Amazon Elastic Container Service

Best Practices Guide

Amazon Elastic Container Service: Best Practices Guide

Table of Contents

Introduction

Amazon Elastic Container Service (Amazon ECS) is a highly scalable and fast container management service that you can use to manage containers on a cluster. This guide covers many of the most important operational best practices for Amazon ECS. It also describes several core concepts that are involved in how Amazon ECS based applications work. The goal is to provide a concrete and actionable approach to operating and troubleshooting Amazon ECS based applications.

This guide is revised regularly to incorporate new Amazon ECS best practices as they're established. If you have any questions or comments about any of the content in this guide, raise an issue in the GitHub repository. For more information, see Amazon ECS Best Practices Guide on GitHub.

Best Practices - Networking

Modern applications are typically built out of multiple distributed components that communicate with each other. For example, a mobile or web application might communicate with an API endpoint, and the API might be powered by multiple microservices that communicate over the internet.

This guide presents the best practices for building a network where the components of your application can communicate with each other securely and in a scalable manner.

Topics

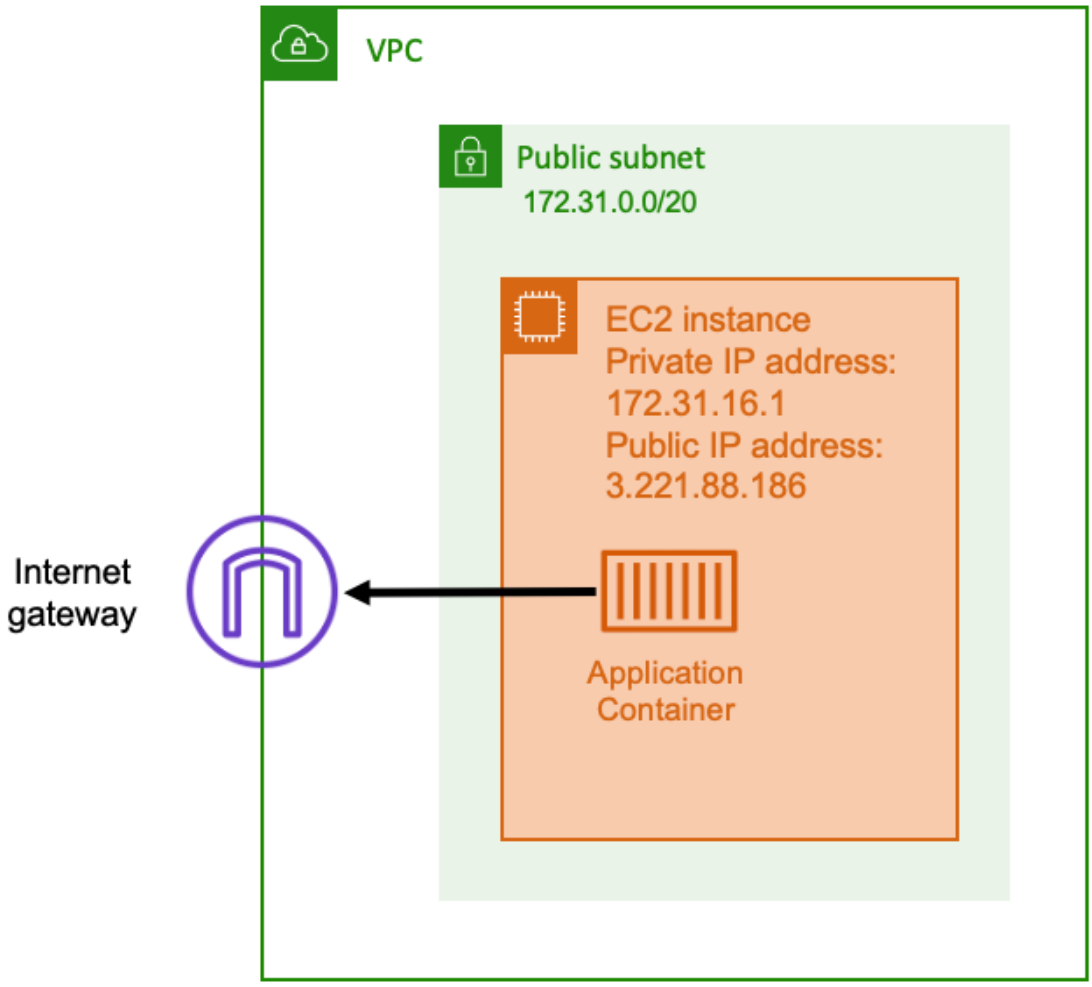
Connecting to the internet

Most containerized applications have a least some components that need outbound access to the internet. For example, the backend for a mobile app requires outbound access to push notifications.

Amazon Virtual Private Cloud has two main methods for facilitating communication between your VPC and the internet.

Topics

Using a public subnet and internet gateway



By using a public subnet that has a route to an internet gateway, your containerized application can run on a host inside a VPC on a public subnet. The host that runs your container is assigned a public IP address. This public IP address is routable from the internet. For more information, see Internet gateways in the *Amazon VPC User Guide*.

This network architecture facilitates direct communication between the host that runs your application and other hosts on the internet. The communication is bi-directional. This means that not only can you establish an outbound connection to any other host on the internet, but other hosts on the internet might also attempt to connect to your host. Therefore, you should pay close attention to your security group and firewall rules. This is to ensure that other hosts on the internet can't open any connections that you don't want to be opened.

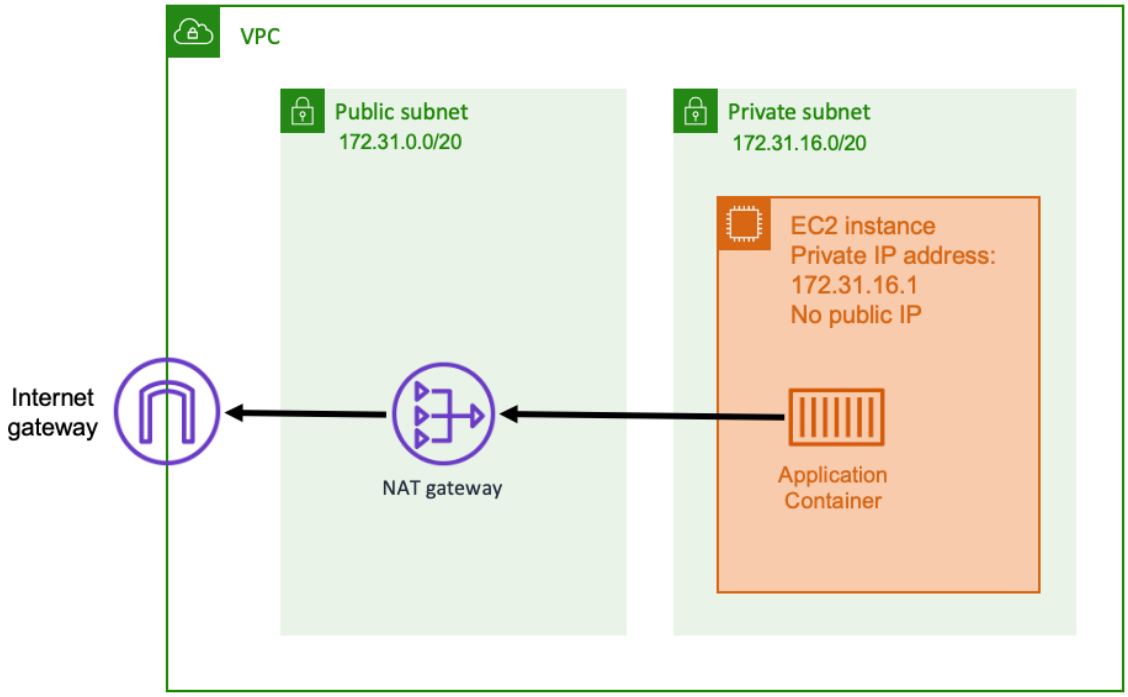
For example, if your application is running on Amazon EC2, make sure that port 22 for SSH access is not open. Otherwise, your instance could receive constant SSH connection attempts from malicious bots on the internet. These bots trawl through public IP addresses. After they find an open SSH port, they attempt to brute-force passwords to try to access your instance. Because of this, many organizations limit the usage of public subnets and prefer to have most, if not all, of their resources inside of private subnets.

Using public subnets for networking is suitable for public applications that require large amounts of bandwidth or minimal latency. Applicable use cases include video streaming and gaming services.

This networking approach is supported both when you use Amazon ECS on Amazon EC2 and when you use it on AWS Fargate.

- Using Amazon EC2 — You can launch EC2 instances on a public subnet. Amazon ECS uses these EC2 instances as cluster capacity, and any containers that are running on the instances can use the underlying public IP address of the host for outbound networking. This applies to both the `host` and `bridge` network modes. However, the `awsvpc` network mode doesn't provide task ENIs with public IP addresses. Therefore, they can't make direct use of an internet gateway.
- Using Fargate — When you create your Amazon ECS service, specify public subnets for the networking configuration of your service, and ensure that the **Assign public IP address** option is enabled. Each Fargate task is networked in the public subnet, and has its own public IP address for direct communication with the internet.

Using a private subnet and NAT gateway



By using a private subnet and a NAT gateway, you can run your containerized application on a host that's in a private subnet. As such, this host has a private IP address that's routable inside your VPC, but isn't routable from the internet. This means that other hosts inside the VPC can make connections to the host using its private IP address, but other hosts on the internet can't make any inbound communications to the host.

With a private subnet, you can use a Network Address Translation (NAT) gateway to enable a host inside a private subnet to connect to the internet. Hosts on the internet receive an inbound connection that appears to be coming from the public IP address of the NAT gateway that's inside a public subnet. The NAT gateway is responsible for serving as a bridge between the internet and the private VPC. This configuration is often preferred for security reasons because it means that your VPC is protected from direct access by attackers on the internet. For more information, see NAT gateways in the *Amazon VPC User Guide*.

This private networking approach is suitable for scenarios where you want to protect your containers from direct external access. Applicable scenarios include payment processing systems or containers storing user data and passwords. You're charged for creating and using a NAT gateway in your account. NAT gateway hourly usage and data processing rates also apply. For redundancy purposes, you should have a NAT gateway in each Availability Zone. This way, the loss in availability of a single Availability Zone doesn't compromise your outbound connectivity. Because of this, if you have a small workload, it might be more cost effective to use private subnets and NAT gateways.

This networking approach is supported both when using Amazon ECS on Amazon EC2 and when using it on AWS Fargate.

- Using Amazon EC2 — You can launch EC2 instances on a private subnet. The containers that run on these EC2 hosts use the underlying hosts networking, and outbound requests go through the NAT gateway.
- Using Fargate — When you create your Amazon ECS service, specify private subnets for the networking configuration of your service, and don't enable the **Assign public IP address** option. Each Fargate task is hosted in a private subnet. Its outbound traffic is routed through any NAT gateway that you have associated with that private subnet.

Receiving inbound connections from the internet

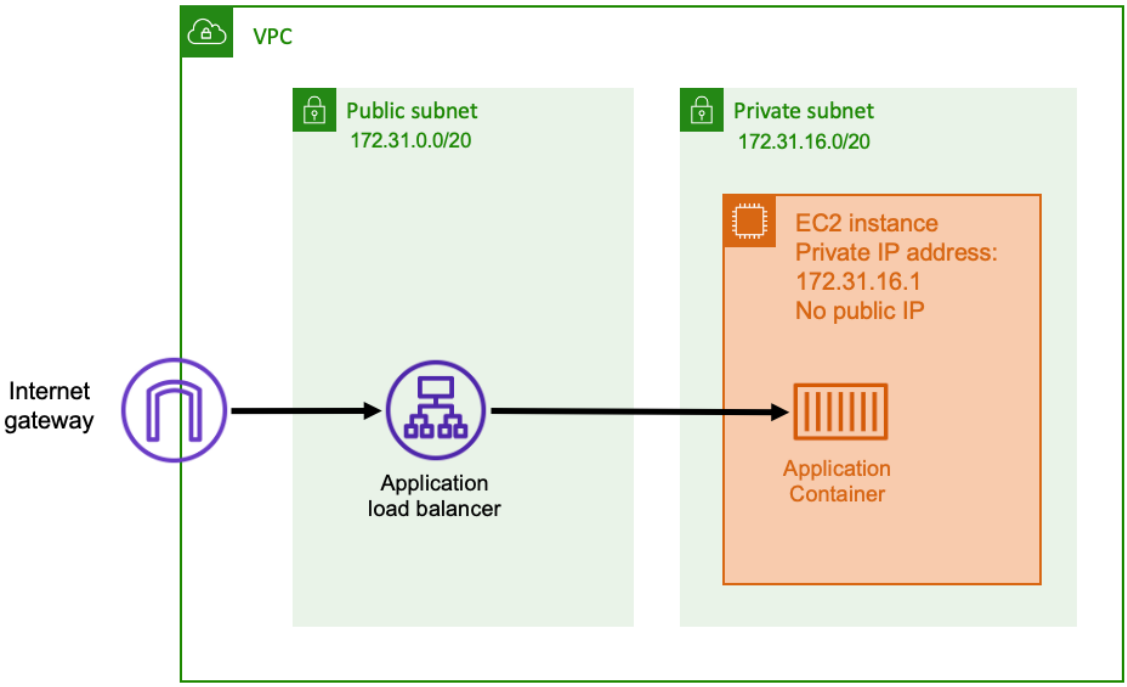
If you run a public service, you must accept inbound traffic from the internet. For example, your public website must accept inbound HTTP requests from browsers. In such case, other hosts on the internet must also initiate an inbound connection to the host of your application.

One approach to this problem is to launch your containers on hosts that are in a public subnet with a public IP address. However, we don't recommend this for large-scale applications. For these, a better approach is to have a scalable input layer that sits between the internet and your application. For this approach, you can use any of the AWS services listed in this section as an input.

Topics

Application Load Balancer

An Application Load Balancer functions at the application layer. It's the seventh layer of the Open Systems Interconnection (OSI) model. This makes an Application Load Balancer suitable for public HTTP services. If you have a website or an HTTP REST API, then an Application Load Balancer is a suitable load balancer for this workload. For more information, see What is an Application Load Balancer? in the *User Guide for Application Load Balancers*.



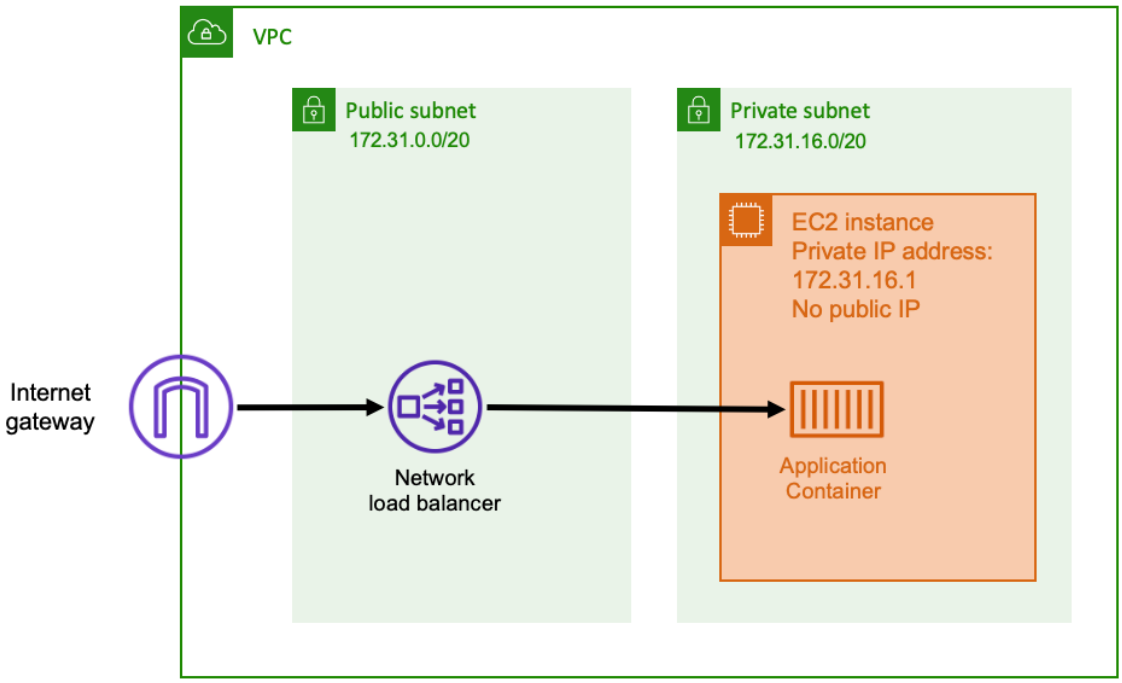
With this architecture, you create an Application Load Balancer in a public subnet so that it has a public IP address and can receive inbound connections from the internet. When the Application Load Balancer receives an inbound connection, or more specifically an HTTP request, it opens a connection to the application using its private IP address. Then, it forwards the request over the internal connection.

An Application Load Balancer has the following advantages.

- SSL/TLS termination — An Application Load Balancer can sustain secure HTTPS communication and certificates for communications with clients. It can optionally terminate the SSL connection at the load balancer level so that you don't have to handle certificates in your own application.
- Advanced routing — An Application Load Balancer can have multiple DNS hostnames. It also has advanced routing capabilities to send incoming HTTP requests to different destinations based on metrics such as the hostname or the path of the request. This means that you can use a single Application Load Balancer as the input for many different internal services, or even microservices on different paths of a REST API.
- gRPC support and websockets — An Application Load Balancer can handle more than just HTTP. It can also load balance gRPC and websocket based services, with HTTP/2 support.
- Security — An Application Load Balancer helps protect your application from malicious traffic. It includes features such as HTTP de sync mitigations, and is integrated with AWS Web Application Firewall (AWS WAF). AWS WAF can further filter out malicious traffic that might contain attack patterns, such as SQL injection or cross-site scripting.

Network Load Balancer

A Network Load Balancer functions at the fourth layer of the Open Systems Interconnection (OSI) model. It's suitable for non-HTTP protocols or scenarios where end-to-end encryption is necessary, but doesn't have the same HTTP-specific features of an Application Load Balancer. Therefore, a Network Load Balancer is best suited for applications that don't use HTTP. For more information, see What is a Network Load Balancer? in the *User Guide for Network Load Balancers*.



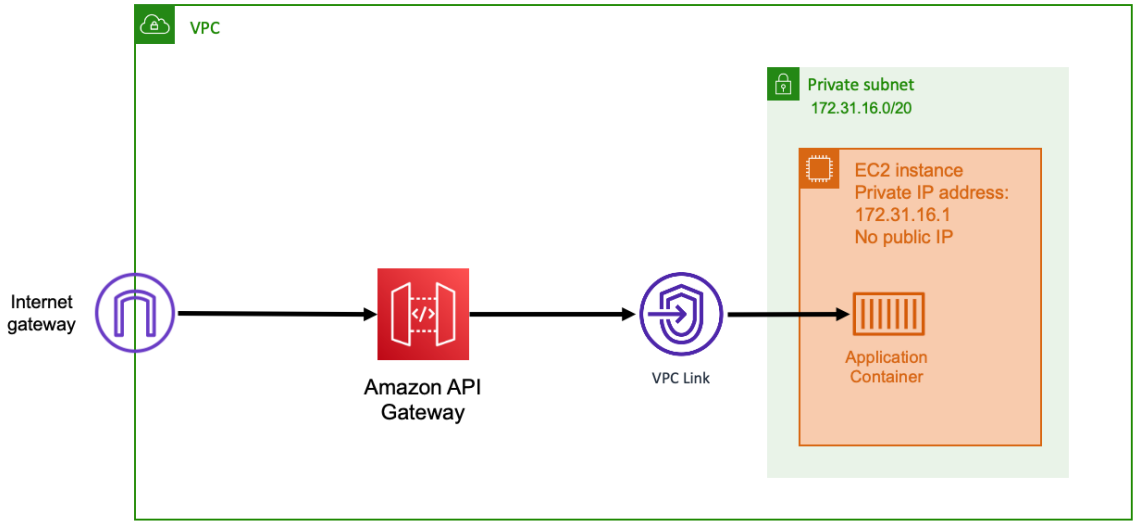
When a Network Load Balancer is used as an input, it functions similarly to an Application Load Balancer. This is because it's created in a public subnet and has a public IP address that can be accessed on the internet. The Network Load Balancer then opens a connection to the private IP address of the host running your container, and sends the packets from the public side to the private side.

Because the Network Load Balancer operates at a lower level of the networking stack, it doesn't have the same set of features that Application Load Balancer does. However, it does have the following important features.

- End-to-end encryption — Because a Network Load Balancer operates at the fourth layer of the OSI model, it doesn't read the contents of packets. This makes it suitable for load balancing communications that need end-to-end encryption.
- TLS encryption — In addition to end-to-end encryption, Network Load Balancer can also terminate TLS connections. This way, your backend applications don’t have to implement their own TLS.
- UDP support — Because a Network Load Balancer operates at the fourth layer of the OSI model, it's suitable for non HTTP workloads and protocols other than TCP.

Amazon API Gateway HTTP API

Amazon API Gateway HTTP API is a server less ingress that's suitable for HTTP applications with sudden bursts in request volumes or low request volumes. For more information, see What is Amazon API Gateway? in the *API Gateway Developer Guide*.



The pricing model for both Application Load Balancer and Network Load Balancer include an hourly price to keep the load balancers available for accepting incoming connections at all times. In contrast, API Gateway charges for each request separately. This has the effect that, if no requests come in, there are no charges. Under high traffic loads, an Application Load Balancer or Network Load Balancer can handle a greater volume of requests at a cheaper per-request price than API Gateway. However, if you have a low number of requests overall or have periods of low traffic, then the cumulative price for using the API Gateway should be more cost effective than paying a hourly charge to maintain a load balancer that's being underutilized.

API Gateway functions using a VPC link that allows the AWS managed service to connect to hosts inside the private subnet of your VPC, using its private IP address. It can detect these private IP addresses by looking at AWS Cloud Map service discovery records that are managed by Amazon ECS service discovery.

API Gateway supports the following features.

- SSL/TLS termination
- Routing different HTTP paths to different backend microservices

Besides the preceding features, API Gateway also supports using custom Lambda authorizers that you can use to protect your API from unauthorized usage. For more information, see Field Notes: Serverless Container-based APIs with Amazon ECS and Amazon API Gateway.

Choosing a network mode

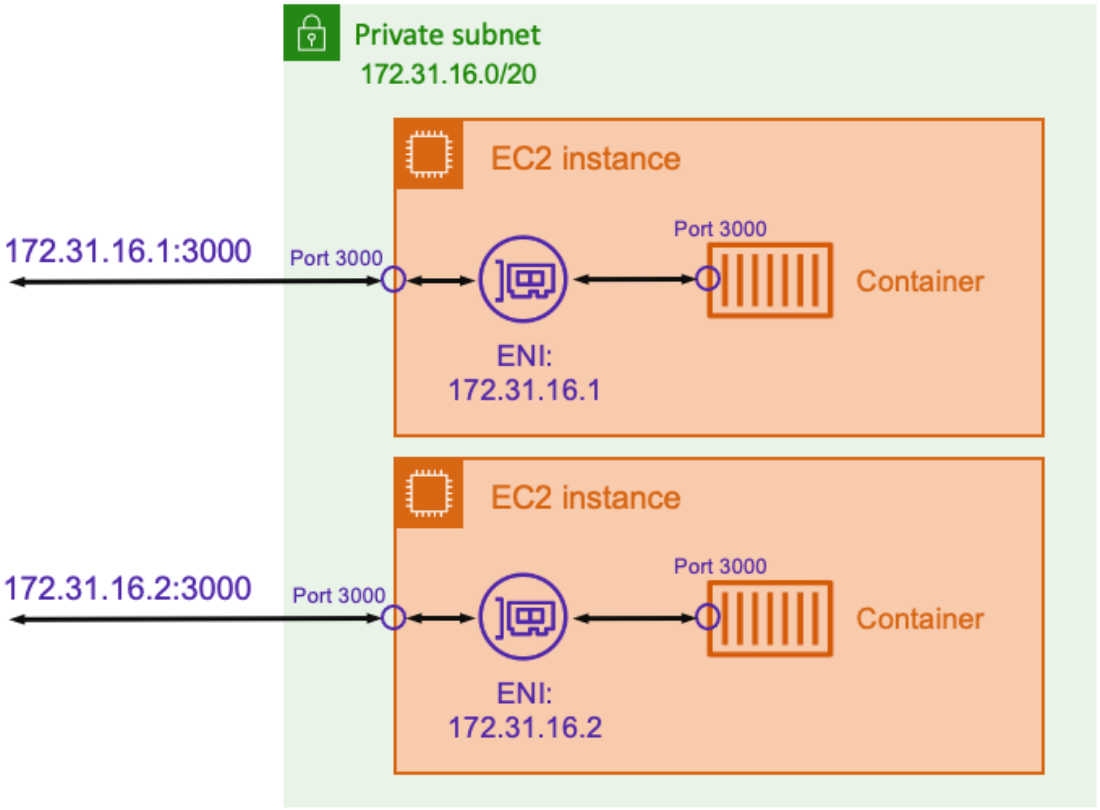
The approaches previously mentioned for architecting inbound and outbound network connections can apply to any of your workloads on AWS, even if they aren't inside a container. When running containers on AWS, you need to consider another level of networking. One of the main advantages of using containers is that you can pack multiple containers onto a single host. When doing this, you need to choose how you want to network the containers that are running on the same host. The following are the options to choose from.

Topics

- AWSVPC mode (p. 12)

Host mode

The `host` network mode is the most basic network mode that's supported in Amazon ECS. Using host mode, the networking of the container is tied directly to the underlying host that's running the container.



Assume that you're running a Node.js container with an Express application that listens on port `3000` similar to the one illustrated in the preceding diagram. When the `host` network mode is used, the container receives traffic on port 3000 using the IP address of the underlying host Amazon EC2 instance. We do not recommending using this mode.

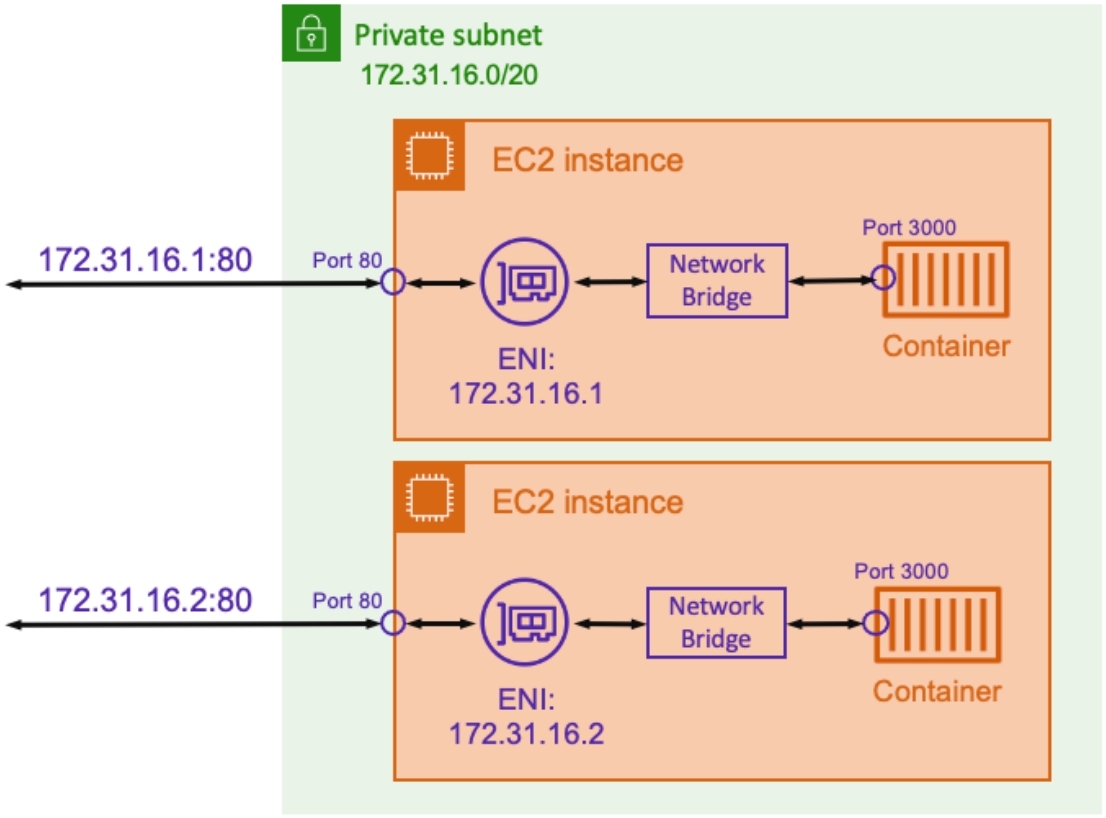
There are significant drawbacks to using this network mode. You can't run more than a single instantiation of a task on each host. This is because only the first task can bind to its required port on the Amazon EC2 instance. There's also no way to remap a container port when it's using `host` network mode. For example, if an application needs to listen on a particular port number, you can't remap the port number directly. Instead, you must manage any port conflicts through changing the application configuration.

There are also security implications when using the `host` network mode. This mode allows containers to impersonate the host, and it allows containers to connect to private loopback network services on the host.

The `host` network mode is only supported for Amazon ECS tasks hosted on Amazon EC2 instances. It's not supported when using Amazon ECS on Fargate.

Bridge mode

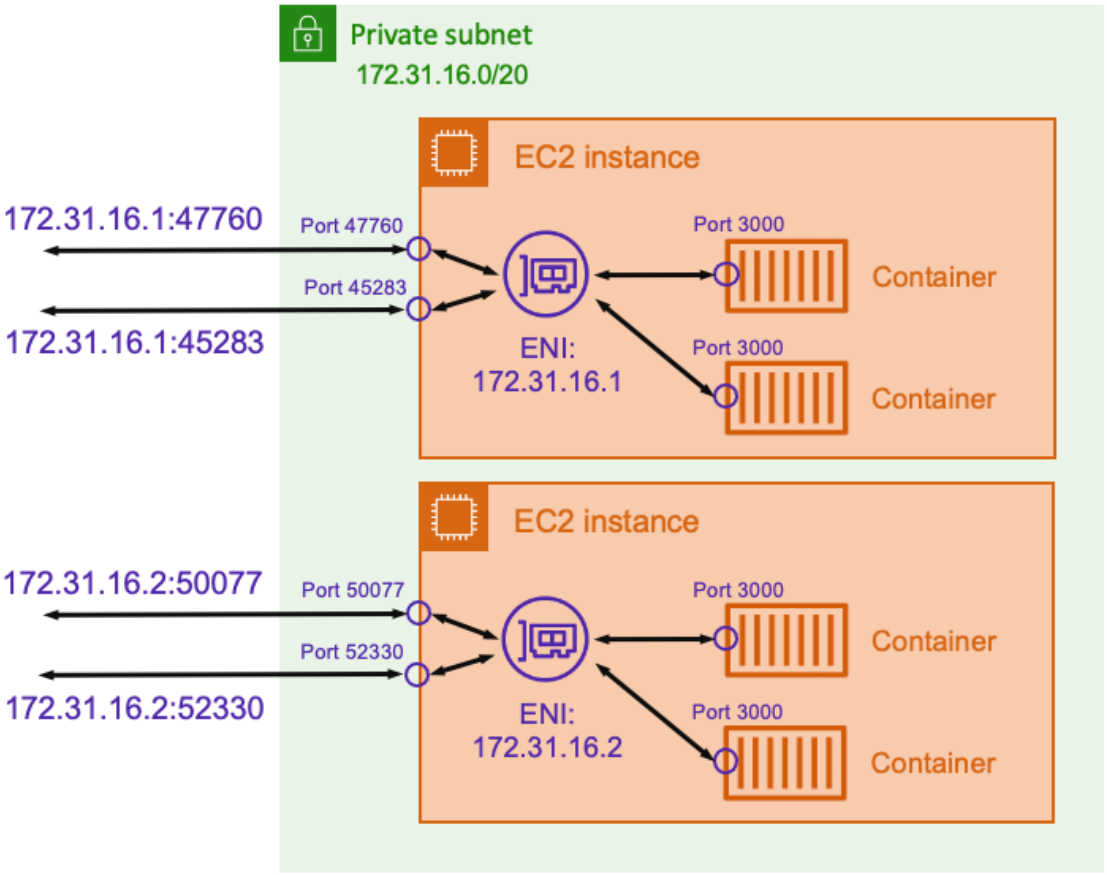
With `bridge` mode, you're using a virtual network bridge to create a layer between the host and the networking of the container. This way, you can create port mappings that remap a host port to a container port. The mappings can be either static or dynamic.



With a static port mapping, you can explicitly define which host port you want to map to a container port. Using the example above, port `80` on the host is being mapped to port `3000` on the container. To communicate to the containerized application, you send traffic to port `80` to the Amazon EC2 instance's IP address. From the containerized application's perspective it sees that inbound traffic on port `3000`.

If you only want to change the traffic port, then static port mappings is suitable. However, this still has the same disadvantage as using the `host` network mode. You can't run more than a single instantiation of a task on each host. This is because a static port mapping only allows a single container to be mapped to port 80.

To solve this problem, consider using the `bridge` network mode with a dynamic port mapping as shown in the following diagram.



By not specifying a host port in the port mapping, you can have Docker choose a random, unused port from the ephemeral port range and assign it as the public host port for the container. For example, the Node.js application listening on port `3000` on the container might be assigned a random high number port such as `47760` on the Amazon EC2 host. Doing this means that you can run multiple copies of that container on the host. Moreover, each container can be assigned its own port on the host. Each copy of the container receives traffic on port `3000`. However, clients that send traffic to these containers use the randomly assigned host ports.

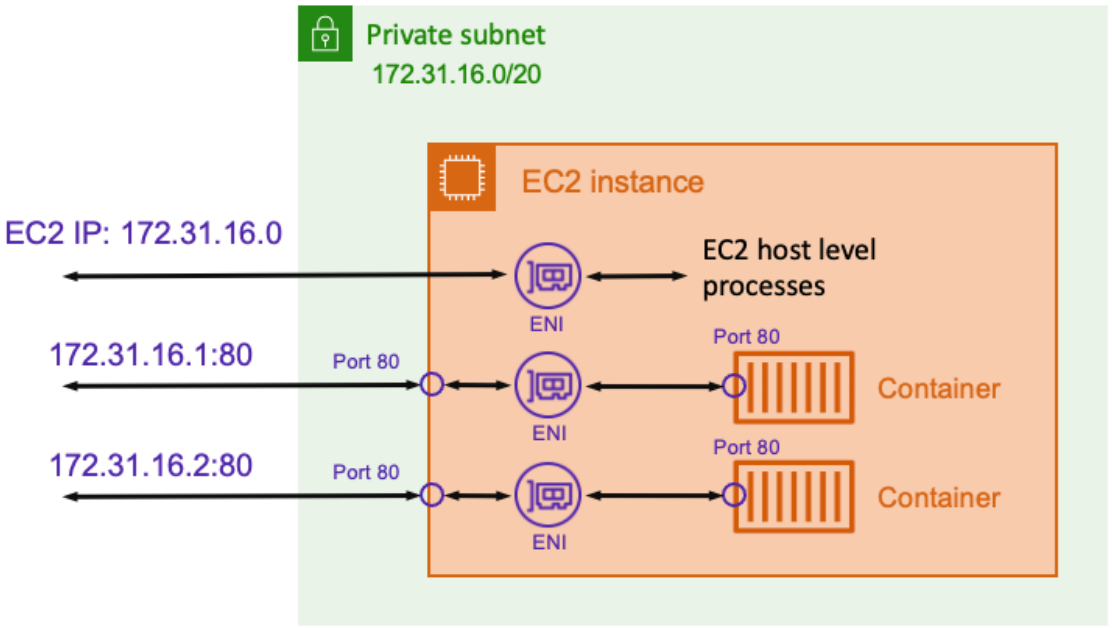
Amazon ECS helps you to keep track of the randomly assigned ports for each task. It does this by automatically updating load balancer target groups and AWS Cloud Map service discovery to have the list of task IP addresses and ports. This makes it easier to use services operating using `bridge` mode with dynamic ports.

However, one disadvantage of using the `bridge` network mode is that it's difficult to lock down service to service communications. Because services might be assigned to any random, unused port, it's necessary to open broad port ranges between hosts. However, it's not easy to create specific rules so that a particular service can only communicate to one other specific service. The services have no specific ports to use for security group networking rules.

The `bridge` network mode is only supported for Amazon ECS tasks hosted on Amazon EC2 instances. It is not supported when using Amazon ECS on Fargate.

AWSVPC mode

With the `awsvpc` network mode, Amazon ECS creates and manages an Elastic Network Interface (ENI) for each task and each task receives its own private IP address within the VPC. This ENI is separate from the underlying hosts ENI. If an Amazon EC2 instance is running multiple tasks, then each task's ENI is separate as well.



In the preceding example, the Amazon EC2 instance is assigned to an ENI. The ENI represents the IP address of the EC2 instance used for network communications at the host level. Each task also has a corresponding ENI and an private IP address. Because each ENI is separate, each container can bind to port `80` on the task ENI. Therefore, you don't need to keep track of port numbers. Instead, you can send traffic to port `80` at the IP address of the task ENI.

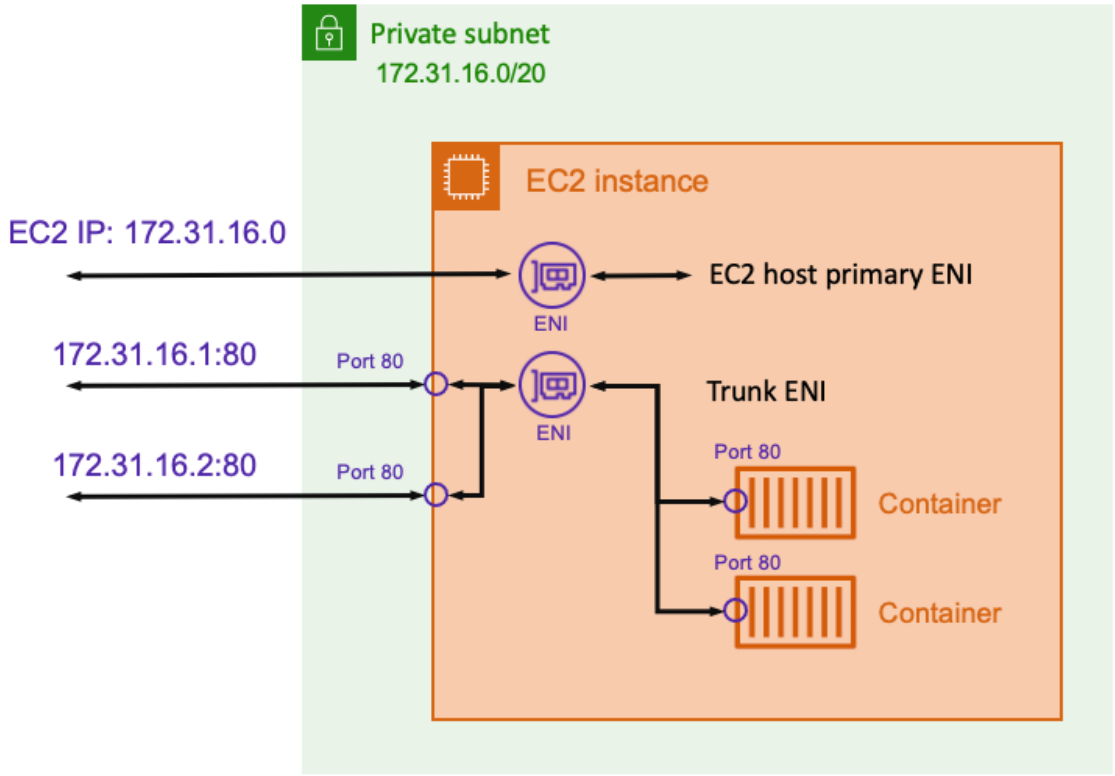
The advantage of using the `awsvpc` network mode is that each task has a separate security group to allow or deny traffic. This means you have greater flexibility to control communications between tasks and services at a more granular level. You can also configure a task to deny incoming traffic from another task located on the same host.

The `awsvpc` network mode is supported for Amazon ECS tasks hosted on both Amazon EC2 and Fargate. Be mindful that, when using Fargate, the `awsvpc` network mode is required.

When using the `awsvpc` network mode there are a few challenges you should be mindful of.

Increasing task density with ENI Trunking

The biggest disadvantage of using the `awsvpc` network mode with tasks that are hosted on Amazon EC2 instances is that EC2 instances have a limit on the number of ENIs that can be attached to them. This limits how many tasks you can place on each instance. Amazon ECS provides the ENI trunking feature which increases the number of available ENIs to achieve more task density.



When using ENI trunking, two ENI attachments are used by default. The first is the primary ENI of the instance, which is used for any host level processes. The second is the trunk ENI, which Amazon ECS creates. This feature is only supported on specific Amazon EC2 instance types.

Consider this example. Without ENI trunking, a `c5.large` instance that has two vCPUs can only host two tasks. However, with ENI trunking, a `c5.large` instance that has two vCPU's can host up to ten tasks. Each task has a different IP address and security group. For more information about available instance types and their density, see Supported Amazon EC2 instance types in the *Amazon Elastic Container Service Developer Guide*.

ENI trunking has no impact on runtime performance in terms of latency or bandwidth. However, it increases task startup time. You should ensure that, if ENI trunking is used, your autoscaling rules and other workloads that depend on task startup time still function as you expect them to.

For more information, see Elastic network interface trunking in the *Amazon Elastic Container Service Developer Guide*.

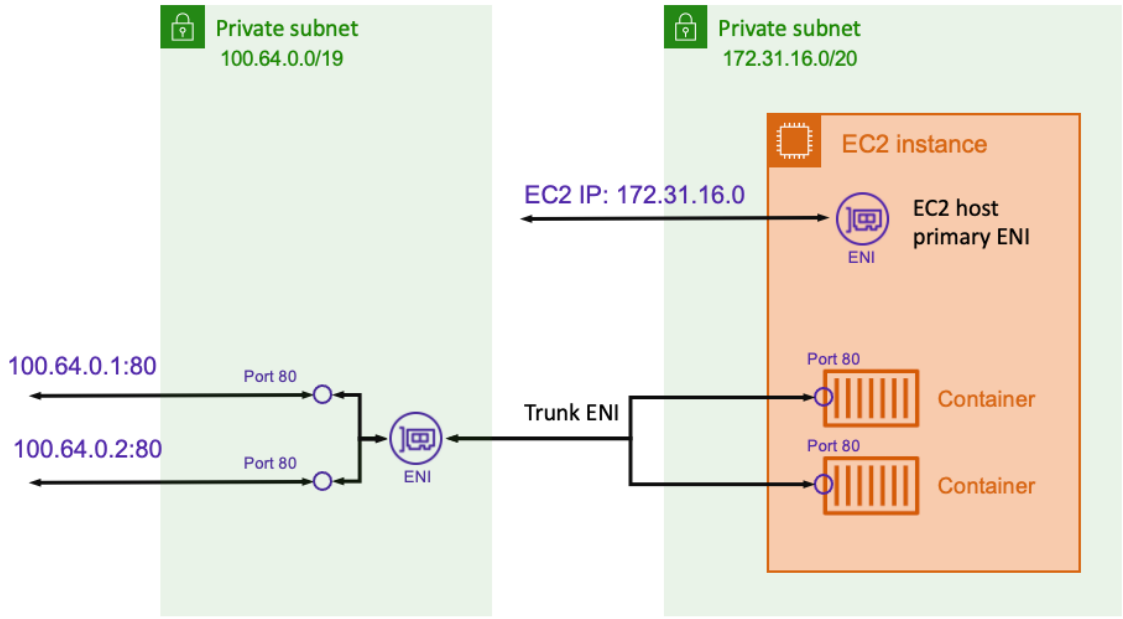
Preventing IP address exhaustion

By assigning a separate IP address to each task, you can simplify your overall infrastructure and maintain security groups that provide a great level of security. However, this configuration can lead to IP exhaustion.

The default VPC on your AWS account has pre-provisioned subnets that have a `/20` CIDR range. This means each subnet has 4,091 available IP addresses. Note that several IP addresses within the `/20` range are reserved for AWS specific usage. Consider this example. You distribute your applications across three subnets in three Availability Zones for high availability. In this case, you can use approximately 12,000 IP addresses across the three subnets.

Using ENI trunking, each Amazon EC2 instance that you launch requires two IP addresses. One IP address is used for the primary ENI, and the other IP address is used for the trunk ENI. Each Amazon ECS task on the instance requires one IP address. If you're launching an extremely large workload, you could run out of available IP addresses. This might result in Amazon EC2 launch failures or task launch failures. These errors occur because the ENIs can't add IP addresses inside the VPC if there are no available IP addresses.

When using the `awsvpc` network mode, you should evaluate your IP address requirements and ensure that your subnet CIDR ranges meet your needs. If you have already started using a VPC that has small subnets and begins to run out of address space, you can add a secondary subnet.



By using ENI trunking, the Amazon VPC CNI can be configured to use ENIs in a different IP address space than the host. By doing this, you can give your Amazon EC2 host and your tasks different IP address ranges that don't overlap. In the example diagram, the EC2 host IP address is in a subnet that has the `172.31.16.0/20` IP range. However, tasks that are running on the host are assigned IP addresses in the `100.64.0.0/19` range. By using two independent IP ranges, you don't need to worry about tasks consuming too many IP addresses and not leaving enough IP addresses for instances.

Using IPv6 dual stack mode

The `awsvpc` network mode is compatible with VPCs that are configured for IPv6 dual stack mode. A VPC using dual stack mode can communicate over IPv4, IPv6, or both. Each subnet in the VPC can have both an IPv4 CIDR range and an IPv6 CIDR range. For more information, see IP addressing in your VPC in the *Amazon VPC User Guide*.

You can't disable IPv4 support for your VPC and subnets to address IPv4 exhaustion issues. However, with the IPv6 support, you can use some new capabilities, specifically the egress-only internet gateway. An egress-only internet gateway allows tasks to use their publicly routable IPv6 address to initiate outbound connections to the internet. But the egress-only internet gateway doesn't allow connections from the internet. For more information, see Egress-only internet gateways in the *Amazon VPC User Guide*.

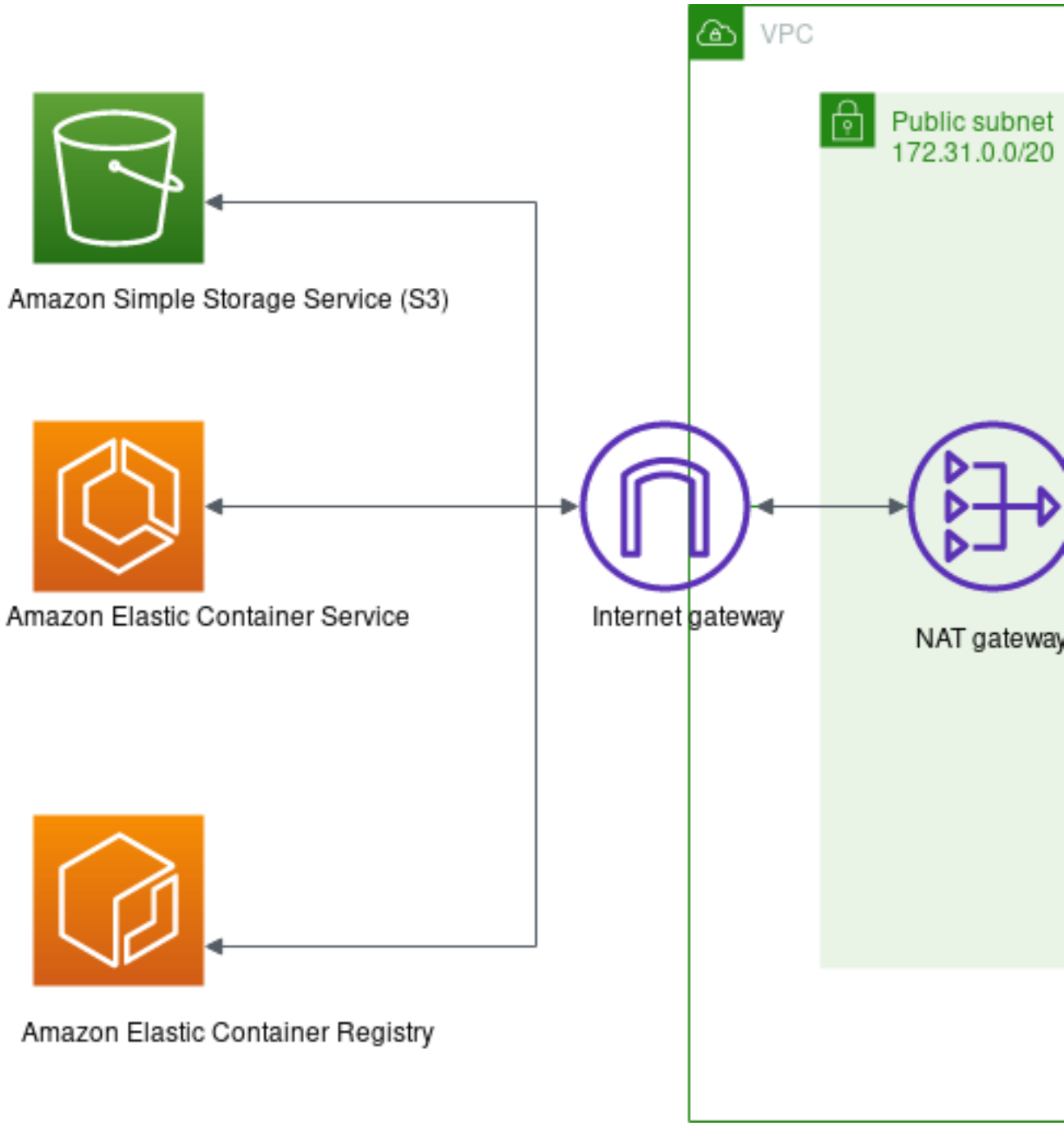
Connecting to AWS services from inside your VPC

For Amazon ECS to function properly, the ECS container agent that runs on each host must communicate with the Amazon ECS control plane. If you're storing your container images in Amazon ECR, the Amazon EC2 hosts must communicate to the Amazon ECR service endpoint, and to Amazon S3, where the image layers are stored. If you use other AWS services for your containerized application, such as persisting data stored in DynamoDB, double-check that these services also have the necessary networking support.

Topics

NAT gateway

Using a NAT gateway is the easiest way to ensure that your Amazon ECS tasks can access other AWS services. For more information about this approach, see Using a private subnet and NAT gateway (p. 4).



The following are the disadvantages to using this approach:

- You can't limit what destinations the NAT gateway can communicate with. You also can't limit which destinations your backend tire can communicate to without disrupting all outbound communications from your VPC.
- NAT gateways charge for every GB of data that passes through. If you use the NAT gateway for downloading large files from Amazon S3, or doing a high volume of database queries to DynamoDB, you're charged for every GB of bandwidth. Additionally, NAT gateways support 5 Gbps of bandwidth and automatically scale up to 45 Gbps. If you route through a single NAT gateway, applications that

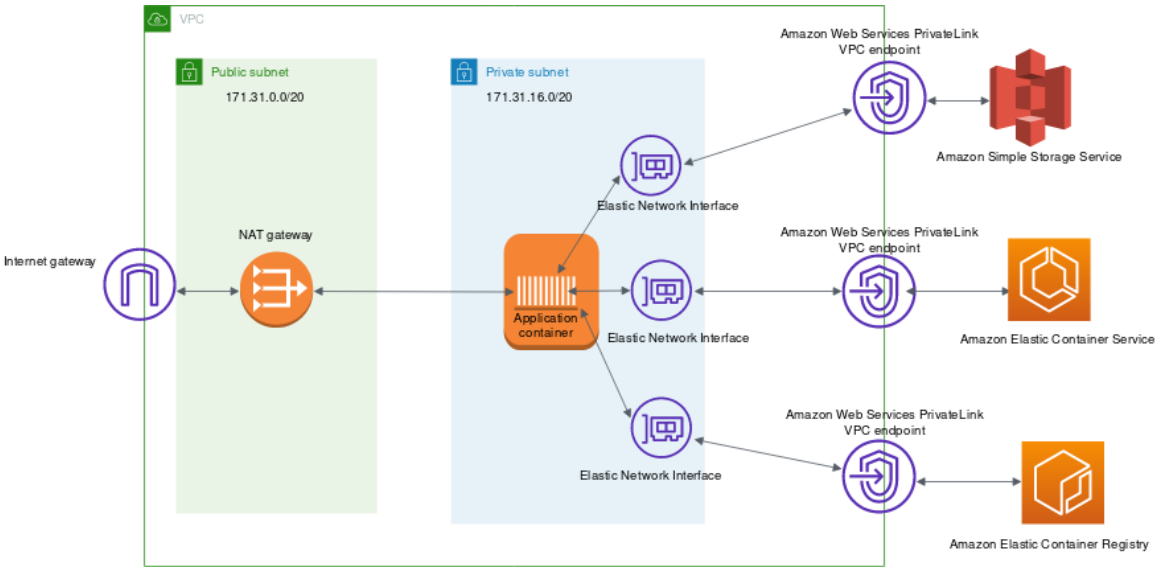
require very high bandwidth connections might encounter networking constraints. As a workaround, you can divide your workload across multiple subnets and give each subnet its own NAT gateway.

AWS PrivateLink

AWS PrivateLink provides private connectivity between VPCs, AWS services, and your on-premises networks without exposing your traffic to the public internet.

One of the technologies used to accomplish this is the VPC endpoint. A VPC endpoint enables private connections between your VPC and supported AWS services and VPC endpoint services. Traffic between your VPC and the other service doesn't leave the Amazon network. A VPC endpoint doesn't require an internet gateway, virtual private gateway, NAT device, VPN connection, or AWS Direct Connect connection. Amazon EC2 instances in your VPC don't require public IP addresses to communicate with resources in the service.

The following diagram shows how communication to AWS services works when you are using VPC endpoints instead of an internet gateway. AWS PrivateLink provisions elastic network interfaces (ENIs) inside of the subnet, and VPC routing rules are used to send any communication to the service hostname through the ENI, directly to the destination AWS service. This traffic no longer needs to use the NAT gateway or internet gateway.



The following are some of the common VPC endpoints that are used with the Amazon ECS service.

- S3 gateway VPC endpoint
- DynamoDB VPC endpoint
- Amazon ECS VPC endpoint
- Amazon ECR VPC endpoint

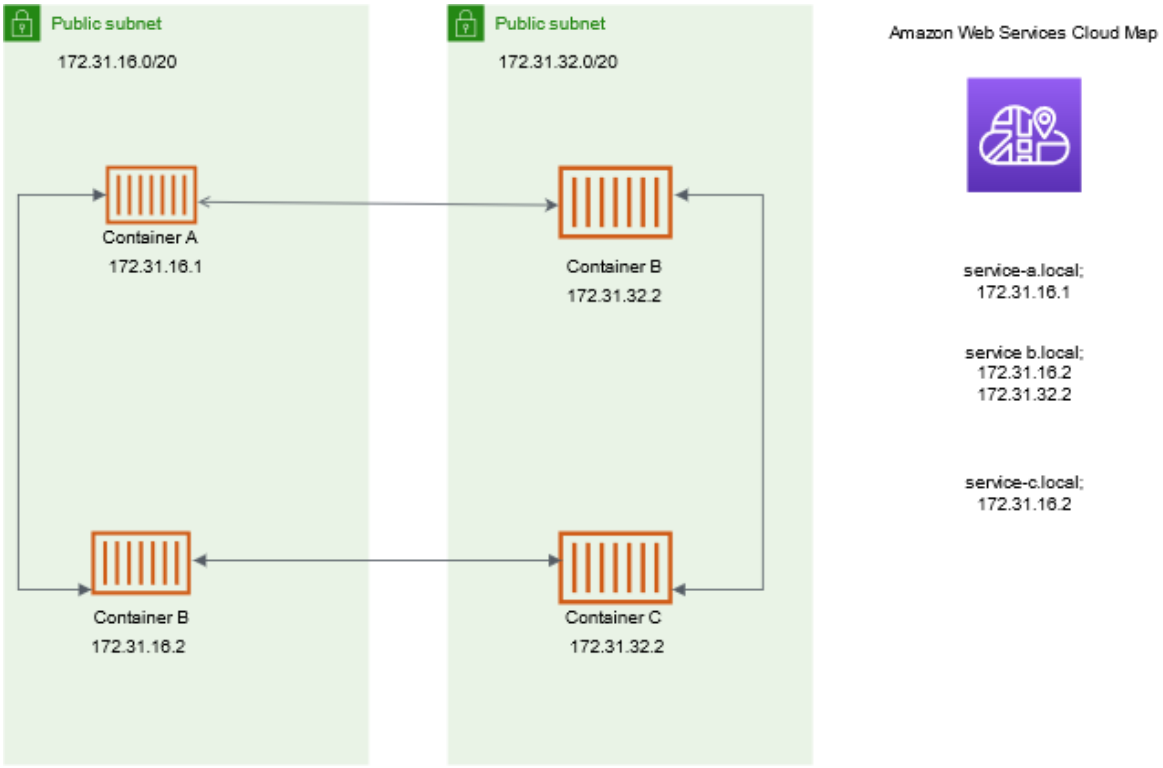
Many other AWS services support VPC endpoints. If you make heavy usage of any AWS service, you should look up the specific documentation for that service and how to create a VPC endpoint for that traffic.

Networking between Amazon ECS services in a VPC

Using Amazon ECS containers in a VPC, you can spilt monolithic applications into separate parts that can be deployed and scaled independently in a secure environment. However, it can be challenging to make sure that all of these parts, both in and outside of a VPC, can communicate with each other. There are several approaches for facilitating communication, all with different advantages and disadvantages.

Using service discovery

One approach for service-to-service communication is direct communication using service discovery. In this approach, you can use the AWS Cloud Map service discovery integration with Amazon ECS. Using service discovery, Amazon ECS syncs the list of launched tasks to AWS Cloud Map, which maintains a DNS hostname that resolves to the internal IP addresses of one or more tasks from that particular service. Other services in the Amazon VPC can use this DNS hostname to send traffic directly to another container using its internal IP address. For more information, see Service discovery in the *Amazon Elastic Container Service Developer Guide*.



In the preceding diagram, there are three services. `serviceA` has one container and communicates with `serviceB`, which has two containers. `serviceB` must also communicate with `serviceC`, which has one container. Each container in all three of these services can use the internal DNS names from AWS Cloud Map to find the internal IP addresses of a container from the downstream service that it needs to communicate to.

This approach to service-to-service communication provides low latency. At first glance, it's also simple as there are no extra components between the containers. Traffic travels directly from one container to the other container.

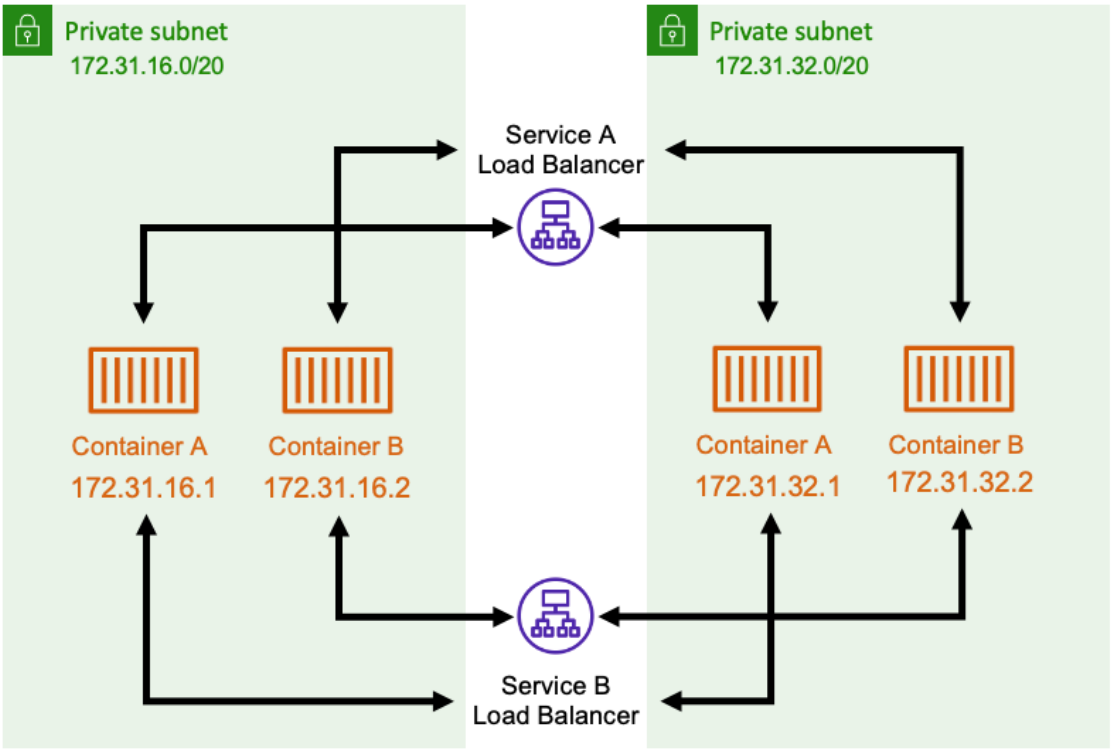
This approach is suitable when using the `awsvpc` network mode, where each task has its own unique IP address. Most software only supports the use of DNS `A` records, which resolve directly to IP addresses. When using the `awsvpc` network mode, the IP address for each task are an `A` record. However, if you're using `bridge` network mode, multiple containers could be sharing the same IP address. Additionally, dynamic port mappings cause the containers to be randomly assigned port numbers on that single IP address. At this point, an `A` record is no longer be enough for service discovery. You must also use an `SRV` record. This type of record can keep track of both IP addresses and port numbers but requires that you configure applications appropriately. Some prebuilt applications that you use might not support `SRV` records.

Another advantage of the `awsvpc` network mode is that you have a unique security group for each service. You can configure this security group to allow incoming connections from only the specific upstream services that need to talk to that service.

The main disadvantage of direct service-to-service communication using service discovery is that you must implement extra logic to have retries and deal with connection failures. DNS records have a time-to-live (TTL) period that controls how long they are cached for. It takes some time for the DNS record to be updated and for the cache to expire so that your applications can pick up the latest version of the DNS record. So, your application might end up resolving the DNS record to point at another container that's no longer there. Your application needs to handle retries and have logic to ignore bad backends.

Using an internal load balancer

Another approach to service-to-service communication is to use an internal load balancer. An internal load balancer exists entirely inside of your VPC and is only accessible to services inside of your VPC.



The load balancer maintains high availability by deploying redundant resources into each subnet. When a container from `serviceA` needs to communicate with a container from `serviceB`, it opens a connection to the load balancer. The load balancer then opens a connection to a container from

`service B`. The load balancer serves as a centralized place for managing all connections between each service.

If a container from `serviceB` stops, then the load balancer can remove that container from the pool. The load balancer also does health checks against each downstream target in its pool and can automatically remove bad targets from the pool until they become healthy again. The applications no longer need to be aware of how many downstream containers are there. They just open their connections to the load balancer.

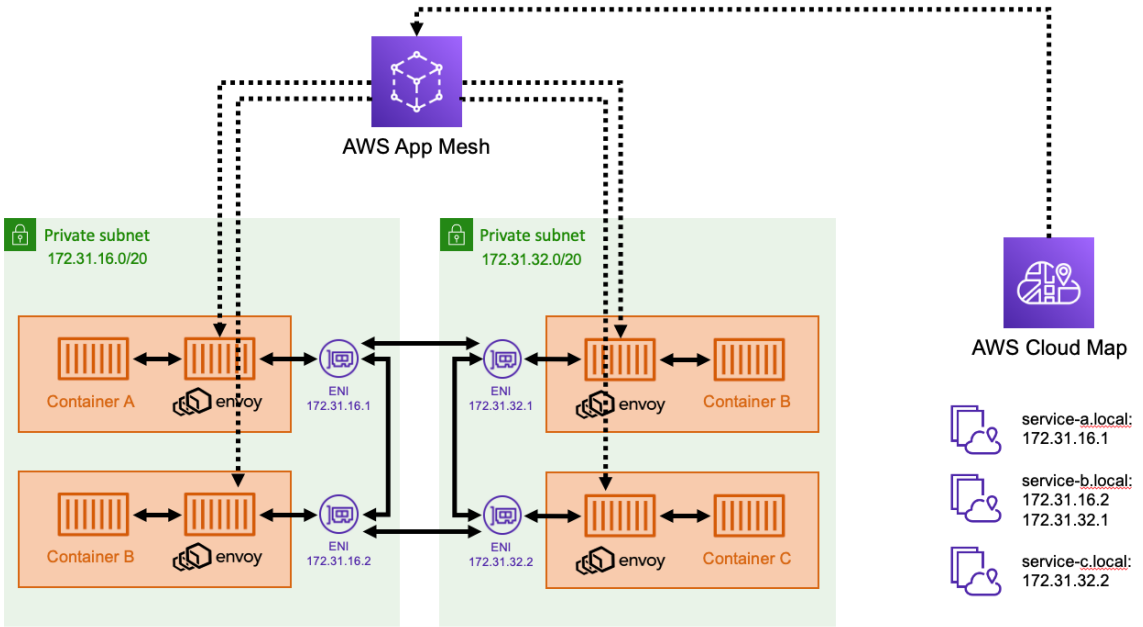
This approach is advantageous to all network modes. The load balancer can keep track of task IP addresses when using the `awsvpc` network mode, as well as more advanced combinations of IP address and port when using the `bridge` network mode. It evenly distribute traffic across all the IP address and port combinations, even if several containers are actually hosted on the same Amazon EC2 instance, just on different ports.

The one disadvantage of this approach is cost. To be highly available, the load balancer needs to have resources in each Availability Zone. This adds extra cost because of the overhead of paying for the load balancer and for the amount of traffic that goes through the load balancer.

However, you can reduce overhead costs by having multiple services share a load balancer. This is particularly suitable for REST services that use an Application Load Balancer. You can create path-based routing rules that route traffic to different services. For example, `/api/user/*` might route to a container that's part of the `user` service, whereas `/api/order/*` might route to the associated `order` service. With this approach, you only pay for one Application Load Balancer, and have one consistent URL for your API. However, you can split the traffic off to various microservices on the backend.

Using a service mesh

AWS App Mesh is a service mesh that can help you manage a large number of services and have better control of how traffic gets routed among services. App Mesh functions as an intermediary between basic service discovery and load balancing. With App Mesh, applications don't directly interact with each other, but they also don't use a centralized load balancer either. Instead, each copy of your task is accompanied by an Envoy proxy sidecar. For more information, see What is AWS App Mesh in the *AWS App Mesh User Guide*.



In the preceding diagram, each task has an Envoy proxy sidecar. This sidecar is responsible for proxying all inbound and outbound traffic for the task. The App Mesh control plane uses AWS Cloud Map to get the list of available services and the IP addresses of specific tasks. Then App Mesh delivers the configuration to the Envoy proxy sidecar. This configuration includes the list of available containers that can be connected to. The Envoy proxy sidecar also conducts health checks against each target to ensure that they're available.

This approach provides the features of service discovery, with the ease of the managed load balancer. Applications don't implement as much load balancing logic within their code because the Envoy proxy sidecar handles that load balancing. The Envoy proxy can be configured to detect failures and retry failed requests. Additionally, it can also be configured to use mTLS to encrypt traffic in transit, and ensure that your applications are communicating to a verified destination.

There are a few differences between an Envoy proxy and a load balancer. In short, with Envoy proxy, you're responsible for deploying and managing your own Envoy proxy sidecar. The Envoy proxy sidecar uses some of the CPU and memory that you allocate to the Amazon ECS task. This adds some overhead to the task resource consumption, and additional operational workload to maintain and update the proxy when needed.

App Mesh and an Envoy proxy allows for extremely low latency between tasks. This is because Envoy proxy runs collocated to each task. There's only one instance to instance network jump, between one Envoy proxy and another Envoy proxy. This means there's also less network overhead compared to when using load balancers. When using load balancers, there are two network jumps. The first is from the upstream task to the load balancer, and the second is from the load balancer to the downstream task.

Networking services across AWS accounts and VPCs

If you're part of an organization with multiple teams and divisions, you probably deploy services independently into separate VPCs inside a shared AWS account or into VPCs that are associated with multiple individual AWS accounts. No matter which way you deploy your services, we recommend that you supplement your networking components to help route traffic between VPCs. For this, several AWS services can be used to supplement your existing networking components.

- AWS Transit Gateway — You should consider this networking service first. This service serves as a central hub for routing your connections between Amazon VPCs, AWS accounts, and on-premises networks. For more information, see What is a transit gateway? in the *Amazon VPC Transit Gateways Guide*.
- Amazon VPC and VPN support — You can use this service to create site-to-site VPN connections for connecting on-premises networks to your VPC. For more information, see What is AWS Site-to-Site VPN? in the *AWS Site-to-Site VPN User Guide*.
- Amazon VPC — You can use Amazon VPC peering to help you to connect multiple VPCs, either in the same account, or across accounts. For more information, see What is VPC peering? in the *Amazon VPC Peering Guide*.
- Shared VPCs — You can use a VPC and VPC subnets across multiple AWS accounts. For more information, see Working with shared VPCs in the *Amazon VPC User Guide*.

Optimizing and troubleshooting

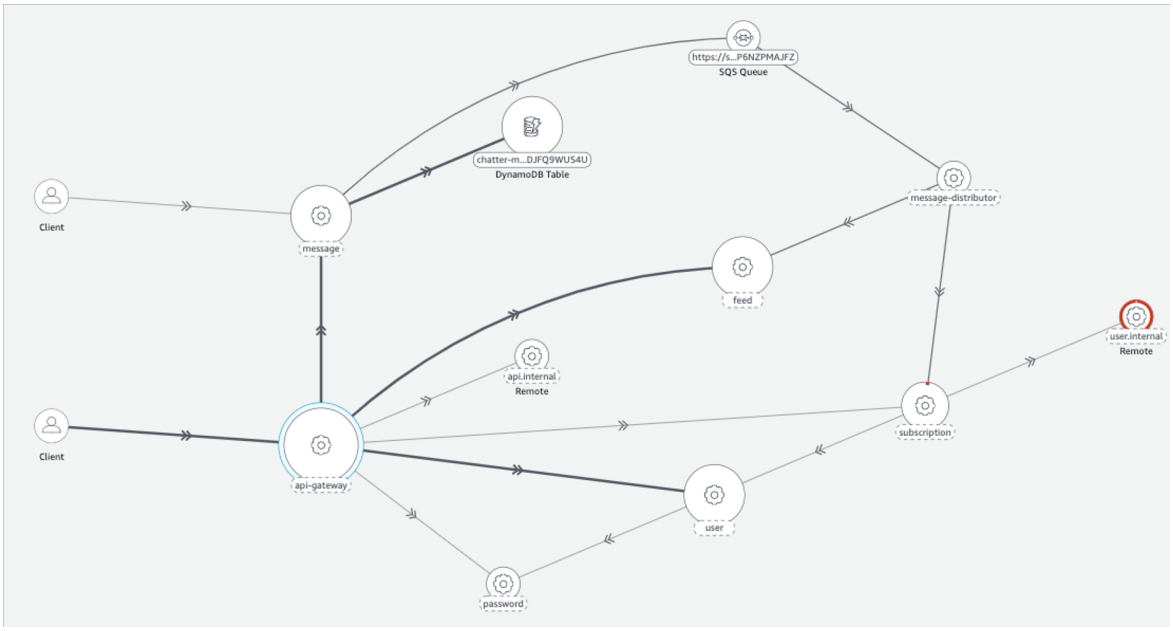
The following services and features can help you to gain insights about your network and service configurations. You can use this information to troubleshoot networking issues and better optimize your services.

CloudWatch Container Insights

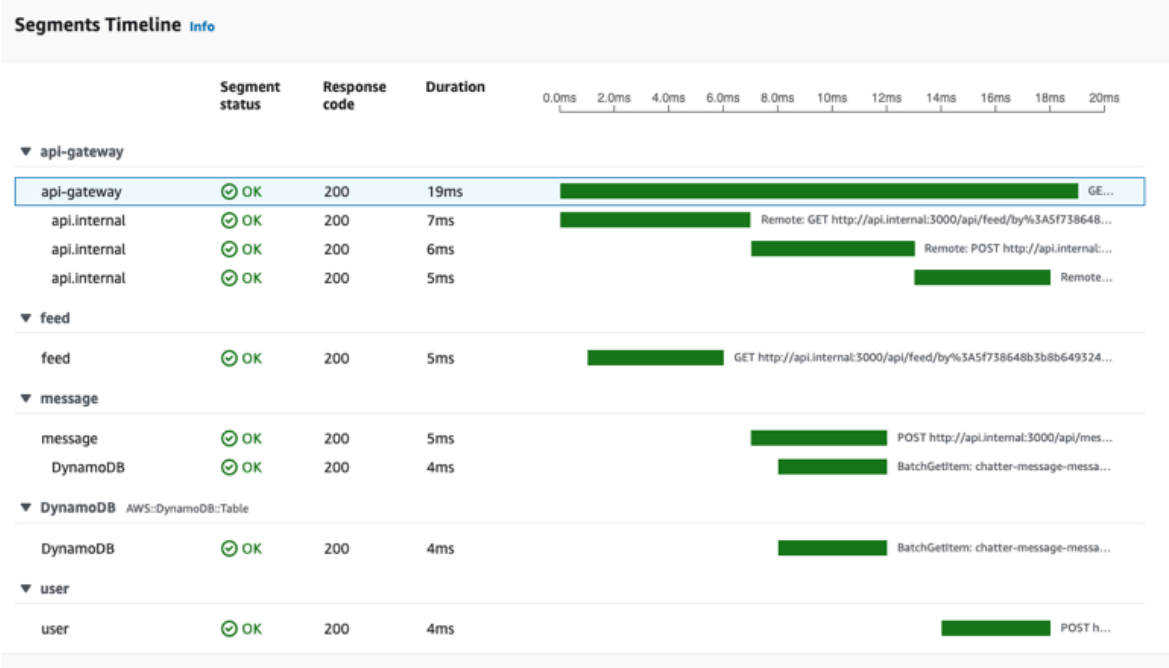
CloudWatch Container Insights collects, aggregates, and summarizes metrics and logs from your containerized applications and microservices. Metrics include the utilization of resources such as CPU, memory, disk, and network. They're available in CloudWatch automatic dashboards. For more information, see Setting up Container Insights on Amazon ECS in the *Amazon CloudWatch User Guide*.

AWS X-Ray

AWS X-Ray is a tracing service that you can use to collect information about the network requests that your application makes. You can use the SDK to instrument your application and capture timings and response codes of traffic between your services, and between your services and AWS service endpoints. For more information, see What is AWS X-Ray in the *AWS X-Ray Developer Guide*.



You can also explore AWS X-Ray graphs of how your services network with each other. Or, use them to explore aggregate statistics about how each service-to-service link is performing. Last, you can dive deeper into any specific transaction to see how segments representing network calls are associated with that particular transaction.



You can use these features to identify if there's a networking bottleneck or if a specific service within your network isn't performing as expected.

VPC Flow Logs

You can use Amazon VPC flow logs to analyze network performance and debug connectivity issues. With VPC flow logs enabled, you can capture a log of all the connections in your VPC. These include connections to networking interfaces that are associated with Elastic Load Balancing, Amazon RDS, NAT gateways, and other key AWS services that you might be using. For more information, see VPC Flow Logs in the *Amazon VPC User Guide*.

Network tuning tips

There are a few settings that you can fine tune in order to improve your networking.

nofile ulimit

If you expect your application to have high traffic and handle many concurrent connections, you should take into account the system quota for the number of files allowed. When there are a lot of network sockets open, each one must be represented by a file descriptor. If your file descriptor quota is too low, it will limit your network sockets,. This results in failed connections or errors. You can update the container specific quota for the number of files in the Amazon ECS task definition. If you're running on Amazon EC2 (instead of AWS Fargate), then you might also need to adjust these quotas on your underlying Amazon EC2 instance.

sysctl net

Another category of tunable setting is the `sysctl` net settings. You should refer to the specific settings for your Linux distribution of choice. Many of these settings adjust the size of read and write buffers. This can help in some situations when running large-scale Amazon EC2 instances that have a lot of containers on them.

Best Practices - Auto scaling and capacity management

Amazon ECS is used to run containerized application workloads of all sizes. This includes both the extremes of minimal testing environments and large production environments operating at a global scale.

With Amazon ECS, like all AWS services, you pay only for what you use. When architected appropriately, you can save costs by having your application consume only the resources that it needs at the time that it needs them. This best practices guide shows how to run your Amazon ECS workloads in a way that meets your service-level objectives while still operating in a cost-effective manner.

Topics

Determining task size

One of the most important choices to make when deploying containers on Amazon ECS is your container and task sizes. Your container and task sizes are both essential for scaling and capacity planning. In Amazon ECS, there are two resource metrics used for capacity: CPU and memory. CPU is measured in units of 1/1024 of a full vCPU (where 1024 units is equal to 1 whole vCPU). Memory is measured in megabytes. In your task definition, you can declare resource reservations and limits.

When you declare a reservation, you're declaring the minimum amount of resources that a task requires. Your task receives at least the amount of resources requested. Your application might be able to use more CPU or memory than the reservation that you declare. However, this is subject to any limits that you also declared. Using more than the reservation amount is known as bursting. In Amazon ECS, reservations are guaranteed. For example, if you use Amazon EC2 instances to provide capacity, Amazon ECS doesn't place a task on an instance where the reservation can't be fulfilled.

A limit is the maximum amount of CPU units or memory that your container or task can use. Any attempt to use more CPU more than this limit results in throttling. Any attempt to use more memory results in your container being stopped.

Choosing these values can be challenging. This is because the values that are the most well suited for your application greatly depend on the resource requirements of your application. Load testing your application is the key to successful resource requirement planning and better understanding your application's requirements.

Stateless applications

For stateless applications that scale horizontally, such as an application behind a load balancer, we recommend that you first determine the amount of memory that your application consumes when it

serves requests. To do this, you can use traditional tools such as `ps` or `top`, or monitoring solutions such as CloudWatch Container Insights.

When determining a CPU reservation, consider how you want to scale your application to meet your business requirements. You can use smaller CPU reservations, such as 256 CPU units (or 1/4 vCPU), to scale out in a fine-grained way that minimizes cost. But, they might not scale fast enough to meet significant spikes in demand. You can use larger CPU reservations to scale in and out more quickly and therefore match demand spikes more quickly. However, larger CPU reservations are more costly.

Other applications

For applications that don't scale horizontally, such as singleton workers or database servers, available capacity and cost represent your most important considerations. You should choose the amount of memory and CPU based on what load testing indicates you need to serve traffic to meet your service-level objective. Amazon ECS ensures that the application is placed on a host that has adequate capacity.

Configuring service auto scaling

An Amazon ECS service is a managed collection of tasks. Each service has an associated task definition, a desired task count, and an optional placement strategy. Amazon ECS service auto scaling is implemented through the Application Auto Scaling service. Application Auto Scaling uses CloudWatch metrics as the source for scaling metrics. It also uses CloudWatch alarms to set thresholds on when to scale your service in or out. You provide the thresholds for scaling, either by setting a metric target, referred to as *target tracking scaling*, or by specifying thresholds, referred to as *step scaling*. After Application Auto Scaling is configured, it continually calculates the appropriate desired task count for the service. It also notifies Amazon ECS when the desired task count should change, either by scaling it out or scaling it in.

To use service auto scaling effectively, you must choose an appropriate scaling metric. We discuss how to choose a metric in the following sections.

Characterizing your application

Properly scaling an application requires knowing the conditions where the application should be scaled in and when it should be scaled out. In essence, an application should be scaled out if demand is forecasted to outstrip capacity. Conversely, an application can be scaled in to conserve costs when resources exceed demand.

Identifying a utilization metric

To scale effectively, it's critical to identify a metric that indicates utilization or saturation. This metric must exhibit the following properties to be useful for scaling.

- The metric must be correlated with demand. When resources are held steady, but demand changes, the metric value must also change. The metric should increase or decrease when demand increases or decreases.
- The metric value must scale in proportion to capacity. When demand holds constant, adding more resources must result in a proportional change in the metric value. So, doubling the number of tasks should cause the metric to decrease by 50%.

The best way to identify a utilization metric is through load testing in a pre-production environment such as a staging environment. Commercial and open-source load testing solutions are widely available. These solutions typically can either generate synthetic load or simulate real user traffic.

To start the process of load testing, you should start by building dashboards for your application's utilization metrics. These metrics include CPU utilization, memory utilization, I/O operations, I/O queue depth, and network throughput. You can collect these metrics with a service such as CloudWatch Container Insights. Or, do so by using Amazon Managed Service for Prometheus together with Amazon Managed Grafana. During this process, make sure that you collect and plot metrics for your application's response times or work completion rates.

When load testing, begin with a small request or job insertion rate. Keep this rate steady for several minutes to allow your application to warm up. Then, slowly increase the rate and hold it steady for a few minutes. Repeat this cycle, increasing the rate each time until your application's response or completion times are too slow to meet your service-level objectives (SLOs).

While load testing, examine each of the utilization metrics. The metrics that increase along with the load are the top candidates to serve as your best utilization metrics.

Next, identify the resource that reaches saturation. At the same time, also examine the utilization metrics to see which one flattens out at a high level first. Or, examine which one reaches peak and then crashes your application first. For example, if CPU utilization increases from 0% to 70-80% as you add load, then stays at that level after even more load is added, then it's safe to say that the CPU is saturated. Depending on the CPU architecture, it might never reach 100%. For example, assume that memory utilization increases as you add load, and then your application suddenly crashes when it reaches the task or Amazon EC2 instance memory limit. In this situation, it's likely the case that memory has been fully consumed. Multiple resources might be consumed by your application. Therefore, choose the metric that represents the resource that depletes first.

Last, try load testing again after doubling the number of tasks or Amazon EC2 instances. Assume that the key metric increases, or decreases, at half the rate as before. If this is the case, then the metric is proportional to capacity. This is a good utilization metric for auto scaling.

Now consider this hypothetical scenario. Suppose that you load test an application and find that the CPU utilization eventually reaches 80% at 100 requests per second. When more load is added, it doesn't make CPU utilization raise anymore. However, it does make your application respond more slowly. Then, you run the load test again, doubling the number of tasks but holding the rate at its previous peak value. If you find the average CPU utilization falls to about 40%, then average CPU utilization is a good candidate for a scaling metric. On the other hand, if CPU utilization remains at 80% after increasing the number of tasks, then average CPU utilization isn't a good scaling metric. In that case, more research is needed to find a suitable metric.

Common application models and scaling properties

Software of all kinds are run on AWS. Many workloads are homegrown, whereas others are based on popular open-source software. Regardless of where they originate, we have observed some common design patterns for services. How to scale effectively depends in large part on the pattern.

The efficient CPU-bound server

The efficient CPU-bound server utilizes almost no resources other than CPU and network throughput. Each request can be handled by the application alone. Requests don't depend on other services such as databases. The application can handle hundreds of thousands of concurrent requests, and can efficiently utilize multiple CPUs to do so. Each request is either serviced by a dedicated thread with low memory overhead, or there's an asynchronous event loop that runs on each CPU that services requests. Each replica of the application is equally capable of handling a request. The only resource that might be depleted before CPU is network bandwidth. In CPU bound-services, memory utilization, even at peak throughput, is a fraction of the resources available.

This type of application is suitable for CPU-based auto scaling. The application enjoys maximum flexibility in terms of scaling. It can be scaled vertically by providing larger Amazon EC2 instances or Fargate vCPUs to it. And, it can also be scaled horizontally by adding more replicas. Adding more replicas, or doubling the instance size, cuts the average CPU utilization relative to capacity by half.

If you're using Amazon EC2 capacity for this application, consider placing it on compute-optimized instances such as the `c5` or `c6g` family.

The efficient memory-bound server

The efficient memory-bound server allocates a significant amount of memory per request. At maximum concurrency, but not necessarily throughput, memory is depleted before the CPU resources are depleted. Memory associated with a request is freed when the request ends. Additional requests can be accepted as long as there is available memory.

This type of application is suitable for memory-based auto scaling. The application enjoys maximum flexibility in terms of scaling. It can be scaled both vertically by providing larger Amazon EC2 or Fargate memory resources to it. And, it can also be scaled horizontally by adding more replicas. Adding more replicas, or doubling the instance size, can cut the average memory utilization relative to capacity by half.

If you're using Amazon EC2 capacity for this application, consider placing it on memory-optimized instances such as the `r5` or `r6g` family.

Some memory-bound applications don't free the memory that's associated with a request when it ends, so that a reduction in concurrency doesn't result in a reduction in the memory used. For this, we don't recommend that you use memory-based scaling.

The worker-based server

The worker-based server processes one request for each individual worker thread one after another. The worker threads can be lightweight threads, such as POSIX threads. They can also be heavier-weight threads, such as UNIX processes. No matter which thread they are, there's always a maximum concurrency that the application can support. Usually the concurrency limit is set proportionally to the memory resources that are available. If the concurrency limit is reached, additional requests are placed into a backlog queue. If the backlog queue overflows, additional incoming requests are immediately rejected. Common applications that fit this pattern include Apache web server and Gunicorn.

Request concurrency is usually the best metric for scaling this application. Because there's a concurrency limit for each replica, it's important to scale out before the average limit is reached.

The best way to obtain request concurrency metrics is to have your application report them to CloudWatch. Each replica of your application can publish the number of concurrent requests as a custom metric at a high frequency. We recommend that the frequency is set to be at least once every minute. After several reports are collected, you can use the average concurrency as a scaling metric. This metric is calculated by taking the total concurrency and dividing it by the number of replicas. For example, if total concurrency is 1000 and the number of replicas is 10, then the average concurrency is 100.

If your application is behind an Application Load Balancer, you can also use the `ActiveConnectionCount` metric for the load balancer as a factor in the scaling metric. The `ActiveConnectionCount` metric must be divided by the number of replicas to obtain an average value. The average value must be used for scaling, as opposed to the raw count value.

For this design to work best, the standard deviation of response latency should be small at low request rates. We recommend that, during periods of low demand, most requests are answered within a short time, and there isn't a lot of requests that take significantly longer than average time to respond. The average response time should be close to the 95th percentile response time. Otherwise, queue overflows might occur as result. This leads to errors. We recommend that you provide additional replicas where necessary to mitigate the risk of overflow.

The waiting server

The waiting server does some processing for each request, but it is highly dependent on one or more downstream services to function. Container applications often make heavy use of downstream services like databases and other API services. It can take some time for these services to respond, particularly

in high capacity or high concurrency scenarios. This is because these applications tend to use few CPU resources and their maximum concurrency in terms of available memory.

The waiting service is suitable either in the memory-bound server pattern or the worker-based server pattern, depending on how the application is designed. If the application's concurrency is limited only by memory, then average memory utilization should be used as a scaling metric. If the application's concurrency is based on a worker limit, then average concurrency should be used as a scaling metric.

The Java-based server

If your Java-based server is CPU-bound and scales proportionally to CPU resources, then it might be suitable for the efficient CPU-bound server pattern. If that is the case, average CPU utilization might be appropriate as a scaling metric. However, many Java applications aren't CPU-bound, making them challenging to scale.

For the best performance, we recommend that you allocate as much memory as possible to the Java Virtual Machine (JVM) heap. Recent versions of the JVM, including Java 8 update 191 or later, automatically set the heap size as large as possible to fit within the container. This means that, in Java, memory utilization is rarely proportional to application utilization. As the request rate and concurrency increases, memory utilization remains constant. Because of this, we don't recommend scaling Java-based servers based on memory utilization. Instead, we typically recommend scaling on CPU utilization.

In some cases, Java-based servers encounter heap exhaustion before exhausting CPU. If your application is prone to heap exhaustion at high concurrency, then average connections are the best scaling metric. If your application is prone to heap exhaustion at high throughput, then average request rate is the best scaling metric.

Servers that use other garbage-collected runtimes

Many server applications are based on runtimes that perform garbage collection such as .NET and Ruby. These server applications might fit into one of the patterns described earlier. However, as with Java, we don't recommend scaling these applications based on memory, because their observed average memory utilization is often uncorrelated with throughput or concurrency.

For these applications, we recommend that you scale on CPU utilization if the application is CPU bound. Otherwise, we recommend that you scale on average throughput or average concurrency, based on your load testing results.

Job processors

Many workloads involve asynchronous job processing. They include applications that don't receive requests in real time, but instead subscribe to a work queue to receive jobs. For these types of applications, the proper scaling metric is almost always queue depth. Queue growth is an indication that pending work outstrips processing capacity, whereas an empty queue indicates that there's more capacity than work to do.

AWS messaging services, such as Amazon SQS and Amazon Kinesis Data Streams, provide CloudWatch metrics that can be used for scaling. For Amazon SQS, `ApproximateNumberOfMessagesVisible` is the best metric. For Kinesis Data Streams, consider using the `MillisBehindLatest` metric, published by the Kinesis Client Library (KCL). This metric should be averaged across all consumers before using it for scaling.

Capacity and availability

Application availability is crucial for providing an error-free experience and for minimizing application latency. Availability depends on having resources that are accessible and have enough capacity to meet

demand. AWS provides several mechanisms to manage availability. For applications hosted on Amazon ECS, these include autoscaling and Availability Zones (AZs). Autoscaling manages the number of tasks or instances based on metrics you define, while Availability Zones allow you to host your application in isolated but geographically-close locations.

As with task sizes, capacity and availability present certain trade-offs you must consider. Ideally, capacity would be perfectly aligned with demand. There would always be just enough capacity to serve requests and process jobs to meet Service Level Objectives (SLOs) including a low latency and error rate. Capacity would never be too high, leading to excessive cost; nor would it never be too low, leading to high latency and error rates.

Autoscaling is a latent process. First, real-time metrics must be delivered to CloudWatch. Then, they need to be aggregated for analysis, which can take up to several minutes depending on the granularity of the metric. CloudWatch compares the metrics against alarm thresholds to identify a shortage or excess of resources. To prevent instability, configure alarms to require the set threshold be crossed for a few minutes before the alarm goes off. It also takes time to provision new tasks and to terminate tasks that are no longer needed.

Because of these potential delays in the system described, it's important that you maintain some headroom by over-provisioning. Doing this can help accommodate short-term bursts in demand. This also helps your application to service additional requests without reaching saturation. As a good practice, you can set your scaling target between 60-80% of utilization. This helps your application better handle bursts of extra demand while additional capacity is still in the process of being provisioned.

Another reason we recommend that you over-provision is so that you can quickly respond to Availability Zone failures. AWS recommends that production workloads be served from multiple Availability Zones. This is because, if an Availability Zone failure occurs, your tasks that are running in the remaining Availability Zones can still serve the demand. If your application runs in two Availability Zones, you need to double your normal task count. This is so that you can provide immediate capacity during any potential failure. If your application runs in three Availability Zones, we recommend that you run 1.5 times your normal task count. That is, run three tasks for every two that are needed for ordinary serving.

Maximizing scaling speed

Autoscaling is a reactive process that takes time to take effect. However, there are some ways to help minimize the time that's needed to scale out.

Minimize image size. Larger images take longer to download from an image repository and unpack. Therefore, keeping image sizes smaller reduces the amount of time that's needed for a container to start. To reduce the image size, you can follow these specific recommendations:

- If you can build a static binary or use Golang, build your image `FROM` scratch and include only your binary application in the resulting image.
- Use minimized base images from upstream distro vendors, such as Amazon Linux or Ubuntu.
- Don’t include any build artifacts in your final image. Using multi-stage builds can help with this.
- Compact `RUN` stages wherever possible. Each `RUN` stage creates a new image layer, leading to an additional round trip to download the layer. A single `RUN` stage that has multiple commands joined by `&&` has fewer layers than one with multiple `RUN` stages.
- If you want to include data, such as ML inference data, in your final image, include only the data that's needed to start up and begin serving traffic. If you fetch data on demand from Amazon S3 or other storage without impacting service, then store your data in those places instead.

Keep your images close. The higher the network latency, the longer it takes to download the image. Host your images in a repository in the same AWS Region that your workload is in. Amazon ECR is a high-performance image repository that's available in every Region that Amazon ECS is available in. Avoid traversing the Internet or a VPN link to download container images. Hosting your images in the same

Region improves overall reliability. It mitigating the risk of network connectivity issues and availability issues in a different Region. Alternatively, you can also implement Amazon ECR cross-region replication to help with this.

Reduce load balancer health check thresholds. Load balancers perform health checks before sending traffic to your application. The default health check configuration for a target group can take 90 seconds or longer. During this, it checks the health status and receiving requests. Lowering the health check interval and threshold count can make your application accept traffic quicker and reduce load on other tasks.

Consider cold-start performance. Some application use runtimes such as Java perform Just-In-Time (JIT) compilation. The compilation process at least as it starts can show application performance. A workaround is to rewrite the latency-critical parts of your workload in languages that don't impose a cold-start performance penalty.

Use step scaling, not target-tracking scaling policies. You have several Application Auto Scaling options for Amazon ECS tasks. Target tracking is the easiest mode to use. With it, all you need to do is set a target value for a metric, such as CPU average utilization. Then, the auto scaler automatically manages the number of tasks that are needed to attain that value. However, we recommend that you use step scaling instead so that you can more quickly react to changes in demand. With step scaling, you define the specific thresholds for your scaling metrics, and how many tasks to add or remove when the thresholds are crossed. And, more importantly, you can react very quickly to changes in demand by minimizing the amount of time a threshold alarm is in breach. For more information, see Service Auto Scaling in the *Amazon Elastic Container Service Developer Guide*.

If you're using Amazon EC2 instances to provide cluster capacity, consider the following recommendations:

Use larger Amazon EC2 instances and faster Amazon EBS volumes. You can improve image download and preparation speeds by using a larger Amazon EC2 instance and faster Amazon EBS volume. Within a given Amazon EC2 instance family, the network and Amazon EBS maximum throughput increases as the instance size increases (for example, from `m5.xlarge` to `m5.2xlarge`). Additionally, you can also customize Amazon EBS volumes to increase their throughput and IOPS. For example, if you're using `gp2` volumes, use larger volumes that offer more baseline throughput. If you're using `gp3` volumes, specify throughput and IOPS when you create the volume.

Use bridge network mode for tasks running on Amazon EC2 instances. Tasks that use `bridge` network mode on Amazon EC2 start faster than tasks that use the `awsvpc` network mode. When `awsvpc` network mode is used, Amazon ECS attaches an elastic network interface (ENI) to the instance before launching the task. This introduces additional latency. There are several tradeoffs for using bridge networking though. These tasks don't get their own security group, and there are some implications for load balancing. For more information, see Load balancer target groups in the *Elastic Load Balancing User Guide*.

Handling demand shocks

Some applications experience sudden large shocks in demand. This happens for a variety of reasons: a news event, big sale, media event, or some other event that goes viral and causes traffic to quickly and significantly increase in a very short span of time. If unplanned, this can cause demand to quickly outstrip available resources.

The best way to handle demand shocks is to anticipate them and plan accordingly. Because autoscaling can take time, we recommend that you scale out your application before the demand shock begins. For the best results, we recommend having a business plan that involves tight collaboration between teams that use a shared calender. The team that's planning the event should work closely with the team in charge of the application in advance. This gives that team enough time to have a clear scheduling plan. They can schedule capacity to scale out before the event and to scale in after the event. For more information, see Scheduled scaling in the *Application Auto Scaling User Guide*.

If you have an Enterprise Support plan, be sure also to work with your Technical Account Manager (TAM). Your TAM can verify your service quotas and ensure that any necessary quotas are raised before the event begins. This way, you don't accidentally hit any service quotas. They can also help you by prewarming services such as load balancers to make sure your event goes smoothly.

Handling unscheduled demand shocks is a more difficult problem. Unscheduled shocks, if large enough in amplitude, can quickly cause demand to outstrip capacity. It can also outpace the ability for autoscaling to react. The best way to prepare for unscheduled shocks is to over-provision resources. You must have enough resources to handle maximum anticipated traffic demand at any time.

Maintaining maximum capacity in anticipation of unscheduled demand shocks can be costly. To mitigate the cost impact, find a leading indicator metric or event that predicts a large demand shock is imminent. If the metric or event reliably provides significant advance notice, begin the scale-out process immediately when the event occurs or when the metric crosses the specific threshold that you set.

If your application is prone to sudden unscheduled demand shocks, consider adding a high-performance mode to your application that sacrifices non-critical functionality but retains crucial functionality for a customer. For example, assume that your application can switch from generating expensive customized responses to serving a static response page. In this scenario, you can increase throughput significantly without scaling the application at all.

Last, you can consider breaking apart monolithic services to better deal with demand shocks. If your application is a monolithic service that's expensive to run and slow to scale, you might be able to extract or rewrite performance-critical pieces and run them as separate services. These new services then can be scaled independently from less-critical components. Having the flexibility to scale out performance-critical functionality separately from other parts of your application can both reduce the time it takes to add capacity and help conserve costs.

Cluster capacity

Earlier in this topic, we discussed how to scale out the replica account for your by using scaling metrics. Your tasks also need to run on resources, including CPU and memory resources. This again relates back the topic of capacity. In Amazon ECS, capacity is provided through two primary providers: AWS Fargate and Amazon EC2.

You can provide capacity to an Amazon ECS cluster in several ways. For example, you can launch Amazon EC2 instances and register them with the cluster at start-up using the Amazon ECS container agent. However, this method can be challenging because you need to manage scaling on your own. Therefore, we recommend that you use Amazon ECS capacity providers. They manage resource scaling for you. There are three kinds of capacity providers: Amazon EC2, Fargate, and Fargate Spot.

The Fargate and Fargate Spot capacity providers handle the lifecycle of Fargate tasks for you. Fargate provides on-demand capacity, and Fargate Spot provides Spot capacity. When a task is launched, ECS provisions a Fargate resource for you. This Fargate resource comes with the memory and CPU units that directly correspond to the task-level limits that you declared in your task definition. Each task receives its own Fargate resource, making a 1:1 relationship between the task and compute resources.

Tasks that run on Fargate Spot are subject to interruption. Interruptions come after a two-minute warning. These occur during periods of heavy demand. Fargate Spot works best for interruption-tolerant workloads such as batch jobs, development or staging environments. They're also suitable for any other scenario where high availability and low latency isn't a requirement.

You can run Fargate Spot tasks alongside Fargate on-demand tasks. By using them together, you receive provision “burst” capacity at a lower cost.

ECS can also manage the Amazon EC2 instance capacity for your tasks. Each Amazon EC2 Capacity Provider is associated with an Amazon EC2 Auto Scaling Group that you specify. When you use the

Amazon EC2 Capacity Provider, ECS cluster auto scaling maintains the size of the Amazon EC2 Auto Scaling Group to ensure all scheduled tasks can be placed.

Cluster capacity best practices

Add headroom to your service, not to the Capacity Provider. Amazon EC2 Capacity Providers offer a target capacity value. If you set the value lower than 100%, ECS provisions more Amazon EC2 instances than necessary to accommodate your tasks. Having several Amazon EC2 instances ready to accept tasks can be helpful. However, when you use Amazon Virtual Private Cloud, launching new tasks require additional time to download the image and attach a network interface. This added latency could be detrimental to your bottom line.

Therefore, we recommend that you do the following. Instead of reducing the target capacity of your Capacity Provider, increase the number of replicas in your service by modifying the target tracking scaling metric or the step scaling thresholds of the service. For more information about the related scaling policies, see Target Tracking Scaling Policies or Step Scaling Policies in the *Amazon Elastic Container Service Developer Guide*. The Amazon EC2 Capacity Provider provisions the capacity that's needed for additional tasks by adding additional instances to the Auto Scaling Group. This helps to ensure that both the compute and application resources are available when you need them. For example, it can help by doubling the number of tasks in an ECS Service to accommodate an immediate 100% burst in demand.

Choosing Fargate task sizes

If you run your tasks on AWS Fargate, you must declare the task CPU and memory limits in your task definition. ECS uses these limits to determine the Fargate instance type to run your task on. The limits that you determine must be greater than or equal to any reservations that you declared. In most cases, you can set them to the sum of the reservations of each of the container that's declared in your task definition. Then, also round the number up to the nearest Fargate instance size. For more information about the available sizes, see Task CPU and memory in the *Amazon Elastic Container Service Developer Guide*.

Choosing the Amazon EC2 instance type

If you use Amazon EC2 to provide capacity for your ECS cluster, you can choose from a large selection of instance types. All Amazon EC2 instance types and families are compatible with ECS.

To determine which instance types you can use, start by eliminating the instance types or instance families that don't meet the specific requirements of your application. For example, if your application requires a GPU, you can exclude any instance types that don't have a GPU. However, you should also consider other requirements, too. For example, consider the CPU architecture, network throughput, and if instance storage is a requirement. Next, examine the amount of CPU and memory provided by each instance type. As a general rule, the CPU and memory must be large enough to hold at least one replica of the task that you want to run.

You can choose from the instance types that are compatible with your application. With larger instances, you can launch more tasks at the same time. And, with smaller instances, you can scale out in a more fine-grained way to save costs. You don't need to choose a single Amazon EC2 instance type that to fit all the applications in your cluster. Instead, you can create multiple Auto Scaling Groups,. Each group can have a different instance type. Then, you can create an Amazon EC2 Capacity Provider for each one of these groups. Last, in the Capacity Provider strategy of your service and task, you can select the Capacity Provider that best suits its needs.

Using Amazon EC2 Spot and FARGATE_SPOT

Spot capacity can provide significant cost savings over on-demand instances. Spot capacity is excess capacity that's priced significantly lower than on-demand or reserved capacity. Spot capacity is suitable for batch processing and machine-learning workloads, and development and staging environments. More generally, it's suitable for any workload that tolerates temporary downtime.

Understand that the following consequences because Spot capacity might not be available all the time.

- First, during periods of extremely high demand, Spot capacity might be unavailable. This can cause Fargate Spot task and Amazon EC2 Spot instance launches to be delayed. In these events, ECS services retry launching tasks, and Amazon EC2 Auto Scaling groups also retry launching instances, until the required capacity becomes available. Fargate and Amazon EC2 don't replace Spot capacity with on-demand capacity.
- Second, when the overall demand for capacity increases, Spot instances and tasks might be terminated with only a two-minute warning. After the warning is sent, tasks should begin an orderly shutdown if necessary before the instance is fully terminated. This helps minimize the possibility of errors. For more information about a graceful shutdown, see Graceful shutdowns with ECS .

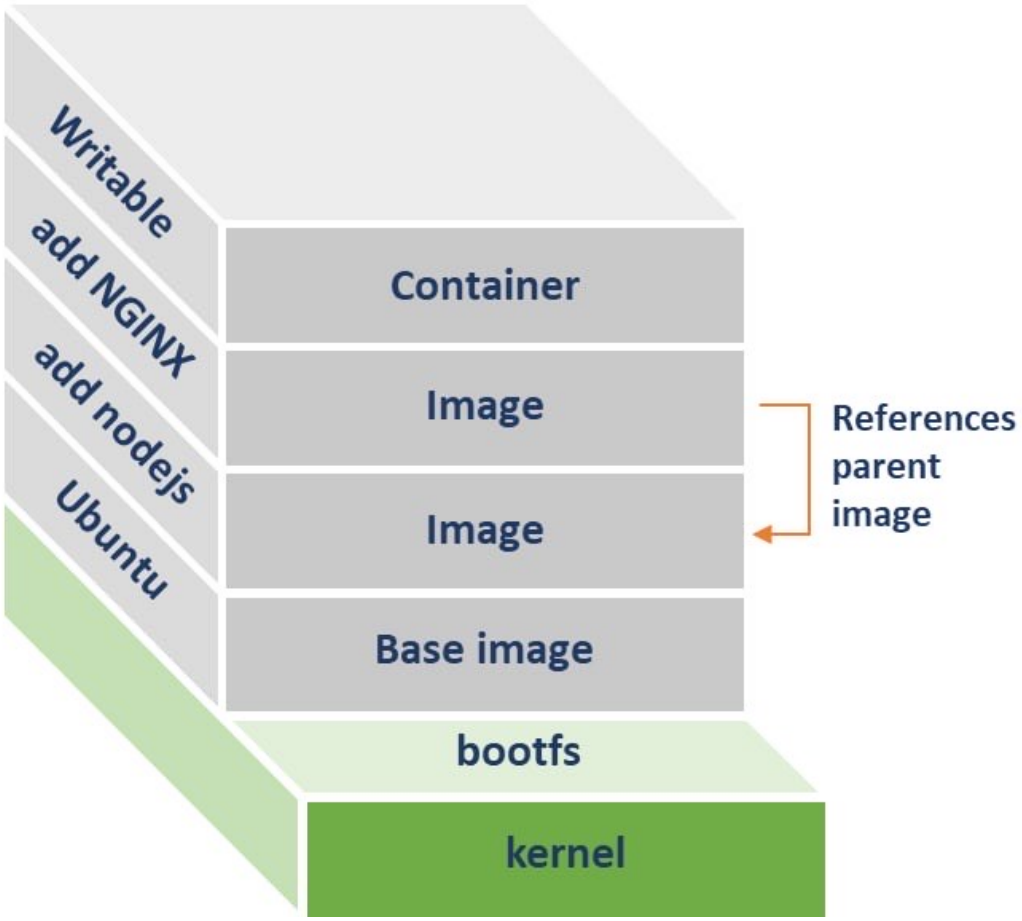
To help minimize Spot capacity shortages, consider the following recommendations:

- **Use multiple Regions and Availability Zones.** Spot capacity varies by Region and Availability Zone. You can improve Spot availability by running your workloads in multiple Regions and Availability Zones. If possible, specify subnets in all the Availability Zones in the Regions where you run your tasks and instances.
- **Use multiple Amazon EC2 instance types.** When you use Mixed Instance Policies with Amazon EC2 Auto Scaling, multiple instance types are launched into your Auto Scaling Group. This ensures that a request for Spot capacity can be fulfilled when needed. To maximize reliability and minimize complexity, use instance types with roughly the same amount of CPU and memory in your Mixed Instances Policy. These instances can be from a different generation, or variants of the same base instance type. Note that they might come with additional features that you might not require. An example of such a list could include m4.large, m5.large, m5a.large, m5d.large, m5n.large, m5dn.large, and m5ad.large. For more information, see Auto Scaling groups with multiple instance types and purchase options in the *Amazon EC2 Auto Scaling User Guide*.
- **Use the capacity-optimized Spot allocation strategy.** With Amazon EC2 Spot, you can choose between the capacity- and cost-optimized allocation strategies. If you choose the capacity-optimized strategy when launching a new instance, Amazon EC2 Spot selects the instance type with the greatest availability in the selected Availability Zone. This helps reduce the possibility that the instance is terminated soon after it launches.

Best Practices - Persistent storage

You can use Amazon ECS to run stateful containerized applications at scale by using AWS storage services, such as Amazon EFS, Amazon EBS, or FSx for Windows File Server, that provide data persistence to inherently ephemeral containers. The term *data persistence* means that the data itself outlasts the process that created it. Data persistence in AWS is achieved by coupling compute and storage services. Similar to Amazon EC2, you can also use Amazon ECS to decouple the lifecycle of your containerized applications from the data they consume and produce. Using AWS storage services, Amazon ECS tasks can persist data even after tasks terminate.

By default, containers don't persist the data they produce. When a container is terminated, the data that it wrote to its writable layer gets destroyed with the container. This makes containers suitable for stateless applications that don't need to store data locally. Containerized applications that require data persistence need a storage backend that isn't destroyed when the application's container terminates.



A container image is built off a series of layers. Each layer represents an instruction in the Dockerfile that the image was built from. Each layer is read-only, except for the container. That is, when you create a container, a writable layer is added over the underlying layers. Any files the container creates, deletes, or modifies are written to the writable layer. When the container terminates, the writable layer is also deleted simultaneously. A new container that uses the same image has its own writable layer. This layer doesn't include any changes. Therefore, a container's data must always be stored outside of the container writable layer.

With Amazon ECS, you can run stateful containers using volumes. Amazon ECS is integrated with Amazon EFS natively, and uses volumes that are integrated with Amazon EBS. For Windows containers, Amazon ECS integrates with FSx for Windows File Server to provide persistent storage.

Topics

Choosing the right storage type for your containers

Applications that are running in an Amazon ECS cluster can use a variety of AWS storage services and third-party products to provide persistent storage for stateful workloads. You should choose your storage backend for your containerized application based on the architecture and storage requirements of your application. For more information about AWS storage services, see Cloud Storage on AWS.

For Amazon ECS clusters that contain Linux instances or are used with Fargate, Amazon ECS integrates with Amazon EFS and Amazon EBS to provide container storage. The most distinctive difference between Amazon EFS and Amazon EBS is that you can simultaneously mount an Amazon EFS filesystem on thousands of Amazon ECS tasks. In contrast, Amazon EBS volumes don't support concurrent access. Given this, Amazon EFS is the recommended storage option for containerized applications that scale horizontally. This is because it supports concurrency. Amazon EFS stores your data redundantly across multiple Availability Zones and offers low latency access from Amazon ECS tasks, regardless of Availability Zone. Amazon EFS supports tasks that run on both Amazon EC2 and Fargate.

Suppose you have an application such as a transactional database that requires sub-millisecond latency and doesn't need a shared filesystem when it's scaled horizontally. For such an application, we recommend using Amazon EBS volumes for persistent storage. Currently, Amazon ECS supports Amazon EBS volumes for tasks hosted on Amazon EC2 only. Support for Amazon EBS volumes isn't available for tasks on Fargate. Before using Amazon EBS volumes with Amazon ECS tasks, you must first attach Amazon EBS volumes to container instances and manage volumes separately from the lifecycle of the task.

For clusters that contain Windows instances, FSx for Windows File Server provides persistent storage for containers. FSx for Windows File Server filesystems supports multi-AZ deployments. Through these deployments, you can share a filesystem with Amazon ECS tasks running across multiple Availability Zones.

You can also use Amazon EC2 instance storage for data persistence for Amazon ECS tasks that are hosted on Amazon EC2 using bind mounts or Docker volumes. When using bind mounts or Docker volumes, containers store data on the container instance file system. One limitation of using a host filesystem for container storage is that the data is only available on a single container instance at a time. This means that containers can only run on the host where the data resides. Therefore, using host storage is only recommended in scenarios where data replication is handled at the application level.

Amazon EFS volumes

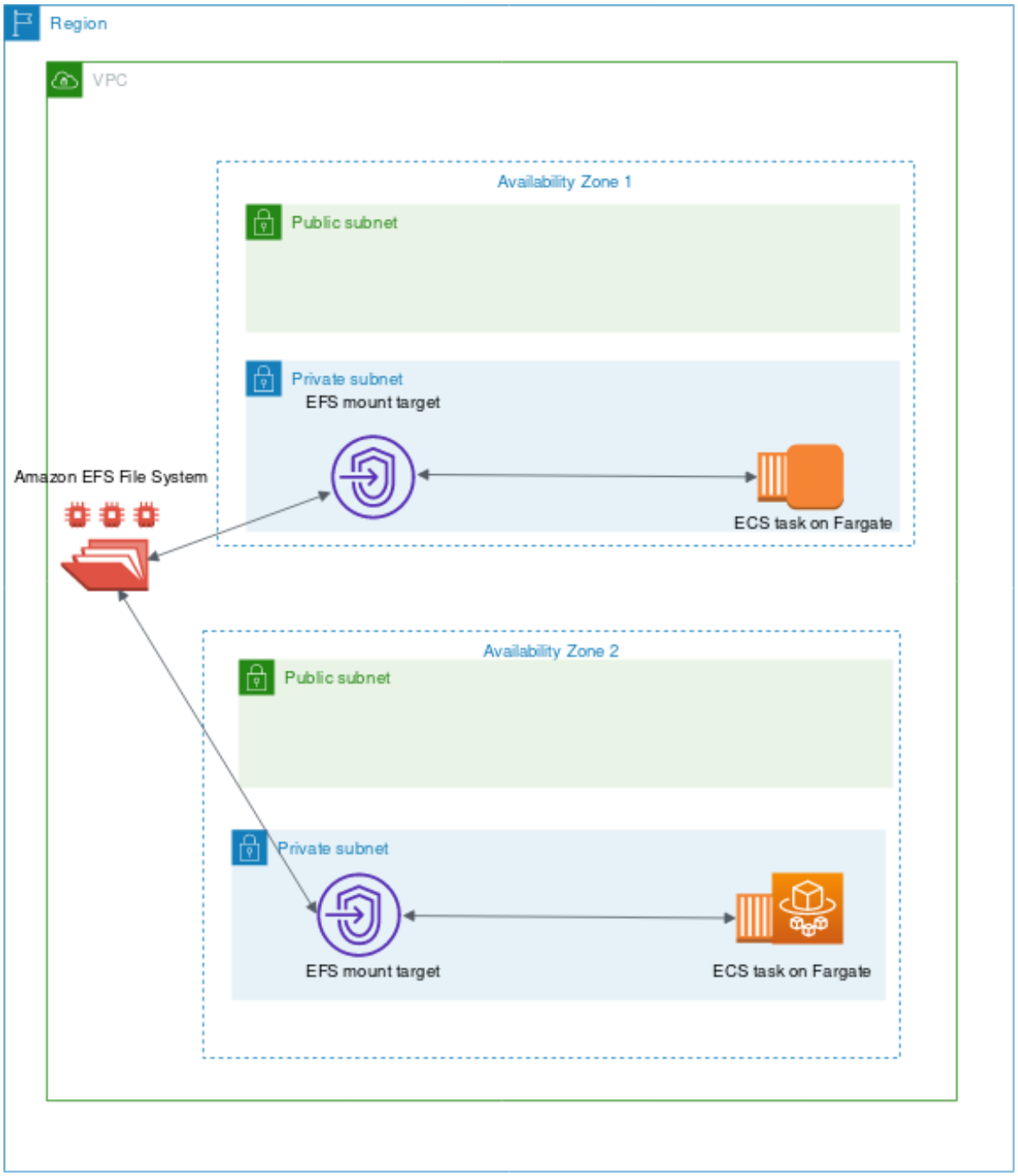
Amazon Elastic File System (Amazon EFS) provides a simple, scalable, fully managed elastic NFS file system. It's built to be able to scale on demand to petabytes without disrupting applications. It can scale in or out as you add and remove files.

You can run your stateful applications in Amazon ECS by using Amazon EFS volumes to provide persistent storage. Amazon ECS tasks that run on Amazon EC2 instances or on Fargate using platform version `1.4.0` and later can mount an existing Amazon EFS file system. Given that multiple containers can mount and access an Amazon EFS file system simultaneously, your tasks have access to the same data set regardless of where they're hosted.

To mount an Amazon EFS file system in your container, you can reference the Amazon EFS file system and container mount point in your Amazon ECS task definition. The following is a snippet of a task definition that uses Amazon EFS for container storage.

```
...
"containerDefinitions":[
    {
        "mountPoints": [
            {
                "containerPath": "/opt/my-app",
                 "sourceVolume": "Shared-EFS-Volume"
            }
    }
  ]
...
"volumes": [
    {
      "efsVolumeConfiguration": {
        "fileSystemId": "fs-1234",
        "transitEncryption": "DISABLED",
        "rootDirectory": ""
      },
      "name": "Shared-EFS-Volume"
    }
  ]
```

Amazon EFS stores data redundantly across multiple Availability Zones within a single Region. An Amazon ECS task mounts the Amazon EFS file system by using an Amazon EFS mount target in its Availability Zone. An Amazon ECS task can only mount an Amazon EFS file system if the Amazon EFS filesystem has a mount target in the Availability Zone the task runs in. Therefore, a best practice is to create Amazon EFS mount targets in all the Availability Zones that you plan to host Amazon ECS tasks in.



For more information, see Amazon EFS volumes in the *Amazon Elastic Container Service Developer Guide*.

Security and access controls

Amazon EFS offers access control features that you can use to ensure that the data stored in an Amazon EFS file system is secure and accessible only from applications that need it. You can secure data by enabling encryption at rest and in-transit. For more information, see Data encryption in Amazon EFS in the *Amazon Elastic File System User Guide*.

In addition to data encryption, you can also use Amazon EFS to restrict access to a file system. There are three ways to implement access control in EFS.

- **Security groups**—With Amazon EFS mount targets, you can configure a security group that's used to permit and deny network traffic. You can configure the security group attached to Amazon EFS to permit NFS traffic (port 2049) from the security group that's attached to your Amazon ECS instances or, when using the `awsvpc` network mode, the Amazon ECS task.
- **IAM**—You can restrict access to an Amazon EFS filesystem using IAM. When configured, Amazon ECS tasks require an IAM role for file system access to mount an EFS file system. For more information, see Using IAM to control file system data access in the *Amazon Elastic File System User Guide*.

  IAM policies can also enforce predefined conditions such as requiring a client to use TLS when connecting to an Amazon EFS file system. For more information, see Amazon EFS condition keys for clients in the *Amazon Elastic File System User Guide*.
- **Amazon EFS access points**—Amazon EFS access points are application-specific entry points into an Amazon EFS file system. You can use access points to enforce a user identity, including the user's POSIX groups, for all file system requests that are made through the access point. Access points can also enforce a different root directory for the file system. This is so that clients can only access data in the specified directory or its sub-directories.

Consider implementing all three access controls on an Amazon EFS file system for maximum security. For example, you can configure the security group attached to an Amazon EFS mount point to only permit ingress NFS traffic from a security group that's associated with your container instance or Amazon ECS task. Additionally, you can configure Amazon EFS to require an IAM role to access the file system, even if the connection originates from a permitted security group. Last, you can use Amazon EFS access points to enforce POSIX user permissions and specify root directories for applications.

The following task definition snippet shows how to mount an Amazon EFS file system using an access point.

```
"volumes": [
    {
      "efsVolumeConfiguration": {
        "fileSystemId": "fs-1234",
        "authorizationConfig": {
          "acessPointId": "fsap-1234",
          "iam": "ENABLED"
        },
        "transitEncryption": "ENABLED",
        "rootDirectory": ""
      },
      "name": "my-filesystem"
    }
]
```

Performance

Amazon EFS offers two performance modes: General Purpose and Max I/O. General Purpose is suitable for latency-sensitive applications such as content management systems and CI/CD tools. In contrast, Max I/O file systems are suitable for workloads such as data analytics, media processing, and machine learning. These workloads need to perform parallel operations from hundreds or even thousands of containers and require the highest possible aggregate throughput and IOPS. For more information, see Amazon EFS performance modes in the *Amazon Elastic File System User Guide*.

Some latency sensitive workloads require both the higher I/O levels that are provided by Max I/O performance mode and the lower latency that are provided by General Purpose performance mode. For this type of workload, we recommend creating multiple General Purpose performance mode file

systems. That way, you can spread your application workload across all these file systems, as long as the workload and applications can support it.

Throughput

All Amazon EFS file systems have an associated metered throughput that's determined by either the amount of provisioned throughput for file systems using *Provisioned Throughput* or the amount of data stored in the EFS Standard or One Zone storage class for file systems using *Bursting Throughput*. For more information, see Understanding metered throughput in the *Amazon Elastic File System User Guide*.

The default throughput mode for Amazon EFS file systems is bursting mode. With bursting mode, the throughput that's available to a file system scales in or out as a file system grows. Because file-based workloads typically spike, requiring high levels of throughput for periods of time and lower levels of throughput the rest of the time, Amazon EFS is designed to burst to allow high throughput levels for periods of time. Additionally, because many workloads are read-heavy, read operations are metered at a 1:3 ratio to other NFS operations (like write).

All Amazon EFS file systems deliver a consistent baseline performance of 50 MB/s for each TB of Amazon EFS Standard or Amazon EFS One Zone storage. All file systems (regardless of size) can burst to 100 MB/s. File systems with more than 1TB of EFS Standard or EFS One Zone storage can burst to 100 MB/s for each TB. Because read operations are metered at a 1:3 ratio, you can drive up to 300 MiBs/s for each TiB of read throughput. As you add data to your file system, the maximum throughput that's available to the file system scales linearly and automatically with your storage in the Amazon EFS Standard storage class. If you need more throughput than you can achieve with your amount of data stored, you can configure Provisioned Throughput to the specific amount your workload requires.

File system throughput is shared across all Amazon EC2 instances connected to a file system. For example, a 1TB file system that can burst to 100 MB/s of throughput can drive 100 MB/s from a single Amazon EC2 instance can each drive 10 MB/s. For more information, see Amazon EFS performance in the *Amazon Elastic File System User Guide*.

Cost optimization

Amazon EFS simplifies scaling storage for you. Amazon EFS file systems grow automatically as you add more data. Especially with Amazon EFS *Bursting Throughput* mode, throughput on Amazon EFS scales as the size of your file system in the standard storage class grows. To improve the throughput without paying an additional cost for provisioned throughput on an EFS filesystem, you can share an Amazon EFS file system with multiple applications. Using Amazon EFS access points, you can implement storage isolation in shared Amazon EFS file systems. By doing so, even though the applications still share the same file system, they can't access data unless you authorize it.

As your data grows, Amazon EFS helps you automatically move infrequently accessed files to a lower storage class. The Amazon EFS Standard-Infrequent Access (IA) storage class reduces storage costs for files that aren't accessed every day. It does this without sacrificing the high availability, high durability, elasticity, and the POSIX file system access that Amazon EFS provides. For more information, see Amazon EFS storage classes in the *Amazon Elastic File System User Guide*.

Consider using Amazon EFS lifecycle policies to automatically save money by moving infrequently accessed files to Amazon EFS IA storage. For more information, see Amazon EFS lifecycle management in the *Amazon Elastic File System User Guide*.

When creating an Amazon EFS file system, you can choose if Amazon EFS replicates your data across multiple Availability Zones (Standard) or stores your data redundantly within a single Availability Zone. The Amazon EFS One Zone storage class can reduce storage costs by a significant margin compared to Amazon EFS Standard storage classes. Consider using Amazon EFS One Zone storage class for workloads that don't require multi-AZ resilience. You can further reduce the cost of Amazon EFS One Zone storage

by moving infrequently accessed files to Amazon EFS One Zone-Infrequent Access. For more information, see Amazon EFS Infrequent Access.

Data protection

Amazon EFS stores your data redundantly across multiple Availability Zones for file systems using Standard storage classes. If you select Amazon EFS One Zone storage classes, your data is redundantly stored within a single Availability Zone. Additionally, Amazon EFS is designed to provide 99.999999999% (11 9's) of durability over a given year.

As with any environment, it's a best practice to have a backup and to build safeguards against accidental deletion. For Amazon EFS data, that best practice includes a functioning, regularly tested backup using AWS Backup. File systems using Amazon EFS One Zone storage classes are configured to automatically back up files by default at file system creation unless you choose to disable this functionality. For more information, see Data protection for Amazon EFS in the *Amazon Elastic File System User Guide*.

Use cases

Amazon EFS provides parallel shared access that automatically grows and shrinks as files are added and removed. As a result, Amazon EFS is suitable for any application that requires a storage with functionalities like low latency, high throughput, and read-after-write consistency. Amazon EFS is an ideal storage backend for applications that scale horizontally and require a shared file system. Workloads such as data analytics, media processing, content management, and web serving are some of the common Amazon EFS use cases.

One use case where Amazon EFS might not be suitable is for applications that require sub-millisecond latency. This is generally a requirement for transactional database systems. We recommend running storage performance tests to determine the impact of using Amazon EFS for latency sensitive applications. If application performance degrades when using Amazon EFS, consider Amazon EBS io2 Block Express, which provides sub-millisecond, low-variance I/O latency on Nitro instances. For more information, see Amazon EBS volume types in the *Amazon EC2 User Guide for Linux Instances*.

Some applications fail if their underlying storage is changed unexpectedly. Therefore, Amazon EFS isn't the best choice for these applications. Rather, you might prefer to use a storage system that doesn't allow concurrent access from multiple places.

Docker volumes

Docker volumes are a feature of the Docker container runtime that allow containers to persist data by mounting a directory from the filesystem of the host. Docker volume drivers (also referred to as plugins) are used to integrate container volumes with external storage systems, such as Amazon EBS. Docker volumes are only supported when hosting Amazon ECS tasks on Amazon EC2 instances.

Amazon ECS tasks can use Docker volumes to persist data using Amazon EBS volumes. This is done by attaching an Amazon EBS volume to an Amazon EC2 instance and then mounting the volume in a task using Docker volumes. A Docker volume can be shared among multiple Amazon ECS tasks on the host.

The limitation of Docker volumes is that the file system the task uses is tied to the specific Amazon EC2 instance. If the instance stops for any reason and the task gets placed on another instance, the data is lost. You can assign tasks to instances to ensure the associated EBS volumes are always available to tasks.

For more information, see Docker volumes in the *Amazon Elastic Container Service Developer Guide*.

Amazon EBS volume lifecycle

There are two key usage patterns with container storage and Amazon EBS. The first is when an application needs to persist data and prevent data loss when its container terminates. An example of this type of application would be a transactional database like MySQL. When a MySQL task terminates, another task is expected to replace it. In this scenario, the lifecycle of the volume is separate from the lifecycle of the task. When using EBS to persist container data, it's a best practice to use task placement constraints to limit the placement of the task to a single host with the EBS volume attached.

The second is when the lifecycle of the volume is independent from the task lifecycle. This is especially useful for applications that require high-performing and low latency storage but don't need to persist data after the task terminates. For example, an ETL workload that process large volumes of data may require a high throughput storage. Amazon EBS is suitable for this type of workload as it provides high performance volumes that provideup to 256,000 IOPS. When the task terminates, the replacement replica can be safely placed on any Amazon EC2 hosts in the cluster. As long as the task has access to a storage backend that can meet its performance requirements, the task can perform its function. Therefore, no task placement constraints are necessary in this case.

If the Amazon EC2 instances in your cluster have multiple types of Amazon EBS volumes attached to them, you can use task placement constraints to ensure that tasks are placed on instances with an appropriate Amazon EBS volume attached. For example, suppose a cluster has some instances with a `gp2` volume, while others use `io1` volumes. You can attach custom attributes to instances with `io1` volumes and then use task placement constraints to ensure your I/O intensive tasks are always placed on container instances with `io1` volumes.

The following AWS CLI command is used to place attributes on an Amazon ECS container instance.

```
aws ecs put-attributes \
     --attributes name=EBS,value=io1,targetId=<your-container-instance-arn>
```

Amazon EBS data availability

Containers are typically short lived, frequently created, and terminated as applications scale in and out horizontally. As a best practice, you can run workloads in multiple Availability Zones to improve the availability of your applications. Amazon ECS provides a way for you to control task placement using task placement strategies and task placement constraints. When a workload persists its data using Amazon EBS volumes, its tasks need to be placed in the same Availability Zone as the Amazon EBS volume. We also recommend that you set a placement constraint that limits the Availability Zone a task can be placed in. This ensures that your tasks and their corresponding volumes are always located in the same Availability Zone.

When running standalone tasks, you can control which Availability Zone the task gets placed by setting placement constraints using the availability zone attribute.

```
attribute:ecs.availability-zone == us-east-1a
```

When running applications that would benefit from running in multiple Availability Zones, consider creating a different Amazon ECS service for each Availability Zone. This ensures that tasks that need an Amazon EBS volume are always placed in the same Availability Zone as the associated volume.

We recommend creating container instances in each Availability Zone, attaching Amazon EBS volumes using launch templates, and adding custom attributes to the instances to differentiate them from other container instances in the Amazon ECS cluster. When creating services, configure task placement constraints to ensure that Amazon ECS places tasks in the right Availability Zone and instance. For more information, see Task placement constraint examples in the *Amazon Elastic Container Service Developer Guide*.

Docker volume plugins

Docker plugins such as Portworx provide an abstraction between the Docker volume and the Amazon EBS volume. These plugins can dynamically create an Amazon EBS volume when your task that needs a volume starts. Portworx can also attach a volume to a new host when a container terminates, and its subsequent replica gets placed on a different container instance. It also replicates each container's volume data among Amazon ECS nodes and across Availability Zones. For more information, see Portworx.

FSx for Windows File Server

FSx for Windows File Server provides fully managed, highly reliable, and scalable file storage that is accessible over the industry-standard Server Message Block (SMB) protocol. It's built on Windows Server, delivering a wide range of administrative features such as user quotas, end-user file restore, and Microsoft Active Directory (AD) integration. It offers single-AZ and multi-AZ deployment options, fully managed backups, and encryption of data at rest and in transit.

Amazon ECS supports the use of FSx for Windows File Server in Amazon ECS Windows task definitions enabling persistent storage as a mount point through SMBv3 protocol using a SMB feature called GlobalMappings.

To setup the FSx for Windows File Server and Amazon ECS integration, the Windows container instance must be a domain member on an Active Directory Domain Service (AD DS), hosted by an AWS Directory Service for Microsoft Active Directory, on-premises Active Directory or self-hosted Active Directory on Amazon EC2. AWS Secrets Manager is used to store sensitive data like the username and password of an Active Directory credential which is used to map the share on the Windows container instance.

To use FSx for Windows File Server file system volumes for your containers, you must specify the volume and mount point configurations in your task definition. The following is a snippet of a task definition that uses FSx for Windows File Server for container storage.

```
{
 "containerDefinitions": [{
  "name": "container-using-fsx",
  "image": "iis:2",
  "entryPoint": [
   "powershell",
   "-command"
  ],
  "mountPoints": [{
   "sourceVolume": "myFsxVolume",
   "containerPath": "\\mount\\fsx",
   "readOnly": false
  }]
 }],
 "volumes": [{
  "fsxWindowsFileServerVolumeConfiguration": {
   "fileSystemId": "fs-ID",
   "authorizationConfig": {
    "domain": "ADDOMAIN.local",
    "credentialsParameter": "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SecretName"
   },
   "rootDirectory": "share"
  }
 }]
}
```

For more information, see Amazon FSx for Windows File Server volumes in the *Amazon Elastic Container Service Developer Guide*.

Security and access controls

FSx for Windows File Server offers the following access control features that you can use to ensure that the data stored in an FSx for Windows File Server file system is secure and accessible only from applications that need it.

Data encryption

FSx for Windows File Server supports two forms of encryption for file systems. They are encryption of data in transit and encryption at rest. Encryption of data in transit is supported on file shares that are mapped on a container instance that supports SMB protocol 3.0 or newer. Encryption of data at rest is automatically enabled when creating an Amazon FSx file system. Amazon FSx automatically encrypts data in transit using SMB encryption as you access your file system without the need for you to modify your applications. For more information, see Data encryption in Amazon FSx in the *FSx for Windows File Server User Guide*.

Folder level access control using Windows ACLs

The Windows Amazon EC2 instance access Amazon FSx file shares using Active Directory credentials. It uses standard Windows access control lists (ACLs) for fine-grained file- and folder-level access control. You can create multiple credentials, each one for a specific folder within the share which maps to a specific task.

In the following example, the task has access to the folder `App01` using a credential saved in Secrets Manager. Its Amazon Resource Name (ARN) is `1234`.

```
"rootDirectory": "\\path\\to\\my\\data\App01",
"credentialsParameter": "arn-1234",
"domain": "corp.fullyqualified.com",
```

In another example, a task has access to the folder `App02` using a credential saved in the Secrets Manager. Its ARN is `6789`.

```
"rootDirectory": "\\path\\to\\my\\data\App02",
"credentialsParameter": "arn-6789",
"domain": "corp.fullyqualified.com",
```

Use cases

Containers aren't designed to persist data. However, some containerized .NET applications might require local folders as persistent storage to save application outputs. FSx for Windows File Server offers a local folder in the container. This allows for multiple containers to read-write on the same file system that's backed by a SMB Share.

Best Practices - Speeding up deployments

You can choose rolling updates for your Amazon ECS service. Deployments might take longer than you expect, but you can modify a few options to speed up your deployments. For context, by choosing this deployment type, you tell the Amazon ECS service scheduler to replace any currently running tasks with new tasks whenever a new service deployment is started. The deployment configuration determines the specific number of tasks that Amazon ECS adds to or removes from the service during a rolling update. The following is an overview of the deployment process:

1. The scheduler starts your application.
2. The scheduler then decides if your application is ready for web traffic.
3. When you scale down or create a new version of the application, the scheduler decides whether your application is safe to stop. At the same time, it must maintain the availability of the application during the rolling deployment.

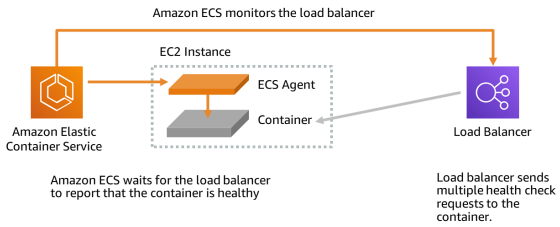
The strategy of maintaining task availability can cause deployments to take longer than you expect.

To speed up deployment times, modify the default load balancer, Amazon ECS agent, service and task definition options. The following sections detail how you can modify all of these to speed up deployments.

Topics

Load balancer health check parameters

The following diagram describes the load balancer health check process. The load balancer periodically sends health checks to the Amazon ECS container. The Amazon ECS agent monitors and waits for the load balancer to report on the container health. It does this before it considers the container to be in a healthy status.



There are two options in the health check that affect the deployment speed. One is for the Amazon ECS task definition. The other is for the load balancer.

- `HealthCheckInterval`: 30 seconds (default)

  For more information about the task definition health check interval parameter, see Health Check in the *Amazon Elastic Container Service Developer Guide*.
- `HealthyThresholdCount`: 5 (default)

By default, the load balancer requires five passing health checks before it reports that the target container is healthy. Each check is made 30 seconds apart. In total, each time takes two minutes and 30 seconds (`5*30/60`).

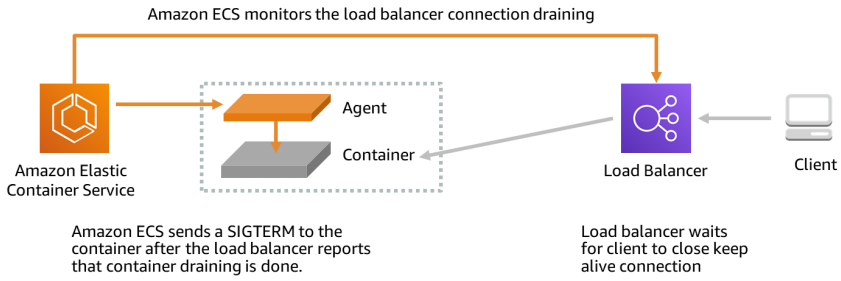
If your service starts up and stabilizes in under 10 seconds, set the options to the following values to have the load balancer wait a total of 10 seconds:

- `HealthCheckInterval`: 5
- `HealthyThresholdCount`: 2

Load balancer connection draining

To allow for optimization, clients maintain a keep alive connection to the container service. This is so that subsequent requests from that client can reuse the existing connection. When you want to stop traffic to a container, you notify the load balancer.

The following diagram describes the load balancer connection draining process. When you tell the load balancer to stop traffic to the container, it periodically checks to see if the client closed the keep alive connection. The Amazon ECS agent monitors the load balancer and waits the load balancer to report that the keep alive connection is closed.



The amount of time that the load balancer waits is the deregistraion delay. You can configure the following load balancer setting to speed up your deployments.

- `deregistration_delay.timeout_seconds`: 30 (default)

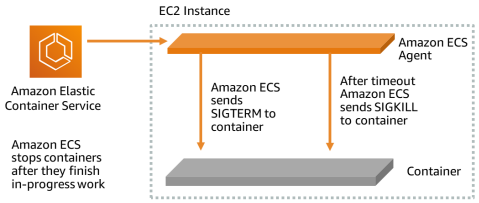
When you have a service with a response time that's under one second, set the parameter to the following value to have the load balancer only wait five seconds before it breaks the connection between the client and the backend service:

- `deregistration_delay.timeout_seconds`: 5

> **Note**
> Do not set the value to 5 seconds when you have a service with long-lived requests, such as slow file uploads or streaming connections.

SIGTERM responsiveness

The following diagram shows how Amazon ECS terminates a task. Amazon ECS first sends a SIGTERM signal to the task to notify the application needs to finish and shut down, and then the sends a SIGKILL message. When applications ignore the SIGTERM, the Amazon ECS service must wait to send the SIGKILL signal to terminate the process.



The amount of time that Amazon ECS waits is determined by the following Amazon ECS agent option:

- `ECS_CONTAINER_STOP_TIMEOUT`: 30 (default)

  For more information about the container agent parameter, see Container agent configuration in the *Amazon Elastic Container Service Developer Guide*.

To speed up the waiting period, set the Amazon ECS agent option to the following value:

> **Note**
> If your application takes more than one second, multiply the value by two and use that number as the value.

- `ECS_CONTAINER_STOP_TIMEOUT`: 2

In this case, the Amazon ECS waits two seconds for the container to shut down, and then Amazon ECS sends a SIGKILL message when the application didn't stop.

You can also modify the application code to trap the SIGTERM signal and react to it. The following is example in JavaScript:

```
process.on('SIGTERM', function() {
  server.close();
})
```

This code causes the HTTP server to stop listening for any new requests, finish answering any in-flight requests, and then the Node.js process terminates. This is because its event loop has nothing left to do. Given this, if it takes the process only 500 ms to finish its in-flight requests, it terminates early without having to wait out the stop timeout and get sent a SIGKILL.

Container image type

The time that it takes a container to start up varies, based on the underlying container image. For example, a fatter image (full versions of Debian, Ubuntu, and Amazon1/2) might take longer to start up because there are a more services that run in the containers compared to their respective slim versions (Debian-slim, Ubuntu-slim, and Amazon-slim) or smaller base images (Alpine).

Container image pull behavior

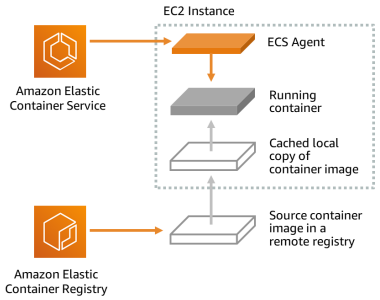
Container image pull behavior for Fargate launch types

Fargate does not cache images, and therefore the whole image is pulled from the registry when a task runs. The following are our recommendations for images used for Fargate tasks:

- Use a larger task size with additional vCPUs. The larger task size can help reduce the time that is required to extract the image when a task launches.
- Use a smaller base image.
- Have the repository that stores the image in the same Region as the task.

Container image pull behavior for Amazon EC2 launch types

When the Amazon ECS agent starts a task, it pulls the Docker image from its remote registry, and then caches a local copy. When you use a new image tag for each release of your application, this behavior is unnecessary.



The following Amazon ECS agent parameter determines the image pull behavior:

- `ECS_IMAGE_PULL_BEHAVIOR`: default

For more information about the container agent parameter, see Container agent configuration in the *Amazon Elastic Container Service Developer Guide*.

To speed up deployment, set the Amazon ECS agent parameter to one of the following values:

- `ECS_IMAGE_PULL_BEHAVIOR`: `once`

  The image is pulled remotely only if it wasn't pulled by a previous task on the same container instance or if the cached image was removed by the automated image cleanup process. Otherwise, the cached image on the instance is used. This ensures that no unnecessary image pulls are attempted.

- `ECS_IMAGE_PULL_BEHAVIOR`: `prefer-cached`

  The image is pulled remotely if there is no cached image. Otherwise, the cached image on the instance is used. Automated image cleanup is disabled for the container to ensure that the cached image isn't removed.

Setting the parameter to either of the preceding values can save time because the Amazon ECS agent uses the existing downloaded image. For larger Docker images, the download time might take 10-20 seconds to pull over the network.

Task deployment

To ensure that there's no application downtime, the deployment process is as follows:

1. Start the new application containers while keeping the exiting containers running.
2. Check that the new containers are healthy.
3. Stop the old containers.

Depending on your deployment configuration and the amount of free, unreserved space in your cluster it may take multiple rounds of this to complete replace all old tasks with new tasks.

There are two ECS service configuration options that you can use to modify the number:

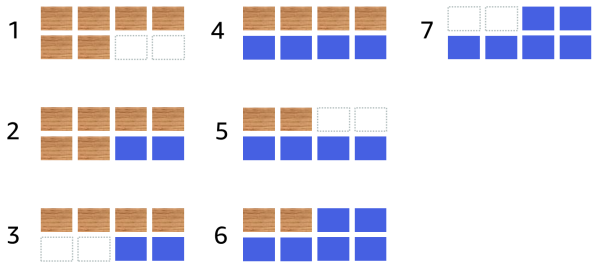
- `minimumHealthyPercent`: 100% (default)

  The lower limit on the number of tasks for your service that must remain in the `RUNNING` state during a deployment. This is represented as a percentage of the desiredCount. It's rounded up to the nearest integer. This parameter enables you to deploy without using additional cluster capacity.

- `maximumPercent`: 200% (default)

  The upper limit on the number of tasks for your service that are allowed in the `RUNNING` or `PENDING` state during a deployment. This is represented as a percentage of the desiredCount. It's rounded down to the nearest integer.

Consider the following service that has six tan tasks, deployed in a cluster that has room for eight tasks total. The default Amazon ECS service configuration options don't allow the deployment to go below 100% of the six desired tasks.



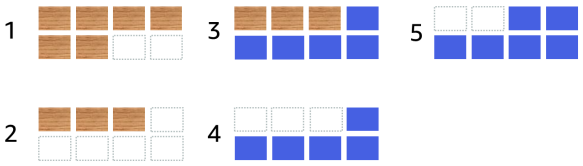
The deployment process is as follows:

1. The goal is to replace the tan tasks with the blue tasks.
2. The scheduler starts two new blue tasks because the default settings require that there are six running tasks.
3. The scheduler stops two of the tan tasks because there will be a total of six tasks (four tan and two blue).
4. The scheduler starts two additional blue tasks.
5. The scheduler shuts down two of the tan tasks.
6. The scheduler starts two additional blue tasks.
7. The scheduler shuts down the last two orange tasks.

In the above example, if you use the default values for the options, there is a 2.5 minute wait for each new task that starts. Additionally, the load balancer might have to wait 5 minutes for the old task to stop.

You can speed up the deployment by setting the `minimumHealthyPercent` value to 50%.

Consider the following service that has six tan tasks, deployed in a cluster that has room for eight tasks total.

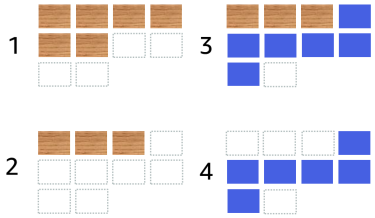


The deployment process is as follows:

1. The goal is to replace the tan tasks with the blue tasks.
2. The scheduler stops three of the tan tasks. There are still three tan tasks running which meets the `minimumHealthyPercent` value.
3. The scheduler starts five blue tasks.

4. The scheduler stops the remain three tan tasks.
5. The scheduler starts the final blue tasks.

You could also add additional free space so that you can run additional tasks.



The deployment process is as follows:

1. The goal is to replace the tan tasks with the blue tasks.
2. The scheduler stops three of the tan tasks
3. The scheduler starts six blue tasks
4. The scheduler stops the three tan tasks.

Use the following values for the ECS service configuration options when your tasks are idle for some time and don't have a high utilization rate.

- `minimumHealthyPercent`: 50%
- `maximumPercent`: 200%

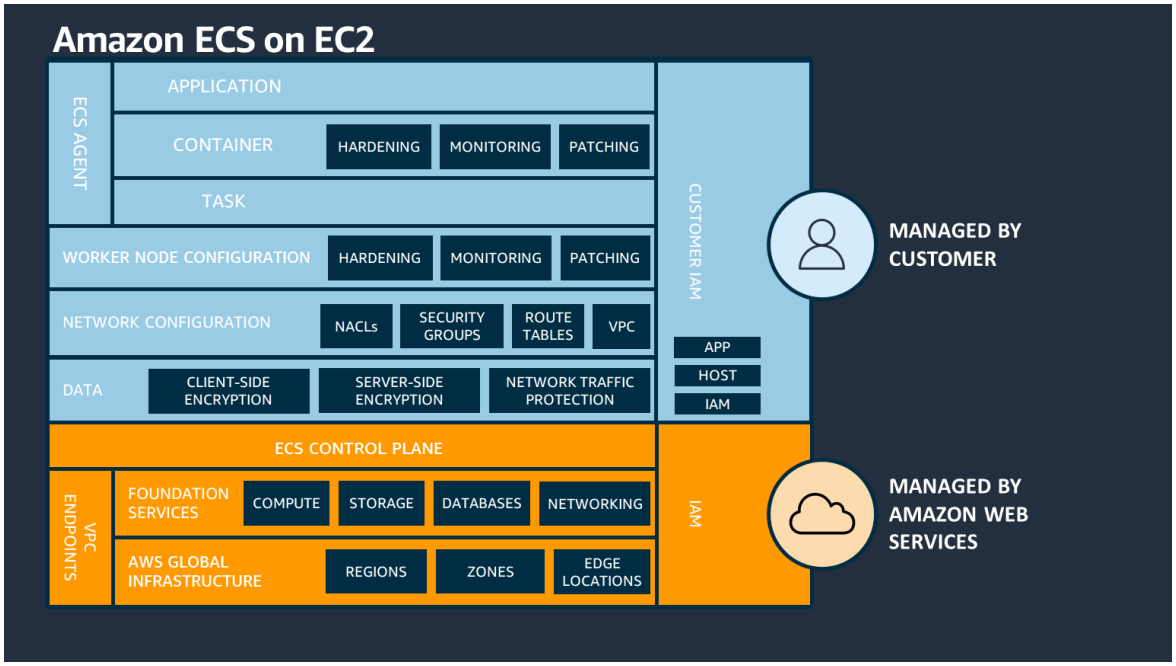
Best Practices - Security

This guide provides security and compliance recommendations for protecting your information, systems, and other assets that are reliant on Amazon ECS. It also introduces some risk assessments and mitigation strategies that you can use to have a better grip on the security controls that are built for Amazon ECS clusters and the workloads that they support. Each topic in this guide starts with a brief overview, followed by a list of recommendations and best practices that you can use to secure your Amazon ECS clusters.

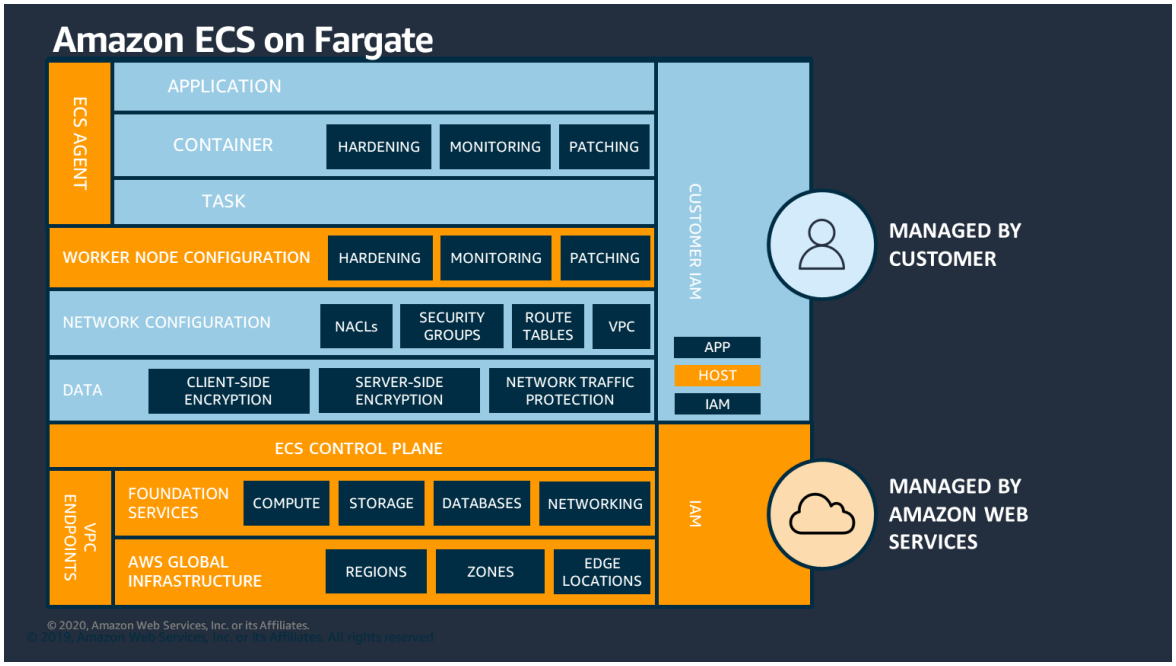
Topics

Shared responsibility model

The security and compliance of a managed service like Amazon ECS is a shared resposibility of both you and AWS. Generally speaking, AWS is responsible for security "of" the cloud whereas you, the customer, are responsible for security "in" the cloud. AWS is responsible for managing of the Amazon ECS control plane, including the infrastructure that's needed to deliver a secure and reliable service. And, you're largely responsible for the topics in this guide. This includes data, network, and runtime security, as well as logging and monitoring.



With respect to infrastructure security, AWS assumes more responsibility for AWS Fargate resources than it does for other self-managed instances. With Fargate, AWS manages the security of the underlying instance in the cloud and the runtime that's used to run your tasks. Fargate also automatically scales your infrastructure on your behalf.



Before extending your services to the cloud, you should understand what aspects of security and compliance that you're responsible for.

For more information about the shared responsibility model, see Shared Responsibility Model.

AWS Identity and Access Management

You can use AWS Identity and Access Management (IAM) to manage and control access to your AWS services and resources through rule-based policies for authentication and authorization purposes. More specifically, through this service, you control access to your AWS resources by using policies that are applied to IAM users, groups, or roles. Among these three, IAM users are accounts that can have access to your resources. And, an IAM role is a set of permissions that can be assumed by an authenticated identity, which isn't associated with a particular identity outside of IAM. For more information, see Policies and permissions in IAM?.

Managing access to Amazon ECS

You can control access to Amazon ECS by creating and applying IAM policies. These policies are composed of a set of actions that apply to a specific set of resources. The action of a policy defines the list of operations (such as Amazon ECS APIs) that are allowed or denied, whereas the resource controls what are the Amazon ECS objects that the action applies to. Conditions can be added to a policy to narrow its scope. For example, a policy can be written to only allow an action to be performed against tasks with a particular set of tags. For more information, see How Amazon ECS works with IAM in the *Amazon Elastic Container Service Developer Guide*.

Recommendations

We recommend that you do the following when setting up your IAM roles and policies.

Follow the policy of least privileged access

Create policies that are scoped to allow users to perform their prescribed jobs. For example, if a developer needs to periodically stop a task, create a policy that only permits that particular action. The following example only allows a user to stop a task that belongs to a particular `task_family` on a cluster with a specific Amazon Resource Name (ARN). Refering to an ARN in a condition is also an example of using resource-level permissions. You can use resource-level permissionsto specify the resource that you want an action to apply to.

> **Note**
> When referencing an ARN in a policy, use the new longer ARN format. For more information, see Amazon Resource Names (ARNs) and IDs in the *Amazon Elastic Container Service Developer Guide*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecs:StopTask"
      ],
      "Condition": {
        "ArnEquals": {
          "ecs:cluster": "arn:aws:ecs:<region>:<aws_account_id>:cluster/<cluster_name>"
        }
      },
      "Resource": [
        "arn:aws:ecs:<region>:<aws_account_id>:task-definition/<task_family>:*"
      ]
    }
  ]
```

```
}
```

Let the cluster resource serve as the administrative boundary

Policies that are too narrowly scoped can cause a proliferation of roles and increase administrative overhead. Rather than creating roles that are scoped to particular tasks or services only, create roles that are scoped to clusters and use the cluster as your primary administrative boundary.

Isolate end-users from the Amazon ECS API by creating automated pipelines

You can limit the actions that users can use by creating pipelines that automatically package and deploy applications onto Amazon ECS clusters. This effectively delegates the job of creating, updating, and deleting tasks to the pipeline. For more information, see Tutorial: Amazon ECS standard deployment with CodePipeline in the *AWS CodePipeline User Guide*.

Use policy conditions for an added layer of security

When you need an added layer of security, add a condition to your policy. This can be useful if you're performing a privileged operation or when you need to restrict the set of actions that can be performed against particular resources. The following example policy requires multi-factor authorization when deleting a cluster.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecs:DeleteCluster"
      ],
      "Condition": {
        "Bool": {
          "aws:MultiFactorAuthPresent": "true"
        }
      },
    "Resource": ["*"]
    }
  ]
}
```

Tags applied to services are propagated to all the tasks that are part of that service. Because of this, you can create roles that are scoped to Amazon ECS resources with specific tags. In the following policy, an IAM principals starts and stops all tasks with a tag-key of `Department` and a tag-value of `Accounting`.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:StartTask",
                "ecs:StopTask",
                "ecs:RunTask"
            ],
            "Resource": "arn:aws:ecs:*",
            "Condition": {
```

```
                "StringEquals": {"ecs:ResourceTag/Department": "Accounting"}
            }
        }
    ]
}
```

Periodically audit access to the Amazon ECS APIs

A user might change roles. After they change roles, the permissions that were previously granted to them might no longer apply. Make sure that you audit who has access to the Amazon ECS APIs and whether that access is still warranted. Consider integrating IAM with a user lifecycle management solution that automatically revokes access when a user leaves the organization. For more information, see Amazon ECS security audit guidelines in the *Amazon Web Services General Reference*.

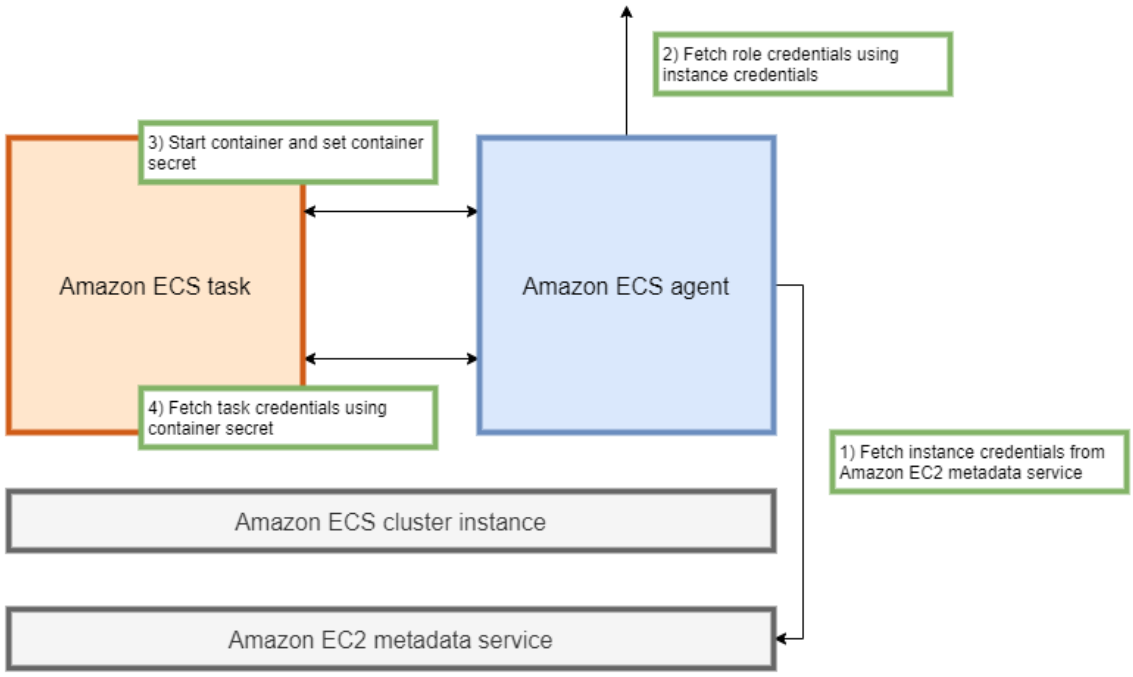
Using IAM roles with Amazon ECS tasks

We recommend that you assign a task an IAM role. Its role can be distinguished from the role of the Amazon EC2 instance that it's running on. Assigning each task a role aligns with the principle of least priviledged access and allows for greater granular control over actions and resources.

When assigning IAM roles for a task, you must use the folllowing trust policy so that each of your tasks can assume an IAM role that's different from the one that your EC2 instance uses. This way, your task doesn't inherit the role of your EC2 instance.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "ecs-tasks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

When you add a task role to a task definition, the Amazon ECS container agent automatically creates a token with a unique credential ID (for example, `12345678-90ab-cdef-1234-567890abcdef`) for the task. This token and the role credentials are then added to the agent's internal cache. The agent populates the the environment variable `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` in the container with the URI of the credential ID (for example, `/v2/credentials/12345678-90ab-cdef-1234-567890abcdef`).



You can manually retrieve the temporary role credentials from inside a container by appending the environment variable to the IP address of the Amazon ECS container agent and running the `curl` command on the the resulting string.

```
curl 192.0.2.0$AWS_CONTAINER_CREDENTIALS_RELATIVE_URI
```

The expected output is as follows:

```
{
 "RoleArn": "arn:aws:iam::123456789012:role/SSMTaskRole-SSMFargateTaskIAMRole-
DASWWSF2WGD6",
 "AccessKeyId": "AKIAIOSFODNN7EXAMPLE",
 "SecretAccessKey": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
 "Token": "IQoJb3JpZ2luX2VjEEM/Example==",
 "Expiration": "2021-01-16T00:51:53Z"
}
```

Newer versions of the AWS SDKs automatically fetch these credentials from the `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` environment varible when making AWS API calls.

The output includes an access key-pair consisting of a secret access key ID and a secret key which your application uses to access AWS resources. It also includes a token that AWS uses to verify that the credentials are valid. By default, credentials assigned to tasks using task roles are valid for six hours. After that, they are automatically rotated by the Amazon ECS container agent.

Task execution role

The task execution role is used to grant the Amazon ECS container agent permission to call specific AWS API actions on your behalf. For example, when you use AWS Fargate, Fargate needs an IAM role that allows it to pull images from Amazon ECR and write logs to CloudWatch Logs. An IAM role is also required when a task references a secret that's stored in AWS Secrets Manager, such as an image pull secret.

Note
If you're pulling images as an authenticated user, you're less likely to be impacted by the changes that occurred to Docker Hub's pull rate limits. For more information see, Private registry authentication for container instances.
By using Amazon ECR and Amazon ECR Public, you can avoid the limits imposed by Docker. If you pull images from Amazon ECR, this also helps shorten network pull times and reduces data transfer changes when traffic leaves your VPC.

Important
When you use Fargate, you must authenticate to a private image registry using `repositoryCredentials`. It's not possible to set the Amazon ECS container agent environment variables `ECS_ENGINE_AUTH_TYPE` or `ECS_ENGINE_AUTH_DATA` or modify the `ecs.config` file for tasks hosted on Fargate. For more information, see Private registry authentication for tasks.

Amazon EC2 container instance role

The Amazon ECS container agent is a container that runs on each Amazon EC2 instance in an Amazon ECS cluster. It's initialized outside of Amazon ECS using the `init` command that's available on the operating system. Consequently, it can't be granted permissions through a task role. Instead, the permissions have to be assigned to the Amazon EC2 instances that the agents run on. The actions list in the example `AmazonEC2ContainerServiceforEC2Role` policy need to be granted to the `ecsInstanceRole`. If you don't do this, your instances cannot join the cluster.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeTags",
                "ecs:CreateCluster",
                "ecs:DeregisterContainerInstance",
                "ecs:DiscoverPollEndpoint",
                "ecs:Poll",
                "ecs:RegisterContainerInstance",
                "ecs:StartTelemetrySession",
                "ecs:UpdateContainerInstancesState",
                "ecs:Submit*",
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

In this policy, the `ecr` and `logs` api actions allow the containers that are running on your instances to pull images from Amazon ECR and write logs to Amazon CloudWatch. The `ecs` actions allow the agent to register and de-register instances and to communicate with the Amazon ECS control plane. Of these, the `ecs:CreateCluster` action is optional.

Service-linked roles

You can use the service-linked role for Amazon ECS to grant the Amazon ECS service permission to call other service APIs on your behalf. Amazon ECS needs the permissions to create and delete network

interfaces, register, and de-register targets with a target group. It also needs the necessary permissions to create and delete scaling policies. These permissions are granted through the service-linked role. This role is created on your behalf the first time that you use the service.

Note
If you inadvertantly delete the service-linked role, you can recreate it. For instructions, see Create the service-linked role.

Recommendations

We recommend that you do the following when setting up your task IAM roles and policies.

Block access to Amazon EC2 metadata

When you run your tasks on Amazon EC2 instances, we strongly recommend that you block access to Amazon EC2 metadata to prevent your containers from inheriting the role assigned to those instances. If your applications have to call an AWS API action, use IAM roles for tasks instead.

To prevent tasks running in **bridge** mode from accessing Amazon EC2 metadata, run the following command or update the instance's user data. For more instruction on updating the user data of an instance, see this AWS Support Article. For more information about the task definition bridge mode, see task definition network mode.

```
sudo yum install -y iptables-services; sudo iptables --insert FORWARD 1 --in-interface
 docker+ --destination 192.0.2.0/32 --jump DROP
```

For this change to persist after a reboot, run the following command that's specific for your Amazon Machine Image (AMI):

- Amazon Linux 2

```
sudo iptables-save | sudo tee /etc/sysconfig/iptables && sudo systemctl enable --now
 iptables
```

- Amazon Linux

```
sudo service iptables save
```

For tasks that use `awsvpc` network mode, set the environment variable `ECS_AWSVPC_BLOCK_IMDS` to `true` in the `/etc/ecs/ecs.config` file.

You should set the `ECS_ENABLE_TASK_IAM_ROLE_NETWORK_HOST` variable to `false` in the ecs-agent config file to prevent the containers that are running within the `host` network from accessing the Amazon EC2 metadata.

Use `awsvpc` network mode

Use the network `awsvpc` network mode to restrict the flow of traffic between different tasks or between your tasks and other services that run within your Amazon VPC. This adds an additional layer of security. The `awsvpc` network mode provides task-level network isolation for tasks that run on Amazon EC2. It is the default mode on AWS Fargate. it's the only network mode that you can use to assign a security group to tasks.

Use IAM Access Advisor to refine roles

We recommend that you remove any actions that were never used or haven't been used for some time. This prevents unwanted access from happening. To do this, review the results produced by IAM Access

Advisor, and then remove actions that were never used or haven't been used recently. You can do this by following the following steps.

Run the following command to generate a report showing the last access information for the referenced policy:

```
aws iam generate-service-last-accessed-details --arn arn:aws:iam::123456789012:policy/
ExamplePolicy1
```

use the `JobId` that was in the output to run the following command. After you do this, you can view the results of the report.

```
aws iam get-service-last-accessed-details --job-id 98a765b4-3cde-2101-2345-example678f9
```

For more information, see IAM Access Advisor.

Monitor AWS CloudTrail for suspicious activity

You can monitor AWS CloudTrail for any suspicious activity. Most AWS API calls are logged to AWS CloudTrail as events. They are analyzed by AWS CloudTrail Insights, and you're alerted of any suspicious behavior that's associated with `write` API calls. This might includea spike in call volume. These alerts include such information as the time the unusual activity occurred and the top identity ARN that contributed to the APIs.

You can identify actions that are performed by tasks with an IAM role in AWS CloudTrail by looking at the event's `userIdentity` property. In the following example, the `arn` includes of the name of the assumed role, `s3-write-go-bucket-role`, followed by the name of the task, `7e9894e088ad416eb5cab92afExample`.

```
"userIdentity": {
    "type": "AssumedRole",
    "principalId": "AROA36C6WWEJ2YEXAMPLE:7e9894e088ad416eb5cab92afExample",
    "arn": "arn:aws:sts::123456789012:assumed-role/s3-write-go-bucket-
role/7e9894e088ad416eb5cab92afExample",
    ...
}
```

Note
When tasks that assume a role are run on Amazon EC2 container instances, a request is logged by Amazon ECS container agent to the audit log of the agent that's located at an address in the `/var/log/ecs/audit.log.YYYY-MM-DD-HH` format. For more information, see Task IAM Roles Log and Logging Insights Events for Trails.

Network security

Network security is a broad topic that encompasses several subtopics. These include encryption-in-transit, network segmentation and isolation, firewalling, traffic routing, and observability.

Encryption in transit

Encrypting network traffic prevents unauthorized users from intercepting and reading data when that data is transmitted across a network. With Amazon ECS, network encryption can be implemented in any of the following ways.

- **With a service mesh (TLS):**

  With AWS App Mesh, you can configure TLS connections between the Envoy proxies that are deployed with mesh endpoints. Two examples are virtual nodes and virtual gateways. The TLS certificates can come from AWS Certificate Manager (ACM). Or, it can come from your own private certificate authority.
  - Enabling Transport Layer Security (TLS)
  - Enable traffic encryption between services in AWS App Mesh using ACM certificates or customer provided certs
  - TLS ACM walkthrough
  - TLS file walkthrough
  - Envoy
- **Using Nitro instances:**

  By default, traffic is automatically encrypted between the following Nitro instance types: C5n, G4, I3en, M5dn, M5n, P3dn, R5dn, and R5n. Traffic isn't encrypted when it's routed through a transit gateway, load balancer, or similar intermediary.
  - Encryption in transit
  - What's new accouncement from 2019
  - This talk from re:Inforce 2019
- **Using Server Name Indication (SNI) with an Application Load Balancer:**

  The Application Load Balancer (ALB) and Network Load Balancer (NLB) support Server Name Indication (SNI). By using SNI, you can put multiple secure applications behind a single listener. For this, each has its own TLS certificate. We recommend that you provision certificates for the load balanacer using AWS Certificate Manager (ACM) and then add them to the listener's certificate list. The AWS load balancer uses a smart certificate selection algorithm with SNI. If the hostname that's provided by a client matches a single certificate in the certificate list, the load balancer chooses that certificate. If a hostname that's provided by a client matches multiple certificates in the list, the load balancer selects a certificate that the client can support. Examples include self-signed certificate or a certificate generated through the ACM.
  - SNI with Application Load Balancer
  - SNI with Network Load Balancer
- **End-to-end encryption with TLS certificates:**

  This involves deploying a TLS certificate with the task. This can either be a self-signed certificate or a certificate from a trusted certificate authority. You can obtain the certificate by referencing a secret for the certificate. Otherwise, you can choose to run an container that issues a Certificate Signing Request (CSR) to ACM and then mounts the resulting secret to a shared volume.
  - Maintaining transport layer security all the way to your containers using the Network Load Balancer with Amazon ECS part 1
  - Maintaining Transport Layer Security (TLS) all the way to your container part 2: Using AWS Certificate Manager Private Certificate Authority

Task networking

The following recommendations are in consideration of how Amazon ECS works. Amazon ECS doesn't use an overlay network. Instead, tasks are configured to operate in different network modes. For example, tasks that are configured to use `bridge` mode acquire a non-routable IP address from a Docker network that runs on each host. Tasks that are configured to use the `awsvpc` network mode acquire an IP address from the subnet of the host. Tasks that are configured with `host` networking use the host's network interface. `awsvpc` is the preferred network mode. This is because it's the only mode that you can use to assign security groups to tasks. It's also the only mode that's available for AWS Fargate tasks on Amazon ECS.

Security groups for tasks

We recommend that you configure your tasks to use the `awsvpc` network mode. After you configure your task to use this mode, the Amazon ECS agent automatically provisions and attaches an Elastic Network Interface (ENI) to the task. When the ENI is provisioned, the task is enrolled in an AWS security group. The security group acts as a virtual firewall that you can use to control inbound and outbound traffic.

Service mesh and Mutual Transport Layer Security (mTLS)

You can use a service mesh such as AWS App Mesh to control network traffic. By default, a virtual node can only communicate with its configured service backends, such as the virtual services that the virtual node will communicate with. If a virtual node needs to communicate with a service outside the mesh, you can use the `ALLOW_ALL` outbound filter or by creating a virtual node inside the mesh for the external service. For more information, see Kubernetes Egress How-To Walkthrough.

App Mesh also gives you the ability to use Mutual Transport Layer Security (mTLS) where both the client and the server are mutually authenticated using certificates. The subsequent communication between client and server are then encrypted using TLS. By requiring mTLS between services in a mesh, you can verify that the traffic is coming from a trusted source. For more information, see the following topics:

- mTLS authentication
- mTLS Secret Discovery Service (SDS) walkthrough
- mTLS File walkthrough

AWS PrivateLink

AWS PrivateLink is a networking technology that allows you to create private endpoints for different AWS services, including Amazon ECS. The endpoints are required in sandboxed environments where there is no Internet Gateway (IGW) attached to the Amazon VPC and no alternative routes to the Internet. Using AWS PrivateLink ensures that calls to the Amazon ECS service stay within the Amazon VPC and do not traverse the internet. For instructions on how to create AWS PrivateLink endpoints for Amazon ECS and other related services, see Amazon ECS interface Amazon VPC endpoints.

> **Important**
> AWS Fargate tasks don't require a AWS PrivateLink endpoint for Amazon ECS.

Amazon ECR and Amazon ECS both support endpoint policies. These policies allow you to refine access to a service's APIs. For example, you could create an endpoint policy for Amazon ECR that only allows images to be pushed to registries in particular AWS accounts. A policy like this could be used to prevent data from being exfiltrated through container images while still allowing users to push to authorized Amazon ECR registries. For more information, see Use VPC endpoint policies.

The following policy allows all AWS principals in your account to perform all actions against only your Amazon ECR repositories:

```
{
  "Statement": [
    {
      "Sid": "LimitECRAccess",
      "Principal": "*",
      "Action": "*",
      "Effect": "Allow",
      "Resource": "arn:aws:ecr:region:your_account_id:repository/*"
    },
  ]
```

```
}
```

You can enhance this further by setting a condition that uses the new `PrincipalOrgID` property. This prevents pushing and pulling of images by an IAM principal that isn't part of your AWS Organizations. For more information, see aws:PrincipalOrgID.

We recommended applying the same policy to both the `com.amazonaws.region.ecr.dkr` and the `com.amazonaws.region.ecr.api` endpoints.

Amazon ECS container agent settings

The Amazon ECS container agent configuration file includes several environment variables that relate to network security. `ECS_AWSVPC_BLOCK_IMDS` and `ECS_ENABLE_TASK_IAM_ROLE_NETWORK_HOST` are used to block a task's access to Amazon EC2 metadata. `HTTP_PROXY` is used to configure the agent to route through a HTTP proxy to connect to the internet. For instructions on configuring the agent and the Docker runtime to route through a proxy, see HTTP Proxy Configuration.

Important
These settings aren't available when you use AWS Fargate.

Recommendations

We recommend that you do the following when setting up your Amazon VPC, load balancers, and network.

Use network encryption where applicable

You should use network encryption where applicable. Certain compliance programs, such as PCI DSS, require that you encrypt data in transit if the data contains cardholder data. If your workload has similar requirements, configure network encryption.

Modern browsers warn users when connecting to insecure sites. If your service is fronted by a public facing load balancer, use TLS/SSL to encrypt the traffic from the client's browser to the load balancer and re-encrypt to the backend if warranted.

Use `awsvpc` network mode and security groups when you need to control traffic between tasks or between tasks and other network resources

You should use `awsvpc` network mode and security groups when you need to control traffic between tasks and between tasks and other network resources. If your service behind an ALB, use security groups to only allow inbound traffic from other network resources using the same security group as your ALB. If your application is behind an NLB, configure the task's security group to only allow inbound traffic from the Amazon VPC CIDR range and the static IP addresses assigned to the NLB.

Security groups should also be used to control traffic between tasks and other resources within the Amazon VPC such as Amazon RDS databases.

Create clusters in separate Amazon VPCs when network traffic needs to be strictly isolated

You should create clusters in separate Amazon VPCs when network traffic needs to be strictly isolated. Avoid running workloads that have strict security requirements on clusters with workloads that don't have to adhere to those requirements. When strict network isolation is mandatory, create clusters in separate Amazon VPCs and selectively expose services to other Amazon VPCs using Amazon VPC endpoints. For more information, see Amazon VPC endpoints.

Configure AWS PrivateLink endpoints when warranted

You should configure AWS PrivateLink endpoints endpoints when warranted. If your security policy prevents you from attaching an Internet Gateway (IGW) to your Amazon VPCs, configure AWS PrivateLink endpoints for Amazon ECS and other services such as Amazon ECR, AWS Secrets Manager, and Amazon CloudWatch.

Use Amazon VPC Flow Logs to analyze the traffic to and from long-running tasks

You should use Amazon VPC Flow Logs to analyze the traffic to and from long-running tasks. Tasks that use `awsvpc` network mode get their own ENI. Doing this, you can monitor traffic that goes to and from individual tasks using Amazon VPC Flow Logs. A recent update to Amazon VPC Flow Logs (v3), enriches the logs with traffic metadata including the vpc ID, subnet ID, and the instance ID. This metadata can be used to help narrow an investigation. For more information, see Amazon VPC Flow Logs.

> **Note**
> Because of the temporary nature of containers, flow logs might not always be an effective way to analyze traffic patterns between different containers or containers and other network resources.

Secrets management

Secrets, such as API keys and database credentials, are frequently used by applications to gain access other systems. They often consist of a username and password, a certificate, or API key. Access to these secrets should be restricted to specific IAM principals that are using IAM and injected into containers at runtime.

Secrets can be seamlessly injected into containers from AWS Secrets Manager and Amazon EC2 Systems Manager Parameter Store. These secrets can be referenced in your task as any of the following.

1. They're referenced as environment variables that use the `secrets` container definition parameter.
2. They're referenced as `secretOptions` if your logging platform requires authentication. For more information, see logging configuration options.
3. They're referenced as secrets pulled by images that use the `repositoryCredentials` container definition parameter if the registry where the container is being pulled from requires authentication. Use this method when pulling images from Docker Hub. For more information, see Private registry authentication for tasks.

Recommendations

We recommend that yo do the following when setting up secrets management.

Use AWS Secrets Manager or Amazon EC2 Systems Manager Parameter Store for storing secret materials

You should securely store API keys, database credentials, and other secret materials in AWS Secrets Manager or as an encrypted parameter in Amazon EC2 Systems Manager Parameter Store. These services are similar because they're both managed key-value stores that use AWS KMS to encrypt sensitive data. AWS Secrets Manager, however, also includes the ability to automatically rotate secrets, generate random secrets, and share secrets across AWS accounts. If you deem these important features, use AWS Secrets Manager otherwise use encrypted parameters.

Note
Tasks that reference a secret from AWS Secrets Manager or Amazon EC2 Systems Manager Parameter Store require a **Task Execution Role** with a policy that grants the Amazon ECS access to the desired secret and, if applicable, the AWS KMS key used to encrypt and decrypt that secret.

Important
Secrets that are referenced in tasks aren't rotated automatically. If your secret changes, you must force a new deployment or launch a new task to retrieve the latest secret value. For more information, see the following topics:

- AWS Secrets Manager: Injecting data as environment variables
- Amazon EC2 Systems Manager Parameter Store: Injecting data as environment variables

Retrieving data from an encrypted Amazon S3 bucket

Because the value of environment variables can inadvertently leak in logs and are revealed when running `docker inspect`, you should store secrets in an encrypted Amazon S3 bucket and use task roles to restrict access to those secrets. When you do this, your application must be written to read the secret from the Amazon S3 bucket. For instructions, see Setting default server-side encryption behavior for Amazon S3 buckets.

Mount the secret to a volume using a sidecar container

Because there's an elevated risk of data leakage with environment variables, you should run a sidecar container that reads your secrets from AWS Secrets Manager and write them to a shared volume. This container can run and exit before the application container by using Amazon ECS container ordering. When you do this, the application container is subsequently mount the volume where the secret was written. Like the Amazon S3 bucket method, your application must be written to read the secret from the shared volume. Because the volume is scoped to the task, the volume is automatically deleted after the task stops. For an example of a sidecar container, see the aws-secret-sidecar-injector project.

Note
On Amazon EC2, the volume that the secret is written to can be encrypted with a AWS KMS customer managed key. On AWS Fargate, volume storage is automatically encrypted using a service managed key.

Additional resources

- Passing secrets to containers in an Amazon ECS task
- Chamber is a wrapper for storing secrets in Amazon EC2 Systems Manager Parameter Store

Compliance

Your compliance responsibility when using Amazon ECS is determined by the sensitivity of your data, and the compliance objectives of your company, and applicable laws and regulations.

AWS provides the following resources to help with compliance:

- Security and compliance quick start guides: These deployment guides discuss architectural considerations and provide steps for deploying security and compliance-focused baseline environments on AWS.
- Architecting for HIPAA Security and Compliance Whitepaper: This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.

- AWS Services in Scope by Compliance Program: This list contains the AWS services in scope of specific compliance programs. For more information, see AWS Compliance Programs.

Payment Card Industry Data Security Standards (PCI DSS)

It's important that you understand the complete flow of cardholder data (CHD) within the environment when adhering to PCI DSS. The CHD flow determines the applicability of the PCI DSS, defines the boundaries and components of a cardholder data environment (CDE), and therefore the scope of a PCI DSS assessment. Accurate determination of the PCI DSS scope is key to defining the security posture and ultimately a successful assessment. Customers must have a procedure for scope determination that assures its completeness and detects changes or deviations from the scope.

The temporary nature of containerized applications provides additional complexities when auditing configurations. As a result, customers need to maintain an awareness of all container configuration parameters to ensure compliance requirements are addressed throughout all phases of a container lifecycle.

For additional information on achieving PCI DSS compliance on Amazon ECS, refer to the following whitepapers.

- Architecting on Amazon ECS for PCI DSS compliance
- Architecting for PCI DSS Scoping and Segmentation on AWS

HIPAA (U.S. Health Insurance Portability and Accountability Act)

Using Amazon ECS with workloads that process protected health information (PHI) requires no additional configuration. Amazon ECS acts as an orchestration service that coordinates the launch of containers on Amazon EC2. It doesn't operate with or upon data within the workload being orchestrated. Consistent with HIPAA regulations and the AWS Business Associate Addendum, PHI should be encrypted in transit and at-rest when accessed by containers launched with Amazon ECS.

Various mechanisms for encrypting at-rest are available with each AWS storage option, such as Amazon S3, Amazon EBS, and AWS KMS. You may deploy an overlay network (such as VNS3 or Weave Net) to ensure complete encryption of PHI transferred between containers or to provide a redundant layer of encryption. Complete logging should also be enabled and all container logs should be directed to Amazon CloudWatch. For more information, see Architecting for HIPAA Security and Compliance.

Recommendations

You should engage the compliance program owners within your business early and use the AWS shared responsibility model to identify compliance control ownership for success with the relevant compliance programs.

Logging and monitoring

Logging and monitoring are an important aspect of maintaining the reliability, availability, and performance of Amazon ECS and your AWS solutions. AWS provides several tools for monitoring your Amazon ECS resources and responding to potential incidents:

- Amazon CloudWatch Alarms
- Amazon CloudWatch Logs
- Amazon CloudWatch Events
- AWS CloudTrail Logs

You can configure the containers in your tasks to send log information to Amazon CloudWatch Logs. If you're using the AWS Fargate launch type for your tasks, you can view the logs from your containers. If you're using the Amazon EC2 launch type, you can view different logs from your containers in one convenient location. This also prevents your container logs from taking up disk space on your container instances.

For more information about Amazon CloudWatch Logs, see **Monitor Logs from Amazon EC2 Instances** in the Amazon CloudWatch User Guide. For instruction on sending container logs from your tasks to Amazon CloudWatch Logs, see Using the `awslogs` log driver.

Container logging with Fluent Bit

AWS provides a Fluent Bit image with plugins for both Amazon CloudWatch Logs and Amazon Kinesis Data Firehose. This image provides the capability to route logs to Amazon CloudWatch and Amazon Kinesis Data Firehose destinations (which include Amazon S3, Amazon OpenSearch Service, and Amazon Redshift). We recommend using Fluent Bit as your log router because it has a lower resource utilization rate than Fluentd. For more information, see Amazon CloudWatch Logs for Fluent Bit and Amazon Kinesis Data Firehose for Fluent Bit.

The AWS for Fluent Bit image is available on:

- Amazon ECR on Amazon ECR Public Gallery
- Amazon ECR repository (in most Regions of high availability)
- Docker Hub

The following shows the syntax to use for the Docker CLI.

```
docker pull public.ecr.aws/aws-observability/aws-for-fluent-bit:tag
```

For example, you can pull the latest AWS for Fluent Bit image using this Docker CLI command:

```
docker pull public.ecr.aws/aws-observability/aws-for-fluent-bit:latest
```

Also refer to the following blog posts for more information on Fluent Bit and related features:

- Fluent Bit for Amazon EKS on AWS Fargate
- Centralized Container Logging with Fluent Bit
- Building a scalable log solution aggregator with AWS Fargate, Fluentd, and Amazon Kinesis Data Firehose

Custom log routing - FireLens for Amazon ECS

With FireLens for Amazon ECS, you can use task definition parameters to route logs to an AWS service or AWS Partner Network (APN) destination for log storage and analytics. FireLens works with Fluentd and Fluent Bit. We provide the AWS for Fluent Bit image. Or, you can alternatively use your own Fluentd or Fluent Bit image.

You should consider the following conditions and considerations when using FireLens for Amazon ECS:

- FireLens for Amazon ECS is supported for tasks that are hosted both on AWS Fargate and Amazon EC2.
- FireLens for Amazon ECS is supported in AWS CloudFormation templates. For more information, see AWS::ECS::TaskDefinition FirelensConfiguration in the **AWS CloudFormation User Guide**.
- For tasks that use the `bridge` network mode, containers with the FireLens configuration must start before any of the application containers that rely on it start. To control the order that your containers start in, use dependency conditions in your task definition. For more information, see Container dependency.

AWS Fargate security

We recommend that you take into account the following best practices when you use AWS Fargate.

Use AWS KMS to encrypt ephemeral storage

You should have your ephemeral storage encrypted by AWS KMS. For Amazon ECS tasks that are hosted on AWS Fargate using platform version `1.4.0` or later, each task receives 20 GB of ephemeral storage. The amount of storage isn't adjustable. For such tasks that were launched on May 28, 2020 or later, the ephemeral storage is encrypted with an AES-256 encryption algorithm using an encryption key managed by AWS Fargate.

Example: Launching an Amazon ECS task on AWS Fargate platform version 1.4.0 with ephemeral storage encryption

The following command will launch an Amazon ECS task on AWS Fargate platform version 1.4. Because this task is launched as part of the Amazon ECS cluster, it uses the 20 GB of ephemeral storage that's automatically encrypted.

```
aws ecs run-task --cluster clustername \
  --task-definition taskdefinition:version \
  --count 1
  --launch-type "FARGATE" \
  --platform-version 1.4.0 \
  --network-configuration
 "awsvpcConfiguration={subnets=[subnetid],securityGroups=[securitygroupid]}" \
  --region region
```

SYS_PTRACE capability for kernel syscall tracing

The default configuration of Linux capabilities that are added or removed from your container are provided by Docker. For more information about the available capabilities, see Runtime privilege and Linux capabilities in the **Docker run** documentation.

Tasks that are launched on AWS Fargate only support adding the `SYS_PTRACE` kernel capability.

Refer to the tutorial video below that shows how to use this feature through the Sysdig Falco project.

#ContainersFromTheCouch - Troubleshooting your AWS Fargate Task using SYS_PTRACE capability

The code discussed in the previous video can be found on GitHub here.

Task and container security

You should consider the container image as your first line of defense against an attack. An insecure, poorly constructed image can allow an attacker to escape the bounds of the container and gain access to the host. You should do the following to mitigate the risk of this happening.

Recommendations

We recommend that you do the following when setting up your tasks and containers.

Create minimal or use distroless images

Start by removing all extraneous binaries from the container image. If you're using an unfamiliar image from Docker Hub, inspect the image to refer to the contents of each of the container's layers. You can use an application such as Dive to do this.

Alternatively, you can use **distroless** images that only include your application and its runtime dependencies. They don't contain package managers or shells. Distroless images improve the "signal to noise of scanners and reduces the burden of establishing provenance to just what you need." For more information, see the GitHub documentation on distroless.

Docker has a mechanism for creating images from a reserved, minimal image known as **scratch**. Formore information, see Creating a simple parent image using **scratch** in the Docker documentation. With langages like Go, you can create a static linked binary and reference it in your Dockerfile. The following example shows how you can accomplish this.

```
############################
# STEP 1 build executable binary
############################
FROM golang:alpine AS builder
# Install git.
# Git is required for fetching the dependencies.
RUN apk update && apk add --no-cache git
WORKDIR $GOPATH/src/mypackage/myapp/
COPY . .
# Fetch dependencies.
# Using go get.
RUN go get -d -v
# Build the binary.
RUN go build -o /go/bin/hello
############################
# STEP 2 build a small image
############################
FROM scratch
# Copy our static executable.
COPY --from=builder /go/bin/hello /go/bin/hello
# Run the hello binary.
ENTRYPOINT ["/go/bin/hello"]
This creates a container image that consists of your application and nothing else, making
 it extremely secure.
```

The previous example is also an example of a multi-stage build. These types of builds are attractive from a security standpoint because you can use them to minimize the size of the final image pushed to your container registry. Container images devoid of build tools and other extraneous binaries improves your security posture by reducing the attack surface of the image. For more information about multi-stage builds, see creating multi-stage builds.

Scan your images for vulnerabilities

Similar to their virtual machine counterparts, container images can contain binaries and application libraries with vulnerabilities or develop vulnerabilities over time. The best way to safeguard against exploits is by regularly scanning your images with an image scanner. Images that are stored in Amazon ECR can be scanned on push or on-demand (once every 24 hours). Amazon ECR currently uses Clair, an open-source image scanning solution. After an image is scanned, the results are logged to the

Amazon ECR event stream in Amazon EventBridge. You can also see the results of a scan from within the Amazon ECR console or by calling the DescribeImageScanFindings API. Images with a `HIGH` or `CRITICAL` vulnerability should be deleted or rebuilt. If an image that has been deployed develops a vulnerability, it should be replaced as soon as possible.

Docker Desktop Edge version 2.3.6.0 or later can scan local images. The scans are powered by Snyk, an application security service. When vulnerabilities are discovered, Snyk identifies the layers and dependencies with the vulnerability in the Dockerfile. It also recommends safe alternatives like using a slimmer base image with fewer vulnerabilities or upgrading a particular package to a newer version. By using Docker scan, developers can resolve potential security issues before pushing their images to the registry.

- Automating image compliance using Amazon ECR and AWS Security Hub explains how to surface vulnerability information from Amazon ECR in AWS Security Hub and automate remediation by blocking access to vulnerable images.

Remove special permissions from your images

The access rights flags `setuid` and `setgid` allow running an executable with the permissions of the owner or group of the executable. Remove all binaries with these access rights from your image as these binaries can be used to escalate privileges. Consider removing all shells and utilities like `nc` and `curl` that can be used for malicious purposes. You can find the files with `setuid` and `setgid` access rights by using the following command.

```
find / -perm /6000 -type f -exec ls -ld {} \;
```

To remove these special permissions from these files, add the following directive to your container image.

```
RUN find / -xdev -perm /6000 -type f -exec chmod a-s {} \; || true
```

Create a set of curated images

Rather than allowing developers to create their own images, create a set of vetted images for the different application stacks in your organization. By doing so, developers can forego learning how to compose Dockerfiles and concentrate on writing code. As changes are merged into your codebase, a CI/CD pipeline can automatically compile the asset and then store it in an artifact repository. And, last, copy the artifact into the appropriate image before pushing it to a Docker registry such as Amazon ECR. At the very least you should create a set of base images hat developers can create their own Dockerfiles from. You should avoid pulling images from Docker Hub. You don't always know what is in the image and about a fifth of the top 1000 images have vulnerabilties. A list of those images and their vulnerabilities can be found at https://vulnerablecontainers.org/.

Scan application packages and libraries for vulnerabilities

Use of open source libraries is now common. As with operating systems and OS packages, these libraries can have vulnerabilities . As part of the development lifecycle these libraries should be scanned and updated when critical vulnerabilities are found.

Docker Desktop performs local scans using Snyk. It can also be used to find vulnerabilities and potential licensing issues in open source libraries. It can be integrated directly into developer workflows giving you the ability to mitigate risks posed by open source libraries. For more information, see the following topics:

- Open Source Application Security Tools includes a list of tools for detecting vulnerabilities in applications.
- Docker scanning cheatsheet

Perform static code analysis

You should perform static code analysis before building a container image. It's performed against your source code and is used to identify coding errors and code that could be exploited by a malicious actor, such as fault injections. SonarQube is a popular option for static application security testing (SAST), with support for a variety of different programming languages.

Run containers as a non-root user

You should run containers as a non-root user. By default, containers run as the `root` user unless the `USER` directive is included in your Dockerfile. The default Linux capabilities that are assigned by Docker restrict the actions that can be run as `root`, but only marginally. For example, a container running as `root` is still not allowed to access devices.

As part of your CI/CD pipeline you should lint Dockerfiles to look for the `USER` directive and fail the build if it's missing. For more information, see the following topics:

- Dockerfile-lint is an open-source tool from RedHat that can be used to check if the file conforms to best practices.
- Hadolint is another tool for building Docker images that conform to best practices.

Use a read-only root file system

You should use a read-only root file system. A container's root file system is writable by default. When you configure a container with a `RO` (read-only) root file system it forces you to explicitly define where data can be persisted. This reduces your attack surface because the container's file system can't be written to unless permissions are specifically granted.

> **Note**
> Having a read-only root file system can cause issues with certain OS packages that expect to be able to write to the filesystem. If you're planning to use read-only root file systems, thoroughly test beforehand.

Configure tasks with CPU and Memory limits (Amazon EC2)

You should configure tasks with CPU and memory limits to minimize the following risk. A task's resource limits set an upper bound for the amount of CPU and memory that can be reserved by all the containers within a task. If no limits are set, tasks have access to the host's CPU and memory. This can cause issues where tasks deployed on a shared host can starve other tasks of system resources.

> **Note**
> Amazon ECS on AWS Fargate tasks require you to specify CPU and memory limits because it uses these values for billing purposes. One task hogging all of the system resources isn't an issue for Amazon ECS Fargate because each task is run on its own dedicated instance. If you don't specify a memory limit, Amazon ECS allocates a minimum of 4MB to each container. Similarly, if no CPU limit is set for the task, the Amazon ECS container agent assigns it a minimum of 2 CPUs.

Use immutable tags with Amazon ECR

With Amazon ECR, you can and should use configure images with immutable tags. This prevents pushing an altered or updated version of an image to your image repository with an identical tag. This protects

against an attacker pushing a compromised version of an image over your image with the same tag. By using immutable tags, you effectively force yourself to push a new image with a different tag for each change.

Avoid running containers as privileged (Amazon EC2)

You should avoid running containers as privileged. For background, containers run as `privileged` are run with extended privileges on the host. This means the container inherits all of the Linux capabilities assigned to `root` on the host. It's use should be severely restricted or forbidden. We advise setting the Amazon ECS container agent environment variable `ECS_DISABLE_PRIVILEGED` to `true` to prevent containers from running as `privileged` on particular hosts if `privileged` isn't needed. Alternatively you can use AWS Lambda to scan your task definitions for the use of the `privileged` parameter.

Note
Running a container as `privileged` isn't supported on Amazon ECS on AWS Fargate.

Remove unnecessary Linux capabilities from the container

The following is a list of the default Linux capabilities assigned to Docker containers. For more information about each capability, see Overview of Linux Capabilities.

```
CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_FOWNER, CAP_FSETID, CAP_KILL,
CAP_SETGID, CAP_SETUID, CAP_SETPCAP, CAP_NET_BIND_SERVICE,
CAP_NET_RAW, CAP_SYS_CHROOT, CAP_MKNOD, CAP_AUDIT_WRITE,
CAP_SETFCAP
```

If a container doesn't require all of the Docker kernal capabilties listed above, consider dropping them from the container. For more information about each Docker kernal capability, see KernalCapabilities. You can find out which capabilities are in use by doing the following:

- Install the OS package libcap-ng and run the `pscap` utility to list the capabilities that each process is using.
- You can also use capsh to decipher which capabilities a process is using.
- Refer to Linux Capabilities 101 for more information.

Use a customer managed key (CMK) to encrypt images pushed to Amazon ECR

You should use a customer managed key (CMK) to encrupt images that are pushed to Amazon ECR. Images that are pushed to Amazon ECR are automatically encrypted at rest with a AWS Key Management Service (AWS KMS) managed key. If you would rather use your own key, Amazon ECR now supports AWS KMS encryption with customer managed keys (CMK). Before enabling server side encryption with a CMK, review the Considerations listed in the documentation on encryption at rest.

Runtime security

Runtime security provides active protection for your containers while they're running. The idea is to detect and prevent malicious activity from occuring your containers.

With secure computing (`seccomp`) you can prevent a containerized application from making certain syscalls to the underlying host operating system's kernel. While the Linux operating system has a few hundred system calls, the most of them aren't necessary for running containers. By restricting which syscalls can be made by a container, you can effectively decrease your application's attack surface.

To get started with seccomp, you can use `strace` to generate a stack trace to see which system calls your application is making. You can use a tool such as `syscall2seccomp` to create a seccomp profile from the data gathered from the stack trace. For more information, see strace and syscall2seccomp.

Unlike the SELinux security module, seccomp can't isolate containers from each other. However, it protects the host kernel from unauthorized syscalls. It functions by intercepting syscalls and only allowing those that have been allow-listed to pass through. Docker has a default seccomp profile that is suitable for a majority of general purpose workloads.

Note
It's also possible to create your own profiles for things that require additional privileges.

AppArmor is a Linux security module similar to seccomp, but it restricts a container's capabilities including accessing parts of the file system. It can be run in either `enforcement` or `complain` mode. Because building AppArmor profiles can be challenging, we recommend that you use a tool like bane. For more information about AppArmor, see the official AppArmor page.

Important
AppArmor is only available for Ubuntu and Debian distributions of Linux.

Recommendations

We recommend that you take the folowing actions when setting up your runtime security.

Use a third-party solution for runtime defense

Use a third-party solution for runtime defense. If you're familiar with how Linux security works, create and manage seccomp and AppArmor profiles. Both are open-source projects. Otherwise, consider using a different third-party service instead. Most use machine learning to block or alert on suspicious activity. For a list of available third-party solutions, see AWS Marketplace for Containers.

Add or remove Linux capabilities using seccomp policies

Use seccomp to have greater control over Linux capabilities and to avoid syscall check errors. Seccomp works as a syscall filter that revokes the permission to run certain syscalls or to use specific agruments.

AWS Partners

You can use any of the following AWS Partner products to add additional security and features to your Amazon ECS workloads. For more information, see Amazon ECS Partners.

Aqua Security

You can use Aqua Security to secure your cloud-native applications from development to production. The Aqua Cloud Native Security Platform integrates with your cloud-native resources and orchestration tools to provide transparent and automated security. It can prevent suspicious activity and attacks in real time, and help to enforce policy and simplify regulatory compliance.

Palo Alto Networks

Palo Alto Networks provides security and protection for your hosts, containers, and serverless infrastructure in the cloud and throughout the development and software lifecycle.

Twistlock is supplied by Palo Alto Networks and can be integrated with Amazon ECS FireLens. With it, you have access to high fidelity security logs and incidents that are seamlessly aggregated into several AWS services. These include Amazon CloudWatch, Amazon Athena, and Amazon Kinesis. Twistlock secures workloads that are deployed on AWS container services.

Sysdig

You can use Sysdig to run secure and compliant cloud-native workloads in production scenarios. The Sysdig Secure DevOps Platform has embedded security and compliance features to protect your cloud-native workloads, and also offers enterprise-grade scalability, performance, and customization.

Document history for the Amazon ECS Best Practices Guide

The following table describes the documentation releases for the Amazon ECS Best Practices Guide.

update-history-change	update-history-description	update-history-date
Deployment best practices (p. 74)	Added best practices for speeding up Amazon ECS deployments.	August 10, 2021
Security best practices (p. 74)	Added best practices for security management for Amazon ECS workloads.	May 26, 2021
Auto scaling and capacity management best practices (p. 74)	Added best practices for auto scaling and capacity management for Amazon ECS workloads.	May 14, 2021
Persistent storage best practices (p. 74)	Added best practices for persistent storage for Amazon ECS workloads.	May 7, 2021
Networking best practices (p. 74)	Added best practices for networking management for Amazon ECS workloads.	April 6, 2021
Initial release (p. 74)	Initial release of the Amazon ECS Best Practices Guide	April 6, 2021