FEB 21, 2024 READ TIME: 18 MIN

Data Products, Data Contracts, and Change Data Capture

Technology > Confluent

Adam Bellemare
Staff Technologist, Office of the CTO

Change data capture (CDC) has long been one of the most popular, reliable, and quickest ways to connect your database tables into data streams. It is a powerful pattern and one of the most common and easiest ways to bootstrap data into Apache Kafka®. But it comes with a relatively significant drawback—it exposes your database's internal data model to the downstream world.

Downstream consumers subscribing to the data are directly coupled to the source's internal data model, breaching the source's isolation. Propagating the source's data model downstream is a recipe for disaster, as it provides a directly coupled and brittle interface. Even simple internal model changes and refactorings can cause significant breaking changes to the downstream consumers, causing outages, system failures, and false reports. Perhaps worst of all, a *silent* breakage can cause untold damage over a long period, resulting in a loss of customer trust.

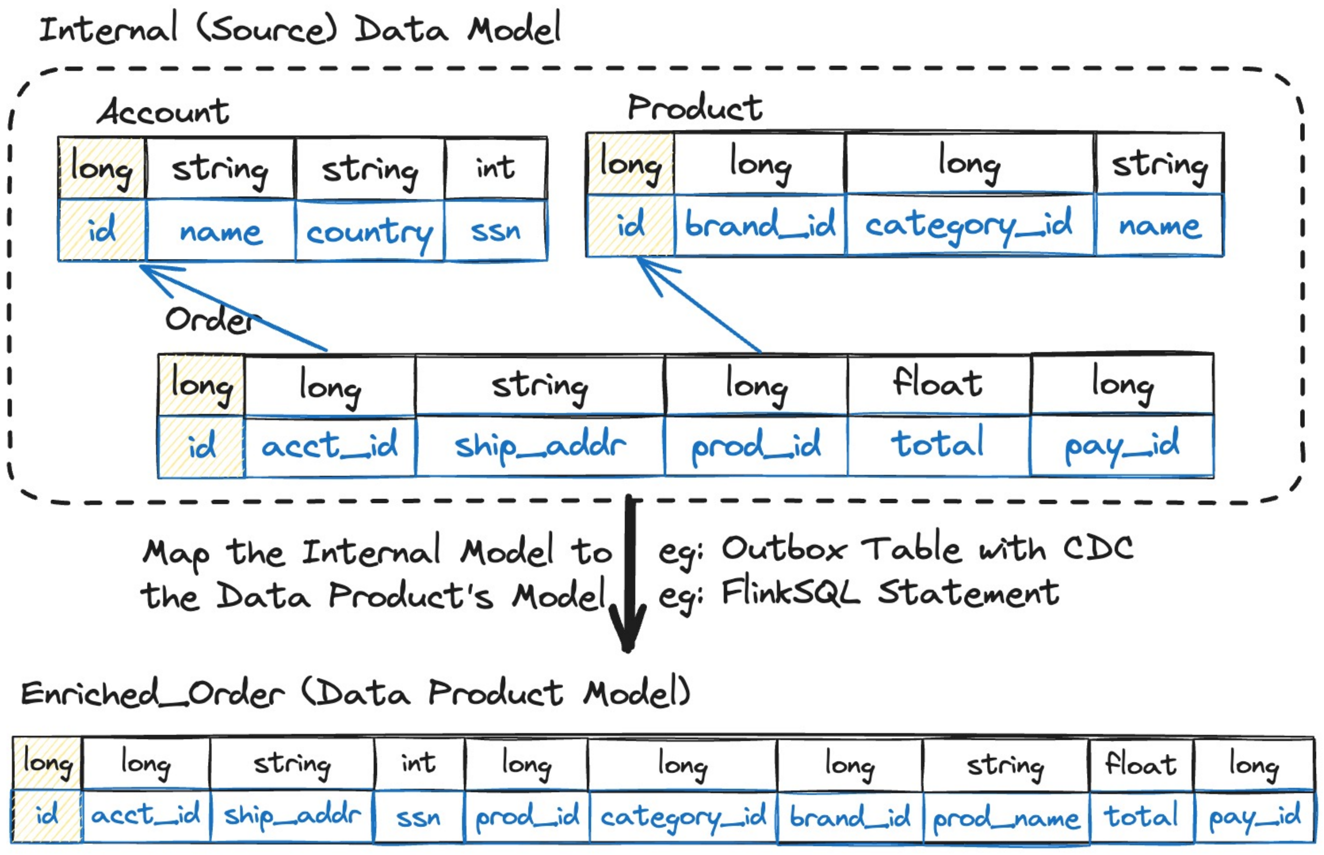
We need a better option.

First-class data products, enforced by **data contracts**, power the next logical evolution of the CDC pattern. Data products, particularly those providing data via an event stream, Confluent envisions *streaming* data products, and how you can implement them in Confluent Cloud. We'll then augment the story using Apache Flink® SQL to build data products using multiple disparate sources, along with organizing, monitoring, and governing your resultant data products. And finally, we'll look to the future of a unified stream/table data product and what that may mean for your business.

What are data products?

You may have heard about data products before, likely as part of a data mesh. A data product is a trustworthy data set purpose-built to share and reuse with other teams and services. It's a formalization of responsibilities, technology, and processes to make getting the data you and your services need easy.

The data product owner is responsible for maintaining the code and processes that generate the data product, particularly the mapping between the internal data model and the external data product model. This mapping provides the decoupling between the two models and ensures that the internal and external models can update (largely) independently of each other. In the following example (which we revisit throughout the post), the internal model is a normalized schema of **Account, Product, and Order**, while the external model is simply the denormalized **enriched_order**.

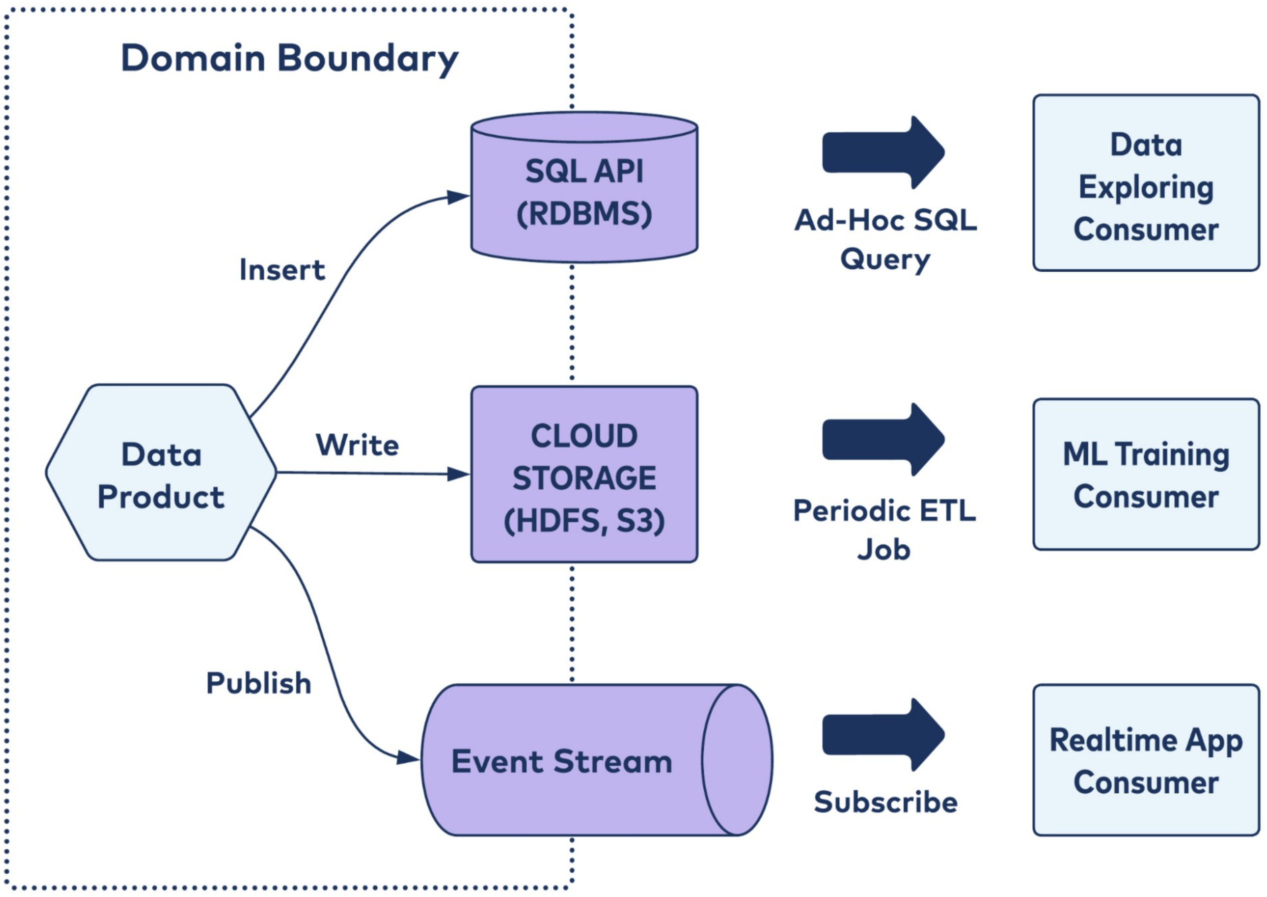

Internal (Source) Data Model
Account
long string string int
id name country ssn
Product
long long long string
id brand_id category_id name
Order
long long string long float long
id acct_id ship_addr prod_id total pay_id
Map the Internal Model to the Data Product's Model
eg: Outbox Table with CDC
eg: FlinkSQL Statement
Enriched_Order (Data Product Model)
long long string int long long long string float long
id acct_id ship_addr ssn prod_id category_id brand_id prod_name total pay_id

A data product also has a data contract. Major components include:

- **Schemas:** A formal declaration of the data's structure, types, fields, defaults, constraints, rules, evolutionary capabilities, and embedded documentation.
- **Metadata:** Additional information about the data product, including SLAs, business tags, classification tags, data access location, policies, and versioning information.
- **Dedicated Ownership:** A designated product owner is responsible for managing the changes and features of the data product. They work with its users and consumers to meet their needs, including negotiating change requests and providing support.

A data product is accessible via one of more **ports** (or modes). These are the endpoints that a data product user can directly connect to to access the underlying data. Common ports include:

- **Kafka topics**, containing a stream of near real-time events
- **Parquet tables**, containing a materialization of the stream or as a standalone periodically updated data set
- **Direct access to database tables** containing a selection of data available for querying
- **REST/gRPC API**, providing a selection of pre-fabricated data sets



A data product accessible by three different ports (Confluent's Practical Data Mesh e-Book). Each port contains exactly the same data organized according to the format.

Much like any other product, it should also be easy to use a data product. Important data product features include:

- **Self-Service:** Allows others to use the data confidently without out-of-band coordination. Make it easy to find, use, and share data products.
- **Interoperability:** The data product should be compatible with other related data products. For example, using common IDs for core business entities, like accountId, userId, and productId, makes it easy to correlate and join with other data products.
- **Security Compliance:** Components like field-level encryption, encryption at rest, and role-based access controls (RBACs).
- **Legal Compliance:** This includes requirements like GDPR, the right to be forgotten, and enforcing that data remains in specific jurisdictions.

There are several common patterns for creating a streaming data product, each of which has seen significant success with many of our customers. Remember that a data product's purpose is to provide the *actual data* as the product and, in our case, as a stream of reliable near-real-time events.

Streams offer a fast, low-latency means to easily connect your data to your business use cases. Composing a typical streaming data product consists of several main components:

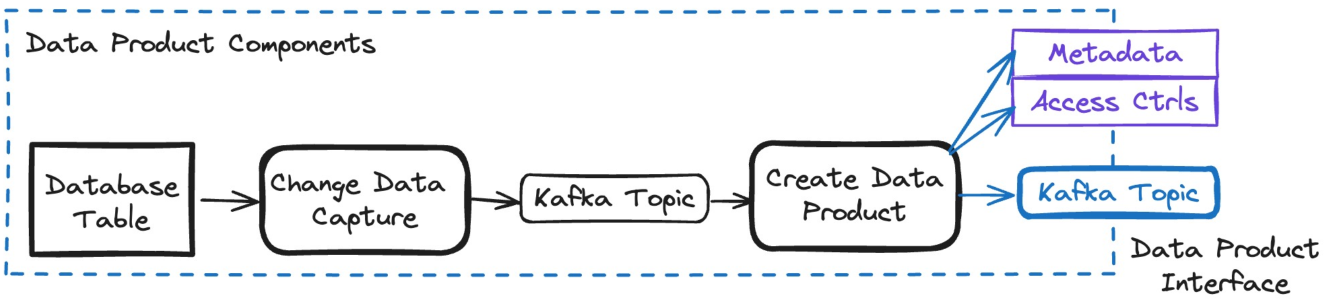
- The data sources, including database tables, Kafka topics, or files in S3
- The mapping to select and reformulate the source data into a model acceptable for external consumption (Note that this mapping isolates the internal and the external data model)
- The topic for the resultant stream of data
- The metadata, tags, schemas, access controls, descriptions, and documentation that support data product usage

WARNING:

Do not neglect the social responsibilities of publishing a data product! Requirements may include:

- Delivering on-call support, including getting out of bed in the middle of the night to meet SLAs
- Providing support and assistance for data usage
- Negotiating schema changes and updates

The following image shows a simple data product created from data stored in a database. The dotted line represents the boundary of the components making up the data product, including the processes, intermediate storage, and the final selection and publishing of the data and metadata.



You may have noticed that this data product looks suspiciously like many data streaming pipelines. You're right to notice that. A data product is simply the formalization of the most important parts of providing data for others. It includes a well-defined contract, reliable content, secured access, and metadata to help you decide if it's the right data for your use cases.

In the remainder of this blog post, we'll explore some of the most common options for building data products, including the outbox pattern, using connectors, using Flink SQL, the role of data governance, and what the future holds.

Building a data product via change data capture and outbox pattern

Change data capture is one of the most common and powerful ways to get data from an existing system into an event stream. CDC captures data from the source database, typically by tailing the underlying transactional log. (Check out our detailed blog post for more info.) As a result, you can access a stream of near-real-time updates to the internal domain model of the database tables, letting you power rich streaming applications and analytics.

However, the resultant streams are coupled directly to the source's internal data model. Any changes to the internal model can affect your data streams. Simple changes like adding a column to a table may not cause any problems, whereas combining and deleting tables may cause your CDC connector and consumers to come to a screeching, error-filled halt.

The outbox table pattern is a popular technique to isolate the internal data model. You declare a database table to act as an outbox, define the schema to match the desired event schema, and populate it whenever you update your internal model. Note that you must use atomic transactions to ensure that your internal data model remains consistent with the outbox. Finally, you use a CDC connector to emit the outbox events into an event stream.

What's great about the outbox is that you use the source database's query engine and data type checking to handle consistency. You can't write any data to the outbox if it doesn't match the schema, which provides you with a strong, well-defined schema for your data product right from the source.

You can also leverage the query engine for richer operations, such as denormalizing data from the source. Outbox tables work with *any database that supports transactions* and are a simple way to get started with well-defined **data contracts** without investing in yet another tool.

Keep in mind that the outbox pattern requires database processing and storage resources. Requirements will vary depending on transaction volume, data size, and query complexity (such as denormalizing or aggregating by time). Test the performance impact before you deploy to production to ensure your system still meets its SLAs and avoid unpleasant surprises.

Let's look at an example.

Say we have three tables inside a relational database: **Account**, **Product**, and **Order**. Each table has a primary `id` key (shaded in yellow). The Order table has two foreign-key references, one to Account and one to Product.

We want to enrich an **Order** event with **Account** and **Product** information and write it to the outbox whenever we create a new order. We'd modify our code by wrapping it into a transaction like so:

Copy

```
def insert_order(long acct_id,
                 String ship_addr,
                 long prod_id,
                 float total,
                 long pay_id){
    //Create the `order` object
    val order = Order(acct_id, ship_addr, prod_id, total, pay_id)

    //Transaction is atomic
    openTransaction {
          //Update the internal model
          writeToOrderTable(order)
          //Get the `account` and `product` for the outbox
          //Note that these lookups consume database resources
          val account = getAccount(acct_id)
          val product = getProduct(prod_id)
```

```
            //Denormalize and select fields to write to outbox
            //Note that the extra write consumes database resources
            writeToOutboxTable(order, account, product)
      }
      //Event is in the outbox, according to outbox Schema.
  }
```

Note that in this code block, we have decided to get the data from each of the Account and Product tables. The `writeToOutboxTable` function takes all three rows as parameters, then denormalizes, selects, and forms the data according to the outbox schema and then writes it to the outbox. Finally, set up a CDC connector to pull the data out of the outbox asynchronously and write it to Kafka.

Coming soon: Two-phase commit (2PC) support with KIP-939

Confluent and Kafka will soon offer more flexibility in sourcing data to data products from operational data stores. With KIP-939, the addition of a two-phase commit protocol to Kafka means data processing systems can safely write data to both operational databases and a data stream.

With the outbox table, we had to rely on a transaction to update both Order and Outbox, as Kafka doesn't currently support participation in 2PC. However, once KIP-939 is complete (target release pending), you'll be able to rewrite your code to simply write directly to Kafka in your transaction.

Copy

```
//Transaction is atomic
openTransaction {
      //Update the internal model
      writeToOrderTable(order)
      //Get the `account` and `product` for the outbox
      //Note that these lookups consume database resources
      val account = getAccount(acct_id)
      val product = getProduct(prod_id)
      //Denormalize and select fields for outbox, then write to Kafka!
      //
      //Note: This is the only change!
      //
      writeToKafka(order, account, product)

}
//Data in the order table and kafka topic are updated atomically!
```

Transactions ensure eventual data consistency in both your database and your streams. Post KIP-939, you'll no longer need a connector to get your data out of your database, though you'll still need to consider any performance trade-offs.

So far, we've looked at the scenario where all of your source data is within a single database, and you can leverage its query engine to denormalize and structure your data product. But let's look at another common scenario: where the data you want to use to build your data product is scattered across multiple sources. This is where Flink SQL can really help us out.

Building a data product using Flink SQL

Let's revise our outbox table example. Instead of a single database, the **Account**, **Product**, and **Order** data are each stored in different locations—one in a PostgreSQL database, one in a native event stream, and one in a document database (say MongoDB).

For the **Account** data in PostgreSQL, we'll use the PostgreSQL Debezium CDC connector, then simplify and flatten the data (within the connector). Similarly, for the **Product** schema in MongoDB, we'll use the MongoDB Debezium CDC connector and emit it to a topic. And for **Order,** we won't need to do anything as it's already in a topic.

Each topic contains event-carried state based on its primary key (I have previously called these events "facts"). What's great about this is that we can materialize the **Account**, **Product**, and **Order** topics into their own Flink tables and then join them together whenever a new event arrives.

The Flink SQL code looks like this:

Copy

```
CREATE TABLE enriched_orders AS
SELECT o.id as order_id,
      a.id as account_id,
      a.country,
      o.ship_addr,
      o.total,
      o.pay_id,
      p.brand_id,
      p.cat_id,
      p.name
FROM orders AS o
LEFT JOIN accounts FOR SYSTEM TIME AS OF `o.$rowtime`AS a
ON a.id = o.account_id
LEFT JOIN products FOR SYSTEM TIME AS OF `o.$rowtime`AS p
ON p.id = o.product_id
```

Assumptions:

- Each record is partitioned according to its primary key
- Rowtime represents the Kafka record timestamp

The resulting **enriched_orders** topic contains denormalized events from three separate sources. When the upstream system creates, updates, or deletes an **order**, the Flink SQL logic will create a new **enriched_order** event. We can also compact the **enriched_orders** topic if we would like to reduce the amount of data stored.

Consumers can read the **enriched_order** data from the beginning of time, from a given offset or timestamp, or simply jump to the latest data.

In this example, the data product components:

1. The CDC connectors
2. The intermediate Kafka topics
3. The Flink SQL table and stream declarations
4. The Flink SQL join job
5. The **enriched_orders** topic as the output port

One helpful technique in organizing the components that make up your data products is to tag them with organizational metadata. For instance, you can tag the connectors, topics, schemas, and Flink SQL jobs (coming soon!) with a unique ID (such as the data product name). This can help you find all the components associated with your data product and help you organize and manage your dependencies.

What is the best method for building a data product?

So far, we’ve covered the outbox pattern, the Flink SQL pattern, and introduced KIP-939 as an upcoming option. But when would you choose one over the other?

The outbox pattern is a good choice when:

- The data required for the data product is entirely within the database
- You require a lot of denormalization

- You don't want to run another downstream process to compile the data product

The outbox pattern has some drawbacks that you will need to consider:

- It imposes extra processing loads on the database, sometimes excessively so
- It must periodically clean out the outbox table to avoid unbounded growth
- It requires deep integration with the application source code
- It cannot join data outside of the database
- There is no replayability if you built the data product incorrectly

In general, the outbox is a good pattern for emitting denormalized data from a singular database, provided the performance and software modification impact is minimal.

The Flink SQL pattern is a good choice when:

- You want to isolate database performance from data product composition resources
- Your data product requires data from multiple sources
- Your database administrator will not let you build an outbox, due to privacy, security, or performance concerns
- You can rewind the Flink SQL job to reprocess Kafka topics, letting you rebuild a data product in the case of an error or a poorly defined schema
- You want a more granular view into how your data products are composed, via stream lineage

The main drawback to the Flink SQL pattern is that the query lifecycle is independent of the source database system. You'll need to validate database schema changes against the Flink SQL queries. Singular ownership of the data product allows the data product owner to orchestrate these changes.

KIP-939 remains a promising future option that will give us more flexibility. In many use cases, we'll be able to remove connectors and outbox tables entirely, publishing directly to the stream and using Flink SQL to construct the data product. Alternatively, we can leverage the source database's engine to denormalize and construct the data product in code, writing it to the data product's topic directly in the transaction. We'll look into this subject more once KIP-939 is accepted and moving into beta release.

Regardless of which pattern you use, we're still only half done. Let's take a look at how you promote your well-formed data into a data product.

Creating a data product in Confluent

A data product is created with *intention*. It is not just another data set in a catalog populated by an automated bot. The data product owner is responsible for defining, maintaining, and supporting the data product throughout its lifecycle.

Data product consumers couple their queries and applications to this schema, which forms part of the data contract. The data product owner must appropriately model and denormalize the data, hide internal fields, and consider privacy and security requirements.

A data product must include metadata, ownership and help information, usage details, and documentation. **Tags** provide an additional mechanism for organizing and curating data products, acting as hooks for discovery, search, automation, and security.

But how do we correlate the metadata, tagging, schemas, streams, and data? Let's take a look at the Confluent Data Portal.

Data products? Meet Data Portal.

Our customers have told us that they love the idea of streaming data products—it's been one of the most discussed subjects with our most forward-thinking customers. In fact, many of them have converged on a similar idea but lacked the tooling to implement it on their own. We listened to them and worked with them to develop our Confluent Data Portal, which we announced last year at Current 2023, and went to General Availability on December 4, 2023.

Data Portal combines several essential components to make building, sharing, discovering, and using streaming data products easy. Major components underpinning it include:

- **Stream Catalog** – Increase collaboration and productivity with self-service data discovery that allows teams to classify, organize, and find the needed data streams.
- **Stream Quality** – Deliver trusted, high-quality data streams to the business and maintain data integrity as services evolve.
- **Stream Lineage** – Understand complex data relationships and uncover more insights with interactive, end-to-end maps of data streams.

Data Portal combines these components into an easy-to-use portal for your data product producers, consumers, and managers (find more details in our general availability announcement blog post). Users can discover, search, inspect, manage, and access data products across your organization. Remember that only those streams you deliberately promote to a data product will show up in the portal, significantly reducing clutter and easing your users' data discovery.

Data Portal showing a purchase_orders data product

Building and declaring a data product is an accomplishment in and of itself. It helps teams all across your company access streaming data for both operational and analytical use cases.

And while it would be nice if we could declare our data products once and never have to change them, the reality is that the one big constant in life is *change*. Data changes over time, be it due to technological changes, business requirements, or organizational structures. The only real constant is change.

So, how can we manage these changing needs and sources? How do we minimize disruption to our data product users without unduly burdening the creator? Let's look at how data contracts provide us with various options for evolving and changing our data products.

Evolving schemas and handling changes

Schemas play an outsized critical role in data streaming. They define the data format, including names, types, fields, defaults, and documentation. The most popular streaming formats include Avro®, Protobuf, and JSON Schema.

The producer writes the data according to the schema, while the consumer reads and uses the data according to the schema. If the producer tries to write data that doesn't adhere to the schema, the Kafka client throws an exception and forces the producer to try again using the correct format. Meanwhile, the consumer is isolated from any data that does not adhere to the schema. It is a very elegant separation of concerns. The consumer doesn't need to guess or interpret the format and meaning of the underlying data, while the producer has a precise and well-defined definition for the data it must produce.

But while schemas provide a strong data definition, the form of the data can change over time. Let's briefly take a look at the two main types of changes:

- **Compatible changes:** Some changes to your data product schema may be forward, backward, or fully compatible. These are relatively easy evolutions, such as adding a field with a default value (check out more information in this course, including a hands-on guide to schemas). Consumers should not be forced to update their code.
- **Breaking changes:** Other changes are not evolutionarily compatible and can incur much work (Avro, Protobuf, and JSON Schema vary in what they consider breaking changes). These changes typically include significantly restructuring the event's schema and requiring consumers to refactor their code.

While compatible changes are relatively easy to handle (again, see docs or video), breaking changes can pose more difficulty. This is where our data contract capabilities can reduce friction, taking incompatible changes and smoothing them over for your consumers.

Let's take a look at an example: Our previous account/product/order data model was pretty good, but we failed to account for non-U.S. citizens in our definition of the account user's identity. Other countries have identifications that are not Social Security numbers (SSN), which doesn't match our schema declaration (Remember, we're specifying the ISO-3166 country codes in `country`).

A user account with a U.S.-specific `ssn`

If we decide to change the event schema (and the upstream outbox definition), we'd have to update our schema from Version 0 to Version 1 (note the changes in green Version 0 to red Version 1).

A breaking schema change due to renaming `ssn` to `national_id`.

Data contracts provide a framework for rules and enforcement to reduce the complexity of migrating data during a breaking schema change. We call this feature **complex data migration.** Complex data migration rules introduce major versioning for schemas that allow breaking compatibility without having to do messy data migrations between topics —the only other option available until now.

Before data contracts, for complex schema evolution, you would often have to duplicate your data by writing records both in the old format to the existing topic and in the new format to a new topic, doubling your storage and network costs. Additionally, your consumers would need to manually switch over from the old topic to the new topic at a specific correlation point, lest they miss reading some data. This process is messy, risky, painful, and expensive.

With complex data migration, we can write **rules** that tell existing consumer clients how to manage new major versions of schemas. Additionally, these rules also specify to **new** clients how to read **old** data that may be in your stream, such as compacted topics storing a complete current state. Consumers will simply upgrade or downgrade the data they need for their clients. Let's look at an example.

Here's a migration rule to convert a field name from `ssn` to `national_id`, a typically incompatible change:

```
{
    "schema": "...",
    "ruleSet": {
      "migrationRules": [
        {
          "name": "changeSsnToNationalId",
          "kind": "TRANSFORM",
          "type": "JSONATA",
          "mode": "UPGRADE",
          "expr": "$merge([$sift($, function($v, $k) {$k != 'ssn'}), {'national_id': $.'ssn'
        },
          {
          "name": "changeNationalIdToSsn",
          "kind": "TRANSFORM",
          "type": "JSONATA",
          "mode": "DOWNGRADE",
          "expr": "$merge([$sift($, function($v, $k) {$k != 'national_id'}), {'ssn': $.'nati
          }
      ]
    }
  }
```

The upgrade rule allows new consumers to read old messages, while the downgrade rule allows old consumers to read new messages. The consumer clients use these rules to convert incompatible events into a suitable format for their use cases.

The following figure shows two consumers, each written against the v1 or the v2 schema. The consumer written against `v1` of the schema `DOWNGRADES` the newer `v2` events to `v1`, which its business logic can understand and process. Meanwhile, the consumer written against `v2` `UPDRADES` the historical `v1` events to `v2`.

More complicated breaking changes may require rewriting your data product entirely, such as a major upstream database refactoring, a significant shift in business objectives, or the revision of a beta data product that simply wasn't working out. Breaking changes are much like any other breaking API change, including those for a REST API (e.g., **/v1/** and **/v2/** APIs**)**. Support both for a fixed period, deprecate the old one, and help migrate your consumers onto the new one. Fortunately, these intensive breaking changes are far less common than evolutionary-compatible changes.

Monitoring and debugging data flows

Next up, monitoring your data product and figuring out if it's working as expected. Confluent provides connector monitoring capabilities so you can keep an eye on your database-sourced data. Similarly, you can monitor your Flink SQL statements, though this is an area that we will continue to improve on as Flink continues to mature.

You can access a curated set of metrics through Confluent's metrics API. One metric to look at is the Consumer Group Lag for your data products to ensure that your data product consumers remain healthy and able to keep up with the rate of data. You can integrate the metrics with your existing observability platforms, including New Relic, Datadog, Grafana Cloud, and Dynatrace.

Stream Lineage is one of the top ways to monitor your data product dependencies upstream and down. You can access a point-in-time lineage view to visualize precisely what your dependencies were doing at a given moment.

Data Portal provides the lineage of each of your streaming data products right out of the box. The following screenshot shows the UI for accessing the stream lineage for **purchase_orders**.

purchase_orders' *stream lineage available through the Data Portal*

Clicking on **Stream Lineage** brings us to the actual lineage of the data product, including upstream sources and downstream users.

Audit logging is another key feature for tracking the granting of access to data products, providing another layer of security and observability.

But what about digging deeper? What if we want to look inside the topics to see why we're getting unexpected results? You have two main options:

- Message browser lets you explore messages directly, which can be very helpful when you receive an error of a malformed message at a specific partition and offset.
- Flink SQL or ksqlDB provides you with full ad hoc data exploration capabilities. Write SQL queries to explore your data, find outliers, and test processing.

These tools let you debug your data products with ease. You can validate your input topics, inspect intermediate topics, and validate that the proper output events are being generated.

But what about tables?

While it's fairly evident that we believe strongly in streaming data products, we know that there are also many use cases for data at rest in the form of tables. Kafka users often sink events into cloud storage, such as S3, GCS, or Azure Blob Storage, repurposing it for analytical use cases and periodic batch-based processing. And while people have been happy enough to sink data to Parquet format for quite some time, we've seen significant interest in converting streams into Apache Iceberg and Apache Hudi formats.

The biggest benefit of a stream-first approach to data products is that you effectively get the table representation for free (barring the cost of sinking it into cloud storage). How? You've already done all the challenging work of providing a well-defined and contextualized data product with a contract, schema, evolution rules, metadata, tagging, ownership, documentation, and support channels.

You've executed a shift left in your data value by creating your data products as near-real-time streams, as close to the source as possible. You've made streams that can power any form of low or high-latency use cases, operational or analytical. There is no need to reinvent the wheel for tables.

Do you want the data in Iceberg? Use the Tabular Iceberg connector (hub) as a custom managed connector in Confluent Cloud to generate an Iceberg representation of the same streaming data product. How about in Hudi? Use the Hudi connector in precisely the same way (binary).

The fact of the matter is that ***it is far easier to take data in motion and slow it down into data at rest than it is to take data at rest and spur it into motion.*** Shifting left to a stream-first approach makes it easier, cheaper, and safer to make important time-sensitive decisions. Shifting left provides you with the capabilities to take your well-defined streaming data products and convert them into other formats, be they columnar, graph, document, or relational.

If you're still a bit skeptical about stream-first data products, don't worry. We hear you. But stay tuned, as we are hard at work reimagining streams and tables to provide you with all the advantages of both models.

In summary

Stream-based data products provide a powerful innovation in the data space. By building data products as close to the source as possible (a shift left), you unlock both near-real-time and batch-based use cases all across your business. Your data product provides a single source of well-defined, high-quality, interoperable, and supported data that your systems and peers can freely consume, transform, and remodel as needed. Data products form the bedrock of a healthy modern data ecosystem.

Data contracts do a lot of heavy lifting. They provide the formal agreement of the form and function of the data product and its API to all of its users. They act as a barrier between the internal and external data models, providing the data producer users with a stable yet evolvable API and a well-defined boundary into other business domains. Users can browse through contracts and find the data product of your choice in the **Data Portal**, or register their own data products for others to use.

CDC and **Flink** form a powerful new pairing in realizing data products. Isolate your internal domain models, join with data from across your company, and eliminate costly, brittle, point-to-point pipelines. With fully managed connectors, stream processing, and governance, we can help you focus on actually using your data instead of struggling just to make it usable.

We are committed to making it as easy as possible for you to stream the data you need, as you need it, where you need it. To learn more, check out our documentation for Data Governance, Flink SQL, and Custom Managed Connectors. Or, if you want to get better acquainted with data products (in the context of data mesh), check out our free e-book. If you're ready to try out Confluent:

- Sign up for Confluent Cloud and get a 30-day trial with $400 of free credit.
- Download Confluent Platform and get a 30-day free trial on unlimited brokers or always on a single broker.

Related resources

- Video: How To Improve Data Quality with Domain Validation Rules
- Video: How to Evolve Your Schemas with Migration Rules

Adam Bellemare is a staff technologist at Confluent, and formerly a data platform engineer at Shopify, Flipp, and BlackBerry. He has worked in the data space for over a decade, with a successful history in event-driven microservices, distributed data architectures, and integrating streaming data across organizations. He is also the author of the O'Reilly titles "Building Event-Driven Microservices" and "Building an Event-Driven Data Mesh."

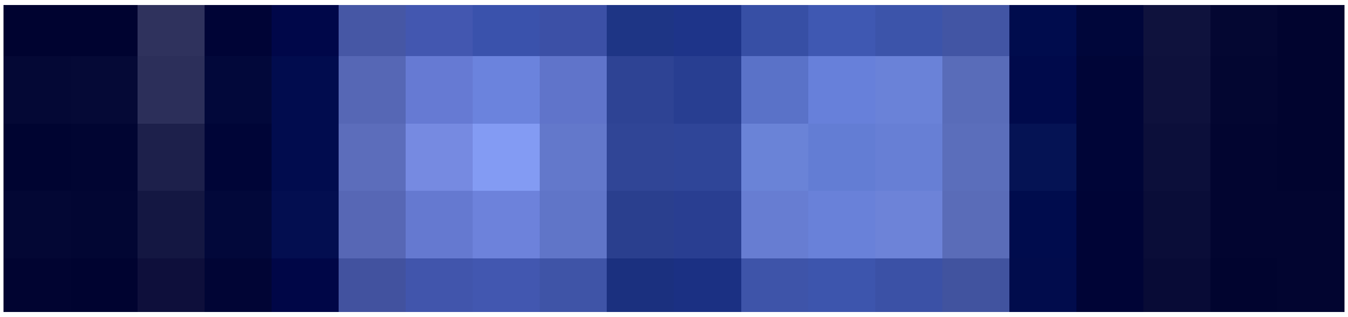
Introducing Data Portal in Stream Governance

Learn about the self-service interface for discovering, exploring, and accessing Kafka topics on Confluent Cloud.

Read The Blog Post >

Did you like this blog post? Share it now

Technology < Confluent

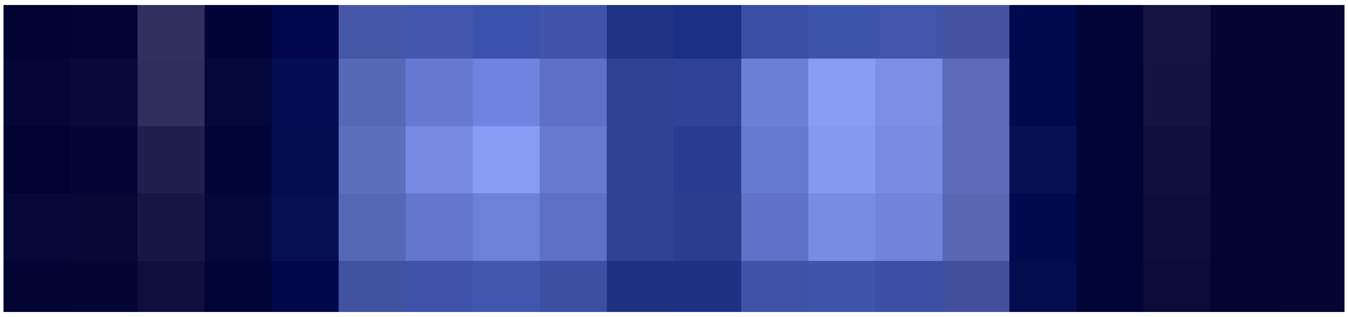


Introducing Data Portal in Stream Governance

DEC 4, 2023

Today, we’re excited to announce the general availability of Data Portal on Confluent Cloud. Data Portal is built on top of Stream Governance, the industry’s only fully managed data governance suite for Apache Kafka® and data streaming.

OLIVIA GREENE
DAVID ARAUJO



New with Confluent Platform: Seamless Migration Off ZooKeeper, Arm64 Support, and More

FEB 14, 2024

Over the last two years, we have periodically announced Confluent Platform releases, each building on top of the innovative feature set from the previous release. Today, we will talk about some of these core features that make hybrid and on-premises data...

ROHIT BAKHSHI

NITIN MUDDANA