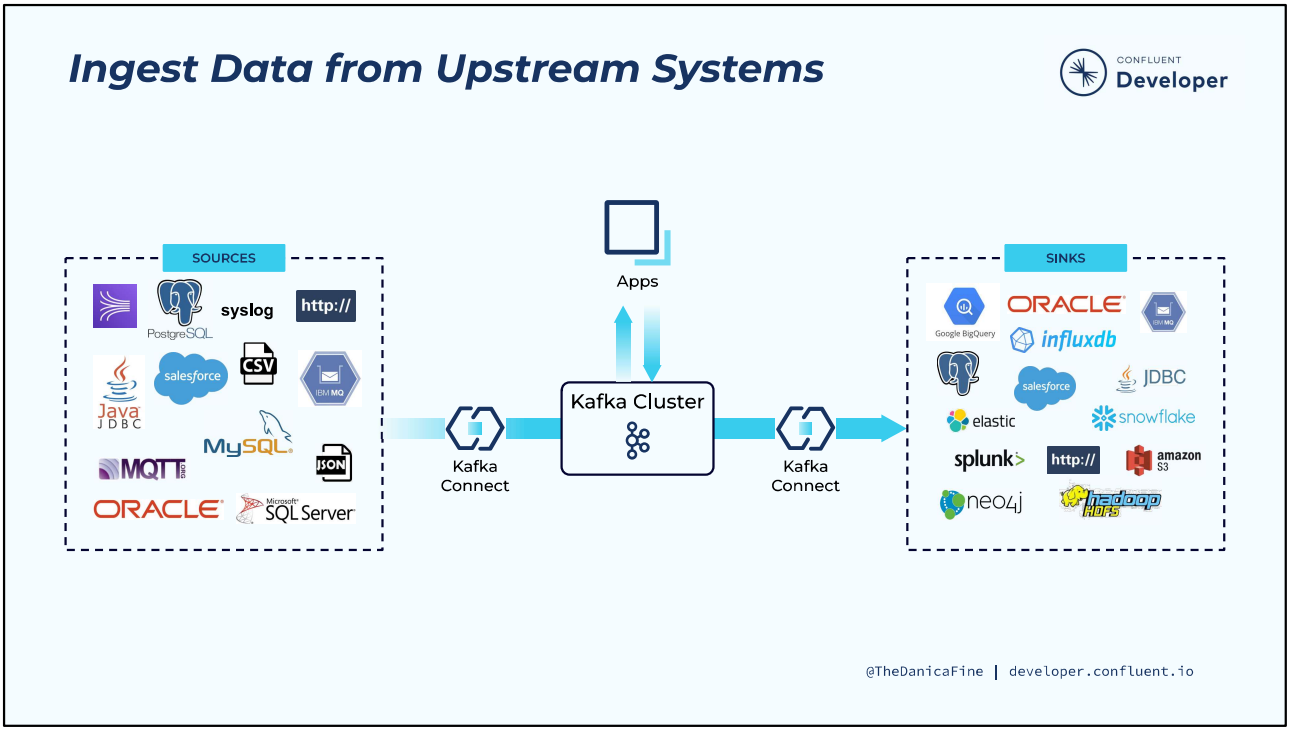

CONFLUENT
Developer
Kafka Connect 101

What's Covered

CONFLUENT Developer

1. Introduction to Kafka Connect
2. Hands On: Getting Started with Kafka Connect
3. Running Kafka Connect
4. Connectors, Configuration, Converters, and Transforms
5. Hands On: Using SMTs with a Managed Connector
6. Hands On: Confluent Cloud Managed Connector API
7. Hands On: Confluent Cloud Managed Connector CLI
8. Deploying Kafka Connect
9. Running Kafka Connect in Docker
10. Hands On: Run a Self-Managed Connector in Docker
11. Kafka Connect's REST API
12. Monitoring Kafka Connect
13. Errors and Dead Letter Queues
14. Troubleshooting Confluent Managed Connectors
15. Troubleshooting Self-Managed Kafka Connect

Hi, I'm Danica Fine with Confluent, and today I'm going to tell you about Kafka Connect.



Kafka Connect is a component of Apache Kafka that's used to perform streaming integration between Kafka and other systems such as databases, cloud services, search indexes, file systems, and key-value stores.

(And, before we dive more deeply into this subject, If you're new to Apache Kafka, I recommend that you take a look at the Apache Kafka 101 course to get started.)

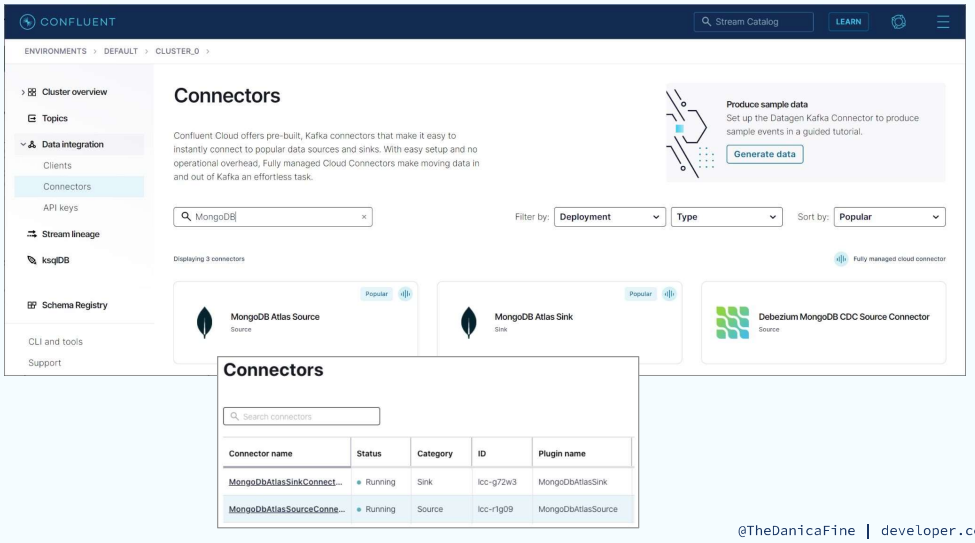
Kafka Connect makes it easy to stream data from numerous sources ***into*** Kafka, and stream data ***out of*** Kafka to numerous targets. The diagram you see here shows a small sample of these sources and targets. There are literally hundreds of different connectors available for Kafka Connect. Some of the most popular ones include:

- RDBMS (Oracle, SQL Server, DB2, Postgres, MySQL)
- Cloud Object stores (Amazon S3, Azure Blob Storage, Google Cloud Storage)

- Message queues (ActiveMQ, IBM MQ, RabbitMQ)
- NoSQL and document stores (Elasticsearch, MongoDB, Cassandra)
- Cloud data warehouses (Snowflake, Google BigQuery, Amazon Redshift)

Confluent Cloud Managed Connectors

CONFLUENT Developer



@TheDanicaFine | developer.confluent.io

We'll be focusing on running Kafka Connect more in the course modules that follow, but, for now, you should know that one of the cool features of Kafka Connect is that it's flexible.

You can choose to run Kafka Connect yourself or take advantage of the numerous fully-managed connectors provided in Confluent Cloud for a fully cloud-based integration solution. In addition to managed connectors, Confluent provides fully-managed Apache Kafka, Schema Registry, and stream processing with KSQL.

How Kafka Connect Works

CONFLUENT Developer

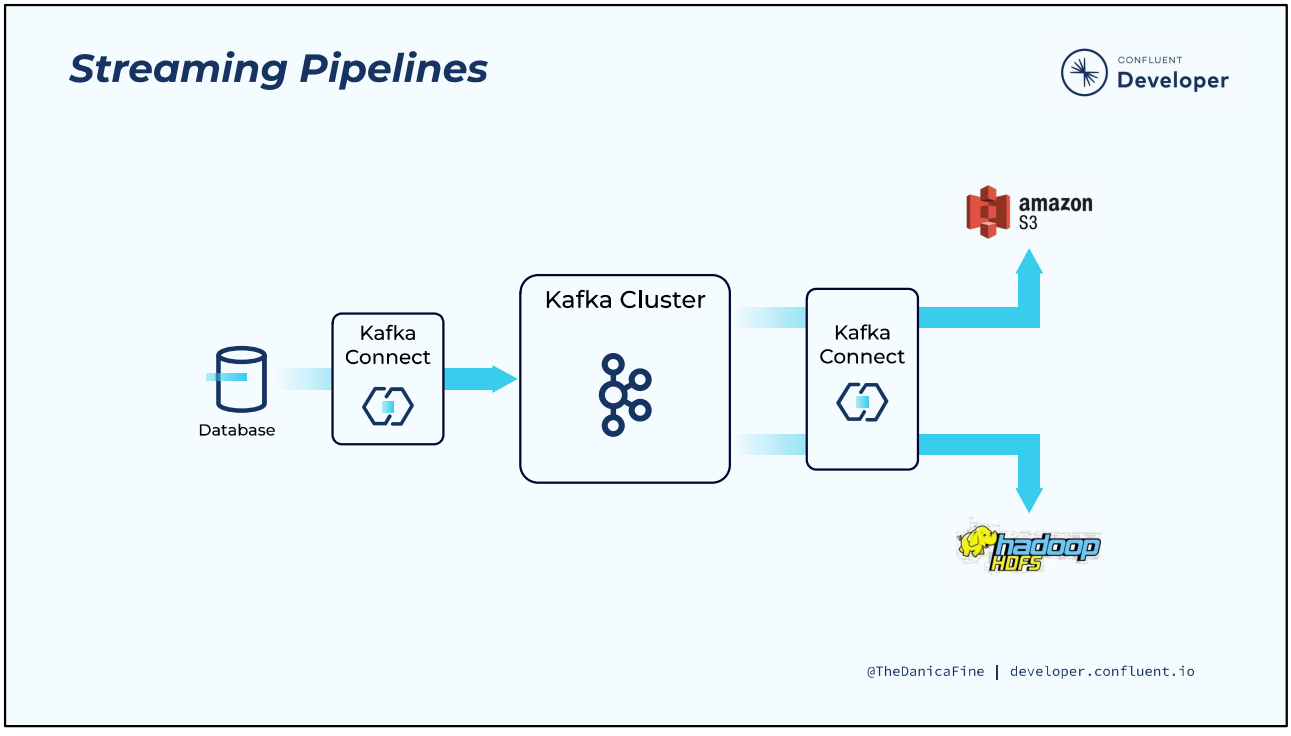

```
{
    "connector.class":
        "io.confluent.connect.jdbc.JdbcSourceConnector",
    "connection.url":
        "jdbc:mysql://asgard:3306/demo",
    "table.whitelist":
        "sales,orders,customers"
}
```

@TheDanicaFine | developer.confluent.io

Kafka Connect runs in its own process, separate from the Kafka brokers. It is distributed, scalable, and fault tolerant, giving you the same features we know and love about Kafka itself.

But the best part of Kafka Connect is that using it requires no programming. It's completely configuration-based, making it available to a wide range of users – not just developers. In addition to ingest and egress of data, Kafka Connect can also perform lightweight transformations on the data as it passes through.

Anytime you are looking to stream data into Kafka from another system, or stream data from Kafka to elsewhere, Kafka Connect should be the first thing that comes to mind. Let's take a look at a few common use cases where Kafka Connect is used.

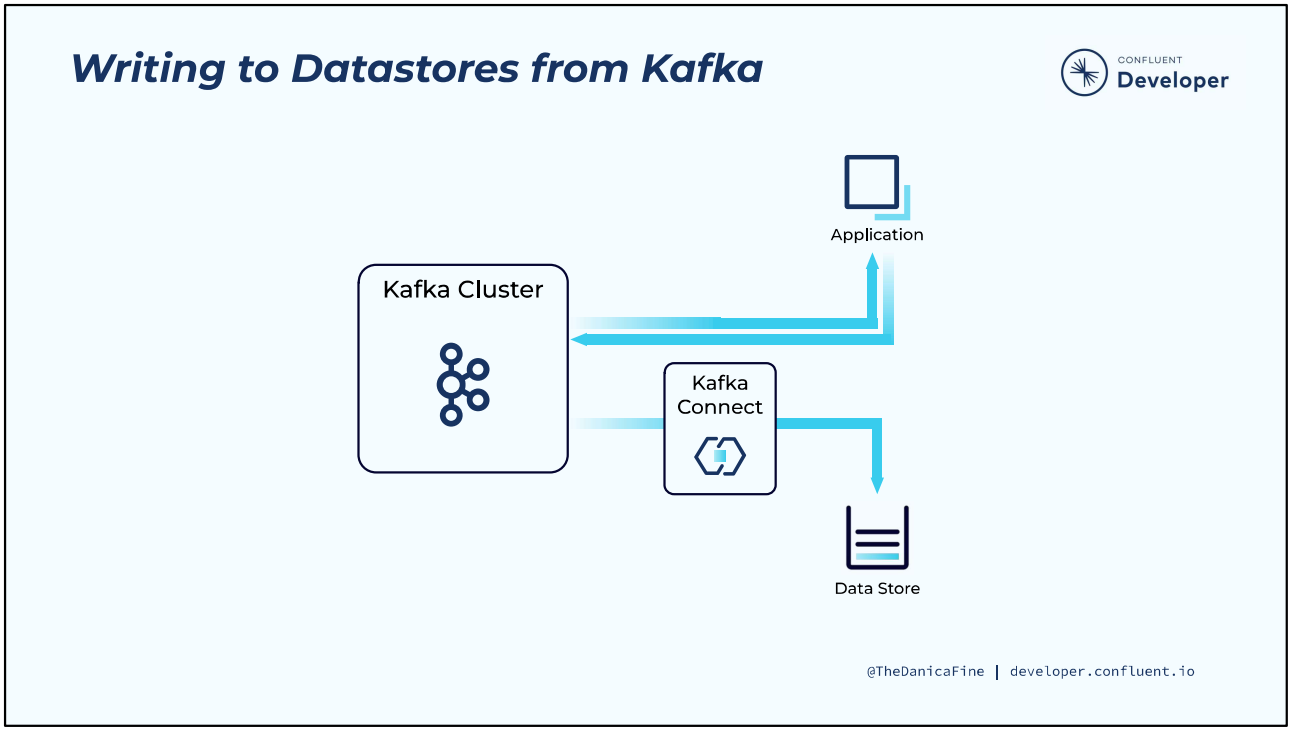


Kafka Connect can be used to ingest real-time streams of events from a data source and stream it to a target system for analytics. In this particular example, our data source is a transactional database.

We have a Kafka connector polling the database for updates and translating the information into real-time events that it produces to Kafka.

That in and of itself is great, but there are several other useful things that we get by adding Kafka to the mix:
- First of all, having Kafka sit between the source and target systems means that we're building a loosely coupled system. In other words, it's relatively easy for us to change the source or target without impacting the other.
- Additionally Kafka acts as a buffer for the data, applying back-pressure as needed
- And also, since we're using Kafka, we know that the system as a whole is scalable and fault tolerant.

Because Kafka stores data up to a configurable time interval per data entity (topic), it's possible to stream the same original data to multiple downstream targets. This means that you only need to move data into Kafka once while allowing it to be consumed by a number of different downstream technologies for a variety of business requirements or even to make the same data available to different areas in a business.

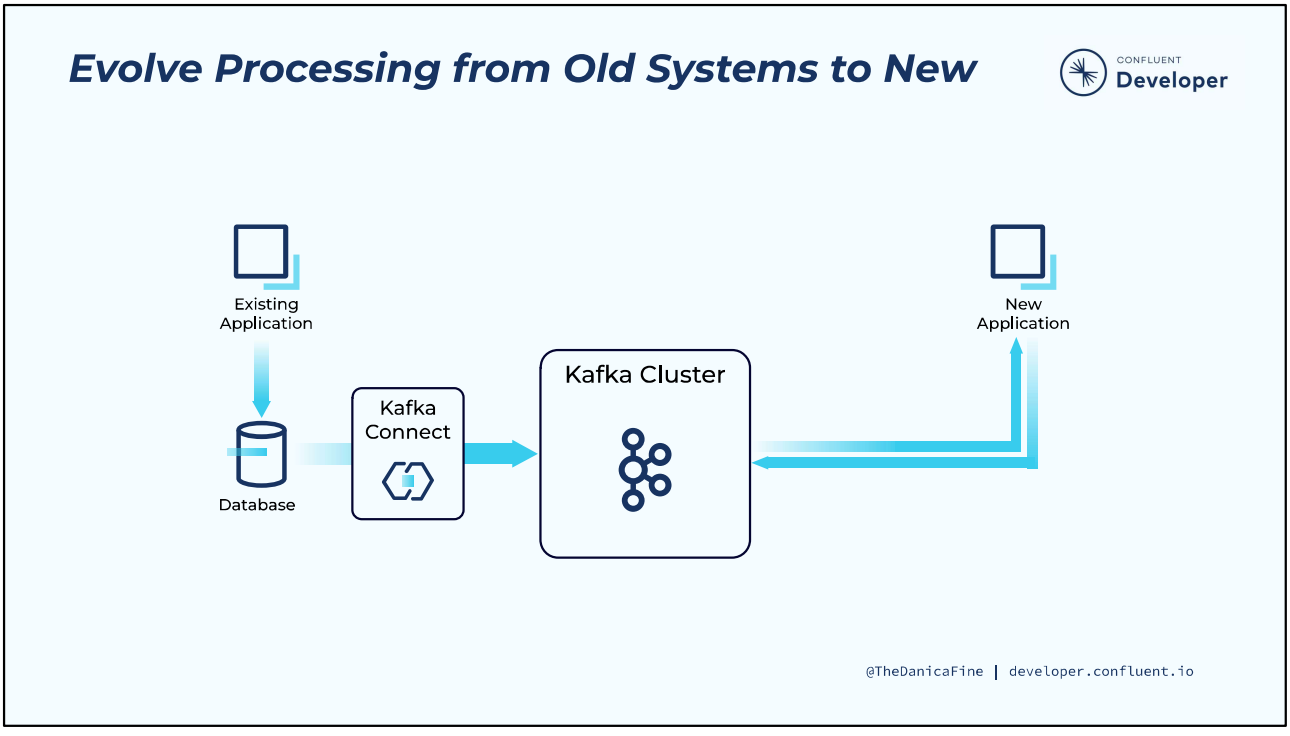


As another use case, you may want to write data created by an application to a target system. This of course could be a number of different application use cases, but suppose that we have an application producing a series of logging events, and we'd like those events to also be written to a document store or persisted to a relational database.

Imagine that you added this logic to your application directly. You'd have to write a decent amount of boilerplate code to make this happen, and whatever code you do add to your application to achieve this will have nothing to do with the application's business logic. Plus, you'd have to maintain this extra code, determine how to scale it along with your application, how to handle failures, restarts, etc…

Instead, you could add a few simple lines of code to produce the data straight to Kafka and allow Kafka Connect to handle the rest. As we saw in the last example, by moving the data to Kafka, we're free to set up Kafka connectors to move the data to whatever

downstream datastore that we need, and it’s fully decoupled from the application itself.

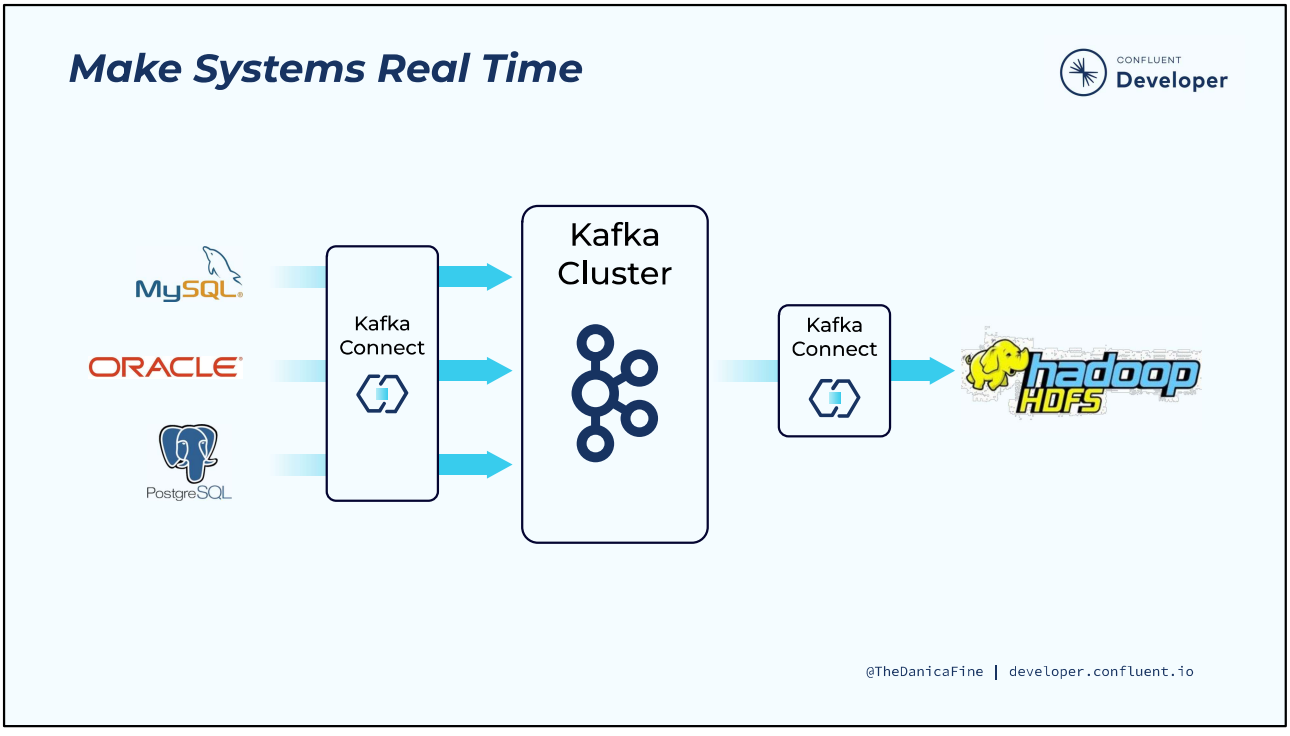


Before the advent of more recent technologies (such as NoSQL stores, event streaming platforms, and microservices) relational databases (RDBMS) were the de facto place to which all application data was written. These data stores still have a hugely important role to play in the systems that we build – but… not always. Sometimes we will want to use Kafka to serve as the message broker between independent services as well as the permanent system of record. These two approaches are very different, but unlike technology changes in the past, there is a seamless route between the two.

By utilizing change data capture (CDC), it's possible to extract every INSERT, UPDATE, and even DELETE from a database into a stream of events in Kafka. And we can do this in near real-time. Using underlying database transaction logs and lightweight queries, CDC has a *very* low impact on the source database, meaning that the existing application can continue running without any changes, all the while new applications can be built, driven by the stream of events captured from the underlying database. When the original

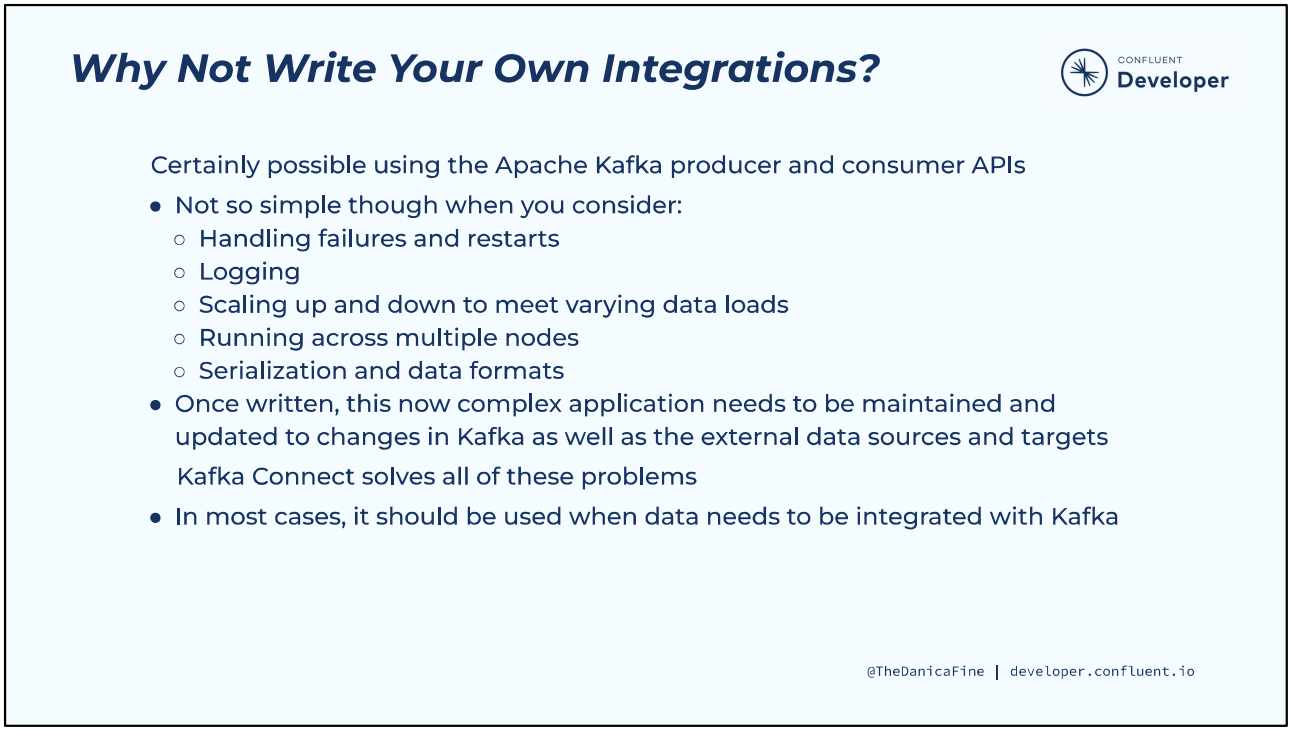
application records something in the database – for example, an order is accepted – any application subscribed to the stream of events in Kafka will be able to take an action based on the events – for example, a new order fulfilment service.

https://developer.confluent.io/learn-kafka/data-pipelines/kafka-data-ingestion-with-cdc/



And this is an incredibly valuable thing because many organizations have data at rest in databases and they'll continue to do so!

But the real value of data lies in our ability to access it as close to when it is generated as possible. By using Kafka Connect to capture data soon after it's written to a database and translating it into a stream of events, you can create so much more value. Doing so unlocks the data so you can move it elsewhere, for example adding a search index or analytics cluster. Alternatively the event stream can be used to trigger applications as the data in the database changes, say to recalculate an account balance or make a recommendation.



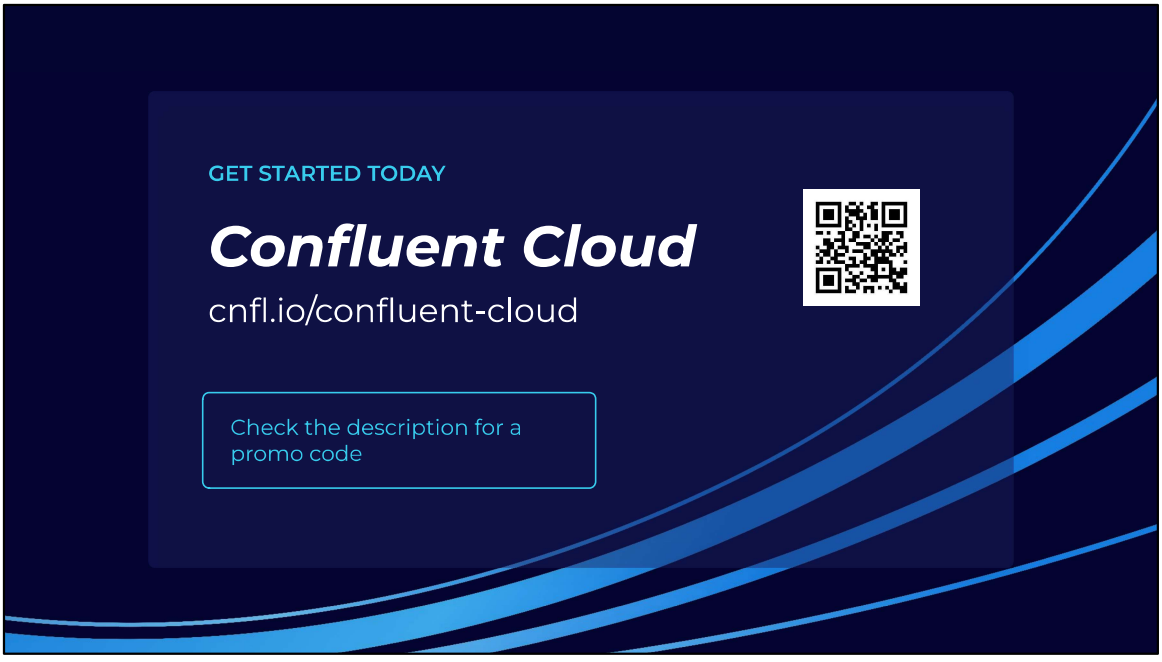
All of this sounds great, but I'm sure you're asking, Why Kafka Connect? Why Not Write Our Own Integrations?

Apache Kafka has its own very capable producer and consumer APIs and client libraries available in many languages, including C/C++, Java, Python, and Go. So it makes sense for you to wonder why you wouldn't just write your own code to move data from a system and write it to Kafka – doesn't it make sense to write a quick bit of consumer code to read from a topic and push it to a target system?

The problem is that if you are going to do this properly, then you need to be able to account for and handle failures, restarts, logging, scaling out and back down again elastically, and also running across multiple nodes. And that's all before we've thought about serialization and data formats. Of course, once you've done all of these things, you've written something that is probably rather like Kafka Connect, but without the many years of development, testing, production validation, and community that exists around

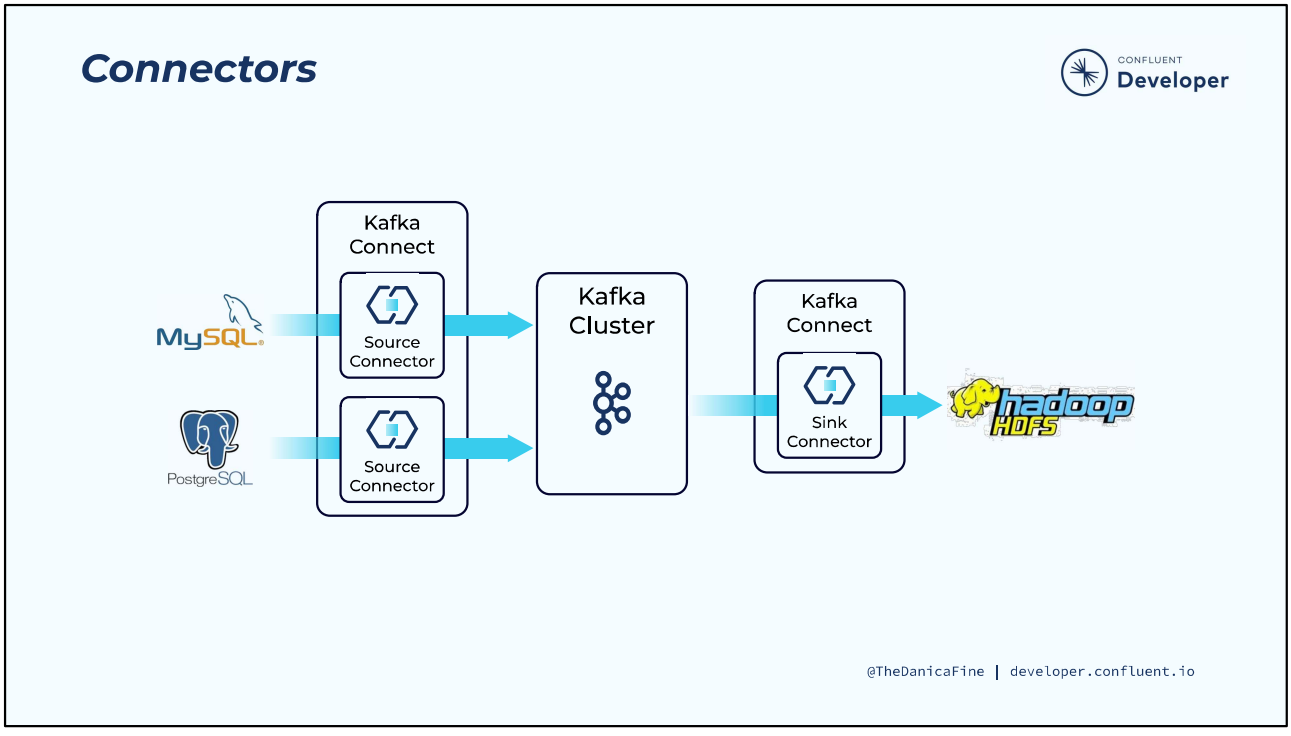
Kafka Connect. Even if you have built a better mousetrap, is all the time that you've spent writing that code to solve this problem worth it? Would your effort result in something that significantly differentiates your business from anyone else doing similar integration?

The bottom line is that integrating external systems with Kafka is a solved problem. There may be a few edge cases where a bespoke solution is appropriate, but by and large, you'll find that Kafka Connect will become the first thing you think of when you need to integrate a data system with Kafka.



Throughout this course, we'll be introduce you to Kafka Connect through hands-on exercises that will have you produce data to and consume data from Confluent Cloud. If you haven't already signed up for Confluent Cloud, sign up now so when your first exercise asks you to log in, you are ready to do so. Be sure to use the promo code when signing up to get the $101 of free usage that it provides.

Hi, Danica Fine here; let's learn how to run Kafka Connect!



When running Kafka Connect, instances of connector plugins provide the integration between external systems and the Kafka Connect framework. These connector plugins are reusable components that define how Source Connectors ought to capture data from data sources to a Kafka topic and also how Sink Connectors should copy data from Kafka topics to be recognized by a target system. By taking care of all of this boilerplate logic for you, the plugins allow you to hit the ground running with Kafka Connect and focus on your data.

There are hundreds of connector plugins available for a variety of data sources and sinks. There are dozens of fully-managed connectors available for you to run entirely through Confluent Cloud. Plus, connectors can also be downloaded from Confluent Hub for use with self-managed Kafka Connect.

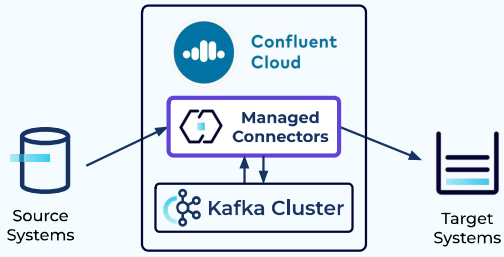
Let's dive a little bit more into the fully-managed and self-managed connectors and what those mean to you.

Confluent Cloud Managed Connectors

CONFLUENT Developer

Confluent Cloud provides fully managed connectors

- You just select the connector and identify details regarding the source/target system
  - GUI/CLI/API to create and manage connector instances
- Confluent takes care of the rest on your behalf
  - Provisioning, execution, failures, and so on
- Managed connector limitations
  - Some connectors are not yet available
  - Some connector transformations are not yet available
  - Some config settings may not be available
  - Network connectivity requirements



@TheDanicaFine | developer.confluent.io

Confluent Cloud offers pre-built, fully managed, Apache Kafka® Connectors that make it easy to instantly connect to popular data sources and sinks. With a simple UI-based configuration and elastic scaling with no infrastructure to manage, Confluent Cloud Connectors make moving data in and out of Kafka an effortless task, giving you more time to focus on application development.

To start, you simply select the connector and fill in a few configuration details about your source or target system. This can be done using the Confluent Cloud console, the confluent CLI, or the Confluent Connect API.

From there, Confluent takes care of the rest on your behalf

- Using the configuration settings you specified, your connector instance is provisioned and run.
- The execution of the connector instance is monitored
- Should the connector fail, you'll have access to troubleshooting to help identify the root cause, correct the issue, and restart the connector and its tasks

All in all, you can relax knowing that all of these tasks are being handled for you.

That said, there are a few limitations regarding managed connectors

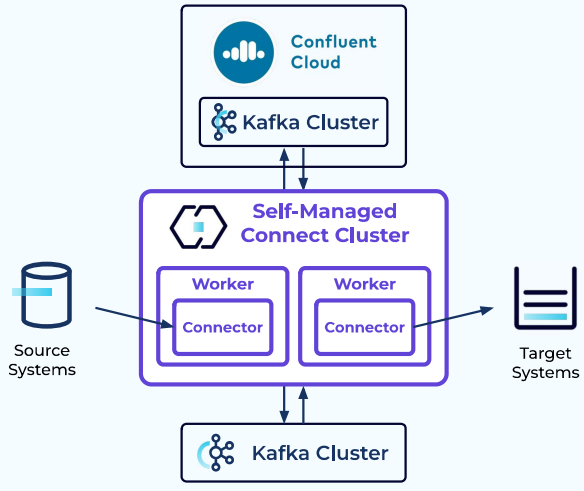
- Some self-managed connectors that are available on Confluent Hub for installation in self-managed Kafka Connect clusters are not yet available in Confluent Cloud
- Some fully-managed Confluent Cloud connectors are not available for all cloud providers
- Some configuration settings available for self-managed connectors may not be available for Confluent managed connectors
- Some single message transformations (SMT) that are available for use in self-managed Kafka Connect clusters are not available in Confluent Cloud
- Because Confluent Cloud fully-managed connectors are going to be accessing your data sources and data sinks, they need to be accessible over the Internet

Be sure to keep those things in mind as you choose which connector options are best for you.

Self-Managed Kafka Connect

CONFLUENT Developer

- Self-managed Kafka Connect consists of one or more Connect clusters depending upon the requirement
- Each Connect cluster consists of one or more Connect worker(s)
  - Connector instances run on Connect workers



@TheDanicaFine | developer.confluent.io

So long as you have access to a Kafka Cluster, Kafka Connect can also be run as a self-managed Kafka Connect Cluster, but as you can see from the diagram, there is a lot more involved with doing so.

- Self-managed Kafka Connect consists of one or more Connect clusters depending upon the requirement
- Each cluster consists of one or more connect worker machines on which the individual connector instances run

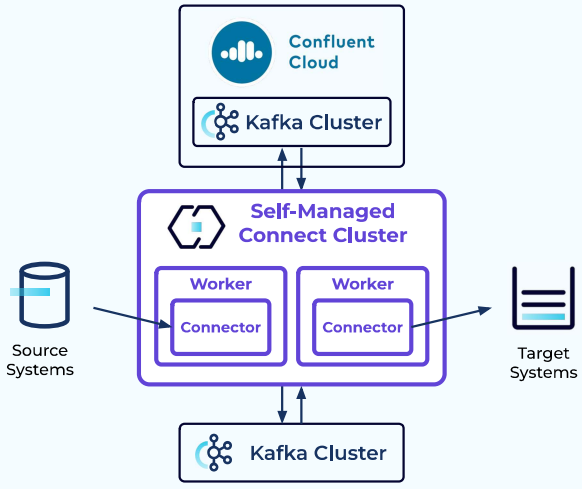
Regardless of how you choose to run Kafka Connect, it's helpful to understand the individual Kafka Connect components and how they work together.

Kafka Connect Workers

CONFLUENT Developer

Kafka Connect workers are JVM processes

- Can be deployed on bare metal or containers, e.g.
  - Bare-metal on-premises install of Confluent Platform
  - IaaS Compute (AWS EC2, Google Compute Engine, etc) install of Confluent Platform
  - Terraform:
    - AWS
    - Google
  - Docker
    - On-premises
    - Cloud-based



@TheDanicaFine | developer.confluent.io

Ultimately, Kafka Connect workers are just JVM processes. You can deploy on bare metal or containers.

A few options present themselves:

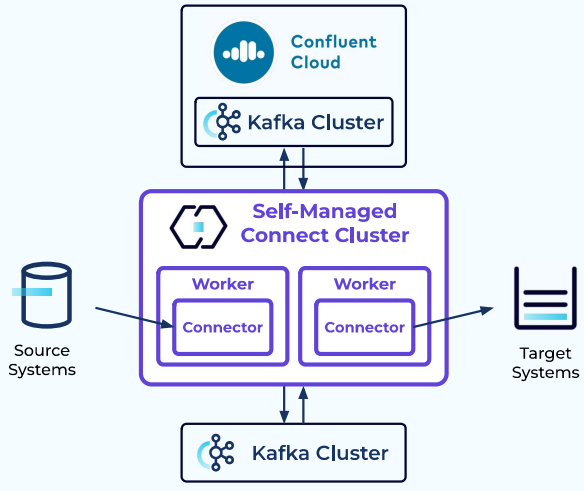
- Bare-metal on-premises install of Confluent Platform
- IaaS Compute (AWS EC2, Google Compute Engine, etc) install of Confluent Platform
- Terraform:
  - AWS
  - Google
- Docker
  - On-premises
  - Cloud-based

Managing a Kafka Connect Cluster

CONFLUENT Developer

Management responsibilities include:

- Worker configuration
- Scaling the Connect cluster up/down to suit demand changes
- Monitoring for problems
  - Troubleshooting
  - Corrective actions

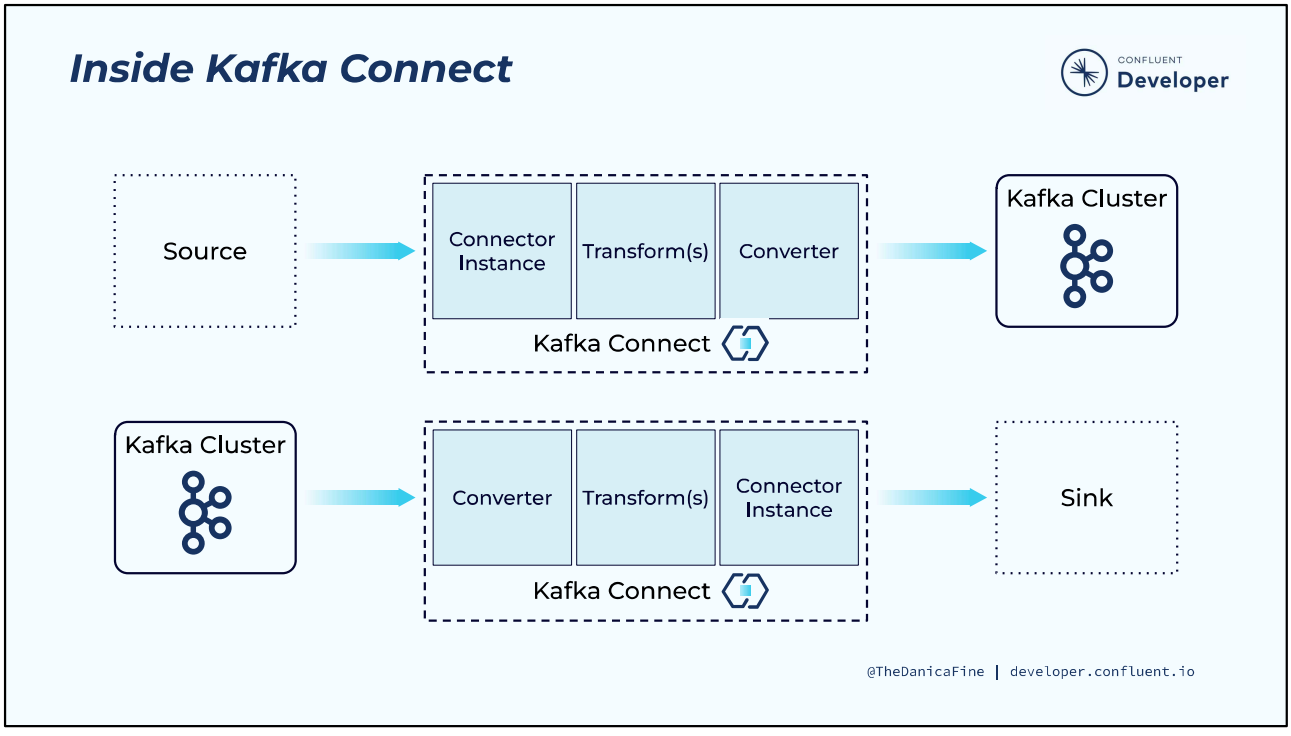


@TheDanicaFine | developer.confluent.io

Once your Kafka Connect cluster is up and running, there's a bit of management that needs to be done.

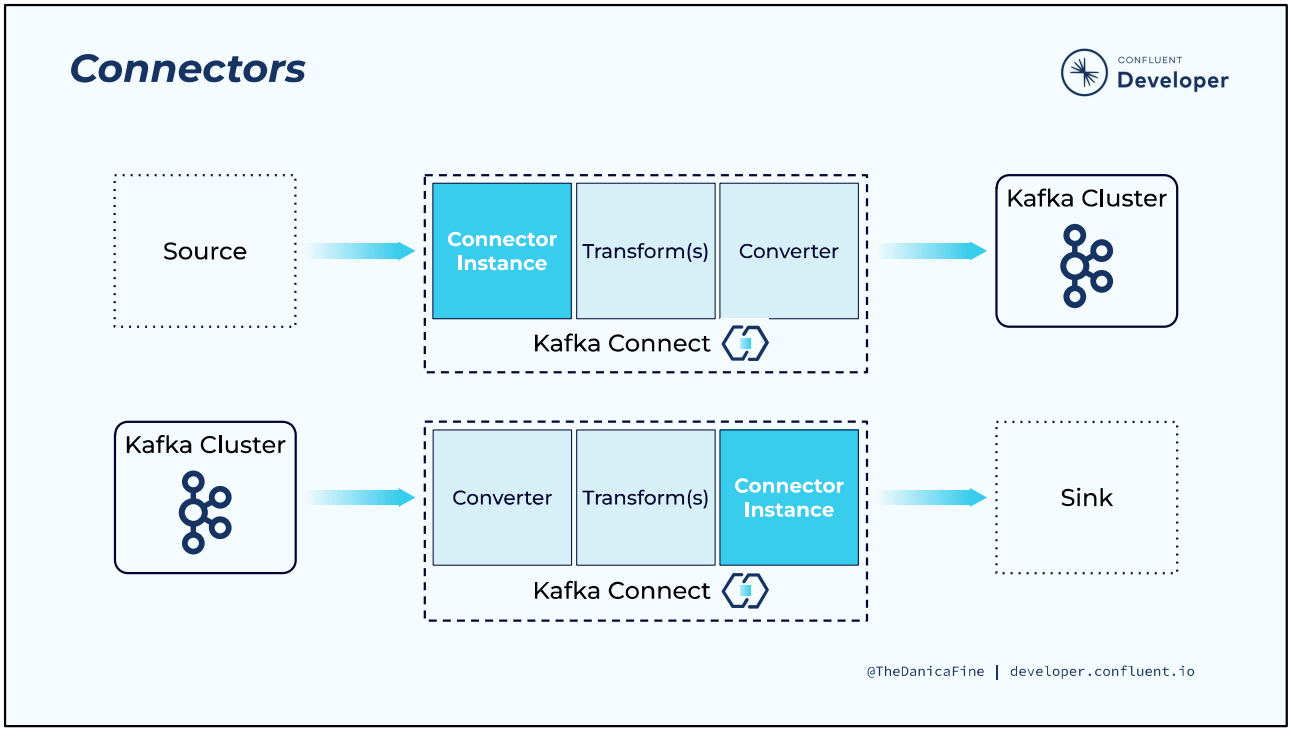
- Connect workers have a number of default configuration settings that you may need to alter
- Depending on the needs of your systems, you might need to scale the Connect cluster up or down to suit demand changes
- And of course, you'll be monitoring for problems
  - Troubleshooting
  - Corrective actions

Hi, Danica Fine here; let's take a deeper look into some of the components of Kafka Connect and the critical roles they play in moving your data.



Kafka Connect is built around a pluggable architecture of several components, which together provide very flexible integration pipelines. To get the most out of Kafka Connect it's important to understand these components and their roles:

- Connectors are responsible for the interaction between Kafka Connect and the external technology being integrated with
- Converters handle the serialization and deserialization of data
- Transformations can optionally apply one or more transformations to the data passing through the pipeline



The key component of any Kafka Connect pipeline is a connector instance which is a logical job that defines where data should be copied to and from. All of the classes that implement or are used by a connector instance are defined in its connector plugin. Written by the community, a vendor, or occasionally written bespoke by the user, the plugin integrates Kafka Connect with a particular technology. These plugins are reusable components that you can download, install, and use without writing code.

For example:

- The Debezium MySQL source connector uses the MySQL bin log to read events from the database and stream these to Kafka Connect
- The Elasticsearch sink connector takes data from Kafka Connect, and using the Elasticsearch APIs, writes the data to Elasticsearch
- The S3 connector from Confluent can act as both a source and sink connector, writing data to S3 or reading it back in

A SOURCE connector plugin knows how to talk to a specific SOURCE system and generate records that Kafka Connect then writes into Kafka. On the downstream side, the connector instance configuration specifies the topics to be consumed and Kafka Connect reads those topics and sends them to the SINK connector that knows how to send those records to a specific SINK system.

So the connectors know how to work with the records and talk to the external system, but Kafka Connect workers are acting as the conductor and taking care of the rest. We will define what a worker is shortly.

Add a Connector Instance with the REST API

CONFLUENT Developer

```
curl -X PUT -H  "Content-Type:application/json"
http://localhost:8083/connectors/sink-elastic-01/config \
    -d '{
"connector.class":
"io.confluent.connect.elasticsearch.ElasticsearchSinkConnector",
    "topics"          : "orders",
    "connection.url" : "http://elasticsearch:9200",
    "type.name"       : "_doc",
    "key.ignore"      : "false",
    "schema.ignore"   : "true"
}'
```


To specify a connector you include its name in your configuration - each connector's documentation will give you the particular classname string to use.
As you may expect, connectors have different configuration properties specific to the technology with which they're integrating. A Cloud connector will need to know the region, the credentials, and the endpoint to use. A database connector will need to know the names of the tables, the database hostname, and so on.

Here's an example of creating an Elasticsearch sink connector instance with a call to Kafka Connect's REST API.

Add a Connector Instance with ksqlDB

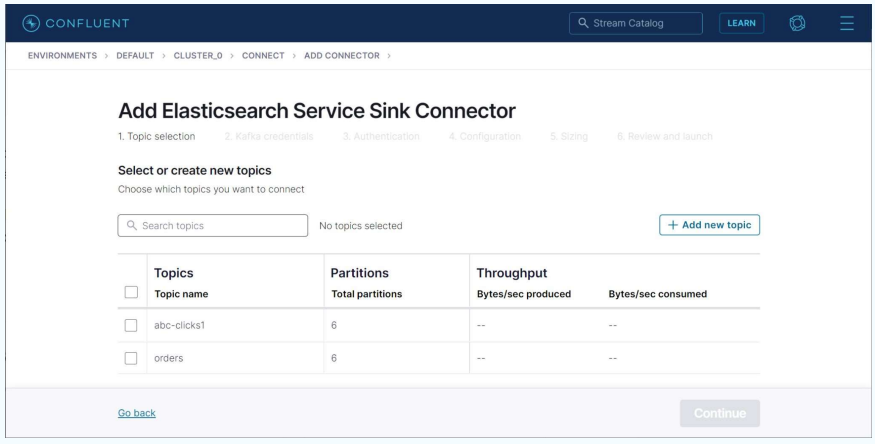
CONFLUENT Developer

```
CREATE SINK CONNECTOR sink-elastic-01 WITH (
  'connector.class' =
'io.confluent.connect.elasticsearch.ElasticsearchSinkConnector',
  'topics'          = 'orders',
  'connection.url'  = 'http://elasticsearch:9200',
  'type.name'       = '_doc',
  'key.ignore'      = 'false',
  'schema.ignore'   = 'true'
);
```


You can also use ksqlDB to manage connectors. Here is the syntax for adding the previous Elasticsearch sink connector instance.

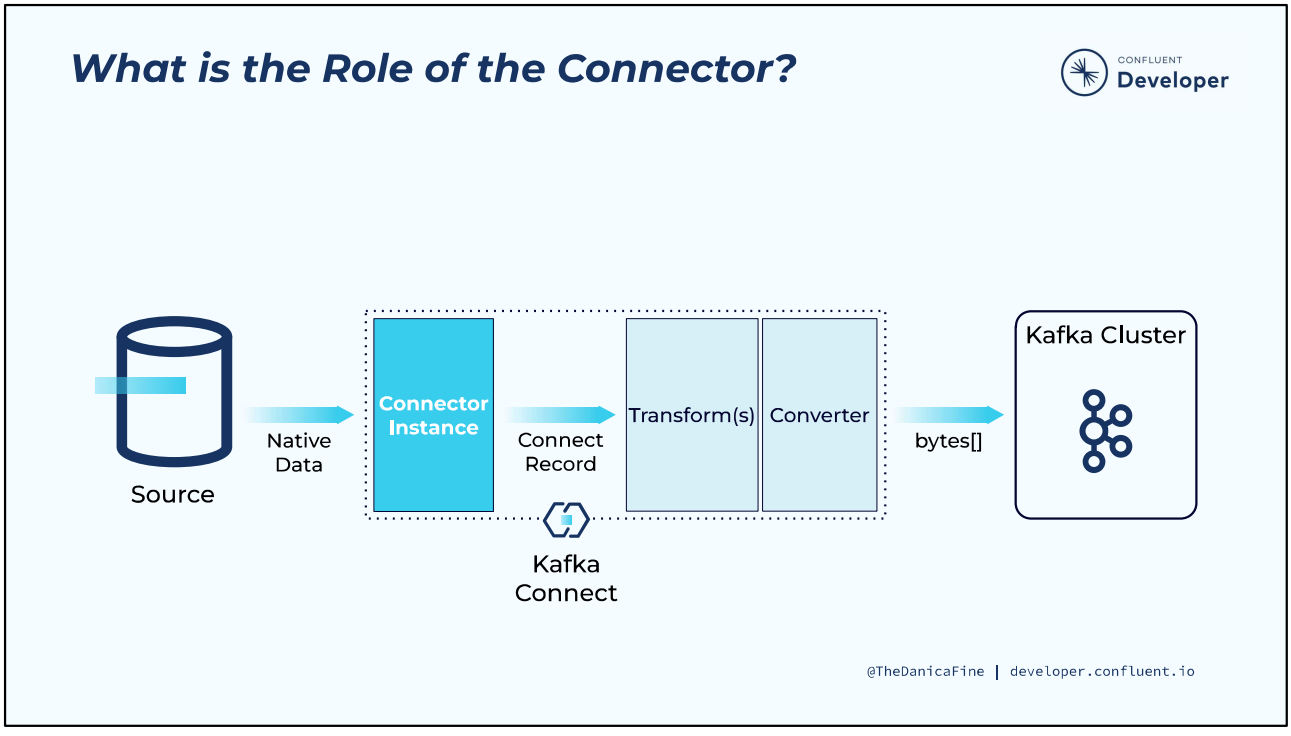
Add a Connector Instance with the Console UI

CONFLUENT Developer



@TheDanicaFine | developer.confluent.io

In addition to using the Kafka Connect REST API directly, you can add connector instances using the Confluent Cloud Console.



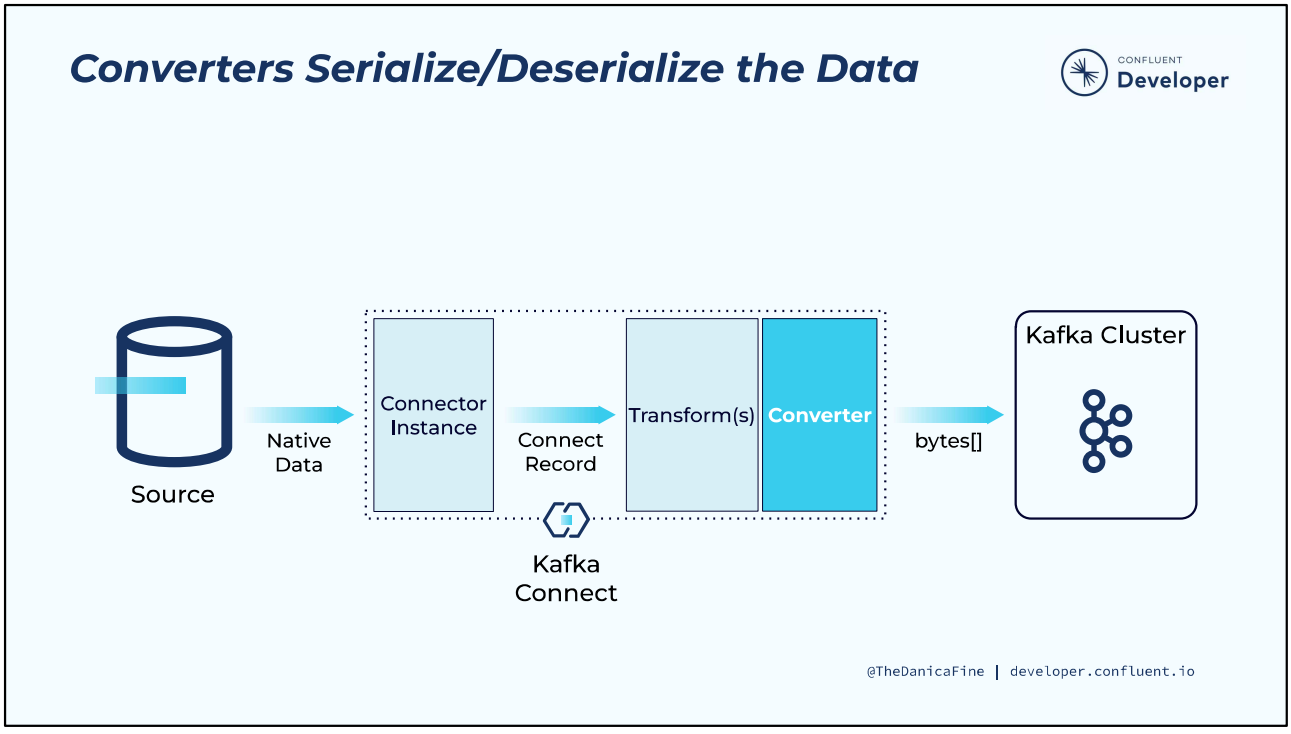
It's important to understand that the connector plugins themselves don't read from or write to (consume/produce) Kafka itself. The plugins just provide the interface between Kafka and the external technology. This is a deliberate design.

- Source connectors interface with the source API and extract the payload + schema of the data, and pass this internally as a generic representation of the data.
- Sink connectors work in reverse - they take a generic representation of the data, and the sink connector plugin writes that to the target system using its API.

Kafka Connect and its underlying components take care of writing data received from source connectors to Kafka topics as well as reading data from Kafka topics and passing it to sink connectors.

Now, this is all hidden from the user – when you add a new connector instance, that's all you need to configure and Kafka Connect does the rest to get the data flowing. But understanding

the next piece in the puzzle is important to help you avoid some of the common pitfalls with Kafka Connect, and that is converters. Technically, transforms sit between connectors and converters, but we'll visit those later.



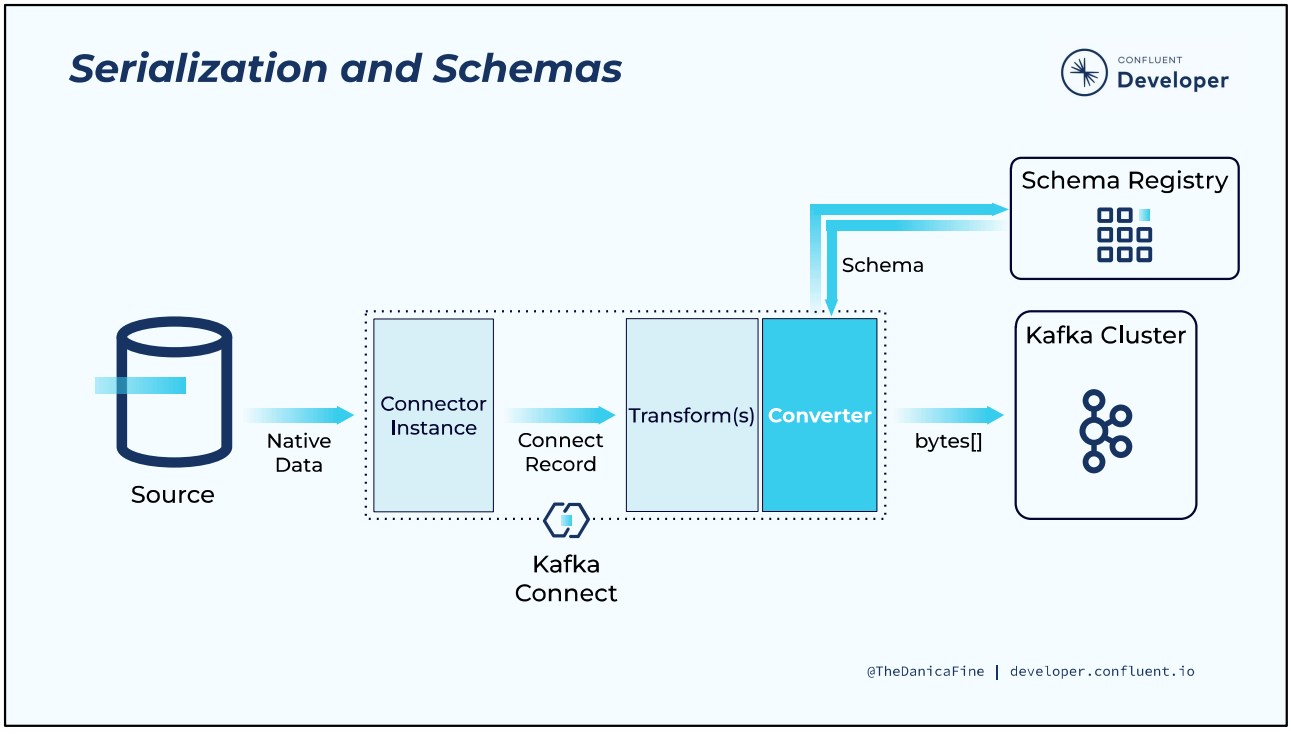
Converters are responsible for the serialization and deserialization of data flowing between Kafka Connect and Kafka itself. You'll sometimes see similar components referred to as SerDes ("SerialiserDeserialiser") in Kafka Streams, or just plain old serializers and deserializers in the Kafka Client libraries.

There are a ton of different converters available, but some common ones include:

- Avro - io.confluent.connect.avro.AvroConverter
- Protobuf - io.confluent.connect.protobuf.ProtobufConverter
- String - org.apache.kafka.connect.storage.StringConverter
- JSON - org.apache.kafka.connect.json.JsonConverter
- JSON Schema - io.confluent.connect.json.JsonSchemaConverter
- ByteArray - org.apache.kafka.connect.converters.ByteArrayConverter

While Kafka doesn't care about how you serialize your data (as far as it's concerned, it's just a series of bytes), you should care about

how you serialize your data! In the same way that you would take a carefully considered approach to how you design your services and model your data, you should also be deliberate in your serialization approach.

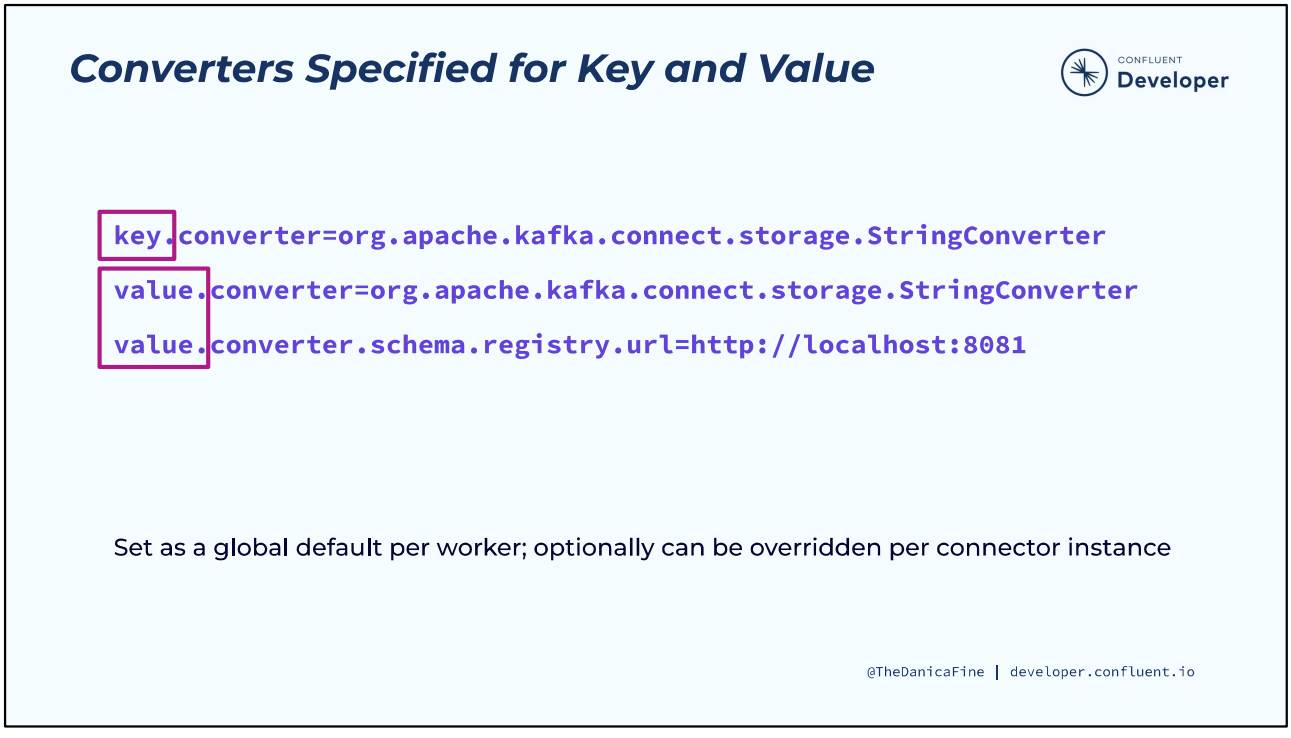


As well as managing the straightforward matter of serializing data flowing into Kafka and deserializing it on its way out, converters have a crucial role to play in the persistence of schemas. Almost all data that we deal with has a schema; it's up to us whether we choose to acknowledge that in our designs or not. You can consider schemas as the API between applications and components of a pipeline. Schemas are the contract between one component in the pipeline and another, describing the shape and form of the data.

When you ingest data from a source such as a database, as well as the rows of data, you have the metadata that describes the fields—the data types, their names, etc. Having this schema metadata is valuable, and you will want to retain it in an efficient manner. A great way to do this is by using a serialization method such as Avro, Protobuf, or JSON Schema. All three of these will serialize the data on to a Kafka topic and then store the schema separately in the Confluent Schema Registry. By storing the schema for data, you can easily utilize it in your consuming applications and pipelines. You can also use it to enforce data

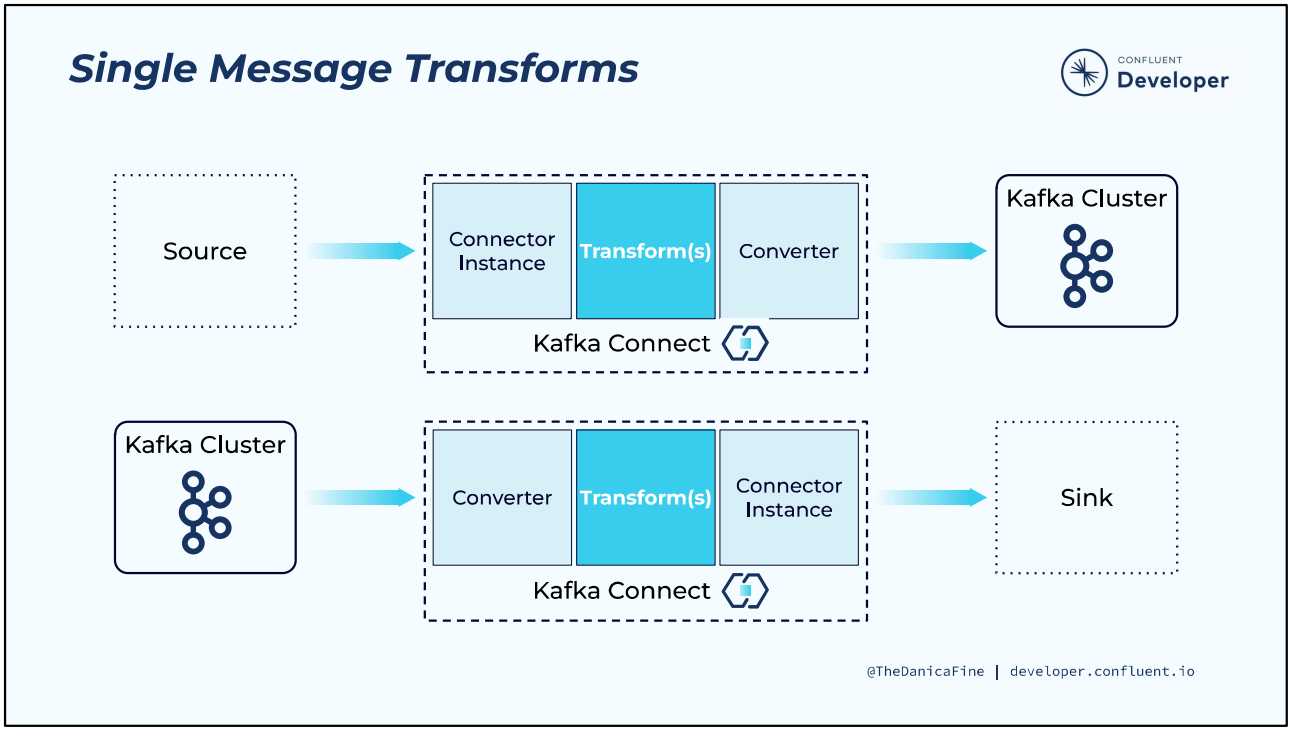
hygiene in the pipeline by ensuring that only data that is compatible with the schema is stored on a given topic.

You can opt to use serialization formats that don't store schemas like JSON, String, and byte array, and in some cases, these are valid. If you use these, just make sure that you are doing so for deliberate reasons and have considered how else you will handle schema information.



Converters are specified separately for the value of a message, and its key.

Note that these converters are set as a global default per connect worker, but they can be overridden per connector instance.



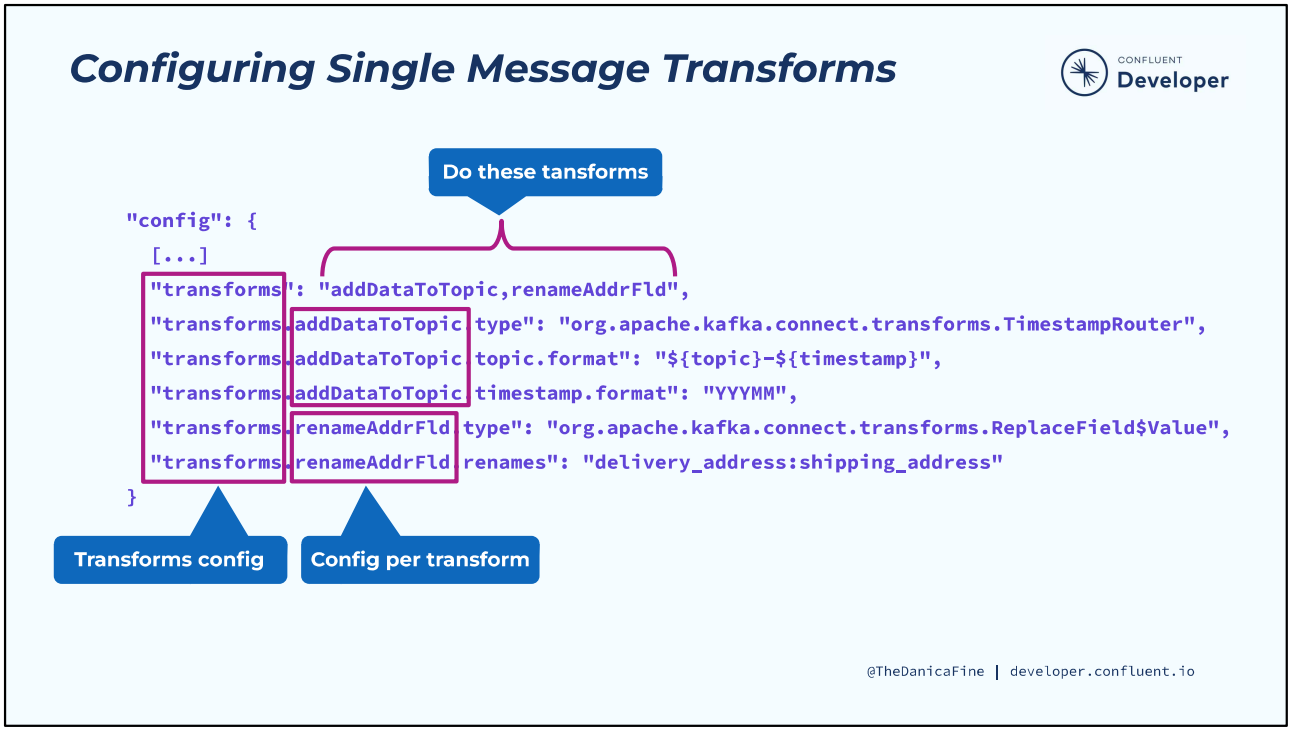
The third and final key component in Kafka Connect is the transform piece. Unlike connectors and converters, these are entirely optional. You can use them to modify data from a source connector before it is written to Kafka, and modify data read from Kafka before it's written to the sink. Transforms operate over individual messages as they move, so they're known as Single Message Transforms or SMTs.

Common uses for SMTs include:

- Dropping fields from data at ingest, such as personally identifiable information (PII) if specified by the system requirements
- Adding metadata information such as lineage to data ingested through Kafka Connect
- Changing field data types
- Modifying the topic name to include a timestamp
- Renaming fields

For more complex transformations, including aggregations and

joins to other topics or lookups to other systems, a full stream processing layer in ksqlDB or Kafka Streams is recommended.



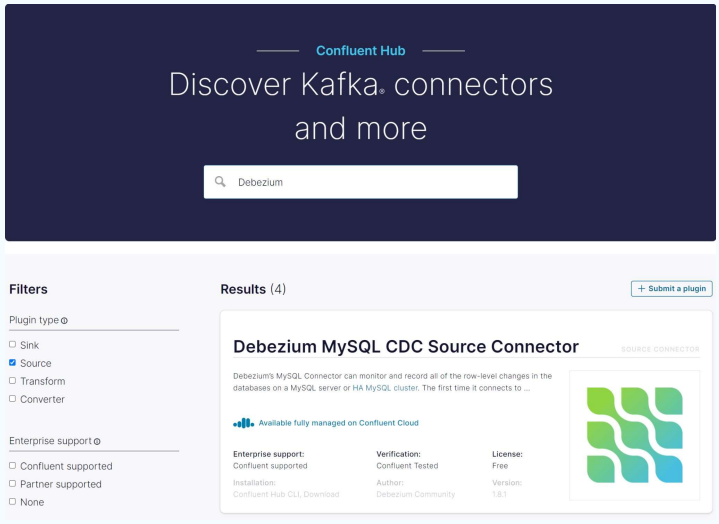
Configuring Single Message Transforms is relatively straight forward for managed connectors when using the Confluent Cloud UI as we will see in the exercise that follows. It is a bit tricky though when doing so manually, e.g. for self-managed connectors.

- `transforms` is the prefix for Single Message Transform configuration
- Each transformation has a label which can be set to something that is descriptive, e.g. `addDateToTopic` `renameAddrFld`
- Each transformation has a type
- You will often see transformation types with a $Value or $Key suffix - this denotes whether it's the Value or Key part of the message to which the tranform will be applied
- After that there will be zero or more additional configuration options, depending on the transformation being used
- The `addDateToTopic` example adds a Year/Month suffix to the target topic
- The `renameAddrFld` example renames the delivery_address

- field to shipping_address

Obtaining Plugins and Writing Your Own

CONFLUENT Developer



@TheDanicaFine | developer.confluent.io

Connectors, transforms, and converters are all specified as part of the Kafka Connect API, and you can consult the Javadocs to write your own.

Apache Kafka and Confluent have several converters and transforms built in already, but you can install more if you need them. You will find these along with hundreds of connectors in the Confluent Hub.

Now that we know the theory, let's take the previous hands-on exercise a step further and transform the input records before writing them to a Kafka topic. In the exercise that follows we'll do exactly that and see how easy it is to do so using Confluent cloud.

In this hands-on exercise, we'll walk through the process of creating a fully-managed Datagen connector, and also configure it to use the cast value single message transform to cast a few fields from the data before it's written to Kafka.

Hands On: Use SMTs with a Managed Connector

CONFLUENT Developer

@TheDanicaFine | developer.confluent.io

In the Confluent Cloud console, navigate into the kc-101 cluster. Expand Data integration and select Connectors.

0:31 First things first, we're using the Datagen connector in this exercise so let's find and select it using the filter.

0:38 We will once again generate sample data using the Orders record schema, but since we want to transform individual messages using a Single Message Transform, we will need to use the advanced configuration options.

Let's add our first SMT.

0:55 We could accept the default label for this transform, but it makes the configuration easier to read if we give it a name that corresponds to the SMT that will be used so let's do this. We're going to create an SMT to cast fields from each message, so we'll call it "castValues".

1:15 We also need to identify which SMT we want to use. In the Transform type list, select org.apache.kafka.connect.transforms.Cast$Value

1:20 For this SMT, we need to enter a list of fields we want to transform and the type we want to cast them to. We do this by specifying the field name and the desired type separated by the colon. We can use any number of these using a comma-delimited list. Set the value of spec equal to orderid:string, orderunits:int32

1:35 The configuration of our first SMT is complete.

required configuration parameters.

- Click Add a single message transform.
- Set the value of Transform name equal to convertTimestamp.
- In the Transform type list, select org.apache.kafka.connect.transforms.TimestampConverter$Value.
- Set the value of target.type equal to string.

This tells the SMT the resulting value should be type string.

- Set the value of field equal to ordertime.
- Set the value of format equal to yyyy-MM-dd.

This is the format the ordertime field will be changed to by the SMT. That completes the SMT configuration.

Let's continue to the next step.

1:59 For this Datagen connector instance, we will once again write the output records to the orders topic.

2:03 Select the orders topic and click Continue

2:08 In order for our connector to communicate with our cluster, we need to provide an API key for it. You can use an existing API key and secret, or create one here, as we're doing.

- With Global access selected, click Generate API key & download. Click Continue.
- We will also use the default sizing for this instance of the connector.Click Continue.

2:25 Before we launch the connector, let's examine its JSON configuration and identify the SMT related settings.

- Scroll the Confluent Cloud console down so the connector JSON configuration is visible.

2:33 Notice the configuration for the two transforms is included in the connector configuration.

2:37 You could also create the connector using either the Confluent Cloud Connect API or confluent CLI and this same JSON configuration. Other than having to provide the actual value for the API key and secret, the JSON is ready to use.

2:52 Let's now launch the connector and observe the result of the SMTs.

target records to keep them in view

- Enter values in the Jump to offset value until the ordertime format transition from original to transformed is in view and then quickly click pause

3:28 As you can see in the current view, offsets 69 and 70 have the original ordertime format and the messages written after offset 70 have the updated ordertime format. Notice also the change in data type for the orderid and orderunits fields.

3:44 Before we end the exercise, let's delete the connector so we don't unnecessarily deplete any Confluent Cloud promotional credits.

- Navigate to Data integration and select Connectors
- Select DatagenSourceConnector_0
- Click Delete
- Enter DatagenSourceConnector_0 in the confirmation field and click Confirm
- Click Cluster overview

3:52 Let's also delete the orders topic.

- Navigate to Topics.
- Select orders
- Select the Configuration tab
- Click Delete topic
- Enter orders in the provided field to confirm the delete action and click Continue

3:54 We will not delete the kc-101 cluster at this time since we will be using it in other exercises that are part of this course.

This concludes this exercise.

In this exercise we'll be using the Confluent Cloud Connector API.

Hands On: Confluent Cloud Managed Connector API

CONFLUENT Developer

@TheDanicaFine | developer.confluent.io

Exercise Environment Preparation Steps

In order to do this exercise using the Confluent Cloud Connector API, you first need to do a bit of preparation to make sure you have all of the tools you need. Complete the following steps to set up your environment. Prior to doing so, you will need to sign up for Confluent Cloud at https://confluent.cloud.

Confluent Cloud APIs are a core building block of Confluent Cloud. You can use them to manage your own account or to integrate Confluent into your product. The goal of this exercise is to demonstrate various Confluent Connect API REST calls that can be used to create, configure, and monitor Confluent managed connectors running in Confluent Cloud. We will also use the org API to identify the cluster ID for the kc-101 cluster as well as the cluster API to create a Kafka topic that we will be streaming data in and out of.

0:52 Start off by using the Confluent Cloud Org API. This requires a Confluent Cloud API key, which we'll generate using the Confluent CLI.

1:01 First, log into Confluent Cloud. Note that we're using the save flag, so that our credentials will be used for subsequent commands in this exercise. They'll remain active until you run the logout command.

1:14 As usual, we need an API Key to connect to our cloud cluster. Like we saw in the setup instructions, we can generate a new API Key here using:

- Run command:

- `confluent api-key create --resource cloud`

1:20 Next, we will convert the API key and secret to a base64 encoded string. This string is used in the authorization header that will be included in the REST calls to the Confluent Cloud API. At the same time, we will assign this string to an environment variable which will make it easier to submit the REST calls during the rest of this exercise.

- Run command:

  ```
  CLOUD_AUTH64=$(echo -n <API Key>:<API Secret> |
  base64 -w0)
  ```

1:40 We need to also convert the kc-101 cluster API key and secret to a base64 encoded string to use in the authorization header for REST calls to the Cluster API. It will be slightly easier for it since we can reference the environment variables created earlier from the java.config file.

- Run command:

  ```
  CLUSTER_AUTH64=$(echo -n $CLOUD_KEY:$CLOUD_SECRET |
  base64 -w0)
  ```

1:58 Now we are ready to issue REST calls to both Confluent Cloud control plane APIs as well as cluster APIS.

2:05 Let's first list our environments by issuing a REST call to the org API.

- Run command:

  ```
  curl --request GET \
  --url 'https://api.confluent.cloud/org/v2/environments'
\
  --header 'Authorization: Basic '$CLOUD_AUTH64''
  ```

2:11 Locate the default environment in the response and note its id. Using this id, we can now obtain the id for the kc-101 cluster.

2:20 Note: We will be using this environment id in later `curl` commands so keep it handy.

- Run command:

  ```
  curl --request GET \  --url
  'https://api.confluent.cloud/cmk/v2/clusters?environment
  =<env ID>' \  --header 'Authorization: Basic
  ```

as well.

- Run command:

```
curl --request POST \
--url
'https://pkc-6ojv2.us-west4.gcp.confluent.cloud:443/kafk
a/v3/clusters/<cluster ID>/topics' \
--header 'Authorization: Basic '$CLUSTER_AUTH64'' \
--header 'content-type: application/json' \
--data '{
  "topic_name": "transactions",
  "partitions_count": 6,  "replication_factor": 3
```

2:43 Before proceeding, we should take a moment to verify that the Transactions topic was created successfully.

- Run command:

```
curl --request GET \
  --url
'https://pkc-6ojv2.us-west4.gcp.confluent.cloud:443/kafk
a/v3/clusters/<cluster ID>/topics' \
  --header 'Authorization: Basic '$CLUSTER_AUTH64'' | jq
'.'
```

Stream sample data to a Kafka topic using the DatagenSource connector

2:52 Before we create the DatagenSource connector instance, it's always a good idea to list the fully-managed connector plugins that are available to us for streaming using our Confluent Cloud environment. Note that this list may be a little different depending upon what cloud provider Confluent Cloud is running in (and that's why it's good to check!).

- Run command:

```
curl --request GET \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster ID>/connector-plugins' \
  --header 'Authorization: Basic '$CLOUD_AUTH64''
```

3:12 It is a bit difficult to spot the different plugins in the command output. For our sanity, let's run the command again and pipe the output to the jq command to get it in a friendlier format.

- Run command:

```
curl --request GET \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
```

```
'.'
```

3:24 Alright, now we can actually tell which individual plugins are available. As you can see, the list is quite long and includes the DatagenSource connector.

3:34 Let's create a DatagenSource connector instance using the API. To do so, we use a PUT or POST REST call. PUT is somewhat easier because it will create the connector if it doesn't exist, or update it if it already exists. If it already exists and there's no update to make, it won't error—so PUT is the idempotent way of updating a connector.

- Run command:

```
curl --request PUT \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster
ID>/connectors/DatagenSourceConnector_2/config' \
  --header 'Authorization: Basic '$CLOUD_AUTH64'' \
  --header 'content-type: application/json' \
  --data '{
    "connector.class": "DatagenSource",
    "name": "DatagenSourceConnector_2",
    "kafka.api.key": "'$CLOUD_KEY'",
    "kafka.api.secret": "'$CLOUD_SECRET'",
    "kafka.topic": "transactions",
    "output.data.format": "AVRO",
    "quickstart": "TRANSACTIONS",
    "tasks.max": "1"
}'
```

2:57 To verify the connector instance's status, let's first list all connector instances in the cluster.

- Run command:

```
curl --request GET \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster ID>/connectors' \
  --header 'Authorization: Basic '$CLOUD_AUTH64''
```

4:05 Next, let's check the status of the__ DatagenSourceConnector_2__ connector instance we just created.

- Run command:

```
curl --request GET \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster
```

```
'.'
```

4:11 The current status is provisioning. This is expected as it takes a moment for the connector instance to be fully provisioned and up and running. Before we continue, it's a good idea to repeat this command periodically until we see the status has changed to Running.

4:26 Once the status shows the connector instance is Running, we can consume records from the destination topic to verify that sample data is being produced as expected.

- Run command:

```
kafka-avro-console-consumer \
--bootstrap-server ${BOOTSTRAP_SERVERS} \
--property schema.registry.url=${SCHEMA_REGISTRY_URL} \
--property
basic.auth.credentials.source=${BASIC_AUTH_CREDENTIALS_S
OURCE} \
--property
basic.auth.user.info=${SCHEMA_REGISTRY_BASIC_AUTH_USER_I
NFO} \
--consumer.config ~/.confluent/java.config \
--topic transactions \
--max-messages 10 \
--from-beginning
```

4:36 We should now have a decent amount of sample data in the transactions topic. So that we don't unnecessarily exhaust any Confluent Cloud promotional credits, we can go ahead and delete the DatagenSource connector instance.

- Run command:

```
curl --request DELETE \
  --url
'https://api.confluent.cloud/connect/v1/environments/ <en
v
ID>/clusters/<cluster
ID>/connectors/DatagenSourceConnector_2' \  --header
'Authorization: Basic '$CLOUD_AUTH64''
```

4:52 By this point, we've set up the start of our data pipeline – we made a connector to generate data to a Kafka topic. Next, we can establish the downstream side of the pipeline by setting up a MySql Sink Connector. This connector will consume records from the transactions topic and write them out to a corresponding table in our MySql database that is running in the local docker container that we started during the exercise environment setup steps.

5:19 Let's take a look at the request we'll be using to set up this connector. Notice that ssl.mode is set to prefer. This configuration parameter tells Confluent Cloud to

connect using TLS if the destination host is set up to do so. Otherwise, a PLAINTEXT connection will be established. For this demonstration, the local host is an AWS EC2 instance that does not have TLS set up so the connection will be nonsecure and the sample data will be unencrypted across the wire. In a production environment, we would want to be sure to set up the destination host to support TLS.

In our terminal window, enter command but do not run the command quite yet:

```
curl --request PUT \
    --url
'https://api.confluent.cloud/connect/v1/environments/env
-6vkq3/clusters/<cluster
ID>/connectors/MySqlSinkConnector_2/config' \
    --header 'Authorization: Basic '$CLOUD_AUTH64'' \
    --header 'content-type: application/json' \
    --data '{
        "connector.class": "MySqlSink",
        "name": "MySqlSinkConnector_2",
        "topics": "transactions",
        "input.data.format": "AVRO",
        "input.key.format": "STRING",
        "kafka.api.key": "'$CLOUD_KEY'",
        "kafka.api.secret": "'$CLOUD_SECRET'",
              "connection.host":
"ec2-54-175-153-98.compute-1.amazonaws.com",
        "connection.port": "3306",
        "connection.user": "kc101user",
        "connection.password": "kc101pw",
        "db.name": "demo",
        "ssl.mode": "prefer",
        "pk.mode": "none",
        "auto.create": "true",
        "auto.evolve": "true",
        "tasks.max": "1"

}'
```

5:55 Notice also the connection host. This should be set to the address of the host on which the mysql database Docker container was established during the exercise setup steps. In the case of the demonstration since an EC2 instance was being used, the sample command specifies the public endpoint address assigned to the AWS EC2 instance. This value can be obtained in the AWS console display of the EC2 instance details.

6:10 Now we can continue by running the curl command.

- Run the curl command

- Run command:

```
curl --request GET \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster
ID>/connectors/MySqlSinkConnector_2/status' \  --header
'Authorization: Basic '$CLOUD_AUTH64'' | jq '.'
```

6:18 We see the status of provisioning and need to wait for the status to be Running before continuing. We'll run it a few more times to check…

6:29 Once the status shows the connector instance is Running, we can run a query on the MySql database to verify the connector has written records to the transactions table.

- Run command:

```
docker exec -t mysql bash -c 'echo "SELECT * FROM
transactions LIMIT10 \G" | mysql -u root
-p$MYSQL_ROOT_PASSWORD demo'
```

Success!

6:41 Let's continue with our tour of the Confluent Cloud API.

6:45 Another useful command allows us to inspect the config for a given connector as follows.

- Run command:

```
curl --request GET \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster
ID>/connectors/MySqlSinkConnector_2/config' \  --header
'Authorization: Basic '$CLOUD_AUTH64'' | jq '.'
```

6:50 There are also use cases where you might need to pause a connector instance temporarily either for debugging or maintenance purposes. Here is the command to do this.

- Run command:

```
curl --request PUT \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster
ID>/connectors/MySqlSinkConnector_2/pause' \  --header
```

- Run command:

```
curl --request GET \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster
ID>/connectors/MySqlSinkConnector_2/status' \  --header
'Authorization: Basic '$CLOUD_AUTH64'' | jq '.'
```

Confirmed! Let's now resume the connector and its task.

- Run command:

```
curl --request PUT \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster
ID>/connectors/MySqlSinkConnector_2/resume' \
  --header 'Authorization: Basic '$CLOUD_AUTH64''
``
```

7:13 And finally, let's verify both the connector and task are once again running.

- Run command:

```
```sh
curl --request GET \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster
ID>/connectors/MySqlSinkConnector_2/status' \  --header
'Authorization: Basic '$CLOUD_AUTH64'' | jq '.'
```

7:20 Everything is up and in the Running state. So that wraps up our tour of the Confluent Connect API.

Note that if you run this demonstration yourself, you'll need to tear down the environment after doing so to avoid unnecessarily accruing cost to the point that your promotional credits are exhausted.

Let's walk through that tear down process now for this environment.

7:40 First things first, we should delete the sink connector.

- Run command:

```
curl --request DELETE \
  --url
'https://api.confluent.cloud/connect/v1/environments/<en
v ID>/clusters/<cluster
```
```

7:44 Since there's no running source connector, we're safe to go ahead and delete the transactions topic.

- Run command:

```
curl --request DELETE \
  --url
'https://pkc-6ojv2.us-west4.gcp.confluent.cloud:443/kafk
a/v3/clusters/<cluster ID>/topics/transactions' \
  --header 'Authorization: Basic '$CLUSTER_AUTH64''
```

7:50 And finally, we will shut down the MySQL Docker container and free its resources.

- Run command:

```
docker-compose down -v
```

And with that, you should have a decent idea of the REST API and the various commands available to you through it.

In this exercise, we'll be exploring the Confluent command line interface.

Exercise Environment Preparation Steps

0:16 In order to do this exercise using the Confluent Cloud Connector CLI, you first need to do a bit of preparation to make sure you have all of the tools you need. Complete the following steps to set up your environment. Prior to doing so, you will need to sign up for Confluent Cloud.

The confluent command line interface (CLI) is a convenient tool that enables developers to manage both Confluent Cloud and Confluent Platform. Built-in autocompletion is a convenient feature to help you quickly write commands. And with authentication and machine readable output, the CLI supports automated workflows as well.

The primary goal of this exercise is to demonstrate various confluent CLI commands that can be used to create, configure, and monitor Confluent managed connectors running in Confluent Cloud.

Confluent Cloud Managed Connector CLI exercise steps

0:55 We will start by logging into Confluent Cloud—and note that we're using the --save flag so that our credentials will be used for subsequent commands in this exercise. They'll remain active until you run the logout command.

- Run command:
  `confluent login --save`

Enter email and password

1:08 Let's start by setting the active environment and cluster that CLI commands apply to.

1:16 First, let's obtain the environment ID.

- Run command:
  `confluent environment list`

1:20 In the Confluent org being used for this demo, there is a single environment and its name is default.

1:25 Let's set it as the active environment for the confluent CLI command.

- Run command:
  `confluent environment use <default env ID>`

1:29 Next, we'll set the default cluster to use; let's obtain the cluster ID.

- Run command:
  `confluent kafka cluster list`

1:34 In the Confluent org being used for this demo, there is a single cluster and its name is kc-101.

1:38 Let's set it as the active cluster for the confluent CLI command.

- Run command:
  `confluent kafka cluster use <kc-101 cluster ID>`

1:42 Let's now create a new Kafka topic named transactions. We will set this as the target for the Datagen source connector.

- Run command:

`confluent kafka topic create transactions`

1:52 Notice the topic was created with default values of 6 partitions and a replication factor of 3.

1:58 Before proceeding, let's verify the transactions topic was created successfully.

- Run command:

  ```
  confluent kafka topic list
  ```

Stream sample data to a Kafka topic using the DatagenSource connector

2:04 Before we create the DatagenSource connector instance, let's list the fully managed connector plugins that are available for streaming with our Confluent Cloud environment.

2:14 Note: This list may be a little different depending upon what cloud provider Confluent Cloud is running in.

- Run command:

  ```
  confluent connect plugin list
  ```

2:21 As you can see, the list is quite long and includes the DatagenSource connector. Next, let's create a DatagenSource connector instance.

First, we need to update the file containing the connector instance configuration. In this demo we will use VSCode.

2:29 Note: This file was created on your behalf and included in the GitHub repo that was cloned earlier during the environment setup steps.

- Run command:

  ```
  code ~/learn-kafka-courses/kafka-connect-101
  ```

- In VSCode, navigate to the delta-configs directory and open file env.delta.

- Note: This file was created as part of the environment set up steps.
  - On line 8, copy the value assigned to CLOUD_KEY.
  - In VSCode, locate and open file datagen-source-config.json.
  - On line 7, replace <key> with the copied value assigned to CLOUD_KEY.
  - Return to file env.delta and on line 9, copy the value assigned to CLOUD_SECRET.
  - Return to file datagen-source-config.json and on line 8, replace <secret> with the copied value assigned to CLOUD_SECRET.
  - Save file datagen-source-config.json and close VSCode.

2:40 We can now create the DatagenSource managed connector instance.

- Run command:

```
confluent connect create --config datagen-source-config.json
```

2:42 To verify the connector instance's status, let's first list all connector instances in the cluster.

- Run command:

```
confluent connect list
```

2:48 The DatagenSource connector instance appears in the list with a status of Provisioning. This is expected as it takes a moment for the connector instance to be fully provisioned and running. We need to repeat this command periodically until we see the status has changed to Running before we continue.

3:05 Using the connector instance ID that was included in the list command output, let's use the describe option to obtain additional details about the connector instance.

- Run command:

```
confluent connect describe <connector ID>
```

3:15 Next, let's consume records from the transactions topic to verify sample data is being produced.

- Run command:

```
confluent kafka topic consume -b transactions \
  --value-format avro \
  --api-key $CLOUD_KEY \
  --api-secret $CLOUD_SECRET \
  --sr-endpoint $SCHEMA_REGISTRY_URL \
  --sr-api-key <SR key> \
  --sr-api-secret <SR secret>
```

Note: You will need to replace <SR key> and <SR secret> with their respective values which you can find on line 18 in ~/.confluent/java.config in the form of basic.auth.user.info=<SR key>:<SR secret>

- Once records begin to appear, press Ctrl+C to end the consumption.

3:21 We should now have sufficient sample data in the transactions topic. So that we don't unnecessarily exhaust any Confluent Cloud promotional credits, let's delete the

DatagenSource connector instance.

- Run command:

  ```
  confluent connect delete <connector ID>
  ```

3:34 Let's now establish the downstream side of our data pipeline. We will use the MySQLSink connector for this. It will consume records from the transactions topic and write them out to a corresponding table in our MySQL database that is running in the local Docker container that we started during the exercise environment setup steps.

First we need to update the file containing the connector instance configuration.

Note: This file was created on your behalf and included in the GitHub repo that was cloned earlier during the environment setup steps.

- Run command:

  ```
  code ~/learn-kafka-courses/kafka-connect-101
  ```

- In VSCode, navigate to the delta-configs directory and open file env.delta.

Note: This file was created as part of the environment set up steps.

- On line 8, copy the value assigned to CLOUD_KEY.
- In VSCode, locate and open file__ mysql-sink-config.json__.
- On line 8, replace <key> with the copied value assigned to CLOUD_KEY.
- Return to file env.delta and on line 9, copy the value assigned to CLOUD_SECRET.
- Return to file datagen-source-config.json and on line 9, replace <secret> with the copied value assigned to CLOUD_SECRET.
- On line 10, replace <mysql-host-endpoint> with the public endpoint of the host where the MySql database is running.

4:02 Notice that ssl.mode is set to prefer. This tells Confluent Cloud to connect using TLS if the destination hose is set up to do so. Otherwise a PLAINTEXT connection will be established. For this demonstration, the local host is an AWS EC2 instance that does not have TLS set up so the connection will be nonsecure and the sample data will be unencrypted across the wire. In a production environment, we would want to be sure to set up the destination host to support TLS.

4:31 Notice also the connection host. This should be set to the address of the host on which the mysql database Docker container was established during the exercise setup steps.. In the case of the demonstration since an EC2 instance was being used, the sample command specifies the public endpoint address assigned to the AWS EC2

instance. This value can be obtained in the AWS console display of the EC2 instance details.

We can now save and close the configuration file.

- Save file datagen-source-config.json and close VSCode.

4:43 Let's now create the MySQL Sink connector instance.

- Run command:

```
confluent connect create —config mysql-sink-config.json
```

4:47 To verify the connector instance's status, let's first list all connector instances in the cluster.

- Run command:

```
confluent connect list
```

4:53 The MySQL Sink connector instance appears in the list with a status of Provisioning. This is expected as it takes a moment for the connector instance to be fully provisioned and running. We need to repeat this command periodically until we see the status has changed to Running before we continue.

5:09 Using the connector instance ID that was included in the list command output, let's use the describe option to obtain additional details about the connector instance.

- Run command:

```
confluent connect describe <connector ID>
```

5:20 Next, let's run a query on the MySQL database to verify the connector has written records to the transactions table.

- Run command:

```
docker exec -t mysql bash -c 'echo "SELECT * FROM
transactions LIMIT 10 \G" | mysql -u root
-p$MYSQL_ROOT_PASSWORD demo'
```

Success!

5:29 Let's continue now with our tour of using the Confluent CLI with Confluent Cloud managed connectors.

5:36 Perhaps we want to pause a connector instance temporarily. Here is the

command to do this.

- Run command:

  `confluent connect pause <connector ID>`

5:40 Let's verify both the connector and task are paused using the status command.

- Run command:

  `confluent connect describe <connector ID>`

5:49 Confirmed.. Let's now resume the connector and its task.

- Run command:

  `confluent connect resume <connector ID>`

5:52 And let's verify both the connector and task are once again running.

- Run command:

  `confluent connect describe <connector ID>`

5:57 The connector and its task are once again in a Running state.

6:03 If you run this demonstration yourself, you need to tear down the environment after doing so to avoid unnecessarily accruing cost to the point your promotional credits are exhausted.

6:10 Let's walk through that tear down process now for this environment.

6:16 First we delete MySqlSinkConnector_2.

- Run command:

  `confluent connect delete <connector ID>`

6:20 Next we will delete the transactions topic.

- Run command:

  `confluent kafka topic delete transactions`

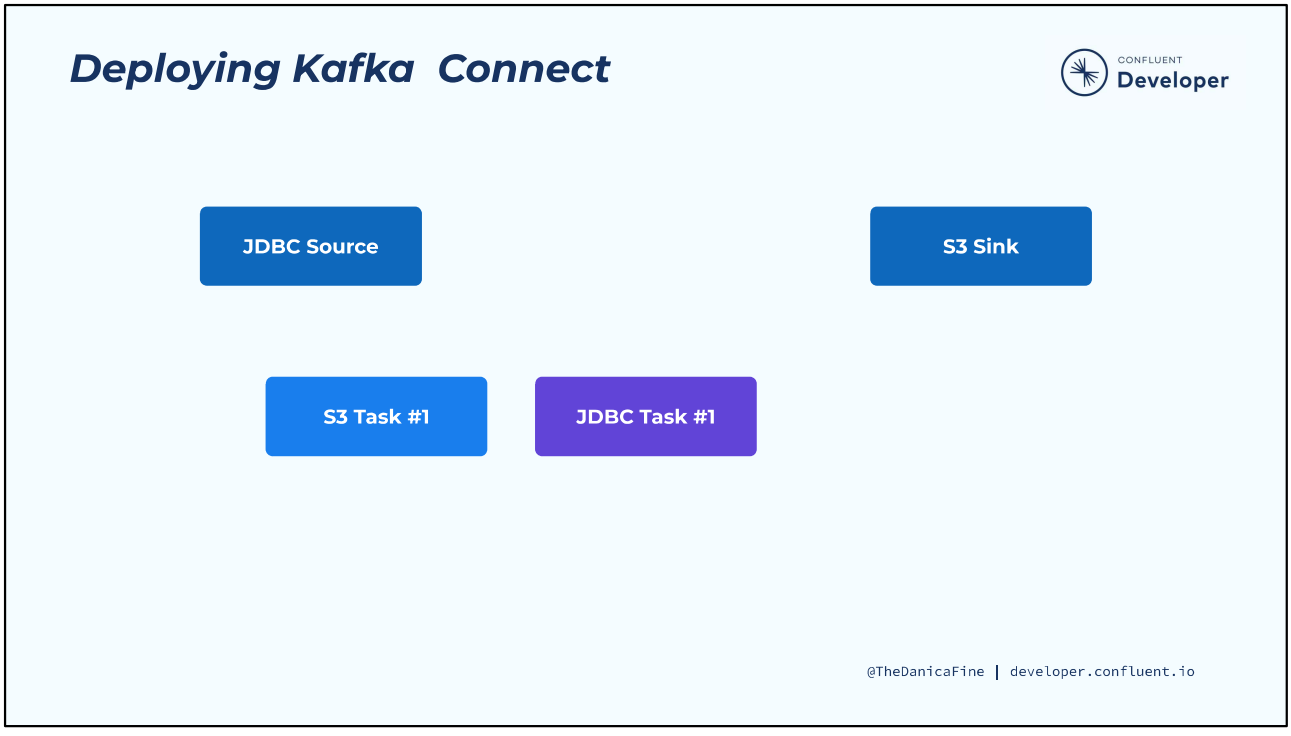
6:23 And finally, we will shut down the mysql Docker container and free its resources.

- Run command:

  `docker-compose down -v`

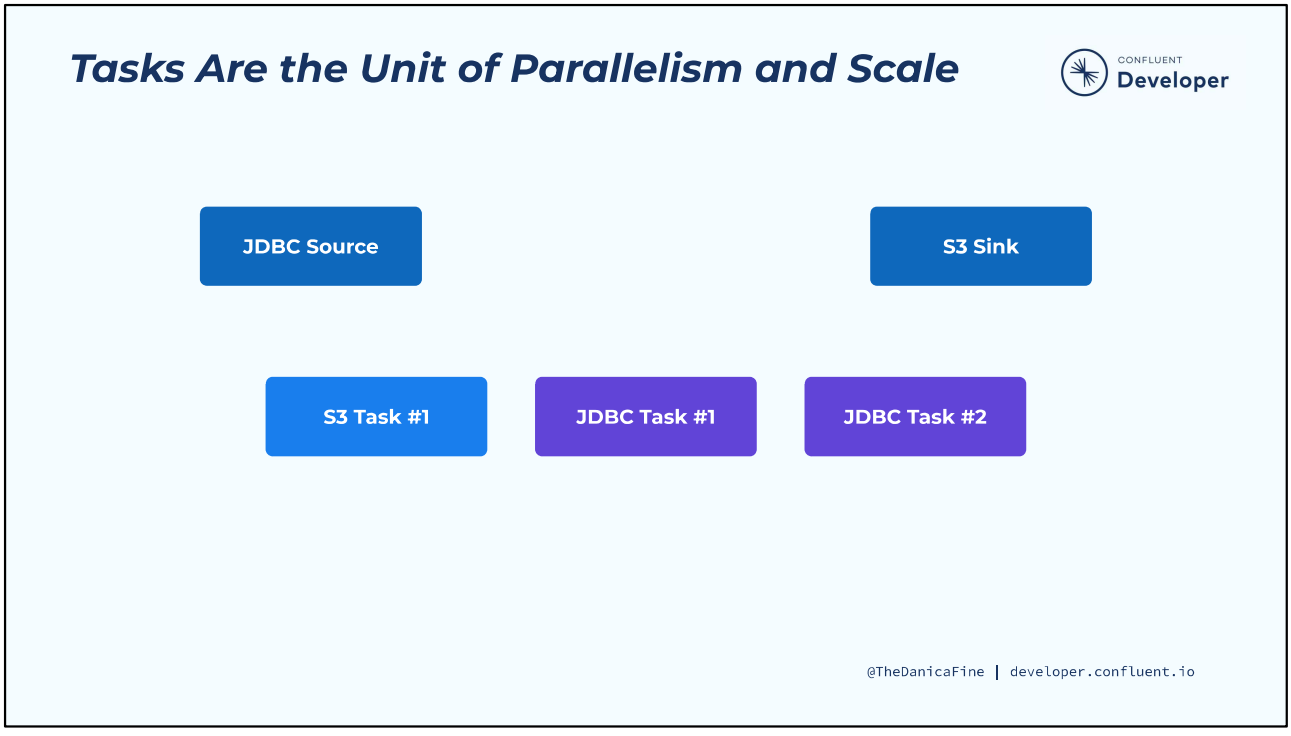
This concludes this demonstration.

Hi, Danica Fine here; up until now, we've learned a ton about Kafka Connect, its components, and how to interact with it. But now let's see how to deploy it.

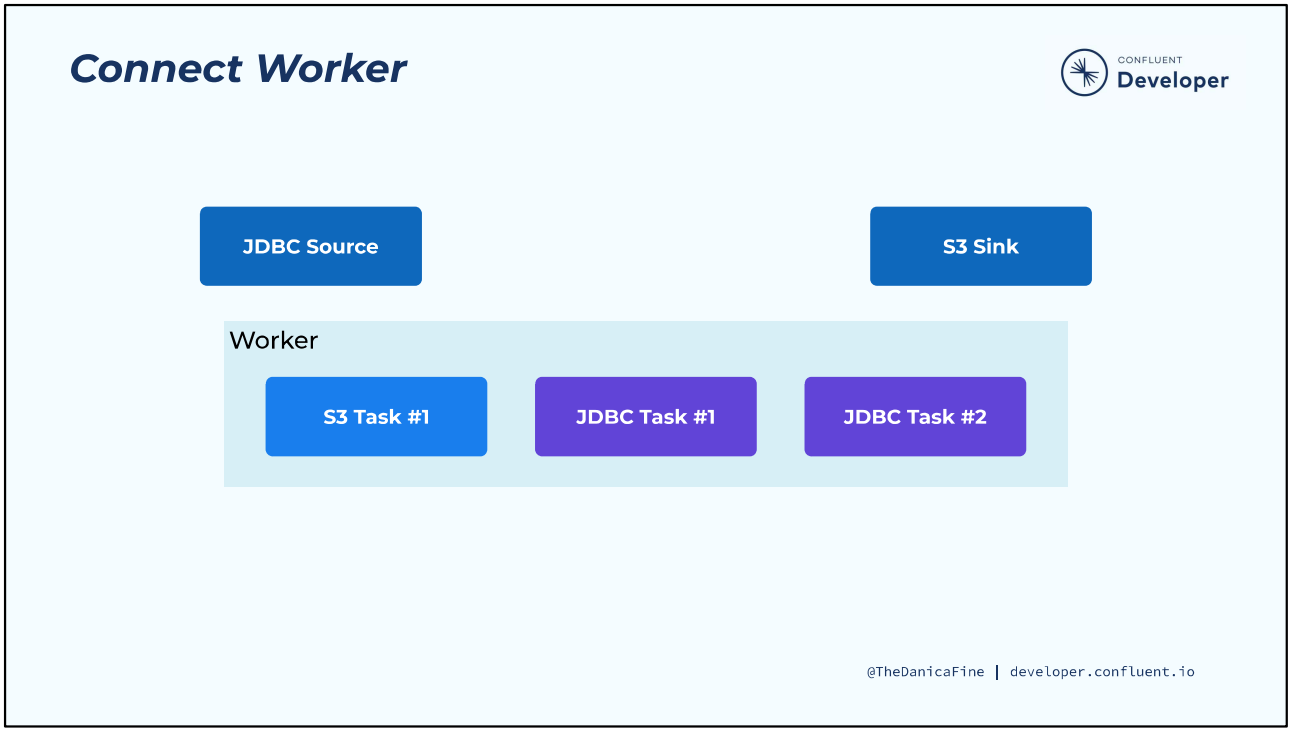


In this module, you will learn about the distributed and standalone deployment methods for Kafka Connect. But the requirement to deploy Kafka Connect infrastructure only applies if you are implementing self-managed Kafka Connect. If you use Confluent managed connectors, all infrastructure deployment is taken care of by Confluent.

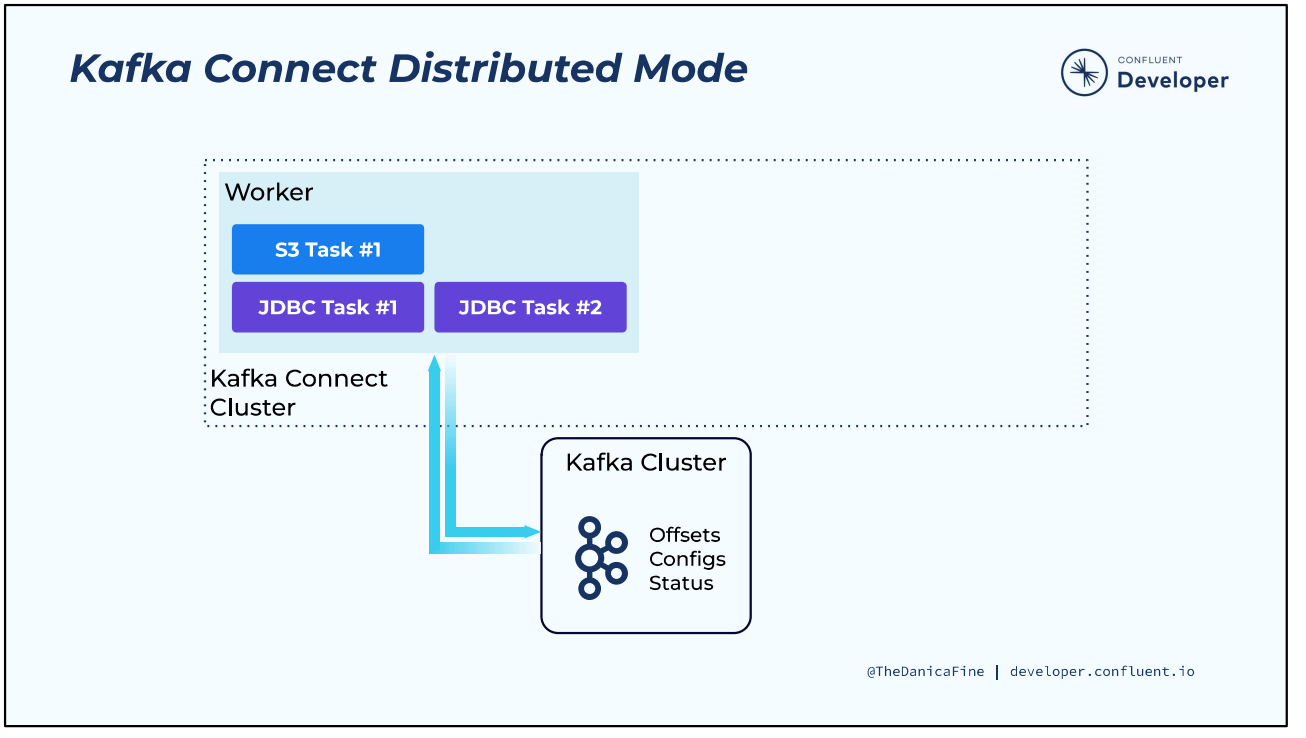
So now that we have learned a bit about the components within Kafka Connect, let's now turn to how we can actually run a connector. When we add a connector instance, we specify its logical configuration. It's physically executed by a thread known as a task. In this diagram we see two logical connectors, each with one task.



The execution of a connector's ingest or egress of data can also be parallelized (if the connector supports it). In that case, additional tasks are spawned. This could mean that when ingesting from a database, multiple tables are read at once, or when writing to a target data store, data is read concurrently from multiple partitions of the underlying Kafka Topic to increase throughput.



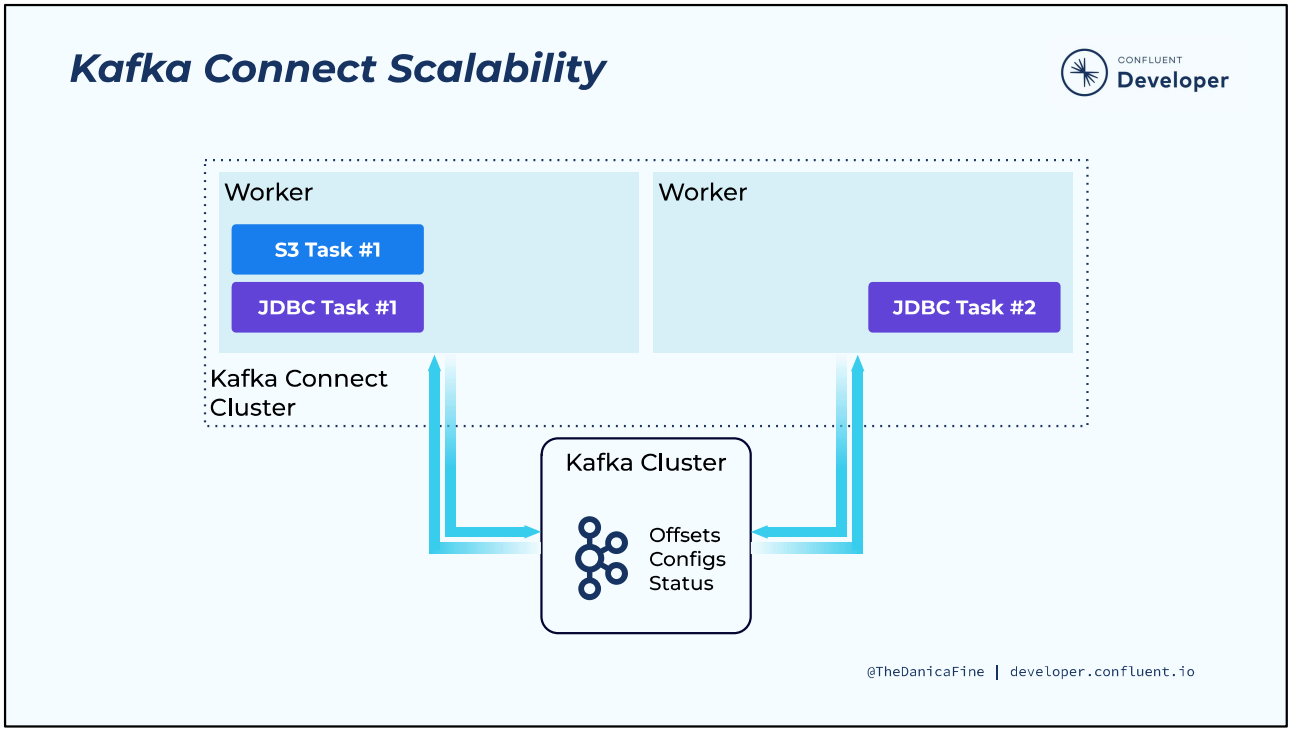
But where do the tasks actually run? Kafka Connect runs under the Java virtual machine (JVM) as a process known as a worker. Each worker can execute multiple connectors. When you look to see if Kafka Connect is running, or want to look at its log file, it's the Worker process that you're looking at. Tasks are executed by Kafka Connect Workers.



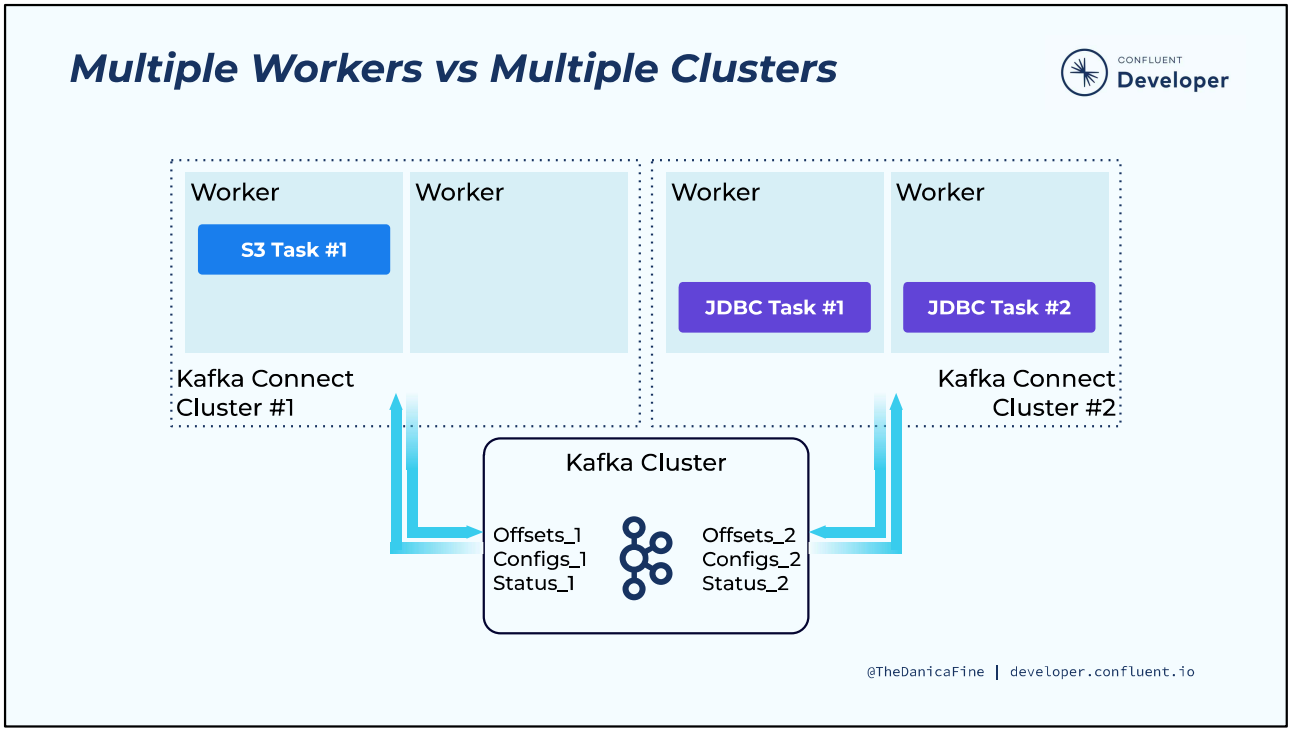
A Kafka Connect worker can be run in one of two deployment methods: standalone or distributed. Each has its pros and cons, which we will now discuss. The way in which you configure and operate Kafka Connect in these two modes is different.

Despite its name, the distributed deployment mode is equally valid for a single worker deployed in a sandbox or development environment. In this mode, Kafka Connect uses Kafka topics to store state pertaining to connector configuration, connector status, and more. The topics are configured to retain this information indefinitely, known as compacted topics. Connector instances are created and managed via the REST API that Kafka Connect offers.

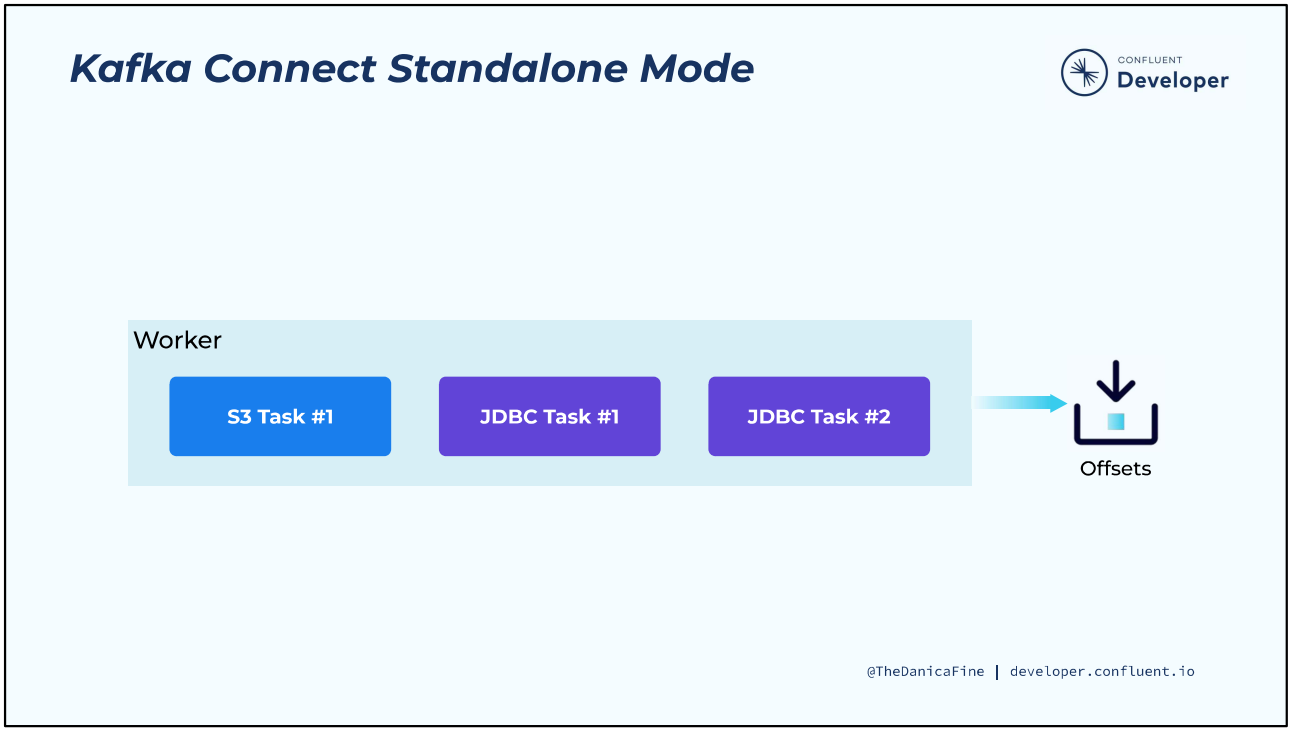
The distributed mode is the recommended best practice for most use cases.



Since all offsets, configs, and status information for the distributed mode cluster is maintained in Kafka topics, this means that you can add additional workers easily, as they can read everything that they need from Kafka. When you add workers from a Kafka Connect cluster, the tasks are rebalanced across the available workers to distribute the workload. If you decide to scale down your cluster (or even if something outside your control happens and a worker crashes), Kafka Connect will rebalance again to ensure that all the connector tasks are still executed.



The minimum number of workers recommended is two so that you have fault tolerance. But of course, you can add additional workers to the cluster as your throughput needs increase. You can opt to have fewer, bigger clusters of workers, or you may choose to deploy a greater number of smaller clusters in order to physically isolate workloads. Both are valid approaches and are usually dictated by organizational structure and responsibility for the respective pipelines implemented in Kafka Connect.



On the other hand, in standalone mode, the Kafka Connect worker uses files to store its state. Connectors are created from local configuration files, not the REST API. Consequently, you cannot cluster workers together, meaning that you cannot scale for throughput or have fault-tolerant behavior.

Because there is no clustering, you can know for certain on which machine a connector's task will be executing (i.e., the machine on which you've deployed the standalone worker). This means that standalone mode is appropriate if you have a connector that needs to execute with server locality, for example, reading from files on a particular machine or ingesting data sent to a network port at a fixed address.

You can satisfy this same requirement using Kafka Connect in the distributed mode with a single worker Connect cluster. This provides the benefit of having offsets, configs, and status information stored in a Kafka topic.

Hi, Danica Fine here; Kafka Connect runs as a JVM process, so there are a ton of options for how and where we can run it. But let's see how to run Kafka Connect in Docker.

Kafka Connect Images on Docker Hub

CONFLUENT Developer

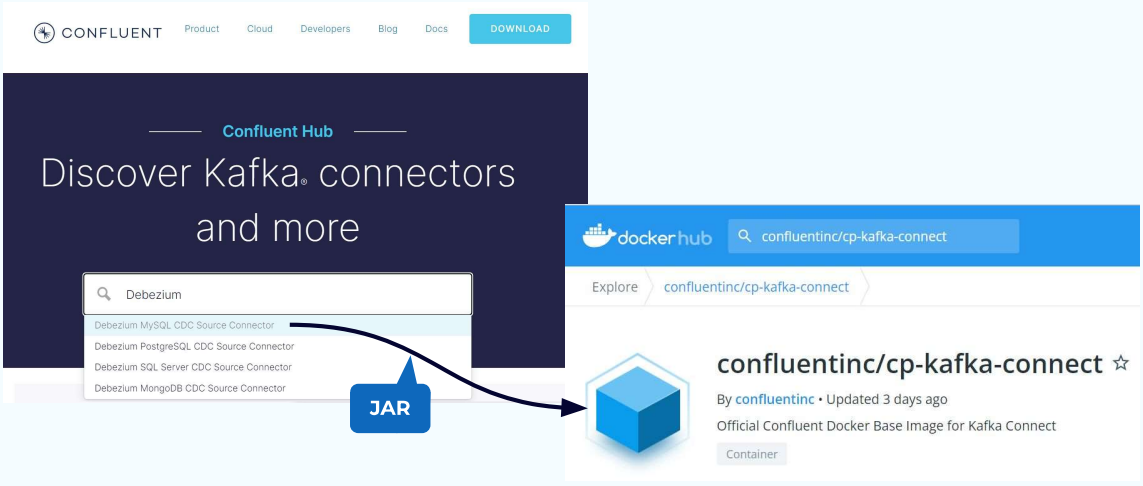
@TheDanicaFine | developer.confluent.io

You can run a Kafka Connect worker directly as a JVM process on a virtual machine or bare metal, but you might prefer the convenience of running it in a container, using a technology like Kubernetes or Docker. Note that containerized Connect via Docker will be used for many of the examples in this series.

Confluent maintains its own image for Kafka Connect, cp-kafka-connect, which provides a basic Connect worker to which you can add your desired JAR files for sink and source connectors, single message transforms, and converters.

Adding Connectors to a Container

CONFLUENT Developer



@TheDanicaFine | developer.confluent.io

You can use Confluent Hub to add your desired JARs, either by installing them at runtime or by creating a new Docker image. Of course, there are pros and cons to either of these options, and you should choose based on your individual needs.

Add to Container Image at Runtime

CONFLUENT Developer

```
kafka-connect:
  image: confluentinc/cp-kafka-connect:7.1.0-1-ubi8
  environment:
    CONNECT_PLUGIN_PATH: /usr/share/java,/usr/share/confluent-hub-components

  command:
    - bash
    - -c
    - |
      confluent-hub install --no-prompt neo4j/kafka-connect-neo4j:2.0.2
      /etc/confluent/docker/run
```


Adding your dependencies at runtime means that you don't have to create a new image, but it does increase installation time each time your container is run, and it also requires an internet connection. It's a good option for prototyping work but probably not for a production deployment.

Your JARs should be in a location that causes them to be class loadable by the Connect process, and you'll need to add an environmental variable that identifies their location (note that in production you will likely have many more environmental variables than just this one). Also make sure to specify the correct version of the Connect base image. Finally, you should add a command that overrides the base image's default command so that you can call the Confluent Hub utility, which will install the connectors specified (in this case, the Neo4j connector).

Build a New Container Image

CONFLUENT Developer

```
FROM confluentinc/cp-kafka-connect:7.1.0-1-ubi8

ENV CONNECT_PLUGIN_PATH: "/usr/share/java,/usr/share/confluent-hub-components"

RUN confluent-hub install --no-prompt neo4j/kafka-connect-neo4j:2.0.2
```


The second way to add dependencies, and the option probably most often used in production deployments, is to build a new image.

Make sure to use the correct Confluent base image version and also check the specific documentation for each of your connectors.

Add Connector Instance at Container Launch

CONFLUENT Developer

- You can add a connector instance using a launch script that does the following:
  - Launch Connect
  - Wait for the Connect listener to respond
  - Add the connector instance using the Connect REST API

Typically, you will add connector instances once the worker process is running by manually submitting the configuration or via an external automation. However, you may find—perhaps for demo purposes—that you want a self-sufficient container that also adds the connector instance when it starts. To do this, you can use a launch script that looks like this:

“To do this, you can use a script that launches connect, waits for the connect listener to respond, and adds the connector instance using the connect rest api.”

```
# Launch Kafka Connect
/etc/confluent/docker/run &
#
# Wait for the Connect listener to respond
echo "Waiting for Kafka Connect to start listening on localhost ⏳"
while : ; do
  curl_status=$$(curl -s -o /dev/null -w %{http_code}
http://localhost:8083/connectors)
```

```
  echo -e $$(date) " Kafka Connect listener HTTP state: "
$$curl_status " (waiting for 200)"
  if [ $$curl_status -eq 200 ] ; then
    break
  fi
  sleep 5
done

# Add the connector instance using the Connect REST API
echo -e "\n--\n+> Creating Data Generator source connector
instance"
curl -s -X PUT -H  "Content-Type:application/json"
http://localhost:8083/connectors/source-datagen-01/config \
    -d '{
    "connector.class":
"io.confluent.kafka.connect.datagen.DatagenConnector",
    "key.converter":
"org.apache.kafka.connect.storage.StringConverter",
    "kafka.topic": "ratings",
    "max.interval":750,
    "quickstart": "ratings",
    "tasks.max": 1
}'
sleep infinity
```

In this exercise, we'll learn how to run and maintain a self-managed connector with Confluent Cloud.

Hands On: Run a Self-Managed Connector in Docker

CONFLUENT Developer

@TheDanicaFine | developer.confluent.io

Exercise Environment Preparation Steps

Complete the following steps to set up the environment used for the course exercises. Prior to doing so, you will need to sign up for Confluent Cloud at https://confluent.cloud.

Note: Steps 1-15 are the same as those included in 6. Hands On: Confluent Cloud Managed Connector API and 7. Hands On: Confluent Cloud Managed Connector CLI. If you already completed them for either exercise, you can skip to step 16 of this exercise setup.

We will use various CLI during the course exercises including `confluent` so these need to be available on the machine you plan to run the exercise on. Downloading Confluent Platform will accomplish this.

Running a Self-Managed Connector with Confluent Cloud exercise steps

0:30 As mentioned previously in this course, the rate at which managed connectors are being added to Confluent Cloud is impressive, but you may find that the connector you want to use with Confluent Cloud isn't yet available. This is one scenario where you will need to run your own Kafka Connect worker which then connects to Confluent Cloud. In this exercise, we will do exactly that.

- 0:50 Establish a self-managed Kafka Connect cluster
    - This cluster will run in Docker containers and be configured to write to and read from Kafka topics in our

    - kc-101 Confluent Cloud cluster
- 1:00 Create Kafka Connect data pipelines
    - Pipeline one will use a self-managed Debezium MySQL source connector to stream data from our local MySQL database to a Kafka topic in Confluent Cloud
    - Pipeline two will use a self-managed Elasticsearch sink connector to stream data from a Kafka topic in our kc-101 Confluent Cloud cluster to a local instance of Elasticsearch
    - Pipeline three will use a self-managed Neo4j sink connector to stream data from a Kafka topic in our kc-101 Confluent Cloud cluster to a local instance of Neo4j

1:34 Our self-managed Kafka Connect cluster will require the Confluent Cloud Java configuration settings we previously obtained. We will make them available via environment variables. We will do this using the delta_configs/env.delta that was created in the exercise setup steps.

1:48 Before we do this though, let's review the contents of java.config.

- Run command:

```
cat ~/.confluent/java.config
```

1:54 Observe the settings, including the Confluent Cloud cluster endpoint that is used for the bootstrap servers property. It also includes values for the cluster API key and secret which clients need to authenticate with Confluent Cloud. The settings also include the Schema Registry endpoint and API key and secret. These are also needed by clients.

2:14 Now, let's review the contents of delta_configs/env.delta.

- Run commands:

```
cd ~/learn-kafka-courses/kafka-connect-101
cat delta_configs/env.delta
```

2:18 As you can see, this file contains numerous export commands that create environment variables set to the value of a corresponding Java client configuration setting.

2:27 We will establish these variables for our current command shell.

- Run command:

```
source delta_configs/env.delta
```

2:30 And now we will verify the previous step was successful.

- Run command:

- Run command:

```
code docker-compose.yml
```

2:48 The first Docker container named connect-1 is one of the two Kafka Connect worker nodes we will be working with. Let's now examine several configuration settings that are specified in the container's environment: settings.

- 3:01 In line 15, CONNECT_BOOTSTRAP_SERVERS is set equal to the $BOOTSTRAP_SERVERS which is one of the environment variables we made available in step 3. This is the endpoint address of the kc-101 Confluent Cloud cluster.
- 3:12 In line 16, we see that the group.id property is set to kc-101. This is what tells the connect-1 worker node which Connect cluster it should join upon startup. All other worker nodes with the same group.id value belong to the same Connect cluster.
- 3:27 In lines 17-19, we see three internal topics the worker nodes use to keep in sync with one another. The names for these topics need to be unique for each Connect cluster.
- 3:37 In lines 29-31, three more environment variables are used to set values used by the Connect workers to connect to Schema Registry which in our case is located in Confluent Cloud.
- 3:48 In lines 46, 51, and 56 we see another environment variable used to set the value that the worker as well as the underlying producer and consumer will use to authenticate with the Confluent Cloud cluster.
- 3:57 In lines 64-67 we see that the connector plugins that we will be using during our exercises are being downloaded from Confluent Hub and installed on the connect-1 worker node. The connect-2 worker node container configuration also includes these same plugin installation steps. Connector plugins must be installed on every node in the connect cluster.

4:17 Scrolling down, we will find that the environment variables are also used to define the connect-2 worker node as well as the local control-center node.

4:22 Scrolling down further, you will see additional Docker containers defined including:

- MySQL
- Elasticsearch
- Neo4j

4:31 Let's now start the Kafka Connect cluster as well as the other Docker containers.

- Run command:

```
docker-compose up -d
```

4:43 And finally verify the Kafka Connect workers are ready.

- Run command:

```
bash -c ' \
echo -e "\n\n=============\nWaiting for Kafka Connect to
start listening on localhost ⏳\n=============\n"
while [ $(curl -s -o /dev/null -w %{http_code}
http://localhost:8083/connectors) -ne 200 ] ; do
  echo -e "\t" $(date) " Kafka Connect listener HTTP
state: " $(curl -s -o /dev/null -w %{http_code}
http://localhost:8083/connectors) " (waiting for 200)"
  sleep 15
Done
echo -e $(date) "\n\n--------------\n\o/ Kafka Connect
is ready! Listener HTTP state: " $(curl -s -o /dev/null
-w %{http_code} http://localhost:8083/connectors)
"\n--------------\n"'
```

4:47 Make sure that the Elasticsearch, Debezium, and Neo4j connector plugins are available.

- Run command:

```
curl -s localhost:8083/connector-plugins | jq
'.[].class'|egrep
'Neo4jSinkConnector|MySqlConnector|ElasticsearchSinkConn
ector'
```

- Expected command output:

```
    "io.confluent.connect.elasticsearch.ElasticsearchSinkCon
nector"
    "io.debezium.connector.mysql.MySqlConnector"
    "streams.kafka.connect.sink.Neo4jSinkConnector"
```

Note: If jq isn't present on your machine, the output from this command and others in this exercise that use jq will not appear as expected.

4:53 Let's now verify that our self-managed Kafka Connect cluster is using our Confluent Cloud cluster.

- In the Confluent Cloud console, navigate to the kc-101 cluster, expand Data Integration, select Clients, and then select Consumers.

Stream data from MySql to a Kafka Topic

5:04 Next, we will stream data from our local MySQL database to a Confluent Cloud Kafka topic using the Debezium MySQL Source connector. We will start by viewing a sample of records in the database.

- In a new terminal window or tab, run command:

```
docker exec -t mysql bash -c 'echo "SELECT * FROM ORDERS
ORDER BY CREATE_TS DESC LIMIT 10 \G" | mysql -u root
-p$MYSQL_ROOT_PASSWORD demo'
```

5:18 These are the records that we will stream to a Kafka topic in Confluent Cloud. Before we continue, let's start a process to generate additional rows to the MySQL database.

- In the same terminal window or tab, run command:

```
docker exec mysql /data/02_populate_more_orders.sh
```

5:28 Let's now create the MySQL source connector instance.

- Return to the original terminal window and run command:

```
curl -i -X PUT -H  "Content-Type:application/json" \

http://localhost:8083/connectors/source-debezium-orders-
01/config \
    -d '{
            "connector.class":
"io.debezium.connector.mysql.MySqlConnector",
            "value.converter":
"io.confluent.connect.json.JsonSchemaConverter",
            "value.converter.schemas.enable": "true",
            "value.converter.schema.registry.url":
"'$SCHEMA_REGISTRY_URL'",

"value.converter.basic.auth.credentials.source":
"'$BASIC_AUTH_CREDENTIALS_SOURCE'",
            "value.converter.basic.auth.user.info":
"'$SCHEMA_REGISTRY_BASIC_AUTH_USER_INFO'",
            "database.hostname": "mysql",
            "database.port": "3306",
            "database.user": "kc101user",
            "database.password": "kc101pw",
            "database.server.id": "42",
```

```
            "database.history.kafka.bootstrap.servers":
      "'$BOOTSTRAP_SERVERS'",

      "database.history.consumer.security.protocol":
      "SASL_SSL",
            "database.history.consumer.sasl.mechanism":
"PLAIN",

      "database.history.consumer.sasl.jaas.config":
      "org.apache.kafka.common.security.plain.PlainLoginModule
      required username=\"'$CLOUD_KEY'\"
      password=\"'$CLOUD_SECRET'\";",

      "database.history.producer.security.protocol":
      "SASL_SSL",
            "database.history.producer.sasl.mechanism":
"PLAIN",

      "database.history.producer.sasl.jaas.config":
      "org.apache.kafka.common.security.plain.PlainLoginModule
      required username=\"'$CLOUD_KEY'\"
      password=\"'$CLOUD_SECRET'\";",
            "database.history.kafka.topic":
"dbhistory.demo",
            "topic.creation.default.replication.factor":
"3",
            "topic.creation.default.partitions": "3",
            "decimal.handling.mode": "double",
            "include.schema.changes": "true",
            "transforms": "unwrap,addTopicPrefix",
            "transforms.unwrap.type":
      "io.debezium.transforms.ExtractNewRecordState",

      "transforms.addTopicPrefix.type":"org.apache.kafka.conne
      ct.transforms.RegexRouter",
            "transforms.addTopicPrefix.regex":"(.*)",

      "transforms.addTopicPrefix.replacement":"mysql-debezium-
      $1"
        }'
```

5:34 Let's check the status of the connector instance.

- Run command:

```
curl -s
"http://localhost:8083/connectors?expand=info&expand=sta
tus" | \
       jq '. | to_entries[] | [ .value.info.type, .key,
.value.status.connector.state,.value.status.tasks[].stat
e,.value.info.config."connector.class"]|join(":|:")' | \
       column -s : -t| sed 's/\"//g'| sort
```

- Expected output:

```
source  |  source-debezium-orders-01  |  RUNNING  |
RUNNING  |  io.debezium.connector.mysql.MySqlConnector
```

5:41 Let's view the records in the Confluent Cloud Kafka topic.

- In the Confluent Cloud console, navigate to the kc-101 cluster, select Topics, select the mysql-debezium-asgard.demo.ORDERS topic, and select the Messages tab.

Observe the records being written to the topic by the MySQL source connector.

5:49 Before continuing, let's stop the MySQL data generator we have running.

- In the second terminal window/tab where the data generator is running, press Ctrl+C to stop it.
- Close this terminal window/tab.

Stream Data from Kafka to Elasticsearch

5:55 Our next step in this exercise is to stream the data that was sourced from the MySQL database to an Elasticsearch sink.

6:02 To start, we will create an Elasticsearch sink connector instance. Notice that for this connector, we set tasks.max equal to 2. This is primarily for instructional purposes since the load doesn't really need to be spread out across our two connect worker nodes. We will examine how these multiple tasks are distributed in later exercises.

- In our terminal window, run command:

```
curl -i -X PUT -H  "Content-Type:application/json" \

http://localhost:8083/connectors/sink-elastic-orders-01/
```

```
        "io.confluent.connect.json.JsonSchemaConverter",
                    "value.converter.schemas.enable": "true",
                    "value.converter.schema.registry.url":
        "'$SCHEMA_REGISTRY_URL'",

        "value.converter.basic.auth.credentials.source":
        "'$BASIC_AUTH_CREDENTIALS_SOURCE'",
                    "value.converter.basic.auth.user.info":
        "'$SCHEMA_REGISTRY_BASIC_AUTH_USER_INFO'",
                    "topics":
"mysql-debezium-asgard.demo.ORDERS",
                    "connection.url":
"http://elasticsearch:9200",
                    "key.ignore": "true",
                    "schema.ignore": "true",
                    "tasks.max": "2"
            }'
```

6:24 Let's check the status of the connector instance.

- Run command:

```
curl -s
"http://localhost:8083/connectors?expand=info&expand=sta
tus" | \
       jq '. | to_entries[] | [ .value.info.type, .key,
.value.status.connector.state,.value.status.tasks[].stat
e,.value.info.config."connector.class"]|join(":|:")' | \
       column -s : -t| sed 's/\"//g'| sort | grep
ElasticsearchSinkConnector
```

- Expected output:

```
source  |  source-debezium-orders-01  |  RUNNING  |
RUNNING  | RUNNING  |
io.confluent.connect.elasticsearch.ElasticsearchSinkConn
ector
```

6:27 Now let's inspect the data that is written to Elasticsearch.

- Run command:

```
curl -s
http://localhost:9200/mysql-debezium-asgard.demo.orders/
```

```
    -H 'content-type: application/json' \
    -d '{ "size": 5, "sort": [ { "CREATE_TS": { "order":
"desc" } } ] }' |\
    jq '.hits.hits[]._source | .id, .CREATE_TS'
```

Stream Data from Kafka to Neo4j

6:33 Our next step in this exercise is to stream the data that was sourced from the MySQL database to a Neo4j sink. Notice that for this connector, we set tasks.max equal to 2.

- In our terminal window, run command:

```
curl -i -X PUT -H  "Content-Type:application/json" \
http://localhost:8083/connectors/sink-neo4j-orders-01/config
\
    -d '{
            "connector.class":
"streams.kafka.connect.sink.Neo4jSinkConnector",
          "value.converter":
"io.confluent.connect.json.JsonSchemaConverter",
            "value.converter.schemas.enable": "true",
            "value.converter.schema.registry.url":
"'$SCHEMA_REGISTRY_URL'",

"value.converter.basic.auth.credentials.source":
"'$BASIC_AUTH_CREDENTIALS_SOURCE'",
            "value.converter.basic.auth.user.info":
"'$SCHEMA_REGISTRY_BASIC_AUTH_USER_INFO'",
            "topics":
"mysql-debezium-asgard.demo.ORDERS",
            "tasks.max": "2",
            "neo4j.server.uri": "bolt://neo4j:7687",
            "neo4j.authentication.basic.username":
"neo4j",
            "neo4j.authentication.basic.password":
"connect",

"neo4j.topic.cypher.mysql-debezium-asgard.demo.ORDERS":
"MERGE (city:city{city: event.delivery_city}) MERGE
(customer:customer{id: event.customer_id,
delivery_address: event.delivery_address, delivery_city:
```

```
event.delivery_city, delivery_company:
event.delivery_company}) MERGE (vehicle:vehicle{make:
event.make, model:event.model}) MERGE
(city)<-[:LIVES_IN]-(customer)-[:BOUGHT{order_total_usd:
event.order_total_usd,order_id:event.id}]->(vehicle)"
        } '
```

6:44 Let's check the status of the connector instance.

- Run command:

```
curl -s
"http://localhost:8083/connectors?expand=info&expand=sta
tus" | \
       jq '. | to_entries[] | [ .value.info.type, .key,
.value.status.connector.state,.value.status.tasks[].stat
e,.value.info.config."connector.class"]|join(":|:")' | \
       column -s : -t| sed 's/\"//g'| sort | grep
Neo4jSinkConnector
```

- Expected output:

```
source  |  source-debezium-orders-01  |  RUNNING  |
RUNNING  | RUNNING  |
io.confluent.connect.elasticsearch.Neo4jSinkConnector
```

6:50 In this exercise we:

- Implemented a self-managed Kafka Connect cluster and associated it with a Confluent Cloud cluster
- Created a source connector instance that consumed records from a local MySQLdatabase and wrote corresponding records to a Kafka topic in the Confluent Cloud cluster
- Created two sink connector instances that consumed records from a Kafka topic in the Confluent Cloud cluster and wrote corresponding records out to a local Elasticsearch instance and a local Neo4j instance

7:22 If you run this demonstration yourself, you need to tear down the environment after doing so to avoid unnecessarily accruing cost to the point your promotional credits are exhausted.

Let's walk through that tear down process now for this environment.

First, we will shut down the Docker containers and free the resources they are using.

let's set the cluster context for the `confluent` CLI.

- Run commands:
  `confluent kafka cluster list`

  `confluent kafka cluster use <kc-101 cluster ID>`

We can now use the list command to identify the topic names we need to delete.

- Run command:

  `confluent kafka topic list`

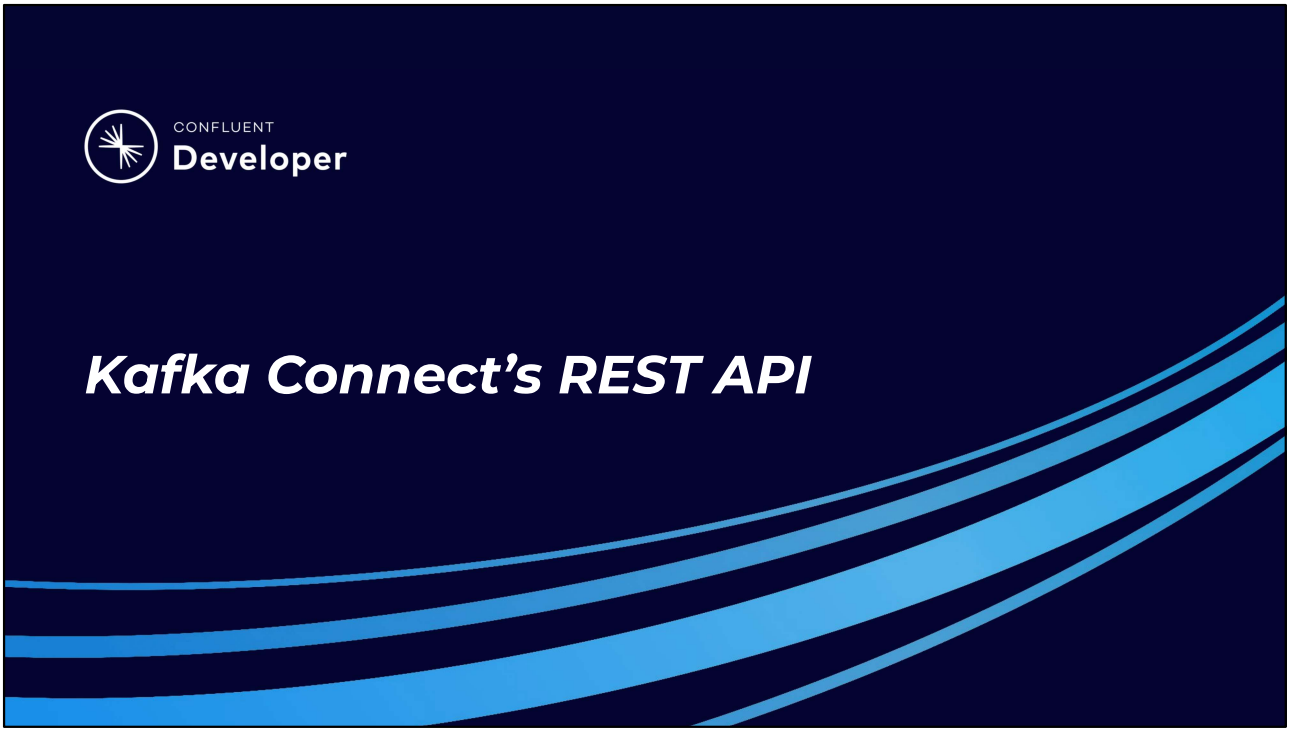
And now we can delete each of these topics.

- Run command:

  `confluent kafka topic delete <topic>`

Repeat command for each topic listed in the previous step.

This concludes this exercise.



In this hands-on demonstration, we'll walk through features of the Confluent Connect REST API using basic command line examples.

Getting Basic Connect Cluster Information

0:16 Get basic Connect cluster information including the worker version, the commit that it's on, and its Kafka cluster ID with the following command:

```
curl http://localhost:8083/
```

0:26 Note that the cluster ID sets this cluster apart from other Connect clusters that may be running a separate set of connectors.

Listing Installed Plugins

0:33 The command below lists the plugins that are installed on the worker. Note that plugins need to be installed first in order to be called at runtime later.

```
curl -s localhost:8083/connector-plugins
```

0:42 Kafka does ship with a few plugins, but generally you will need to install plugins yourself. The best place to get them is Confluent Hub, where you will find a large number of plugins and a command line tool to install them. Recall that the Docker containers for our two Connect workers included commands to download and install four connector plugins from Confluent Hub.

Formatting the Result of the Installed Plugin List

```
curl -s localhost:8083/connector-plugins | jq '.'
```

1:13 Now it is much easier to see the details for each of the available plugins. These plugins need to be installed on all workers in the Connect cluster so that if a connector instance or task is moved to a worker due to a rebalance, the plugin is available to run it.

Create a Connector Instance

1:26 To create a connector instance, you PUT or POST a JSON file with the connector's configuration to a REST endpoint on your Connect worker. PUT is somewhat easier because it will create the connector if it doesn't exist, or update it if it already exists. If it already exists and there's no update to make, it won't error—so PUT is the idempotent way of updating a connector.

```
curl -i -X PUT -H  "Content-Type:application/json" \

http://localhost:8083/connectors/source-debezium-orders-
00/config \
    -d '{
            "connector.class":
"io.debezium.connector.mysql.MySqlConnector",
            "value.converter":
"io.confluent.connect.json.JsonSchemaConverter",
            "value.converter.schemas.enable": "true",
            "value.converter.schema.registry.url":
"'$SCHEMA_REGISTRY_URL'",

"value.converter.basic.auth.credentials.source":
"'$BASIC_AUTH_CREDENTIALS_SOURCE'",
            "value.converter.basic.auth.user.info":
"'$SCHEMA_REGISTRY_BASIC_AUTH_USER_INFO'",
            "database.hostname": "mysql",
            "database.port": "3306",
            "database.user": "debezium",
            "database.password": "dbz",
            "database.server.id": "42",
            "database.server.name": "asgard",
            "table.whitelist": "demo.orders",
            "database.history.kafka.bootstrap.servers":
"'$BOOTSTRAP_SERVERS'",

"database.history.consumer.security.protocol":
```

```
      "SASL_SSL",
                  "database.history.consumer.sasl.mechanism":
"PLAIN",

      "database.history.consumer.sasl.jaas.config":
      "org.apache.kafka.common.security.plain.PlainLoginModule
      required username=\"'$CLOUD_KEY'\"
      password=\"'$CLOUD_SECRET'\";",

      "database.history.producer.security.protocol":
      "SASL_SSL",
                  "database.history.producer.sasl.mechanism":
"PLAIN",

      "database.history.producer.sasl.jaas.config":
      "org.apache.kafka.common.security.plain.PlainLoginModule
      required username=\"'$CLOUD_KEY'\"
      password=\"'$CLOUD_SECRET'\";",
                  "database.history.kafka.topic":
"dbhistory.demo",
                  "topic.creation.default.replication.factor":
"3",
                  "topic.creation.default.partitions": "3",
                  "decimal.handling.mode": "double",
                  "include.schema.changes": "true",
                  "transforms": "unwrap,addTopicPrefix",
                  "transforms.unwrap.type":
      "io.debezium.transforms.ExtractNewRecordState",

      "transforms.addTopicPrefix.type":"org.apache.kafka.conne
      ct.transforms.RegexRouter",
                  "transforms.addTopicPrefix.regex":"(.*)",

      "transforms.addTopicPrefix.replacement":"mysql-debezium-
      $1"
          }'
```

List Connector Instances

1:50 Use the following command to list of all extant connectors:

```
      curl -s -XGET "http://localhost:8083/connectors/"
```

Inspect Config and Status for a Connector

1:54 Inspect the config for a given connector as follows:

```
curl -i -X GET -H  "Content-Type:application/json" \

http://localhost:8083/connectors/sink-elastic-orders-00/config
```

2:02 You can also look at a connector's status. While the config command shows a connector's static configuration, the status shows the connector as a runtime entity:

```
curl -s
"http://localhost:8083/connectors?expand=info&expand=status" | \
      jq '. | to_entries[] | [ .value.info.type, .key,
.value.status.connector.state,.value.status.tasks[].state, .value.info.config."connector.class"] |join(":|:")' |
\
      column -s : -t| sed 's/\"//g'| sort
```

Delete a Connector

2:12 If something is wrong in your setup and you don't think a config change would help, or if you simply don't need a connector to run anymore, you can delete it by name:

```
curl -s -XDELETE
"http://localhost:8083/connectors/sink-elastic-orders-00
"
```

2:24 Or you can make a nifty interactive delete list with the tool `peco` by piping the connector list `stdout` through it, finally `xarg-ing` to a `cURL` call to the delete API:

```
curl -s "http://localhost:8083/connectors" | \
        jq '.[]' | \
      peco | \
    xargs -I{connector_name} curl -s -XDELETE
"http://localhost:8083/connectors/"\{connector_name\}
```

It returns an interactive list of connector instances. To delete one, arrow down to

configuration (see Create a Connector Instance above). Because `PUT` is used to both create and update connectors, it's the standard command that you should use most of the time (which also means that you don't have to completely rewrite your configs).

Inspect Task Details

The following command returns the connector status:

```
curl -s -XGET
"http://localhost:8083/connectors/source-debezium-orders
-00/status" | jq '.'
```

2:36 If your connector fails, the details of the failure belong to the task. So to inspect the problem, you'll need to find the stack trace for the task. The task is the entity that is actually running the connector and converter code, so the state for the stack trace lives in it.

```
curl -s -XGET
"http://localhost:8083/connectors/source-debezium-orders
-00/tasks/0/status" | jq '.'
```

Restart the Connector and Tasks

3:08 If after inspecting a task, you have determined that it has failed and you have fixed the reason for the failure (perhaps restarted a database), you can restart the connector with the following:

```
curl -s -XPOST
"http://localhost:8083/connectors/source-debezium-orders
-00/restart"
```

3:18 Keep in mind though that restarting the connector doesn't restart all of its tasks. You will also need to restart the failed task and then get its status again as follows:

```
curl -s -XPOST
"http://localhost:8083/connectors/source-debezium-orders
-00/tasks/0/restart"
```

```
curl -s -XGET
"http://localhost:8083/connectors/source-debezium-orders
-00/tasks/0/status" | jq '.'
```

all of its tasks at exactly the same time. The tasks are running in a thread pool, so there's no fancy mechanism to make this happen simultaneously.

3:43 A connector and its tasks can be paused as follows:

```
curl -s -XPUT
"http://localhost:8083/connectors/source-debezium-orders
-00/pause"
```

3:54 Just as easily, a connector and its tasks can be resumed:

```
curl -s -XPUT
"http://localhost:8083/connectors/source-debezium-orders
-00/resume"
```

Display All of a Connector's Tasks

4:01 A convenient way to display all of a connector's tasks at once is as follows:

```
curl -s -XGET
"http://localhost:8083/connectors/sink-neo4j-orders-00/t
asks" | jq '.'
```

4:06 This information is similar to what you can get from other APIs, but it is broken down by task, and configs for each are shown.

Get a List of Topics Used by a Connector

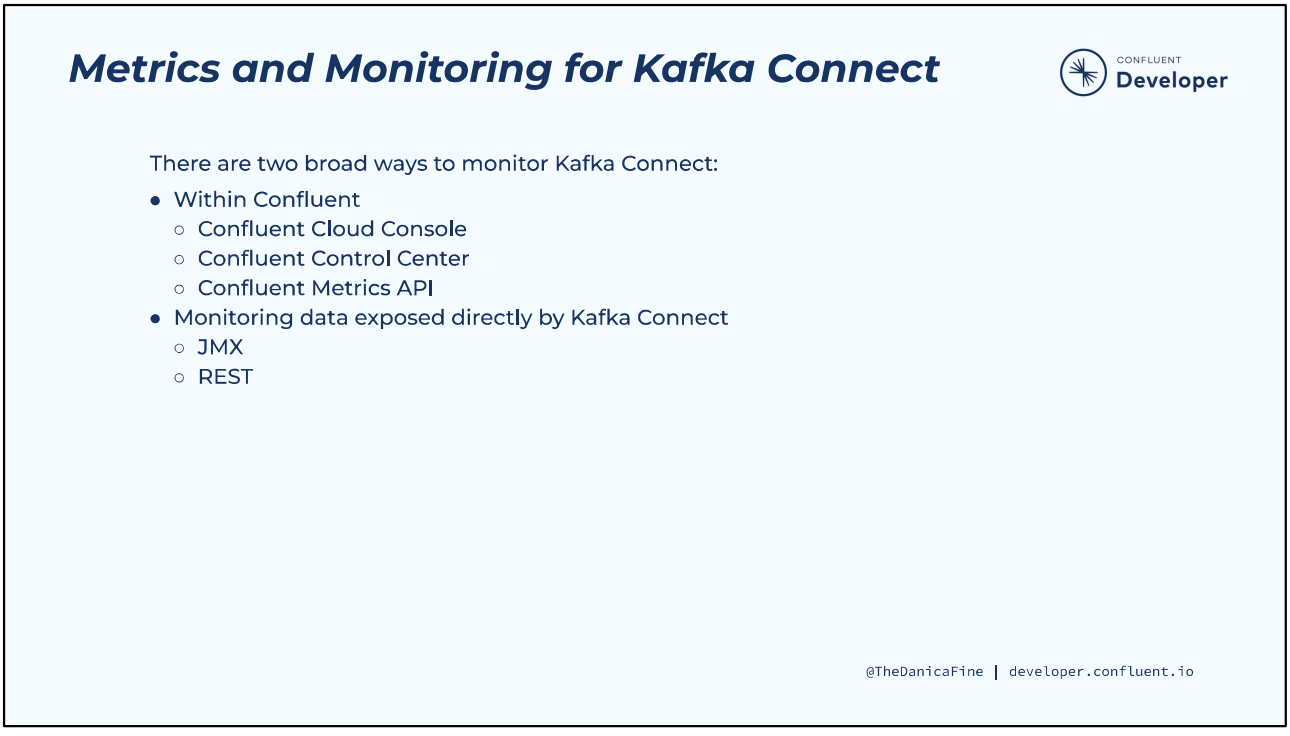
4:20 As of Apache Kafka 2.5, it is possible to get a list of topics used by a connector:

```
curl -s -XGET
"http://localhost:8083/connectors/source-debezium-orders
-00/topics" | jq '.'
```

4:23 This shows the topics that a connector is consuming from or producing to. This may not be particularly useful for connectors that are consuming from or producing to a single topic. However, some developers, for example, use regular expressions for topic names in Connect, so this is a major benefit in situations where topic names are derived computationally.

This could also be useful with a source connector that is using SMTs to dynamically

Hi, I'm Danica Fine, here to introduce you to methods for monitoring your Kafka Connect instances.
Now that you know a bit more about how to configure and run Kafka Connect, the next step is to understand how to monitor Connect clusters and connectors and act on any irregularities you may encounter. Let's dive in!

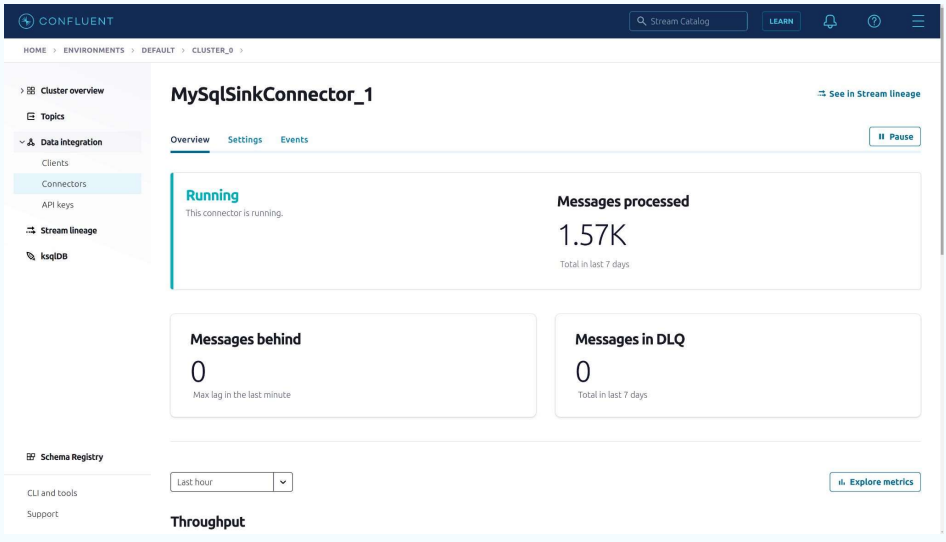


There are two broad ways to monitor Kafka Connect:

- Within the Confluent Kafka ecosystem, the Confluent Cloud Console and Confluent Platform Control Center are the easiest options to get started with monitoring a connector instance. The Confluent Metrics API is another option that can be used to collect metrics that may then be integrated with third-party monitoring tools such as Datadog, Dynatrace, Grafana, and Prometheus.
- Another option compatible for Confluent Kafka and Apache Kafka is to monitor data exposed directly by Kafka Connect, such as JMX and REST.

Managed Connector Overview

CONFLUENT Developer



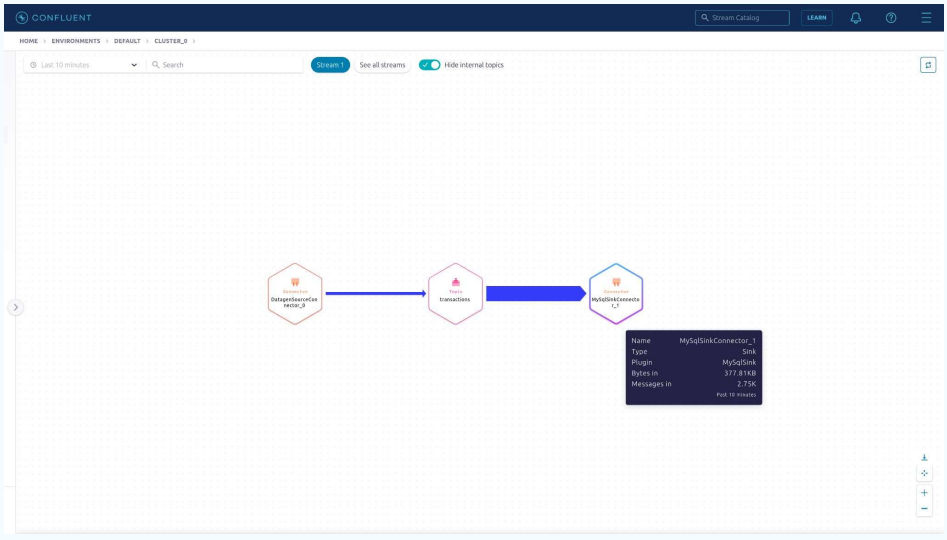
@TheDanicaFine | developer.confluent.io

From the Confluent Cloud UI, there are several views for monitoring an individual connector's status. A good place to start is the Connector Overview window. Here you can find a connector's current status, how many messages it has processed, whether there is any lag occurring, and also whether any potentially problematic messages have been written to the dead letter queue.

The Overview window also includes an option to open the Stream lineage window which shows where the connector fits within related event streams.

Confluent Stream Lineage

CONFLUENT Developer



@TheDanicaFine | developer.confluent.io

Stream lineage provides a graphical UI of event streams and data relationships with both a bird's eye view and drill-down magnification for answering questions like:

- Where did data come from?
- Where is it going?
- Where, when, and how was it transformed?

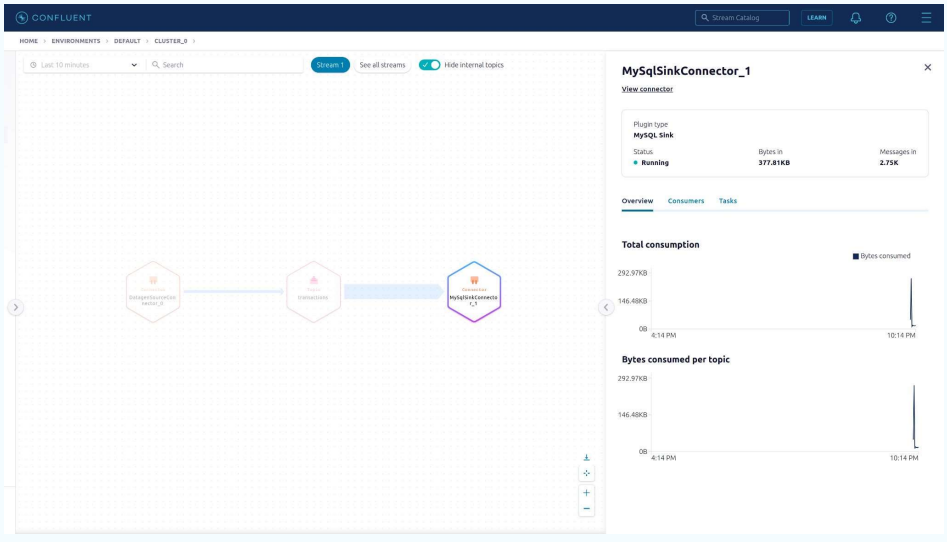
Viewing a connector in Stream lineage lets you easily identify its relationship within event streams.

- For source connectors, you can see what topic the connector is producing records to.
- For sink connectors, you can see what topic or topics the connector is consuming records from.

Mousing over the connector displays a popup with its details. If the connector or other elements of the event stream are clicked, a corresponding connector overview tab opens.

Stream Lineage - Connector Overview

CONFLUENT Developer

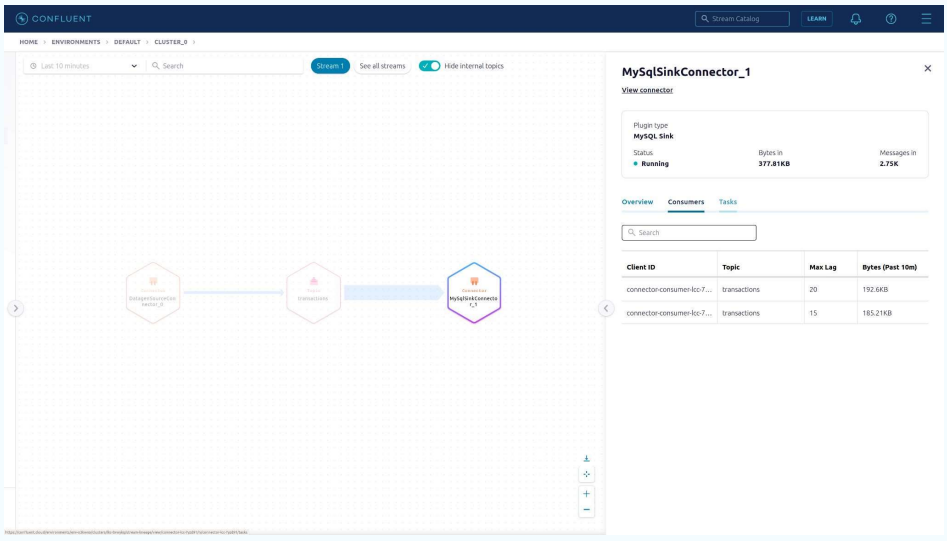


The Stream lineage connector overview tab displays much of the same details that the primary Connector Overview window displays.

A Consumers and a Tasks tab is also available.

Stream Lineage - Connector Consumers

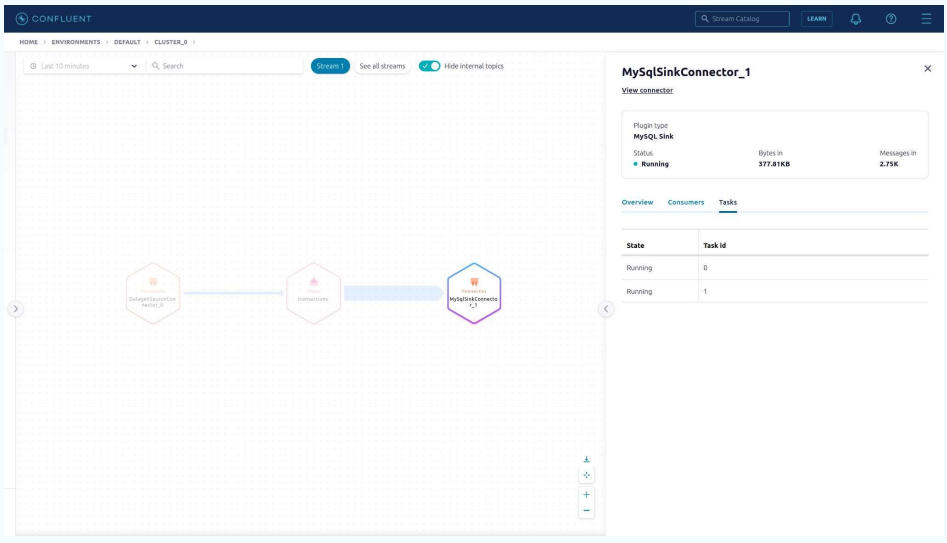
CONFLUENT Developer



The Consumers tab displays the list of consumer clients being used by the connector instance to read from the Kafka topic partitions. If a connector instance has multiple tasks, this list will contain multiple consumer clients.

Stream Lineage - Connector Tasks

CONFLUENT Developer

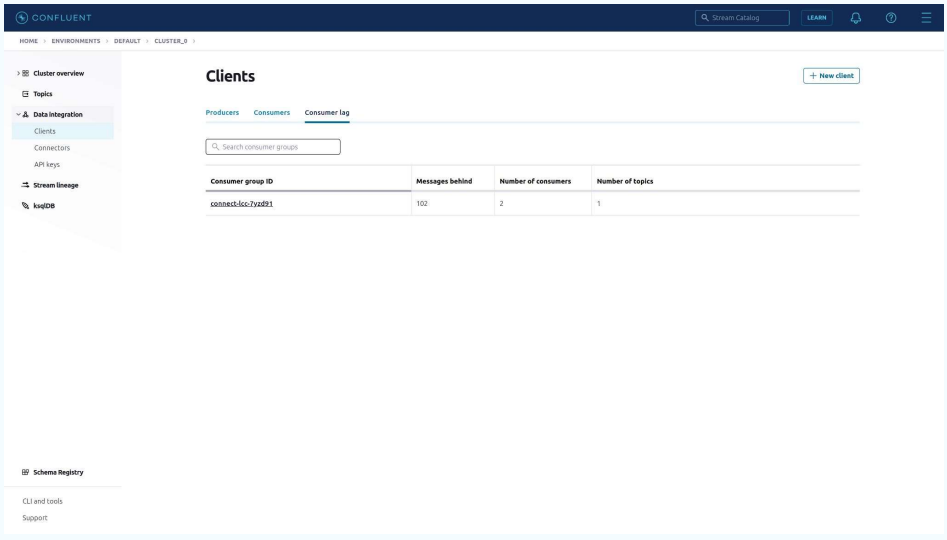


@TheDanicaFine | developer.confluent.io

The Tasks tab shows the connector's tasks and their status.

Confluent Consumer Lag Tab

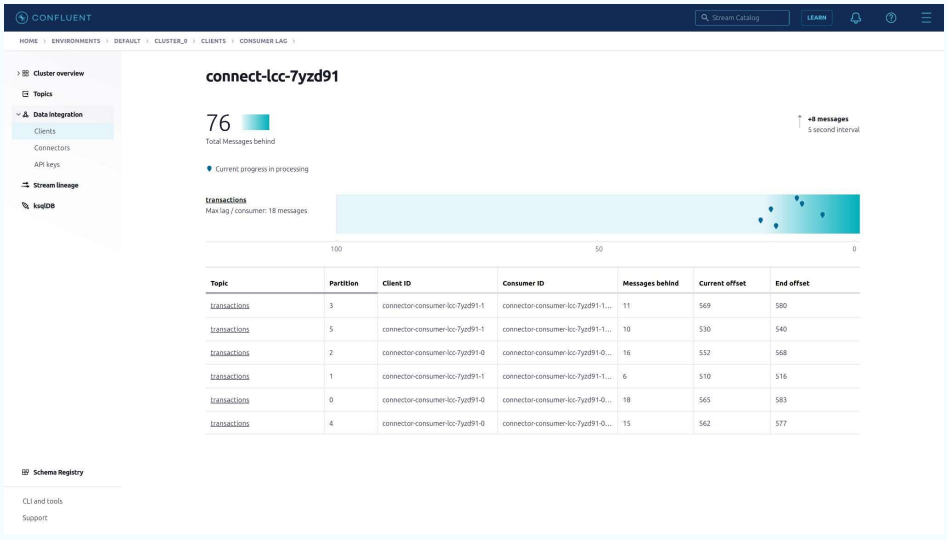
CONFLUENT Developer



You can view consumption lag information related to the connector by navigating to the Clients window under Data Integration.
To see the consumer lag for a particular connector, navigate to the Consumer lag tab and select the consumer group whose ID includes the connector ID.

Confluent Consumer Lag Details

CONFLUENT Developer



The window that appears shows the current lag for each partition of the topic being consumed by the connector.

Most of the connector information that is provided by the Confluent Cloud UI is available in the Confluent Control Center for self-managed connectors being run in conjunction with a Confluent Platform Kafka cluster.

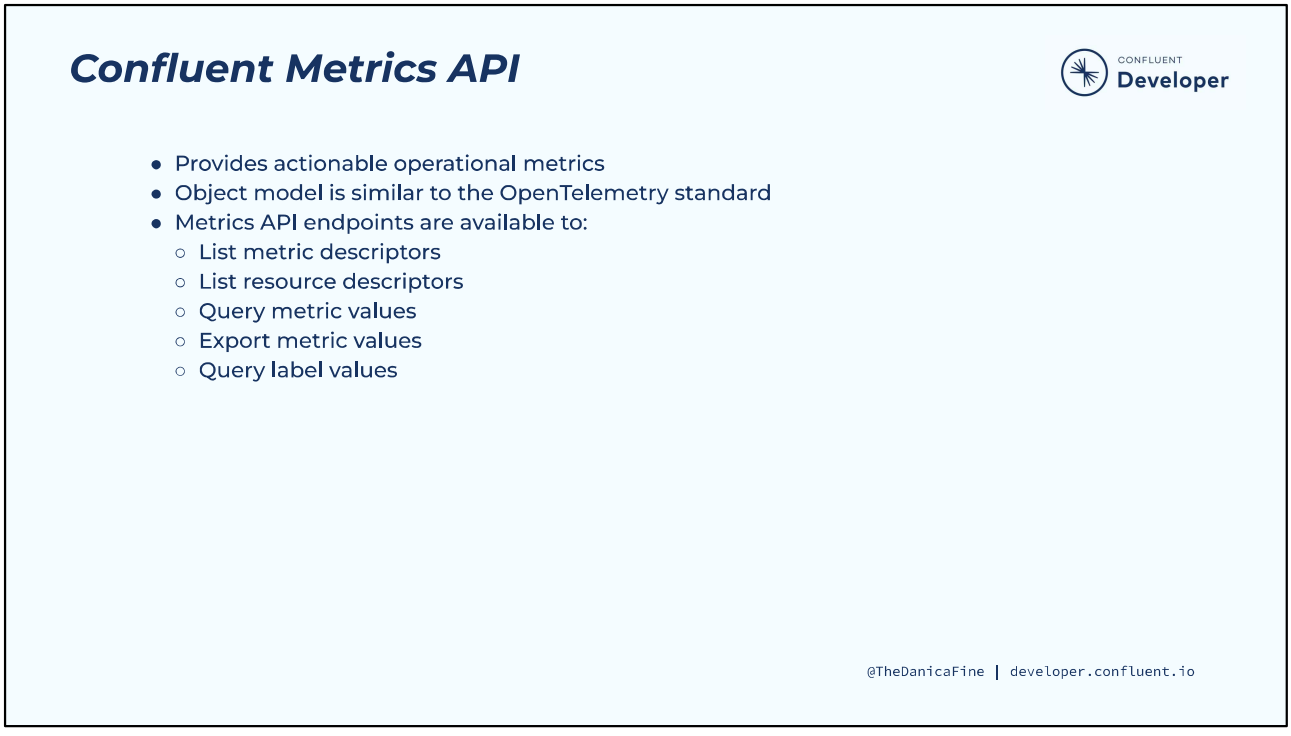
Third-Party Monitoring Integration

CONFLUENT Developer

- Datadog, Dynatrace, and Grafana Cloud
  - Input a Cloud API key
  - Select resources to monitor
  - See metrics in prebuilt dashboard
- Prometheus
  - Can scrape the Confluent Cloud Metrics API directly using the `export` endpoint
    - Endpoint returns the single most recent data point for each metric
  - Prometheus exposition or Open Metrics format

Datadog and Grafana Cloud provide integrations and Dynatrace provides an extension that allows users to input a Cloud API key, select resources to monitor, and see metrics in minutes in prebuilt dashboards.

Prometheus servers can scrape the Confluent Cloud Metrics API directly by making use of the export endpoint. This endpoint returns the single most recent data point for each metric, for each distinct combination of labels in the Prometheus exposition or Open Metrics format.

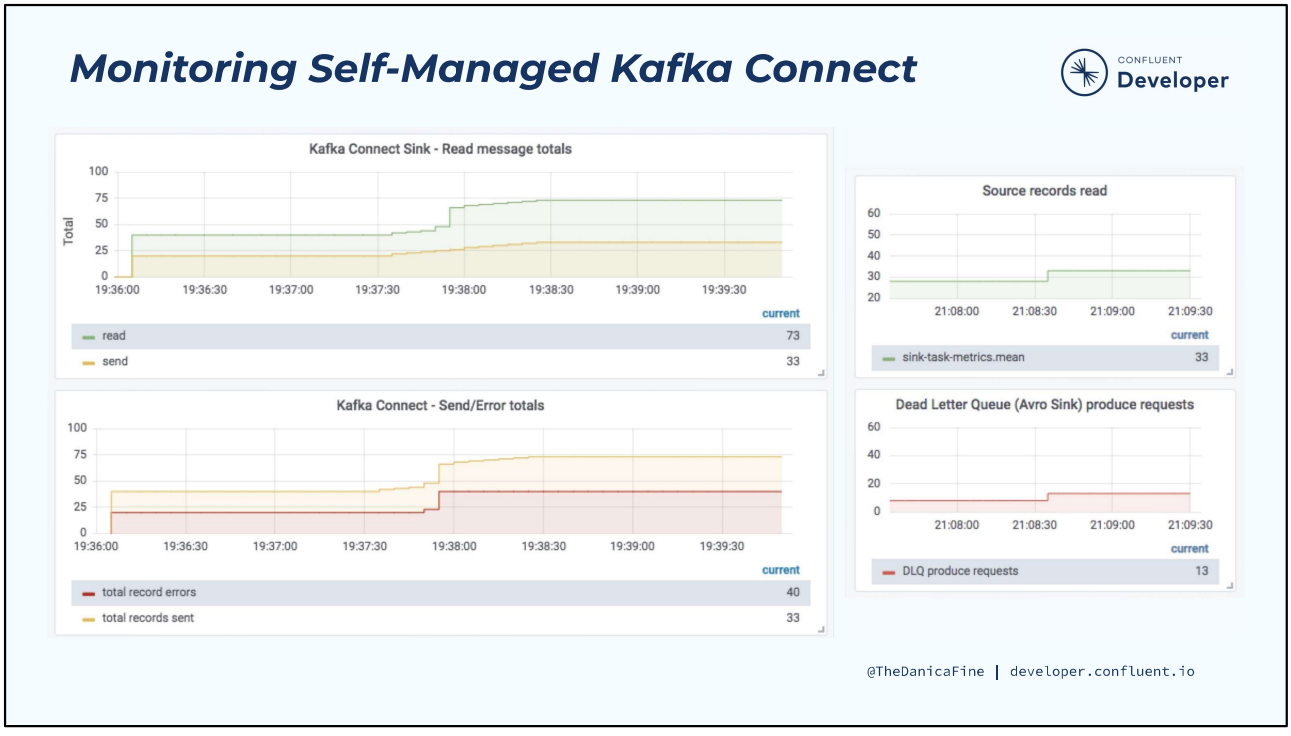


The Confluent Cloud Metrics API provides actionable operational metrics about your Confluent Cloud deployment. This is a queryable HTTP API in which the user will POST a query written in JSON and get back a time series of metrics specified by the query.

The object model for the Metrics API is designed similarly to the OpenTelemetry standard.

Metrics API endpoints are available to:

- List metric descriptors
- List resource descriptors
- Query metric values
- Export metric values
- Query label values



Kafka Connect exposes various data for monitoring over JMX and REST, and this collection is ever expanding (see, for example, KIP-475). This is an option when Confluent tools are not available, e.g. with self-managed Kafka Connect. To use JMX, you just need to be familiar with the MBeans that are exposed, and you need a tool for gathering data from them and visualizing it.

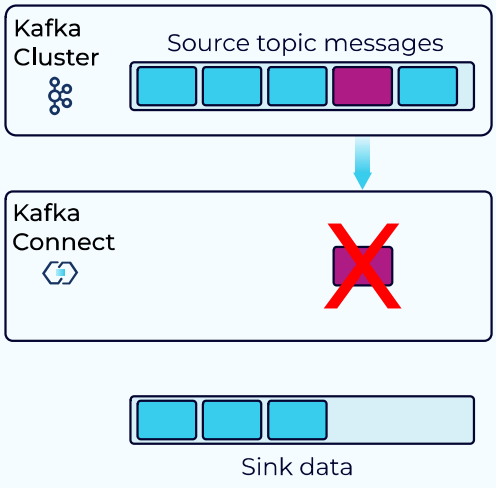
In the JMX-based setup in this image, we see data for total messages read by a sink connector, as well as information about Connect error totals, source records read, and dead letter queue requests. With a little bit of tooling, you can build alerting on this data and expand your own observability framework. All of this setup can be a lot of work, though, so if you can do it in a fully managed way, it is far easier.

Kafka Connect also exposes information about the status of tasks and connectors on its REST interface, which we covered in the previous module.

Hi, Danica Fine here; error-handling is bound to come up with any technology. Let's learn about our options for handling errors in Kafka Connect.

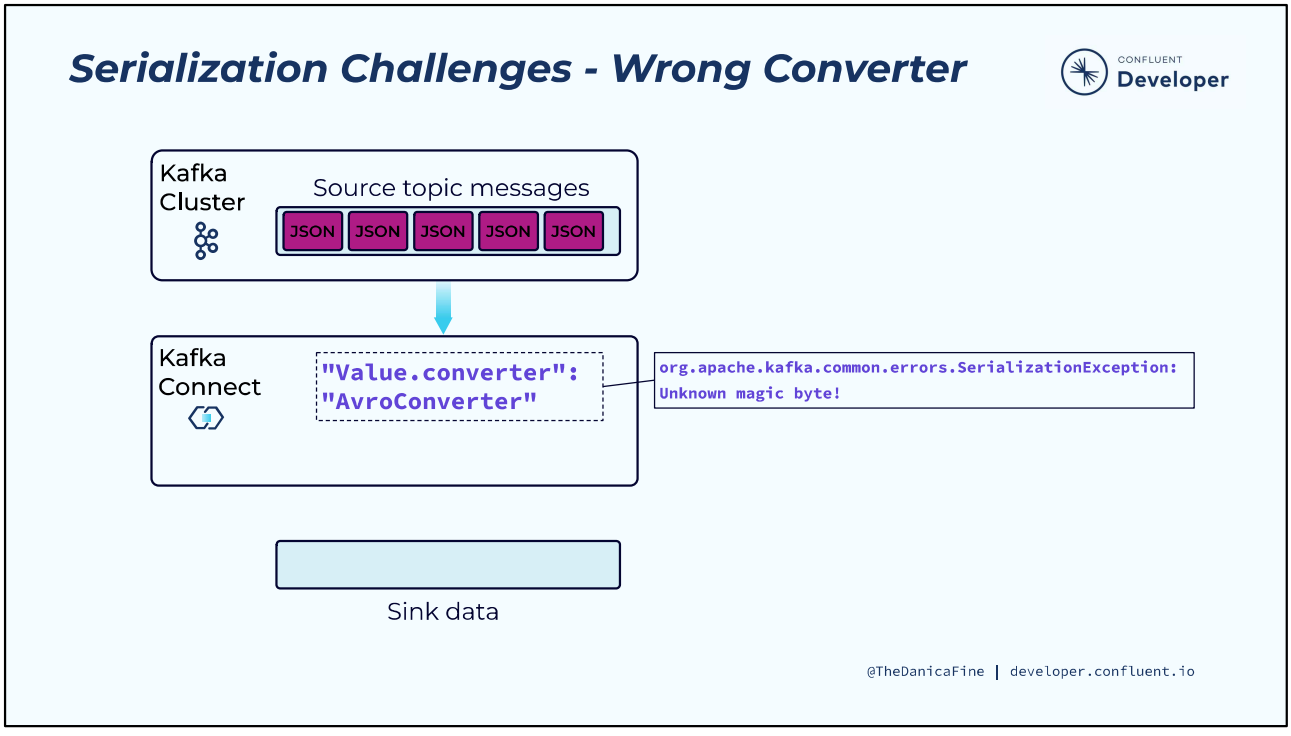
Error Handling in Kafka Connect

CONFLUENT Developer



The bottom line is that Kafka Connect supports several error-handling patterns, including fail fast, silently ignore, and dead letter queues.

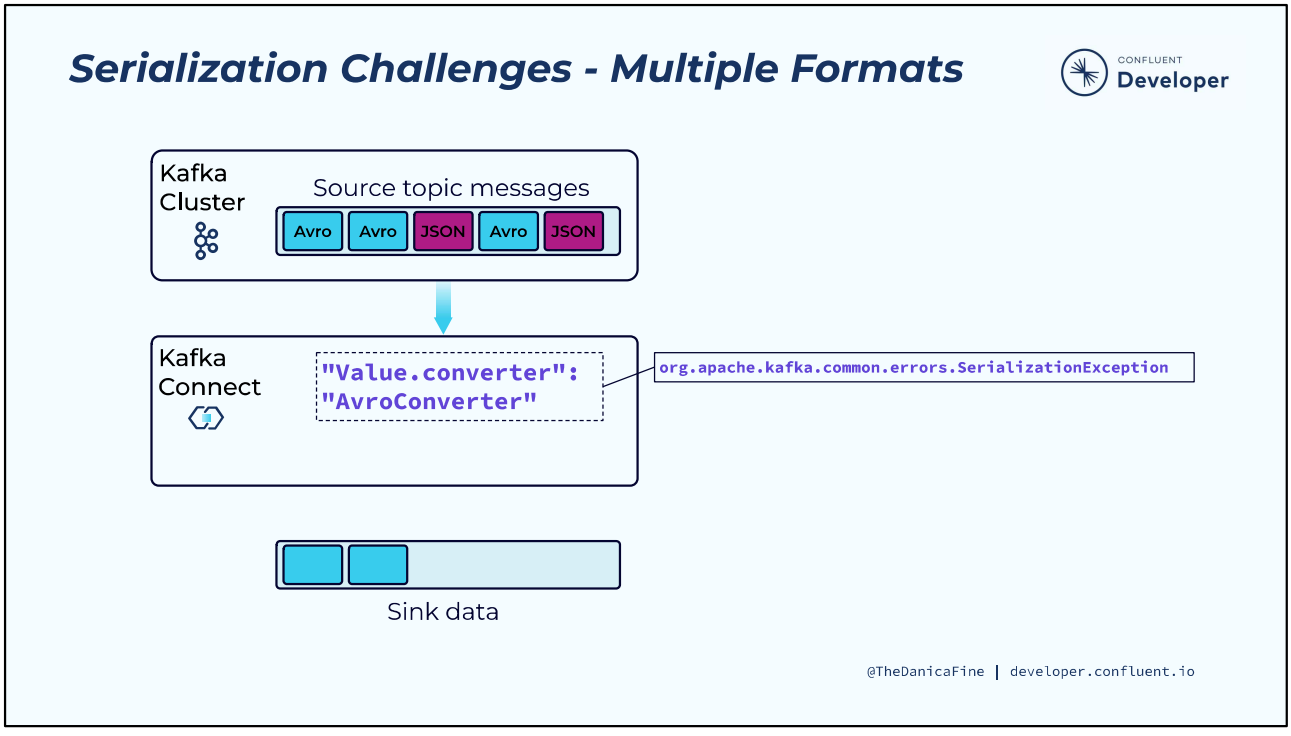
Obviously these different patterns are going to be useful in certain scenarios. Let's examine some cases where these error handling patterns are utilized.



As you set up and try various options with Connect, you might find that you have accidentally configured Connect to use an incorrect Converter. For example, if topic messages were serialized as JSON data and Connect attempts to deserialize them using the Avro converter, an “Unknown magic byte!” exception would be triggered.

In this situation, the Avro deserializer is trying to process a message from a Kafka topic, and the message is either not Avro, or it is Avro, but it wasn’t created with the Confluent Schema Registry serializer. Either way, the message doesn’t match the expected wire format, thus the reference to “magic bytes.” The error is quite specific, but basically it means that the data is in a format other than that which the Avro deserializer expects.

The simple fix in this case would be to update the Connector instance configuration to use the correct JSON Converter.

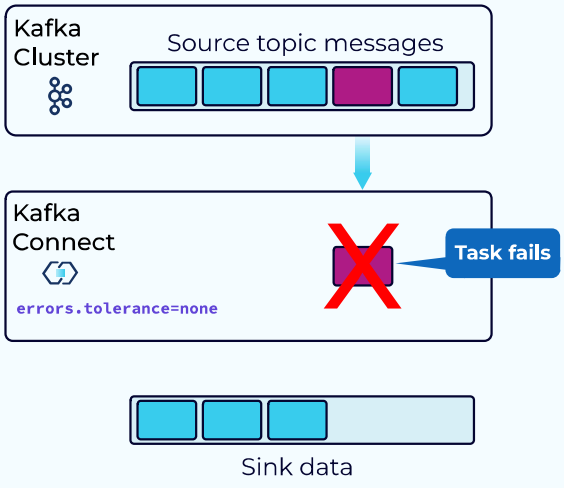


Another scenario you might come across is where a topic contains messages with multiple serialization formats. This might happen if the serialization method was changed for messages being written to a topic, for example at first the messages were serialized as JSON and then at some point this was changed to Avro. And if the producer clients were not all updated at the same time so that multiple producers were writing the topic using different serialization formats, the topic messages might alternate from one format to another. Since the connector instance can only be configured to use a single converter, exceptions will occur when the converter attempts to deserialize a message with the other format.

You can address both of these scenarios with configuration for error tolerances and dead letter queues. Let's take a look at these now.

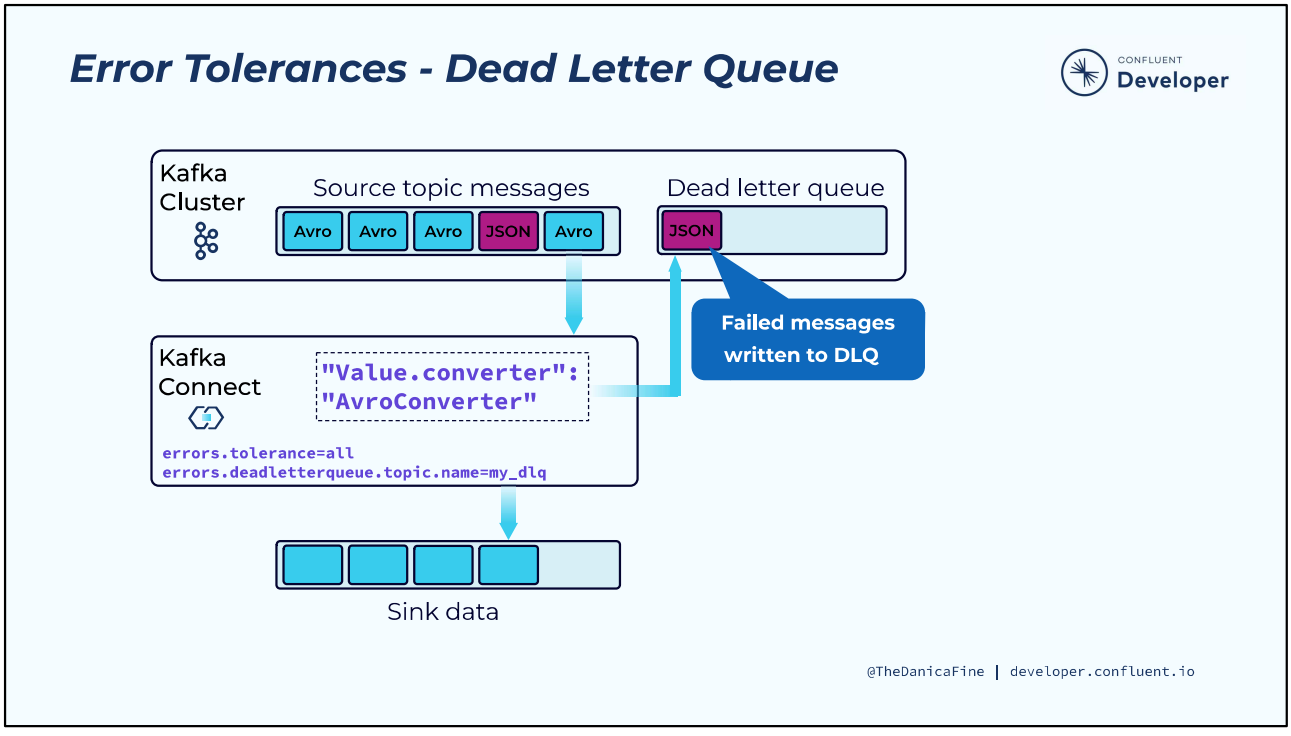
Error Tolerances - Fail Fast (Default)

CONFLUENT Developer



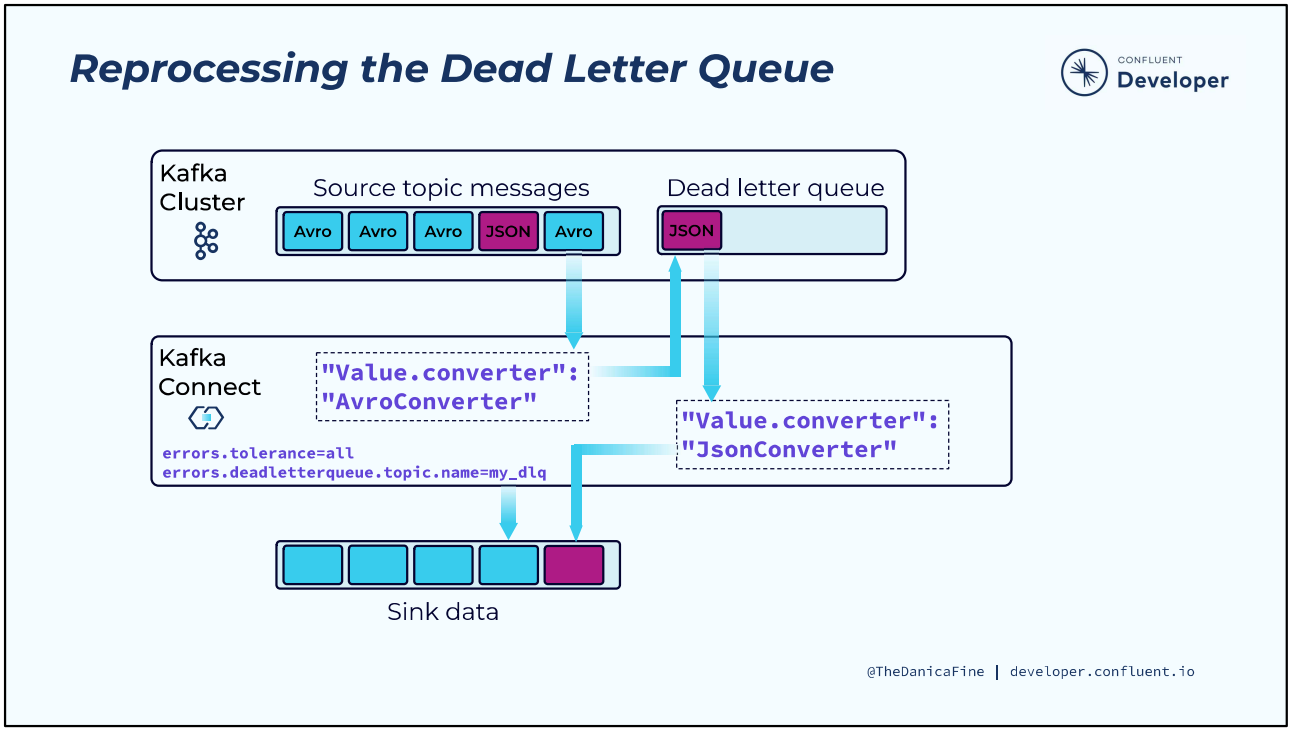
@TheDanicaFine | developer.confluent.io

By default, if Connect receives a serialization error like the ones that were just covered, the corresponding connector task is going to stop and you will have to deal with the issue and then restart it. This is safe behavior because if there's an invalid message, Connect won't process it.

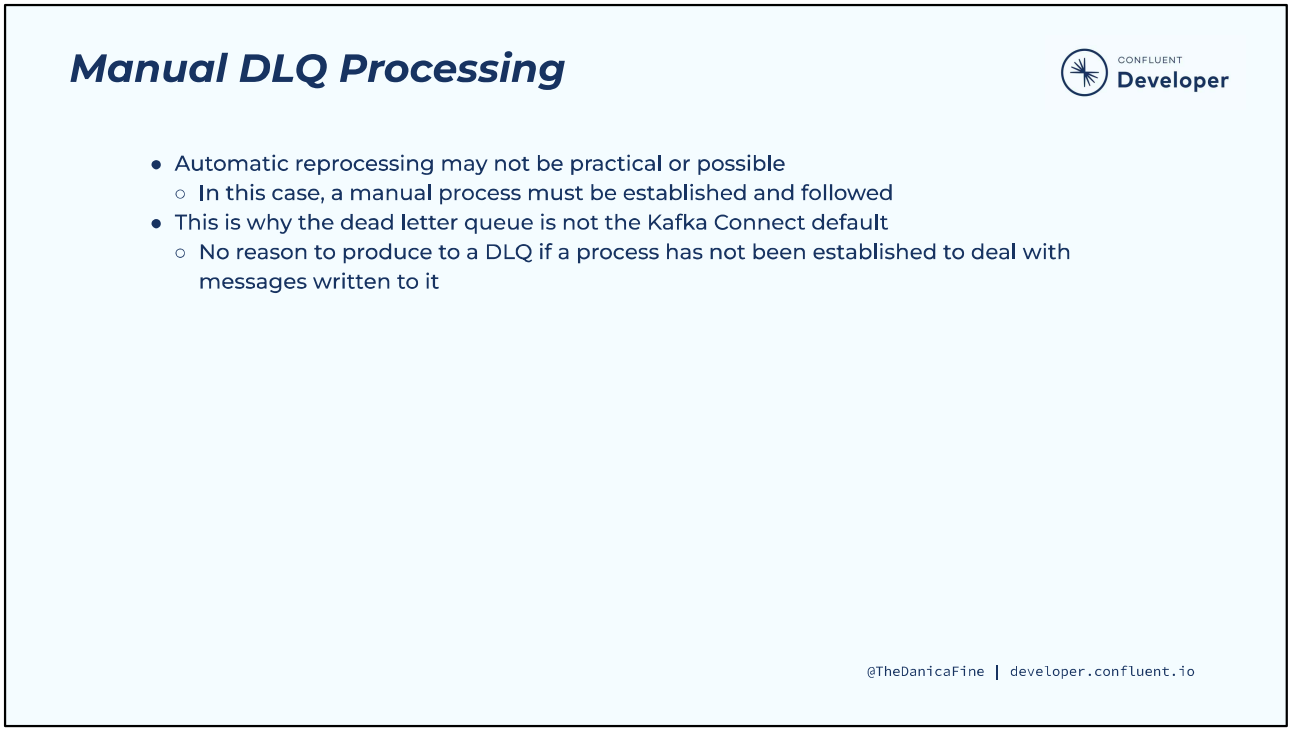


A dead letter queue in Kafka is just another Kafka topic to which messages can be routed by Kafka Connect if they fail processing in some way. The term is employed for its familiarity; a dead letter queue as traditionally conceived is part of an enterprise messaging system and is a place where messages are sent based on some routing logic that classifies them as having nowhere to go, and as potentially needing to be processed at a later time.

In Kafka Connect, the dead letter queue isn't configured automatically as not every connector needs one. Kafka Connect's dead letter queue is where failed messages are sent, instead of silently dropping them. Once the messages are there, you can inspect their headers, which will contain reasons for their rejection, and you can also look at their keys and values.



The dilemma whereby an Avro and a JSON producer are sending to the same topic has a solution in the dead letter queue. Basically you set the dead letter queue to receive the erroring messages, then reprocess them from the dead letter queue with the appropriate converter, and send them on to the sink. So, for example, if Avro messages are correctly proceeding to the sink, but JSON messages are erroring into the dead letter queue, you could add another connector with a JSON converter to process them out of the dead letter queue and send them on to the sink. This would allow you to complete the processing of the source topic, which is not possible with a single connector.



The dead letter queue reprocessing solution is easy to work out computationally, and it's simple to configure and get running. However, if the option doesn't exist for some reason or messages are failing for reasons that are hard to identify, you will have to try something else. You may be able to develop a consumer to convert the messages if you know how to deal with a particular failure mode, but often in these situations, you will end up having to manually handle the erroring messages—so you should have a process for a person to review them.

This is ultimately why dead letter queues aren't a default in Kafka Connect, because you need a way of dealing with the dead letter queue messages, otherwise you are just producing them somewhere for no reason. In effect, you should make sure you have thought through how you will process the results of a dead letter queue before you set one up.

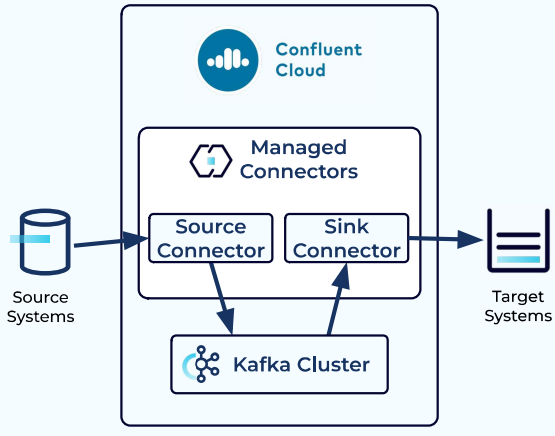
Module title slide style

Troubleshooting Managed Connectors

CONFLUENT Developer

The following tools can be used to troubleshoot managed connectors:

- Confluent Cloud UI
- Confluent CLI
- Confluent Connect API
- Connect log events

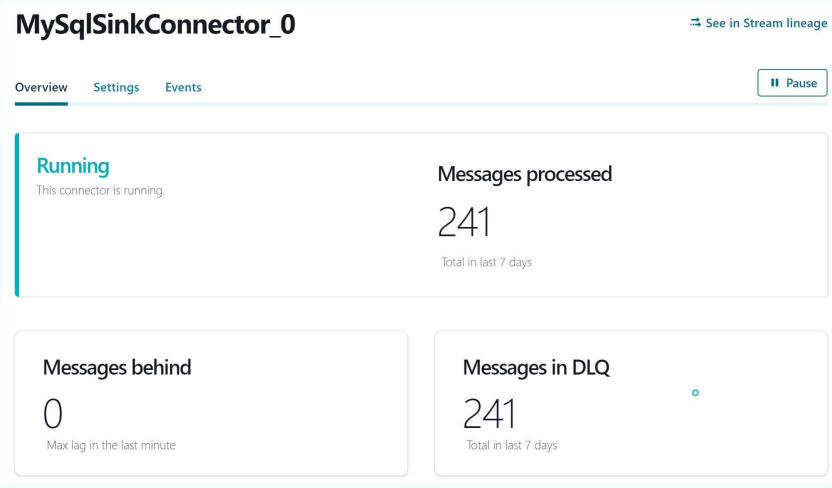


@TheDanicaFine | developer.confluent.io

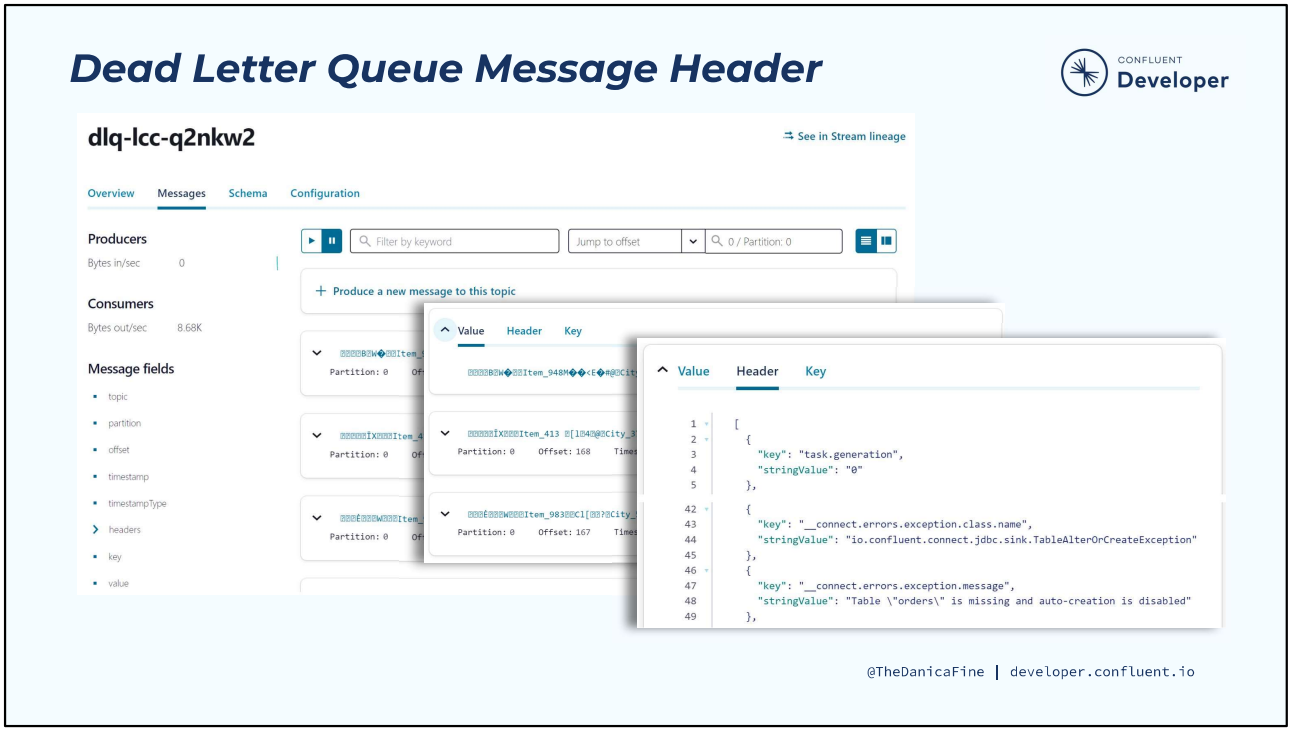
There are several tools that you can use to troubleshoot managed connectors. Depending upon the type of problem, one of these tools may work better than others. Let's now walk through a scenario taking a look at how each of these tools can be use to troubleshoot the problem.

Troubleshoot the Dead Letter Queue

CONFLUENT Developer



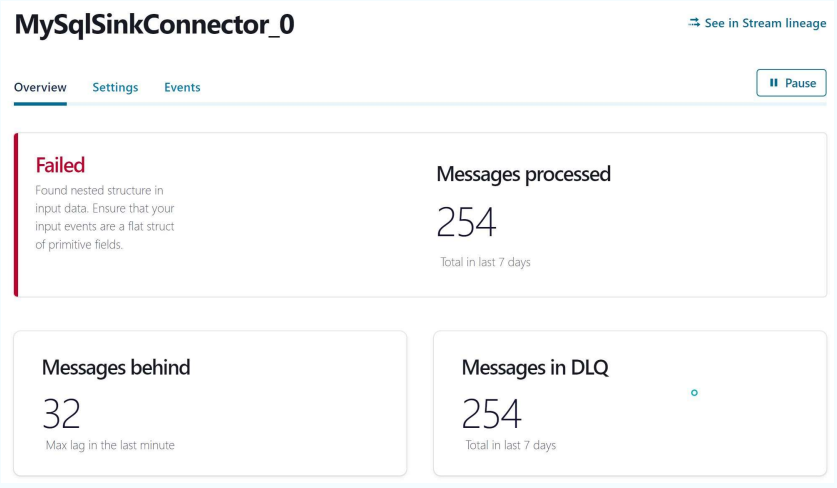
One problem that you may experience is a sink connector is unable to process the messages from the Kafka topic it is configured to consume from. It could be that it is just a subset of these messages that it is unable to process or it could be all messages from the topic as shown in this example. Depending upon how the connector is configured, To troubleshoot this, you can click the dead letter queue tile in the connector overview window. This will navigate the UI to the associated Kafka topic where the dead letter queue messages are being written.



In the dead letter queue topic view, select the messages tab, drill into one of the messages and select the header tab. Then scroll down in the header information to identify the possible cause for the message ending up in the dead letter queue. In this example, we see that the connector wasn't configured to auto-create the destination table if it didn't already exist. To correct this, you would simply updated the connector configuration setting auto create table to true. You could do so using either the Confluent Cloud UI, the Connect API, or the Confluent CLI.

Troubleshoot a Failed Connector

CONFLUENT Developer



@TheDanicaFine | developer.confluent.io

Let's continue with the use case from the previous slide. The connector has been configured to auto-create the destination table in the MySql database and the Confluent Cloud UI now indicates the connector failed. You can investigate this using several tools. Let's now look at each of these.

Confluent CLI

CONFLUENT Developer

```
$confluent connect describe lcc-o3n6o9
Connector Details
+--------+--------------------------------+
| ID     | lcc-o3n6o9                     |
| Name   | MySqlSinkConnector_0           |
| Status | FAILED                         |
| Type   | sink                           |
| Trace  | Found nested structure in      |
|        | input data. Ensure that your   |
|        | input events are a flat struct |
|        | of primitive fields.           |
+--------+--------------------------------+


Task Level Details
  Task ID |         State
----------+------------------------
        0 | USER_ACTIONABLE_ERROR
```


The Confluent CLI is one of the tools that can be used to investigate connector failures. The describe command provides similar detail as the Confluent Cloud UI.

Confluent Connect API

CONFLUENT Developer

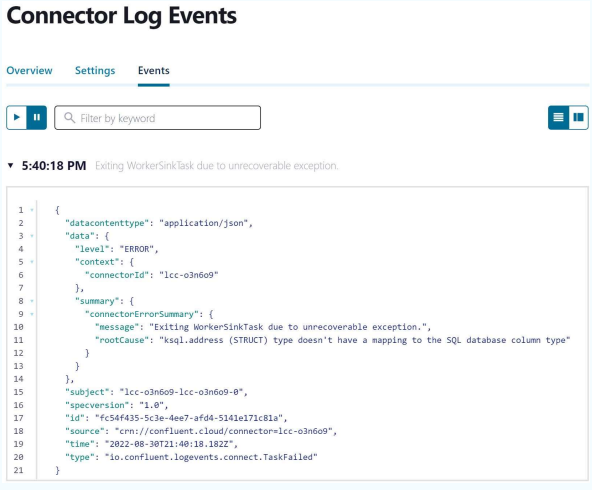
```
GET /connect/v1/environments/{environment_id}/clusters/{kafka_cluster_id}/connectors/{connector_name}/status
```


```
{
  "name": "MySqlSinkConnector_0",
  "connector": {
    "state": "FAILED",
    "worker_id": "MySqlSinkConnector_0",
    "trace": "Found nested structure in input data. Ensure that your input events are a flat struct of
primitive fields.\n"
  },
  "tasks": [
    {
      "id": 0,
      "state": "USER_ACTIONABLE_ERROR",
      "worker_id": "MySqlSinkConnector_0",
      "msg": ""
    }
  ],
  "type": "sink"
}
```


The Confluent Connect API status request provides similar detail as the Confluent Cloud UI and Confluent UI.

Confluent Cloud Connect Log Events

CONFLUENT Developer

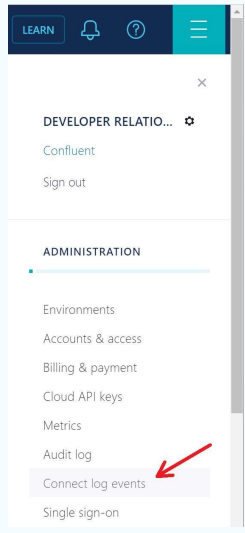


Confluent Cloud Connect log events are available on the connector events tab. They may provide additional detail regarding connector problems. In this example, the log event adds to the previous trace information about ensuring your input events are a flat struct of primitive fields.

Consuming Connect Log Events

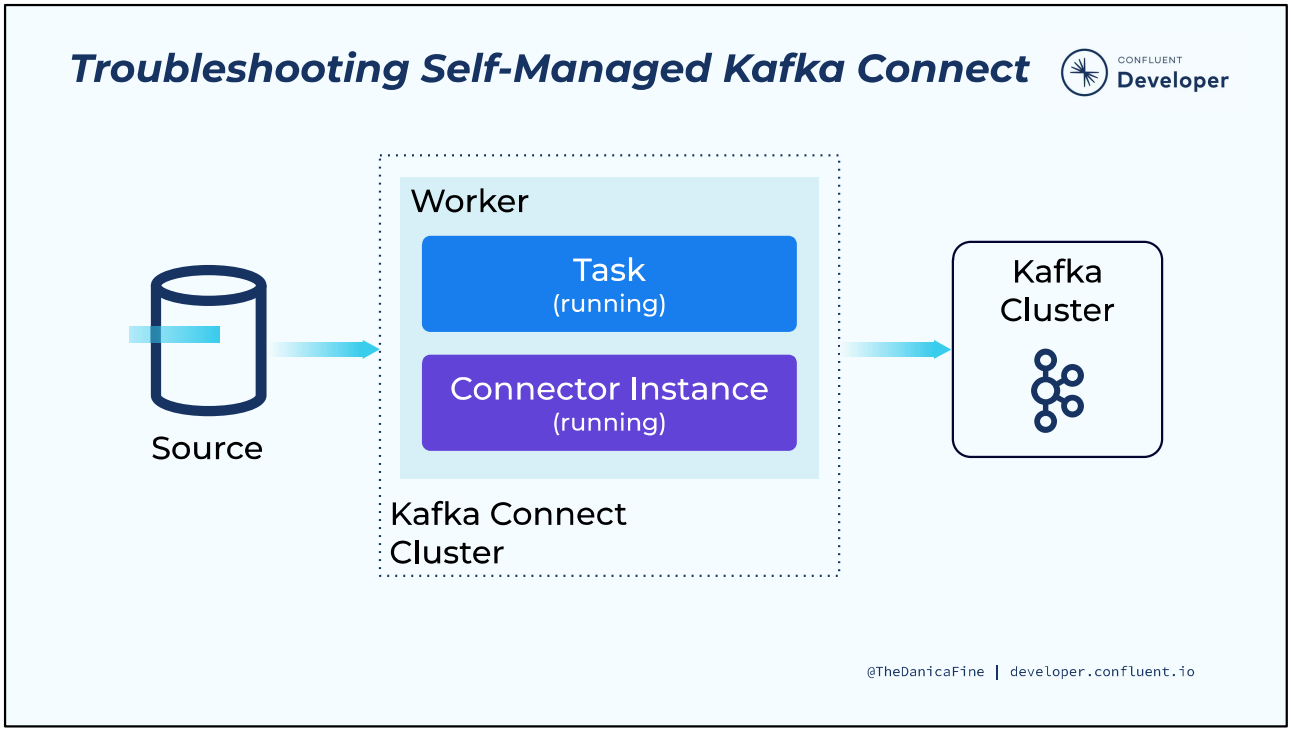
CONFLUENT Developer

```
$confluent kafka topic consume -b confluent-connect-log-events
Starting Kafka Consumer. Use Ctrl-C to exit.
{"datacontenttype":"application/json","data":{"level":"ERROR",
"context":{"connectorId":"lcc-o3n6o9"},"summary":{"connectorEr
rorSummary":{"message":"Exiting WorkerSinkTask due to
unrecoverable exception.","rootCause":"ksql.address (STRUCT)
type doesn\u0027t have a mapping to the SQL database column
type"}}},"subject":"lcc-o3n6o9-lcc-o3n6o9-0","specversion":"1.
0","id":"fc54f435-5c3e-4ee7-afd4-5141e171c81a","source":"crn:/
/confluent.cloud/connector=lcc-o3n6o9","time":"2022-08-30T21:4
0:18.182Z","type":"io.confluent.logevents.connect.TaskFailed"}
% Headers: [content-type="application/cloudevents+json;
charset=UTF-8"]
```

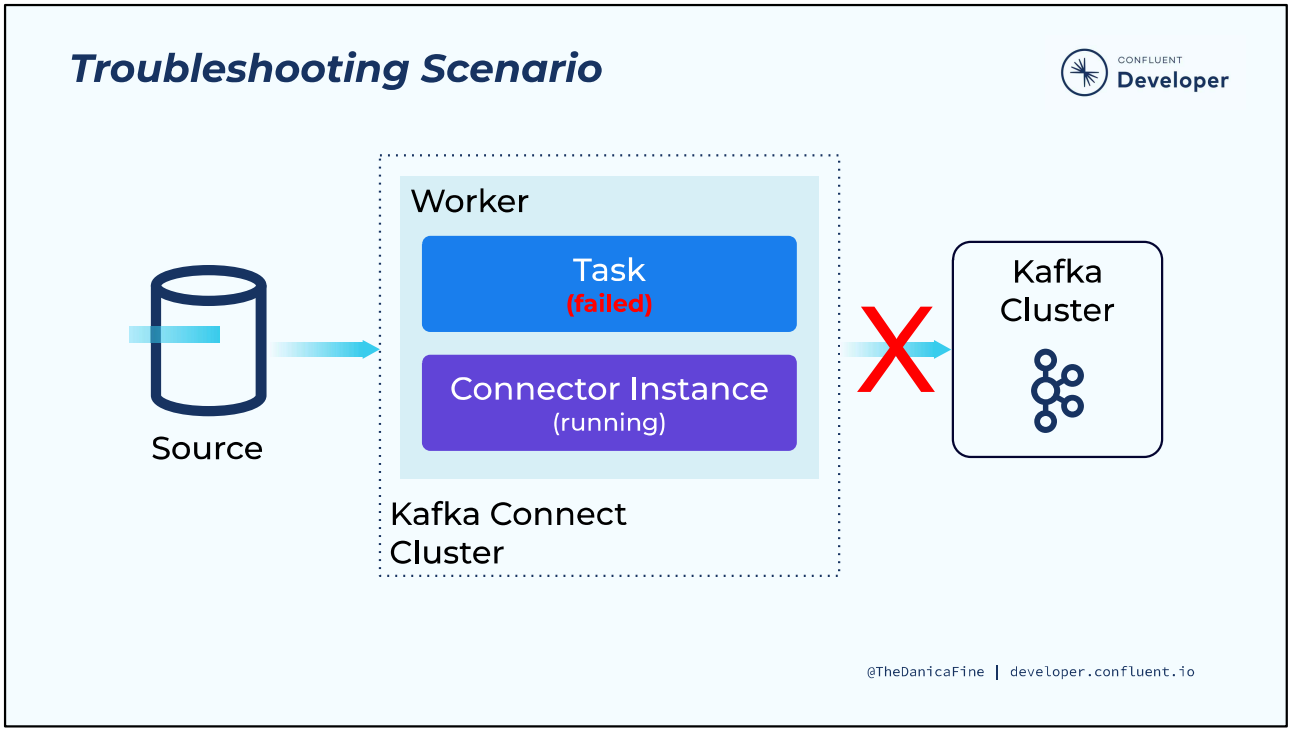


Connect log events can also be accessed using the Confluent CLI consume command. Detailed information regarding how to accomplish this can be found by clicking the triple bar icon in the upper right corner of the Confluent Cloud UI and choosing the Connect log events menu.

Module title slide style



Given that Kafka Connect is a data integration framework, troubleshooting is just a necessary part of using it. This has nothing to do with Connect being finicky (on the contrary, it's very stable). Rather, there are keys and secrets, hostnames, and table names to get right. Then there are the external systems that you are integrating with, each of which needs to be visible and accessible by Connect, and each of which has its own security model. Basically, if you've done any integration work in the past, the situation is familiar.



Your Connect worker is running, your source connector is running—but no data is being ingested.

Because connectors consist of tasks, one of the first things you should consider is that one or more of its tasks has failed, independently of the connector. To verify this, you'll need to gather more information from the Connect API.

Getting Connector and Task Status

CONFLUENT Developer

```
$ curl -s "http://localhost:8083/connectors/jdbc-sink/status" | \
jq '.connector.state'
"RUNNING"
```

```
$ curl -s "http://localhost:8083/connectors/jdbc-sink/status" | \
jq '.tasks[0].state'
"FAILED"
```


You might start by using curl to get the status of the connector instance and its tasks.

Here we see what was represented in the previous diagram. The connector instance status is RUNNING while the task status is FAILED.

But we need more details about this in order to identify what caused the failure.

Getting Task Status

CONFLUENT Developer

```
$curl -s "http://localhost:8083/connectors/jdbc-sink/status" | jq '.tasks[0].trace' | sed 's/\\n/\n/g; s/\\t/\t/g'

"org.apache.kafka.connect.errors.ConnectException: Exiting WorkerSinkTask due to unrecoverable exception.
   at org.apache.kafka.connect.runtime.WorkerSinkTask.deliverMessages(WorkerSinkTask.java:618)
   at org.apache.kafka.connect.runtime.WorkerSinkTask.poll(WorkerSinkTask.java:334)
   at org.apache.kafka.connect.runtime.WorkerSinkTask.iteration(WorkerSinkTask.java:235)
   at org.apache.kafka.connect.runtime.WorkerSinkTask.execute(WorkerSinkTask.java:204)
   at org.apache.kafka.connect.runtime.WorkerTask.doRun(WorkerTask.java:200)
   at org.apache.kafka.connect.runtime.WorkerTask.run(WorkerTask.java:255)
   at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:515)
   at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
   at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
   at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
   at java.base/java.lang.Thread.run(Thread.java:829)
Caused by: org.apache.kafka.connect.errors.ConnectException: java.sql.SQLException: No suitable driver found for
jdbc:mysql://localhost/demo
   at io.confluent.connect.jdbc.util.CachedConnectionProvider.getConnection(CachedConnectionProvider.java:59)
   at io.confluent.connect.jdbc.sink.JdbcDbWriter.write(JdbcDbWriter.java:64)
   at io.confluent.connect.jdbc.sink.JdbcSinkTask.put(JdbcSinkTask.java:84)
   at org.apache.kafka.connect.runtime.WorkerSinkTask.deliverMessages(WorkerSinkTask.java:584)
   ... 10 more
Caused by: java.sql.SQLException: No suitable driver found for jdbc:mysql://localhost/demo
   at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:702)
   at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:189)
   at io.confluent.connect.jdbc.dialect.GenericDatabaseDialect.getConnection(GenericDatabaseDialect.java:247)
   at io.confluent.connect.jdbc.util.CachedConnectionProvider.newConnection(CachedConnectionProvider.java:80)
   at io.confluent.connect.jdbc.util.CachedConnectionProvider.getConnection(CachedConnectionProvider.java:52)
   ... 13 more
"
```


You can also use curl to request the stack trace for the task and pipe the results through jq (a remarkably capable JSON formatter). The illustrated command requests the stack trace for the first element in the tasks array.

Next, read through the trace and look for clues. In this instance, upon reviewing, you notice that there is a Connect exception and also that a driver is missing.

Kafka Connect Log4j Logging

CONFLUENT Developer

- The log is the source of truth

```
$ confluent local services connect log
$ docker-compose logs kafka-connect
$ cat /var/log/kafka/connect.log
```

- The Log4j properties files controls what is logged, the log message layout, and where log files are stored

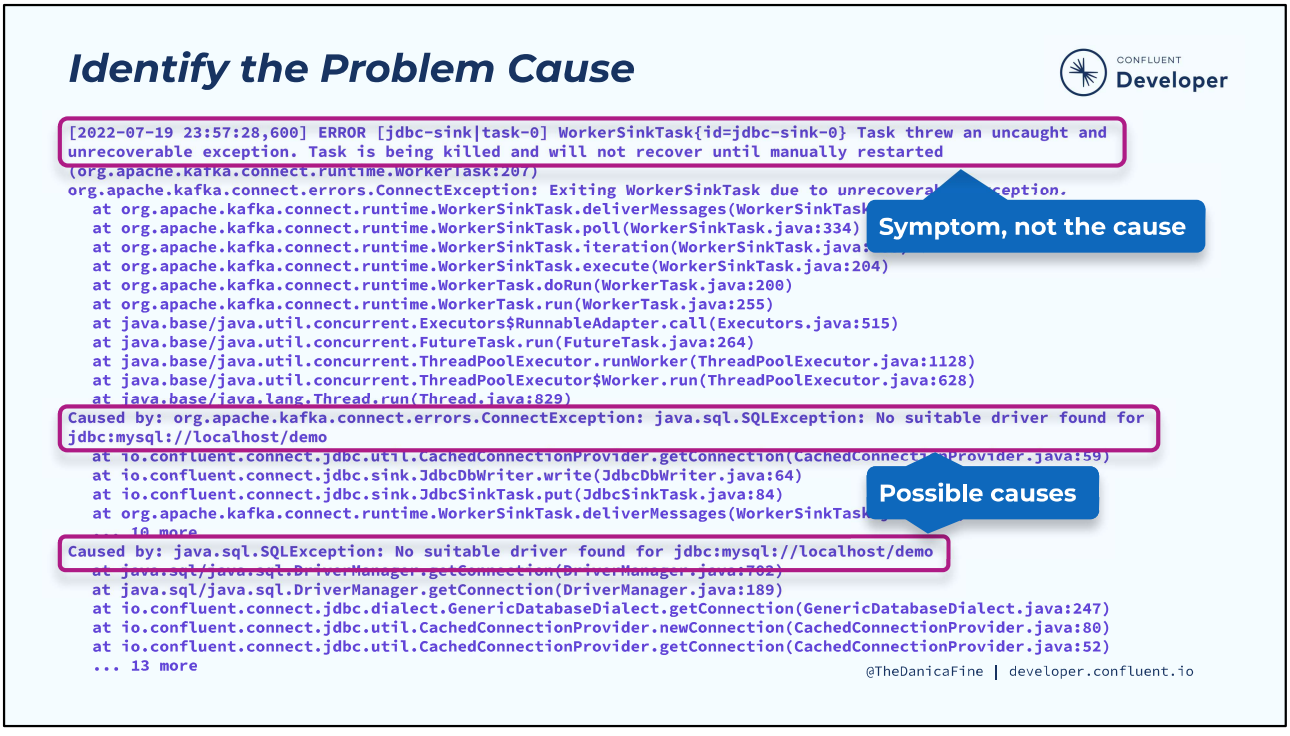
```
/etc/kafka/connect-log4j.properties (default location)
```

@TheDanicaFine | developer.confluent.io

In addition to the stack trace, you should read the log. There are different ways to access the log, depending on how you are running Connect:

- If you are just running the Confluent CLI locally, the command is confluent local services connect log
- If you are using Docker, it's docker logs, plus the name of the container
- If you are running completely vanilla Connect using Apache Kafka, you can just read the log files with cat, or more likely tail (the location varies by installation)

Connector contexts were added to logging in Apache Kafka 2.3 with KIP-449, and they make the diagnostic process a lot easier.



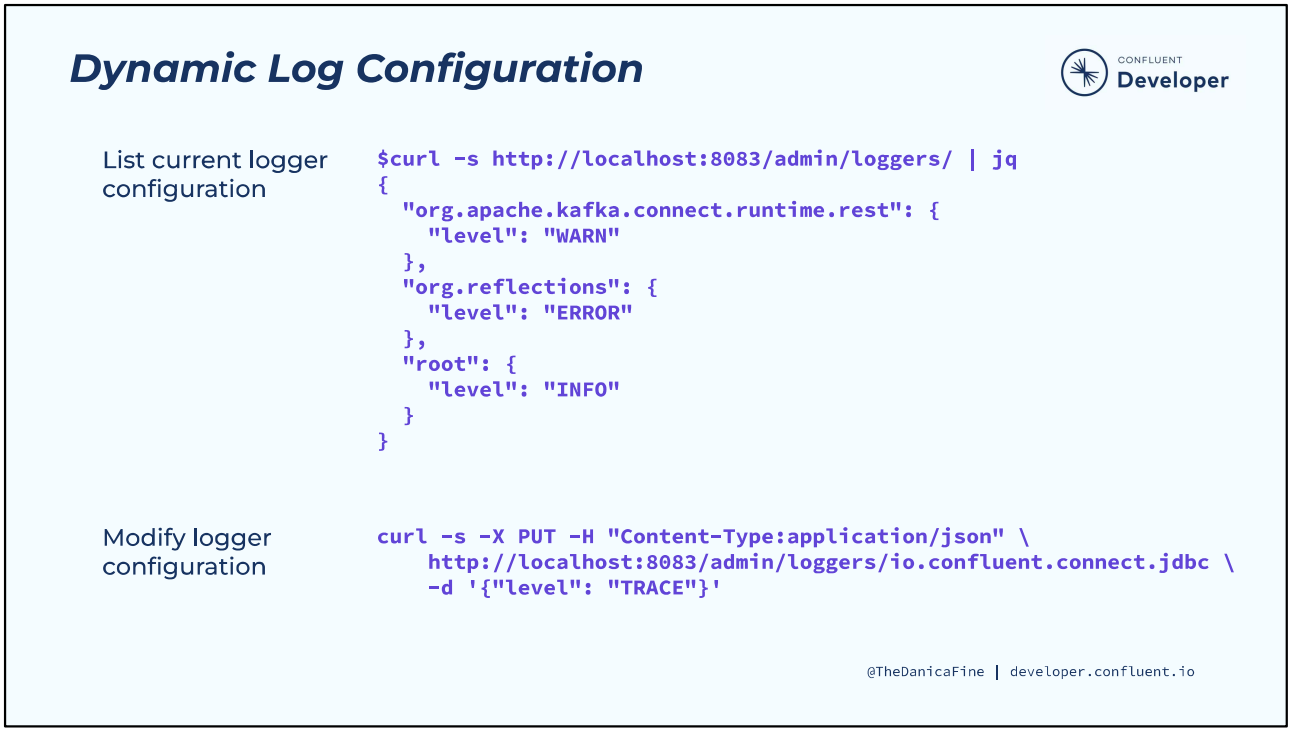
“Task is being killed and will not recover until manually restarted”

This is a general error, a symptom of the problem rather than the cause, and it doesn’t reveal any information about the underlying problem. When you see this, this is a sign that you need to search further in the stack trace or the connect worker log for your problem. For example, you can see this error in the connect worker log, and it’s a sign that you should look further in the log to the exceptions in order to identify your problem.

At this point, you are only at the beginning of troubleshooting the problem but at least you know where to look. With a bit of research, you would find the documentation for the JDBC connector indicates the MySQL JDBC driver needs to be installed on the connect worker machine when a MySQL database is part of the pipeline.

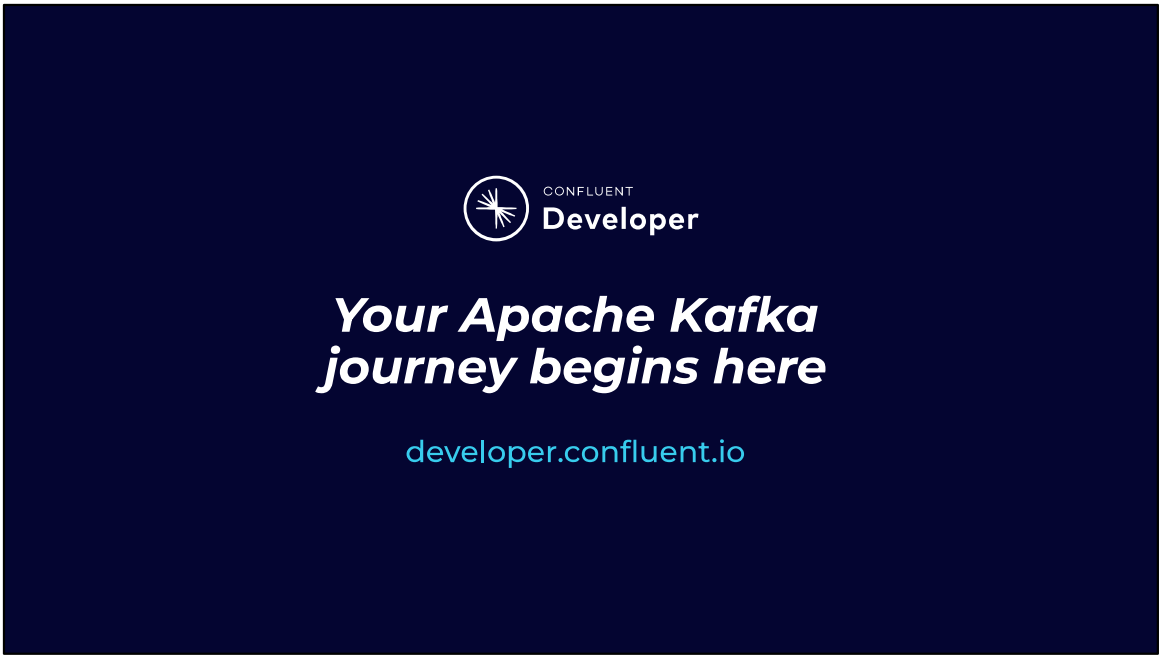
If your research doesn’t bear fruit, you might consider posting your problem to the Confluent Community Forum, but just keep in mind that a useful post will elaborate upon the “Task is being killed” error

alone.



Dynamic log configuration arrived in Apache Kafka 2.4. It means you can change the level of logging detail without having to restart the worker.

For example, perhaps there is a particular connector such as io.debezium above that you'd like to log at TRACE level to try and troubleshoot. If you set everything to TRACE, it would be overwhelming. Using dynamic log configuration, you can conveniently do so at runtime via REST without restarting Connect, and targetting the specific logger of interest.


CONFLUENT
Developer
Your Apache Kafka
journey begins here
developer.confluent.io