**Migrating a Monolith to micro service and its challenges**

Let us say we have an application a huge monolith a single code base multiple teams working on it.

It contains all the features of the entire application let us say we split the monolith into a set of micro services talking to each other and because we do that, we get lot of benefits.

**Benefits:**

**Faster delivery:** Small size encourages frequent releases with less risk.

since each of your component in your entire application is very small you can deploy that frequently.

**Isolation:** Single microservice cannot crash entire system

Similarly, if the component crashes or goes down because of this isolation it will ensures that it will not affect the other parts of the system.

**Scaling:** We can Scale individual services based on their use.

You can individually scale your components or so your payment module or your payment service is used lot more you can increase its scaling independently of scaling any other module.

**Culture:** Well defined ownership, high trust team.

It improves your culture now each team is working on single business capabilities and you get flexibility.

**Flexibility:** Polygot services, own persistence systems etc.

As long as the contracts the API between two micro services are not changed, they remain intact you can change the language or frame work that you use to deploy a micro service.

You can change your data base systems each micro service can use its own set of queues and things like that.

since now we converted a monolith into distributed system, we also get a set of challenges.

**challenges**

set1:

Service Discovery

Load Balancing

Fault tolerance

Distributed Tracing

Telemetrics

Security(mTLS, Policies, patches)

set2:

Granularity

Bounded Contexts

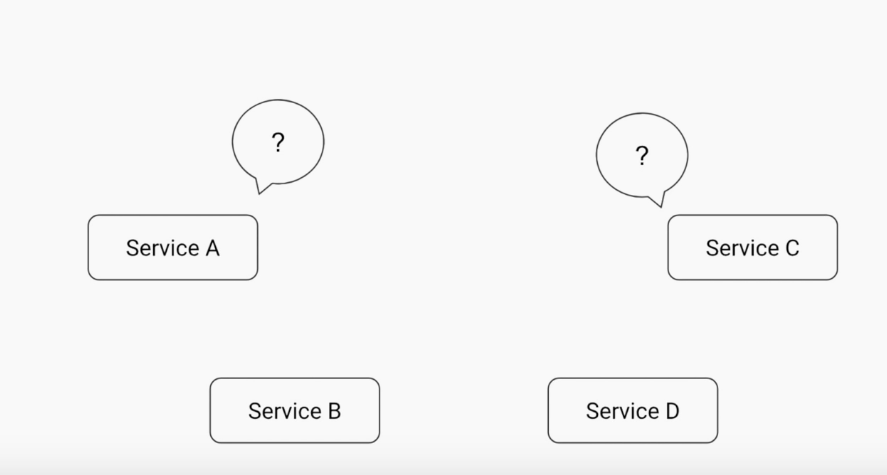
Data modelling

Independently releasable

Service contracts

Smart services, dumb-pipes

Service discovery:



Now let’s say we split monolith into micro sercice.

ServiceA

ServiceB

ServiceC

ServiceD

Here Each micro service could be running on a different machine/different vm with its own ip and port.

How one service will get to know about IP addresses of other services.

one option is hard code the IP.

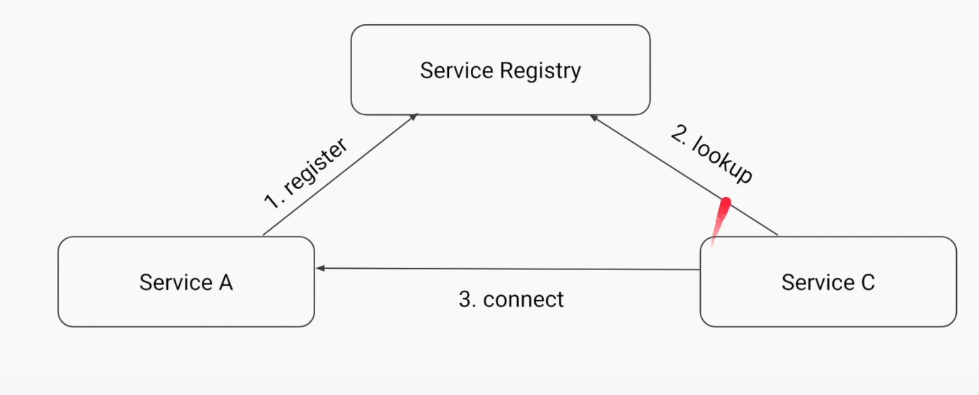
But the problem In cloud if your service has restarted then it will get a different IP.

or

We may have multiple copies of the same service (as replicas) so each copy will have a different IP.

As a result, we cannot hard code IP.

So essentially, we need a registry of service discovery.



let us say we have a single service registry

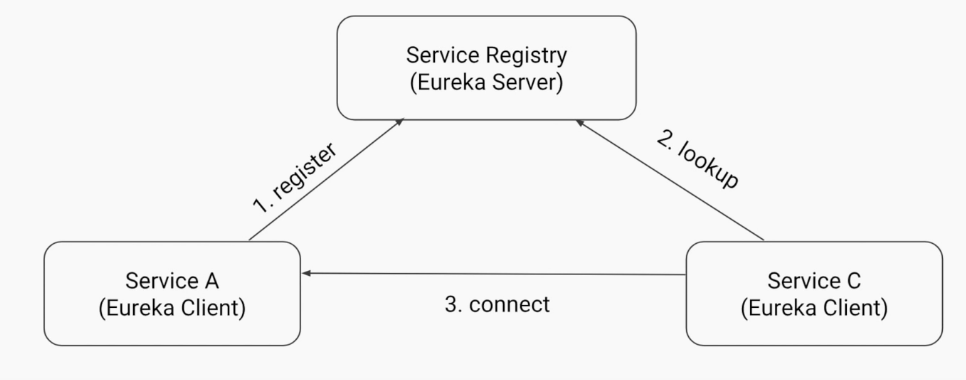
whenever any service comes up or it starts it will register itself to its registry.

my name is service A and my IP address is this.

you please keep it in your registry.

whenever there is another service which want to talks to this service it will lookup for service A in service registry and it will get service A IP address.

now the service C can talk to its service A.



Service Registry(Eureka Server)

Servcice A (Eureka Client)

Service C(Eureka Client)

In this project we have two components:

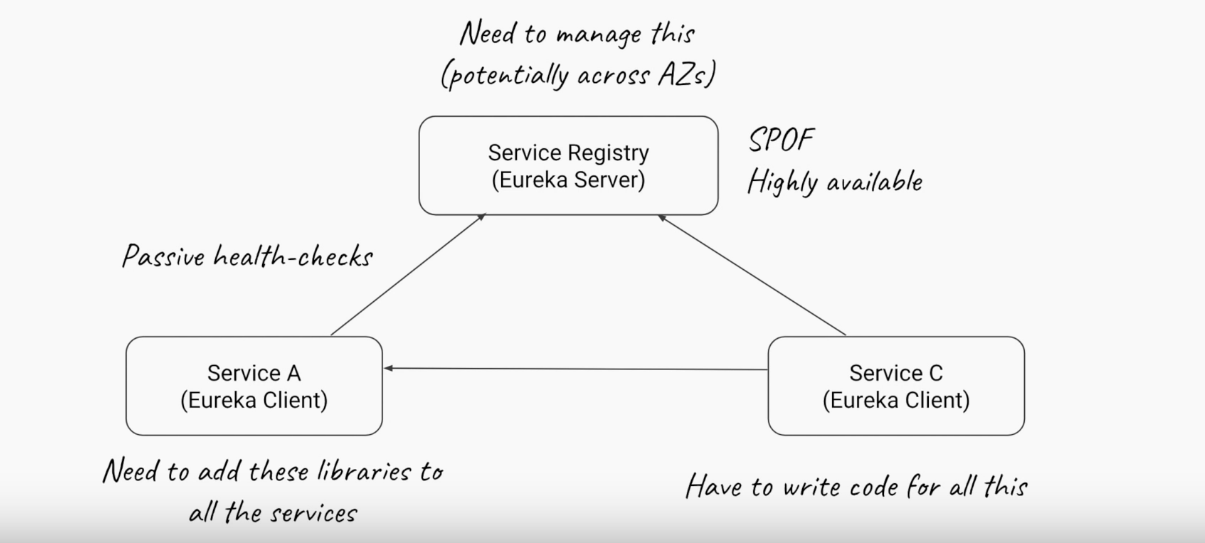
1st component:

The first component is an application is also called as eureka server this is a spring boot application it acts as your service registry. So this is the application all your clients or all your services will talk to register itself and to lookup other services.

2nd Component:

The second component is called the eureka client it is a library that you need add it to your each of the micro services projects. Once you do that (also you need to add a set of properties) so when you start the service it will register to the eureka server and whenever eureka client wants to lookup from the server it will give back the ip address and it can connect to it.

But there are a certain problems with it.



One is since this is an application. (ie eureka server)

Eureka server is a standalone application we have to manage it.

This is also a single point of failure now.

Assume If we have 10 micro services all are talking to the single service registry.

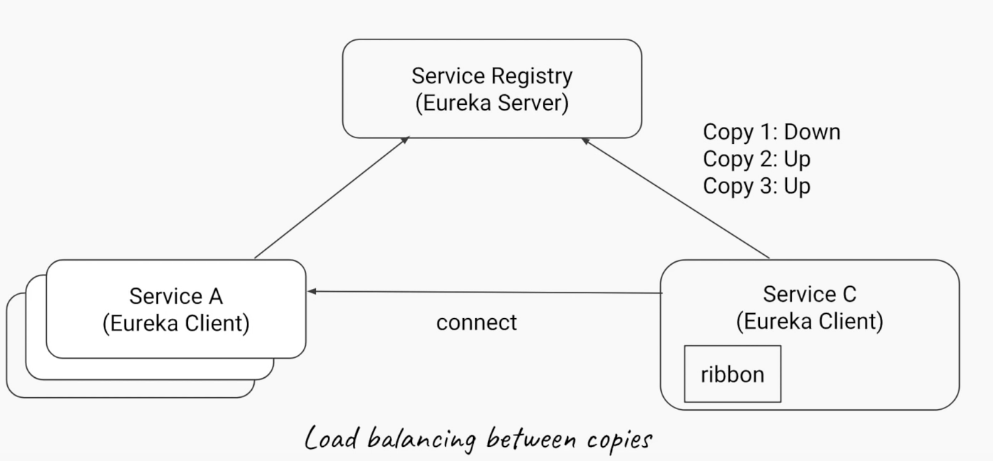
Even if for few mins if your service registry goes down for few mins then it will affect the entire system.

So that is why it is SPOF and it needs to be highly available all the time.

There is this pear awareness and clustering concepts available in eureka server where you can deploy multiple copies of this eureka server.

The other problem is you need to add lib and code to your micro services for this entire thing to work.

Load Balancing-spring cloud



How we took an example of payment service requiring scaling let us say we have serviceA which has multiple copies all are running at the same time.

The service C which consumes serviceA

In Service C we need add library called as ribbon. Service C is client

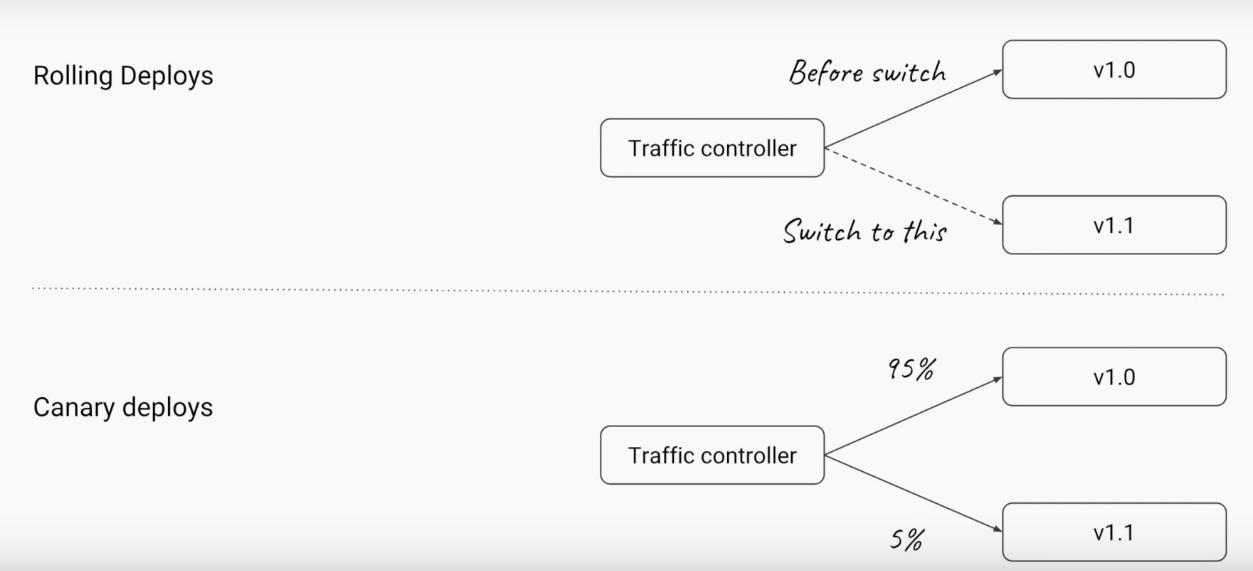
which will get all the addresses of all the copies of serviceA from registry.

and internally it will decide how to do the load balancing between copies.

So first time it can choose to connect to copy one.

second time it might connect to copy two etc

Canary / Rolling Deploys.

**Rolling Deploys**

Lets say we have v1.0 of service and all our traffic is going to that.

you want to able to start the version 1.1

new update to your service and at the right time you want to able to switch all the traffic from 1.0 to 1.1

and you want to ensure that while you do this transition your service is not getting effected.

your traffic is still getting handled properly.

but behind the seens you are doing an update to your service.

**Canary Deploys**

let us say you have updated a feature.

and u want to test it with a very small crud before deploying it to wider audience.

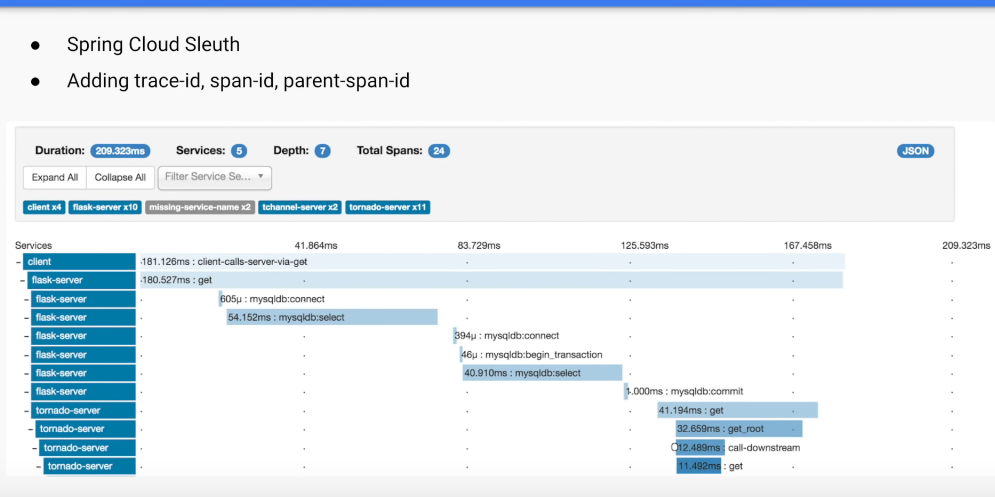
So let us say you want to expose your new feature only to 5% of users.

So some how u need to have an ability to push only 5% of u r traffic into the new version of the service and the rest of the 95% will still goto your old service and once now you think this feature is really working well you slowly migrate more and more traffic to this v1.1 and eventually you will migrate the entire 100% traffic to this new version. So this concept is called canary deploys.

**Distributed tracing**

spring cloud sleuth

Adding trace-id, span-id, parent-span-id



There another problem your flow of an application can go through multiple micro services how will you debug that in this case so whenever a request comes in you need to assign it a unique code this is also called as a traceID and then that unique code every micro service when it logs for that request it needs to log with that unique code so that then you can trace the entire flow across all the services using that particular unique id.

**Security:**

Mutual TLS between services

Network policies / whitelisting

Transparent patching

Another problem with micro services is security.

How do you ensure that the two services those are talking to each other they use https for that you need to install certificates on both the services?

And there is also a concept of certificate rotation.

So every 30 days you should ensure that certificates are renewed or replaced so if some one gets holded the certificates they cannot use the micro services all the time.

The second security issue is networking policies.

How do you ensure you have a set of rules that this service is allowed to be talking to only service b and service c but not service d. you need to be able to have these set of rules that which sevice can talk to which other service.

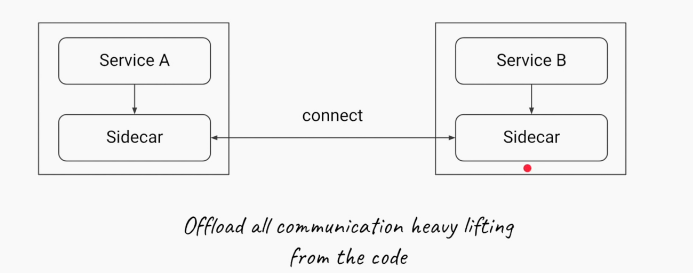
**Service Mesh:**

How the service mesh is intended to solve the problems?

ans: The main moto of service mesh is Do not burden my code with all these infrastructure related decisions. So all these problems are not related to actual business code, they are related to communication.

we want someone or some platform or some code to take of all the responsibilities with in our application code or do it somewhere else in a different plane.

**Sidecars**



Let us say we have initially serviceA which is talking to service B.

Now lets say we have new program / application it also built in java called as side car.

which is always running where the micro service is running.

ServiceA machine will have ServiceA and its own sidecar.

ServiceB machine/container will have ServiceB and its own side car.

Whenever ServiceA wants to connect to ServiceB it will not talk directly.

It will talk to the sidecar.

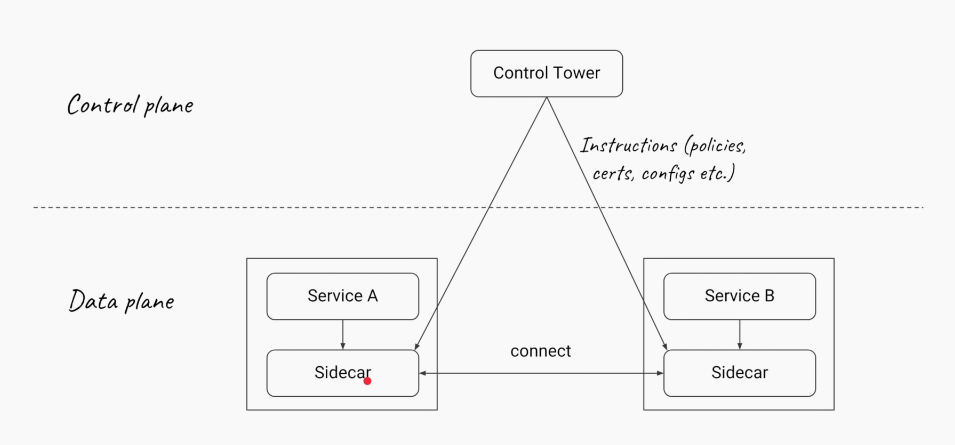
So side car will act as proxy for serviceA.

And then it is side cars responsibility to talk to other service's Side car.

so this side car proxy offloading all communication or infra structure related difficulties that we spoke about from within u r core we are offloading it to the side car.

this is the advantage of having side car.

**Data plane vs control plane**

****

Let us say let me add one more component

**Control Tower**

Control Tower could be a program which always manages the side car

we do not want to put the code in the side car we want to manage the side car by some one else.

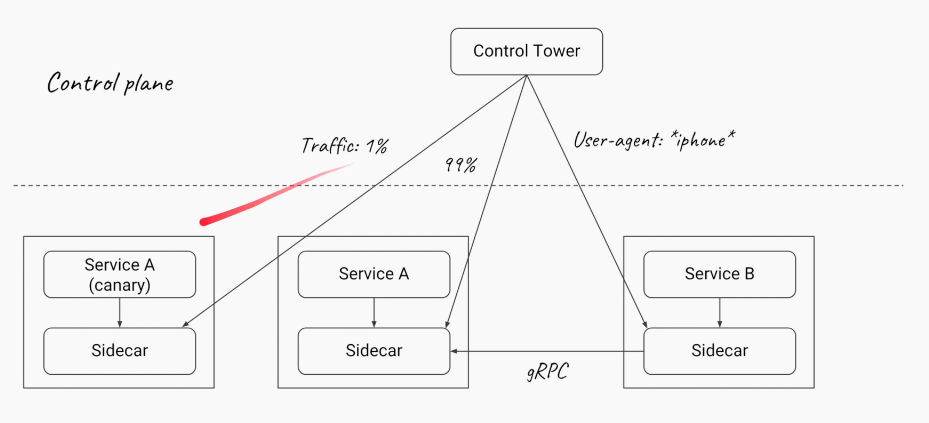
let us say that someone else is called control tower.

this control tower will be responsible for pushing all that configuration about service discovery networking policies traffic management, load balancing certificates, mutual tls authentications and so on and so forth.

that control tower takes care of all these problems and challenges.

we do not want to deal those things in our service code.

**Traffic Management**

****

If we go one step even further.

What we can do is since now we have this control tower component this program which dynamically takes care all these side cars. we can do all those things we spoke about.

so in this example let us say we have serviceA which is updated one that we want to test out we can ask the control tower hey just sent one % of my traffic to this update serviceA (ie canary version) while rest of the 99% of the traffic goes to un-updated serviceA or stable serviceA.

Similarly check the headers of the request if the request is coming from iphone send that request to ServiceB.

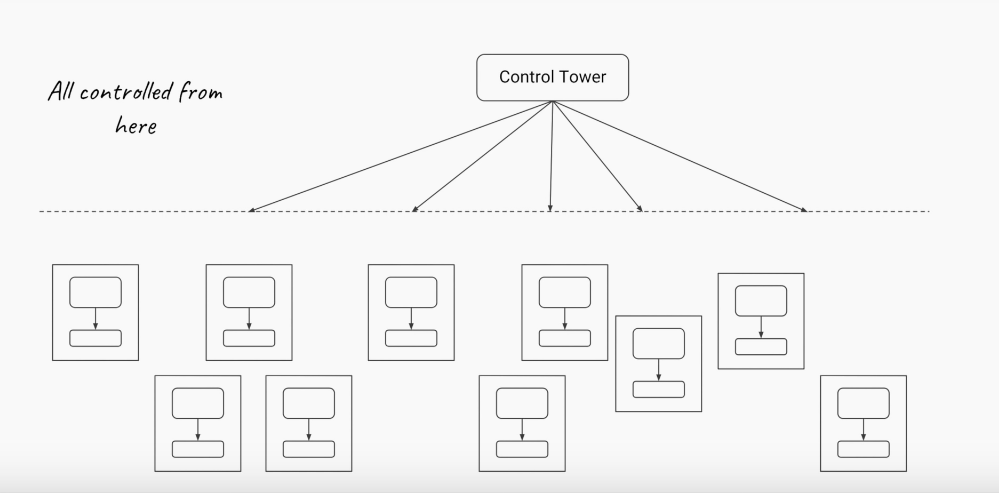
otherwise all ways send to ServiceA.

one more adv of control tower and side car is since now your services are not talking to eachother directly side car can decide which communication protocols to use let us intially it is using http 1.

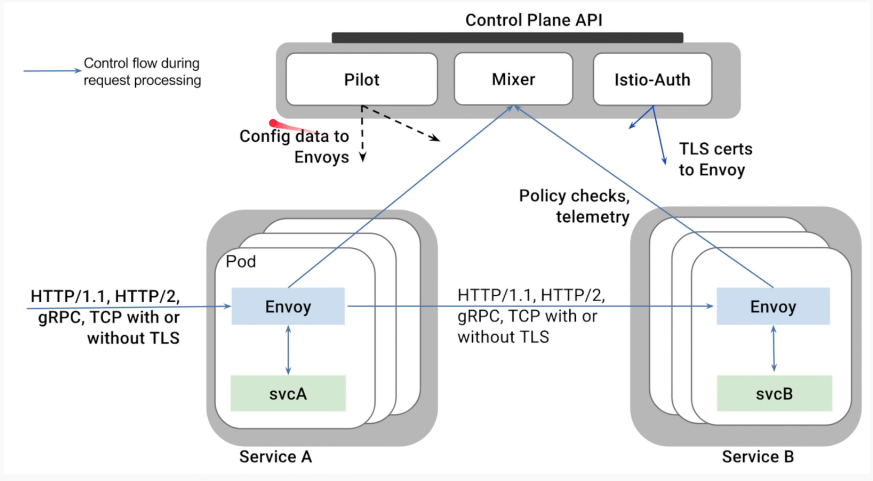
it can switch it http 2 or it can switch to gRPC.

We can understand the adv whenever we have 100 of micro services so we no need to write the single code to manage the services.

so this whole concept control tower and side car is called service mesh.



Important Note: service mesh allows you to offload this un differentiate heavy burden or heavy lifting from your code on to a control tower and driven by the side cars one of the projects you implement service mesh Istio + Envoy.



Envoy is a project which acts as a side car that we talked about.

In the diagram service A has side car envoy.

Service B also has side car envoy.

In the control pan we have the project called Istio.

configurations are push down to the side car.

TLS certificates are also push down to the side cars.

Metrics and network policies of which services can talk to what service is also push down to side cars by this control pane.

And then between the envoys they can choose which protocols to be used.

Istio and envoy are combined and one such service mesh project.

**Summary of Responsibilities of the service mesh**

Service Discovery

Load Balancing (extensive set of algorithms)

Multi protocol support (HTTP / gRPC)

Fault tolerance (Circuit breaking, Rate limiting, Auto-retries)

Scaling

Telemetrics(including wire-level like MongoDB, DynamoDB)

Distributed Tracing

Security(mTLS, Policies)