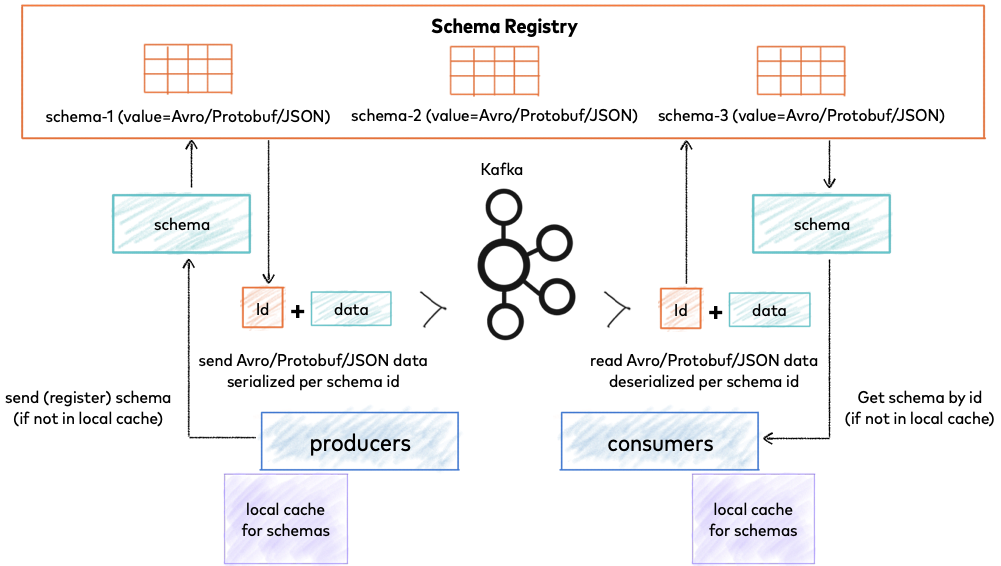
Confluent Schema Registry provides a serving layer for your metadata. It provides a RESTful interface for storing and retrieving your [Avro®](https://avro.apache.org/docs/current/spec.html), [JSON Schema](https://json-schema.org/), and [Protobuf](https://developers.google.com/protocol-buffers/) [schemas](https://docs.confluent.io/platform/current/schema-registry/schema_registry_onprem_tutorial.html#schema-registry-tutorial-definition). It stores a versioned history of all schemas based on a specified [subject name strategy](https://docs.confluent.io/platform/current/schema-registry/serdes-develop/index.html#sr-schemas-subject-name-strategy), provides multiple [compatibility settings](https://docs.confluent.io/platform/current/schema-registry/avro.html#schema-evolution-and-compatibility) and allows evolution of schemas according to the configured compatibility settings and expanded support for these schema types. It provides serializers that plug into Apache Kafka® clients that handle schema storage and retrieval for Kafka messages that are sent in any of the supported formats.

Schema Registry lives outside of and separately from your Kafka brokers. Your producers and consumers still talk to Kafka to publish and read data (messages) to topics. Concurrently, they can also talk to Schema Registry to send and retrieve schemas that describe the data models for the messages.



*Confluent Schema Registry for storing and retrieving schemas*

Schema Registry is a distributed storage layer for schemas which uses Kafka as its underlying storage mechanism. Some key design decisions:

* Assigns globally unique ID to each registered schema. Allocated IDs are guaranteed to be monotonically increasing and unique, but not necessarily consecutive.
* Kafka provides the durable backend, and functions as a write-ahead changelog for the state of Schema Registry and the schemas it contains.
* Schema Registry is designed to be distributed, with single-primary architecture, and ZooKeeper/Kafka coordinates primary election (based on the configuration).

# Why Schema Registry?

Kafka, at its core, only transfers data in byte format. There is no data verification that’s being done at the Kafka cluster level. In fact, Kafka doesn’t even know what kind of data it is sending or receiving; whether it is a string or integer.

Diagram

Description automatically generated

Producer sending data in byte format to Kafka Cluster and being consumed by a consumer.

Due to the decoupled nature of Kafka, producers and consumers do not communicate with each other directly, but rather information transfer happens via Kafka topic. At the same time, the consumer still needs to know the type of data the producer is sending in order to deserialize it. Imagine if the producer starts sending bad data to Kafka or if the data type of your data gets changed. Your downstream consumers will start breaking. We need a way to have a common data type that must be agreed upon.

That’s where [Schema Registry](https://docs.confluent.io/current/schema-registry/index.html) comes into the picture. It is an application that resides outside of your Kafka cluster and handles the distribution of schemas to the producer and consumer by storing a copy of schema in its local cache.

Diagram

Description automatically generated

Schema Registry Architecture

With the schema registry in place, the producer, before sending the data to Kafka, talks to the schema registry first and checks if the schema is available. If it doesn’t find the schema then it registers and caches it in the schema registry. Once the producer gets the schema, it will serialize the data with the schema and send it to Kafka in binary format prepended with a unique schema ID. When the consumer processes this message, it will communicate with the schema registry using the schema ID it got from the producer and deserialize it using the same schema. If there is a schema mismatch, the schema registry will throw an error letting the producer know that it’s breaking the schema agreement.

# Data Serialization Formats

Now that we know how schema registry works, what kind of data serialization format are we using with the schema registry? There are a few important points that we should consider when choosing the right data serialization format:

* If the serialization format is binary.
* If we can use schemas to enforce strict data structures.

Following are some [data serialization formats](https://en.wikipedia.org/wiki/Comparison_of_data-serialization_formats)as per the above considerations:

A picture containing chart

Description automatically generated

Comparison of various data serialization formats

# AVRO is the winner

[Avro](https://avro.apache.org/) is an open-source binary data serialization format that comes from the Hadoop world and has many use cases. It offers rich data structures and offers code generation on statically typed programming languages like C# and Java.

* Avro has support for primitive ( int, boolean, string , float etc.) and complex ( enums,arrays, maps,unions etc.) types.
* Avro schemas are defined using JSON.
* It is very fast.
* Avro embedded documentation prevents us from the guessing game of what each field means.
* We can have a default value for fields which is very useful when we evolve our schemas.

Let’s look at a sample Avro schema file:

Sample AVRO schema

You can see how easy it is to write an Avro schema. It is also a standard data serialization format for the [Confluent Schema Registry](https://www.confluent.io/blog/avro-kafka-data/).

# Schema Evolution

With time, our AVRO schemas will evolve. We will add new fields or update existing fields. With evolving schemas, our downstream consumers should be able to consume messages seamlessly without sending a production alert at 3 AM. Schema Registry is specifically built for data evolution by versioning each schema change.

Diagram

Description automatically generated

Schema Evolution

When a schema is first created, it gets a unique schema ID and a version number. With time, our schemas will evolve, we add new changes and if changes are compatible, we get a new schema ID and our version number increments. There are two ways to tell if a schema is compatible: by using a [maven plugin](https://docs.confluent.io/current/schema-registry/develop/maven-plugin.html#schema-registry-test-compatibility) (in Java) or by simply making a [REST](https://docs.confluent.io/current/schema-registry/using.html#sr-test-compat-latest) call. This compatibility check compares schema on the local machine with the schema on the schema registry.

There are various [patterns](https://docs.confluent.io/current/schema-registry/avro.html#compatibility-types) for schema evolution:

Forward Compatibility: update producer to V2 version of the schema and gradually update consumers to the V2 version.

Diagram

Description automatically generated

Forward Compatibility

Backward Compatibility: update all consumers to the V2 version of the schema and then update producer to V2 version.

Diagram

Description automatically generated

Backward Compatibility

Full Compatibility: when schemas are both forward and backward compatible.

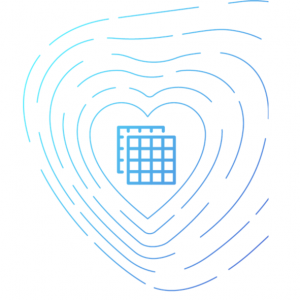
# Conclusion

Schema Registry is a simple concept but it’s really powerful in enforcing data governance within your Kafka architecture. Schemas reside outside of your Kafka cluster, only the schema ID resides in your Kafka, hence making schema registry a critical component of your infrastructure. If the schema registry is not available, it will break producers and consumers. So it is always a best practice to ensure your schema registry is highly available.

[Part 1](https://www.confluent.io/blog/schemas-contracts-compatibility) of this blog series by Gwen Shapira explained the benefits of schemas, contracts between services, and compatibility checking for schema evolution. In particular, using Confluent Schema Registry makes this really easy for developers to use schemas, and it is designed to be highly available. But it’s important to configure it properly from the start and manage it well, or else the schemas may not be available to the applications that need them.

There are three common types of operational mistakes we have observed customers make in self-managing their own Schema Registry:

* Bad designs
* Inconsistent configurations
* Operational faux pas



The impact of these mistakes include duplicate schema IDs, lost schemas, and inaccessible services. Here, we explore 17 pitfalls that operators make, because we want to make sure your Schema Registry is rock solid if you decide to self-manage this important component in your architecture. Actually, we recommend that you consider another alternative to self-managing Schema Registry, and the next blog post in this series reveals what that alternative is!

## **Bad designs**

|  |
| --- |
| **Mistake #1:** Co-locating Schema Registry instances on Apache Kafka® brokers |

The Schema Registry application itself requires about 1 GB for heap, but other than that, it does not need a lot of CPU, memory or disk. Given its relatively low footprint, operators may be tempted to co-locate Schema Registry with other services, like a Kafka broker. However, co-locating a Schema Registry instance on a host with any other application means that its uptime is entirely dependent on the co-located services behaving properly on the host. Therefore, to isolate failures, it is best practice to deploy Schema Registry on its own.

|  |
| --- |
| **Mistake #2:** Creating separate Schema Registry instances within a company |

Separate schema registries may not stay separated forever. Over time, organizations restructure, project scopes change, and an end system that was used by one application may now be used by multiple applications. If that happens and schema IDs are no longer globally unique, there may be collisions between schema IDs. For consistency in schema definitions and operational simplicity, deploy a single global Schema Registry cluster across an entire company, geographical areas, or clusters in a multi-datacenter design.

|  |
| --- |
| **Mistake #3:** Incorrectly setting up Schema Registry in multi-datacenter deployments |

In a multi-datacenter design, the same schemas and schema IDs must be available in both datacenters. Whether the design is active-active or active-passive, designate one Kafka cluster as the primary for Schema Registry. The primary cluster:

* Coordinates primary election among all the Schema Registry instances
* Contains the schemas topic, to which primary instances back up newly registered schemas

All Schema Registry instances need access to the primary-eligible instances to forward new schema registrations, and they need access to the designated primary cluster because they subscribe directly to the schemas topic. [Confluent Replicator](https://www.confluent.io/confluent-replicator/) then copies the Kafka schemas topic from the primary cluster to the other cluster for backup. For Schema Registry best practices in a multi-datacenter deployment, refer to the white paper [Disaster Recovery for Multi-Datacenter Apache Kafka Deployments](https://www.confluent.io/white-paper/disaster-recovery-for-multi-datacenter-apache-kafka-deployments/).

|  |
| --- |
| **Mistake #4:** Not deploying a virtual IP (VIP) in front of multiple Schema Registry instances |

Client applications can connect to any Schema Registry instance for producing or consuming messages. All instances have local caches mapping schemas to schema IDs so they can provide schema information to all consumers. For new schema registration, secondaries forward new schema registration requests to the primary.

If a Schema Registry instance IP address ever changes, the clients need to update their connection information. To ease that burden, it is easier to use VIPs for the addresses, so clients don’t have to update their connection information if IP addresses change.

## **Inconsistent configurations**

Multiple Schema Registry instances deployed across datacenters provide resiliency and high availability, thereby allowing any instance to communicate schemas and schema IDs to Kafka clients. However there needs to be some consistency between Schema Registry instances. If there isn’t, it is possible to end up with duplicate schema IDs, depending on the view of the current primary instance.

|  |
| --- |
| **Mistake #5:** Configuring different names for the schemas topic in different Schema Registry instances |

There is a commit log with all the schema information, which gets written to a Kafka topic. All Schema Registry instances should be configured to use the same schemas topic, whose name is set by the configuration parameter kafkastore.topic. This topic is the schema’s source of truth, and the primary instances read the schemas from this topic. The name of this topic defaults to \_schemas, but sometimes customers choose to rename it. This has to be the same for all Schema Registry instances, otherwise it may result in different schemas with the same ID.

|  |
| --- |
| **Mistake #6:** Mixing the election modes among the Schema Registry instances in the same cluster |

In a single-primary architecture, only the primary Schema Registry instance writes to that Kafka topic. There is an election process to coordinate primary election among the Schema Registry instances: One instance is elected primary, and the rest are secondary.

Since Confluent Platform 4.0, either the Kafka group protocol or ZooKeeper can coordinate the election—but not both. Mixing election modes could result in multiple primaries and duplicate schema IDs. We generally recommend using the Kafka group protocol, especially if connecting to Confluent Cloud or access to ZooKeeper is unavailable.

|  |
| --- |
| **Mistake #7:** Configuring different settings between Schema Registry instances that should be the same |

In addition to the election mode needing to be the same between the Schema Registry instances, there are a few other configuration parameters that must match in order to avoid unintended side effects. You can typically leave those other parameters as default (e.g., the group ID for the consumer used to read the Kafka store topic). If you override any default settings, they must be consistently overridden in all instances.

|  |
| --- |
| **Mistake #8:** Configuring the same host.name for all Schema Registry instances |

Nevertheless, there is one configuration parameter that must differ between Schema Registry instances, and that is host.name. This should be unique per instance to prevent potential problems and keep consistency on which instance is primary.

## **Operational faux pas**

|  |
| --- |
| **Mistake #9:** Bringing up Schema Registry without any security features enabled |

Securing Schema Registry is just as critical as securing your Kafka cluster, because the schema forms the contract for how different applications and organizations talk to each other through Kafka. Therefore, not restricting access to the Schema Registry might allow an unauthorized user to mess with the service in such a way that client applications can no longer be served schemas to deserialize their data.

Schema Registry has a REST API that allows any application to integrate with it to retrieve schemas or register new ones. Allow end user [REST API calls to Schema Registry over HTTPS](https://docs.confluent.io/current/schema-registry/docs/security.html#configuring-the-rest-api-for-http-or-https) instead of the default HTTP.

|  |
| --- |
| **Mistake #10:** Mis-configuring SSL keys, certificates, keystores, or truststores |

Configuring Schema Registry for SSL encryption and SSL or SASL authentication to the Kafka brokers is important for securing their communication. This requires working with the security team in your company to get the right keys and certificates and configuring the proper [keystores and truststores](https://docs.confluent.io/current/kafka/authentication_ssl.html#sr) to ensure that Schema Registry can securely communicate with the brokers. We have observed many customers spending time troubleshooting wrong keys or certificates, which slows down their ability to spin up new services.

|  |
| --- |
| **Mistake #11:** Manually creating the schemas topic with incorrect configuration settings |

The primary Schema Registry instance registers all new schemas and backs it up to a schemas topic in Kafka. This Kafka topic is the source of truth for all schema information and schema-to-schema ID mapping. By default, Schema Registry automatically creates this topic if it does not already exist, and it creates it with the right configuration settings: replication factor of three and retention policy set to compact (versus delete). But if you override the defaults and accidentally misconfigure it, you risk losing this topic, which is the source of truth for schemas.

|  |
| --- |
| **Mistake #12:** Deleting the schemas topic from the Kafka cluster |

Once the schemas topic is created, it is important to ensure that it is always available and never to delete it. If someone were to delete this topic, producers would not be able to produce data with new schemas, because Schema Registry would be unable to register new schemas. We hope it doesn’t happen to you, but I have to mention it because this has happened before. Really.

|  |
| --- |
| **Mistake #13:** Not backing up the schemas topic |

Should the schema topic be accidentally deleted, operators must be prepared to restore it. Therefore it is a best practice to backup the schemas topic on a regular basis. If you already have a multi-datacenter Kafka deployment, you can backup this topic to another Kafka cluster using [Confluent Replicator](https://www.confluent.io/confluent-replicator/). You can also use a [Kafka sink connector](https://www.confluent.io/hub/) to copy the topic data from Kafka to a separate storage (e.g., AWS S3). These will continuously update as the schema topic updates.

|  |
| --- |
| **Mistake #14:** Restarting Schema Registry instances before restoring the schemas topic |

[Restoring the schemas topic](https://docs.confluent.io/current/schema-registry/docs/deployment.html#backup-and-restore) requires a series of steps. Kafka operators often do not have control over pausing the client applications, which may try to register new schemas at random intervals, particularly if the configuration parameter auto.register.schemas is left at its default of true. Therefore, the schemas topic should be fully restored before Schema Registry instances are restarted, so that when Schema Registry does restart and read the schemas topic, it reads the schemas in order and schemas IDs maintain their proper sequence.

|  |
| --- |
| **Mistake #15:** Not monitoring Schema Registry |

Like any other component, you have to [monitor](https://docs.confluent.io/current/schema-registry/docs/monitoring.html) the Schema Registry instances to know that they are healthy and able to service clients. The last thing you want is for a Kafka developer to be the one to alert the operations team that the Schema Registry service is unreachable. Rather, the operations team should be the first to know through good monitoring practices!

|  |
| --- |
| **Mistake #16:** Upgrading Java on the host machine to a version that is not compatible with Schema Registry |

Schema Registry is a Java application. It has been validated on some of the more recent versions of Java, including Java 11, but not all versions like Java 9 or 10. During the lifecycle of self-managed Schema Registry, users may deploy the latest version of Java on their host machine by getting an image with the latest Java version preinstalled or upgrading Java on an existing image. However, if they do this without checking for Schema Registry compatibility, they may end up trying to run Schema Registry with an incompatible Java version that causes it to not even start. If you have questions about Java compatibility, see the [documentation](https://docs.confluent.io/current/installation/versions-interoperability.html#java).

|  |
| --- |
| **Mistake #17:** Poorly managing a Kafka cluster that causes broker problems |

The source of truth for schemas is stored in a Kafka topic, so the primary Schema Registry instance needs access to that Kafka topic to register new schemas. Schema Registry communicates with the Kafka cluster to write to the schemas topic, and any broker problems or cluster issues can negatively impact Schema Registry access to the schema’s topic. Therefore that cluster needs to be highly available to ensure that new schemas can be properly registered and written to that topic.