# Design system

## Design Youtube:

- Repeat the question.
- ask lot of questions.
  - break it down to what they really want you to build
  - what are the requirement.
  - think out loud when you approach the problem.
- example of questions:
  - What parts of YouTube do you want me to design?
  - Obviously there's a lot of things to YouTube, like recommendations and editing content and channels and advertisements and managing payments to people.
  - what part do you want be to focus on.
    - upload videos, or playing them.
  - How much video are we talking about, how much traffic are we talking about.
  - requirement terms of latencies or availability. & ask about the budget as well
    - it shows you what type of trade of are we talking about.

## Working Backwards:

- Once you get all the requirement, start designing backwards from the customer experience.

> For example, YouTube.
> Okay, I need to vend massive amounts of videos
> all around the world at very low latency
> and at massive scale.
> Well, that probably means I need to use a CDN
> for my most popular videos at least.
> Where does the data come from that feeds that CDN,

- who are those customers
  - are they all around the world.
- what are their use cases
- which use cases do you need to concern yourself with.

## Defining scaling requirements

- Is it hundreds of users ? Millions or users ?

    - you need horizontal calling.
    - how often are users coming ? what transaction rate do you need to support ?
- Also define the scale of the data.
    - Hundreds of videos? Millions?
- Youtube example: millions of users, millions of videos
    - You will need every trick in the book for horizontally scaled servers and data storage.
- Some internal tool might not need this level of complexity, however:
    - Always prefer the simplest solution that will work.
    - Vertical scaling still has its place.

## Defining latency requirements

- How fast is fast enough ?
    - use CDN usage.
- Youtube example
    - caching video recommendations.
    - caching video metadata, descriptions.

## Defining availability requirement

- How much downtime can you tolerate?
- look for single point of failure and try to replicate it - do a backup.

> if you've been explicitly told that this is a high availability service. And they might not tell you, right.
> Sometimes they might expect you to arrive at these answers yourself.
> And again, if you just work backwards
> from the customer experience that can inform you as to what the optimal customer experience might look like.
>
> So, if they don't tell you what the latency requirements are say, well, you know, as a customer,
> I expect pages to render pretty quickly.
> I would expect, you know, this webpage to render in under one second.
> That means I need to have very fast access to this data locally available, probably in a CDN somewhere.

## Think Out Loud

## Be Honest

- Don’t pretend to know stuff you don’t know. That won’t end well.
- If you’re steered into a direction you’re unfamiliar with, say so.
- But don’t just give up, let the interviewer help you.
- This is an opportunity to learn.

# Defending your Design

- don’t get defensive - take feedback constructively.

## Cold Start Microservice

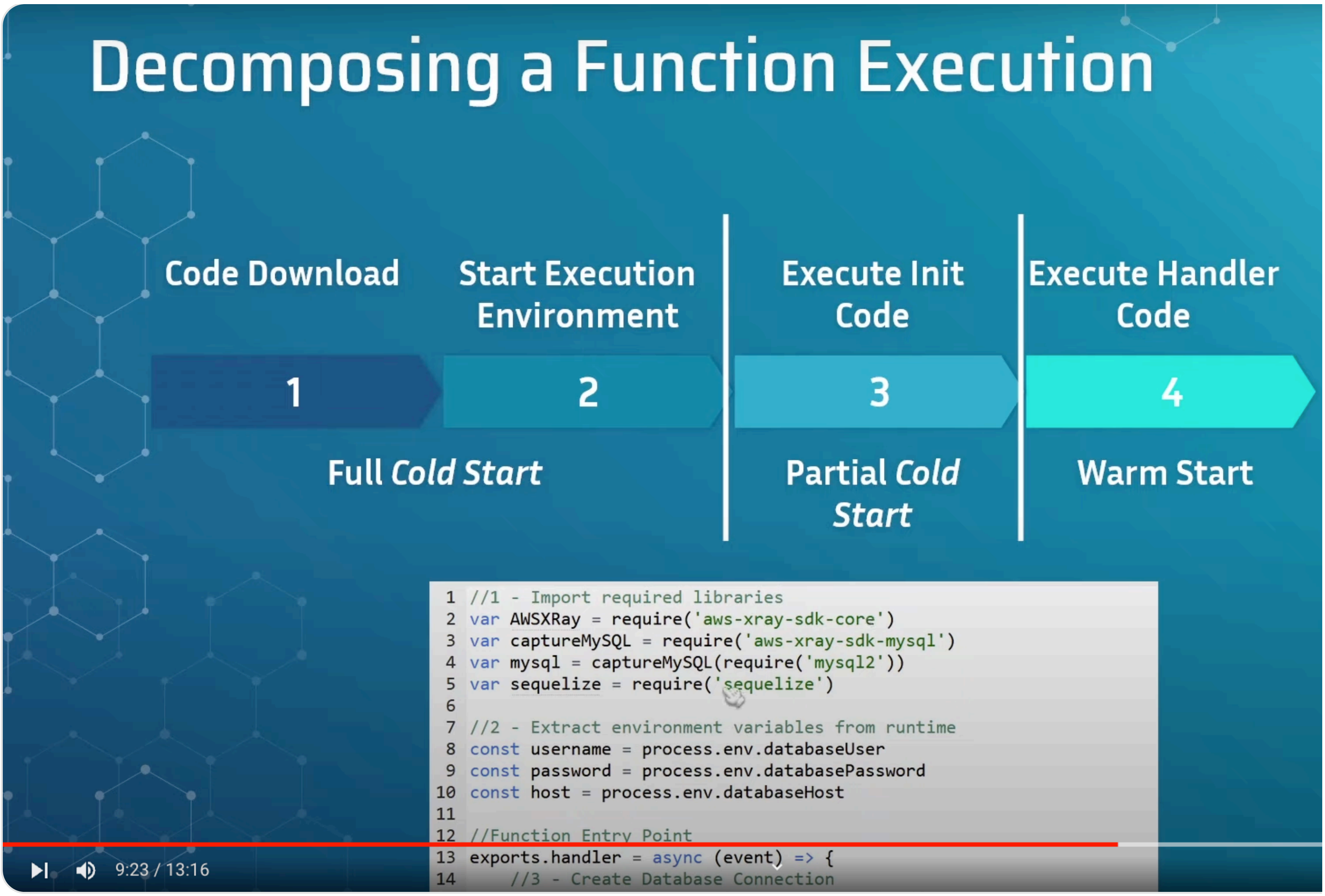
Cold start is the time it takes for a serverless function to initialize and execute when it is invoked for the first time or after a period of inactivity

- Warm-Up Strategies
  - Scheduled Tasks:
    - after deployment ping health check of the service to ensure warm up.
    - Or every 1 scheduled tasks to ping the health check.
  - using caching solutions
    - prepopulate when deployment is happing.
- Proactive Scaling.
  - use AWS auto Scaling: it can be configured using gantry.
    - monitor the health of your services and automatically adjust the number of instances based on defined policies.
    - This helps mitigate cold starts by maintaining a minimum number of warm instances.
- Optimizing Container Orchestration
  - Use AWS Fargate: Fargate allows for serverless container orchestration, reducing the likelihood of cold starts by abstracting away the underlying infrastructure.

**Problem description:**

The cold start happens when you do your very first request after prod deployment OR there have been very long down time with no request.

- In addition in finding and allocating resources to your request, there are 4 steps need to be handled in order to return a response.
  - ***Code Download***: from Zip file or download from S3.
  - ***Start Execution Environment***: be smart when you choose your Programming language. normally when it comes for lambdas Engineers typically use either `nodejs` or `python` which as less start execution time compared to `Java` for example.
  - ***Execute Init Code***:  eveything before the handler function typically `require` . best to minimise the requirment
    - 🟥 Bad Code: `import * as Helpers from "./utils/helpers"` → don’t import everything.
    - 🟩 Good Code: `import { isOddNumber } from "./utils/helpers"` → only import the thing you need.
  - ***Execute Handler Code***: Execute the main `handler` function.
- Note: Cold Start also occurs while scaling up.



Decomposing a Function Execution

Code Download | Start Execution Environment | Execute Init Code | Execute Handler Code

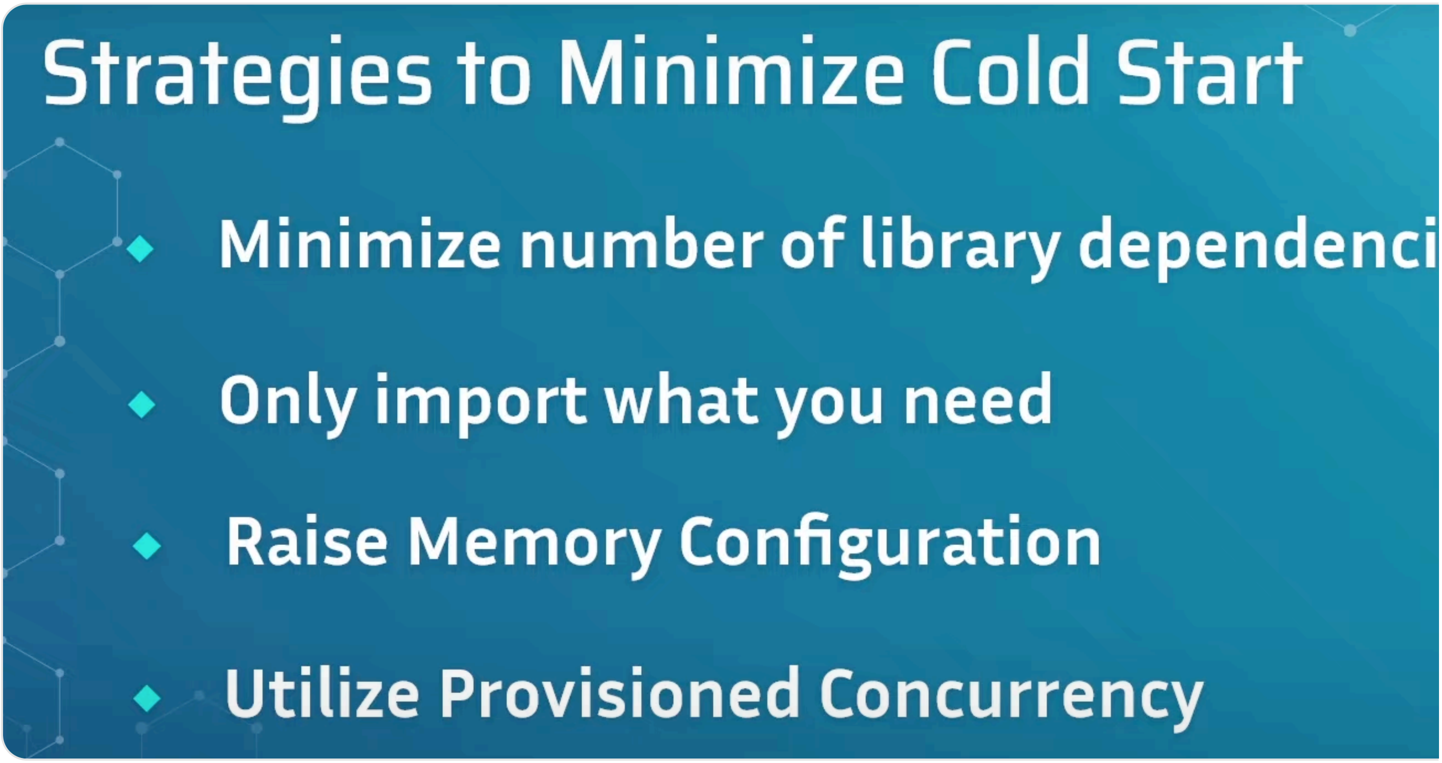
1 | 2 | 3 | 4

Full *Cold Start* | Partial *Cold Start* | Warm Start

```
//1 - Import required libraries
var AWSXRay = require('aws-xray-sdk-core')
var captureMySQL = require('aws-xray-sdk-mysql')
var mysql = captureMySQL(require('mysql2'))
var sequelize = require('sequelize')

//2 - Extract environment variables from runtime
const username = process.env.databaseUser
const password = process.env.databasePassword
const host = process.env.databaseHost

//Function Entry Point
exports.handler = async (event) => {
    //3 - Create Database Connection
```



Strategies to Minimize Cold Start

- Minimize number of library dependenci
- Only import what you need
- Raise Memory Configuration
- Utilize Provisioned Concurrency

### Optimising Docker Images: Best Practices

https://www.squash.io/how-to-improve-docker-container-performance/

- Use offical Base Images and Alpine ones

    - example `FROM node:20-alpine` do not use ubuntu image then install node it will be large image size.
- Minimise the number of Layers example
    - Do this `RUN apt-get update && apt-get install -y package1 package2 package3`
    - Not this

```
RUN apt-get update
RUN apt-get install -y package1
RUN apt-get install -y package2
```

- Use `.dockerignore` to Exclude Unnecessary Files
    - for example exclude `node_modules*/` `logs/` use don't need to copy it only the `src` you need.
- Use Specific Tags for Base Images
    - use specific version instead of `latest` so it won't break.
- Optimising Image size
    - Use a minimal base image: Choose a base image that only includes the necessary dependencies for your application example `From node:20-alpine`.
    - Avoid unnecessary packages and dependencies: Only include the packages and dependencies required by your application.
    - Remove unnecessary files: Clean up any unnecessary files and directories in your image.
- Use multi-stage docker build
    - the one with `From node:21` can do `test`, `lint` & `build` → build means turn `TS` to `JS`.
    - the final image that run into a container will be `From node:21-alpine` , which takes the already built project and CMD ['./listen.js '] listen to it.
        - `COPY --from=deps /workdir/dist/appsapi .` `from` the previous image build
    - Note: `alpine` does not have `tsc` compiler to compile TS to JS.
    - you don't need to have the entire `src` at all only the `dist` built code you need and then run it.
- use default docker cache mechanism in user advantage. the order in DockerFile is important
    - the things that are less likely to be changed will be placed on top. will insure to be cached.
    - the things that more likely to be changed will be placed on the button. as they will invalidate the cache until the end of the file commands.

## Managing Docker Containers: Tips and Tricks

- Use Appropriate Resource Limits
    - Set the necessary amount of CPU and memory resources.
    - This helps prevent one container from monopolising the resources and affecting the performance of other containers.
    - `docker run --cpus=1 --memory=512m my-container`
- Monitor Resource Usage

    - Monitoring the resource usage of your Docker containers is essential to identify bottlenecks and optimise performance.
    - `docker stats --all`
- Use Docker Volumes for Persistent Data
    - Docker volumes are a great way to manage persistent data for your containers. By using volumes, you can separate data from the container's filesystem, making it easier to manage and backup.
    - `docker run -v myvolume:/data my-container`
- Clean Up Unused Containers and Images
    - unused containers and images can accumulate and consume valuable disk space. It's important to regularly clean up these unused resources to optimize disk usage
- Utilize Docker Compose for Complex Deployments
    - By using Docker Compose, you can easily define the relationships between different containers and manage their configuration. This simplifies the deployment process and ensures consistency across different environments.

### Improving Docker Networking: Strategies for Efficiency

- reach docs

### Scaling Docker Applications: Techniques for Performance

- Load Balancing
- Horizontal Scaling
    - Horizontal scaling involves adding more instances of Docker containers to handle increased demand. By horizontally scaling your application, you can distribute the workload across multiple containers
- Vertical Scaling
    - increase of cpu and memory of each container
- Caching
    - use Redis cache to minimize the load on db. as your dockers are stateless.
- Monitoring and Optimisation
    - AWS ECS will provide dashboard to monitor performance.

### Container Orchestration with Docker

- What is Container Orchestration?
    - Container orchestration is the process of automating the deployment, management, and scaling of containers. It helps in efficiently running and coordinating multiple containers across a cluster of hosts. Orchestration platforms provide features such as service discovery, scaling, load balancing, high availability, and fault tolerance.
    - it provide load balance and auto scaling functionality out of the box.
- Why Use Container Orchestration?

    - Container orchestration simplifies the management of complex containerised applications by abstracting away the underlying infrastructure. Here are some key reasons to use container orchestration
        - ***Scalability***: auto scanning: Orchestration platforms enable you to easily scale your applications horizontally by adding or removing containers as per the demand.
        - ***High Availability***: Orchestration tools monitor the health of containers and automatically restart or replace failed containers, ensuring high availability of your applications.
        - ***Load Balancing***: Orchestration platforms distribute incoming traffic across multiple containers, optimising resource utilisation and improving performance.
        - ***Service Discovery:*** Orchestration tools provide built-in service discovery mechanisms, allowing containers to find and communicate with each other seamlessly.
            - if they are in the same AWS-ECS cluster , gantry env.

# Databases

## In SQL DB if how to make search faster ?

- we index the column.

## Why SQL Indexes?

- Indexes are used by queries to find data from tables quickly.
- Indexes are created on tables and views.
- with out indexes we will result of query scans.
- example `CREATE Index IX_tblEmployee_Salary ON tblEmployee (SALARY ASC).`

## What are the types of Indexes in SQL?

- Clustered Index: determines the physical order of data in a table like `id` of the table type `int Primary Key`
    - you can only have one clustered Index in one table.
    - you can drop the clustered index in `id` and do a composite clustered index example (Gender & Salary) index. just like DynamoDB.
    - the clustered Index determines the order which table rows are sorted.
- Non Clustered Index:

## Advantage vs disadvantage of Indexes

- Advantage:
    - `SELECT` clause can benefit from indexes.
    - `DELETE` & `UPDATE` can benefit from indexes as well.
    - `SELECT` with `ORDER BY` the index.

    - `GROUP BY`
- Disadvantage:
    - Add additional Disk Space: for non Clustered.
    - `DELETE` , `UPDATE` & `INSERT`: can be slow if we have too many indexes, the record as to be deleted from all the index tables.
        - you have to choose the right amount of indexes on your table.
    - non covering query can be slower than covering query: as the non covering query needs to look up in the actual date from the test of fields of our query is for example `SELECT * ..etc`

# Kubernetes (EKS):

- Managed Kubernetes Service: you don't need to worry about the master nodes. AWS will take care of it.
- AWS manages Master Nodes.
- Necessary apps pre-installed.
- Scaling and backups.
- you need to focus on your application worker node only.

### How to use EKS?

- Setup or preparation steps.
    - Create AWS account.
    - Create a VPC.
    - Create an IAM role with Security Group.
        - AWS user with least permissions.
- Create Cluster Control Plane: the master node (using the IAM role).
    - choose cluster name, k8s version.
    - choose region and VPC for your cluster.
    - set security for your cluster.
- Create Worker Nodes and connect to cluster.
    - these worker nodes will be some EC2 Instances with cpu and ram configurations.
    - Create as a Node Group (Group of Nodes) for autoscalling.
    - Choose cluster it will attach to.
    - Define Security Group, select instance type, resourcees
    - Autoscaling configuration → define  max and min number of Nodes.

# AWS - ECS

## Types of ECS

- Serverless with Fargate.
- self managed with EC2:

    - you require to manage these EC2, update security and update dependencies.

### ECS support auto scaling.

- to handle variable volume.

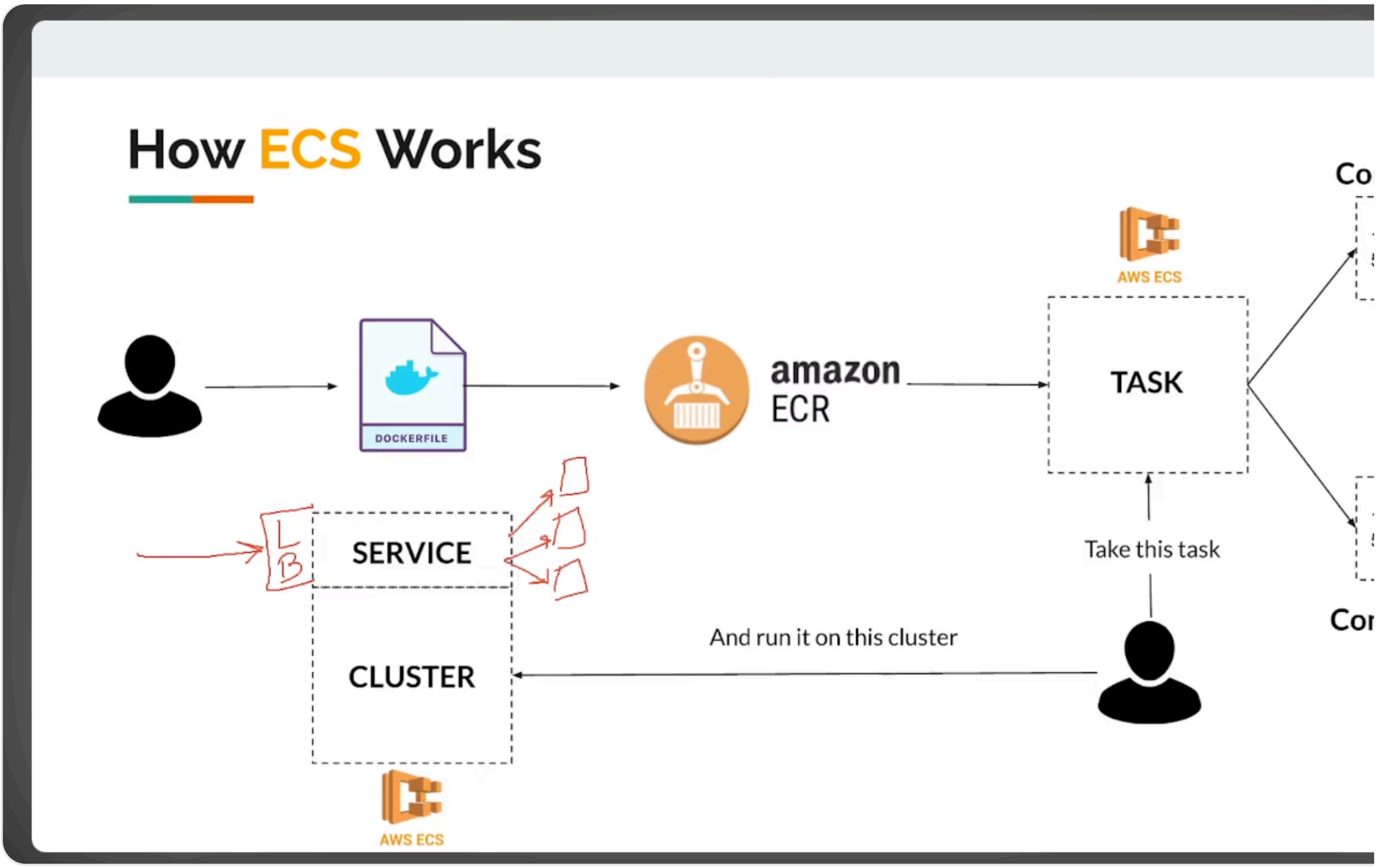
### ECS great with services that requries to be live all time

- like  a stateless RESTful API.

### CES is cost effective

### How ECS works.

- task definition: tells ecs how you want to spin up your docker containers
    - a task can contain more than one container.
    - some applications comes in pair, exmaple: rest-api (stateless) & database (stateful).
        - in your task you can specify internal ports than open between these two containers
- Cluster:
    - where your ec2 runners lives: if you use fargate your these ec2 instances will be hidden from you as you don't need to worry about them.
    - you define a `Service` inside your `Cluster` .
        - the `Service` allows you to specify a minimum number of tasks. therefore containers running on this cluster at any point of time.
        - for example: you have very popular application, you need lots of infrastructure to run it on. you would define a service that says at any given point I need at least 10 tasks of this type.
            - what it means you have 10 `Contanier1` & 10 `Container2`
    - `Service` comes with great tools, it will always monitor your running containers, making sure they are live and healthy.
        - it comes with mentoring dashboard, cpu and memory utilisation.
    - How to make your application scalable?
        - you have application load balancer. and connect to your containers.
        - you have auto scaling solution comes with Service.



How ECS Works

## When to move from Monolith to microservice

-