

# Kafka Cost Control

## Table of Contents

1. User Manual .....	3
1.1. Introduction .....	4
1.1.1. Graphql .....	4
1.1.2. Authorization .....	4
1.2. Config samples .....	5
1.3. Configuring your cost control application .....	6
1.3.1. Configuring aggregation types .....	6
1.3.2. Configuring transformations .....	6
1.4. Pricing rules .....	8
1.4.1. Listing pricing rules .....	8
1.4.2. Setting a pricing rule .....	8
1.4.3. Removing a pricing rule .....	9
1.5. Context data .....	10
1.5.1. Listing existing context data .....	10
1.5.2. Setting context data .....	10
1.5.3. Removing context data .....	11
1.6. Reprocess .....	12
1.6.1. Using the UI .....	12
2. Installation .....	13
2.1. Prerequisites .....	14
2.2. Topics and AVRO schemas .....	15
2.2.1. Reference AVRO schemas .....	15
2.2.2. Topics .....	15
2.3. Kubernetes .....	17
2.3.1. Kafka metric scraper .....	17
2.3.2. Kafka cost control .....	17
2.3.3. Helm Chart (Strimzi only) .....	18
2.4. Metric database .....	19
2.4.1. DuckDB Integration .....	19
2.4.2. TimescaleDB Database Schema .....	20
2.5. Kafka connect .....	21
2.5.1. Configuration of the connectors .....	21
2.6. Grafana .....	23
2.6.1. Database connection .....	23
2.7. Troubleshooting .....	24
2.7.1. Kafka cost control is never ready .....	24

3. Architecture .....	25
3.1. Introduction and Goals .....	26
3.1.1. Requirements Overview .....	26
3.1.2. Quality Goals .....	27
3.1.3. Stakeholders .....	28
3.2. Architecture Constraints .....	29
3.3. System Scope and Context .....	30
3.4. Solution Strategy .....	31
3.4.1. Used Technologies .....	31
3.4.2. Time based aggregations & scraping intervals .....	31
3.5. Building Block View .....	32
3.5.1. Whitebox Overall System .....	32
3.5.2. MetricsScraper .....	32
3.5.3. PricingRules .....	33
3.5.4. ContextProvider .....	33
3.6. Runtime View .....	36
3.6.1. Metrics Ingestion from Confluent Cloud .....	36
3.6.2. Metrics using Kafka Admin API .....	36
3.6.3. Other sources of metrics .....	36
3.6.4. Metrics Enrichment .....	37
3.6.5. Metrics Grouping .....	38
3.7. Deployment View .....	39
3.8. Risks and Technical Debts .....	40
3.9. Glossary .....	41

# **1. User Manual**

# 1.1. Introduction

This user manual will help you understand kafka Cost Control and how to use it properly. This document assumes that you already have a running application. If not please see the [Installation](#) section.

At this point you should have access to the Kafka Cost Control UI and to the Grafana Dashboard.

## 1.1.1. Graphql

Kafka cost control provides a graphql endpoint at: `<your-host>/graphql-ui`

In addition, there is a ready to use GraphQL UI. You can access it by going to the following URL: `<your-host>/graphql-ui`

## 1.1.2. Authorization

TODO explain basic auth stuff TODO explain the localstorage trick

## 1.2. Config samples

If you want to quickly get started, you can create pricing rules and context data using the [config sample folder](#). All you need is node JS 20+.

Be sure to edit the files `pricing-rules.json` and `context-data.json` to match your environment.

To persist the configuration you can use the following command:

```
node index.js --url=https://<your-host>/graphql --user=admin --password=your-password
```

If you have issues you can try to add the `--verbose` options. This will display all the requests.

## 1.3. Configuring your cost control application

Your cost control instance can be configured either by setting environment variables, or (if you are using one of the provided container images) by mounting an `application.properties` file at `/deployments/config/application.properties`:

```
# inside your application.properties  
quarkus.profile=ccloud
```

### 1.3.1. Configuring aggregation types

In cost control, each metric is associated with an aggregation type. The aggregation type determines how multiple measurements within a time window are combined to produce a single value. Possible aggregation types are `SUM` and `MAX`. If no aggregation type is specified for a metric, then `SUM` will be used. To specify an aggregation type for a metric, add a line to your `application.properties` file like this:

```
# inside your application.properties  
cc.metrics.aggregations.<metric-name>=max
```

For example, to aggregate retained byte measurements using the `MAX` aggregation type, add the following line to your `application.properties` file:

```
cc.metrics.aggregations.confluent_kafka_server_retained_bytes=max
```

Note that these settings may only be set in an `application.properties` file, and not via environment variables. This is due to the fact that Quarkus cannot unambiguously map environment variable names to property names if said property names contain user-defined parts (like `confluent_kafka_server_retained_bytes` in the example above). For more information, see [the relevant section of the Quarkus documentation](#).

### 1.3.2. Configuring transformations

Currently, there is one kind of transformation that can be added to the metrics processing pipeline: `splitTopicMetricAmongPrincipals`. This transformation is configured by supplying a map of metric names to context keys in `application.properties`. For example:

```
cc.metrics.transformations.splitMetricAmongPrincipals={bytesin:'writers',bytesout:'readers'}
```

This will expect `bytesin` metrics to have a `writers` key in their context, which should be a comma-separated list of principals (e.g. applications, teams, ...) that can write to this topic. When a `bytesin` metric for any topic is encountered, the metric will be replaced with `n` metrics (where `n` is the length of the `writers` list in that metric's context). The name of the topic will be moved into that metric's

context under the `topic` key and the name of a principal from the `writers` list will move into the metric's `name` field. The value of each generated metric will be the value of the original metric divided by `n`.

In effect this takes a single metric that says "90 kb have been written to topic ABC by writers X,Y,Z" and transforms it into 3 metrics:

- Principal X wrote 30 kb to topic ABC
- Principal Y wrote 30 kb to topic ABC
- Principal Z wrote 30 kb to topic ABC

The advantage over the original metric is that each such transformed metric will be put into its own database row, which will make it easier to aggregate by principal (e.g. answer questions like "how much did principal X produce in the past month"). This type of transformation is especially relevant when doing cost control based on vanilla Kafka broker metrics (as is the case in Strimzi deployments, for example), because here only topic-level metrics are generated natively. This transformation allows to turn these topic-focused metrics into principal-focused ones.

Some metrics that are configured this way may not have a matching context key in the context. To handle such cases, cost control offers three different strategies that can be enabled by setting the `cc.metrics.transformations.config.splitMetricAmongPrincipals.missingKeyHandling` property in `application.properties`:

- **ASSIGN\_TO\_FALLBACK**: The metric will be handled as if the context key existed and contained a single principal name. This fallback principal name is set to "unknown" per default but can be changed by setting the `metrics.transformations.config.splitMetricAmongPrincipals.fallbackPrincipal` property in `application.properties`.
- **DROP**: The metric will not be forwarded downstream and will be dropped.
- **PASS\_THROUGH**: The metric will be passed downstream without any changes. This is the default behavior.

Note that `cc.metrics.transformations.config.splitMetricAmongPrincipals.missingKeyHandling` is expected to be a map of metric names to one of the above strategies. For example:

```
cc.metrics.transformations.config.splitMetricAmongPrincipals.missingKeyHandling={bytes
in:'ASSIGN_TO_FALLBACK',bytesout:'DROP'}
```

If no strategy is specified for a metric, then `ASSIGN_TO_FALLBACK` will be used.

## 1.4. Pricing rules

Pricing rules are a way to put a price on each metric. The price will be applied on the **hourly** aggregate. Also, it's common for metrics to be in **bytes** and not Megabyte or Gigabyte. Keep that in mind when setting the price. For example, if you want to have a price of **1.0\$** per **GB** you will need to set the price to  $1.0/1024^3 = 0.000976276\text{\$}$  per **byte**.

Pricing rules are stored in kafka in a compacted topic. The key should be the metric name.

### 1.4.1. Listing pricing rules

#### From the UI

Simply go to the pricing rules tab of the UI. You should see the metric name and it's cost.

#### Using Graphql

```
query getAllRules {
  pricingRules {
    creationTime
    metricName
    baseCost
    costFactor
  }
}
```

### 1.4.2. Setting a pricing rule

#### From the UI

*Not available yet.*

#### Using Graphql

```
mutation saveRule {
  savePricingRule(
    request: {metricName: "whatever", baseCost: 0.12, costFactor: 0.0001}
  ) {
    creationTime
    metricName
    baseCost
    costFactor
  }
}
```

### 1.4.3. Removing a pricing rule

#### From the UI

*Not available yet.*

#### Using Graphql

```
mutation deleteRule {  
  deletePricingRule(request: {metricName: "whatever"}) {  
    creationTime  
    metricName  
    baseCost  
    costFactor  
  }  
}
```

## 1.5. Context data

Context data are a way to attach a context (attributes basically) to a kafka item (topic, principal, ...). Basically define a set of key/values for an item that match a regex. It is possible that one item match multiple regex (and thus multiple context), but in this case you have to be careful to not have conflicting key/values.

You can have as much key/values as you want. They will be used to sum up prices in the dashboard. It is therefore important that you have at least one key/value that defined the cost unit or organization unit. For example: `organization_unit=department1`.

The context data are stored in kafka in a compacted topic. The key is free for the user to choose.

### 1.5.1. Listing existing context data

#### From the UI

Go to the tab *Context Data* in the UI. You should see all the context with their validity time, type, regex and context key/values.

If you are signed in, you can use the context tester via the button *test context data* to check what context key/values currently apply to your topic or principal.

#### Using Graphql

```
query getContextData {  
    contextData {  
        id  
        creationTime  
        validFrom  
        validUntil  
        entityType  
        regex  
        context {  
            key  
            value  
        }  
    }  
}
```

### 1.5.2. Setting context data

If you want to create a new context, you can omit the id if you want. If no id is set, the API will generate one for you using a UUID. If you use an id that is not yet in the system, this means you're creating a new context item.

## From the UI

In the Context Data tab via the button *Add context data*. You need to be signed in for the button to be visible.

## Using Graphql

```
mutation saveContextData {
  saveContextData(
    request: {id: "323b603d-5b5f-48d2-84fc-4e784e942289", entityType: TOPIC, regex: ".*collaboration", context: [{key: "app", value: "agoora"}, {key: "cost-unit", value: "spoud"}, {key: "domain", value: "collaboration"}]}
  ) {
    id
    creationTime
    entityType
    regex
    context {
      key
      value
    }
  }
}
```

### 1.5.3. Removing context data

## From the UI

*Not available yet.*

## Using Graphql

```
mutation deleteContextData {
  deleteContextData(request: {id: "323b603d-5b5f-48d2-84fc-4e784e942289"}) {
    id
    creationTime
    entityType
    regex
    context {
      key
      value
    }
  }
}
```

## 1.6. Reprocess

Reprocessing should only be used when you made a mistake, fixed it and want to reprocess the raw data. Reprocessing will induce a lag, meaning data will not be live for a little while. Depending on how much data you want to reprocess this can take minutes or hours. So be sure to know what you are doing. After the reprocessing is done, the data will be live again. Reprocessing will **NOT** lose data. They will just take a bit of time to appear live again.

Be aware that in the reprocessing action may take a while to complete (usually about 1 min). This is why you should be patient with the request.

The process is as follows:

- use request reprocessing
- KafkaCostControl MetricProcess kafka stream application will stop
- Wait for all consumers to stop and for kafka to release the consumer group (this may take time)
- KafkaCostControl will look for the offset of the timestamp requested for the reprocessing (if not timestamp requested, it will just see to zero)
- KafkaCostControl will self-destruct in order for kubernetes to restart it (you may see a restart count increasing)
- KafkaCostControl kafka stream application will resume from the offset defined by the timestamp you gave

The metric database should be independent. This means it should be able to accept updates. Otherwise, you will need to clean the database yourself before a reprocessing.

### 1.6.1. Using the UI

- Go to the *Others* tab.
- Choose a date for the start time of the reprocessing (empty means from the beginning of time). You can help yourself with the quick button on top.
- Click on reprocess
- Confirm the reprocessing

### Using Graphql

```
mutation reprocess {  
  reprocess(areYouSure: "no", startTime:"2024-01-01T00:00:00Z")  
}
```

## 2. Installation

## 2.1. Prerequisites

This installation manual assumes that

1. You have a Kafka cluster
2. You have a schema registry
3. You have a Kubernetes cluster

## 2.2. Topics and AVRO schemas

Kafka cost control uses internal topic to compute pricing. You will have to create those topic before deploying the application. The documentation will show the default names, you can change them but don't forget to adapt the aggregator configuration.

### 2.2.1. Reference AVRO schemas

Some schemas will reference [EntityType](#). Please add it to your schema registry and reference it when needed.

### 2.2.2. Topics

Topic name	Clean up policy	Key	Value
context-data	compact	<i>String</i>	<a href="#">ContextData</a>
pricing-rules	compact	<i>String</i>	<a href="#">PricingRule</a>
aggregated	delete	<a href="#">AggregatedDataKey</a>	<a href="#">AggregatedDataWindowed</a>
aggregated-table-friendly	delete	<a href="#">AggregatedDataKey</a>	<a href="#">AggregatedDataTableFriendly</a>
metrics-raw-telegraf-dev	delete	<i>None</i>	<i>String</i>

#### Context data

This topic will contain the additional information you wish to attach to the metrics. SEE TODO for more information. This topic is compacted and it is important that you take care of the key yourself. If you wish to delete a context-data you can set *null* as payload (and provide the key you want to delete).

#### Pricing rule

This topic will contain the price of each metric. Be aware that most of the metric will be in *bytes*. So if you want for example to have a price of *1.0\$* per *GB* you will need to set the price to  $1.0/1024^3 = 0.000976276\$$  per *byte*. The key should be the metric name. If you wish to remove a price value, send the payload *null* with the key you want to delete. See TODO on how to use the API or the UI to set the price.

#### Aggregated

This topic will contain the enriched data. This is the result topic of the aggregator.

#### Aggregated table friendly

This is the exact same thing as *aggregated* except there are no hashmap and other nested field. Everything has been flattened. This topic makes it easy to use Kafka Connect with a table database.

## Metrics raw telegraf

You can have multiple raw topics. For example one per environment or one per kafka cluster. The topic name is up to you, just don't forget to configure it properly when you deploy telegraf (see Kubernetes section). Give some special consideration to the `retention.ms` setting for the raw metrics topics. For example, if you want to distribute the cost of your monthly bill based on the raw metrics scraped over the course of the month then it is a good idea to retain the scraped data for more than 30 days. This gives people time to ask questions about their bill and also gives the opportunity to reprocess the metrics with new pricing rules/context if needed.

## 2.3. Kubernetes

You can find all the deployment files in the [deployment](#) folder. This folder use [Kustomize](#) to simplify the deployment of multiple instances with some variations.

The kubernetes deployment is in two parts. One part is the **kafka control software** (processing, ui, dashboard, etc.) and the other part is the **kafka metric scrapper**. You may have multiple kafka metric scrapper deployment (one per kafka cluster for example), but you should need only one kafka cost control deployment.

### 2.3.1. Kafka metric scrapper

This part will be responsible to scrape kafka for relevant metrics. Depending on what metrics you want to provide you will need a user with read access to kafka metric but also kafka admin client. Read permission is enough ! You don't need a user with write permission.

This documentation will assume that you use the [dev/](#) folder, but you can configure as much Kustomize folders as you want. The [dev/](#) folder is a good starting point if you have confluent cluster running.

Copy the environment sample file:

```
cd deployment/kafka-metric-scrapper/dev  
cp .env.sample .env  
vi .env
```

Edit the environment file with the correct output topic, endpoints and credentials.

Be sure to edit the **namespace** in the [kustomization.yaml](#) file.

Deploy the dev environment using kubectl

```
cd /deployment/kafka-metric-scrapper  
kubectl apply -k dev
```

Wait for the deployment to finish and check the output topic for metrics. You should receive new data every minute.

### 2.3.2. Kafka cost control

For this part we will deploy the kafka stream application that is responsible to enrich the metrics, TimescaleDB for storing the metrics, kafka connect instance to sink the metric into the database, a grafana dashboard and a simple UI to define prices and contexts.

This documentation will assume that you use the [dev/](#) folder, but you can configure as much Kustomize folders as you want.

Copy the environment sample file:

```
cd deployment/kafka-cost-control/app  
cp .env.sample .env  
vi .env
```

Edit the environment file with the correct credentials. The database password can be randomly generated. It will be used by kafka connect and grafana.

Be sure to edit the **namespace** in the [kustomization.yaml](#) file.

You also may want to adapt the ingress files to use a proper hosts. You will need two hosts, one for grafana and one for the kafka cost control application.

Deploy the application using kubectl

```
cd /deployment/kafka-metric-scrapper  
kubectl apply -k app
```

### 2.3.3. Helm Chart (Strimzi only)

If your Kafka cluster is managed by the Strimzi operators, you can use the provided Helm chart (under [helm/kcc-strimzi](#)) to deploy Cost Control with all the required components (Telegraf, Aggregator, UI, TimescaleDB, Kafka Connect) as well as all the required users and topics. The Helm chart still assumes that you already have a Schema Registry deployed. See the relevant README file for more information.

A component that is special to the Strimzi deployment is the Context Operator. This operator detects [KafkaUser](#) resources with a specific annotation and automatically creates a corresponding [Context](#) in Cost Control. This allows you to manage your contexts directly from Kubernetes. See the README in [strimzi-operator](#) for more information.

## 2.4. Metric database

For storing the metrics, we recommend using either an external time series database or, alternatively, the built-in DuckDB integration.

The advantage of the DuckDB integration is that it is easy to set up and does not require any additional downstream components (e.g. Kafka Connect and a time series database). However, the DuckDB integration is not as scalable as a dedicated external database. It currently also only supports downloading an export of the metrics for some time period. These exports can be downloaded in JSON and CSV format for offline analysis (e.g. in a Jupyter notebook). Note that when the DuckDB integration is used, a stream of aggregated metrics is still sent to Kafka, so that it is possible to switch to using an external time series database at any time.

The time series database approach is more scalable and allows leveraging the full power of the database for querying and analyzing the metrics. However, it also requires additional components to be set up, such as Kafka Connect and the database itself. Feel free to choose the database that suits your needs, but be careful to choose one that is compatible with Kafka connect (or is otherwise capable of receiving data from Kafka) so you can easily transfer metrics from Kafka to your database. In this example we will assume that you're using TimescaleDB because it's the one we provide Kubernetes manifest for.

In the following sections we will show you how to set up both the DuckDB integration and TimescaleDB. Feel free to skip the section that does not apply to your use case.

### 2.4.1. DuckDB Integration

The DuckDB integration is disabled by default. To enable it, set the following environment variables in your Aggregator deployment:

```
containers:
  - name: kafka-cost-control
    image: spoud/kafka-cost-control:latest
    env:
      # enable the DuckDB integration
      - name: CC_OLAP_ENABLED
        value: "true"
      # path in the container where the DuckDB database will be stored
      # if this is not set, the data will be stored in-memory (i.e. it will be
      # lost when the container is restarted)
      - name: CC_OLAP_DATABASE_URL
        value: "jdbc:duckdb:/home/jboss/kafka-stream/duckdb.db"
```

Once the integration is enabled, you can export collected metrics from the DuckDB database by using curl:

```
# Get all metrics from the last 30 days in CSV format
curl -H "Accept: text/csv" http://localhost:8083/olap/export > out.csv
# Get all metrics from the last 30 days in JSON format
```

```

curl -H "Accept: application/json" http://localhost:8083/olap/export > out.json
# Get all metrics for all of February 2025 in CSV format
curl -H "Accept: text/csv" "http://localhost:8083/olap/export?fromDate=2025-02-
01T00:00:00Z&toDate=2025-03-01T00:00:00Z" > out.csv

```

Note that when using the `fromDate` and `toDate` parameters, the times must be specified in UTC in the ISO 8601 format (e.g. `2025-02-01T00:00:00Z`). Otherwise the API will return a 400 Bad Request error.

## 2.4.2. TimescaleDB Database Schema

Feel free to adapt the partition size to fit your needs. In this example we put 7 days but please follow the [TimescaleDB documentation](#) to choose the right partition size for your use case. We recommend a value between 7 days and 1 month. Note that the Helm and Kustomize templates already execute this script when the database is created.

```

CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;
CREATE TABLE "kafka_aggregated-table-friendly"
(
    "startTime"          TIMESTAMP      NOT NULL,
    "endTime"            TIMESTAMP      NOT NULL,
    "entityType"         VARCHAR        NOT NULL,
    "initialMetricName" VARCHAR        NOT NULL,
    "name"               VARCHAR        NOT NULL,
    "value"              DOUBLE PRECISION NOT NULL,
    "cost"               DOUBLE PRECISION NULL,
    "tags"               JSONB          NOT NULL,
    "context"            JSONB          NOT NULL,
    PRIMARY KEY ("startTime", "endTime", "entityType", "initialMetricName", "name")
);
SELECT create_hypertable('kafka_aggregated-table-friendly', by_range('startTime',
INTERVAL '7 day'));

```

To prevent the database from being overwhelmed by the amount of data, we recommend creating a retention policy. In this example we will keep the data for 2 years:

```
SELECT add_retention_policy('kafka_aggregated-table-friendly', INTERVAL '2 years');
```

if you want to run the scripts above manually, you can use the interactive cli.

```
kubectl exec -it -n <namespace> timescaledb-0 -- psql -U postgres -d postgres
```

## 2.5. Kafka connect

To write data from the kafka metric topic to the timeserie database we will use Kafka Connect.

Please refer to the kubernetes manifest to deploy a kafka connect cluster.

### 2.5.1. Configuration of the connectors

Don't forget to adapt the hosts, users and password

```
{  
  "name": "kafka-cost-control-aggregated",  
  "config": {  
  
    "tasks.max": "1",  
    "topics": "aggregated-table-friendly",  
    "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",  
    "connection.url": "jdbc:postgresql://timescaledb-  
service:5432/postgres?sslmode=disable",  
    "connection.user": "postgres",  
    "connection.password": "password",  
    "insert.mode": "upsert",  
    "auto.create": "false",  
    "table.name.format": "kafka_${topic}",  
    "pk.mode": "record_value",  
    "pk.fields": "startTime,endTime,entityType,initialMetricName,name",  
    "key.converter": "org.apache.kafka.connect.storage.StringConverter",  
    "value.converter": "io.confluent.connect.avro.AvroConverter",  
    "value.converter.schema.registry.url": "https://schema-registry-host",  
    "value.converter.basic.auth.credentials.source": "USER_INFO",  
    "value.converter.basic.auth.user.info": "schema-registry-user:schema-registry-  
password",  
    "transforms": "flatten",  
    "transforms.flatten.type": "org.apache.kafka.connect.transforms.Flatten$Value",  
    "transforms.flatten.delimiter": "_"  
  }  
}
```

To configure the connector, you can simply create a kubernetes tunnel to the running kafka connect service.

```
kubectl -n <namespace> port-forward service/kafka-connect-service 8083:8083
```

Then in another terminal you can use the curl command to create the connector. Don't forget to edit the different users and passwords required (kafka, schema registry and database)

```
curl -X POST -H "Content-Type: application/json" --data @kafka-connect-config.json
```

```
http://localhost:8083/connectors
```

You can check the status of the connectors with the following command:

```
curl -X GET http://localhost:8083/connectors/kafka-cost-control-aggregated/status | jq
```

The status should be running

## 2.6. Grafana

Go to your grafana dashboard (you should have configured the host in the deployment). The default credentials are `admin:admin`. You will be asked to create a new password.

### 2.6.1. Database connection

Go to the administration page and search for the plugin called [PostgreSQL](#). It should normally be already installed, if not install it. You can then click on [add new data source](#).

Field	Value	Info
Name	grafana-postgresql-datasource	<i>should be the default</i>
Host URL	timescaledb-service:5432	<i>Kubernetes service for the database</i>
Database Name	postgres	<i>this is the default if you didn't change it</i>
Username	postgres	<i>this is the default if you didn't change it</i>
Password	postgres	<i>this is the password you created in the .env file of the deployment</i>
TLS/SSL Mode	disabled	<i>unless you configured it</i>
TimescaleDB	on	<i>this will improve performance</i>

You can keep the rest of the value as default.

TODO import dashboard

## 2.7. Troubleshooting

### 2.7.1. Kafka cost control is never ready

If the kafka-cost-control pod is never ready there are good chances that it is waiting on a topic before it can start. If you look closely in the log you will see a message like this:

```
2024-02-06 15:46:34,739 WARN [io.qua.kaf.str.run.KafkaStreamsProducer] (pool-5-thread-1) Waiting for topic(s) to be created: [non-existing-topic]
```

As soon as you create the missing topic(s), you should be good to go. Look again at the [Topics](#) section for more information on how to create a topic.

### **3. Architecture**

## 3.1. Introduction and Goals

Many organizations have introduced Kafka either on premise or in the cloud in recent years. Kafka platforms are often used as a shared service for multiple teams. Having all costs centralized in a single cost center means that there is no incentive to save costs for individual users or projects.

Kafka Cost Control gives organizations transparency into the costs caused by applications and allow to distribute platform costs in a fair way to its users by providing a solution that

- shows usage statistics per application and organizational unit
- allows defining rules for platform cost distribution over organizational units or applications
- works for most organizations, no matter if they use Confluent Cloud, Kubernetes or on-prem installations

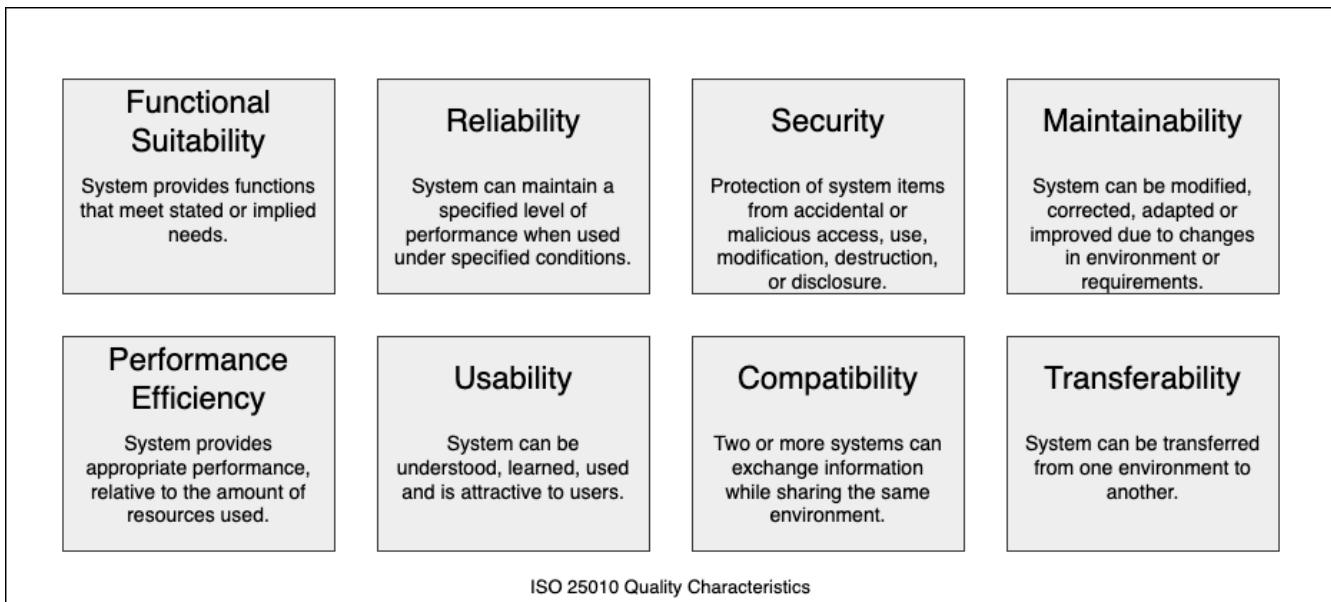
### 3.1.1. Requirements Overview

1. Collection and aggregation of usage metrics and statistics from one or multiple Kafka clusters.  
Aggregation by time:
  - hourly (for debugging or as a metric to understand costs in near real-time)
  - daily
  - weekly
  - monthly
2. Management of associations between client applications, projects and organizational units (OU)
  - automatic recognition of running consumer groups
  - automatic detection of principals/clients
  - creation, modification and deletion of contexts (projects and OUs)
  - interface to hook in custom logic for automatic assignment of clients to projects and OUs
  - manual assignment of auto-detected principals or consumer groups to projects and OUs
  - context can change in time, each item should have a start and end date (optional). This means that an item (ex a topic) can switch ownership at any point in time
3. Visualization of usage statistics
  - Costs and usage statistics can be broken down interactively
    - Summary view: total costs for timespan (day, week, month) per OU
    - Detail View OU by category: costs by category (produce, consume, storage) for the selected OU in the selected timespan
    - Detail View OU by application/principal/consumer-group/topic
  - Data must be made available in a format that can be used to display it with standard software (e.g. Kibana, Grafana, PowerBI), so that organizations can integrate it into an existing application landscape
  - provisioning of a lightweight default dashboard e.g. as a simple SPA, so that extra tooling is

- not mandatory to view the cost breakdown
  - Items not yet classified should be easily identifiable, so we know what configuration is missing (for example a topic has no OU yet)
4. Management of rules, that describe how costs are calculated (aka pricing rules)
  5. Management of rules, that describe how costs are calculated, e.g.
    - fixed rates for available metrics, i.e. CHF 0.15 per consumed GB
    - base charge, i.e. CHF 0.5 per principal per hour
    - rules can be changed at any time, but take effect at a specified start time
    - optional: backtesting of rules using historical data
  6. Access Control
    - only authorized users can modify rules, OUs and projects
    - unauthenticated users should be able to see statistics
  7. Observability
    - expose metrics so that the cost control app can be monitored
    - proper logging
  8. Export of end-of-month reports as CSV or Excel for further manual processing
  9. Ability to reprocess raw data in case a mistake was made. For example we see at the end of the month that an item was wrongly attributed to an OU. We should be able to correct this and reprocess the data.

### 3.1.2. Quality Goals

1. **Transferability / Extensibility:** Kafka Cost Control should be modular, so that company-specific extensions can be added.  
A core layer should contain common base functionality. Company specific terms or features should be separated into dedicated modules.
2. **Maintainability:** Reacting to changing requirements and implementing bug fixes should be possible within weeks.



### 3.1.3. Stakeholders

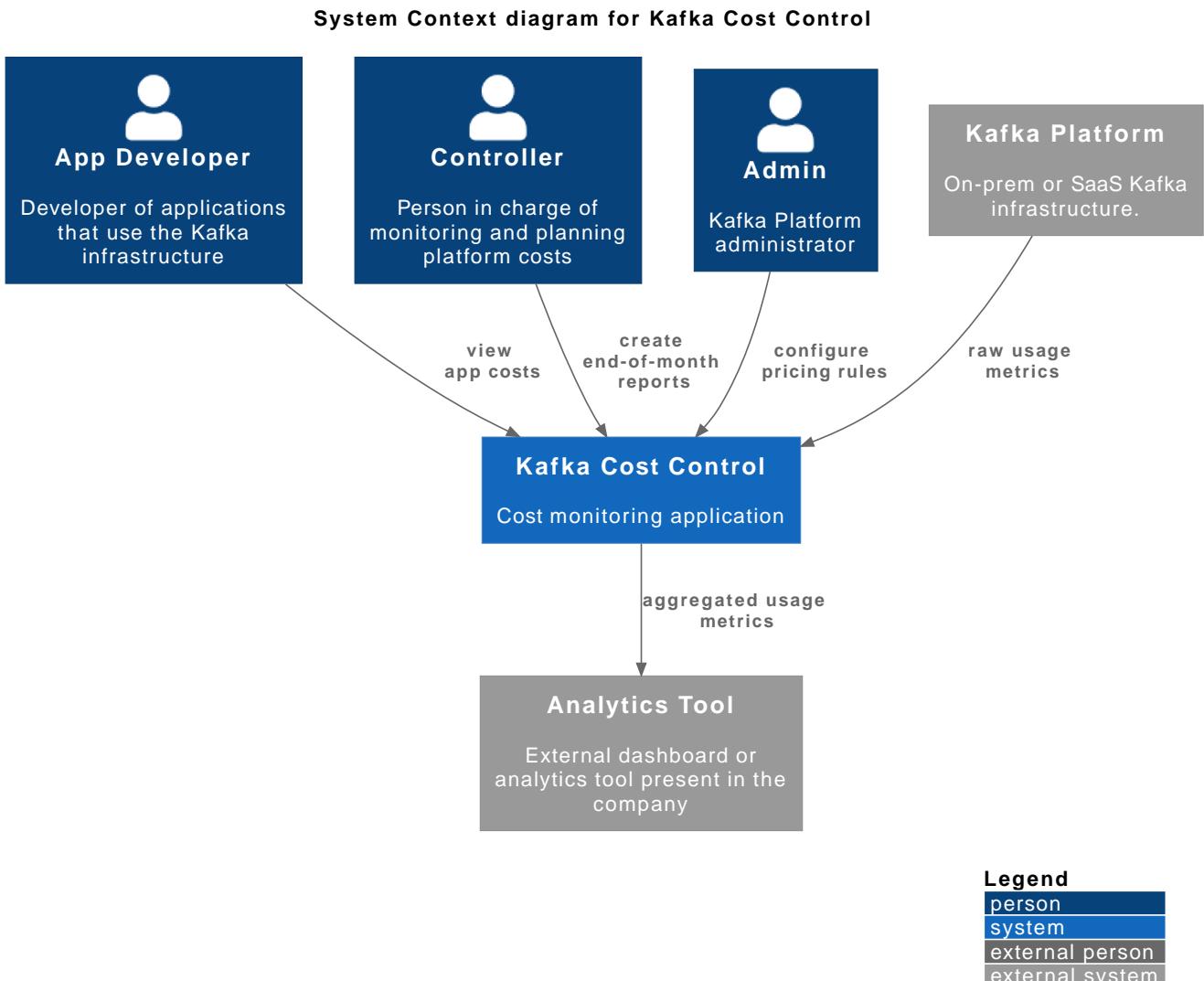
Role/Name	Expectations
<i>Kafka user</i>	Should be able to see their usage. Should take ownership of resources.
<i>Management</i>	Should have an overview of the costs and usage of Kafka.

## 3.2. Architecture Constraints

Constraint	Explanation
JVM based	use common language at SPOUD and many clients to make sure many can contribute
Hosting On-Site (not SaaS only)	Companies may not want to expose usage data to a SaaS provider

### 3.3. System Scope and Context

Kafka Cost Control is a standalone application that needs to integrate into an existing IT landscape.



## 3.4. Solution Strategy

### 3.4.1. Used Technologies

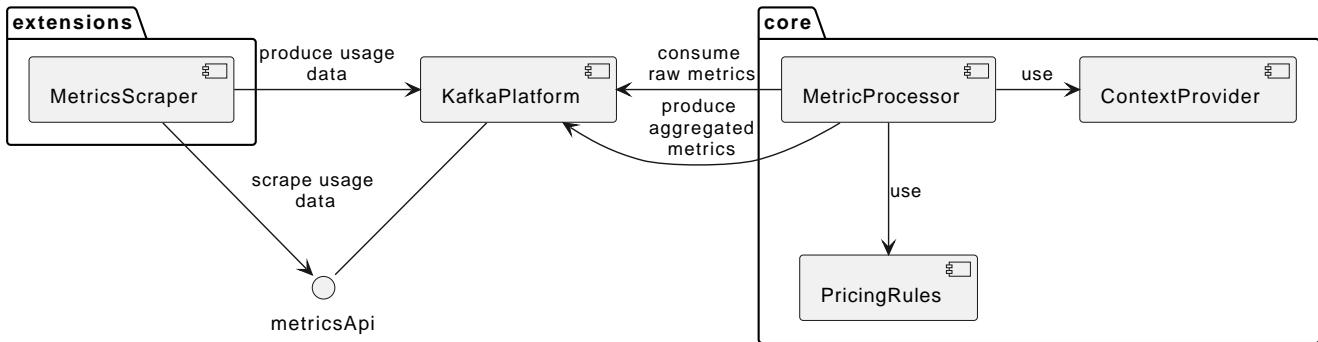
Technology	Reason
Telegraf	<ul style="list-style-type: none"><li>Used for scraping metrics from data sources like Prometheus agents or Confluent Cloud API.</li><li>Versatile and lightweight tool that can be run in all environments.</li><li>supports Kafka</li></ul>
Kafka	for storing metrics, context info and pricing rules, reduces number of solution dependencies
Kafka Streams	for enriching metrics and storing pricing + context data into KTables
DataStore	A datastore, e.g. a SQL DB, will be used for the time based aggregations (e.g. end of month reporting). Avoids complex calendar logic in Kafka Streams.

### 3.4.2. Time based aggregations & scraping intervals

- **MetricsScraper** should ingest metrics with an interval of 1 minute for confluent cloud metrics. Other data sources can have longer intervals.
- **MetricsProcessor** aggregates metrics with short time windows of 60 minutes
  - variable cost is usually defined as *cost unit/minute*
  - The window value is the accumulated cost for one hour (interpolation may be needed when data points are missing)
  - this allows some tolerance for gaps in metrics and varying ingestion intervals

## 3.5. Building Block View

### 3.5.1. Whitebox Overall System



Building block	Description
PricingRules	Stores rules for turning usage information into costs
ContextProvider	Manages contextual information that can be used to enrich metrics with company-specific information. E.g. relations between clientIds, applications, projects, cost centers, ...
MetricProcessor	<ul style="list-style-type: none"><li>Defines interfaces for metrics, that must be used by <i>MetricsScraper</i></li><li>Aggregates metrics into time buckets</li><li>Produces enriched data streams which includes contextual information</li></ul>
MetricsScraper	<ul style="list-style-type: none"><li>uses a metric source, such as JMX or the confluent cloud metrics API to collect usage metrics</li><li>transforms the collected metrics into a format that is defined by <i>MetricProcessor</i></li></ul>

### 3.5.2. MetricsScraper

#### Confluent Cloud

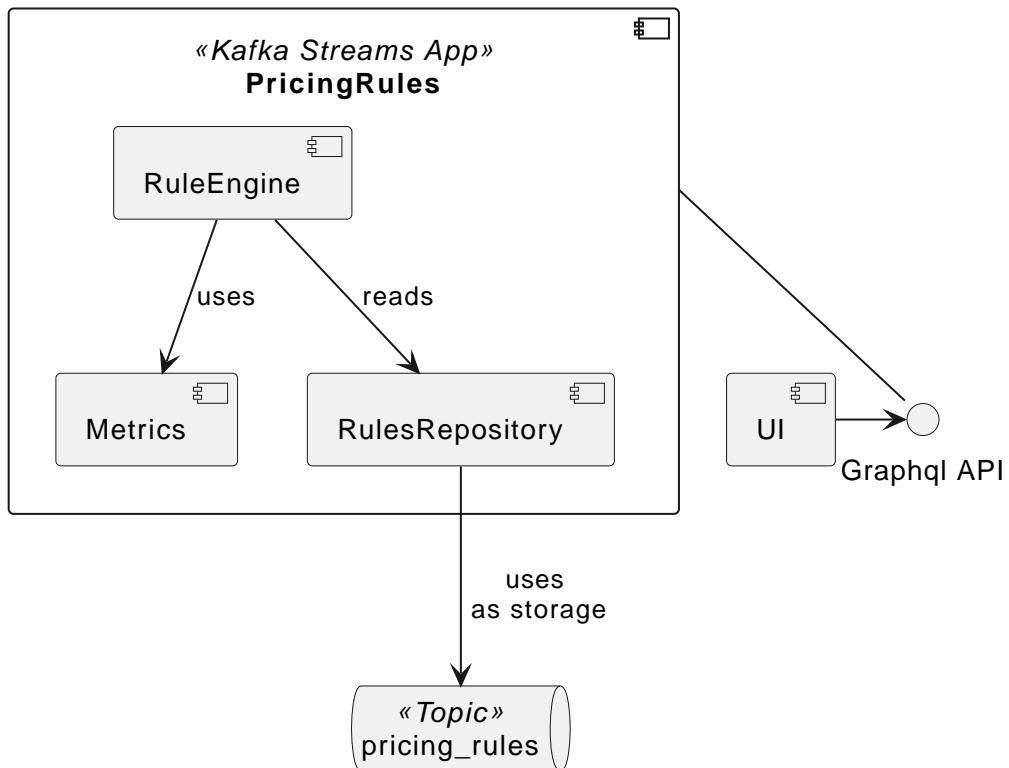
Confluent exposes many metrics in prometheus format. These will be scraped with telegraf. Some information are missing from the prometheus export endpoint and need to be fetched with custom queries/requests. This is done with a java application which exposes them as prometheus endpoint. Docs: <https://docs.confluent.io/cloud/current/monitoring/metrics-api.html>

Additional metric	endpoint/query
Partition count of a topic	<b>AdminClient</b> to list all topics, partition count and replication factor

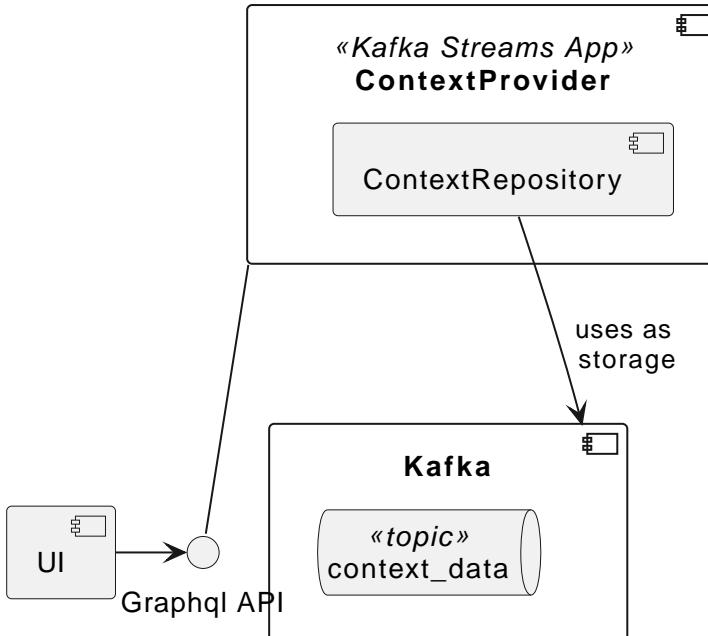
registered schemas for a topic	<p>http requests to schema registry needed.</p> <ol style="list-style-type: none"> <li>1. <b>GET</b> <code>\{{schema_registry_url}}/schemas/</code></li> <li>2. Group by topic and create gauge metric for schema count by topic</li> </ol> <p><b>WARNING:</b> Does not account for soft deleted topics in confluent cloud! Possible sanity check: sum of all schemas should equal total number of schemas reported by <a href="https://api.telemetry.confluent.cloud/v2/metrics/cloud/descriptors/metrics?resource_type=schema_registry">https://api.telemetry.confluent.cloud/v2/metrics/cloud/descriptors/metrics?resource_type=schema_registry</a> If the sum is smaller, then soft deletes were made (using the max version number might yield a better result in this case).</p> <p><b>WARNING:</b> If a non default naming strategy is used for subjects, then linking schemas/subjects to topics is not possible for all schemas.</p>
--------------------------------	---

[] | /github/workspace/images/.svg

### 3.5.3. PricingRules



### 3.5.4. ContextProvider



## Context format

- metrics are defined in the core
- a metric belongs to at least one of the dimensions
  - topic
  - consumer group
  - principal
- a context object can be attached to existing dimensions as a AVRO key-value pair to provide the needed flexibility

*topic context as JSON record in a topic, record key="car-claims"*

```
{
  "creationTime": "2024-01-01T00:00:00Z",
  "validFrom": "2024-01-01T00:00:00Z",
  "validUntil": null,
  "entityType": "TOPIC",
  "regex": "car-claims",
  "context": {
    "project": "claims-processing",
    "organization_unit": "non-life-insurance",
    "sap_psp_element": "1234.234.abc"
  }
}
```

*topic context rule as JSON record in a topic, record key="default-rule-since-2020"*

```
{
  "creationTime": "2024-01-01T00:00:00Z",
  "validFrom": "2024-01-01T00:00:00Z",
```

```

    "validUntil": null,
    "entityType": "TOPIC",
    "regex": "^([a-zA-Z0-9-]+)\\.([a-zA-Z0-9-]+)\\.([a-zA-Z0-9-]+)-.*$",
    "context": {
        "tenant": "$1",
        "app_id": "$2",
        "component_id": "$3"
    }
}

```

If naming conventions are very clear they could also be provided as a file / configuration.

*principal context as JSON record in a topic, record key="cluster-id-principal-default-ctxt"*

```

{
    "creationTime": "2024-01-01T00:00:00Z",
    "validFrom": "2024-01-01T00:00:00Z",
    "validUntil": null,
    "entityType": "PRINCIPAL",
    "regex": "u-4j9my2",
    "context": {
        "project": "claims-processing",
        "organization_unit": "non-life-insurance",
        "sap_psp_element": "1234.234.abc"
    }
}

```

## INFO

Context objects will be started as AVRO messages. We use JSON as a representation here for simplicity.

## Context Lookup

State stores in Kafka Streams will be used to construct lookup tables for the context.

The key is a string and is a free value that can be set by the user. If no key is provided the API should create random unique key. The topic is compacted, meaning if we want to delete an item we can send a null payload with its key.

*Table 1. context lookup table*

Key	Value
<type>_<cluster-id>_<principal_id>	<context-object>
PRINCIPAL_lx1dfsg_u-4j9my2_2024-01-01	{..., "regex": "u-4j9my2", "context": {...}}
b0bd9c9a-08e6-46c7-9f71-9eafe370da6c	<context-object>

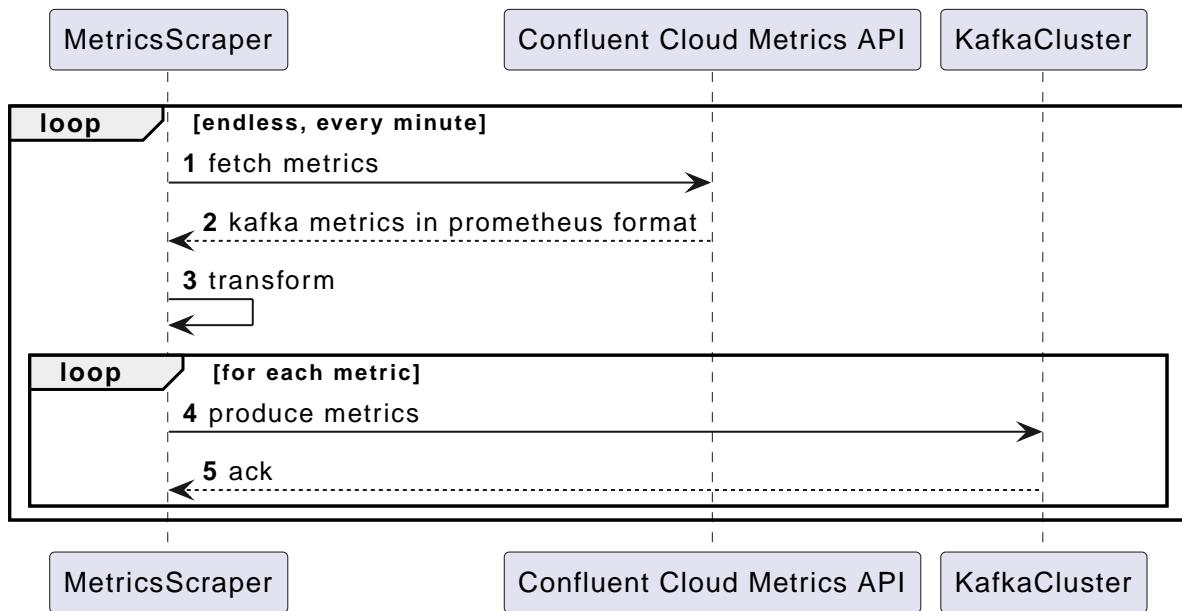
Once the table has been loaded, aggregated metrics can be enriched with a KTable - Streams join.

## 3.6. Runtime View

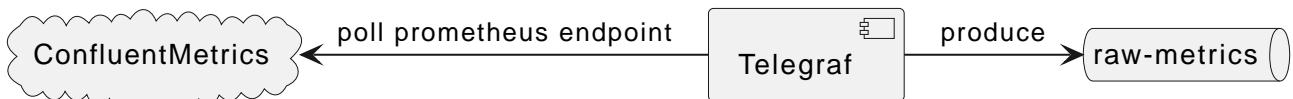
### 3.6.1. Metrics Ingestion from Confluent Cloud

Process to gather and aggregate metrics from Confluent Cloud.

The Confluent Metrics Scraper calls the endpoint `api.telemetry.confluent.cloud/v2/metrics/cloud/export?resource.kafka.id={CLUSTER-ID}` with Basic Auth in an interval of 1 Minute to obtain all metrics in Prometheus format.

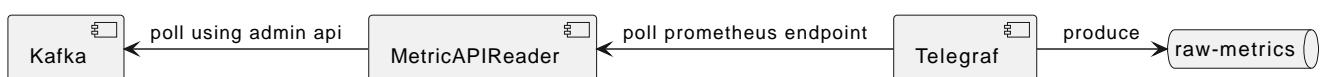


Telegraf is used to poll data using Confluent prometheus endpoint.



### 3.6.2. Metrics using Kafka Admin API

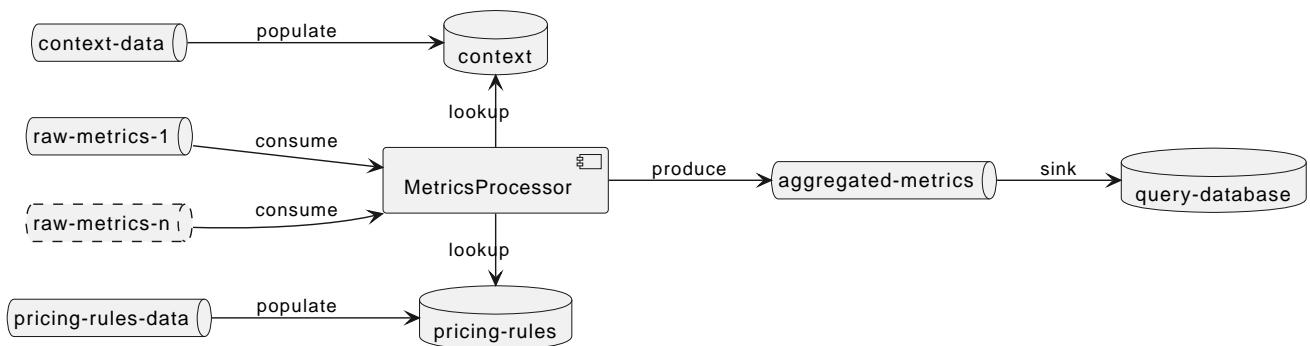
Some information can be gathered from the Kafka Admin API. We will develop a simple application that connects to the Kafka Admin API and exposes metrics as Prometheus endpoint. We can then reuse Telegraf to publish those metrics to kafka.



### 3.6.3. Other sources of metrics

Anyone can publish to the raw metrics topic. The metrics should follow the Telegraf format. Recommendation: use one topic per source of metrics. The MetricEnricher application will anyway consume multiple raw metric topics.

### 3.6.4. Metrics Enrichment



1. Metrics are consumed from all the raw data topics.
2. Metrics are aggregated by the MetricsProcessor. Here we:
  - aggregate by hours
  - attach context
  - attach pricing rule
3. The aggregates are stored in the **aggregated-metrics** topic.
4. The aggregated metrics are stored into the query database.

The storage procedure into the query database must be idempotent in order to reprocess the enrichment in case of reprocessing.

#### Enrichment for topics

*metric with topic name from confluent cloud*

```
{  
  "fields": {  
    "gauge": 40920  
  },  
  "name": "confluent_kafka_server_sent_bytes",  
  "tags": {  
    "env": "sdm",  
    "host": "confluent.cloud",  
    "kafka_id": "lkc-x5zqx",  
    "topic": "mobiliar-agoora-state-global",  
    "url": "https://api.telemetry.confluent.cloud/v2/metrics/cloud/export?resource.kafka.id=lkc-x5zqx"  
  },  
  "timestamp": 1704805140  
}
```

#### Enrichment for principals

*metric with principal id from confluent cloud*

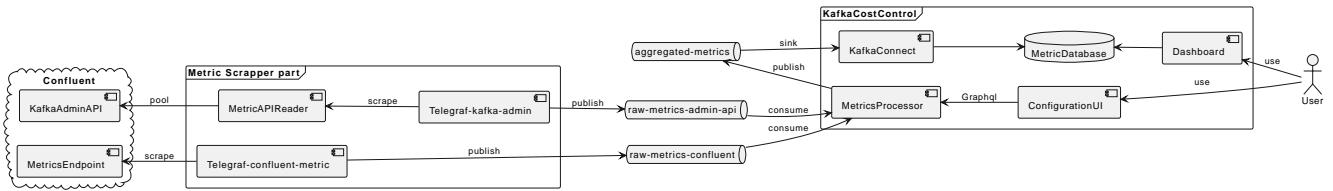
```
{  
  "fields": {  
    "gauge": 0  
  },  
  "name": "confluent_kafka_server_request_bytes",  
  "tags": {  
    "env": "sdm",  
    "host": "confluent.cloud",  
    "kafka_id": "lkc-x5zqx",  
    "principal_id": "u-4j9my2",  
    "type": "ApiVersions",  
    "url": "https://api.telemetry.confluent.cloud/v2/metrics/cloud/export?resource.kafka.id=lkc-x5zqx"  
  },  
  "timestamp": 1704805200  
}
```

### 3.6.5. Metrics Grouping

- confluent\_kafka\_server\_request\_bytes by kafka\_id (Cluster) and principal\_id (User) for the type Produce as sum stored in produced\_bytes
- confluent\_kafka\_server\_response\_bytes by kafka\_id (Cluster) and principal\_id (User) for the type Fetch as sum stored in fetched\_bytes
- confluent\_kafka\_server\_retained\_bytes by kafka\_id (Cluster) and topic as min and max stored in retained\_bytes\_min and retained\_bytes\_max
- confluent\_kafka\_server\_consumer\_lag\_offsets by kafka\_id (Cluster) and topic as list of consumer\_group\_id stored in consumergroups

maybe more are possible.

## 3.7. Deployment View



## 3.8. Risks and Technical Debts

- Difficulty to get context data
  - Will the customer be willing to make the effort to provide the necessary data?
- Difficulty to put a set price on each kafka item
- How to integrate general cost like operation, etc. (not linked to a particular kafka item)
- Difficulty of integration with companies cost dashboard

## 3.9. Glossary

Term	Definition
<i>OU</i>	<i>Organization Unit</i>