

Governance for Data-in-Motion

Paul Harvener, Principal Consultant, Data–Blitz

Overview:

At Data-Blitz, we've noticed a growing interest in governance and data quality control for data in motion. This paper serves as a comprehensive guide for software engineers looking to create a proof of concept (POC) focused on governance for data in motion. It includes all the necessary artifacts to build and execute such a POC successfully. POCs involving distributed systems often demand detailed knowledge of each component involved, which can be daunting. Frequently, these efforts are constrained by limited time allocations from management. This guide addresses these challenges by providing a step-by-step process for designing and executing a Data Governance POC for data in motion, leveraging the Confluent Platform Schema Registry and Data Contracts. All the required artifacts are provided in the appendices and to a linked Git repository containing all the prebuilt artifacts for easy access. Data in motion refers to data actively moving through a system's pipeline. Wikipedia defines data in motion as:

***“ Data in transit, also referred to as data in motion and data in flight, is data en route between source and destination, typically on a computer network.
Data in transit can be separated into two categories: information that flows over the public or untrusted network such as the Internet and data that flows in the confines of a private network such as a corporate or enterprise local area network.
Data in transit is used as a complement to the terms data in use, and data at rest which together define the three states of digital data. ”***

This paper examines data in motion within stream processing using Kafka. Confluent through the Confluent Platform Enterprise Addition provides tools designed to enforce data quality and transformation efficiently. Although the idea is not new, in the past, ensuring data quality in motion primarily focused on encryption, typically implemented via SSL/TLS. While encryption has become largely standardized, data transformation and semantics have often been handled by custom implementations on both the producer and consumer sides. Many organizations developed custom data producers to validate data against a defined standard, rejecting any data that failed to meet this benchmark.

This use case naturally led to the development of more reusable solutions. Confluent began addressing this challenge with the release of the Confluent Schema Registry in 2018. The Schema Registry validated the structure of data against predefined schemas and provided a framework for migrating schema changes. It allowed consumers to work with either the old or new schema, enabling organizations to roll out changes

without disrupting existing systems. Over time, older consumers could gradually transition to the new schema without breaking functionality during the interim, a game-changing approach that greatly improved the quality of data in motion.

More recently, Confluent introduced tools for controlling data semantics as well. This paper explores the use of Confluent Data Contracts between data Producers and Consumers, providing a comprehensive solution for managing both structure and semantics in data streams.

Confluent Data Contracts shift the responsibility for ensuring data quality to Producers by establishing a binding and enforceable agreement between data Producers and Consumers. This approach, known as "**shift-left**", empowers producers to uphold the integrity of the data's structure and meaning before it reaches consumers. With the contract in place, consumers can rely on the incoming data's accuracy and consistency, confident that it aligns with the agreed-upon standards. In a streaming architecture, data contracts provide critical transparency into data dependencies and usage, ensuring consistent, reliable event streams while serving as a definitive reference point for understanding the flow and structure of data in motion.

Kafka-based messaging systems are the logical choice for enforcing data governance in motion because they act as the central conduit through which all data passes in a distributed architecture. Kafka serves as the backbone for real-time data streams, connecting data producers and consumers in a way that naturally lends itself to centralized enforcement of governance policies. By establishing data contracts at this central point of flow, Kafka ensures that data governance rules, such as schema validation, data quality checks, and security policies, are consistently applied before data reaches its destination.

With Kafka, all data passes through a single pipeline, making it the ideal place to enforce governance policies across multiple systems and teams. As data is produced, Kafka can validate it against predefined schemas, check for compliance with data contracts, and flag any discrepancies immediately, allowing issues to be caught and corrected before they propagate downstream. This centralized enforcement relieves pressure on downstream consumers, who can trust that the data they receive is already clean, consistent, and compliant with governance rules. It eliminates the need for multiple systems to enforce their governance measures independently, reducing complexity and ensuring uniformity across the data ecosystem.

Kafka's ability to handle large volumes of data in real time also makes it perfect for flagging bad data as soon as it appears, providing actionable insights that producers can use to resolve issues at the source. This proactive approach improves data quality and reduces the time and cost associated with identifying and correcting bad data later in the pipeline. Additionally, Kafka's integration with monitoring and alerting tools enhances visibility into data flows, enabling more effective governance.

While data at rest remains important, governance in motion, enforced through Kafka, plays a critical role in ensuring that data is of high quality before it even reaches storage. The governance of data at rest complements this by ensuring that once the data is stored, it remains secure, compliant, and accurately reflects the original

stream. In this way, Kafka messaging systems bridge the gap between real-time governance and long-term data management, ensuring that data is governed throughout its entire lifecycle from the moment it's generated to when it's archived or analyzed. This integrated governance model helps maintain the integrity, reliability, and security of data, whether in motion or at rest, making Kafka a cornerstone for modern data governance strategies.

Confluent Platform's Data Governance Ecosystem

- **Schema Registry:**

The Confluent Schema Registry is a centralized service that manages schemas for Kafka topics. It ensures producers and consumers use consistent and compatible data formats, enabling seamless schema evolution and data governance. A schema plays a central role in a data contract by defining the structure, data types, and constraints for the key and value fields of the data being transmitted. Schemas ensure that both producers and consumers of data are aligned on the data's format and structure.

Supported Schema Formats

The Confluent Schema Registry supports different schema formats, each with its strengths:

1. **Avro:** A compact and fast binary format that is widely used for serialization in Kafka. Avro supports schema evolution, making it ideal for use cases where data structures change over time.
2. **JSON:** A human-readable format that is commonly used for data serialization. While it is not as space-efficient as Avro, JSON is easier to inspect and debug.
3. **Protobuf:** A binary format developed by Google, known for its efficiency and flexibility. Protobuf supports both forward and backward schema compatibility, making it useful in distributed systems.

ARVO is currently the most used streaming protocol. It is an optimized form of JSON but uses its own schema definition format. As stated before, this paper only describes streaming governance using ARVO.

Key Components of the Schema Registry

1. **Schema Storage:**

- Stores schemas for Kafka topics and versions them.

2. **Subject:**

- A Subject represents a schema namespace, typically associated with a Kafka topic.
- Subjects schemas are divided between value schemas and key schemas.
- Subjects can have multiple versions, representing schema evolution.

3. Migration:

- Define how schemas evolve and interact with existing versions.
- Types: Backward, Forward, and No Compatibility.

Backward Compatibility

When using the Kafka Schema Registry in **BACKWARD** compatibility mode, the goal is to ensure that the new schema can successfully read data produced by older versions of the schema. This mode focuses on making sure producers who upgrade to the new schema do not break existing consumers reading data from the older schema's format.

Allowed (Non-Breaking) Changes Under BACKWARD Compatibility:

1. Adding New Fields With Defaults:

You can safely add new fields to the schema as long as each new field includes a default value. This ensures that data encoded with the old schema (which lacks the new fields) can still be read by the new schema. When encountering older data, the new schema uses the default values for the missing fields.

2. Removing Fields That Had Defaults:

If a field previously had a default value, you can remove it without breaking backward compatibility. Older data that included this field is still readable because the new schema can interpret the data's absence of the field as falling back to a scenario consistent with older versions (though it effectively ignores that field's data now).

3. Adding Symbols to Enums:

You can add new symbols (values) to an enum. Older data that doesn't use the new symbols remains perfectly readable by the new schema, and no errors are introduced.

4. Reordering Fields or Changing Documentation:

Reordering fields in an Avro schema or changing documentation (``doc`` fields) does not break backward compatibility. Avro identifies fields by name, not position, and documentation does not affect data interpretation

Breaking Changes Under BACKWARD Compatibility

1. Removing Required Fields (Without Defaults):

If you remove a field that did not have a default value, the new schema will not know how to interpret older data containing that field. This breaks backward compatibility.

2. Changing Field Types Incompatibly:

Altering a field's type to something that older data cannot be interpreted as (e.g., changing from ``int`` to ``string`` without a union that supports both) breaks backward compatibility. The new schema won't be able to read older data correctly if its fields have incompatible types.

3. Renaming Fields Without Aliases:

If a field name changes, the new schema no longer recognizes the old field name found in historical data, making it impossible to read that old data without additional migration mechanisms (like aliases).

4. Removing Enum Symbols:

Dropping an enum symbol expected by older data results in unreadable data for that field, breaking backward compatibility.

5. Changing Defaults of Existing Fields:

Updating a default value that already existed can break backward compatibility if the new default value conflicts with expectations or assumptions of older data readers.

Summary:

Backward compatibility allows newer schemas to consume older data without issues, as long as changes are limited to adding fields with defaults, adding enum symbols, and minor schema adjustments that don't break the interpretation of existing fields. More disruptive changes, such as removing required fields or altering field types, are considered breaking under BACKWARD compatibility.

Forward Compatibility

Forward compatibility means that older consumers using the old schema should be able to read data produced by newer producers using the new schema. In other words, changes to the schema should be made in such a way that old readers can still interpret the new data without errors.

Allowed (Non-Breaking) Changes Under FORWARD Compatibility:

1. Adding New Fields (with Defaults):

Adding a new field that old consumers do not know about is safe as long as the field can be ignored by the old schema. Avro's name-based resolution means the old schema simply won't look for this new field, and thus won't break. Providing a default value is still a good practice, though it mainly affects backward compatibility.

- **Note:** While not strictly required for forward compatibility (since old readers ignore unknown fields), defaults ensure smoother backward and full compatibility.

2. Reordering Fields or Adding Documentation:

Changing the order of fields or adding documentation (`doc` fields) does not affect how old consumers read data. Avro uses field names to match data rather than relying on order. Documentation changes are informational only.

Potentially Allowed (With Caution):

1. Adding Enum Symbols:

If the new schema includes additional enum values, older schemas won't recognize these new values if they appear in the data. If older readers encounter these new enum values, they may fail, making this potentially a breaking change for forward compatibility. Generally, avoid adding new enum symbols if old consumers must understand them.

Breaking Changes Under FORWARD Compatibility:

1. Removing Fields Expected by Old Schema Without Defaults:

If the old schema expects a certain field (it has no default) and the new schema's data no longer provides it, the old consumers can't fulfill their expectation for that field. This breaks forward compatibility since old consumers can't read data they assume must exist.

2. Changing Field Types Incompatibly:

If the new schema changes a field's type to something the old schema cannot interpret, old consumers won't be able to read that field's data correctly. For example, changing an `int` to a `string` could cause reading issues for the old schema.

3. Removing Enum Symbols:

If older consumers rely on certain enum values and you remove them in the new schema, old consumers expecting those symbols can't decode the data properly.

4. Renaming or Removing Fields Without a Clear Migration Path:

Old consumers that look for a field by its original name won't find it if it's been removed or renamed without aliases. Since old consumers do not have knowledge of the rename, they cannot read the data as intended.

5. Altering Union Composition:

Adding or removing types from a union field in a way that old schemas cannot handle breaks forward compatibility. If old consumers expect certain union branches that no longer exist, they can't properly decode the data.

Summary:

To maintain forward compatibility, ensure that old consumers can still interpret the new data. Avoid removing fields that older consumers need, changing field types incompatibly, or removing enum symbols. Adding completely new fields is usually safe since old consumers ignore unknown fields. The key is that data produced with the new schema must still fulfill the expectations of the old schema so old consumers can read it without error.

Full Compatibility (Forward and Backward)

Full compatibility means the schema must maintain both backward and forward compatibility at the same time. In other words, new schemas must be able to read all previously produced data (backward compatibility), and old schemas must be able to read data produced by the new schema (forward compatibility).

Allowed (Non-Breaking) Changes Under Full Compatibility

1. Adding New Fields with Default Values :

- Backward Compatibility: The new schema can read older data that doesn't include the new field by using the default value.
- Forward Compatibility: Older schemas ignore unknown fields, so they remain unaffected.

2. Reordering Fields :

- Avro uses field names rather than positions, so reordering fields does not affect reading older or newer data.

3. Adding Documentation or Changing doc Fields :

- Documentation changes do not affect how data is interpreted.\

4. Making a Required Field Optional by Adding a Default :

- Older data (lacking this field) can still be interpreted by the new schema using the default value.
- Older schemas ignore the new optional field, remaining forward compatible.

5. Adding Enum Symbols (With Caution) :

- If older schemas encounter a new enum symbol, they may not know how to handle it. To maintain true full compatibility, ensure the new symbol is not actually encountered by older consumers or plan a coordinated upgrade.

Breaking Changes Under Full Compatibility

1. Removing Required Fields Without Defaults :

- Backward Break: The new schema won't know how to interpret old data that depended on the removed field.
- Forward Break: Older schemas expecting this field won't find it in the new data.

2. Changing Field Types Incompatibly :

- Altering a field's type (e.g., from `int` to `string`) breaks older and newer schemas' ability to interpret data consistently

3. Renaming Fields Without Aliases :

- If you rename a field without providing aliases or a strategy for old schemas to recognize it, old schemas break. Similarly, the new schema can't interpret old data correctly.

4. Changing Default Values of Existing Fields :

- Old data might depend on the original default values. Changing them disrupts backward compatibility. Old schemas interpreting new data may rely on the old defaults, breaking forward compatibility.

5. Removing Enum Symbols :

- If older schemas expect certain symbols and they're removed, they can't interpret the new data. If newer schemas interpret old data that includes now-removed symbols, they fail.

6. Altering Union Types Incompatibly :

- Removing or changing union branches can break how both old and new schemas understand the data.

Summary:

To maintain full compatibility, schema evolution must be done very carefully. You can add new fields (with defaults), reorder fields, add documentation, or make fields optional by adding defaults. However, removing fields, changing field types, renaming fields, or removing enum symbols will break compatibility both backward and forward. By adhering to these rules, you ensure that both older and newer consumers and producers can continue interacting seamlessly, regardless of which schema version they use.

Schema Registry Migration Summary:

While the Confluent Schema Registry supports communication and synchronization of schema evolution between producers and consumers, it sometimes lacks the flexibility to accurately represent real-world scenarios of schema changes. As noted, many modifications rely on having attributes defined with default values. In Avro, providing defaults is the primary mechanism for indicating optional fields, which can feel like a limitation since it's the only built-in approach. Having defaults can sometimes undermine the very purpose of maintaining a schema.

Later, we will introduce Confluent Platform's data contracts and migration rules to address these shortcomings. In the meantime, the following table summarizes the legitimate schema changes allowed under the three compatibility modes: backward, forward, and full.

Schema Migration Matrix

Schema Change	Backward Compatible?	Forward Compatible?	Fully Compatible?
Add a new field with a default value	Yes	Yes	Yes
Remove a field that had a default	Yes	Yes (if old readers ignore it)	Yes (generally safe)
Add an enum symbol	Yes	Potentially No (old readers may fail if they see it)	Potentially (use with caution)
Remove a required field (no default)	No	No	No
Change a field's type to an incompatible type	No	No	No
Rename a field without providing an alias	No	No	No
Change the default value of an existing field	Potentially No (if old data relies on it)	Potentially No (old readers expect the old default)	No
Remove an enum symbol	No	No	No
Reorder fields	Yes	Yes	Yes
Add or change documentation (`doc` fields)	Yes	Yes	Yes
Modify union types by adding/removing branches	Generally No	Generally No	Generally No
...			

Confluent Kafka Platform Schema migration options

Key and Value Schemas

In Kafka topics, both the key and value of a message can have their distinct schemas. Within the Confluent Schema Registry, these schemas are represented separately but are stored under the same Subject. By default:

- The **key schema** is named subject-key.
- The **value schema** is named subject-value.

Schemas, Subjects, Topics, and Aliases

Before diving deeper, let's clarify some key terms and their relationships within the Schema Registry ecosystem:

- **Kafka Topics:**

A Kafka topic is a stream of messages, where each message consists of a key-value pair. Either the key, the value, or both can be serialized using Avro, JSON, or Protobuf formats.

- **Schemas:**

A schema defines the structure and data types of your message data. It determines which fields are included, their types, and how they're represented.

- **Subjects:**

Schema Registry introduces the concept of a subject, serving as a namespace within which schemas evolve. Subjects let you manage and version schemas independently of the Kafka topic's name.

By default, the subject name strategy derives the subject name from the topic name. However, the association between a Kafka topic and a schema (subject) is not strictly one-to-one, allowing greater flexibility and reuse of schemas across topics.

- **Subject Aliases**

Starting with Confluent Platform 7.4.1 and in Confluent Cloud, you can configure one subject to act as an alias for another. For example, suppose you've registered a schema under the subject

``my.package.Foo``, but you now want to apply that same schema to messages produced to a Kafka topic named ``mytopic``.

With the default ``TopicNameStrategy``, most clients expect the schema for ``mytopic``'s record values to be available under the subject ``mytopic-value``. Instead of re-registering the schema under a different subject or changing the subject naming strategy, you can simply set up an alias. This allows the schema defined in ``my.package.Foo`` to be used as if it were registered under ``mytopic-value``, streamlining schema management and reducing overhead.

Schema Versioning and Subject Association

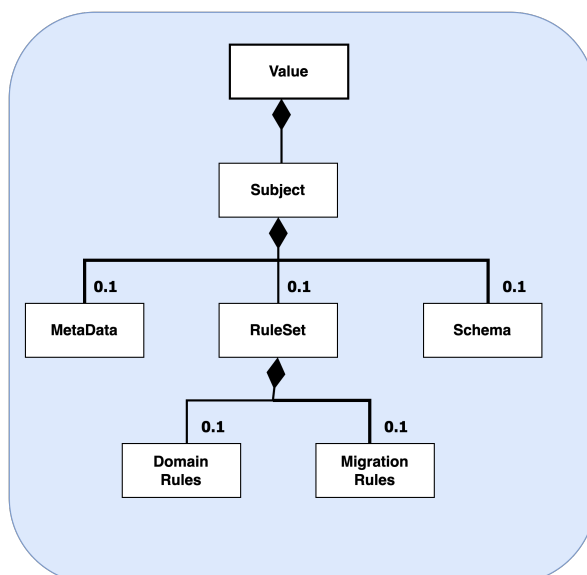
Schemas within a subject are versioned to allow for changes over time without breaking existing integrations. When a schema is updated, the Schema Registry assigns it a new version number under the same subject. This versioning system allows for schema evolution, where new fields can be added or existing fields can be modified while ensuring backward and forward compatibility.

Each subject is typically associated with a Kafka topic, ensuring that every message sent to that topic conforms to the schema defined in the subject. The versioning mechanism allows consumers to interpret data according to the correct version of the schema, maintaining data consistency even as schemas evolve.

By linking subjects to topics, the Schema Registry ensures that data contracts are enforced across Kafka topics, guaranteeing that data producers and consumers adhere to the agreed-upon schema at all times.

Composition of a Subject

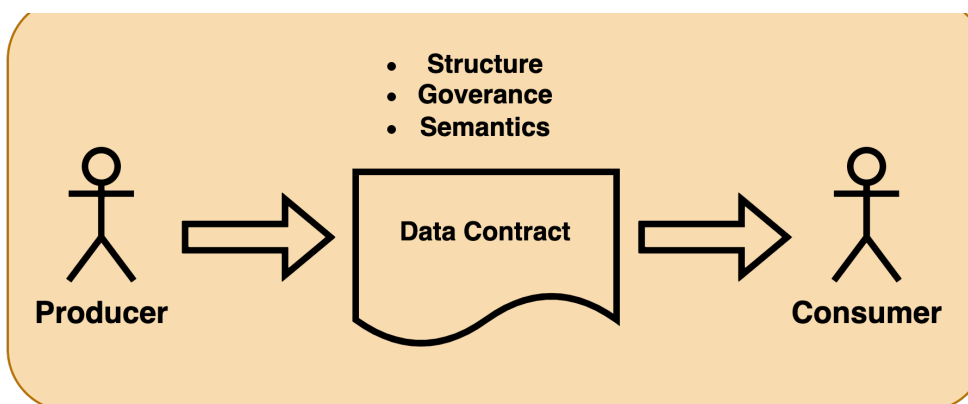
Below is diagram illustrating the composition of a Subject.



As shown in the diagram, a subject can optionally include metadata, a ruleSet, and a schema. The ruleSet itself can be composed of both domain rules and migration rules.

Confluent Data Contracts:

Confluent Data Contracts enhance the capabilities of the Confluent Schema Registry by allowing more than just schemas to be included. A Data Contract may consist of a schema alone, or it can also include optional Metadata and Data Quality Rules. Together, these components are versioned and stored as a Subject within the Schema Registry. The diagram below illustrates this structure:



The diagram above illustrates the structure of a Subject. The Subject is represented as a JSON document and submitted to the Confluent Schema Registry, where it is stored and assigned a version and unique ID. The Schema Registry provides a RESTful API that enables the provisioning of Schemas, Metadata, and Rulesets for a specific Subject.

Rules within the Rulesets can reference both Metadata and Schema attributes. Rules can be designed to validate or enforce conditions based on specific attributes defined in the schema, as well as information provided in the Metadata. This allows for more comprehensive validation logic, ensuring that the data adheres to both structural and contextual semantic requirements before being accepted or processed.

Metadata:

Metadata plays a critical role in enriching the data beyond its structural definition (as provided by the schema). Metadata can serve both informational and operational purposes, providing context and enabling more sophisticated governance and validation processes through rule sets.

At a basic level, metadata can be purely informational, providing descriptive details about the data, such as:

- **Data lineage:** Describing where the data originated, the transformations it has undergone, or who the data producer is.
- **Ownership and stewardship:** Indicating who is responsible for the data, both technically and from a business perspective.
- **Usage guidelines:** Offering information on how the data should be used, including any relevant compliance, privacy, or policy constraints.
- **Classification:** Labeling the data for categorization purposes, such as confidential, personal data, financial data, etc.

Metadata for Rule Sets:

Metadata is not just for documentation purposes; it can also serve as a functional part of the data governance system. In this capacity, it acts as a driver for rules within rule sets that operate at runtime. Rules can reference metadata to influence how the data is validated, transformed, or processed. Examples include:

- **Contextual validation:**

Metadata might include thresholds or business rules, such as a maximum allowable value for a field. Rules can reference this metadata to perform real-time validation.

- **Data retention policies:**

Metadata can specify how long data should be stored or under what conditions it must be archived or deleted, allowing rules to automatically manage these operations.

- **Tagging for rule enforcement:** Data might be tagged with specific attributes, such as "requires encryption" or "subject to GDPR." Rules can then reference these tags to enforce compliance, such as ensuring that sensitive data is encrypted before being transmitted or stored.
- **Conditional logic:** Based on metadata, rules can apply different validation or transformation logic. For example, if metadata indicates that data comes from a particular region, region-specific validation rules may be applied.

Dynamic Rule Referencing

The combination of schema attributes and metadata allows rules to dynamically adjust based on the data context. For instance:

- A rule might validate a field's value against a range specified in metadata (e.g., a credit score field must be within a range set by a regulatory requirement stored in the metadata).
- Rules could enforce that certain fields are required, optional, or subject to specific conditions, depending on metadata tags like "high priority" or "sensitive data."

In this way, metadata adds a layer of flexibility and context that allows for more dynamic and adaptive rule enforcement, enabling systems to respond to a wide range of business, operational, and compliance needs. It helps ensure that the data is not only structurally correct (validated by the schema) but also contextually correct, meeting business requirements and policy standards defined in the metadata

Data Contract RuleSets:

Data quality rules are critical in ensuring that data meets predefined standards before it is processed by downstream systems. These rules enforce consistency, accuracy, and compliance with business requirements, enhancing the overall reliability of data within a streaming architecture. Confluent's stream governance package, when enabled within the Schema Registry configuration, allows the enforcement of these data quality rules alongside schemas and metadata.

Types of Data Contract RuleSets

Data contract rule sets are available in two forms. Both types can leverage expression logic using the Google Common Expression Language (CEL) or JSONata. Additionally, Data Contracts support custom Rule Executors implemented in Java, allowing for more specialized validation and transformation.

- **domainRules:**

Domain Rules focus on validating the current data against domain logic and constraints to ensure data quality and correctness

- **migrationRules:**

Migration Rules focus on evolving schemas over time, enabling smooth transitions between schema versions without breaking existing producers or consumers.

In a single ruleSet, you might combine both domain and migration rules: domain rules to maintain data quality and correctness now, and migration rules to ensure compatibility as your schema and business requirements evolve.

POC Examples:

All the upcoming examples will utilize an Avro schema and demonstrate scenarios such as Subjects with Complex Migration Rules, Domain Validation Rules leveraging Dead Letter Queues (DLQ),. These examples will run on the Confluent Platform Enterprise Edition, set up using Docker Compose. The host machine is expected to have access to Docker Hub and be capable of establishing a local network within the Docker Compose environment. We will use `console-avro-producer` and console-avro-consumer for message production and consumption, respectively.

This document provides all the necessary content to successfully execute each example without requiring additional references to documents covering the detailed usage of each runtime layer. While understanding the implementation layers is important, the focus here is to ensure everything needed to run the examples is contained in one place.

Additionally, all the referenced artifacts, including the example configurations and schemas, are contained within the appendices of this document or can be cloned from the GitHub repository at

<https://github.com/data-blitz-demos/data-in-motion>

Example 1 Data Structural Governance:

In our first example, we demonstrate sending AVRO messages to a specified topic, with the Schema Registry enforcing the schema structure. Any messages that do not conform will cause the producer to fail, illustrating a basic form of data governance. We will use this as a baseline for other examples, each introducing additional forms of data governance for data in motion. Please follow the steps below.

1. We will be deploying our Confluent Platform Kafka ecosystem using Docker Compose, so you'll need to install Docker on your machine. You can download Docker from the following link:

<https://www.docker.com/products/docker-desktop>

2. We will use *curl* to interact with the Confluent REST interfaces for the REST Proxy, Broker, and Schema Registry. Please download and install curl from the following link:

<https://curl.se/download.html>

Note: Mac users can also use Brew

Choose the correct installation for your machine's operating system.

3. Download and install the jq library, a command-line tool that allows you to parse, filter, and transform JSON data efficiently. You can get it from the following link:

<https://jqlang.github.io/jq/download/>

Note: Mac users can also use Brew

Choose the correct installation for your machine's operating system.

4. Optionally, download and install Visual Studio Code VSC from the following link.

<https://code.visualstudio.com/download>

Choose the correct installation for your machine's operating system. **Note:** This is optional; you can run the following examples directly from an operating system terminal.

5. Download and install Java. Follow the directions for your specific operating system

<https://www.java.com/en/>

6. Download and install the Confluent Platform on your machine. Unzip it into a directory of your choice, which we will refer to as `EXAMPLE_HOME`.

https://docs.confluent.io/platform/current/installation/installing_cp/zip-tar.html

Note: If we were using the downloaded distribution as our Kafka runtime, it would be referred to as `CONFLUENT_HOME`. However, in these examples, we will use Docker to run the Confluent Platform Kafka ecosystem. Despite this, we still need to run the **kafka-avro-console-producer** and **kafka-avro-console-consumer** located in the `bin` directory of the installation. This base directory will be referred to as `EXAMPLE_HOME`.

7. If you choose **not** to download from the above Git Repo then, copy and paste the text from Appendix 1 into a file named `docker-compose.yml` within the `EXAMPLE_HOME` directory.
8. Then, at the command prompt type,

docker-compose up -d

or use the docker-compose up plugin with VSC

The Confluent Platform ecosystem will begin starting once all the Docker images have been downloaded. This process may take some time, depending on the speed of your internet connection. Once complete, you should see the following response.

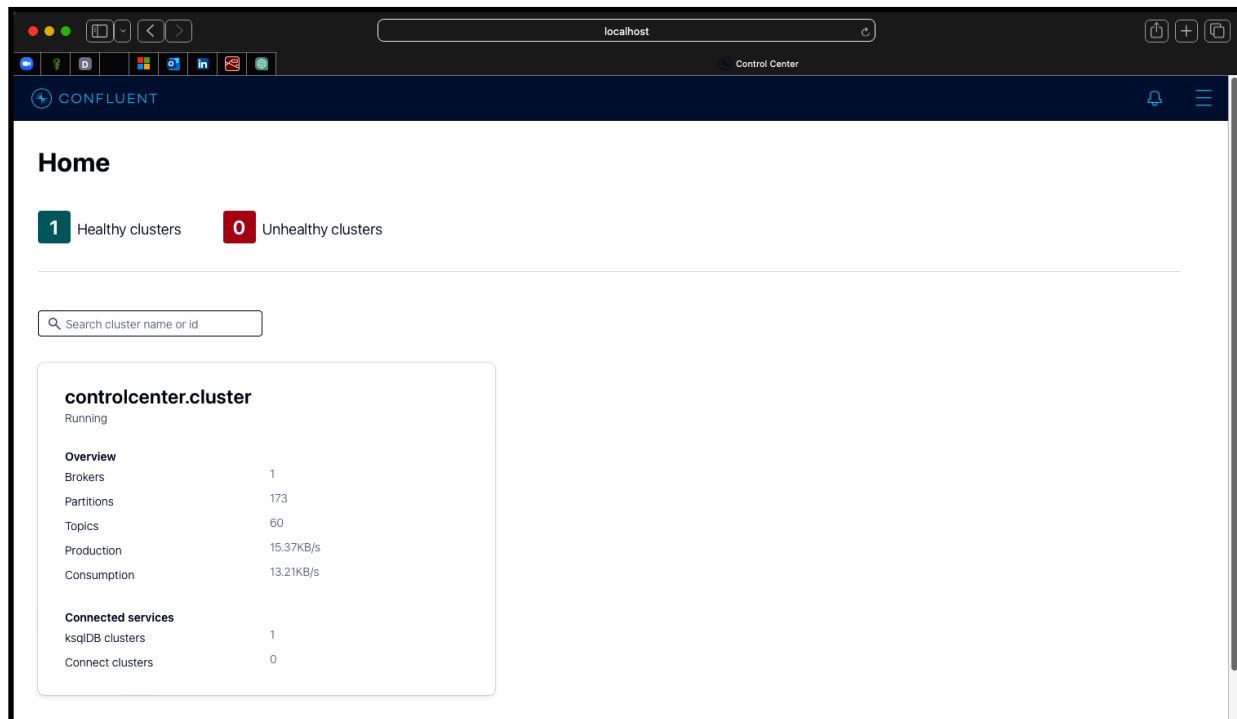
*** Executing task: docker compose -f "docker-compose.yml" up -d --build**

```
[+] Running 9/10
  ✔ Network confluent-770_default Created
    4.0s
  ✔ Container zookeeper Started
    1.3s
  ✔ Container broker Started
    1.6s
  ✔ Container schema-registry Started
    1.8s
  ✔ Container rest-proxy Started
    2.6s
  ✔ Container connect Started
    2.5s
  ✔ Container ksqldb-server Started
    2.8s
  ✔ Container ksqldb-cli Started
    3.5s
  ✔ Container control-center Started
    3.5s
  ✔ Container ksql-datagen Started
    3.5s
```

Notice we have started the whole Confluent Platform ecosystem. We will only be using the Zookeeper, Broker, Schema_Registry, and Control Center in these examples. The primary reason for taking this approach is to have access to the Confluent Control Center, a useful UI for viewing and provisioning the Kafka deployments.

9. If you choose **not** to download from the above Git Repo, then copy and paste the text from Appendix 2 into a file named order-transaction. avsc and then save it into the EXAMPLE_HOME directory. This is the AVRO schema we will be using for all of the examples.
10. Next, Let's bring up the Confluent Control Center. In your favorite web browser, go to the following URL.

URL: ***http://localhost:9021/clusters***



screenshot: 1

The screenshot above shows the Confluent Control Center UI homepage. Sometimes, the Control Center might initially display a cluster as "Unhealthy." To resolve this, click on "Overview" followed by "Brokers" in the left-hand menu. This will prompt the system to recheck the cluster's health, typically updating the status to "Healthy."

11. Register the AVRO schema **order-transaction.avsc** with the Confluent Schema Registry. This schema will be used for all the examples throughout this document.

```
jq -n --rawfile schema order-transaction.avsc '{schema: $schema}' |  
curl http://localhost:8081/subjects/order-transactions-value/versions --json @-
```

We should get the following response:

```
{"id":1}%
```

This response confirms that we successfully added the order-transaction schema to the Schema Registry, with the schema **id** set to 1. The avro-console-producer will use this **id** to identify the schema for the produced messages, with the schema **id** prepended to each message. The consumer will use this **id** to retrieve the correct schema from the Schema Registry for validation on the consumer side.

Notice the URL path: **/subjects/order-transactions-value/**. Since we're using the default Subject naming strategy, TopicNameStrategy, this indicates that we've created a Subject named order-transactions, and the schema will apply to the value schema of the key-value message. More about subjects will be covered when we start adding metadata and rule sets to the pipeline. If we also used a schema for the message key, it would be specified as **order-transactions-key**.

12. Start the avro-console-consumer. The kafka-avro-console-consumer is located in the **bin** directory of the Confluent Platform Distribution, which is located in the EXAMPLE_HOME directory. Start a new terminal and at the command prompt:

```
./bin/kafka-avro-console-consumer \  
--bootstrap-server localhost:9092 \  
--from-beginning \  
--topic order-transactions \  
--property schema.registry.url=http://localhost:8081
```

This command starts the **kafka-avro-console-consumer**, which will listen to and consume messages from the **order-transactions** topic. It will always read messages from the beginning of the topic upon startup. Additionally, it will use the Schema Registry running on `localhost:8081` when a schema **id** is prepended to the message.

We should get the following response:

[2024-10-22 21:19:45,890] INFO KafkaAvroDeserializerConfig values:

```
auto.register.schemas = true
avro.reflection.allow.null = false
avro.use.logical.type.converters = false
basic.auth.credentials.source = URL
basic.auth.user.info = [hidden]
bearer.auth.cache.expiry.buffer.seconds = 300
bearer.auth.client.id = null
bearer.auth.client.secret = null
bearer.auth.credentials.source = STATIC_TOKEN
bearer.auth.custom.provider.class = null
bearer.auth.identity.pool.id = null
bearer.auth.issuer.endpoint.url = null
bearer.auth.logical.cluster = null
bearer.auth.scope = null
bearer.auth.scope.claim.name = scope
bearer.auth.sub.claim.name = sub
bearer.auth.token = [hidden]
context.name.strategy = class io.confluent.kafka.serializers.context.NullContextNameStrategy
http.connect.timeout.ms = 60000
http.read.timeout.ms = 60000
id.compatibility.strict = true
key.subject.name.strategy = class io.confluent.kafka.serializers.subject.TopicNameStrategy
latest.cache.size = 1000
latest.cache.ttl.sec = -1
latest.compatibility.strict = true
max.schemas.per.subject = 1000
normalize.schemas = false
proxy.host =
proxy.port = -1
rule.actions = []
rule.executors = []
rule.service.loader.enable = true
schema.format = null
schema.reflection = false
schema.registry.basic.auth.user.info = [hidden]
schema.registry.ssl.cipher.suites = null
schema.registry.ssl.enabled.protocols = [TLSv1.2, TLSv1.3]
schema.registry.ssl.endpoint.identification.algorithm = https
schema.registry.ssl.engine.factory.class = null
schema.registry.ssl.key.password = null
schema.registry.ssl.keymanager.algorithm = SunX509
schema.registry.ssl.keystore.certificate.chain = null
schema.registry.ssl.keystore.key = null
schema.registry.ssl.keystore.location = null
schema.registry.ssl.keystore.password = null
schema.registry.ssl.keystore.type = JKS
schema.registry.ssl.protocol = TLSv1.3
schema.registry.ssl.provider = null
schema.registry.ssl.secure.random.implementation = null
schema.registry.ssl.trustmanager.algorithm = PKIX
schema.registry.ssl.truststore.certificates = null
schema.registry.ssl.truststore.location = null
schema.registry.ssl.truststore.password = null
schema.registry.ssl.truststore.type = JKS
schema.registry.url = [http://localhost:8081]
specific.avro.key.type = null
specific.avro.reader = false
specific.avro.value.type = null
```

```
use.latest.version = false
use.latest.with.metadata = null
use.schema.id = -1
value.subject.name.strategy = class io.confluent.kafka.serializers.subject.TopicNameStrategy
(io.confluent.kafka.serializers.KafkaAvroDeserializerConfig:372)
```

13. Start the **kafka-avro-console-producer**, which can be found in the ``bin`` directory of the Confluent Platform distribution within the ``EXAMPLE_HOME`` directory. Open a new terminal window and, at the command prompt, enter the following

```
./bin/kafka-avro-console-producer \
--topic order-transactions \
--broker-list localhost:9092 \
--property schema.registry.url=http://localhost:8081 \
--property value.serializer=io.confluent.kafka.serializers.KafkaAvroSerializer \
--property value.schema.id=1
```

This command starts the **kafka-avro-console-producer**, which will write/produce messages to the order-transactions topic. Once started, it will wait for messages to be input in the terminal. The producer will use the Schema Registry running on localhost:8081. Since this is a "**shift left**" pipeline processing model, the produced messages must conform to the ``order-transaction`` schema or schema **id** 1 at the source level. This approach guards against bad data being propagated to downstream consumers.

We should get the following response:

```
[2024-10-22 21:35:57,044] INFO KafkaAvroSerializerConfig values:
  auto.register.schemas = true
  avro.reflection.allow.null = false
  avro.remove.java.properties = false
  avro.use.logical.type.converters = false
  basic.auth.credentials.source = URL
  basic.auth.user.info = [hidden]
  bearer.auth.cache.expiry.buffer.seconds = 300
  bearer.auth.client.id = null
  bearer.auth.client.secret = null
  bearer.auth.credentials.source = STATIC_TOKEN
  bearer.auth.custom.provider.class = null
  bearer.auth.identity.pool.id = null
  bearer.auth.issuer.endpoint.url = null
  bearer.auth.logical.cluster = null
  bearer.auth.scope = null
  bearer.auth.scope.claim.name = scope
  bearer.auth.sub.claim.name = sub
  bearer.auth.token = [hidden]
  context.name.strategy = class io.confluent.kafka.serializers.context.NullContextNameStrategy
  http.connect.timeout.ms = 60000
  http.read.timeout.ms = 60000
  id.compatibility.strict = true
```

```
key.subject.name.strategy = class io.confluent.kafka.serializers.subject.TopicNameStrategy
latest.cache.size = 1000
latest.cache.ttl.sec = -1
latest.compatibility.strict = true
max.schemas.per.subject = 1000
normalize.schemas = false
proxy.host =
proxy.port = -1
rule.actions = []
rule.executors = []
rule.service.loader.enable = true
schema.format = null
schema.reflection = false
schema.registry.basic.auth.user.info = [hidden]
schema.registry.ssl.cipher.suites = null
schema.registry.ssl.enabled.protocols = [TLSv1.2, TLSv1.3]
schema.registry.ssl.endpoint.identification.algorithm = https
schema.registry.ssl.engine.factory.class = null
schema.registry.ssl.key.password = null
schema.registry.ssl.keymanager.algorithm = SunX509
schema.registry.ssl.keystore.certificate.chain = null
schema.registry.ssl.keystore.key = null
schema.registry.ssl.keystore.location = null
schema.registry.ssl.keystore.password = null
schema.registry.ssl.keystore.type = JKS
schema.registry.ssl.protocol = TLSv1.3
schema.registry.ssl.provider = null
schema.registry.ssl.secure.random.implementation = null
schema.registry.ssl.trustmanager.algorithm = PKIX
schema.registry.ssl.truststore.certificates = null
schema.registry.ssl.truststore.location = null
schema.registry.ssl.truststore.password = null
schema.registry.ssl.truststore.type = JKS
schema.registry.url = [http://localhost:8081]
use.latest.version = false
use.latest.with.metadata = null
use.schema.id = -1
value.subject.name.strategy = class io.confluent.kafka.serializers.subject.TopicNameStrategy
(io.confluent.kafka.serializers.KafkaAvroSerializerConfig:372)
```

14. Let's produce some messages. In the terminal running in the EXAMPL_HOME directory, running the **kafka-avro-console-producer**. Copy and paste the following JSON object after the previous response type/paste:.

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34,
"productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505,
"firstName": "Amanda", "lastName": "Murray", "email": "amanda.murray@yahoo.com", "gender": "Female",
"age": 67, "address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744",
"creditCardNumberType": "AMEX", "creditCardNumber": "5541123132728247"}
```

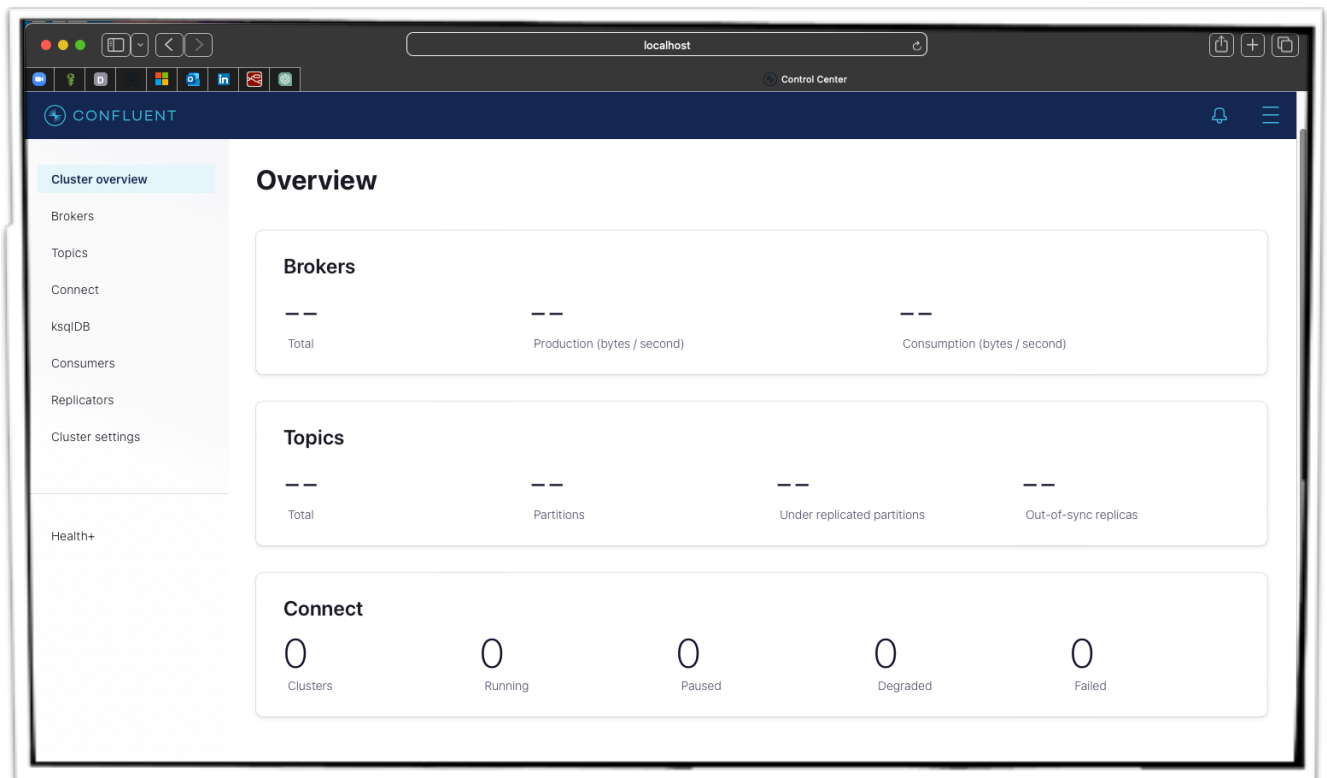
12. Let's check the **kafka-avro-console-consumer**. In the terminal running the **kafka-avro-console-consumer** we should see the following:

```
(io.confluent.kafka.serializers.KafkaAvroDeserializerConfig:372)
{"transactionId":"1f04e109-73a8-495c-a7b9-674c7779a130","productId":"4304360364601","price":874.34,"productDescription":"Stainless steel garden trowel with ergonomic handle","timestamp":1729184505,"firstName":"Amanda","lastName":"Murray","email":"amanda.murray@yahoo.com","gender":"Female","age":67,"address":"9900 Curtis Field Suite 242","city":"West Katieland","state":"VA","zipCode":"41744","creditCardNumberType":"AMEX","creditCardNumber":"5541123132728247"}
```

What just happened? We built a **"shift left"** pipeline using the Confluent Platform. First, we created an AVRO message schema and registered it with the Schema Registry. Then, we produced schema-compliant messages, which were written to the **order-transactions** topic. The consumer read the messages and validated them against the schema of Id 1, and since they passed the schema check, they were successfully written to stdout.

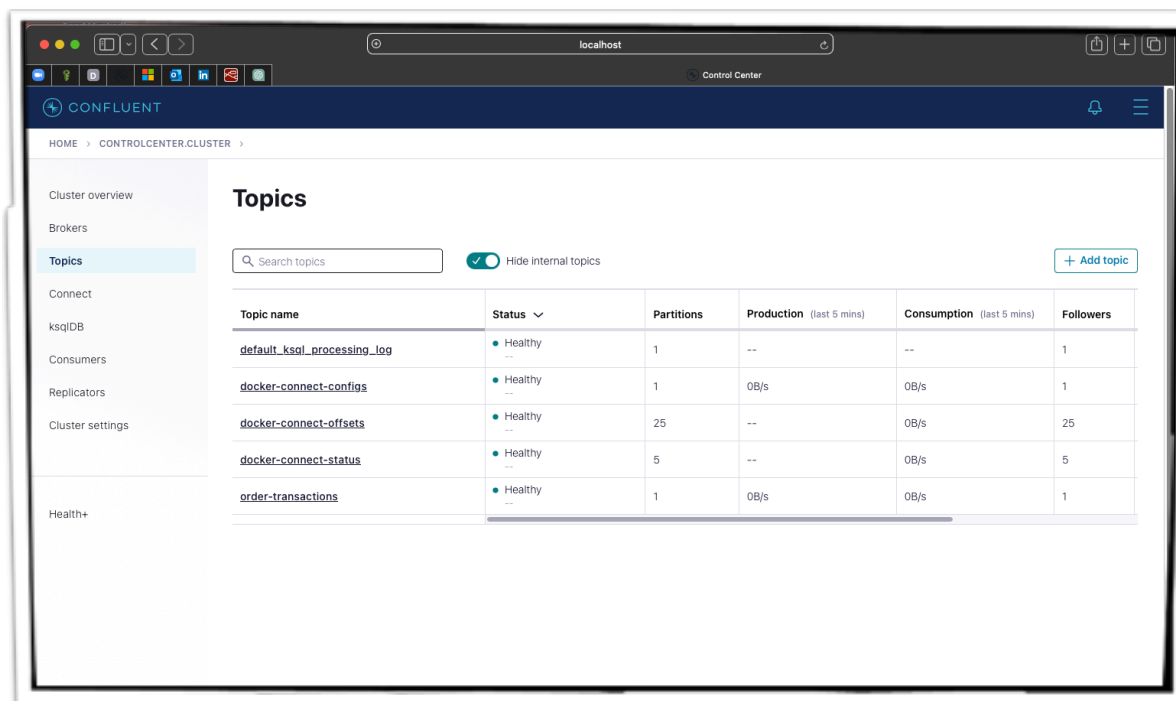
13. Let's check in with the Confluent Control Center. Start your favorite browser and go to link,

<http://localhost:9021/clusters/overview>



Next, click on the Topics link on the left. This will take you to the Topics page.

Cluster overview->Topics

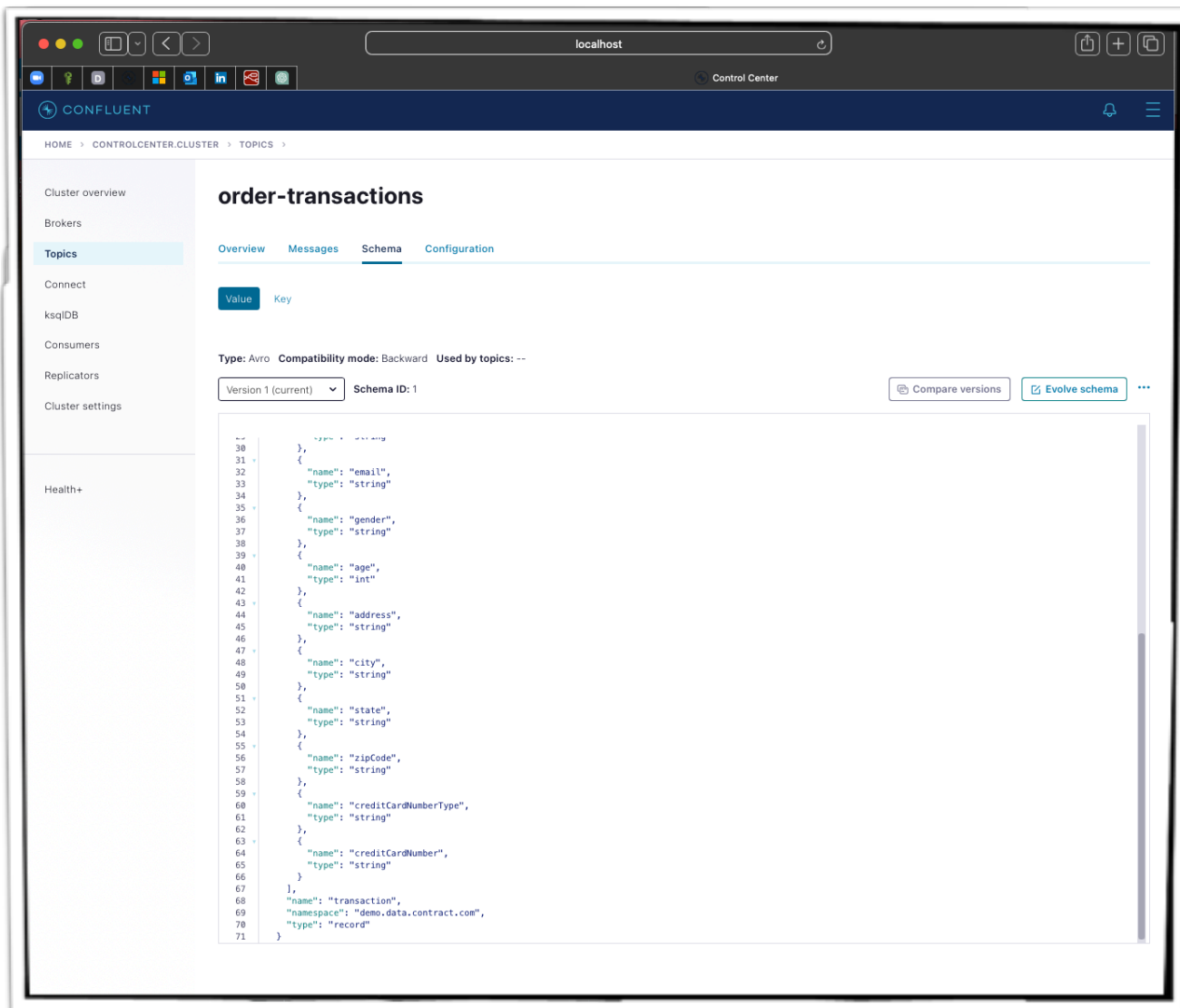


Notice the Topic **order-transactions** we created and used in our first example. Next, Let's drill into the Topic order-transactions. Click on the orders-transactions topic.

Noticed the top of the Messages and Schema links. Click on the Messages link. Then, type 0 in the top left text box. This will allow us to see messages in partition 0.

Notice the message the producer sent to the consumer is present. This is a good tool to observe messages for correctness. Next, go back to the previous page, and click the schema link.

Cluster overview->Topics->order-transactions-schema



The screenshot shows the Confluent Control Center interface for the 'order-transactions' topic. The left sidebar contains navigation links: Cluster overview, Brokers, Topics (selected), Connect, ksqldb, Consumers, Replicators, and Cluster settings. The main content area displays the 'order-transactions' topic with tabs for Overview, Messages, Schema (selected), and Configuration. Below the tabs, there are buttons for 'Value' and 'Key'. The 'Schema' tab shows the current schema version (Version 1) and its ID (1). The schema is defined in Avro format with the following structure:

```

30  },
31  {
32    "name": "email",
33    "type": "string"
34  },
35  {
36    "name": "gender",
37    "type": "string"
38  },
39  {
40    "name": "age",
41    "type": "int"
42  },
43  {
44    "name": "address",
45    "type": "string"
46  },
47  {
48    "name": "city",
49    "type": "string"
50  },
51  {
52    "name": "state",
53    "type": "string"
54  },
55  {
56    "name": "zipCode",
57    "type": "string"
58  },
59  {
60    "name": "creditCardNumberType",
61    "type": "string"
62  },
63  {
64    "name": "creditCardNumber",
65    "type": "string"
66  },
67  ],
68  "name": "transaction",
69  "namespace": "demo.data.contract.com",
70  "type": "record"
71  }

```

Buttons for 'Compare versions' and 'Evolve schema' are visible at the top right of the schema editor.

Notice the Schema we entered, and the *id* is set to 1

The example above demonstrates our initial pipeline with governance for data in motion, ensuring data adheres to a predefined schema. Schema validation is performed on the producer side, near the data source, an approach referred to as a "shift left" pipeline. Validating data as early as possible prevents bad data from propagating through the system. If the schema validation fails, the producer halts processing. While this behavior is technically correct, it may not be optimal for production

environments. In the following examples, we'll explore more practical solutions by integrating rules into the pipeline, which will be included in the Subject.

14. Next, let's introduce some invalid data to test the schema validation. We'll ensure that this data is flagged as invalid because it does not conform to the defined schema. In the **kafka-avro-console-producer** terminal, paste the following invalid data:

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": 4304360364601, "price": 874.34, "productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName": "Sam", "lastName": "Murray", "email": "sam.murray@yahoo.com", "gender": "Male", "age": 12, "address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "5541123132728247"}
```

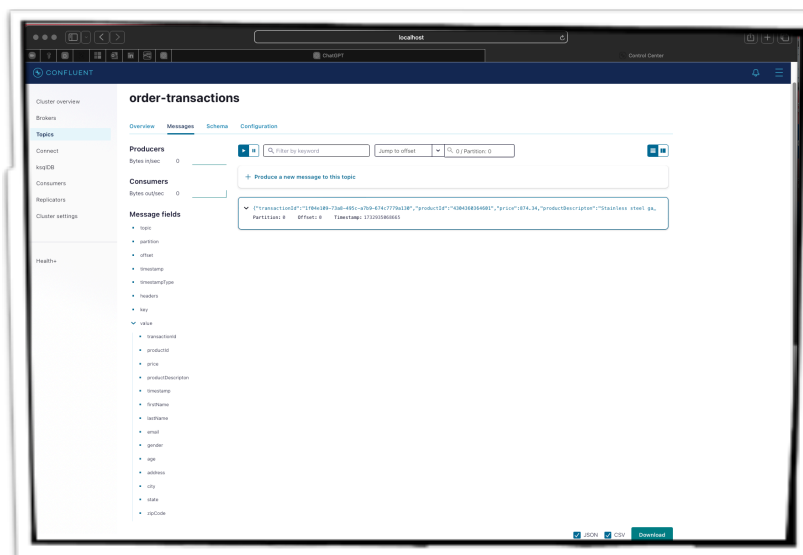
Notice the productId field is not a string but instead is a number. This is enough to violate the schema. We should get the following response.

```
{ "name": "creditCardNumberType", "type": "string" }, { "name": "creditCardNumber", "type": "string" } }
at io.confluent.kafka.formatter.AvroMessageReader.readFrom(AvroMessageReader.java:130)
at io.confluent.kafka.formatter.SchemaMessageReader.readMessage(SchemaMessageReader.java:405)
at kafka.tools.ConsoleProducer$$anon$1$$anon$2.hasNext(ConsoleProducer.scala:67)
at kafka.tools.ConsoleProducer$.loopReader(ConsoleProducer.scala:90)
at kafka.tools.ConsoleProducer$.main(ConsoleProducer.scala:99)
at kafka.tools.ConsoleProducer.main(ConsoleProducer.scala)
```

Caused by: org.apache.avro.AvroTypeException: Expected string. Got VALUE_NUMBER_INT

The **kafka-avro-console-producer** faulted (i.e. crashed) returning the above exception. Let's check the Control Center to make sure the bad message was not produced to the topic order-transactions.

Cluster overview->Topics->order-transactions-messages



We noticed that there is only one message in the topic **order–transactions**. This was the initial message we wrote. The second message, which violated the schema, did not get written to the topic.

Example 2 Data Quality Governance:

In the previous example, we demonstrated how to send Avro messages to a specified topic with the Schema Registry ensuring schema compliance. Now, we'll enhance the pipeline by incorporating Data Quality Governance, which evaluates data values and checks their semantic correctness. Since Avro schemas alone do not natively enforce value constraints for individual fields, we'll leverage **Confluent Platform Data Contracts** to implement these validations.

Continuing from the earlier example, we'll work with the order-transaction schema and introduce Data Quality Rules into the Subject's value schema, representing the pipeline's data. Confluent Platform Data Contracts allow us to define Data Quality Rules using either the **Google Common Expression Language (CEL)** or **JSONata**. Each of these DSLs provides a way to specify constraints. In this example, we'll use CEL, which is generally faster—an important consideration since every message in the topic must undergo validation. Rulesets are associated with a specific subject and can target either the value schema or the key schema. Here, we'll focus on the value schema.

For the order-transaction subject, we'll apply rules to the order-transaction-value schema. In our first Data Quality Rule example, every message written to the order-transaction topic must comply with the following rule: the age of an individual must be greater than 18. The rule is defined as follows:

```
{
  "ruleSet": {
    "domainRules": [
      {
        "name": "checkForMinors",
        "kind": "CONDITION",
        "type": "CEL",
        "mode": "WRITE",
        "expr": "message.age > 17"
      }
    ]
  }
}
```

If you prefer not to download the JSON object from the Git repository, you can copy and paste it into a file named order-transaction-ruleSet-simple.json and save it in the EXAMPLE_HOME directory. Here's a breakdown of the structure: the ruleSet object contains an array called domainRules, where each element represents an individual rule evaluated in sequence. In this case, the rule set contains only one rule.

1. Just for consistency, let's bring down Confluent Platform Kafka Ecosyse. In the EXAMPLE_HOME directory, at the command prompt type,

docker-compose down

or use the docker-compose down plugin with VSC

Expected Response:

** Executing task: docker compose -f "docker-compose.yml" down*

```
[+] Running 10/10
✓ Container control-center    Removed          2.0s
✓ Container ksql-datagen      Removed          10.3s
✓ Container rest-proxy        Removed          1.9s
✓ Container ksqldb-cli         Removed          1.0s
✓ Container ksqldb-server     Removed          1.3s
✓ Container connect           Removed          2.5s
✓ Container schema-registry   Removed          1.2s
✓ Container broker            Removed          10.3s
✓ Container zookeeper         Removed          0.6s
✓ Network confluent-770_default Removed          0.1s
```

Keep in mind that we are not using volumes with docker-compose. This implies that all Subjects, Schema, and RuleSets are deleted when bringing down docker-compose.

2. Again, in the EXAMPL_HOME directory, at the prompt,

docker-compose up

or use the docker-compose up plugin with VSC

Expected Response:

** Executing task: docker compose -f "docker-compose.yml" up -d --build*

```
[+] Running 9/10
🔄 Network confluent-770_default Created          3.0s
✓ Container zookeeper         Started          0.9s
✓ Container broker            Started          1.1s
✓ Container schema-registry   Started          1.4s
✓ Container connect           Started          1.8s
✓ Container rest-proxy        Started          1.7s
✓ Container ksqldb-server     Started          2.0s
✓ Container ksqldb-cli         Started          2.6s
✓ Container ksql-datagen      Started          2.5s
✓ Container control-center    Started
```

2. Once again, in a new terminal, let's add the schema **order-transaction.avsc** to the Schema Registry. Create a new terminal type:

```
jq -n --rawfile schema order-transaction.avsc '{schema: $schema}' |
curl http://localhost:8081/subjects/order-transactions-value/versions --json @-
```

We should get the following response:

```
{“id”:1}
```

3. Next, let's add the RuleSet to the Subject's value schema, type:

```
curl http://localhost:8081/subjects/order-transactions-value/versions \
--json @order-transaction-ruleSet-simple.json
```

We should get the following response:

```
{“id”:2,“version”:2,“ruleSet”:{“domainRules”:
[{"name":“checkForMinors”,“kind”:“CONDITION”,“mode”:“WRITE”,“type”:“CEL”,“expr”:“message.age >
17”,“disabled”:false}],“schema”:{“type”:“record”,“name”:“transaction”,“namespace”:“demo.data.contract.com”,“fields”:[{“name”:
“transactionId”,“type”:“string”,“name”:“productId”,“type”:“string”,“name”:“price”,“type”:“double”,“name”:
“productDescription”,“type”:“string”,“name”:“timestamp”,“type”:“long”,“name”:“firstName”,“type”:“string”,“name”:
“lastName”,“type”:“string”,“name”:“email”,“type”:“string”,“name”:“gender”,“type”:“string”,“name”:“age”,“type”:
“int”,“name”:“address”,“type”:“string”,“name”:“city”,“type”:“string”,“name”:“state”,“type”:“string”,“name”:
“zipCode”,“type”:“string”,“name”:“creditCardNumberType”,“type”:“string”,“name”:“creditCardNumber”,“type”:“string”}]}]}
```

Notice in the top left of the response “id”:2. By adding RuleSet to the Subject value schema, we versioned the value schema by 1. The schema version is now equal to 2.

4. Once again, start the **kafka-arvo-console-consumer**. Start a new terminal at the prompt type:

```
./bin/kafka-avro-console-consumer \
--bootstrap-server localhost:9092 \
--from-beginning --topic order-transactions \
--property schema.registry.url=http://localhost:8081
```



Expected response:

```
use.latest.with.metadata = null
```

```
use.schema.id = -1
value.subject.name.strategy = class io.confluent.kafka.serializers.subject.TopicNameStrategy
(io.confluent.kafka.serializers.KafkaAvroDeserializerConfig:372)
```

The consumer blocks until a message is available on the topic **order-transactions**

5. Next, let's start the kafka-avro-console-producer. Create a new terminal and type:

```
./bin/kafka-avro-console-producer \
--topic order-transactions \
--broker-list localhost:9092 \
--property value.schema.id=2 \
--property bootstrap.servers=localhost:9092
```

Expected response:

```
use.latest.version = false
use.latest.with.metadata = null
use.schema.id = -1
value.subject.name.strategy = class io.confluent.kafka.serializers.subject.TopicNameStrategy
(io.confluent.kafka.serializers.KafkaAvroSerializerConfig:372)
```

The Producer blocks until we paste data to the bottom of the terminal window

6. First, let's add some good data to the kafka-avro-console-producer. Cut and paste the following JSON object to the bottom of **kafka-avro-console-producer** terminal :

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34,
"productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName":
"Amanda", "lastName": "Murray", "email": "amanda.murray@yahoo.com", "gender": "Female", "age": 67,
"address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744",
"creditCardNumberType": "Mastercard", "creditCardNumber": "2541123132728249"}
```

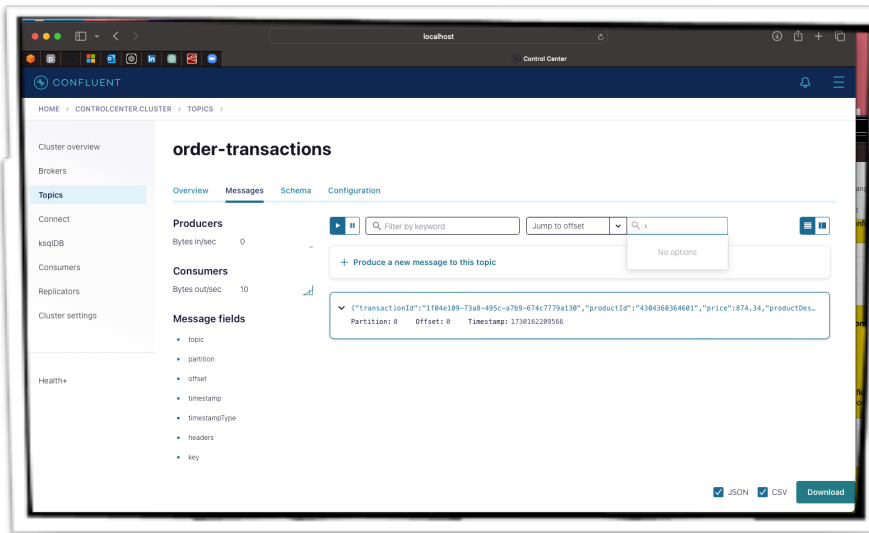
Expected response: Nothing

7. Now, switch to the **kafka-avro-console-consumer** terminal. You should see the following,

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescription": "Stainless steel
garden trowel with ergonomic
handle.", "timestamp": 1729184505, "firstName": "Amanda", "lastName": "Murray", "email": "amanda.murray@yahoo.com", "gender": "Female", "age":
67, "address": "9900 Curtis Field Suite 242", "city": "West
Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Mastercard", "creditCardNumber": "2541123132728249"}
```

As we can see, it worked. The **kafka-arvo-console-consumer** received the message from the producer.

Cluster overview->Topics->order-transactions-messages



This implies that it first passed a schema check, and then passed the rule that the age field must be greater than 1. Check what happened with Control Center,

As we can see, there is only one message on the topic **order-transactions**.

8. Next, Let's try to write some bad data. The age will be set to 13. Go back to the producer terminal and paste the following JSON object

```
{ "transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 87.325,
  "productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName":
  "Doug", "lastName": "Smith", "email": "doug.smith@yahoo.com", "gender": "Male", "age": 13, "address": "9900
  Curtis Field Suite 242", "city": "West Kateland", "state": "VA", "zipCode": "41744", "creditCardNumberType":
  "Visa", "creditCardNumber": "5541123132728247" }
```

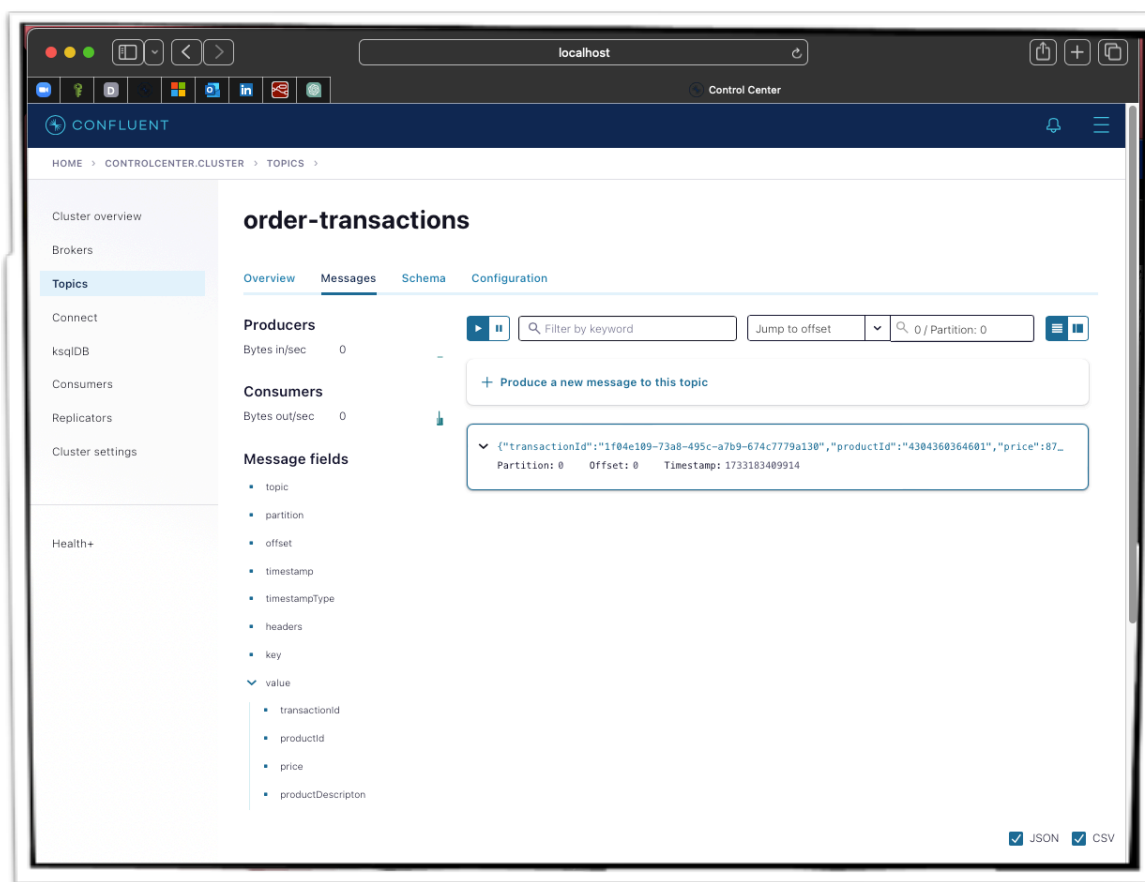
We should have the following response:

```
Caused by: org.apache.kafka.common.errors.SerializationException: Rule failed: checkForMinors
  at io.confluent.kafka.schemaregistry.rules.ErrorAction.run(ErrorAction.java:32)
  at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.runAction(AbstractKafkaSchemaSerDe.java:834)
  at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:732)
  at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:660)
  at io.confluent.kafka.serializers.AbstractKafkaAvroSerializer.serializeImpl(AbstractKafkaAvroSerializer.java:144)
  ... 6 more
Caused by: io.confluent.kafka.schemaregistry.rules.RuleConditionException: Expr failed: 'message.age > 17'
  at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:718)
  ... 8 more
```


The **kafka-avro-console-producer** encountered an error, throwing an exception due to the rule expression "message.age > 17" failing. This behavior is expected for the **kafka-avro-console-producer**, as it is designed primarily for testing and demonstrations. In a production environment, a producer would typically log the exception and continue processing.

9. Let's check in again Control Center,

Cluster overview->Topics->order-transactions-messages



As we can see, the bad message was not written to the topic **order-transactions**. The only message present is the good message sent earlier.

11. Many times, returning an exception and logging it is not enough. This is a note-worthy condition that may occur often. For this scenario, we may want to use what's known as a dead letter queue (DLQ). With the Confluent Platform, a dead letter queue is a separate topic. When a message fails to pass subject-driven constraints, it is written to a separate topic defined in the rule. Below is an example of using a dead letter Q with our previous example.

```
{
  "ruleSet": {
    "domainRules": [
      {
        "name": "checkForMinors",
        "kind": "CONDITION",
        "type": "CEL",
        "mode": "WRITE",
        "expr": "message.age > 17",
        "params": {
          "dlq.topic": "order-transactions-dlq"
        },
        "onFailure": "DLQ"
      }
    ]
  }
}
```

In the above rule, we have added the configuration for a dead letter queue. If the rule “checkForMinors” fails the logic in the “expr”, the message will be written to the topic **order-transactions-dlq**.

6. *If you prefer not to download the JSON object from the Git repository, cut and paste the above JSON object into the file **order-transaction-rule-set-dlq.json** located in the EXAMPLE_HOME directory.*
12. Next, Let's update the RuleSet for the Subject value schema, order-transactions-value the above ruleSet. Start a new terminal and enter:

```
curl http://localhost:8081/subjects/order-transactions-value/versions \
--json @order-transaction-ruleSet-simple-dlq.json
```

13. Return to the faulted **kafka-avro-console-producer** terminal and rerun the producer and type:

```
./bin/kafka-avro-console-producer \  
--topic order-transactions \  
--broker-list localhost:9092 \  
--property value.schema.id=2 \  
--property bootstrap.servers=localhost:9092 \  
--property dlq.auto.flush=true
```

14. Next, Let's add the bad data. Cut and paste the following JSON object to the bottom of the running producer terminal paste:

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 87.325,  
"productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName":  
"Doug", "lastName": "Smith", "email": "doug.smith@yahoo.com", "gender": "Male", "age": 13, "address": "9900  
Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType":  
"Visa", "creditCardNumber": "5541123132728247"}
```

We should get the following response

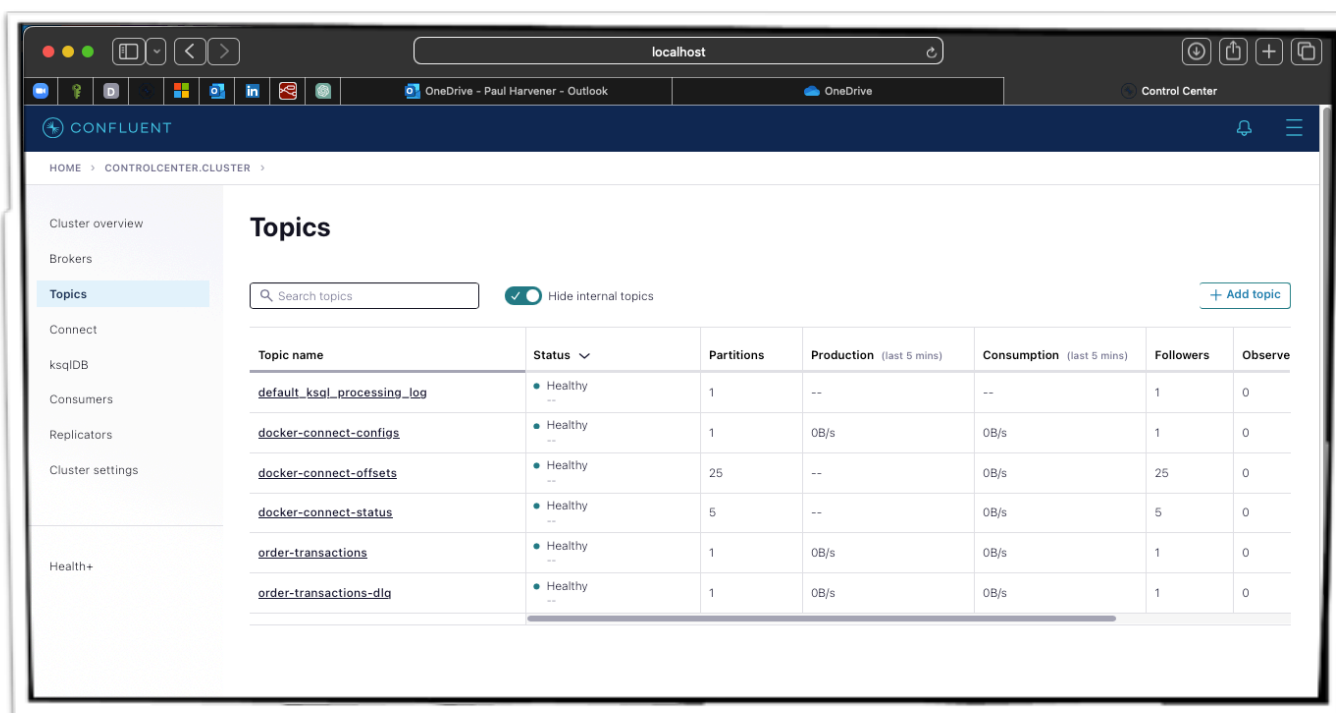
```
Caused by: org.apache.kafka.common.errors.SerializationException: Rule failed: checkForMinors  
at io.confluent.kafka.schemaregistry.rules.DlqAction.run(DlqAction.java:139)  
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.runAction(AbstractKafkaSchemaSerDe.java:834)  
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:732)  
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:660)  
at io.confluent.kafka.serializers.AbstractKafkaAvroSerializer.serializeImpl(AbstractKafkaAvroSerializer.java:144)  
... 6 more  
Caused by: io.confluent.kafka.schemaregistry.rules.RuleConditionException: Expr failed: 'message.age > 17'  
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:718)  
... 8 more
```

Notice we got the same results without the DLQ.

15. Next, Let's check in with the Control Center.

goto URL: <http://localhost:9021/clusters> Topics->Topics

Cluster overview->Topics

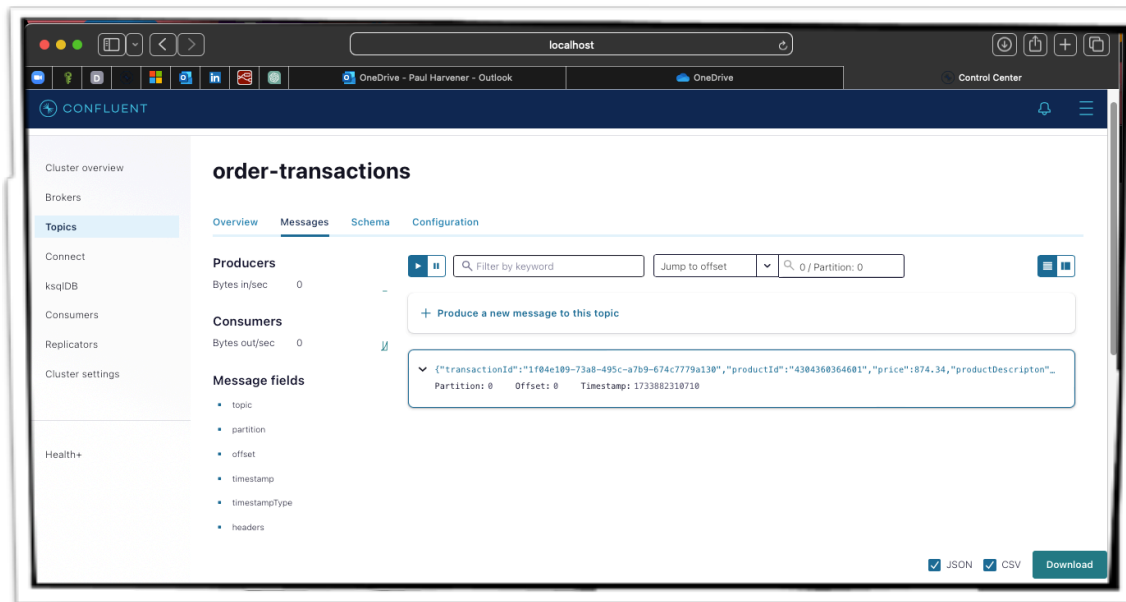


The screenshot shows the Confluent Control Center interface. The left sidebar contains navigation links: Cluster overview, Brokers, Topics (selected), Connect, ksqlDB, Consumers, Replicators, Cluster settings, and Health+. The main content area is titled 'Topics' and includes a search bar, a 'Hide internal topics' toggle, and an '+ Add topic' button. Below these is a table listing topics.

Topic name	Status	Partitions	Production (last 5 mins)	Consumption (last 5 mins)	Followers	Observe
default_ksql_processing_log	Healthy	1	--	--	1	0
docker-connect-configs	Healthy	1	0B/s	0B/s	1	0
docker-connect-offsets	Healthy	25	--	0B/s	25	0
docker-connect-status	Healthy	5	--	0B/s	5	0
order-transactions	Healthy	1	0B/s	0B/s	1	0
order-transactions-dlq	Healthy	1	0B/s	0B/s	1	0

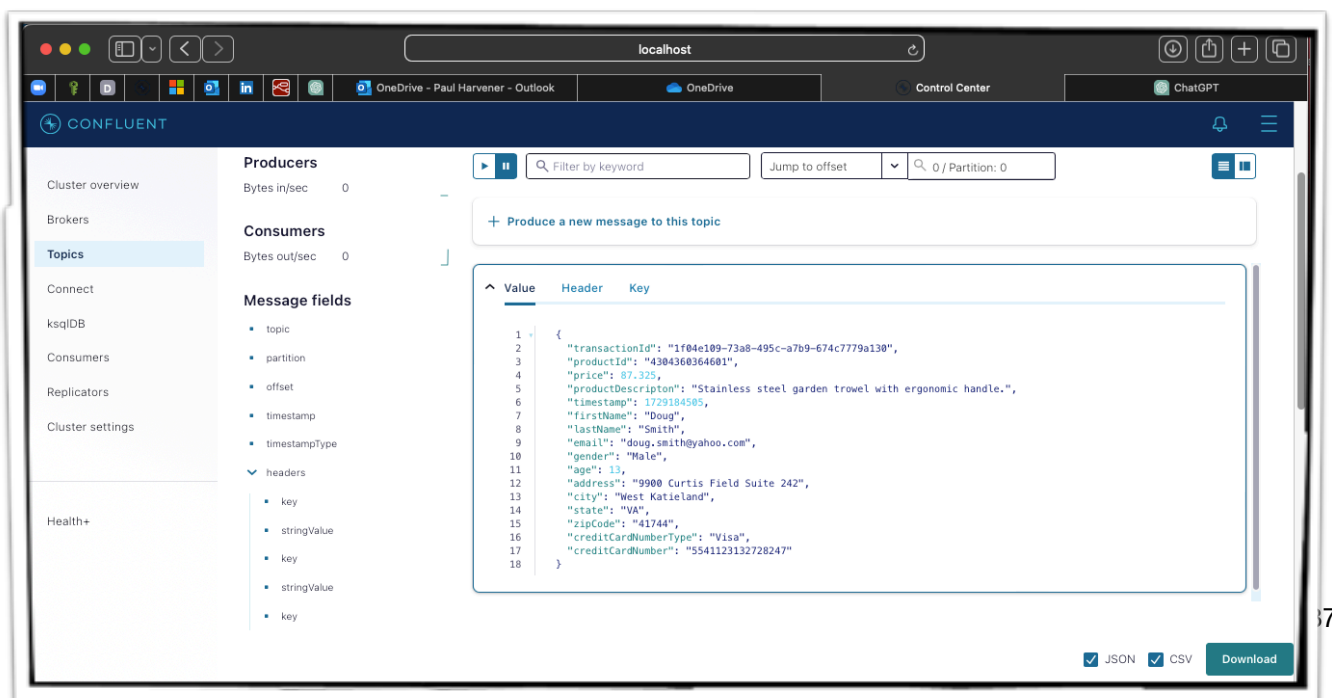
A new topic named **order-transactions-dlq** was automatically created because the broker has auto-creation of topics enabled. This setting is not recommended for production systems. Next, let's examine the `order-transactions` topic to verify that the erroneous message was not written.

Cluster overview->Topics->order-transactions-messages 0



Observe that there is only one message present, the original valid message. Now, let's delve into the **order-transactions-dlq** topic.

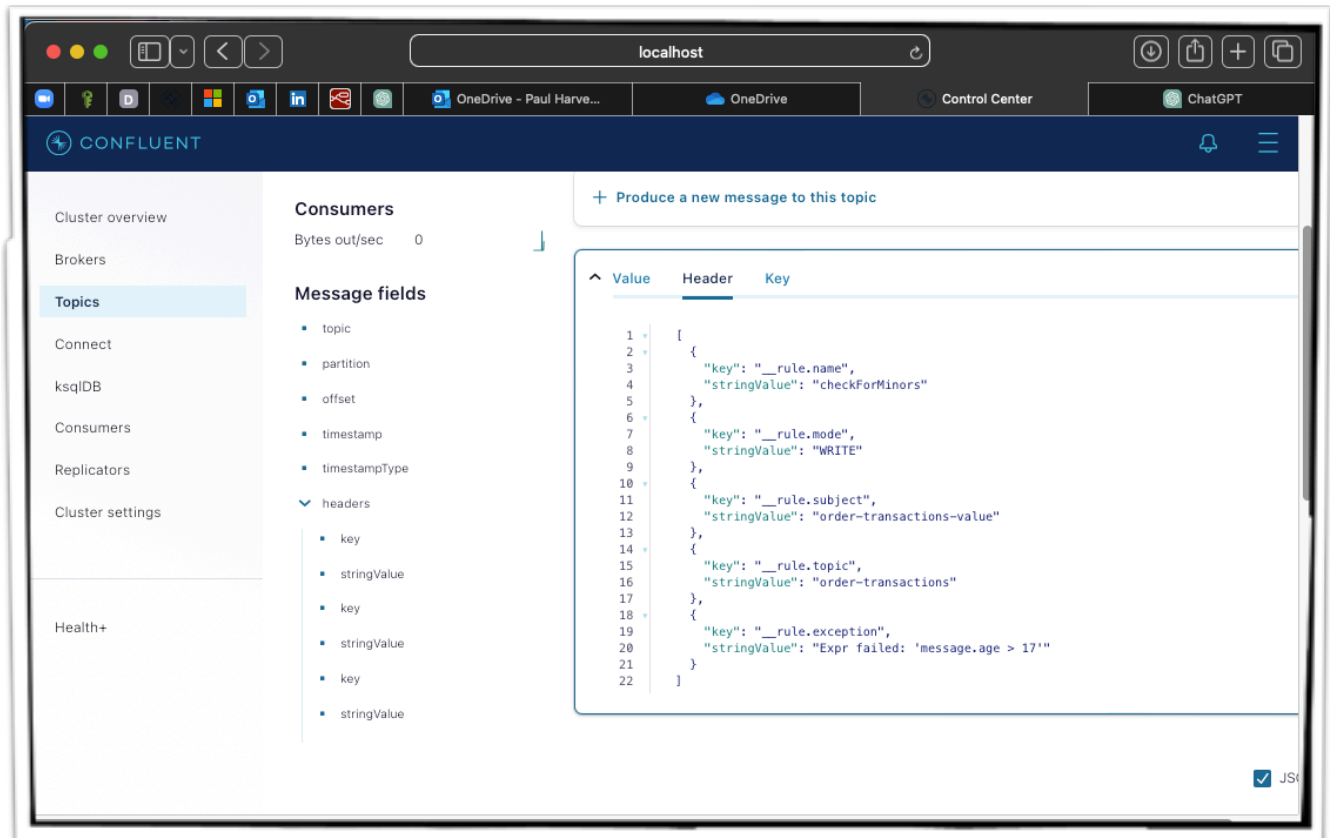
Cluster overview->Topics->order-transactionsdlq—>messages 0



As we can see, the bad message value was written to the DLQ. Next, we will examine the header of the message to determine the reason the message was rejected.

Notice the last entry in the array, `_rule.exception`, with the string value: "Expr failed "message.age >

Cluster overview->Topics->order-transactionsdlq->header 0



17". Dead letter queues play a critical role in managing data governance in motion. Organizations should periodically review the dead letter queue to ensure proper oversight. The Confluent Platform Enterprise Edition offers robust support for implementing semantic governance, enabling the enforcement of message structure and data quality rules. Next, we will explore a more complex use case involving data quality, applying advanced rules through rule-based expressions.

Example 3 Complex Data Quality Governance:

In this example, we will provide semantic governance around the use of credit cards. We will use the **order-transaction.avsc** schema as before. We will force the data to adhere to the following constraints.

- We will allow customers who are 18 years old or older
- We will only support Visa, Mastercard, or AMEX credit cards
- Visa: The leading digit of the credit card number must be 5
- Visa: The card number must be 16 digits long
- Mastercard: The leading digit of the credit card number must be 2
- Mastercard: The card number must be 16 digits long
- AMEX: The leading digit of the credit card number must be 3
- AMEX: The card number must be 15 digits long

1. First, Let's restart docker-compose. This will delete everything done thus far and initialize the environment at the command prompt type,

docker-compose down

or use the docker-compose down plugin with VSC

docker-compose up -d

or use the docker-compose up plugin with VSC

2. Once again, register the AVRO schema **order transaction. avsc** with the Confluent Schema Registry.

jq -n --rawfile schema order-transaction.avsc '{schema: \$schema}' |

curl http://localhost:8081/subjects/order-transactions-value/versions --json @-

3. If you prefer not to download the JSON object from the Git repository, copy and paste the JSON object in Appendix 3 into a file called **order-transaction-ruleSet-complex.json** into the

EXAMPLE_HOME directory. In this example, we will use the Google Common Expression Language CEL to implement the RuleSet

4. Add the following RuleSet to the Subjects **order-transaction-value**, first create a new terminal, issue the following curl command,

```
curl http://localhost:8081/subjects/order-transactions-value/versions \  
--json @order-transaction-ruleSet-complex.json
```

5. Next, let's start **kafka-avro-console-producer** and **kafka-avro-console-consumer**

In terminal already open run the following,

```
./bin/kafka-avro-console-producer \  
--topic order-transactions \  
--broker-list localhost:9092 \  
--property value.schema.id=2 \  
--property bootstrap.servers=localhost:9092 \  
--property dlq.auto.flush=true
```

In a new terminal run the following,

```
./bin/kafka-avro-console-consumer \  
--bootstrap-server localhost:9092 \  
--from-beginning --topic order-transactions \  
--property schema.registry.url=http://localhost:8081
```

6. Let's review the rules associated with the use case. Open the JSON object content in the file order-transaction-ruleSet-complex.json in the EXAMPLE_HOME directory

```
{
```



```

"ruleSet": {
  "domainRules": [
    {
      "name": "customers_under_age_of_18_not_supported",
      "kind": "CONDITION",
      "type": "CEL",
      "mode": "WRITE",
      "expr": "message.age >= 18",
      "params": {
        "dlq.topic": "order-transactions-dlq"
      },
      "onFailure": "DLQ"
    },
    {
      "name": "unsupported_credit_card_type",
      "kind": "CONDITION",
      "type": "CEL",
      "mode": "WRITE",
      "expr": "message.creditCardNumberType in [\"AMEX\", \"Visa\", \"Mastercard\"]",
      "params": {
        "dlq.topic": "order-transactions--dlq"
      },
      "onFailure": "DLQ"
    },
    {
      "name": "visa_card_number_is_not_16_digits_long",
      "kind": "CONDITION",
      "type": "CEL",
      "mode": "WRITE",
      "expr": "message.creditCardNumberType == \"Visa\" ? size(message.creditCardNumber) == 16:true",
      "params": {
        "dlq.topic": "order-transactions-dlq"
      },
      "onFailure": "DLQ"
    },
    {
      "name": "visa_card_number_first_digit_is_not_5",
      "kind": "CONDITION",
      "type": "CEL",
      "mode": "WRITE",
      "expr": "message.creditCardNumberType == \"Visa\" ? message.creditCardNumber.matches(\"^5[0-9]{15}$\"):true",
      "params": {
        "dlq.topic": "order-transactions-dlq"
      },
      "onFailure": "DLQ"
    }
  ]
}

```

The **domainRules** array contains an ordered list of rules, each of which is evaluated separately. The first rule implements the logic from the previous example: all **order transactions** must be initiated by an adult aged 18 or older. If this rule fails, the resulting message will be sent to the dead-letter queue (DLQ).

The next rule verifies the credit card type. Only Visa, MasterCard, or AMEX are accepted. If a transaction includes an unsupported card type, the rule will fail, and the message will be sent to the same DLQ. While it's possible to have separate dead-letter queues for each rule, in this example, a single DLQ suffices. As demonstrated earlier, the header of the message sent to the DLQ includes the failure reason and additional information needed to debug the issue.

The subsequent two rules apply specifically to Visa credit card types. The first ensures that the credit card number is the correct length for Visa cards, 16 digits. If the number is not 16 digits long, this triggers an error. The next rule checks whether the leading digit of the credit card number is five. If the leading digit is not five, and the card type is Visa, this is also considered an error. All errors are routed to the same DLQ.

The remainder of the ruleset applies similar logic for MasterCard and AMEX card types. Note that additional rules can be appended to the **domainRules** array, which represents a sequential chain of rules applied to each record written to the **order-transactions** topic.

7. Next, we will test these rules with a set of messages. First, let's send an **order-transaction** for a Visa card that meets all the requirements. Specifically, the credit card number starts with the digit 5 and is exactly 16 characters long. Copy and paste the following JSON object to the terminal running the **kafka-avro-console-producer**.

```
{ "transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName": "Sam", "lastName": "Murray", "email": "sam.murray@yahoo.com", "gender": "male", "age": 54, "address": "9900 Curtis Field Suite 242", "city": "West Kateland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "5541123132728247" }
```

8. Then let's look at the terminal running the **kafka-avro-console-consumer**. We should see the following,

```
{ "transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName": "Sam", "lastName": "Murray", "email": "sam.murray@yahoo.com", "gender": "male", "age": 54, "address": "9900 Curtis Field Suite 242", "city": "West Kateland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "5541123132728247" }
```

As we can see, the message successfully passed all the rule constraints. The age was greater than 18, the credit card type was Visa, the credit card number was 16 digits long, and it started with the digit 5.

- Next, we will test these rules with an invalid message. Specifically, we'll send a Visa transaction with a credit card number that starts with the digit 8, which violates the rule expecting it to start with a 5. Copy and paste the following JSON object into the terminal running the **kafka-avro-console-producer**.

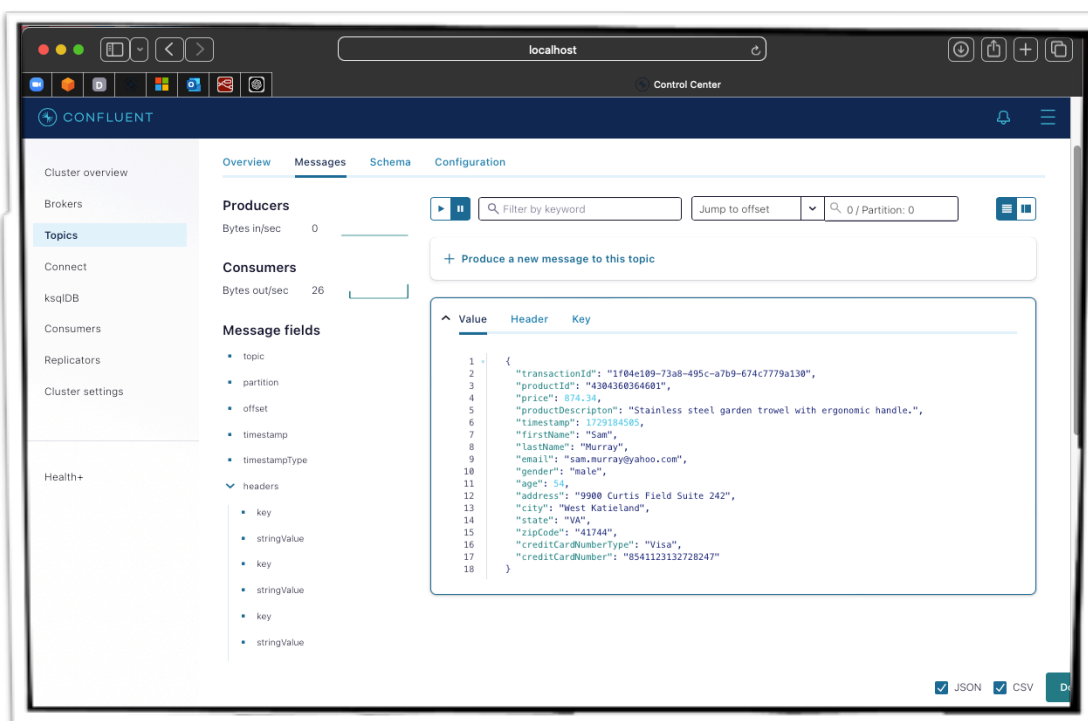
```
{ "transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName": "Sam", "lastName": "Murray", "email": "sam.murray@yahoo.com", "gender": "male", "age": 54, "address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "8541123132728247"} Notice that the kafka-avro-console-producer failed and threw an exception. As we can see, the rule enforcing that the first digit of a Visa card number must be 5 was violated. Consequently, the message did not pass validation and was not sent to the kafka-avro-console-consumer.
```

Caused by: org.apache.kafka.common.errors.SerializationException: Rule failed: visa_card_number_first_digit_is_not_5 at io.confluent.kafka.schemaregistry.rules.DlqAction.run(DlqAction.java:139)
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.runAction(AbstractKafkaSchemaSerDe.java:834)
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:732)
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:660)
at io.confluent.kafka.serializers.AbstractKafkaAvroSerializer.serializeImpl(AbstractKafkaAvroSerializer.java:144)
... 6 more

Caused by: io.confluent.kafka.schemaregistry.rules.RuleConditionException: Expr failed: 'message.creditCardNumberType == "Visa" ? message.creditCardNumber.matches("^5[0-9]{15}\$"):true' at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:718)

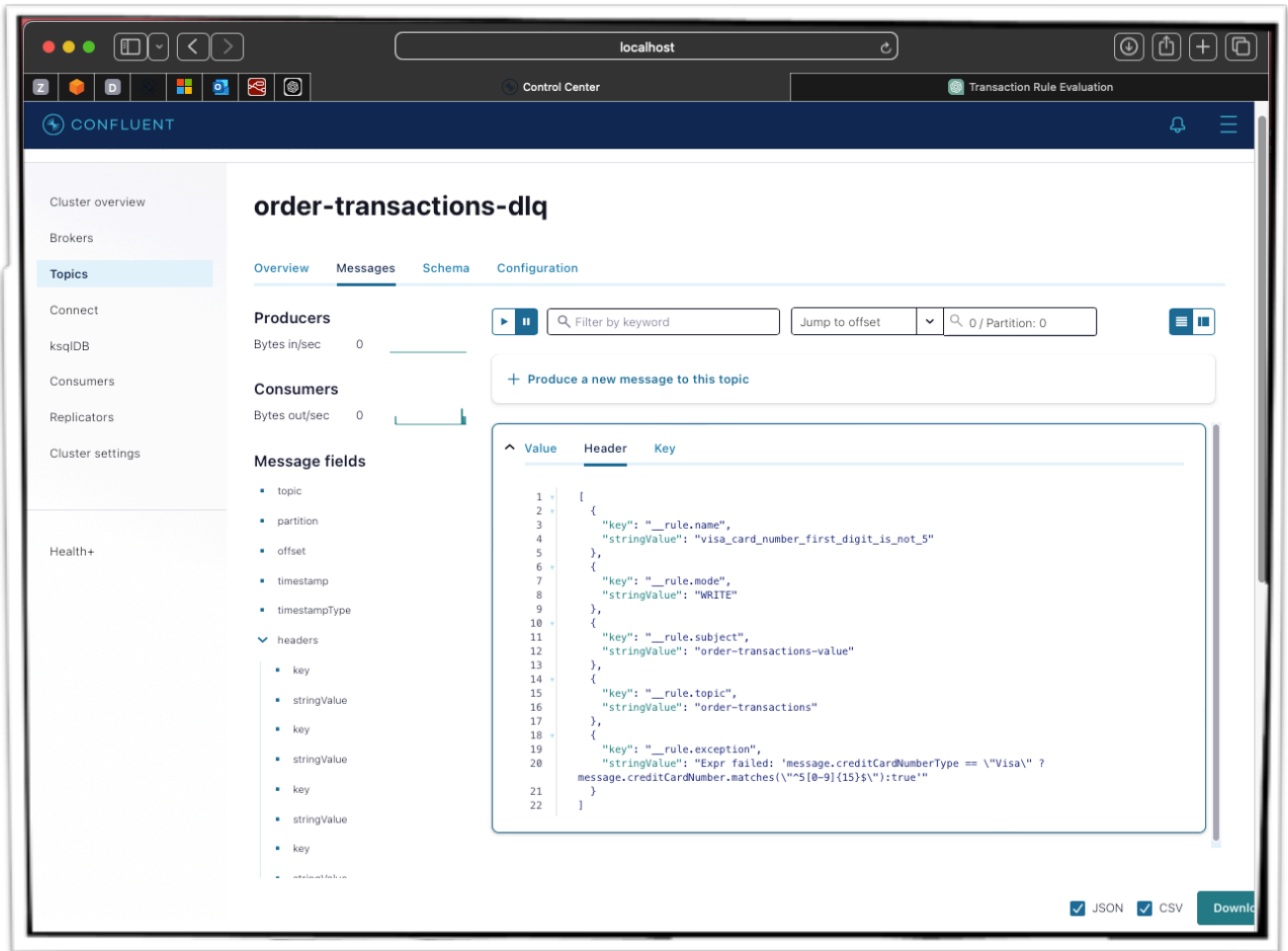
- Let's check the terminal running the **kafka-avro-console-consumer**. As we can see, nothing happened, the faulty message was filtered out and not delivered to the consumer. Now, let's open Control Center and view the dead-letter queue (DLQ) topic.

Cluster overview->Topics->order-transactions-dlq->0->messages->Value



The faulty message is written to the dead-letter queue (DLQ) exactly as it was sent. Notice that the first digit of the credit card number is 8. Next, let's examine the header for this message.

Cluster overview->Topics->order-transactions-dlq->0->messages->Header



As shown in the screenshot above, the header contains metadata about the failed document. The rule that failed is identified by the first object in the array, where `__rule.name` has a stringValue of "visa_card_number_digit_is_not_5".

11. Next, we will test these rules with other invalid message. Specifically, we'll send a Visa transaction with a credit card number that is not 16 digits long, which violates the rule expecting it to be 16 digits long. Copy and paste the following JSON object into the terminal running the **kafka-avro-console-producer**.

```
{ "transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price":
874.34, "productDescripton": "Stainless steel garden trowel with ergonomic handle.", "timestamp":
1729184505, "firstName": "Amanda", "lastName":
"Murray", "email": "amanda.murray@yahoo.com", "gender": "Female", "age": 67, "address": "9900 Curtis
Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType":
"Visa", "creditCardNumber": "554112313272824" }
```

We get the following response,

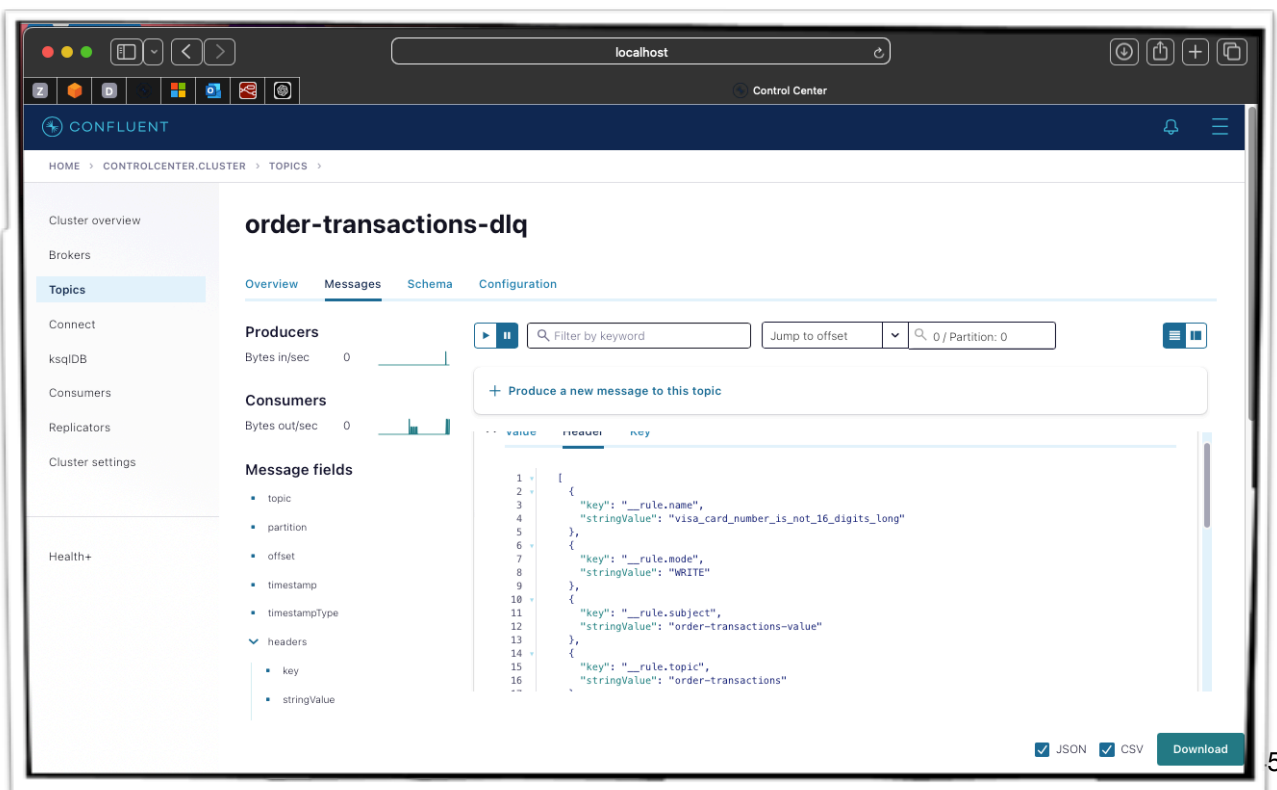
*Caused by: org.apache.kafka.common.errors.SerializationException: Rule failed: visa_card_number_is_not_16_digits_long
at io.confluent.kafka.schemaregistry.rules.DlqAction.run(DlqAction.java:139)
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.runAction(AbstractKafkaSchemaSerDe.java:834)
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:732)
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:660)
at io.confluent.kafka.serializers.AbstractKafkaAvroSerializer.serializeImpl(AbstractKafkaAvroSerializer.java:144)
... 6 more*

*Caused by: io.confluent.kafka.schemaregistry.rules.RuleConditionException: Expr failed: 'message.creditCardNumberType ==
"Visa" ? size(message.creditCardNumber) == 16:true'
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:718)
... 8 more*

Lets check the dead letter queue (DLQ),

The new faulty message has been added to the dead letter queue, as observed.\

Title



The screenshot shows the Confluent Control Center interface for the 'order-transactions-dlq' topic. The 'Messages' tab is selected, showing a list of messages. The message fields include topic, partition, offset, timestamp, timestampType, and headers. The headers section shows a key-value pair: 'key' with value 'order-transactions-value'. The message value is a JSON object containing rule information, including the rule name 'visa_card_number_is_not_16_digits_long' and the rule mode 'WRITE'.

12. What happens if more than one rule fails. How is this accounted for? The next input object will have the wrong length for the credit card number and it will not start with 5. Copy and paste the following JSON object into the terminal running the **kafka-avro-console-producer**.

```
{ "transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName": "Amanda", "lastName": "Murray", "email": "amanda.murray@yahoo.com", "gender": "Female", "age": 67, "address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "854112313272824" }
```

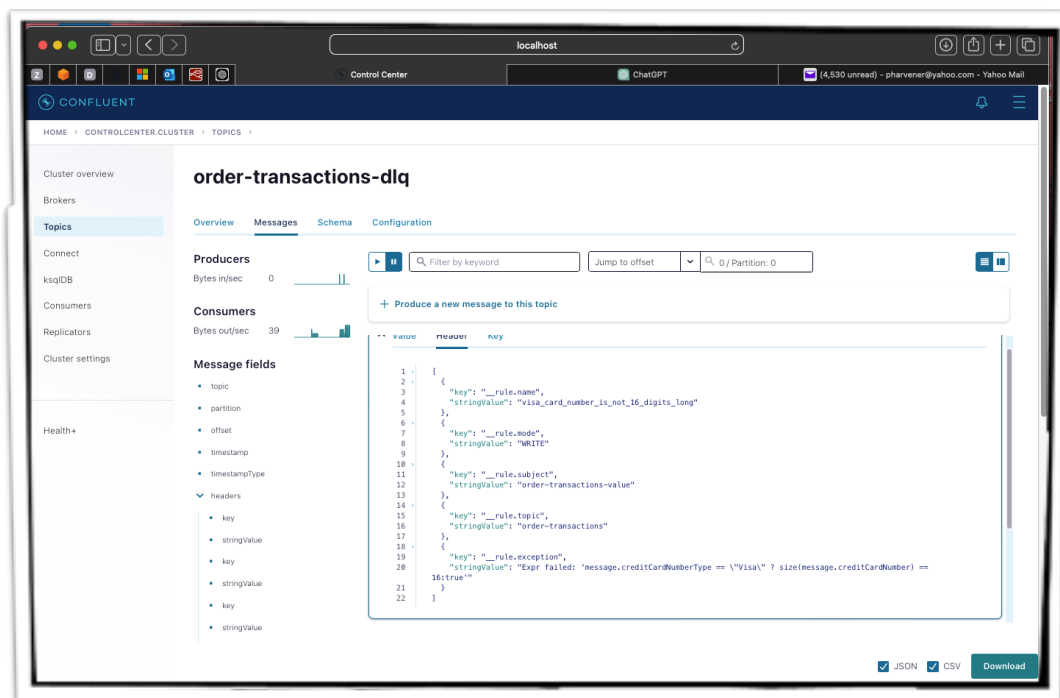
We get the following results,

*Caused by: org.apache.kafka.common.errors.SerializationException: Rule failed: visa_card_number_is_not_16_digits_long at io.confluent.kafka.schemaregistry.rules.DlqAction.run(DlqAction.java:139)
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.runAction(AbstractKafkaSchemaSerDe.java:834)
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:732)
at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:660)
at io.confluent.kafka.serializers.AbstractKafkaAvroSerializer.serializeImpl(AbstractKafkaAvroSerializer.java:144)
... 6 more*

Caused by: io.confluent.kafka.schemaregistry.rules.RuleConditionException: Expr failed: 'message.creditCardNumberType == "Visa" ? size(message.creditCardNumber) == 16:true'

at io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe.executeRules(AbstractKafkaSchemaSerDe.java:718)

It seems that only the first failing rule is reported in the exception stack trace. Let's examine the dead letter queue and the message header.



Control Center produces the same results: only the first failing rule is reported. This behavior makes sense because, in the rule set, the check for the credit card number length precedes the check for whether the credit card number starts with 5.

Example 4: Data Transformation Governance:

As mentioned earlier, data transformation is a critical use case. Using a "shift left" approach, we aim to apply data transformations as early in the pipeline as possible. This example demonstrates how to govern data transformations using the Confluent Platform Data Contracts framework. Specifically, we will introduce a schema migration that forces a breaking change. For this demonstration, we will use AVRO as the messaging format for streaming data.

The Confluent Platform Data Contracts framework supports breaking schema changes by partitioning schema versions into compatibility groups. This capability is essential for ensuring that both legacy and updated consumers can process messages correctly, regardless of schema changes.

To achieve this, we use migration rules implemented with JSONata, a lightweight query and transformation language for JSON data. These migration rules enable seamless schema compatibility across different versions, ensuring smooth communication between producers and consumers even when schema-breaking changes occur.

In this case, two primary rules are defined: the UPGRADE rule and the DOWNGRADE rule. The UPGRADE rule allows new consumers, who expect the current schema, to read messages from older schema versions. Conversely, the DOWNGRADE rule allows older consumers, who expect the previous schema, to read messages that use the current schema.

If the system is planning to upgrade all consumers to the latest schema version, the DOWNGRADE rule can be omitted, as there will no longer be a need to support the older schema.

In each migration rule, we leverage the JSONata function called `$sift()`, which allows for the transformation of fields within the data. The `$sift()` function removes a field by its name and enables the addition of a new field with a different name. This approach ensures that schema changes, such as removing attributes or renaming fields, can be handled dynamically, allowing both old and new consumers to process the messages without disruption.

By using migration rules in this way, data transformation governance ensures smooth compatibility across schema versions, even when breaking changes occur. This is a powerful feature for managing data evolution while maintaining flexibility and ensuring uninterrupted data flows for both producers and consumers. Below is a simple form of Migration Rules discussed.

```
"migrationRules": [
  {
    "name": "changeFirstNameToFirst_name",
    "kind": "TRANSFORM",
    "type": "JSONATA",
    "mode": "UPGRADE",
    "expr": "$merge([$sift($,function($v,$k) {$k != 'firstName'}),{'first_name': $. 'firstName'}})]"
  },
  {
    "name": "changeFirst_nameToFirstName",
    "kind": "TRANSFORM",
    "type": "JSONATA",
    "mode": "DOWNGRADE",
    "expr": "$merge([$sift($,function($v,$k) {$k != 'first_name'}),{'firstName': $. 'first_name'}})]"
  }
]
```

In this data transformation governance setup, the discussed migration rules will be added to the existing rule set within the data contract framework. As outlined, we have two primary rules to manage schema evolution: a UPGRADE rule for transforming older schema messages to be readable by consumers using the latest schema, and a DOWNGRADE rule for making newer schema messages accessible to consumers expecting an older schema.

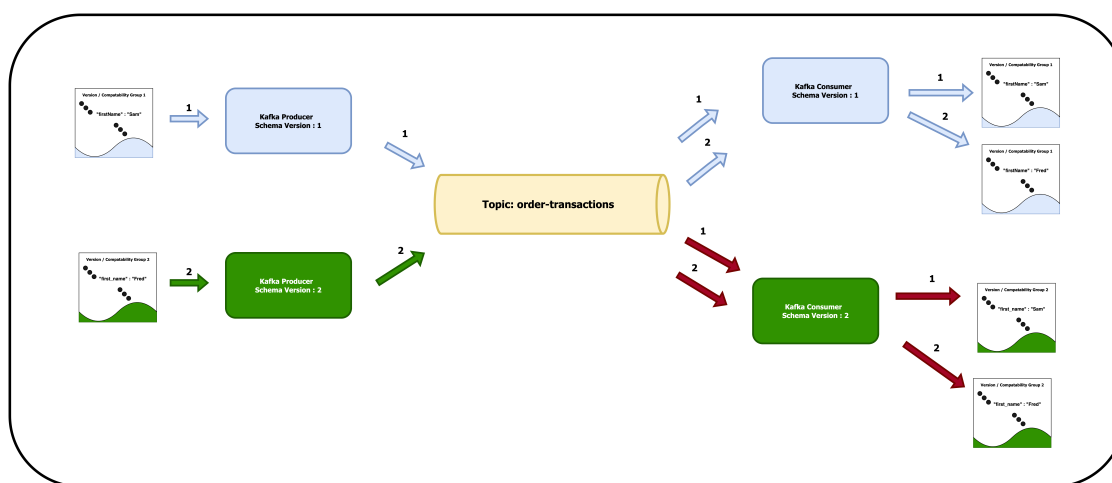
The application of these rules is determined by a compatibility group attribute in the metadata. This attribute identifies the schema version, allowing the system to determine whether to apply the UPGRADE or DOWNGRADE rule based on the compatibility needs of each consumer. By embedding the compatibility group in the metadata, we achieve a flexible, version-aware transformation process, which can dynamically adjust based on the specific schema being consumed.

An example of the metadata controlling the major version will look like this

```
{
  "metadata": {
    "compatibility_group": 2
  }
}
```

By incorporating JSONata into these migration rules, we enable precise, rule-based control over data transformation, ensuring consistent compatibility across versions and maintaining data integrity as schema evolution continues.

Below is a diagram illustrating this example. It features two separate Kafka producers. The first producer sends messages using the original schema, `order-transactions`, while the second sends messages using the new schema, `order-messages-migration`. Each schema belongs to a distinct compatibility group. The key difference between these schemas is the naming convention for a field: the original schema uses `firstName`, while the new schema uses `first_name`. This setup demonstrates a scenario where downstream consumers include two types, one expecting the original schema and the other expecting the new schema. As noted, the difference between these schemas introduces a breaking change to the pipeline. Migration rules resolve this issue, enabling seamless communication across both data sources.



Next, two consumers are initiated: one expects messages conforming to the old schema (compatibility group 1) and is represented in light blue, while the other expects messages conforming to the new schema (compatibility group 2) and is represented in red.

The top Kafka producer (light blue) sends message 1 using the original schema. This message is written to the `order-transactions` topic. Both the existing data quality rules and the new migration rules are applied. As a result, the top consumer (light blue) receives the message in its original schema format, as expected. Simultaneously, the message is transformed to conform to the new schema and delivered to the second consumer (red), ensuring compatibility.

Next, the bottom Kafka producer (red) sends message 2 using the new schema. This message is translated to the original schema format and sent to the top consumer (light blue), meeting its expectations for the old schema. At the same time, message 2 is delivered to the bottom consumer (red) in its original form, conforming to the new schema.

Let's run the example:

1. First, Let's restart docker-compose. This will delete everything done thus far and initialize the environment.

2. Next, Let's run example 3 over again. This will represent the existing pipeline. We will later add the migration rules to the rule set. We will briefly go over this sequence of actions.

In a new terminal:

```
jq -n --rawfile schema order-transaction.avsc "{schema: $schema}" |
curl http://localhost:8081/subjects/order-transactions-value/versions --json @-

curl http://localhost:8081/subjects/order-transactions-value/versions \
--json @order-transaction-ruleSet-complex.json

./bin/kafka-avro-console-producer \
--topic order-transactions \
--broker-list localhost:9092 \
--property value.schema.id=2 \
--property bootstrap.servers=localhost:9092 \
--property dlq.auto.flush=true
```

In a new terminal:

```
./bin/kafka-avro-console-consumer \
--bootstrap-server localhost:9092 \
--from-beginning --topic order-transactions \
--property schema.registry.url=http://localhost:8081
```

In the first terminal (i.e. producer terminal), send the following valid JSON Document.

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName": "Amanda", "lastName": "Murray", "email": "amanda.murray@yahoo.com", "gender": "Female", "age": 67, "address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "5541123132728247"}
```

Then in the consumer terminal, we should see the following.

```
(io.confluent.kafka.serializers.KafkaAvroDeserializerConfig:372)
{"transactionId":"1f04e109-73a8-495c-a7b9-674c7779a130", "productId":"4304360364601", "price":874.34, "productDescription":"Stainless steel garden trowel with ergonomic handle.", "timestamp":1729184505, "firstName":"Amanda", "lastName":"Murray", "email":"amanda.murray@yahoo.com", "gender":"Female", "age":67, "address":"9900 Curtis Field Suite 242", "city":"West Katieland", "state":"VA", "zipCode":"41744", "creditCardNumberType":"Visa", "creditCardNumber":"5541123132728247"}
```

Notice everything is working as before.

3. Next, Let's create a breaking change to the order-transaction-value schema. Copy and paste the JSON document in Appendix 4 into a file called order-transaction-value-breaking-change in the EXAMPLE_HOME directory.

Notice that we removed the field first name and replaced it with a field first_name. This is a breaking change. None of the schema of migration change methods will support this.

4. Create a new Terminal and enter the following command.

```
jq -n --rawfile schema order-transaction-breaking-change.avsc "{schema: $schema}" |  
curl http://localhost:8081/subjects/order-transactions-value/versions --json @-
```

And we get the following response.

```
{  
  "error_code": 409,  
  "message": "Schema being registered is incompatible with an earlier schema for subject \"order-transactions-value\", details:  
  [{  
    "errorType": "READER_FIELD_MISSING_DEFAULT_VALUE",  
    "description": "The field \"first_name\" at path \"/fields/5\" in the new schema has  
    no default value and is missing in the old schema",  
    "additionalInfo": "first_name",  
    "oldSchemaVersion": 2,  
    "oldSchema": "{  
      \"type\": \"record\",  
      \"name\": \"transaction\",  
      \"namespace\": \"demo.data.contract.com\",  
      \"fields\": [{  
        \"name\": \"transactionId\",  
        \"type\": \"string\",  
        \"name\":  
        \"productId\",  
        \"type\": \"string\",  
        \"name\": \"price\",  
        \"type\": \"double\",  
        \"name\": \"productDescription\",  
        \"type\": \"string\",  
        \"name\":  
        \"timestamp\",  
        \"type\": \"long\",  
        \"name\": \"firstName\",  
        \"type\": \"string\",  
        \"name\": \"lastName\",  
        \"type\": \"string\",  
        \"name\":  
        \"email\",  
        \"type\": \"string\",  
        \"name\": \"gender\",  
        \"type\": \"string\",  
        \"name\": \"age\",  
        \"type\": \"int\",  
        \"name\": \"address\",  
        \"type\":  
        \"string\",  
        \"name\": \"city\",  
        \"type\": \"string\",  
        \"name\": \"state\",  
        \"type\": \"string\",  
        \"name\": \"zipCode\",  
        \"type\": \"string\",  
        \"name\": \"creditCardNumberType\",  
        \"type\": \"string\",  
        \"name\": \"creditCardNumber\",  
        \"type\": \"string\"  
      }],  
      \"validateFields\": false,  
      \"compatibility\": \"BACKWARD\"  
    }  
  ]  
}
```

Notice that this resulted in an error, with the message clearly identifying the issue: a breaking change. The content compatibility mode here is set to "BACKWARD", which is the default schema evolution mode. However, this breaking change would still not have been resolved even if the schema evolution mode were set to "FORWARD" or "FULL." Without migration rules, we would be limited to evolving our schema strictly within these compatibility constraints, unable to effectively handle breaking changes.

5. Next, Let's create the migration rules to handle this breaking change within the existing domain rule set. Copy the JSON object from Appendix 4 and paste it into a file named order-transaction-ruleset-complex-migration.json, then save this file in the EXAMPLE_HOME directory.
6. Start by configuring the Schema Registry to perform compatibility checks only within defined compatibility groups. Start a new terminal and enter,

```
curl http://localhost:8081/config/order-transactions-value \  
-X PUT --json "{ \"compatibilityGroup\": \"compatibility_group\" }"
```

We should get the following response,

```
{  
  \"compatibilityGroup\": \"compatibility_group\"  
}
```

Going forward, for all schema versions of the subject order-transaction-value, the compatibility_group field within the metadata object will specify the Compatibility Group associated with each schema version.

- Next, add metadata to the subject's value schema to segment breaking changes across different compatibility groups. This is achieved by embedding the metadata JSON object directly within the old/existing schema definition. Enter the following into the terminal,

```
jq -n --rawfile schema order-transaction.avsc "{ schema: \$schema, metadata: { properties: { compatibility_group: 1 } } }" |
curl http://localhost:8081/subjects/order-transactions-value/versions --json @-
```

We should get the following response,

```
{
  "id": 1,
  "version": 1,
  "metadata": {
    "properties": {
      "compatibility_group": "1"
    }
  },
  "schema": {
    "type": "record",
    "name": "transaction",
    "namespace": "demo.data.contract.com",
    "fields": [
      {
        "name": "transactionId",
        "type": "string"
      },
      {
        "name": "productId",
        "type": "string"
      },
      {
        "name": "price",
        "type": "double"
      },
      {
        "name": "productDescription",
        "type": "string"
      },
      {
        "name": "timestamp",
        "type": "long"
      },
      {
        "name": "first_name",
        "type": "string"
      },
      {
        "name": "last_name",
        "type": "string"
      },
      {
        "name": "email",
        "type": "string"
      },
      {
        "name": "gender",
        "type": "string"
      },
      {
        "name": "age",
        "type": "int"
      },
      {
        "name": "address",
        "type": "string"
      },
      {
        "name": "city",
        "type": "string"
      },
      {
        "name": "state",
        "type": "string"
      },
      {
        "name": "zipCode",
        "type": "string"
      },
      {
        "name": "creditCardNumberType",
        "type": "string"
      },
      {
        "name": "creditCardNumber",
        "type": "string"
      }
    ]
  }
}
```

- Now, Let's add the metadata to the subject order-transaction-value schema to allow breaking changes into different compatibility group. We do this by adding the metadata JSON object in line with the new schema definition. Let's version the old schema to the new. In a new terminal enter the following.

```
jq -n --rawfile schema order-transaction-migration.avsc "{ schema: \$schema, metadata: { properties: { compatibility_group: 2 } } }" |
curl http://localhost:8081/subjects/order-transactions-value/versions --json @-
```

We should get the following response,

```
{
  "id": 2,
  "version": 2,
  "metadata": {
    "properties": {
      "compatibility_group": "2"
    }
  },
  "schema": {
    "type": "record",
    "name": "transaction",
    "namespace": "demo.data.contract.com",
    "fields": [
      {
        "name": "transactionId",
        "type": "string"
      },
      {
        "name": "productId",
        "type": "string"
      },
      {
        "name": "price",
        "type": "double"
      },
      {
        "name": "productDescription",
        "type": "string"
      },
      {
        "name": "timestamp",
        "type": "long"
      },
      {
        "name": "first_name",
        "type": "string"
      },
      {
        "name": "last_name",
        "type": "string"
      },
      {
        "name": "email",
        "type": "string"
      },
      {
        "name": "gender",
        "type": "string"
      },
      {
        "name": "age",
        "type": "int"
      },
      {
        "name": "address",
        "type": "string"
      },
      {
        "name": "city",
        "type": "string"
      },
      {
        "name": "state",
        "type": "string"
      },
      {
        "name": "zipCode",
        "type": "string"
      },
      {
        "name": "creditCardNumberType",
        "type": "string"
      },
      {
        "name": "creditCardNumber",
        "type": "string"
      }
    ]
  }
}
```

- Now, we are ready to start 2 separate consumers. one for compatibility group 1 and the other for compatibility group 2. Create a new terminal for each and enter the following in the first terminal, This will be for consuming messages in compatibility group 1,

```
./bin/kafka-avro-console-consumer \
--topic order-transactions \
--bootstrap-server localhost:9092 \
--property auto.register.schemas=false
```

We should get the following response,



```
value.subject.name.strategy = class io.confluent.kafka.serializers.subject.TopicNameStrategy
(io.confluent.kafka.serializers.KafkaAvroDeserializerConfig:372)
```

Enter the following in the second terminal, This will be for compatibility group 2

```
./bin/kafka-avro-console-consumer \  
--topic order-transactions \  
--bootstrap-server localhost:9092 \  
--property auto.register.schemas=false \  
--property use.latest.version=true
```

We should get the following response,



```
value.subject.name.strategy = class io.confluent.kafka.serializers.subject.TopicNameStrategy  
(io.confluent.kafka.serializers.KafkaAvroDeserializerConfig:372)
```

Notice that we started 2 consumers on the same topic. By applying the Migration Rules, specifically the UPGRADE and DOWNGRADE rules, the producer can send messages that comply with either the original schema or the new schema that addresses the breaking changes.

Each consumer will receive the version specific to their designated compatibility group, ensuring seamless compatibility across schema versions, Let's run two producers; one for the original schema, compatibility group 1, and another producer to produce messages for the new schema, compatibility group 2. Start terminal for a producer using the original schema or compatibility group 1.

```
./bin/kafka-avro-console-producer \  
--topic order-transactions \  
--broker-list localhost:9092 \  
--property value.schema.id=1 \  
--property bootstrap.servers=localhost:9092 \  
--property dlq.auto.flush=true
```

We should get the following response,



```
value.subject.name.strategy = class io.confluent.kafka.serializers.subject.TopicNameStrategy
(io.confluent.kafka.serializers.KafkaAvroSerializerConfig:372)
```

Next start a terminal for a producer using the new schema or compatibility group 2,

```
./bin/kafka-avro-console-producer \
--topic order-transactions \
--broker-list localhost:9092 \
--property value.schema.id=2 \
--property bootstrap.servers=localhost:9092 \
--property dlq.auto.flush=true
```

We should get the following response,



```
value.subject.name.strategy = class io.confluent.kafka.serializers.subject.TopicNameStrategy
(io.confluent.kafka.serializers.KafkaAvroSerializerConfig:372)
```

11. Now we have the complete scenario setup as described in the above diagram. First Let's a message compliant with the original schema, compatibility group 1. Cut and paste the following JSON object in the producer terminal representing compatibility group one.

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName": "Sam", "lastName": "Murray", "email": "sam.murray@yahoo.com", "gender": "Female", "age": 67, "address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "5541123132728247"}
```

12. Now Let's check both of the consumers to verify the correct version of the message has been delivered to the consumer expecting the original schema and the other consumer expecting the new scheme. Below is the response found in the consumer expecting the original schema.

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescription": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName": "Sam", "lastName": "Murray", "email": "sam.murray@yahoo.com", "gender": "Female", "age": 67, "address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "5541123132728247"}
```

Notice the field “firstName”:”Sam” is in the correct format for the original schema.

Next, Let's look at the response from consumers expecting the message to be translated to the schema.

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescripton": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "first_name": "Sam", "lastName": "Murray", "email": "sam.murray@yahoo.com", "gender": "Female", "age": 67, "address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "5541123132728247"}
```

Notice the field “first_name”:”Sam” is in the correct format for the new schema. The UPGRADE migration worked. The produced message was in the old format, but the upgrade rule upgraded the message to the new scheme format, which is the format this producer expected

13. Now Let's try sending the message in the new scheme format and see how the DOWNGRADE migration roll worked. Cut and paste the following JSON object onto the producer expecting the new schema, compatibility group 2.

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescripton": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "first_name": "Fred", "lastName": "Murray", "email": "fred.murray@yahoo.com", "gender": "Female", "age": 67, "address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "5541123132728247"}
```

14. Now Let's check both of the consumers to verify the correct version of the message has been delivered to the consumer expecting the original schema and the other consumer expecting the new scheme. Below is the response found in the consumer expecting the original schema.

```
{"transactionId": "1f04e109-73a8-495c-a7b9-674c7779a130", "productId": "4304360364601", "price": 874.34, "productDescripton": "Stainless steel garden trowel with ergonomic handle.", "timestamp": 1729184505, "firstName": "Sam", "lastName": "Murray", "email": "sam.murray@yahoo.com", "gender": "Female", "age": 67, "address": "9900 Curtis Field Suite 242", "city": "West Katieland", "state": "VA", "zipCode": "41744", "creditCardNumberType": "Visa", "creditCardNumber": "5541123132728247"}
```

Notice the field “firstName”:Sam” is in the correct format for the original schema.

```
{"transactionId":"1f04e109-73a8-495c-a7b9-674c7779a130","productId":"4304360364601","price":874.34,"productDescription":"Stainless steel garden trowel with ergonomic handle","timestamp":1729184505,"firstName":"Fred","lastName":"Murray","email":"fred.murray@yahoo.com","gender":"Female","age":67,"address":"9900 Curtis Field Suite 242","city":"West Katieland","state":"VA","zipCode":"41744","creditCardNumberType":"Visa","creditCardNumber":"5541123132728247"}
```

Notice the field “firstName”:Fred” is in the correct format for the original schema.

Knowledge check to consumer expecting the new format, compatibility group 2

```
{"transactionId":"1f04e109-73a8-495c-a7b9-674c7779a130","productId":"4304360364601","price":874.34,"productDescription":"Stainless steel garden trowel with ergonomic handle","timestamp":1729184505,"first_name":"Fred","lastName":"Murray","email":"fred.murray@yahoo.com","gender":"Female","age":67,"address":"9900 Curtis Field Suite 242","city":"West Katieland","state":"VA","zipCode":"41744","creditCardNumberType":"Visa","creditCardNumber":"5541123132728247"}
```

Notice the field “first_name”:Fred” is in the correct format for the new schema.

The migration rules behaved as expected. The downgrade migration rule was executed to take the message produced in the new format, compatibility group 2 and downgraded it to the old schema for the consumer expecting the old schema.

The upgrade migration rule also worked correctly. When the message was produced using the old schema, it upgraded the message to the new Schemin, which that particular consumer was expecting.

As shown in his overall example, we can use consulate platform data contracts using migration rules to mediate the scheme of migration that involves breaking changes. This is a very powerful tool Dillon has multiple versions of a given scheme to be used once. It provides a service level

Conclusion: Governance with Data-in-Motion

The implementation of robust governance for data-in-motion is essential for ensuring data quality, consistency, and compliance in modern distributed systems. By leveraging tools such as the Confluent Schema Registry, Data Contracts, and associated governance frameworks, organizations can establish centralized, enforceable standards that streamline data management and improve pipeline reliability.

Key components like schema validation, metadata enrichment, and dynamic rule sets enable both structural and semantic data governance at the source, embodying the "shift-left" philosophy. This approach minimizes downstream errors and reduces the operational burden on consumers. Advanced features such as compatibility groups and migration rules further enhance adaptability, allowing systems to evolve seamlessly while maintaining compatibility across schema versions.

Through practical examples and clear methodologies, this paper demonstrates how to design and execute governance strategies that balance real-time enforcement with long-term data lifecycle management. As data-driven architectures continue to scale, effective governance of data-in-motion ensures not only operational efficiency but also adherence to critical business, regulatory, and compliance requirements, solidifying Kafka-based systems as a cornerstone of modern data ecosystems.

Appendix 1: docker-compose.yml:

The Confluent Platform ecosystem used in the above examples. Cut and paste the following section into a file named docker-compose.yml. This is aYAML file, make sure all the columns maintain their spacing.

```
---
version: "2"
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.7.1
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  broker:
    image: confluentinc/cp-server:7.7.1
```

hostname: broker
container_name: broker
depends_on:
- *zookeeper*
ports:
- *"9092:9092"*
- *"9101:9101"*
environment:
KAFKA_BROKER_ID: 1
KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:29092,PLAINTEXT_HOST://localhost:9092
KAFKA_METRIC_REPORTERS: io.confluent.metrics.reporter.ConfluentMetricsReporter
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
KAFKA_CONFLUENT_LICENSE_TOPIC_REPLICATION_FACTOR: 1
KAFKA_CONFLUENT_BALANCER_TOPIC_REPLICATION_FACTOR: 1
KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
KAFKA_JMX_PORT: 9101
KAFKA_JMX_HOSTNAME: localhost
KAFKA_CONFLUENT_SCHEMA_REGISTRY_URL: http://schema-registry:8081
CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS: broker:29092
CONFLUENT_METRICS_REPORTER_TOPIC_REPLICAS: 1
CONFLUENT_METRICS_ENABLE: "true"
CONFLUENT_SUPPORT_CUSTOMER_ID: "anonymous"

schema-registry:
image: confluentinc/cp-schema-registry:7.7.1
hostname: schema-registry
container_name: schema-registry
depends_on:
- *broker*
ports:
- *"8081:8081"*
environment:
SCHEMA_REGISTRY_HOST_NAME: schema-registry
SCHEMA_REGISTRY_RESOURCE_EXTENSION_CLASS: io.confluent.kafka.schemaregistry.rulehandler.RuleSetResourceExtension
SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS: "broker:29092"
SCHEMA_REGISTRY_LISTENERS: http://0.0.0.0:8081

connect:
image: cnfldemos/cp-server-connect-datagen:0.6.4-7.6.0
hostname: connect
container_name: connect
depends_on:
- *broker*
- *schema-registry*
ports:
- *"8083:8083"*
environment:
CONNECT_BOOTSTRAP_SERVERS: "broker:29092"
CONNECT_REST_ADVERTISED_HOST_NAME: connect
CONNECT_GROUP_ID: compose-connect-group
CONNECT_CONFIG_STORAGE_TOPIC: docker-connect-configs
CONNECT_CONFIG_STORAGE_REPLICATION_FACTOR: 1
CONNECT_OFFSET_FLUSH_INTERVAL_MS: 10000
CONNECT_OFFSET_STORAGE_TOPIC: docker-connect-offsets
CONNECT_OFFSET_STORAGE_REPLICATION_FACTOR: 1

```
CONNECT_STATUS_STORAGE_TOPIC: docker-connect-status
CONNECT_STATUS_STORAGE_REPLICATION_FACTOR: 1
CONNECT_KEY_CONVERTER: org.apache.kafka.connect.storage.StringConverter
CONNECT_VALUE_CONVERTER: io.confluent.connect.avro.AvroConverter
CONNECT_VALUE_CONVERTER_SCHEMA_REGISTRY_URL: http://schema-registry:8081
# CLASSPATH required due to CC-2422
CLASSPATH: /usr/share/java/monitoring-interceptors/monitoring-interceptors-7.7.1.jar
CONNECT_PRODUCER_INTERCEPTOR_CLASSES: "io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor"
CONNECT_CONSUMER_INTERCEPTOR_CLASSES: "io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor"
CONNECT_PLUGIN_PATH: "/usr/share/java:/usr/share/confluent-hub-components"
CONNECT_LOG4J_LOGGERS: org.apache.zookeeper=ERROR,org.I0Itec.zkclient=ERROR,org.reflections=ERROR
```

control-center:

```
image: confluentinc/cp-enterprise-control-center:7.7.1
hostname: control-center
container_name: control-center
depends_on:
  - broker
  - schema-registry
  - connect
  - ksqldb-server
ports:
  - "9021:9021"
environment:
  CONTROL_CENTER_BOOTSTRAP_SERVERS: "broker:29092"
  CONTROL_CENTER_CONNECT_CONNECT-DEFAULT_CLUSTER: "connect:8083"
  CONTROL_CENTER_KSQL_KSQLDB1_URL: "http://ksqldb-server:8088"
  CONTROL_CENTER_KSQL_KSQLDB1_ADVERTISED_URL: "http://localhost:8088"
  CONTROL_CENTER_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"
  CONTROL_CENTER_REPLICATION_FACTOR: 1
  CONTROL_CENTER_INTERNAL_TOPICS_PARTITIONS: 1
  CONTROL_CENTER_MONITORING_INTERCEPTOR_TOPIC_PARTITIONS: 1
  CONFLUENT_METRICS_TOPIC_REPLICATION: 1
  PORT: 9021
```

ksqldb-server:

```
image: confluentinc/cp-ksqldb-server:7.7.1
hostname: ksqldb-server
container_name: ksqldb-server
depends_on:
  - broker
  - connect
ports:
  - "8088:8088"
environment:
  KSQL_CONFIG_DIR: "/etc/ksql"
  KSQL_BOOTSTRAP_SERVERS: "broker:29092"
  KSQL_HOST_NAME: ksqldb-server
  KSQL_LISTENERS: "http://0.0.0.0:8088"
  KSQL_CACHE_MAX_BYTES_BUFFERING: 0
  KSQL_KSQL_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"
  KSQL_PRODUCER_INTERCEPTOR_CLASSES: "io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor"
  KSQL_CONSUMER_INTERCEPTOR_CLASSES: "io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor"
  KSQL_KSQL_CONNECT_URL: "http://connect:8083"
  KSQL_KSQL_LOGGING_PROCESSING_TOPIC_REPLICATION_FACTOR: 1
  KSQL_KSQL_LOGGING_PROCESSING_TOPIC_AUTO_CREATE: "true"
  KSQL_KSQL_LOGGING_PROCESSING_STREAM_AUTO_CREATE: "true"
```

ksqldb-cli:

image: confluentinc/cp-ksqldb-cli:7.7.1

container_name: ksqldb-cli

depends_on:

- broker
- connect
- ksqldb-server

entrypoint: /bin/sh

tty: true

ksql-datagen:

image: confluentinc/ksqldb-examples:7.7.1

hostname: ksql-datagen

container_name: ksql-datagen

depends_on:

- ksqldb-server
- broker
- schema-registry
- connect

*command: "bash -c "echo Waiting for Kafka to be ready... && *
*cub kafka-ready -b broker:29092 1 40 && *
*echo Waiting for Confluent Schema Registry to be ready... && *
*cub sr-ready schema-registry 8081 40 && *
*echo Waiting a few seconds for topic creation to finish... && *
*sleep 11 && *
tail -f /dev/null" "

environment:

KSQL_CONFIG_DIR: "/etc/ksql"

STREAMS_BOOTSTRAP_SERVERS: broker:29092

STREAMS_SCHEMA_REGISTRY_HOST: schema-registry

STREAMS_SCHEMA_REGISTRY_PORT: 8081

rest-proxy:

image: confluentinc/cp-kafka-rest:7.7.1

depends_on:

- broker
- schema-registry

ports:

- 8082:8082

hostname: rest-proxy

container_name: rest-proxy

environment:

KAFKA_REST_HOST_NAME: rest-proxy

KAFKA_REST_BOOTSTRAP_SERVERS: "broker:29092"

KAFKA_REST_LISTENERS: "http://0.0.0.0:8082"

KAFKA_REST_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"

Appendix 2: Example Avro Schema order-transaction.avsc:

The following is the AVRO schema used in our examples. Please copy and paste into a file called order – transaction.avsc in the EXAMPLE_HOME directory

```
{  
  "name": "transaction",
```

```
"namespace": "demo.data.contract.com",
"type": "record",
"fields": [
  {
    "name": "transactionId",
    "type": "string"
  },
  {
    "name": "productId",
    "type": "string"
  },
  {
    "name": "price",
    "type": "double"
  },
  {
    "name": "productDescription",
    "type": "string"
  },
  {
    "name": "timestamp",
    "type": "long"
  },
  {
    "name": "firstName",
    "type": "string"
  },
  {
    "name": "lastName",
    "type": "string"
  },
  {
    "name": "email",
    "type": "string"
  },
  {
    "name": "gender",
    "type": "string"
  },
  {
    "name": "age",
    "type": "int"
  },
  {
    "name": "address",
    "type": "string"
  },
  {
    "name": "city",
    "type": "string"
  },
  {
    "name": "state",
    "type": "string"
  },
  {
    "name": "zipCode",
    "type": "string"
  },
  {
```

```

    "name": "creditCardNumberType",
    "type": "string"
  },
  {
    "name": "creditCardNumber",
    "type": "string"
  }
]
}

```

Appendix 3: order-transaction-ruleSet-complex.json:

The following is the rule-set used in example 3. Please copy and paste into a file called order-transaction-ruleSet-complex.json in the EXAMPLE_HOME directory

```

{
  "ruleSet": {
    "domainRules": [
      {
        "name": "customers_under_age_of_18_not_supported",
        "kind": "CONDITION",
        "type": "CEL",
        "mode": "WRITE",
        "expr": "message.age >= 18",
        "params": {
          "dlq.topic": "order-transactions-dlq"
        }
      },
      {
        "name": "unsupported_credit_card_type",
        "kind": "CONDITION",
        "type": "CEL",
        "mode": "WRITE",
        "expr": "message.creditCardNumberType in [\"AMEX\", \"Visa\", \"Mastercard\"]",
        "params": {
          "dlq.topic": "order-transactions-dlq"
        }
      },
      {
        "name": "visa_card_number_is_not_16_digits_long",
        "kind": "CONDITION",
        "type": "CEL",
        "mode": "WRITE",
        "expr": "message.creditCardNumberType == \"Visa\" ? size(message.creditCardNumber) == 16:true",
        "params": {
          "dlq.topic": "order-transactions-dlq"
        }
      }
    ],
    "onFailure": "DLQ"
  }
}

```

```

    "name": "visa_card_number_first_digit_is_not_5",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType == \"Visa\" ? message.creditCardNumber.matches(\"^5[0-9]{15}$\"):true",
    "params": {
        "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
},
{
    "name": "mastercard_card_number_is_not_16_digits_long",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType == \"Mastercard\" ? size(message.creditCardNumber) == 16:true",
    "params": {
        "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
},
{
    "name": "mastercard_card_number_first_digit_is_not_2",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType == \"Mastercard\" ? message.creditCardNumber.matches(\"^2[0-9]{15}$\"):true",
    "params": {
        "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
},
{
    "name": "amex_card_number_is_not_15_digits_long",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType == \"AMEX\" ? size(message.creditCardNumber) == 15:true",
    "params": {
        "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
},
{
    "name": "amex_card_number_first_digit_is_not_3",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType == \"AMEX\" ? message.creditCardNumber.matches(\"^3[0-9]{14}$\"):true",
    "params": {
        "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
}
}
}
}
}

```

Appendix 4:

order-transaction-breaking-change.json:

The following is the Avro schema is used in example 1. Please copy and paste into a file called order-transaction-breaking-change.json in the EXAMPLE_HOME directory

```
{
  "name": "transaction",
  "namespace": "demo.data.contract.com",
  "type": "record",
  "fields": [
    {
      "name": "transactionId",
      "type": "string"
    },
    {
      "name": "productId",
      "type": "string"
    },
    {
      "name": "price",
      "type": "double"
    },
    {
      "name": "productDescription",
      "type": "string"
    },
    {
      "name": "timestamp",
      "type": "long"
    },
    {
      "name": "first_name",
      "type": "string"
    },
    {
      "name": "lastName",
      "type": "string"
    },
    {
      "name": "email",
      "type": "string"
    },
    {
      "name": "gender",
      "type": "string"
    },
    {
      "name": "age",
      "type": "int"
    },
    {
      "name": "address",
      "type": "string"
    },
  ]
}
```



```
{
  "name": "city",
  "type": "string"
},
{
  "name": "state",
  "type": "string"
},
{
  "name": "zipCode",
  "type": "string"
},
{
  "name": "creditCardNumberType",
  "type": "string"
},
{
  "name": "creditCardNumber",
  "type": "string"
}
]
```

Appendix 5:

order-transaction-ruleset-complex-mirgration.json:

The following is the rule-set used in example 4. Please copy and paste into a file called order-transaction-ruleset-complex-migration.json in the EXAMPLE_HOME directory

```
{
  {
    "name": "customers_under_age_of_18_not_supported",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.age >= 18",
    "params": {
      "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
  },
  {
    "name": "unsupported_credit_card_type",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType in [\"AMEX\", \"Visa\", \"Mastercard\"]",
    "params": {
      "dlq.topic": "order-transactions--dlq"
    },
    "onFailure": "DLQ"
  },
  {
```

```

    "name": "visa_card_number_is_not_16_digits_long",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType == \"Visa\" ? size(message.creditCardNumber) == 16:true",
    "params": {
        "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
},
{
    "name": "visa_card_number_first_digit_is_not_5",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType == \"Visa\" ? message.creditCardNumber.matches(\"^5[0-9]{15}$\"):true",
    "params": {
        "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
},
{
    "name": "mastercard_card_number_is_not_16_digits_long",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType == \"Mastercard\" ? size(message.creditCardNumber) == 16:true",
    "params": {
        "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
},
{
    "name": "mastercard_card_number_first_digit_is_not_2",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType == \"Mastercard\" ? message.creditCardNumber.matches(\"^2[0-9]{15}$\"):true",
    "params": {
        "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
},
{
    "name": "amex_card_number_is_not_15_digits_long",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",
    "expr": "message.creditCardNumberType == \"AMEX\" ? size(message.creditCardNumber) == 15:true",
    "params": {
        "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
},
{
    "name": "amex_card_number_first_digit_is_not_3",
    "kind": "CONDITION",
    "type": "CEL",
    "mode": "WRITE",

```

```

    "expr": "message.creditCardNumberType == \"AMEX\" ? message.creditCardNumber.matches(\"^3[0-9]{14}$\"):true",
    "params": {
      "dlq.topic": "order-transactions-dlq"
    },
    "onFailure": "DLQ"
  }
},
"migrationRules": [
  {
    "name": "changeFirstNameToFirst_name",
    "kind": "TRANSFORM",
    "type": "JSONATA",
    "mode": "UPGRADE",
    "expr": "$merge([$sift($,function($v,$k) {$k != 'firstName'}),{'first_name': $.firstName}])"
  },
  {
    "name": "changeFirst_nameToFirstName",
    "kind": "TRANSFORM",
    "type": "JSONATA",
    "mode": "DOWNGRADE",
    "expr": "$merge([$sift($,function($v,$k) {$k != 'first_name'}),{'firstName': $.first_name}])"
  }
]
}
}
}

```

About Data-Blitz:

Data-Blitz specializes in cloud computing, event processing, and data governance. Our team is highly skilled in cloud-based architectures, data in motion, data pipelines, and AI. We provide expert support in Data Engineering, AWS, Azure, and GCP cloud environments, Apache/Confluent Kafka, IoT Edge devices, NoSQL and SQL databases, and Lucene-based search engines like Elasticsearch and Solr. Our expertise also extends to AI, with a focus on PyTorch, TensorFlow, Generative AI, and prompt engineering using LangChain and vector databases. We have experienced architects, developers, and agile scrum leads who are rigorously vetted and only begin billing after demonstrating proficiency with the target technology. By maintaining low margins, we offer competitive pay to our consultants, helping us attract and retain top talent.

About the Author:

I'm **Paul Harvener**, a Principal Consultant at Data-Blitz. I have a background in Computational Mathematics, Applied Physics, and Computer Science. My career began in the defense sector, where I specialized in Operations Research, modeling tactical warfare, and simulating satellite communication systems using Discrete Event Processing, Dynamic Programming, Monte Carlo Simulation, and Markov Chains. I later transitioned to the unclassified world of the telecommunications industry, designing telephony management applications. Next, I moved to a truly transformational company called Borland. My tenure at Borland saw me developing distributed processing systems using CORBA, which ultimately inspired me to co-found Data-Blitz, where we specialize in High-Performance, Distributed, Real-Time, and Event-driven data processing systems. In my early days at Data-Blitz, I authored a product called Plumber. Plumber was a distributed compute engine that allowed organizations to deploy truly distributed application stacks across an arbitrary set of computers. It ran under MESO, Vagrant, and was later ported to Docker. Today, this concept is known as Serverless Computing. But in the end, there's nothing serverless about it. Plumber was sunsetted in 2019 when various public cloud vendors offered a much more flexible approach. You can reach me at **pharvener@data-blitz.com**.