University of Bamberg

Distributed Systems Group

Seminar paper

As part of the Master seminar

Modeling Service-Oriented and Cloud-native Software Architectures

With the subject:

Modeling a microTOSCA based application and microservices design patterns

Submitted by:

Shashidhar Reddy Nimmagari

Supervisor: Prof. Dr. Guido Wirtz

Bamberg, Winterterm 2021/2022

Contents

List of Figures

List of Tables

Listings

Abbreviations

API	Application Programming Interface
UI	User Interface
SOA	Software Oriented Architecture

Abstract
Microservices gained a lot of momentum in recent past years. The microservice-based architecture helps an application to be structured as a group of loosely coupled, independent collaborative services. In this paper, we will define and design a microTOSCA model to depict the architecture of an e-commerce application using the OASIS standard TOSCA. We are also going to design a monolithic microservices architecture for the e-commerce application. Then a technique is proposed to mine the architecture using Kubernetes application deployment. Furthermore, We are going to explore microservices patterns that are available and which of them are present in the e-commerce application that We are modeling using microTOSCA.

1 Introduction

With the evolution of technologies, microservices have become very prominent these days. The advantages of the Microservices outweigh the disadvantages as they are being adopted hugely by some of the very big players in the market like Amazon, Apple, Netflix, and Uber. The microservice-based architecture has become a pattern style for e-commerce applications. The architecture is made up of a collection of small services, each of which runs in its own process and communicates through an HTTP resource Application Programming Interface (API). These services would be administered centrally, and they would be built in a variety of computer languages.

microTOSCA is a repository of the custom TOSCA types which is used for analyzing, modelling and refactoring the service oriented architectures[10]. This paper starts by dealing with the background work related to microservices, microTOSCA, Kubernetes and microservices patterns, and TOSCA. Then we are going to model the architecture of an e-commerce application using the monolithic architecture as well as microTOSCA topology. Then, a technique is presented to mine the architecture modeled by the microTOSCA, which will be done by using Kubernetes deployment and also by using the tools of microTOSCA like microMiner. Afterward, We are going to explore microservices patterns present in the exemplary application and also talk about which microservice patterns are present in the e-commerce application that we are modeling using microTOSCA.

2 Background

2.1 Microservices:

Microservices are a type of architectural solution that can be used to design complicated web-based applications. Microservices have grown in popularity as a natural progression from Software Oriented Architecture (SOA), a method for overcoming the drawbacks of old monolithic systems. They break a large software project in to small, independent parts, which in turn communicates through Application Programming Interface (API).

Advantages:

- Scaling is the most important aspect when considering a cloud application as it directly impacts the cost of the ongoing project. Since the modules are individual, It is very easy to scale in and scale out the resources.
- Small and fast deployments: It has small codebases and scopes, which leads to quick deployments which can lead to the exploration of continuous deployments.
- Easy to understand: Developers can understand it easily as the functionalities of the services are very simple.
- Improved fault isolation: If a single module fails, it does not affect the whole large application. That single module failure can be dealt with individually.

Disadvantages:

- Complexity in Interface control: Each microservice will be interacting with its own API. We can make any changes to that microservice. But in case we are changing the API, the application which is using that API might get affected. So in these situations, It is quite complex to control the interfaces.
- Debugging: The large application will be consisting of many microservices. Each individual microservice will have its own logs. If there is any problem or error, It will be really difficult to find the source of the problem.
- Upfront costs: Hiring a skilled developer who understands the technology requires a lot of budgets. The companies also need to invest in the infrastructure as well

2.2 Architectures:

2.2.1 Monolithic Architecture:

A monolithic application consists a Database, user interface for client side, and server side. All the services and User Interface (UI) are packaged inside a single WAR. Many startup companies choose the monolithic style because it is very easy and comfortable to work with small teams. Although it has some advantages, the disadvantages outweigh them because it is quite tough to get adopted with the new technologies. Scaling is also a big issue in monolithic-based applications. If any single module or a function goes down, the whole application goes down.

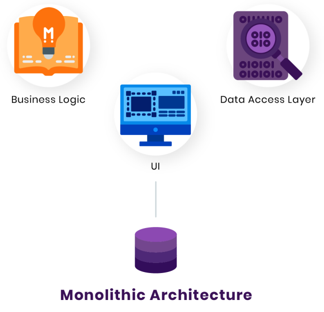

Figure 1: Monolithic architecture[1]

2.2.2 Service Oriented Architecture(SOA):

SOA provides a way by which the components of the software can be used again and in an inter-operable way via service interfaces. The features of the application will be broken down into several components, which are known as services. These services are very flexible, highly reliable, and can be scaled according to the increasing and decreasing complexity. This architecture is suitable for designing complex systems such as banks, while monolithic architecture does not suit the banking system that much.

2.2.3 Microservice based architecture:

Microservices break the application into smaller parts which reduces duplication and makes it easy to understand. They can be deployed and operated individually, so it is easy to scale each service independently, unlike SOA. Services will communicate only with the APIs while hiding their implementation from the other services, and each service will be having its own datastore. Microservice architecture focuses mainly on business priority, whereas the monolithic layer focuses mainly on technology layers and databases. In SOA, data is shared between the services, while in the microservices, Each service can have its own storage.

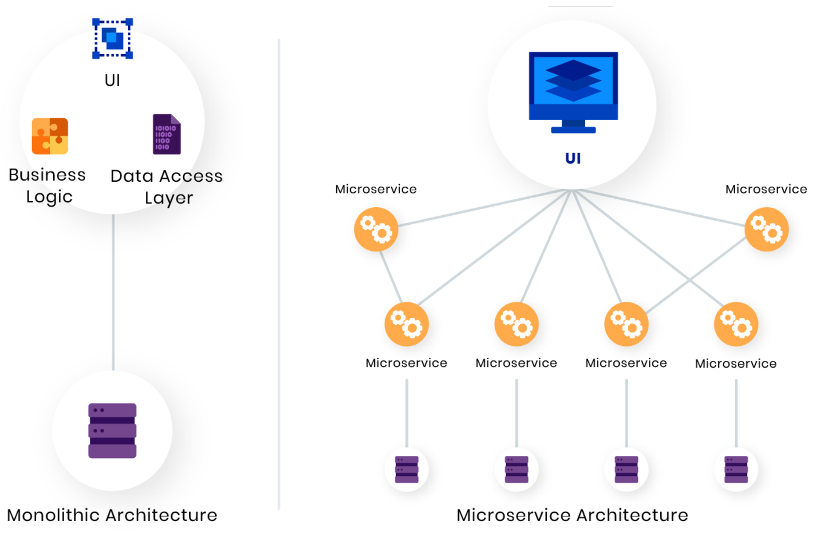

Figure 2: Monolithic vs Microservice architecture[1]

2.3 Microservices Patterns:

A pattern is a solution that can be reused to a problem that may occur in a particular context.[9] This pattern concept was created by Christopher Alexander, who also created pattern language. We can take an example to understand what actually these design patterns are. For example, in a company, there are three teams developing three different projects with three different processes. One of the projects may give fewer errors compared to the others. By using the design patterns, teams working on different projects can use the same pattern, which can be useful to build similar kinds of applications. Another example can be Amazon, the design pattern followed by amazon may not be the best pattern for other companies like Zalando because of the scaling the latter company needs. It can be comparatively less or even more.

There are different types of relationships used in the pattern, and they are predecessor, Successor, Alternative, Generalization, and Specialization. Apart from the relationships, there are also different types of patterns available. They are shown in the figure 3. Some description is given about some of the Patterns that are available.

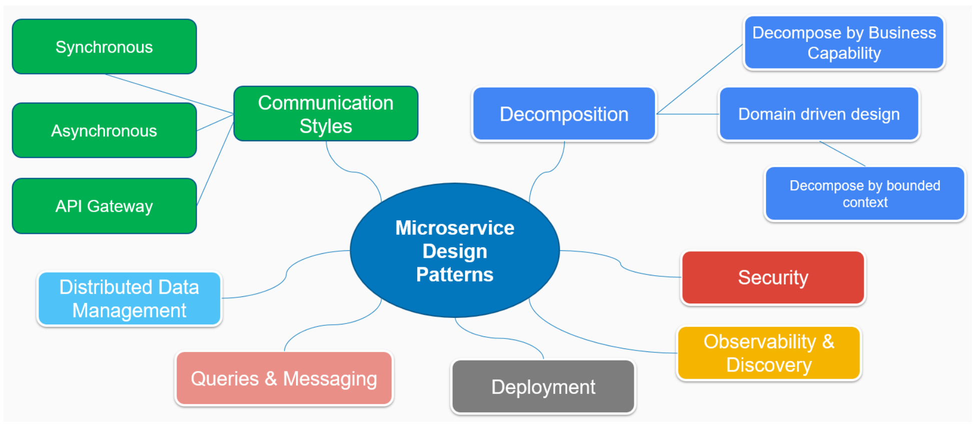
Synchronous
Asynchronous
API Gateway
Communication Styles
Decomposition
Decompose by Business Capability
Domain driven design
Decompose by bounded context
Microservice Design Patterns
Distributed Data Management
Security
Observability & Discovery
Queries & Messaging
Deployment

Figure 3: Patterns[2]

- **Decomposition Patterns:** To decide the decomposition of a system into services is a difficult task. To decompose an application into services, first, the architecture of the application must be stable. For this, we have two kinds of decomposition patterns. Decompose by Business capability, which is responsible for organizing the services based on the business capabilities, and Decompose by sub-domain, which is responsible for organizing the services around domain-driven design sub-domains. There might be some issues in these two patterns which include, how to identify those business capabilities and sub-domains because it requires a good understanding of the businesses.

- **Communication patterns:** while developing a microservice-based application, there will be several services that will be communicating with each other. So, making the design and architectural decisions need a lot of attention. These communication patterns are divided into five groups: Communication style, Discovery, Reliability, Transactional messaging, and External API. All these are responsible for different purposes like which IPC mechanism to select, ensuring the reliability of the communication etc,

- **Data consistency patterns:** These patterns are responsible for implementing the management of transactions. As we know earlier, microservice-based applications can have a different database for each service, and this might be a problem in some cases to achieve consistency. So, using a traditional method for a modern application is not a suitable option for us.

- **Patterns for querying the data:** As described above, when we have multiple services using a different kinds of databases. In some situations, queries of one service needed to be joined with other services. As the services can only interact with the APIs, It is quite challenging to achieve this task. One way of solving this problem can be using the saga pattern through which we can achieve the consistency. If required, we can also use API composition pattern which can interact with other APIs.

- **Service Deployment Patterns:** Deployment of an application is an integral part when developing any kind of application. It may be monolithic-based or microservice-based architecture. It is quite straightforward to deploy monolithic when compared to microservice-based applications because, in monolithic, a load balancer can be used to run multiple instances of the application, but in the latter, there are numerous services running independently. So we need a deployment infrastructure that is highly automated. The ideal way of deploying the microservice application is using another deployment platform. There are two ways of deploying these services. The first approach uses VM's and containers and deploys them. The other can be using a serverless approach.

- **Patterns for Automated testing of services:** Unlike monolithic architecture, the microservice architecture makes it easier to test the services because they are individual and smaller in size when compared to the whole application altogether. There are three different kinds of patterns to test these services. They are Consumer-driven contract tests, Consumer-side contract tests, and Service component tests. These tests are responsible for verifying the client of a service communicates and meets the expectations of the services and also tests the service individually.

- **Patterns for handling cross-cutting concerns:** Observability patterns and discovery patterns are the main concerns every service must implement. It takes so much time to reimplement the configuration parameters when creating the new service. One option is applying the Microservice Chassis pattern, and the other is Externalized Configuration. Microservice Chassis pattern is responsible for building the services on the same framework which handles the concerns.

- **Security Patterns:** All the services interact with the API, So users get authenticated by the API gateway, which will later pass all the information to the services. It also passes the access tokens(JSON Web Token), which is responsible for getting all the information about the users.

- **Observability Patterns:** In a monolithic application, It is quite difficult to troubleshoot as everything lies in one single program. It is very important to understand the runtime along with the troubleshooting problems. We can look into the log file to understand how the requests are handled. It is even more difficult to diagnose problems when it comes to microservices architecture applications. It is because when a particular problem occurs, there will be no log file available to have a look at how the requests are handled. As there are many requests arising from multiple services, those requests can bounce between other services and then reach the client. There are different patterns available to design the services related to observability services they are:

  Health check API can return the health of that service. Log aggregation helps in maintaining the logging activity. Distributed tracing is responsible for assigning unique IDs for tracking the requests. Exception tracking is responsible for tracking the exceptions, Application metrics, which is responsible for maintaining the metrics, and the final one, Audit logging, which is responsible for keeping track of user actions.

2.4 Kubernetes:

Kubernetes is responsible for managing, scaling, deploying, and automating the applications which are containerized. It is an open-source platform that has many benefits like resource efficiency, scaling, and development speed. Kubernetes modules, including the creating of a Kubernetes cluster to update the app, are shown in figure 4

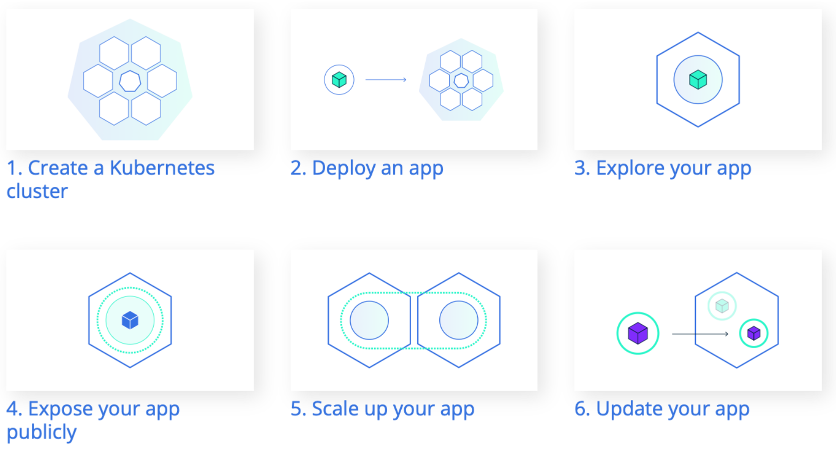

Figure 4: Kubernetes modules[3]

In the first module, a Kubernetes cluster is created using a Minikube. Minikube is responsible for creating the virtual machine on our local machine. Later it deploys that cluster. A detailed description of the Kubernetes cluster is given in the next paragraph. Once the creation and running of the Kubernetes cluster are done, we can deploy the applications on it. The applications should be containerized for deployment. In the third module, the app can be explored for knowing about pods and nodes and how to troubleshoot the applications. Later the app will be exposed publicly, and based on the traffic, scaling up or down is done as the users will be expecting the application to give new rolling updates. This can be done using kubectl.

Kubernetes cluster consists of Nodes, Nodes process, Master, and kubelet. Master is the one that is responsible for managing the cluster and also scaling rollouts etc... A node is considered as a virtual machine or physical computer. Kubelet is an agent responsible for managing the Node and also communicates with the master. Kubernetes cluster also has Deployment, App, Pods, Services and Labels, and Selectors. Deployment is nothing but what to create and what to update. After the deployment is created, the master schedules apps on the nodes. Code for creating deployments via kubect1 (k8s CLI):

$ kubectl create deployment DEPLOYMENT-NAME–image=ADDRESS.

An app has to be containerized. The number of ideal replicas for an app is 3. Pods are the smallest deployable units of the Kubernetes; They provide an abstraction for a group of containerized applications by providing a unique IP address that cannot be accessed outside of that cluster. Services provide an abstraction for a set of pods with a unique IP address. Those sets of pods are determined by a Label selector, which filters a set of objects.

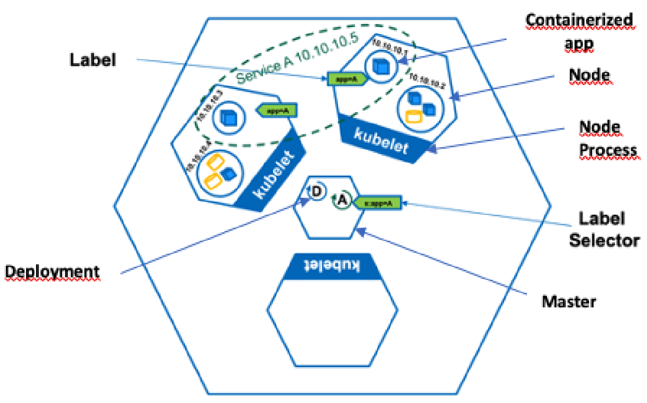

Figure 5: Kubernetes Cluster[3]

2.5 TOSCA:

It stands for "Topology And Orchestration Specification For Cloud Applications". There are 2 main goals for TOSCA standard, they are:

- It provides a specification of the cloud applications which are portable. It provides this specification using a YAML file and a language which can also be read by the machines.
- Automation and management of the cloud applications are specifies using the TOSCA standard.

First, the cloud application will be specified as a service template that is composed of a topology template. A topology template is nothing but a topology graph where the components of an application are represented as nodes, and the relations among those components(nodes) are represented as edges. The node types, relationship types, and group types are defined for extending or inheriting and specializing their structure.

2.6 MicroTOSCA:

As mentioned earlier in the introduction, microTOSCA is a repository of the custom TOSCA types which is used for analyzing, modelling, and refactoring the service-oriented architectures.[10]

It is basically a type-system that enables the representation of microservice-based architectures as typed topology graphs in TOSCA. As there will be different integration components in an application like API gateways, load balancers, and message queues, etc., they will be modeled by the nodes. The runtime interactions are represented by the arcs. By using all these representations, We can model architecture for the e-commerce application.

2.7 microTOSCA toolchain:

This toolchain is madeup of microMINER and microFRESHNER which are integrated by microTOSCA. The microTOSCA toolchain helps to analyze and design the microservice-based applications.

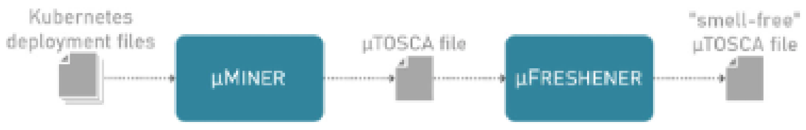

Figure 6: microTOSCA toolchain[4]

After the microservice-based architecture of an application is mined from its Kubernetes deployment, those deployment files are sent to the microMINER, which generates the microTOSCA topology graph, which is then sent to MicroFRESHNER. The microFRESHNER then detects the architectural smells which are not following the microservices design principles, which is out of the picture in this seminar paper.

3 Modelling the architectures

3.1 Modelling a monolithic architecture for a e-commerce application:

In the recent decade, monolithic architecture is being chosen by developers because of the simplicity it offers. The application is not only responsible for performing a single task. It can perform any kind of task to execute a function. It is executed in a single platform by combining all the modules into a single program.

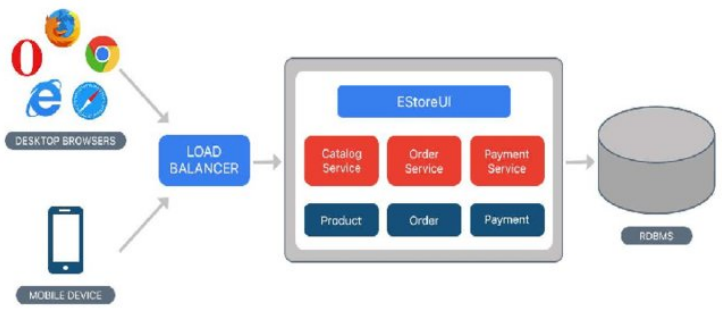

Figure 7: Monolithic architecture for e-commerce application[5]

The above figure depicts how an e-commerce application is deployed as one single program.

3.2 Modelling a e-commerce application architecture using the microTOSCA topology:

microTOSCA topogy defines different nodes, relationships and groups that are used to define orchestration specification of cloud applications which can be found at[6].

There are three types of nodes, which are service, communication pattern, and database. If we take the e-commerce application as an example, the service can be a cart for handling the products list. We will be having two or more components in the design. So, for establishing the communication between these components, we need communication patterns that can be of two types: Message Routers and Message Broker. Message Router is used for transmitting the routing information, and Message Broker is used for implementing the message broking Asynchronously or Synchronously. The database is used to store the data. In the e-commerce example, the product catalog consists of a MongoDB database. The database can be of any type(MongoDB, Redis, etc.).

Apart from the nodes and Communication patterns, we also have relationship named InteractsWith which can be used to denote a relation of a source node to a target node. This InteractsWith relationship has three properties, namely circuit_breaker, timeout, and dynamic_discovery. dynamic_discovery is used to identify whether the target node is dynamically discovered. Circuit_breaker and timeouts are used to know whether the relationship is being established by setting a circuit breaker or timeouts.

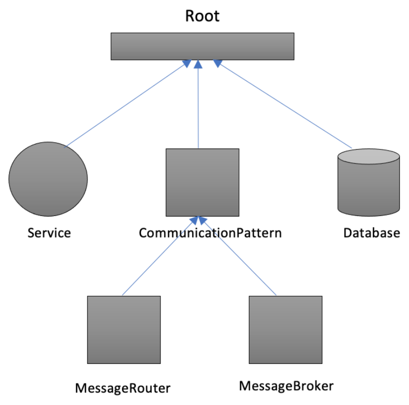

Figure 8: microTOSCA nodes and relationships[6]

The E-commerce application which I am modeling with microTosca model types consists of 7 services which are E-commerce home page (e-Commerce App), API gateway, Cart, Product Catalogue, Orders, Payment and shipping, Two communication Patterns (i.e., Message Routers and Message broker), and three databases (i.e., Redis, MongoDB, MySql). The application is intended to make the external users order the products through the e-commerce app page, where it directs the users to the API gateway. The users thereafter can access the other services like Product Catalog, cart, and orders. Each service has a database that has the newly updated records of the products. All the services are dynamically discovered by a message router.

The cart service is interacting with payment service to process the payment order. The cart service is also interacting with message router, which is dynamically discovering the payment service. Furthermore, if the payment service is unresponsive, the circuit_breaker set is used to avoid such cases. The cart service uses the message broker to queue the payment of the products which are successfully processed into the shipping service. The product catalog has a different database, as each service can use the database as per its requirement. The users can have a look at their orders via the service of the orders, which is responsible for storing all the orders. The product catalog and order services are also interacting through a message router.

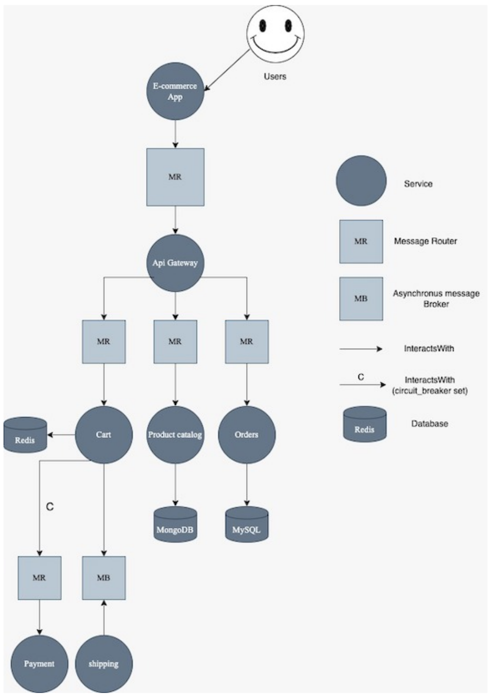

Figure 9: e-commerce application architecture using microTOSCA topology[7]

4 Mining the e-commerce application architecture

microTOSCA topology graph is used to model the architecture of the application. To generate this, we are going to mine it by deploying the application in the Kubernetes. This process is done by using a microMINER which is responsible for generating the microTOSCA topology graphs, which is modeling the architecture of the e-commerce application.

4.1 Mining the e-commerce application from Kubernetes deployment

First, the information will be taken from the application deployment in the Kubernetes. Then we monitor the packets that are being interacted with and their network packets. Afterward, the topology graph is refined after analyzing the network packets.

microMINER prototype is used to mine the architecture of the e-commerce application. It can also be considered that the main goal of this microMINER can be, make it easy to deploy in any other automation technologies other than the Kubernetes. This can be done by using the python programming language. For enacting the packet sniffing, the tshark command-line can be used. In the tshark command line, the following command can be used to mine the architecture of the application:

```
$ sudo python3.8 -m microMiner generate kubernetes source target [time] [test][name]
```

Source contains the path which will be having all the Kubernetes manifest files. These files will be detailing the application deployment. Target is used to specify where the microTOSCA specification can be stored. This microMINER will be running on the Kubernetes cluster master node. So, it helps the microMINER to interact with the master node where the Kubernetes engine will be running and also for managing the deployment of the e-commerce application.

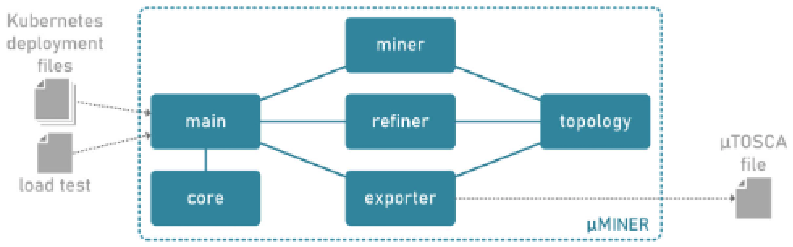

Figure 10: microMINER architecture[4]

There are different modules in the microMINER. They are main, miner, refiner, export, and the topology. The steps involved in the microMINER are as follows:

1. First the Kubernetes deployment files of the application are sent to the main module, which also offers a command-line interface. This main module coordinates the miner, refiner and the exporter modules.
2. The main module invokes the miner to start static and dynamic mining steps. Then the refiner is invoked, which helps in mining the topology graph.
3. Afterward, the exporter is responsible for exporting the architecture which is mined to microTOSCA.

Here, First step 2 is responsible for building and refining the topology graph, which is responsible for modeling the architecture of the e-commerce application. Then step 3 takes that topology graph and sends it to microTOSCA. To support the adaption of deploying this application into any other technologies other than the Kubernetes deployment, the main module first interacts with the core module.

5 Microservice patterns present in the e-commerce application and how they are used in the design

5.1 Decompose by Business capabilities pattern:

Services in the application should be defined according to their business capabilities. Business capability can be a business object. In our case, i.e., e-commerce application, Product catalog management is responsible for products, and Shipping management is responsible for shipping the orders. The microservice architecture will have the services according to these capabilities. By using this pattern, the architecture will be a bit stable as the services are loosely coupled.

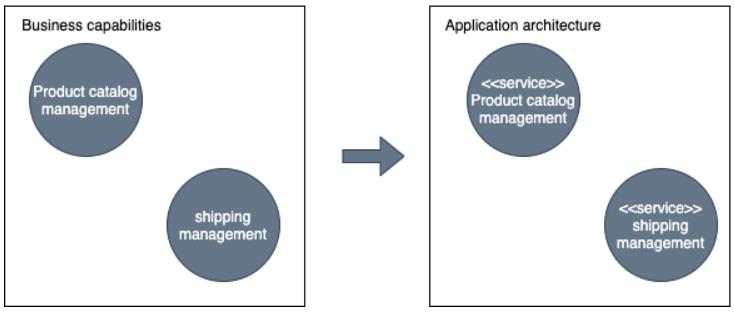
Business capabilities
Product catalog management
shipping management
Application architecture
<<service>> Product catalog management
<<service>> shipping management

Figure 11: Decompose by Business capabilities[8]

5.2 Decompose by Subdomains pattern:

There may be multiple domains and subdomains in an architecture based on their requirements. That will be different for every application. In the e-commerce application, We can consider the cart service as a domain and the payment service as well as shipping services as the subdomains as they are under the main domain cart. The graph is shown in the figure below.

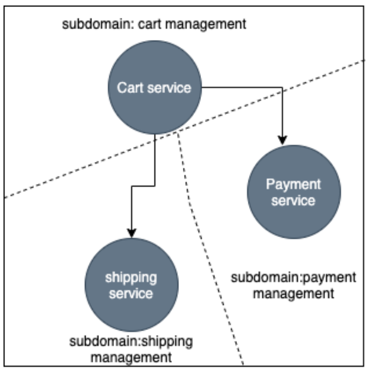
subdomain: cart management
Cart service
Payment service
subdomain:payment management
shipping service
subdomain:shipping management

Figure 12: Decompose by Subdomains[8]

5.3 Database per service pattern:

As we already know how the services are structured in the microservice architecture, each has its own database, and it only interacts with that services API. The database of one service will not be accessed by the other databases. There are different ways for a service to keep the data private. Private-tables-per-service, Schema-per-service, and Database-server-per-service. Every service will be having its own server for the database. The figure below depicts how this pattern is designed.

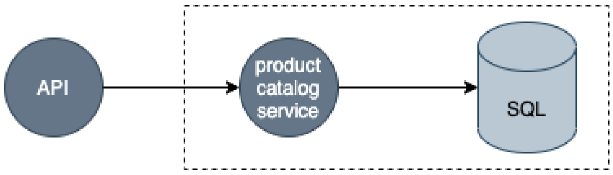

Figure 13: Database per service[8]

order table for the product catalog is shown below

ID	Product ID	Status
1	13	In bag
2	42	Delivered
3	07	wishlist

Table 1: Product catalog

5.4 API gateway pattern:

The APIs which are being provided by the microservices are quite different when compared to the client's expectations. Clients' requirements will be wide-ranging when it comes to browser support on desktop and mobile various other devices. We are taking the e-commerce application as an example here. It should be available on various devices, which can include desktops, mobiles, Ipads, etc. Adopting one size which can fit in every device doesn't work that well. We need to design an API gateway in such a way that each kind of device should have its own API. In the below figure, there are three kinds of API for there different kinds of devices for different clients they are web application API, Mobile Application API, and Public API gateway. There are also some disadvantages in using this pattern; the complexity will be increased because each API gateway needs to be managed and deployed individually. The cost for the building will also get increased.

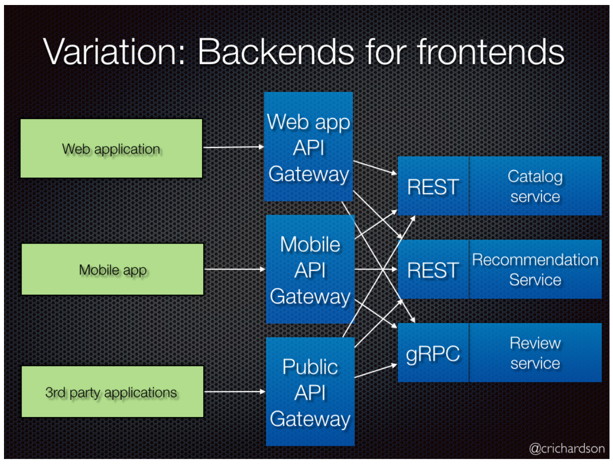

Figure 14: Backends for Frontends[8]

5.5 Command-Query Responsibility Segregation(CQRS) pattern:

The idea for the CQRS is based on the CQS concept by Bertrand Meyer. Basically, the objects will have two methods. One is Command methods responsible for only running actions while returning nothing. The other one is query methods which will return values but has no effect overall. The write and read requests will be two different models. For example, if a service using a particular database is not supporting the query. It can implement a query that retrieves the data which is owned by another service.

For example, In the e-commerce application, we have delivery service and order service, and we want to filter findOrderHistory() operation. Here only the order service stores the details of the orders. Delivery service cannot sort by the OrderCreationDate attribute as it is not storing the order details. The API composer can solve this by implementing an in-memory join. It takes the details from the Delivery service and joins them with the orders received from the order service.

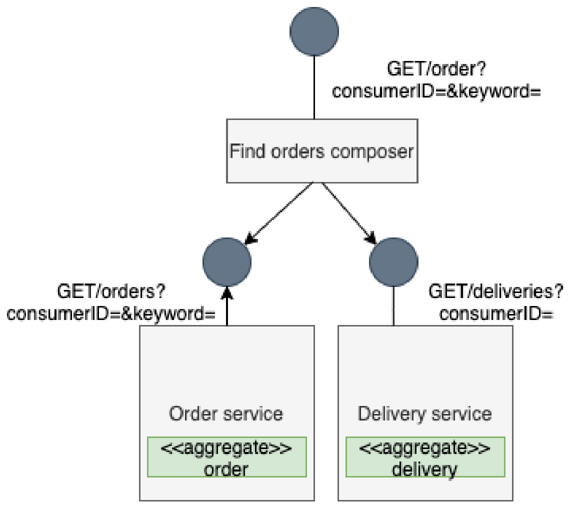

Figure 15: CQRS[9]

There can be a disadvantage here. The API composer should join the data sets, which are very large. So it is a bit cumbersome to do that. The solution can be receiving only the orders based on the IDS that are matching. But for this, we need bulk fetch API.

5.6 Access token pattern:

A microservice-based application(e-commerce application in our case) uses so many services to run and manage the application. For receiving the client requests, API Gateway is the only entry point. This API Gateway is the one that is responsible for authenticating these requests from the clients and for forwarding those to the other services. It authenticates those requests from the clients and sends the access token(JSON). The services can include these access tokens when they are communicating with the other services.

6 Conclusion

In this paper, We have presented microTOSCA, which enables microservice-based architectures by using the TOSCA standard. Then the application is deployed in Kubernetes, which is later mined using a microTOSCA tool called microMINER for getting a microTOSCA topology graph. Then microservice patterns, which are helpful in the design of the architecture, are described in detail. The mining technique used for getting the microTOSCA topology graph can also be extended in such a way that it should be able to work with Docker swarm, OpenShift, etc., which are based on container orchestration.

Finally, It is important to note that our data mining, modeling solutions are on the expensive side, especially when the size of the application is growing. As the application grows, the number of components will also get increased. So, enhancing the microMINER according to the scaling of the application can give better results which can be explored in the future.

References

[1] A. e. D., *July 27).* Best Architecture for an MVP: Monolith, SOA, Microservices, or Serverless? Rubygarage. [Online]. Available: https://rubygarage.org/blog/monolith-soa-microservices-serverless

[2] [Online]. Available: https://subscription.packtpub.com/book/web_development/9781789535297/1/ch01lvl1sec03/overview-of-microservice-integration-patterns

[3] L. K. Basics, *(n.* d.). Kubernetes. [Online]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/

[4] J. Soldani, G. Muntoni, D. Neri, and A. Brogi, "The tosca toolchain: Mining, analyzing, and refactoring microservice-based architectures," in *Software: Practice and Experience.* Published, 2021. [Online]. Available: https://doi.org/10.1002/spe.2974

[5] K. Gos and W. Zabierowski, *The Comparison of Microservice and Monolithic Architecture*, vol. 10, no. 1109, pp. 150–153, 2020.

[6] [Online]. Available: https://github.com/di-unipi-socc/microTOSCA/blob/master/microTOSCA.yml

[7] [Online]. Available: https://giters.com/di-unipi-socc/M2

[8] M. A. pattern, *(n.* d.). Microservices.Io. [Online]. Available: https://microservices.io/patterns/microservices.html

[9] C. M. P. Richardson, *1. edition.* Manning Publications, 2019.

[10] [Online]. Available: https://github.com/di-unipi-socc/microTOSCA

[11] A. . Qian, *May 21).* Advantages and Disadvantages of Microservices Architecture. Cloud Academy. [Online]. Available: https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/

[12] S. . Diguer, *July 28).* Microservices Advantages and Disadvantages: Everything You Need to Know. Solace. [Online]. Available: https://solace.com/blog/microservices-advantages-and-disadvantages/