Microservices Orchestration vs. Choreography: A Decision Framework

Alan Megargel
School of Computing and Information Systems
Singapore Management University
Singapore
alanmegargel@smu.edu.sg

Christopher M. Poskitt
School of Computing and Information Systems
Singapore Management University
Singapore
cposkitt@smu.edu.sg

Venky Shankararaman
School of Computing and Information Systems
Singapore Management University
Singapore
venks@smu.edu.sg

***Abstract*—Microservices-based applications consist of loosely coupled, independently deployable services that encapsulate units of functionality. To implement larger application processes, these microservices must communicate and collaborate. Typically, this follows one of two patterns: (1) choreography, in which communication is done via asynchronous message-passing; or (2) orchestration, in which a controller is used to synchronously manage the process flow. Choosing the right pattern requires the resolution of some trade-offs concerning coupling, chattiness, visibility, and design. To address this problem, we propose a decision framework for microservices collaboration patterns that helps solution architects to crystallize their goals, compare the key factors, and then choose a pattern using a weighted scoring mechanism. In cases where there is no clear preference, a hybrid pattern is suggested which inherits some strengths of both choreography and orchestration. We demonstrate the framework by evaluating the needs of three industry case studies (Danske Bank, LGB Bank, Netflix), showing that it leads to appropriate patterns being suggested. We are not aware of any existing decision frameworks to guide solution architects in choosing a microservices collaboration pattern.**

Keywords—microservices, orchestration, choreography, event-based, invocation-based, service-oriented architecture

I. INTRODUCTION

Microservices constitute an implementation approach to Service-Oriented Architecture (SOA) principles and patterns, with emphasis on service development and deployment using modern software engineering tools and practices [1]. Microservices are characterized as being modular, small in size, independently deployable, and organized around business capabilities [2]. These characteristics allow for the components of complex applications to be independently monitored, tested, updated, or scaled; benefits that led to the adoption of microservices by companies such as Amazon, Netflix, and Uber. Microservices have also been deployed in more traditional enterprises, such as banking, due to their effectiveness at integrating or replacing parts of monolithic legacy systems [3].

Decomposing a complex application into a collection of microservices also introduces some challenges, most notably, in determining how they should *collaborate* in order to implement different application processes. What might be a simple language-level function call in a monolith is instead some network-level communication between services. Should that communication then be synchronous, e.g. an HTTP invocation of a RESTful API; or should it be an asynchronous message exchanged over a publish/subscribe protocol? Should the individual microservices embed the 'larger' application process logic, or should this concern be separated and managed in a different part of the application?

In practice, microservices typically collaborate according to one of two different design patterns: *choreography* or *orchestration* [2], [4]. Using choreography, microservices communicate asynchronously by publishing events via a message broker; there is no central controller. Microservices must subscribe and set themselves up for the next iteration of the application process. Using orchestration, a *composite* microservice is introduced to act as a controller, synchronously invoking other microservices using a request/reply protocol to manage the steps of the application process. In orchestration, the microservices do not embed any knowledge of the application process flow, leaving it solely as the responsibility of the controller. Choosing the right collaboration pattern boils down to resolving a series of trade-offs resulting from their different communication styles. Microservice choreography leads to solutions with less coupling and less chattiness, whereas orchestration leads to solutions with better process flow visibility.

In this article, we address the problem of identifying which of the microservice collaboration patterns is best suited to the needs of a given problem or application process. We propose a *decision framework* to help identify the goals of the solution architect, the factors that are most important to them, and then guide them to an appropriate choice of collaboration patterns using a weighted scoring mechanism. In cases where there is no clear preference, a 'hybrid' pattern is suggested which inherits some strengths of both choreography and orchestration. We demonstrate the framework to evaluate the needs drawn from three industry case studies—*Danske Bank* [5], *LGB Bank (anonymized)*, and *Netflix*—showing that the considered factors and scoring mechanisms lead to appropriate collaboration patterns being suggested. Finally, we discuss the management implications arising from using the decision framework.

978-1-6654-3579-6/21/$31.00 ©2021 IEEE

II. Related Work

In this section, we review related work in the area of microservices design challenges, and the use of decision frameworks to guide solution architects in overcoming these challenges. First, we review the fundamental microservice design principles. Then we review microservice design challenges related to service composition and documentation, followed by a review of the two dominant microservices collaboration patterns – choreography and orchestration. Finally, we review related work on architecture decision frameworks.

A. Microservice Design Principles

In order to achieve the agility and scalability goals of a microservices-based architecture, the design of each microservice should be guided by the following fundamental design principles [3]:

1) Do One Thing Well: "Microservices should be highly cohesive [6], [7], in that they encapsulate elements (methods and data) that belong together. A microservice has a specific responsibility enforced by explicit boundaries. It is the only source of a function or truth; i.e. the microservice is designed to be the single place to get, add, or change a 'thing', e.g. a customer, or a product". Domain-Driven Design (DDD) [8] has proven to be a good approach for identifying optimal microservice granularity and bounded context [9].

2) No Bigger than a Squad: "Each microservice is small enough that it can be built and maintained by a squad (small team) working independently. A single squad / team should comfortably own a microservice, whereby the full context of the microservice is able to be understood by a single person". A squad should be no more than 10 people [10].

3) Don't Share Data Stores: "Only one microservice is to own its underlying data [6]. This implies moving away from normalized and centralised shared data stores. Microservices that need to share data, can do so via API interaction or event-based interaction".

4) Independent Release Cadence: "Microservices should be loosely coupled [7] and therefore should have their own release cadence and evolve independently. It should always be possible to deploy a microservice without redeploying any other microservices. Microservices that must always be released together could be redesigned and merged into one microservice".

Following the above design principles, small-sized, loosly-coupled, and independently depoyable microservices are more suitable for cloud deployent as compared to monoliths [3]. Small-sized deployment objects also makes containerization (e.g. using Docker) practical, as well as continuous integration (i.e. scripted build, test, and deploy) using agile DevOps tools and methods [1]. Cloud-based microservices are typically exposed to external third parties as well as to internal user interfaces, via an API Gateway. A conceptual microservices-based architecture is illustrated in Fig. 1 below. As an architectural principle, the user interface should only contain presentation logic, and all of the business logic and data should reside in the microservices layer [3].

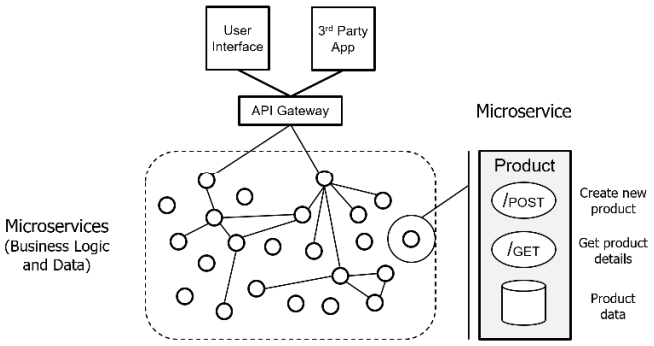

Fig. 1. Conceptual Microservices-based Architecture [3]

B. Microservice Design Challenges

Organizations are increasingly challenged to migrate from a monolithic application architecture to a microservices-based architecture [3]. SOA implementations without microserves have challenges related to monolithic deployment, and bounded data modeling, along with a complex protocol stack for implementing solutions. An Enterprise Service Bus (ESB), an SOA design pattern, does however provide a centralized view of application process flows. On the other hand, a microservices-based architecture provides greater autonomy of services, reduces data structure dependencies, and provides elastic service scalability through individual service instantiation. A microservices-based architecture does however sacrifice the centralized view of application process flows, which is now distributed across services [4]. With or without microservices, a common challenge of SOA implementations is the collaboration of multiple services to fulfill a 'larger' application process [11].

A microservices-based architecture has desirable characteristics which support continuous integration (i.e. an automated build, test and deploy of individual microservices) such as modularity, scalability and separation of concerns [12]. However, because of the modular and independent nature of microservices, the visibility of an application process becomes challenging in terms of process documentation [13]. For example, one user-triggered request is no longer processed by a single monolithic system: rather it is processed by many microservices collaborating together to fulfil the end-to-end application process. A recent survey of architects across multiple industries revealed the challenges of documenting microservices-based applications to be: a) "documentation is mainly achieved manually", b) "documentation is wrong or out-of-date", c) "documentation is incomplete", d) "no appropriate visualization for different stakeholders", and e) "tools' lack of providing runtime Key Performance Indicators (KPI)" [13].

Microservice containerization and container orchestration is well defined by the tool providers, e.g. Docker and Kubernetes. Container orchestration tools simplify the management of container-based systems using features such as deployment automation, auto-scaling and self-healing. Yussupov [12] leverages the classic Enterprise Integration Patterns (EIP) [14], Pipes and Filters, to propose a "pattern-based microservices composition meta-model" for designing container-based applications.

C. Microservices Collaboration

Microservices collaboration can be defined as the action of a number of microservices working together in order to satisfy a business need [15]. In a monolithic application, all the functionalities required for fulfilling the business need are found within the code segment of a large application, as illustrated in Fig. 2 below. The business need is satisfied through the interaction between different functions, which is achieved through inter-process communication, usually between the various code segments running on the same hardware platform (server) using function or language-level method calls [2].

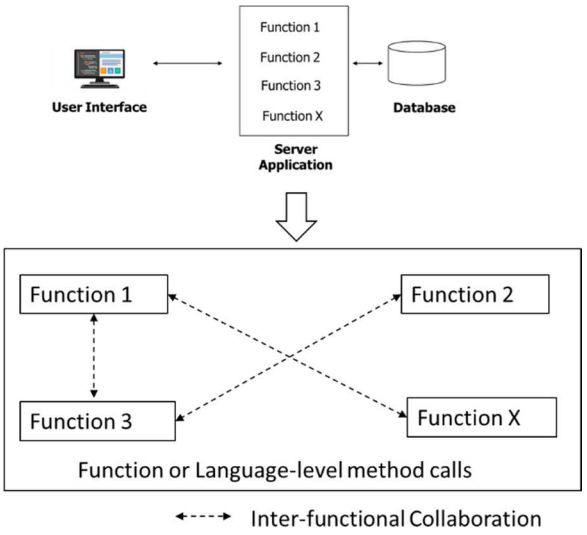

Fig. 2. Inter-Functional Collaboration in a Monolithic Application

However, in a microservices-based architecture, an application is comprised of multiple microservices, each executing a part of the required functionality, as illustrated in Fig. 3 below. Rather than language-level method calls as in a monolith, microservices communicate via standards-based protocols, e.g. HTTP for invocation-based communication, or a message-based communication via a message broker. With each microservice encapsulating a single application function (business logic), instantiated independently in a decoupled way, an abstract mechanism is needed to coordinate the communications between microservices in order to execute the end-to-end application process.

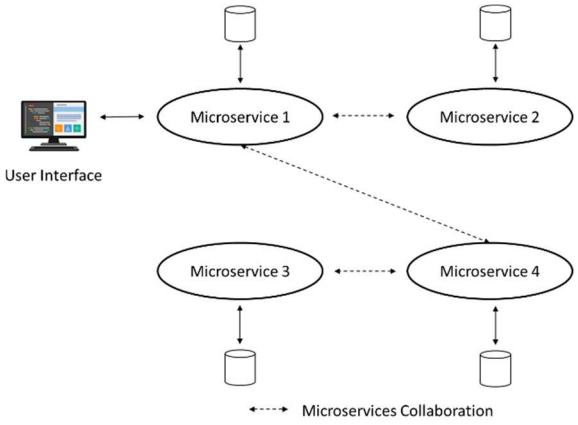

Fig. 3. Microservices Collaboration

In a microservices-based architecture, the business need can only be satisfied through collaboration between the microservices [2]. For example, the business need of an order fulfilment process for a web-based retailer may require three microservices to collaborate, namely payment, inventory, and shipment [15]. The payment microservice is responsible for ensuring the payment is processed, the inventory microservice is responsible for reserving the items ordered and updating the inventory, and the shipment microservice is responsible for ensuring the items ordered are shipped to the customer. This collaboration can be designed using two different architectural patterns; either the choreography pattern, or the orchestration pattern [3].

D. Decision Frameworks

Decision frameworks provide a way to facilitate and enhance decision making by assisting the solution architect in deciding on the technology and architecture of a solution for a given business requirement [16]. While decision frameworks vary in design and purpose, they generally address three common elements which include: a) helping to identify clear goals, b) illuminating key questions that help decision participants to scope problems, and c) providing support to make a final choice having considered the pros and cons of each option. Scoring mechanisms are typically used to help compare the options. For example, Griffin [17] developed a decision framework to assist solution architects in deciding on the technology best suited to support decentralized control of a distributed business process.

Goossens [18] has developed a high-level decision framework for helping organizations make well-supported software architecture design decisions regarding three categories, namely communication between microservices, integration, and management of the microservices. This framework provides support on how to solve the three challenges on a conceptual level but does not provide any support in translating these findings into designing a working solution. The framework reported in this paper attempts to focus only on one challenge, namely microservices collaboration, and provides a method for comparing the different collaboration patterns and choosing the most suitable pattern for a given business requirement.

While various authors have compared microservice collaboration patterns descriptively (e.g. Richardson [2], Cerny [4], Haj Ali [19]) or experimentally (e.g. Singhal [20], Rudrabhatla [21]), we are not aware of any existing decision frameworks to guide solution architects in choosing a microservice collaboration pattern.

III. Microservice Collaboration Patterns

This section covers the microservice collaboration patterns which are commonly used, as follows:

A. Microservice Choreography Pattern

Using this collaboration pattern, there is no ‘controller’ of the end-to-end application process flow. Instead, microservices will publish an event (via a message broker) whenever there is a state change, whereby other microservices subscribing to that event then set themselves

up for the next iteration of the process. Fig. 4 below illustrates an example of microservice choreography, for an order management application using multiple microservices.

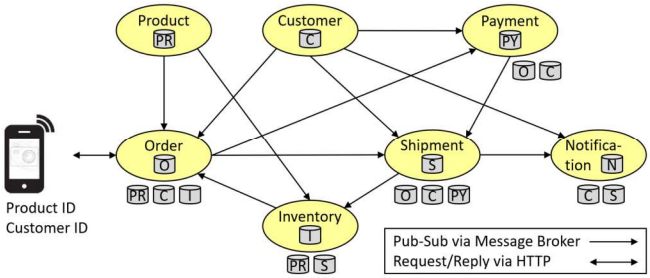

Fig. 4. Example of Microservice Choreography

In the above example, whenever the Customer microservice changes state, e.g. a customer record is created or updated, then it publishes an event (message) via a message broker. The Order microservice subscribes to the event, because it needs to know the customer's name; the Payment microservice subscribes to the event because it needs to know the customer's credit card details; the Shipment microservice subscribes to the event because it needs to know the customer's mailing address; and the notification microservice subscribes to the event because it needs to know the customer's email address. For event-based collaboration to work, all of these microservices need to know the new customer information before the next order occurs. Note that there are five copies of the Customer database, in this example.

Strengths of the choreography pattern include the following [2], [4], [19]:

- *Loosely Coupled*
  Microservices can be deployed independently. Any active process data will be queued up in the message broker during deployment time, so there will be no interruption.
- *Low Chattiness*
  Data is exchanged between microservices only if there is a state change. This pattern is suitable for microservices which are deployed across the wide area network.

Weaknesses of the choreography pattern include the following [2], [4], [19]:

- *Poor Process Visibility*
  End-to-end processes are difficult to monitor, e.g. the runtime state of a process. Furthermore, point-to-point connections can lead to 'spaghetti' architectures which are inherently unmanageable. This pattern is less suitable for complex processes which involve a large number of microservices.
- *Complex Design*
  Due to the poor visibility of end-to-end processes, and the need for a message broker to intermediate the process flow, the design of applications becomes relatively complex.
- *Poor Reusability (Weak Atomicity)*
  Microservices must maintain copies of databases (or database tables), other than the one that they own. Weak atomicity makes microservices less reusable for assembly into other applications.
- *Indeterminate Response Time*
  If a microservice goes offline temporarily during a process iteration, the process will eventually complete when the microservice goes back online. This feature makes the response time indeterminate, and therefore this pattern is not suitable if a user interface needs an immediate response.

B. *Microservice Orchestration Pattern*

In this collaboration pattern, a composite microservice acts as the 'controller' which orchestrates the end-to-end application process flow by invoking multiple atomic services in a sequence. Microservice invocation is done via request/reply interaction, during the process execution. Fig. 5 below illustrates an example of microservice orchestration, for the same order management application described above.

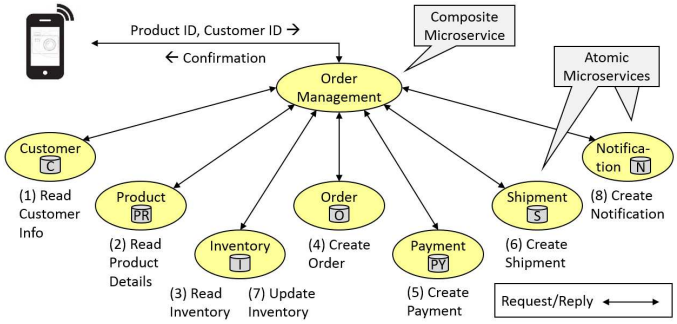

Fig. 5. Example of Microservice Orchestration

In the above example, invocation-based request-reply interaction is used instead of event-based publish-subscribe interaction. Composite microservices may invoke any other composite or atomic microservices, in order to orchestrate an application process. Atomic microservices should never invoke each other. Atomic microservices have exclusive access to their own data. Note that there is only one copy of each database, in this example.

Strengths of the orchestration pattern include the following [2], [4], [19]:

- *Clear Process Visibility*
  End-to-end processes are easy to monitor, e.g. the runtime state of a process. This pattern is more suitable for complex processes which involve a large number of microservices.
- *Simple Design*
  Due to the clear visibility of end-to-end processes, and point-to-point style of invocation-based communication, the design of applications becomes relatively simple.
- *High Reusability (Strong Atomicity)*
  Microservices encapsulate a single entity (object), and have exclusive access to data that they own. Strong atomicity makes microservices more reusable for assembly into other applications.

- *Predictable Response Time*
  Each step in the application process flow uses invocation-based request-reply interaction. This makes response times predictable, even for error responses, therefore this pattern is suitable if a user interface needs an immediate response.

Weaknesses of the orchestration pattern include the following [2], [4], [19]:

- *Tightly Coupled*
  Microservices can be deployed independently, but require downtime during deployment in order to avoid interruption of the application process flow. This can be overcome by using an active-active load-balanced configuration.
- *High Chattiness*
  Data is exchanged between microservices during each step of the application process flow, therefore this pattern is not suitable for microservices which are deployed across the wide area network.

C. Choreography with a Process Engine Pattern

This final collaboration pattern is essentially a hybrid of the other two, in that it combines the asynchronicity and flexibility of choreography with the process visibility of orchestration [22]. Instead of having atomic microservices subscribing to state changes in each other, a composite microservice (the 'controller') becomes responsible for ensuring the execution of steps by publishing and subscribing to events. Fig. 5 below illustrates an example of this hybrid pattern for the same scenario as described earlier, i.e. propagating an update of some customer's details.

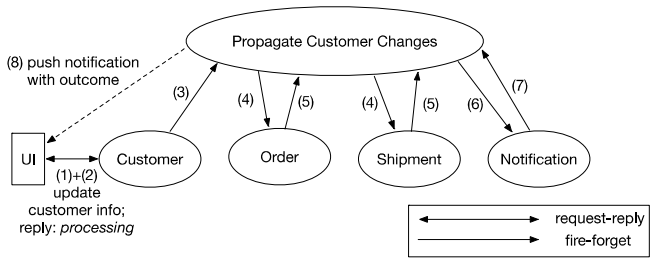

Fig. 5. Example of Choreography with a Process Engine

In the above example, an invocation-based request-reply from the UI to the Customer microservice is used to update a customer's details. The microservice replies to indicate that it is processing the update, then publishes an event (message) via a message broker to indicate its state has changed. The controller (Propagate Customer Changes) subscribes to this event, and in turn publishes its own events that are subscribed to by atomic microservices that need to know about the changes. The controller has full visibility of the end-to-end process, and is able to perform any necessary actions once the process has been completed (or fails), e.g. publishing a push notification to the web application.

In our example, the controller is implemented as a composite microservice, but it could alternatively be implemented using the worklist and queue services of Conductor, Netflix's workflow orchestration engine [23]. This 'hybrid' pattern inherits some strengths of both choreography and orchestration. For example, it is loosely coupled due to the use of brokers, yet it retains process visibility. However, the chattiness is higher than in pure choreography, and while the introduction of a controller does help simplify the design, it also introduces a potential bottleneck in terms of scalability [24].

IV. Decision Framework

This section describes a framework for deciding which microservices collaboration pattern to use. The framework employs a set of six collaboration factors derived from the previous section, including: a) distribution of microservices across the wide area network (WAN), impacting 'chattiness', b) predictability of response time from the application process, c) loose coupling of microservices to the process, impacting deployment, d) reusability (i.e. atomicity) of microservices, e) complexity (i.e. number of microservices in the application process), and f) runtime visibility of the application process flow.

As a first step, the company's architecture team, or lead architect, is meant to assess the general ability of each microservices collaboration pattern to handle/provide the six collaboration factors, and then establish this assessment as an internal Technology Architecture Standard to guide on-going solution architectures. Here, the lead architect needs to answer the question, how capable is each collaboration pattern (choreography and orchestration) in satisfying each of the six collaboration factors? In terms of 'Ability', each collaboration factor is to be assessed using a five-point Likert scale; 1-Incabable, 2-Slightly Incapable, 3-Neutral, 4-Capable, 5-Very Capable. A worked example is provided in Table 1 below.

TABLE I
Assessment of Ability (Worked Example)

	Ability (1-5)	
Ability to Handle/Provide	**Choreo-graphy**	**Orches-tration**
Distribution Across Networks - µ-services deployment across the WAN	5	2
Predictability of Response Time - from an application process	3	5
Loose Coupling (Hot Deployments) - deployment without process interruption	5	3
Service Reusability (Atomicity) - degree to which µ-services are reusable	1	5
Design Complexity - number of µ-services in an application process	2	5
Runtime Process Visibility - ability to monitor the runtime state of a process	2	4

For each on-going solution, the solution architect together with business users are meant to assess the priority of each of the six collaboration factors, in satisfying the business need of the solution. In terms of business 'Priority', each collaboration factor is to be assessed using a five-point Likert scale; 1-Not a Priority, 2-Low Priority, 3-Medium Priority, 4-High Priority, 5-Essential. For each of the six collaboration factors, the assessed priority is then multiplied by the assessed ability for each collaboration pattern. The total score for each collaboration pattern then serves as an

indication to the solution architect as to which collaboration pattern should be selected for the solution.

Three worked examples of the framework are provided in the following subsections. Here, we are demonstrating the use of the framework on known industry cases. The first case, Danske Bank, results in choreography as the selected collaboration pattern. The second case, LGB Bank, results in orchestration as the selected pattern. The third case, Netflix, results in no clear pattern preference and recommends a 'hybrid' collaboration pattern. Note: we are simply demonstrating the use of the framework here, and we will leave a real evaluation of the framework involving industry partners to future work.

A. Danske Bank Case

Our first case study is the Danske Bank Foreign Exchange (FX) Core system, a legacy monolithic solution that was recently re-implemented using microservices [5]. Some key requirements of the case study included: a) that all communication should be asynchronous, b) services should be loosely coupled, c) ability to deploy in private data centers and eventually private clouds, and d) no explicit requirement for FX Core microservices to be reused in other applications.

A worked example of the Danske Bank assessment is shown in Table II below, which indicates that a choreography collaboration pattern should be selected. This outcome is predominantly due to the business needs for loose coupling and distribution across networks: choreography is 'very capable' (5) for both, and they have also been scored as 'essential' (5) for Danske Bank owing to requirements 'b' and 'c' respectively.

TABLE II
DANSKE BANK ASSESSMENT (WORKED EXAMPLE)

		Priority x Ability	
Business Need	**Priority (1-5)**	**Choreo-graphy**	**Orches-tration**
Distribution Across Networks	5	5x5=25	5x2=10
Predictability of Response Time	2	2x3=6	2x5=10
Loose Coupling (Hot Deployments)	5	5x5=25	5x3=15
Service Reusability (Atomicity)	1	1x1=1	1x5=5
Design Complexity	1	1x2=2	1x5=5
Runtime Process Visibility	1	1x2=2	1x4=4
	TOTAL	**61**	**49**

B. LGB Bank Case

In the case of Large Global Bank (LGB Bank), based on a case from a real bank but named anonymously, their application process involved automating the reselling of third party travel insurance, initiated via a mobile device. Constraints of the application process included: a) all required microservices deployed on a single local area network, b) immediate response from the application process sent back to the mobile device (e.g. insurance policy number), c) active-active load-balanced deployment of microservices, to ensure no interruption of the process, d) all microservices designed to be reused by multiple other application processes, e) a large number of atomic microservices (nine to be exact) involved in the application process, and f) the application process must be easily documented.

A worked example of the LGB Bank assessment is shown in Table III below, which indicates that an orchestration collaboration pattern should be selected. This outcome is due to several high priority or essential LGB Bank business needs that orchestration is 'very capable' (5) for. For example, service reusability was identified as 'essential' (5) owing to constraints 'd' and 'e'. Similarly, predictability of response time was identified as 'essential' (5) owing to constraint 'b'.

TABLE III
LGB BANK ASSESSMENT (WORKED EXAMPLE)

		Priority x Ability	
Business Need	**Priority (1-5)**	**Choreo-graphy**	**Orches-tration**
Distribution Across Networks	2	2x5=10	2x2=4
Predictability of Response Time	5	5x3=15	5x5=25
Loose Coupling (Hot Deployments)	3	3x5=15	3x3=9
Service Reusability (Atomicity)	5	5x1=5	5x5=25
Design Complexity	4	4x2=8	4x5=20
Runtime Process Visibility	4	4x2=8	4x4=16
	TOTAL	**61**	**99**

C. Netflix Case

If the decision framework does not return a clear preference for choreography or orchestration, then adopting the 'hybrid' pattern – choreography with a process engine – may be the appropriate choice to make.

Netflix described the challenges of implementing their business processes in a recent article [25]. Key factors include: a) a growing number of microservices and increasing complexity of existing choreographed processes, b) the need to be able to track and visualize process flows, c) the ability to support complex workflows that run over multiple days, d) the need to scale to millions of concurrently running workflows, and e) the need to be able to force tasks to synchronously complete.

A worked example in Table IV below returns no clear preference for choreography or orchestration (e.g. scores within 10 points), and thus a hybrid approach combining the key strengths of both is appropriate. In such a case, a 'hybrid' pattern - choreography with a process engine – may be adopted. In the Netflix case, this outcome arises because it has a mix of high priority business needs that can be capably supported by choreography or orchestration. For example, choreography is 'very capable' (5) for loose coupling, which is 'essential' (5) for Netflix owing to factors 'a' and 'd'. On the other hand, orchestration is 'capable' (4) for runtime process visibility, which is 'high priority' (4) for Netflix owing to factor 'b'.

TABLE IV
NETFLIX ASSESSMENT (WORKED EXAMPLE)

		Priority x Ability	
Business Need	**Priority (1-5)**	**Choreo-graphy**	**Orches-tration**
Distribution Across Networks	5	5x5=25	5x2=10
Predictability of Response Time	4	4x3=12	4x5=20
Loose Coupling (Hot Deployments)	5	5x5=25	5x3=15
Service Reusability (Atomicity)	2	2x1=2	2x5=10
Design Complexity	2	2x2=4	2x5=10
Runtime Process Visibility	4	4x2=8	4x4=16
	TOTAL	**76**	**81**

V. MANAGEMENT IMPLICATIONS

The framework proposed in this paper is meant to guide the solution architect in deciding which microservice collaboration pattern to use for specific application processes. However, it is important to note that both collaboration patterns, as well as the 'hybrid' pattern, may be employed concurrently across a company's multiple application processes, on a case-by-case basis, depending on the design goals of each application.

Generally, the choreography collaboration pattern is preferred when there are a few microservices involved in the application process, the microservices are deployed across the wide area network, and deployment must be done without interrupting the application process. The downside of employing this pattern is that there is a cost of maintaining a message broker, and application processes are difficult to monitor at runtime due to the event-based nature of the microservices collaboration. Another downside of using this pattern is that multiple copies of the same database would need to be propagated across all of the microservices that use that data, which would have implications on database maintenance and storage costs, and would limit reusability of those microservices to be assembled into new applications.

Generally, the orchestration collaboration pattern is preferred when there are many microservices involved in the application process, the microservices are deployed on a single local area network, and an immediate response is needed from the application process. The downside of employing this pattern is that microservices are tightly coupled to the process controller through invocation-based communication, and therefore active-active load-balancing is needed in order not to interrupt the application process during deployments. One of the main benefits of this pattern, is that the process is easily visible through documentation, or simply by looking at the composite microservice (process controller) code. The other main benefit of this pattern is service reusability (atomicity), in that: a) microservices stand alone (are atomic) as they are decoupled for the processing logic, and b) microservices have clear ownership of their underlying data, with no need to propagate data redundantly across multiple other microservices (as with choreography).

The 'hybrid' collaboration pattern – choreography with a process engine – implemented in Netflix's Conductor [25], has been shown to add process visibility to the choreography pattern with a trade-off of increased chattiness [24]. All other aspects of the choreography pattern remain the same, e.g. loose coupling. In at least one study, a limitation of Netflix Conductor was revealed, as it was not able to execute workflows involving hundreds of tasks [24]. Alternatives to Netflix Conductor include: Amazon 'Step Functions', Uber 'Cadence', and Zeebe 'Zeebe'.

Both of the primary collaboration patterns can claim 'agility' as a benefit, although through different means. The choreography collaboration pattern enables agility because the reuse of existing microservices is not considered in the design, due to poor atomicity; meaning agility is not hampered from having to design for and manage service reuse. Conversely, the orchestration collaboration pattern enables agility precisely due to the reuse of existing atomic microservices. Reuse of software assets is typically an organizational challenge, rather than a technical challenge. It is our observation that organizations that are ineffective in managing software reuse, tend to decide on the choreography collaboration pattern, regardless of other factors.

VI. CONCLUSION

In this article, we have compared the two main microservice collaboration patterns – choreography and orchestration - and have proposed a decision framework to help solution architects choose appropriate patterns for their applications. The framework encourages them to identify their goals in a way that aligns to the key distinguishing properties of the patterns, then suggests an option based on a weighted scoring mechanism. Finally, we applied it in three case studies (Danske Bank, LGB Bank, Netflix), where contrasting requirements in coupling and reusability led to different collaboration patterns being suggested. We are not aware of any existing decision frameworks to guide solution architects in choosing a microservices collaboration pattern.

We extended our framework to support a 'hybrid' collaboration pattern – choreography with process engine – which combines aspects of both choreography and orchestration. For example, in Netflix's Conductor [25], microservices communicate asynchronously via queues (as in choreography) but also have a centralized process engine (as in orchestration). This flexibility would allow for collaboration patterns where a degree of loose coupling and runtime process visibility are simultaneously provided [22].

For future work, we intend to test our decision framework with industry partners who are in the process of migrating from a legacy monolithic application architecture to a microservices-based architecture.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their detailed reviews, which have helped to improve the quality of this paper.

REFERENCES

[1] O. Zimmermann, "Microservices tenets", *Computer Science-Research and Development*, *32*(3), 301-310, 2017.

[2] C. Richardson, “Microservices patterns.” Manning Publications Company, 2018.

[3] A. Megargel, V. Shankararaman, and D.K. Walker, "Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example," in *Software Engineering in the Era of Cloud Computing*: Springer, pp. 85-108, 2020.

[4] T. Cerny, M.J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: current and future directions," *ACM SIGAPP Applied Computing Review,* vol. 17, no. 4, pp. 29-45, 2018.

[5] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S.T. Larsen, and S. Dustdar, "Microservices: Migration of a Mission Critical System," *IEEE Transactions on Services Computing*, 2018.

[6] F.J. Frey, C. Hentrich, and U. Zdun, "Capability-based service identification in service-oriented legacy modernization," in *Proceedings of the 18th European Conference on Pattern Languages of Program*, p. 10: ACM, 2015.

[7] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: a systematic approach to service decomposition," in *European Conference on Service-Oriented and Cloud Computing*, pp. 185-200: Springer, 2016.

[8] E. Evans, and E.J Evans, “Domain-driven design: tackling complexity in the heart of software,” *Addison-Wesley Professional*, 2004.

[9] H. Vural and M. Koyuncu, “Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice?,” IEEE Access, 9, 32721-32733, 2021.

[10] A. Crawford, “Build a DevOps culture and squads: Building a culture is at the core of adopting the IBM Garage Methodology,” Available: https://www.ibm.com/garage/method/practices/culture/practice_building_culture/

[11] A.L. Lemos, F. Daniel, and B. Benatallah, "Web service composition: a survey of techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1-41, 2015.

[12] V. Yussupov, U. Breitenbücher, C. Krieger, F. Leymann, J. Soldani, and M Wurster. “Pattern-based Modelling, Integration, and Deployment of Microservice Architectures,” in *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)* (pp. 40-50), IEEE, 2020.

[13] M. Kleehaus and F. Matthes, “Challenges in Documenting Microservice-Based IT Landscape: A Survey from an Enterprise Architecture Management Perspective.”, in *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, (pp. 11-20), IEEE, 2019.

[14] G. Hohpe and B. Woolf, “Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.” Addison-Wesley, 2004.

[15] B. Rücker and M. Schimak, “Know the Flow! Microservices and Event Choreographies,” 2017. Available: https://www.infoq.com/articles/microservice-event-choreographies/

[16] N. Upadhyay, "SDMF: Systematic decision-making framework for evaluation of software architecture," *Procedia computer science,* vol. 91, pp. 599-608, 2016.

[17] P.R. Griffin, A. Megargel, and V. Shankararaman, "A decision framework for decentralised control of distributed processes: Is blockchain the only solution?," In *Handbook of Research on Blockchain Architecture, Strategy, and Business Value*. IGI Global, 2019.

[18] B. Goossens, "Decision-Making in a Microservice Architecture," University of Twente, 2019.

[19] M. Haj Ali, “Measuring the Modeling Complexity of Microservice Choreography and Orchestration: The Case of E-commerce Applications (Doctoral dissertation, Université d'Ottawa/University of Ottawa)”, 2021

[20] N. Singhal, U. Sakthivel, and P. Raj, "Selection mechanism of micro-services orchestration vs. choreography," *International Journal of Web & Semantic Technology*, vol. 10, no. 1, pp. 1-13, 2019.

[21] C.K. Rudrabhatla, “Comparison of event choreography and orchestration techniques in microservice architecture. *International Journal of Advanced Computer Science and Applications*, *9*(8), 18-22, 2018.

[22] L.A. Weir, “Is BPM Dead, Long Live Microservices?,” 2018. Available: http://www.soa4u.co.uk/2018/02/is-bpm-dead-long-live-microservices.html

[23] M. Camilli, C. Bellettini, L. Capra, and M. Monga, “A formal framework for specifying and verifying microservices based process flows,” in *International Conference on Software Engineering and Formal Methods*, (pp. 187-202). Springer, Cham, 2017.

[24] A. Oliveira, “Development of an Orchestration Engine for the DS4NP Platform,*”* (Doctoral dissertation, Universidade de Coimbra), 2020.

[25] V. Baraiya and V. Singh, "Netflix Conductor: A microservices orchestrator," 2016. Available: https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40