CHAPTER 4

Cloud Native Architecture and Design Patterns

The *Pattern Language* is an organized and coherent set of patterns, each of which describes a problem and the core of a solution that can be used in many ways within a specific field of expertise.

When architects and designers work on a particular problem, it is unusual for them to think of a new solution that is completely distinct from existing ones. They often recall or remember a similar problem they already solved and reuse the essence of that solution. Their problem may in fact recur again and again in various projects and implementations. Using the earlier solution to solve this recurring problem has a name; it is called using a pattern.

A software pattern is a solution to a recurring problem within a given context. Each pattern describes a context, a problem, and a solution. Patterns reflect how the code or components are developed and interact with each other. Using patterns simplifies design and architecture problems.

Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Software architects and designers who know available software architecture and design patterns can recognize when one can be applied in a design scenario. This chapter explains the details of patterns with real-time problem scenarios.

This chapter begins by explaining what software architecture patterns are and how they can be used in your design. It then briefly covers all the commonly available patterns and provides detailed information on cloud native-related patterns including Gang of Four patterns, enterprise integration patterns, microservices patterns, etc.

S. R. Goniwada, *Cloud Native Architecture and Design*, https://doi.org/10.1007/978-1-4842-7226-8_4

In this chapter, I will cover the following topics:

- Evolution of software architecture patterns
- Software architecture pattern usage
- Architecture styles
- Gang of Four patterns, including the enterprise integration pattern
- Details of cloud native and microservices patterns
- Infrastructure patterns, testing patterns, database patterns, and transactional patterns
- Anti-patterns
- Do's and don'ts of pattern usage

Evolution of Design Patterns

Economic changes in the 19th century provided the catalyst for the rise of modern architecture and the creation of some iconic buildings. Christopher Alexander was a vocal critic of utilized space and developed theories for architectural and urban design. He published a theory of architecture: *The Timeless Way of Building in 1979, A Pattern Language in 1977 and the Oregon Experiment 1975*.

This *Pattern Language*, as it's called, details 253 patterns that serve as generic guiding principles for design.

Design patterns in computer science achieved prominence when *Design Pattern: Elements of Reusable Object-Oriented Software* by the "Gang of Four" was published in 1994 by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The *Design Pattern* book used objects and interfaces instead of walls and doors, but at the core of both kinds of patterns are solutions to a problem in a context.

The next progression in the pattern world was *Studies in Computational Science: Parallel Programming Paradigms*, a book about programming techniques written by Per Brinch Hansen. He was a Danish-American computer scientist known for his work in operating systems, concurrent programming, and parallel and distributed computing. The author's main point is that the lack of proper programming techniques is the source of many difficulties in computing. This book mainly addresses concurrent programs, divide-and-conquer paradigms, parallel parallelism, etc.

The next progression in the pattern world was *Pattern-Oriented Software Architecture: A System of Patterns, Volume 1*, which was written in 1996 by Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad. This book details how to design application and middleware software to run in concurrent and networked environments, event handling, synchronization, services access and configuration, and concurrency.

The next progression in the pattern world was *Smalltalk Best Practices Pattern* written by Kent Beck in 1997. This book is all about choosing names of objects, variables, and methods; how to break logic into methods; and how to communicate your implementation. Smalltalk is one of the most influential programming languages and was one of the first object-oriented programming languages, so all other languages that come after Smalltalk like Java, Python, Ruby, etc., were influenced by Smalltalk.

The next progression in the pattern world was *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects* written by Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann in 2000. It is the second volume in the Pattern-Oriented Software Architecture series. This book focuses on networking and concurrency.

The next progression in the pattern world was *Pattern of Enterprise Application Architecture* written by Martin Fowler in 2002. This book is about enterprise architecture patterns like how to layer an enterprise application, how to organize domain logic, how to design web-based applications, and how to implement distributed design.

The next progression in the pattern world was *Enterprise Integration Pattern: Designing, Building, and Deploying Messaging Solutions* written by Gregor Hohpe and Bobby Woolf in 2003. This book covers enterprise integration and messaging with both synchronous and asynchronous loosely coupled patterns.

The next progression in the pattern world was *Head First Design Pattern* written by Eric Freeman, Elisabeth Freeman, Bert Bates, and Kathy Sierra in 2004. In this book, the authors illustrated already available patterns in a graphical way with simple understandable terms and with examples.

The next book in the pattern world was *Software Architecture Patterns* written by Mark Richards in 2015. In this book, the author provides details of modern-day architecture patterns such as event-driven architecture, microservices, layered architecture, etc.

The next book in the pattern world was *Microservice Patterns* written by Chris Richardson in 2018. In this book, the author provides the details of microservices patterns such as event-driven architecture, microservices, etc.

There are various other books available on the market about patterns, and each author gives details based on their experience with the best possible examples. In this book, I am not covering the entire set of patterns but providing brief details of existing patterns that are relevant to a cloud native architecture.

This chapter covers the details of relevant cloud native patterns, including object-oriented, enterprise application, and enterprise integration patterns. It also provides some examples.

What Are Software Patterns?

A *software pattern* is a solution to a recurring problem within a given context. Each pattern describes a context, a problem, and a solution. Patterns reflect how code or components are developed and interact with each other. Using patterns simplifies design and architecture problems. Some people interpret what is and isn't a pattern differently. One person's pattern can be another person's architecture style or building blocks. In general, a pattern has a pattern name, problem, solution, and consequences.

When an architect and designer work on a particular problem, it is unusual for them to think of a new solution that is completely distinct from existing ones. They often recall a similar problem they have already solved and reuse the essence of that solution in the new situation. In fact, the same problem may recur again and again in various projects and implementations. Using the earlier solution to solve this recurring problem is called using a *pattern*.

- Patterns can be seen as building blocks of more complex solutions.
- Their function is a common language used by technology architects and designers to describe solutions.

Architecture Style, Architecture Pattern, and Design Pattern

The architecture style, architecture pattern, and design pattern are not mutually exclusive but complement each other, and all of them can provide some insight into the development of a solution. There are small differences between all three and,

again, different interpretations from person to person. Some say architecture styles and architecture patterns are the same, but others say they are different. I will try to highlight the differences based on my experience, and the rest I leave to you to judge.

An architecture style describes how to organize components of the architecture and code. It is the highest granularity of architecture, and it specifies the layers and high-level modules of the application, as well as how they interact with each other.

Architecture patterns help to specify the fundamental structure of an application.

Design patterns are more localized and solve a particular problem within the codebase. Examples include the factory pattern, singleton pattern, etc.

Anti-pattern

An *anti-pattern* describes a recurring solution to a problem that generates negative consequences. An anti-pattern is about applying a wrong solution to the right problem without having knowledge or analysis of either problem or applying patterns. The term was coined in 1995 by Andrew Koenig.

An anti-pattern from the developer's perspective is comprised of technical problems and solutions that are encountered. From an architecture perspective, it resolves problems in how systems are structured, and from a managerial perspective, an anti-pattern addresses common problems in software engineering.

In a nutshell, leveraging patterns is a valuable approach, but that doesn't mean you have to use a particular pattern. A common mistake by architects and designers is when they engineer a problem by using patterns. You need to understand the context and solution to the problem before applying the pattern in your context.

Cloud Native Data Management Pattern for Microservices

The following are cloud native data management patterns for microservices.

Event Sourcing Pattern

The event sourcing pattern is not new; Martin Fowler wrote about it in his book *Pattern of Enterprise Application Architecture* in 2002. The event sourcing pattern has not been used much, but it gains a lot of importance with the emergence of cloud native event-driven architecture. According to Fowler:

> *"Event Sourcing ensures that all changes to the application state are stored as a sequence of events. Not just query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes."*

In your application, when any activity occurs, it should be through an event. Without an event, the system may not function. The event can be anything such as clicking a button, clicking the back button, sending a request to an API, scheduling a job, transferring a payment, withdrawing a certain amount, purchasing a product, viewing reviews, etc. You need to use these events to track, audit, log, and restore marketing, etc. These events are difficult to store in the database by using create, read, update, and delete (CRUD) operations. You need a special type of data store to store all kinds of events.

The event sourcing pattern defines an approach to handling an operation on data that is driven by a sequence of events, and each of the event records is stored as a new record. The event-driven services publish the list of events with a description like an event name, time, date, user, etc., to the event store. It uses the event-centric approach to persist data. A business object is persisted in an event store with the sequence of state-changing events. Whenever an object's state changes, a new event happens to the sequence of events. The event store publishes events to the consumers, so the current state is derived from the event store.

Stream

The stream (the event store allows you to define and create as many streams as required for your domain) comprises a log of all events that have occurred during the state of an object. The event store can provide output as in a traditional database, and it provides much more such as time traveling through the system and root-cause analysis. The data in the event store is immutable and provides methods for audit logs.

Event Store

Figure 4-1 shows an overview of an event sourcing pattern, including storing events, externally consuming an application of an events, and querying an event for a specific state or current state. For example, the user performs various activities in an ecommerce application like logging in, searching for an item, selecting a brand, adding a brand or removing a brand, etc. These user activities are called *events*. The e-commerce application publishes all the user activities to the event stream by using event-driven systems like Kafka and stores them in the event store database like EventStoreDB.

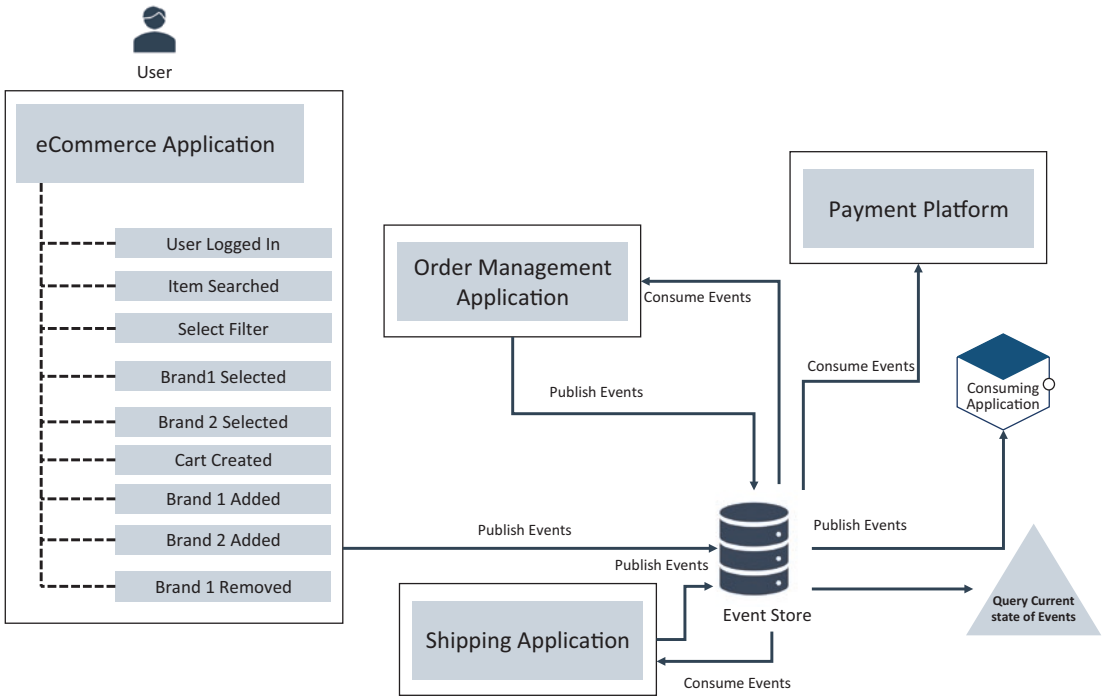

Figure 4-1. *Event source*

All the events are immutable and stored using an append-only operation; the event capture and event store are published seamlessly in the background without affecting the performance of an application or the user experience. All the events are simple event objects with characteristics such as timestamps, user IDs, etc.

The event sourcing enables the following:

- You can do a complete rebuild of an application or service state by rerunning the events from the event log on a system or service.

- Temporal queries can be used to determine the application state at any point in time. This can be achieved by initializing a blank state and rerunning all the events up to a particular point in time.
- Event replay can be used to repair a corrupted state of an application or service due to an incorrect event being received. This can be done by initializing a blank application or service state and replaying all the events while replacing the incorrect with the correct one.

There are multiple databases such as EventStoreDB, IBM DB2 Event Store, and NEventStore designed for storing events.

Every event has a name; in this example, the event name is the shopping cart `experienceUser1`. All the events are stored in a flat representation of an entity.

The following are the benefits of event sourcing:

- It enables accurate audit logging in an application.
- It makes it possible to implement temporal queries that determine the state of an entity at any point in time.
- It helps to implement the accountability required in the compliance.
- It is used to guarantee that all changes to a service resource state are based on events; it solves data consistency issues in a distributed architecture by atomically saving and publishing events and enabling event subscribers.

These are typical use cases of event sourcing:

- Enterprises in the finance industry such as banks, trading, and insurances are mandated to do regulation. Event sourcing helps to store audits and makes it easy to monitor the action of events.
- Up-to-date record-keeping in government agencies.
- User activity in the retail application for marketing.

These are some considerations necessary when using event sourcing:

- Event sourcing typically improves the performance of updates, but it takes time to construct an aggregated state. Using a snapshot may decrease the amount of time needed by taking a snapshot and replying to the events from that point on.

- The event structure may change over time. Therefore, the application or service should have a versioning strategy and be able to handle events with different versions.

Command and Query Responsibility Segregation Pattern

The *command and query responsibility segregation* (CQRS) pattern isolates the updated operation data from reading operation data. Implementing CQRS increases the system performance, provides low overhead on the command database, and provides a higher degree of security.

In a traditional architecture, as shown in Figure 4-2, usually the system uses a single model for both command and query operations.

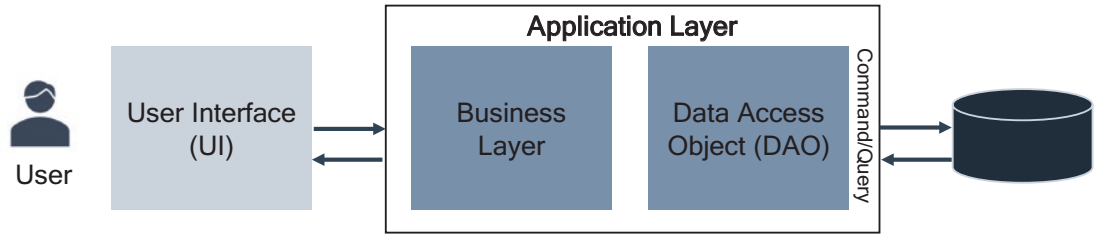

Figure 4-2. *Traditional architecture data operations*

In Figure 4-2, the application layer consists of business logic and DAOs and uses the same database for all CRUD operations. This type of architecture works well for basic CRUD operations. In more complex or legacy applications, this approach becomes unwieldy.

In one of my projects in early 2012, the client had a Temenos T24 core banking application, and it was very old and unable to support the business expansion; sometimes it failed to scale to meet the demand, which impacted the bank business. All the services from various channels like web, mobile, and branch were requesting both command and read operations from the same application with one monolithic database. Both read and write workloads are often asymmetrical, with very different performance and scale requirements.

The following are the drawbacks of this kind of architecture:

- Data conflicts can occur when both read and write operations are performed on the same sets of data.

- Performance degradation may occur due to the load on the data store and data access objects.
- Security becomes complex for security at rest and security in transit.

As our needs become more sophisticated, we are steadily moving away from that model. We need to look at the storage differently.

Approaching CQRS in two different ways, you can do the following:

- Segregate the application layer based on the command and query responsibility. The write request and read request are handled by two different objects.
- Split up the data storage, having separate reads and writes by using the event source.

As shown in Figure 4-3, the database is split up into application layers between the command and query model.

Application Layer Command and Query

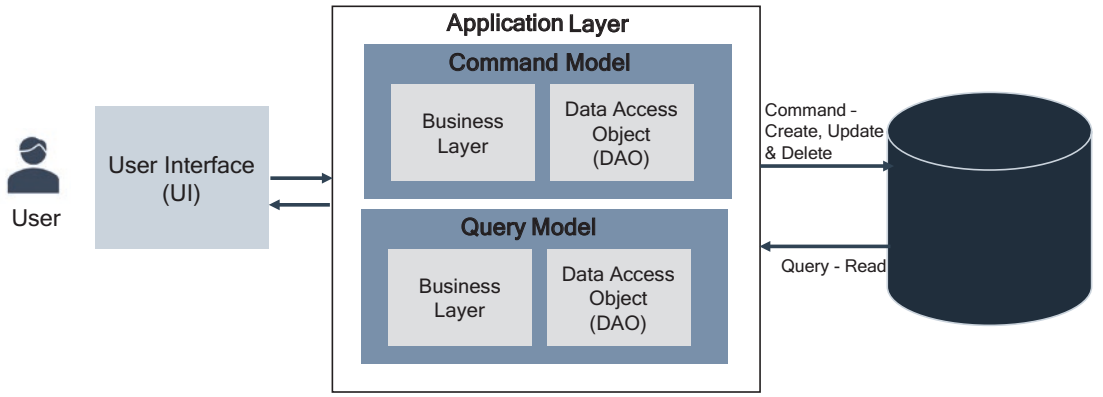

Figure 4-3. *Command and query model in the application layer*

Having separate models means different object models can be running in different processes and separate VMs or containers. There could be a separate request from the UI for commands (create, update and delete) and queries (read). This type of CQRS has both pros and cons, but it does not solve the industry problems. There is no change in the database load, and it may not improve the performance and security; however, complexity in the application layer is reduced.

Command and Query in the Database

As shown in Figure 4-4, we can split storage between the commands and queries by using event sourcing. These separate reads and writes go into different databases: the command database for creating, updating, and deleting and the querying database for read-only operations. The commands are usually task and transaction-based rather than data-centric. A query never modifies the data and returns a value object or DTO that does not encapsulate any domain knowledge.

For greater isolation, this model physically separates the read data from the write data. In this case, the read database can use its own data schema that is optimized for reading operations, and this type of architecture provides flexibility to choose the type of databases such as RDBMS or NoSQL, etc.

In the previous example of Temenos T24, we adopted the second model. We created the operational data store (ODS) from Temenos T24. We designed an event sourcing mechanism between the T24 database to the ODS database in near-real-time mode. From the enterprise service bus (ESB), we orchestrated all the read requests like statements, etc., to ODS with all the debit and credit orchestrated to the Temenos T24 system.

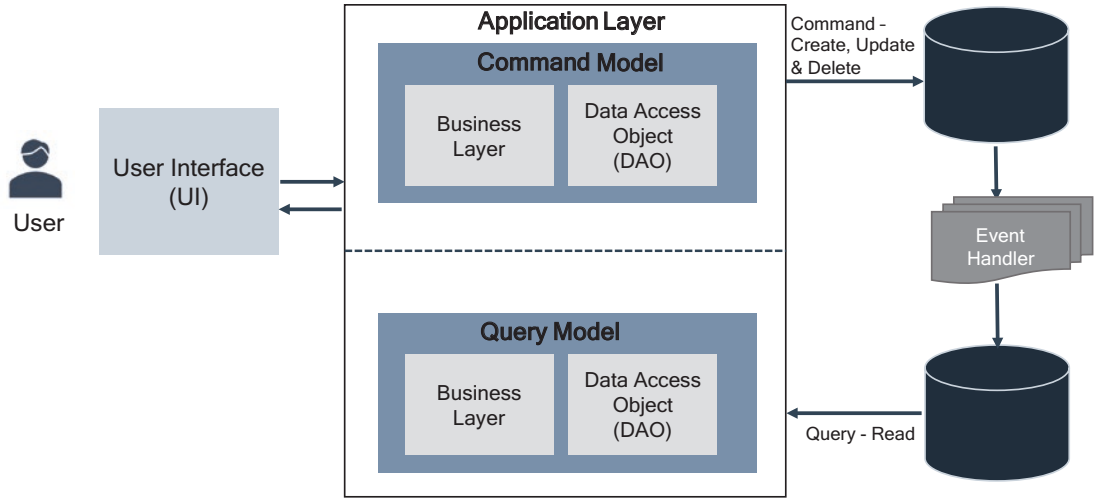

Figure 4-4. *Command and query in database*

As mentioned, the CQRS provides a separation of concerns. The command side is all about business or transactions and does not place much importance on queries or different materialized views over the data or optimized APIs from the nonrelational database, etc. On the other hand, the query side is all about read access. The main purpose is making queries fast and efficient. In many business systems, based on my experience, I can say approximately 70 percent of requests for read purpose from the users.

Separating the read side and the write side into separate models within a bounded context provides the ability to scale each one of them independently. The read data model could be de-normalized or could be a materialized view, which in turn increases the performance of the query execution.

The way event sourcing helps with CQRS is to have part of the application writing to an event store or stream topic. This is paired with an event handler that subscribes to the queue topic, transforms and cleanses the event, and writes the materialized view to read the store.

The following are the benefits of CQRS:

- CQRS allows the read and query workloads to scale independently.
- The query side can use a schema or materialized views that are optimized for queries, and the command side uses a schema that is optimized for updates.
- It is easier to manage security; that is, only command domains can perform writes on data.
- Segregating the query and command sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the command model, and the query model can be relatively simple.
- By storing materialized views in the query database, the application can avoid complex joins when querying.
- There are various options to use a query database, from RDBMS to NoSQL databases.
- The query database can provide data to the various analytical purposes.

The following are issues of CQRS:

- The idea of CQRS is simple, but the implementation is complex; you need very highly skilled resources.
- The best way to implement CQRS is to use event-driven architecture; you need to take care of data cleansing, message failures, etc.
- The query data may be stale due to replication time lag.

The following are the use cases for CQRS:

- You read more query-based use cases than command-based use cases, for example, social networking systems, retail bank applications, etc.
- In complex business logic, you want to simplify the understanding of the domain dividing problem into command and query.

Data Partitioning Pattern

A partition allows a table, index, or index-organized table to be subdivided into smaller chunks, where each chunk of such a database object is called a *partition*. Each partition has its name.

Data partitioning divides the data set and distributes the data over multiple servers or shards. Each shard is an independent database, and collectively, the shard makes up a single database. The portioning helps manageability, performance, high availability, security, operational flexibility, and scalability. This makes technologies an ideal fit for microservices data storage.

The *data partitioning pattern* addresses these issues of scale:

- High query rates exhausting the CPU capacity of the server
- Larger data sets exceeding the storage capacity of a single machine
- Working set sizes larger than the system's RAM, thus stressing the I/O capacity of disk drives

You can use the following strategies for database partitioning:

- *Horizontal partitioning (sharding)*: Each partition is a separate data store, but all partitions have the same schema. Each partition is known as shards and holds a subset of data.

- *Vertical partitioning*: Each partition holds a subset of the fields for items in the data store; the fields are divided according to how you access the data.
- *Functional partitioning*: Data is aggregated according to how it is used by each bounded context in the system.

You can combine multiple strategies in your application; for example, you apply horizontal partitioning for high availability and use a vertical partitioning strategy to store data based on data access.

The database, either RDBMS or NoSQL, provides different criteria to share the database.

- Range or interval partitioning
- List partitioning
- Round-robin partitioning
- Hash partitioning

Round-robin partitioning distributes the rows of a table among the nodes in a round-robin fashion. The range, list, hash partitioning, and an attribute called the *partitioning key* must be chosen among the table attributes. The partition of the table rows is based on the value of the partitioning key.

In range partitioning, a given range of values is assigned to a partition, and the data distributed among the nodes in such a way that each partition contains rows for which the partitioning key value lies within its range. The list strategy similar to the range, but a list of values is assigned one by one. The hash partitioning is based on the partition key and the hash values.

Horizontal Partitioning or Sharding

Applications in an enterprise require a database to store business data. When the business grows, the data size grows exponentially; at some point in time the database performs very badly with limited CPU, single storage capacity, performance, or query throughput. There should be a limit to increase the CPU, memory, etc. Therefore, you can't go beyond certain limitations.

Sharding is a common idea in database architectures. By sharing a table, you can store new chunks of data across multiple physical nodes to achieve horizontal scalability.

By horizontally scaling out, you can enable a flexible database design that increases the performance and high availability of data.

Figure 4-5 shows horizontal partitioning or sharding; in this example, user employee details are divided into two shards, HS1 and HS2, based on ID/key. Each shard holds the data for a contiguous range of shard keys. Sharding spreads the load over more nodes, which reduces contention and improves performance.

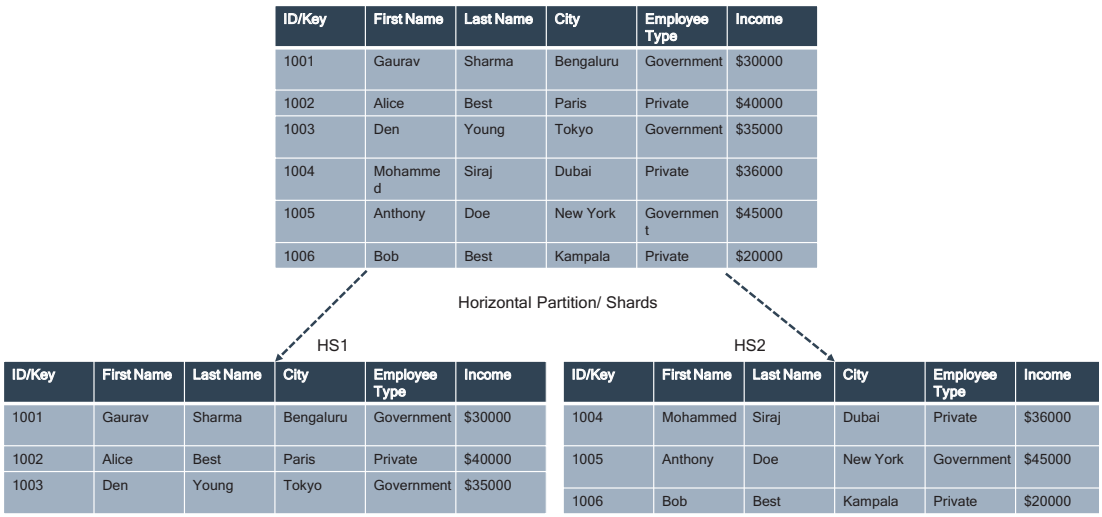

Figure 4-5. *Horizontal partition/shards*

The shards don't have to be the same size. It's more important to balance the number of requests. Some shards might be large, and other shards might be smaller; you can choose the key based on the access operation. The smaller size is more frequent and faster; the larger size is less frequent and slow.

Besides achieving the scalability and throughput of service level agreements (SLAs), sharding can potentially improve unplanned outages, and each node collaborates to make sure always available. Some database vendors use the master-slave architecture style for sharding.

Range Based or Interval Partitioning/Sharding

Range-based sharding separates the date based on ranges of the data value. Shard keys with range values are separated into a separate chunk. Each shard in an architecture preserves the same schema of the master database. Interval partitioning is an extension to range partitioning in which, beyond a point in time, partitions are defined by the interval.

Range-based shards support more efficient range queries. Given a range query on the shard key, the query router can easily determine which chunks overlap that range and route the query to only to those shards that contain these values in a chunk.

Each partitioning, as shown in Figure 4-6, creates a dedicated partition for certain values or value ranges in a table. In the previous example, the partition is based on the income. The income less than $35,000 is shard into one, and the income greater than $35,000 is in another shard.

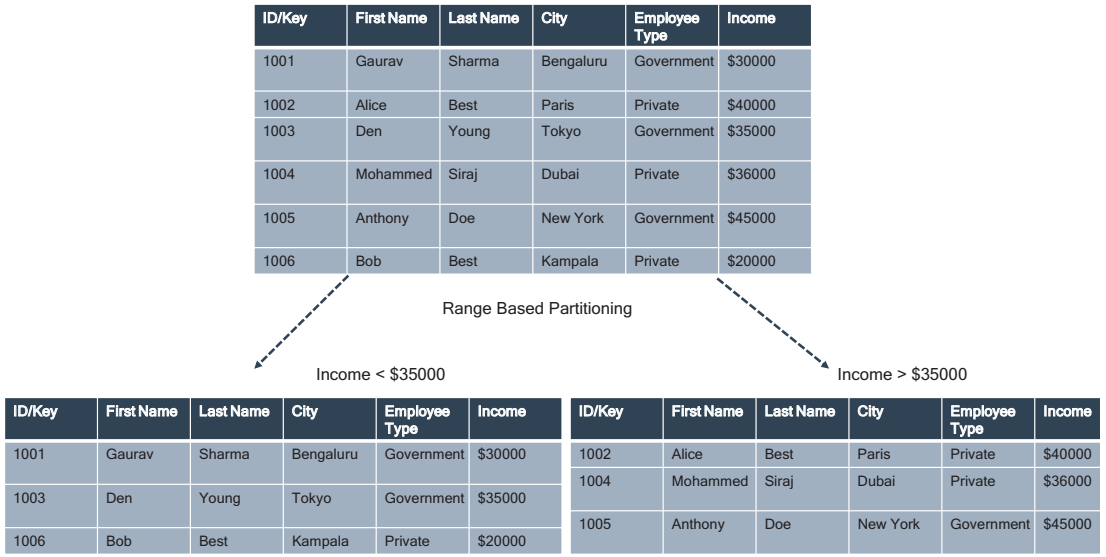

Figure 4-6. *Range-based sharding*

Partitions may be created or dropped as needed, and applications may choose to use range partitioning to manage data at a fine level of details.

The range partitioning specification usually takes a range of values to determine one partition, but it is also possible to define a partition for a single value. When one row is inserted or modified, the target partition is determined by the defined ranges. If a value does not fit one of these ranges, an error is raised. To prevent this kind of error, create another partition to accommodate these kinds of data that are not part of the range.

The range-based partitioning can result in the uneven distribution of data, which may negate some of the benefits of sharding.

Consider the range or interval partition in the following cases:

- Large tables are frequently scanned by a range predicate on a good partitioning column.
- You want to maintain a rolling window of data.
- You cannot complete any housekeeping activity on large tables in a required time, but you can divide them into smaller logical chunks based on the partition range column.

Hash Partitioning/Sharding

Hash partitioning is a partitioning technique where a hash key is used to distribute rows evenly across the different partitions.

Hashing is the process of converting a given key into another value and refers to the conversion of a column's primary key value to a database page number on which the rows will be stored.

Hash sharding takes a shard key's value and generates a hash value from it. The hash value is then used to determine in which shard the data should reside. With a uniform hashing algorithm such as Ketama (it is an implementation of a consistent hashing algorithm, meaning you can add or remove servers from the pool without causing a complete remap of all keys), the data with close shard keys is unlikely to be placed on the same shard.

In Figure 4-7, the table is partitioned by using the hash function on the ID/key column.

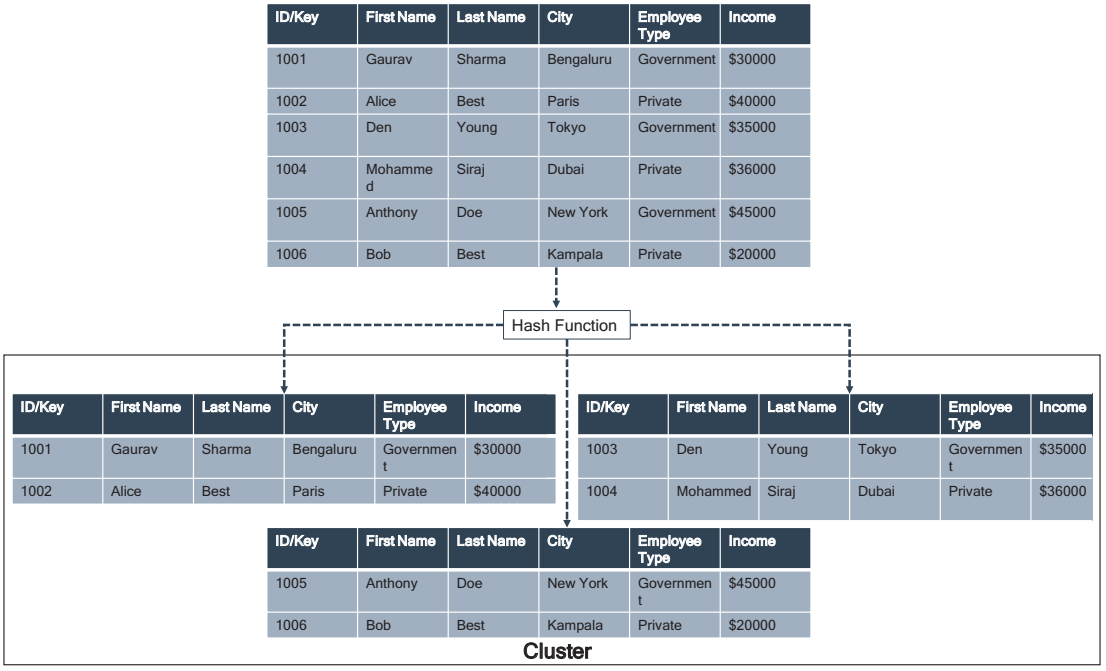

Figure 4-7. *Hash partitioning/sharding*

Partitioning by hash is used primarily to ensure an even distribution of data among a predetermined number of partitions and is focused on data distribution instead of data grouping.

As a rule of thumb, hash partitioning can be used in the following cases:

- To enable partial or full parallel partition-wise joins with likely equalized partitions
- To distribute data evenly among the nodes
- To randomly distribute data to avoid I/O bottlenecks

List Partition

The list partitioning concept is like range partitioning. As detailed, the range partitioning is done by assigning a range of values to each partition. In the list partition, we assign a set of values to each partition.

You should use list partitioning when you want to specifically map rows to partitions based on discrete values. For example, all users in Asia and Europe are stored in one partition, and users in America and Africa are stored in different partitions.

List partitioning is useful when we have a column that can contain only a limited set of values; even range partitioning can be used, but list partition allows you to equally distribute the rows by assigning a proper set of values to each partition.

Round-Robin Partitioning

The round-robin portioning is used to achieve an equal distribution of rows to partitions. With this technique, the new rows are assigned to partitions on a rotation basis. There is no partition key; rows are distributed randomly across all partitions, and therefore load balancing is achieved.

Vertical Partitioning

Vertical partitioning splits the data vertically to reduce I/O and the performance associated with fetching items that are frequently accessed.

In this example, as shown in Figure 4-8, different attributes of employees are stored in different partitions. VS1 holds data that is accessed more frequently, and, in another partition, VS2 holds the employee type and income, which are accessed intermittently.

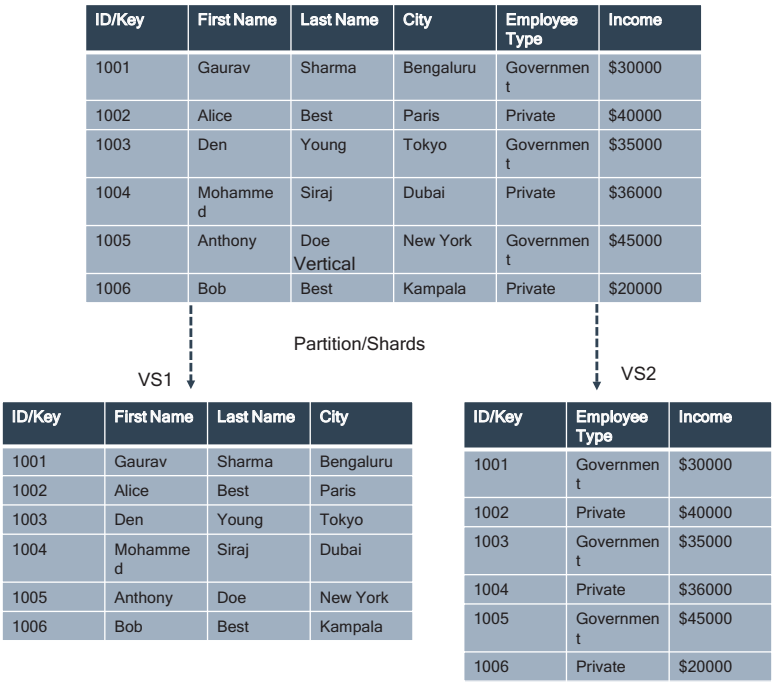

Figure 4-8. *Vertical partitioning*

The following are the benefits of vertical partitioning:

- Slow-access data can be separated from more dynamic data.
- Sensitive data can be stored in a separate partition with additional security controls.
- This strategy can reduce the amount of concurrent access.

Data Replication

Replication is the continuous copying of data changes from the primary database to the secondary database. The two databases are generally located in different servers, resulting in a load balancing framework by distributing various database queries and providing a failover capability. This kind of distribution satisfies the failover and fault tolerance characteristics.

Replication can serve many nonfunctional requirements such as the following:

- *Scalability*: Handling higher query throughput than a single machine can handle
- *High availability*: Keeping the system running even when one or more nodes go down
- *Disconnected operations*: Allowing an application to continue working when there is a network problem
- *Latency*: Placing data geographically closer to users so that users can interact with the data faster

In some cases, replication can provide increased read capacity as the client can send read operations to different servers. Maintaining copies of data in different nodes and different data centers can increase data locality and availability of the distributed application. You can also maintain additional copies of dedicated purposes, such as disaster recovery, reporting, or backup.

There are two types of replications:

- Leader-based or leader-followers replication
- Quorum-based replication

Leader-Based or Leader-Followers Replication

In leader-based replication, one replica is designed as a leader while another replica is a follower. Clients always send their write queries to the leader. Leaders write the data to its local storage first and then send the data change to its followers. When the client wants to read from the database, it can query either the leader or the follower. The leader is responsible for making decisions on behalf of the entire cluster and propagating the decisions to all the nodes in a cluster.

In Figure 4-9, there is a single leader with asynchronous and synchronous replication. The user sends an update request to update the first name to the leader. The leader updates first and then sends a synchronous request to Follower 1 and Follower 2. After the leader receives an OK response from Follower 1 and Follower 2, the leader sends an OK status to the user for a successful update. The leader replicates asynchronously to Follower 3, but the leader doesn't wait to receive any OK from Follower 3.

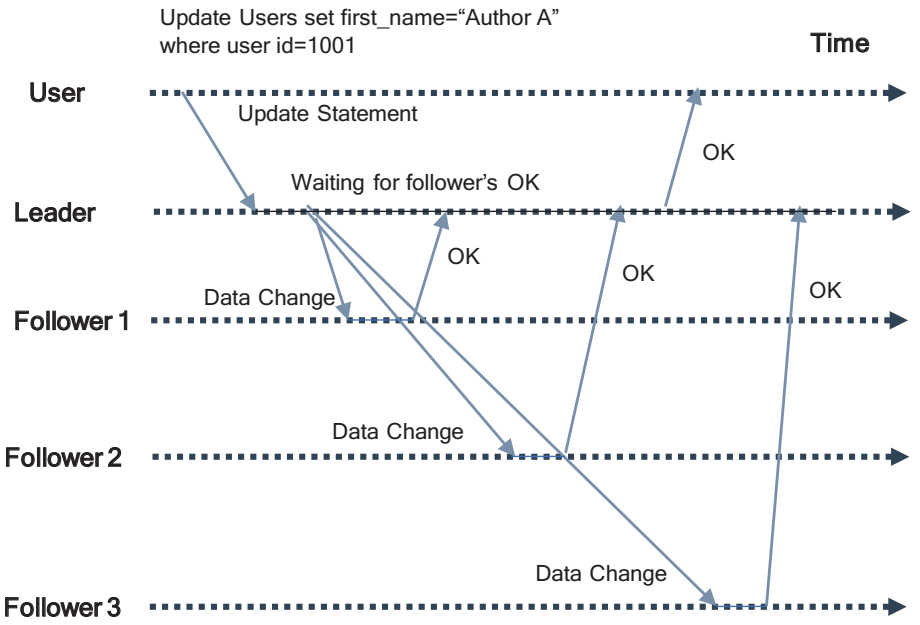

Figure 4-9. *Single leader with two synchronous and one asynchronous replication*

In a multileader example, there are two data centers (DCs) or clusters across geographies to provide high availability or latency to various users. In this model, you need to have two separate sets of leaders and followers in each cluster or DC and replicate each as mentioned in Figure 4-10; however, both need to synchronize and resolve any conflicts or inconsistencies. In this case, both leaders talk to each other over a conflict resolution object to sync each other.

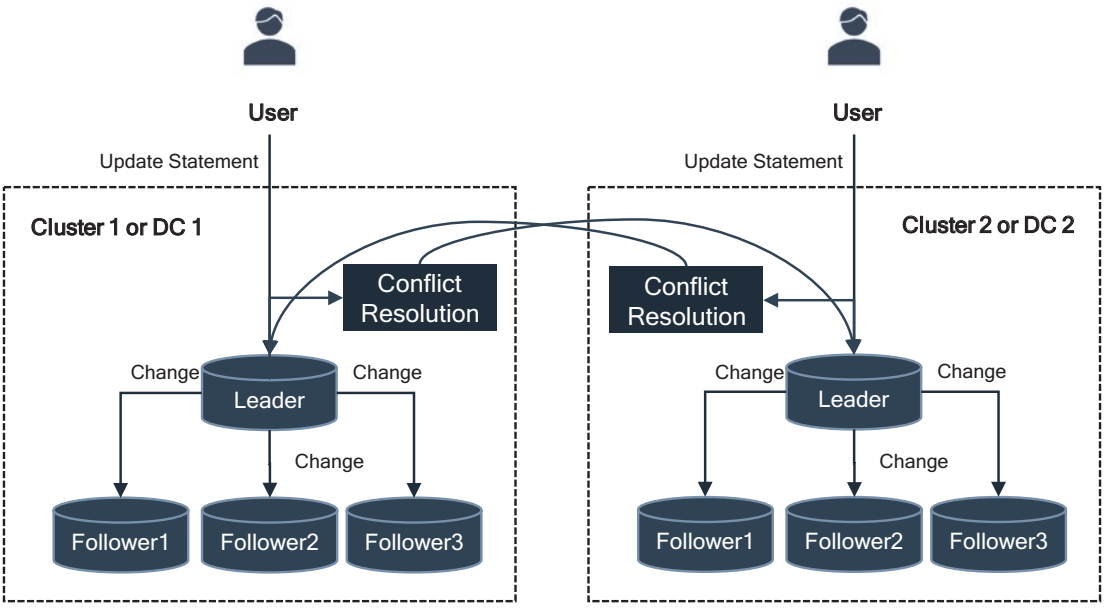

Figure 4-10. *Multileader-based replication across clusters or data centers*

Every server in a node or cluster or DC at startup looks for an existing leader. If no leader is found, it triggers leader selection. The leader in each cluster is a must; without the leader, there is no acceptance of any request from the user. Only the leader handles the client request, not the followers. If a request is sent to a follower, then the follower sends a request to the leader to act.

How are the leaders selected?

An election will be conducted to select a leader. If the existing leader is not available, then the database cluster uses the Raft consensus algorithm to choose the leader.

Raft is designed to select a leader by ensuring each node in the cluster agrees upon the same series of state transitions.

The Raft protocol was developed by Diego Ongaro and John Oosterhout (Stanford University) in 2014. Raft was designed for better understandability of how consensus can be achieved. The consensus is a method to involve multiple servers agreeing on one value; once they decide on a value, that decision is final.

According to Raft, each node in a replicated server cluster can stay in either leader, follower, or candidate. At the time of election to choose the leader, the servers can ask other servers to vote; hence, they are called candidates when they have requested votes.

Figure 4-11 shows the step-by-step process of how servers apply Raft consensus to choose a leader. A leader election is started by a candidate server; it starts the election by increasing the term counter, voting itself as a new leader, and sending a message to all other nodes. Here, Follower 3 is a candidate and sends messages to Follower 2 and Follower 1. A server will vote only once per term, on a first-come, first serve basis. If a candidate receives a majority vote, then it becomes a new leader. Here Follower 3 receives a maximum vote and then is selected as a new leader. Raft uses a randomized election timeout to ensure that split-vote problems are resolved quickly.

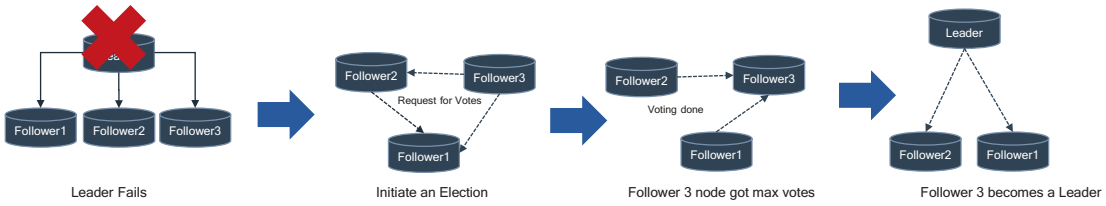

Figure 4-11. *Election process to choose a leader*

The high availability of leaders is achieved using a Failover pattern. A timeout with heartbeats is used to detect whether the replica is dead or alive. When one or more followers fall behind a leader by a certain configurable unit, it is called a replication lag and can cause strange side effects. Various consistency models can be used for deciding how an application should behave under replication lag.

Quorum-Based Replication

A cluster quorum disk is the storage medium on which the configuration database is stored for a cluster computing network. The cluster configuration database, also called a *quorum*, informs the cluster which physical server(s) should be active at any given time. The quorum disk comprises a shared block device that allows concurrent read-write access by all nodes in a cluster.

In this replication, the client is responsible for copying the data to multiple replicas. The nodes do not actively copy data among each other. The size of the replica group doesn't change even when some replicas are down. The client sends both read and write to multiple replicas. A cluster agrees that it received an update when a majority of the nodes in the cluster have to acknowledge the update. This number is called a *quorum*. The number of quorums will be decided by the following formula:

No of quorum = n/2+1

If you have five nodes in a cluster, then n=5 nodes, and then 5/2+1= 3 (round off). If you have a cluster of five nodes, you need a quorum of three.

In the quorum, how to decide how many failures can be tolerated equals the size of the cluster minus quorum. If you have five nodes and three quorums, then node-quorum = failure, 5-3=2. A cluster of five nodes can tolerate two of them failing.

You can use this formula to calculate nodes in a cluster:

2f+1

f=failure (2*2+1=5)

Figure 4-12 depicts a quorum-based replication pattern that shows quorum write, quorum read, and read repair after a node (replica 3) outage. In that case, it is sufficient to acknowledge the write. Thus, when the user receives two OK responses from the cluster, this satisfies the n/2+1 = 3/2+1=2.

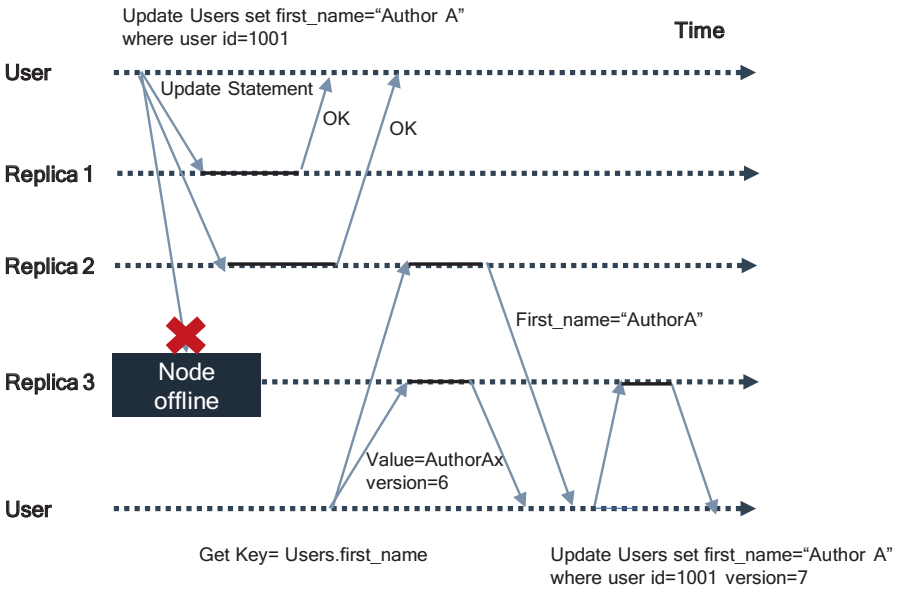

Figure 4-12. *Quorum-based replication*

If there are *n* replicas, every write must be confirmed by *w* nodes to be considered successful, and we must query at least *r* nodes for each read. The quorum allows the system to tolerate unavailable nodes as follows:

- If $w < n$, we can still process writes if a node is unavailable.
- If $r < n$, we can still process reads if a node is unavailable.
- With $n=3$, $w=2$, $r=2$, you can tolerate one available node.
- With $n=5$, $w=3$, $r=3$, you can tolerate two unavailable nodes.

The cluster can function only if the majority of servers are up and running. You need to consider the following:

- *The throughput of a write operation*: Every time data is written to the cluster, it needs to be copied to multiple servers. Every node in a cluster adds overhead to complete all the writes. The latency of data is a directly proportionate number of servers forming the quorum; therefore, if you increase the number of nodes, then it impacts the throughput.
- *The number of failures that need to be tolerated*: The number of failures tolerated depends on several nodes in a cluster; adding one more node doesn't give more fault tolerance. For example, 100 developers cannot complete the entire project in 1 day instead of 5 developers in 20 days.

Even if a client always performs quorum reads and writes, conflicts are likely to occur.

- Two clients may write to the same key at the same time (use concurrency control to manage this).
- If an error occurs during writing or if a node fails and needs to be re-created, a write may be present on fewer than *w* replicas.

The result is that replicas disagree about what a particular value in the database should be. In such a case, the application must be handled by using a concurrency algorithm.

Martin Fowler wrote in his blog about how to choose the optimal servers in a cluster, as shown in Figure 4-13. He says the decision is based on the number of tolerated failures and approximate impact on the throughput. The throughput column shows the approximate relative throughput to highlight how the throughput degrades with the number of servers. The number will vary from system to system. For further reading, refer to Raft Thesis and Zookeeper's paper (https://raft.github.io/).

Number of Servers	Quorum	Number of Tolerated Failures	Representative Throughput
1	1	0	100
2	2	0	85
3	2	1	82
4	3	1	57
5	3	2	48
6	4	2	41
7	5	3	36

Figure 4-13. *Deciding on the number of servers in a cluster*

In the quorum, write and read are not sufficient, as some failure scenarios can cause clients to see data inconsistency. Each server does not have any visibility of data on another server. The inconsistency can be resolved only when data is read from multiple nodes in a cluster.

Cloud Native API Management Patterns for Microservices

These are patterns for microservices.

Idempotent Service Operation

There are idempotent operations on HTTP methods. If a REST service is idempotent, the consumer of an API can make that same call repeatedly while producing the same result; in other words, making multiple identical requests has the same effect as making a single request.

When you design REST APIs, you must take into consideration that consumers can make mistakes. The consumer application can write client code in such a way that there can be duplicate requests coming to the API. In distributed architecture, failure may occur when invoking service. A lost request should be retired, but a lost response may cause unintended side effects if retired automatically.

These duplicate requests may be unintentional or intentional, you must design fault-tolerant APIs in such a way that the duplicate requests do not leave the system unstable.

The *idempotent service pattern* is used to provide a guarantee that service invocations are safe to repeat in the case of failures that could lead to a response message being lost. The idempotent requests can be processed multiple times without side effects.

When designing APIs, you must follow REST principles such as stateless, uniform interface, code on demand, etc. You will have automatically idempotent REST APIs for HTTP methods.

- GET, PUT, DELETE, HEAD, OPTIONS, and TRACE are idempotent.
- POST is not idempotent.

The GET, HEAD, OPTIONS, and TRACE methods should not have any significance when taking an action other than retrieval. These methods ought to be called “safe” methods. The POST, PUT, and DELETE are represented as “unsafe” requests and require special handling in the case of exceptional situation (e.g., state reconciliation).

POST is an HTTP method used to send data to a server to create/update data from the consumer. When you invoke POST requests many times, you will use the same resources on a server, so POST is not idempotent.

GET, HEAD, OPTIONS, and TRACE are used for requesting resources from a backend application; therefore, these methods never change a resource state on a backend application. They are purely for retrieving application data, so invoking multiple requests will not affect data on a server, so these methods are idempotent.

The PUT method is used to update a resource in a back-end application. If you call PUT multiple times, you are updating the existing record or overwriting the record. Therefore, it not changing any records; hence, PUT is idempotent.

The DELETE method is used to delete a record in a back-end application. The first request deletes a record in an application, and then the consumer will receive an HTTP response 200 (OK) or 204 (No Content) if the consumer sends the same request again and again, the DELETE method tries to find a record that was deleted earlier, or the HTTP returns 404 (Not Found) message. Here only the response is different, but there is no change in record status; hence, the DELETE method is idempotent.

Optimistic Concurrency Control in API

Concurrency control means an object will ensure the correct results are received for concurrent operations. Concurrency is required to avoid conflict between concurrent requests. There are two kinds of concurrency control.

- Optimistic concurrency control
- Pessimistic concurrency control

The *optimistic* concurrency control allows concurrency conflicts to happen. If they happen, the control makes sure the previously requested data is not changed. It doesn't lock any records to ensure the record wasn't changed in the time between the select and submit operations.

The *pessimistic* concurrency control blocks an operation of a transaction and does not allow another request to access a particular API or data.

Concurrency locking is not new; you, me, and everyone experienced concurrency issues in RDBMS, but how does the concurrency control impact our APIs? What happens when two users update the same record at the same time? Will you send any error messages? What response code will you use? In the REST API, several consumers interact with a single resource, each consumer holding a copy of the state. Let's imagine author A (you) and author B (me) are editing content on the same topic at the same time. You edit the content faster than me, and you submit the changes. When I complete my editing, I submit the changes, but I overwrite your changes. To avoid this type of conflict, you need a concurrency mechanism in APIs.

Conflict mostly occurs in response to the HTTP PUT method request as this method is used for the update operation. You need to use the concurrency control designed into the HTTP protocol to protect the integrity of your data.

An entity tag, specified by the ETag HTTP header, is an opaque token that the server associates with the particular state of a resource. It is an optional header in the HTTP request, and it is kind of like a version stamp for a resource. Whenever the resource changes, the ETag should change accordingly.

The API consumer and provider use the ETag value to determine whether a request to a resource is up-to-date by comparing the value of the ETag header on an incoming request to the value of the ETag header present on the server. If a value matches, then the consumer will get up-to-date information; if not matched, the consumer should refresh the request to receive the updated details.

With the previous example of content editing using an HTTP, imagine you want to modify some content in a server. What will you do? You use GET requests to fetch content and make a local modification and then issue a PUT request to update on the server.

With a single client, the interaction is happening without any issues. The concurrency is required when two or more requests try to modify same content, as shown in Figure 4-14.

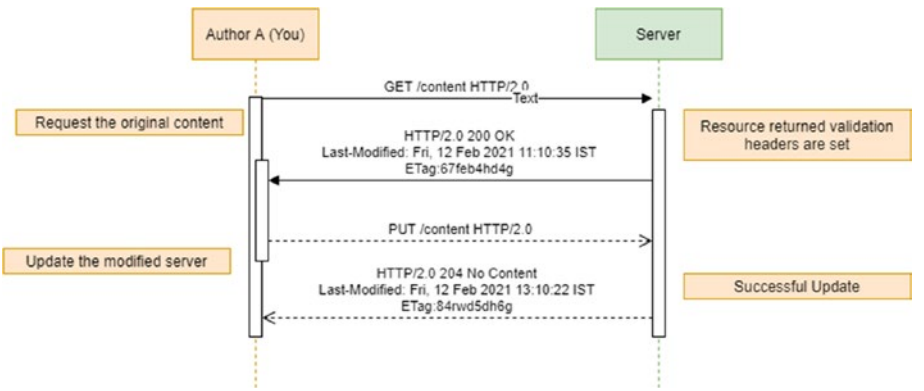

Figure 4-14. *Single request*

Say author A gets the content and modifies it locally, and author B requests the same content and modifies it locally. If both authors attempt to put their modifications back on the server, the modification of author A will be lost when author B's PUT overwrites, as shown in Figure 4-15. In this situation, both the authors are aware of this situation.

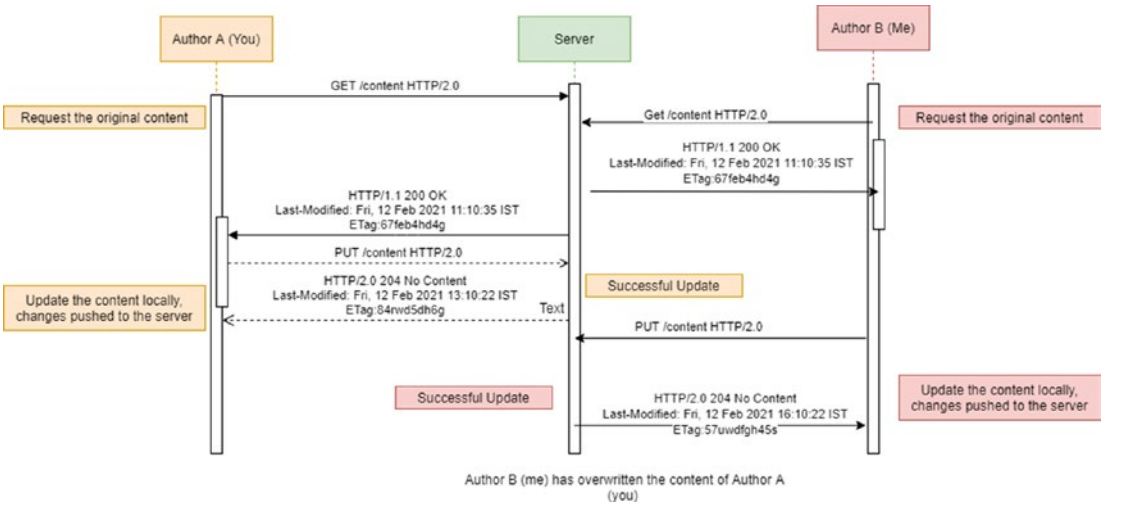

Figure 4-15. *Concurrency condition: author A updates lost*

To avoid these concurrency issues, as shown in Figure 4-16, you need to use the ETag header with the conditional request If-Match. This allows you to implement optimistic locking to avoid conflicts. With optimistic locking, each author is able to edit the content, and the author notifies with conflicts in a content.

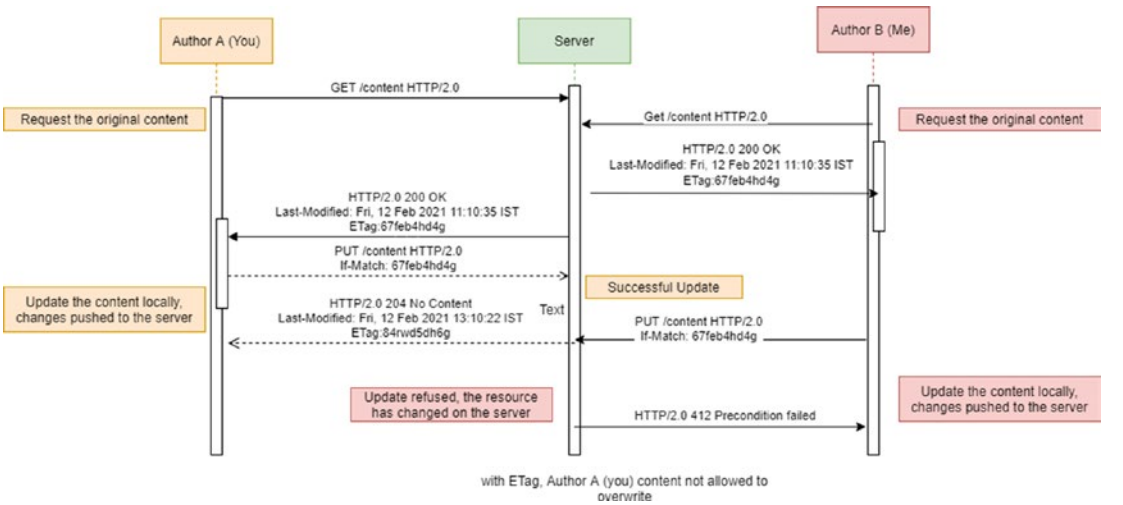

Figure 4-16. *Optimistic locking with ETag*

Figure 4-16 shows an implementation of optimistic locking by using an ETag and the If-Match header. If the ETag header does not match the value of the content on the server, the server rejects the change with 412 Precondition Failed. Author B is notified of the conflict and can try again after updating the local copy.

You need to make sure that when you are using optimistic locking, this condition is not suitable for everything, such as if both author A and author B update their photo at the same time on the same album. This is a feature, not a conflict.

Circuit Breaker

The circuit breaker pattern is used to check the availability of an external service, detect failures, and prevent them from happening constantly. In a distributed cloud native application, calls to remote resources and services can fail due to transient faults such as slow network connections and slow execution by microservices. These faults correct themselves after some time, and cloud and cloud native applications should handle this kind of situation.

For example, your mobile application needs to retrieve data from microservices hosted in the cloud platform. During business hours, your application might access 100 transactions per second (100 tps); in this case, your microservice is not available due to various faults such as network, slowness, etc. In this scenario, your microservice should be able to handle quickly and gracefully without waiting for each service request to time out.

The circuit breaker pattern was popularized by Michael T. Nygard in his book *Release It!*, which can prevent an application from repeatedly trying to execute an operation that's likely to fail. This allows it to continue without waiting for the fault to be fixed or wasting CPU cycles while it determines that the fault is long-lasting.

As illustrated in Figure 4-17, the idea of the circuit breaker is to wrap a protected function call in a circuit breaker object, which monitors failure. Once the failure reaches a certain threshold, the circuit breaker trips, and all calls to the circuit breaker return with an error, which means the circuit breaker acts as a proxy for operations that could potentially fail.

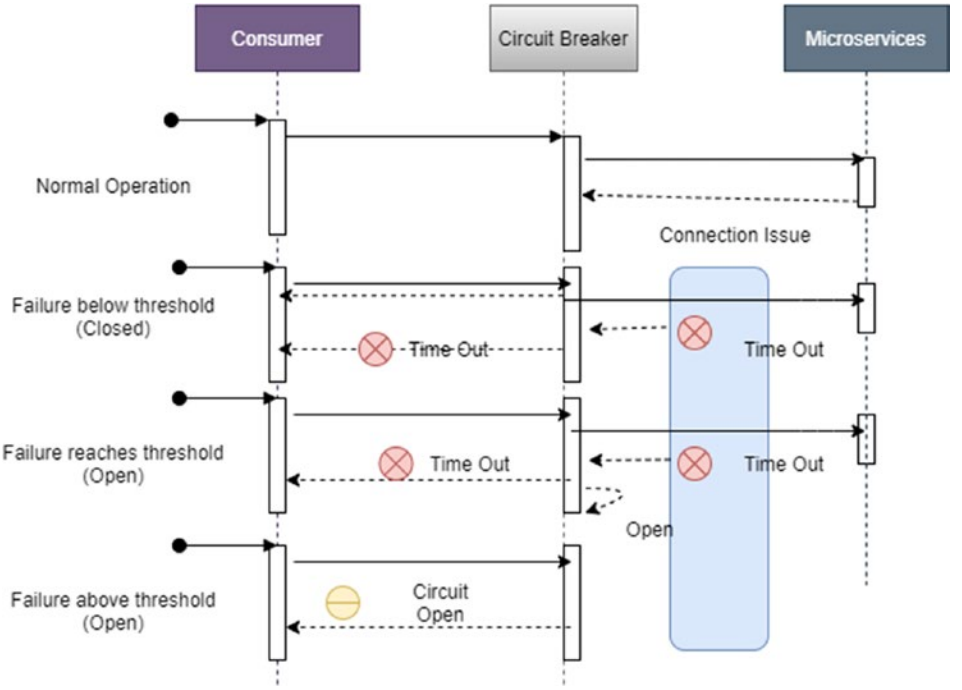

Figure 4-17. *Circuit breaker sequence diagram*

As shown in Figure 4-18, the circuit breaker pattern is implemented as a state machine that mimics the state of an electric circuit breaker.

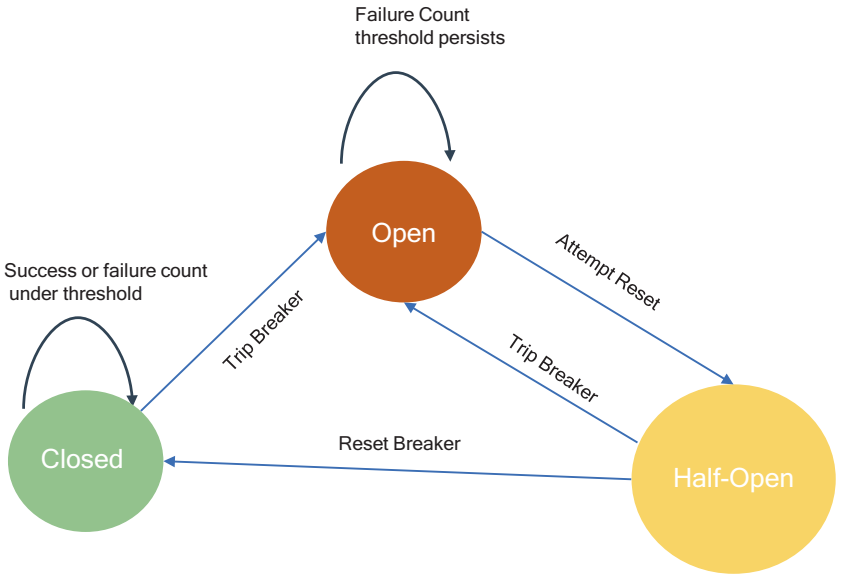

Figure 4-18. *Circuit breaker pattern states*

Closed: The operation executes normally. The circuit breaker maintains a count of the recent failures. If the number of recent failures exceeds a threshold within a given period, the proxy is placed into the open state. At this point, the proxy starts a timeout timer, and when this timer expires, the proxy is placed into the half-open state.

Open: The request from the application fails immediately, and an exception is returned to the application.

A half-open state is used to prevent a recovering service from being hit with a large number of requests. As a service recovers, it may be able to support a limited volume of requests until the recovery is complete, but while recovery is in progress a flood of work may cause the service to time out or fail again.

The circuit breaker pattern should be implemented asynchronously to offload the logic to detect failures from the logic to execute the actual operation. The implementation requires some form of persistence (to record the number of successful and unsuccessful operation execution). There are various tools are available in the industry like Istio, Hashicorp Consul, etc., to support the circuit breaker implementation.

Use this pattern in the following case:

- To prevent an application from attempting to invoke a remote service or access a shared resource if this operation is highly likely to fail

This pattern might not be suitable for the following:

- For handling access to local private resources in an application, such as in-memory data structure. In this environment, using a circuit breaker would simply add overhead to your application.
- As a substitute for handling exceptions in the business logic of your applications.

Service Discovery

The API gateway needs to know the location (IP address and port) for each microservice with which it communicates. In a traditional architecture and system, you could probably hardwire the location because this application is not dynamic. In a cloud native modern application like microservices, finding the needed location is a nontrivial problem.

Infrastructure services such as MQs usually have a static physical location that can be specified by using server OS environment variables. However, in cloud native microservices, determining the location of an application is not easy.

Application services are assigned a location and set of instances of service changes dynamically because of autoscaling, container orchestration, etc. Consequently, the API gateway needs to use the system's service discovery mechanism either in server-side discovery or in client-side discovery.

The service registry is a key part of discovery. It is a database containing the network locations of service instances. This is a single point of failure and therefore should be highly available and up-to-date.

Client-Side Discovery Pattern

When using this pattern, the client is responsible for determining the network locations of available service instances and load balancing requests across them, as shown in Figure 4-19. The client queries a service registry, which stores available service instances. The client then uses a load balancing algorithm to select one of the available service instances and makes a request.

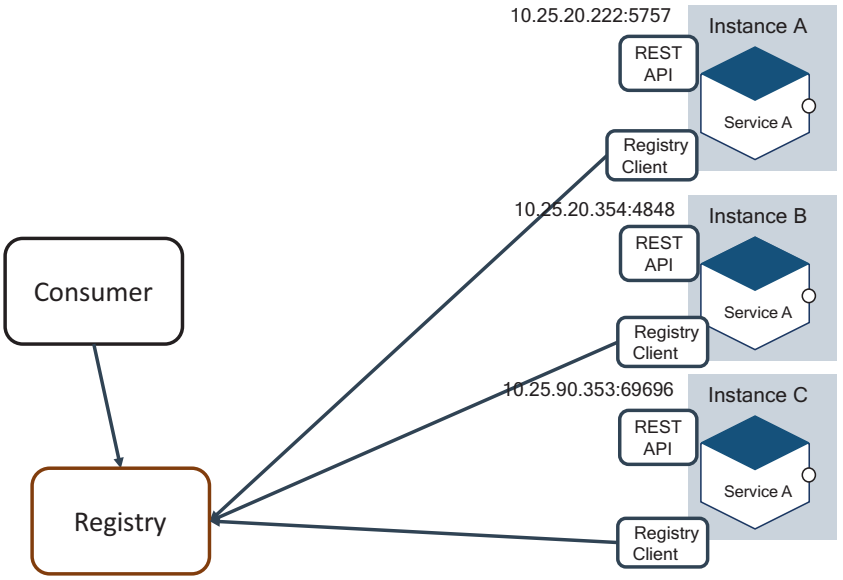

Figure 4-19. *Client-side registry*

The network location of a service instance is registered with the registry when it starts and removes when it terminates. The registration of services is refreshed regularly by using a heartbeat mechanism.

The client-side registry pattern has a few benefits and drawbacks. The following are the benefits:

- It is relatively simple, without additional components required except for the registry.
- The client can make intelligent, application-specific load balancing decisions such as using hashing consistently.

The drawbacks are as follows:

- The client is coupled with the service registry and potentially complicated with load balancing.
- You must implement client-side service discovery logic for each programming language and framework used by your service clients.

Server-Side Discovery Pattern

The client request to a service via a load balancer. The load balancer queries the registry and routes each request to an available service instance, as shown in Figure 4-20.

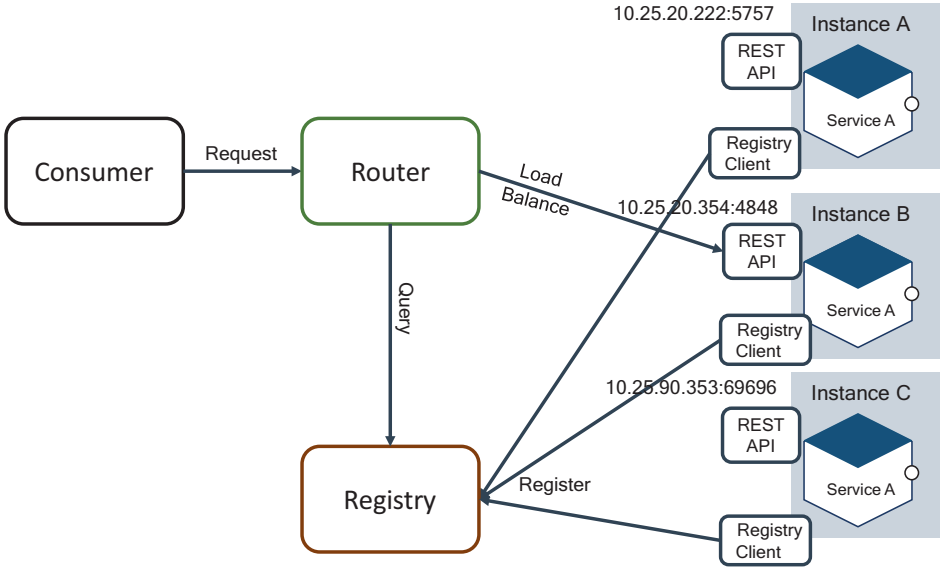

Figure 4-20. *Server-side registry*

As with client-side discovery, service instances are registered with the service registry.

The server-side pattern has several benefits and drawbacks. The benefits are as follows:

- Compared to client-side discovery, the client does not need to know how to deal with discovery. The discovery is abstracted away from the client. Instead, a client simply requests the router.
- This eliminates discovery logic for each programming language and framework used by your service consumers.
- Some cloud environments provide this functionality like cloud ELBs.

The drawbacks are as follows:

- Unless it is part of the cloud environment, the router is another system component that must be installed and configured. It will also need to be replicated for availability and capacity.
- More network hops are required than the client-side discovery.

Service Versioning

There are basic principles for designing an API exposed by microservices, the first of which is enforcing strong contracts. A microservice provides a versioned, well-defined contract to its clients and other microservices, and each service must not break it until it's determined no other microservices relies on it. Figure 4-21 illustrates the relationship between service producers like microservices and consumers such as web applications or mobile applications. The service producer registers all its services in service registries like Netflix Eureka and consumer contacts in the registry for service discovery, and later it connects to microservices for consumption of the service data.

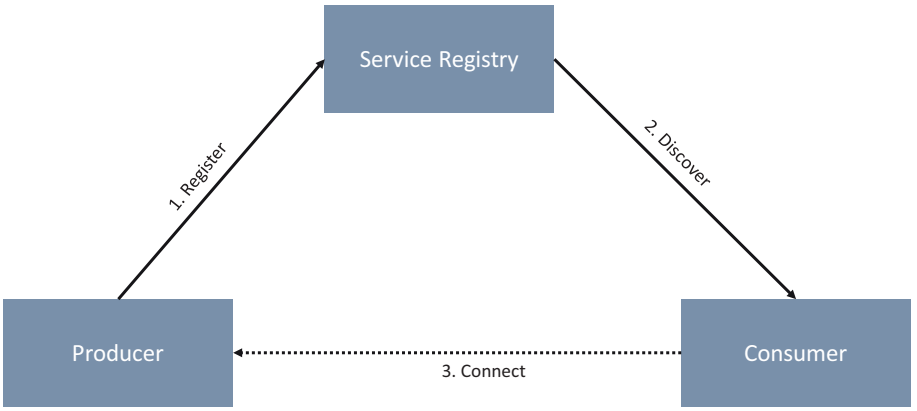

Figure 4-21. *Service registry*

There are two options for versioning the exposed API of a microservice. If you need to provide additional information on an HTTP method like a GET or POST or PUT operation, then the change is unlikely to be backward compatible. In that case, you need to look at ways of handling this problem.

The following are the two most common ways of handling versioning:

- Versioning in the URI
- Versioning in the header

URI Versioning

URI versioning is when you change the URI of the resource to contain version information, for example, `/customer/v1.1/{id}`. URI versioning gives you the ability to version an entire resource hierarchy. If you model a version like this, it enables resources for the automated navigation or discovery of resources.

The drawback of URI versioning is you need to change the resource name and location. This introduces a complex creation of URI aliases that make it difficult to track the production version, and it may break existing software links that do not include version information.

Here are two ways you can version in URI versioning:

- *Versioning at multiple hierarchy nodes (complicated)*: `-/customer/version/2/account/version/2`
- *Versioning as a query parameter*: `/customer?version=2`

There are multiple options to deal with this problem.

- Copy your old data into a new `V2` database and keep the two entirely separate.
- Update your schema in place and add code to `V1` to handle the new schema.

Header Versioning

In the header versioning, you need to include version information in a special header of each request and response. For example, say you need to add a header with `x-version:3.1`. In this approach, the resource name and location remain unchanged throughout your URL hierarchy, so you want to create URI aliases.

The drawback of the header versioning approach is that information can't be readily encoded into software links. It works only with custom clients that know how to encode the special header.

Based on the URI and header versioning mechanism, you can consider either forward- or backward-compatible methods.

- *Forward compatibility*: When developing a service, you make sure that this version will be compatible with future versions and won't be impacted when other services are updated (e.g., a new feature added). Achieving forward compatibility is a complex task since you have to deal with several unknown or unexpected features. The most common concept is to simply ignore unrecognized elements.
- *Backward compatibility*: The new version of a service is compatible with today's version, so existing clients can start using this new version as if there was no change. It can be verified by thoroughly testing the new version with old data sets.

There are a few different types of changes that are important for service versioning such as the major release, minor release, new capability, bug fix, etc.

You need a version number for each release. When a client requests a certain service, a service proxy forwards the request to a version of the service that is compatible with the version of the client. Therefore, all the clients have a single endpoint. While implementing the versioning, the governance of the API is of utmost importance to avoid any software breaks.

Cloud Native Event-Driven Patterns for Microservices

These are the event-driven patterns.

Asynchronous Nonblocking I/O

Compared to all the other characteristics of infrastructure such as CPU, memory, and disk, the network is slow.

- A high-end modern system is capable of moving data between the CPU and main memory at the speed of around 6 GB per second.
- A common local area network (LAN) of I/O is about 12.5 MB per second.
- Today's hard disk provides a lot of storage and transfer speeds of around 50–60MB per second.
- A CPU can execute approximately more than a billion instructions per second.

The I/O performance has not increased as quickly as CPU and memory performance, partially due to neglect and physical limitation. In a cloud native architecture, all system tasks are I/O-bound, and the I/O speed often limits the overall system performance.

According to Amdahl's law, as shown in Figure 4-22, improved CPU performance alone has a limited effect on the overall system speed. This law gives a theoretical speedup in the latency of the execution of a task at fixed workloads that can be expected of a system whose resources are improved.

$$\text{Execution time after improvement} = \frac{\text{Time affected by improvement}}{\text{Amount of Improvement}} + \text{Time unaffected by improvement}$$

Figure 4-22. *Amdahl's law*

Currently, the network is ubiquitous; it is the distribution of communications infrastructure and wireless technologies throughout the environment to enable continuous improvement. In the 5G world, network slicing enables the multiplexing of virtualized and independent logical networks on the same physical network infrastructure. Once the 5G network is rolled out, the speed of the network increases tenfold. Even though the network speed increases tenfold, it cannot match the speed of the CPU and memory. There are four fundamental performance metrics for I/O systems of your application.

- *Bandwidth (B)*: This is the amount of data that can be transferred in unit time from one service A to another service B, as shown in Figure 4-23. It is the capacity of the network like your Internet bandwidth of 1Mbps, 1Gbps, etc.
- *Latency (L)*: This is the time taken for the smaller transfer from service A to service B, as shown in Figure 4-23. The measuring units in time are transaction per second (tps), etc. For example, if the request that starts at service A is 0 seconds and reaches service B in 2 seconds, then your transaction rate is 2tps.
- *Throughput (T)*: This is the amount of data moved successfully from service A to service B in a given time period, as shown in Figure 4-23. It is measured bits per second as in Mbps and Gbps.
- *Response time (R)*: This is the time taken from the time service A sends a request to service B until the time that the service indicates the request has completed and reaches service A, as shown in Figure 4-23. For example, the response time is 4ms between your services, etc.

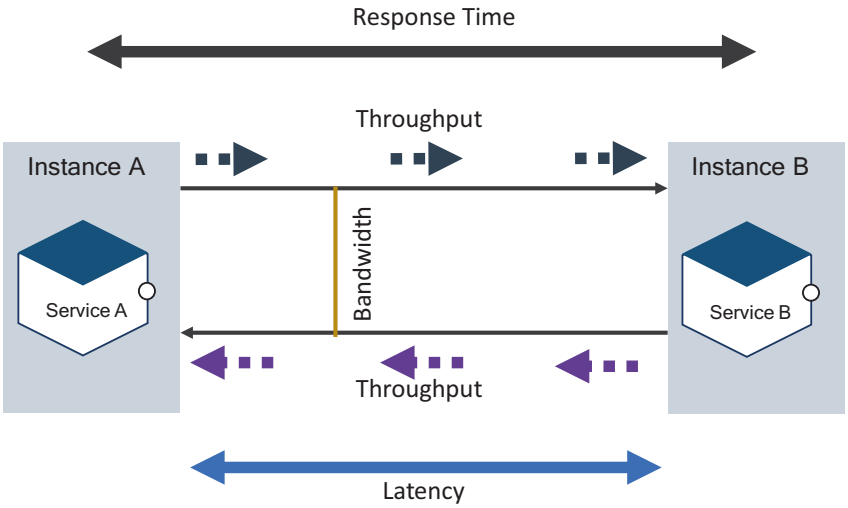

Figure 4-23. *Relationship of BLTR*

What is synchronous and asynchronous messaging?

As shown in Figure 4-24 A, synchronous messaging involves a sender that waits for the server to respond to the request with a message. The thread is blocked between the sender and the receiver. The sender cannot send another request until receiving a response from an earlier request.

As shown in Figure 4-24 B, asynchronous messaging involves a sender that does not wait for a message from the server. An event is used to trigger a message from a server. Even if the sender is down, the message processes the request. The server callback is sent once the server completes its execution. Here there is no blocking of threads.

Blocking I/O means that a given thread, after initiating an I/O operation, cannot perform further calculations until the result is fully received, which means when an API call is invoked to connect with the microservices, the thread that handles that connection is blocked until there is some data to read. Until the relevant operation is complete, that thread cannot do anything else but wait.

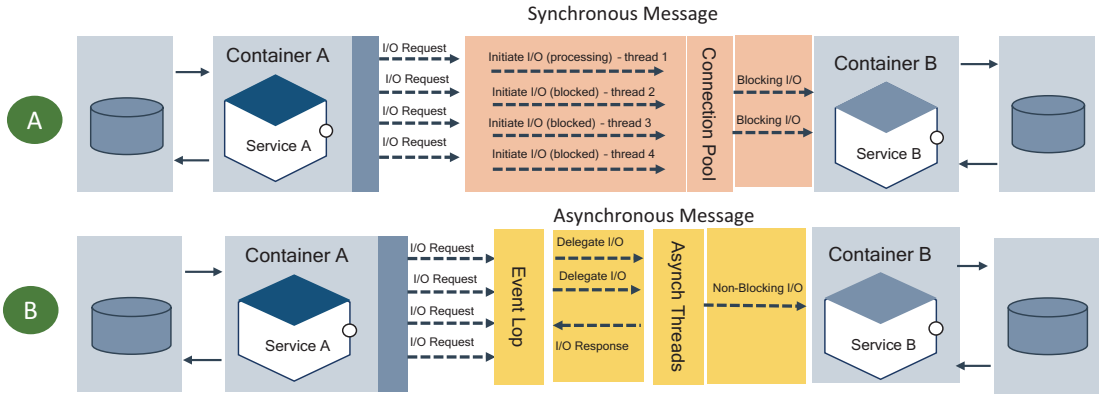

Figure 4-24. *Synchronous and asynchronous blocking processing*

In the synchronous I/O, a thread starts an I/O operation and immediately enters a wait state until the I/O request has been completed. This type of processing consumes a large number of resources.

The asynchronous nonblocking I/O pattern helps in saving the I/O cost where the total cost of I/O is more than the cost of the processing.

The asynchronous nonblocking I/O pattern immediately returns from I/O calls. On completion, an event is emitted, or a callback is executed. The interesting characteristic of this pattern is the fact there is no blocking or waiting at the user level. The entire operation is shifted to the kernel space. This allows the application to take advantage of additional CPU time while the I/O operations happen in the background on the kernel level. In other words, the services implementing nonblocking I/O can overlap the I/O operations with additional CPU-bound operations or can dispatch additional I/O operations in the meantime.

Use nonblocking I/O pattern for good performance under highly concurrent I/O. Most business use cases in modern architecture are based on asynchronous communication by using events; that is called *event-driven architecture* (explained in Chapter 6).

Stream Processing

Stream processing is a technique that lets consumers query continuous data streams and detect conditions quickly in a near-real-time fashion. Detecting the condition varies depending on the type of database and infrastructure you are using. Stream processing allows applications to exploit a limited form of parallel processing more easily. An

application that supports stream processing can manage multiple computational units without explicitly managing allocation, synchronization, or communication among those units. The stream processing pattern simplifies parallel software and hardware by restricting the parallel computation that can be performed.

For the incoming data, a series of operations is applied to each element in the stream, and the operation can entail multiple tasks in the incoming series of data, which can be performed in parallel or serial or both. This workflow is referred to as a *stream processing pipeline*, which includes the generation of the data, the processing of the data, and the delivery of the analyzed data to the consumer.

Stream processing takes on data via aggregation, analytics, transformations, enrichment, and ingestion.

In the Figure 4-25 example, for each input data, the stream processing engine operates in real time on data and provides output. The output is delivered to a streaming analytics application and added to the output streams.

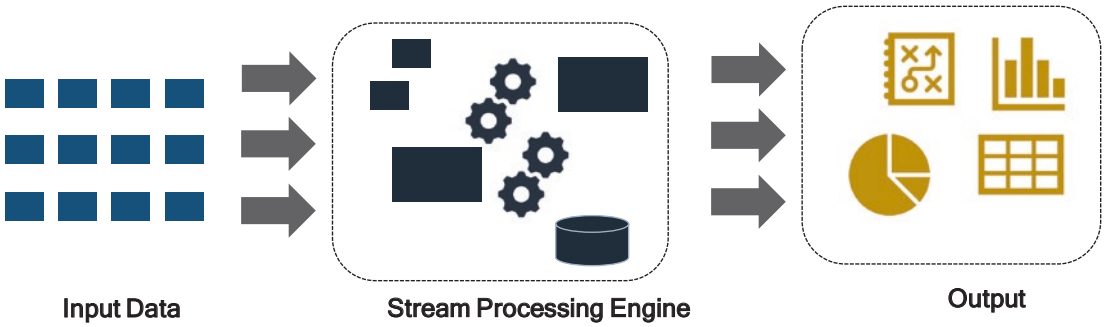

Figure 4-25. *Stream processing*

The stream processing pattern addresses many challenges in the modern architecture of real-time analytics and event-driven applications.

- Stream processing can handle data volumes that are much larger than the data processing systems.
- Stream processing easily models the continuous flow of data.
- Stream processing decentralizes and decouples the infrastructure.

The following are the typical use cases of stream processing:

- Trading
- Smart patient care
- IoT sensors
- Social media events
- Geospatial data processing

You can use tools such as Apache Kafka, Apache Flink, Solace, AWS Kinesis, etc.

Cloud Native Design Pattern for Microservices

The following are design patterns.

Mediator

Partitioning a system into many objects generally enhances the reusability, but proliferating interconnections between those objects tends to reduce it again.

Mediator is a behavioral design pattern and was written about in the Gang of Four pattern book. This pattern is about reducing the dependencies between two objects. This pattern restricts the direct communications between the objects and forces them to collaborate via the mediator object.

The mediator object (which encapsulates all interconnections), as shown in Figure 4-26, acts as the hub of communication; it is responsible for controlling and coordinating the interconnections of its clients and promotes loose coupling by keeping objects from referring to each other explicitly.

- Define an object (mediator) that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Design an intermediary to decouple many peers.
- Promote many-to-many relationships between interacting peers to full object status.

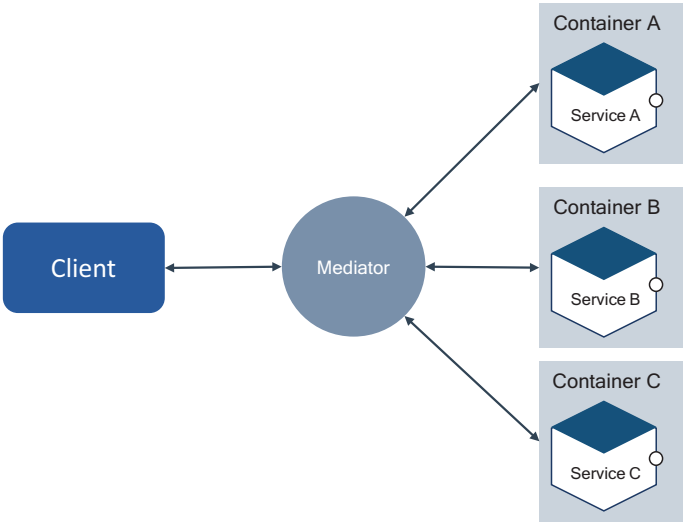

Figure 4-26. *Mediator pattern*

Services are not coupled with one another directly. Instead, each service talks to the mediator, which in turn knows and conducts the orchestration of others. The many-to-many mapping between colleagues that would otherwise exist has been promoted to full object status. This new abstraction provides a locus on indirection where additional leverage can be hosted.

Orchestration

Orchestration is like a conductor in a music concert. In a concert, an orchestrator takes a composer's musical sketch and turns it into a score of an orchestra, ensemble, or choral group, assigning the instruments and voices according to the composer's intentions.

Some say that orchestration is an anti-pattern. In the microservices world, based on my experience across industries, there are various use cases where orchestration is beneficial. Yes, orchestration is a single point of failure (SPOF) in an entire implementation, but that doesn't mean this is an anti-pattern.

Companies such as Netflix and Uber each created an orchestration tool. They are called Conductor and Cadence, respectively. Conductor is used in a workflow that adds Netflix idents to videos. (Idents are those four-second videos with the Netflix logo that appear at the beginning and end of the show.) You can use BPMN tools for orchestration, but with the caveat that you need to make sure of the context, use cases, etc., before

you decide on an orchestration mechanism. Since there are many risks associated with orchestrating microservices, it is prudent for you to limit your orchestration to places that need it.

For example, we used orchestration for a microservices implementation in the telecom industry. This use case is about laying optical fiber in an entire country, which requires a flow of information across systems for an approval process, billing process, order progression, calculation of charged coupled device (CCD), V21, and NH21 validation of optical fiber. The client had a legacy workflow system that was very old and didn't scale as required. To start with, the customer wants to replace the flow system and later do digital decoupling of enterprise systems. What we did was we replace the flow with the orchestrator, and we created microservices for each task to connect synchronously with all the enterprise legacy applications.

Strangler Pattern

The digital decoupling of monolithic applications from scratch is a challenge. It consumes a lot of time and effort and involves a lot of risks. The main thing is to maintain the business continuity. You cannot apply the big-bang approach when decoupling legacy monolithic applications to microservices; it must be done incrementally, as shown in Figure 4-27. Feature of a legacy system can be replaced with microservices iteratively, but, finally, the new system with microservices eventually replaces all the features of the old system. You need to "strangle" the old system iteratively and allow the new system to evolve.

The fundamental strategy to adopt is event interception (i.e., the new microservices decide which events or requests will be passed on to the applications), which can be used to gradually move functionality to the strangler.

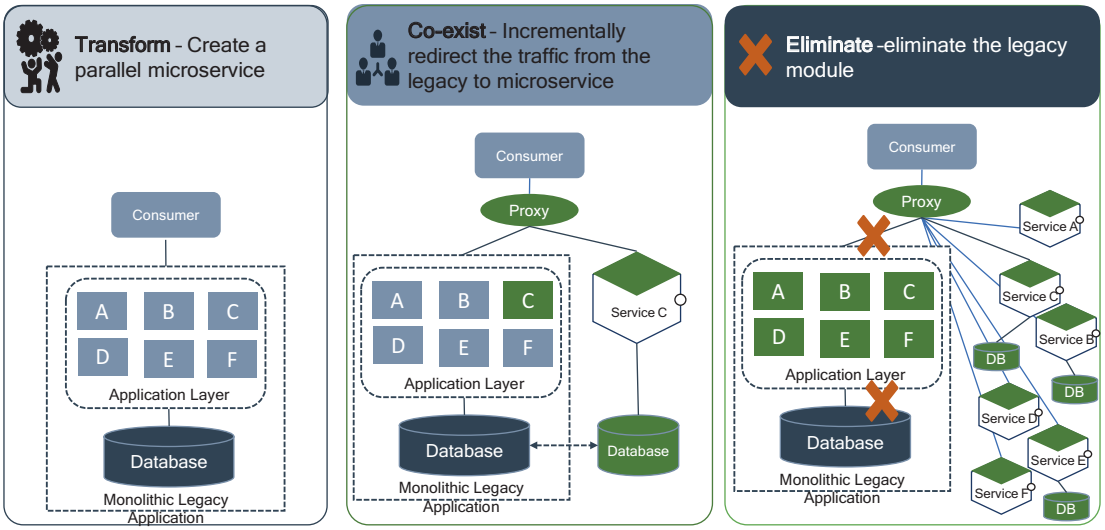

Figure 4-27. *Strangulation steps*

The proxy routes these requests either to the legacy application or to the new services. Existing features can be migrated to the new microservices gradually, and consumers can continue using the same interface, unaware that any migration has taken place.

This pattern helps to minimize the risk from decoupling and iterate the process smoothly. You can also set the percentage of users to an old or new application; once new microservices stabilize, then you route all users to new microservices. Over the time, as features are migrated to microservices, the monolithic legacy application is eventually strangled and gradually decommissioned.

Bulkhead Pattern

The bulkhead pattern is a type of application design that is tolerant of failure. It enforces the principle of damage containment and provides a higher degree of resilience by partitioning the system. In general, the objective of this pattern is to avoid faults in one part of a system taking the entire system down.

The bulkhead pattern, as shown in Figure 4-28, gets its name from cargo ship design. In a ship, a bulkhead is a dividing wall or barrier between other compartments. This means that if a portion of a ship hits a rock or iceberg, that portion fills with water, and the rest of the portion is unaffected. This prevents damage caused to the entire cargo ship and avoids sinking. If there are no partitions in the ship, the entire ship will sink. The bulkhead enforces a principle of damage containment.

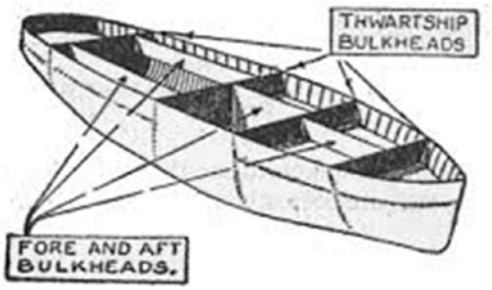

Figure 4-28. *Bulkhead in cargo ship*

The bulkhead pattern is analogous to the bulkhead on a ship and employs the same technique in cloud native architecture by separating your application into independent microservices. A failure in one service does not propagate to other services.

Assume that your consumer sends requests to multiple services simultaneously; during this time, your service is unable to respond in a timely manner due to various reasons. At that point, the request from the consumer to other services is also affected. Eventually, the consumer can no longer send requests to other services in your system.

In a microservices world, you cannot completely avoid dependencies across microservices to provide final responses to the consumers; therefore, you need to maintain intercommunication between microservices to complete the transaction.

To implement this pattern, you need to make sure that all your services work independently of each other and that failure in one will not create a failure in another service. The pattern also depends on what kind of faults you want to protect the system.

In Figure 4-29, service A and service B use service C, because both services depend on common functionality that resides in service C. Suddenly, service A becomes overloaded by multiple requests from the consumer; this will impact service C as service A needs dependent functionality from service C. In this case, service A is bombarded with requests to service C. In the meantime, the user sends a request to service B, so service B needs to call service C to fulfill a request to their consumers. However, service B is unable to get a response, or the response is very slow from service C, which will impact their consumer. This is all caused by both service A and B depending on service C and service C being unable to pool equally for both the services.

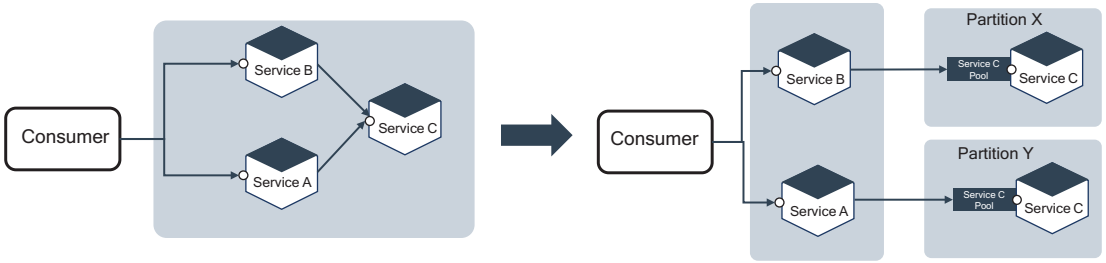

Figure 4-29. *Bulkhead in microservices*

To minimize this impact on service B, you need to adopt a bulkhead approach to partition service C into an equal pool of requests to serve its consumers. You don't need a separate database for service C; both instances of service C in each partition can share a database.

How does the bulkhead pattern work?

Figure 4-30 illustrates the bulkhead pattern with a connection example. This is a classic example for all synchronous connections, for example, in a database. The services request a connection to the database, and each head in this pattern has a single responsibility to manage the respective tasks. One component failing will not impact the whole.

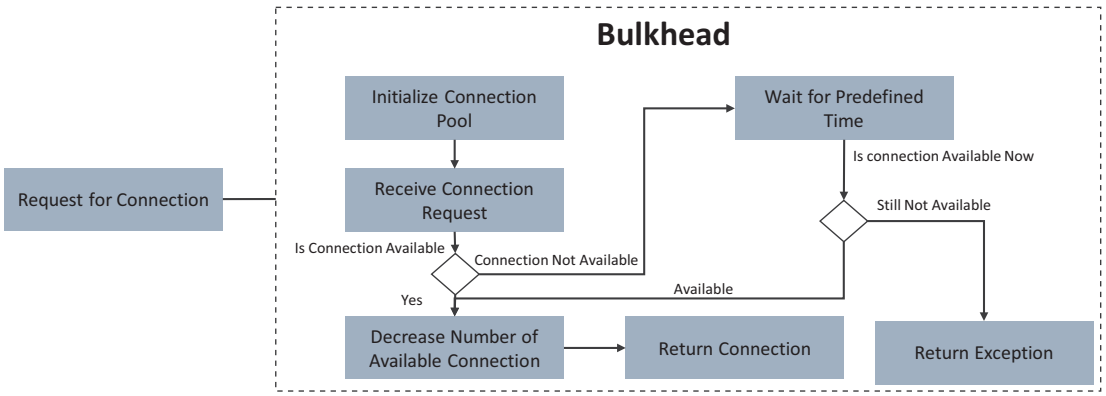

Figure 4-30. *Bulkhead pattern example*

While implementing the bulkhead pattern, you need to analyze the impact of the failure and how to minimize the damage caused by a failure. One more important thing you need to consider is to not generalize this approach for all your services as each service has its failures. Applying this pattern should be feasible both technically and financially.

Usually, you can use the bulkhead pattern to fix the following problems:

- Whenever you want to scale a service independent of another service
- Fault-isolated components of varying risk or availability requirements
- Protecting the application from cascading failures

Anti-corruption Pattern

The anti-corruption pattern, as shown in Figure 4-31, is a layer between the new modernized microservices and the legacy monolithic application. This pattern is useful in decoupling legacy applications into microservices.

In the journey of modernizing your monolithic application into a cloud native application, the journey cannot be done in one release or two releases; it takes many releases and takes months or years depending on the complexity of the system. Therefore, your approach should be iterative to decouple monolithic systems into microservices. In this case, you need to deploy both monolithic legacy applications and microservices to production, so your new microservices can't be executed silo without interacting with the legacy monolithic application.

A monolithic application was built on old technologies and communication protocols and may not be compatible with new technologies like event-driven architecture or API consumption, etc. If your microservice application needs to interact with a monolithic application, it cannot be done directly calling incompatible communication protocols, so you need a middle layer to marshall and unmarshall requests between the legacy monolithic and microservice applications and also between microservices and other enterprise applications in the organization. This middle layer is called an *anti-corruption pattern*.

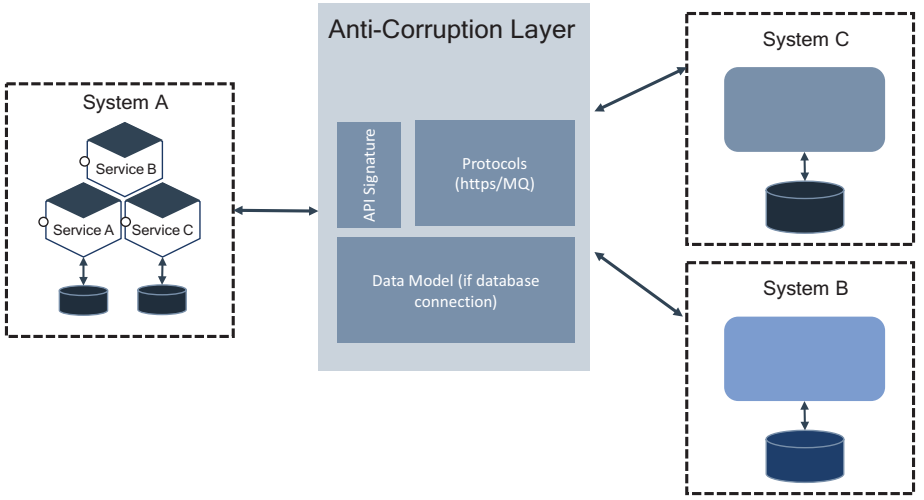

Figure 4-31. *Anti-corruption pattern*

Here is the functionality of an anti-corruption layer:

- Façade for other system; hides the implementation of service C and service D
- Establishing API contract signature
- Communication across systems with respective protocols like HTTP(S), MQ, etc.
- Data model interface if you are interacting with a database directly
- Translating the semantics

You can use any tools like ESBs and custom components as an anti-corruption layer. The following are some of the drawbacks of the anti-corruption layer pattern:

- This layer may add latency between the systems.
- Scaling of anti-corruption layer does not meet the requirements.
- The anti-corruption layer is a single point of failure and requires additional care to make sure it has high availability.

Cloud Native Runtime Pattern for Microservices

Here are the runtime patterns.

Fail Fast

This pattern states that if a service has a problem in serving a request, it should fail fast. An annoying situation is to wait for a response. It is OK for the consumer to get a Not Available or Not Found error rather than waiting for a minute for a response.

In a distributed cloud native architecture, you should know that every service will fail and design your application for resiliency. You can't design a robust microservice and expect no failures at any point in time. You need to embrace failures.

Failures can happen for a variety of reasons like an error in your service, exception in your service, another dependent service not being available, a network failure, etc.

The circuit breaker and bulkhead patterns help you to implement failures in your services, as shown in Figure 4-32.

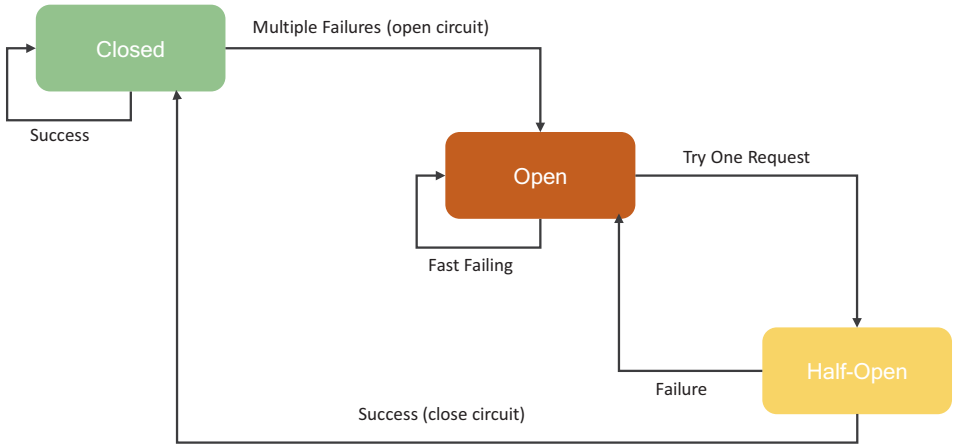

Figure 4-32. *Fail fast implementation*

Write an algorithm to detect the health of the system-based metrics. Certain metrics like the CPU usage of the containers will be evaluated for each scenario, and prediction methods are implemented that try to forecast failures based on these metrics. If the performance of the service is below the threshold, then you need to inject a boot request to the respective service to restart.

There are various online prediction methods to track failures and errors, such as symptom monitoring by using Bayesian predictors, co-occurrence predictors, pattern-based predictors, rule-based predictors, time-series predictors, and system model predictors. More details of failure management are covered in the "Microservices Architecture and Design" Chapter 5.

Retry

The *retry pattern* enables a cloud native application to handle transient failures when it tries to connect to services by transparently retrying a failed operation as mentioned earlier. The retry pattern improves the stability of an application by enabling the service consumer to handle anticipated, temporary failures of the service by retrying to invoke the same service operation that previously failed.

The retry approach is not new; you have been using the retry mechanism in all MQ-based applications. In MQs, you can configure several retries before sending to the dead letter queue. As shown in Figure 4-33, you need to adopt a similar approach in a cloud native application.

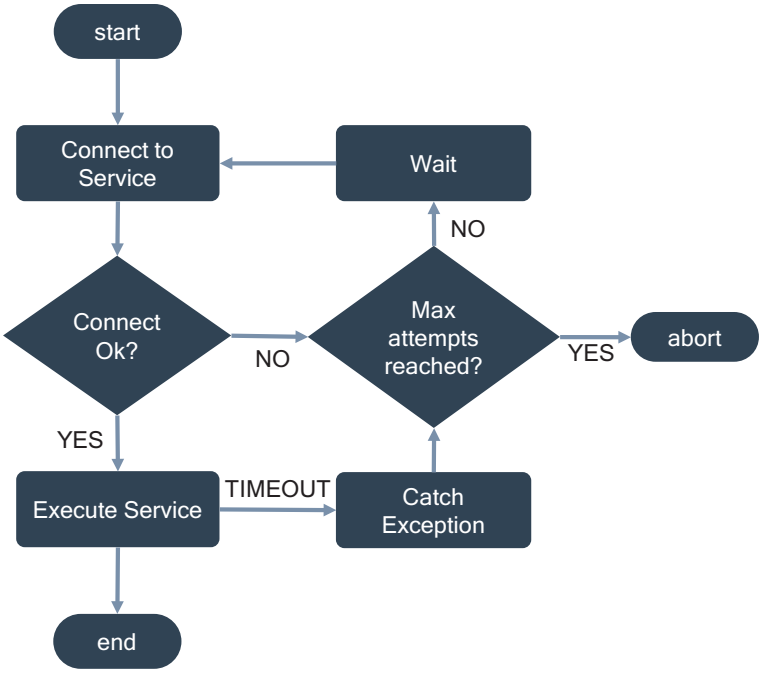

Figure 4-33. *Flow diagram of retry mechanism*

There are some considerations you need to consider when using this retry pattern.

- If you receive any indication that the fault is not transient or unlikely for a normal service request to be successful if repeated, for example, an authentication failure, then you should not use the retry mechanism.

- If the specific fault is unusual, it might have been caused by extraordinary circumstances such as a network packet lost in transit. In this case, the client code should use the retry mechanism immediately.
- If the fault is caused due to the unavailability of services, the service consumer should wait for a suitable time before retrying the request. Be careful here; you cannot retry a service infinitely.
- Set a retry count before you terminate or throw an error if the service is not available.

Sidecar

The sidecar pattern segregates the technical configuration from the functional implementation of a microservice and deploys it in a container alongside the functional microservices container. It is like a sidecar on a scooter, as shown in Figure 4-34.

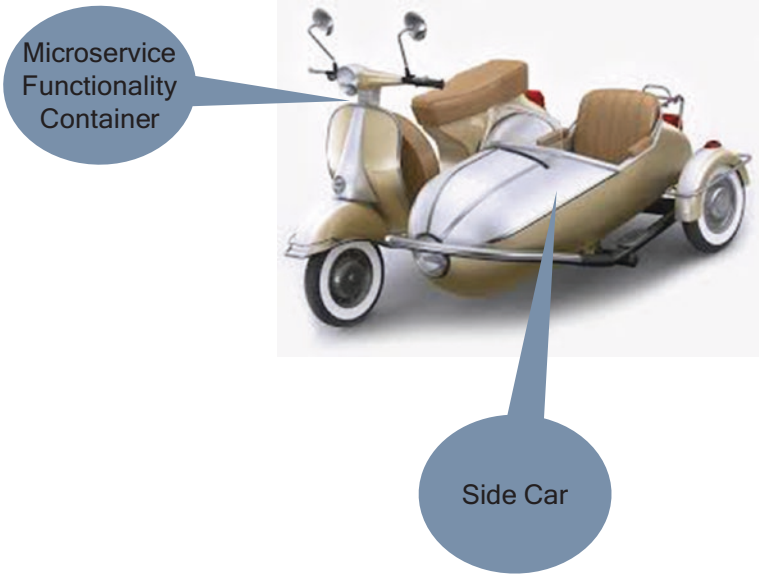

Figure 4-34. *Sidecar on a scooter*

This pattern allows you to add several configuration details from the third party without modifying the microservice. It is a single-node pattern made up of two containers. One container for the application container contains the core business logic, and another container is for the technical configuration details.

The objective of the sidecar container supplements and improves the application container without the knowledge of the application container. The sidecar container is co-scheduled onto the same machine through the container group, and it goes wherever the main container goes.

The sidecar container contains peripheral details of the application container such as platform abstraction, proxy to remote services, logging and configuration, etc.

There is no burden on the main microservices application logic container if you use the sidecar pattern as follows:

- Sidecar is independent of the main application container in terms of the environment, programming language, etc.
- It uses the same resources as the main microservice application.
- There is no latency when you separate the technical details to the sidecar; it runs on the same node.
- It reduces the burden on the application logic.
- There's no dependency on the platform code in the main logic.

The sidecar container uses a service mesh; refer to Chapter 5 for more details about service meshes.

Avoid the use of a sidecar when your application uses synchronous activity and your application code is small; it's not worth separating the technical functionalities from main components and also not suitable for microservices that undergo frequent changes.

Init Containers

Initializing logic for any program is common, if you remember how constructors work in an object-oriented program. The constructor will be called whenever an object gets initiated. The objective of the constructor is to prepare the object to execute the normal business functions.

Similarly, in a cloud native architecture, Kubernetes uses the same logic. There you are using constructors, but in Kubernetes, you need to use init containers.

A Kubernetes pod, as shown in Figure 4-35, can have multiple containers running microservices within it; similar multiple methods in a Java class also have one or more init containers, like the constructors in a Java class, and the init containers run before any application containers are started.

The init containers must complete successfully before the microservice containers start because the main microservice containers have prerequisites before they start. The prerequisites are setting up permissions on the file system, installing application seed data, initializing tools and libraries, etc. These prerequisites cannot be part of the main microservice containers; these prerequisites are part of the init containers.

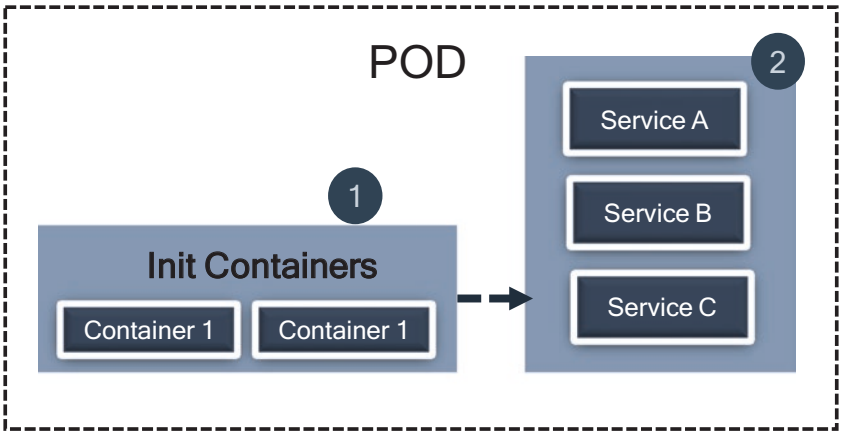

Figure 4-35. *Init container*

The init containers are small and complete the lifecycle very fast. For the pod to be successful, the init container must complete the initialization, or the entire pod will restart. The bottom line is that the init containers are mandatory for any pod to run successfully.

Saga Pattern

The saga pattern is an important pattern in the microservices world to ensure the consistency of the data in a distributed architecture without having a single atomicity, consistency, isolation, and durability (ACID) transaction. This pattern commits multiple compensatory transactions at different stages.

The two-phase commit transaction handles the ACID properties when the commit of the first transactions depends on the completion of a second. It is useful especially when you have to update multiple entities at the same time, like confirming the credit card transaction and crediting your account.

However, when you are working with a microservices transaction, then things get more complicated. Each service has its database, and you can no longer leverage the benefit of local two-phase commit to maintain the consistency of your whole system.

There are many scenarios such as the merchant payment, ecommerce application, etc., where the saga pattern is useful in a distributed microservices environment.

The saga pattern is a sequence of local transactions where each transaction updates data within a single service. The first transaction is initiated by a customer, and each subsequent step is triggered by the completion of the previous one.

In the order process use case, the saga pattern implementation looks like Figure 4-36. Each microservice depends on the other; there are sequences of steps of microservices.

Step 1: Order microservices (the order is created)

Step 2: Payment microservices (the payment is processed)

Step 3: Stock microservices (prepare order and inventory management)

Step 4: Shipping microservice (ship items by using the shipping address)

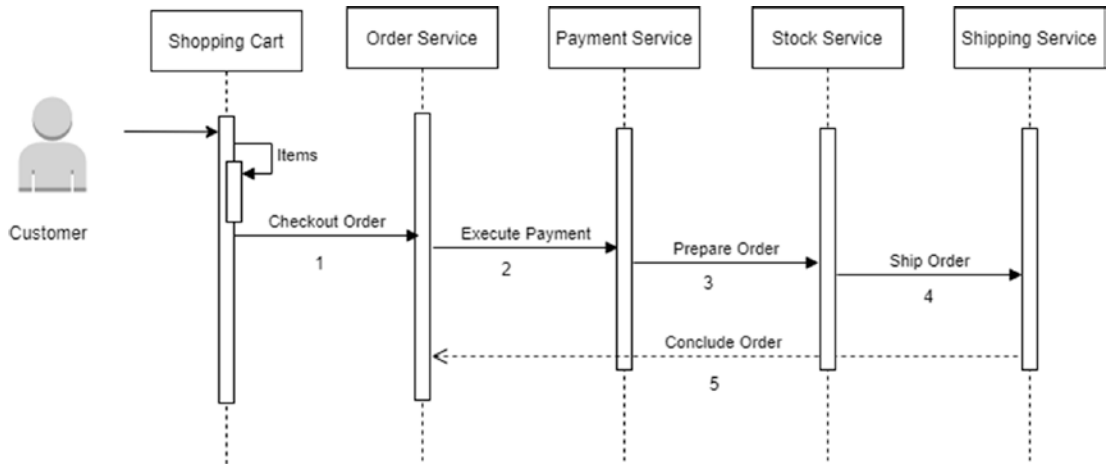

Figure 4-36. *Sequence steps in order*

To implement these use cases, you can choose from the following options:

- *Event-driven system with choreography*: Each microservice produces and listens to other microservices and self-decides whether an action needs to be taken or not.
- *Orchestration*: Central orchestration software or one microservice acts as an orchestration to coordinate saga's decision-making and sequence of business logic.

Event Driven and Choreography

In the choreography approach, as shown in the Figure 4-37, the Order microservice initiates a transaction and publishes an event, and payment services listen to these events and complete their local transaction. The Payment microservice publishes events, and the Stock microservices listens and consumes the payment event and executes its local transaction and publishes a new event. The final Shipping microservice consumes the event and executes the local transaction. The entire distributed transaction ends when the Shipping microservice completes its local transaction and there is no further publishing of events.

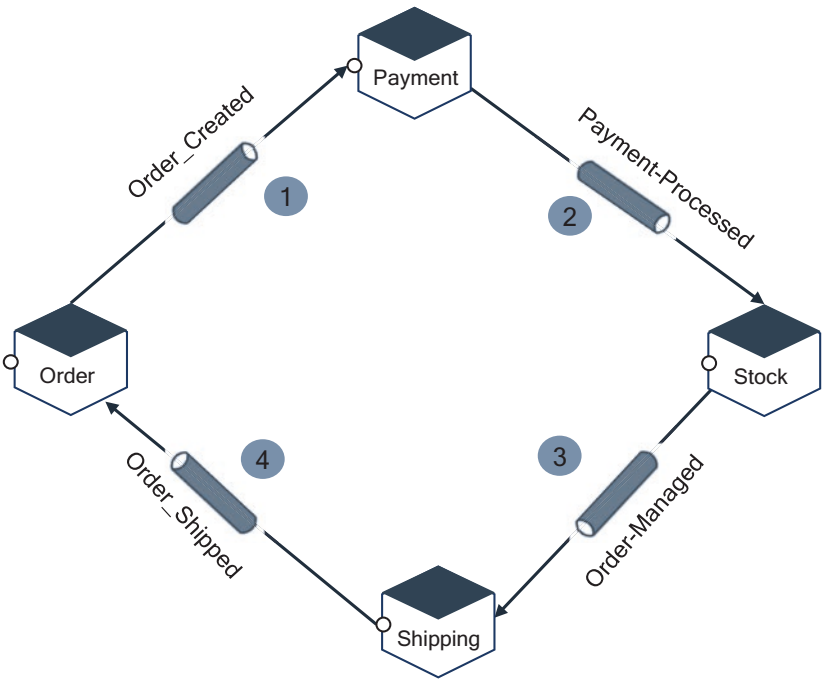

Figure 4-37. *Service interaction in choreography*

In the order process transaction, if the customer cancels its order or stocks are not available after a payment is processed, then you need to roll back the entire transaction and process a payment return to the customer. In this case, you need to implement another compensatory transaction.

In Figure 4-38, if an item is out of stock or a customer canceled an order, the Stock microservice publishes an event, and the Payment microservice consumes an event and processes a refund by compensating a transaction. The Payment microservice publishes an event, the Order microservices consume and update, and the order is canceled.

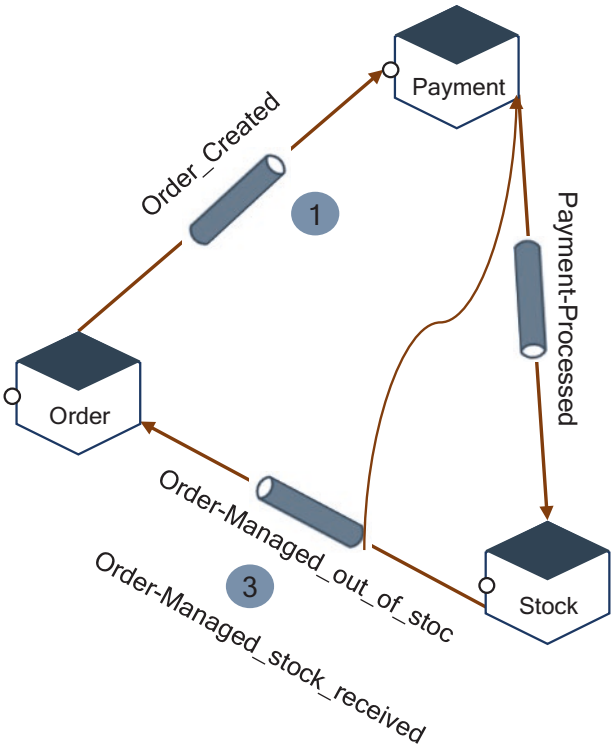

Figure 4-38. *Compensatory transaction*

Orchestrator-Based Saga Pattern

In this approach, either we use orchestration tools like Netflix Conductor/Apache Airflow/Uber Cadence or we create a new microservice with the responsibility of orchestrating each microservices. The saga pattern orchestrator communicates with participated microservices in a synchronous style or point-to-point messaging style with commands about an action.

As shown in Figure 4-39, the orchestrator sends a request to each service.

1. The orchestrator sends an Execute Payment to the Payment microservice, and it replies after execution.
2. The orchestrator sends Stock Manage to the Stock microservices and it replies with Stock Managed.
3. The orchestrator sends Process ship to customer to the Shipping microservices and replies after shipped.

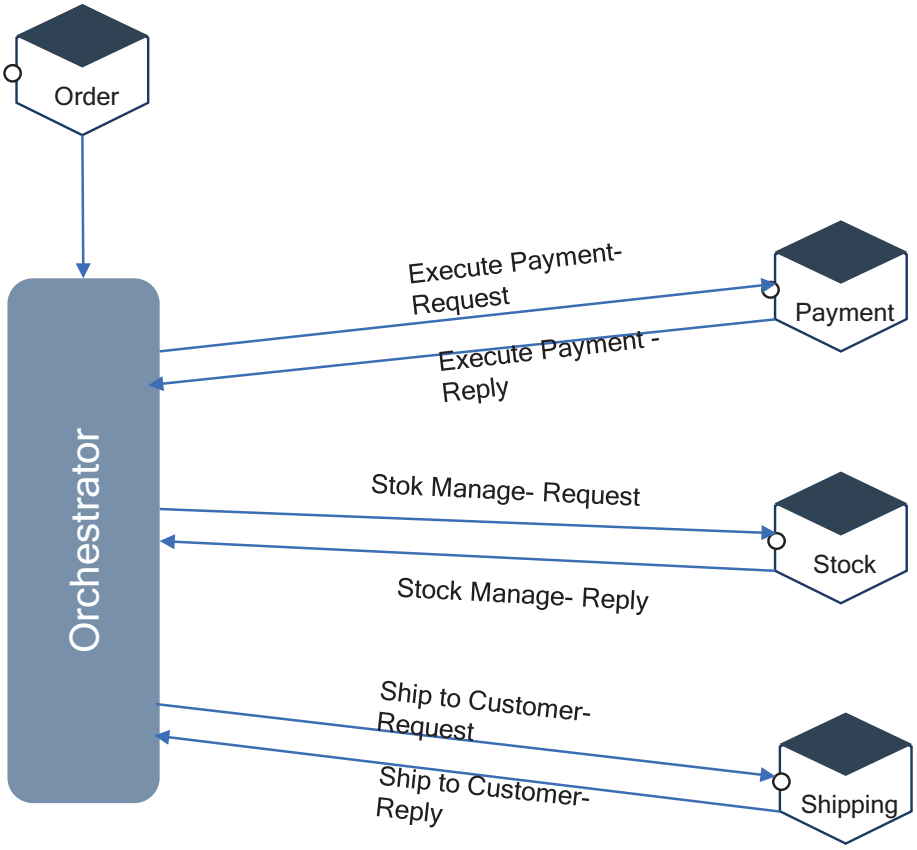

Figure 4-39. *Orchestration saga*

Rollback in the orchestration is easier. If the stock is not available or the customer cancels the order, then the orchestrator sends a command message to each service to compensate for the transaction.

Of the two, the chorography approach is the better and recommended approach to implement over the orchestrator approach.

Summary

Software architects must be familiar with software architecture patterns, as they are powerful tools when designing a cloud native architecture. Architecture patterns provide a proven solution to recurring problems for a given context.

Leveraging patterns gives architects a high-level structure of the cloud native system and provides a grouping of design decisions that have been repeated and used successfully. Using them reduces complexity by placing constraints on the design and allows us to anticipate the qualities that the cloud native system will exhibit once it is implemented.

In this chapter, you learned about some of cloud native patterns related to data, microservices, and event-driven architecture. You can use these patterns at design time and runtime.

The focus of the next chapter is how to architect and design cloud native elements such as microservices, event-driven elements, serverless, and data.