Running Containers on AWS

- Building Scalable and Resilient Containerized Applications with Continuous Delivery and Infrastructure as Code

First published: December 2023

About the author

Victor (Wei) Gu is a Sr. Containers & Serverless Architect at Amazon Web Services. He works with AWS customers to design microservices and cloud native solutions using Amazon EKS/ECS and AWS serverless services. He specializes in Kubernetes, Spark on Kubernetes, MLOps and DevOps.

Victor has strong research and technical writing ability. He published 12 journal and proceeding papers, 3 AWS blogs, and participated in several open-source projects as a key contributor. Victor was invited to speak in KubeCon Europe 2023 on the topic of Data Platform on Kubernetes.

Originally from China, Victor studied and worked in Canada for over 10 years and now he resides in Houston, TX, with his wife and two kids.

Who this book is for

This book is targeted towards Cloud engineers, SREs and solution architects, who are looking to upskill and dominate container management on AWS.

The book is also beneficial for DevOps engineers who want to automate container infrastructure provision and containerized application deployment with CI/CD, GitOps and IaC tools. In order to learn from this book, you should have a basic understanding of containers, Docker, Kubernetes and AWS services.

What you will learn

1	Gain insights into the essentials of AWS container services
2	Run and manage containers on Amazon ECS, EKS, and AWS App Runner
3	Understand the benefits and limitations of using Fargate
4	Explore layers in container image and secure container images with ECR
5	Pick the appropriate AWS container service based on your requirements
6	Deploy containerized applications using CI/CD pipelines and GitOps
7	Deploy container platforms consistently and reliably using IaC
8	Implement monitoring and logging solutions for containerized applications

Book Description

Running containers in AWS provides the added benefit of leveraging the power of the AWS cloud while taking advantage of the benefits of containers technology. By mastering AWS container services like Amazon ECS, EKS, and Fargate, you can streamline your development and deployment processes and deliver reliable, scalable applications.

By supporting portability, consistency and efficiency, Containers technology becomes a valuable tool for organizations that need to develop and deploy applications quickly and reliably across different environments. Additionally, the rise of cloud computing and microservices architectures has increased the demand for containers as a way to manage and orchestrate complex distributed systems.

This book is a comprehensive guide that teaches how to run and manage containers on AWS by leveraging all the best practices learned from years of customer interactions. You will learn about the container service landscape on AWS, understand the features of each service and then be able to pick the proper services for your needs. You will get a deeper understanding on how Amazon ECS, EKS and Fargate support containers in terms of scalability, availability, agility and security. Filled with real-world examples, the book shows how to deploy container platforms quickly, consistently, and reliably using Infrastructure as Code and observe platform data using monitoring and logging tools.

By the end of this book you will effectively set up container platforms in AWS using IaC tools and deploy containerized applications with CI/CD or GitOps tools.

Table of Contents

1

Navigating the Container Landscape on AWS with Registry and Orchestration

Running containers on AWS provides the added benefit of leveraging the power of the AWS cloud while taking advantage of the benefits of containers technology. As I am writing the book, nearly 80 percent of all containers in the cloud run on AWS. Mastering AWS container services makes easier for you to run applications in the cloud, so you can focus on innovation and your business needs. In the first chapter, we will start with the basics of container, including its key components and the benefits of working with container. We will navigate the various services on AWS that support storing docker images secretly and managing when and where your containers run.

In this chapter we're going to cover the following main topics:

- Container 101
- Container Image registry
- Container orchestration services

Container 101

To run containers on AWS, let's first understand what is container and the key components in the container ecosystem.

What is container?

"A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings."

- *From Docker Company*

https://www.docker.com/resources/what-container/

The preceding paragraph is the most popular definition of container. It mentions two key concepts:

Container, a running software which doesn't reply on any external dependencies.

Container image, a static executable package which will become container(s) at runtime. One container image can be executed as one or many containers in one or different physical machines.

To execute container image as container on a host operating system, you need a **container engine or runtime**. Docker is the most used engine, but there are some other options and we will explore them at the end of this section.

Important note

You may hear Docker everyday but it may mean different things each time.

First, Docker means Docker Inc., the company who started the container frenzy and introduced an explosion of new container tools, standards and APIs.

Second, Docker means the engine or runtime for creating container from container image. It actually has multiple layers where runc is the low-level runtime, containerd is the high-level runtime, and a docker daemon process.

Third, Docker means Docker CLI tool that gives you the ability to interact with the Docker daemon using docker commands.

As we've come to understand, Docker is just one component of the container ecosystem. With containers becoming increasingly popular and more companies entering this domain, standards are necessary to enhance interoperability within the ecosystem. This enables software to be run on various platforms and operating systems, reducing reliance on a single company or project. Currently, there are two major **container standards**:

- **Open Container Initiative (OCI):** a set of standards for defining the container image format, container runtime, and distribution. The OCI was established in 2015 by Docker and was one of the first efforts at creating standards for the container ecosystem. The OCI specifications standardize what a container is and what it should be able to do.

  Container Runtime Interface (CRI): an interface used by Kubernetes to control the different runtimes that create and manage containers. The CRI describes how Kubernetes will interact with any runtime. As long as a container runtime implements the CRI API, the runtime can have its unique low-level implementation on creating and starting containers.

  Figure 1.1 shows the relationship between container standard and container runtime. The orange entities are the standards and runtimes initialized by Docker while green entities are the standards and runtimes initialized by Kubernetes community. **containerd** is a high-level container runtime that came from Docker and now an open source project. It implements both OCI and CRI standards. It pulls images from image registries, manages them and then hands over to a lower-level runtime, which uses the features of the Linux kernel to create container processes. **CRI-O** is another high-level container runtime which implements CRI standard. It was born out of Red Hat, IBM, Intel, SUSE and others. **runc** is a low-level container runtime used by both containerd and CRI-O. By following OCI standard, runc provides all the low-level functionality for containers, interacting with low-level Linux features, like namespaces and control groups:

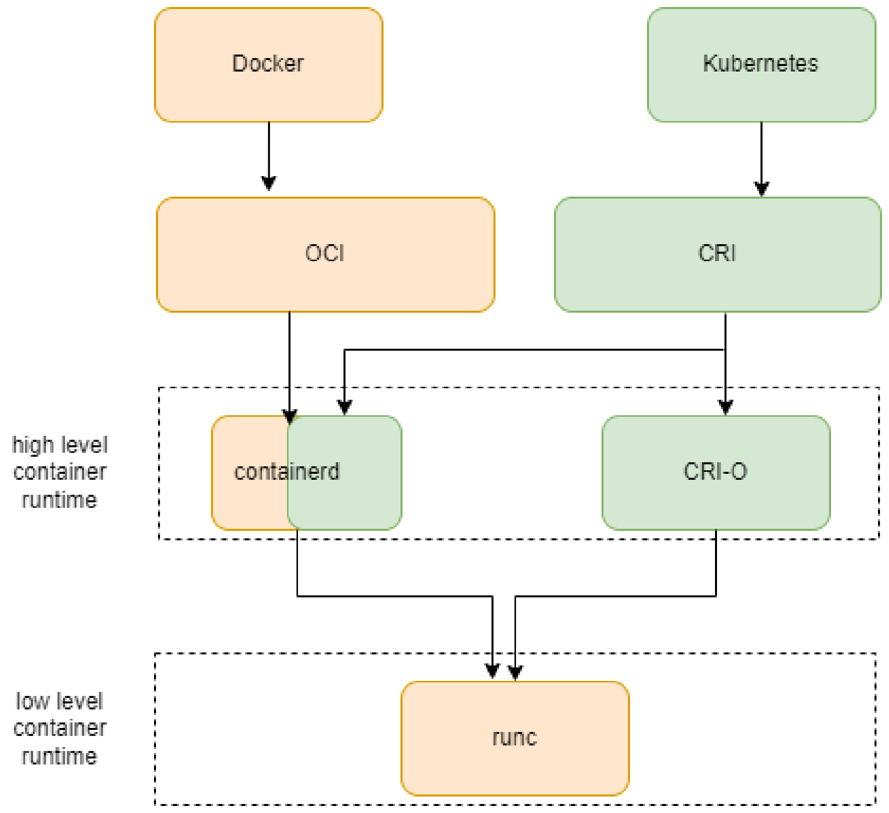


Figure 1.1 - The relationship between container standard and container runtime

Note

You may hear the news that "*Kubernetes is deprecating Docker*". What happened is Dockershim has been removed as of Kubernetes v1.24. Dockershim is a little software shim that allows kubelet to use Docker. After removing Dockershim, kubelet will use containerd or CRI-O as its container runtime.

What are the benefits of containerization?

Containers offer several benefits for software development and deployment:

Portability: Containers offer a high level of portability since they bundle all the dependencies and libraries required to execute an application. This ensures that the application can be transferred smoothly from one machine to another, or from a development environment to a production environment without any concerns about compatibility problems.

Consistency: Containers offer a uniform runtime environment, irrespective of the underlying infrastructure, ensuring that applications operate consistently across various environments. As a result, it becomes simpler to test, deploy, and maintain applications, regardless of the underlying infrastructure.

Lightweight and Scalability: Containers are designed to be minimal and efficient, with only the essential components required to run the application included. This makes them significantly smaller than traditional virtual machines, which need to include an entire operating system and its dependencies. Because of lightweight, containers can be quickly duplicated and scaled up or down in response to demand, allowing for the efficient management of sudden surges in traffic or workload without impacting the application's performance. The lightweight feature also makes it simpler to scale applications horizontally by deploying additional container instances rather than vertically, which involves upgrading the server's hardware.

Security: Containers offer an additional layer of security by isolating applications from one another and from the underlying infrastructure. This isolation helps prevent unauthorized access, data leaks, and other security breaches that can compromise the application and its data.

In general, containerization can apply to a broad range of applications. Some applications are better suited for containerization than others. Applications that are stateless, modular, and can run independently are ideal for containerization. Some application types are:

- **Web applications**: Web applications typically comprise multiple components, such as front-end web components, back-end APIs, and databases, each of which can be containerized independently. This provides greater flexibility and agility for development teams, as they can quickly and easily move the partial of the application from one environment to another.

- **Microservices**: by containerizing microservices, each component can operate within its own container, which provides isolation and minimizes the risk of failures that could affect other components. This approach also allows for easier testing and deployment of individual microservices, without affecting the entire application. In addition, containerization enables efficient scaling of microservices as it permits easy duplication and replication of the containers that host individual microservices. This can be done quickly and seamlessly in response to changes in demand, without affecting the performance of the other microservices.

- **DevOps tools** (**like Jenkins and Argo CD**): containerization allows for faster deployment and scaling of deployment agents, as new containers can be spun up and down quickly to meet changing demands. This makes it easier to handle sudden spikes in build or deployment queues.

- **Machine learning applications**: Machine learning applications that use machine learning libraries and frameworks like TensorFlow, PyTorch, and Scikit-learn can also be containerized. Containerization provides a more flexible and efficient way of deploying, scaling, and maintaining these applications, making it easier for data science teams to manage their machine learning workflows.

How can I get started?

In this section, you will learn how to set up the tools to create container images and run container in your local host.

Docker

As mentioned previously, Docker is the most used runtime for container image and Docker CLI is the most used command line for building container image, sending images to image registry and calling runtime to execute images.

As shown in *Figure 1.2*, when working with Docker, you use Docker CLI, which communicates with Docker daemon in the background. Docker daemon is responsible for building container image at local or fetching images from remote container image registry. Docker daemon can operate on containers and host kernel through containerd and runc:

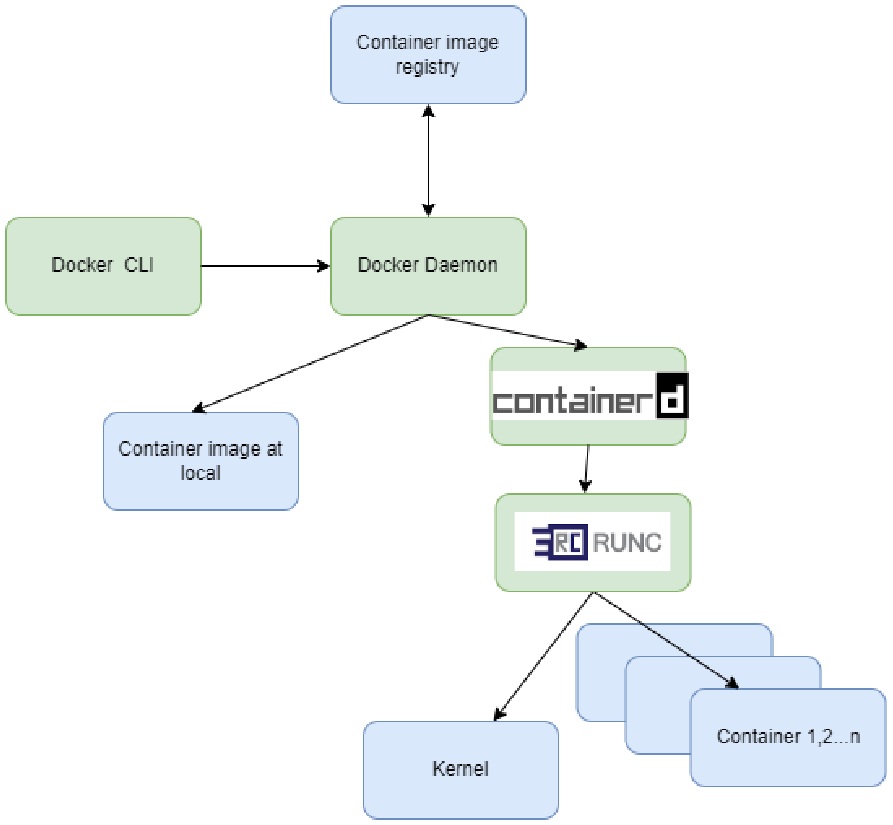


Figure 1.2 - Docker workflow

Docker offers two product types: Docker Engine and Docker Desktop.

- Docker Engine is an open source containerization platform that supports packaging, deploying and running your applications in containers. Docker Engine is available on a variety of Linux platforms, macOS and Windows 10. In Linux, it supports Ubuntu, Debian and Fedora. You can get the installation details at https://docs.docker.com/engine/install/

- **Docker Desktop** includes Docker Engine and provides a straightforward **Graphical User Interface** (**GUI**) that lets you manage your containers, applications, and images directly from your local host. Docker Desktop can be used either on its own or as a complementary tool to the CLI. You can get the installation details at https://docs.docker.com/desktop/.

Podman

Podman is a daemonless open source tool for developing, managing, and running containers on your host machine. It was first created by Red Hat in 2018 and now is open source and free to download.

As shown in *Figure 1.3*, Podman manages containers by interacting directly with the Linux kernel using the runc container runtime process instead of relying on a daemon process. Podman relies on a library called containers/images for pulling container images from container registry servers. It also uses containers/storage to manage images on disk on the local host. Podman is compatible with Docker but provides numerous security advantages, due to its capability to operate as an unprivileged user (rootless) and to function without the daemon process (daemon-less).

Podman can be installed on Linux, macOS and Windows. You can get the installation details at https://podman.io/getting-started/installation.

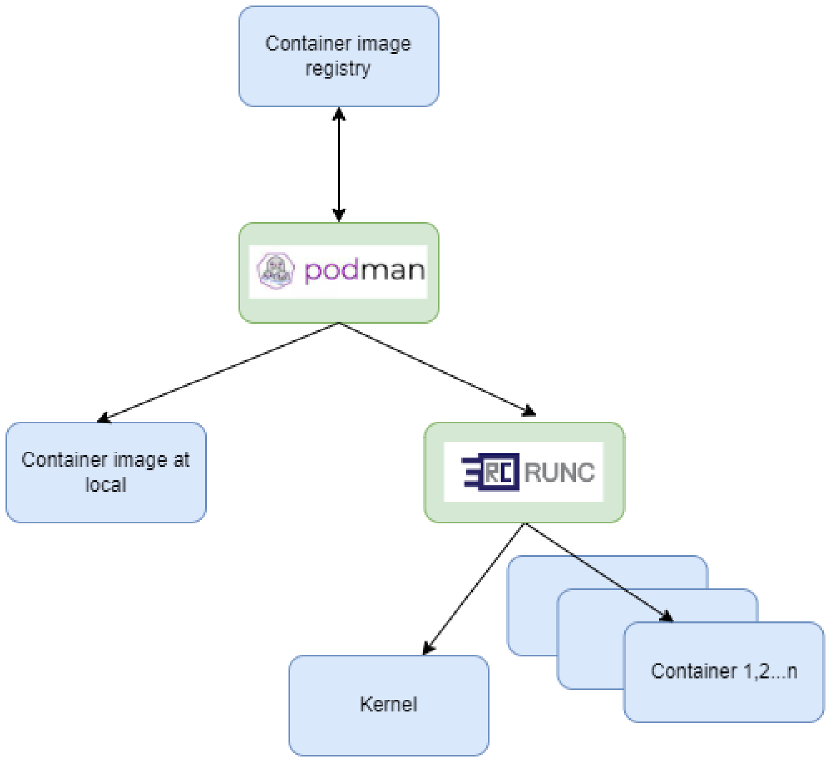


Figure 1.3 - Podman workflow

In this section, we provided an overview of the container ecosystem and introduced key concepts, such as containers, container images, container runtimes, and container standards. It's important to note that Docker is just one tool among many for running containers. We also discussed the benefits of using containers and identified the types of applications that are suitable for containerization. In the next section, we will begin exploring AWS services for containers.

Container image registry

To run containers on AWS, you need to first push your container images to a place where can be accessed by AWS services. A **container image registry** is such a storage system that allows you to store and distribute container images. A container image registry acts as a centralized location that enables various computing services to retrieve images, and for developers to upload images from their local host. Container image registries can be hosted either on-premises or in the cloud. You can use managed registry service from cloud like Amazon ECR or build your own with open source solution like Harbor. In this section, we will start with Amazon ECR and move to other alternative solutions.

Amazon Elastic Container Registry (ECR)

Amazon ECR is an AWS managed container image registry service that is secure, scalable, and reliable. You can use your preferred CLI (like Docker or Podman) to push, pull, and manage OCI images, and OCI compatible artifacts in ECR. The key features of ECR are:

- **Secure**: Amazon ECR provides secure and private container image management by integrating with **AWS Identity and Access Management** (**IAM**) to manage user permissions and policies. Furthermor, it supports Image scanning which helps detect software vulnerabilities in your container images. With image scanning, each newly pushed image can be scanned, and the scan results can be retrieved from Amazon ECR or Inspector to help you identify any security issues.
- **Highly available**: Amazon ECR is a highly available service that can replicate container images across cross-region and cross-account to provide high availability and durability.
- **Lifecycle policies**: Amazon ECR allows you to define lifecycle policies to manage the retention and deletion of container images based on criteria such as image age, tag status, or total number of images. This will help to clean up unused images.
- **Helm chart support**: besides container image, ECR also supports Helm chart (a set of YAML manifests for deploying Kubernetes applications).
- **Public Repository**: ECR public allows you to publicly share and distribute your container images with anyone around the world.
- **Cost-effective**: ECR offers a cost-effective pricing model where you only pay for the amount of data you store in the registry and the data transfer out of the registry. Additionally, there are no upfront costs or minimum fees associated with using ECR. AWS customer also gets free storage as part of the AWS Free Tier.

You will learn more about ECR in *Chapter 4*, such as how to upload container images and Helm charts, perform security scans, and sign images. Moving forward, we will examine other container registry options that are compatible with the AWS environment.

Other top container registries

Although not as seamlessly as ECR, most container registries available in the market can still function with AWS services. We have compiled a list of popular options and will provide a comparison table at the end.

Docker Hub Container Registry

Docker Hub is one of the most widely used container registries on the market that allows developers to store and share Docker container images. **Docker Hub** provides a free tier for public repositories and a paid subscription service for private repositories with additional features. The pricing is based on the amount of storage and data transfer used.

Docker Hub can integrate with GitHub for automated builds and deployments. It also offers webhooks, allowing for automated workflows triggered by new container image pushes, as well as automated scanning for vulnerabilities and compliance issues.

Harbor Container Registry

Harbor was created by VMware in 2014, switched to an open-source model in 2016 and now is maintained by the **Cloud Native Computing Foundation** (**CNCF**). It is a container registry that requires the installation, configuration and management from the user.

Harbor offers role-based access control for managing user access, supports vulnerability scanning with various tools, enables image signing and verification using Notary, and provides replication and synchronization across multiple instances for better distribution and redundancy. Harbor also provides a web-based GUI for managing repositories, users, and images, along with API and webhook support for automation and integration with other systems.

Red Hat Quay

Red Hat Quay is a container registry that was created by CoreOS in 2014 and now is maintained by Red Hat. Quay has several distributions which may be confusing sometimes.

- Project Quay: an open-source version of Quay that allows developers to store, manage, and distribute container images.
- Red Hat Quay.io: a commercial container registry product that is hosted on Red Hat's cloud infrastructure. It offers additional enterprise-level features like role-based access control, audit logging, and automation capabilities.
- Red Hat Quay: another commercial container registry product from Red Hat but for private-cloud deployments. It can be deployed on Red Hat OpenShift as a built-in Operator.

The following table compares all the container registries mentioned previously with ECR:

	Amazon ECR	Docker Hub	Harbor	RedHat Quay
Authentication	AWS IAM	Password or Access Token	AD, LDAP, RBAC, and OIDC	LDAP, Keystone, OIDC, Google and Github
Cross-region replication	✓	✗	✓	✓
MFA for Image Push/Pull	✓	✓	✓	✗
SLA Availability	99.9%	n/a	self-hosted	n/a
Garbage collection	✓	✗	✓	✓
Image Scanning	✓ free	✓ free but limited	✓	✓

Table 1.1- Comparison of container images registries

Based on the comparison, Amazon ECR offers the best service level agreement (SLA) and provides the most managed container services. Considering that it can seamlessly intergate with other AWS servies, Amazon ECR is your first container registry choice in AWS environment.

Container orchestration services

After obtaining container images and an image registry to store them, the subsequent requirement is a container orchestration service that manages the underlying infrastructure to execute the containers. AWS provides a range of container orchestration services catering to diverse technical and business needs.

Amazon Elastic Container Service (ECS)

Amazon ECS is a fully managed container orchestration service that helps you easily deploy, manage, and scale containerized applications. Launched in 2014, Amazon ECS was the first AWS container service that provided native support for containers. Amazon ECS provides a simple and intuitive API for deploying and managing containers, making it easy for developers and DevOps teams to get started quickly.

Figure 1.4 shows the key components in Amazon ECS:

An **Amazon ECS cluster** is a logical grouping of tasks or services. An ECS cluster needs to be inside Amazon **Virtual Private Cloud** (**VPC**).

Task definition(s) is a `JSON` format file that describes one or more containers that form your application. It comprises several parameters, including the Docker image to use, resource requirements (such as CPU and memory), port mappings, environment variables, IAM roles, and networking settings.

Task is the instantiation of a task definition within an ECS cluster. When an ECS task is launched, it is assigned a unique task ID, and its state can be monitored and managed through the ECS console or API. Tasks can be launched as part of a service or individually as a standalone task. They can also be scaled up or down based on demand, and new tasks can be launched automatically in response to failures or load spikes.

Service is a logical group of tasks that are deployed and managed together in an ECS cluster. An ECS service enables you to specify the number of tasks to run, the task definition to use, and how to distribute traffic across the tasks.

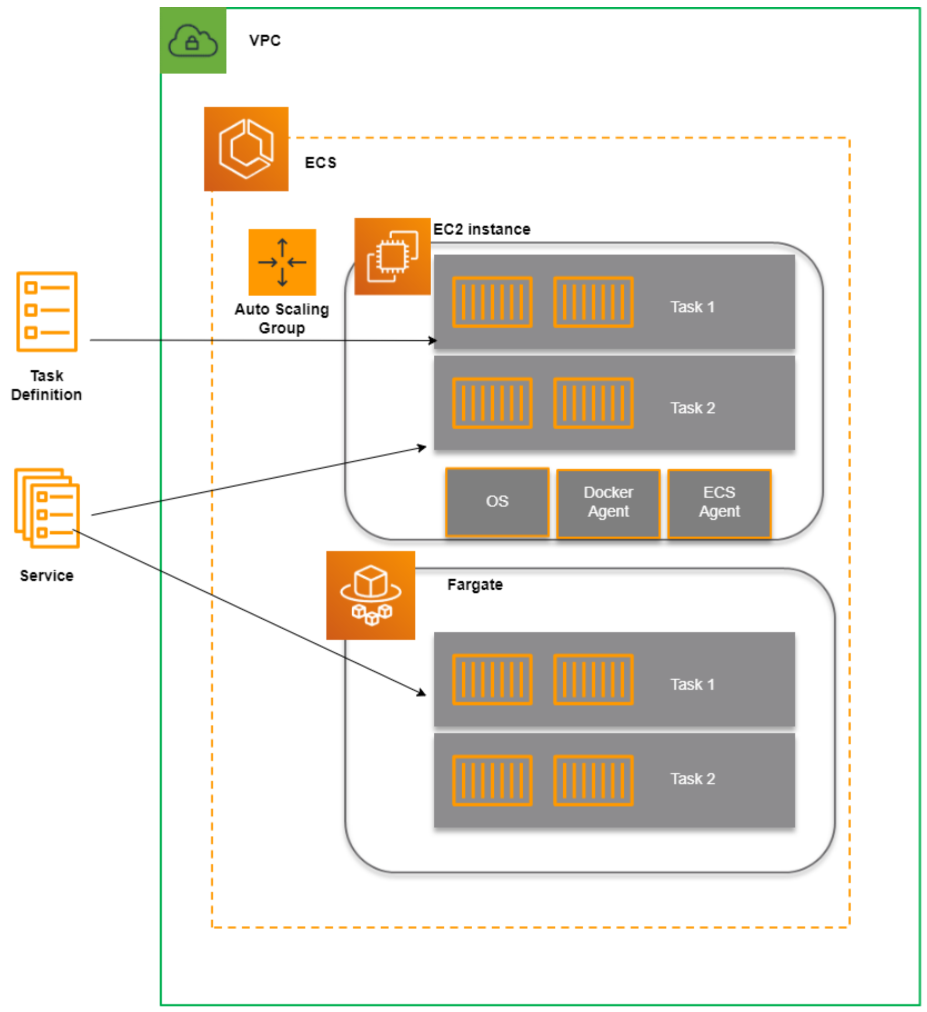


Figure 1.4 - The overview architecture of ECS

Elastic Compute Cloud (EC2) instance is one compute capacity you can use in ECS. When you launch EC2 instances for ECS, you need to ensure that they have the ECS agent installed and running. The ECS agent is a component that runs on each EC2 instance in an ECS cluster and communicates with the ECS service to manage containerized workloads. It helps to launch and stop containers, monitors the health of containers, and reports back to the ECS service about the state of the containers.

When use EC2 as compute provider, you also need to use Auto Scaling Groups to provide managed scale-in and scale-out actions. Auto Scaling Groups in ECS work by defining a set of rules that specify the desired minimum, maximum, and desired number of EC2 instances in the ECS cluster. These rules are based on metrics such as CPU usage or network traffic, and they are used to determine when to add or remove EC2 instances from the cluster.

Fargate is another compute capacity in ECS that allows you to run containers without having to manage the underlying EC2 instances. With Fargate, you simply define your containerized application and its resource requirements, and AWS automatically provisions the necessary compute resources to run the application. You don't have to worry about scaling or managing the underlying infrastructure, as AWS takes care of it for you.

Chapter 5 provides comprehensive details on ECS scaling, configuration, and a practical demonstration of a microservice implementation using ECS.

Note

While Fargate relieves users of the need to provision and manage their own servers, the service itself relies on servers provided by AWS to host containerized applications. It's important to note that Fargate only supports running containers on certain operating systems, including:

Amazon Linux 2

Windows Server 2019 Full

Windows Server 2019 Core

Windows Server 2022 Full

Windows Server 2022 Core

Figure 1.5 shows two ways to trigger tasks in ECS from outside. *Figure 1.5 (a)* shows to use Elastic Load Balancing to distribute external traffic to your tasks in ECS. Your tasks need to be registered under a target group of the load balancer first. Elastic Load Balancing has three types: **Application Load Balancer** (**ALB**), **Network Load Balancer** (**NLB**) and **Classic Load Balancer**. We recommend that you use ALB for your Amazon ECS tasks so that you can take advantage of these latest features, like path-based routing and priority rules.

Figure 1.5 (b) shows the approach to enable Amazon ECS service discovery by using AWS Cloud Map and DNS records in Route 53. This involves registering your tasks with Cloud Map, and then creating a Route 53 DNS record that enables other applications to discover your tasks. Other applications can then query the Route 53 DNS to retrieve the endpoint of your ECS tasks, which they can use to establish a connection.

Chapter 2 will provide a thorough guide to network services that facilitate container communication, such as Elastic Load Balancer, Cloud Map, and Route 53.

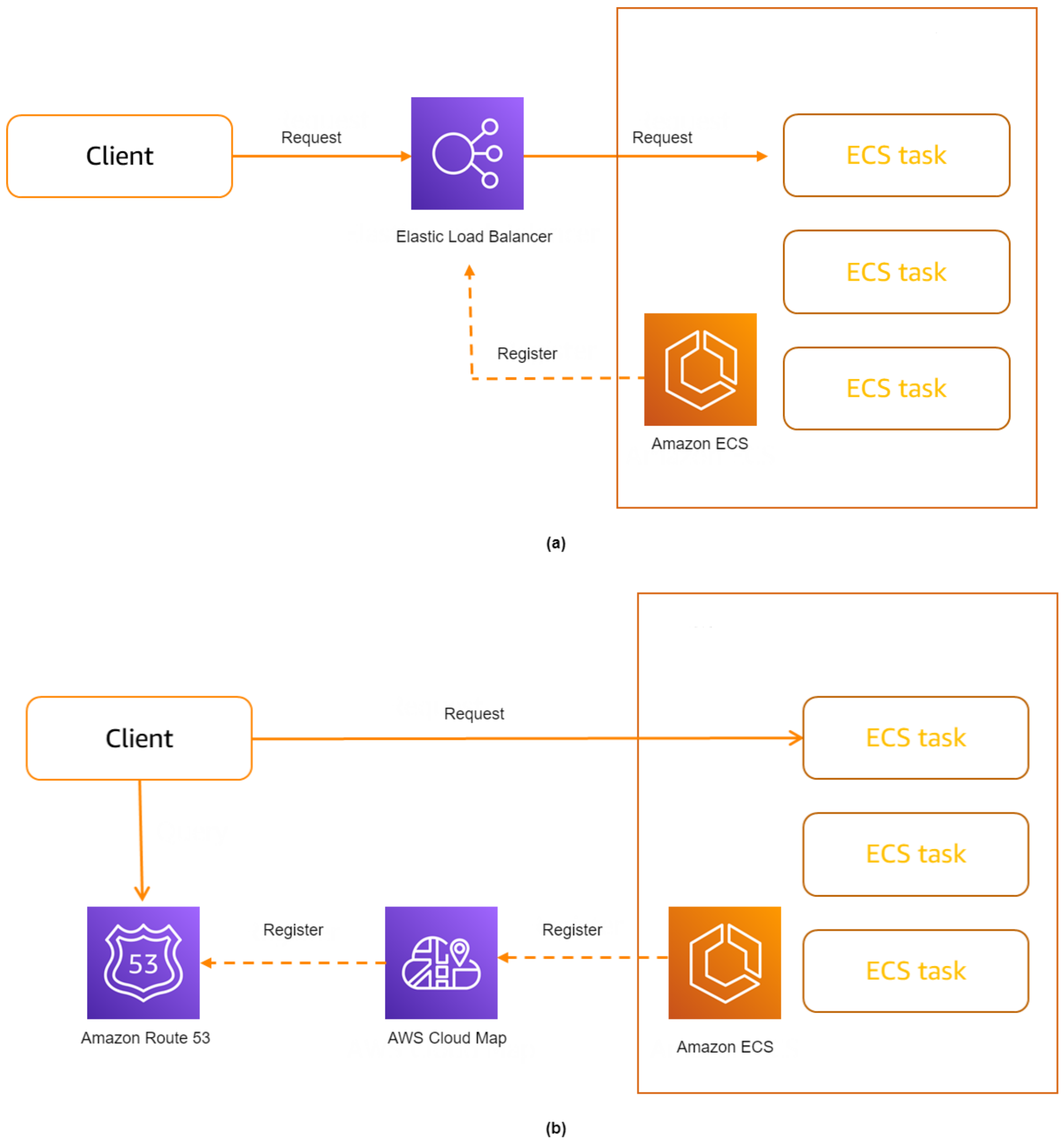


Figure 1.5 - ECS service connection options

There are several ways to get started with ECS. The first way is to use AWS console. You need to create the cluster and then follow the steps to create task definition and service.

Secondly, you can use AWS Copilot CLI, a command-line interface tool to build, deploy, and operate containerized applications on ECS. The AWS Copilot CLI automates the process of creating and configuring ECS clusters, networking, and load balancers for your applications, as well as deploying, scaling, and monitoring your containers. It also provides features such as automated SSL/TLS certificate generation and renewal and automatic application scaling. You can get more details about the AWS Copilot at https://docs.aws.amazon.com/AmazonECS/latest/developerguide/getting-started-aws-copilot-cli.html

Thirdly, you can use Infrastructure-as-Code tools like AWS Cloud Development Kit (CDK) or Terraform to create ECS clusters, tasks and services. While AWS Copilot CLI is designed specifically for ECS, AWS CDK and Terraform can be used to manage a wide variety of cloud services. This means that DevOps teams can use these tools to create and maintain nearly all the AWS services in their account. More detailed information on Infrastructure-as-Code for AWS container services can be found in *Chapter 12*.

Amazon Elastic Kubernetes Service (EKS)

Kubernetes has become the standard for container orchestration due to its open-source nature, scalability, flexibility, portability, and strong community support. It allows for the management of large-scale containerized applications and provides a consistent deployment experience across different environments.

Amazon **Elastic Kubernetes Service** (**EKS**) is a fully managed service that makes it easy to run Kubernetes on AWS. It eliminates the need for you to install, operate, and maintain your own Kubernetes clusters. Amazon EKS uses upstream Kubernetes code to provide the latest features and security updates, and also ensures that the Kubernetes API is compatible with the upstream version. EKS customers can use the same Kubernetes tools and APIs they are familiar with to deploy and manage their applications and can also take advantage of the extensive Kubernetes ecosystem, including a wide range of open-source tools and plugins.

As shown in *Figure 1.6*, the major components of Amazon EKS include:

- **EKS Control Plane**: This is the management layer of the Kubernetes cluster that runs the Kubernetes API server, etcd, and other control plane components. With Amazon EKS, AWS operates and manages the control plane for you, so you can focus on running your applications. The EKS Control Plane is highly available and scalable, with multiple copies of the Kubernetes API server and etcd running across multiple **Availability Zones** (**AZs**) in Amazon managed VPC to ensure resilience and fault tolerance. AWS also provides automatic updates and patches to the Control Plane to ensure that it is always up-to-date and secure.
- **Worker Nodes**: Your containerized applications are deployed on EC2 instances in your VPC. These instances are registered as nodes in your EKS cluster and are managed by the EKS control plane. Each node has the following components:
    1. container runtime: a software package that is for pulling container images from registry and running the containers.
    2. kubelet: an agent that monitors the status of its hosting node and manage node-level resources such as disk and network usage.
    3. kube-proxy: a network proxy that manages network connectivity between different pods and services by maintaining network rules.

- Node group: a set of Amazon EC2 instances that are used to run as Kubernetes worker nodes. Each node group can be thought of as a logical grouping of EC2 instances that share the same instance type, AMI, and scaling configuration. Node groups can be either managed node groups or self-managed node groups. Managed node groups are fully managed by EKS, while self-managed node groups are managed by the customer.
- Fargate: besides EC2 instance, Amazon EKS can also run containers on AWS Fargate which is a serverless computing engine that allows you to run containerized workloads with on-demand and optimized compute capacity, without the need to manage the underlying infrastructure or resources. When using Fargate with EKS, customers need to create and manage Fargate profiles, which are used to define the Kubernetes namespaces and selectors for pods that should be scheduled on Fargate. In *Chapter 8*, we will explore the details of using Fargate with both ECS and EKS and dive deep into the benefits and limits of using Fargate.

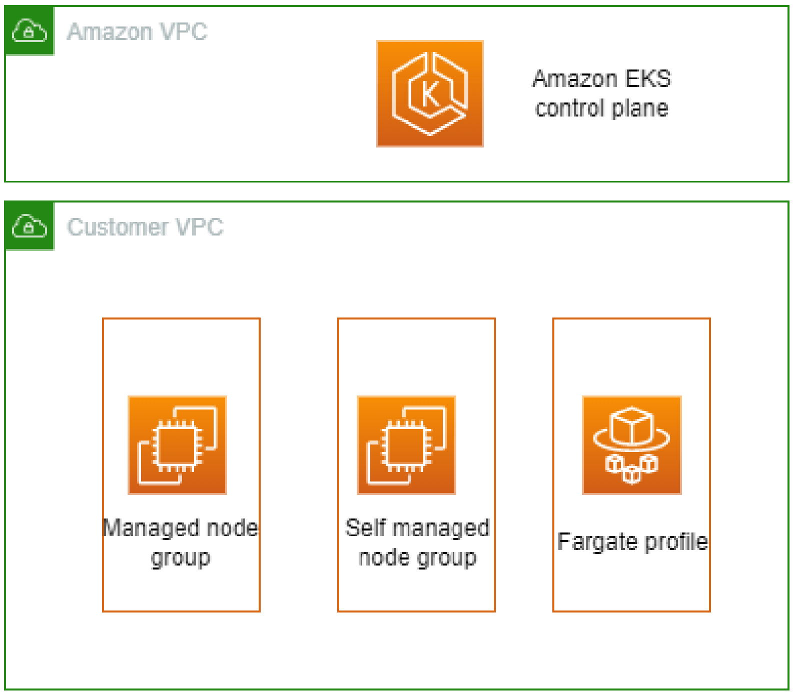


Figure 1.6 - the overview architecture of EKS

Chapters 6 and *7* will cover EKS cluster autoscaling, high availability deployment, secured service connections and more. These topics will be presented through a hands-on lab that will provide practical experience with these concepts.

Figure 1.7 shows one example of using EKS to host containerized applications. The application image is deployed as pods in EC2 instances in two different availability zones for high availability. A service object is created for providing a stable IP address for accessing the set of pods. Elastic load balancers can be created in public subnets for routing external traffic to the cluster internal service:

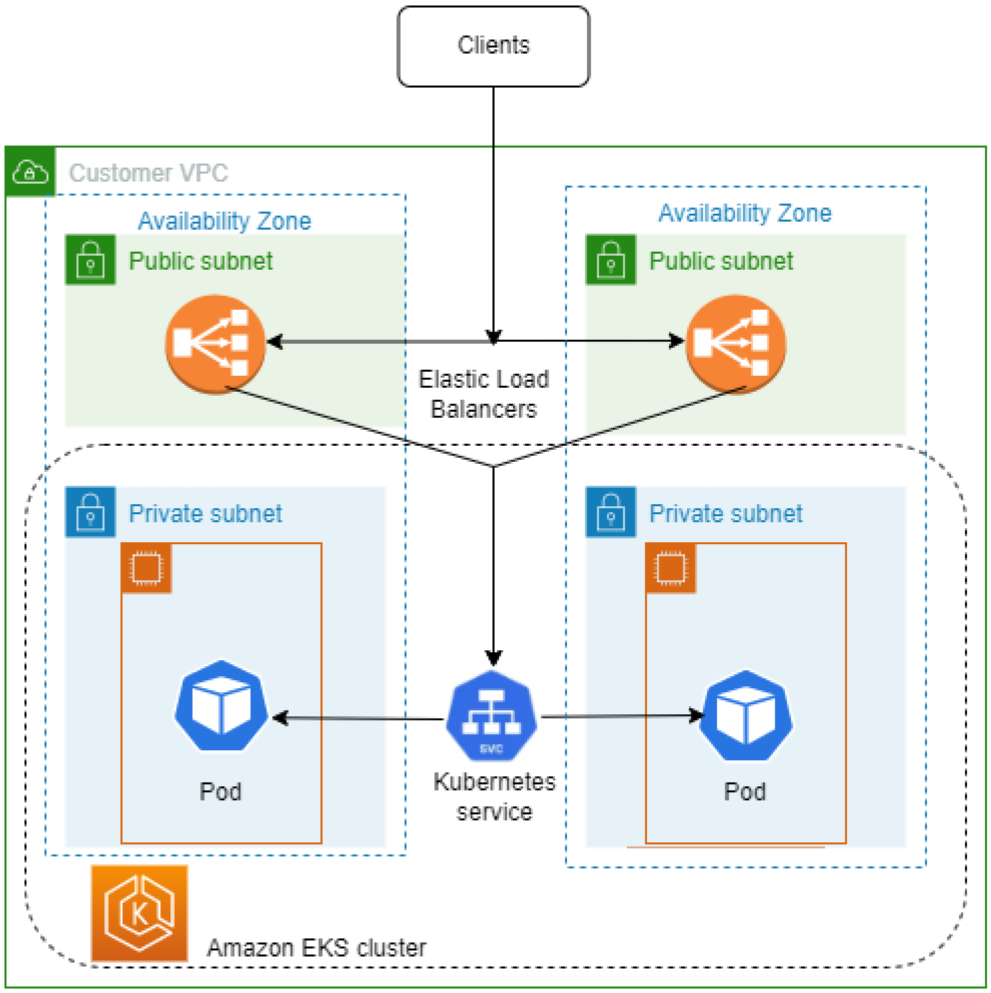


Figure 1.7 - EKS Application Networking

Do you know?

AWS offers different EKS distribution options which allow you to install and use EKS under different infrastructures:

Amazon EKS in Local Zones: EKS in Local Zones is a deployment option that enables customers to run Kubernetes clusters in a specific geographic location close to end-users. With EKS Local Zones, customers can run Kubernetes in a single, highly available zone located in metropolitan areas, reducing application latency and providing higher performance.

Amazon EKS on Outposts: EKS on Outposts is a fully managed service that enables customers to run Kubernetes on AWS Outposts, which are customizable compute and storage racks built with AWS hardware. EKS on Outposts enables customers to utilize native AWS services, infrastructure, and operating models in any data center, co-location space, or on-premises facility.

Amazon EKS Anywhere: EKS Anywhere is a deployment option for Amazon EKS that allows customers to create and operate Kubernetes clusters on-premises, in their own data centers, or in the cloud. With EKS Anywhere, customers can use the same EKS APIs, tools, and ecosystem to manage their Kubernetes clusters anywhere they choose.

You can get started with Amazon EKS with AWS console or `eksctl` command-line tool. `Eksctl` automates many of the tasks involved in setting up and configuring an EKS cluster, allowing you to easily create and manage clusters using a simple `YAML` configuration file. For more information about `eksctl`, please go to https://docs.aws.amazon.com/eks/latest/userguide/getting-started-eksctl.html

AWS App Runner

AWS App Runner is a fully managed service which simplifies the deployment of containerized web applications and APIs for developers without requiring prior experience with containers. With App Runner, the web application is automatically built, deployed and exposed through load balancer with encryption. It integrates with source code repositories like GitHub or AWS CodeCommit, and container registries like Amazon ECR.

Figure 1.8 shows the major components in App Runner:

- **App Runner service**: refers to a deployed version of your application. App Runner uses it to deploy and control the application based on either source code repository or container image.
- **Repository provider**: the repository service that can build and deploy container images from source code stored in a Git repository or directly deploy container image from ECR.
- Runtime: A base image used to deploy a source code repository. App Runner includes various managed runtimes for different programming platforms, such as Python, Node JS, Java, .NET, PHP Ruby and Go.
- **App Runner managed AWS resources**, App Runner will also create and manage AWS resources for running your applications, like Application Load Balancer for networking, Auto Scaling group for underlining EC2 instance auto scaling. In each EC2 instance, App runner will install and maintain container runtime, orchestration agent and Operation System.

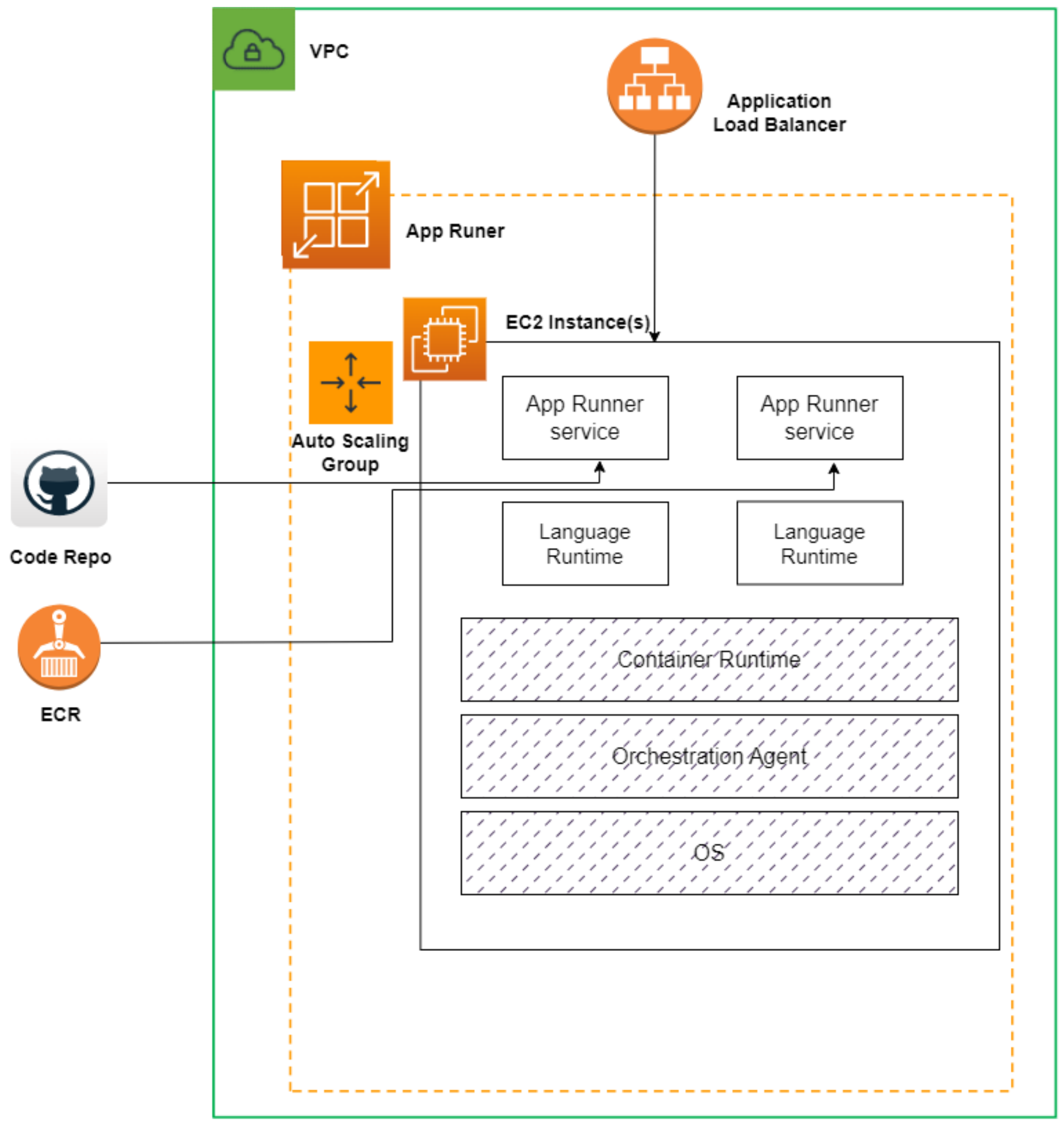


Figure 1.8 - the overview architecture of AWS App Runner

Compared with ECS and EKS, AWS App Runner provides a quick and easy way to deploy containerized web applications and APIs while reducing the operational overhead further. It also provides monitoring and logging capabilities, which allow developers to track performance and troubleshoot issues easily.

You can use the App Runner by simply following the instructions in AWS console. The major steps are:

- 1. Configure the connection between your source code repository or ECR image repository with App Runner.
- 2. Set up the deployment option (manual or automatic) and provide build commands for your application.

- 3. Configure your App Runner service with Environment variables, virtual CPU and memory, Auto scaling, IAM roles, networking and etc.

You can also put those settings into an `apprunner.yaml` configuration file and use AWS CLI to create App Runner service base on the configuration file. The example `apprunner.yaml` is like the following:

```
version: 1.0

runtime: python3

build:

  commands:

    build:

      - pip install pipenv

      - pipenv install

  env:

    - name: DJANGO_SETTINGS_MODULE

      value: "django_apprunner.settings"

run:

  runtime-version: 3.7.7

  command: pipenv run gunicorn django_apprunner.wsgi --log-file -

  network:

    port: 8000

    env: MY_APP_PORT

  env:

    - name: MY_VAR_EXAMPLE

      value: "example"
```

Other AWS services for running containers

AWS offers several services that allow you to run containerized applications, in addition to container orchestration services like ECS and EKS. AWS Lambda and AWS Elastic Beanstalk are two such services that provide a higher level of abstraction by hiding the container orchestration layer. This allows you to perform less control over container scheduling and configuration.

AWS Lambda

AWS Lambda can be used to run containerized applications. To run a containerized application on Lambda, you need to package your application as a Docker image and upload it to a container registry. You can then use the AWS Lambda console, CLI, or SDK to create a Lambda function and specify the container image to use. When the Lambda function is invoked, AWS will create a new container instance and run your application inside it. You need to pay attention on the following things when using containers on Lambda:

- **Base images**: you have to choose the base images that are packed with a runtime interface client which will be used to interact with your function code with Lambda platform. AWS provides various base images for all the supported Lambda runtimes (Python, Node.js, Java, .NET, Go, Ruby).
- **Image type and permission**: Lambda is compatible solely with Linux-based container images and does not support multi-architecture container images. It is important to ensure that the container image you choose can operate on a read-only file system. By default, your function code can access a writable "/tmp" directory with the storage between 512 MB and 10,240 MB in increments of 1-MB.
- **Image size**: Lambda supports a maximum uncompressed image size of 10 GB, including all layers.
- **Other limits on Lambda**: the well-known limits on Lambda functions also apply on running containers on Lambda, such as cold start and 15 minutes time out (can be extended to 30 minutes). You should consider those limits during the design.

AWS Elastic Beanstalk

AWS Elastic Beanstalk is a fully managed service that makes it easy to deploy, manage, and scale web applications and services. It abstracts away the underlying infrastructure and automatically handles the deployment, scaling, and monitoring of applications.

AWS Elastic Beanstalk supports a variety of application architectures, including containerized applications. By using container, you can define your own runtime environment, application dependencies and configurations inside container image. The containers running inside the Elastic Beanstalk can get values from the environment variables defined in the Elastic Beanstalk console. Elastic Beanstalk will only handle capacity provisioning, load balancing, scaling, and application health monitoring.

AWS Elastic Beanstalk can actually use ECS to coordinate container deployments. As shown in *Figure 1.9*, Elastic Beanstalk automatically creates and configures ECS resources while building the environment. We only recommend using AWS Elastic Beanstalk for container deployment to those who are already AWS Elastic Beanstalk users and are looking to transition to containerization. For new users, we recommend exploring alternative AWS container services:

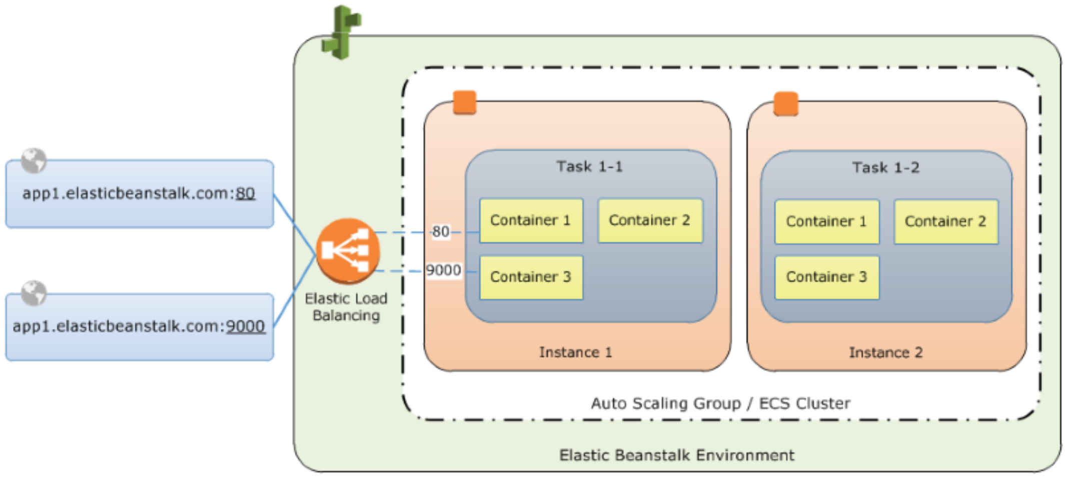


Figure 1.9 - AWS Elastic Beanstalk uses ECS to container platform (source: AWS Elastic Beanstalk Developer Guide)

Summary

In this chapter, we discovered the key concepts in the container ecosystem, such as container, container image, container runtime and container standard. We also explained the difference between container and docker and understand the main standards for container, one from Docker and one from kubernetes community. The benefits of using containers and the types of applications that are suitable for containerization are also being discussed.

To operate containers, specific tools are required. In your local host, you can use Docker or Podman to build and manage container images and run them as containers. In cloud environments, AWS provides various services for container image management and orchestration. To store container images, AWS provides ECR, but other alternative tools like Docker Hub, Harbor, and Red Hat Quay can also be utilized. For container orchestration, AWS offers ECR, EKS, and App Runner. Additionally, containers can be run on AWS Lambda or AWS Elastic Beanstalk.

Further reading

For more information on the topics covered in this chapter, please refer to the following:

- **Open Container Initiative** (**OCI**): https://opencontainers.org/
- **Container Runtime Interface** (**CRI**): https://kubernetes.io/docs/concepts/architecture/cri/
- Getting Started with Podman: https://podman.io/getting-started/
- Dockerfile best practices: https://github.com/hexops/dockerfile
- Amazon ECR User Guide: https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html

- Amazon ECS Best Practices Guide: https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/intro.html
- Amazon EKS User Guide: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html
- AWS App Runner User Guide: https://docs.aws.amazon.com/apprunner/latest/dg/what-is-apprunner.html

4
Storing and Managing Container Images and Helm Charts with Amazon ECR

To run your container workloads in AWS, you first need a place to store your container images and allow other AWS services to access those images securely. In *Chapter 1*, we briefly walked you through the container registry tools on the market. Amazon ECR, a fully managed Docker container registry from AWS, is recommended since it can be seamlessly integrated with the other AWS container services. In this chapter, we will start with outlining best practices for building container images, and then we will look at Amazon ECR closely, after which we will delve into Amazon ECR to explore its various features for image storage and access. We will also give you the image repository design best practice. We'll also provide guidance on designing image repositories based on best practices. Lastly, we'll demonstrate advanced usage patterns with Amazon ECR, aimed at advancing your container image building skills to new heights.

The content in this chapter is organized in the following manner:

- Container image build best practices
- Amazon ECR dive deep
- Advanced usage with Amazon ECR

Technical requirements

Starting in this chapter, we will reinforce your learning by doing hands-on labs. To make a consistent programming environment, we will first help you create a Cloud 9 environment and then installed the required tools over there. For this chapter, all the sample code can be found at

https://github.com/victorgu-github/Running-Containers-on-AWS-ebook/tree/main/Chapter4-ECR

AWS Cloud9 is an integrated development environment (IDE) that offers a cloud-based environment for writing, running, and debugging code. You can go to AWS Free Tire website (https://aws.amazon.com/free) and register a free AWS account with credits which should be enough for practicing all the labs in this book. Once you registered a new account, let's go to AWS Cloud 9 Console (https://us-west-2.console.aws.amazon.com/cloud9control/home?region=us-west-2) and create a new Cloud 9 environment. Please make sure you are under AWS region `us-west-2`**.** In the rest of the book, if don't mention specifically, the default region is `us-west-2`:

1. Select **Create environment**

2. In the new page, fill up Name with **awscontainerdemo** and change New EC2 instance selection to **t3.small**; Change Network settings to **Secure Shell (SSH)** and then Click **Create**.
3. In the Cloud 9 Console, when the new environment is ready, click **Open** to go inside the Cloud 9 IDE.
4. Resize the Cloud 9's storage EBS size from default 10 GB to 100 GB by running the following code. Once done, the Cloud 9 instance will be rebooted:

```
pip3 install --user --upgrade boto3

export instance_id=$(curl -s http://169.254.169.254/latest/meta-data/instance-id)

python -c "import boto3

import os

from botocore.exceptions import ClientError

ec2 = boto3.client('ec2')

volume_info = ec2.describe_volumes(

    Filters=[

        {

            'Name': 'attachment.instance-id',

            'Values': [

                os.getenv('instance_id')

            ]

        }

    ]

)

volume_id = volume_info['Volumes'][0]['VolumeId']

try:

    resize = ec2.modify_volume(

            VolumeId=volume_id,

            Size=100

    )

    print(resize)

except ClientError as e:
    if e.response['Error']['Code'] == 'InvalidParameterValue':
        print('ERROR MESSAGE: {}'.format(e))"
if [ $? -eq 0 ]; then
    sudo reboot
fi
```

5. Upgrade AWS CLI with the following code:

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -
o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
aws --version
```

   The expected result will be the following (you may get newer versions):

```
aws-cli/2.11.9 Python/3.11.2 Linux/4.14.309-231.529.amzn2.x86_64
exe/x86_64.amzn.2 prompt/off
```

6. Log in with the command line:

```
aws configure
```

   Verify who you are by running the following command:

```
aws sts get-caller-identity
```

Now your cloud9 environment is all set. You can re-login anytime for implementing the labs in this book.

Best Practices for building Container images

In this section, we will explain the container image layer structure, discuss base layer selection, and show you how to build container images efficiently and securely.

Container image format

As discussed in *Chapter 1*, **container image** is a static executable package which will become container(s) at runtime. A container image comprises several layers that are formed into one virtual filesystem. Each layer can be seen as an intermediate image, which is a change on the layer below. Container images are created using a Dockerfile. Every command you specify (`FROM`, `RUN`, `COPY`, etc.) in your Dockerfile causes the change in the previous image, which leads to a new image layer.

Figure 4.1 shows the structure of the image created by using the Dockerfile in the repo at https://github.com/victorgu-github/Running-Containers-on-AWS-ebook/tree/main/Chapter4-ECR/sample-nodejs-app/Dockerfile.

The layer at the bottom usually we call it the base layer. In this case, it is an image layer for node application (tag version 16). In the Dockerfile, we define some steps to create directory, copy application files into the layer and then run `npm install` to install dependent libraries. Every command will create an extra layer adding on the top. Every layer is immutable after creation, but you can keep adding more. The top layer has read-write permissions while the rest layers have read-only permissions:

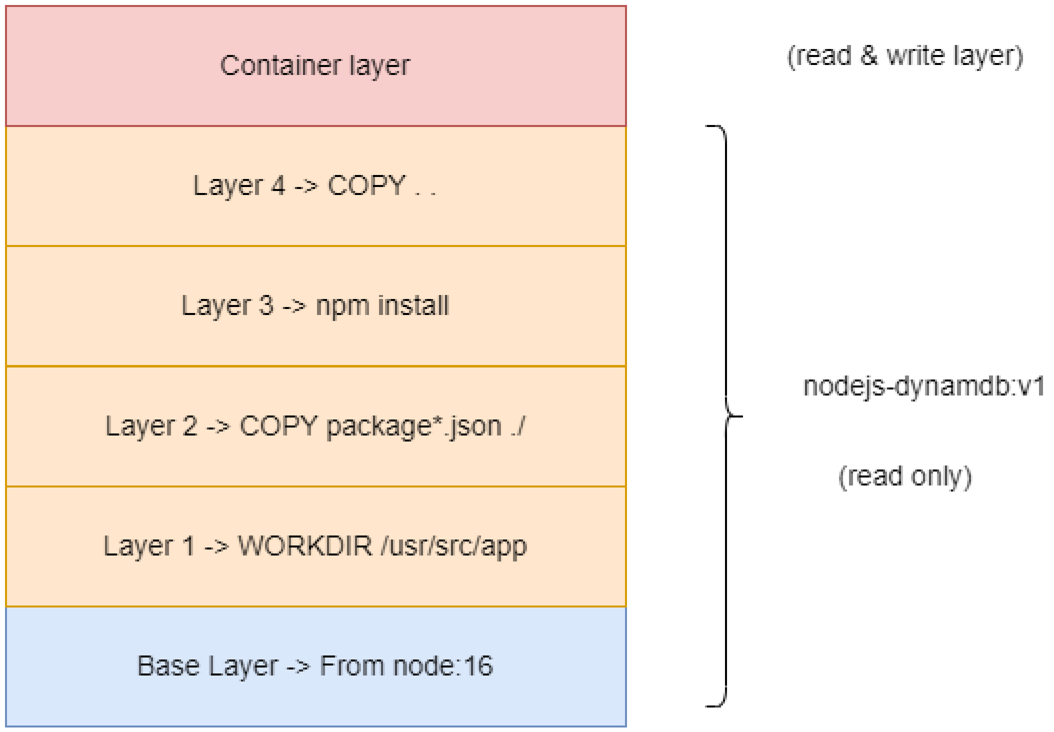


Figure 4.1 - Container image layer structure

In the image format, layers are stored as blobs and are identified by the digests. The layer's digest is generated by hashing its blob content. The most used hashing algorithm are SHA-256 or SHA-512. Besides image layer, each image also needs a manifest for describing the list of layer included. You can use the following docker command to read an image's manifest:

```
docker manifest inspect my-image:tag
```

Best practices for building images

In this section, we will go through some best practices for building your container images efficiently and securely.

Using minimal and official base images

As the foundation of the container image, how to choose the base image is critical to your entire image building procedure. There are many base images available on the market varying in terms of size, dependent libraries and utilities included, which eventually affect the performance and security.

Do you know the following statistics about container base image?

Size: Apline base image has 5.7 MB while Red Hat UBI takes 228 MB

Security: Ubuntu base images have significantly more vulnerabilities and older releases have more vulnerabilities.

Popularity: Debian is the most widely used distribution;RHEL, the least, by far.

The data source is the following:
https://academic.oup.com/gigascience/article/10/6/giab025/6291571
We suggest the following tips when you choose base image:

- Choose it as small as possible. Smaller base image will bring better performance and leave minimal attack surface.
- Use *Distroless* images which only contain your application and its runtime dependencies. Unlike the base images based on standard Linux distribution, Distroless images do not contain package managers, shells or any other programs.
- Choose a set of base images for your organization not just one. It is hard to choose the best one fitting for all the scenarios. Web development team wants to use the `nodejs` base image, while data science team needs an image based on python. Have a discussion with all the development teams in your company and make the list. Regularly go back to check the list and review it base on your current technical stacks.
- Choose the base image from Docker official images which are a curated set of container images published by Docker inc. For AWS customers, you can find and pull Docker Official Images directly from Amazon ECR public gallery at https://gallery.ecr.aws/docker/ for better performance and extra cost saving (directly pull images from Docker may cause data transmission cost in AWS).

Picking base images would be one of the first steps for converting your applications to containers. Your buniess requirements and preferences should also be considered. Start your container journey by practing the tips above and hold a meeting with your teams to gather feedbacks.

Using multi-stage builds

Multi-stage builds is a method for creating leaner and more secured container images. For example, you can have a stage for compiling and building your application, which can subsequently be copied to later stages. By doing this, you can ensure that only the final stage is used for creating the image, and that any dependencies and tools associated with building your application are discarded. This leaves behind a lightweight and modular production-ready image.
We have a sample ASP.NET Core application demoing how to write multi-stage builds in Dockerfile- https://github.com/victorgu-github/Running-Containers-on-AWS-ebook/tree/main/Chapter4-ECR/sample_aspnetapp

```
# use microsoft sdk image to build
```

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /source

# copy csproj and restore as distinct layers
COPY *.sln .
COPY aspnetapp/*.csproj ./aspnetapp/
RUN dotnet restore

# copy everything else and build app
COPY aspnetapp/. ./aspnetapp/
WORKDIR /source/aspnetapp
RUN dotnet publish -c release -o /app --no-restore

# final stage/image, use UBI
FROM registry.access.redhat.com/ubi8/dotnet-50-runtime
WORKDIR /app
COPY --from=build /app ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

In the Dockerfile above, you see two **From** statements. The first **From** is for using `.NET SDK` base image to compile the application. The built result will be copied into the second image which is Red hat UBI8 with dot net core 5 runtime.

Ordering Dockerfile commands appropriately and minimize steps

When using `docker build` command to generate an image based on a Dockerfile, Docker caches each step (or layer) in the Dockerfile to expedite subsequent builds. If any step changes in the Dockerfile, the cache will be invalidated not just for that specific step, but for all subsequent steps as well. Therefore, it's crucial to be mindful of the order in which you arrange your Dockerfile commands, placing layers that are likely to change towards the end of the Dockerfile.

To optimize the size of your Docker image, it is recommended to combine the RUN, COPY, and ADD commands as much as possible. Each command in a Dockerfile creates a new layer in the image, and each layer adds to the overall size. By minimizing the number of layers, you can reduce the size of the final image and improve performance.

Scanning your image for security check regularly

Continuously analyzing the security posture of your images using vulnerability detection tools is very important because container images may contain binaries and application libraries with vulnerabilities, or may develop vulnerabilities over time. There are two ways to scan your images: first scan your images right after docker build in CI pipeline; second, some container repository can continuously scan the images inside and we will explain this feature in Amazon ECR in the next section. Some popular tools can be integrated in your CI pipeline are Snyk, Trivy, Clair and Anchore.

Hands-on lab

In this section, please follow the instruction to run the lab. It will reinforce the learning on container image layers and help you better understand the best practices of building container images. We will run the lab in the Cloud 9 environment previously created.

Lab 1 - docker build and push

Let's first clone the code repo and go to the sample-`nodejs-app` folder in *Chapter 4* and then run `docker build`:

```
git clone https://github.com/victorgu-github/Running-Containers-on-AWS-ebook.git

cd Running-Containers-on-AWS-ebook/Chapter4-ECR/sample-nodejs-app/

docker build . -t nodejs-dynamdb:v1
```

Then let's check the build results by running `docker image ls`. As shown in the following command, you get two images at local, `node:16` is the base image defined in Dockerfile and `nodejs-dynamdb:v1` is the newly built image:

```
$ docker image ls

REPOSITORY        TAG        IMAGE ID        CREATED          SIZE

nodejs-dynamdb    v1         ba33a1a1080b    5 seconds ago    1.01GB

node              16         26ed31aaee8c    11 days ago      910MB
```

Now let's use the following commands to push the new `nodejs` application image to an ECR repository. Please replace `aws_account_id` with yours in all the following commands:

1. `Create an ECR repo with aws cli first`:

```
aws ecr create-repository --repository-name team-a/nodejs-dynamdb --region us-west-2
```

2. `Authenticate your Docker client to the Amazon ECR registry`:

```
aws ecr get-login-password --region us-west-2 | docker login --username AWS --password-stdin aws_account_id.dkr.ecr.us-west-2.amazonaws.com
```

3. `Tag your image with the Amazon ECR registry`

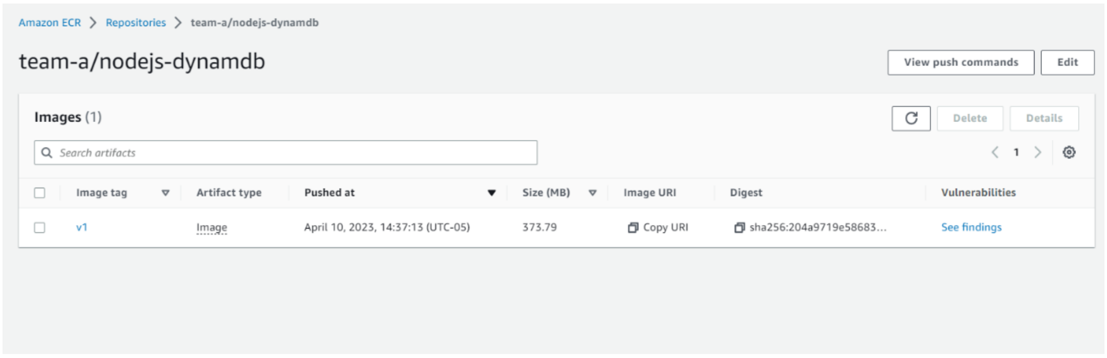
```
docker tag nodejs-dynamdb:v1 aws_account_id.dkr.ecr.us-west-
2.amazonaws.com/team-a/nodejs-dynamdb:v1
```

# Run docker image ls again. you will see a new record created but points to the same image
`docker image ls`

4. Push the image using the docker push command:

```
docker push aws_account_id.dkr.ecr.us-west-2.amazonaws.com/team-
a/nodejs-dynamdb:v1
```

5. Now go to Amazon ECR console in the region `us-west-2`. You should see the image as in the following screenshot:



team-a/nodejs-dynamdb

Images (1)

Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest	Vulnerabilities
v1	Image	April 10, 2023, 14:37:13 (UTC-05)	373.79	Copy URI	sha256:204a9719e58683...	See findings

Figure 4.2 - The new image in ECR

Congratulations. Now you've done the first lab. Feel free to redo the lab with your own base image or application code.

Lab 2 - Exploring container image layers using dive and simplifying with Slim

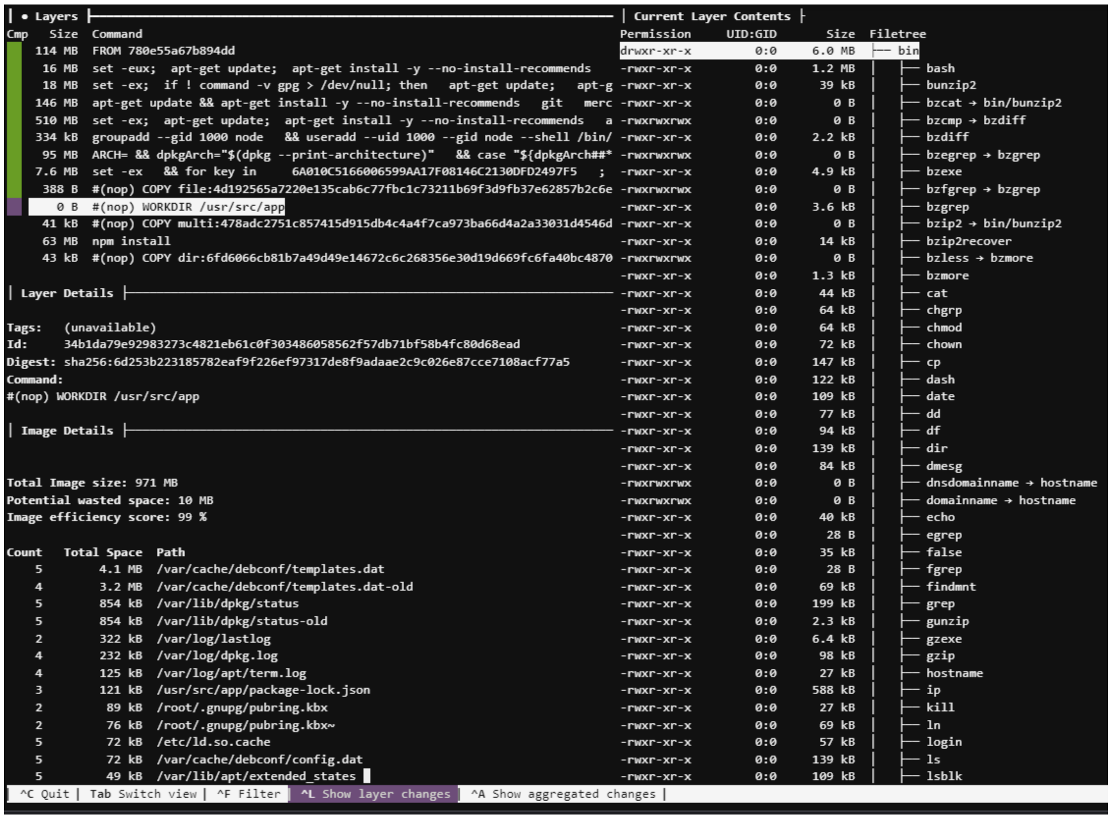
Let's first install dive and use it to explore the image with the following commands:

```
curl -OL
https://github.com/wagoodman/dive/releases/download/v0.9.2/dive_0.9.2_
linux_amd64.rpm

sudo rpm -i dive_0.9.2_linux_amd64.rpm

dive nodejs-dynamdb:v1
```

As shown in *Figure 4.3*, you can see all the layers inside the image. Every `WORKDIR`, `COPY` and `RUN` commands defined in Dockerfile introduce an additional layer on top of the base image:



```
| • Layers ├──────────────────────────────────────────────────────────────
Cmp   Size  Command
    114 MB  FROM 780e55a67b894dd
     16 MB  set -eux;  apt-get update;  apt-get install -y --no-install-recommends
     18 MB  set -ex;  if ! command -v gpg > /dev/null; then   apt-get update;   apt-g
    146 MB  apt-get update && apt-get install -y --no-install-recommends   git   merc
    510 MB  set -ex;  apt-get update;  apt-get install -y --no-install-recommends   a
    334 kB  groupadd --gid 1000 node   && useradd --uid 1000 --gid node --shell /bin/
     95 MB  ARCH= && dpkgArch="$(dpkg --print-architecture)"   && case "${dpkgArch##*
    7.6 MB  set -ex   && for key in     6A010C5166006599AA17F08146C2130DFD2497F5   ;
     388 B  #(nop) COPY file:4d192565a7220e135cab6c77fbc1c73211b69f3d9fb37e62857b2c6e
       0 B  #(nop) WORKDIR /usr/src/app
     41 kB  #(nop) COPY multi:478adc2751c857415d915db4c4a4f7ca973ba66d4a2a33031d4546d
     63 MB  npm install
     43 kB  #(nop) COPY dir:6fd6066cb81b7a49d49e14672c6c268356e30d19d669fc6fa40bc4870

| Layer Details ├─────────────────────────────────────────────────────────

Tags:    (unavailable)
Id:      34b1da79e92983273c4821eb61c0f303486058562f57db71bf58b4fc80d68ead
Digest: sha256:6d253b223185782eaf9f226ef97317de8f9adaae2c9c026e87cce7108acf77a5
Command:
#(nop) WORKDIR /usr/src/app

| Image Details ├─────────────────────────────────────────────────────────


Total Image size: 971 MB
Potential wasted space: 10 MB
Image efficiency score: 99 %

Count   Total Space  Path
    5        4.1 MB  /var/cache/debconf/templates.dat
    4        3.2 MB  /var/cache/debconf/templates.dat-old
    5        854 kB  /var/lib/dpkg/status
    5        854 kB  /var/lib/dpkg/status-old
    2        322 kB  /var/log/lastlog
    4        232 kB  /var/log/dpkg.log
    4        125 kB  /var/log/apt/term.log
    3        121 kB  /usr/src/app/package-lock.json
    2         89 kB  /root/.gnupg/pubring.kbx
    2         76 kB  /root/.gnupg/pubring.kbx~
    5         72 kB  /etc/ld.so.cache
    5         72 kB  /var/cache/debconf/config.dat
    5         49 kB  /var/lib/apt/extended_states
```

```
| Current Layer Contents ├
Permission     UID:GID       Size  Filetree
drwxr-xr-x         0:0     6.0 MB  ├── bin
-rwxr-xr-x         0:0     1.2 MB  │   ├── bash
-rwxr-xr-x         0:0      39 kB  │   ├── bunzip2
-rwxr-xr-x         0:0        0 B  │   ├── bzcat → bin/bunzip2
-rwxrwxrwx         0:0        0 B  │   ├── bzcmp → bzdiff
-rwxr-xr-x         0:0     2.2 kB  │   ├── bzdiff
-rwxrwxrwx         0:0        0 B  │   ├── bzegrep → bzgrep
-rwxr-xr-x         0:0     4.9 kB  │   ├── bzexe
-rwxrwxrwx         0:0        0 B  │   ├── bzfgrep → bzgrep
-rwxr-xr-x         0:0     3.6 kB  │   ├── bzgrep
-rwxr-xr-x         0:0        0 B  │   ├── bzip2 → bin/bunzip2
-rwxr-xr-x         0:0      14 kB  │   ├── bzip2recover
-rwxrwxrwx         0:0        0 B  │   ├── bzless → bzmore
-rwxr-xr-x         0:0     1.3 kB  │   ├── bzmore
-rwxr-xr-x         0:0      44 kB  │   ├── cat
-rwxr-xr-x         0:0      64 kB  │   ├── chgrp
-rwxr-xr-x         0:0      64 kB  │   ├── chmod
-rwxr-xr-x         0:0      72 kB  │   ├── chown
-rwxr-xr-x         0:0     147 kB  │   ├── cp
-rwxr-xr-x         0:0     122 kB  │   ├── dash
-rwxr-xr-x         0:0     109 kB  │   ├── date
-rwxr-xr-x         0:0      77 kB  │   ├── dd
-rwxr-xr-x         0:0      94 kB  │   ├── df
-rwxr-xr-x         0:0     139 kB  │   ├── dir
-rwxr-xr-x         0:0      84 kB  │   ├── dmesg
-rwxrwxrwx         0:0        0 B  │   ├── dnsdomainname → hostname
-rwxrwxrwx         0:0        0 B  │   ├── domainname → hostname
-rwxr-xr-x         0:0      40 kB  │   ├── echo
-rwxr-xr-x         0:0       28 B  │   ├── egrep
-rwxr-xr-x         0:0      35 kB  │   ├── false
-rwxr-xr-x         0:0       28 B  │   ├── fgrep
-rwxr-xr-x         0:0      69 kB  │   ├── findmnt
-rwxr-xr-x         0:0     199 kB  │   ├── grep
-rwxr-xr-x         0:0     2.3 kB  │   ├── gunzip
-rwxr-xr-x         0:0     6.4 kB  │   ├── gzexe
-rwxr-xr-x         0:0      98 kB  │   ├── gzip
-rwxr-xr-x         0:0      27 kB  │   ├── hostname
-rwxr-xr-x         0:0     588 kB  │   ├── ip
-rwxr-xr-x         0:0      27 kB  │   ├── kill
-rwxr-xr-x         0:0      69 kB  │   ├── ln
-rwxr-xr-x         0:0      57 kB  │   ├── login
-rwxr-xr-x         0:0     139 kB  │   ├── ls
-rwxr-xr-x         0:0     109 kB  │   ├── lsblk
```

```
| ^C Quit | Tab Switch view | ^F Filter | ^L Show layer changes | ^A Show aggregated changes |
```

Figure 4.3 - Exploring container image layers using Dive

Next, let's try to use Slim (previously DockerSlim) to optimize the image and reduce its size. To install Slim, please run the following commands:

```
curl -sL
https://raw.githubusercontent.com/slimtoolkit/slim/master/scripts/inst
all-slim.sh | sudo -E bash -
```

Then use Slim to rebuild the image:

```
slim build --target nodejs-dynamdb:v1
```

By running docker image `ls` to check the result, you can see the new …`/nodejs-dynamdb.slim` image only takes 90.8 MB while the original one takes 1.01 GB:

```
$ docker image ls

REPOSITORY                                                         TAG      IMAGE ID       CREATED            SIZE

349361870252.dkr.ecr.us-west-2.amazonaws.com/team-a/nodejs-dynamdb.slim   latest
2705df195b27   7 seconds ago       90.8MB

349361870252.dkr.ecr.us-west-2.amazonaws.com/team-a/nodejs-dynamdb       v1
ba33a1a1080b   About an hour ago   1.01GB

nodejs-dynamdb                                                     v1       ba33a1a1080b   About an hour ago   1.01GB
```

```
node                                   16       26ed31aaee8c   11 days ago        910MB
```

This is the end of the lab2. For more advanced usage of Slim, please go to https://github.com/slimtoolkit/slim

Amazon ECR deep dive

In this section, we will examine Amazon ECR in detail. We will start by explaining how images are stored in Amazon ECR and then delve into how to utilize policies to manage images. Additionally, we will explore the image scanning feature in Amazon ECR for detecting software vulnerabilities in your container images. You will gain insight into Amazon ECR repository design patterns, ranging from single account to multi-region and multi-account setups. Finally, we will demonstrate how to use Amazon ECR to store your helm chart, publish your images/charts in public repositories, and monitor ECR repository metrics using CloudWatch.

Figure 4.4 shows how Amazon ECR stores images conceptually. Each AWS account has one Amazon ECR registry service per region. A registry contains multiple repositories containing images. An image is actually a manifest that refers to an image blob identified by its digest. Tags are labels used to name images to make them more human identifiable. A tag can only point to one image, but multiple tags can point to the same image.

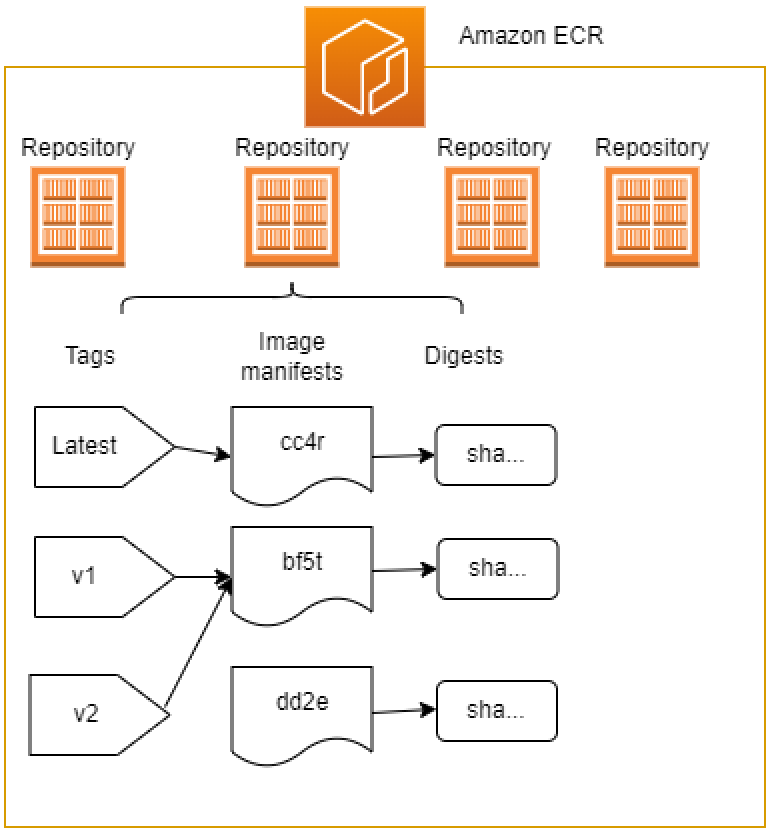


Figure 4.4 The conceptual architecture of Amazon ECR

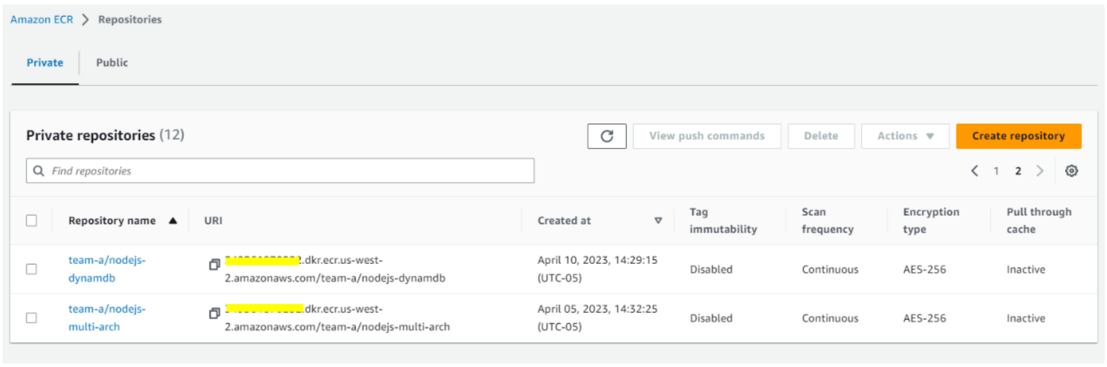
The following is an example of an image reference link in ECR:

```
11111111111.dkr.ecr.us-west-2.amazonaws.com/team-a/nodejs-multi-arch:v1
```

`11111111111.dkr.ecr.us-west-2.amazonaws.com` is the ECR registry domain. In this example, the ECR service is in the region `us-west-2`.

team-a is the repository name. Repository is a virtual concept in ECR. As shown in *Figure 4.5*, images under the same repository (that is, **team-a**) are not grouped together but are shown as flat file structure. Repository is optional but recommended to use. You can use policy to manage the images under the same repository in a batch.

nodejs-multi-arch:v1 is the image name and tag.



Private repositories (12)

Repository name	URI	Created at	Tag immutability	Scan frequency	Encryption type	Pull through cache
team-a/nodejs-dynamdb	[redacted].dkr.ecr.us-west-2.amazonaws.com/team-a/nodejs-dynamdb	April 10, 2023, 14:29:15 (UTC-05)	Disabled	Continuous	AES-256	Inactive
team-a/nodejs-multi-arch	[redacted].dkr.ecr.us-west-2.amazonaws.com/team-a/nodejs-multi-arch	April 05, 2023, 14:32:25 (UTC-05)	Disabled	Continuous	AES-256	Inactive

Figure 4.5 Repository flat structure in ECR

Figure 4.5 shows that ECR store images in a flat strcutre. The images under the same repository name will be listed together but are not grouped heirtically.

Amazon ECR best practices

Amazon ECR is a managed AWS service for storing your container images. You can start using it without preliminary training. But the following tips would help you maximize the value of using Amazon ECR.

Managing images with policy

Amazon ECR has two set of policies, one for controlling access to your images and another for managing the lifecycle of the images.

- **repository policy**- This is a resource-based policy for control access to repositories. It is a type of IAM policy that is designed to manage and regulate access to Amazon ECR repositories. Usually, it works along with IAM policy to assign AWS users the access to ECR. IAM policy are is used to apply permissions for the entire Amazon ECR service while repository policy is for defining the access on repository level. The following code shows an example of the repository policy, allowing for a specific user to pull images from any repository:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ECRRepositoryPolicy",
            "Effect": "Allow",
            "Principal": {"AWS": "arn:aws:iam::account-
id:user/username"},
            "Action": [
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage"
            ]
        }
    ]
}
```

- **lifecycle policy**- This provides the lifecycle management of images in a repository.

  A lifecycle policy consists one or more rules that describe the actions to be taken by Amazon ECR. This allows for automatic management of container images by setting expiration criteria based on either the age or count of images. Failure to take any action will result in an accumulation of outdated container images in the registry, leading to increased costs and potential security risks. The following code shows an example of using the lifecycle policy to expire (delete) untagged images older than 30 days:

```
{
    "rules": [
        {
            "rulePriority": 1,
            "description": "Expire images older than 30 days",
            "selection": {
                "tagStatus": "untagged",
                "countType": "sinceImagePushed",
                "countUnit": "days",
                "countNumber": 30
```

```
                    },
                    "action": {
                            "type": "expire"
                    }
            }
    ]
}
```

You can get all the parameters used by lifecycle policy at https://awscli.amazonaws.com/v2/documentation/api/2.0.34/reference/ecr/put-lifecycle-policy.html

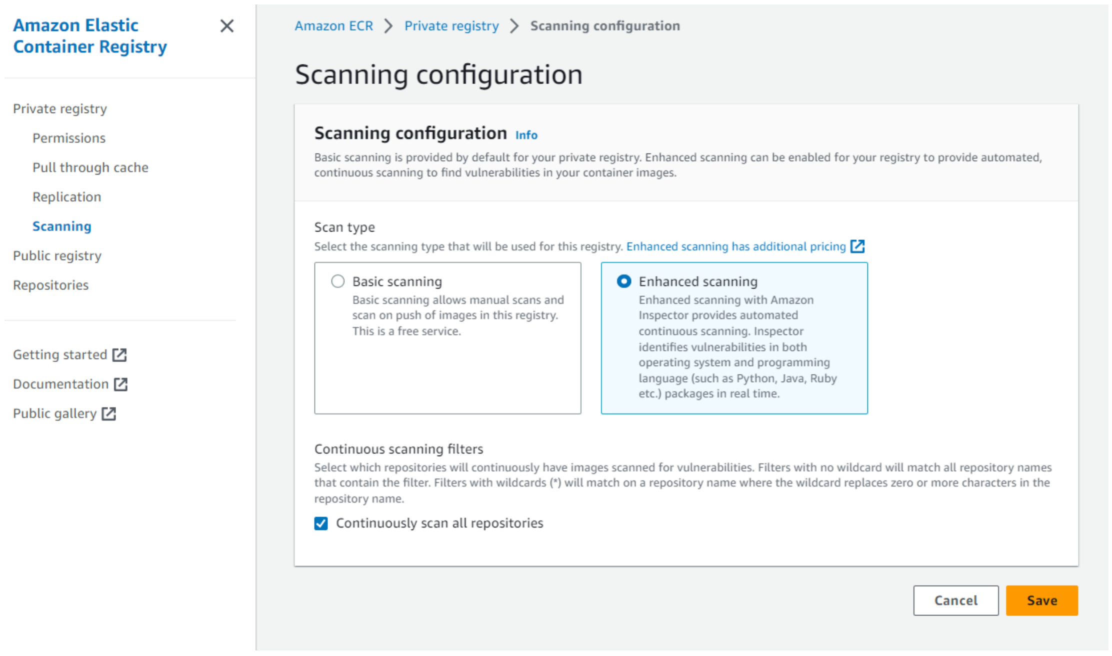
Identifying and fixing security vulnerabilities with ECR image scanning

Amazon ECR offers two scanning types: **basic** and **enhanced**. The basic one is enabled by default to all the images in your private repositories. Whenever you push images to ECR, those images will be scanned unless you set the filter to turn off the scan on certain repositories. The basic scan adopts the **Common Vulnerabilities and Exposures** (**CVEs**) database from the open-source Clair project. You can more information about the CVEs database the Clair tool at https://github.com/quay/clair.

Enhanced scanning can scan your images for both operating systems and programing language package vulnerabilities. An integration between Amazon Inspector and Amazon ECR enables vulnerability scanning capabilities and allows you to view the scan findings within both services. The configuration for scanning is established at the level of the private registry and is defined on a per-Region basis.

Note

Please note that both basic and enhanced scanning in Amazon ECR support Linux based containers only.



Scanning configuration

Scanning configuration Info

Basic scanning is provided by default for your private registry. Enhanced scanning can be enabled for your registry to provide automated, continuous scanning to find vulnerabilities in your container images.

Scan type

Select the scanning type that will be used for this registry. Enhanced scanning has additional pricing

○ Basic scanning
Basic scanning allows manual scans and scan on push of images in this registry. This is a free service.

● Enhanced scanning
Enhanced scanning with Amazon Inspector provides automated continuous scanning. Inspector identifies vulnerabilities in both operating system and programming language (such as Python, Java, Ruby etc.) packages in real time.

Continuous scanning filters

Select which repositories will continuously have images scanned for vulnerabilities. Filters with no wildcard will match all repository names that contain the filter. Filters with wildcards (*) will match on a repository name where the wildcard replaces zero or more characters in the repository name.

✓ Continuously scan all repositories

Figure 4.6 Amazon ECR scanning configuration panel

As shown in *Figure 4.6*, you can go to AWS console and switch the scan types under **Private registry** -> **Scanning**. By default, **Continuously scan all repositories** is selected, which means Amazon ECR will continually scan all images during their lifetime. You can reduce the scan duration to **180 days** or **30 days**.

Enabling your repository design with multiple account and multiple region

We recommend using single central ECR design per region, which means that you only need to have one ECR service in one shared AWS account in one region. As shown in *Figure 4.7*, the built images would be pushed into **Dev/Test** repositories. Upon successful completion of the testing phase, the images are moved to a production repository from where images can be pulled and run on a production Amazon ECS/EKS cluster that may be located in another location. The non-prod ECS/EKS cluster can be deployed into the same AWS account or have its own account. You can set up the cross account IAM policy to enable the cross-account access. Also, by using the repository policy mentioned above, you can give developers the access to **Dev/Test** repositories only while allow administrators and build agents to access the prod repositories.

Since Amazon ECR is a regional service, to enable high availability, you can use ECR's cross-Region replication feature to duplicate images from one region to another. Multi-region design is also used when you need to lower image pulling latency. By copying images to multiple regions, the container platform should be able to pull images from its local region. As shown in *Figure 4.7*, you can copy all the images needs to be replicated under one repository and then enable replication feature for that repository. The replication feature cannot be relayed. For example, if you configured cross-region replication from region `us-west-2` to region `us-east-1` and from region `us-east-1` to region `us-east-2`, an image pushed to `us-west-2` replicates to only `us-east-1`, it doesn't replicate again to `us-east-2`. You should set up the replication from region `us-west-2` to region `us-east-2` instead:

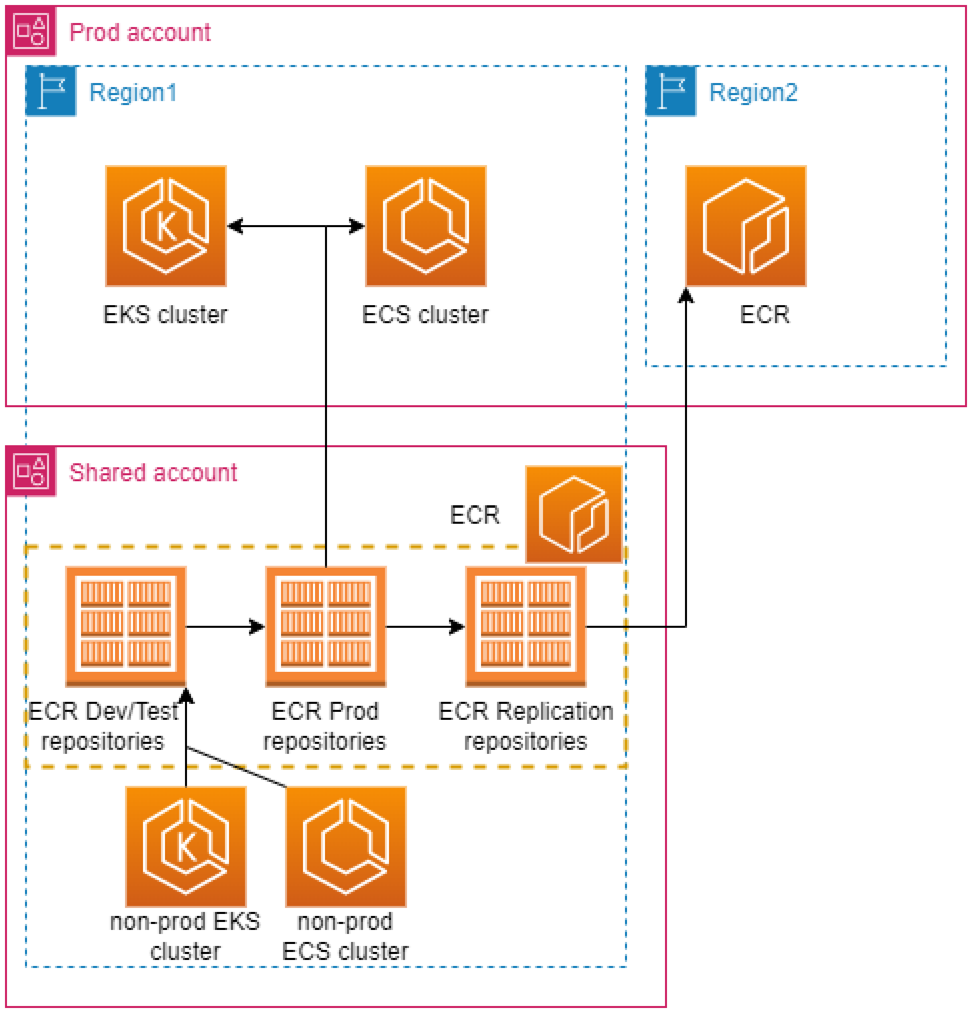


Figure 4.7 - Cross-account and cross-region ECR repository design

Using both private repository and public repository

When we discuss the repositories in ECR, if not specify, you are talking about the private repositories which secured by Amazon IAM-based authentication. You have to use AWS account credentials before pulling images from ECR private repositories. The public repository in ECR, on the other hand, doesn't require authentication for pulling images (although you still need to authenticate the Docker CLI for push).

You can use ECR public repositories to share your images back to the open source community. AWS creates the Amazon ECR public gallery which is a public website for easily searching container images hosted in Amazon ECR public repositories. AWS also publishes its service container images, like images for load balancer in public gallery. Visit the Amazon ECR Public Gallery at https://gallery.ecr.aws

Storing helm charts in ECR

Besides container images, Amazon ECR also supports helm chart which is a package format used to deploy a set of Kubernetes resources. There is a bit of difference between pushing and pulling helm chart from ECR and that of images.

- You have to authenticate your helm client to the Amazon ECR registry when you pull or push helm charts by following the commands:

  ```
  aws ecr get-login-password \
       --region us-west-2 | helm registry login \
       --username AWS \
       --password-stdin aws_account_id.dkr.ecr.us-west-2.amazonaws.com
  ```

- When creating a repository to store your helm chart, the name of your repository should match the name you used when creating the helm chart.

- ECR only supports helm chart with OCI format. The chart reference is always prefixed with `oci://`. The push and pull (install) helm chart from ECR would look like the following:

  ```
  helm push helm-test-chart-0.1.0.tgz oci://aws_account_id.dkr.ecr.us-west-2.amazonaws.com/

  helm install ecr-chart-demo oci://aws_account_id.dkr.ecr.region.amazonaws.com/helm-test-chart --version 0.1.0
  ```

Understanding your ECR usage with CloudWatch metrics

CloudWatch metrics can be utilized to gain insight into the utilization of Amazon ECR resources in your account. Amazon ECR sends two types of metrics to CloudWatch automatically. One is repository pull count. It is under **CloudWatch** -> **Metrics**->**All metrics** -> **AWS/ECR namespace/ Repository Metrics** namespace. Another is API call count (like, get ECR token api, upload layer api). It is under AWS/Usage namespace.

Advanced usage with Amazon ECR

In this section, we would like to fortify your learning on Amazon ECR by introducing you to two advanced usage patterns. One is to build multi-architecture container images to make your container ecosystem more reliability and resilience by allowing it running on a wider range of hardware architectures; Another is to enhance the image security by using cryptographic signatures.

Multi-architecture container images

Multi-architecture container images are the container images that can run on multiple processor architectures, such as x86, ARM, and others. The container images you built on x86 machine cannot be used on ARM machines directly. With multi-architecture images, you will build images on a wider range of hardware and optimize images for each specific hardware architectures in that reduce the amount of processing power and memory required to run your applications.

In AWS, most EC2 instances are using x86 based architecture. Since 2018, AWS started using a 64-bit ARM-based CPU processor **Graviton** in their EC2 instances. After several version updates, the latest Graviton3 delivers up to 40% better price performance over comparable current generation x86-based instances. An increasing number of customers are interested in migrating to Graviton processors to leverage the performance optimizations offered by these processors and realize cost savings.

While you can use the tool like **buildx** (https://www.docker.com/blog/faster-multi-platform-builds-dockerfile-cross-compilation-guide/) to build multi-architecture images on the same host machine, the simulation of homogenous hardware will bring some unexpected errors during the cross-compile. In *Figure 4.8*, we demo how to use native build for each processor and combine the results as a multi-architecture image to support both x86 processor and ARM processor.

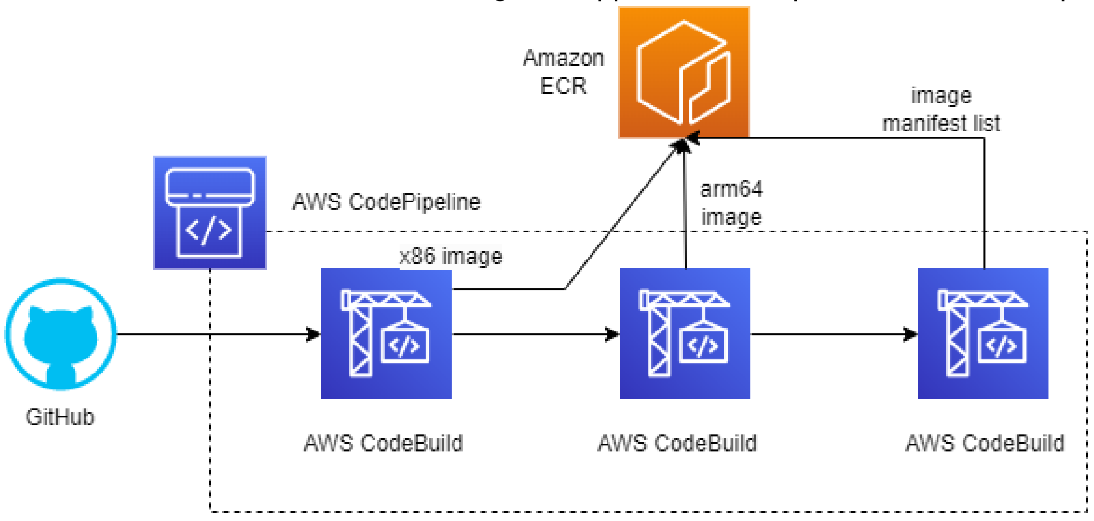


Figure 4.8 - Multi-architecture image build workflow

You will see the detailed instructions in the following lab for implementing the multi-architecture workflow.

Lab 3 - Multi-architecture image build with AWS CodeBuild

We will use the cloud9 environment created before for this lab.

1. Create a new repository `nodejs-multi-arch` in ECR with the following command:

```
aws ecr create-repository --repository-name nodejs-multi-arch --
region us-west-2
```

2. Create a build project in CodeBuild to build the image based on x86:
I. Go to AWS CodeBuild Console (https://us-west-2.console.aws.amazon.com/codesuite/codebuild/home?region=us-west-2), choose **Create build project**
II. Fill up the Create build project panel with the following:

- Project name: **build-x86**
- Source provider : **GitHub**
- Choose **Connect using OAuth** and Click **Connect to GitHub**
- Repository URL: **https://github.com/victorgu-github/Running-Containers-on-AWS-ebook**
- Environment image: **Managed image**-> **Amazon Linux 2** -> **Standard** -> **aws/codebuild/amazonlinux2-x86_64-standard:3.0**
- Select **Privileged**
- Expand **Additional configurations** and move to the **Environment variables** and add the following variables:

`AWS_DEFAULT_REGION`	`us-west-2`	Plaintext
`AWS_ACCOUNT_ID`	[your AWS account ID]	Plaintext
`IMAGE_REPO_NAME`	`team-a/nodejs-multi-arch`	Plaintext
`IMAGE_TAG`	x64	Plaintext
`docker_user`	[your docker account name]	Plaintext
`docker_pwd`	[your docker account password]	Plaintext

Table 4.1 - Build x86 environment varaibales

- Buildspec name: **Chapter4-ECR/sample-nodejs-app/buildspec.yml**
- `Click` **Create build project** `button`

3. Attach the IAM policy. After creating a new CodeBuild project, if you need to modify the IAM role associated with the project to allow interaction with the Amazon ECR API, you can follow these steps:

- On the CodeBuild console, choose the **`build-x86`** project

- Choose the Build details
- Under Service role, choose the link that looks like **`arn:aws:iam::111111111111:role/service-role/codebuild- build-x86-service-role`**
- A new browser tab should open.
- Choose Attach policies.
- In the Search field, enter **`AmazonEC2ContainerRegistryPowerUser`**.
- Select **`AmazonEC2ContainerRegistryPowerUser`**.
- Choose Attach policy.
- `Go to` **build-x86** `project in CodeBuild, click` **Start build** `button`

4. Create a build project in CodeBuild to build the image based on ARM64. Repeat the steps in Step 2 but with the following setting:

- Project name: **build-arm64**
- Image: **aws/codebuild/amazonlinux2-aarch64-standard:2.0**
- Service role: choose **Existing service role**, the one used in step 3.
- Environment variables:

`AWS_DEFAULT_REGION`	`us-west-2`	Plaintext
`AWS_ACCOUNT_ID`	[your AWS account ID]	Plaintext
`IMAGE_REPO_NAME`	`team-a/nodejs-multi-arch`	Plaintext
`IMAGE_TAG`	`arm64`	Plaintext
`docker_user`	[your docker account name]	Plaintext
`docker_pwd`	[your docker account password]	Plaintext

Table 4.2 - Build arm64 environment varaibles

- Buildspec name: **Chapter4-ECR/sample-nodejs-app/buildspec.yml**
- `Once build-arm64 project created, go inside and click` **Start build** `button`

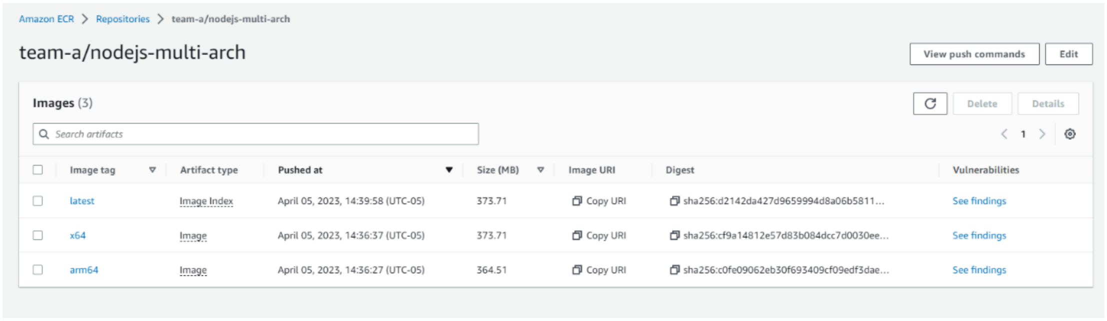
5. Create an Image Index to link images built under difference processors together. Repeat the steps in Step 2 but with the following setting:

- Project name: **image-manifest**
- Service role: Choose **Existing service role**, the one used in step 3.
- Environment variables:

`AWS_DEFAULT_REGION`	`us-west-2`	Plaintext
`AWS_ACCOUNT_ID`	[your AWS account ID]	Plaintext
`IMAGE_REPO_NAME`	`team-a/nodejs-multi-arch`	Plaintext
`IMAGE_TAG`	latest	Plaintext

Table 4.3 - Multi-arch image manifest environment variables

- Buildspec name: **Chapter4-ECR/sample-nodejs-app/buildspec-manifest.yml**
- `Once image-manifest project created, go inside and click` **Start build** `button`


Amazon ECR > Repositories > team-a/nodejs-multi-arch

team-a/nodejs-multi-arch

Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest	Vulnerabilities
latest	Image Index	April 05, 2023, 14:39:58 (UTC-05)	373.71	Copy URI	sha256:d2142da427d9659994d8a06b5811...	See findings
x64	Image	April 05, 2023, 14:36:37 (UTC-05)	373.71	Copy URI	sha256:cf9a14812e57d83b084dcc7d0030ee...	See findings
arm64	Image	April 05, 2023, 14:36:27 (UTC-05)	364.51	Copy URI	sha256:c0fe09062eb30f693409cf09edf3dae...	See findings

Figure 4.9 - Multi-architecture images stored in Amazon ECR

`Once all the build projects are created and completed, you can check the results in ECR. Under the repository` **team-a/nodejs-multi-arch**, you should see three records as shown in *Figure 4.9*. You can test the image by pulling it to your local. You only need to use **team-a/nodejs-multi-arch:latest** and the docker CLI will pull the right version based on your host's processor.

Container image signing

Container image signing is the process of digitally signing a container image to guarantee its authenticity and integrity. By using cryptographic keys to sign the image and creating a digital signature, it verifies that the image has not been altered, and ensures that the image being used is the exact same one that was originally signed. Container image signing enhances security, ensures compliance with regulations. It helps prevent attacks such as malware injection or tampering with the image content and provides a verifiable record of the image's origin and authenticity.

Figure 4.10 shows a general workflow of signing and verifying an image with a private/public key pair:

- The step first is to build a container image. The application source code and Dockerfile is saved in Code Repo in Github and the CI/CD pipeline compiles the code, builds it as a container image and pushes it to the `Dev/Test` repository in ECR.
- Once the image is built, you can use a signer to sign the image with your private key and then move the signed image to the prod repository. The signer can be a component in your

build agent. The following lab will walk you through how to use Cosign as signer. The signer will sign the image with the private key inside a private/public key pair.

- The signed image will be verified with the public key before pulling into a cluster. You can use the policy keeper tool like kyverno to make sure that only signed images can be deployed after verification:

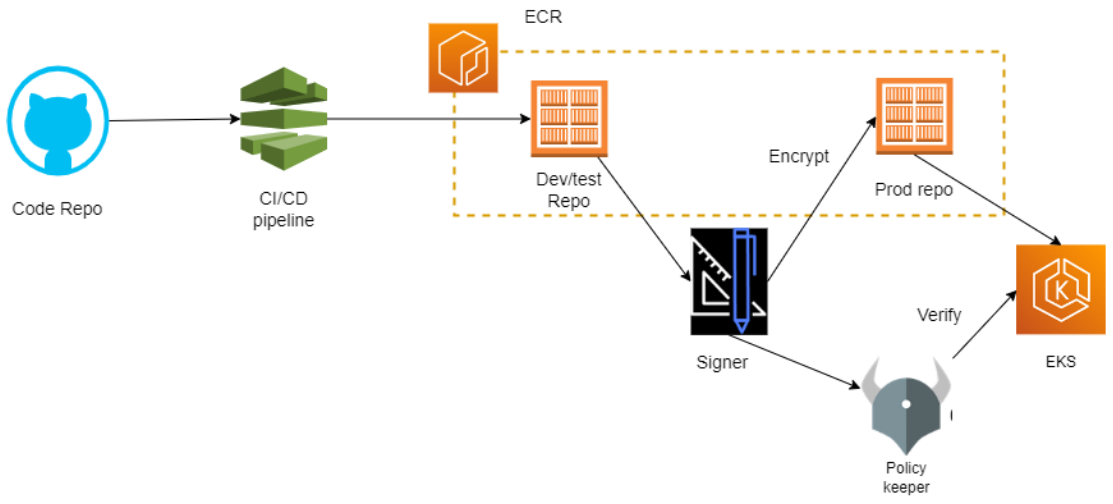


Figure 4.10 - Image signing and verifying workflow

Let's practice the image signing with lab 4.

Lab 4 - Signing your image with Cosign

We will use the cloud9 environment created earlier for this lab.
Let's first install the Cosign with the following command:

```
wget
"https://github.com/sigstore/cosign/releases/download/v2.0.0/cosign-2.0.0.x86_64.rpm"

sudo rpm -ivh cosign-2.0.0.x86_64.rpm
```

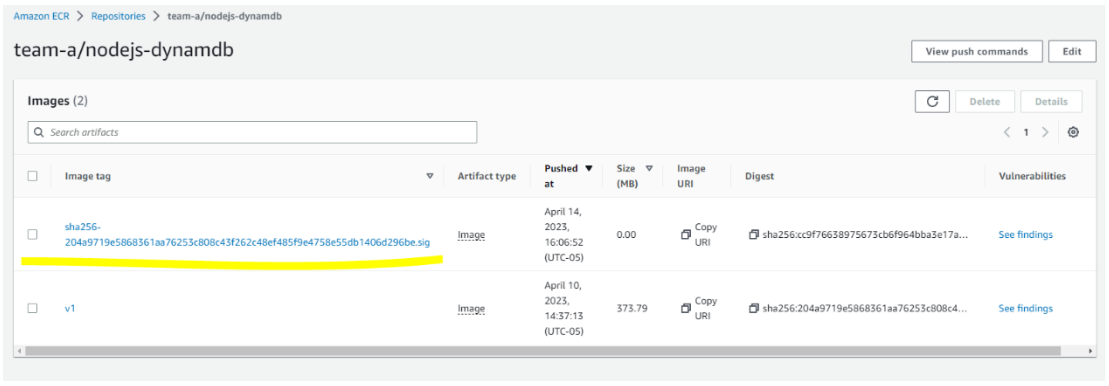
Then we can use the default **keyless signing** method in Cosign to sign the image **349361870252.dkr.ecr.us-west-2.amazonaws.com/team-a/nodejs-dynamdb:v1** we created in Lab 1. You will be asked to register an account with cosign and then login to get a verification code:

```
# authenticate with ECR

aws ecr get-login-password --region us-west-2 | docker login --username AWS --password-stdin aws_account_id.dkr.ecr.us-west-2.amazonaws.com

cosign sign <your account id>.dkr.ecr.us-west-2.amazonaws.com/team-a/nodejs-dynamdb:v1
```

Go to the repository **team-a/nodejs-dynamdb** in ECR. You will see that the signature is saved along with the image as shown in the following screenshot. The keyless mode in this lab is for demo purpose. You should use your public/private key pair for signing and verifying in your production environment. Your public/private key pair can be saved in AWS KMS:



team-a/nodejs-dynamdb

Images (2)

Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest	Vulnerabilities
sha256-204a9719e5868361aa76253c808c43f262c48ef485f9e4758e55db1406d296be.sig	Image	April 14, 2023, 16:06:52 (UTC-05)	0.00	Copy URI	sha256:cc9f76638975673cb6f964bba3e17a...	See findings
v1	Image	April 10, 2023, 14:37:13 (UTC-05)	373.79	Copy URI	sha256:204a9719e5868361aa76253c808c4...	See findings

Figure 4.11 - Image signature in ECR pushed by Cosign

For more advanced usage of cosign for image signing, please go to https://github.com/sigstore/cosign

Summary

In this chapter, we have covered various best practices for building container images. We have demonstrated how to select appropriate base images, create images with multi-stage build, and scan images for vulnerabilities. Additionally, we have introduced open-source tools such as dive and slim, which can be used to explore image layers and reduce image size.

Moving on, we have provided tips for using Amazon ECR, which includes managing images with policies and utilizing image scanning features to identify potential software vulnerabilities. We recommend utilizing a central ECR service to host all images within an organization, and leveraging policy and replication features to facilitate cross-account and cross-region access. Moreover, we have introduced important features in ECR such as Helm chart, public repository, and repository metrics.

Towards the end of the chapter, we have showcased two advanced usage patterns related to images and ECR. First, we have discussed the use of multi-architecture images to enable container systems to run on a wider range of hardware architectures. Second, we have explained how image signing technology can be employed to enhance image security through the use of asymmetric encryption. In the next chapter, you will start gaining the knowledge of deploying container images to Amazon ECS.