High availability and disaster recovery for applications that are based on containers: Part 2

Prescriptive guidance paper

Table of Contents

Introduction

This article is the second in a series that describes approaches and advice for developers and site reliability engineers to build resiliency for cloud-native applications. It is based on tests that were conducted on a sample application to determine when data might be lost or become inconsistent in an application system failure. To make tests realistic, the sample application is BlueCompute, which is based on a microservices architecture.

The first article in this series discussed a few key concepts of resiliency along the following domains:

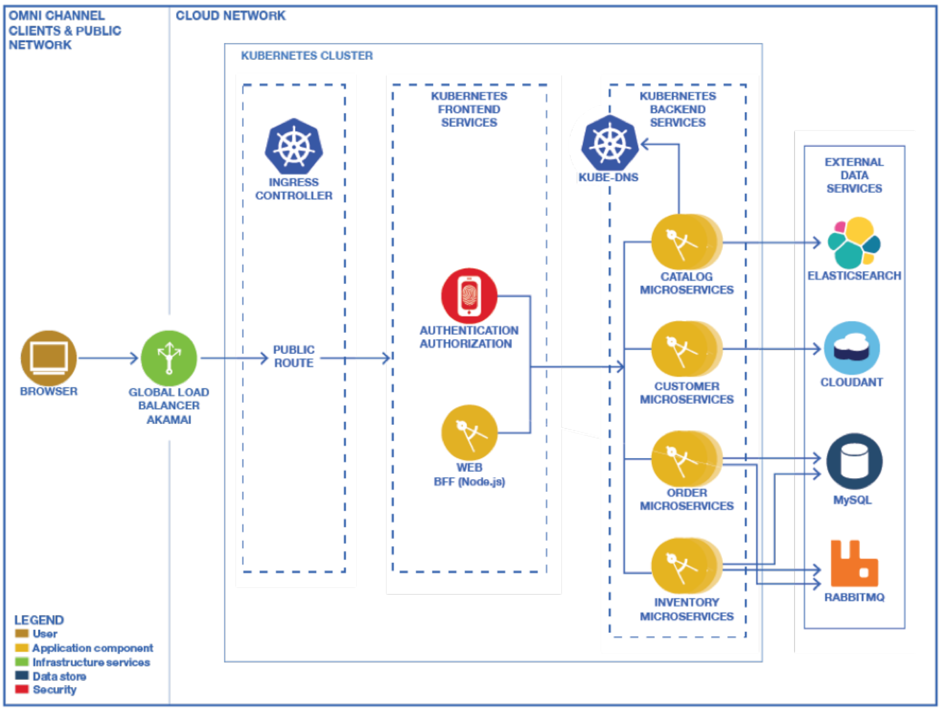
- Technologies: storage types, databases
- Architecture topologies: single cluster, multicluster, multiple sites, site status (active-standby)
- Data consistency features: ACID and BASE databases, CAP theorem

In this article, you explore a few failure scenarios to understand resiliency pitfalls that are common in developing a microservices application. Then, you learn guidelines that are important for application developers to consider to build applications that can be deployed securely with resiliency capabilities. In this context, *developer* is used to refer to a person who is as either a developer or designer of applications.

A real case, the BlueCompute app

The BlueCompute application was tested to derive the resiliency considerations that follow. If you read the first article in this series, you can skip this section.

BlueCompute is built with several microservices, as shown in the following diagram:



The *catalog microservice* uses Elasticsearch as search engine and repository for the catalog data. The real catalog data is on a different system, such as SAP. Updates to the catalog are constantly pushed into Elasticsearch by an automated process.

The only concerns about the catalog data are the performance and high availability (HA) of the service. The data isn't critical and can be quickly restored by an automated procedure.

The *customer microservice* allows customers to register and provide personal information to the site. After the customer registers the initial data, it is rarely accessed. The use case doesn't imply any transactional requirements and can be handled by a Cloudant® NoSQL database that provides great flexibility in the data structure.

The *order microservice* creates an order for the customer. This microservice interacts with the inventory service to get the availability of the ordered item and submits the request to update the item availability after a successful order.

The *order submission microservice* handles the concurrency of users for the same catalog items. For example, this microservice manages cases when multiple customers try to order the last item in the catalog. This scenario can be addressed by using different database types (SQL and NoSQL).

A record in the order database is created only in these situations:

- The inventory provides availability of the item
- A request is submitted to the inventory to process the order

In this way, the customer sees a record for the order only when item is booked by the inventory microservice. The order microservice then waits asynchronously on the message queue to get the result of the operation.

The *inventory microservice* requires a transactional database, which in this case is a MySQL database. This microservice also requires queuing, as it handles order processing and the assignment of an item to a single customer request. The inventory microservice accepts the request from the order submission microservice only in these situations:

- The item is available in the requested quantity
- A message event is created in the queue to handle the processing of the order
- The item availability is reduced in the requested quantity

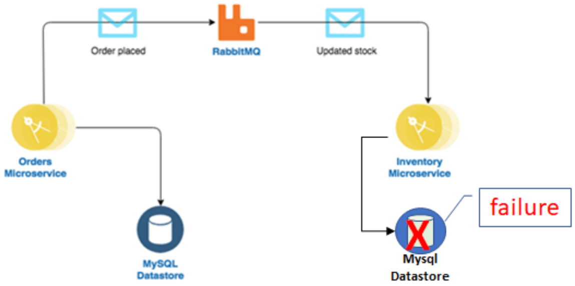
When the inventory is completed, a message is sent to notify the order submission microservice. Failures can happen in the inventory microservice, the database, and the queue. Some failures make the service unavailable. Others can cause a loss of data and inconsistency with the order service.

The complexity of the order and inventory service design depends on the nonfunctional requirements and how they are fulfilled by the data services.

Failure scenario 1: Inventory microservice failure

This first scenario tests the failure of the inventory microservice's database and explores the application behavior that simulates the order of several items. Because of the database failure, the inventory microservice can't check the availability and compare it against the request.

The following diagram shows the failure scenario:



The test is run in 3 phases:

- A preparation phase where the inventory database is checked for the items to be ordered
- A second phase where the user requests to order the items while the database is failing
- A third phase, after the database is recovered, where the user checks the order status while the inventory database is also checked to verify that it was updated

The test results from second phase show that the user can still order items. The result of the last phase, after recovery, is that the user sees that the requested items are ordered, and the items are in the order database. However, the requested number of items isn't subtracted from the inventory database.

The expected result depends on application design behavior. A simple case is that the user can't complete the order and receives a message about temporary order service unavailability, which suggests that the user can retry later. No updates are made to the order or inventory because the back-end service is unavailable.

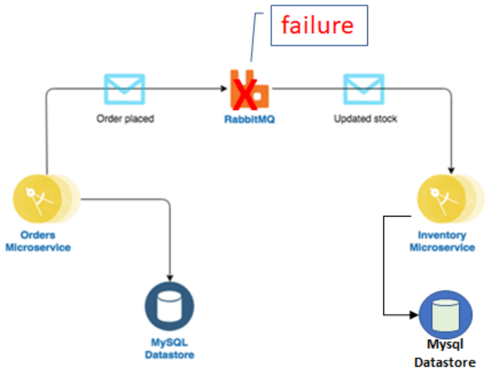
In a more complex case, the order is cached so that it is accepted in the user interface and in the order status of the order database. After the inventory database recovers, the number of items is subtracted from the inventory, the order database is updated, and the order status of the user interface and order database changes from "accepted" to "confirmed". Caching can be activated on the messaging system.

The test results show data inconsistency because the items are ordered but the inventory isn't updated. Because a transaction involves both order and inventory, a problem exists. Without correct transaction management that spans multiple microservices, the inventory microservice data can't be consistent with the order microservice data.

Failure scenario 2: Messaging failure

The second scenario tests a failure of the messaging microservice and explores the application behavior to order several items of an object. Typically, the orders microservice communicates with the inventory microservice to check for the items' availability compared to the number of items that are requested. Then, the inventory numbers are updated based on the number of orders that were requested. Because of the messaging failure, the orders microservice can't communicate with the inventory microservice.

The following diagram shows the failure scenario:



The test is run in 3 phases:

- A preparation phase where the inventory database is checked for the items to be ordered
- A second phase where the user requests to order the items while the messaging is failing
- A third phase where the user checks the order status after the database is recovered while the inventory database is also checked to verify that it was updated

The test results from the second phase show that the user receives the message “your order was not processed, please try again”, so it seems that the order can’t be requested.

The results from the last phase, after recovery, are that the user finds the items of the objects that were requested as ordered, and the items are in the orders database. However, the number of items in the inventory database isn’t changed.

The expected result is that the user can’t order the items. No order update or update of inventory is made because the inventory isn’t available. Based on the test results, data inconsistency is generated because the items are ordered but the inventory isn’t updated.

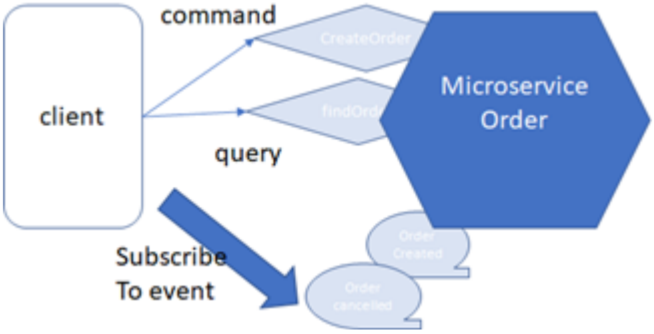
Because the order and inventory databases must make a transaction, a problem exists. Without correct transaction management that spans multiple microservices, the inventory microservice data can’t be consistent with the order microservice data.

Application design patterns to handle failures

To address the 2 failure scenarios, you can use several methods. The following methods use the BlueCompute application as an example. To better understand the methods, review a few key concepts of application design patterns.

Microservices applications and design patterns

BlueCompute is a microservices-based application. A *microservice* is a software component that implements a business function for the entire application. The microservice functions are exposed by APIs for running commands and queries. For example, the order microservice can run a command such as createOrder or queries such as findOrderbyId. This microservice also publishes events when a state changes, such as orderCreated or orderDeleted. This diagram shows an example of an order microservice:



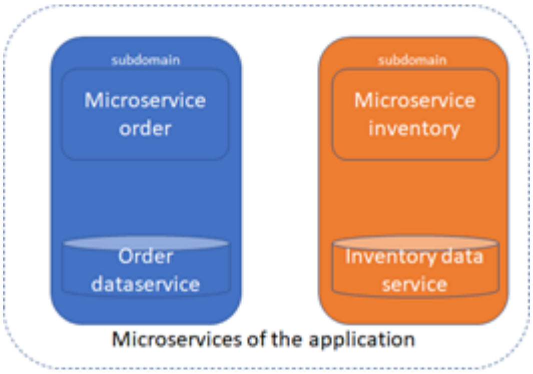
This microservice architecture that is used in the BlueCompute application has many benefits:

- Benefits for development:
  - Development teams can be small and autonomous.
  - Different technologies can be used for development.
- Benefits for deployment and management:
  - Continuous delivery is enabled for each component.
  - Each microservice is small and easily maintained.
  - Each microservice is independently deployable.
- Benefits for nonfunctional requirements:
  - Services are independently scalable.
  - Fault isolation is improved, so the failure of a component doesn't make the whole app fail.

If you look at the component design, you can identify a few design patterns. For example, the Domain Driven Design pattern is used to separate functions that are related to different domains that are implemented in different services. Order, inventory, customer, and catalog are distinct subdomains.

The Database per Service pattern is also applied. Each data service has its own database. This choice has a few advantages: services can be loosely coupled so that they can be developed, deployed, and scaled independently, and each service can choose the database type based on the service requirement. For example, if needed, you can use replication and sharding features.

This diagram shows an example of 2 components of the BlueCompute application, order and inventory, with the described patterns:

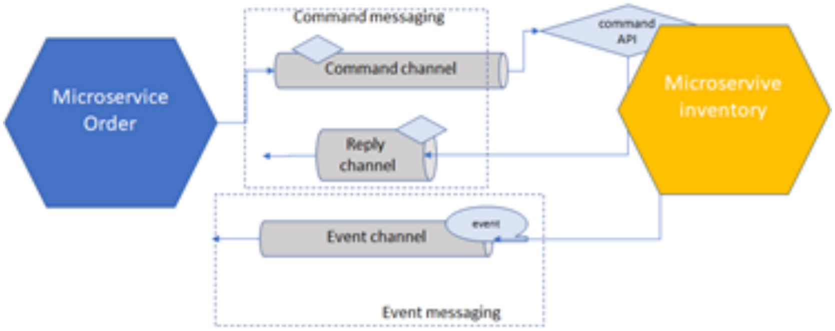


A microservice application is a composite system, so communication is necessary to implement interprocess communication (IPC) for commands, queries, and events. Communication can be done in 2 ways:

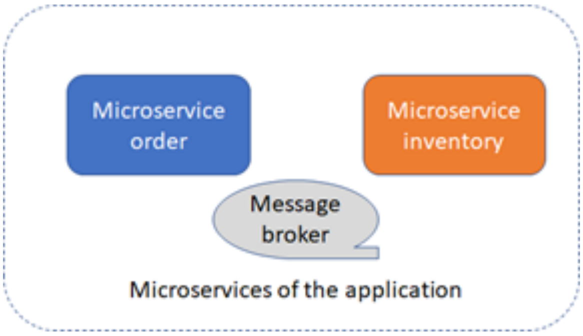
- Through protocols such as REST or gRPC, with a message format that is text-based like JSON or XML, or in a binary format like Protocol Buffer or Avro
- Through messaging, such as IBM MQ or MQTT

Communication can be synchronous, where the client waits for the response to proceed, or asynchronous, where the client doesn't block its execution. Usually, REST or gRPC are a synchronous type, while messaging is asynchronous.

Asynchronous messaging involves commands, documents, or event exchange. Command exchanges are usually done through a point-to-point channel. Event exchange is usually done through a publish/subscribe channel.



Communication is usually done through a message broker instead of direct messaging and can be done through messaging exchange. The use of a message broker can provide benefits, such as decoupling or increasing the availability of the system because a message can be received or sent without all involved parties being available. For those reasons, asynchronous communication can be a valid alternative to synchronous ones when the service API isn't RESTful, usually by using a messaging-based architecture with a broker.



It's common to use a message broker for domain event messages that are used to notify a change in the status of a database. Exchange of domain events are a significant part to manage a transaction.

Challenges for data consistency with microservices to manage transactions

Based on subdomain division and the Database per Service pattern, a few drawbacks exist, especially for business transaction functions that are implemented as separate subdomains.

In the BlueCompute application, order and inventory are implemented as separated subdomains, or separate microservices. The order microservice is a business transaction. To be completed from a data consistency point of view, the order microservice must check the inventory service for item availability, update it, and ensure that no one can update data in the meantime.

Each microservice has its own database to store its data. A service can't retrieve data from other domains because it doesn't have access to that data.

In this context, a database transaction that involves different databases can't be requested for that data. In a monolithic application, this problem is solved by using a database transaction that uses ACID properties of the database to guarantee such data consistency.

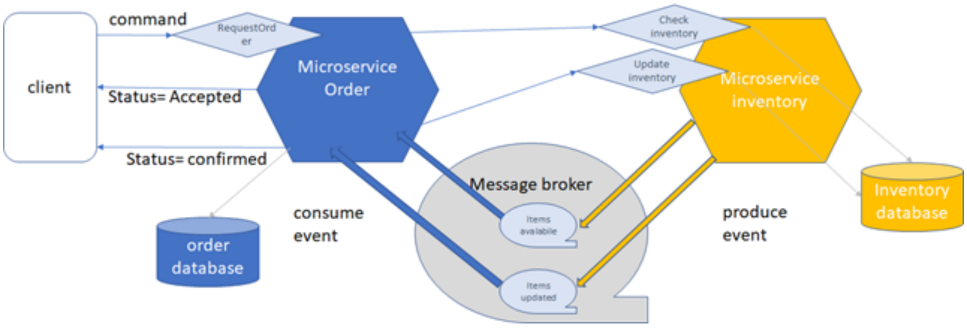
In this microservice-type application, all data is available only at the local or single database level and is available to the related microservices only.

Managing transactions among microservices must address at least 3 main issues:

- The local transaction. Each microservice must update its local database. In this context, *local* means the database that is associated with the microservice in the database per the service design pattern. How can you guarantee local data consistency for those updates with the domain event notification to the requester microservice?
- The global transaction. How can you guarantee global data consistency for a transaction that spans multiple services, such as order and inventory?
- Data query. A microservice needs to query data that is owned by another microservice database and join them. Data is accessible only through API query. How do you address data information retrieval in this context?

Microservice data consistency in a distributed transaction

A typical microservice application design pattern involves a Database per Service as a subdomain decomposition and the use of domain event messaging through a broker. The command and query communication can be synchronous, such as REST, or asynchronous, such as message channels. The following diagram shows a typical scenario for communication between 2 microservices by using REST for command or query communication and messaging for domain event communication.



To manage a transaction that spans microservices, local consistency must be guaranteed, so an atomic database change is made for each database of the microservice. Then, consistency must be ensured among the data of the different microservices.

The following discussion addresses the previous issues for the described design pattern:

- Achieving a local ACID transaction in a distributed system
- Managing distributed transactions by using sagas

- Achieving queried data that is owned by other microservice databases with Command Query Responsibility Segregation

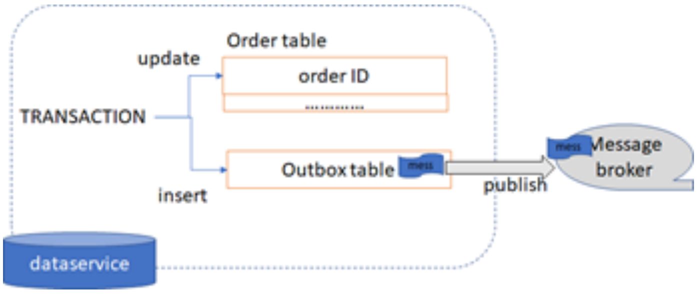
Achieving a local ACID transaction in a distributed system

Messages with message brokers are used by a microservice during a transaction that updates its database. For example, in the BlueCompute application, both order and inventory use an SQL database.

Domain events are generated, or published, to notify the remote requester for the local database change. Both operations must be atomic, or a database can fail before it sends the message and generate an inconsistency.

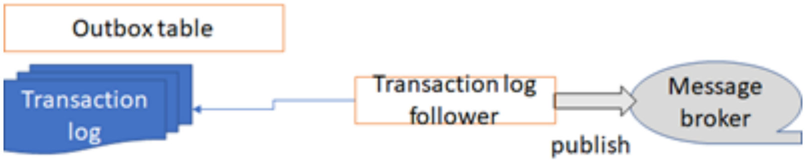
It's necessary to guarantee transactions between the update of the database and the message notification. Otherwise, an inconsistency can occur. For example, an update of the database without a notification of the change can cause a second update request.

A mechanism that uses a database table as a message queue is called a transactional outbox. A database that must make a change uses an outbox table to insert the notify message for the message broker. The insert action follows the update action on the order table, and both are part of a transaction. Atomicity is guaranteed because the transaction is a local ACID transaction. The outbox table is like a temporary message queue where the events are maintained if they're submitted to the real message broker.



When the number of microservices and events increase, the access to the table to fire the change event can affect the performance of the database. In that case, it might be worthwhile to follow a different approach: *publishing by transaction log tailing*.

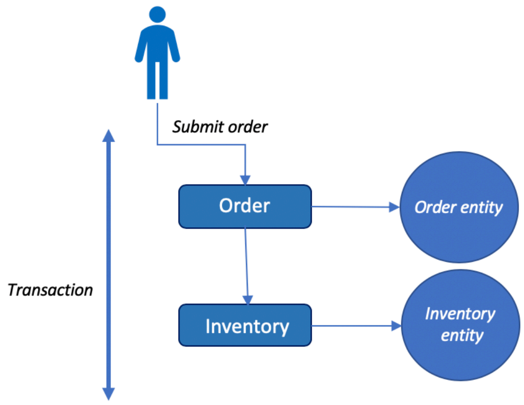
For each application that committed an update, an entry exists in the database's transaction log. A transaction log miner follower reads the transaction log and publishes changes to the message broker.



Reference: https://pradeeploganathan.com/patterns/transactional-outbox-pattern/

Managing distributed transactions by using sagas

For the BlueCompute application, most of the errors and inconsistencies considerations were about the order and inventory microservices, and more specifically, about their interactions. The 2 microservices implement a business transaction for the user who opens BlueCompute to buy a specific product. They're distinct processes and can't implement the scenario as a single transaction in a traditional way, as represented in the following diagram.



You can use 2 options to handle distributed transactions: a 2-phase commit (2PC) and sagas. While they're both possible, in the practice of the microservices, the saga pattern is the typical approach.

The main reason why 2PC isn't used with microservices is that the required locking mechanism can deeply affect the performance of the solution, especially when the complexity and the number of microservices tends to increase.

A saga is made of a sequence of tasks that each implement a microservice local transaction. For each task of the sequence, a compensating task must be defined that can revert the status of the microservice before the execution.

If any of the tasks fail, the compensating tasks are run for each of the tasks that are completed. The final goal of compensating tasks is to restore system to the state before the saga.

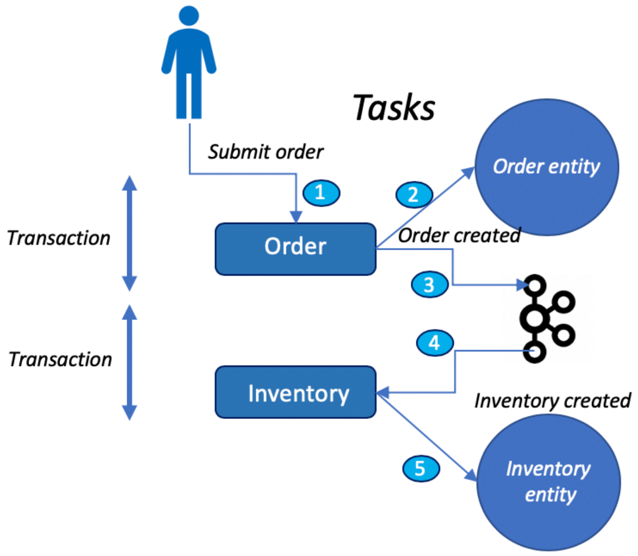
The saga is completed if all the tasks are completed. The saga fails when a task fails and all the compensating actions are completed. A saga must always be completed in one way or the other:

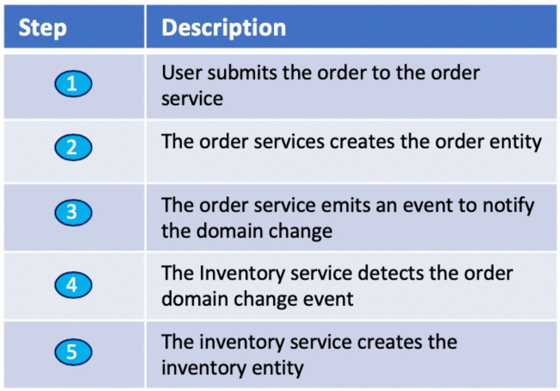
- Choreographed sagas
- Orchestrated sagas

In choreographed sagas, the sequence of tasks is driven by a chain of events that is generated by each task when it ends. The first task is triggered by the input to the saga and when it ends, it generates an event that the next task of the saga consumes.

The saga is completed when either the last service doesn't emit an event, or it emits an event but there are no consumers for it. By using the same approach, the compensating tasks are triggered as a chain of events so that each compensating task responds to error events that come from other microservices that are in the saga.

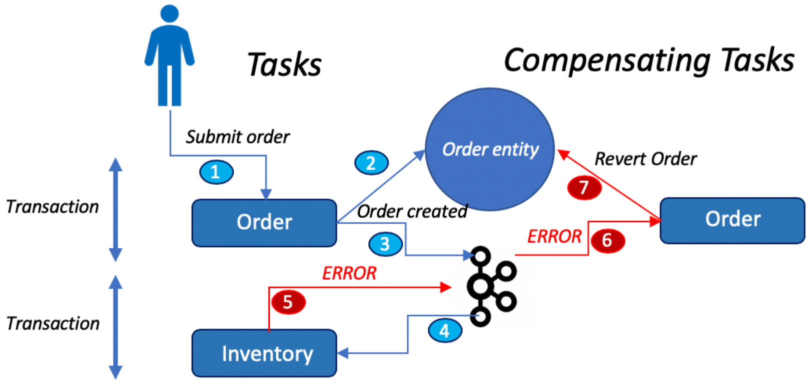
The next 2 diagrams illustrate the order-inventory scenario as a choreographed saga. The first shows a choreographed saga:

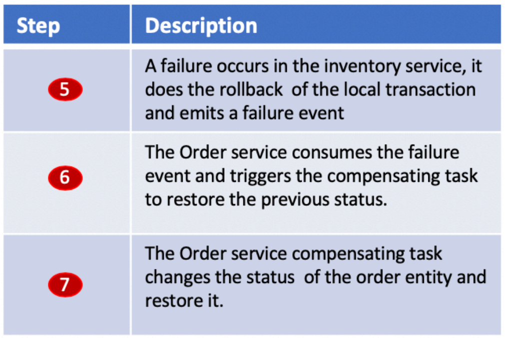




Step	Description
1	User submits the order to the order service
2	The order services creates the order entity
3	The order service emits an event to notify the domain change
4	The Inventory service detects the order domain change event
5	The inventory service creates the inventory entity

The second shows a choreographed saga with failures:





Step	Description
5	A failure occurs in the inventory service, it does the rollback of the local transaction and emits a failure event
6	The Order service consumes the failure event and triggers the compensating task to restore the previous status.
7	The Order service compensating task changes the status of the order entity and restore it.

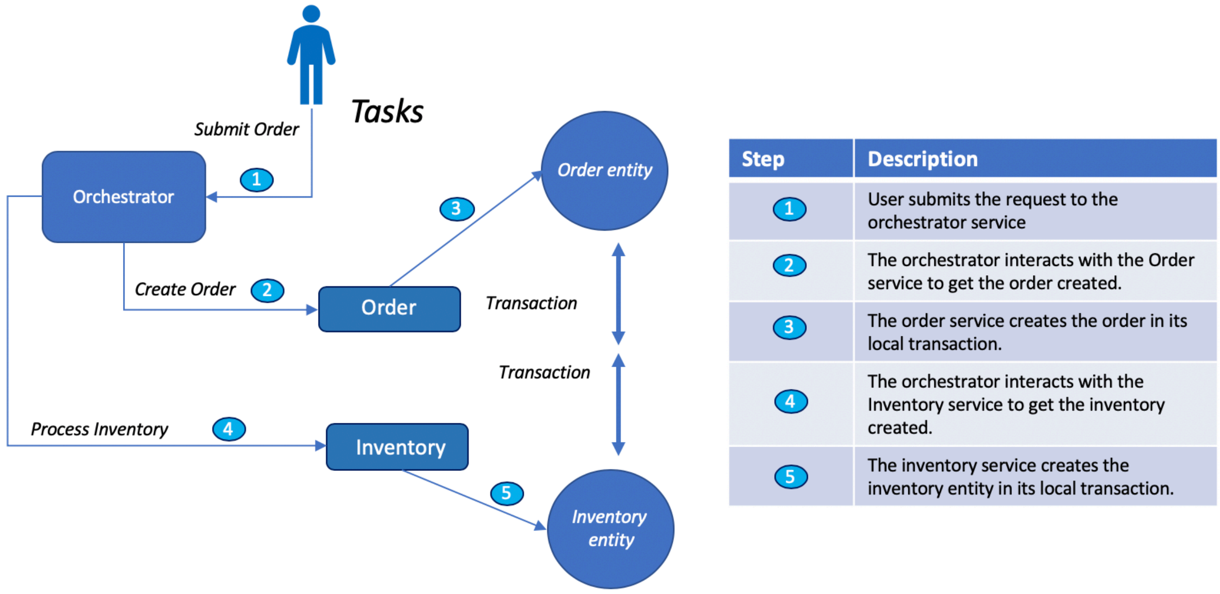
While the choreography can be efficient and performant, it can be difficult to coordinate the interactions when the number of microservices increases. Controlling cyclic dependencies is a common issue. Another pain point is the increasing complexity of logic that each microservice must implement to participate in the saga.

An orchestrated saga uses a centralized orchestrator to drive the execution of the microservices that are participating in the saga. The orchestrator is a microservice that coordinates the tasks that are run in the saga and the associated compensating tasks in case errors occur.

Where the saga execution logic is distributed among the participants, you take a different approach to choreography. The orchestrated saga takes advantage of the centralized logic to simplify the error handling (compensation) and handle advanced scenarios, such as avoiding concurrency issues where the saga instances access the same resources.

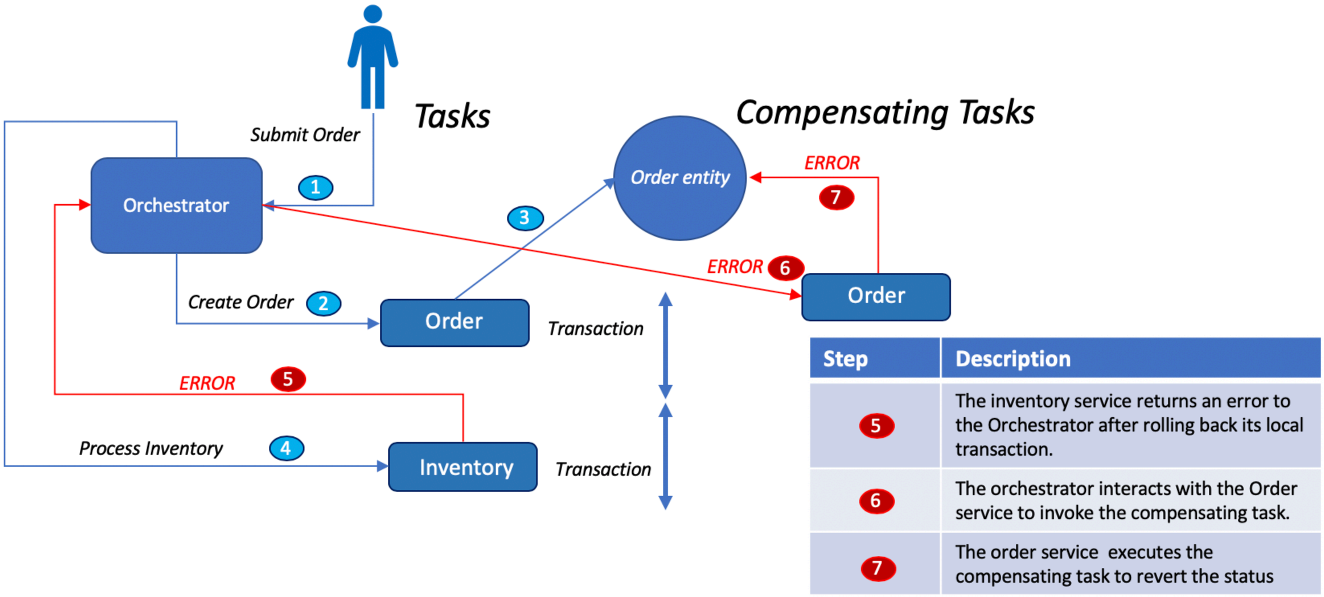
The orchestrator is another microservice to handle. Because of its role in coordinating the saga, it can easily concentrate too much logic that belongs to the coordinated business microservices.

The following diagram illustrates the order-inventory scenario as an orchestrated saga:



Step	Description
1	User submits the request to the orchestrator service
2	The orchestrator interacts with the Order service to get the order created.
3	The order service creates the order in its local transaction.
4	The orchestrator interacts with the Inventory service to get the inventory created.
5	The inventory service creates the inventory entity in its local transaction.

This diagram shows an orchestrated saga with failures:



Step	Description
5	The inventory service returns an error to the Orchestrator after rolling back its local transaction.
6	The orchestrator interacts with the Order service to invoke the compensating task.
7	The order service executes the compensating task to revert the status

Use events for both choreographed and orchestrated sagas because they help to add resiliency and maintain decoupled microservices. Expect the possibility of duplicate events. As a design consequence, the tasks and the related compensating tasks in a saga must be idempotent because they can be run multiple times.

Achieving queried data that is owned by other microservice databases with CQRS

In a traditional monolithic application, the data is usually in a single database. Views and queries that are based on the SQL join of multiple tables can be used to read useful data in the way that the business use cases require.

For example, you might have a table for the inventory, a table for the order, and a table for the catalog entries. When an order is created, it points to the related inventory and catalog entries so that a join between the 3 tables can immediately provide complete information about the user request.

For microservices architectures, the Database per Service pattern is often adopted. It isn't always possible to use the same approach as for a monolithic application. The requirement is still valid, and the business use cases might require data to be collected and combined from multiple microservices before it is provided to the client.

Command Query Responsibility Segregation (CQRS) is a common pattern that addresses the need to implement a query that retrieves data from multiple microservices. It consists in a database that is a READONLY replica of the microservices' key data. The application subscribes to the domain events of the microservices that own the data and keeps the replica in sync. The READONLY replica is a single database where SQL queries can be run to achieve the desired results.

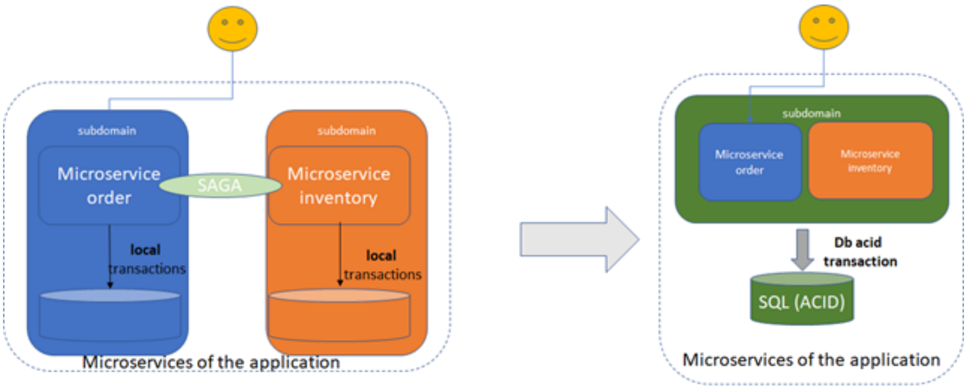
CQRS also applies to traditional and monolithic architectures and doesn't necessary imply the use of distinct databases. In this case, the replica can be synchronous or asynchronous depending on the application requirements and architecture.

For microservices, the CQRS is implemented by using a distinct database where the microservices data is asynchronously replicated and potentially adapted to the model that is optimized for reading the data. As consequence, the READONLY database must be considered as eventually consistent.

Alternative function aggregation for business transactions: single subdomain

To ensure strong data integrity requirements (ACID), sometimes a different subdomain definition for entities might be suitable. For example, you might aggregate the function's orders and inventory in a single subdomain to apply strong consistency inside the transactional boundary.

Depending on the context, you can decide which approach is best and whether an aggregation is better than keeping the microservices separated. Of course, if the transaction involves more than 2 microservices, an aggregation of all of them is a major impact compared to the microservices architecture. Aggregation isn't necessarily preferred, but it is an option.



Resiliency pattern for microservices applications communications

To increase the reliability of microservices, you can use techniques to reduce the impacts of a failure or the outage of a specific service to the whole application. For example, to prevent a cascade impact of a service's failure to other services, you can apply these patterns:

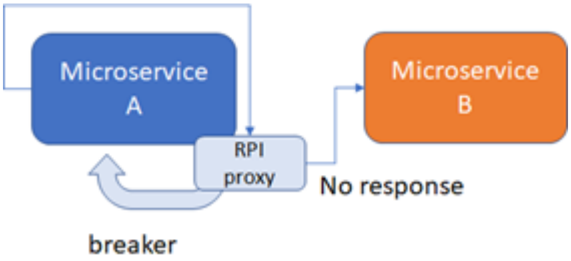
- Circuit breaker for synchronous communication

- Managing duplicate messages in message broker messaging system.

Circuit Breaker for synchronous communication

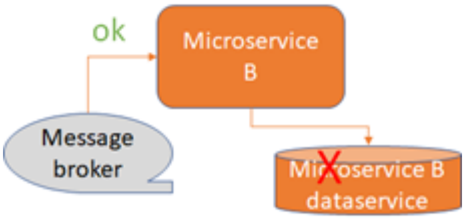
What about the failure of services? For example, interprocess communication (IPC) can be disrupted by service failure, network failure, or target overload. You can apply the Circuit Breaker pattern to manage this scenario.

Because IPC uses a proxy interface, it's wise to design it to manage call failures that deal with network timeouts or excessive load. The management of this scenario can be done by using a circuit breaker that stops further remote calls without trying to invoke them for an interval of time and returns a message to the client such as "temporary unavailable". Istio provides an open-source implementation of such a mechanism.

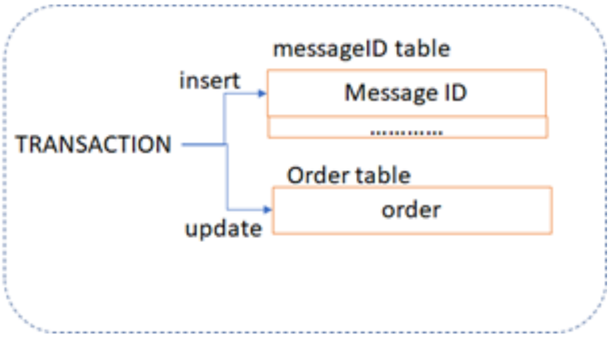


Managing duplicate messages in a message broker messaging system

In another example, the failure of a client, network, or message broker can cause a message to be delivered multiple times. Imagine that a client fails after it processes a message and updates its database, but before it acknowledges the message. The message broker delivers the unacknowledged message again, either to that client when it restarts or to another replica of the client. A message of "order created" and "order cancelled" is consumed, but the "order created" message isn't acknowledged. If the broker resends an "order created" message, it might override the "order cancelled" message.



One of the ways to manage this scenario is to track messages at the consumer by using a table to store the messageID of the message that it receives and to run a transaction between that insert and the order table update. In this way, a second insert of the same ID is refused as duplicate.



Bridging the gap between development and operations

You can design your application behavior to prevent and avoid failures and data inconsistency. From an application point of view, this can be done in 2 complementary ways: by using asynchronous communication to address a microservice failure or by using a design pattern such as saga to manage transactions.

In general, you can achieve resiliency through different approaches that are used at same time:

- Change the application flow; for example, to move from an interactive synchronous application interface to an asynchronous interface with an order to be requested and asynchronously confirmed and fulfilled
- Change the infrastructure, configuring redundancy and data replication
- A mix of both to address problems and balance costs

Designing the best solution for resiliency that addresses your business needs affordably requires a strong collaboration between the development and operations organizations.

Scenario 3: Zone failure in a multizone region

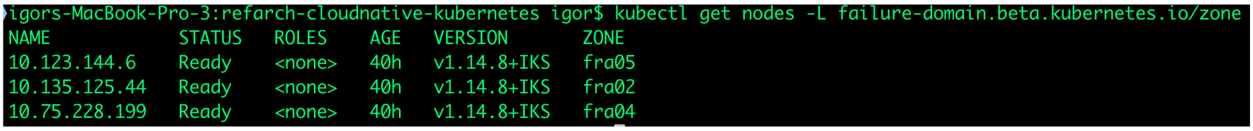
This scenario involves the multizone installation of a Kubernetes cluster. If such a cluster is consumed from the public cloud, such as the IBM Cloud Kubernetes Service, the deployment is done by the cloud provider and is transparent to the consumer. To better discuss resiliency concepts, this scenario covers a consumer-operated deployment.

In a multizone cluster Kubernetes installation, your cluster was installed in multiple data centers or in multiple separated zones in a single data center. Zones as separate data centers is the most resilient approach but is also the most expensive. Zones that are internal to a single data center can provide separate fault domains at a lower cost, but with less resiliency to large failures.
The data centers must be in the same region and be close to each other in the network to reduce latency. If your data centers are in different regions, you can’t use multizone Kubernetes. In that case, deploy multiple Kubernetes clusters.

Multizone means that you still work in a single cluster, but you want to decrease the risk of a data center failure. Multizone allows resource reallocation in the cluster and automatic failure discovery.

Kubernetes added multizone support in version 1.2. Kubernetes uses the failure-domain.beta.kubernetes.io/zone label for nodes to specify a node zone. The Kubernetes scheduler uses those labels to spread the Pods by using zone-specific information.

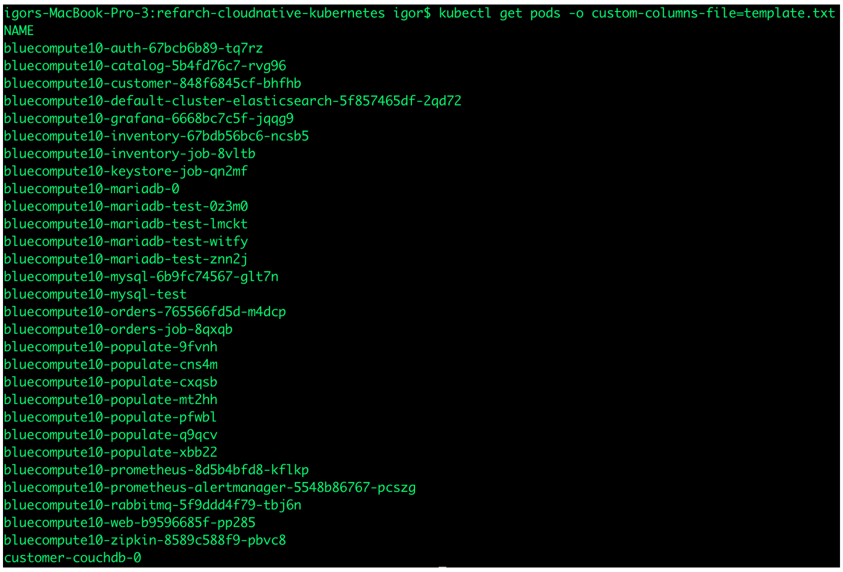
Multizone doesn’t mean that Kubernetes automatically increases the number of instances of your microservices. After cluster creation, you can see all nodes as original workers. This example shows the output of a cluster with 3 zones that each have one worker:



```
igors-MacBook-Pro-3:refarch-cloudnative-kubernetes igor$ kubectl get nodes -L failure-domain.beta.kubernetes.io/zone
NAME            STATUS   ROLES    AGE   VERSION       ZONE
10.123.144.6    Ready    <none>   40h   v1.14.8+IKS   fra05
10.135.125.44   Ready    <none>   40h   v1.14.8+IKS   fra02
10.75.228.199   Ready    <none>   40h   v1.14.8+IKS   fra04
```

The first worker is in fra05, the second is in fra02, and the third is in fra04. They’re in different locations in the same region: Frankfurt.

For the BlueCompute application, you still see the same number of instances. But you have the capacity to reschedule your application if a data center fails. The Kubernetes scheduler uses zone information to schedule microservices the same way as it uses anti-affinity.



```
igors-MacBook-Pro-3:refarch-cloudnative-kubernetes igor$ kubectl get pods -o custom-columns-file=template.txt
NAME
bluecompute10-auth-67bcb6b89-tq7rz
bluecompute10-catalog-5b4fd76c7-rvg96
bluecompute10-customer-848f6845cf-bhfhb
bluecompute10-default-cluster-elasticsearch-5f857465df-2qd72
bluecompute10-grafana-6668bc7c5f-jqqg9
bluecompute10-inventory-67bdb56bc6-ncsb5
bluecompute10-inventory-job-8vltb
bluecompute10-keystore-job-qn2mf
bluecompute10-mariadb-0
bluecompute10-mariadb-test-0z3m0
bluecompute10-mariadb-test-lmckt
bluecompute10-mariadb-test-witfy
bluecompute10-mariadb-test-znn2j
bluecompute10-mysql-6b9fc74567-glt7n
bluecompute10-mysql-test
bluecompute10-orders-765566fd5d-m4dcp
bluecompute10-orders-job-8qxqb
bluecompute10-populate-9fvnh
bluecompute10-populate-cns4m
bluecompute10-populate-cxqsb
bluecompute10-populate-mt2hh
bluecompute10-populate-pfwbl
bluecompute10-populate-q9qcv
bluecompute10-populate-xbb22
bluecompute10-prometheus-8d5b4bfd8-kflkp
bluecompute10-prometheus-alertmanager-5548b86767-pcszg
bluecompute10-rabbitmq-5f9ddd4f79-tbj6n
bluecompute10-web-b9596685f-pp285
bluecompute10-zipkin-8589c588f9-pbvc8
customer-couchdb-0
```

How many data centers should be used?

Use a 3-zone cluster. You can also use a 2-zone cluster, but it isn’t as cost-efficient. Finding and maintaining more than 3 data centers in the same region for an application is difficult. Don’t forget that latency between all your data centers must be small. Stateful microservices depend on latency. They need a time to synchronize the state.

Why is a 3-zone installation cheaper than a 2-zone installation?

To improve resiliency, increase number of nodes and enable scheduling between them. Two zones sound reasonable for a stable cluster, but it’s better to use 3. The reasons to use a 3-zone cluster instead of a 2-zone cluster are as follows:

- An odd number of zones can recover from split brain failure scenarios. If a communication failure occurs, the zones might be active, but one can’t connect to the others. In that case, the components in each zone are still active but can’t determine whether the others failed or are still active. The two instances that can still communicate can assume that the third is off and become the master instances. The instance that can’t communicate with the others then detects that it is isolated and acts as a subordinate instance.
- Multiple databases work in HA mode only when distribution is to at least 3 separated Kubernetes nodes, such as CouchDB.
- Kubernetes masters tolerate a failure of one or more masters if most remain up, which implies that you need at least 3 zones to get the full benefit from multizone deployment.
- A 3-zone installation is cheaper than a 2-zone installation. Consider this example:
  - The BlueCompute application requires 8 GB of RAM and 3 cores.
  - In a 2-zone option, if a zone fails, you need capacity to put all workloads into one zone, so the second zone must have the same 8 GB of RAM and 3 cores.
  - In a 3-zone option, if a zone fails, you still need the same capacity but can split it between two zones with 4 GB RAM and 1.5 cores in each.

	1 zone	2 zones	3 zones
Number of workers	1	2	3
RAM for each zone	8	8	4
CPU for each zone	3	3	1.5
Total RAM	8	16	12
Total CPU	3	6	4.5

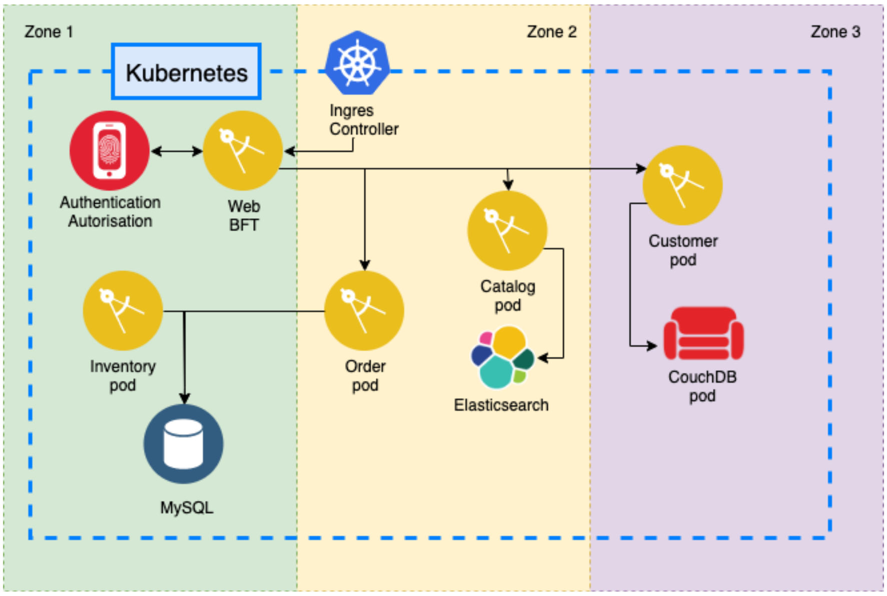
To understand whether you have enough capacity to cover all workloads if a zone fails, use this formula:

(total RAM) – (RAM for each zone) ≥ (required RAM)
and
(total CPU) – (CPU for each zone) ≥ (required CPU)

The master branch of the BlueCompute application has no replicas for microservices. When Kubernetes starts containers on another node, the application has an outage while it waits in case zone failure occurs. To decrease this outage or completely remove it, use one of these options:

- Configure the application to have a functional requirement RTO (recovery time objective) with zero downtime. To achieve that RTO, use replicas (ReplicaSet) to BlueCompute stateless microservices and increase the resiliency of the stateful application.
- To decrease outage time, you can prepare the Kubernetes cluster to move workloads from one zone to another. You can duplicate the Docker registry and create a local cache of Docker images on each node. The Docker image downloading process is time-consuming and can overload your network.

After you deploy the master branch of BlueCompute into a 3-zone installation, you have an application that works but that has some solution design issues in resiliency.



The BlueCompute application still has an RTO that is greater than zero. To improve resiliency, take these steps:

1) Increase the number of replicas of your stateless microservices.
2) Prepare your databases to lose part of the cluster.

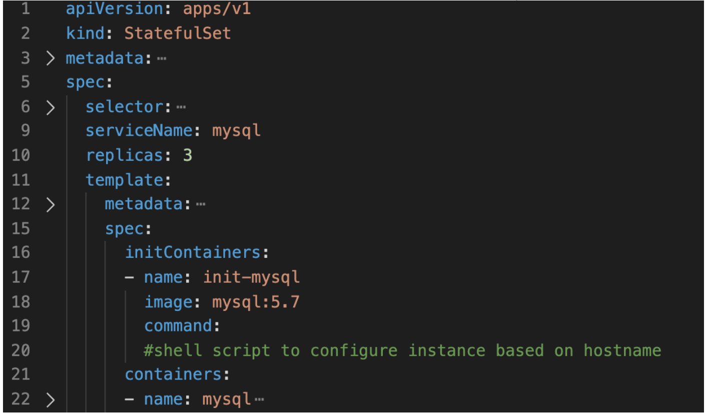
If your application has no complex logic for affinity, anti-affinity, and node selectors and you don’t use stateful microservices, you can add new replicas by using a Kubernetes ReplicaSet or Deployment. The BlueCompute application has no complex logic where you can place Pods. With many rules for selectors, Kubernetes scheduler might be in a situation where it can’t find a suitable node for a Pod. In that case, your Pod is in a Pending state.

To determine how to improve resiliency for a database, you must know its cluster’s architecture and cluster-specific information for that database. For the BlueCompute application, you use 3 types of databases, MySQL, CouchDB, and Elasticsearch.

A few differences exist between stateless and stateful services scaling. You can’t simply increase the number of instances for stateful microservice. To create a database cluster in Kubernetes, you can use StatefulSet. In that case, all instances differ in small ways from each other. You must specify which one instance is the master and which are subordinates.

MySQL cluster

A traditional MySQL cluster contains 3 instances with one master and 2 subordinates. This example shows an overview of StatefulSet for MySQL:



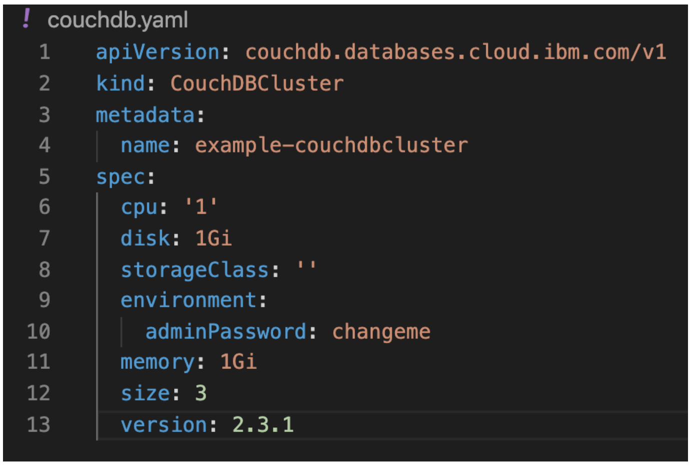
```
apiVersion: apps/v1
kind: StatefulSet
metadata:⋯
spec:
  selector:⋯
  serviceName: mysql
  replicas: 3
  template:
    metadata:⋯
    spec:
      initContainers:
      - name: init-mysql
        image: mysql:5.7
        command:
        #shell script to configure instance based on hostname
      containers:
      - name: mysql⋯
```

As you can see, initContainer is also included, which is responsible for the main container configuration. The initContainer code specifies the correct hostnames and selects a master for the MySQL cluster. The Kubernetes scheduler tries to put all instances on different nodes. Another way to get a MySQL cluster is to use Kubernetes Operators.

CouchDB cluster

Kubernetes Operator for Apache CouchDB is now available. IBM Cloudant, which is built on CouchDB, is a standard database for all cloud-native applications in IBM Cloud. CouchDB provides a crash-resistant storage engine and clustering with data redundancy. It is written in Erlang and has a good fault-tolerance implementation.

After the CouchDB cluster operator is installed, you need to apply only a single YAML configuration to your cluster.

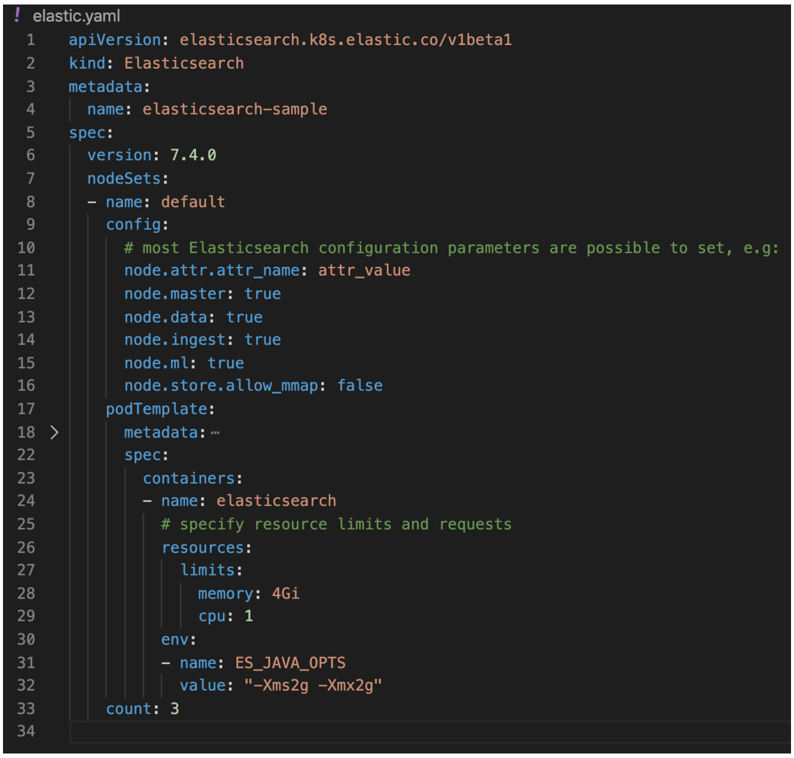


```yaml
couchdb.yaml
apiVersion: couchdb.databases.cloud.ibm.com/v1
kind: CouchDBCluster
metadata:
  name: example-couchdbcluster
spec:
  cpu: '1'
  disk: 1Gi
  storageClass: ''
  environment:
    adminPassword: changeme
  memory: 1Gi
  size: 3
  version: 2.3.1
```

Each database node in an Apache CouchDB cluster requires a separate node, as it contains anti-affinity logic. For the minimum installation, it requires 3 separate workers. The Apache CouchDB operator has its own logic to spread cluster across different zones. If you lose 1 of the 3 zones, the cluster continues to work. If you lose 2 of the 3 zones, Apache CouchDB cluster declines all inbound traffic to save the last replica of the data. It automatically starts to work after it discovers a new replica.

Elasticsearch cluster

The same approach can be used to create an Elasticsearch cluster. You can use multiple implementations for an Elasticsearch operator, such as this one from Elastic. To create a cluster, apply this configuration:



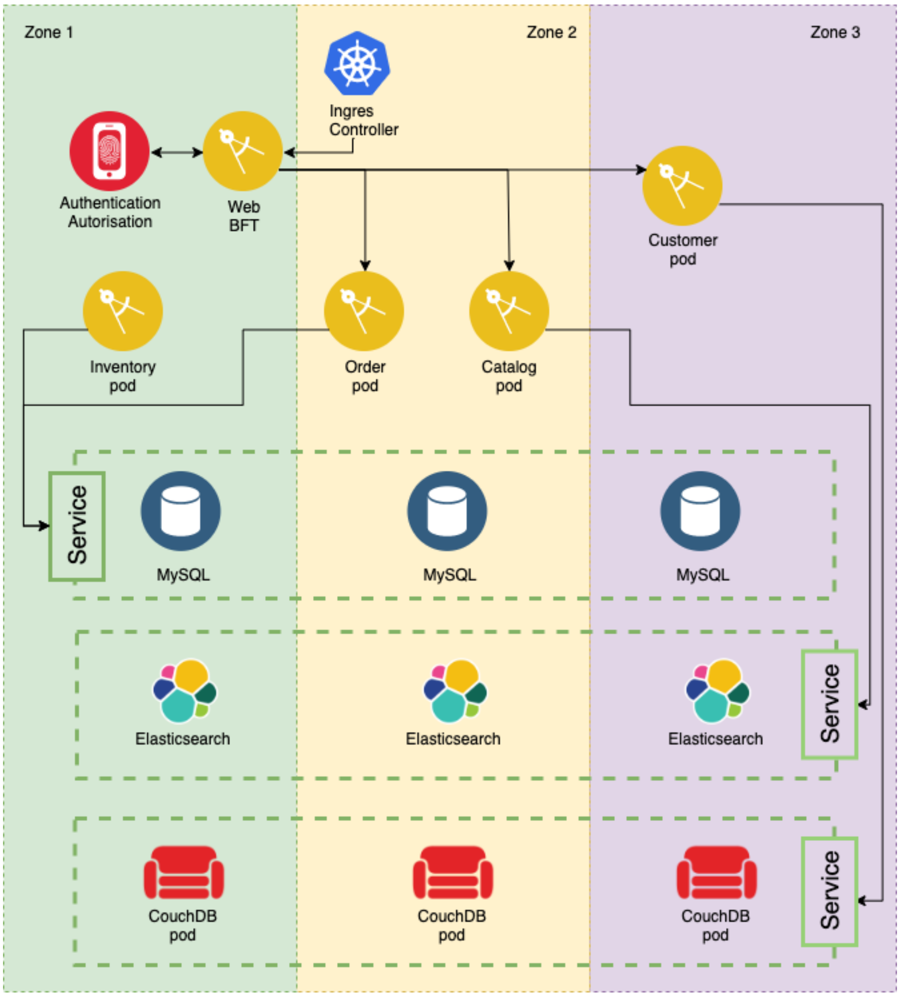
```yaml
elastic.yaml
apiVersion: elasticsearch.k8s.elastic.co/v1beta1
kind: Elasticsearch
metadata:
  name: elasticsearch-sample
spec:
  version: 7.4.0
  nodeSets:
  - name: default
    config:
      # most Elasticsearch configuration parameters are possible to set, e.g:
      node.attr.attr_name: attr_value
      node.master: true
      node.data: true
      node.ingest: true
      node.ml: true
      node.store.allow_mmap: false
    podTemplate:
      metadata: ⋯
      spec:
        containers:
        - name: elasticsearch
          # specify resource limits and requests
          resources:
            limits:
              memory: 4Gi
              cpu: 1
          env:
          - name: ES_JAVA_OPTS
            value: "-Xms2g -Xmx2g"
    count: 3

```

The configuration creates an Elasticsearch cluster that has 3 nodes. The Kubernetes scheduler tries to assign replicas to different zones.

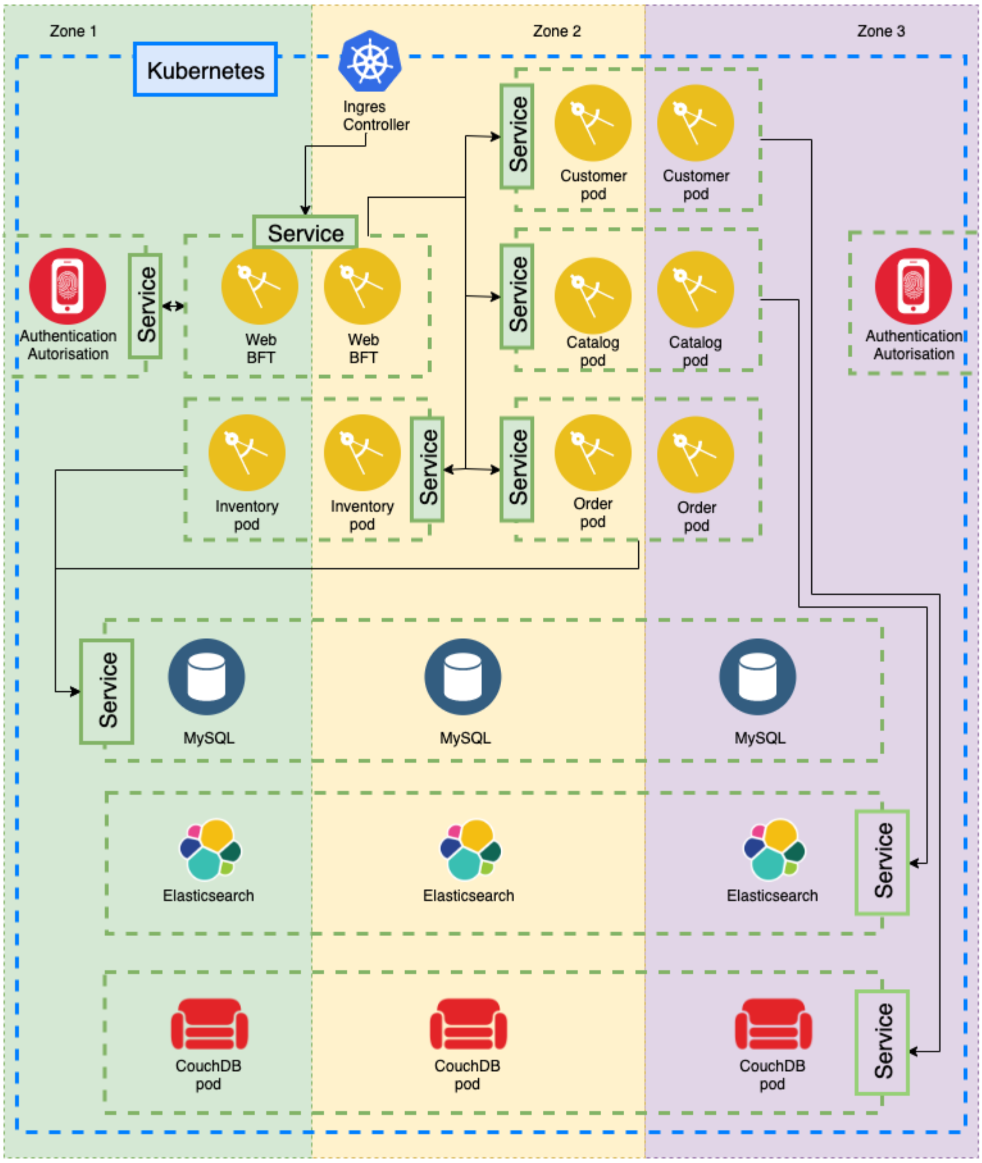
BlueCompute HA architecture for 3-zone cluster

This example shows 3 zones, and each microservice has one replica. If a zone fails, the Kubernetes scheduler can reschedule unavailable microservices to another zone. The 3 databases in the cluster have a more complex configuration. Three different StatefulSets are installed, which configured 3 replicas for each database. This configuration provides resilience and HA, and throughput is increased for all databases.



The application is ready for zone failing, and cluster configuration for databases will save data consistency. But the application doesn’t have zero RTO. Kubernetes still needs some time to redeploy the application to another zone if a failure occurs.

To achieve lower RTO, add replicas to all the services. This example shows the architecture:



This setup requires more resources than the initial one. But if any zone fails, the whole setup is still functional.

If a zone fails, databases are available and you don't lose any data, as the RPO is zero. But Kubernetes can't automatically re-create all the initial Pods for all the databases. The cluster administrator must configure and add those Pods manually or by using scripts. In many cases, Kubernetes users try to move databases out of the cluster. The main benefit of using databases in the cluster is to simplify the management and deployment process. It's important to know each scenario and the related behavior.

Failure scenarios

ID	Scenario	Behavior
1	A stateless Pod fails (authentication, web BFT, inventory, customer, catalog, or order)	The traffic is routed to a replica, and Kubernetes automatically restarts the Pod.

2	The network of a zone fails	Masters that are in the live zones continue to work. Kubernetes automatically discovers the unavailability of workers and reschedules Pods. The network issues must be fixed manually.
3	A full zone fails	
4	A zone is overloaded; not enough resource is in the zone	Kubernetes distributes the workload across the available worker nodes in all the zones, optimizing the workload for each zone
5	One of the workers in a zone fails	See "The network of a zone fails"
6	A zone fails and another zone doesn't have enough capacity	Kubernetes restarts the Pods of the failed zones. The overall cluster survives with decreased performance. Resources need to be added to the surviving zones in the cluster.
7	Application errors occur, but the application is still up and running	This kind of problem must be managed at the application level. Evaluate the usage of probes that are supported by Kubernetes.
12	Pods' limits consume more than the nodes have	Performance is decreased. Nodes need to be added to the surviving zones in the cluster.
13	A MySQL Pod fails	The cluster continues to work with the remaining 2 Pods.
14	An Elasticsearch Pod fails	
15	A CouchDB Pod fails	

Conclusion

Many failure scenarios can happen in a microservices application, but several approaches are effective to help the application survive those failures. This paper showed scenarios where a single cluster is deployed and focused on the failures that occur in the microservices of an application. The paper also discussed multizone deployment. Even a multizone deployment of a Kubernetes cluster is still a single cluster that can survive if a zone is lost.

Authors

Fabio Cerri, Senior Technical Staff Member, Technology Innovation & Automation, Journey to Cloud CoE, Global Technical Services, Delivery & Integrated Operations, IBM Corporation

Igor Khapov, Virtualisation Development Lead, IBM Academy of Technology Member
IBM Systems

Eduardo Patrocinio, Lead Garage Architect, Distinguished Engineer, IBM Garage, IBM Cloud and Cognitive Software, IBM Corporation

Gianroberto Piras, Architect, Global Technical Services, IBM Corporation

Raffaele Pullo, Distinguished Engineer, Hybrid Cloud Architecture, Global Technical Services, Chief Technology Officer Global Technical Services Italy, IBM Corporation