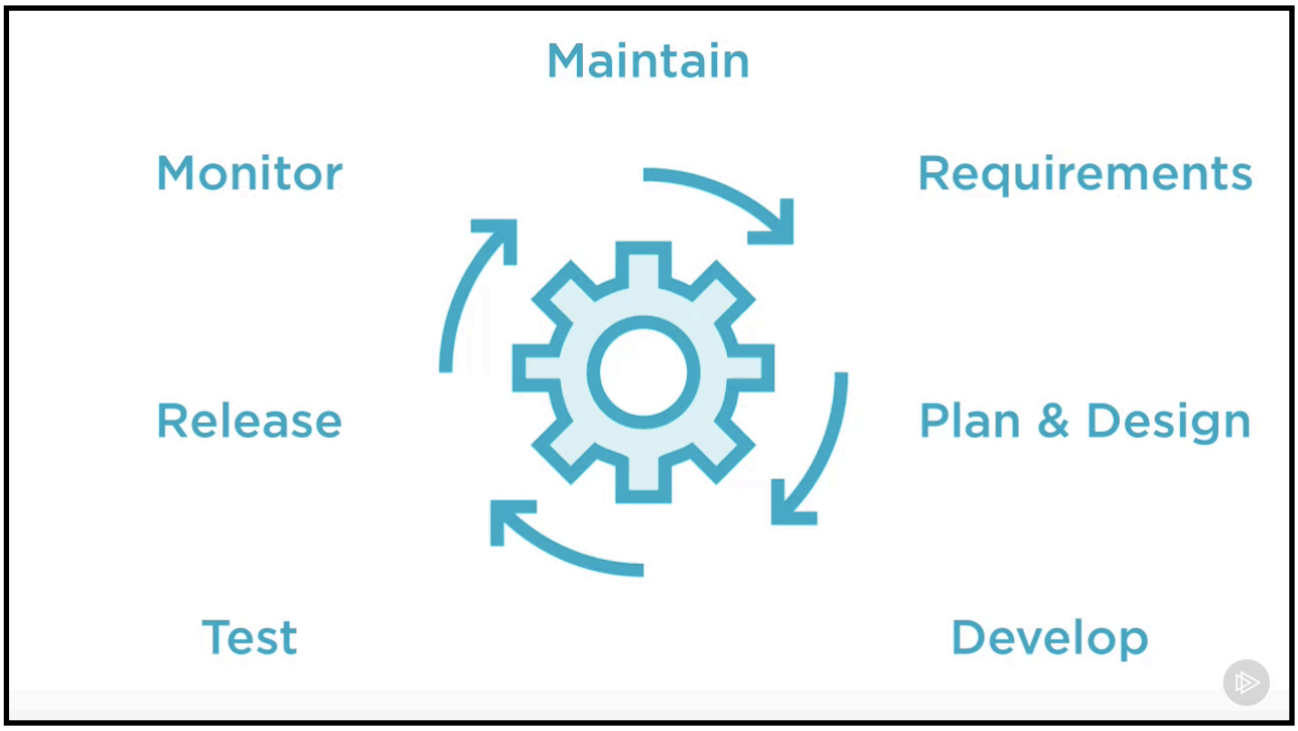
SOFTWARE DEVELOPMENT LIFE-CYCLE & MICROSERVICES

No matter if you follow the waterfall or agile model, the software development has a certain lifecycle that turns a project into a product. This project involves human beings to design, develop, deploy, and execute this product in production. Microservices have an impact on this development lifecycle. They have an impact on the technology, as well as the way your team is organized, or the way you run your software in production.

Let's focus on the typical Agile software development lifecycle :-



During the **analysis phase,** we gather the requirements. During the first sprints, this is the foundation of the application that defines the minimal requirements to get the application off the ground.

Then comes the **plan and design phase**, where we plan this sprint to make sure that we can meet the requirements in the given timeline, as well as pulling the design elements from the requirements.

The **development phase** is the longest. This is where we go into action, and develop all that was planned. This goes hand in hand with the **testing phase**. Testing is a vast topic, as it can cover unit testing, integration testing, UI testing, performance, and stress testing.

At the end of this sprint, we **release** all the planned features into production. Now the product is alive and kicking.

Now's when our customers start using our application that now needs to be **monitored**. This is an important phase, as it will give us some feedback on how our customers use the application. Based on these results, we can build a list of requirements for the next iteration.

And of course in the long run, we'll have to **maintain the application**.

During this development lifecycle, several different people are involved :-

Users and business analysts will gather their requirements, and write down the user stories for a given sprint.
Developers, web designers, and architects will design and develop the software.

Project managers will coordinate the project, and make sure to synchronize everyone, so the target can be reached.

Once running in production, the **operational team** is responsible for monitoring, dealing with security, and making sure the software is up and running 24/7.

An organization will typically have several of these projects, bigger ones, smaller ones, crucial runs, running 24/7, side projects running only certain occasions. They could be related or not, developed in different countries by different teams. Each project is mostly independent, so some might have used a waterfall approach, some agile. Some projects can take a few weeks to be built, others a few years. **Some will succeed and some will fail. The succeeding project will turn into software. And finally, each software will get deployed.**

Some will be deployed on servers or virtual machines, others will get deployed directly on the cloud. **This is a quick resume of what most IT companies do when building software. Microservices are also pieces of software, and they roughly follow the same development lifecycle with some slight changes.**

MICROSERVICES

"***The term microservices describes a software development style that has grown from recent trends to set up practices meant to increase the speed and efficiency of developing and managing software solutions at scale.”***

This set of practices is **technology agnostic**, meaning that there is not one single technology or programming language to build microservices. In fact, you can build microservices with roughly all programming languages. It's more about applying a certain number of principles and architectural patterns that will finally create a microservice architecture.

Let's now focus on the word itself:-

The word ‘Microservices’ is made of **‘Micro’** and **‘Services’**.

 # MICRO :-
You could ask how big is a microservice?
What's big, what's small?
Who decides the measure?
Is it the number of lines of code, the number of developers in the team?
Well, we don't know.
There is no universal measure defining the ideal size of a microservice.

Each microservice is supposed to do one thing and do it well. So the micro refers to the scope of the service functionality, not the lines of code.

A microservice is a service built around a specific business capability, which can be independently deployed. We call that bounded context. So to build a large enterprise application, we have to identify the sub-domains of our main business domain, and build each sub-domain as a microservice.

For example, an ecommerce application has a large domain, but when you think about it, you can split it up into several smaller sub-domains, such as user management, catalog, invoicing, and so on.

 # SERVICE :-
A service is an independently-deployable component of bounded code that supports interoperability through message-based communication.
Here you might say hey, I've already done something like that, it was called service-oriented architecture. If you've implemented SOA, you're already familiar with the concept of modularity and message-based communication.

James Lewis and Martin Fowler, both of Thoughtworks, were the ones who coined the term microservice. The original definition they gave summarizes it all -
“The microservice architectural style is an approach to developing a single application as a suite of small services, each running on its own process and communicating with lightweight mechanisms."

By lightweight mechanisms, they meant HTTP resources. These services are built around business capabilities, the famous bounded context, and are independently deployable.

Sam Newman, also from Thoughtworks, had a more concise definition -
“Microservices are small, autonomous services that work together.”

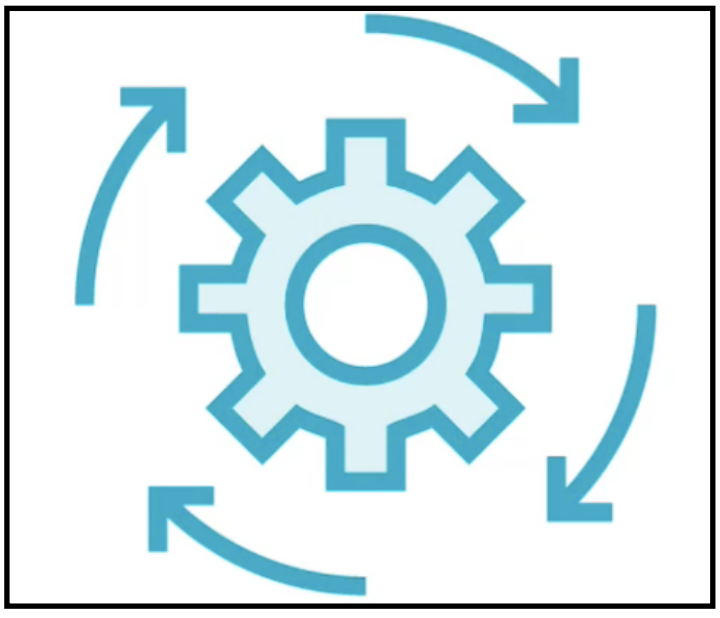
This quote emphasizes the level of independence, the limited scope, and the decomposition of an application into highly-cohesive and loosely-coupled services.

BACK TO: SDLC & MICROSERVICES

The way we've been developing software is basically one project results in one product, one application, which is one piece of software.

The bigger the project, the longer its lifecycle. It might take you a few months or years to go from gathering the requirements to actually releasing it into production.
The bigger the project, the bigger the teams, as you need more people to design, more people to develop, and more people to test.

Let's call this big piece of software a monolith for now :-

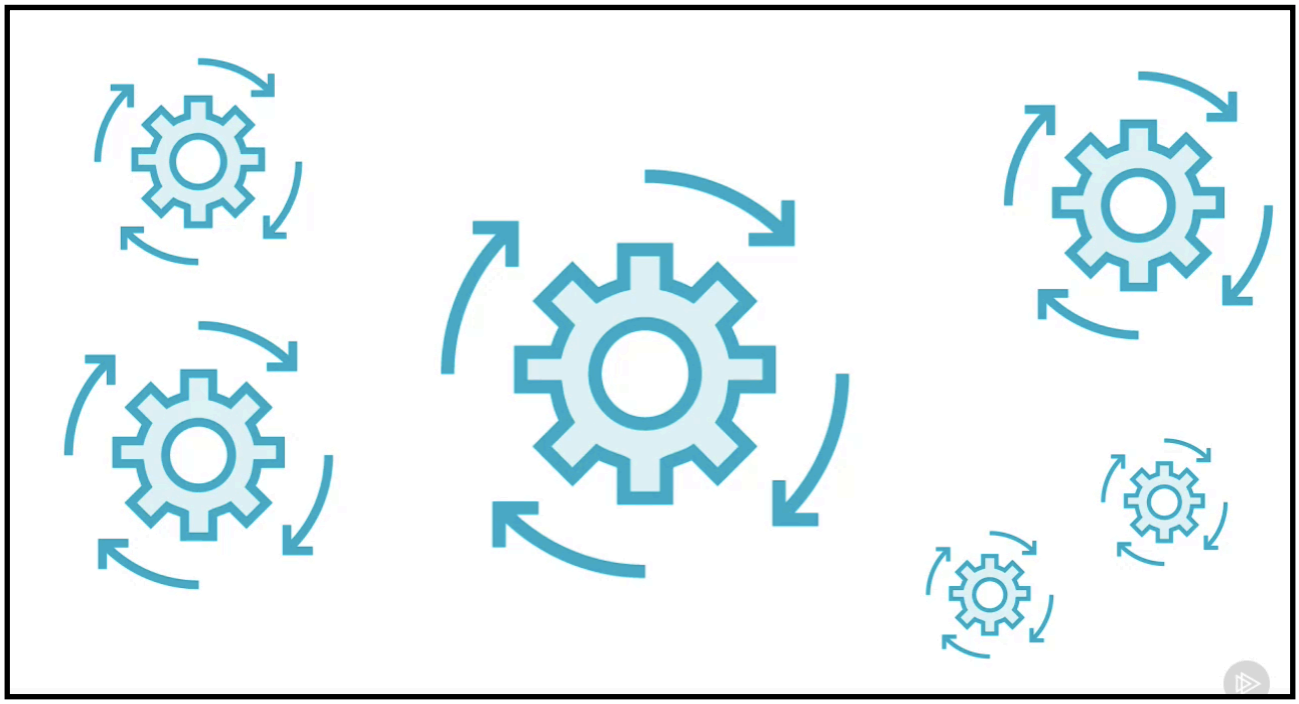


Now, if you manage to split this monolith into smaller microservices, then you can concentrate on something smaller.

The development lifecycle is still the same, with the same different phases as it has to go through. But a smaller project iterates faster from requirements, to development, to releasing into production. This is good for time to market.

A smaller project also means a smaller team. In fact, instead of having several teams located in different offices, each working on their own task and not communicating much with the other teams, you can have a single one working closely together.

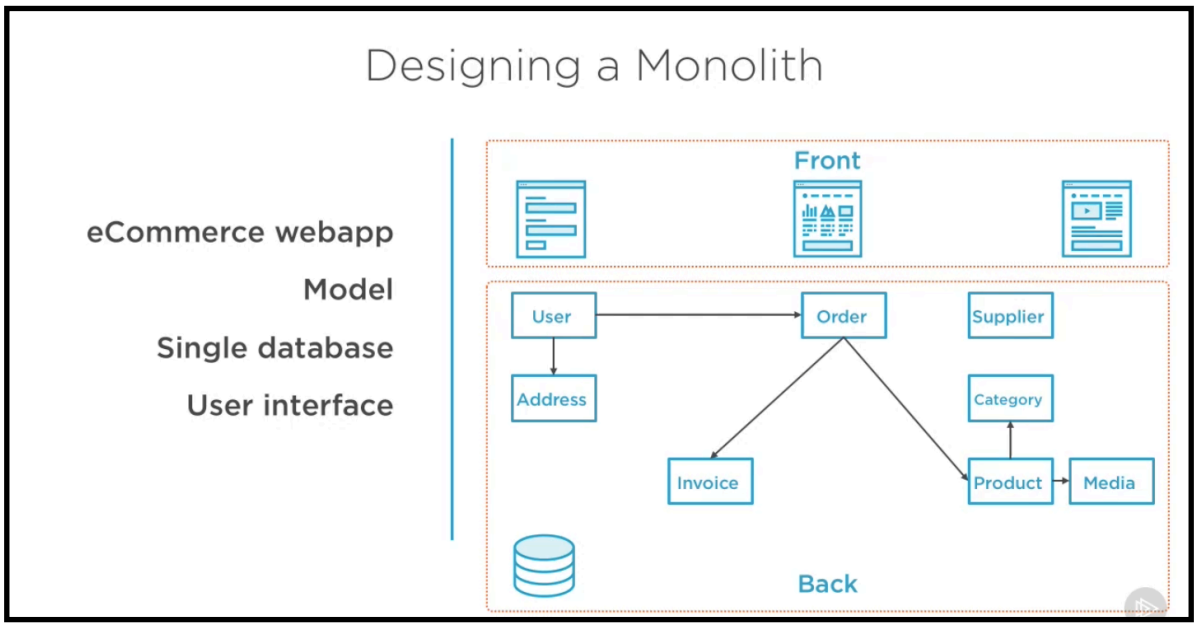
Microservices will force you to be more agile, to other users, the developers, the operational team, the QA team, all together working on the same product.



But of course, if you've subdivided an application in smaller pieces, that means you'll end up with several projects living in parallel. Because they are small, they will all benefit from having small teams, faster development lifecycle, and getting to production on time.

Each microservice lives independently, but they all rely on each other. Being small in terms of business functionalities means that you need to interact with other microservices, so they can all together achieve the tasks of your application. They all get deployed in production at their own pace, on-premise on the cloud, living side by side with other microservices.

BUILDING A MONOLITH



Let's imagine that we need to build an ecommerce web application that sells products, creates purchase orders, checks the availability of the product from the supplier, generates invoices, and ships the goods to the customers.

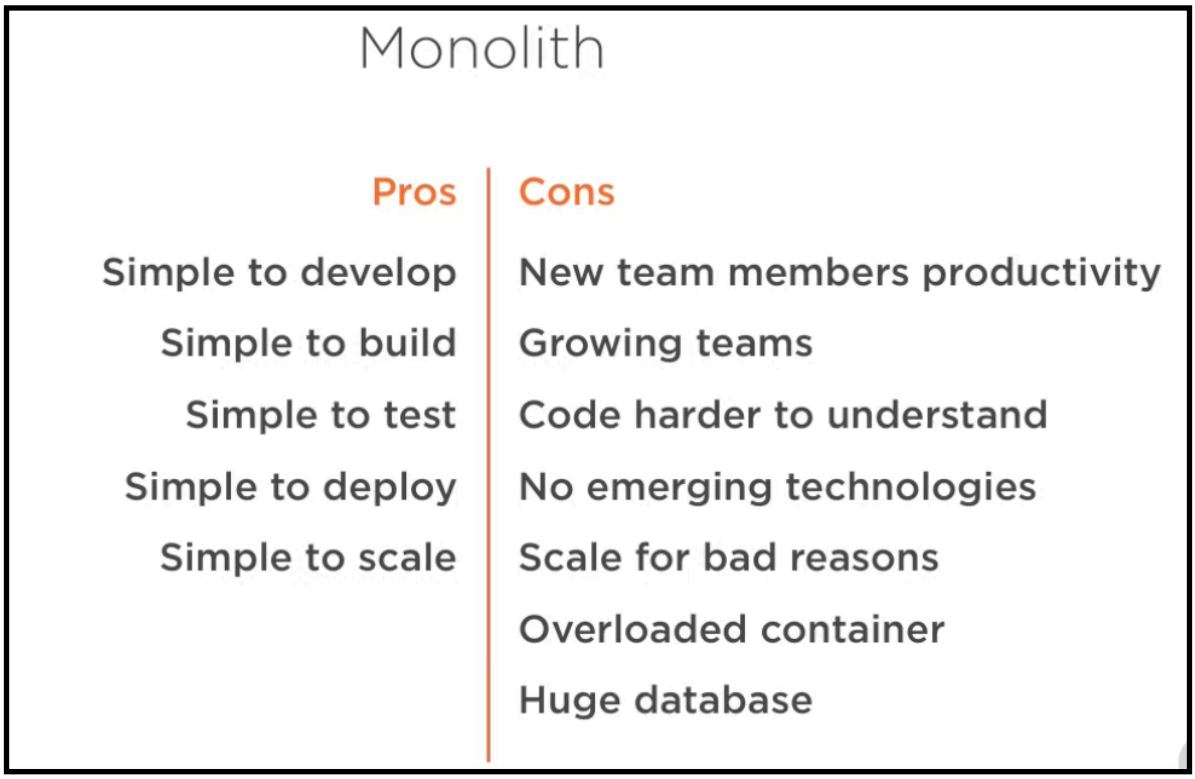
We will first model the application. -
Our customers need a user profile with an address where we can deliver our products.

This user puts an order which basically is a list of products he or she wants to buy. A product belongs to a category, for example, a book, a CD, a chair, and can be described with medias. Think of a PDF file for a book or an MP3 file for a CD.

Once an order is created, the application needs to contact the suppliers and creates an invoice for the customer, so he or she can pay. All this information will be stored in a database.

And of course we need a user interface made of a few screens. I'm sure you've seen this kind of architecture before. **We have a frontend that talks to a backend.**

The entire backend will be deployed as a single monolithic application with its own single database. Being responsive, its single-user interface supports a variety of different devices, including desktop browsers, mobile, or tablets. The application might also expose APIs for third-parties to consume. You can even run multiple instances of the application behind a load balancer in order to scale and improve availability. We've been using this kind of architecture for a long time, it works, and has proven to be efficient. So why shall we care about microservices?



Monolith have a number of benefits, in fact they are simple.

They are **simple to develop,** the technical stack is limited to a few frameworks, languages, and databases.

The code can fit in a single IDE and **can easily be built.**

They are **simple to test,** of course you still have to mock a few external services when doing integration tests, but most of your tests test a single piece of software.

Monoliths are also **simple to deploy.** You simply need to deploy the final artifact on the appropriate runtime, configure a few properties, and that's it.

They are also **simple to scale,** you just run multiple instances of the same application behind a load balancer, and you're done.

But what if you want to add more functionalities to our E-Commerce application?

Once the application becomes large, and the teams grow in size, the monolithic approach has a number of drawbacks. The bigger the application is, the **harder for a new team member to become productive.**

And **you will need a few extra teams to handle the size of a growing monolith.** Once the application reaches a certain size, it will be necessary to divide up the organization into teams that focus on specific functional areas. For example, a team just developing the UI, another one for the backend, another one for testing, for deploying.

The problem with a monolithic application is that it prevents the teams from working independently. The teams must be coordinated.

The code of the application will get harder to understand and modify. As a result, development typically slows down, followed by the quality of code.

You **won't be able to easily take advantage of emerging technologies**, but instead bet on a longterm commitment to a technologies tag.

What if a small part of the application needs a different database? You'll need to change a lot of code just for that. You'll end up scaling the application for bad reasons. If you need more CPU to handle all the invoices, let's say for Christmas sales, you will need to scale the entire application, not just the invoices. **The web container hosting the application will get overloaded. The larger the application, the longer the container will take to start up, and the more resources it will consume. The database contains all the data and might become huge, which will have an impact on the application's performance.**

BUILDING MICROSERVICES

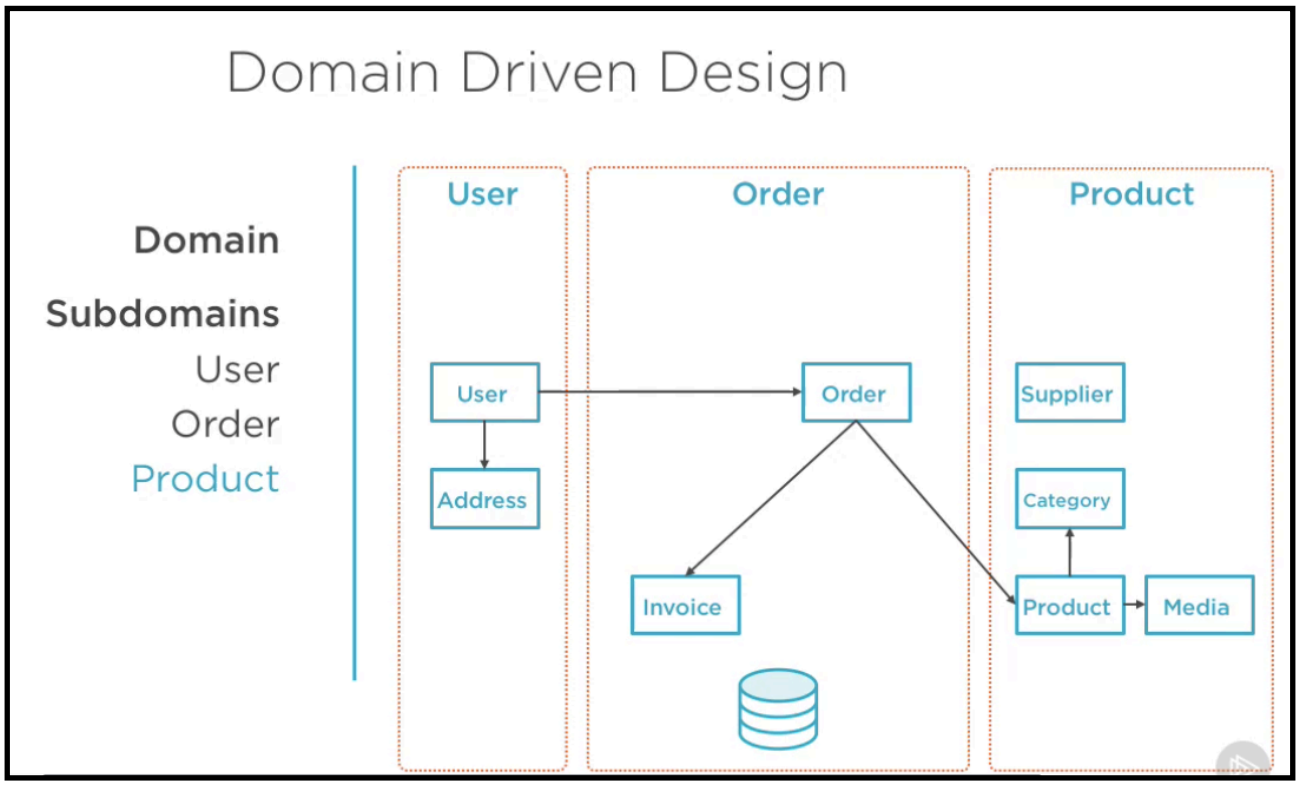
Let's start our journey in building our ecommerce application the microservice way, the journey being long. Let's start by the beginning, the design phase.

We need to define an architecture that structures the application as a set of loosely-coupled collaborating services. For that, we can define services corresponding to domain-driven design subdomains. A domain here in our ecommerce application consists of multiple subdomains. Each subdomain corresponds to a different part of the business.

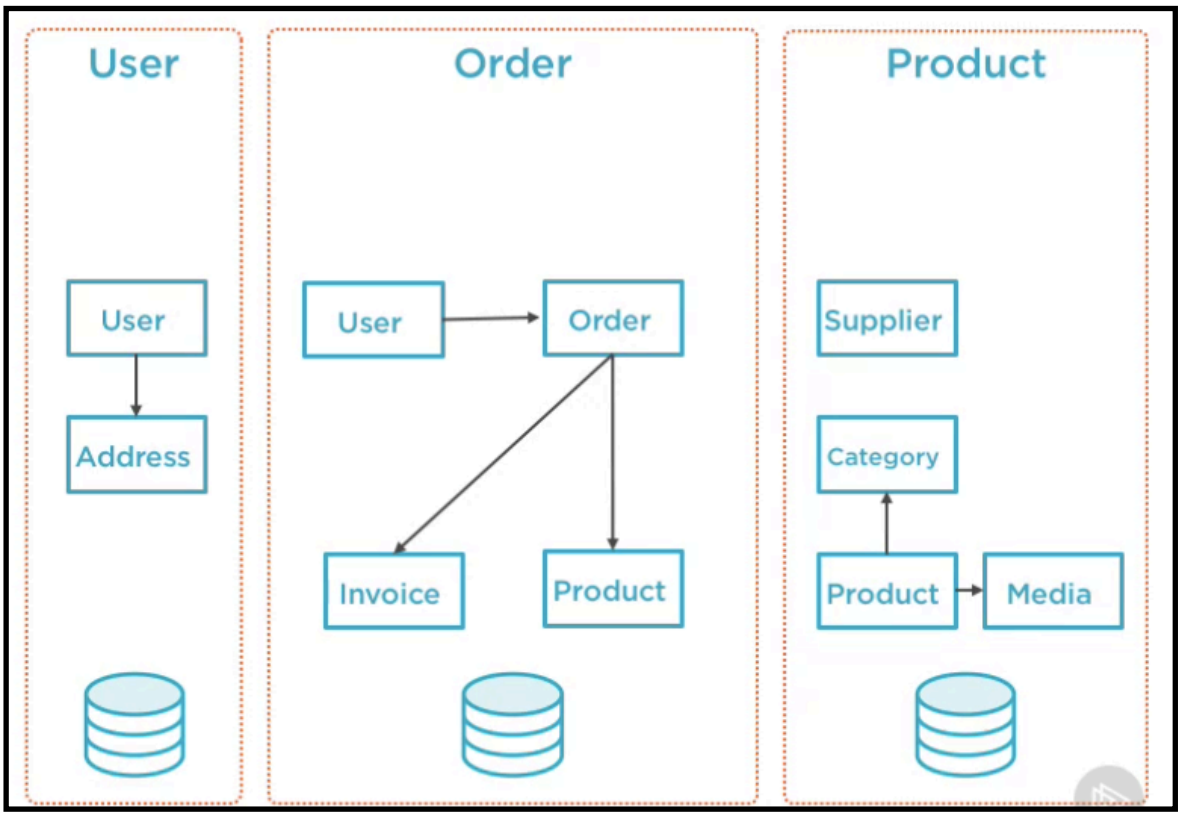
So if we look carefully at our domain model, **we notice that we can subdivide into three subdomains.** The user subdomain deals with the user authentication, its profile, its address.

Then we can see that there is a whole piece processing purchase orders, invoices with batch rates, and discounts.

Finally, we have an entire subdomain related to products and the suppliers selling these products.



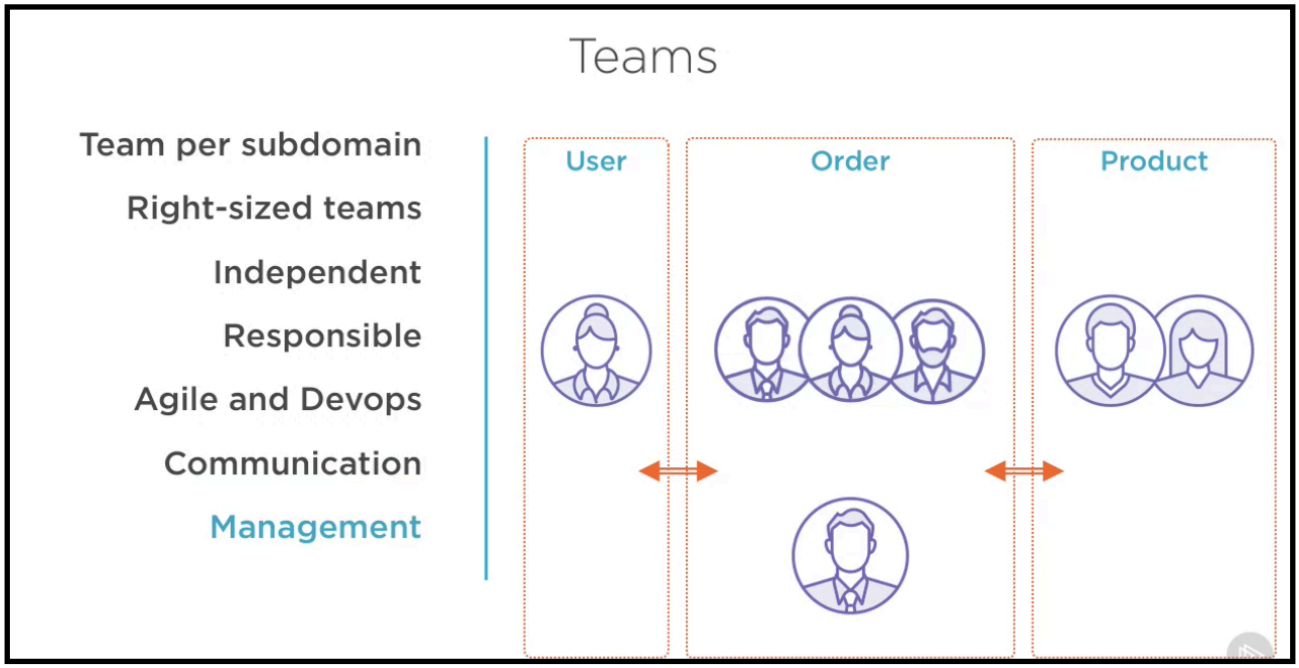
But now we have a problem, we have dependencies between these three subdomains. They should be totally independent. ***Domain-driven design gives us a set of techniques and patterns to help us in designing our model. One technique is to duplicate the entities that depend on each other.***



For example, here we duplicate the user. Being separate entities, we can even specialize them. In the user domain, the user can have a profile with social media, and in the order domain, a user can have information about payment such as credit card. Same for the product that is duplicated in both domains. Each subdomain being responsible for its own processing and data, it should have its own database. In fact, **sharing databases is discouraged and is an empty pattern in the microservice world.**

ORGANISATION IN MICROSERVICES

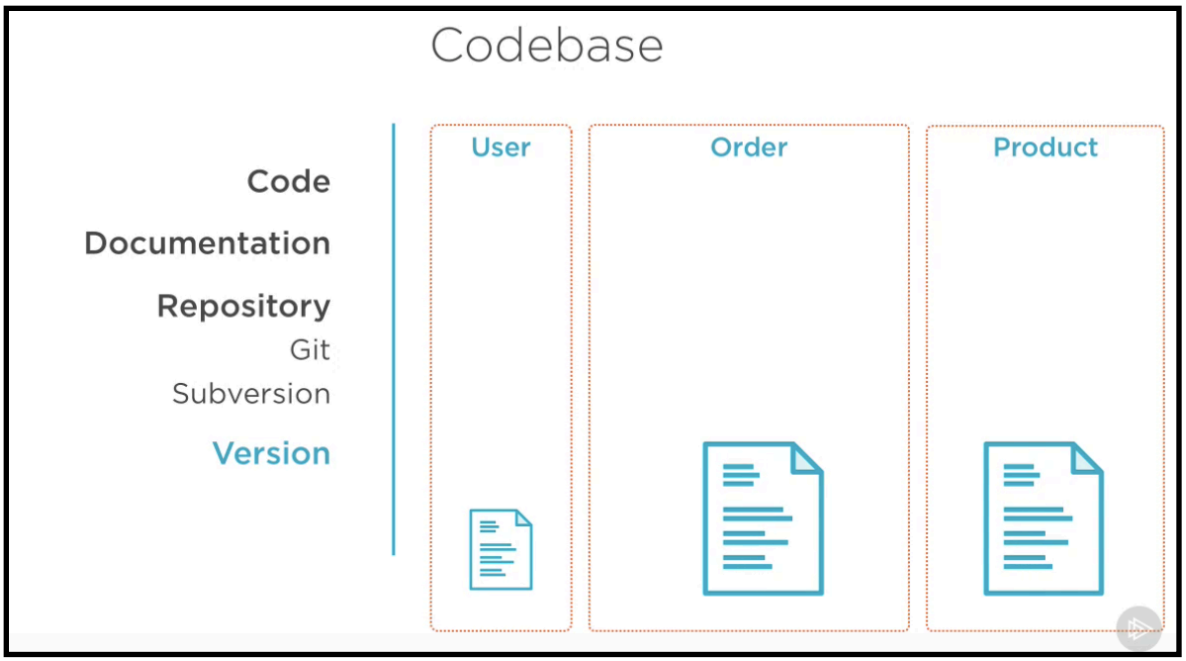
Microservices also have an impact on the way you organize your teams and your code.



The first impact is that you go from one team per domain to one team per subdomain. Also, from the design phase, it got clear that the user subdomain covers less functionalities than the order subdomain. **This means you can right-size each team.** The smaller team for user, the bigger team for product, and slightly bigger for the order subdomain that covers quite a lot of business functionalities. **Each team is now totally independent, but more important is solely responsible for the entire lifecycle of the product, from development to deployment.**

That's when you need to embrace Agile and Devops methodologies, so each one with its different skills learns how to work together on the same product. Even if the teams are all independent, remember that those microservices end up communicating with each other. This means that you will still have to manage and orchestrate the integration between the teams at a certain point in time.

Each team being independent and separated, they can each have the code and document in separate repositories. That is using different version control software such as Git, Subversion, or Mercurial. Remember that each software is independent, therefore versioning is important. For example, the purchase order team might be developing the 1. 2 version of the microservice, while the product team will be testing the 4. 8 version of the microservice.



DATA STORE IN MICROSERVICES

 # In the monolith approach, the entire data of the application is stored in a single database. With microservices, this is empty pattern and has to be fixed.

In a microservice architecture, most services need to persist data in some kind of database. Services must be loosely coupled, so that they can be developed, deployed, and scaled independently. Therefore each needs its own independent data storage.

In fact, this makes sense. Different services have different data storage requirements. One will heavily rely on transactions, the other one will be write mostly, another one read only. In fact, for some services, a relational database would be the best choice, for example, for our product microservice.

Other services might need a NoSQL database such as document database, which is good for storing unstructured data. Our user microservice could use a graph database or even an LDAP directory.

Using a database per service ensures that the services are loosely coupled. Changes to one service's database does not impact any other services.

 # Having separate databases bring another challenge, data synchronization. If a customer changes an email address in the user subdomain, the change has to be replicated in the order subdomain. Same for product.

In the microservices world, there is no distributed transactions. This means that when you update the user subdomain, you can't span a long-running two-face commit distributed transaction to the order subdomain. That would have a performance impact on your application. Without such distributed transaction, you lose the immediate consistency of your data. You then have to move to an **eventual consistency model**.

 # Eventual consistency model:-
This means that a service publishes an event when its data changes. The other services consume that event and update that data. This is related to the capture-data change and event-sourcing patterns. This mean that for a period of time, the same product might have a different price or description on both subdomains. **There are several was of publishing events, including tools such as Akka, Kafka, or RabbitMQ. A tool like Debezium uses events to handle captured data change.**

USER INTERFACE (UI) IN MICROSERVICES

Not all microservices have a user interface, but when they do, there are several techniques we can use if we need to aggregate them.

One benefit of microservices is that each team develops in a relatively isolated and independent manner. They can develop and maintain their own set of graphical components, aggregate them, work with designers, and have the best user experience for their use cases.

But one challenge quickly arises, how to implement a unique user interface that displays data from multiple microservices.

Consider, for example, our purchase order detail page, which displays the name and the price of the product, but also its availability. We need to go to microservices, and most important, **we want our users to feel that they are interacting with a single application. So the idea is to do a user interface composition.**

There are actually two design patterns that we can use:-
Server-side page composition, this allows you to build web pages on the server by composing HTML fragments developed by multiple microservices teams.

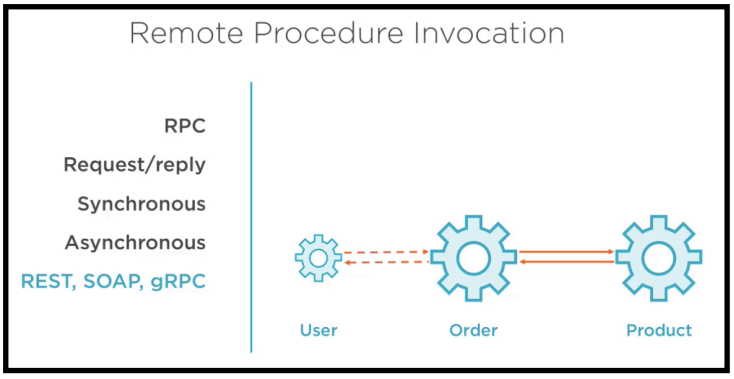
Client-side composition, this is where the browser builds a single UI interface composing UI fragments. This means that we need a UI team that is responsible to implement the application skeleton that aggregates multiple microservice UI components. And of course, this composition has to be responsive, taking into account all different devices.

SERVICES IN MICROSERVICES

Once our subdomain is isolated, then its package deployed and executed independently of one another. These microservices can be more or less complex, can store data or not, have a user interface or not, but most of them will end up communicating with each other. When they do, they need to expose and consume APIs, use a communication protocol, and choose a communication style.

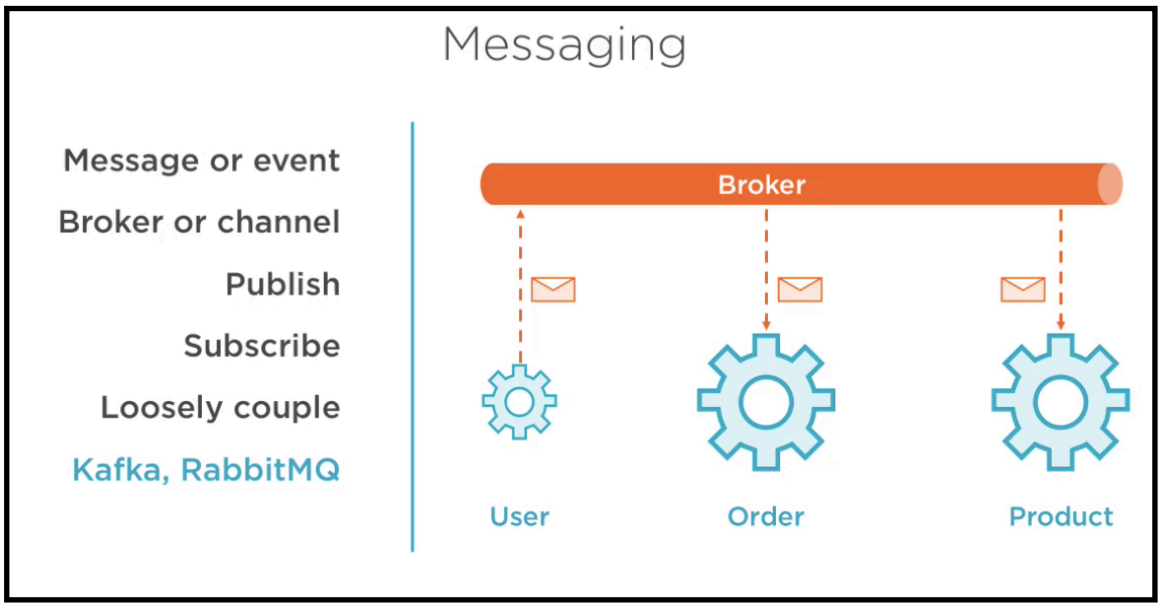
Let's focus on the two main communication families-
Remote Procedure Invocation and messaging.

Remote Procedure Invocation, also known Remote Procedure Call, is the simplest and familiar interprocess-communication protocol. **It works on the request/reply principle.** A service requests something to another service, and this one replies back.



For example, our purchase order microservice requests that the product microservice to get the price of a specific product. **This call can be synchronous,** meaning that the caller will hand until the request comes back, **or asynchronous, for example,** the user microservice invokes the order microservice to get a copy of a specific purchase order by email. This can be done asynchronously. The order microservice will send back a notification once the email has been sent. **There are numerous examples of RPI technologies such as REST, SOAP, or gRPC.**

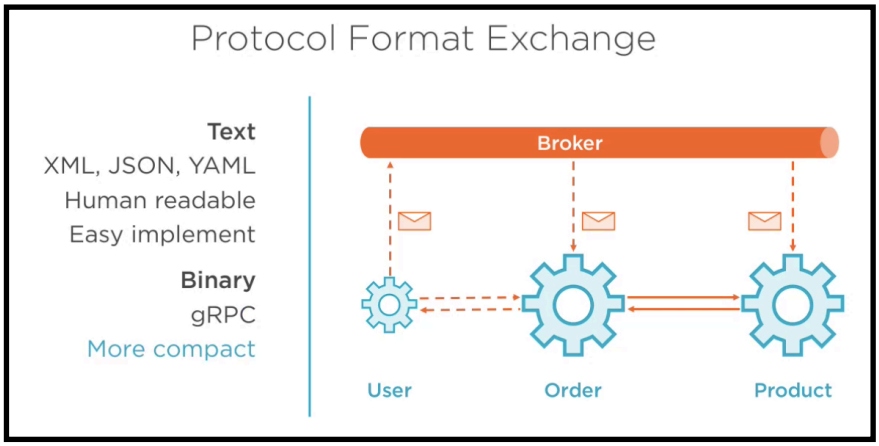
Messaging, is when microservices exchange messages or events via a broker or channel. This works as follows.



When a microservice wants to interact with the other, it publishes a message to the broker. The other microservices subscribe to that broker if interested in such messages, and receive the messages at a later stage. These microservices can then proceed to update their own state.

Asynchronous messages play a significant role in keeping things loosely coupled in a microservice architecture. They also improve availability since the message broker buffers messages until the consumer is able to process them. There are numerous examples of message brokers such as Apache Kafka or Rabbit MQ.

Protocol Format Exchange, No matter if we use messaging or Remote Procedure Invocation, we need to define the format of the data microservices exchanges. Depending on the needs of your architecture, you can use text or binary.

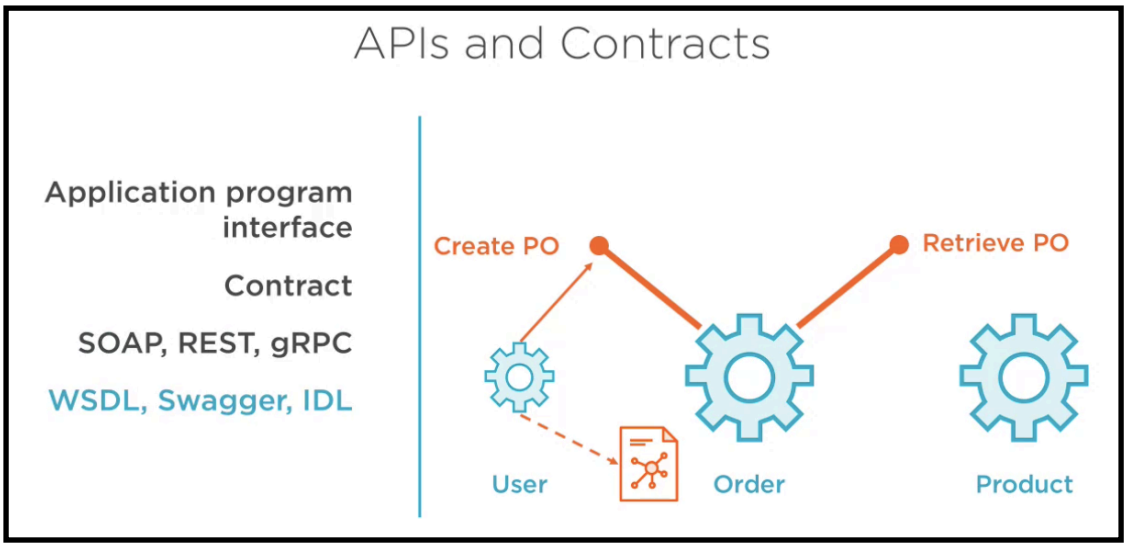


Text messages can take several formats, even if the most used today are XML, JSON, or YAML. The main advantage is that they are human readable, easy to implement, and easy to debug. You can also exchange binary messages. Binary protocols such as gRPC, for example, are more compact, but more difficult to handle.

We just saw that each microservice can communicate through Remote Procedure Call, messaging, exchange text or binary messages, but how does each team know how to invoke an external microservice?
Through APIs and contracts.

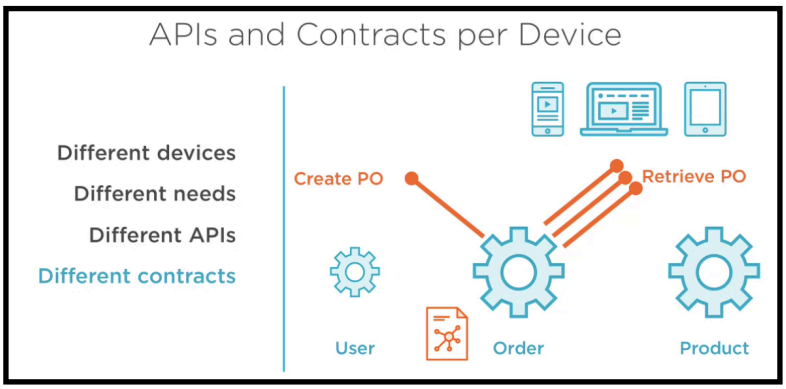
An API, or application program interface, is a set of routines, data structure, and protocols exposed by the microservice.

Let's say our order microservice exposes two APIs, one to create a new purchase order, and a second one to retrieve an existing one. For the other services to know what's exposed and how to invoke it, the order microservice exposes a contract. This way the user microservice gets the contract, reads it, and discovers how to invoke a remote API to create a purchase order.



Depending if you use SOAP, REST, or gRPC, you will have to describe your microservices API using WSDL, Swagger, or Interface Definition Language. Like most applications today, we need to be aware of the multiplicity of devices, as well as network constraints.

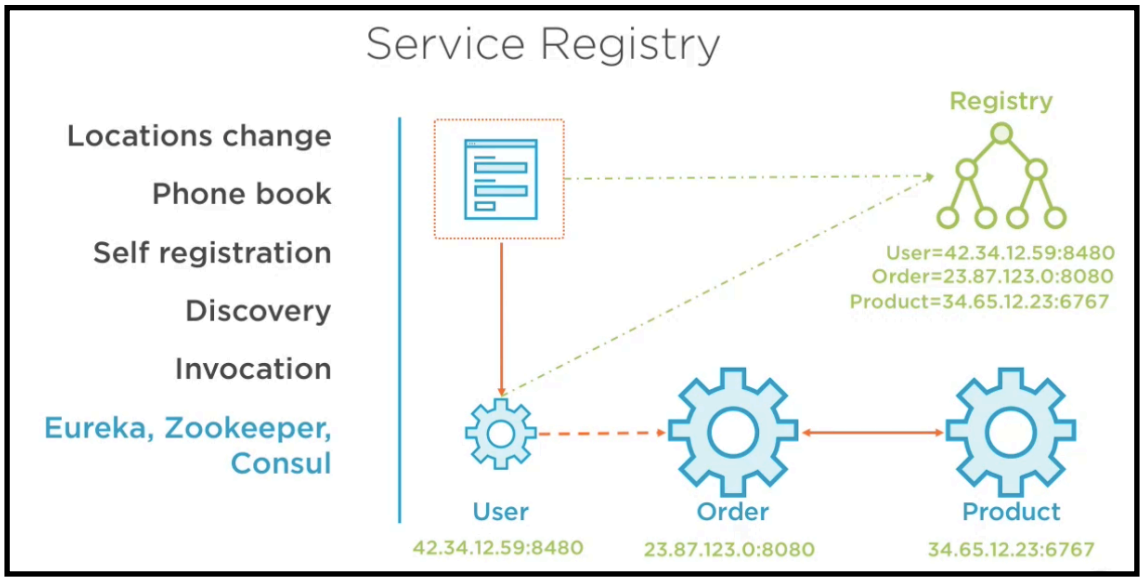
If a laptop connected to a good optical fiber network needs to retrieve a specific purchase order, we might want to give it all the details, including a PDF representation, but if the same API call happens from a mobile device stuck in public transport with bad internet connection, we might want to show only a subset of information. As you can see, each device has different needs, therefore we should have different APIs and contracts per device. This is a common practice that also has to be applied to microservices architecture.



DISTRIBUTED SERVICES IN MICROSERVICES

SERVICE REGISTRY

When we have microservices talking to each other through a network, we need to implement a few extra patterns to make sure the system is reliable. **A microservice-based application typically runs in an environment where the number of instances of service and their network location change dynamically. So how does the client of a microservice discover its location if it's constantly changing? The answer is by using a service registry.**

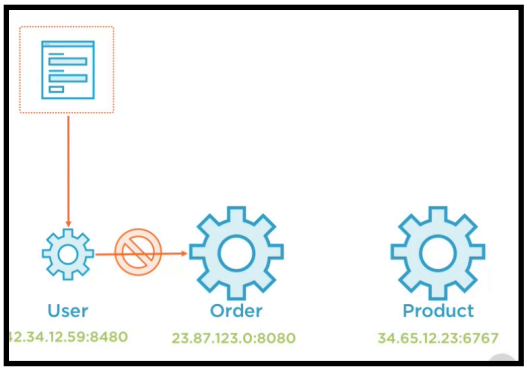


A service registry is a phone book of services with their locations, letting clients look up services by their logical names. The first thing our microservices have to do is self registration. This means that they need to register the network location on startup, and lately deregister on shutdown.

When making a request to a service, the client needs first to discover the location of a service instance by querying the registry. And then it can invoke the needed microservice. **Famous service registries are Eureka, Zookeeper, or Consul for example.**

Cross-Origin Resource Sharing (CORS)

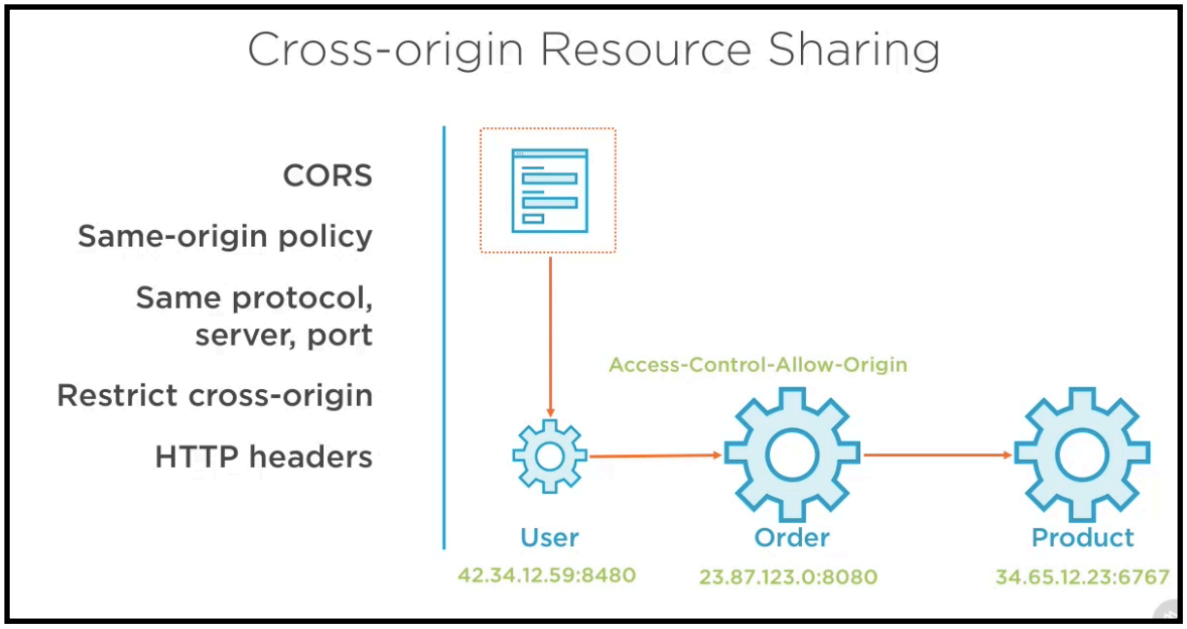
When dealing with microservices located in different servers, we quickly need to deal with CORS. **In HTTP, the same-origin policy is very restrictive. Under this policy, a document hosted on the user microservice can only interact with other documents that are also on that same server.** In short, the same-origin policy enforces that documents that interact with each other have the same origin.



An origin is made up of the protocol, HTTP or HTTPS, the host, and the port number. But in a microservice architecture, services are located in different origins and need to talk to each other, that is crossing the origin. **For security reasons, browsers restrict cross-origin HTTP requests initiated from within scripts.**

<- (BY DEFAULT)

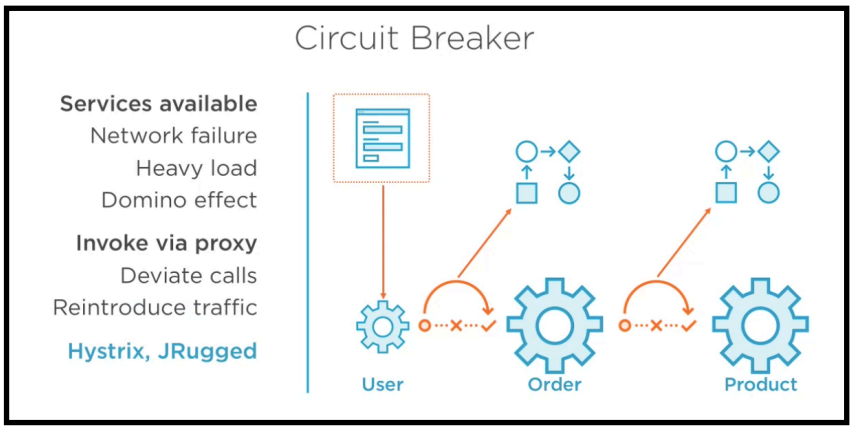
To allow cross-origin resource sharing, microservices have to use additional HTTP headers to let a user agent gain permission to access selected resources from a server on a different origin. That's typically dealing with the access-control-allow-origin family of HTTP headers. But CORS is not the only thing that can retain a call to be made between two microservices. With Remote Procedure Invocation, the services need to be available, as there is no intermediate broker such as messaging.



CIRCUIT BREAKER

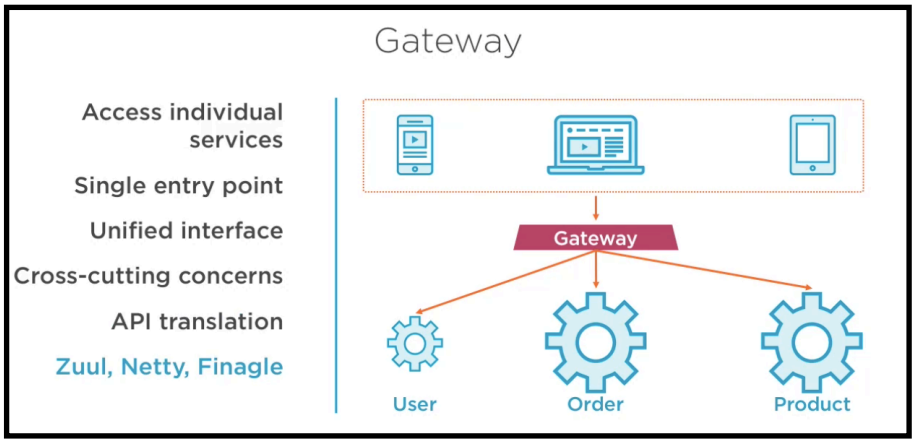
When our user microservice synchronously invokes the purchase order, there is always the possibility that the purchase order is unavailable. This can be because of network failure, or because purchase order is under heavy load and is essentially unusable. The failure of the purchase order microservice can potentially cascade to the user microservice, and then through all the entire application. **We call that the domino effect. One failure in one system can trigger other systems to fail.**

To avoid this, we need to introduce a circuit breaker. **A circuit breaker is a way to invoke a remote service via a proxy in order to deviate the call if needed.** For example, if the number of consecutive failures crosses a certain threshold, the circuit breaker will stop attempting to invoke the remote service and will deviate the calls. After the timeout expires, the circuit breaker will start allowing a limited number of requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. The circuit breaker slowly attempts to reintroduce traffic. **There are already a few circuit breakers out there such as Hystrix or JRugged.**



GATEWAY

Let's look back at our user interfaces. **Each microservice has its own set of graphical components, but at the end of the day, they must be aggregated into a single application. How do these components access the individual services?**



One solution is to have a 1:1 relationship between the component and the microservice, but then each call needs to deal with cross-cutting concerns such as security.

A better approach is to have an API gateway, that is the single entry point for all clients. This allows each client to have a unified interface to all microservices. The gateway can then handle requests in one of two ways. Some requests are simply routed to the appropriate service, others can handle cross-cutting concerns like authentication, authorization, or determining the location of services via the registry.

A gateway can also be the ideal place to insert API translation. Different devices need different data, therefore the gateway can expose a different API for each client. There are a few gateways that can be used out of the box in a microservice architecture such as **Zuul, Netty, or Finagle.**

SECURITY IN MICROSERVICES

Handling security is not specific to microservices. Any system nowadays needs some sort of security mechanism. But with microservices, a few extra cares have to be taken. **Authentication and authorization are the terms used when controlling access to a service and enforcing policies.**

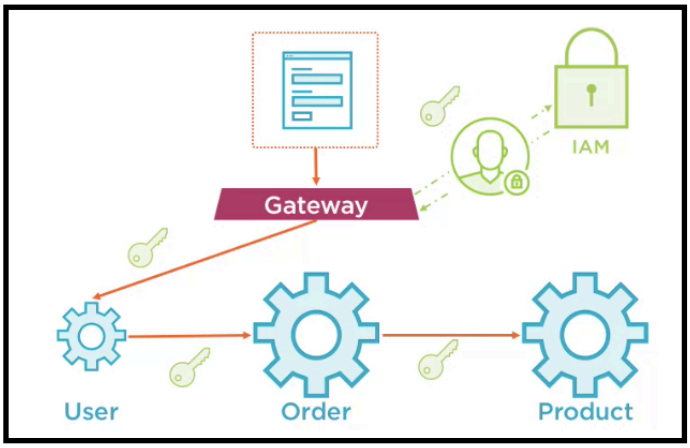
Authentication is the process of asserting that somebody really is who he claims to be. For example, asking for a user and password. **Authorization** refers to rules that determine who is allowed to do what. For example, only administrators can remove existing users from our user microservice. For that, we use an identity and access management system. It addresses the need to ensure appropriate access to resources across our distributed system. This means that our microservices don't have to deal with logging form, authenticating users, or storing credentials. They delegate authentication and authorization to the identity and access management system. For example, a user tries to authenticate, with the wrong credential, it won't be allowed to access the user microservice. If the credentials are correct, the call can then be made to the service.

With single sign-on, once logged in, users don't have to log in again to access a different microservice. This is where you can find authentication protocol such as Kerberos, OpenID Connect, OAuth 2. 0, or SAML.

Here you can see the benefit of having a gateway as a single entry point for client requests. It authenticates requests and forwards them to other services, which might in turn invoke other services.

Some well-known identity and access management systems are Okta, Keycloak, or Shiro.

In a distributed system, it is critical to assert the authenticity of requests in a consistent way across all services. In other words, once authenticated, how does one microservice communicate the identity of the requester to the other microservices?
The answer is through access tokens.



An access token securely stores information about a user and is then exchanged between services. Each service needs to make sure the token is valid and takes the user information out of it to verify that a user is authorized to perform the operation or not. Tokens can follow the JSON Web Token specification. Another possibility is to use cookies between service calls. You can see again the benefit of the gateway as it centralizes user interface calls and access token control.

SCALIBILITY IN MICROSERVICES

One major advantage of a microservice architecture is that you can scale each microservice independently depending on its needs. But being distributed, the system also has to be available, and for that, there are a few techniques to know.

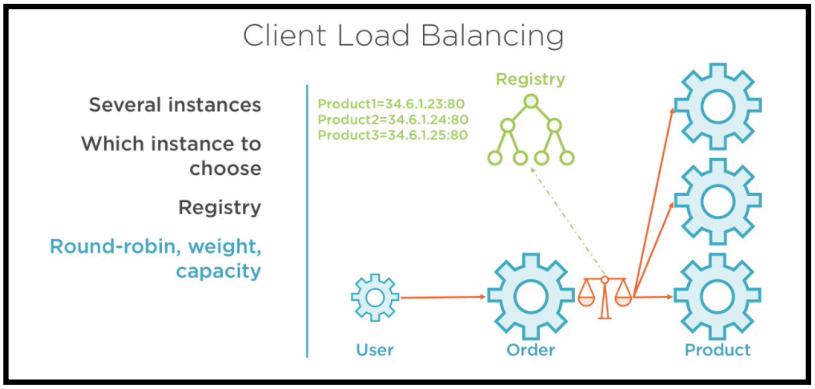
For Christmas, our application will surely get under load, because customer's will buy more products. Our user microservice might not be too affected by that, but we will definitely need to scale our purchase order and product microservices.

There are several ways of doing it actually :-

Vertical scaling means that you scale by adding more power to an existing machine.
For example, the machine for our purchase order microservice gets more CPU and RAM, and it works fine for this particular microservice.

Horizontal scaling means that you scale by adding more machines.
So our product microservice gets replicated into different machines. We talk about service replication or clustering.

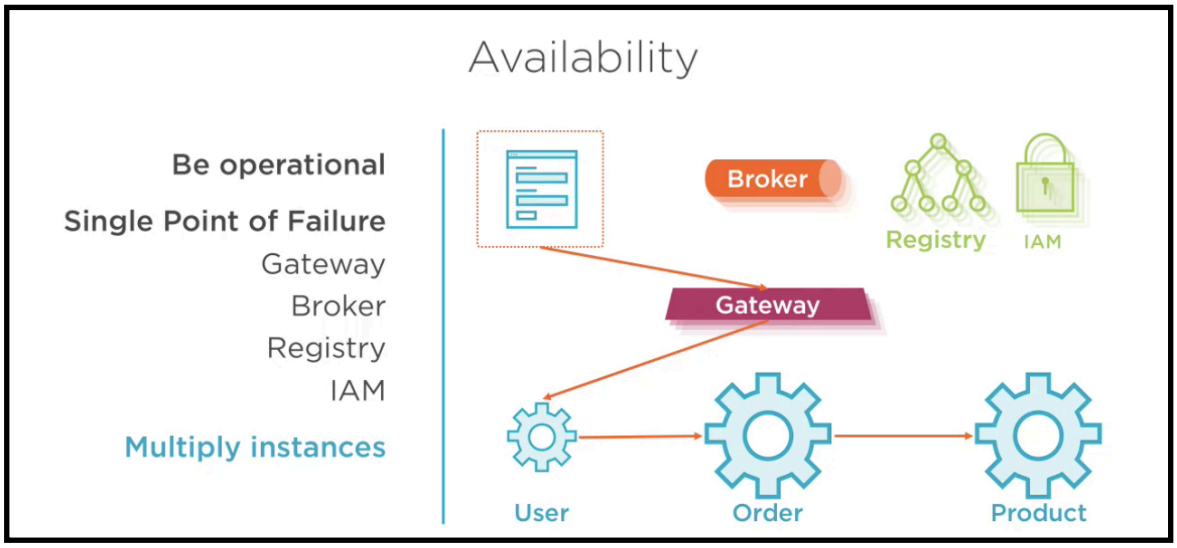
Services may scale up and down based upon certain predefined metrics.



When we scale horizontally, we end up with several instances of the same microservice located on different servers. When the purchase order invokes the product, which of the multiple available instances will it use? Remember that we have a registry, so all the instances of product are registered there. **So it's just a matter of having a client-load balancing on the purchase order.**

The load balancer will pick up from among the registered instances of the product microservice. It looks for the number of instances, say x, then the load balancer chooses from the among the candidate instances which one to route the request to. **The load balancer decides based on whatever criteria it likes, round-robin, or based on a weight, capacity of the service. Ribbon or Meraki are common client load-balancing tools.**

AVAILABILITY IN MICROSERVICES



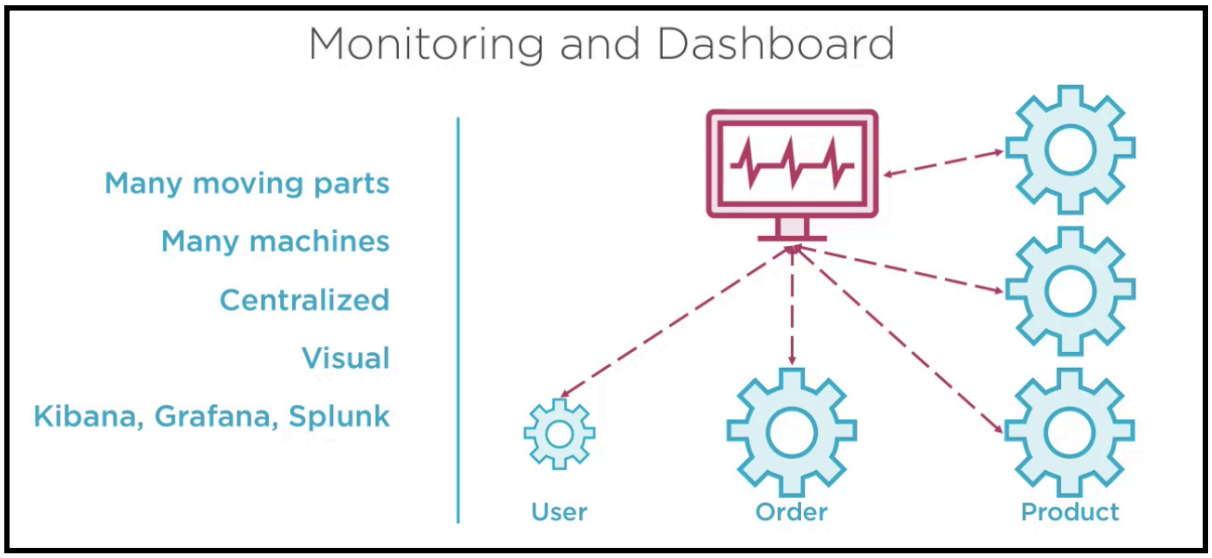
One thing is scaling up when Christmas arrives to handle the load, but the other thing is having a system that is available. **Availability means the probability for a system to be operational at a given time. Available systems report availability in terms of minutes or hours of downtime per year.** For example, a high available system is expected to be available 99. 999% of the time.

In our architecture, we have a few single points of failure, also called SPOF. A single point of failure is a part of a system that, if it fails, will stop the entire system from working.
For example, we have only one gateway, one message broker, one service registry, and one identity and access management system. For a distributed system to be continuously available, every request received by a non-falling node in the system must result in a response. This means that if the service registry is down, the user microservice will not be able to locate the purchase order microservices, and the entire system will be unavailable. **To fix that, all the single points of failure need to be scaled horizontally, so we have multiple instances. They also need to be clustered so they can keep in sync.**

MONITORING MY MICROSERVICES

One of the most important aspects of a distributed system is monitoring. This allows you to take proactive action if, for example, a service is consuming unexpected resources or not responding.

There are so many moving parts and so many machines involved in a microservice architecture that you quickly need to monitor what's happening. You need to rapidly view the running instances, seeing their failure and success rates, and spotting Button x.

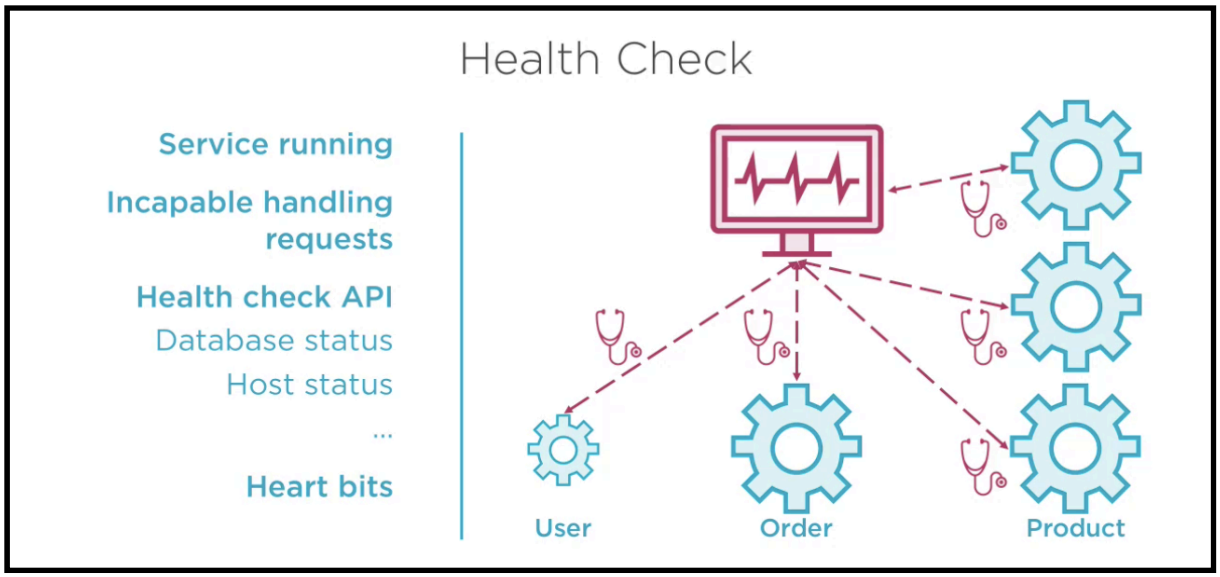


One key feature of monitoring is that it has to be centralized. Imagine having to log onto each machine, check the logs or the process manually. This is not possible when you have dozens, hundreds, and sometimes thousands of devices to monitor. And when the architecture is too complex, the information needs to be visual. **Monitoring tools need a dashboard where you can quickly visualize what's wrong. Monitoring dashboards such as Kibana, Grafana, or Splunk allow you to visualize all sorts of information.**

HEALTH CHECK

One very important monitoring information is health check. **Sometimes a microservice instance can be running, but incapable of handling requests. For example, it might have run out of database connections. So, How to detect that a running microservice is unable to handle requests? One way is to have a health check API on each microservice, that is a HTTP endpoint that returns the health of the service, and can be pinged by the centralized monitoring.**

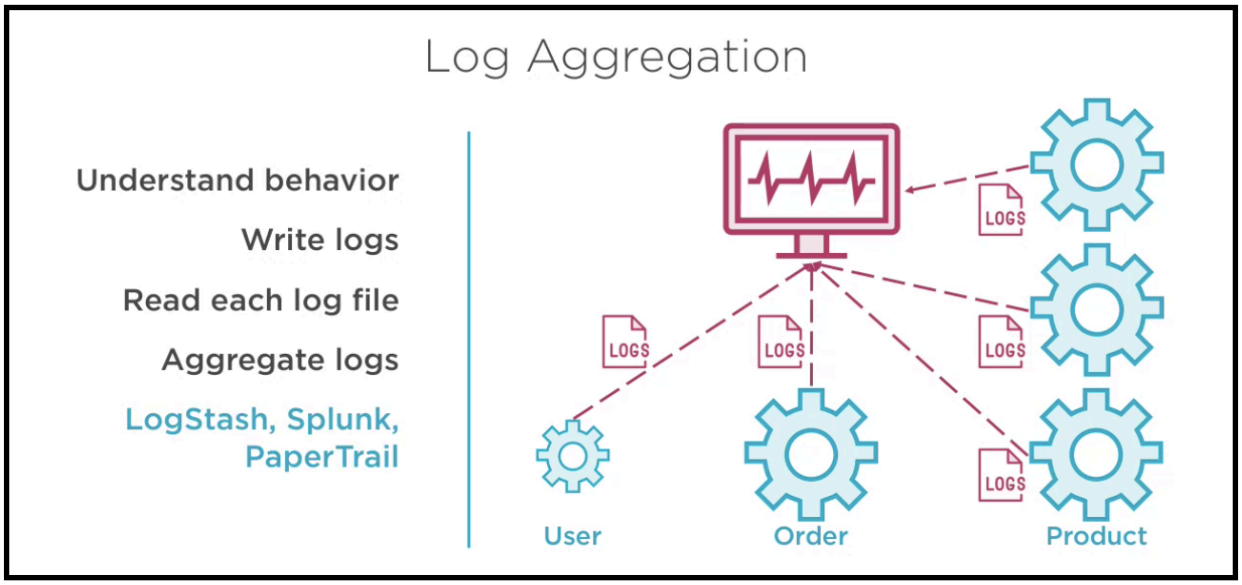
The health check API performs various checks such as status of the database, the status of the host, the disk space, the available memory, and so on. The centralized monitoring periodically invokes these endpoints to check the health of each service, therefore the health of the entire system. Think of it as heart beats.



LOG AGGREGATOR

Our application consists of multiple microservices that are running on multiple machines. Thanks to health check, we know that they are alive, but how can we understand the behavior of the application and troubleshoot problems? Well, each microservice writes information about what it is doing to a log file in a standardized format.

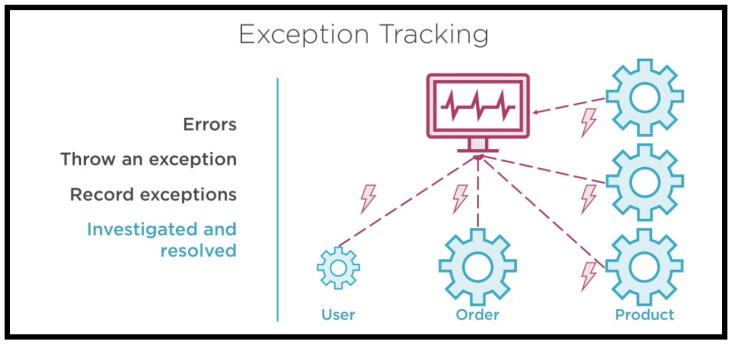
The log file contains errors, warnings, information, and debug messages. But because it's not efficient to read each log file of each microservice to understand what's going on, we use a log aggregator. It is a centralized logging service that aggregates log from each service instance. The administrators can then search and analyze the logs from the dashboards. There are a few log aggregators you can choose from such as **LogStash, Splunk, or PaperTrail.**



EXCEPTION TRACKING

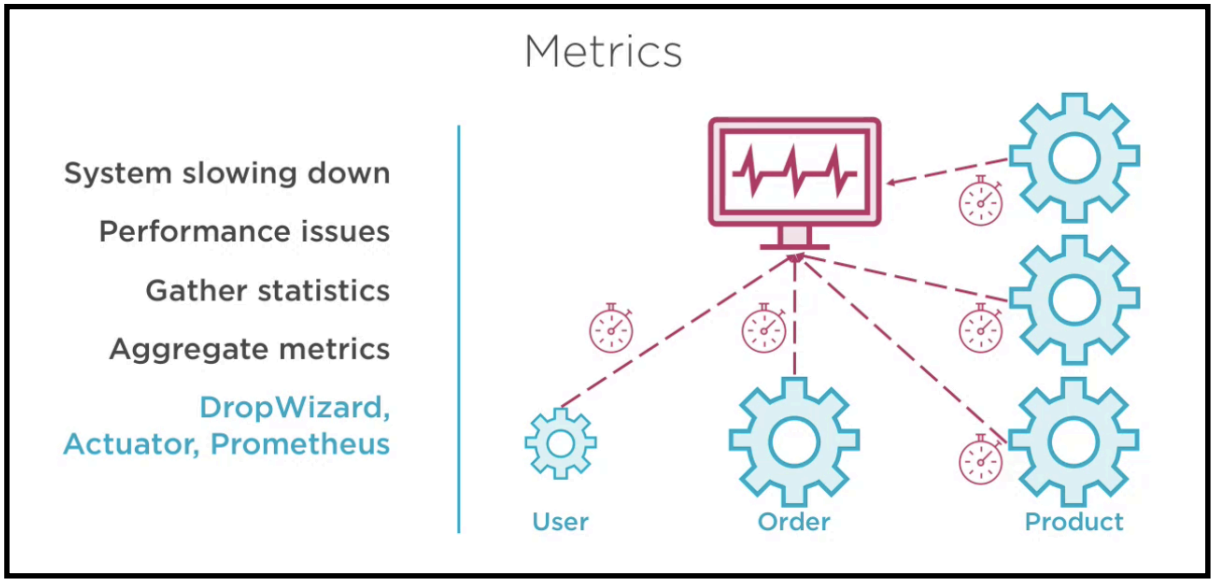
Sometimes errors occur when sending requests. When an error occurs, a microservice throws an exception, which contains an error message and a stack trace. This exception appears in the middle of the log file with all sorts of information and can get difficult to find. That's why exception should be recorded in a centralized exception tracking system, so they can be investigated and resolved by developers.

This is crucial in understanding our system errors, fix them, and hopefully see the evolution of errors declining through time.



METRICS

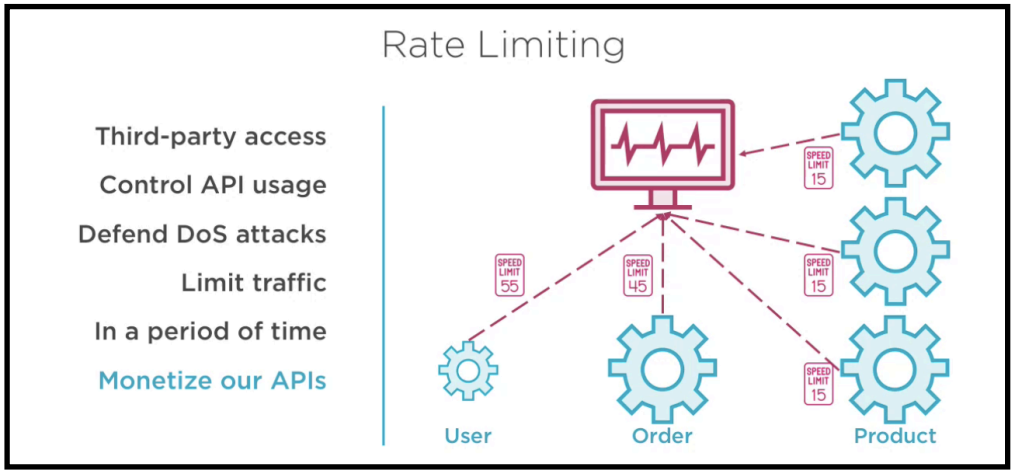
We have health checks, logs, but how do we know that the system is slowing down, or how do we spot performance issues. We need to instrument our microservices to gather statistics about individual operations. How long does it take to answer an HTTP request, how long creating a purchase order takes, how long is the database access? We then have to aggregate these metrics in a centralized metric service, which provides reporting and alerting. There are several tools you can use such as **DropWizard, Spring Actuator, or Prometheus.**



RATE LIMITING

In a microservice architecture, it is important to understand the behavior of users. It is useful to know what actions a user has recently performed, successful logins, logouts, which microservice has been solicited for this particular user interaction, which pages have been visited, how many products were browsed, how many products were bought?

The user activity has to be recorded in a centralized system, so we can understand better the parts of our system that are heavily used by our users, and therefore optimize or scale them if needed. But it's not only users who can access your microservices. All the microservers can, and sometimes it could be third-party microservices invoking your APIs. That's why you should control your API usage by setting up rate limiting.

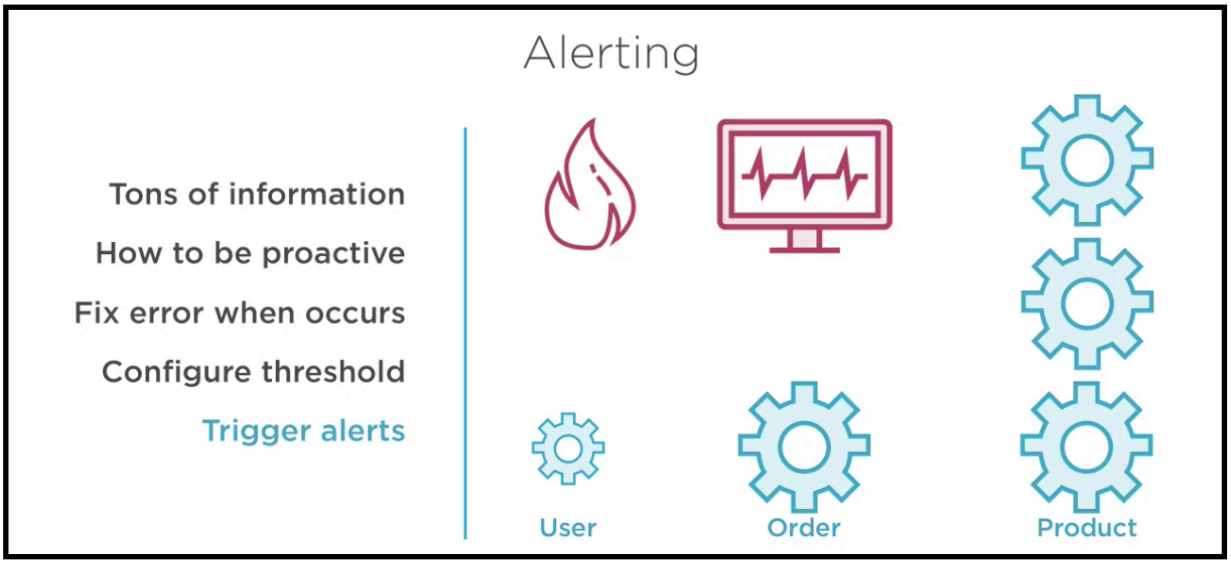


Rate limiting is not new, and not just related to microservices. In fact, **the main reason to implement rate limiting was to defend applications against DoS attacks, denial of service.** It was a way to apply policies to limit traffic coming from specific sources, specific customers, API addresses, and so forth.

Nowadays with microservices and exposed APIs, **rate limiting became a way to manage our APIs.** **We can rate limit how many HTTP requests a client can make in a given period of time. It's also a way to monetize our APIs.** For example, a specific customer has the right to access our product API only 100 times per day. If the customer wants more accesses, he or she must pay more for that.

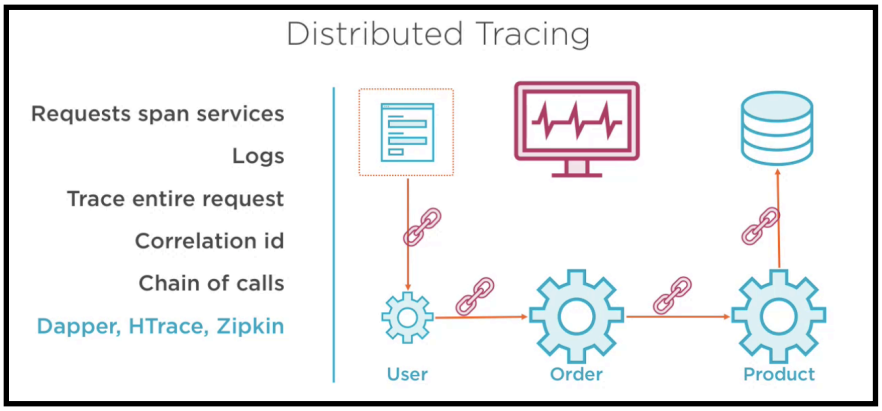
ALERTING

Now that we have tons of information about the health of the system, exception, metrics & audits, how to be proactive? We as administrators can't spend our day looking at the centralized monitoring system. Instead, we need to configure alerts that are triggered when certain messages appear in the logs, so they can forget about non-relevant information. Therefore, when a certain threshold is reached, the monitoring system should generate an alert.



DISTRIBUTED TRACING

Now with monitoring all over and alerts, we could think that we are done, but there is something specific to distributed systems that has to be taken into account, distributed tracing.



Requests often span multiple services, for example, a user hits the user microservice, gets logged in, creates a shopping cart, selects a few products from the database, and generates a purchase order. So far we have logged every event separately and can trace the entire request going from the user interface, to the database, through three microservices.

With distributed tracing, we instrument services with code that assigns each request a unique correlation identifier that is then passed between services. This creates a chain of calls. It provides useful insights, for example, the whole of all response time of an entire invocation, and the sources of latency. Whereas logs and metrics give us insights of an individual operation. There are a few tracing systems you can use in your microservice architecture such as **Dapper, HTrace, or Zipkin.**

DEPLOYING MY MICROSERVICES

Each of our microservices must be independently deployable and scalable. Microservices also need to be isolated from one other, that's why deployment is important. We've been digging into our microservice architecture from databases, to distributed services, gateways, service registries, and so on, but at the end of the day, all we've seen needs to get deployed either on physical or virtual servers, and this could be on-premise or in the cloud.

With **physical servers,** each server has its own memory, network, processing, and storage.

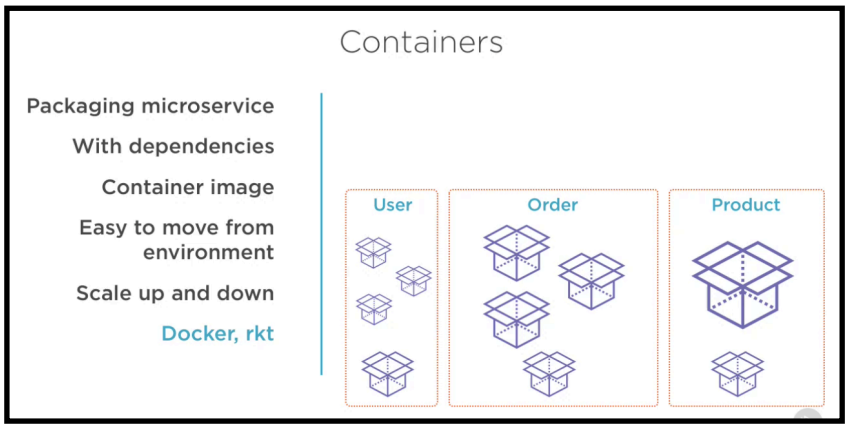
With a **virtual infrastructure**, you have the same physical server with all these physical resources of course, but you get virtual CPU, virtual memory, virtual network interfaces, so you can constrain the resources consumed by a service.

Then depending if you need physical or virtual servers, you can deploy each single microservice instance on its own host. This way, each microservice instance is isolated from other another. **There is no possibility of conflicting resources.** But you can also run several instances of different services on a single host. This is a more efficient resource utilization, but you need to make sure the microservices do not end up conflicting. And remember that you can have the same segregation with virtual servers too.

CONTAINERS

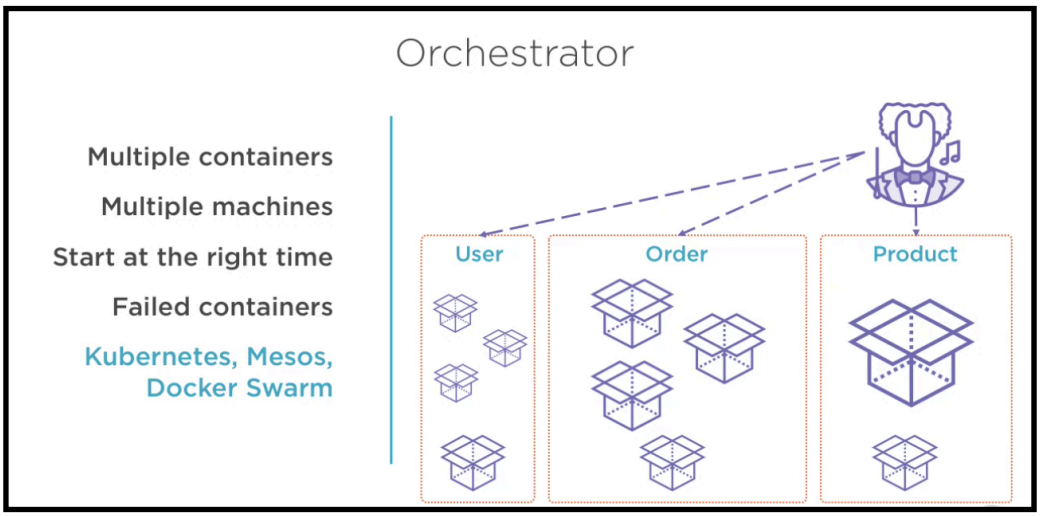
In above scenario, our architecture started to get a bit crowded, and we could have got lost in packaging a microservice with its right dependencies such as operating system and file system.

One way to simplify the deployment is to package each microservice as a container image, and deploy it as a container. It's the container's role to encapsulate the details of the technology used to build a service. It can also impose limits on the CPU and memory usage. By providing an image that contains all the microservice dependencies, it becomes easy to move it from development to testing, and finally to production. Being consistent, also services can now be started and stopped in exactly the same way. It also becomes straightforward to scale up and down a service by changing the number of container instances.



Docker is the de facto container management system, but there is also **Rocket**.

ORCHESTRATOR



Containers are a way of packaging our microservices, but we need to orchestrate all these containers by running multiple containers across multiple machines. We also need to start the right containers at the right time, figure out how they can talk to each other, handle storage consideration, and deal with failed containers or hardware.

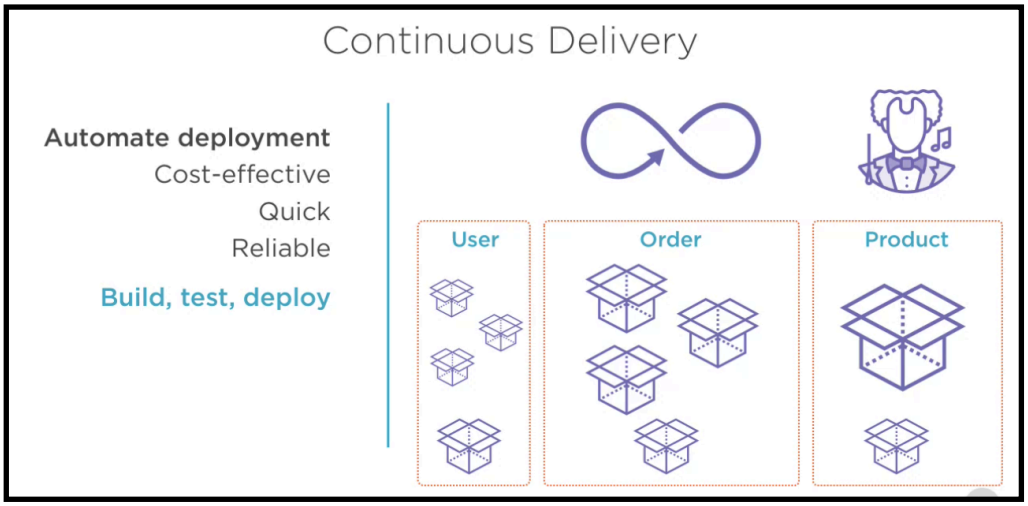
Doing all this manually would be a nightmare. Luckily we have orchestrators such as Kubernetes, Docker Swarm, Mesos, and Marathon to automate these tasks, reducing operational burden.

CONTINUOUS DELIVERY

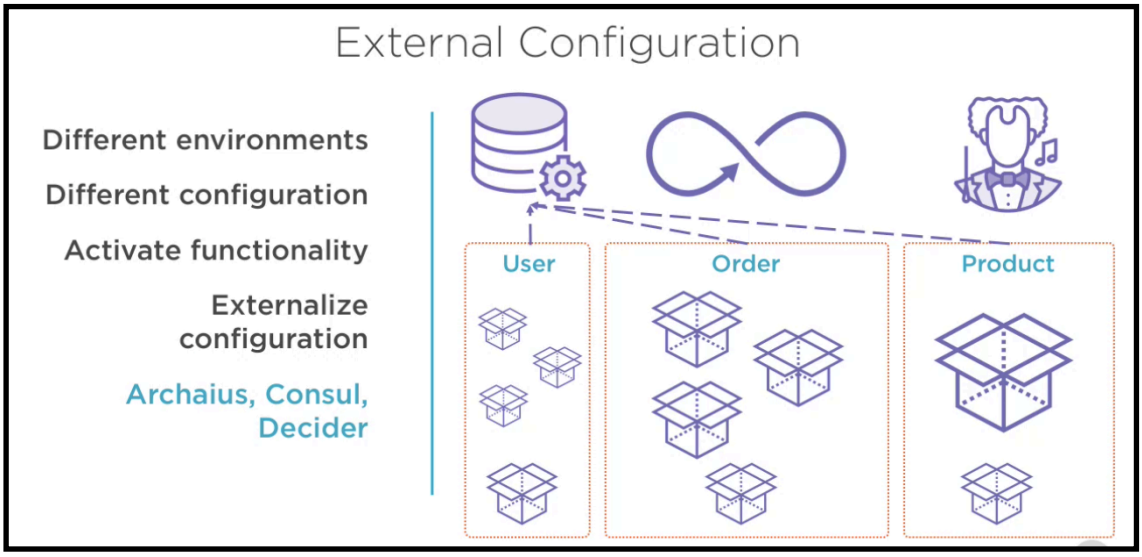
Even if every piece of architecture is a container that can be orchestrated, we want this entire deployment to be automated. Deploying our ecommerce application should be as **cost effective** and **quick** as possible, but most important, **reliable.** That's the aim of continuous delivery.

Continuous delivery is meant to get all the code from our version control, build each microservice, pass all the unit tests, acceptance test, performance test, release the code, package it into a container, and trigger the orchestrator so it can deploy all the pieces of our architecture.

The more frequently continuous delivery run, the more robust a deployment into production can be. There are a few continuous delivery tools such as **Jenkins, Asgard, or Aminator.**



EXTERNAL CONFIGURATION



Containers, orchestrators, and continuous delivery are not just for production, they have to be used in all our environments. Depending on the complexity or the architecture, we can have a development, test, QA, or staging environment. These different environments are really needed in a microservice architecture.

It reaches a point when you need to integrate all the microservices with a gateway, the messaging broker, the service registry, and so on, and test the entire application.

We also need to take microservices versioning into account. This means that you can test a 1. 2 version in the staging environment, while running a 1. 0 version in production.

So now we have several microservices running on different environments. **Different environments usually mean different runtime configuration for the same microservice.**

For example, we will increase the log level in development, but not in production, or the database location and credentials will also be different between environments.

Another example is the ability to activate a new functionality at runtime, and leave the team to access the new functionality's impact on the rest of the systems with the ability to deactivate it. That means that **when we develop our microservices, we need to take into account the possibility to externalize all its configuration into a centralized configuration store.**
It can be a database, but other tools can be used such as **Archaius, Consul, or Decider.**

CHALLENGES

Enterprises today live in a globally-competitive world. They are distributed over contents, do business 24/7 over the internet, across different countries, deal with different currencies and time zones. They need applications to fulfill their business needs, which are getting more and more complex.

These applications need to store and process a huge amount of data, be mobile, use geo localization, be valuable, scalable, secure, communicate with external systems, and aggregate online services. This is what companies are trying to achieve while minimizing costs, and using open, robust, and agile technologies. Most companies have to combine these complex new challenges while maintaining their existing enterprise information systems.

In this hyper-competitive international market, all companies have become software companies, and time-to-market is a differentiator. Organizations look for small batches of work that they can move from concept to production quickly. And this is where microservices can help.

It's a common misstep to start down the microservice path for its own sake without thinking about the specific benefits you are targeting.

So if you are considering adopting a microservice architecture for your organization, you should ask yourself a few questions -
Could you go beyond agile and dev-ops practices?
How effective the existing architecture is in turns of modularity and replaceability?
Are there opportunities to improve?
What are the pain points of your current architecture?
What is the measure of success?
When you move to a new technology, you need to measure that you have succeeded.

BUSINESS CONCERNS

Business implication it could have in your organization :-

Each microservice is generally built by a full-stack team, which reduces potential communication difficulties between disbarred teams.

A smaller team is easier to manage, therefore any changes are easy to make and faster to happen.

Less dependencies between teams result in faster code to production.

Remember that software estimation is difficult, **the smaller the system is, the easier for the team is to estimate the evolution and roadmap of the system, ie. easier s/w estimation.**

If one team is responsible for the entire product, that means you really should embrace agile and devops practices, because you will have users, developers, designers, and operational team all working together.

If two microservices talk to each other, that means two separate teams need to synchronize and will occasionally need a central management.

When building a new team or getting people on board an existing one, you need to think of recruitment :-

Microservices are **trendy**, so **people will get motivated to come and work for you.**

You will be able to **easily mix experience and junior developers** as the team is small, and interactions are easy. This will create a **common knowledge** shared by your team that will **collectively learn about the product they are building.**

Training :-

One benefit of microservices is that you can pick up the appropriate technological stack and develop your product. The other benefit is that being small, you can **embrace new technologies faster.**

But there are **so many technologies, so many programming languages, so many data stores** that you'll have to constantly train your team. Also, if each microservice team has its own technology, moving people around teams might get difficult, and **more training will be required.**

Standard :-

When you pick up a technology stack, you would hope it is standard as possible, so you can rely on it for future releases. But **there is no such a thing as a standard technological stack for microservices.**

Your choices will more or less rely on some sort of standards such as HTTP or SQL, but not on a standard single stack.

There is no one single framework or product that will make everything work.

It's an **integration of several technologies, languages, frameworks, and devices.** And because of that, **you won't have a single point of support when you need help.**

Open-Source :-

Most tools, frameworks, library, or languages are open source. **Being so broad, microservices benefit from very dynamic open-source communities.** Back in the days, software used to be close source, and you would not know what was going inside a technology. Today, open communities are flourished throughout the planet, **giving you access to all the needed code and documentation.**

Cost :-

One big advantage with microservices is that you can **start small, start cheap, with only a few microservices, and then grow up when needed.** The first iterations of your product shouldn't cost you much.

Small team, open-source products, cheap deployment on the cloud, all that makes it possible to start with a small budget, and then invest more when your business is running.

Time to Market :-

Microservices will get you **faster to the market,** and you will **start earning revenues faster** too.

Small batches of work demand fewer people to finish, are easier to iterate on, and **can be moved to production faster.**

TECHNICAL CONCERNS

Design :-

One of the biggest challenges in designing microservices is deciding how to partition the system into microservices. Our ecommerce application was split into three sub modules, but why not four or five? We could've created an extra microservice to manage invoices, and another one to deal with shipping products.

How small should you design your microservices?

Designing microservices is very much an art, as it depends on the complexity of your domain. Fortunately there are a number of strategies that can help around domain-driven design. Once the designing phase is finished, you can make the best architectural choices for your own microservice.

Technical Diversity :-

Choose the most appropriate programming languages and frameworks. make sure your microservice has the best performance by adopting the right database, the right platform. In other words, pick up the right tool for the right job.

Microservices are technology and language agnostic, so it is quite possible for a single organization to utilize multiple runtime platforms in a microservice architecture.

User Interface :-

Choose the appropriate web framework to create your graphical components.

Again being small, you shouldn't have thousands of components, but instead **have a small set of very well-designed components**, so your users have a **great browsing experience.**

You can also **concentrate on making sure your user interface is responsive and fits well on all devices.**

Need to integrate all the UI together.

But if at the end of the day your user interface is the meeting point of several microservices components, then **aggregating several front ends to make the user feel it's one can be tricky.**

Distributed:-

In a microservices architecture, every microservice is a separate application with its own data storage and communicating over a network. This creates an environment that is highly distributed, and with that come challenges.

In fact, these challenges are known for quite awhile, and I recommend reading about the network fallacies. Its author ironically said that the network is reliable, latency is 0, bandwidth is infinite, the network is secure, topology doesn't change, there's one administrator, and transport cost is 0. Of course this is a lie.

So it is critical that an organization is prepared to address the complexity implied by moving to a distributed system world. Network failure will occur no matter how much and how hard you test. The key concern is not how to avoid failure, but how to deal with failure.

It is important for services to automatically take corrective actions to ensure user experience is not impacted. That's why implementing circuit breaker and scalability is important.

Data Store :-

Once you model your domain into subdomains, you end up **using a database per subdomain.** This is important, as it **ensures that the services are loosely coupled.** Changes to one service database does not impact any other services. **Each service can use the type of database that is best suited to its needs,** relational, NoSQL, or timed series database.

But remember that if you need to synchronize data across microservices, then you need to implement a few patterns such as capture data change or event sourcing to get eventual consistency.

You will need to leave immediate consistency, as distributed transaction with two-face commits are best to be avoided. **(Avoid distributed transactions)**

Performance :-

Choosing the right database, the right language, the right tools for the right job will give you performance benefits.

You will be able to optimize each microservice, and take it to different benchmarks and performance tuning. But of course when integrating your microservices into the entire distributed system, you will hit the network that could have a performance impact at a larger scale.

Security :-

If security is the system's ability to resist unauthorized attempts while still providing service to legitimate users, then we know **security is difficult.**

If you multiply the number of microservices, you potentially multiply the number of flaws and open doors. Each microservice has to validate the access tokens coming in, passing them toward the microservices dealing with time out and token renewal.

In fact, security in microservices is so important that we see the new term **DevSecOps** immerging. **This is where your team is made of developers, operationals, and security hackers.**

Testing :-

If you stick to your own microservice, then testing is easy. There is less code in a microservice than in a monolith. Less code means less code to test, means less test.

With mocking mechanism, it is easy to test a microservice in isolation without worrying about distributed system.

But when it comes to integrate your microservice and test it in the entire system, things become harder. **(Difficult for integration testing)**

Complex Test Environment:-
Creating a test environment that is similar to your production can also be complex. That's why some companies test specific paths of their system straight into production. This can sound a bit hard, but if you spend a lot of time and money creating a test environment, it could be worth testing in production.

You can even look into **chaos testing.** When your code is running on enough machines and reaches enough complexities, errors are going to happen. Since failure is unavoidable, why not creating them on purpose. So **chaos testing is when you generate failure on purpose.**

Maintainability :-

Less code to write, less code to test, less code to maintain, but more important, less code to understand.

A microservice focuses on the subdomain of your model, so it is easier for the entire team to understand.

Even if your microservice has been running in production for a long time without anybody touching it, if a modification has to be done, it could be done in a timely manner.

Extensibility :-

One beauty of a microservice architecture is that it is extensible by design -

You need a new feature, just implement it as a separate microservice, instead of implementing it into an existing one.

You need to access a third-party service hosted by a partner in the cloud? Just create a microservice to bridge into it.

You want your partners to have access to parts of your system, create special API contracts.

PRODUCTION CONCERNS

Continuous Integration / Delivery

There is this concept called **DURS,** which means **deploy, update, replace, scale.**

In a microservice architecture, we need to have independent DURS. This means that each microservice can be independently deployed, independently updated, replaced, and scaled.

Continuous integration and continuous deployment are **very important in order for microservices-based applications to succeed.**

These practices are **required so that errors are identified earlier via well-automated testing and deployment pipelines.**

This needs to be highly automated, and is **one of the most difficult tasks that your devops team will have to manage well.**

Portability

Your continuous integration system has to build container images of your microservices, so that the same image can then be used across every step of the deployment process.

A container image is a snapshot of a microservice with all the parts it needs, such as libraries, network, or volume resources, and other dependencies. All this is shipped as one package, totally isolated from other containers.

By doing so, you can be assured that the application is portable and will run on numerous platforms. An orchestrator will help you in deploying and scaling your containers across your infrastructure.

Infrastructure

With microservices, in terms of infrastructure, **you usually end up with hybrid infrastructure.** Hybrid infrastructure combines traditional IT, private cloud, and public clouds.

You can mix your infrastructure by deploying some microservices into your traditional IT with your physical virtual servers and dedicated storage on the same network. Other microservices can be deployed in your own private cloud, but also in different public cloud providers in several data centers spread out throughout the world. All major cloud providers have embraced containerization, so deploying your container images would be easy. **You can enable your right mix of on-premise, on-cloud to handle the workload needed by your application.**

Scalability

Scalability is the ability for your application to gracefully meet the demand of stress caused by increased usage, in short, ensuring your application doesn't slow down when pounded by more users than you originally anticipated.

When used correctly, hybrid hosting can provide the scalability you need. You can host your application on-premise in your own data center, and then scale out on a public cloud.

Availability

With scalability comes availability. **Availability is the characteristic of a system to ensure a level of operational performance. This is also called uptime.**

Availability is a double-edged sword. Keeping a monolith available is easy, as there is only one piece in the system, but if it fails, the entire application is not available anymore. With a microservice architecture, if one part of the system is not available, the rest is, and that's one interest of microservices. **It is mostly available by design, but this comes at a cost.**

If one piece of the system is not available, each microservices depending on it must enable graceful degradation, so every microservice needs to be designed so it can handle other pieces not being available.

And also remember that service registry, gateway, or identity, and access management are critical system components that must be highly available. In production, it's another **operational complexity of managing a system comprised of many different service types that have to be constantly available.**

Monitoring

With so many moving parts in a microservice architecture, **monitoring is not an option. You must constantly monitor your system.**

The more microservices, the more service registries, the more databases, the more message brokers, the more logs you'll have to aggregate, the more heart beats you'll have to control, the more metrics you'll have to gather, so you can create useful alerts to detect when the system starts to slow down. **Handling a large volume of logs and other monitoring information requires a substantial infrastructure that needs to be highly available.**

Of course, monitoring your system is mandatory, but you also **need to monitor external third-party API calls and rate limit them, as well as dealing with possible security issues and fraud.**

Monitoring large distributed systems is complex.