EVALUATION OF MICROSERVICE ARCHITECTURE PATTERNS

R.P.D.M Weerasooriya

(209393E)

Degree of Master of Science in Computer Science

Department of Computer Science and Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

July 2022

EVALUATION OF MICROSERVICE ARCHITECTURE PATTERNS

R.P.D.M Weerasooriya

(209393E)

Thesis/Dissertation was submitted in partial fulfillment of the requirements for the degree MSc in Computer Science specializing in Software Architecture

Department of Computer Science and Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

July 2022

Declaration

I declare that this is my own work and this thesis/dissertation does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other University or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: UOM Verified Signature Date: 2022-07-22

The above candidate has carried out research for the PhD/MPhil/Masters thesis/dissertation under my supervision. I confirm that the declaration made above by the student is true and correct.

Name of the supervisor: Prof. Indika Perera

Signature of the supervisor: UOM Verified Signature

Abstract

Microservices has been a very popular word in the Software Industry for quite a long time. Microservices architecture is regarded as a rising trend. There has been a rise in the research carried out in the field of microservices which encourage enterprise software architects and IT executives to witness and be a part of the new evolution. Most of the time we might only have a glance at such topics when our attention is caught. The concept of microservices describes a style of software systems that is highly effective in building enterprise solutions in the current times. The software industry has witnessed that there have been many companies such as Netflix, Amazon, Spotify who have benefited greatly with the use of microservices. So that for many other software organizations are rapidly adhering to incorporating microservices into their enterprise solutions. This is becoming the first choice style for building enterprise applications. But, however, there's not much guidance or information which will help a beginner to determine what the microservice style should be used in the project and how to do it. In order to fill this gap, and be of assistance for architects and developers to identify the most appropriate patterns which most suit their enterprise application, I aim to carry out my research targeted to find out and characterize various microservice architecture patterns reported in the known literature, and perform an evaluation of architectural patterns by case studies and with implementations[1].

ACKNOWLEDGEMENT

First and foremost, I would like to express my gratitude to my supervisor, Dr. Indika Perera for providing abundant guidance, support and encouragement throughout this research. Also I would like to thank my colleagues for sharing knowledge, support and constant encouragement. Last but not least, I express my love and gratitude to my parents for their love, help and support.

Thank you

Table of Contents

Abstract ii

ACKNOWLEDGEMENT iii

Table of Figures vi

LIST OF ABBREVIATIONS vii

1. Introduction 1
- 1.2 Motivation for the Research 3
- 1.3 Research Problem 4
- 1.4 Research Objective 5
- 1.7 Organization of the Thesis 6

2. Literature Review 7
- 2.1 Different Architectural Patterns 7
- 2.2 The Coordination Patterns in Microservices 7
- 2.4 Deployment Strategies & Patterns 7
- 2.5. Data Storage Patterns 8

3. Methodology 8
- 3.1.1 Pattern Identification 9
- 3.2 Implementation 10
  - 3.2.1 Client Side Discovery Pattern 10
  - 3.2.2 Server side discovery pattern 11
  - 3.2.3 API Gateway Pattern 12
  - 3.2.3 State management Pattern 13
  - 3.2.4 Deployment patterns 14

4. Evaluation 16
- 4.1 The Coordination Patterns in Microservices 16
  - 4.1.1 Fine Grained SOA 16
  - 4.1.2 Service Discovery Patterns 18
    - 4.1.2.1 The Client Side Discovery Pattern 19
  - 4.1.2.2 The Server Side Discovery Pattern 20

The Service Registry 21
4.1.3 The API-Gateway Pattern 22
Advantages 23
Disadvantages 23
4.1.4 Backends for frontends 24
Advantages Of BFF 25
Disadvantages of BFF Pattern 25
When to use BFF 27
Solve BFF Associated Problems 27
4.2 Managed State Patterns 29
Advantages 31
Disadvantages 31
4.2.2 Event-Driven State Management 32
4.2.2.1 Saga Pattern 33
Advantages 34
Disadvantages 34
4.2.3 (Event Sourcing) Replicating State in Layered APIs 35
4.3 Deployment Patterns 38
4.3.1 The Multiple Services per Host Pattern. 39
4.3.2 Single Service per Host Pattern 39
4.3.2.1 Service Instance per Virtual Machine Pattern 40
4.3.2.2 Service Instance per Container Pattern 40
4.3.3 Serverless Deployment 41
Single Service per host 43
Serverless 43
4.4. Data Storage Patterns 43
4.4.1 The Database-per-Service Pattern. 43
4.4.2 The Database Cluster Pattern. 45
Database Cluster Pattern 46
5. Conclusion 46
Future Work 48
REFERENCES 49

Table of Figures

Figure	Page Number
1 - Implementation of Client Side Discovery	18
2 - Client-side service discovery config	18
3 - Client side delivery	19
4 - Server Side Discovery Pattern	20
5 - API Gateway Pattern	21
6- Dockerizing an application	22
7 - Serverless applications	23
8 - Overview of Fine Grained SOA	25
9 - Overview of Service Discovery	27
10 - Client Side Discovery Pattern	29
11 - Server Side Discovery Pattern	29
12 - Overview of API Gateway	31
13 - Default Api Gateway Pattern	33
14 - Fan out	35

15 - Fuse	36
16 - Solve Fanout	37
17 - Message Oriented State Patterns	41
18 - Event driven state management	43
19 - Event Sourcing	46
20 Multiple Services Per Host Pattern	50
21 Single Service Per Host Pattern	51
23 Serverless Architecture Overview	52
24 Database per service pattern	56
25 Shared Database Pattern	57

LIST OF ABBREVIATIONS

- API - Application Programming Interface
- REST - Representational State Transfer
- SQL - Structured Query Language
- NoSql - Not Only SQL
- RDBMS - Relational Database Management Section
- HTTP - Hypertext Transfer Protocol
- SOA - Service Oriented Architecture
- GCP - Google Cloud Platform

1. Introduction

Microservices has been a very popular word in the Software Industry for quite a long time. Microservices architecture is regarded as a rising trend. There has been a rise in the research carried out in the field of microservices which encourage enterprise software architects and IT executives to witness and be a part of the new evolution. Most of the time we might only have a glance at such topics when our attention is caught. The concept of microservices describes a style of software systems that is highly effective in building enterprise solutions in the current times. The software industry has witnessed that there have been many companies such as Netflix, Amazon, Spotify who have benefited greatly with the use of microservices. So that for many other software organizations are rapidly adhering to and incorporating microservices into their enterprise solutions. This is becoming the first choice style for building enterprise applications. But, however, there's not much guidance or information which will help a beginner to determine what the microservice style should be used in the project and how to do it[2].

In higher abstraction, one could say that the microservice architectural style is a way to break down a large traditional monolithic application into a number of smaller cohesive services. Every microservice is equipped with the needed resources and communicates in lightweight means such as RESTFul APIs. In the microservice architecture domain, Each microservice could be developed and deployed by a different set of developers independently. These services would not have any restriction on the programming languages and the data storage technologies of each microservice's requirement. These qualities make microservices highly compatible with building high fluid business applications.

Microservice architectures should have high adaptability to accommodate new business requirements that are coming their way. With agile practices being followed in almost all the software development organizations to tolerate inevitable change, building a system in the modern world is not trivial by any means. Number of modifications and new releases required to ensure the customer experience and the accuracy of the business contracts could be daunting.

It is highly improbable that any organization would have the opportunity and freedom of coming up with a microservices architecture from their very first project. It will be a journey full of learning new lessons of different microservice architectures, and technologies. The path to glory of perfection is driven by adaptability and extensibility to fulfill new customer requirements. The ability to concentrate on what could be done right now while also facilitating the development teams to move towards a more productive team is one of the main advantages of adapting to a microservice based ecosystem[3].

1.2 Motivation for the Research

Enterprise software development is an area of research that constantly changes with time. New technologies, inventions along with customer requirements make it a great area of research for anyone interested in computer science. For any enterprise application, the architecture acts as the brain and the heart of it. So it is critical to design your system with a high quality architecture for a great customer based application.

Microservices has been a buzzing topic for some time and lots of large monolith applications are transformed into microservice based systems [2], [5], [6]. Amazon, Spotify, Netflix are some of the world wide giants who have benefited highly by the use of microservices [7], [8]. So it's an important topic that we should be researching passionately about the software engineering field to harvest the rewards for our organizations by learning from the above mentioned industrial giants.

1.3 Research Problem

One of the major concerns that most of the architects and experts in this domain have faced over the last few years with this is that many of them understand it as a prescriptive architecture. Hence there's this conception rooted that certain things must be done in one certain way only. But what has been noted by large tech giants repeating the benefits of microservices is that adopting microservices this way is not practical and can result in huge financial losses. There's a high chance that organizations might not be able to reap those advantages if they do not encourage themselves to be adaptable with new changes but hang on to the initial prescriptive approach in building microservices[5].

Implementation of microservices should be done sensibly and realistically so that it is based on practical rather than theoretical considerations, so that it adheres to the needs of the organization. Every company is not another Spotify or Uber who are giants in microservice based development. Your approach should be to use microservice patterns in a way that best suits your needs.

What the domain experts in this field strongly feel is that there is no one ultimate microservice pattern that yields you all the advantages. It is proposed to consider microservices as a set of many patterns which could exist in any given enterprise application depending on your organization's needs.

It's recommended that the Microservice patterns should inherently be non-monolithic. So we can develop and manage them independently leveraging different development teams. This also helps us to tolerate change easily. This concept is similar to the concept of using databases. We use databases to persist application data but they all share similar goals with different user requirements such as scalability, maintainability etc.

For example, Relational Databases, NoSQL databases, Big Data databases, etc. all share once distinct similarities. That is they can be used to manage data. But the specifications and priorities of each choice could be different[6].

Even though microservices are being used in the industry in a large number of organizations, there are still challenges and problems arising in dealing with microservices.

Architects and Developers are compelled to use designs which are popular, even those patterns are not the most applicable ones to their problem.

In order to fill this gap, and be of assistance for architects and developers to identify the most appropriate patterns which most suit their enterprise application, I aim to carry out my research targeted to distinguish between all the known microservice architecture patterns, and perform an evaluation of architectural patterns by case studies and with implementations.

1.4 Research Objective

I'm planning to perform an evaluation of architectural patterns of microservices in terms of development complexity, interoperability, performance and throughput. This includes patterns in orchestration, coordination, designing, deployment and data storage of microservices. I believe someone starting in this domain would find this useful when it comes to deciding which pattern(s) to be used in their enterprise application which are critical. There is a gap of research evaluating all these patterns in the perspectives of developers and enterprise architects.

- Uniquely Identify each microservice pattern with its distinct characteristics.
- Perform a critical evaluation of these patterns in terms of development complexity, performance, cost etc.
- A developer/architect entering into this domain should be able to make use of this evaluation and decide on the optimal pattern to be used in building the enterprise application.

1.7 Organization of the Thesis

- Chapter 2 focuses on literature review along with some of the microservice architecture patterns.
- Chapter 3 stresses on methodology of how different patterns were identified and implemented.
- Chapter 4 explains the evaluation details of sub architecture patterns in each architecture pattern.

2. Literature Review

2.1 Different Architectural Patterns

Architectural Patterns can be described as a collection of implementation decisions that could be used to solve a design problem over and over again. The architecture patterns used in building your microservices are critical as it affects all cross cutting concerns including performance, ease of development, maintainability etc. So it’s a very important decision to pick which architectural pattern(s) you’re going to incorporate in your solution. A poor decision might cause the enterprise application to perform poorly and the users of the application would be dissatisfied. Similarly, very good choices will allow your application to perform seamlessly catering all the user base. Thus, we identify the following patterns with their distinct pros and cons to give architects an idea of what each pattern could benefit with. Four commonly used patterns emerge [8].

- Orchestration and Coordination-oriented architecture patterns
- Patterns related to service deployments on hosts and virtual machines.
- Patterns associated with data management

2.2 The Coordination Patterns in Microservices

The following contains a taxonomy of the few microservices patterns identified under this topic which are widely adopted in the microservices domain. Each organization can choose to make choices based on pros and cons that are prioritized by the organization more. The recommended approach is to design and implement a microservices architecture using a mixture of these patterns.

These patterns are not dedicated to any type of organization. However, they can be applied to develop any kind of enterprise application if the domain and the requirements could be fulfilled by a given pattern. To put this in another way, what you want to achieve has a higher probability with a collection of microservice patterns, instead of adopting one pattern of microservice architecture .

2.4 Deployment Strategies & Patterns

The following Patterns address the widely used practices used when microservices are required to be deployed in a production environment in a cloud infrastructure. Deploying even a monolithic application takes up lots of work. Your deployment cluster might consist of many hosts. And then you may run many instances of your service in the cluster. It can be accepted that the deployment of any critical service in a short timeline is not trivial.

When it comes to microservice based applications, the deployment pipeline would seem more complex. A microservice-based enterprise application might have many deployments. Each microservice has its own resource requirements. There might also be system requirements to spin up many instances of each service based on the traffic so every client is successfully served. In order to cater these requirements, there are many patterns that any organization could make use of[9].

2.5. Data Storage Patterns

Even in a monolithic system manipulation of data should be done with lots of care. When it comes to a microservice-based architecture it could seem more complicated due to the sheer volume of data and the user queries. If the architects could model the data in a way it's optimized for all the user queries, it would help to reduce the latency of database queries thus increasing the customer experience[10].

3. Methodology

3.1.1 Pattern Identification

I used a literature review protocol along with a methodical analyzing process to determine the patterns from the industrial practices and the known literature . The plan of my research is to identify the main architectural styles and then compare and contrast them with respect to their pros and cons. To identify key patterns with their advantages and disadvantages I referred to the following literature sources.

- Existing Literature based on Microservice Architecture Patterns
- Whitepapers issued on microservice architecture
- Industrial Experts/ Domain Experts

The following main patterns were identified based on the microservice lifecycle. Each of these patterns has lots of sub patterns embedded in them which will be discussed later.

1. Orchestration/Coordination-oriented architecture patterns
2. Deployment strategies
3. Data management

   In addition to the literature review, I thought that state management also can be regarded as a microservice pattern as many of the domains are inherited by adapting state management patterns based on events.

3.1.2 Steps for evaluation.

I'm planning to use literature reviews (research papers/white papers) based on microservice architectures along with the experiences I've gained throughout the 3 year career that I've worked in many leading software engineering companies to evaluate the microservice patterns, I define my goals as follows:

- Developing POC on currency conversion system using each microservice architectural patterns to understand the attributes, and characteristics and perform a better evaluation[13].
- Evaluate each pattern in terms of development complexity, interoperability and performance and throughput for enterprise software solutions.
- Some of the architectural patterns were implemented with hands-on coding to understand their implementation complexity.

3.2 Implementation

3.2.1 Client Side Discovery Pattern

Following is a sample of code implementing client side discovery using the Spring Framework's RestTemplate

```
@Component
class RegistrationServiceProxy @Autowired()(restTemplate: RestTemplate) extends RegistrationServ
ice {

  @Value("${user_registration_url}")
  var userRegistrationUrl: String = _

  override def registerUser(emailAddress: String, password: String): Either[RegistrationError, S
tring] = {

      val response = restTemplate.postForEntity(userRegistrationUrl,
        RegistrationBackendRequest(emailAddress, password),
        classOf[RegistrationBackendResponse])
       ...
}
```

Figure 1 - Implementation of Client Side Discovery

```
@Configuration
@EnableEurekaClient
@Profile(Array("enableEureka"))
class EurekaClientConfiguration {

  @Bean
  @LoadBalanced
  def restTemplate(scalaObjectMapper : ScalaObjectMapper) : RestTemplate = {
    val restTemplate = new RestTemplate()
    restTemplate.getMessageConverters foreach {
      case mc: MappingJackson2HttpMessageConverter =>
        mc.setObjectMapper(scalaObjectMapper)
      case _ =>
    }
    restTemplate
  }
```

Figure 2 - Client-side service discovery is configured using various Spring Cloud annotations[11]

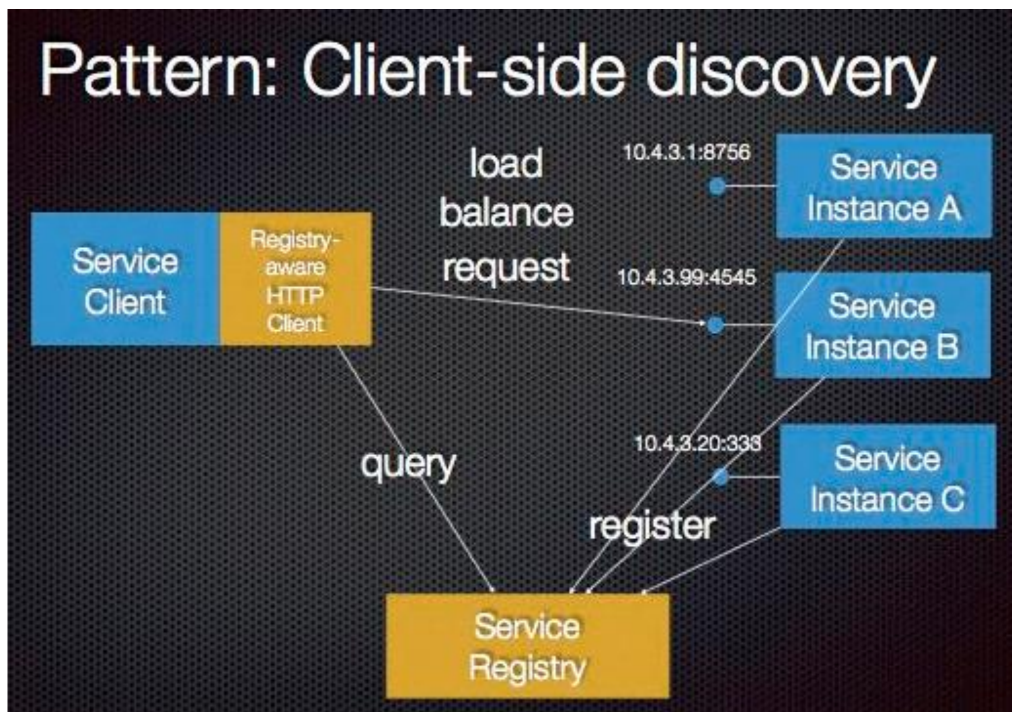

Figure 3 - Client Side Discovery Pattern, **Source** [11]

3.2.2 Server side discovery pattern

An Elastic Load Balancer (ELB) is an example as it acts as a discovery router. When a user creates an HTTP call to the ELB, it balances the traffic amongst instances listening to it.. A load balance balances the traffic coming from the Internet. A load balancer also functions as a Service Registry.

Container orchestrators like Kubernetes, possess a proxy service on every host which renders the services of a service discovery. To call a service, the user interacts first with it using the namespace. Then this service-discoverer is responsible for forwarding the request to one of the containers of the service anywhere in the cluster.

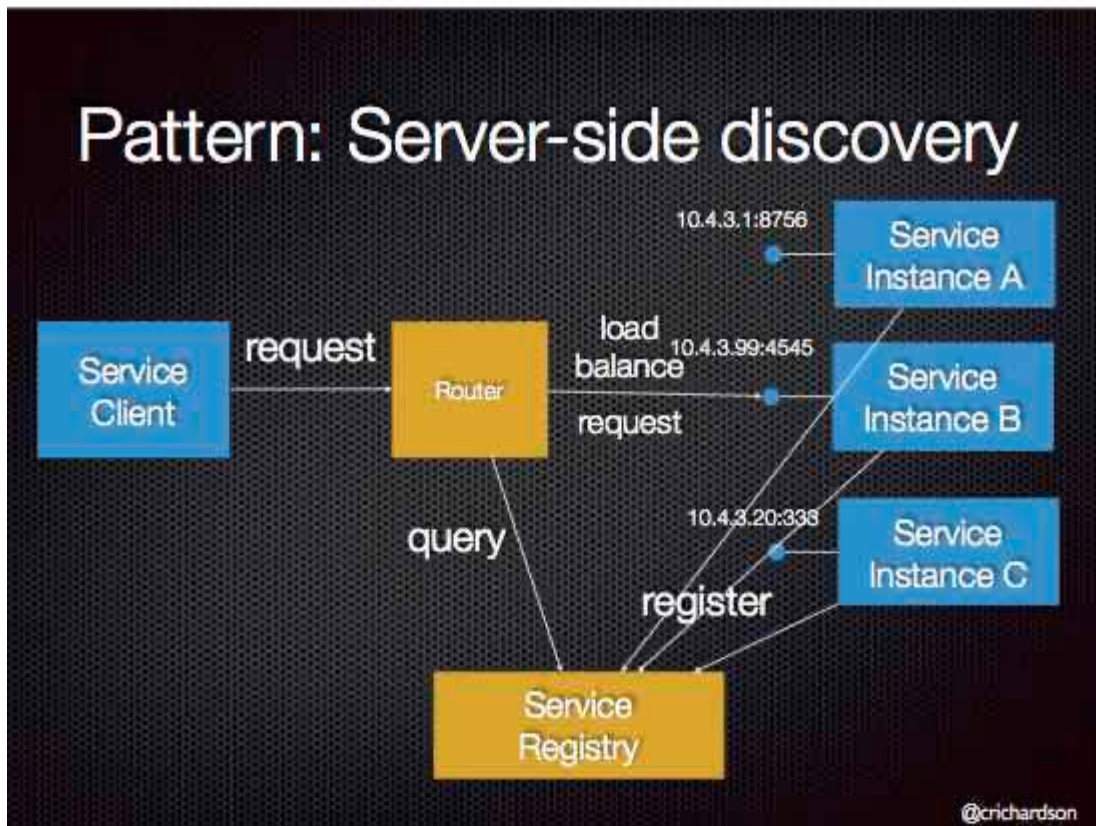

Figure 4 - Server Side Discovery Pattern **Source** -[12]

3.2.3 API Gateway Pattern

Microservices are able to expose their smart contract endpoints to others through a REST API. When building an enterprise, an aggregation of requests is needed on the server side as one client request may need to go through multiple microservices until the desired result is achieved. API-Gateway can be considered as a commonly recommended approach to cater this requirement (Figure 3). This is also a pattern which is used quite widely in the software industry. API Gateway can be considered as the interface that accepts the user requests and forwards them to the relevant backend microservices, also triggering other microservices and aggregating results for users. It provides a dedicated API to each client to accept the client query. And also it performs tasks such as manipulating data and implementing common cross cutting concerns like authentication. And also it can be responsible for tasks such monitoring

which makes it easier for the devops team. I tried implementing the api gateway pattern using the Spring Framework.

```
@Configuration
public class ApiGatewayConfiguration {

    @Bean
    public RouteLocator gatewayRouter(RouteLocatorBuilder builder) {
        return builder.routes()
                .route(p -> p
                        .path("/get")
                        .filters(f -> f
                                .addRequestHeader("MyHeader", "MyURI")
                                .addRequestParameter("Param", "MyValue"))
                        .uri("http://httpbin.org:80"))
                .route(p -> p.path("/currency-exchange/**")
                        .uri("lb://currency-exchange"))
                .route(p -> p.path("/currency-conversion/**")
                        .uri("lb://currency-conversion"))
                .route(p -> p.path("/currency-conversion-feign/**")
                        .uri("lb://currency-conversion"))
                .route(p -> p.path("/currency-conversion-new/**")
                        .filters(f -> f.rewritePath(
                                "/currency-conversion-new/(?&lt;segment&gt;.*)",
                                "/currency-conversion-feign/${segment}"))
                        .uri("lb://currency-conversion"))
                .build();
    }
```

Figure 5 - API Gateway Pattern **Source** -[13]

3.2.3 State management Pattern

State management patterns were implemented using event brokers. An event broker is a component for routing events with low latency with high tolerance. Applications and microservices are decoupled from each other when they communicate via the event broker which offers a significant advantage over the other patterns. Events are published to the event broker via topics following and they are delivered to all the microservices which are subscribed/listening. AWS SQS and Kafka brokers were used in the implementation of state

management patterns[2]. The experience gathered while working with the event-driven communication in the restaurant industry domain was valuable for the evaluation.

3.2.4 Deployment patterns

Few methodologies were undertaken to evaluate the deployment patterns. Applications were dockerized to run in ECS.

```
FROM node:6.9.5

ENV APP_DIR /app

RUN npm install -g yarn
RUN npm install -g pm2
RUN npm install grunt -g

RUN mkdir -p "$APP_DIR"
WORKDIR $APP_DIR

COPY package*.json ./
COPY . .

RUN npm install
RUN npm run build
RUN cp -r .build/* .

RUN yarn --production --pure-lockfile

CMD pm2-docker pm2.config.json
```

Figure - 6 - Dockerizing an application [13]

```yaml
# serverless.yml
service: my-express-application
plugins:
  - serverless-offline
provider:
  name: aws
  stage: dev
  region: us-east-1
  profile: default
functions:
  app:
    handler: index.handler
    events:
      - http:
          path: /
          method: any
      - http:
          path: /*
          method: any
```

Figure - 7 - Serverless applications were deployed in aws serverless [13]

I tried to develop code bases and try out different architecture patterns. Some of the implementations are included here [13].

4. Evaluation

4.1 The Coordination Patterns in Microservices

4.1.1 Fine Grained SOA

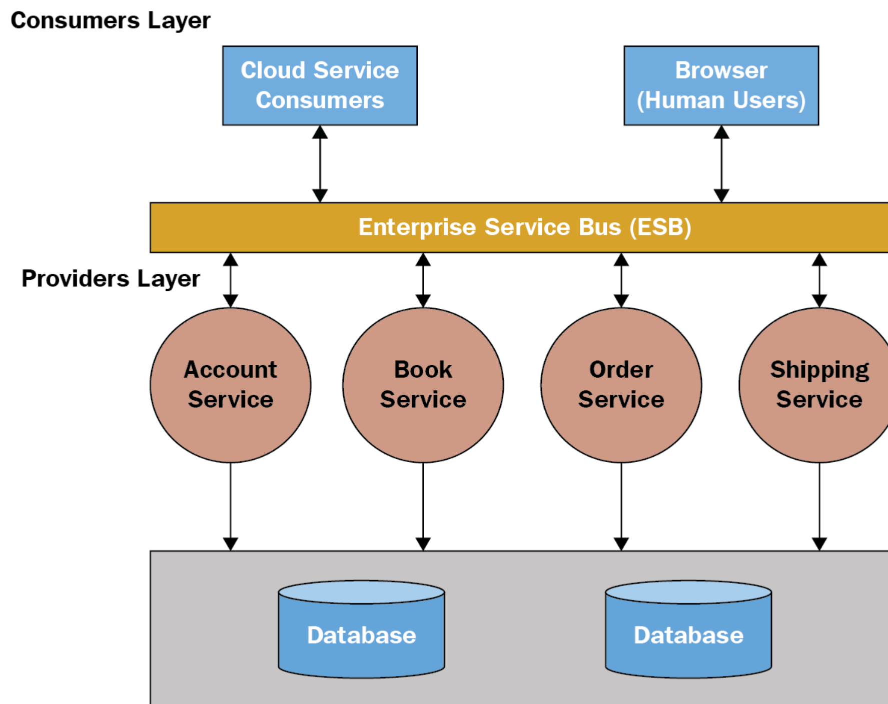

Figure 8 - Overview of Fine Grained SOA, **Source** Packethub [14]

Many domain experts claim that Netflix started this pattern.. Fine-grained SOA as the name suggests it reduces the problems faced by developers with SOA, and applies the same fundamentals. The key difference here is that it focuses on creating more-fine-grained layers[4].

When you break down the components into more fine-grained layers, and then try to include scalability, some inherited issues with the approach could spring up. For example Where you used to make a single http request with the previous implementation of the application, now you have to implement tens of http requests. This is inefficient. When you were used to managing a lower number of components, now you endup managing tens or hundreds or more. The monitoring tools and all the infrastructure are required to be modified.

When tens and hundreds of microservice can cause modifying the state of your central applications, the consistency of that data is compromised as data is accessed by every

microservices without any standards. One of the challenges associated with microservices include spaghetti interprocess communication.

In most scenarios, this pattern could be considered as an extension of service-oriented pattern, where the goal of each service is to provide an interface to other third-party systems. This sometimes ends up creating dependencies with those external systems. Monolithically developed services are too difficult to be modified, scaled and developed. Fine-grained SOA is quite similar to API-led approaches but impacts highly when it's used with existing systems.

Impacts

- ◆ Traffic increases.
- ◆ High maintenance cost.
- ◆ Outdated monitoring solutions are not feasible.
- ◆ The CICD pipeline becomes critical.
- ◆ The need for an orchestrator..

Key Characteristics

- It works well when the traffic is low, but problems emerge when the system needs to be scaled.
- It focuses more on integrations, with microservice having to depend on external systems.
- Inter-process communication is inefficient.
- State management and data consistency is not efficient

4.1.2 Service Discovery Patterns

If you are writing code to call another service that offers an API, you should know the IP and port of the service. Most of the applications in the past are running on physical hosts and their Ips and ports of the services are not varied. Hence you can hard-code these configurations in your code and they would be just fine most of the time.

But in the modern world, this is a critical issue with thousands of microservices in one system. A gist of the issue is illustrated in the following diagram.

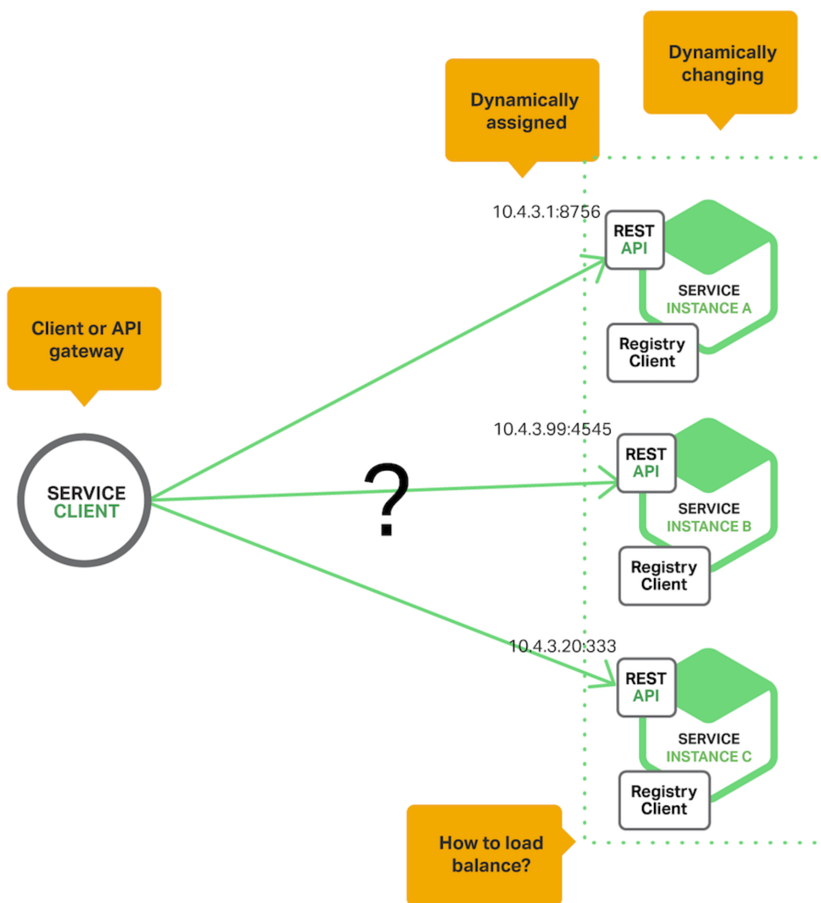

Figure 9 - Overview of Service Discovery

4.1.2.1 The Client Side Discovery Pattern

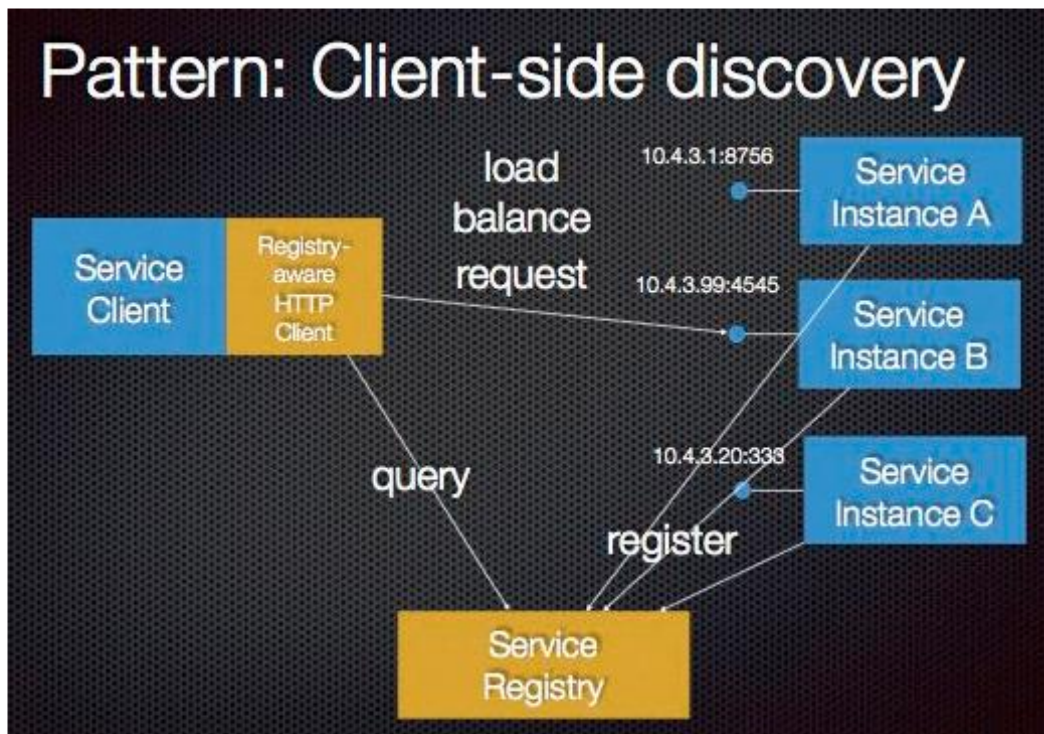

Figure 10 - Client Side Discovery Pattern, **Source** [14]

Every service instance in the enterprise application registers their IP in the service registry when it first enters the ecosystem. This information of service registration of microservices information is updated periodically using a peer-to-peer protocol named heartbeat mechanism.

This pattern is quite simpler than the client side service discovery mechanism.. Because the client is aware of the running services, it can make desired routing decisions more efficiently.

Microservices should interact with each other to complete a given business transaction. In a traditional application, services call another service through a RPC-like method. In legacy system deployment, services are configured to run at predefined hosts and ports. But in the modern world, microservices are mostly deployed in a cloud containerized environment. The cluster configurations change regularly and the legacy approach could not meet these requirements.

Client gets the location of the targeted service from the Service Registry before making a HTTP request. The Service Registry maintains a registry of locations of all microservices in the system.

Advantages

- Less network hops than the network hops in the Server-side Discovery

Disadvantages:

- This pattern ends up creating a dependency between the service components
- The logic needed to retrieve and manage service discovery information needs to be implemented in each of your clients which is troublesome.

4.1.2.2 The Server Side Discovery Pattern

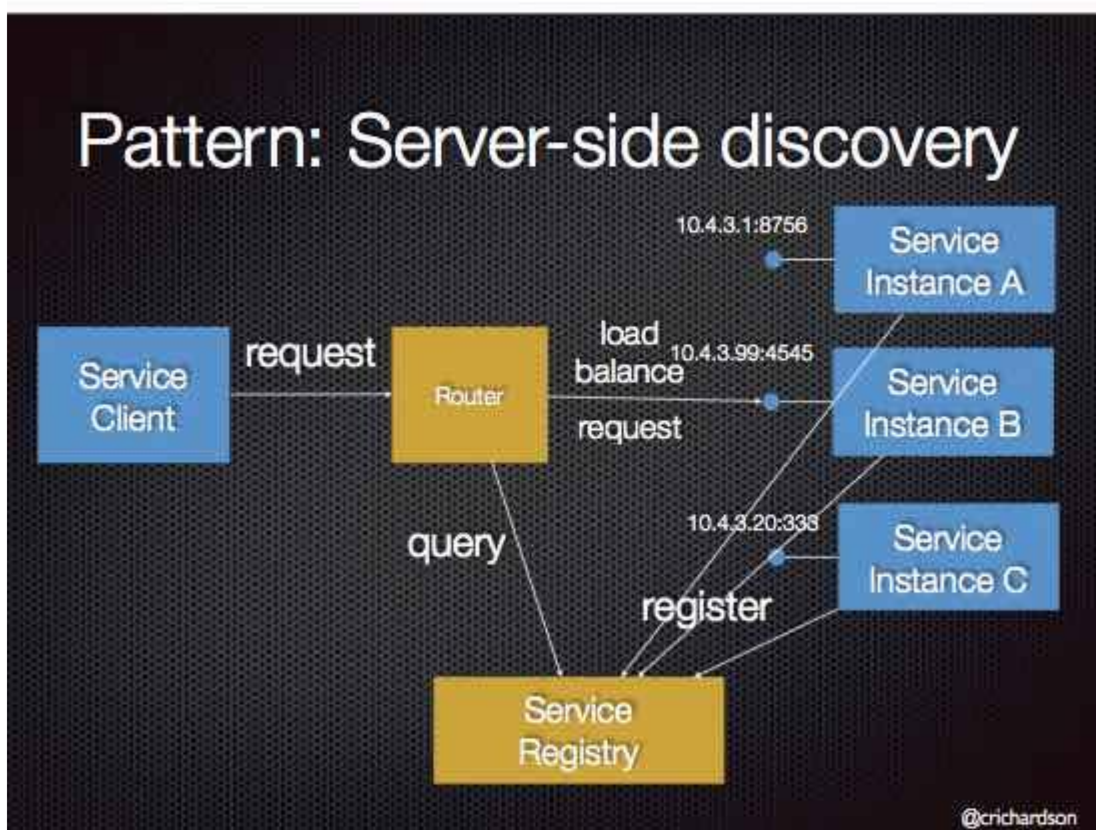

Figure 11 - Server Side Discovery Pattern, **Source** [15]

Container orchestrators like Kubernetes, possess a proxy service on every host which renders the services of a server-side discoverer. To communicate with a service, the user interacts first with the service-discoverer using the port. Then it is responsible for forwarding the request to one of the containers of the service anywhere in the cluster.

Advantages

- The client code is made very simple because it does not have to implement the logic of service discovery. The client could freely make a request to the load balancer. e.g. AWS Application Load Balancer

Disadvantages:

- The application load balancer/router should be set up and configured. The availability, scalability should be managed by itself.
- The application/load balancer router should support HTTP, TCP or Websocket.
- More network hops compared to that of Client Side Discovery

The Service Registry

The service registry acts as a key component in the concept of service discovery. It acts as the data persistence store of Ips and ports of other components in the system. It should be highly scalable to meet high traffic. If Clients are configured to store network URLs from the service registry, those details might become stale and clients might not be able to interact with other components.

When any component bootstraps its lifecycle, first it registers its network details with the service registry using a POST request. After that periodically (generally 30 seconds) the service will update the network details using a PUT request. Generally when a service gracefully terminates from the system, the registration information is removed. Furthermore the entries could be configured with a TTL so registration times out. Any other microservice could obtain the registered service instances by querying the service registry, generally by HTTP GET request.

Client Side Discovery Pattern	Server Side discovery pattern
• Complex client-side code to handle service discovery	• Client code is much simpler as no extra code needs to be implemented
• Less network hops	• More network hops
• Creates a dependency between the client and the Service Registry	• Some cloud environments provide this functionality, e.g. AWS Elastic Load Balancer

• Have to implement client-side service discovery logic in every client.	• The load balancer/ router should be a separate component in the system and should be configured. It should support scalability and high availability
	• Supports HTTPS, HTTP, TCP and web sockets

4.1.3 The API-Gateway Pattern

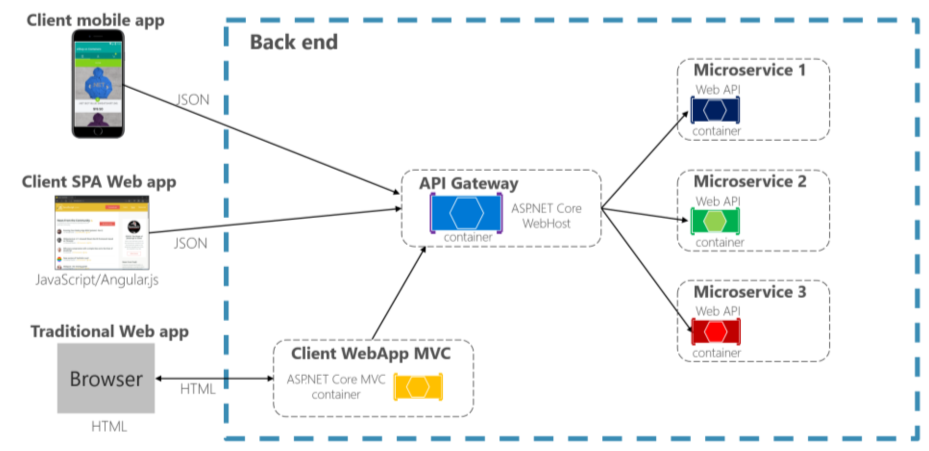

Figure 12 - Overview of API Gateway Pattern

The API Gateway pattern tries to address the following issues.

- Microservices generally supply a fine-grained API. Hence the clients need to interact with multiple services to get one business use case done as it may span across multiple microservices.
- API Gateway could be used to expose a dedidated API for each type of users in your application such as web user, mobile user or any other third party users
- It acts as the interface which separates the internal service components from the public internet. Hence this could be used to enable monitoring of requests, apply common functions like authorization, embed correlation ids to requests for easy monitoring.

- The clients should be adaptable and tolerable to all types of backend services as they are configured to use a diverse set of protocols which might not be web friendly or mobile friendly at all

Advantages

Without providing one sized API that fits all types of clients, the API gateway can expose a different API for each type of client.

Eg- verify that the client request is authenticated to perform the request.

Extensibility is another advantage that is provided with this pattern. It's convenient that this pattern can provide custom APIs quite easily. It's easier to roll out new features compared to other architecture patterns. The Services could be easily modified to have new business functionality based on client needs.

Api Gateway ensures backward compatibility for new services, so that existing clients are not affected by any endpoint changes.

Disadvantages

Api Gateway might seem like a bottleneck. The gateway interface is a critical interface for every client request. If the api gateway is not configured in a proper manner, it could have adverse results on the system and be a bottleneck in terms of implementation.

When every client is dedicated with a predefined custom API, it'll create a lot of hassle when keeping an audit of what version of API is used what client

When the number of microservices in a system increases drastically, the scalability of the API gateway would be limited at some point.

4.1.4 Backends for frontends

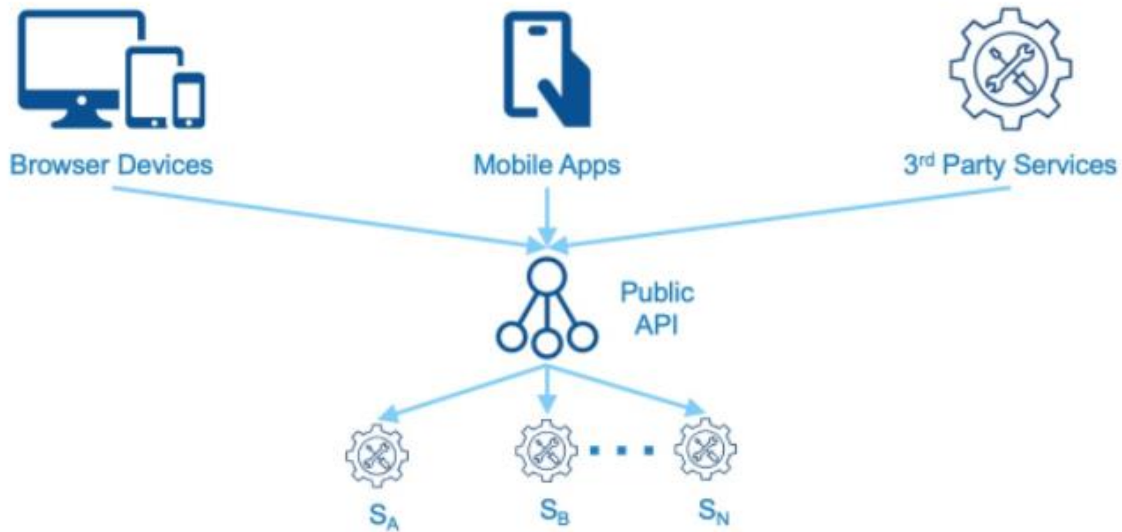

Figure 13 - Default Api Gateway Pattern[17]

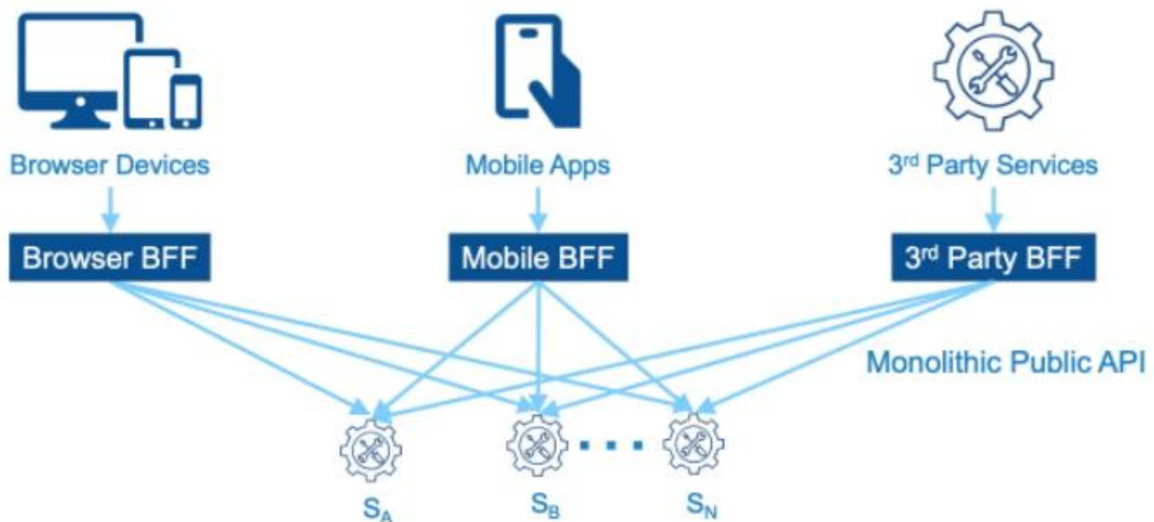

Figure14 - Overview of backends for frontend, **Source** - [17]

A Backend for Frontend is evolved from the API Gateway pattern. When it comes to enterprise applications that support many types of end users such as web app clients. Mobile

app users, desktop application clients, this particular pattern is highly suitable. A BFF is conceptually, a thinner layer between the client and the backend APIs. The process starts when a client (web/mobile) user requests data. First their request is passed through the BFF and into a common backend layer. BFF acts as a “translator” layer, which is a shim that takes these calls and converts them into a common form so that the backend api can understand it. Even Though we are adding a middle layer between the client and the backend, BFF is small pieces of code.

In this pattern, a group of calls is made by various types of clients. And they are being translated into a single backend call. The different types of data translations are carried out when data is fetched from the API to the BFF where the service call is converted into a general call. This helps to maintain a clean code in the backend microservice components. BFF also makes the backend development independent of the type of new client (mobile/web). This pattern helps to establish a single team responsible for processing and transforming data for multiple API ultimately producing efficient throughput to all the types of clients[18].

Advantages Of BFF

The BFF reaps the following benefits.

- Help to reduce the complexity of the client by acting as an aggregator and doing coordination of requests
- Less complex than a monolithic API
- You will be able to deploy your client applications (web/mobile) faster into the market because you can leverage dedicated teams. Front end teams can get the help of predefined back end teams who cater their API requirements through BFF.
- BFF will provide you better results for each type of client facing application as each client will fetch data from the BFF which is optimized for it.

Disadvantages of BFF Pattern

There are two very distinct disadvantages associated with the BFF pattern implementation.

Fan Out- If architects do not plan ahead carefully, BFF could lead to a high degree of fan-out between the BFF and the related backend services it communicates with. If one of the BFF is terminated, this could lead to total failure of one type of client as shown below.

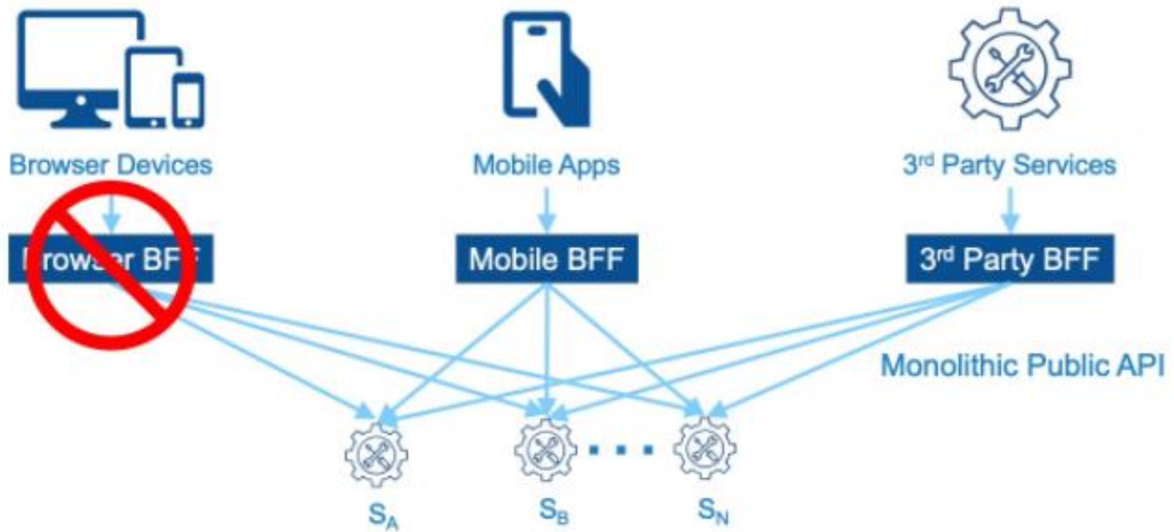

Figure 15 - Fan out, **Source** [18]

- Fuse: Each service, when it communicates with many BFFs would lead to malfunctioning of all BFFs if it's compromised it might halt all operations. Every service is turned into fuse anti-pattern

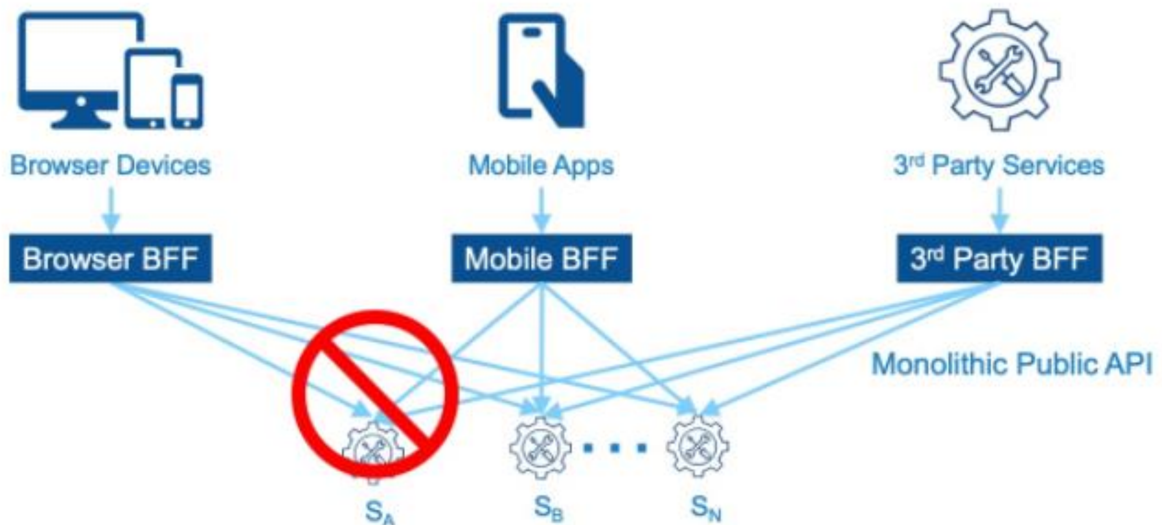

Figure 16 - Fuse, **Source** [18]

- Duplication: BFF would lead you to implement similar code by different teams, this might lead to increment in the cost of development. But teams could identifying which code is duplicated and build shared libraries that get developed once
- Count of Microservices: When the number of types of clients increases, we have to deploy multiple BFFs to accommodate clients. This causes the number of

components that have to be managed by the CICD pipeline and deployed. This overhead could be minimized by practicing and leveraging efficient DevOps practices

When to use BFF

When the data requirements across different types of clients (mobile, browser and other third party) change and the time to deploy a single aggregated API becomes troublesome, BFFs is a go to solution. It's highly recommended to follow the below practices to miss any pitfalls when integrating BFF pattern into your microservice architecture.

Solve BFF Associated Problems

- Solve Fan Out Issue: we should prevent failure of any of the services that BFF coordinates with to make sure the clients are uninterrupted. We should implement fault tolerance. Each type of new client will have its dedicated BFF interface to complete the necessary data transformation. Even Though this increases the number of components to be managed and deployments, we can gain high availability and fault tolerance for all types of clients.

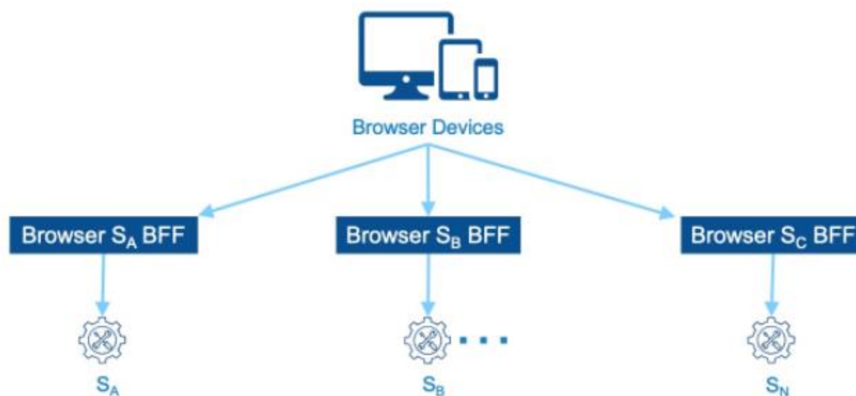

Figure 17 - Solve Fanout, **Source** [19]

- Reuse: If you're not happy with the functionality being overlapped between multiple components, it will be a good idea to educate the teams (engineering, scrum masters and product owners) about the scope and the efforts needed. Creating reusable libraries to be used in BFF is a great engineering principle to lower development costs as well as move new features to market quickly.

	Pattern	Advantages	Disadvantages

Orchestration & Coordination	API Gateway Pattern	• Extension easiness • New feature deployment made easier • Backward compatibility	• Potential bottleneck • Gateway • Duplicate Code • High Component Management overhead
	Service Discovery	• High code maintainabilit y • Fault tolerance	• Service registry complexity • Distributed system complexity
	Fine Grained SOAP	• High Adaptability • Supports multiple protocols • Efficient for medium sized systems	• Development /testing complexity • Implementati on effort • Network-related issue

4.2 Managed State Patterns

The microservice patterns here are responsible for manipulation of state in a microservice enterprise application. State can be considered as one of the most important and critical aspects of a distributed enterprise system. One of the main reasons for that is in traditional system designs they mostly used data queries and CRUD operations on the data though consistency is difficult in a distributed system. It can be concluded that the patterns that deal with state management of microservices are found useful when your number of microservices is growing rapidly. At the first stage of any enterprise application the prime focus of the microservice designs are concentrated on integration of the dependencies and third party services, the hassle of state management arises quickly when the customer base of the application starts increasing. This teaches us one important fact and that is if we are to ensure the quality of data while mitigating corrupted and invalid data we need to consider querying or issuing commands. And it is highly advised to declare state and plan to minimize any side effects resulting by mutating and querying it. By following state management patterns, it's industrially proven that you will be able to gain autonomous, adaptable and highly scalable efficient enterprise applications.

4.2.1 Message Oriented State Patterns

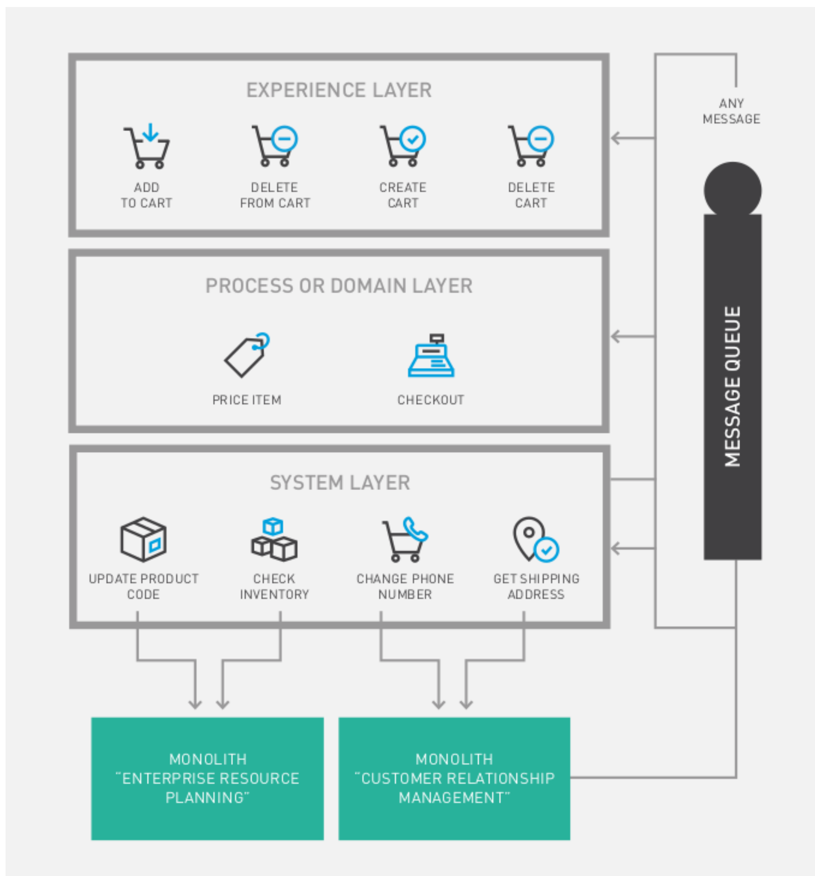

Figure - 18 Message Oriented State Patterns [19]

This is one of the most practiced patterns to minimize any side effect of querying or mutating state. By supplying a queue which is functioning asynchronously, the main method to interact with state changes or to query any other microservices, Every microservice is configured with the time limits necessary to aggregate events and thus generate a consistent external view. This is recommended to be implemented by developers who have good previous experiences in SOA. It's advised by the industrial experts that this pattern should only be used as an intermediate stage. This would be a great choice for an organization who is just starting the footsteps in the microservices path, but it is advised that this pattern would not be the best fit when the enterprise needs to be scaled and the number of components increases.

By configuring components to integrate via a queue, the behavior of each microservice becomes unpredictable, making sure the side effects of the design emerge vividly when a system is exposed to a microservices architecture. Industrial experts regard this microservice pattern to be a transitory step. It is a step where teams would be encouraged to use a queue and convince them about the advantages using queues. Furthermore it's important to note that if an

organization is determined to reap results of this pattern they need to maintain standards about the data that is passed in the queues.

We can make sure to maintain the data integrity without having to replicate the state of business data between databases. Advantages such as asynchronous state which can be sent to different microservices for processing are distinct advantages of integrating a queue in the microservice architecture. A change in data triggers a message over a queue so any other microservices that are listening to these messages through Pub/Sub architecture get notified of the message.

Advantages

Message queues help to decouple the consuming microservices. Those microservices are able to process the messages in the queue independently from message producers, which improves the scalability and reliability, consistency of the enterprise application.

Disadvantages

The interprocess communication loses its efficiency somewhat due to asynchronicity.The consistency of data and state management are compromised due to the unpredictability and the any side effects of a message. Reusability is minimized because predicting the use patterns is difficult. Since the existing microservices patterns are quite similar to each other, problems tend to emerge. Even Though this turns out to work efficiently at high scale it tends to become infeasible. The granularity of the system could diminish if the design of this pattern is poorly carried out. So it’s of vital importance when integrating this pattern in your system.

4.2.2 Event-Driven State Management

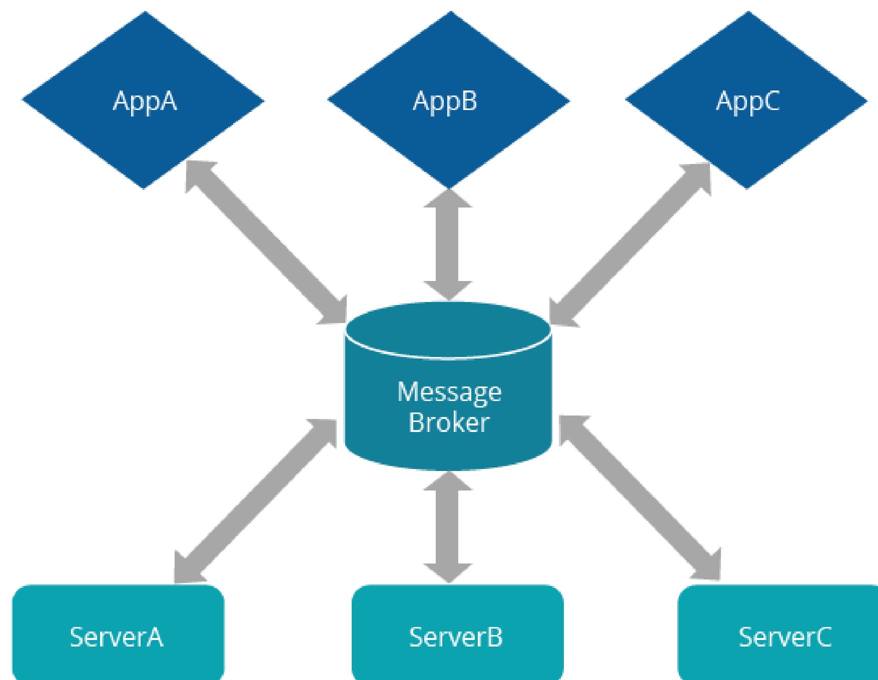

Figure 19 - Event driven state management, **Source** -WSO2 [20]

Event-driven architectures when used together in a microservice based system can lead your system to greater heights and reap multiple advantages like scalability and reliability. Event-driven microservice patterns are traditionally based on a queue. They transform the concept of the design and the message in the queue into the concept of an event[20].

To study deeply about what an event is, An event is any occurence of change of data in a given time. By processing these events any microservice listening to it generates a snapshot of the system on demand. This materialized view could be replayed to view the events in order similar to the initial stage. If you are able to follow and generate a standard, this pattern would work well in microservices architectures[21]. But it's important to notice that the domain of your business application might dictate the usage of this pattern as some of the domains can not be abstracted by a sequence of events.

To ensure data integrity, we should replicate business events to and synchronize them between other microservices. It's important to adapt to use a generic event overview to denote a change in the architecture

When some activity is performed by the client (not necessarily a change of data), an event containing all the information about the activity is sent to all microservices listening to those types of events. These business changes are known as the product of these events which are being generated, sent across and then unlimitaly computed by microservices subscribed[22].

4.2.2.1 Saga Pattern

The Saga pattern can be thought of as a collection of business transactions. When a user starts a business transaction in a SAGA based microservice application, the microservices in the system publishes events to notify other microservices in the system. The microservices which were notified by the initial events process the event and publish new events based on the results to notify many others in the system. And if one of the transactions in this sequence does not complete successfully, the microservices responsible for the transaction emits a new events to notify the whole system. Upon these events every microservice listening will undo the processing that was carried out earlier to minimize so that any adverse impact is minimized[23].

To look deeper at the SAGA microservice pattern furthermore, let's look at the following example. Note that the following events are related to a shopping application. When a user tries to order/purchase any item, the below given events could be emitted:

- *Item Order service* generates the event *order.* All the microservices listening to this event move this to INPROGRESS state. A Saga pattern manages the chain of events.
- The *Item Order Service* tries to update the new entry in the database.
- If the operation is successful, it can accept the order or else deny the order.
- Upon acceptance, it will emit *accepted* event and upon rejection it will emit *rejected* event
- If the order was accepted, it would notify the customer with a success message. Upon rejection, the customer will be notified with the error message.

This approach is very different to that of the other coordination patterns that we have discussed so far[23]. This pattern might cause the complexity of the system as it provides ways which are different to the traditional methods for state manipulation. Since there are no standard patterns that you could apply out of the box, the development team needs to come to a consensus on the standard of system design and implementation to reap the rewards of the Saga pattern.

Advantages

Cohesion is one of the main advantages that could be reaped from following this architecture. The programmers find it convenient to work with and comprehend due to the standardized approach.

Scalability: Since components interact through events they are not dependent on each other, thus components could be scaled independent of the others, and this is a huge advantage for the development team

Speed of change: You will be able to make changes to the application quite easily due to this cohesive architecture.

Disadvantages

Furthermore due to the asynchronous nature of interprocess communication the latency of the business transactions could be higher compared to that of the other microservice patterns. Due to similarities in message generation and the naming conventions, it might create confusion and doubt events and commands in the system. Development and testing complexity of the system is higher due to the asynchronous communication. A skilled development team is a necessity if you’re going ahead with this microservice pattern. Another one of the most important issues is that it's quite difficult to restore the state in a case where any microservice exits non-gracefully.

4.2.3 (Event Sourcing) Replicating State in Layered APIs

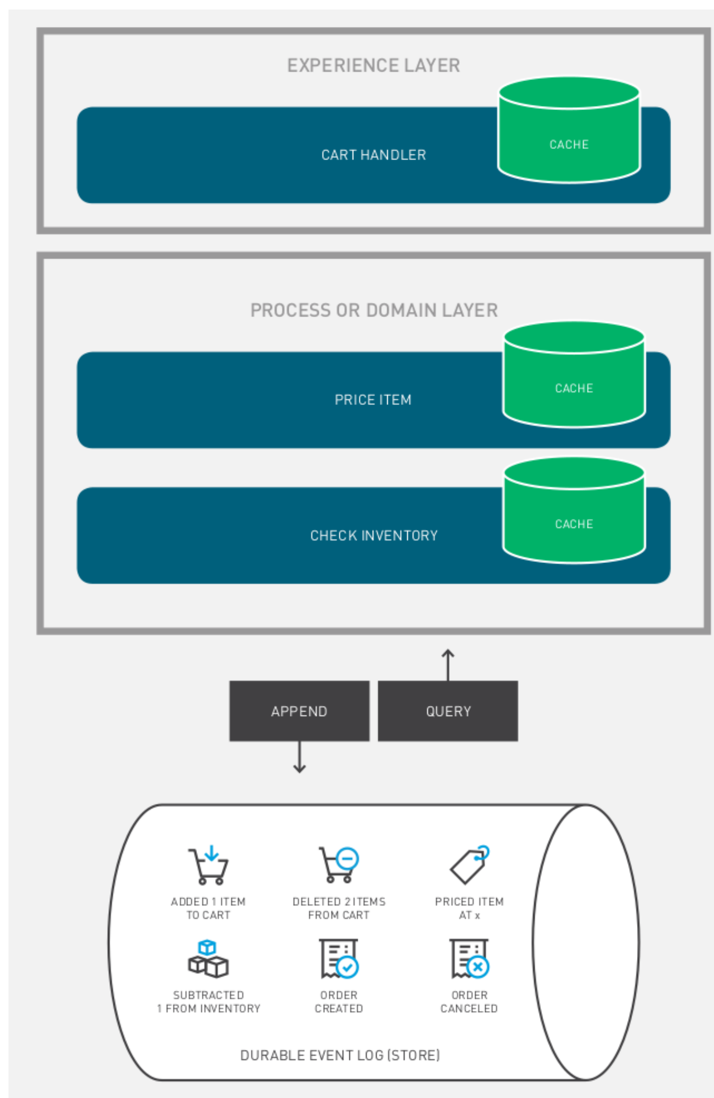

Figure 20 - Event Sourcing [19]

Syncing of state or replaying of events is one of the solutions to the problem that arise from isolating state in enterprise applications where consistency of the application is needed. If we imagine an **order**, **user**, and **authorization** microservice, one common thing among them is that they have a state which is isolated, hence they have to depend on each other to complete a given business transaction. If a failure in a microservice happens, it can affect other microservices to fail thus ending causing the customer query to fail.

The above problem could be resolved by maintaining one place to persist all state mutations that are occurring across all the microservices in the system so that each microservice can generate its own state by consuming the event from this single event store whenever they need. This is the fundamental of event sourcing, where microservices communicate through an event store which leverages them to generate a snapshot of the system by consuming these events at any given time[24].

Event sourcing is a pattern where microservices are focused on maintaining a collection of events persisting the full history of a business transaction as a log of events, instead of persisting the current state only[16]. The microservices have the ability to query state, hence, have made use of this event log. This design makes sure the system is eventually consistent. For example most financial institutes have implemented a system which is more consistent than ensuring every business transaction is consistent. This gives the engineering team a lot of space ultimately speeding up the delivery time since dependencies are minimized[21].

It is quite hard to achieve integrity with data when there are many sources of truth. So the idea here is to maintain a single source of truth for every business transaction occurring in the system, and sync that store with any other microservic as needed. The goal is to send all the changes transformed into events into a permanent log. When it's required to query data, the microservice needs to generate a snapshot of the system by calculating events replayed from the initial event store.

The impacts

This pattern helps to create a cohesive microservice architecture which would be very scalable since the Command-Query Request Separation is included in the design. One other concerning impact would be that It can be a bit hard to understand the overview of the architecture and understand logical dependencies associated with the microservices.

Advantages

One of the main advantages that is gained by adhering to this pattern of microservice architecture is that it promotes a very scalable enterprise application. Furthermore the modifiability and the extensibility of the system is high. The engineering team would find this architecture pattern easy to work with.

Disadvantages

The inter process communication lacks a bit of efficiency due to asynchronicity. Another major concern is that the pattern is not able to be used with every enterprise application since the domain of the application dictates whether it could be abstracted into events. A skilled development team is needed to take up the development work due to the complexities in development.

	Pattern	Advantages	Disadvantages
State Management	Message Oriented State Management	• Autonomous microservices. • High reliability.	• Inefficient IPC due to asynchronicity • Not highly scalable • Development complexity • Testing complexity
	Event-Driven State Management	• Independent components • High scalability • High Maintainability • Failure Safety	• Distributed system complexity • Development complexity • Testing complexity
	(Event Sourcing) Replicating State in Layered APIs	• Increased maintainability	• Development /testing complexity

		• Flexibility • Physical isolation	• High Implementation effort • Inefficient IPC communication due to asynchronicity

4.3 Deployment Patterns

The following Patterns address the widely used practices used when microservices are required to be deployed in a production environment in a cloud infrastructure. Deploying an application which is monolithic, might require you to configure many instances. You normally use many servers which are either physical or virtual. And then you run multiple instances of the application on each one. It can be accepted that the deployment of a traditional application is not trivial. The complications of deploying a microservice-based application is even higher.

An enterprise application consists of many microservices. Microservices could be written by many server side languages. Each microservice should have its deployment pipeline, scaling, and other monitoring requirements taken care of. You might need many instances of each service so every client is successfully served. The resource requirements for microservices are different to each other depending on the nature of the tasks they perform. .In order to cater these requirements, there are a few different microservice deployment patterns that can be made use of[9].

4.3.1 The Multiple Services per Host Pattern.

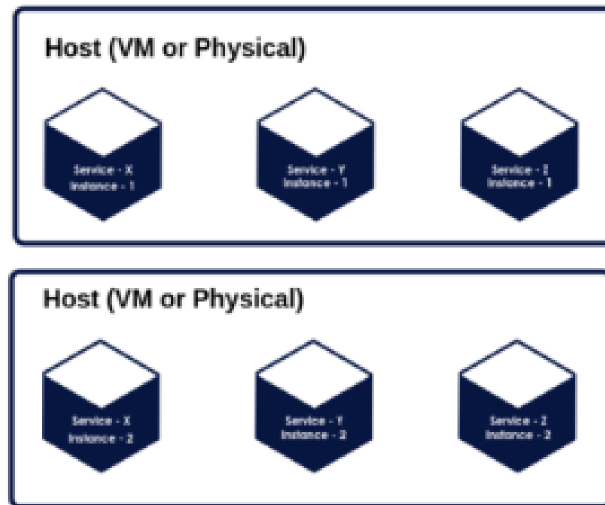

Figure -21 Multiple Services Per Host Pattern, **Source** - [27]

Deploying several instances in one host is one way of running your microservices in a production environment. When you're using this particular deployment pattern, you spin up virtual hosts in your cloud environment and run multiple microservice instances. But this is an outdated approach in service deployment. Each service instance is configured to run on a predefined port on hosts[28].

4.3.2 Single Service per Host Pattern

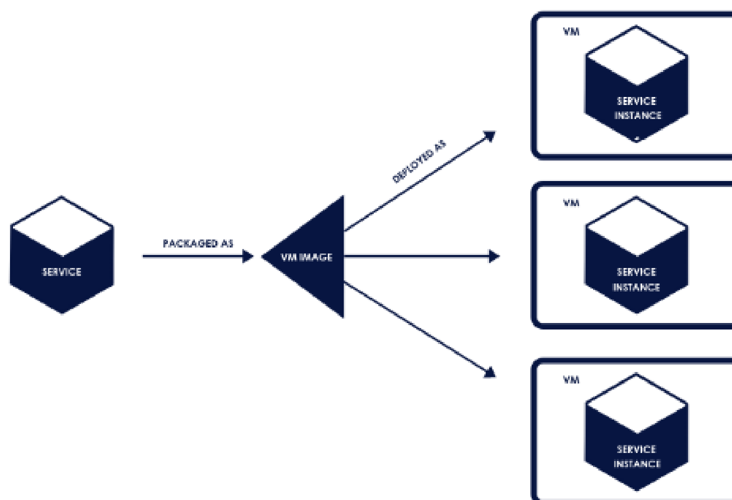

Figure - 22 Single Service Per Host Pattern, **Source** - [27]

One of the other ways of deploying microservices is to deploy one service instance in its own host. This allows you to run each instance separately on its host. Isolation of services is one benefit that can be gained from following this pattern. This helps in gaining the maximum performance for a given service. This pattern could further be divided into two sub patterns which are[28],

1. One service instance in one virtual Machine
2. One service instance per container

4.3.2.1 Service Instance per Virtual Machine Pattern

If you prepare for the deployment following the pattern, one service instance per virtual machine, your first step would be to build an image out of your source code of the microservice using a technology like *Docker*[28]. Each microservice instance is basically a virtual machine that is launched using that built image. They do packaging of its microservices into containers before they are deployed in any host provided by the cloud providers.

4.3.2.2 Service Instance per Container Pattern

In this pattern, microservices are deployed in its own container. Containers are basically a virtualization at the operating system level layer with the help of a hypervisor. A container generally consists of multiple processes running in it. For the process running, it has its own port and the filesystem as if it were running in a host. The resources needed for your processes could be easily configured and managed. For example the resources like container's memory and CPU are at your leverage. Container technologies like Docker and Solaris Zones give you more options like I/O rate limiting to optimize the operations of the processes in your container. Microservice giants like Amazon, Uber, Spotify and Netflix follow this pattern to cater millions of their users.

To follow his pattern, the first step is to package your microservice code base into a container image. A container image is basically a collection of applications and libraries required to run the service. If you're to deploy a microservice written in Node Js, you should have a image created from the base image of Node Js.

After you have built a container image out of your microservice using any of the containerization technologies, the next step is to deploy your microservice as a standalone

service or with multiple containers. In the modern enterprise application development generally you comply to run multiple containers on one instance purchased from your cloud provider. In this case it would be timesaving to integrate container orchestrators such as Kubernetes to orchestrate the microservice containers. A container orchestrator manages all the containers in all the hosts. It could be configured to make decisions on which container should be running in which host of the cluster depending on the resources or requirements of the deployed service and the remaining resources in the cluster[28].

4.3.3 Serverless Deployment

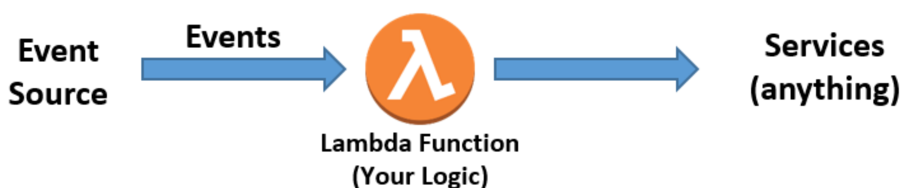

Figure -23 Serverless Architecture Overview, Source - AWS [30]

Almost all Cloud providers provide Serverless deployment options currently. AWS Lambda, GCP cloud functions[31], Azure functions are just a few of the options examples of serverless deployment technologies[34]. They support almost all the languages available. In order to start bootstrapping a serverless microservice, the first step is to package the source code as a compressed file and place it in the serverless environment. You can configure other details like the name of the function and other environmental variables required when the function is triggered by a user. Severless platforms make sure to trigger automatically enough invocations of your microservice to process requests. Most cloud providers like AWS, Azure and GCP have pricing plans so that you are going to be charged for each invocation. The resources consumed by the invocation and the running time affect pricing. The best advantage of adapting to the serverless pattern is that you as a developer or as an administrator in the organization do not have to undergo the burden of managing the servers or containers while the cost of infrastructure is very low.

A Lambda function could be abstracted as a stateless microservice. It processes requests by integrating with other different types of services in the cloud computing ecosystem. As an example, a *lambda* function could be configured (offered by AWS) to be triggered when a video is placed in an object store(storage solutions in AWS) which has the logic to insert the video into a persistence store and start analyzing the video[30].

serverless functions could be invoked irrespective of the cloud provider by following methods:

1. By a web service(REST) call
2. Triggered by other services offered in the cloud computing ecosystem
3. Configure rules as to when and where the function should be invoked

Serverless frameworks offered by all the cloud providers are found to be a highly efficient way to unleash your microservices with low development to market timeline. Serverless would be a great choice in scenarios where your microservice is performing short lived tasks such as CRUD operations, data shimming etc. In this case the BFF pattern that we discussed earlier could be benefited most by the serverless framework. For long running calculations and machine learning tasks this would not be advantageous. The per invocation pricing model ensures that you have to pay for the work your microservice performs only. No charge if there was no invocation at all. Another great relief would be that since the managing work and monitoring cost is low the engineering team can solely focus on the development work.

	Pattern	Advantages	Disadvantages
Deployment Pattern	The Multiple Services per Host Pattern	Traditional Approach Easy service-discovery	Low scalability Low security
	Single Service per host	Isolation of services High scalability High security	High cost due to infrastructure Low security in containerization

	Serverless	Low cost High scalability Low time from development to market	Development complexity Not suitable for long running tasks/ database connection oriented tasks

4.4. Data Storage Patterns

Data persistence and management in an enterprise application system can get complicated and troublesome. The importance of this is even higher in a microservice based architecture. Every microservice in the system depends on querying or mutating data. The efficiency of processing client requests could be improved by efficient data management thus reducing the latency to fetch data from the database. The prime intention is to make data exposed to every microservice within an optimal time. Any increase in the latency of the database queries would derail the performance of the whole system[10].

4.4.1 The Database-per-Service Pattern.

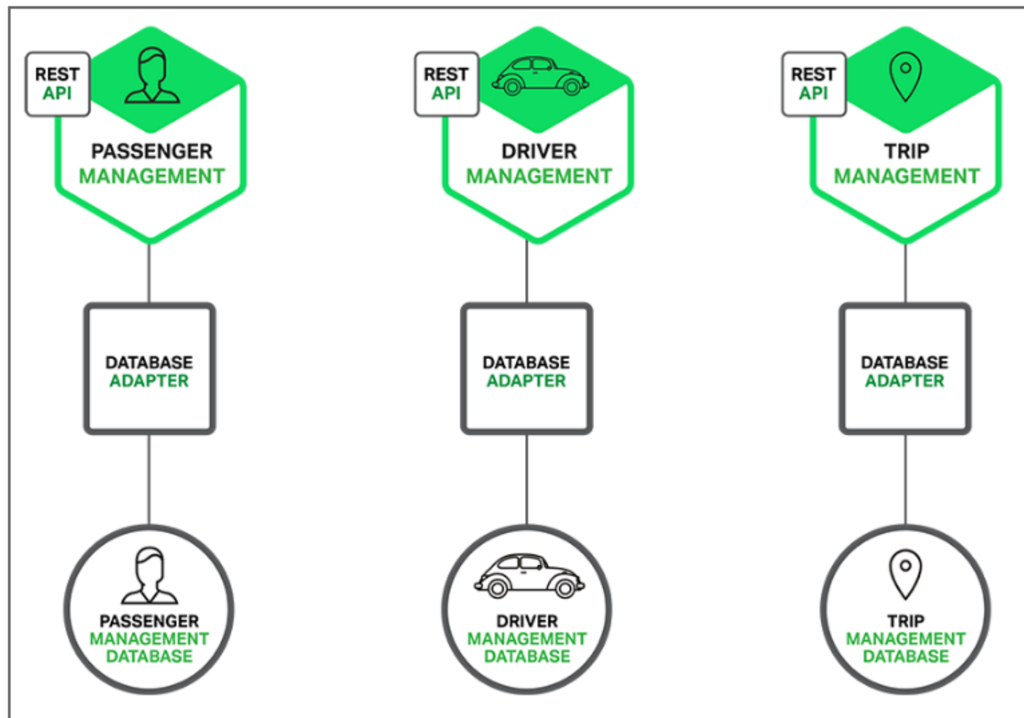

Figure - 24 Database per service pattern, **Source** - Nginx [33]

This pattern focuses on the concept where each microservice maintains a private database. This could be considered as one of the trivial methods to start your microservice journey. It's highly used when you are planning to migrate existing monoliths to a microservice based approach. Every microservice is responsible for managing the data for which it's cohesive. If any other microservice needs to query the data which is managed by another microservice, it should fetch the data through a REST Api but not access the database directly.

Even Though it seems easier to have one database per microservice it gets tricker when this pattern is implemented. One of the major reasons for this is since microservices are not modularized and separated by concern when they are bootstrapped. Since every microservice depends on another microservice to complete any given business transaction, the interactions might depend on a few microservices making every component dependent on each other which reduces scalability of the system. If the architects in the tem could identify the boundary context of all the business transactions performed by the system, you're guaranteed to gain the advantages from adhering to this pattern. You can make sure that all the microservices in the same boundary context have one data store there by reducing the dependencies. If you're bootstrapping a new microservice based system the task would be somewhat trivial but if it's an already existing system the task would sound anormous.

One other problem that might arise when working with this pattern with handson is when you come across business transactions that need interaction of multiple microservices. Another

scenario is when a client query needs data spanning across multiple boundary contexts. In these situations it's important to think and plan for the future so that dependency is minimized and scalability of the system is preserved. One of the most prominent advantages of this pattern is the high scalability of the system where the database is not going to restrict the latency of the customer queries. This pattern grants great independence for the engineering team to develop microservice freely and since the dependencies are minimal[10].

4.4.2 The Database Cluster Pattern.

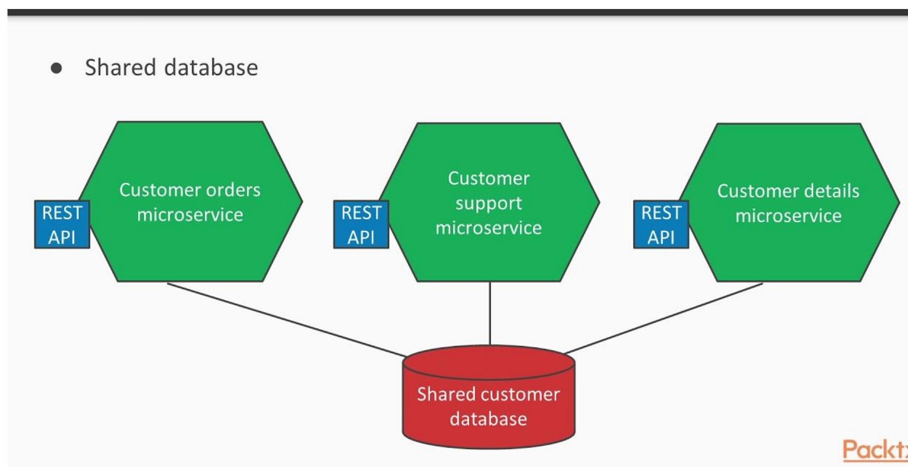

Figure -25 Shared Database Pattern, **Source** - [10]

The second storage pattern focuses on storing data on a database cluster for the entire enterprise application. Experts claim that such a system makes it easier to maintain data in dedicated hardware. But since most of the organizations today make use of cloud providers, this really is no longer an issue. To ensure the consistency of data, microservices are assigned a set of tables. These can only be accessed directly by the owning microservices. In the other scenarios, each microservice would be assigned with a database definition. Each time, the microservice sees the database as if it's owning it. Generally, this is regarded to be a safe choice for architects as this enables them to design and implement microservices in more familiar ways. As your enterprise application attracts more traffic this pattern would hinder the scalability[28].

	Pattern	Advantages	Disadvantages

Data Storage Pattern	Database Per Service Pattern	• High scalability • Low dependency between microservices • High modifiability	• Defining boundary contexts is critical for success • Bad planning might force to have lots of interactions between microservic es, increasing latency • High resource cost
	Database Cluster Pattern	• Easy to design and implement • Low resource cost	Less scalability as the system is grows

5. Conclusion

In the 3rd decade of the 21st century enterprise software is of utmost importance than it has

ever been. It is critical to lay a good rock-solid foundation when you're starting to build your house, the same way enterprise software architecture dictates the good, the bad and the ugly for your software application. When the architecture is designed well, the development of the business features by the engineering team and scalability of the application for new customers across the globe becomes ever so simple. So the role of the architects is a deciding factor inorder to make sure the goals of the system owners and the users of the system are fulfilled.

Architectural patterns in software are temporary. They change with time as anything else does. When new technologies are invented, to cater new customers, new design conventions are treated as patterns. For some time now, microservices has been a trending topic due its numerous advantages over the traditional monolithic applications. And there's not a single microservice pattern that you can implement out of the box to gain all the advantages of a microservices approach. It is a combination of many microservice patterns that are mentioned above that will yield you an efficient system. Most architects regard systems built with microservices to be a prescriptive architecture. But this research confirms that it's a descriptive architecture indeed. Based on the above evaluation of microservice architectures, this research also yields some of the factors dictating certain microservice architecture patterns over another in a given project setup.

1. Size of the system
2. Development complexity
3. Domain
4. Types of users (Mobile/Desktop/ Third party)
5. Time to market
6. Scalability
7. Experience of the development team

In order to transfer this knowledge base of microservice architecture patterns and to help software engineers in architecting enterprise applications, I give you the ArchitectureProposer. It's a web application which gives you the most matching microservice architecture patterns depending on the properties of your project. The user input could be an XML as well as filling up a form.

Future Work

I believe there is huge opportunity and potential for this research to be extended and kept in sync with the latest breakthroughs in the software industry as it constantly changes. And also the ArchitectureProposer could be further improved to give more insightful recommendations to software engineers starting fresh at architecting enterprise applications based on microservices. Now this web application uses a rule based approach, further there is space here to integrate a machine learning based approach to improve the results based on the user inputs.

REFERENCES

1. martinfowler.com. 2021. *Microservices*. [online] Available at: <https://martinfowler.com/articles/microservices.html> [Accessed 27 January 2021].

2. Newman, S., n.d. Building microservices.

3. Richardson, C., n.d. Microservices Patterns.

4. Hulya VuralHulya, VuralMurat, KoyuncuMurat KoyuncuSinem. “A Systematic Literature Review on Microservices” , International Conference on Computational Science and Its Applications (2018).

5. P. D. Francesco, I. Malavolta and P. Lago, "Research on Architecting Microservices: Trends,Focus, and Potential for Industrial Adoption," 2017 IEEE International Conference on Software Architecture (ICSA), Gothenburg, 2017, pp.

6. PMario Villamizar, Oscar Garcés and Harold Castro, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud" 2015 10th Computing Colombian Conference (10CCC)At: Bogotá, Colombia

7. Javad Ghofrani, Daniel Lübke, “Challenges of Microservices Architecture:A Survey on the State of the Practice” 10.1109/SOSE.2016.22.

8. Taibi, D. et al. “Architectural Patterns for Microservices: A Systematic Mapping Study.” CLOSER (2018).

9. 1-30, doi: 10.1109/ICSA.2017.24.4. D. Guo, W. Wang, G. Zeng and Z. Wei, "Microservices Architecture Based Cloudware Deployment Platform for Service Computing," 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE), Oxford, 2016, pp. 358-363, doi:

10. dzone.com. 2021. 6 Data Management Patterns for Microservices - DZone Microservices. [online] Available at: <https://dzone.com/articles/6-data-management-patterns-for-microservices-1> [Accessed 7 March 2021].

11. microservices.io. 2021. Microservices Pattern: Client-side service discovery pattern. [online] Available at: <https://microservices.io/patterns/client-side-discovery.html> [Accessed 27 January 2021].

12. microservices.io. 2021. Microservices Pattern: Server-side service discovery pattern. [online] Available at: <https://microservices.io/patterns/server-side-discovery.html> [Accessed 27 January 2021].

13. GitHub. 2022. *GitHub - dilekamadushan/research-architecture-patterns*. [online] Available at: <https://github.com/dilekamadushan/research-architecture-patterns> [Accessed 20 February 2022].

14. Subscription.packtpub.com. 2021. {{metadataController.pageTitle}}. [online] Available at: <https://subscription.packtpub.com/book/application_development/9781789133608/1/ch01lvl1sec12/service-oriented-architecture-soa> [Accessed 24 January 2021].

15. GitHub. 2022. *GitHub - Netflix/eureka: AWS Service registry for resilient mid-tier load balancing and failover.*. [online] Available at: <https://github.com/Netflix/eureka> [Accessed 5 March 2022].

16. Medium. 2021. Microservices Layered Architecture. [online] Available at: <https://medium.com/microservices-in-practice/microservices-layered-architecture-88a7fc38d3f1> [Accessed 27 January 2021].

17. Nordic APIs. 2021. Building a Backend for Frontend (BFF) For Your Microservices | Nordic APIs |. [online] Available at: <https://nordicapis.com/building-a-backend-for-frontend-shim-for-your-microservices/> [Accessed 24 January 2021].

18. AKF Partners. 2022. *Backend for Frontend (BFF) Pattern: The Dos and Don'ts of the BFF Pattern*. [online] Available at: <https://akfpartners.com/growth-blog/backend-for-frontend> [Accessed 5 March 2022].

19. Mulesoft, 2018. THE TOP SIX MICROSERVICES PATTERNS. [online] Available at: <https://www.mulesoft.com/ty/wp/top-microservices-patterns> [Accessed 21 January 2021]

20. Wso2.com. 2021. WSO2. [online] Available at: <https://wso2.com/whitepapers/event-driven-architecture-the-path-to-increased-agility-and-high-expandability/> [Accessed 26 January 2021].

21. microservices.io. 2021. Microservices Pattern: Event-driven architecture. [online] Available at: <https://microservices.io/patterns/data/event-driven-architecture.html> [Accessed 7 March 2021].

22. Ingeno, J., 2018. Software Architect's Handbook.: Packt Publishing.

23. microservices.io. 2021. *Microservices Pattern: Sagas*. [online] Available at: <https://microservices.io/patterns/data/saga.html> [Accessed 26 January 2021].

24. martinfowler.com. 2021. Event Sourcing. [online] Available at: <https://martinfowler.com/eaaDev/EventSourcing.html> [Accessed 29 April 2021].

25. Mihai Baboi, Adrian Iftene, Daniela Gîfu. "Dynamic Microservices to Create Scalable and Fault Tolerance Architecture" , 23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems.

26. Engineering & Technology, 2008. Time to market [event-driven architecture]. 3(4), pp.56-59.

27. dzone.com. 2021. Right Strategies for Microservices Deployment - DZone Microservices. [online] Available at: <https://dzone.com/articles/right-strategies-for-microservices-deployment> [Accessed 26 January 2021].

28. Venugopal, M. V. L. N. (2017). Containerized Microservices architecture. International Journal of Engineering and Computer Science, 6(11), 1. https://doi.org/10.18535/ijecs/v6i11.20

29. Fetzer, C. (2016). Building Critical Applications Using Microservices. IEEE Security & Privacy, 14(6), 86–89. https://doi.org/10.1109/msp.2016.129

30. Serverless Architectures with AWS Lambda. [online] Available at: <https://aws.amazon.com/whitepapers/latest/serverless-architectures-lambda/aws-lambdathe-basics.html> [Accessed 26 January 2021].

31. W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," 2018 IEEE International Conference on Cloud Engineering (IC2E), Orlando, FL, 2018, pp. 159-169, doi:10.1109/IC2E.2018.00039.

32. Patond, S. B., Satpute, S. R., & Prof.D.D.Sapkal, S. S. P. |. A. S. K. |. (2018). Microservice Oriented Application Development. International Journal of Trend in Scientific Research and Development, Volume-2(Issue-4), 1130–1135. https://doi.org/10.31142/ijtsrd14318

33. NGINX. 2021. Introduction to Microservices | NGINX. [online] Available at: <https://www.nginx.com/blog/introduction-to-microservices/> [Accessed 26 January 2021].

34. Familiar, B. (2015). Microservice architecture. Microservices, IoT, and Azure, 21-31. doi:10.1007/978-1-4842-1275-2_3

35. Familiar, B. (2015). Microservice reference implementation. Microservices, IoT, and Azure, 109-131. doi:10.1007/978-1-4842-1275-2_6

36. O'Reilly Online Learning. 2021. *Software Architecture Patterns*. [online] Available at: <https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html> [Accessed 24 January 2021].

37. Bankar, S., 2018. Cloud Computing Using Amazon Web Services AWS. International Journal of Trend in Scientific Research and Development, Volume-2(Issue-4), pp.2156-2157.