Table Of Contents

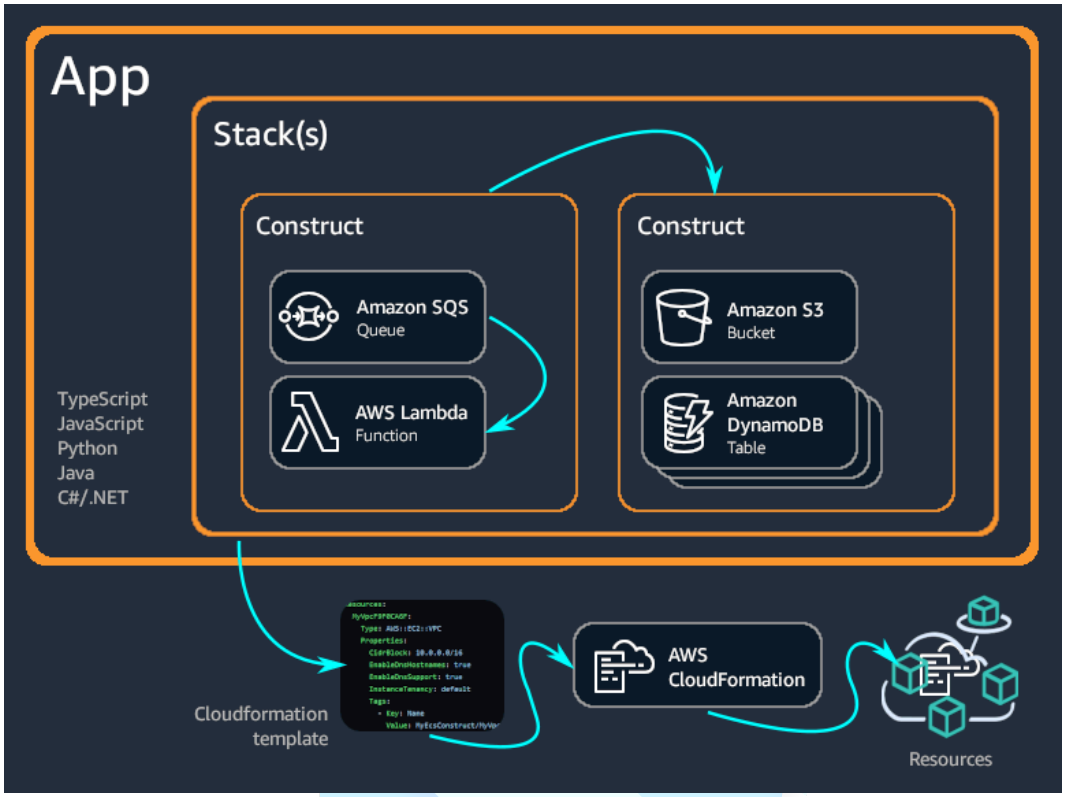
Amazon ECS

- Launch Types
- Cluster
- Autoscaling group
- Task definition
    - For fargate
    - Fro ec2
    - Fro ec2 anywhere
    - For all
- Images
- Service

Tested on:

- Linux Kernel: 5.15.0-102-generic
- CDK Version: 2.138.0
- Operating System: Ubuntu

AWS CDK



Prerequisite

- Node above version^18
- Aws cli
- aws cdk
- Typescript or any other supported language

https://cdkworkshop.com/20-typescript/20-create-project/100-cdk-init.html

```
akash@sky:~$ tar -xf node-v20.12.1-linux-x64.tar.xz
akash@sky:~$ cd node-v20.12.1-linux-x64/
akash@sky:~/node-v20.12.1-linux-x64$ ls
bin  CHANGELOG.md  include  lib  LICENSE  README.md  share
akash@sky:~/node-v20.12.1-linux-x64$ sudo cp -r ./bin/. /bin/
akash@sky:~/node-v20.12.1-linux-x64$ node -v
v20.12.1
```

Install Aws cli

The AWS Command Line Interface (CLI) is a unified tool to manage AWS services. Installing the AWS CLI provides you with commands for interacting with various AWS services directly from your terminal or command prompt.

```
akash@sky:~$curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip"
akash@sky:~$unzip awscliv2.zip
akash@sky:~$sudo ./aws/install
```

Cdk install (version - 2.138.0)

- Framework for defining AWS infrastructure as code.
- Install CDK CLI globally via npm or pip.
- Use CDK CLI for project management, template synthesis, and stack deployment.

```
akash@sky:~$npm install -g aws-cdk
```

Typescript install

- superset of JavaScript with optional static typing.
- Compile to JavaScript for execution.
- Install globally via npm for use in CDK projects, particularly if using TypeScript as the language.

```
akash@sky:~$npm -g install typescript
```

CDK commands

cdk init [TEMPLATE] Create a new, empty CDK project from a template
cdk list [STACKS..] Lists all stacks in the app [aliases: ls]
cdk synthesize [STACKS..] Synthesizes and prints the CloudFormation template for this stack [aliases: synth]
cdk bootstrap [ENVIRONMENTS..] Deploys the CDK toolkit stack into an AWS environment
cdk deploy [STACKS..] Deploys the stack(s) named STACKS into your AWS account

cdk import [STACK]	Import existing resource(s) into the given STACK
cdk watch [STACKS..]	Shortcut for 'deploy --watch'
cdk destroy [STACKS..]	Destroy the stack(s) named STACKS
cdk diff [STACKS..]	Compares the specified stack with the deployed stack or a local template file, and returns with status 1 if any difference is found

Introduction to Amazon S3 with CDK

Amazon Simple Storage Service (Amazon S3) is a scalable object storage service offered by Amazon Web Services (AWS). It provides developers and businesses with secure, durable, and highly available storage for a wide variety of data types, ranging from images and videos to application backups and log files. With Amazon S3, you can store and retrieve any amount of data from anywhere on the web, making it ideal for building cloud-native applications, hosting static websites, and storing data lakes.

Define an S3 bucket

```
const bucket = new s3.Bucket(this, 'MyFirstBucket');
```

Bucket constructs expose the following deploy-time attributes:

- bucketArn - the ARN of the bucket (i.e. arn:aws:s3:::bucket_name)
- bucketName - the name of the bucket (i.e. bucket_name)
- bucketWebsiteUrl - the Website URL of the bucket (i.e. http://bucket_name.s3-website-us-west-1.amazonaws.com)
- bucketDomainName - the URL of the bucket (i.e. bucket_name.s3.amazonaws.com)
- bucketDualStackDomainName - the dual-stack URL of the bucket (i.e. bucket_name.s3.dualstack.eu-west-1.amazonaws.com)
- bucketRegionalDomainName - the regional URL of the bucket (i.e. bucket_name.s3.eu-west-1.amazonaws.com)
- arnForObjects(pattern) - the ARN of an object or objects within the bucket (i.e. arn:aws:s3:::bucket_name/exampleobject.png or arn:aws:s3:::bucket_name/Development/*)
- urlForObject(key) - the HTTP URL of an object within the bucket (i.e. https://s3.cn-north-1.amazonaws.com.cn/china-bucket/mykey)

- virtualHostedUrlForObject(key) - the virtual-hosted style HTTP URL of an object within the bucket (i.e. https://china-bucket-s3.cn-north-1.amazonaws.com.cn/mykey)
- s3UrlForObject(key) - the S3 URL of an object within the bucket (i.e. s3://bucket/mykey)

you'll create a new S3 bucket. S3 buckets are used to store data in the form of objects. They are globally unique within AWS, and you can specify various configuration options when creating them, such as versioning, encryption, and access control.

```
import * as s3 from 'aws-cdk-lib/aws-s3';

const app = new cdk.App();
const stack = new cdk.Stack(app, 'MyStack');

new s3.Bucket(stack, 'MyBucket', {
  removalPolicy: cdk.RemovalPolicy.DESTROY // Optional: Specify removal policy
});
```

Adding an Object to an S3 Bucket

Once you have created an S3 bucket, you can upload objects (files) to it. Objects can be files of any type, and you can upload them using the AWS Management Console, AWS CLI, SDKs, or AWS CDK.

```
import * as s3 from 'aws-cdk-lib/aws-s3';
import * as cdk from 'aws-cdk-lib';

const app = new cdk.App();
const stack = new cdk.Stack(app, 'MyStack');

const bucket = new s3.Bucket(stack, 'MyBucket');

// Upload a file to the bucket
new s3.BucketDeployment(stack, 'DeployFiles', {
  sources: [s3.Source.asset('/path/to/local/folder')],
  destinationBucket: bucket
});
```

Setting Bucket Policy for Public Access

By default, S3 buckets and objects are private, accessible only to the AWS account that owns them. However, you can make objects in a bucket publicly accessible by configuring a bucket policy. Bucket policies are JSON documents that define permissions for buckets and the objects they contain.

```
import * as s3 from 'aws-cdk-lib/aws-s3';
import * as cdk from 'aws-cdk-lib';

const app = new cdk.App();
const stack = new cdk.Stack(app, 'MyStack');

const bucket = new s3.Bucket(stack, 'MyBucket');

// Add a bucket policy to allow public read access
bucket.addToResourcePolicy(new iam.PolicyStatement({
  actions: ['s3:GetObject'],
  resources: [bucket.arnForObjects('*')],
  principals: [new iam.Anyone()],
}));
```

StorageClass

```
new StorageClass(value: string)
```

Parameters

- value string

Properties

Name	Type	Description
value	string	

static DEEP_ARCHIVE	StorageClass	Use for archiving data that rarely needs to be accessed.
static GLACIER	StorageClass	Storage class for long-term archival that can take between minutes and hours to access.
static GLACIER_INSTANT_RETRIEVAL	StorageClass	Storage class for long-term archival that can be accessed in a few milliseconds.
static INFREQUENT_ACCESS	StorageClass	Storage class for data that is accessed less frequently, but requires rapid access when needed.
static INTELLIGENT_TIERING	StorageClass	The INTELLIGENT_TIERING storage class is designed to optimize storage costs by automatically moving data to the most cost-effective storage access tier, without performance impact or operational overhead.
static ONE_ZONE_INFREQUENT_ACCESS	StorageClass	Infrequent Access that's only stored in one availability zone.

BlockPublicAccess

Initializer

```
new BlockPublicAccess(options: BlockPublicAccessOptions)
```

Parameters

- options BlockPublicAccessOptions

Properties

Name	Type
blockPublicAcls?	boolean
blockPublicPolicy?	boolean
ignorePublicAcls?	boolean
restrictPublicBuckets?	boolean
static BLOCK_ACLS	BlockPublicAccess
static BLOCK_ALL	BlockPublicAccess

Enabling Versioning on a Bucket

S3 versioning is a feature that allows you to keep multiple versions of an object in a bucket. When versioning is enabled, S3 automatically generates a unique version ID for each object uploaded to the bucket. This feature is useful for data retention, backup, and recovery purposes.

```
const bucket = new s3.Bucket(stack, 'MyBucket', {
  versioned: true
});
```

Configuring Bucket Encryption

S3 supports server-side encryption to protect data at rest. When you enable encryption for a bucket, S3 encrypts each object stored in the bucket using the specified encryption method. You can choose between S3-managed keys, AWS Key Management Service (KMS) keys, or customer-provided keys for encryption.

```
new s3.Bucket(stack, 'MyBucket', {
  encryption: s3.BucketEncryption.S3_MANAGED // Use S3 Managed Keys
```

```
});
```

Cross-Region Replication

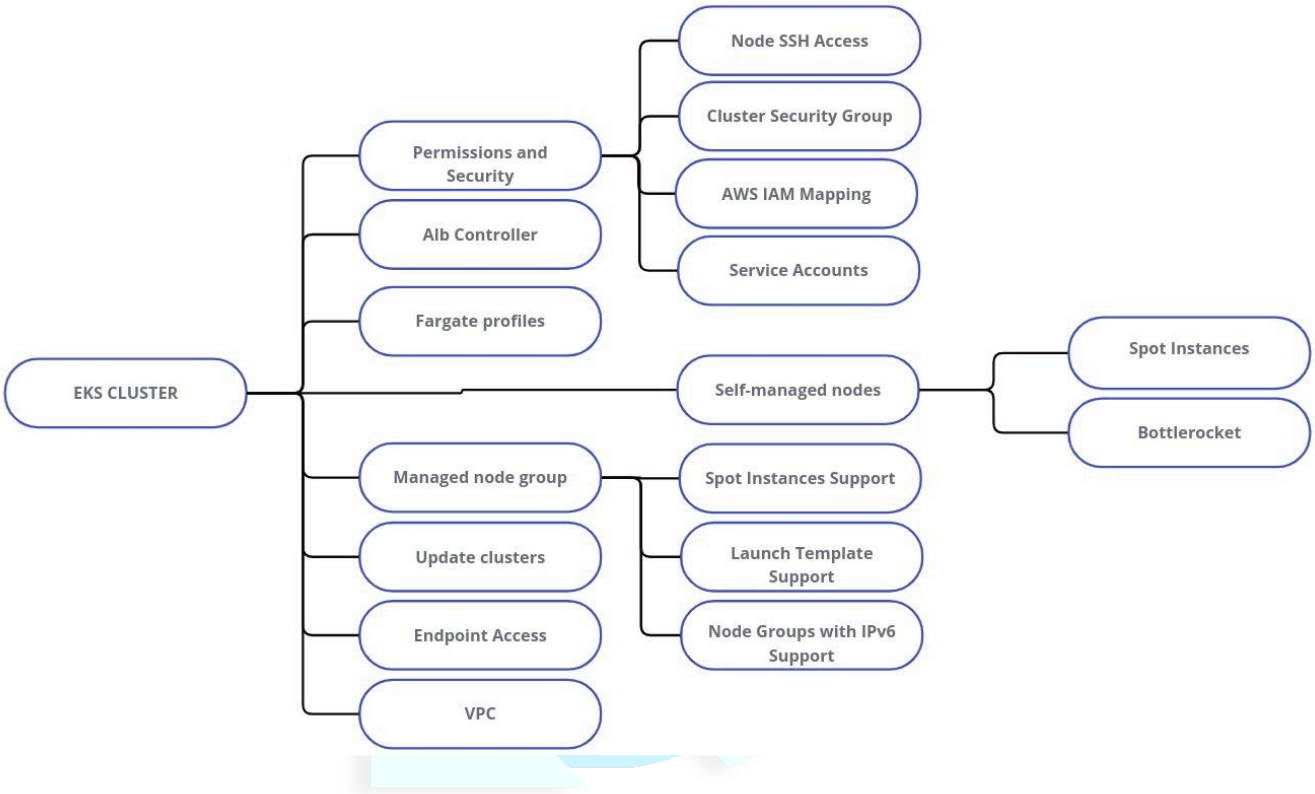
Cross-region replication is a feature that allows you to replicate objects from one S3 bucket to another in a different AWS region. This helps in achieving data resilience, compliance, and disaster recovery objectives. You can configure replication rules to specify which objects to replicate and the destination bucket.

```
import * as s3 from aws-cdk-lib/aws-s3';
import * as cdk from 'aws-cdk-lib';

const app = new cdk.App();
const stack = new cdk.Stack(app, 'MyStack');

const sourceBucket = new s3.Bucket(stack, 'SourceBucket');
const destinationBucket = new s3.Bucket(stack, 'DestinationBucket');

// Configure cross-region replication
sourceBucket.addReplicationRule({
  destination: {
    bucket: destinationBucket
  },
  prefixes: ['']
});
```

Amazon EKS Library

Introduction

- EKS Cluster - The cluster endpoint created by EKS.
- Managed Node Group - EC2 worker nodes managed by EKS.
- Fargate Profile - Fargate worker nodes managed by EKS.
- Auto Scaling Group - EC2 worker nodes managed by the user.



Creating an EKS Cluster

Objective: Create a new Amazon EKS cluster. Amazon EKS provides managed Kubernetes clusters, removing the need for you to manage the control plane and allowing you to focus on deploying and managing containerized applications.

When create an EKS cluster, you're provisioning a managed Kubernetes control plane in AWS. This control plane runs the Kubernetes API server, scheduler, and other core components required for managing containerized workloads. You can specify configurations such as the desired Kubernetes version, instance types for worker nodes, networking options, and more.

Managed Node Groups

Managed Node Groups are the recommended way to allocate cluster capacity.
By default, this library will allocate a managed node group with 2 m5.large instances (this instance type suits most common use-cas es, and is good value for money).
At cluster instantiation time, you can customize the number of instances and their type:

```
new eks.Cluster(this, 'HelloEKS', {
  version: eks.KubernetesVersion.V1_29,
  defaultCapacity: 5,
  defaultCapacityInstance: ec2.InstanceType.of(ec2.InstanceClass.M5, ec2.InstanceSize.SMALL),
});
```

FargateProfile

- Fargate profiles allows an administrator to declare which pods run on Fargate.
- Each profile can have up to five selectors that contain a namespace and optional labels.
- You must define a namespace for every selector.
- The label field consists of multiple optional key-value pairs.
- Pods that match a selector (by matching a namespace for the selector and all of the labels specified in the selector) are scheduled on Fargate.
- If a namespace selector is defined without any labels, Amazon EKS will attempt to schedule all pods that run in that namespace onto Fargate using the profile.
- If a to-be-scheduled pod matches any of the selectors in the Fargate profile, then that pod is scheduled on Fargate.
- If a pod matches multiple Fargate profiles, Amazon EKS picks one of the matches at random.
- Fargate Profile need private subnet to deploy there pod.

Auto Scaling Group

Creating an auto-scaling group and connecting it to the cluster is done using the cluster.addAutoScalingGroupCapacity method:

```
cluster.addAutoScalingGroupCapacity('frontend-nodes', {
  instanceType: new ec2.InstanceType('t2.medium'),
```

```
  minCapacity: 3,
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
});
```

- To connect an already initialized auto-scaling group, use the

```
cluster.connectAutoScalingGroupCapacity()
```

- To connect a self-managed node group to an imported cluster, use the

```
cluster.connectAutoScalingGroupCapacity()
```

Endpoint Access

When you create a new cluster, Amazon EKS creates an endpoint for the managed Kubernetes API server that you use to communicate with your cluster (using Kubernetes management tools such as kubectl

By default, this API server endpoint is public to the internet, and access to the API server is secured using a combination of AWS Identity and Access Management (IAM) and native Kubernetes Role Based Access Control (RBAC).

Alb Controller

To deploy the controller on your EKS cluster, configure the albController property:

```
new eks.Cluster(this, 'HelloEKS', {
  version: eks.KubernetesVersion.V1_29,
  albController: {
    version: eks.AlbControllerVersion.V2_6_2,
  },
});
```

The albController requires defaultCapacity or at least one nodegroup. If there's no defaultCapacity or available nodegroup for the cluster, the albController deployment would fail.

Permissions and Security

Amazon EKS provides several mechanism of securing the cluster and granting permissions to specific IAM users and roles.

The Amazon EKS construct manages the *aws-auth* ConfigMap Kubernetes resource on your behalf and exposes an API through the cluster.awsAuth for mapping users, roles and accounts.

For example, let's say you want to grant an IAM user administrative privileges on your cluster

```
   const eksAdminRole = new iam.Role(this, 'EksAdminRole', {
    assumedBy: new iam.AccountRootPrincipal(),
});
```

Amazon ECS

Amazon Elastic Container Service (Amazon ECS) is a fully managed container orchestration service.

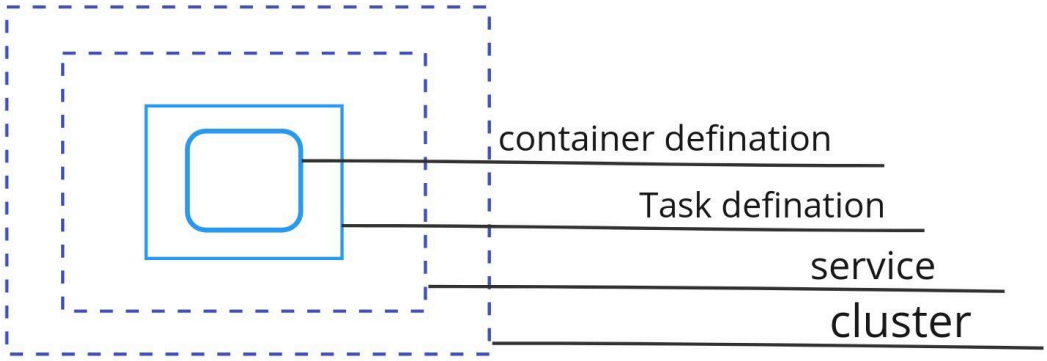
```
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
```

Launch Types:

AWS Fargate vs Amazon EC2 vs AWS ECS Anywhere

There are three sets of constructs in this library:

- Use the Ec2TaskDefinition and Ec2Service constructs to run tasks on Amazon EC2 instances running in your account.
- Use the FargateTaskDefinition and FargateService constructs to run tasks on instances that are managed for you by AWS.
- Use the ExternalTaskDefinition and ExternalService constructs to run AWS ECS Anywhere tasks on self-managed infrastructure.



Clusters

A Cluster defines the infrastructure to run your tasks on. You can run many tasks on a single cluster.
The following code creates a cluster that can run AWS Fargate tasks:

```
declare const vpc: ec2.Vpc;

const cluster = new ecs.Cluster(this, 'Cluster', {
  vpc,
});
```

AutoscalingGroup:

```
cluster.addCapacity('DefaultAutoScalingGroupCapacity', {
  instanceType: new ec2.InstanceType("t2.xlarge"),
  desiredCapacity: 3,
});
```

Task definitions

A task definition describes what a single copy of a task should look like. A task definition has one or more containers; typically, it has one main container (the default container is

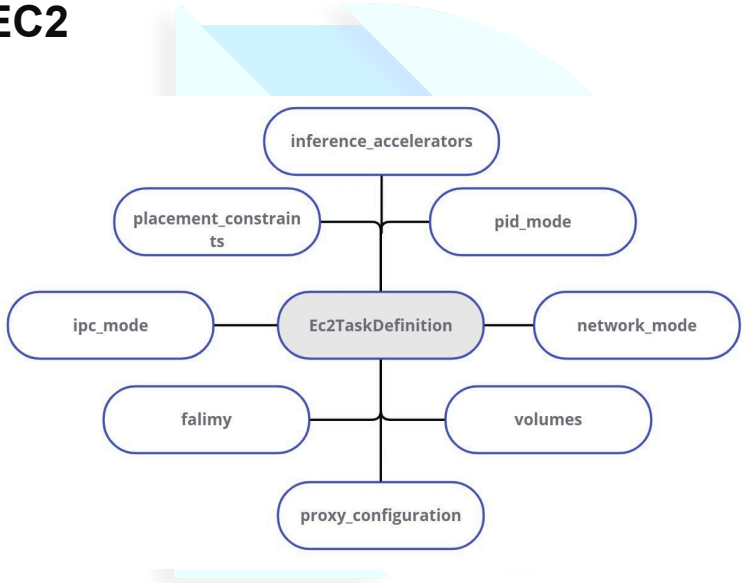
the first one that's added to the task definition, and it is marked essential) and optionally some supporting containers which are used to support the main container, doings things like upload logs or metrics to monitoring services.

To run a task or service:-
For Amazon EC2 launch type, use the Ec2TaskDefinition.
For AWS Fargate tasks/services, use the FargateTaskDefinition.
For AWS ECS Anywhere use the ExternalTaskDefinition.

These classes provide simplified APIs that only contain properties relevant for each specific launch type.
For a , specify the task size (memoryLimitMiB and cpu):

AMAZON EC2

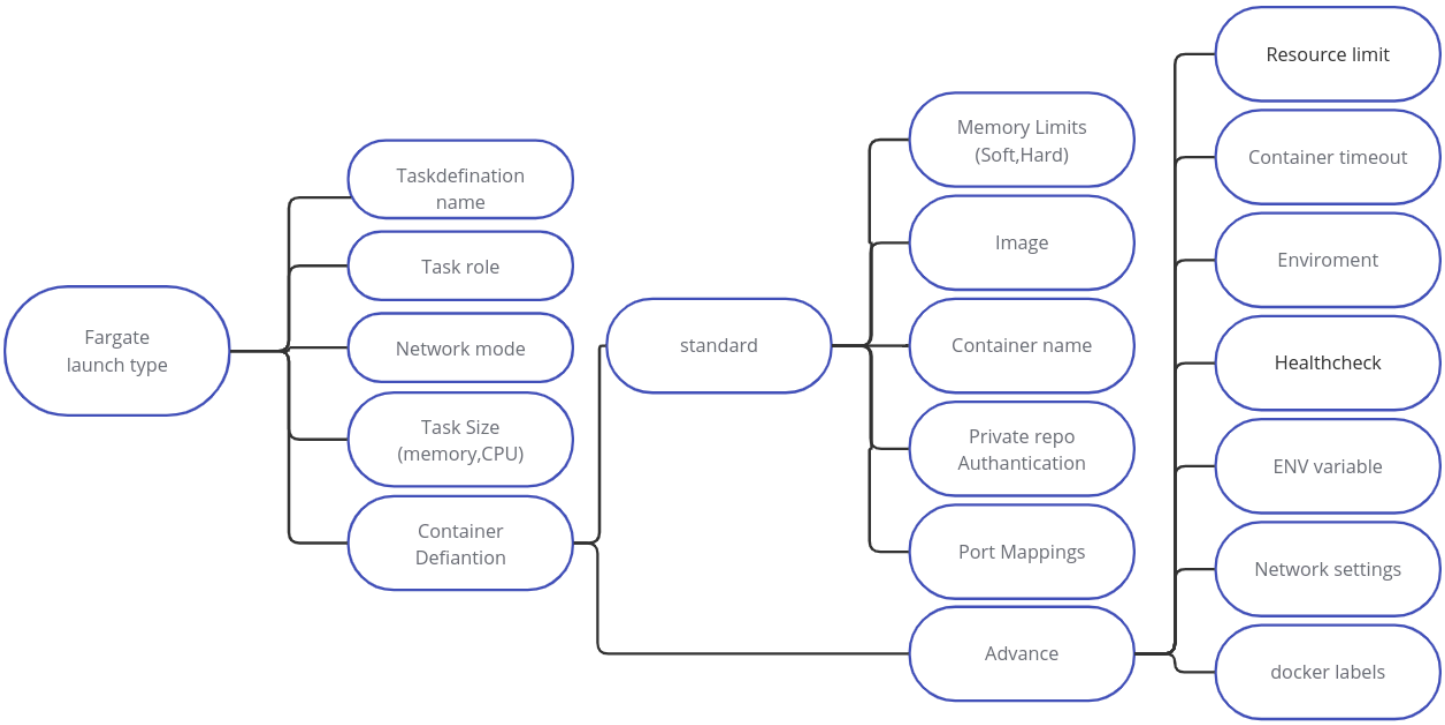
Parameters:-




```
const ec2TaskDefinition = new ecs.Ec2TaskDefinition(this, 'TaskDef', {
  networkMode: ecs.NetworkMode.BRIDGE,
});

const container = ec2TaskDefinition.addContainer("WebContainer", {
  // Use an image from DockerHub
  image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-sample"),
  memoryLimitMiB: 1024,
  // ... other options here ...
});
```

AWS Fargate

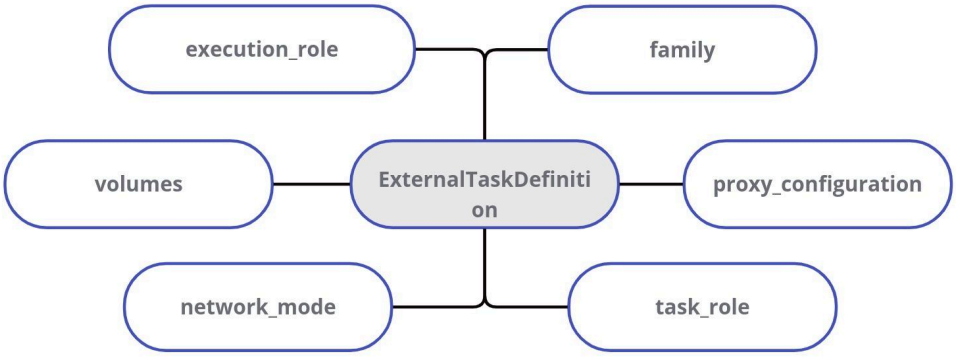
Parameters:-




```
const fargateTaskDefinition = new ecs.FargateTaskDefinition(this, 'TaskDef', {
  memoryLimitMiB: 512,
  cpu: 256,
});
```

ECS Anywhere

Parameters:-




```
const externalTaskDefinition = new ecs.ExternalTaskDefinition(this,
'TaskDef');

const container = externalTaskDefinition.addContainer("WebContainer",
{
  // Use an image from DockerHub
  image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-sample"),
  memoryLimitMiB: 1024,
  // ... other options here ...
});
```

Port mapping:-

To add a port mapping when adding a container to the task definition, specify the portMappings option:

```
portMappings: [{ containerPort: 3000 }],
```

To add port mappings directly to a container definition, call `addPortMappings()`:

```
declare const container: ecs.ContainerDefinition;

container.addPortMappings({
  containerPort: 3000,
});
```

Task Definition for all:-

Note: ECS Anywhere doesn't support volume attachments in the task definition.

To use a TaskDefinition that can be used with either Amazon EC2 or AWS Fargate launch types, use the `TaskDefinition` construct.

When creating a task definition you have to specify what kind of tasks you intend to run: Amazon EC2, AWS Fargate, or both. The following example uses both:

```
const taskDefinition = new ecs.TaskDefinition(this, 'TaskDef', {
  memoryMiB: '512',
  cpu: '256',
  networkMode: ecs.NetworkMode.AWS_VPC,
  compatibility: ecs.Compatibility.EC2_AND_FARGATE,
});
```

To grant a principal permission to run your `TaskDefinition`, you can use the

TaskDefinition.grantRun() method:

```
declare const role: iam.IGrantable;
const taskDef = new ecs.TaskDefinition(this, 'TaskDef', {
  cpu: '512',
  memoryMiB: '512',
  compatibility: ecs.Compatibility.EC2_AND_FARGATE,
});

// Gives role required permissions to run taskDef
taskDef.grantRun(role);
```

Images

Images supply the software that runs inside the container. Images can be obtained from either DockerHub or from ECR repositories, built directly from a local Dockerfile, or use an existing tarball.

- ecs.ContainerImage.fromRegistry(imageName): use a public image.
- ecs.ContainerImage.fromRegistry(imageName, { credentials: mySecret }): use a private image that requires credentials.

- ecs.ContainerImage.fromAsset('./image'): build and upload an image directly from a Dockerfile in your source directory.

Service

A Service instantiates a TaskDefinition on a Cluster a given number of times, optionally associating them with a load balancer. If a task fails, Amazon ECS automatically restarts the task.

```
declare const cluster: ecs.Cluster;
declare const taskDefinition: ecs.TaskDefinition;

const service = new ecs.FargateService(this, 'Service', {
  cluster,
  taskDefinition,
  desiredCount: 5,
});
```