REPLY GLUE

MICROSERVICE INTEGRATION PATTERNS

APPLICATION OF INTEGRATION PATTERNS TO SOLVE MICROSERVICE ARCHITECTURE CHALLENGES

Microservices are the evolution of best-practice application design principles that shape the delivery of solutions to the business in the form of services. Businesses need to change rapidly and constantly upgrade to keep pace with the demands of the customers. Microservices are seen as a bridge to meet such outcomes and many organisations are adopting Microservices for similar purposes.

Having said that, Microservices are not the silver bullet in application development especially if some of the common pitfalls of distributed computing are not carefully trodden over. Scoping the context of operation for a microservice is one complex task and it varies from one organisation to another even if they belong to the same business sector. The process/approach of design might be the same but the end state of microservice layout will vary from business to business. Likewise, building a microservice architecture will have various challenges to overcome. This paper details about the different challenges from an integration perspective in a microservice architecture and different options to address them.

CHALLENGES OF MICROSERVICE ARCHITECTURE

Every organisation's need for architecture maturity varies and depending on that, the path they choose to adopt microservices differs too. Few organisations have the capability to establish a full-scale green field implementation of microservices, where you have the liberty and flexibility to set everything right from the start. But many organisations have the added responsibility to go through a transition phase from a legacy/monolith to modern microservice architecture in a seamless way as much as possible to reach the end state. In both journeys there are challenges that needs to be addressed as part of the newly built microservice architecture.

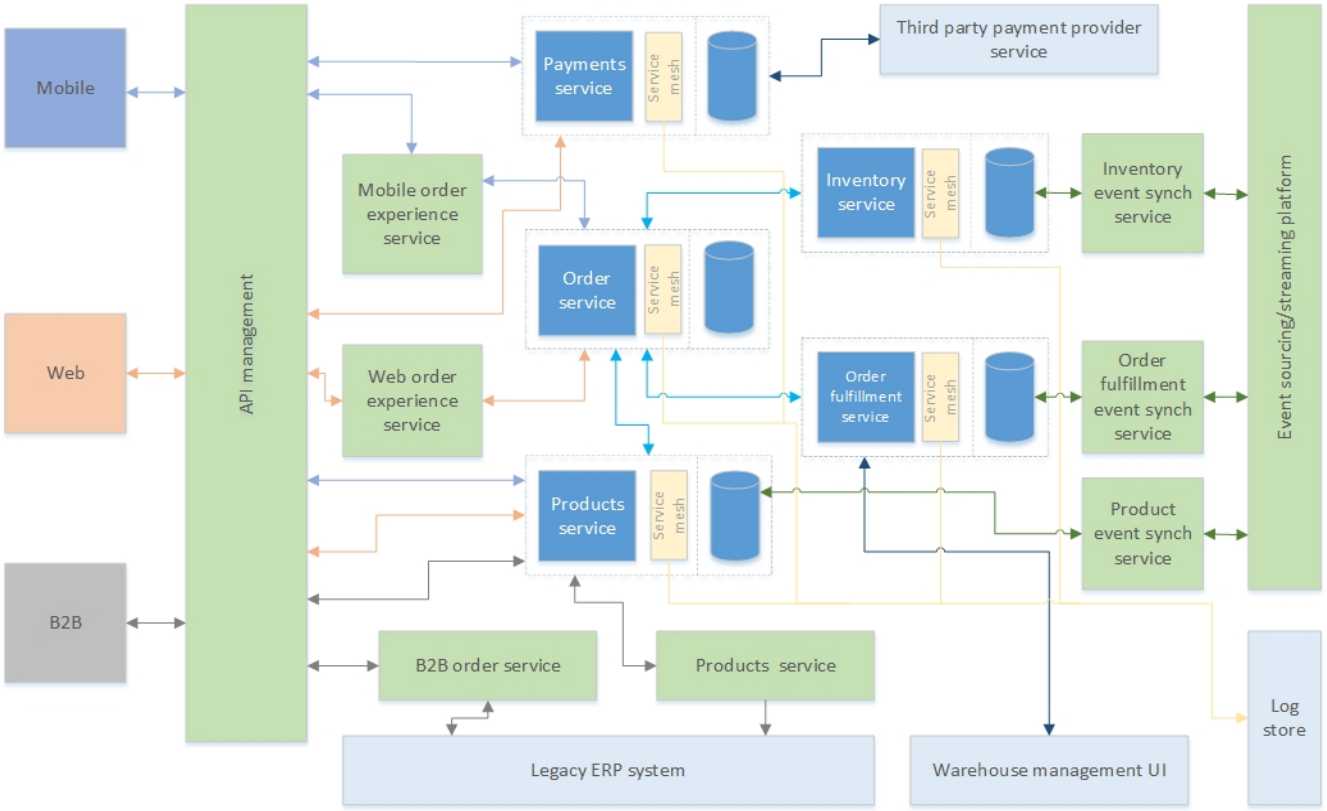
Some of the challenges faced in such an architecture is listed below and this paper applies standard integration patterns to tackle the same:

- Interfacing with legacy system of record
- Modern consumer integration demands
- Seamless integration for existing consumers and supporting incremental shifting from monolith to microservice
- Inter-microservice integration to achieve data consistency and thereby data accessibility for consumers to perform reliable read write operations
- Distributed transaction management across microservices and service composition for unified consumer experience
- Application network resiliency

SOLUTION

Let's take an imaginary use case of a retail business which has an ongoing phased rollout of a multi-channel ecommerce platform backed by a network of microservice applications and a legacy ERP system.

The below figure is a high-level snapshot view of a transitioning microservice architecture. For simple illustration purpose the scope is limited to some of the selected few services that are part of a larger architecture. As represented in the above figure, the retailer has an ecommerce solution built for web and mobile platforms serviced through purpose built microservices like order, payments, inventory, fulfilment, products, etc. Also, the newly built architecture must transition the B2B order management gradually from the legacy system.



Reference architecture of an organisation transitioning to modern microservice design exhibiting integration patterns

INTERFACING WITH LEGACY SYSTEM OF RECORD

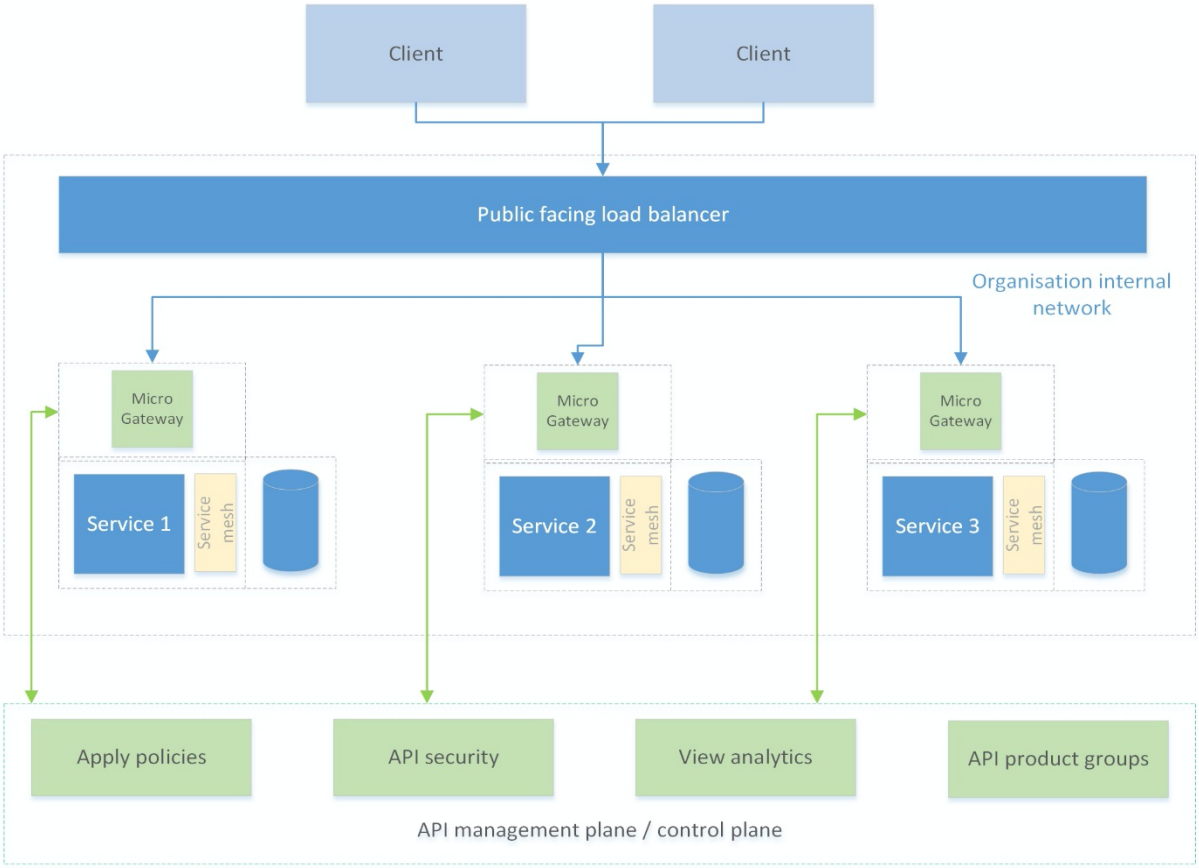
- A well designed microservice should clearly have a bounded data context and should be able to interface seamlessly with other microservices that are part of the platform without any data related issues.
- This works well so long as the interactions are only between the microservices; however, the reality of the situation demands for some functions to be delivered by legacy systems until they are permanently moved.
- In such cases, overloading a particular microservice to deal with the legacy problems like data contract conflicts, interfacing protocol challenges, etc. will be against the design principles and the best way to go about this would be to build an anti-corruption layer i.e., to have an integration micro service that can deal with all the listed legacy issues and the actual micro service application can integrate through it. In the reference architecture, products service and B2B order service is designed and placed accordingly.
- Again, not all the situations will need a façade to integrate both ends to microservice. Especially in Greenfield implementations the legacy is out of the equation and should allow to directly interface as and when required. End of the day it is always extra resources and extra IT footprint to build and manage the facades or integration microservices and so careful assessment is needed looking at clear benefits down the road.

MODERN CONSUMER INTEGRATION DEMANDS

- On the other hand, modern consumers are very demanding and providing them what they need is still a primary function of the microservice. But a microservice designed for global usage to be able to cater to multiple distinct consumer needs will only strain the implementation of the service.
- Hence consumer specific experience providing integration services can solve the problem by abstracting the consumer specific needs to an integration micro service and helps to let the microservice to focus purely on the business logic. In the illustration, mobile order experience service and web order experience service demonstrates the same.
- To summarise at a high level, for non-inter microservice communications especially with legacy or demanding consumer systems, principles of hexagonal architecture design will do the needful by abstracting the non-business specific logic both on the front end and back end from the micro service and keeps them simple and fit for purpose.

SEAMLESS INTEGRATION FOR EXISTING CONSUMERS AND SUPPORTING INCREMENTAL SHIFTING FROM MONOLITH TO MICROSERVICE

- The shift from monolith to microservice is not an immediate change. One of the most common problem to tackle in legacy to cloud native microservice based transformation projects is firstly to support addition of new service, secondly to allow co-existence along with the existing services and then at the end state support fully with the new service.
- This way of incrementally replacing the parts of a monolith with a new service is a very popular design pattern called strangler pattern. Once the new functionality is ready, the old component is strangled, the new service is put into use, and the old component is decommissioned altogether. In the above illustration, the order service is split into two basically one dealt through a newly built microservice catering mobile and web consumers whereas the other order service specifically takes care of B2B orders managed by a legacy backend. The B2B order service will be expected to function until the complete B2B order management piece is moved to be within the responsibilities of the order microservice.
- In terms of supporting a seamless integration with consumers of a microservice, API Gateway plays a key role solving many critical problems. The fundamental principle of design for a microservice architecture is to be flexible to change and at the same time be highly available and scalable. Change causes version upgrades to the microservice and sometimes the new versions can be non-backward compatible. In such situations API gateway routing enables teams to run and serve multiple versions of the microservice matched for different consumers based on their compatibility.



Microgateway – Decentralised API gateway

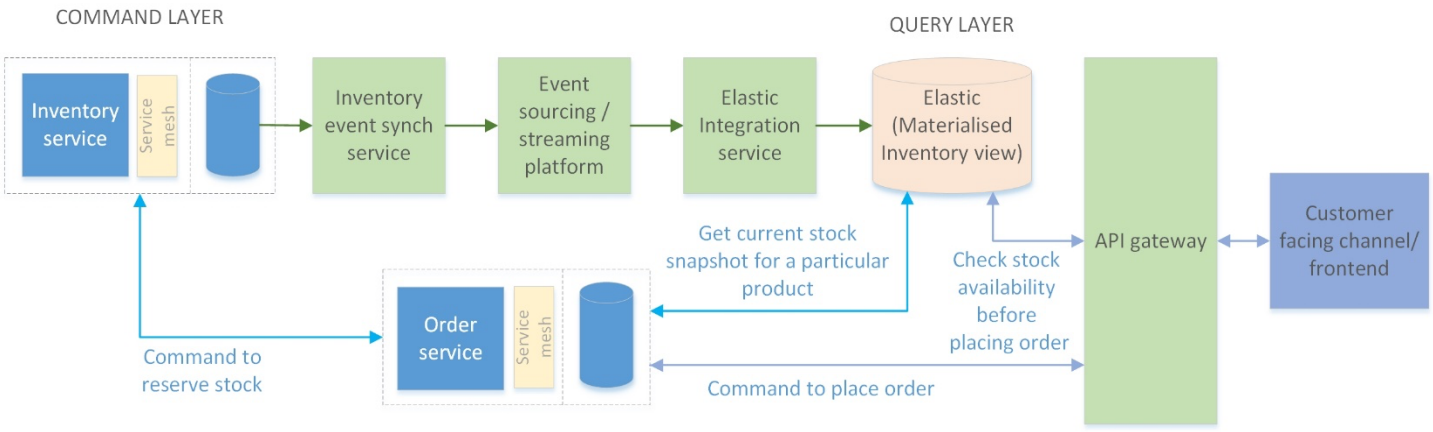
- Having talked about the importance of API gateway in a microservice architecture, the API gateway itself needs to be chosen and built very carefully as it could easily become a single point of failure/a complex integration monolith. After doing all the heavy lifting of distributing the application logic across a number of microservices, leaving the API gateway as vulnerable point can be daunting. But the concept of having a central management plane/control plane for the API management separate and then distributing the actual runtime components as micro gateways that are attached to the microservices is a popular design and makes more sense to the distributed, highly available and scalable qualities of a microservice platform.
- Distribution of micro service functions across multiple servers can cause of maintenance overheads for routine activities like certificate upgrades, whitelisting clients, etc. The API gateway can be an apt place to offload these routines and can add more security and single point of access to the platform and additionally governs the exposure of the microservices by allowing to apply various API management policies.

INTER-MICROSERVICE INTEGRATION TO ACHIEVE DATA CONSISTENCY AND THEREBY RELIABLE DATA ACCESSIBILITY

- Distributed systems need to talk to each other to deliver a business function. Decomposition of massive application into fine grained microservices only raises the demand for more reliable integrations between them that are real-time/near real-time. Every microservice has its defined data boundary of operation but might still be interested in certain specific events or data from other closely related data stores managed and owned by a different microservice to deliver a simplified experience to its consumers. The need is to keep the relevant data consistent

between the respective bounded data layers of the microservices. A separate platform that can allow events publishing, subscribing and streaming is needed as part of the architecture.

- This data consistency needs to be achieved by events synchronization between systems with reliability, sequencing and replay capabilities. Event sourcing pattern can solve the problem. It is an approach to handle operations on data that's driven by a sequence of events, each of which is recorded in an append-only event data store. Each event represents a set of changes to the data performed by the service that publishes the event.
- For instance, picking an illustration of event sourcing from the reference architecture would be a case of an Inventory microservice being interested in receiving updates on fulfilment progress dealt in order fulfilment service to make near real time adjustments to the available stock of the particular product. In this case, the sequence of the events as well as history of changes is critical and hence the events are appended to an event store. Both the publishing to event store and subscribing from an event store though not a compulsion can be dealt through a data bounded integration service depending on the interfacing needs and challenges as detailed in earlier sections.
- Microservices are exposed to consumers to read and update data. Read and write workloads are often asymmetrical, with very different performance and scale requirements. Using the same data layer to execute read and write operations will have technical challenges like concurrent updates, record lock, etc. In such cases the event data store and event sourcing mechanism can be extended to create a replicated view if the read data from the event data store that is built as part of the event sourcing logic. The read data layer can sometime not just be a mere replication of the source data layer and instead can be materialized with enrichment of data from other event stores and presented in a scalable and highly available data layer. This way of achieving low latency and high-performance querying and update requests on microservices is termed as CQRS pattern.
- In the same inventory sync case, as and when change of state occurs to the inventory, the event is a subject of interest for order service and other customer front facing applications to get an accurate snapshot of stock. The below illustration shows how event sourcing can support to enable CQRS pattern. The inventory change events are sourced through an event store into a scalable and highly available elastic database which is used as a query only database. Whereas the update inventory requests like reserve stock are managed on the inventory microservice directly thereby provisioning options to independently scale the services as well as reducing the transaction management complexities.



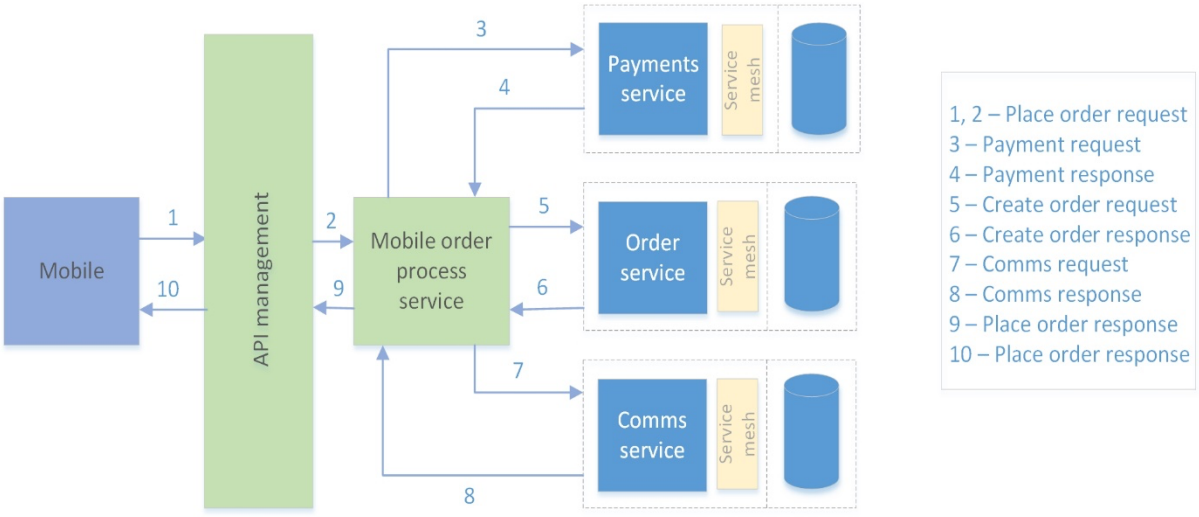
Command Query Responsibility Segregation enabled through event sourcing – Illustration

- Similarly, creation of new product/deletion of an existing product is a subject of interest for the Inventory service to subscribe in order to store the product IDs and against which the stock quantity is primarily maintained.

- Event sourcing pattern using streaming platforms can help build efficient materialized views within the event processing platform and this can provide great flexibility for consumers.
- CQRS and event sourcing techniques can have some delay in achieving real-time consistency and so are suitable for eventual consistency accepted scenarios only. In some situations, the delay caused in data consistency might trigger incorrect or duplicate command requests and the services as well as the client should be designed capable of handling such edge case scenarios.

DISTRIBUTED TRANSACTION MANAGEMENT AND SERVICES COMPOSITION

- A transaction is a grouping of operations that are guaranteed to all complete, or none at all. Transactions are implemented to meet ACID guarantees. In a distributed microservice architecture, transactions are required across service data layers to deliver a single functional request from a consumer perspective. Composing multiple services to provide a unified function to consumers/clients is a common ability in API led architectures.
- The traditional 2 phase commit pattern is too chatty and is not a scalable option. Firstly, it needs each participating system to support the 2-phase commit and then introduces locks during the execution of the transaction in all systems and the performance degrades as the number of systems grows.
- In the reference illustration, consider the case of placing the order only after payment is successful. This needs an orchestrator service to first call the payments service, get the response back and validate if the response was successful and then proceed further to making a request to order service to place the order. The question is who exactly needs to take the responsibility of orchestration. Either the client or an intermediate integration service should encapsulate the backend.

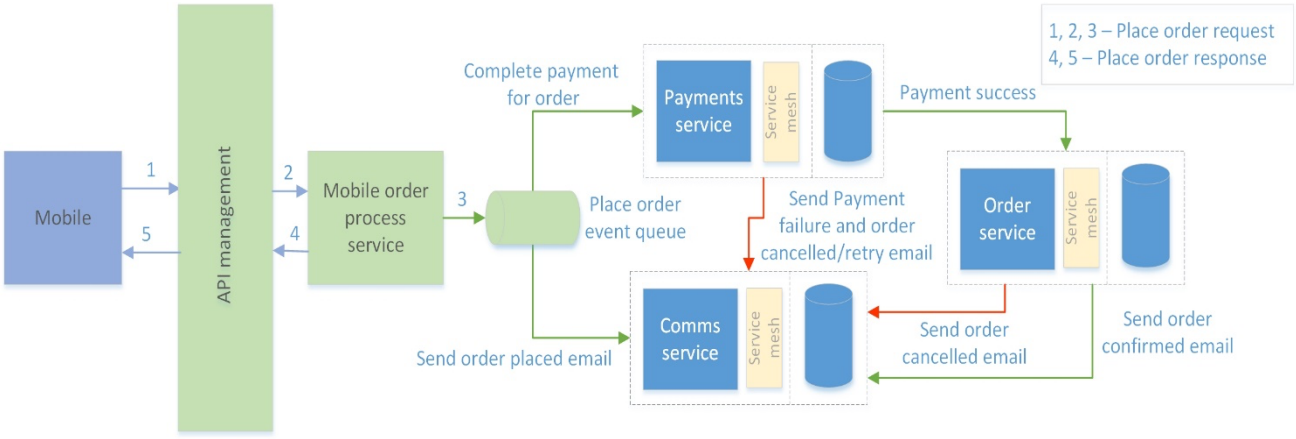


Orchestrator SAGA pattern

- In some cases, the orchestration can be a simple problem provided the interfacing and the data models are smoothly compatible. If the client is already proxied from the backend through an API management capability then the orchestration can be configured in the gateway to provide a singular request response experience to the client. The gateway can also aggregate multiple service invocation responses made in parallel and give a single response back to the client.
- Even though Gateway aggregation/orchestration can be a viable solution, the key points to consider are to not overwhelm the gateway service with high resource requiring operations and at the same time it can be used for

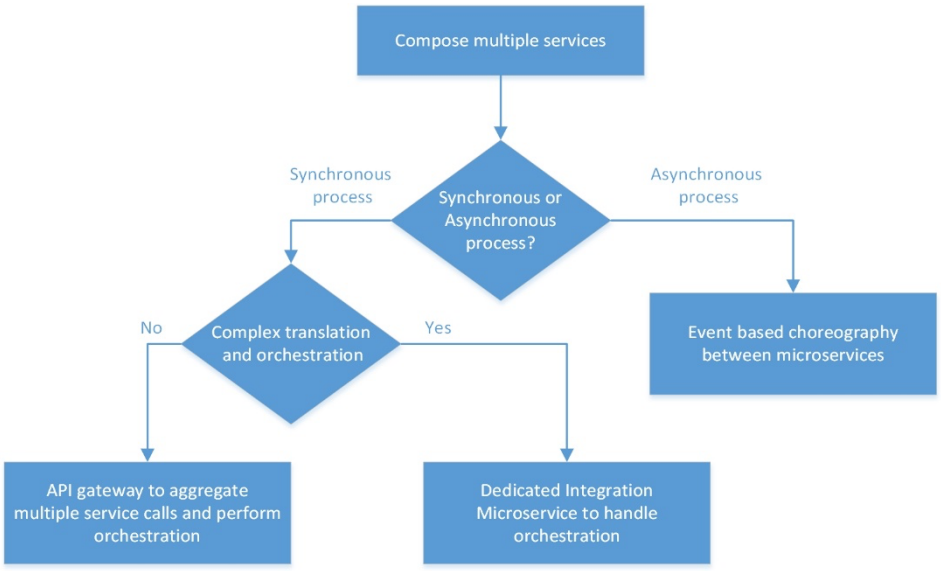
light weight interactions. One another key problem to solve when multi service interfacing is dealt would be to design the fault tolerance side of functions to be reactive and resilient. More details are discussed further under networking fault tolerance section in this paper.

- Integration micro service orchestrating the sequence of service calls is advised in situations where gateway is not in the picture and whereas a microservice itself needs some real-time information from another service for needs like data enrichment, check real-time status of entities, etc. Also, cases, where the microservice shouldn't have to deal with the burden of data semantic conflicts, etc. while interfacing with the other service, an integration micro service to provide the composing function is advised.
- Another pattern to achieve the same is by applying Event choreography pattern. It is completely decoupled and highly scalable solution based on events. The service executes a transaction and then publishes an event to trigger a consequential action in the next service. In case of failures, the services trigger compensating transactions to correct the previously committed services actions. This is illustrated in the below figure.



Event Choreography SAGA pattern

- The main complexity with these saga patterns as opposed to XA transactions are that they can introduce complex programming models into the architecture. The orchestrator pattern if not provisioned with right resourcing can become a single point of failure and introducing changes to event choreography pattern is tedious.



Simple Decision tree to choose right service composition pattern

APPLICATION NETWORK RESILIENCY

- On various situations, like we discussed in the above sections, multiple microservices work together to deliver complex enterprise business functions. Integrating or connecting all the applications, services and data will form an application network and that needs to be resilient to ensure correctness and fail-safe features in a distributed service-based environment. While resiliency is one of the critical essences of a microservice architecture, it needs to be fulfilled at different levels like infrastructure, integration, within the core service logic of a microservice, etc.
- Some of the common communication related problems/patterns seen in an application network are latency, connection failures, timeouts, retries, circuit breaking, fault isolation, etc. Every microservice that is part of the application network has to overcome these problems. Also, there are certain peripheral tasks like configuration, audit logging, monitoring, etc. that are required in each microservice.
- The approach to solve this requirement is to let each microservice build their own logic in very few cases or apply a sidecar/ambassador pattern with the help of an attachment layer like service mesh as a standard in each microservice.
- In the reference architecture illustration, all microservices are attached with a service mesh layer and one of the functions (audit logging) is indicated to be fulfilled by it.
- The service-mesh pattern overcomes some of the challenges in integrating microservices but only offers the commodity features of inter-service communication, which are independent from the business logic of the service, and therefore any application-integration logic related to the business use case should strictly be implemented at each service level.
- Before building a function into a sidecar/service mesh, it is worth considering to see if it would work better as a separate service or a more traditional daemon or whether the functionality could be implemented as a library or using a traditional extension mechanism. Language-specific libraries may have a deeper level of integration and less network overhead.

CONCLUSION

With the segregation of monolithic applications into microservice and cloud-native applications, the requirement to connect these apps is becoming increasingly challenging. The services and applications are dispersed across the network and connected via disparate communication structures. Realizing any business use case requires the integration of the microservices. As a result, cloud-native application integration is one of the most critical yet largely concealed requirements in the modern era of microservices and cloud-native architecture.

Accordingly, it is important to select the most appropriate technology for building integration-savvy services and minimize the development time required to weave together services. Several open-source, cloud based frameworks and IPaaS technologies are in the market to fulfil these application-integration needs in the cloud-native landscape, which we need to evaluate against each specific use case to choose the best fit.

REFERENCES

- https://azure.microsoft.com/en-us/blog/design-patterns-for-microservices/
- https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture/
- https://martinfowler.com/bliki/StranglerFigApplication.html
- https://martinfowler.com/microservices/
- https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749
- https://database.guide/what-is-acid-in-databases/#:~:text=In%20database%20systems%2C%20ACID%20(Atomicity,occur%20while%20processing%20a%20transaction.
- https://en.wikipedia.org/wiki/CAP_theorem