3 Designing a Kafka project

This chapters covers

- Designing a real-world Kafka project
- Determining which data format to use
- Existing issues impacting data usage
- Deciding when data transformation takes place
- How Kafka Connect helps us start a data-streaming path

In our previous chapter, we saw how we can work with Kafka from the command line and how to use a Java client. Now, we will expand on those first concepts and look at designing various solutions with Kafka. We will discuss some questions to consider as we lay out a strategy for the example project we'll start in this chapter. As we begin to develop our solutions, keep in mind that, like most projects, we might make minor changes along the way and are just looking for a place to jump in and start developing. After reading this chapter, you will be well on your way to solving real-world use cases while producing a design to facilitate your further exploration of Kafka in the rest of this book. Let's start on this exciting learning path!

3.1 Designing a Kafka project

Although new companies and projects can use Kafka as they get started, that is not the case for all Kafka adopters. For those of us who have been in enterprise environments or worked with legacy systems (and anything over five years old is probably considered legacy these days), in reality, starting from scratch is not a luxury we always have. However, one benefit of dealing with existing architectures is that it gives us a list of pain points, that we can address. The contrast also helps us to highlight the shift in thinking about the data in our work. In this chapter, we will work on a project for a company that is ready to shift from their current way of handling data and apply this new hammer named Kafka.

3.1.1 Taking over an existing data architecture

Let's look at some background to give us our fictional example inspired by Kafka's ever-growing usage. One topic by Confluent mentioned in chapter 1 (https://www.confluent.io/use-case/internet-of-things-iot/) and also an excellent article by Janakiram MSV, titled "Apache Kafka: The Cornerstone of an Internet-of-Things Data Platform," includes Kafka's use of sensors [1]. Using the topic of sensors as a use case, we will dig into a fictional example project.

Our new fictional consulting company has just won a contract to help re-architect a plant that works on e-bikes and manages them remotely. Sensors are placed throughout the bike that continuously provide events about the condition and status of the internal equipment they are monitoring. However, so many events are generated that the current system ignores most of the messages. We have been asked to help the site owners unlock the potential in that data for their various applications to utilize. Besides this, our current data includes traditional relational database systems that are large and clustered. With so many sensors and an existing database, how might we create our new Kafka-based architecture without impacting manufacturing?

3.1.2 A first change

One of the best ways to start our task is probably not with a big-bang approach—all our data does not have to move into Kafka at once. If we use a database today and want to kick the tires on the streaming data tomorrow, one of the easiest on-ramps starts with Kafka Connect. Although it can handle production loads, it does not have to out of the gate. We will take one database table and start our new architecture while letting the existing applications run for the time being. But first, let's get into some examples to gain familiarity with Kafka Connect.

3.1.3 Built-in features

The purpose of Kafka Connect is to help move data into or out of Kafka without writing our own producers and consumers. Connect is a framework that is already part of Kafka, which makes it simple to use previously built pieces to start your streaming work. These pieces are called *connectors*, and they were developed to work reliably with other data sources [2].

If you recall from chapter 2, some of the producer and consumer Java client real-world code that we used as examples showed how Connect abstracts those concepts away by using them internally with Connect. One of the easiest ways to start is by looking at how Connect can take a typical application log file and move it into a Kafka topic. The easiest option to run and test Connect on your local machine is standalone mode. Scaling can come later if we like what we can do in standalone mode! In the folder where you installed Kafka, locate the following files under the config directory:

- connect-standalone.properties
- connect-file-source.properties

Peeking inside the connect-standalone.properties file, you should see some configuration keys and values that should look familiar from some of the properties we used to make our own Java clients in chapter 2. Knowing the underlying producers and consumer clients can help us understand how Connect uses that same configuration to complete its work by listing items such as `bootstrap.servers`.

In our example, we'll take data from one data source and put that into Kafka so that we can treat data as being sourced from a Kafka file. Using the file connect-file-source.properties, included with your Kafka installation as an example template, let's create a file called alert-source.properties and place the text from listing 3.1 inside as the contents of our file. This file defines the configurations that we need to set up the file alert.txt and to specify the data be sent to the specific topic `kinaction_alert_connect`. Note that this example is following steps similar to the excellent Connect Quickstart guide at https://docs.confluent.io/3.1.2/connect/quickstart.html if you need more reference material. To learn even more detailed information, we recommend watching the excellent presentation of Randall Hauch (Apache Kafka committer and PMC) from the Kafka Summit (San Francisco, 2018) located at http://mng.bz/8WeD.

With configurations (and not code), we can get data into Kafka from any file. Because reading from a file is a common task, we can use Connect's prebuilt classes. In this case, the class is `FileStreamSource` [2]. For the following listing, let's pretend that we have an application that sends alerts to a text file.

Listing 3.1 Configuring Connect for a file source

```
name=alert-source
connector.class=FileStreamSource
tasks.max=1
file=alert.txt
topic=kinaction_alert_connect
```

Specifies the class that interacts with our source file

For standalone mode, 1 is a valid value to test our setup.

Monitors this file for changes

Names the topic where this data will be sent

The value of the `topic` property is significant. We will use it later to verify that messages are pulled from a file into the specific `kinaction_alert_connect` topic. The file alert.txt is monitored for changes as new messages flow in. And finally, we chose `1` for the value of `tasks.max` because we only really need one task for our connector and, in this example, we are not worried about parallelism.

NOTE If you are running ZooKeeper and Kafka locally, make sure that you have your own Kafka brokers still running as part of this exercise (in case you shut them down after the previous chapter).

Now that we have done the needed configuration, we need to start Connect and send in our configurations. We can start the Connect process by invoking the shell script `connect-standalone.sh`, including our custom configuration file as a parameter to that script. To start Connect in a terminal, run the command in the following listing and leave it running [2].

Listing 3.2 Starting Connect for a file source

```
bin/connect-standalone.sh config/connect-standalone.properties \
  alert-source.properties
```

Moving to another terminal window, create a text file named alert.txt in the directory in which we started the Connect service and add a couple of lines of text to this file using your text editor; the text can be anything you want. Now let's use the `console-consumer` command to verify that Connect is doing its job. For that, we'll open another terminal window and consume from the `kinaction_alert_connect` topic, using the following listing as an example. Connect should ingest this file's contents and produce the data into Kafka [2].

Listing 3.3 Confirming file messages made it into Kafka

```
bin/kafka-console-consumer.sh \
  --bootstrap-server localhost:9094 \
  --topic kinaction_alert_connect --from-beginning
```

Before moving to another connector type, let's quickly talk about the sink connector and how it carries Kafka's messages back out to another file. Because the destination (or sink) for this data is another file, we want to look at the connect-file-sink.properties file. A small change is shown in listing 3.4 as the new outcome is written to a file rather than read from a file as we did previously. We'll declare `FileStreamSink` to define a new role as a sink. The topic `kinaction_alert_connect` is the source of our data in this scenario. Placing the text from the following listing in a new file called alert-sink.properties sets up our new configuration [2].

Listing 3.4 Configuring Connect for a file source and a sink

```
name=alert-sink
connector.class=FileStreamSink
tasks.max=1
file=alert-sink.txt
topics=kinaction_alert_connect
```

An out-of-the-box class to which we delegate the work of interacting with our file

For standalone mode, 1 is a valid value to test our setup.

The destination file for any messages that make it into our Kafka topic

Names the topic that the data comes from

If the Connect instance is still running in a terminal, we'll need to close that terminal window or stop the process by pressing Ctrl-C. Then we'll restart it with the file-source and file-sink property files. Listing 3.5 shows how to restart Connect with both our custom alert source and sink properties [2]. The end result should be data flowing from a file into Kafka and back out to a separate destination.

Listing 3.5 Starting Connect for a file source and a sink

```
bin/connect-standalone.sh config/connect-standalone.properties \
  alert-source.properties alert-sink.properties
```

To confirm that Connect is using our new sink, open the sink file we used in our configuration, alert-sink.txt, and verify that you can see the messages that were in the source file and that these were sent to the Kafka topic.

3.1.4 Data for our invoices

Let's look at another requirement, dealing with our invoices for bike orders. Connect easily lets those with in-depth knowledge of creating custom connectors share them with others (to help those of us who may not be experts in these systems). Now that we have used a connector (listings 3.4 and 3.5), it should be relatively simple to integrate a different connector because Connect standardizes interaction with other systems.

To use Connect in our manufacturing example, let's look at applying an existing source connector that streams table updates from a local database to a Kafka topic. Again, our goal is not to change the entire data processing architecture at once. Instead, we'll show how we can bring in updates from a table-based database application and develop our new application in parallel while letting the other system exist as is. Note that this example is following steps similar to the guide at https://docs.confluent.io/kafka-connect-jdbc/current/source-connector/index.html, if you need more reference material.

Our first step is to set up a database for our local examples. For ease of use and to get started quickly, we'll use connectors from Confluent for SQLite. If you can run `sqlite3` in your terminal and get a prompt, then you are already set. Otherwise, use your favorite package manager or installer to get a version of SQLite that works on your operating system.

> **TIP** Check out the Commands.md file in the source code for this chapter to find installation instructions for the Confluent command line interface (CLI) as well as the JDBC connector using `confluent-hub`. The rest of the examples reference commands in the Confluent-installed directory *only* and not in the Kafka-installed directory.

To create a database, we will run `sqlite3 kafkatest.db` from the command line. In this database, we will then run the code in listing 3.6 to create the invoices table and to insert some test data in the table. As we design our table, it is helpful to think of how we

will capture changes into Kafka. Most use cases will not require us to capture the entire database but only changes after the initial load. A timestamp, sequence number, or ID can help us determine which data has changed and needs to be sent to Kafka. In the following listing, the ID or modified columns could be our guide for Connect to let Kafka know which data was modified in the table [3].

Listing 3.6 Creating the invoices table

```
CREATE TABLE invoices(                                  <-- Creates an invoices table
   id INT PRIMARY KEY     NOT NULL,                     <-- Sets an incremental ID so Connect
   title          TEXT    NOT NULL,                        knows which entries to capture
   details        CHAR(50),
   billedamt      REAL,
   modified     TIMESTAMP DEFAULT (STRFTIME('%s', 'now')) NOT NULL
);

INSERT INTO invoices (id,title,details,billedamt)  \    <-- Inserts test data
  VALUES (1, 'book', 'Franz Kafka', 500.00 );               into our table
```

By creating a file in the location etc/kafka-connect-jdbc/kafkatest-sqlite.properties, and after making slight changes to our database table name, we can see how additional inserts and updates to the rows cause messages to be sent into Kafka. Refer to the source code for chapter 3 in the Git repository to find more detailed setup instructions for finding and creating the JDBC connector files in the Confluent installation directory. It is not part of the Apache Kafka distribution like the file connector. Also, if the modified timestamp format gives an error, make sure to check out other options in the source code with this chapter.

Now that we have a new configuration, we need to start Connect to pass it kafka-test-sqlite.properties.

Listing 3.7 Starting Connect for a database table source

```
confluent-hub install confluentinc/kafka-connect-jdbc:10.2.0
confluent local services connect start
...
# See Commands.md for other steps
confluent local services connect connector config jdbc-source
--config etc/kafka-connect-jdbc/kafkatest-sqlite.properties
                                                        <-- We are using our new
                                                            database properties file.
```

Listing 3.7 shows how you can launch Connect with the Confluent CLI tool. The standalone connnect script, `connect-standalone.sh`, could have also been used [3]. Although the power of Kafka Connect is great for moving existing database tables to Kafka, our sensors (which are not database backed) are going to require a different technique.

3.2 Sensor event design

Because there are no existing connectors for our state-of-the-art sensors, we can directly interact with their event system through custom producers. The ability to hook into and write our producers to send data into Kafka is where we will look at the requirements in the following sections.

Figure 3.1 shows that there is a critical path of stages that need to work together. One of the steps is an additional quality check sensor. This sensor can be skipped to avoid processing delays if it goes down for maintenance or failure. Sensors are attached to all of the bikes' internal steps (represented by gears in figure 3.1), and they send messages to the clustered database machines that exist in the current system. There is also an administration console used remotely to update and run commands against the sensors already built into the system.

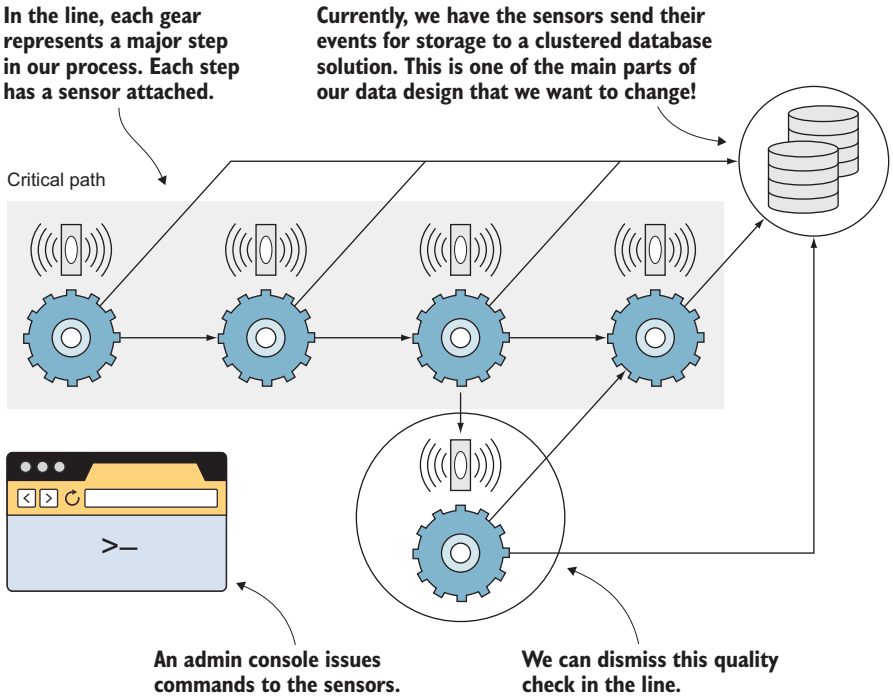

Figure 3.1 Factory setup

3.2.1 Existing issues

Let's start by discussing some of the issues that have come up in most of our previous use cases. The need for data to exist and to be available to users is a deep and challenging problem. Let's look at how we can deal with two of those challenges: data silos and recoverability.

DEALING WITH DATA SILOS

In our factory, the data and the processing are owned by an application. If others want to use that data, they would need to talk to the application owner. And what are the chances that the data is provided in a format that can be easily processed? Or what if it does not provide the data at all?

The shift from traditional "data thinking" makes the data available to everyone in its raw source. If you have access to the data as it comes in, you do not have to worry about the application API exposing it to specific formats or after custom transformations. And what if the application providing the API parses the original data incorrectly? To untangle that mess might take a while if we have to recreate the data from changes to the original data source.

RECOVERABILITY

One of the excellent perks of a distributed system like Kafka is that failure is an expected condition: it's planned for and handled! However, along with system blips, we also have the human element in developing applications. If an application has a defect or a logic issue that destroys our data, what would be our path to correct it? With Kafka, that can be as simple as starting to consume from the beginning topic as we did with the console consumer flag `--from-beginning` in chapter 2. Additionally, data retention makes it available for use again and again. The ability to reprocess data for corrections is powerful. But if the original event is not available, it might be hard to retrofit the existing data.

Because events are only produced once from the sensor source for a specific instance, the message broker can play a crucial part in our consumption pattern. If the message in a queuing system is removed from the broker after a subscriber reads the message, as in version 1.0 of the application in figure 3.2, it is gone from the system. If a defect in an application's logic is found after the fact, analysis would be needed to see if data can be corrected using what was left over from the processing of that original event because it will not be fired again. Fortunately, Kafka brokers allow for a different option.

Beginning with version 1.1, the application can replay those messages already consumed with the new application logic. Our new application code that fixed a logic mistake from version 1.0 can process all the events again. The chance to process our events again makes it easier to enhance our applications without data loss or corruption.

The replay of data can also show us how a value changes over time. It might be beneficial to draw a parallel between replaying the Kafka topic and the idea of a write-ahead log (WAL). With a WAL, we can tell what a value used to be and the changes that happened over time because modifications to values are written in the log before they are applied. WALs are commonly found in database systems and help a system recover if an action fails during a transaction. If you follow the events from the beginning to the end, you would see how data moves from its initial value to its current value.

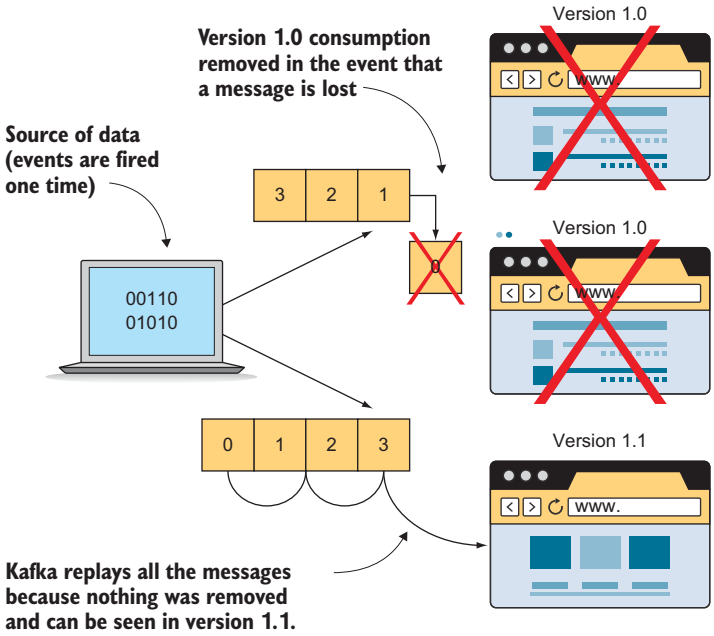

Figure 3.2 Looking at a developer coding mistake

When should data be changed?

Whether data is coming from a database or a log event, our preference is to get the data into Kafka first; then the data will be available in its purest form. But each step before it is stored in Kafka is an opportunity for the data to be altered or injected with various formatting or programming logic errors. Keep in mind that hardware, software, and logic can and will fail in distributed computing, so it's always great to get data into Kafka first, which gives you the ability to replay data if any of those failures occur.

3.2.2 Why Kafka is the right fit

Does Kafka even make sense in our fictional sensor use case? Of course, this is a book about Kafka, right? However, let's quickly try to pinpoint a couple of compelling reasons to give Kafka a try.

One thing that has been made clear by our clients is that their current database is getting expensive to scale vertically. By vertical scaling, we mean increasing things like CPU, RAM, and disk drives in an existing machine. (To scale dynamically, we would look at adding more servers to our environment.) With the ability to horizontally scale our cluster, we can hope to get more overall benefits for our buck. Although the servers that we run our brokers on might not be the cheapest machines money can buy, 32 GB or 64 GB of RAM on these servers can handle production loads [4].

The other item that probably jumped out at you is that we have events being produced continuously. This should sound similar to the definition of stream processing that we talked about earlier. The constant data feed won't have a defined end time or stopping point, so our systems should be ready to handle messages constantly. Another interesting point to note is that our messages are usually under 10 KB for our example. The smaller the message size and the amount of memory we can offer to page caches, the better shape we are in to keep our performance healthy.

During this requirements review for our scenario, some security-minded developers might have noticed there's no built-in disk encryption for the brokers (data at rest). However, that isn't a requirement for the current system. We will first focus on getting our system up and running and then worry about adding security at a later point in our implementation.

3.2.3 Thought starters on our design

One thing to note is which features are available for specific Kafka versions. Although we use a recent version for our examples (at the time of this writing, version 2.7.1), some developers might not have control over the current broker and client versions they are using due to their existing infrastructures. For this reason, it is good to keep in mind when some of the features and APIs we might use made their debut. Table 3.1 highlights some of the past major features but is not inclusive of all versions [5].

Table 3.1 Past Kafka version milestones

Kafka version	Feature
2.0.0	ACLS with prefix support and hostname verification (default for SSL)
1.0.0	Java 9 support and JBOD disk failure improvements
0.11.0.0	Admin API
0.10.2.0	Improved client compatibility
0.10.1.0	Time-based search
0.10.0.0	Kafka streams, timestamps, and rack awareness
0.9.0.0	Various security features (ACLS, SSL), Kafka Connect, and a new consumer client

Another thing to note as we focus on clients in the next few chapters is the feature-improved client compatibility. Broker versions since 0.10.0 can work with newer client versions. This is important because we can now try new versions of clients by upgrading them first, and the brokers can remain at their version until we decide that we want to upgrade them. This comes in handy as you work through this material if you are running against a cluster that already exists.

Now that we have decided to give Kafka a try, this might be a good time to decide how we want our data to exist. The following questions are intended to make us think about how we want to process our data. These preferences impact various parts of our design, but our main focus here is on figuring out the data structure; we will cover the

implementation in later chapters. This list is not meant to be complete, but it is a good starting point in planning our design:

- *Is it okay to lose any messages in the system?* For example, is one missed event about a mortgage payment going to ruin your customer's day and their trust in your business? Or is it a minor issue such as your social media account RSS feed missing a post? Although the latter is unfortunate, would it be the end of your customer's world?
- *Does your data need to be grouped in any way?* Are the events correlated with other events that are coming in? For example, are we going to be taking in account changes? In that case, we'd want to associate the various account changes with the customer whose account is changing. Grouping events up front might also prevent the need for applications to coordinate messages from multiple consumers while reading from the topic.
- *Do you need data delivered in a specific order?* What if a message gets delivered in an order other than when it occurred? For example, you get an order-canceled notice before the actual order. Because product ends up shipping due to order alone, the customer service impact is probably good enough reason to say that the ordering is indeed essential. Or course, not everything will need exact ordering. For example, if you are looking at SEO data for your business, the order is not as important as making sure that you can get a total at the end.
- *Do you only want the last value of a specific item, or is the history of that item important?* Do you care about how your data has evolved? One way to think about this looks at how data is updated in a traditional relational database table. It is mutated in place (the older value is gone and the newer value replaces it). The history of what that value looked like a day ago (or even a month ago) is lost.
- *How many consumers are you going to have?* Will they all be independent of each other, or will they need to maintain some sort of order when reading the messages? If you are going to have a lot of data that you want to consume as quickly as possible, that will inform and help shape how you break up your messages on the tail end of your processing.

Now that we have a couple of questions to ask for our factory, let's try to apply these to our actual requirements. We will use a chart to answer each scenario. We will learn how to do this in the following section.

3.2.4 User data requirements

Our new architecture needs to provide a couple of specific key features. In general, we want the ability to capture messages even if the consuming service is down. For example, if one of the consumer applications is down in our remote plant, we want to make sure that it can later process the events without dropping messages entirely. Additionally, when the application is out of maintenance or comes back up after a failure, we want it to still have the data it needs. For our example use case, we also want the status from our sensors as either working or broken (a sort of alert), and we want to make sure we can see if any part of our bike process could lead to total failure.

Along with the preceding information, we also want to maintain a history of the sensors' alert status. This data could be used in determining trends and in predicting failures from sensor data before actual events lead to broken equipment. We also want to keep an audit log of any users that push updates or queries directly against the sensors. Finally, for compliance reasons, we want to know who did what administration actions on the sensors themselves.

3.2.5 High-level plan for applying our questions

Let's focus closer on our requirements to create an audit log. Overall, it seems like everything that comes in from the management API will need to be captured. We want to make sure that only users with access permissions are able to perform actions against the sensors, and we should not lose messages, as our audit would not be complete without all the events. In this case, we do not need any grouping key because each event can be treated as independent.

The order does not matter inside our audit topic because each message will have a timestamp in the data itself. Our primary concern is that all the data is there to process. As a side note, Kafka itself does allow messages to be sorted by time, but the message payload can include time. However, this specific use case does not warrant this usage.

Figure 3.3 shows how a user would generate two audit events from a web administration console by sending a command to sensor 1 and another to sensor 3. Both commands should end up as separate events in Kafka. To make this a little clearer, table 3.2 presents a rough checklist of things we should consider regarding data for each

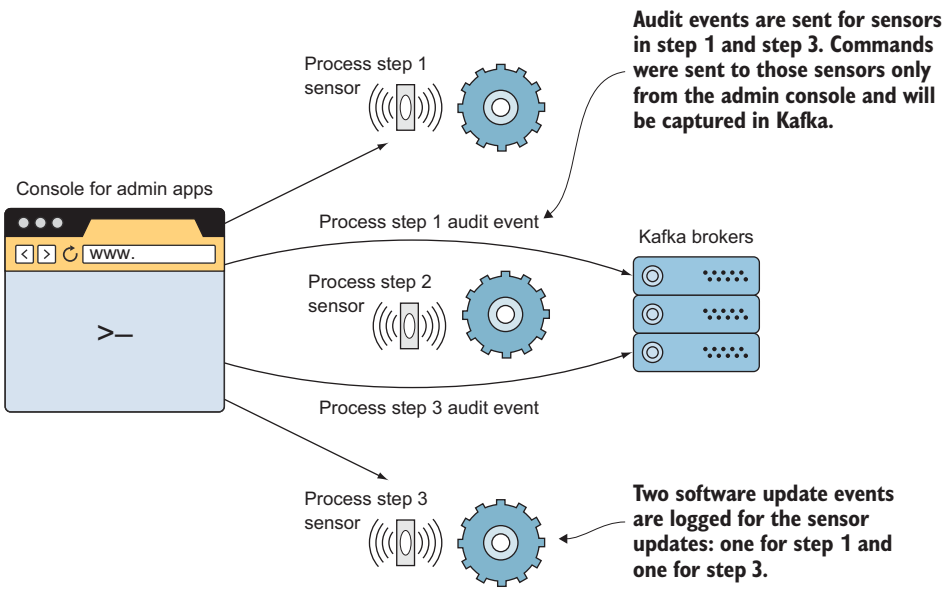

Figure 3.3 Audit use case

requirement. This at-a-glance view will help us when determining the configuration options we want to use for our producer clients.

Table 3.2 Audit checklist

Kafka feature	Concern?
Message loss	Yes
Grouping	No
Ordering	No
Last value only	No
Independent consumer	Yes

In this audit producer, we are concerned with making sure that no data is lost and that consuming applications do not have any worries about data being ordered or coordinated. Furthermore, the alert trend of our status requirements deals with each process in the bike's system with a goal of spotting trends. It might be helpful to group this data using a key. We have not addressed the term *key* in depth, but it can be thought of as a way to group related events.

We will likely use the bikes' part ID names at each stage of the internal system where the sensor is installed because they will be unique from any other name. We want to be able to look across the key at all of the events for a given stage to spot these trends over time. By using the same key for each sensor, we should be able to consume these events easily. Because alert statuses are sent every 5 seconds, we are not concerned about missing a message, as the next one should arrive shortly. If a sensor sends a "Needs Maintenance" message every couple of days, that is the type of information we want to have to spot trends in equipment failure.

Figure 3.4 shows a sensor watching each stage of the process. Those equipment alert events go into Kafka. Although not an immediate concern for our system, Kafka does enable us to pull that data into other data storage or processing system like Hadoop.

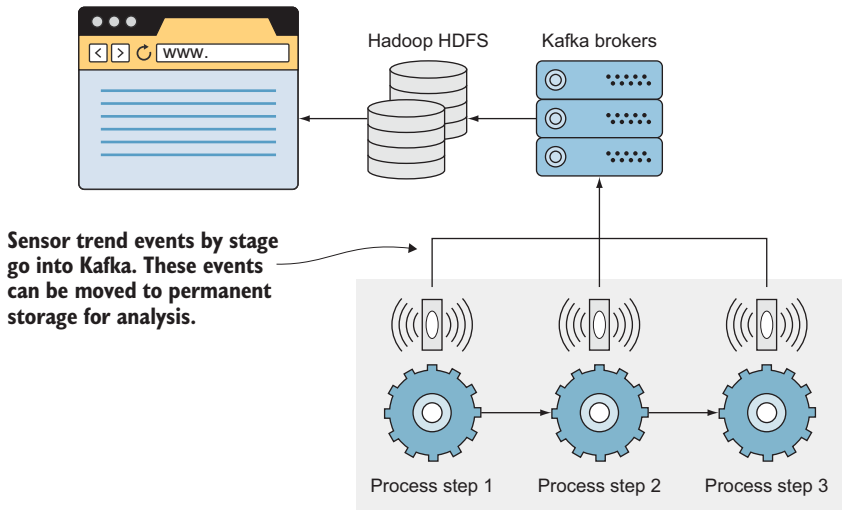

Figure 3.4 Alert trend use case

Table 3.3 Audit checklist

Kafka feature	Concern?
Message loss	No
Grouping	Yes
Ordering	No
Last value only	No
Independent consumer	Yes

Table 3.3 highlights that our goal is to group the alert results by stage and that we are not concerned about losing a message from time to time.

As for alerting on statuses, we also want to group by a key, which is the process stage. However, we do not care about past states of the sensor but rather the current status. In other words, the current status is all we care about and need for our requirements. The new status replaces the old, and we do not need to maintain a history. The word *replace* here is not entirely correct (or not what we are used to thinking). Internally, Kafka adds the new event that it receives to the end of its log file like any other message it receives. After all, the log is immutable and can only be appended to at the end of the file. How does Kafka make what appears to be an update happen? It uses a process called *log compaction*, which we will dig into in chapter 7.

Another difference we have with this requirement is the consumer usage assigned to specific alert partitions. Critical alerts are processed first due to an uptime requirement in which those events need to be handled quickly. Figure 3.5 shows an example of how critical alerts could be sent to Kafka and then consumed to

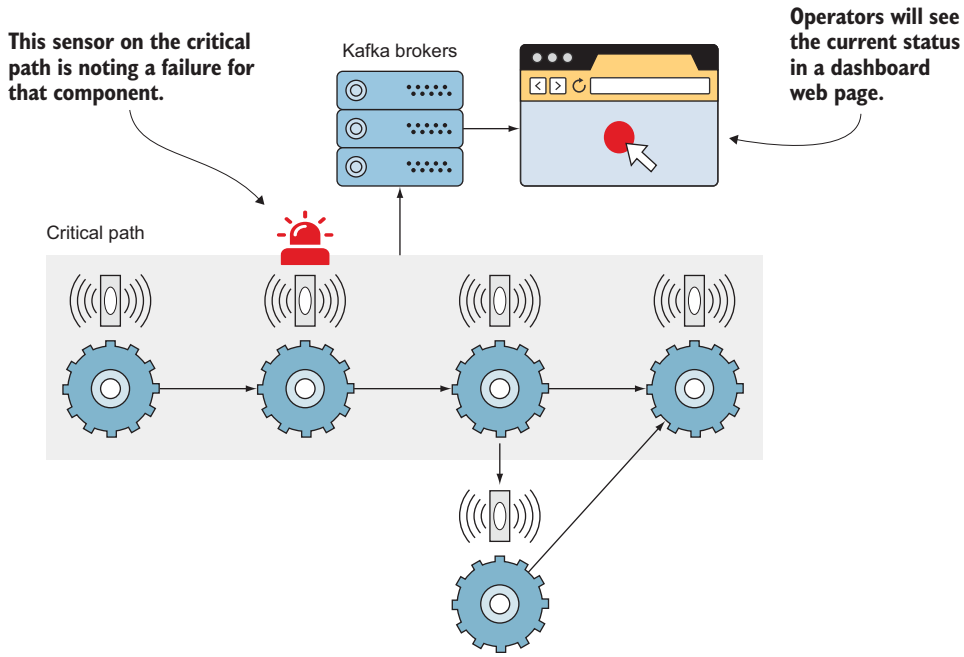

Figure 3.5 Alert use case

populate an operator's display to get attention quickly. Table 3.4 reinforces the idea that we want to group an alert to the stage it was created in and that we want to know the latest status only.

Taking the time to plan out our data requirements will not only help us clarify our application requirements but, hopefully, validate the use of Kafka in our design.

Table 3.4 Audit checklist

Kafka feature	Concern?
Message loss	No
Grouping	Yes
Ordering	No
Last value only	Yes
Independent consumer	No

3.2.6 Reviewing our blueprint

One of the last things to think about is how we want to keep these groups of data organized. Logically, the groups of data can be thought of in the following manner:

- Audit data
- Alert trend data
- Alert data

For those of you already jumping ahead, keep in mind that we might use our alert trend data as a starting point for our alerts topic; you can use one topic as the starting point to populate another topic. However, to start our design, we will write each event type from the sensors to their logical topic to make our first attempt uncomplicated and easy to follow. In other words, all audit events end up on an audit topic, all alert trend events end up on a alert trend topic, and our alert events on an alert topic. This one-to-one mapping makes it easier to focus on the requirements at hand for the time being.

3.3 Format of your data

One of the easiest things to skip, but critical to cover in our design, is the format of our data. XML and JSON are pretty standard formats that help define some sort of structure to our data. However, even with a clear syntax format, there can be information missing in our data. What is the meaning of the first column or the third one? What is the data type of the field in the second column of a file? The knowledge of how to parse or analyze our data can be hidden in applications that repeatedly pull the data from its storage location. Schemas are a means of providing some of this needed information in a way that can be used by our code or by other applications that may need the same data.

If you look at the Kafka documentation, you may have noticed references to another serialization system called Apache Avro. Avro provides schema definition support as well as schema storage in Avro files [6]. In our opinion, Avro is likely what you will see in Kafka code that you might encounter in the real world and why we will focus on this choice out of all the available options. Let's take a closer look at why this format is commonly used in Kafka.

3.3.1 Plan for data

One of the significant gains of using Kafka is that the producers and consumers are not tied directly to each other. Further, Kafka does not do any data validation by default. However, there is likely a need for each process or application to understand what that data means and what format is in use. By using a schema, we provide a way for our application's developers to understand the structure and intent of the data. The definition doesn't have to be posted in a README file for others in the organization to determine data types or to try to reverse-engineer from data dumps.

Listing 3.8 shows an example of an Avro schema defined as JSON. Fields can be created with details such as name, type, and any default values. For example, looking at the field `daysOverDue`, the schema tells us that the days a book is overdue is an `int` with a default value of `0`. Knowing that this value is numeric and not text (such as `one week`) helps to create a clear contract for the data producers and consumers.

Listing 3.8 Avro schema example

```
{
    "type" : "record",                          <-- JSON-defined Avro schema
    "name" : "kinaction_libraryCheckout",
    ...
    "fields" : [{"name" : "materialName",
                 "type" : "string",
                 "default" : ""},

                {"name" : "daysOverDue",        <-- Maps directly to a field name
                 "type" : "int",
                 "default" : 0},                <-- Provides the default value

                {"name" : "checkoutDate",
                 "type" : "int",
                 "logicalType": "date",
                 "default" : "-1"},

                 {"name" : "borrower",
                  "type" : {
                        "type" : "record",
                        "name" : "borrowerDetails",
                        "fields" : [
                           {"name" : "cardNumber",
                            "type" : "string",
                            "default" : "NONE"}
                         ]},
                         "default" : {}
                }
    ]
}
```

Defines a field with a name, type, and default value

By looking at the example of the Avro schema in listing 3.8, we can see that questions such as "Do we parse the `cardNumber` as a number or a string (in this case, `string`)" are easily answered by a developer looking at the schema. Applications could automatically

use this information to generate data objects for this data, which helps to avoid parsing data type errors.

Schemas can be used by tools like Apache Avro to handle data that evolves. Most of us have dealt with altered statements or tools like Liquibase to work around these changes in relational databases. With schemas, we start with the knowledge that our data will probably change.

Do we need a schema when we are first starting with our data designs? One of the main concerns is that if our system's scale keeps getting larger, will we be able to control the correctness of data? The more consumers we have could lead to a burden on the testing that we would need to do. Besides the growth in numbers alone, we might not even know all of the consumers of that data.

3.3.2 Dependency setup

Now that we have discussed some of the advantages of using a schema, why would we look at Avro? First of all, Avro always is serialized with its schema [7]. Although not a schema itself, Avro supports schemas when reading and writing data and can apply rules to handle schemas that change over time. Also, if you have ever seen JSON, it is pretty easy to understand Avro. Besides the data, the schema language itself is defined in JSON as well. If the schema changes, you can still process data [7]. The old data uses the schema that existed as part of its data. On the other hand, any new formats will use the schema present in their data. Clients are the ones who gain the benefit of using Avro.

Another benefit of looking at Avro is the popularity of its usage. We first saw it used on various Hadoop efforts, but it can be used in many other applications. Confluent also has built-in support for most parts of their tooling [6]. Bindings exist for many programming languages and should not be hard to find, in general. Those who have past "bad" experiences and prefer to avoid generated code can use Avro dynamically without code generation.

Let's get started with using Avro by adding it to our pom.xml file as the following listing shows [8]. If you are not used to pom.xml or Maven, you can find this file in our project's root directory.

Listing 3.9 Adding Avro to pom.xml

```
<dependency>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro</artifactId>
  <version>${avro.version}</version>
</dependency>
```

Adds this entry as a dependency to the project's pom.xml file

Because we are already modifying the POM file, let's go ahead and include a plugin that generates the Java source code for our schema definitions. As a side note, you can also generate the sources from a standalone Java JAR, avro-tools, if you do not want to use a Maven plugin. For those who do not prefer code generation in their source code projects, this is not a hard requirement [9].

Listing 3.10 shows how to add the avro-maven-plugin to our pom.xml as suggested by the Apache Avro Getting Started with Java documentation site [8]. The code in this listing omits the configuration XML block. Adding the needed configuration also lets Maven know that we want to generate source code for the Avro files found in the source directory we list and to output the generated code to the specified output directory. If you like, you can change the source and output locations to match your specific project structure.

Listing 3.10 Adding the Avro Maven plugin to pom.xml

```
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>
  <version>${avro.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>schema</goal>
      </goals>
      ...
    </execution>
  </executions>
</plugin>
```

Sets the artifact ID needed in our pom.xml as a plugin

Configures the Maven phase

Configures the Maven goal

Let's start defining our schema by thinking about the data types we want to use, beginning with our alert status scenario. To start, we'll create a new file named kinaction _alert.avsc with a text editor. The following listing shows the schema definition. We will name our Java class `Alert` as we will interact with it after the generation of source code from this file.

Listing 3.11 Alert schema: kinaction_alert.avsc

```
{
  ...
  "type": "record",
  "name": "Alert",
  "fields": [
    {
      "name": "sensor_id",
      "type": "long",
      "doc": "The unique id that identifies the sensor"
    },
    {
      "name": "time",
      "type": "long",
      "doc":
        "Time alert generated as UTC milliseconds from epoch"
    },
    {
```

Names the created Java class

Defines the data types and documentation notes

```
      "name": "status",
      "type": {
        "type": "enum",
        "name": "AlertStatus",
        "symbols": [
          "Critical",
          "Major",
          "Minor",
          "Warning"
        ]
      },
      "doc":
       "Allowed values sensors use for current status"
    }
  ]
}
```

In listing 3.11, which shows a definition of alerts, one thing to note is that `"doc"` is not a required part of the definition. However, there is certainly value in adding details that will help future producer or consumer developers understand what the data means. The hope is to stop others from inferring our data's meaning and to be more explicit about the content. For example, the field `"time"` always seems to invoke developer anxiety when seen. Is it stored in a string format? Is time zone information included? Does it include leap seconds? The `"doc"` field can provide that information. A namespace field, not shown in listing 3.11, turns into the Java package for the generated Java class. You can view the full example in the source code for the book. The various field definitions include the name as well as a type.

Now that we have the schema defined, let's run the Maven build to see what we are working with. The commands `mvn generate-sources` or `mvn install` can generate the sources in our project. This should give us a couple of classes, `Alert.java` and `AlertStatus.java`, that we can now use in our examples.

Although we have focused on Avro itself, the remaining part of the setup is related to the changes we need to make in our producer and consumer clients to use the schema that we created. We can always define our own serializer for Avro, but we already have an excellent example provided by Confluent. Access to the existing classes is accomplished by adding the `kafka-avro-serializer` dependency to our build [10]. The following listing shows the pom.xml entry that we'll add. This is needed to avoid having to create our own Avro serializer and deserializer for the keys and values of our events.

Listing 3.12 Adding the Kafka serializer to pom.xml

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>${confluent.version}</version>
</dependency>
```

Adds this entry as a dependency in the project's pom.xml file

If you are using Maven to follow along, make sure that you place the Confluent repository in your pom file. This information is needed to let Maven know where to get specific dependencies [11].

```
<repository>
    <id>confluent</id>
    <url>https://packages.confluent.io/maven/</url>
</repository>
```

With the build set up and our Avro object ready to use, let's take our example producer, `HelloWorldProducer`, from the last chapter and slightly modify the class to use Avro. Listing 3.13 shows the pertinent changes to the producer class (not including imports). Notice the use of `io.confluent.kafka.serializers.KafkaAvroSerializer` as the value of the property `value.serializer`. This handles the `Alert` object that we created and sent to our new `kinaction_schematest` topic.

Before, we could use a string serializer, but with Avro, we need to define a specific value serializer to tell the client how to deal with our data. The use of an `Alert` object rather than a string shows how we can utilize types in our applications as long as we can serialize them. This example also makes use of the Schema Registry. We will cover more details about the Schema Registry in chapter 11. This registry can have a versioned history of schemas to help us manage schema evolution.

Listing 3.13 Producer using Avro serialization

```
public class HelloWorldProducer {

  static final Logger log =
    LoggerFactory.getLogger(HelloWorldProducer.class);

  public static void main(String[] args) {
    Properties kaProperties = new Properties();
    kaProperties.put("bootstrap.servers",
      "localhost:9092,localhost:9093,localhost:9094");
    kaProperties.put("key.serializer",
      "org.apache.kafka.common.serialization.LongSerializer");
    kaProperties.put("value.serializer",
      "io.confluent.kafka.serializers.KafkaAvroSerializer");
    kaProperties.put("schema.registry.url",
      "http://localhost:8081");

    try (Producer<Long, Alert> producer =
      new KafkaProducer<>(kaProperties)) {
      Alert alert =
        new Alert(12345L,
          Instant.now().toEpochMilli(),
          Critical);

      log.info("kinaction_info Alert -> {}", alert);
```

Sets value.serializer to the KafkaAvroSerializer class for our custom Alert value

Creates a critical alert

```
      ProducerRecord<Long, Alert> producerRecord =
          new ProducerRecord<>("kinaction_schematest",
                               alert.getSensorId(),
                               alert);

      producer.send(producerRecord);
    }
  }
}
```

The differences are pretty minor. The type changes for our `Producer` and `ProducerRecord` definitions, as do the configuration settings for the `value.serializer`.

Now that we have produced messages using `Alert`, the other changes would be on the consumption side of the messages. For a consumer to get the values produced to our new topic, it will have to use a value deserializer; in this case, `KafkaAvroDeserializer` [10]. This deserializer works to get back the value that was serialized by the producer. This code can also reference the same `Alert` class generated in the project. The following listing shows the significant changes for the consumer class `HelloWorldConsumer`.

Listing 3.14 Consumer using Avro serialization

```
public class HelloWorldConsumer {

  final static Logger log =
    LoggerFactory.getLogger(HelloWorldConsumer.class);

  private volatile boolean keepConsuming = true;

  public static void main(String[] args) {
    Properties kaProperties = new Properties();
    kaProperties.put("bootstrap.servers", "localhost:9094");
    ...
    kaProperties.put("key.deserializer",
      "org.apache.kafka.common.serialization.LongDeserializer");
    kaProperties.put("value.deserializer",
      "io.confluent.kafka.serializers.KafkaAvroDeserializer");
    kaProperties.put("schema.registry.url", "http://localhost:8081");

    HelloWorldConsumer helloWorldConsumer = new HelloWorldConsumer();
    helloWorldConsumer.consume(kaProperties);

    Runtime.getRuntime()
      .addShutdownHook(
        new Thread(helloWorldConsumer::shutdown)
      );
  }

  private void consume(Properties kaProperties) {

    try (KafkaConsumer<Long, Alert> consumer =
      new KafkaConsumer<>(kaProperties)) {
      consumer.subscribe(
```

Sets value.serializer to the KafkaAvroSerializer class due to the Alert usage

KafkaConsumer typed to handle Alert values

```
        List.of("kinaction_schematest")
      );

      while (keepConsuming) {
        ConsumerRecords<Long, Alert> records =
          consumer.poll(Duration.ofMillis(250));
        for (ConsumerRecord<Long, Alert> record :
          records) {
            log.info("kinaction_info offset = {}, kinaction_value = {}",
              record.offset(),
              record.value());
        }
      }
    }
  }

  private void shutdown() {
    keepConsuming = false;
  }
}
```

Updates ConsumerRecord to handle Alert values

As with the producer, the consumer client does not require many changes due to the power of updating the configuration deserializer and Avro! Now that we have some ideas about the *what* we want to accomplish and our data format, we are well equipped to tackle the *how* in our next chapter. We will cover more schema-related topics in chapter 11 and move on to a different way to handle our object types in the example project in chapters 4 and 5. Although the task of sending data to Kafka is straightforward, there are various configuration-driven behaviors that we can use to help us satisfy our specific requirements.

Summary

- Designing a Kafka solution involves understanding our data first. These details include how we need to handle data loss, ordering of messages, and grouping in our use cases.
- The need to group data determines whether we will key the messages in Kafka.
- Leveraging schema definitions not only helps us generate code, but it also helps us handle future data changes. Additionally, we can use these schemas with our own custom Kafka clients.
- Kafka Connect provides existing connectors to write to and from various data sources.

References

1 J. MSV. "Apache Kafka: The Cornerstone of an Internet-of-Things Data Platform" (February 15, 2017). https://thenewstack.io/apache-kafka-cornerstone-iot-data-platform/ (accessed August 10, 2017).

2 "Quickstart." Confluent documentation (n.d.). https://docs.confluent.io/3.1.2/connect/quickstart.html (accessed November 22, 2019).

3 "JDBC Source Connector for Confluent Platform." Confluent documentation (n.d.). https://docs.confluent.io/kafka-connect-jdbc/current/source-connector/index.html (accessed October 15, 2021).

4 "Running Kafka in Production: Memory." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/deployment.html#memory (accessed June 16, 2021).

5 "Download." Apache Software Foundation (n.d.). https://kafka.apache.org/downloads (accessed November 21, 2019).

6 J. Kreps. "Why Avro for Kafka Data?" Confluent blog (February 25, 2015). https://www.confluent.io/blog/avro-kafka-data/ (accessed November 23, 2017).

7 "Apache Avro 1.8.2 Documentation." Apache Software Foundation (n.d.). https://avro.apache.org/docs/1.8.2/index.html (accessed November 19, 2019).

8 "Apache Avro 1.8.2 Getting Started (Java)): Serializing and deserializing without code generation." Apache Software Foundation (n.d.). https://avro.apache.org/docs/1.8.2/gettingstartedjava.html#download_install (accessed November 19, 2019).

9 "Apache Avro 1.8.2 Getting Started (Java): Serializing and deserializing without code generation." Apache Software Foundation (n.d.). https://avro.apache.org/docs/1.8.2/gettingstartedjava.html#Serializing+and+deserializing+without+code+generation (accessed November 19, 2019).

10 "Application Development: Java." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/app-development/index.html#java (accessed November 20, 2019).

11 "Installation: Maven repository for jars." Confluent documentation (n.d.). https://docs.confluent.io/3.1.2/installation.html#maven-repository-for-jars (accessed November 20, 2019).