level up and for each component we'll explain when it is needed and the best plan for deploying it in several scenarios. We will not discuss capacity or hardware recommendations at this point, as this will be discussed in depth in the next section. In addition, you should refer to Confluent documentation for installation instructions.

Zookeeper

Zookeeper is a centralized service for managing distributed processes and is a mandatory component in every Apache Kafka cluster. While the Kafka community has been working to reduce the dependency of Kafka clients on ZooKeeper, Kafka brokers still use ZooKeeper to manage cluster membership and elect a cluster controller.

In order to provide high availability, you will need at least 3 ZooKeeper nodes (allowing for one-node failure) or 5 nodes (allowing for two-node failure). All ZooKeeper nodes are equivalent, so they will usually run on identical nodes. Note that the number of ZooKeeper nodes MUST be odd.

Kafka Brokers

Kafka brokers are the main storage and messaging components of Apache Kafka. Kafka is a streaming platform that uses messaging semantics. 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 will usually want at least 3 Kafka brokers in a cluster, each running on a separate server. This way you can replicate each Kafka partition at least 3 times and have a cluster that will survive a failure of 2 nodes without downtime.

If the Kafka cluster is not going to be highly loaded, it is acceptable to run Kafka brokers on the same servers as the ZooKeeper nodes. In this case, it is recommended to allocate separate disks for ZooKeeper (as we'll specify in the hardware recommendations below). For high-throughput use cases we do recommend installing Kafka brokers on separate nodes.

Kafka Connect

Kafka Connect is a component of Apache Kafka that allows it to integrate with external systems to pull data from source systems and push data to sink systems. It works as a pluggable interface, so you plug in Connectors for the systems you want to integrate with. For example, you deploy Kafka Connect with JDBC and Elastic Search connectors to copy data from MySQL to Kafka and from Kafka to Elasticsearch. The full list of available connectors can be found here: https://www.confluent.io/product/connectors/

Kafka Connect can be deployed in one of two modes:

- **Standalone mode:** This is similar to how Logstash or Apache Flume are deployed. You need to get logs from a specific machine to Kafka, you run Kafka Connect with a file connector or spooling directory connector on the machine and it reads the files and send 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 and failover work between them. To start and stop connectors anywhere on the cluster, you connect to any Kafka Connect node (known as "worker") and use a REST API to start, stop, pause, resume, and configure connectors. No matter which node you use to start a connector, Kafka connect will determine the optimal level of parallelism for the connector and start parallel tasks to pull or push data as needed on the least loaded available worker nodes.

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

Kafka Clients and Kafka Streams

Apache Kafka's Java clients and Kafka Streams libraries are used within custom applications. Even though Kafka Streams and the Java client JARs are included in Confluent Platform's Kafka packages and are installed alongside Kafka brokers, they are typically declared as application dependencies using a build manager such as Apache Maven and are deployed with the application that imports them.

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

At the core of Confluent's other clients (C, C++, Python, 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 our Python (confluent-kafka-python) and Go (confluent-kafka-go) clients on librdkafka, we provide consistent APIs and semantics, high performance and high-quality clients in various programming languages.

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

REST Proxy

The Kafka REST Proxy is an open source HTTP server that provides a RESTful interface to a Kafka cluster. It makes it easy 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 - you will choose to use the REST Proxy if you wish to produce and consume messages to/from Kafka using a RESTful http protocol. If your applications only use the native clients (mentioned above), you can choose not to deploy the REST Proxy.

The REST Proxy is typically deployed on a separate set of machines. For additional throughput and high availability, it is recommended to 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 will be directed to the same REST Proxy server, so use of a “sticky” load balancing policy is recommended.

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

Schema Registry

Schema Registry is an open source serving layer for your metadata. It provides a RESTful interface for storing and retrieving 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 that plug into Kafka clients and automatically handle schema storage and retrieval for Kafka messages that are sent in the Avro format. Schema Registry is typically installed on its own servers, although for smaller installations it can safely be installed alongside REST Proxy and Connect workers. For high availability, you’ll want to install multiple Schema Registry servers. With multiple servers, the Schema Registry uses a leader-followers architecture. In this configuration, at most one Schema Registry instance is leader at any given moment. Only the leader is capable of publishing writes to the underlying Kafka log, but all nodes are capable of directly serving read requests. Follower nodes forward write requests them to the current leader. Schema Registry stores all its schemas in Kafka, and therefore Schema Registry nodes do not require storage and can be deployed in containers.

Replicator

Replicator is a new component added to Confluent Platform Enterprise in release 3.1 to help manage multi-cluster deployments of Confluent Platform and Apache Kafka. It provides a centralized configuration of cross-cluster replication. Unlike Apache Kafka's MirrorMaker, it replicates topic configuration in addition to the messages in the topics.

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, Replicator should be installed on all of them. By installing Replicator on a larger number of nodes, Replicator will scale to replicate at higher throughput and will be highly available through a built-in failover mechanism.

Auto Rebalancer

Auto Rebalancer is a new component added to Confluent Platform Enterprise in release 3.1 to optimize resource utilization and help scale Kafka clusters. Auto Balancer evaluates information on the number of brokers, partitions, leaders and sizes of partitions to decide on a balanced placement of partitions on brokers and modify the replicas assigned to each broker to achieve a balanced placement. For example, when a new broker is added to the cluster, Auto Rebalancer will move partitions to the new broker to balance the load between all brokers available in the cluster. To avoid impact on production workload, the rebalancing traffic can be throttled to a fraction of the available network capacity.

Auto Rebalancer can be installed on any machine in the Confluent Platform cluster - it just needs to be able to communicate with the Kafka brokers and ZooKeeper to collect load metrics and send instructions to move partitions. For convenience, we recommend installing it alongside Kafka brokers or on Confluent Control Center node if available.

Confluent Control Center

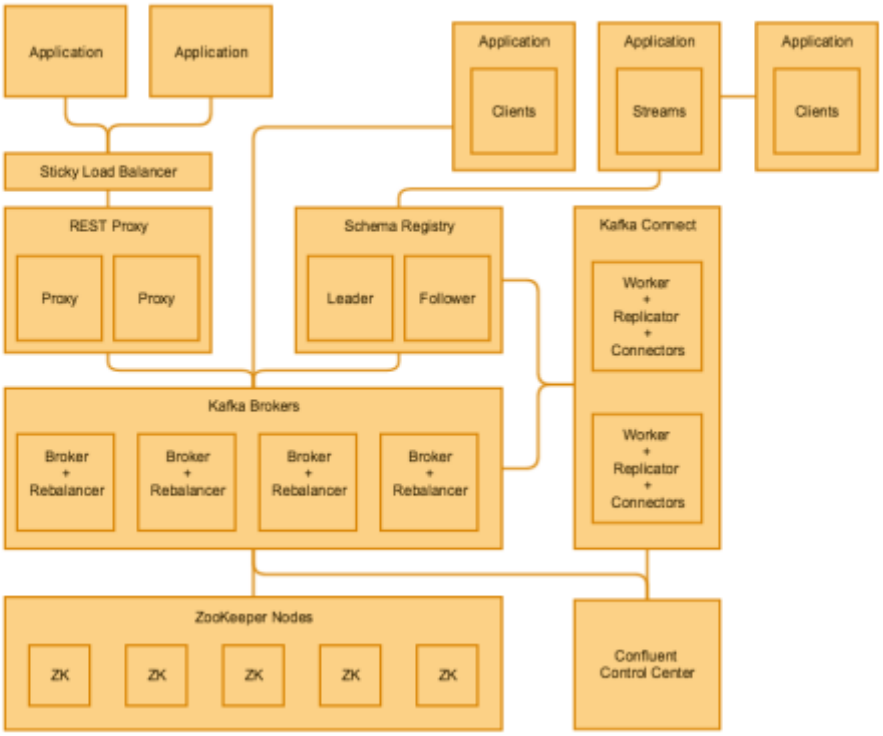
Control Center is Confluent's web based tool for managing and monitoring Apache Kafka. It is part of Confluent Platform Enterprise and provides two key types of functionality for building and monitoring production data pipelines and streaming applications:

- **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 system performance end to end. 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 or measure whether messages sent were received.
- **Multi-cluster monitoring and management** - A single C3 node can monitor data flow in multiple clusters and 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 new sinks to write data into external data systems. Additionally, you can manage, monitor, and configure connectors with Confluent Control Center.

Confluent Control center currently runs on a single machine, and due to the resources required we recommend dedicating a separate machine for Control Center.

Large Cluster Reference Architecture

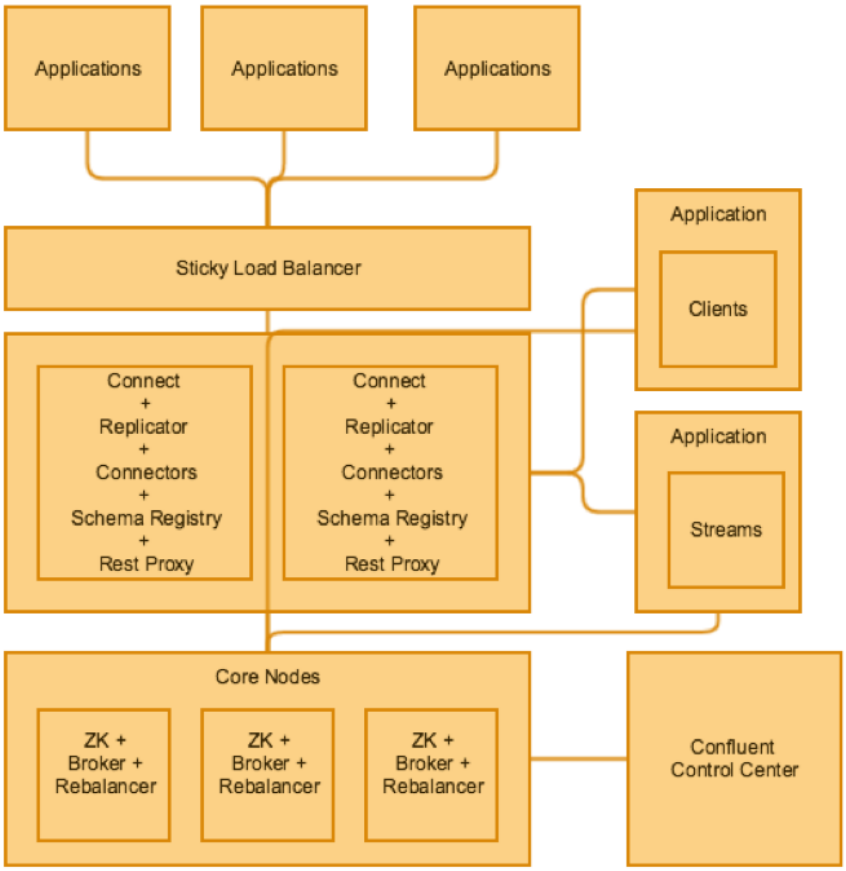
Taking all the recommendations above, a Confluent Platform cluster that is built for high-throughput long-term scalability will have an architecture similar to the following:



This architecture was built to scale. Each component is given its own servers, and if any layer becomes overly loaded it can be scaled independently simply by adding nodes to that specific layer. For example, when adding applications that use the 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 that case, you only need to add REST Proxy nodes in order to scale your entire platform.

Small Cluster Reference Architecture

Usually companies start out by adopting the Confluent Platform for one use case with limited load and when this proves successful they grow the cluster to accommodate additional applications and teams. This architecture is recommended for the early stages where investing in full-scale deployment is usually not required for the success of the project. In those cases, starting with fewer servers and installing multiple components per server is the way to go.



Note that this architecture, although requiring far fewer servers, provides the high-availability of a full scale cluster. As the use case expands, you will notice bottlenecks develop in the system. In this case the correct approach is to start by separating the bottleneck components to their own servers, and when further growth is required, scale by adding servers to each component. With time, this architecture will evolve to resemble the recommended large scale architecture.

Capacity Planning

When planning architecture for Confluent Platform, you need to provide sufficient resources for the planned workload. Storage, memory, CPU and network resources can all be potential bottlenecks and must be considered. Since every component is scalable, usage of storage, memory and CPU can be monitored on each node and additional nodes can be added when required. While most components do not store state, there is no problem to add nodes at any time and immediately take advantage of the added capacity. The main exception is Kafka brokers, which serve as the main storage component for the cluster. It is critical to monitor Kafka brokers closely and to add capacity and rebalance before any broker is overloaded - usually when any resource reaches 60-70% of capacity. The reason is that the rebalancing operation itself takes resources and the more resources you can spare for rebalancing, the less time it will take to rebalance and the sooner the cluster will have the additional capacity.

Performance can be monitored via Confluent Control Center and any 3rd party JMX monitoring tool. Confluent Control Center enables you to configure alerts when SLAs are not met, which will allow you to take action proactively.

Storage

Storage is mostly a concern on **ZooKeeper** and **Kafka brokers**.

For **ZooKeeper** the main concern is low latency writes to the transaction log. Therefore we recommend dedicated disks, specifically for storing the ZooKeeper transaction log (even in small scale deployment where ZooKeeper is installed alongside Kafka Brokers).

Kafka brokers are the main storage for the Confluent Platform cluster and therefore usually require ample storage capacity. Most deployments use 6-12 disks, usually 1TB each. The exact amount of storage you will need obviously depends on the number of topics, partitions, the rate at which applications will be writing to each topic and the retention policies you configure.

You also want to consider the type of storage. SSD and spinning magnetic drives offer different performance characteristics and depending on your use case, SSD performance benefits 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 Kafka brokers to access disk frequently, minimizing seek times will increase throughput.

When selecting a file system, we recommend either EXT4 or XFS - both have been tested and used extensively in production Kafka clusters.

Note that the use of shared-storage devices, while supported, is not recommended. Confluent Platform is not tested with SAN/NAS and very careful configuration is required in order to achieve good performance and availability when using shared storage.

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

Kafka Streams uses RocksDB as a local persistent state store. The exact use of storage depends on the specific streams application. Aggregation, windowed aggregation and windowed join all use

RocksDB stores to store their state. The size used will depend on the number of partitions, unique keys in the stream (cardinality), size of keys and values and the retention for windowed operations (specified in the DSL using until operator). Note that Kafka Streams uses quite a few file descriptors for its RocksDB stores, so make sure to increase number of file descriptors to 64K or above. Since calculating exact usage is complex, we typically allocate generous disk space to streams application to allow for ample local state. 100-300GB is a good starting point for capacity planning.

Memory

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

ZooKeeper uses the JVM heap, and 4GB RAM are typically sufficient. Too small of a heap will result in high CPU due to constant garbage collection while too large heap may result in long garbage collection pauses and loss of connectivity within ZooKeeper cluster.

Kafka brokers use both the JVM heap and the OS page cache. The JVM heap is used for replication of partitions between brokers and for log compaction. Replication requires 1MB (default replica.max.fetch.size) for each partition on the broker. In 3.1, we added a new configuration (**replica.fetch.response.max.bytes**) that limits the total RAM used for replication to 10MB, to avoid memory and garbage collection issues when the number of partitions on a broker is high. For log compaction, calculating the required memory is more complicated and we recommend referring to the Kafka documentation if you are using this feature. For small to medium sized deployments, 4GB heap size is usually sufficient. In addition, it is highly recommended that consumers always read from memory, i.e. from data that was written to Kafka and is still stored in the OS page cache. The amount of memory this requires depends on the rate at this data is written and how far behind you expect consumers to get. If you write 20GB per hour per broker and you allow brokers to fall 3 hours behind in normal scenario, you will want to reserve 60GB to the OS page cache. In cases where 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, you will want to increase the JVM heap size to 1GB or higher.

The more memory you give **Control Center** the better but we recommend at least 32GB of RAM. The JVM heap size can be fairly small (defaults to 3GB) but the application needs the additional memory for

RocksDB in-memory indexes and caches as well as OS page cache for faster access to persistent data.

Kafka Producer Clients can benefit from generous JVM heap sizes to achieve high throughput. Our clients attempt to batch data as it is sent to brokers in order to use the network more efficiently, and in addition they store messages in memory until they are acknowledged successfully by the brokers. Having sufficient memory for the producer buffers will allow the producer to keep retrying to send messages to the broker in events of network issues or leader election rather than block or throw exceptions.

Kafka Streams has several memory areas and the total memory usage will depend on your specific streams application and on the configuration. Starting with Apache Kafka 3.1 and higher there is a streams buffer cache. It defaults to 10MB and controlled through **cache.max.bytes.buffering** configuration. Setting it higher will generally result in better performance for your streams application. In addition, streams uses RocksDB memory stores for each partition involved in each aggregation, windowed aggregation and windowed-join. Kafka Streams exposes the RocksDB configuration and we recommend using the RocksDB tuning guide to size those. In addition, Kafka Streams uses a Kafka consumer for each thread you configure for your application. Each consumer allocates the lower of either 1MB per partition or 50MB per broker. Since calculating all these variables is complex and since more memory generally increases performance of streams applications, we typically allocate large amounts of memory - 32GB and above.

REST Proxy buffers data for both producers and consumers. Consumers use at least 2MB per consumer and up to 64MB in cases of large responses from brokers (typical for bursty traffic). Producers will have a buffer of 64MB each. Start by allocating 1GB RAM and add 64MB for each producer and 16MB for each consumer planned.

Note that in all cases, we recommend using the G1 garbage collection for the JVM to minimize garbage collection overhead.

CPU

Most of the Confluent Platform components are not particularly CPU bound. If you notice high CPU it is usually a result of misconfiguration, insufficient memory, or a bug.

The few exceptions are:

- **Compression**: Kafka Producers and Consumers will compress and decompress data if you configure them to do so. We recommend enabling compression since it improves network and disk utilization, but it does use more CPU on the clients. Kafka brokers older than 0.10.0 decompressed and recompressed the messages before storing them to disk, which increased CPU utilization on the brokers as well.
- **Encryption**: Starting at version 0.9.0, Kafka clients can communicate with brokers using SSL. There is a small performance overhead on both the client and the broker when using encryption, and a larger overhead when it is the consumer that connects over SSL - because the broker needs to encrypt messages before sending them to the consumer, it can't use the normal zero-copy optimization and therefore uses significantly more CPU. Large scale deployment often go to some length to make sure consumers are deployed within the same LAN as the brokers where encryption is often not a requirement.
- **High rate of client requests:** If you have 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 those cases configuring clients to batch requests will improve performance.

Note that many components are multi-threaded and will benefit more from large number of cores than from faster cores.

Network

Large scale Kafka deployments that are using 1GbE will typically become network-bound. If you are planning on scaling the cluster to over 100MB/s, you will need to provision a higher bandwidth network. When provisioning for network capacity, you will want to take into account the replication traffic and leave some overhead for rebalancing operations and bursty clients. Network is one of the resources that are most difficult to provision since adding nodes will eventually run against switch limitations, therefore consider enabling compression to getting better throughput from existing network resources. Note that 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-Premise Deployment

Large Cluster

Component	Nodes	Storage	Memory	CPU
ZooKeeper	5 is recommended for fault tolerance	Transaction log: 512GB SSD Storage: 2 X 1TB SATA, RAID 10	32GB RAM	This is typically not a bottleneck. 2-4 cores suffice.
Kafka Broker	At least 3, more for additional storage, RAM, network throughput	12 X 1TB disk. RAID 10 is optional. Separate OS disks from Kafka storage.	More is better. 64GB RAM+	Usually dual 12 core sockets.
Connect	At least 2 for HA	Other than installation, not needed.	0.5-4GB Heap Depending on connectors	Typically not CPU-bound. More cores is better than faster cores.
Schema Registry	At least 2 for HA	Other than installation, not needed.	1GB Heap Size	Typically not CPU-bound. More cores is better than faster cores.
REST Proxy	At least 2 for HA, more for additional throughput	Other than installation, not needed.	1GB overhead + 64MB per producer and 16MB per consumer	At least 16 cores to handle http requests in parallel and background threads for consumers and producers.
Control Center	1	At least 300GB, preferably SSDs.	32GB+	At least 8 cores. More preferred.

Small Cluster

Component	Nodes	Storage	Memory	CPU
ZooKeeper + Broker	At least 3	12X 1TB disks Dedicated disk for ZooKeeper Transaction log. Dedicated disk (or two) for OS REST Proxy for Kafka.	More is better. 64GB RAM+	Usually dual 12 core sockets.
Connect + Schema Registry + Rest Proxy	At least 2	Other than installation, not needed.	1 GB for connect 1GB for Schema Registry 1 GB + 64 MB per producer + 16 MB per consumer for REST Proxy	At least 16 cores
Control Center	1	At least 300GB, preferably SSDs.	32GB+	At least 8 cores.

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: Take into account that cloud providers use “virtual” cores when sizing machines. Those are typically weaker than modern cores you will use in your data center. You may need to scale the number of cores up when planning cloud deployments.
- Network: Most cloud providers only provide 10GbE on their highest tier nodes. Make sure your cluster has sufficient nodes and network capacity to provide the throughput you need after taking replication traffic into account.

Below are some examples of instance types that can be used in various cloud providers. Note that cloud offerings continuously evolve and there are typically variety of nodes with similar characteristics. As long as you adhere to the hardware recommendations, you will be in good shape. The instance types below are just examples. Remember that the default storage type for most cloud instances is ephemeral. If you expect your cloud deployment to be shut down and restarted without loss of data (say, for a development project), you'll want to configure persistent storage (EBS in Amazon, WASB in Azure, or GCS in Google).

Amazon AWS EC2

Component	Node Type	Memory	CPU	Storage	Network
ZooKeeper	m3.large	7.5GB	2 vCPU	1 x 32GB SSD	moderate
Kafka Broker	D2.xlarge	30.5GB	4 vCPU	3 x 2TB SSD	high
	r3.xlarge m4.2xlarge	30.5GB 32GB	4 vCPU 8 vCPU	Use EBS storage. We recommend configuring r3 instances as "ebs optimized"	high
Connect	c4.xlarge	7.5GB	4 vCPU	Use EBS	high
Schema Registry	m3.large	7.5GB	2 vCPU	1 x 32GB SSD	moderate
REST Proxy	c4.xlarge	30GB	16 vCPU	Use EBS	high
Control Center	m4.2xlarge	32GB	8 vCPU	Use EBS (SSD recommended)	high

Microsoft Azure

Component	Node Type	Memory	CPU	Storage	Network
ZooKeeper	Standard_DS2	7GB	2 vCPU	4 x 14GB SSD	high
Kafka Broker	Standard_DS4	28GB	8 vCPU	16 x 56GB SSD Or WASB storage	high
	Standard_D4_v2	28GB	8 vCPU	16 X 400GB Or WASB storage”	high
Connect	Standard_A3	7GB	4 vCPU	8 X 285GB	high
Schema Registry	Standard_A2	3.5GB	2 vCPU	4 x 135GB	moderate
REST Proxy	Standard_D4	28GB	8 vCPU	16 x 500GB	high
Control Center	Standard_DS4	28GB	8 vCPU	16 x 56GB SSD Or WASB storage	high

Google Cloud Compute Engine

Component	Node Type	Memory	CPU	Storage	Network
ZooKeeper	n1-standard-2	7GB	2 vCPU	Max 16 disks, up to 64TB each	
Kafka Broker	n1-highmem-4	26GB	4 vCPU	Max 16 disks, up to 64TB each	
Connect	n1-standard-4	15GB	4 vCPU	Max 16 disks, up to 64TB each	
Schema Registry	n1-standard-2	7.5GB	2 vCPU	Max 16 disks, up to 64TB each	
REST Proxy	n1-highcpu-16	14.4GB	16 vCPU	Max 16 disks, up to 64TB each	
Control Center	n1-highmem-8	52GB	8 vCPU	Max 16 disks, up to 64TB each	

Conclusion

This paper is intended to share some of our best practices around deployment of the 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 consideration such as workload characteristics, access patterns and SLAs are very important - but are too specific to cover in a general paper. To choose the right deployment strategy for specific cases, we recommend engaging with Confluent's professional services for architecture and operational review.