CONFLUENT

10 Principles for Streaming Services

Confluent, © 2018 Confluent, Inc.

Table of Contents

1. Pick topics with business significance and use sensible keys. 2

2. Decouple publishers from subscribers by avoiding Request Response. 3

3. Apply the Inverse Conway Maneuver 4

4. Apply the Single Writer Principle. 5

5. Keep data sets *alive* within the broker. 6

6. Use the log to Event Source to regenerate a service's state. 7

7. Prefer stream processing over maintaining historic views. 7

8. Use historical views when appropriate. 8

9. Use schemas, especially if data is retained. 8

10. Consider stream management services. 9

11. Conclusion 10

As streaming data becomes an increasingly significant factor for modern, digital-age businesses, organizations need flexible tools for managing data streams efficiently and in real-time. Microservices architectures enable businesses to evolve their systems away from the slow and unresponsive shared-state architectures of the past. Businesses can deploy a microservice-based environment either with event-based or request-response approaches, or a hybrid of the two. The trend in business today is towards hybrid or predominantly event-driven architectures, in which the services themselves raise events.

Such events typically map to the real-world flow of the business they represent. For example, a user makes a purchase, which defines an event. This in turn triggers a series of downstream services (payment, fulfillment, fraud detection, etc.) Businesses are increasingly turning to Apache Kafka® as an active brokering technology for managing these streams of events. Kafka accepts messages and places them into topic to which any service can subscribe. Kafka decouples producing and consuming services and provides improved properties for scalability, availability and data retention.

Confluent Platform enables businesses to run a streaming platform built on Apache Kafka® at scale. Confluent offers connectivity and data compatibility capabilities to simplify operating and maintaining a Kafka cluster in support of a microservices architecture.

When implementing such a solution, there are a number of critical factors that a business must take into consideration to ensure successful deployment. This paper provides 10 principles for streaming services, a list of items to be mindful of when designing and building a microservices system using the Confluent Platform.

The principles covered in this paper are:

1. Pick topics with business significance and use sensible keys.
2. Decouple publishers from subscribers by avoiding Request Response.

© 2014-2020 Confluent, Inc.

3. Apply the Inverse Conway Maneuver.
4. Apply the single writer principle.
5. Keep data sets 'alive' within the broker.
6. Use the log to event source.
7. Prefer stream processing over maintaining historic views.
8. Use historical views when appropriate.
9. Use schemas, especially if data is retained.
10. Consider stream management services.

In the sections that follow, each of these principles is outlined in detail.

1. Pick topics with business significance and use sensible keys.

The more meaningful a topic is to the user, the more effective it will be. The best choices are typically items with real-world counterparts: `orders`, `payments`, `returns`, `invoices`. Each topic will typically have a workflow associated with it `OrderProposed`, `OrderConfirmed` etc.

Give your messages meaningful IDs. Include the service name (this means new service instances, which need to write data, can add their service name to the keyspace and know they will avoid collisions). Include the entity name and assign a unique identifier. Do this on the originating process, so cross-process calls can implement idempotence reliably.

Further to this, if the entities are mutable, as many business events are, add a version identifier. The offset in the log will provide this implicitly, but it is generally better practice to version explicitly so that version carries right the way through your whole data pipeline. See Figure 1, below.

© 2014-2020 Confluent, Inc.

Figure 1. Use Meaningful Topics

2. Decouple publishers from subscribers by avoiding Request Response.

Avoid point-to-point, request-response-styled communication patterns. Request response patterns couple sender and receiver, as the sender has to know who they are calling. Using a broker reverses this using what is termed "receiver driven flow control". This means the receiving service controls what messages they get, not the calling service. This gives them a previously unattainable degree of autonomy. So services can be added, or altered, without ever having to make changes to other services that sit upstream. This is a very desirable property for an architecture to exhibit.

Having said that, be wary of using Kafka for everything. Using a durable, retentive technology for ephemeral tasks, like moving the contents of a shopping cart into a web browser, can be overkill. A lightweight request response protocol often makes more sense in such settings. So while Kafka can support request response, it should be used sparingly for such tasks.

A better approach is to use Kafka Streams to create a streaming, materialized view. This can then be queried from a browser using the ***Queryable State feature***. [1: Thereska, Eno et al. "Unifying Stream Processing and Interactive Queries in Apache Kafka®" Confluent Blog October 26, 2016 https://goo.gl/VTIBGS] Alternatively, of

course, you can roll your own.

3. Apply the Inverse Conway Maneuver

Building microservices typically means decentralising software development across many separate services and hence many separate teams. Among other things, this makes it easier for them to grow. But it's important to avoid the pre-existing, typically siloed shape of most organisations leaking into your software architecture. Conway's Law states that business systems ultimately reflect the organizational structure of the businesses they serve [2: Conway, Melvin "How Do Committees Invent?" Harvard Business Review April, 1968 https://goo.gl/MlBhyZ]. An approach to combat this is to build services that crosscut silos. This is sometimes termed the Inverse Conway Maneuver, an approach where the organization is restructured to reflect the optimal architecture of its supporting systems [3: Bloomberg, Jason "DevOps Insights into Conway's Law" Intellyx June, 2015 https://goo.gl/CdsU66]. Consider this approach and avoid siloed services.

© 2014-2020 Confluent, Inc.

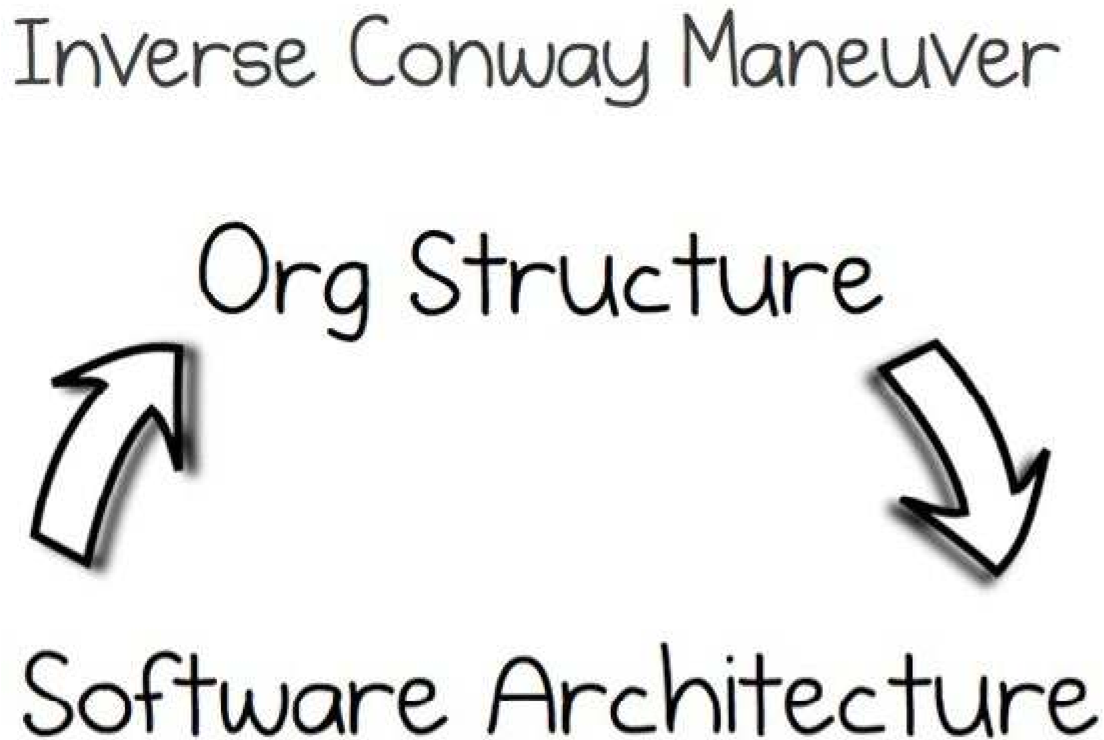


Figure 2. The Inverse Conway Maneuver recommends evolving your team and organizational structure to promote your desired architecture

4. Apply the Single Writer Principle.

As much as possible, have a single service take sole responsibility for changes to each type of data/event. This means all changes are made by a single service, and then propagated downstream. This is an important design principle to incorporate early on. It is typically hard to retrofit into existing flows not designed with write-responsibilities in mind.

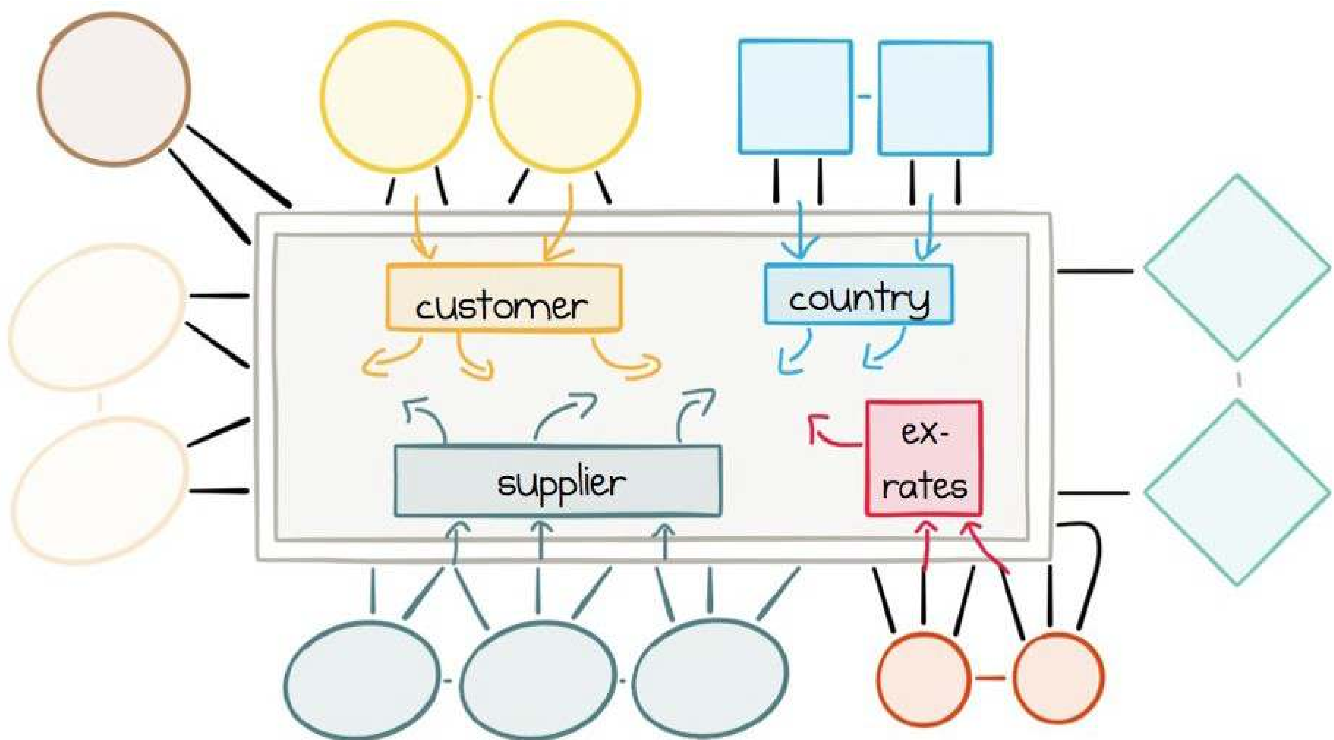
The implication is that local copies, which reside in other services, should always be read-only (i.e. *immutable*). See Figure 3 below. In this simple example, changes to Orders are always directed back to the Order Service. Local copies of Orders within the Payment, Fulfillment, or Stock services are immutable. For more information on why this property is so important see Immutability Changes Everything. [4: Helland, Pat Immutability Changes Everything 2015 https://goo.gl/35gMAu]

© 2014-2020 Confluent, Inc.

1. Use the Single Writer Principle from the Start image::fig-3.png[align=center]

5. Keep data sets *alive* within the broker.

One of the most powerful capabilities provided by Kafka is the option to leave data sets inside the broker long term. Using compacted topics, as described above, this provides a middle ground between what is effectively a stream and what is effectively a table.



In this model, data sets can be kept alive and available for any service to dip into. Such datasets do not suffer the tight couplings associated with shared databases, but they do provide the core aspects of data persistence. Kafka Streams separates out the function for joining these streams and embeds it right in each service. This makes it easier for services to dip into the shared streams and join together any new datasets they need.

© 2014-2020 Confluent, Inc.

6. Use the log to Event Source to regenerate a service's state.

A common pattern with traditional messaging is to immediately record messages received to a database, simply because the messaging system is ephemeral: once the message is acknowledged, it's gone. When using Kafka, the log can be retained for a period of time, or indefinitely (typically making use of the Compacted Topics feature). The log can then be relied upon for reprocessing should some issue require it.

In addition to this, the log, or a Kafka Streams State Store, can be used to Event Source side effects. Event sourcing is an important pattern for stateful services: Services which contain a number of steps, some of which have side effects (make external calls etc). The progression of these steps must be recorded so that, should the service crash, it can resume from the correct position in its workflow. For example, a payment may go to an external provider, and the service would want to record that fact, so that after a crash it was not repeated.

Kafka's distributed log can be used for event sourcing such an intermediary state. Alternatively, the Kafka Streams API provides an even richer set of primitives. This means business processes can be modelled either as a Kafka Streams multi-step workflow, or alternatively the state store can be used directly to store, and recover, such intermediary checkpoints. Another option is to use an external database.

7. Prefer stream processing over maintaining historic views.

Keeping and maintaining data in local databases always comes with risks. Accidents happen. Data becomes corrupted. The local copy gradually diverges from the source.

One way to combat this is to, whenever possible, operate directly on shared streams, rather than persisting to a database and processing there. This stream processing

© 2014-2020 Confluent, Inc.

"middle ground" provides a consistently lighter-weight and more accurate solution in this regard. It is also often quicker to bootstrap, allowing developers to quickly build stream-centric services.

8. Use historical views when appropriate.

Following on from Principle 7, an external database or warehouse, of one form or another, may still be required. For example, to support ad hoc analytical queries that go beyond what is practical with a pure streaming platform.

When creating such an environment, use the Apache Kafka® Connect API to replicate data and keep it read-only. The immutability of data in such an environment is key (See point 4: Apply the Single Writer Principle).

One advantage of having such second level stores, with the history also available in Kafka, is they can be broadened, incrementally. That is to say they can be regenerated, at will, if the base data is retained in Kafka. Use this incremental, iterative approach to keep your services agile and responsive in the face of large datasets. This pattern can also be extended to provide CQRS.

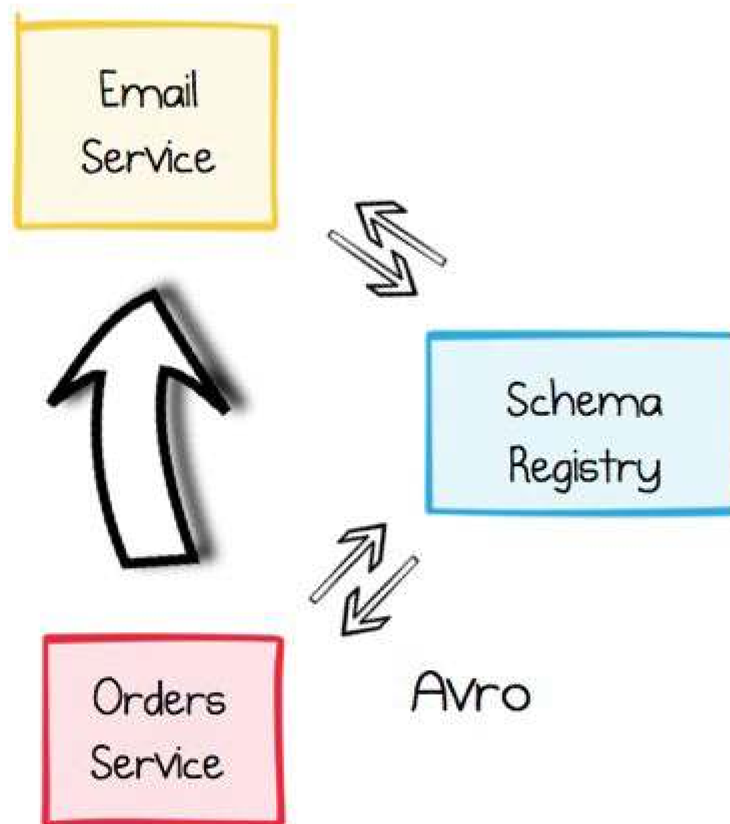
9. Use schemas, especially if data is retained.

Schemas are an important part of keeping data fresh and sharing it between teams and services. Schema-less data doesn't age well. Without a schema in place, leveraging historical data leads to a host of version and compatibility issues. Also, changing the contract between services becomes painful as the architecture grows.

Applying a schema to messages provides a contract. Something that services can program to, protect them against badly behaved services, and allow them to reason about backwards compatibility.

© 2014-2020 Confluent, Inc.

The Confluent Schema Registry helps manage this process in a multi-service environment. It allows administrators or services to register Avro schemas and define their compatibility. This typically proves very useful as service interactions become more complex or data is retained.



10. Consider stream management services.

Stream management is an important consideration for richer use cases where data sets are stored within Kafka long term. Managing such data means assuming many of the traditional roles of the DBA (database administrator), including managing data partitioning, migration, and environment management.

Providing a simple and powerful toolset for transforming streaming data, Kafka Streams is well suited to building such administrative tools, such as handling non-backward-compatible schema changes by downconverting streams or duplicating

© 2014-2020 Confluent, Inc.

streams into both Versioned and Compacted forms.

11. Conclusion

While they do not represent a full implementation strategy, the principles presented here provide a good outline for the kinds of issues that will arise when you look to implement a streaming solution. The best time to be aware of these issues is early on. A system designed with these 10 principles in mind will be better prepared for the challenges a new streaming environment is likely to face. And it will enjoy significant advantages over a system to which these principles must be added later.

These principles support an environment that is both capable and highly adaptable to a rapidly changing business and technology reality. For more information on how Apache Kafka® works with Microservices, see our earlier paper, Microservices in the Apache Kafka® Ecosystem.

© 2014-2020 Confluent, Inc.