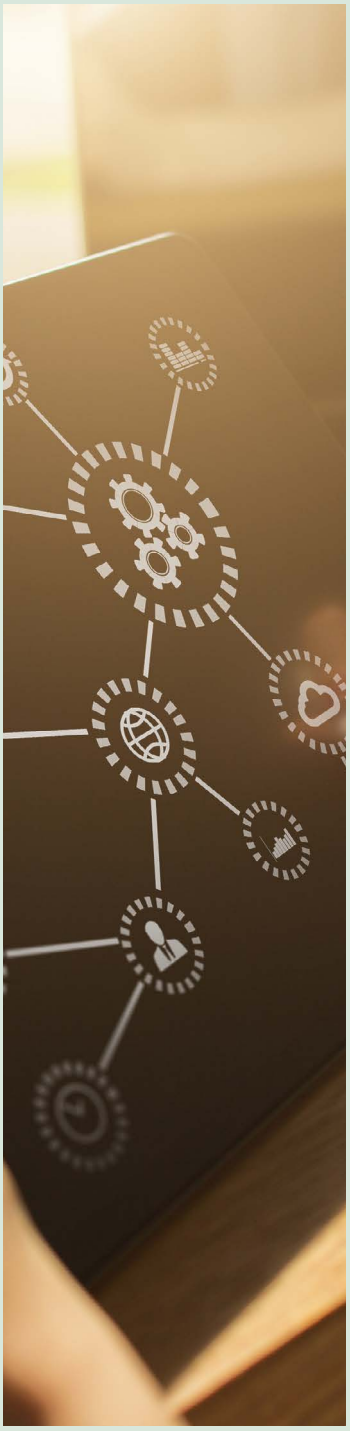
WHITE PAPER



MICROSERVICES, NOT A CAKE FOR EVERY BUSINESS

Infosys®
Navigate your next

Table of Contents

Introduction

Microservices architecture (MSA) has become one of the hottest buzzwords in the IT industry. The gravitation of microservices is so strong that it has become more of a compelling solution option for every business domain/problem for the unique advantages it offers over the traditional architectural styles. It would not be wrong to say that it has become the **de-facto architectural style** for digital transformation programs.

Big Tech companies like **Netflix, Uber, Amazon, eBay** and many others are the live examples which have been benefitted immensely by embracing this new architectural style. Today, Netflix, for example, streams around **250 million hours** of video per day to more than **139 million** subscribers across the globe, and the company continues to grow. All this was possible because they took timely measures to transform their video streaming application from the monolithic architecture to cloud-based microservices architecture.

Not denying the fact that this new architectural style offers numerous advantages, it shouldn't be considered a panacea for every business problem. The downside of it is that it **doesn't guarantee the expected benefits** for every business. Its adoption, therefore, must be **carefully planned** for, as the path to microservices is **paved with hidden problems** and **challenges**. Even for companies like Netflix, it was not a cakewalk then, though, now they are pioneers in microservices. They took nearly 2 years to break their monolith application into microservices. They started this exercise somewhere around 2009 and finally in 2011 announced end of redesigning their structure and organizing it using microservices. It therefore, becomes essential that before jumping onto the microservices bandwagon, **proper assessment** and **fitment analysis** needs to be done, otherwise it can lead to over-engineering of the applications with significant development overhead and infrastructure costs, and the resulting application with too many components/ services will become difficult to support and maintain. In some cases, it would not be wrong to say that traditional monolithic architectural style(s) will be the best fit and more efficient.

In this whitepaper, based on our experience and working knowledge, we have tried to highlight on what's going wrong with microservices adoption, key deciding factors for microservices adoption, monolith to microservices transformation roadmap and finally touch base on some of the microservices best practices. We request the readers to consider this article for reference purpose only and kindly urge them to do their due diligence when adopting MSA architecture.

What's going wrong with Microservices adoption?

Monolith applications are not only business-critical but, in general, are huge, complex, heterogeneous and composed of the varied technology stack. They interact with multiple upstream and downstream systems. To transform them into microservices, a more robust and structured approach must be defined. Below are some of the most common pitfalls which we have observed with microservices adoption, these seems to be a gap in adopting this new architectural style because of which many of such transformation projects got scrapped or are bound to fail

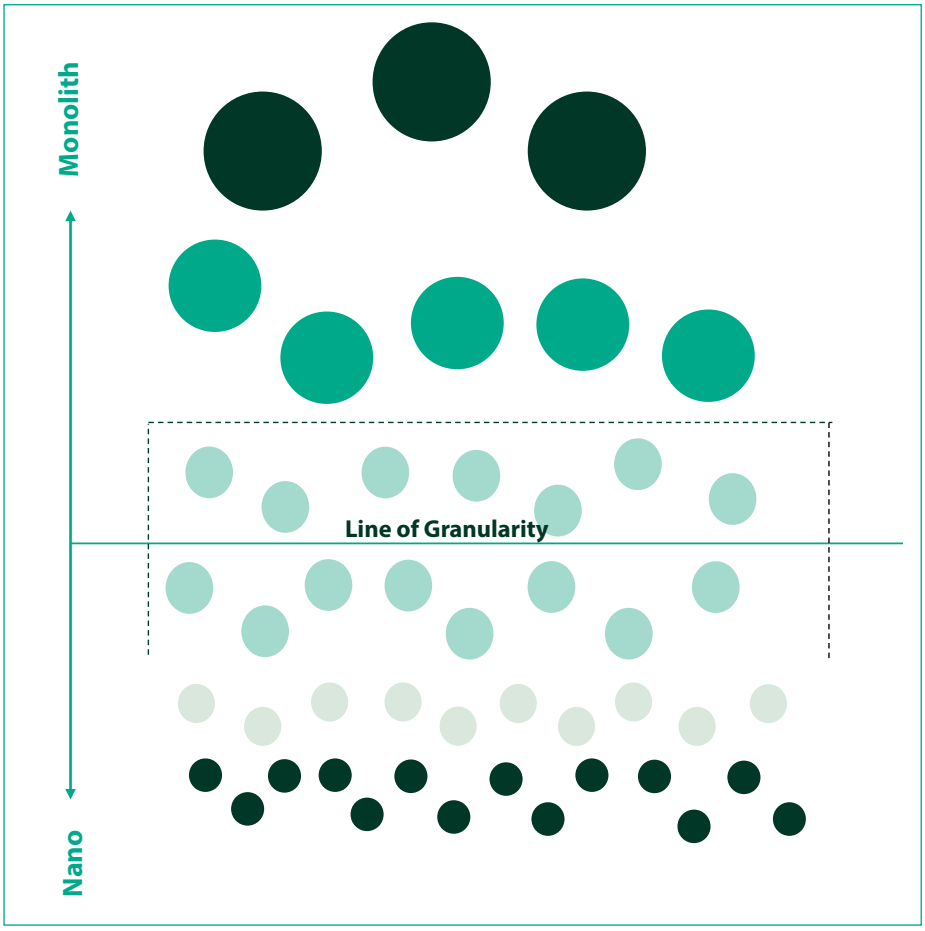
Lack of holistic 360-degree view of the application

It has been an observed trend that most project teams are directly jumping on to the microservices bandwagon without analyzing their needs. In such a hurried state of adoption, they are missing the holistic 360-degree view of the application and concentrating only on the server-side decomposition or refactoring of the existing server-side code into a set of microservices by following the big-bang approach without any view of providing new functionality or business value add. Another most common misconception is to deploy each existing API as a separate service and calling it as a microservice. These are the two most detrimental approaches towards microservices.

Apart from the above two, the other important aspects which are generally missed out are: -

- Applications dependency matrix and the order of their decomposition
- Microservices identification approaches like DDD and bounded context, etc.
- Data considerations
- Impact on UI
- Third party interfaces / software dependencies
- Defining services of proper granularity

This has not only resulted in more rework with increased effort and missed timelines, but at times led to project failures also. So it becomes of utmost importance that project teams must thoroughly perform the E2E application assessment with the required stakeholders and come up with a proper transformation roadmap.



(Source: https://www.opengroup.org/soa/source-book/msawp/p6.htm)

Services granularities not defined properly

Services granularities must be properly defined to prevent the two extreme scenarios where services are defined either too coarse grained (monolith) or fine grained (nano). We have seen the scenarios where the teams were not able to draw this line of service granularity (as shown in the below diagram) and ended up with too fine grained services which resulted in expensive remote calls, chatty communications and un-manageable services. Monolith services comes with their own dark side nature which we are well versed with. So defining the right granularity of the services is more of an art than a science.

Good microservices should be very close to the line of granularity as shown below:

One important point to consider is that the **number of lines of code must not define the granularity of the service.** The service should be granular to such an extent that it is does one thing and does it well. Below are some examples of services with proper granularity

- **CreateOrder** will do only the job of creating the order
- **RegisterCustomer** will do only the job of registering the customer

Let's assume that RegisterCustomer internally talks to another **ValidationService** which validates all the customer related information. ValidationService is a valid microservice which provides validation functionality and encapsulates all the validation logic. This ValidationService internally talks to other small services like **addressValidationService, EmailValidationService, telephonevalidationservice** to validate the customer information. This is where the things start going wrong as each of these services are now nanoservices. However, there's an argument that says **nanoservices have a place in Serverless architecture,** so the debate continues. But in general, **nanoservices are considered as anti-pattern in microservices realm.**

To overcome this challenge, **Domain-Driven Design (DDD)** modelling can be adopted to come up with business specific domains, sub-domains and the corresponding bounded context. Identifying the right set of bounded contexts will help to define services with proper granularity.

Non-involvement of required stake holders while defining microservices

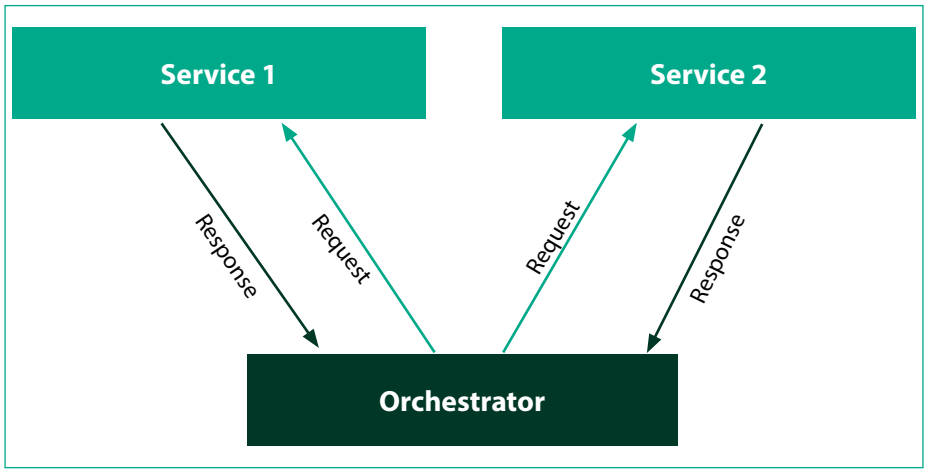
One of the major gaps which we did observe was that **only the development and none other stakeholders (business, QA, support, etc.) were involved** in microservices identification exercise. Practically, it is difficult to find too many independent components /services within an application because of cross-functional processes and data requirements. It can be addressed to some extent with the involvement of required stake holders to identify these independent components. Such un-healthy practices have not only led to the wrong identification of services but dragged the development teams to design and develop inappropriate microservices and keep iterating over solutions.

Tight coupling amongst microservices

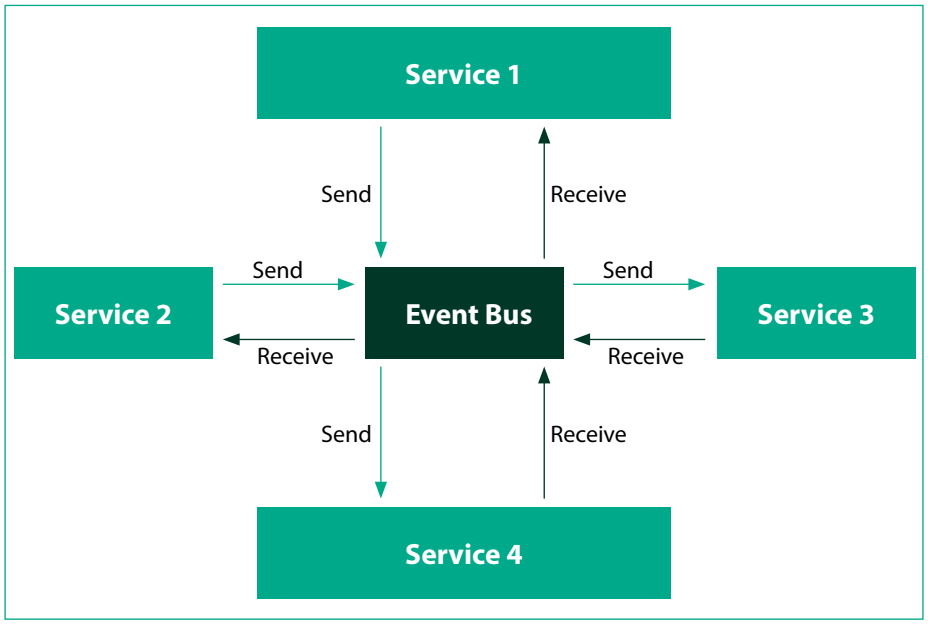
In MSA, each service is an independent entity and performs a specific task or function. Sometimes these services need to interact and share data. From interaction per se, we can consider the following two patterns

1. Service Orchestration
2. Service Choreography

Service Orchestration approach consist of a **central controller** usually called the orchestrator which handles all microservices interactions. It calls one service and wait for the response before invoking / calling the next service. It follows the **request/response** type paradigm.



Service Choreography doesn't follow the centralized collector approach but rather the set of independent services that interact with each other in an **asynchronous fashion using an event bus.** Each service broadcast data as events and the interesting services subscribe to those events, use the data and perform actions.



Depending on the problem statement and the use case in hand, the most appropriate integration pattern must be adopted. For some use cases which involves both synchronous and asynchronous blocks of activity, the **hybrid** or the combination of both these approaches is most appropriate.

Our observation was that; the organizations were using **the traditional request/reply way** of interaction amongst the services. This resulted into very **tight coupling amongst the services as** change in one service request had cascading effect on the dependent services and this defeated the very purpose of developing applications using MSA architecture.

Also, if too many service interactions are needed to accomplish the task, then, it raises the question on services granularity and might be an indication to merge services responsible for all operations related to a particular entity.

"The road to microservices is paved with good intentions. But more than a few teams are jumping on the bandwagon without analyzing their needs first"

—Nathaniel T. Schutta

Key decision factors for Microservices adoption

In this section, we will discuss on some of the key deciding factors or principles which will guide in the microservices adoption and help us avoid accidental complexity into the system.

When should we not go for Microservices?

The famous adage "one size doesn't fit all" holds good for microservices as well, as they are not the right fit for every business/ project. Just because everyone is jumping on the bandwagon cannot be the sole reason for its adoption. **We must spend substantial amount of time to analyze if the given application is fit for MSA or not.** Business value must be the key driving force. The investments made must make sense for the rewards. The rewards can be in terms of increased agility, faster releases, improved performance, and increased customer satisfaction which finally leads to business growth.

It is therefore of utmost importance that proper due diligence has to be done taking into consideration different aspects like business value add, cost-to-benefit analysis, technology complexity, architecture fitment, skill set availability, monitoring and infrastructure overhead and then decide on whether to go with microservice architecture or not.

Based on our experience, we have mentioned couple of below scenarios where MSA is not the right fit. These are for reference purpose only and we request the readers to do proper diligence from their end.

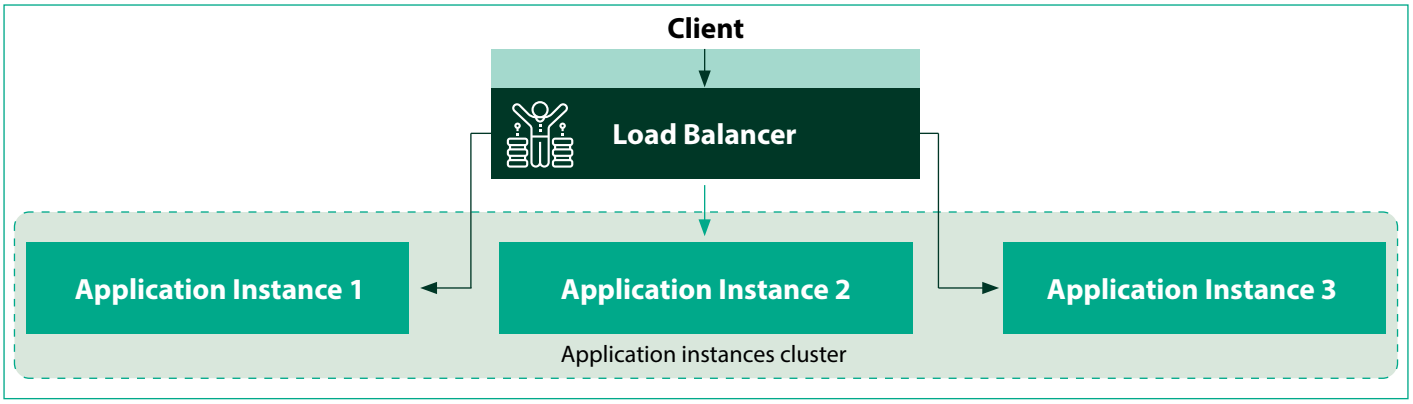
- **Small scale legacy applications** which doesn't have future roadmap
- **Non Complex** applications
- **Intranet based admin applications** which are not business critical
- **No business value add** for the time and effort put in
- Organization **cannot support multiple development teams** working independently and simultaneously.
- **Stable applications** with **low change management**
- **Small scale projects** which **cannot withstand integration** and **infrastructure overhead**
- **Monolith UI** with **significant cross references** of data
- **Tightly coupled application domains** which impose a greater challenge to identify the independent bounded contexts
- **One-time point solutions** with low volumes of change
- Several **dependencies** and external **integration** points
- **Workload is low** and applications NFRs are within the defined SLAs

When should we go for Microservices?

Following are some of the decision factors favoring MSA adoption.

To achieve higher scalability

Till now monolith applications have **scaled horizontally** by running multiple copies of application instances **load balanced across servers** and sharing a common database and cache. It works well when the database read to write ratio is very high and when the transaction growth exceeds the data growth. This approach corresponds to **X-axis scaling** (shown below) in the **"scale cube"** model (for more details please refer Art of scalability[1]).



Normally, over time, such deployments lead to increased transaction volumes with substantial growth in data which increases the write to read ratio. When it happens, monolith applications become very resource-intensive which reduces the applications performance and scalability and other aspects of the application take a hit and on top of that continuous development, constant release and upgrade activities adds to increased complexity and any further X-axis scaling will only engender the situation. In such situations, it's better to consider other dimensions of scaling (Y-axis or Z-axis).

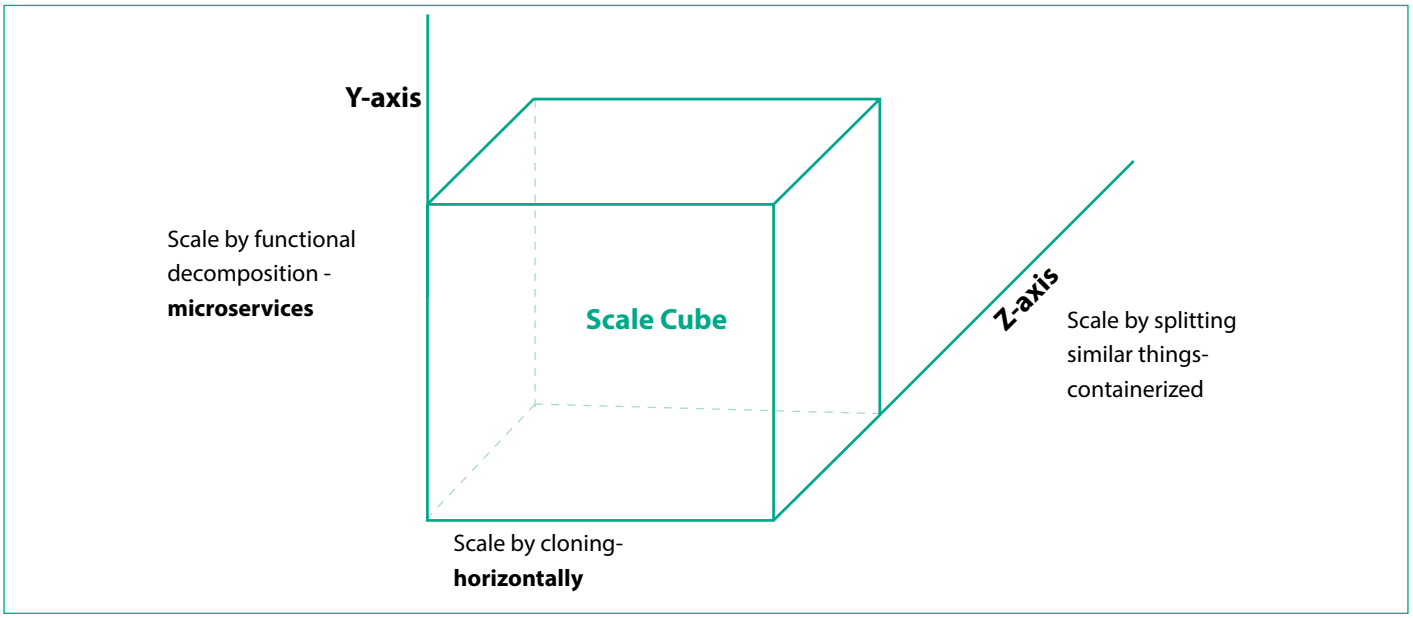


Fig. Scalability Cube

Y-axis scaling refers to **split the application into different services,** each of which is an **independent deployable unit.** Each service is normally composed of one or more related functions.

Microservices support this scaling model, as they are smaller, independent services with their own database. There can be multiple ways of decomposing the application into services but this model mainly focuses on the following two ways of decomposition.

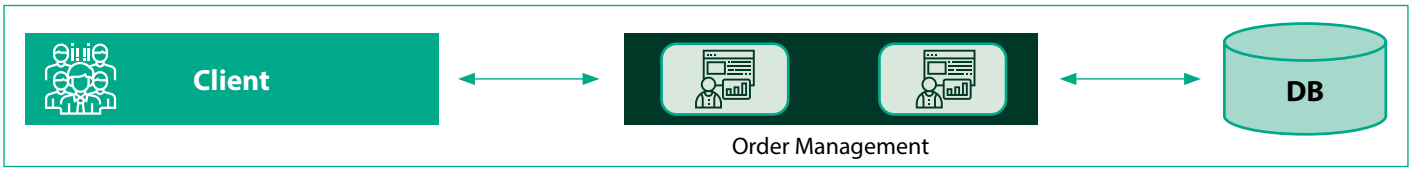
1. Verb based decomposition
2. Noun based decomposition

Verb based decomposition mainly concentrates of defining services which can handle an **atomic operation** or a **single use-case** like **search, payment, add to cart,** etc.

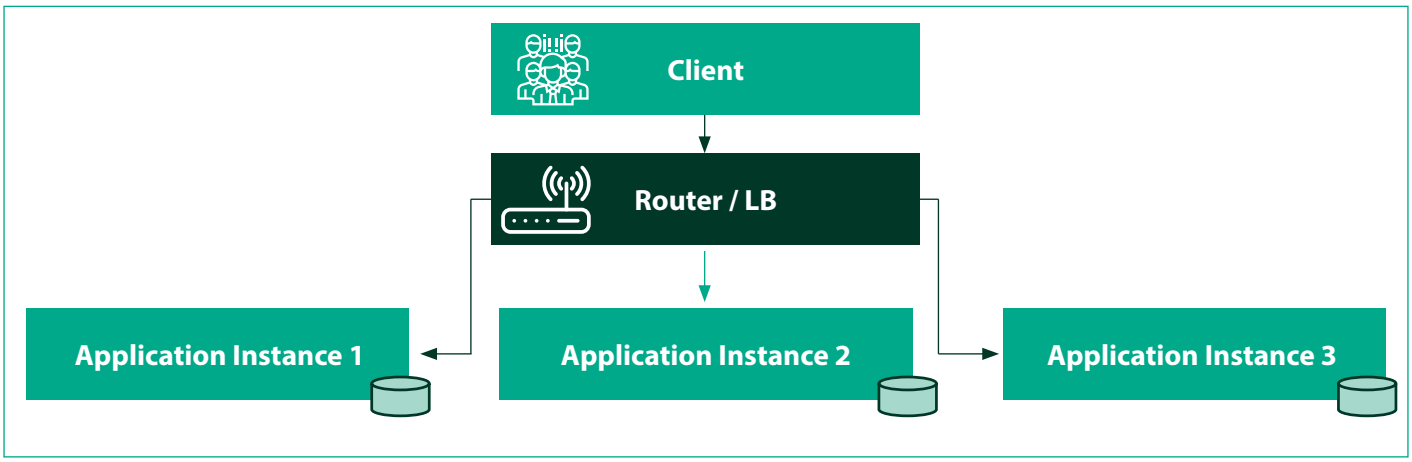
Noun based decomposition normally split application into **set of services which are responsible for all the operations within a particular entity** like **Order management, User management** etc.

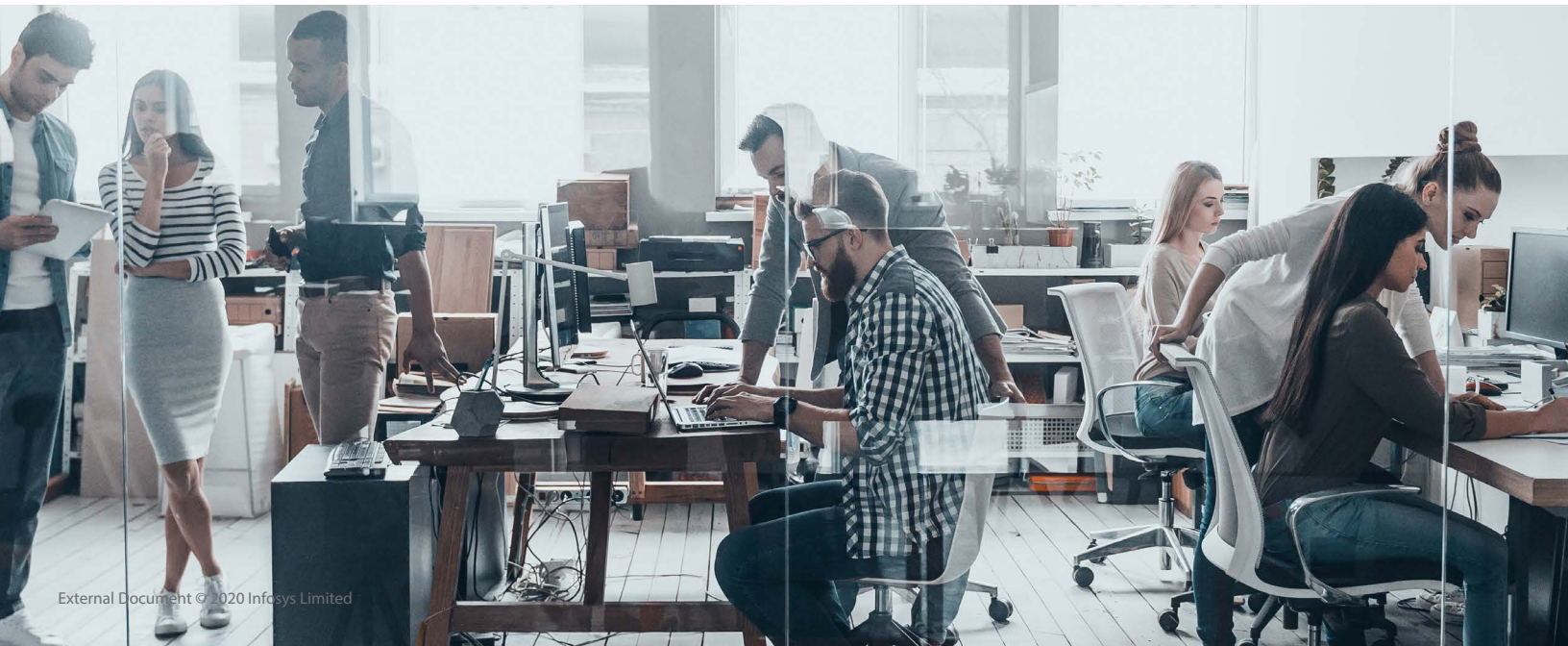
An application can use either of these or a combination of both these decomposition techniques to achieve higher scalability.

- If there are parts of the system which **need to scale independently** from the rest of the system due to their **independent nature** or the **load** or **throughput** characteristics of these components are quite different, then we can go for Y-axis scaling for them.



Z-axis scaling is quite similar to X-axis scaling in the sense that it also runs multiple instances of the application but **the data is partitioned/shared amongst each of these instances** i.e. each instance works only a subset of data and a load-balancer or a router is responsible to route the request to appropriate application instances. For multi-tenant applications this scaling model is the right fit.





To support Polyglot development

If there are parts of the system or business use cases which suits a particular technology stack or which can benefit from a particular technology stack, then it's better to pull them apart as a set of independent services (microservices) from the rest of the system which can be on a different technology stack. For example, **Search functionality** can be developed as an independent entity using **Elastic Search** engine to achieve higher performance and increased scalability.

Following are some of the guiding principles which we should be adhered to for polyglot development

1. Technology stack standardization must be done so that team doesn't start using every technology under the sun! Every technology stack must be evaluated before adopting them.
2. Proven technology stacks for a particular use case must be adopted
3. Proper technical guidance must be provided to the team to make the best use of the chosen technology stack.
4. Teams must be trained on the chosen technology stacks.

To support independent and frequently changing parts

If there are **independent components** within a system **whose lifecycle can be managed separately** from the rest of the application, then, such components are the right candidates for microservices. They can be developed fast, tested quickly and released to the market in no time. This promotes agility and enable to take up new business opportunities/use-cases faster.

Also if there are **frequently changing parts** within an application which need to evolve at a different speed or in different directions then those parts are the good candidates for microservices.

To isolate failures and external dependencies

If an application is **dependent on some unreliable external systems** which are not meeting our availability requirements and can result into failures, then, it is better to bundle such external system calls within a microservice and handle the failure over scenarios. This will prevent from entire system going down in case of any failure on the external system.

Even if the external systems are highly reliable and designed for failure but are the candidates for change in future, then, such system calls must also be bundled within a microservice. One such example can be a payment gateway

Guidelines or Key Recommendations

Following are some of the key guidelines or best practices being followed within the industry for MSA adoption.

Monolith-first mantra

From the domain modeling perspective, any new application development can be classified into following 2 broad categories:

a. Part of **existing business domain**

b. Part of completely **new business domain**

- If we have teams with **reasonable experience of building microservices systems** and SMEs with **extensive knowledge of the existing business domain** and combinedly they can help identifying the right set of bounded contexts or independent entities, then we can start with MSA approach for greenfield applications.
- In case the **team is new to MS**A or we **don't have SMEs with required domain knowledge** or the domain is **too complex** or it is **completely a new business domain,** in either of these cases going directly with MSA architecture will be risky. In such scenarios, the mantra to follow is to follow **Monolith-first approach** and build an iterative model of the system and as we mature our knowledge around the business domain and understand the system complexity and its component boundaries, it then becomes easy to identify the parts which can be taken out into independent existence.

Follow 12 factor App methodology

It is industry proven methodology for building modern web applications. These best practices enable applications to be highly portable and resilient in nature when deployed to the web. They provide the required governance structure for building microservices. Please refer to this wiki[2] to get more details about the same.

Two Pizza team approach

Another key tenant for successful implementation of microservices is to have team size small enough to be fed on two pizzas **("two pizza rule"** coined by Amazon CEO Jeff Bezos). The rationale behind this rule is to form **autonomous, creative** and **innovative** teams which have effective communication amongst the team members and is only possible when the teams are smaller in size and work together. The team members know each other better, form relationships, trust and motivation which ultimately leads to improved productivity.

Form cross functional teams

The rationale behind it is to form teams which can handle the E2E lifecycle of microservices (development, testing, build and deployment) and thereby avoid the dependency on other teams which can slow down the entire DEVOPS chain. The team can be composed of a developer, a QA, a database person and operations guy or the developer must to be able to take on different roles. This practice is very common in big product companies like Google, Netflix and Amazon etc. The mantra is **whoever builds should be responsible for it.**

Define a maturity model

It's better to define the microservices maturity model. This will help to gauge applications current maturity level and define the transformation roadmap to achieve the next maturity level.

Key Challenges in implementing Microservices

Following are some of the key challenges in implementing Microservices: -

Setup proper DevOps Culture

The goal of microservices is to accelerate software delivery through **continuous delivery** and **deployment** and is only possible with cross-functional teams **(DevOps)** who can cooperatively build, test, release, monitor and maintain the applications. Lack of DevOps culture increases dependency on other teams, which will bottle up the releases and impact the overall time to market. So even before we embark on the microservices journey, we need to ensure that DevOps culture is adopted at the organization level.

Querying data across microservices

As per the norms, every microservice must have its data source, which keeps its persistent data private and make accessible only via its API. Some use cases will demand fetching data from multiple data sources. In such scenarios, all the required services need to be invoked to fetch the data, which, not only increases the complexity by creating hard dependency amongst the services but hits the performance as well. Instead, we can follow one of the below approaches which provide a more robust and loosely coupled approach

- API Composition
- Command Query Responsibility Segregation(CQRS)
- Event Sourcing

Distributed Transactions

Since microservices are distributed in nature, it's possible to have transactions that span multiple services and therefore databases. The major challenge with such distributed transactions is to ensure their atomicity as they don't have a global transaction coordinator. However, there are a couple of approaches that can be used to handle distributed transactions.

- **2 phase commit**

Though this approach guarantees transaction atomicity but is not a recommended approach for microservices as it is inherently slow and can have adverse effect on the system throughput during high load. Also it requires significant development effort in every service that can participate in a transaction

- **Eventual Consistency and Compensation/SAGA**

In this, distributed transactions are handled through asynchronous local transactions on related microservices which communicate through an event bus. It is the most recommended approach to handle distributed transactions.

Distributed tracing

In MSA, the request can flow through multiple layers of services which are spread across the network, so it becomes very difficult to trace a particular request to debug a reported issue. Below are some of the solutions to handle distributed tracing.

- Correlate requests with a unique ID (Request ID or Correlation ID). This will be added to all the logs and sent to all downstream service calls
- Spring Cloud Sleuth tracing library
- Using distributed tracking system like Zipkin

Monolith to Microservices transformation roadmap

Once the organizations have decided to transform their existing monolith applications, the major challenge before them would be to define the MSA transformation roadmap. The below section highlights the transformation strategies which we have adopted successfully in a few transformation programs. They can be used for referential purpose and project teams must do proper due diligence in coming up with the contextualized solution approaches.

Transformation strategy

Once the set of applications have been identified which need to be transformed to MSA, we need to define the transformation strategy for each of them. One of the most popular industry strategy is to adopt the principles of **"Domain Driven Design"** to guide us along the way. The domain and technical knowledge of the existing brown field applications will enable to perform domain modelling and come up with the set of bounded contexts which can then be converted into Microservices.

Once bounded contexts are identified and defined, we can follow the below strategies

Apply Strangulation or Incremental Migration Strategy

Instead of going big bang on monolith decomposition, the best strategy would be to follow incremental approach and slowly strangulate the application with 1 or 2 micro services and run them simultaneously alongside monolithic application. Over time, keep on adding new set of micro services which will finally result into lean / no more monolith.

Plan for Canary release / Phased roll out or Incremental rollouts

For microservices deployment the best approach would be to plan for Canary releases or incremental rollouts to reduce the risk of production failure and be sure that services are working as expected and entire business functionality is intact with this new architectural change. With this approach the main advantage gained is that we can do phased rollout of services to a small subset of users and test them thoroughly and once fully satisfied roll out to wider audience.

Segregate frontend and backend tiers

Split out the presentation and business tiers. They must communicate using light weight messaging protocols like REST/ HTTP(s) with proper message format(s) like JSON/XML over HTTP(s). This will promote loose coupling and increase application scalability as each tier can be scaled independently of the other.

Data considerations

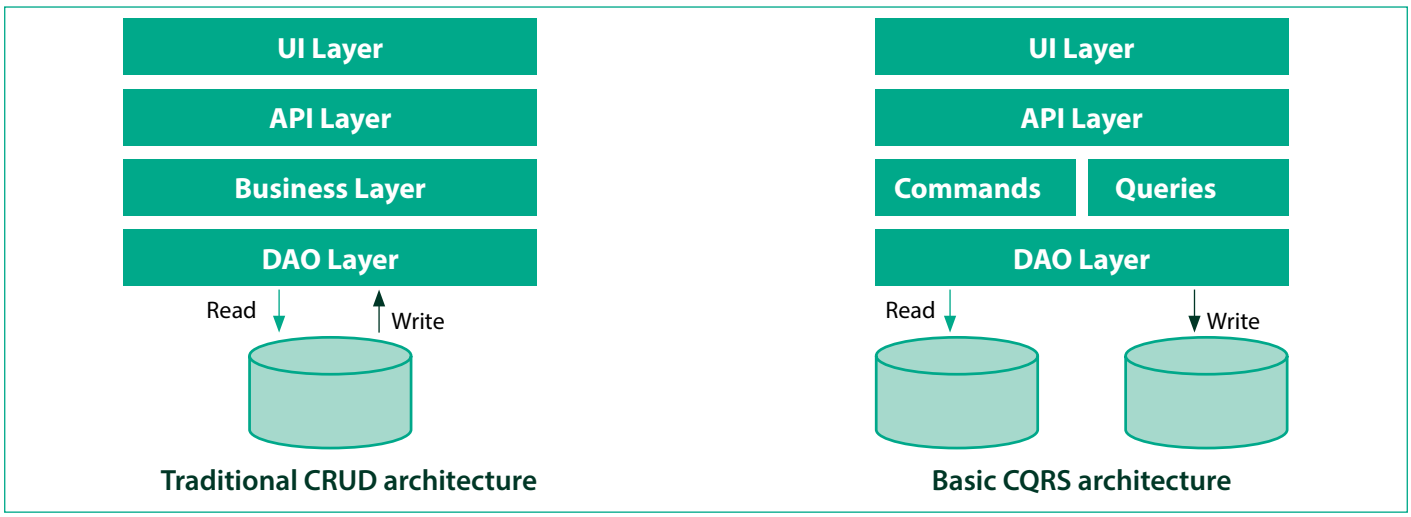
When it comes to data, maintaining data integrity and consistency are the major challenges posed by MSA architecture. Data privacy need to be dealt with and the data owned by a service is accessible via its API only. Apart from this, reporting and querying functionalities poses their own respective challenges. So each aspect of the data must be properly dealt with. Following can be some points for considerations

Choose the appropriate data store

- Based on the application complexity and the defined SLAs, decide on the most appropriate data store. To play safe and ensure least risk, continue with existing data store and perform data decoupling (in terms of schemas, tables, views etc...) amongst services as the first solution. Once data is decoupled successfully, later planning for or exploring another technology becomes easy.
- For services demanding high scalability and throughput validate appropriate data store.

Choose the correct data manipulation strategy

- Considering each service requirements and the involved complexity, decide on the most appropriate data manipulation strategy. General rule of thumb is to go with CRUD (Create, Read, Update and Delete) strategy for services which need **same data model** for both **read** and **write** operations and adopt CQRS (Command and Query responsibility pattern) for services which **need separate data models for both read and write operations.**



Choose the correct data consistency mode

- Decide on the consistency model to go with [Eventual Consistency Vs Strong Consistency]. If immediate consistency is the requirement i.e. any update in any node requires all nodes to agree on the new value before making it available for client reads, then go with Strong consistency and if high throughput and availability takes precedence over immediate consistency then go for eventual consistency. Point to note here is that most of the real world business use cases are already eventual consistent.

Handling the transactions across microservices

- One of the major challenges with microservices architecture is to develop transactional business applications. One of the proven ways to overcome this challenge is to design microservices using DDD (Domain Driven Design), Event sourcing and CQRS (Command Query responsibility segregation) approach. For details you can refer this link[3]

Avoid rewrite from scratch until absolutely necessary

Try to reuse the existing business code/ logic by retrofitting it to the micro services need. Avoid rewriting the entire application from scratch. It should be done in the extreme cases where the existing business functionality need to be revamped or we are going for polyglot development.

Keep new functionalities in standalone Microservice

If the new functionality can have independent existence better to develop it as a standalone micro services rather than making it the part of existing monolithic application. Initially, it will take time and effort both to manage monolith and microservices, but gradually it will prove beneficial.

There can and will be different factors which need to be considered depending on the application complexity and the underlying business domain. So having said that, we have come to the end of this article and hope you might have enjoyed it and got some insights on how we should plan for MSA adoption.

Conclusion

In this article we talked about what's going wrong with microservices adoption and when we should and shouldn't use this new architectural style. While MSA may not be suitable for every problem statement, it is a compelling choice for problems that benefit from the independence constraint. Having said that, we are **not trying to advocate any negative mindset towards microservices** but rather trying to guide the teams to be more cautious and follow a holistic approach towards this new architectural style.

Moreover, Organizations must have a more focused approach towards **"Why Microservices"** rather than **"Why not Microservices"**. This article will help them with some of the necessary concepts / to come up with to do list before they start on the MSA journey. They shouldn't be **attracted to the hype as the cost and challenges are as real as the benefits.** With that said, **Microservices are not a free lunch!**

About the Authors

Madhavi Shailaja Katakam, *Technology Architect*

Madhavi is a Technology Architect in Infosys. She has strong expertise on Java technologies. She also has very good expertise related to various CI and CD aspects-build management using Maven and auto deployments.

Ravi Shankar Anupindi, *Senior Technology Architect*

Ravi is a Senior Technology Architect in Infosys. He has strong expertise on Java technologies and has been working on the cloud native technologies.

His area of interest includes exploring latest technologies and looking at ways to adopt them to derive significant business benefits. He has been involved in many DevOps initiatives for clients.

This paper is the personal point of view of the authors

References

https://blog.christianposta.com/microservices/the-real-success-story-of-microservices-architectures/

https://microservices.io/articles/scalecube.html

https://akfpartners.com/growth-blog/scaling-your-systems-in-the-cloud-akf-scale-cube-explained

https://content.pivotal.io/blog/should-that-be-a-microservice-keep-these-six-factors-in-mind

https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/

https://medium.com/@abinoda/how-teams-get-microservices-wrong-from-the-start-51777c99c059

https://www.opengroup.org/soa/source-book/msawp/p4.htm

https://techbeacon.com/app-dev-testing/challenges-scaling-microservices

https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/

For more information, contact askus@infosys.com

Infosys®
Navigate your next