**Confluent Platform**

Confluent Platform is a comprehensive, enterprise-grade event streaming platform built around Apache Kafka. It extends Kafka's capabilities by adding additional tools, services, and features to create a more robust and user-friendly ecosystem for real-time data streaming and processing.

**Core Components of Confluent Platform**

**Apache Kafka**:

The core messaging and streaming component that provides the backbone for real-time data streams.

Kafka component

* Brokers: Servers that run Kafka and handle requests from producers and consumers.
* Producers: Applications that publish messages to Kafka topics.
* Consumers: Applications that subscribe to topics and read messages from them.
* Topics: Categories or channels to which messages are published.
* Partitions: Subdivisions of a topic that allow for parallelism and scalability.
* Offsets: Identifiers for messages within a partition, allowing consumers to track their position in a topic.

**Zookeeper**

A service used by Kafka for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

**Confluent Control Center**:

A graphical user interface for managing and monitoring Kafka clusters. It provides insights into the health and performance of your Kafka ecosystem, making it easier to monitor topics, brokers, and consumer groups.

**Schema Registry**:

A service for managing and enforcing schemas for Kafka messages, which ensures data compatibility and helps prevent data corruption.

**Kafka Connect**:

A framework for integrating Kafka with other systems, providing pre-built connectors for various data sources and sinks. This makes it easier to move data in and out of Kafka.

**ksqlDB (KSQL)**:

A stream processing engine that allows you to process and analyze data in Kafka using SQL-like queries. It simplifies the creation of real-time data transformations and enrichments.

**REST Proxy**:

Provides a RESTful interface to Kafka, making it easier to produce and consume messages using HTTP without needing Kafka clients.

**Kafka Streams**:

A client library for building applications and micro-services, where the input and output data are stored in Kafka clusters. It allows for complex event processing using Kafka topics.

**Replicator**:

A tool for replicating data between Kafka clusters, useful for disaster recovery, multi-datacenter deployments, and cloud migrations.

**Security and Authentication**

SASL/SSL: Simple Authentication and Security Layer/Secure Sockets Layer, protocols used for securing communication between clients and servers.

ACLs (Access Control Lists): Used to control access to topics and resources within Kafka.

**Monitoring and Management Tools**

JMX: Java Management Extensions, used for monitoring Kafka metrics.

Prometheus and Grafana: Open-source tools for monitoring and visualizing metrics collected from Kafka brokers.

**Confluent Community Version vs. Apache Kafka**

|  |  |  |
| --- | --- | --- |
| **Feature/Aspect** | **Apache Kafka** | **Confluent Community Version** |
| Core Kafka | Yes | Yes |
| Kafka Connect | Yes, but with basic connectors | Yes, with additional community connectors |
| Schema Registry | No | Yes |
| KSQL (ksqlDB) | No | Yes |
| Control Center | No | Yes, limited version |
| Pre-built Connectors | Limited to Kafka Connect plugins | More extensive library of connectors |
| Streams API | Yes | Yes |
| Monitoring and Management | Limited to JMX and third-party tools | Integrated Control Center, limited version |
| Security | Basic security (SSL, SASL) | Enhanced security options |
| Documentation and Support | Community support and documentation | Improved documentation, community support |

Drawback of Confluent Community Version vs. Apache Kafka

|  |  |  |
| --- | --- | --- |
| **Drawback Category** | **Confluent Community Version** | **Apache Kafka** |
| Dependency | Tightly coupled with Confluent products and services, which could affect future directions based on Confluent's strategies. | Less dependent on specific vendor offerings, providing more freedom in technology choices. |
| Customization | May limit customization options due to pre-configured setups and available extensions. | Offers greater flexibility for highly customized solutions, appealing to users with specific requirements. |
| Cost | Potential costs for advanced features and services, such as Confluent Cloud or premium connectors. | Free and open-source, with no hidden costs for the core software. |
| Learning Curve | Simplifies many aspects of Kafka but introduces its own set of tools and services, potentially increasing the learning curve for new users. | Straightforward and well-documented, but lacks the additional tools and services that simplify operations. |
| Operational Overhead | Reduces operational complexity with integrated tools and services, but still requires understanding of Kafka fundamentals. | Requires manual setup, configuration, and management, which can be complex and time-consuming. |
| Scalability | Simplifies scaling with additional tools, but the underlying principles remain the same as Kafka. | Highly scalable, but achieving linear scalability and managing cluster growth requires careful planning and maintenance. |
| Security and Monitoring | Offers enhanced security features and integrates with monitoring tools, reducing the complexity of setup. | Basic security and monitoring capabilities require manual setup and configuration, which can be complex. |
| Support | Provides professional support options through Confluent, catering to organizations valuing direct commercial support. | Community-driven support with extensive documentation and forums, along with commercial support options from third-party vendors. |

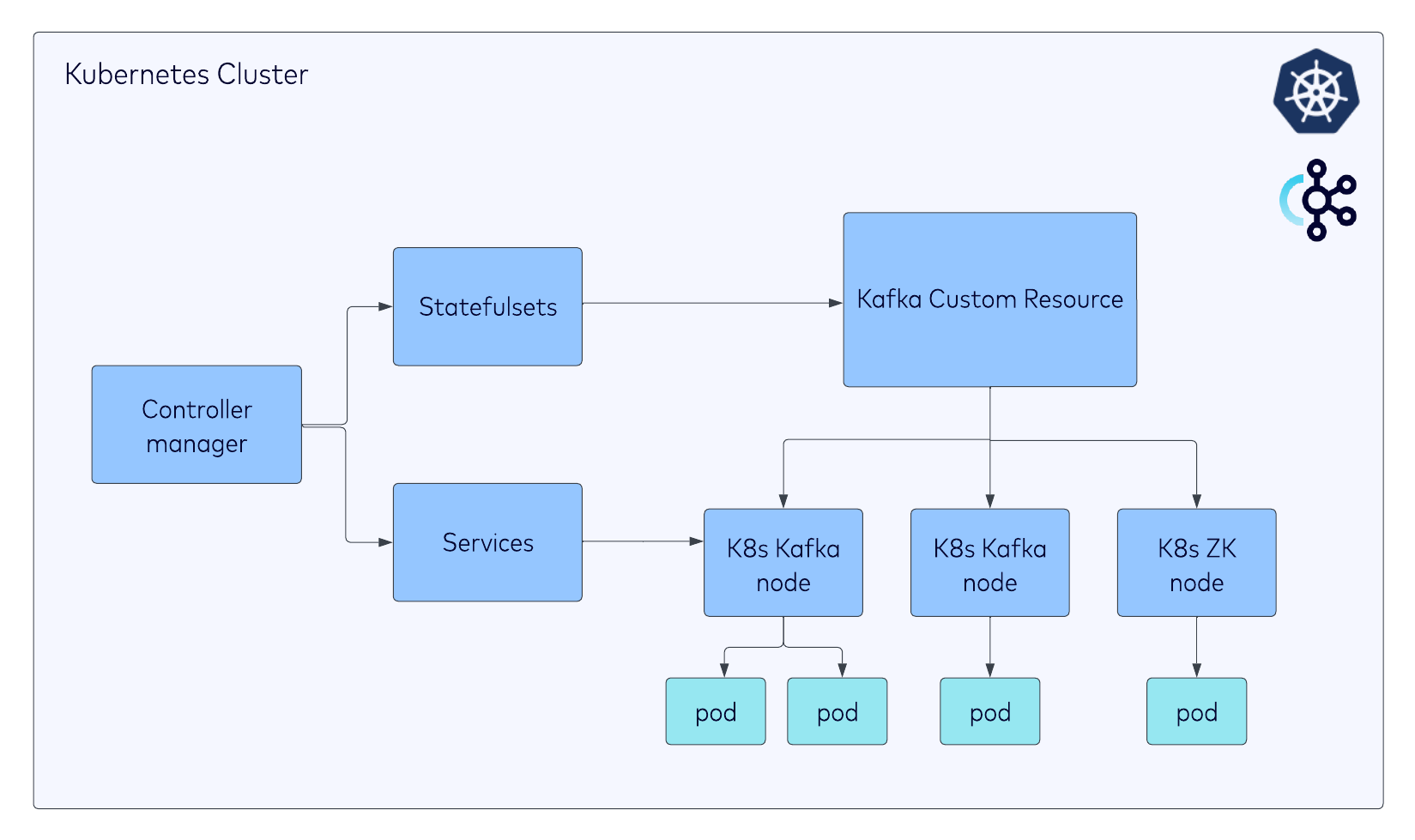
COST

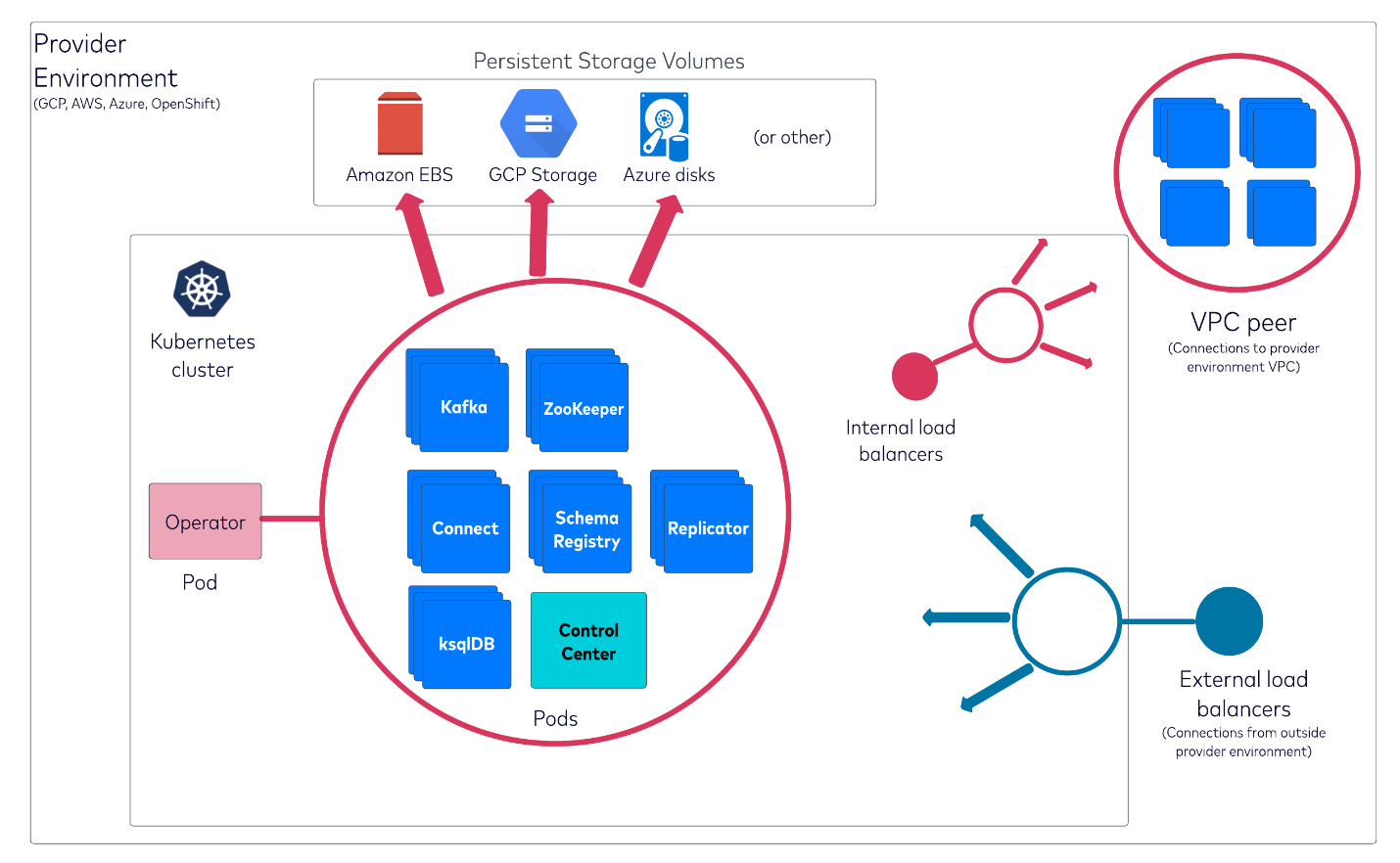
Confluent Community Version:

While free for basic use, may involve costs for advanced features, services, and professional support. However, it offers integrated tools that can reduce operational expenses.

Apache Kafka:

Free and open-source with no initial software costs, but might require third-party tools and support, leading to higher operational expenses over time.





**Installation options**

Using KRaft

Using ZooKeeper

[https://docs.confluent.io/operator/current/co-quickstart.html#co-quickstart-kraft](https://docs.confluent.io/operator/current/co-quickstart.html%23co-quickstart-kraft)

**Comparison of KRaft and ZooKeeper Modes**

|  |  |  |
| --- | --- | --- |
| **Feature** | **KRaft Mode** | **ZooKeeper Mode** |
| Metadata Management | Managed internally within Kafka brokers | Managed by external ZooKeeper ensemble |
| Consensus Protocol | Uses Raft consensus algorithm | Uses ZooKeeper's Zab protocol for consensus |
| Setup Complexity | Simpler, fewer components to manage | More complex, requires managing ZooKeeper separately |
| Operational Overhead | Lower, as there is no need for ZooKeeper | Higher, due to additional ZooKeeper management |
| Maturity and Stability | Newer, still evolving | Mature and widely used |
| Documentation and Support | Growing, but less extensive than ZooKeeper mode | Extensive and well-supported |

**Hardware Requirements:**

Nodes:

Minimum: 3 nodes (for a simple, single data center setup)

Recommended: 5 or more nodes for better fault tolerance and performance.

CPU:

Minimum: Each node should have at least 2 vCPUs.

Recommended: 4 vCPUs per node for better performance.

Memory:

Minimum: At least 8 GB RAM per node.

Recommended: 16 GB RAM per node for improved performance and to accommodate additional services like Zookeeper, Schema Registry, etc., which might run alongside Kafka.

Storage:

Minimum: Each node should have at least 100 GB SSD storage.

Recommended: 500 GB or more SSD storage per node for data retention and log storage.

Resource Requirements:

Network Bandwidth:

High network bandwidth is crucial for Kafka due to its nature of handling large volumes of data in real-time. Ensure your EKS cluster has sufficient network capacity to handle the expected traffic.

Throughput:

Minimum: Depending on your use case, plan for at least 10,000 messages per second throughput.

Recommended: Aim for 50,000+ messages per second throughput for higher loads.

Latency:

Minimum: Aim for sub-millisecond latency under normal conditions.

Recommended: Consistently achieve low latency (<1ms) for optimal performance.

Considerations:

Scalability: Start with a smaller cluster and plan to scale out as needed. Kafka is highly scalable, but initial configuration and data migration can be time-consuming.

High Availability (HA): For production environments, ensure your Kafka cluster is configured for HA. This typically means having multiple brokers across different availability zones.

Monitoring and Management: Utilize tools like Prometheus for monitoring and Grafana for visualization. Confluent Control Center can also be used for managing and monitoring your Kafka cluster.

Security: Implement proper security measures including SSL/TLS encryption for data in transit, SASL/SCRAM or LDAP for authentication, and ACLs for authorization.

Backup and Recovery: Have a robust backup strategy in place. Tools like Confluent Backup can automate this process.

Cost Optimization: Be mindful of the cost implications of running a Kafka cluster on AWS. Optimize your resource allocation and consider using reserved instances or spot instances when possible.

**Message Flow in Kafka**

Producing Messages:

A producer creates a message with data and sends it to a specific topic in Kafka.

The message can optionally have a key, which can be used to determine the partition to which the message should be sent. Without a key, the message is sent to partitions in a round-robin manner.

Kafka appends the message to the specified partition within the topic and assigns it a unique offset.

Message Storage:

Kafka stores messages in partitions. Each partition is an ordered, immutable sequence of messages.

Messages in a partition are assigned a sequential ID number called an offset.

Kafka retains messages for a configurable amount of time or until the log reaches a certain size, based on the configured retention policy.

Consuming Messages:

A consumer reads messages from a Kafka topic.

Consumers can specify a starting offset, allowing them to read messages from any point in the log.

When a consumer reads a message, Kafka does not remove it from the partition. This allows multiple consumers to read the same message independently.

Consumers in a consumer group share the load of reading from partitions. Each partition is read by exactly one consumer in the group, ensuring parallel processing.

better performance with gp2/gp3 EBS

**ERROR -1**

ERROR-> default ebs type on cluster so it will pic default storage class

SOLUTION-> now I have changed default storage class as efs-sc

After above got resource insufficient on nodes

For above issue , installation karpenter

#### <https://karpenter.sh/v0.37/getting-started/migrating-from-cas/>

Compatibility Matrix

KUBERNETES 1.24 1.30

karpenter >= 0.21 0.37.0

on which we want to setup/ run karpenter

Set node affinity

Edit the karpenter.yaml file and find the karpenter deployment affinity rules. Modify the affinity so karpenter will run on one of the existing node group nodes.

The rules should look something like this. Modify the value to match your $NODEGROUP, one node group per line.

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: karpenter.sh/nodepool

operator: DoesNotExist

- key: eks.amazonaws.com/nodegroup

operator: In

values:

- ${NODEGROUP} #############replace

Youtube <https://www.youtube.com/watch?v=dS6UIovSXpA>

<https://www.youtube.com/watch?v=sXYNmnzbEFA>

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Doc

helm template karpenter oci://public.ecr.aws/karpenter/karpenter --version "${KARPENTER\_VERSION}" --namespace "${KARPENTER\_NAMESPACE}" \

--set "settings.clusterName=${CLUSTER\_NAME}" \

--set "serviceAccount.annotations.eks\.amazonaws\.com/role-arn=arn:${AWS\_PARTITION}:iam::${AWS\_ACCOUNT\_ID}:role/KarpenterControllerRole-${CLUSTER\_NAME}" \

--set controller.resources.requests.cpu=1 \

--set controller.resources.requests.memory=1Gi \

--set controller.resources.limits.cpu=1 \

--set controller.resources.limits.memory=1Gi > karpenter.yaml

Phind

Step 1: Configure Node Affinity for Karpenter Deployment

When deploying Karpenter using Helm, you need to customize the deployment to ensure that Karpenter pods run on nodes within your On-Demand node group. This involves setting node affinity rules in the Karpenter deployment YAML generated by Helm.

1. **Generate Karpenter Deployment YAML**:

First, generate the Karpenter deployment YAML file using Helm, as you've done previously. Ensure you're pointing to the correct version of Karpenter and specifying your cluster name and namespace.

helm template karpenter oci://public.ecr.aws/karpenter/karpenter --version "${KARPENTER\_VERSION}" --namespace "${KARPENTER\_NAMESPACE}" \

--set "settings.clusterName=${CLUSTER\_NAME}" \

--set "serviceAccount.annotations.eks\.amazonaws\.com/role-arn=arn:${AWS\_PARTITION}:iam::${AWS\_ACCOUNT\_ID}:role/KarpenterControllerRole-${CLUSTER\_NAME}" \

--set controller.resources.requests.cpu=1 \

--set controller.resources.requests.memory=1Gi \

--set controller.resources.limits.cpu=1 \

--set controller.resources.limits.memory=1Gi > karpenter.yaml

Now karpenter.yaml edit this file

1. **Edit Node Affinity**:

Open the karpenter.yaml file and locate the spec.template.spec.affinity section of the Karpenter deployment. Modify it to ensure Karpenter runs on nodes within your On-Demand node group. You'll need to know the labels associated with your On-Demand node group to configure this correctly.

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: "eks.amazonaws.com/nodegroup"

operator: In

values:

- ON\_DEMAND\_NODEGROUP\_NAME # Replace with your On-Demand node group name­

Step 2: Configure Karpenter Provisioner for Spot Instances

To manage Spot Instances, you'll configure a Karpenter provisioner that targets Spot Instances specifically. This doesn't require changes to the deployment process but rather focuses on the provisioner configuration.

1. Create a Provisioner Configuration:

Define a Karpenter provisioner that specifies the use of Spot Instances. This involves creating a YAML file for the provisioner that includes constraints and preferences for Spot Instances.

apiVersion: karpenter.sh/v1alpha5

kind: Provisioner

metadata:

name: spot-provisioner

spec:

requirements:

- key: "node.kubernetes.io/instance-type"

operator: In

values: ["m5.large", "m5n.large", "m5dn.large"] # Example instance types

- key: "lifecycle"

operator: In

values: ["Ec2Spot"]

limits:

resources:

cpu: "100"

memory: 100Gi

apiVersion: karpenter.sh/v1alpha5

kind: Provisioner

metadata:

name: eks-kafka-spot-ng-manager

spec:

consolidation:

enabled: true

labels:

Name: Eks-Kafka-SpotInstanceManager

environment: staging

limits:

resources:

cpu: 1000

providerRef:

name: eks-kafka-spot-ng-manager

requirements:

- key: karpenter.sh/capacity-type

operator: In

values:

- spot

- key: kubernetes.io/arch

operator: In

values:

- amd64

- key: karpenter.k8s.aws/instance-category

operator: In

values:

- c

- m

- r

- key: karpenter.k8s.aws/instance-generation

operator: Gt

values:

- "2"

- key: kubernetes.io/os

operator: In

values:

- linux

CREATING NEW NODE GROUP  
  
eksctl create nodegroup \

--cluster=eks-kafka \

--region=ap-south-1 \

--name=ondemand-ng \

--node-type=t3.large \

--nodes=3 \

--nodes-min=3 \

--nodes-max=5 \

--node-volume-size=50 \

--node-labels="lifecycle=OnDemand"

eksctl create nodegroup \

--cluster=eks-kafka \

--region=ap-south-1 \

--name=ondemand-nodes \

--node-type=t3.xlarge \

--nodes=3 \

--nodes-min=3 \

--nodes-max=5 \

--node-volume-size=100 \

--node-labels="lifecycle=OnDemand"