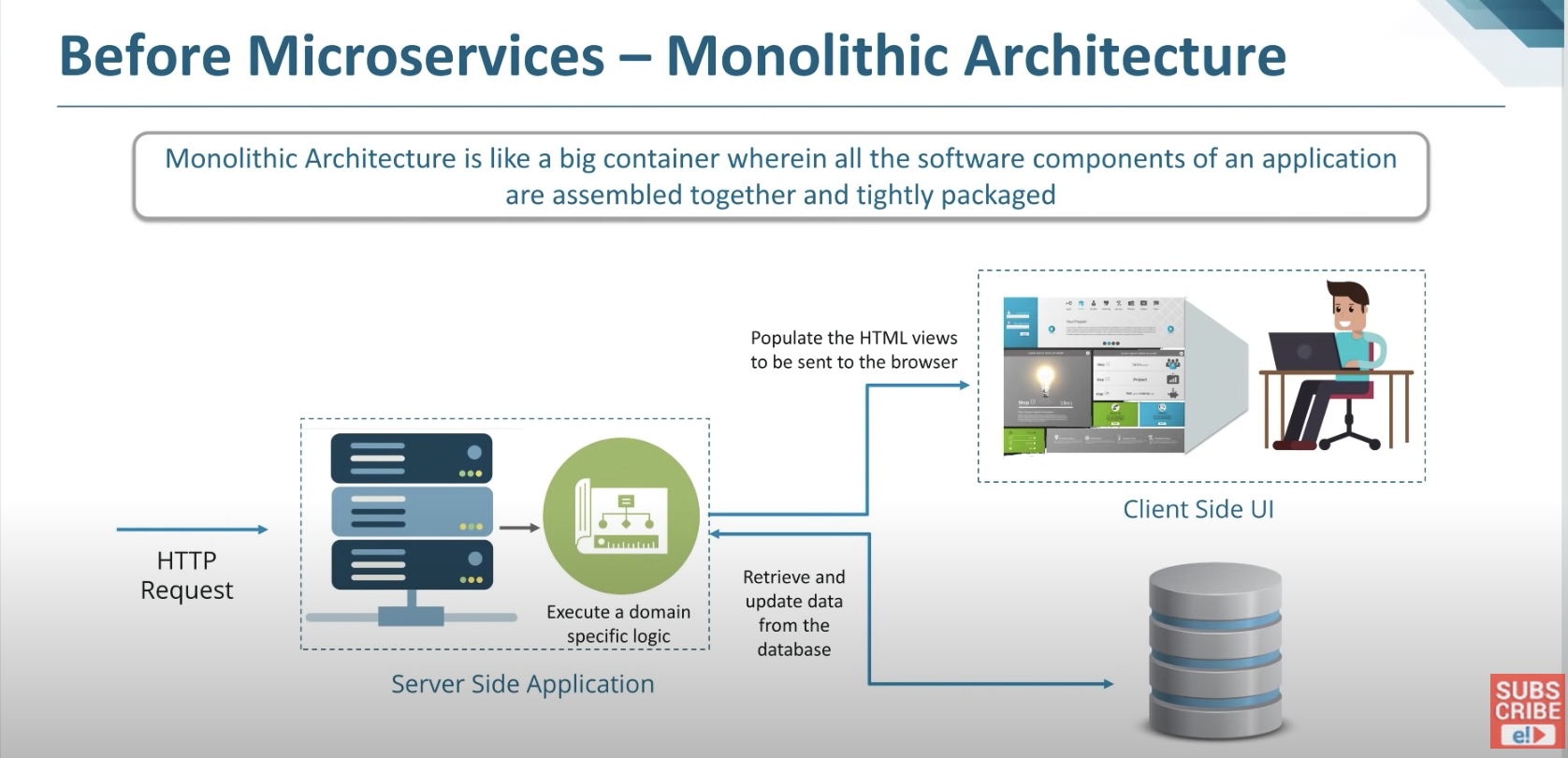
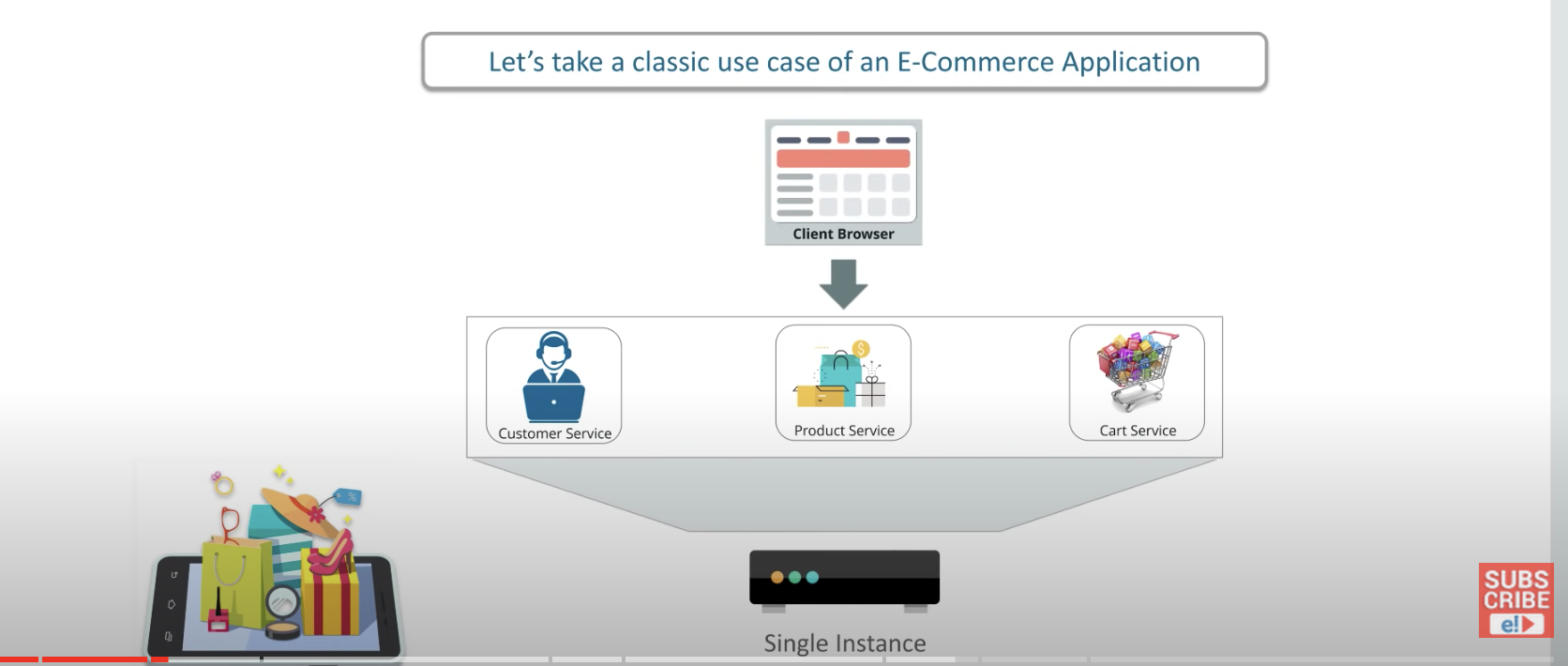
Microservices: <https://www.youtube.com/watch?v=gfWr2_H39N0> , <https://middleware.io/blog/microservices-architecture/>

Monolithic: in this architecture all the software components/entities of an application is inside a single component and all are components are interconnected and inter dependent.

The entire software/application is as a single piece. All the components are tightly coupled, i.e., all the components are its dependent components are required to compile and run the whole application.

example as e-commerce application with services line consumer service, product service, order service, cart service and all these are deployed as a single container.





Advantages

development is very easy

deployment is simple, as it requires only one executable file for deployment

simple to scale as we run multiple copies for same executable

Drawbacks.

large and complex application, as all components are present in a single container it is complex to understand and modify and as a result development slowdowns and quality of code changes.

Slow development and large teams across geo locations.

when we change only the cart service, we need to deploy the whole application. chances that the breaking the other working services

blocks continuous development as change in 1 service can impact other parts of applications and requires deploy of complete executable

if any one of service fails, the complete application goes down.(unavailability of application)

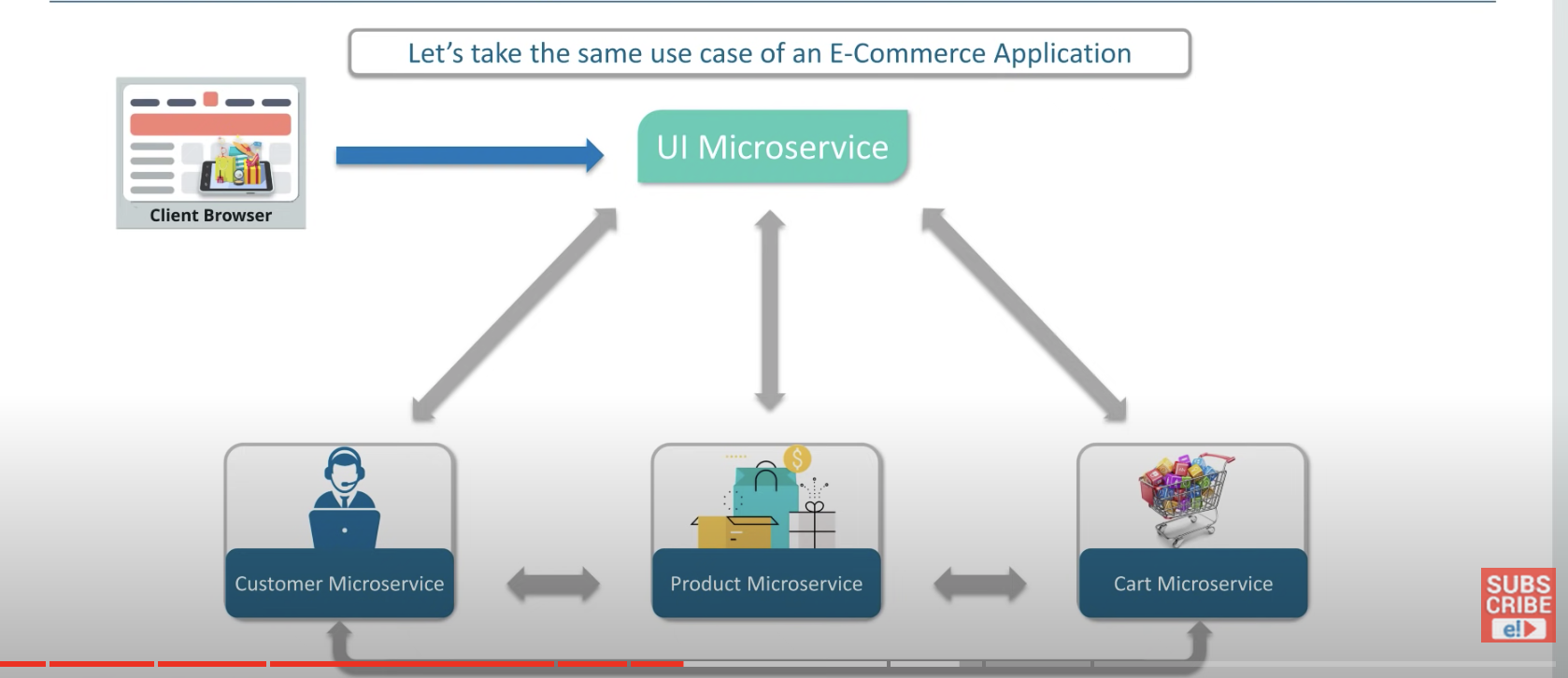
inflexible -- difficult to move to new technology or new framework as it requires the complete application change.

Microservices: it is a architecture in which the complete large application is divided into multiple independent services and which all together acts as one system.

a single business is divided into multiple domain sub services and all services does not share data structures but they all communicate via apis.

here each single microservice can focus on only one business domain or functionality.

same e-commerce example, now here each service (consumer, product, order and cart) are separated into different modules and all are communicated via apis and communications b/w the services are stateless where each pair of request/response is an independent transaction. Here each microservice has its own instances and data model and its own data.

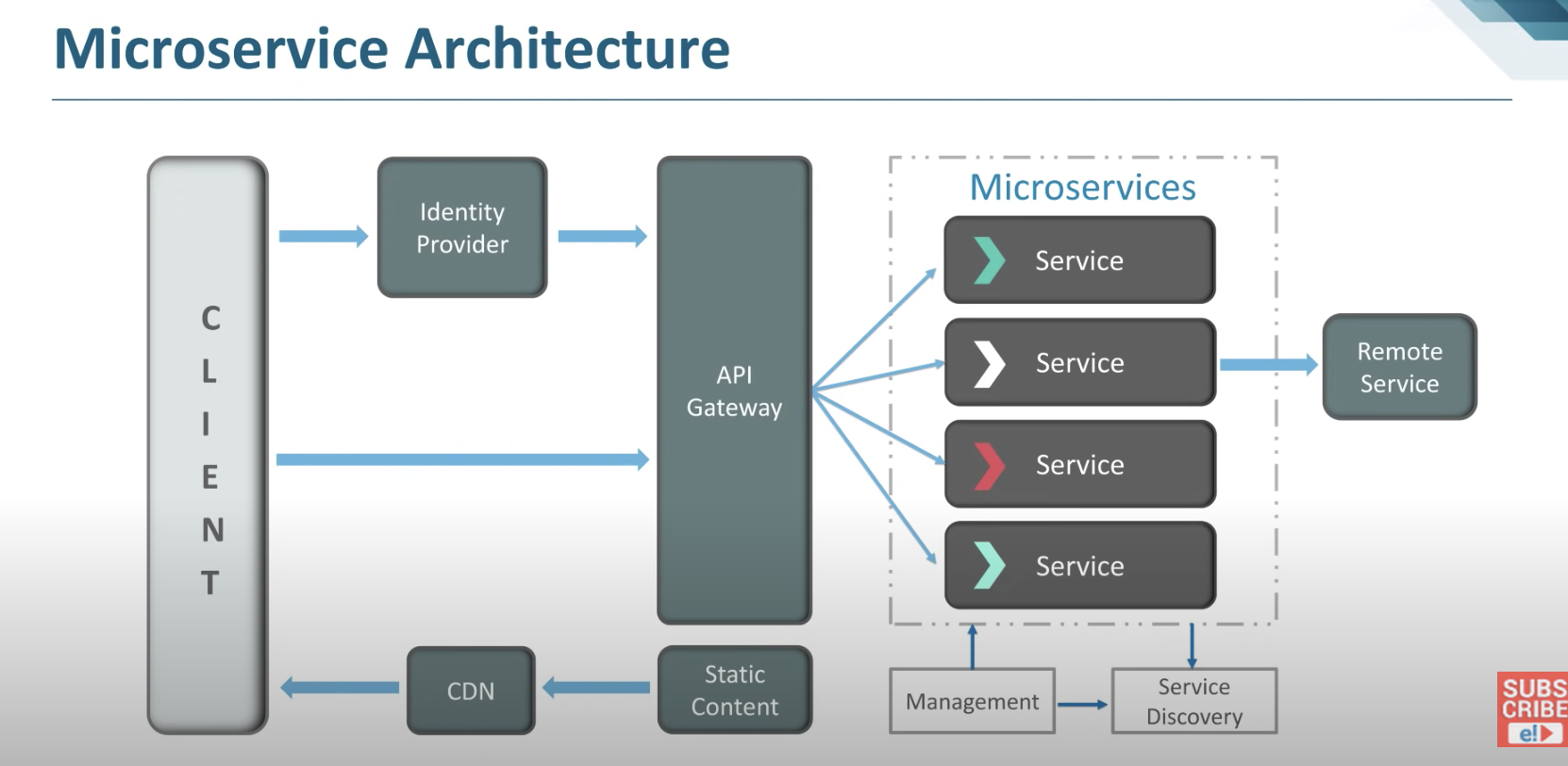


each microservice has its own code base and can be deployed independently without any dependency on other services in an application

internal implementation of services are hidden from other services in an application.

all services don't need to share same set of libraries or same technologies or frameworks.

Microservice architecture:



* **Clients**: Different users send requests from various devices.
* **Identity Provider**: Validate a user's or client's identity and issue security tokens.
* **API Gateway:** Handles the requests from clients.
* **Static Content:** Contains all of the system's content.
* **Management:**Services are balanced on nodes and failures are identified.
* **Service Discovery:** A guide to discovering the routes of communication between microservices.
* **Content Delivery Network:** Includes distributed network of proxy servers and their data centers.
* **Remote Service:** Provides remote access to data or information that resides on networked computers and devices.

Management component:

This is responsible for placing the services on the nodes. Identifying failures, rebalancing services across the nodes.

Service discovery component:

Task of this component is maintain the list of services and which nodes they are located on.

Service discovery is the automatic detection of devices and offered services over a network.

Let’s imagine several microservices that make up a more or less complex application. These will communicate with each other somehow (e.g. API Rest, gRPC).

A microservices-based application typically runs in virtualized or containerized environments. The number of instances of a service and its locations changes dynamically. We need to know where these instances are and their names to allow requests to arrive at the target microservice. This is where tactics such as Service Discovery come into play.

The Service Discovery mechanism helps us know where each instance is located. In this way, a Service Discovery component acts as a registry in which the addresses of all instances are tracked. The instances have dynamically assigned network paths. Consequently, if a client wants to make a request to a service, it must use a Service Discovery mechanism

**The Need for Service Discovery**

A microservice needs to know the location (IP address and port) of every service it communicates with. If we don’t employ a Service Discovery mechanism, service locations become coupled, leading to a system that’s difficult to maintain. We could wire the locations or inject them via configuration in a traditional application, but it isn’t recommended in a modern cloud-based application of this kind.

Dynamically determining the location of an application service isn’t a trivial matter. Things become more complicated when we consider an environment where we’re constantly destroying and distributing new instances of services. This may well be the case for a cloud-based application that’s continuously changing due to horizontal autoscaling to meet peak loads, or the release of a new version. Hence, the need for a Service Discovery mechanism.

The service discovery component is made up of three parts:

* A service provider that originates service instances over a network;
* A service registry, which acts as a database that stores the location of available service instances; and
* A service consumer, which retrieves the location of a service instance from the registry, and then communicates with that instance.

Service discovery also consists of two major [discovery patterns](https://www.techtarget.com/searchapparchitecture/tip/Fundamental-patterns-for-service-discovery-in-microservices):

* A client-side discovery pattern searches the service registry to locate a service provider, selects an appropriate and available service instance using a load balancing algorithm, and then makes a request.
* In a server-side discovery pattern, the router searches the service registry and, once the applicable service instance is found, forwards the request accordingly.

Data residing in the service registry should always be current, so that related services can find their related service [instances at runtime](https://www.techtarget.com/searchapparchitecture/tip/A-quick-rundown-of-multi-runtime-microservices-architecture). If the service registry is down, it will hinder all the services, so enterprises typically use a distributed database, such as Apache ZooKeeper, to avoid regular failures

**How Does Service Discovery Works?**

Service Discovery handles things in two parts. First, it provides a mechanism for an instance to register and say, “I’m here!” Second, it provides a way to find the service once it has registered.

Let’s clarify the concept we’ve discussed so far with an example: a Service Consumer and a Service Provider (a service exposing REST API). The Service Consumer needs the Service Provider to read and write data.

API Gateway:

this is the entry point for the client, The client cannot call directly all the services, this api gateway navigates the client request to appropriate service on backend. When client request multiple responses from multiple services, then this api gateway might aggregate responses from multiple services and return the aggregated response.

Service mesh

In a microservices architecture, the service mesh creates a dynamic messaging layer to facilitate communication. It abstracts the communication layer, which means developers don't have to code in inter-process communication when they create the application.

Service mesh tooling typically uses a sidecar pattern, which creates a proxy container that sits beside the containers that have either a single microservice instance or a collection of services. The sidecar routes traffic to and from the container, and directs communication with other sidecar proxies to maintain service connections.

Two of today's most popular service mesh options are Istio, a project that Google launched alongside IBM and Lyft, and Linkerd, a project under the Cloud Native Computing Foundation. Both Istio and Linkerd are tied to Kubernetes, though they feature notable differences in areas such as support for non-container environments and traffic control capabilities.

Advantages

Loosely coupled as each service is independent and makes development is easy and can be deployed easily without disturbing other services.

Programming language and framework independent, means that the tech and framework need not to be same across all services in an application.

Small focused, means that less effort to re-write the code.

Bounded context, means each microservice does not need to understand the internal implementation of other services in an application

Independent development and independent deployment. Small team

Fault isolation meaning, if a service goes down, the whole application won’t go down.

Mixed tech stack across services in an application

Services can be scaled independently where we can scale single application if needed.

Communicating b/w different microservices can be done via Rest calls, Kafka etc.

Via rest, we use spring boot already provided RestTemplate for this communication and return response from other service.

For kafka we use KafkaTemplate to publish a message to topic and the other service subscribes to that topic and consumes and process the data.

**Spring Cloud**: In Microservices, the Spring cloud is a system that integrates with external systems. This is a short-lived framework designed to build applications quickly. It contributes significantly to microservice architecture due to its association with finite amounts of data processing.

### What issues are generally solved by spring clouds?

The following problems can be solved with spring cloud:

* **Complicated issues caused by distributed systems:** This includes network issues, latency problems, bandwidth problems, and security issues.
* **Service Discovery issues:** Service discovery allows processes and services to communicate and locate each other within a cluster.
* **Redundancy issues:** Distributed systems can often have redundancy issues.
* **Load balancing issues:**Optimize the distribution of workloads among multiple computing resources, including computer clusters, central processing units, and network links.
* **Reduces performance issues:** Reduces performance issues caused by various operational overheads.

### Explain three types of Tests for Microservices?

In Microservice architecture tests are divided into three broad categories:

* At the bottom level test, we can perform a general test like performance and unit tests. These kinds of tests are entirely automated.
* At the middle level, we can perform exploratory tests like the stress tests and usability tests.
* At the top level, we can conduct acceptance tests which are mostly fewer in numbers. It also helps stakeholders to know about different software features.

<https://www.guru99.com/microservices-interview-questions.html>