mitrais | MEMBER OF CAC HOLDINGS GROUP

Adopting Microservices Architecture by Implementing Microservices Patterns

Mitrais is a world-class technology company based in Indonesia and part of the global CAC Holdings Group. Founded in 1991, we have developed and implemented software for over 700 clients, and we are committed to building long-term and high-trust relationships.

www.mitrais.com

Table of Contents

Abstract

Microservices architecture has become a popular application architecture to build new or modernize existing legacy applications. Microservices architecture offers independent deployment and scalability of application modules, which can reduce downtime and allow teams to maintain a smaller code base. However, not all companies that adopt microservices architecture achieve outcome they want. This paper will outline the definition and benefits of a microservices architecture,and expand on the microservices patterns which will reduce the risk of failure when implementing a microservices architecture.

1. Introduction

There is no clear and consistent definition of microservices architecture. Some have tried to define microservices architecture; for example, Chris Richardson stated, “Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are highly maintainable and testable; loosely coupled; independently deployable; organized around business capabilities; owned by a small team”.

Martin Fowler wrote, “Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery”.

From these two definitions, we can see that there is no general consensus as to what microservices architecture encompasses.

What we can observe from industry definitions is that microservices have certain consistent characteristics:

- Small in size (more on this later)
- Focussed on one task
- Aligned with a bounded context; and
- Autonomous

1.1 Microservices Characteristics

1.1.1 Small

The definition of small is relative. This could be the number of lines of code in the application, the number of team members working on the application, size and weight of the application or a range of other measures. James Lewis, Technical Director at Thoughtworks, stated that “a microservice should be as big as my head.” This definition is ambiguous at first, however, the idea is that a microservice should be small enough to be understandable by the team assigned to it. Dave Farley, the author of “Continuous Delivery” stated that “If you could rewrite it in a week or two, then that is how should small a microservice is”.

1.1.2 Focus on One Task

This is related to the ‘Separation of Concern’ principle to organize applications into defined sections, so that each section addresses a specific concern. In microservices, the sections are directly related to problem domains. Therefore, each microservice should focus only on tasks in one single problem domain.

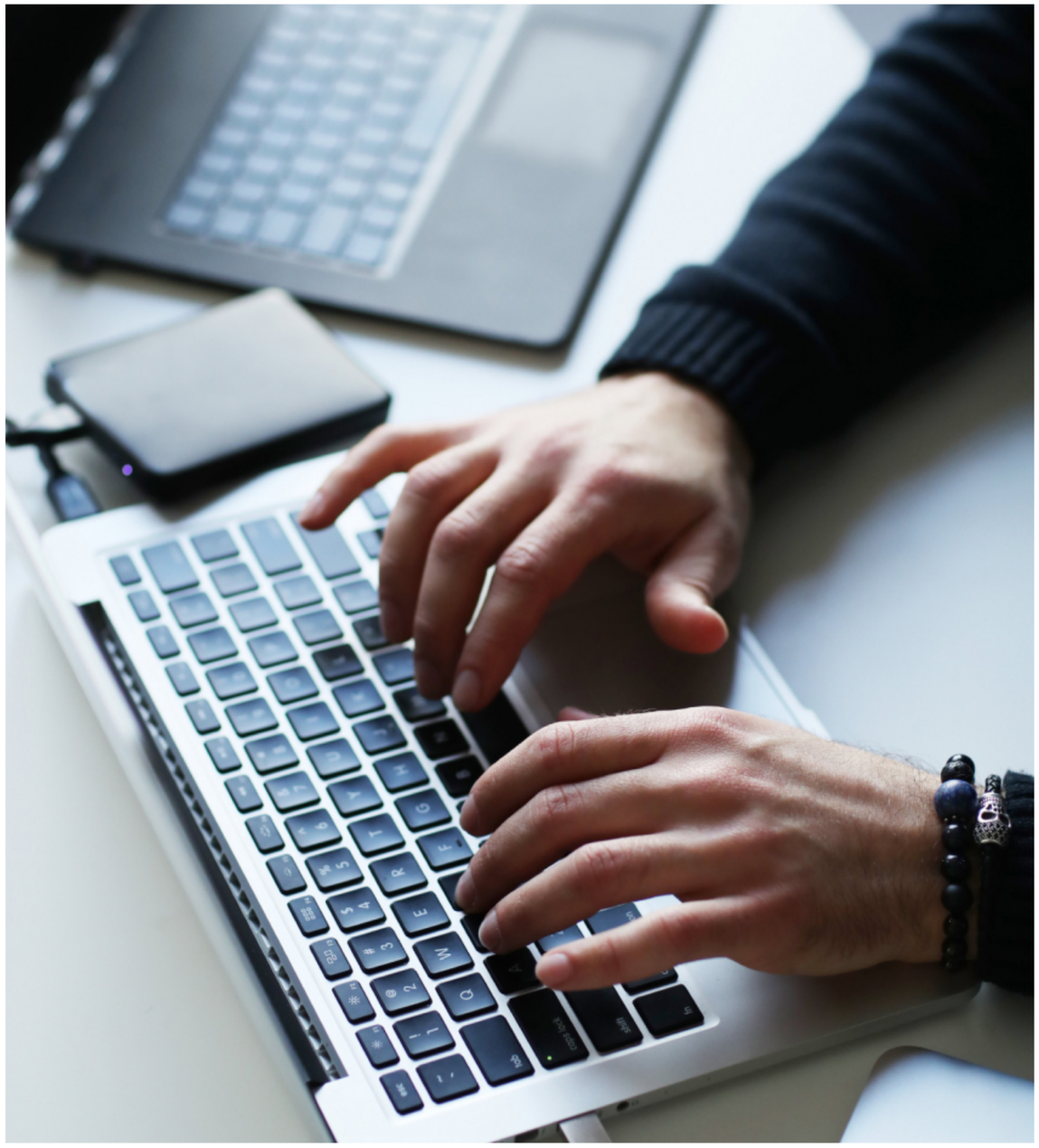
1.1.3 Aligned with a bounded context

‘Bounded Context’ is an idea that comes from “Domain Driven Design” as presented by Eric Evan in his 2003 book. He explained that a ‘Bounded Context’ is a defined part of the software where particular terms, definitions, and rules apply in a consistent way so that it makes a cohesive unit. Although this concept can be applied to any context, in microservices this is key because it provides a clear boundary with which to align the elements that make up the microservices.

In traditional development, applications are broken down into smaller technical services - however, this does not qualify as microservices because it is not aligned with the ‘Bounded Context’ principle, and creates coupling between the services themselves.

1.1.4 Autonomous

What makes microservices autonomous is that any implementation changes can be applied without coordinating with other microservices across the application. This allows microservices to scale enormously in an organization and this provides the greatest opportunity for delivering value in implementing microservices. However, this is also the part that is often misunderstood when implementing microservices. Developers frequently claim that they are implementing this architecture, but they must also build, test, and deploy the microservices together, which runs against the foundational principle of the architecture.



1.2 Microservices Adoption

More and more companies are adopting microservices. Based on the “Cloud Microservices - Global Market Trajectory & Analytics” report from Research and Market, the microservices market will grow 19.2 percent over the period 2020-2027 which is equates to about US$2.8 Billion. This is because companies want to take advantage of the benefits of microservices, such as allowing teams to implement or change features without impacting other services and getting to market faster because teams only need to change a small portion of the applications. However, adopting microservices is quite challenging. O’Reilly in their “Microservices Adoption in 2020” survey found that 10% of the respondents reported “complete success”, 54% reported being “mostly successful”, and 92% reported “at least some success”. While adoption is growing, it would appear that not all companies have achieved a successful microservices implementation.

The biggest challenges that the respondents to the “Microservices Adoption in 2020” survey mentioned are decomposition and complexity. At first, decomposing a large application sounds easy - however doing it the wrong way will only create other problems, like creating a coupled distributed application, creating increased complexity. As a result, rather than utilising microservices, the app ends up as a distributed monolith, which does not offer the benefits of the microservices architecture. To minimize the risk of failure, there are several microservices patterns that can be used as an approach to adopting a microservices architecture. These patterns outline the strategy that is used as common architecture by companies that have already adopted microservices, such as application decomposition strategy, data handling, and deployment strategy.

2. Microservices Pattern

2.1 Decomposition Pattern

One of the microservices characteristics is that they should be small. This is achieved by decomposing large applications into smaller, specific microservices. The challenge then lies in identifying how small each microservice should be. If it is too large, then it will create monolith application challenges, like:

- Longer build time
- Longer deployment time
- Challenging change implementation
- Increased development team capacity required

But if each microservice is too small, you are faced with:

- Complex integration between services
- Coupling of services

Microservices that are too small are a signal that during the decomposition process, the organisation was focused only on the representation of data, not the behaviour of the services. There are several patterns that can be used to help decomposing services that will help mitigate the risks outlined here.

2.1.1 Decompose by Business Capability

This pattern decomposes services corresponding to broad business capabilities. James Denman stated that “A business capability is an expression or the articulation of the capacity, materials, and expertise an organization needs in order to perform core functions”. In other words, a business capability is something that the business does to generate value for themselves, or their clients.

The above picture is an example of the business capabilities of a bank. Each of the business capabilities represents the problem domain of the application. It becomes clearer as to how to identify how many services the bank will have to build into their applications as microservices, and how big or small the services will become.

2.1.2 Decompose by Subdomain

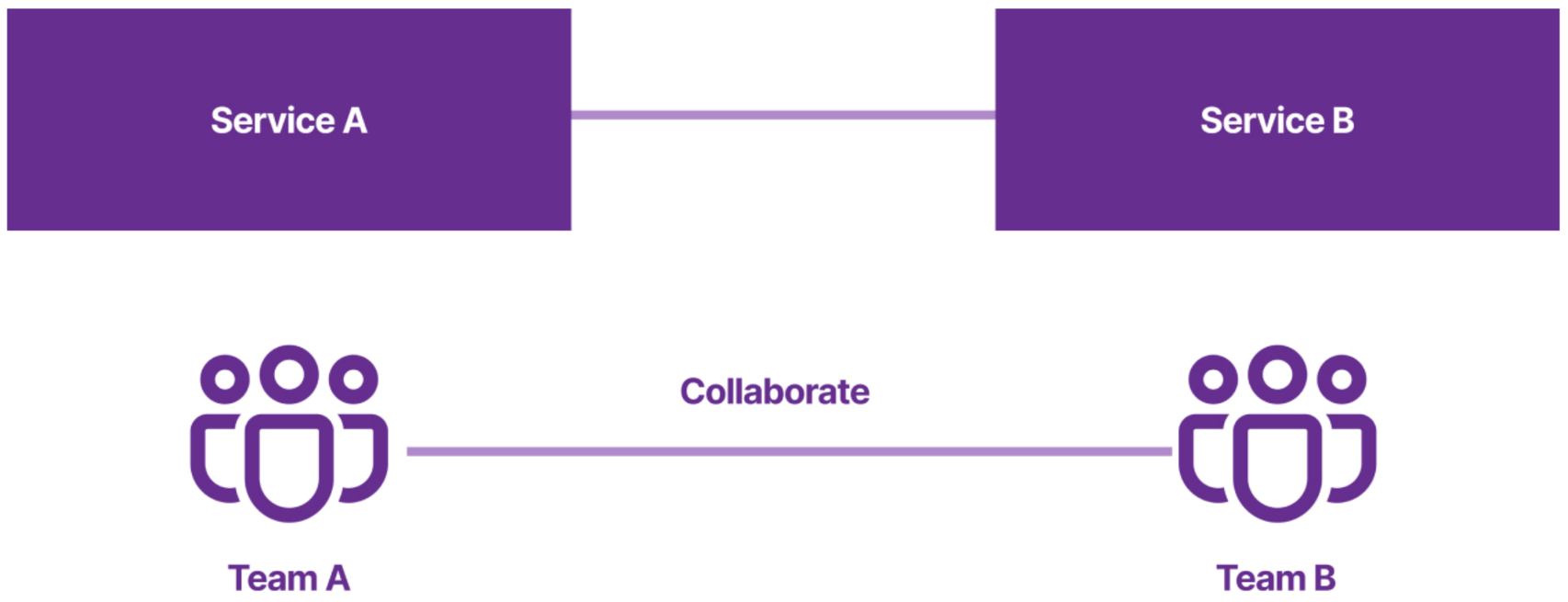
In more complex organizations, decomposing microservices by their business capabilities might not enough. This is because one business capability could contain a large amount of functionality, and so decomposing by business capability and then again into its subdomain is necessary.

Commercial Banking	Customer Relationships	Finance & Controlling
Cash Management	Complaints Management	Accounting
Corporate Financing	Contract Management	Asset Management
Credit Management	Customer Engagement	Controlling
Deposit Management	Customer Identity Management	Payroll
Pensions Management	Customer Management	Settlements & Payments
Syndicated Loan Management	Customer Scoring	Tax Management
Wholesale Trading	Customer Support & Education	Treasury
	Order Management	

Here we see an example of functionalities inside the business capability that represent the subdomain of the problem domain. Think about it as subsystems inside the system, creating smaller and more specific microservices that can run and be maintained independently.

2.1.3 Service per Team

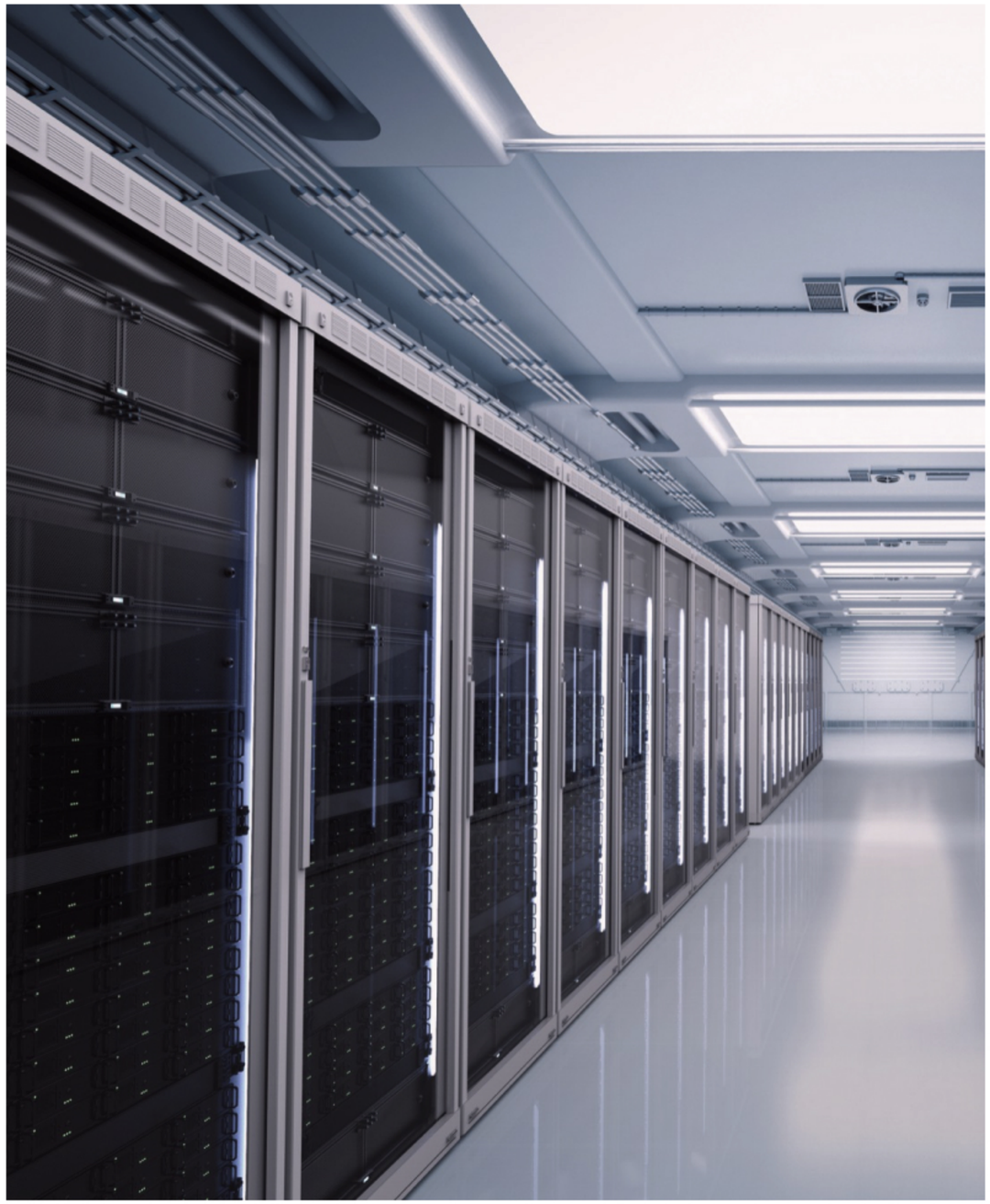
In a Service per team pattern, each of the business capabilities is owned by a single team, and ideally, each team only maintains, builds and operates one service. The size and complexity of the service is scoped to the team's capacity.



The drawback of this pattern is that implementing features that span services is more complicated and requires teams to collaborate, which can slow build time and add to the volume of work. This pattern is more suitable for more digitally mature organizations, and it is recommended to start with Decompose by Business Structure and Decompose by Sub Domain first, and then assign the build of any microservice to each team by problem domain.

2.2 Data Management Pattern

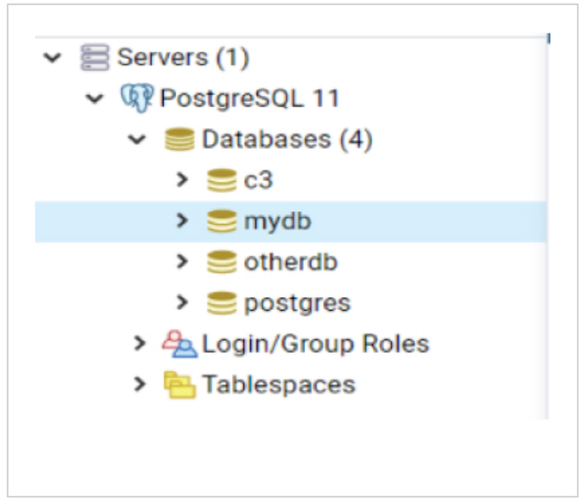
Most applications require data as an input, and produce data as an output, while the data processing itself might involve persisting the data or retrieving data from another source. In a monolithic architecture, this can be achieved by providing data storage that can be accessed by any components inside the application. However, in microservices this can be challenging because each service no longer resides in the same application. There are several patterns that can be used to allow for data processing and retrieval from a variety of sources:



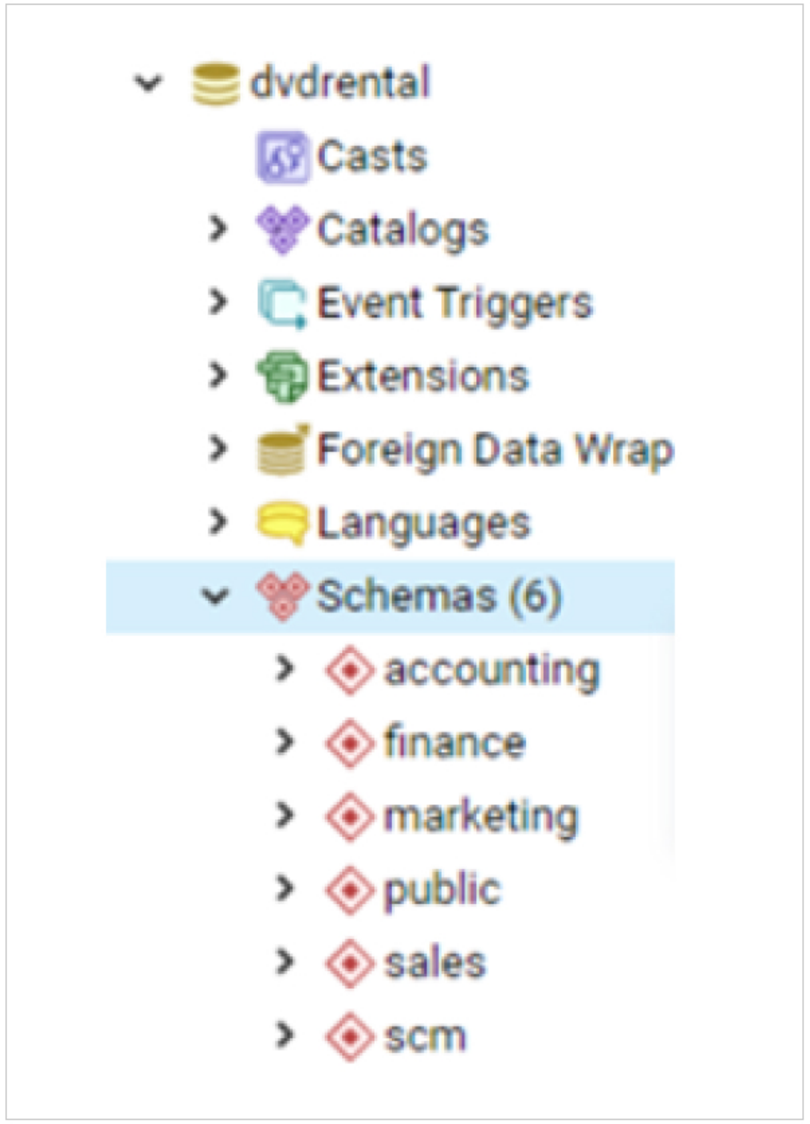
2.2.1 Database per Service

This pattern aims to make services loosely coupled by providing each service with its own data storage so that they can be developed, deployed, and scaled independently. Each service can then have a different type of data storage optimized for their usage, for example RDBMS, graph, or NoSQL.

There are three common approaches to database storage that are commonly used in microservices.



One database per service
Each service will have its own database isolated from other services to ensure that only the services that are responsible for the data output will manage the data.



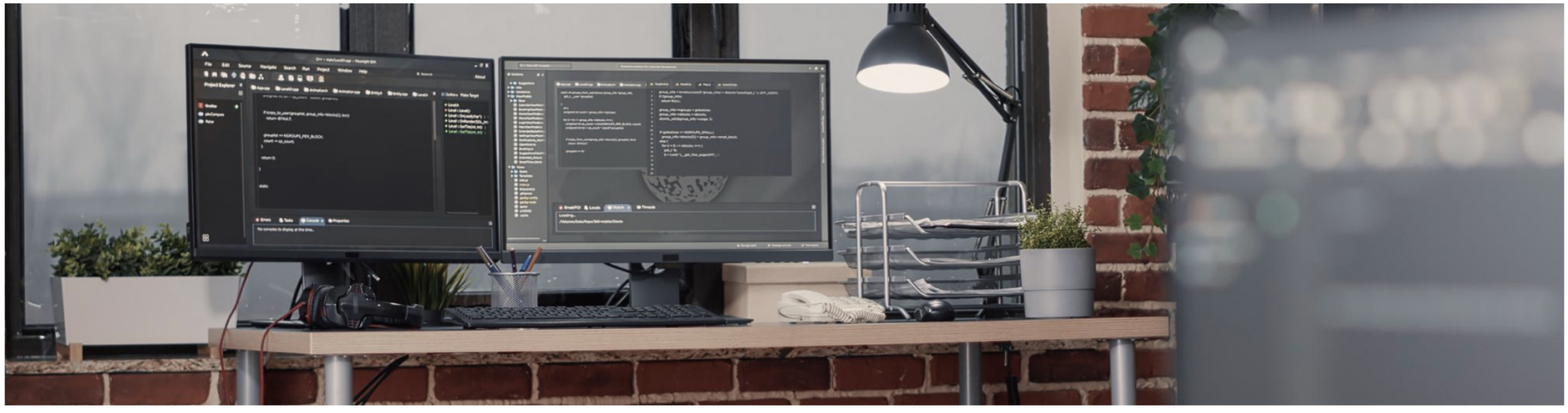
One schema per service
All services will use the same database; however, each service is assigned a specific schema.

Private tables per service
All services utilise the same database and schema; however, each service is assigned a specific table.

2.3 Communication Pattern

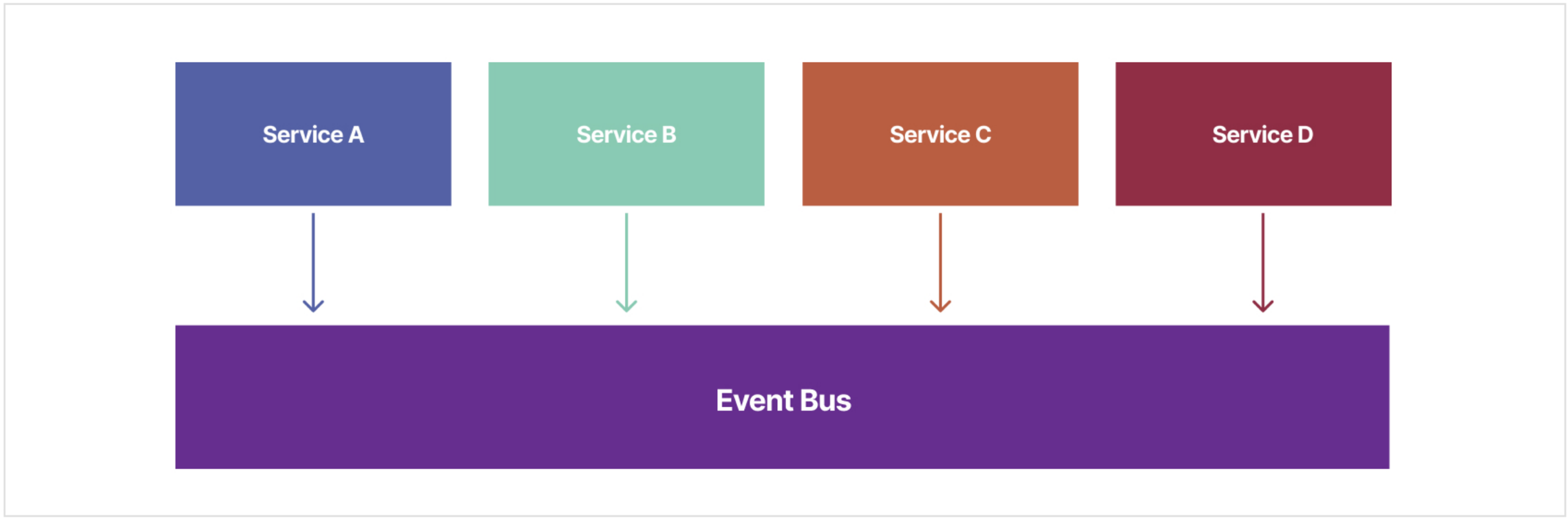
It’s a common practice in microservices that each service has its own data storage. But it’s also not unusual to see that one service might need to access data from other services. One could share the data storage of one service so that other services can access the data, but this is considered an anti-pattern, creating coupling between the services and removing the value of the microservices.

Communication patterns can then facilitate this scenario, where one service needs to communicate with others to perform its task whether that’s in the form of sending or receiving data. There are several patterns available that can be used for this and avoid coupling between each service and maintain the integrity of your microservice architecture.



2.3.1 Domain Events

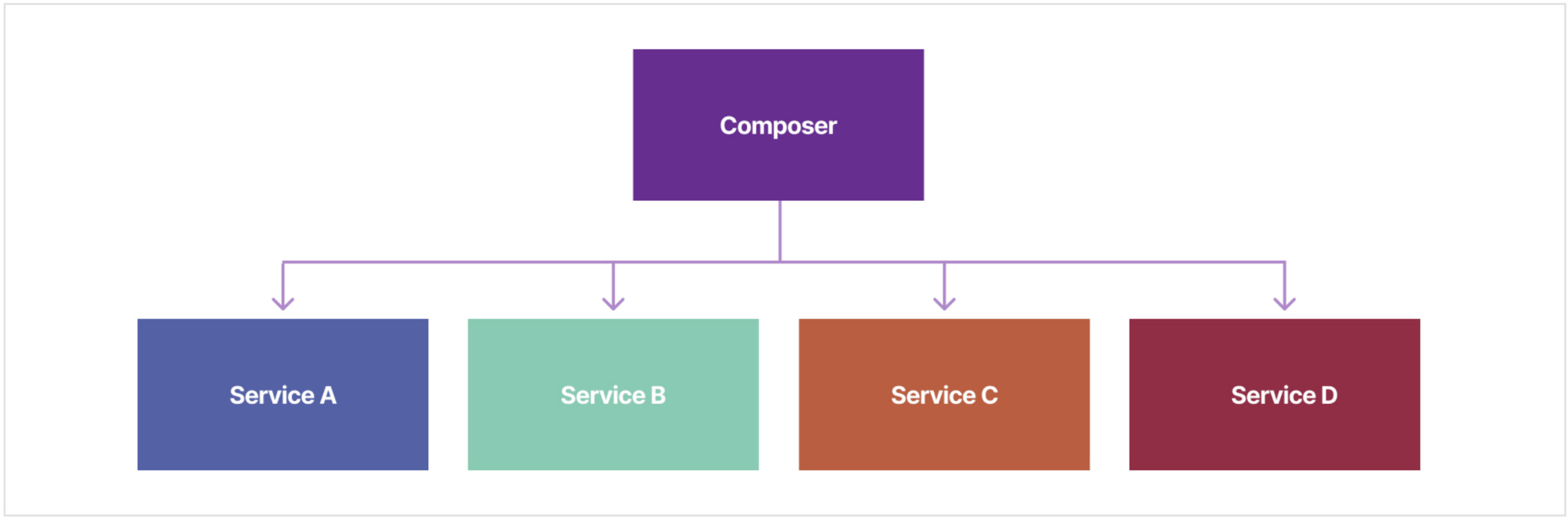
A Domain Events pattern aims to resolve the communication problems by providing a specific mechanism to share the data, without sharing the data storage by publishing any activities that happened inside one service. The activities could be data modification, creation, or deletion.



This pattern uses an event-driven approach, where each service publishes domain events for any domain activities that happened, such as when new data is created. Other services subscribe to the domain events and store the data for their own usage individually, which can be achieved through any asynchronous process, like an event bus.

2.3.1 API Composition

In some scenarios, there is a need to retrieve data across multiple services and aggregate the data, such as when generating a report. In this case, an API Composition pattern should be used. This pattern uses a “composer” as a component that will retrieve data from multiple services and aggregate them as necessary.

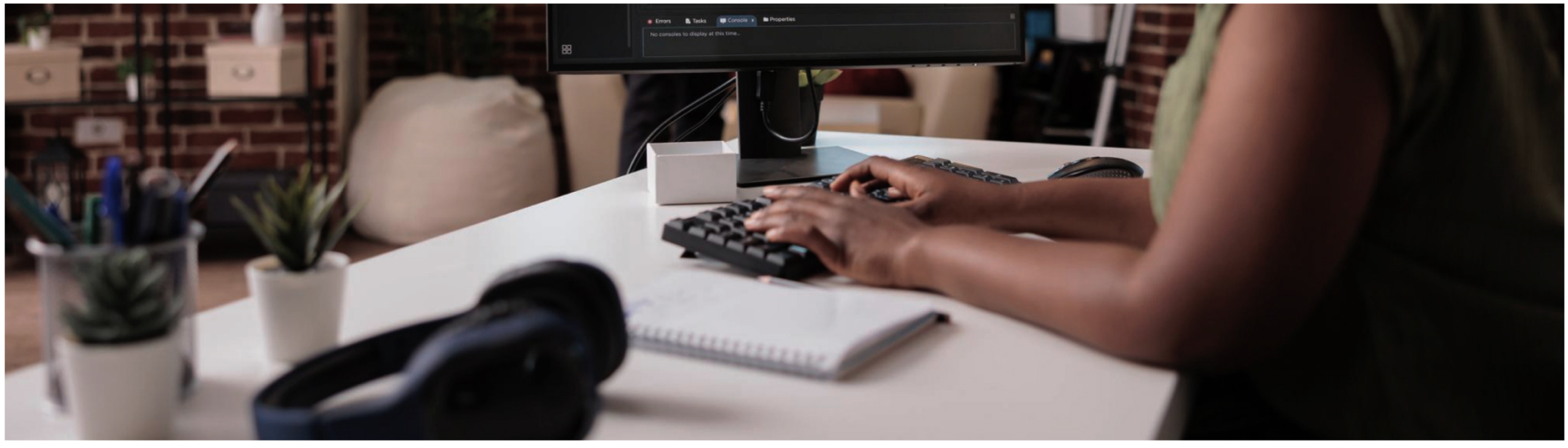


It is recommended to use asynchronous processing to avoid blocking calls, which will reduce the performance. Asynchronous processing calls to other services will be performed without waiting for other calls to finish first. Aggregating the data is performed after all asynchronous calls are finished.

2.4 Transactional Patterns

A transaction is a single unit of work that contains one or more processes, which should result in all processes being completed, or nothing at all. This concept is typically used in the database system to guarantee the consistency of the data. In monolithic applications, this is automatically managed by the database system, however, in microservices each service has its own database. Therefore, each transaction can be automatically managed by the database system if the transaction is performed in a single service. This is referred to as ‘Local Transaction’.

If the transaction spans across multiple services, then a global transaction is needed, which could be achieved through something like Saga as outlined below.

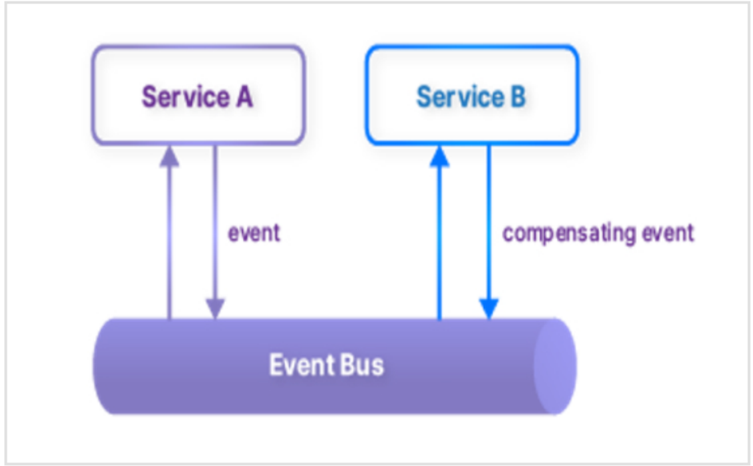


2.4.1 Saga

This pattern was first introduced by Hector Garcia-Molina in his paper “Sagas”. He explains that Saga is a long-lived transaction that can be written as a sequence of transactions that are interleaved with other transactions. In other words, Saga breaks a global transaction into multiple local transactions. There are two styles of Saga implementation - Choreographed and Orchestrated.

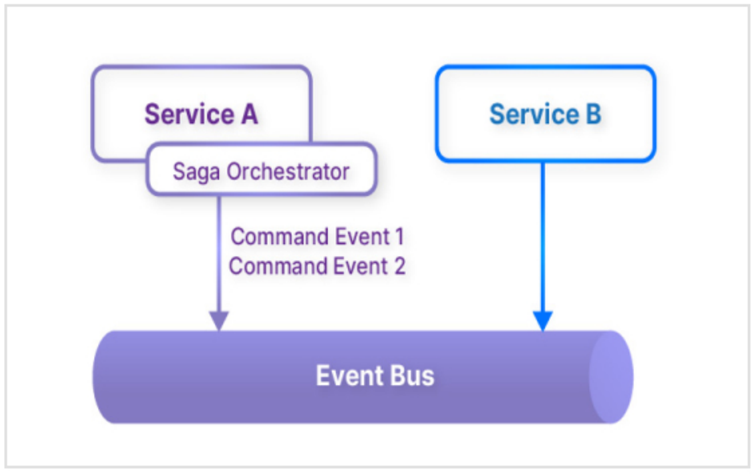
- **Choreographed Saga**

In a Choreographed saga, each local transaction publishes events that trigger local transactions in other services. This means that one service can publish an event that will be subscribed to by other services to create another local transaction. If one of the services failed to process the local transaction, they should publish a compensating event that tells the other participating services to undo the transaction.

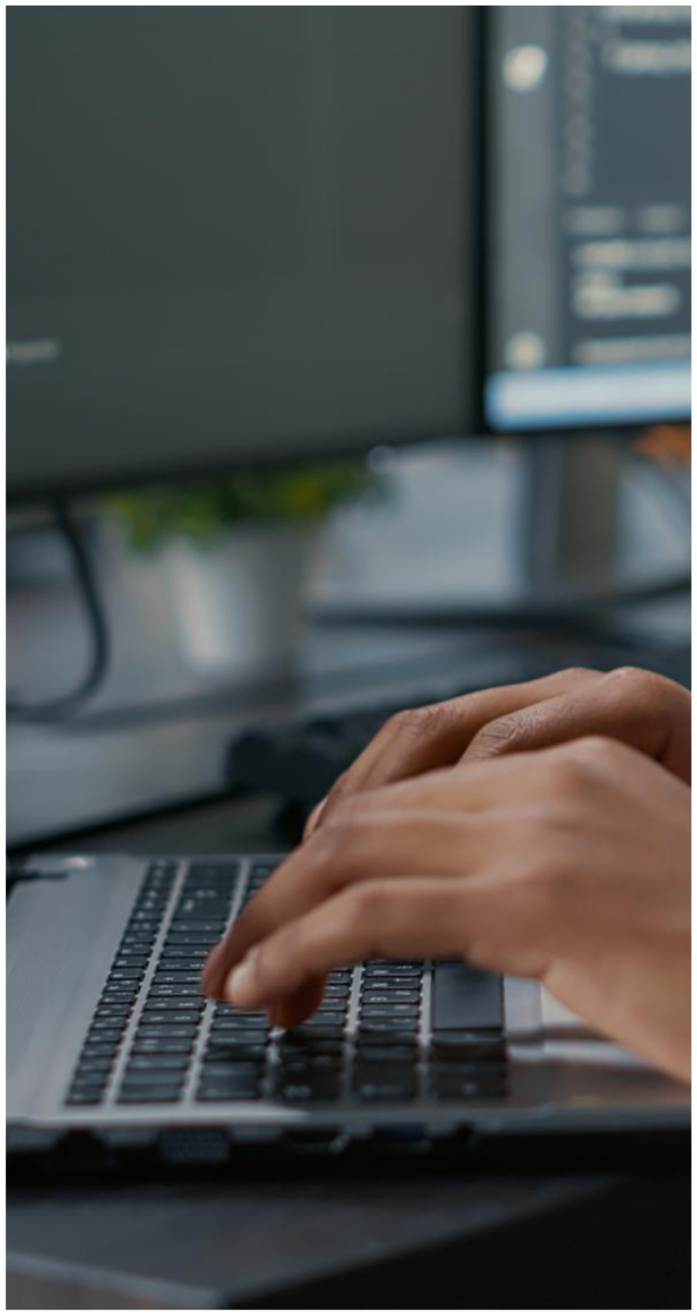


- **Orchestrated Saga**

In an Orchestrated saga, each local transaction is managed by an orchestrator. The orchestrator resides in the service that initiates the transaction. The orchestrator will publish a specific event for a specific service which tells the service to create a local transaction. If the transaction failed, the participating services would publish an event to inform the orchestrator, then the orchestrator will publish a compensating event that tells the other service to undo the transaction.



2.5 Deployment Pattern

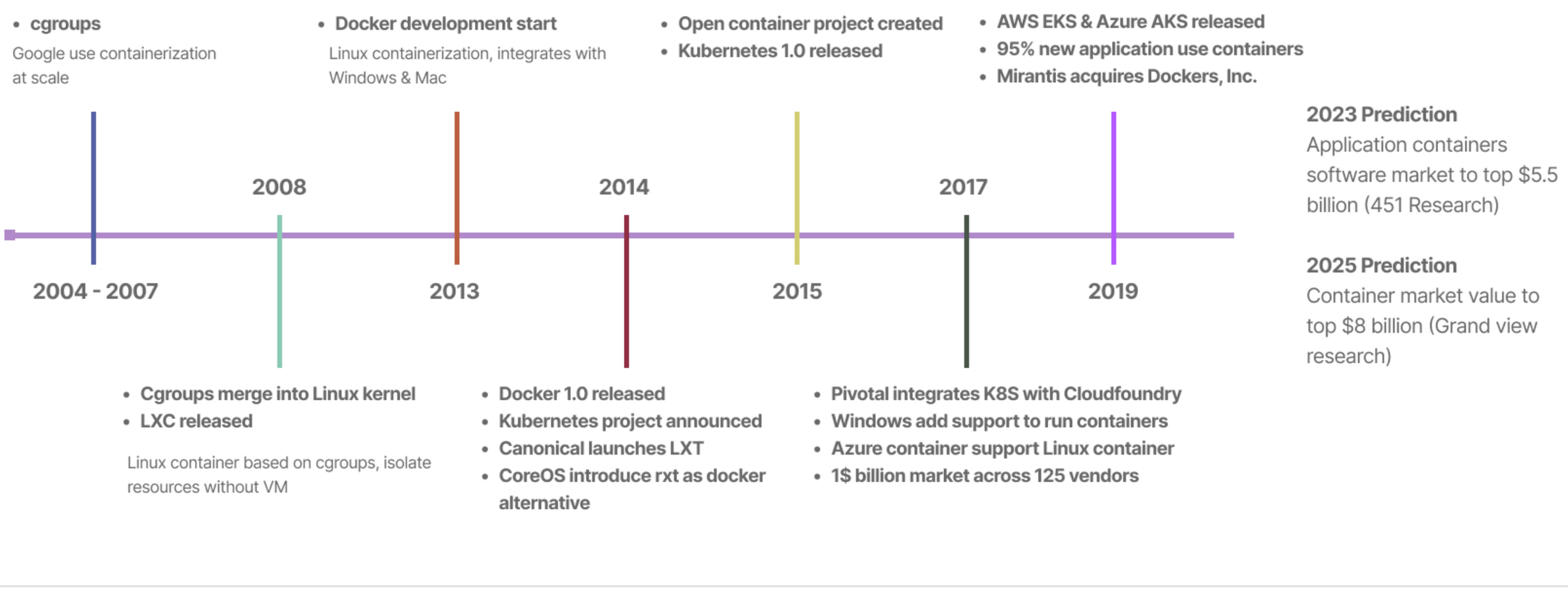


For a Deployment Pattern, there are several approaches that can be used:

- Multiple services per host, and all services are deployed into one host machine.
- One service per host, and each service is deployed as one host machine.
- One service per VM, and each service is deployed as one virtual machine.
- One service per container, and each service is deployed as one container.

The recommended approach is to use one service per container. As stated in the “Microservices Adoption in 2020” survey from O’Reilly, almost half (49%) of respondents who describe their deployments as “a complete success” also deploy most of their microservices (75-100%) in containers. The majority (83%) of respondents who describe their microservices efforts as “Not successful at all” are using some means other than containers to create their instances.

Containerization technology was originally developed in 2004. However, Docker popularized this technology in 2014 and now it has become best practice when deployingmicroservices. The graphic below outlines a brief history of Containerization technology.



There are several reasons why containerization technology is best practice for microservice architecture:

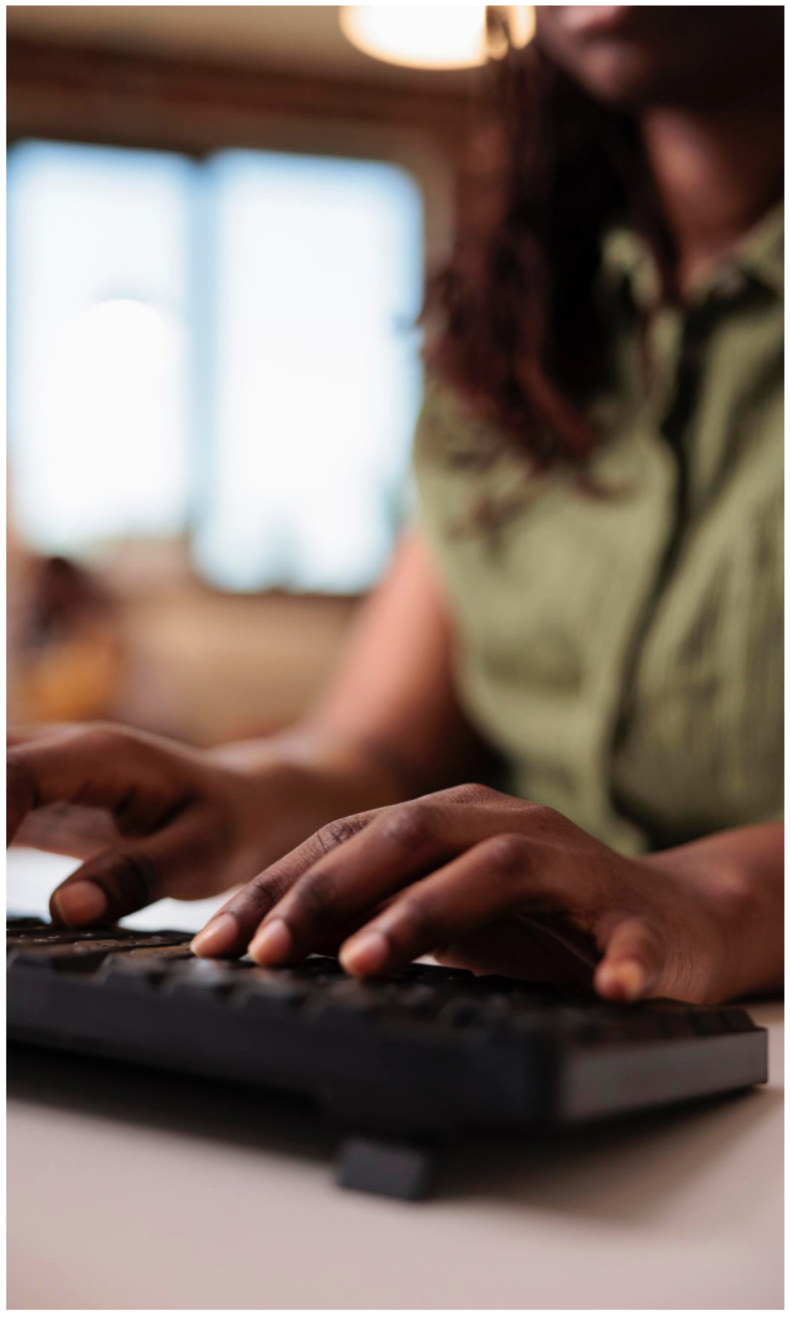
- Less Overhead, it uses a shared kernel (operating system) with the host machine.
- Portable, it can be easily deployed to multiple different operating systems.
- Isolated, each container is isolated from the others even if it is deployed in the same host machine.

Conclusion

Although microservices offer great advantages, they also introduce high levels of complexity to the architecture and can increase the failure risk. Microservices patterns aim to tackle this complexity and minimize the risks associated with microservices development and to give guidance on how to adopt a microservices architecture. None of this should veer organisations from moving towards a microservice architecture, as the value benefits are so great from speed of deployment to ongoing maintenance of code. Instead, developers, scrum masters and those looking to decompose a monolithic application should go into the project eyes wide open and consider the structure of deploying this architecture in your organisation before a single line of code is created. This is key to delivering on the incredible benefits of Microservice architecture.

If you are considering adopting a microservices architecture in your organisation Mitrais highly recommends revisiting the references below and adopting the Microservice patterns described in this paper.

And always remember Lewis's Zen Koan, "a microservice should be as big as my head!"



About Mitrais

Mitrais is a world-class technology company based in Indonesia and a part of the global CAC Holdings Group. We have been recognized as Indonesia's leading provider of offshore development services by Forrester Research, and our goal is to help your business meet and exceed your expectations. Combining Western innovation with Eastern productivity, Mitrais maintains its preeminent position in the Asia Pacific region. As a member of the Microsoft Partner Network with a Gold Application Development competency, we demonstrate the highest level of competence and expertise with Microsoft technologies. Our close working relationship with Microsoft enables us to deliver exceptional software development services. Through collaboration with trusted partners and our team of talented software engineers, we are committed to providing outstanding solutions to our valued clients.

Resources

1. Cloud Microservices - Global Market Trajectory & Analytics. Research and Market. https://www.researchandmarkets.com/reports/4804268/cloud-microservices-global-market-trajectory
2. Denman, James. Business Capability. Tech Target. https://www.techtarget.com/searchapparchitecture/definition/business-capability
3. Evans, Eric. 2003. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional
4. Fowler, Martin. Microservices. MartinFowler.com. https://martinfowler.com/articles/microservices.html
5. Hector et al. 1987. Sagas. Princeton University
6. Lewis, James. Scale, Microservices and Flow. Codecamp. The One with Architecture. https://codecamp.ro/architecture-conference-february
7. Mike et al. Microservices Adoption in 2020. O'Reilly. https://www.oreilly.com/radar/microservices-adoption-in-2020/
8. Richardson, Chris. Microservices Pattern. Microservies.io. https://microservices.io

MITRAIS | MEMBER OF CAC HOLDINGS GROUP

Terima Kasih　　**Thank** You　　**ありがとう**ございました