Amazon ECS Best Practices

Best Practices Guide

Amazon ECS Best Practices: Best Practices Guide

Table of Contents

Introduction

Amazon Elastic Container Service (Amazon ECS) is a highly scalable, fast container management service that makes it easy to run, stop, and manage containers on a cluster. This guide covers many of the most important operational best practices while explaining core topics underpinning how Amazon ECS-based applications work. The goal is to provide a concrete, actionable approach to operating and troubleshooting Amazon ECS-based applications.

This guide will be revised regularly to incorporate new Amazon ECS best practices. If you have any questions or comments about any of the content in this guide, raise an issue in the GitHub repository.

Best Practices - Networking

Modern applications are typically built out of multiple components that communicate with each other as part of a distributed system. For example, you may have a mobile or web application that communicates with an API endpoint. That API may be powered by multiple microservices that communicate with each other. Some of those microservices may require internet access to talk to other supporting APIs on the Internet, or they need to talk to the APIs of AWS services like Amazon DynamoDB or Amazon SQS.

In this guide, we discuss best practices for building the networking that makes it so the distributed components of your application can communicate with each other securely, and in a scalable manner.

Topics

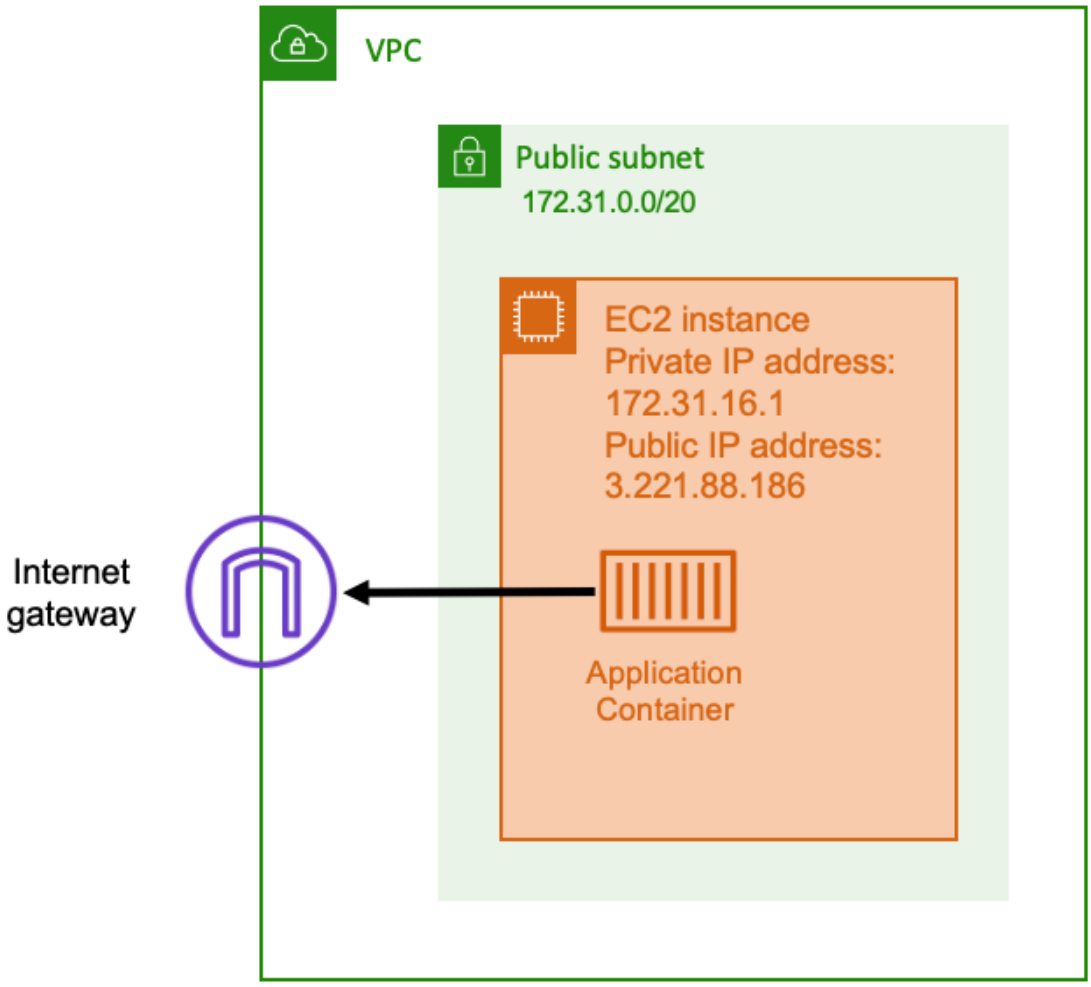
Connecting to the Internet

Many containerized applications have a least a few components that need outbound access to the Internet. For example, the backend for a mobile app that needs to talk to the Apple Push Notification service. Or a social media application that allows users to post website links, and it needs to fetch the content of a user submitted website link in order to generate a preview card for the link based on the Open Graph tags on the HTML page.

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

Topics

Using a public subnet and internet gateway



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

This network architecture allows direct communication between the host that is running your application and other hosts on the Internet. The communication is bi-directional; You can establish an outbound connection to any other host on the internet, but other hosts on the Internet may attempt to connect to your host as well. As a result, you need to pay extra attention to security group and firewall rules to ensure that other hosts on the Internet can't open any connections that you didn't intend.

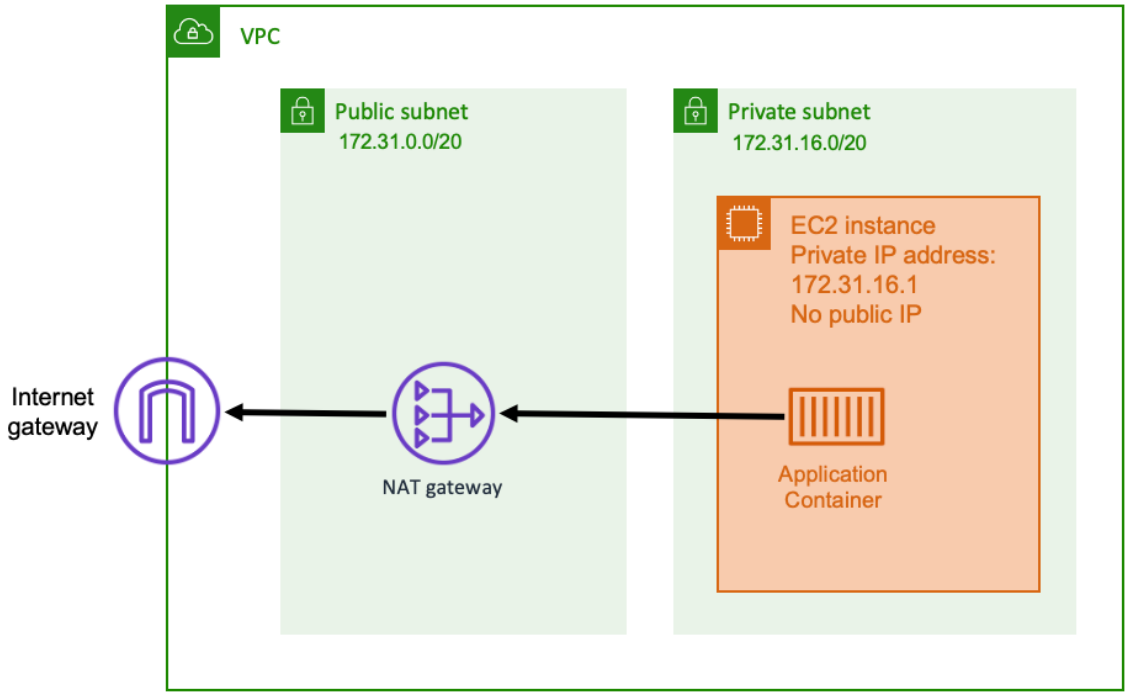
For example, if you are running on Amazon EC2, you want to ensure that port 22 for SSH access is locked down; otherwise it is likely that your instance will receive constant SSH connection attempts from bots. These bots trawl through public IP addresses, and once they find an open SSH port they attempt to brute-force passwords in the hopes of gaining access to your instance. For this reason, many organizations limit the usage of public subnets and prefer to have most, if not all, of their resources inside of private subnets.

However, using public subnets for networking is ideal for applications that are customer-facing and require large amounts of bandwidth or the lowest possible latency. Examples of such applications might include video streaming, or video game servers.

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

- Using Amazon EC2 — Launch EC2 instances on a public subnet. Amazon ECS uses these EC2 instances as cluster capacity, and any containers running on the instances will be able to use the underlying public IP address of the host for outbound networking. This applies when using the `host` and `bridge` network modes, but the `awsvpc` network mode does not provide task ENIs with public IP addresses, so they can't make direct use of an internet gateway.
- Using Fargate — When creating 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



Using a private subnet and a NAT gateway, your containerized application runs on a host that is in a private subnet. The host has a private IP address that is 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 not 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 NAT gateway's public IP address 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 from the Internet. For more information, see NAT gateways in the *Amazon VPC User Guide*.

This private networking approach is ideal for workloads that you want to protect from direct external access, such as payment processing or user data and passwords. You are charged for creating and using a NAT gateway in your account. NAT gateway hourly usage and data processing rates apply as well. For redundancy, you should have a NAT gateway in each Availability Zone so that the loss of a single Availability Zone doesn't compromise your outbound connectivity. For this reason, it may not be cost effective to use private subnets and NAT gateways if you have a small workload.

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

- Using Amazon EC2 — Launch EC2 instances on a private subnet. The containers that run on these EC2 hosts use the underlying hosts networking, and outbound requests will go through the NAT gateway.
- Using Fargate — When creating 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, and its outbound traffic is routed through any NAT gateway you have associated with that private subnet.

Receiving inbound connections from the Internet

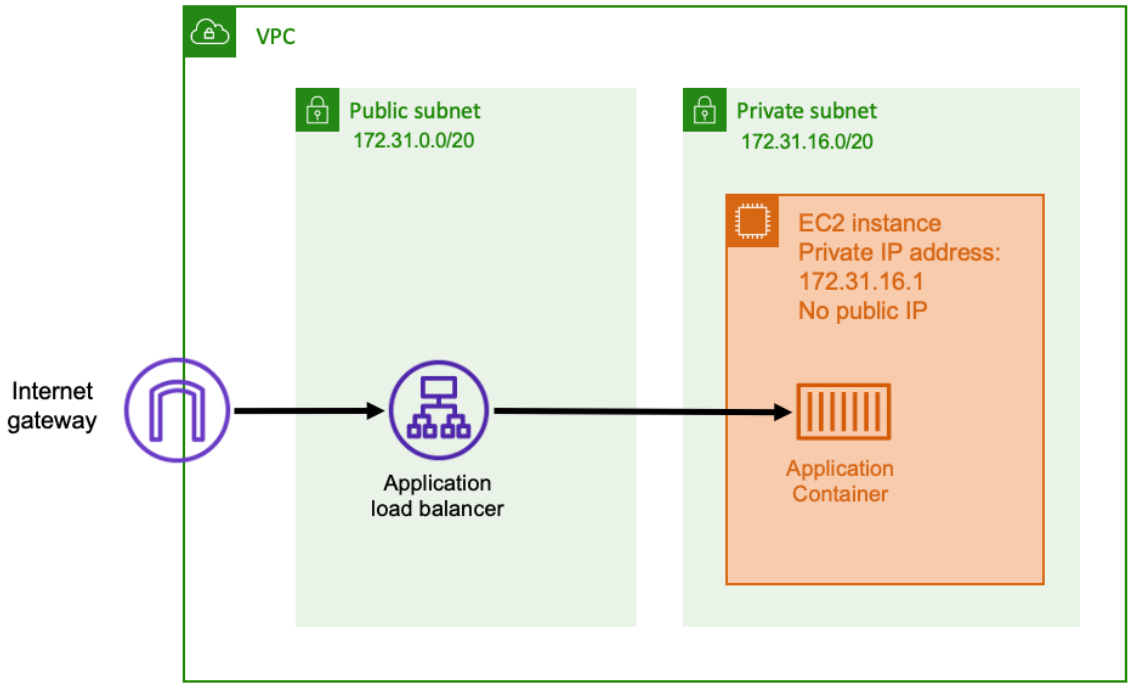
When running a public-facing service, it's likely you need to accept inbound traffic from the Internet as well. For example, you might have a public website that needs to accept inbound HTTP requests from browsers, or a public API that powers a web or mobile phone application. In this scenario, other hosts on the internet must be able to initiate an inbound connection to your application's host instance.

A simple method is to launch your containers on hosts that are in a public subnet with a public IP address. However, this approach is rarely used at scale. It is generally preferred to have a scalable ingress layer that sits between the Internet and your application instead of having arbitrary hosts on the Internet communicate directly with your application. There are a few different AWS services you can use as an ingress.

Topics

Application Load Balancer

An Application Load Balancer functions at the application layer, the seventh layer of the Open Systems Interconnection (OSI) model which makes them ideal for web-facing HTTP services. For example, if you have a website or an HTTP REST API then an Application Load Balancer is a great load balancer for this workload type. For more information, see What is an Application Load Balancer? in the *User Guide for Application Load Balancers*.



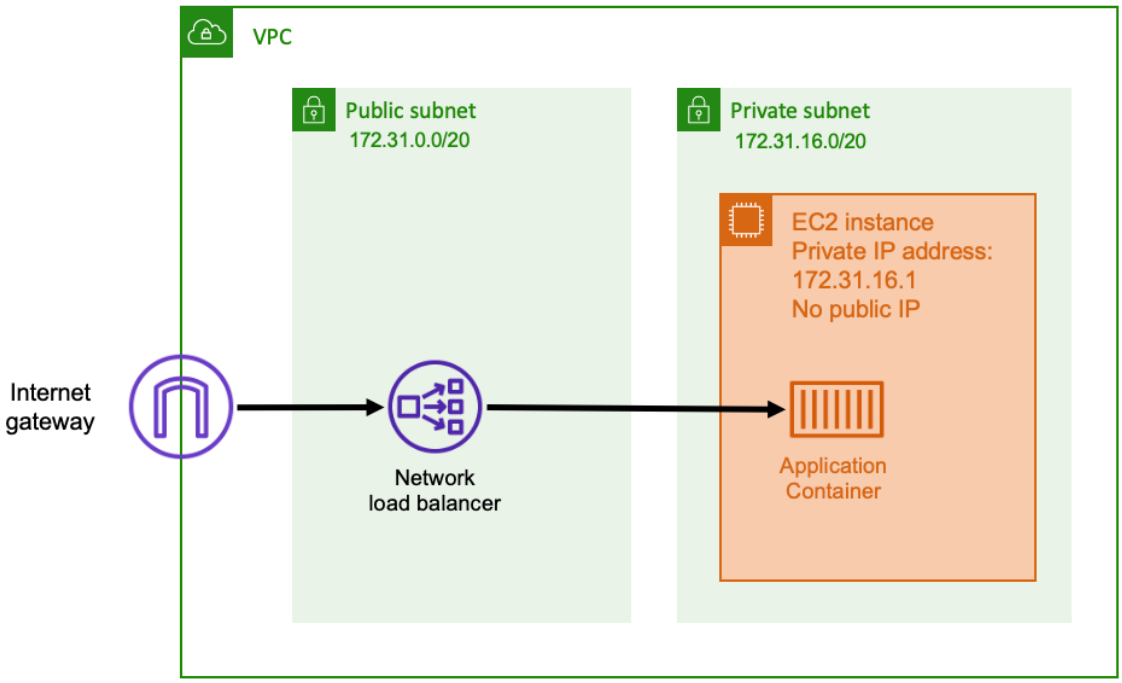
With this architecture, you create an Application Load Balancer in a public subnet so that it has a public IP address and can receive inbound connections from the Internet. When the Application Load Balancer receives an inbound connection and sees an HTTP request, it will open a connection to the application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

Application Load Balancers have the following useful features.

- SSL/TLS termination — Application Load Balancers can handle secure HTTPS communication and certificates for communicating with clients, and 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 — Application Load Balancers can have multiple DNS hostnames, and have powerful routing capabilities to send incoming HTTP requests to different destinations based on characteristics like the hostname, or even the path of the request. This means you can use a single Application Load Balancer as the front facing ingress for many different internal services, or even microservices on different paths of a REST API.
- gRPC support and websockets — Application Load Balancers handle more than just HTTP. They can also load balance gRPC and websocket based services, with HTTP/2 support.
- Security — Application Load Balancers help protect your application from bad traffic. They include features such as HTTP desync mitigations, and it integrates with AWS Web Application Firewall (AWS WAF) which can further filter out bad traffic that includes common attack patterns like 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 is ideal for non-HTTP protocols or situations where end to end encryption is necessary. It doesn't have the deep HTTP specific features that an Application Load Balancer does, but for applications that don't use HTTP, a Network Load Balancer is better. For more information, see What is a Network Load Balancer? in the *User Guide for Network Load Balancers*.



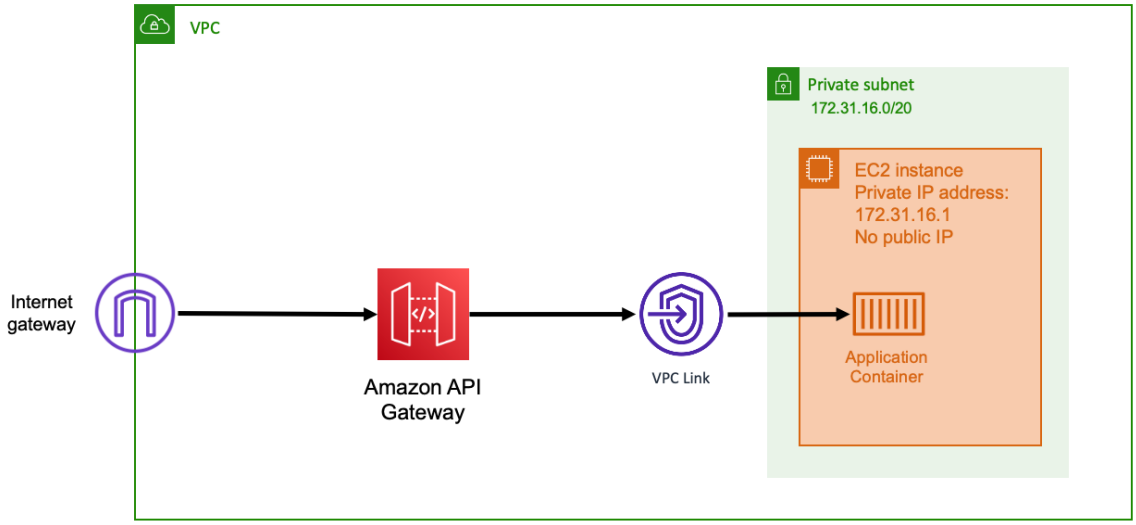
A Network Load Balancer as an ingress is very similar to an Application Load Balancer in that it is created in a public subnet, so that it can have a public IP address that is reachable from the Internet. The Network Load Balancer then opens a connection to the private IP address of the host running 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 can't do many of the things that the higher level 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 need to inspect the contents of packets, like an Application Load Balancer does. This makes it ideal for load balancing communications that need to stay end to end encrypted all the way to your application.
- TLS encryption — Although you can use a Network Load Balancer for full end to end encryption you can also choose to have the Network Load Balancer terminate TLS connections so that 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 is ideal for non HTTP workloads and protocols other than TCP.

Amazon API Gateway HTTP API

Amazon API Gateway HTTP API is a serverless ingress that is ideal for HTTP applications with wide ranging, bursty request volumes, or very low request volumes. For more information, see What is Amazon API Gateway? in the *API Gateway Developer Guide*.



The pricing model for Application Load Balancers and Network Load Balancers include an hourly price to keep the load balancers available for accepting incoming connections at all times. In contrast, API Gateway charges per request. So 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 but if you have a low number of requests overall or periods of extremely low traffic then the cumulative price for using the API Gateway would be cheaper than paying a constant hourly charge to maintain a load balancer that is being underutilized.

For example, an Application Load Balancer always costs at least $16.20 a month because of its hourly charge. On API Gateway HTTP API you could serve 16 million requests before you hit $16 in charges. So low traffic use cases under 16 million requests per month would be cheaper on API Gateway HTTP API as opposed to using an Application Load Balancer. However, an Application Load Balancer that is being utilized by clients with keep alive connections can serve hundreds of millions of requests per month for under $20.

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 discover 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

It also supports the usage of custom Lambda authorizers that help you protect your API from unauthorized usage. For more information, see Field Notes: Serverless Container-based APIs with Amazon ECS and Amazon API Gateway.

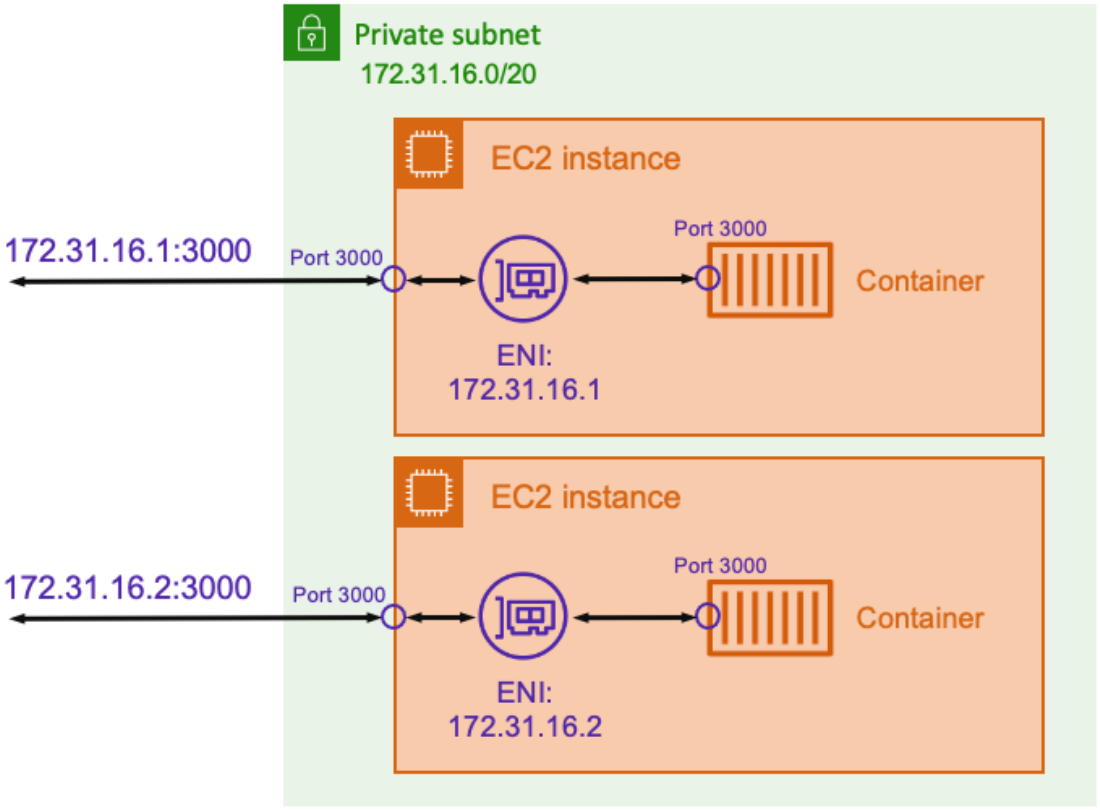
Choosing a network mode

The methods for architecting inbound and outbound network connections apply to all workloads on AWS, even if they aren't inside a container. When running containers on AWS, there is another level of networking to consider. One of the main reasons to use 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

Host mode

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



Using the example above, imagine you are running a Node.js container with an Express application that listens on port `3000`. 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. There is no mapping or virtual network. Instead, the container ports are mapped directly to the same ports on the Elastic Network Interface (ENI) attached to the underlying Amazon EC2 instance.

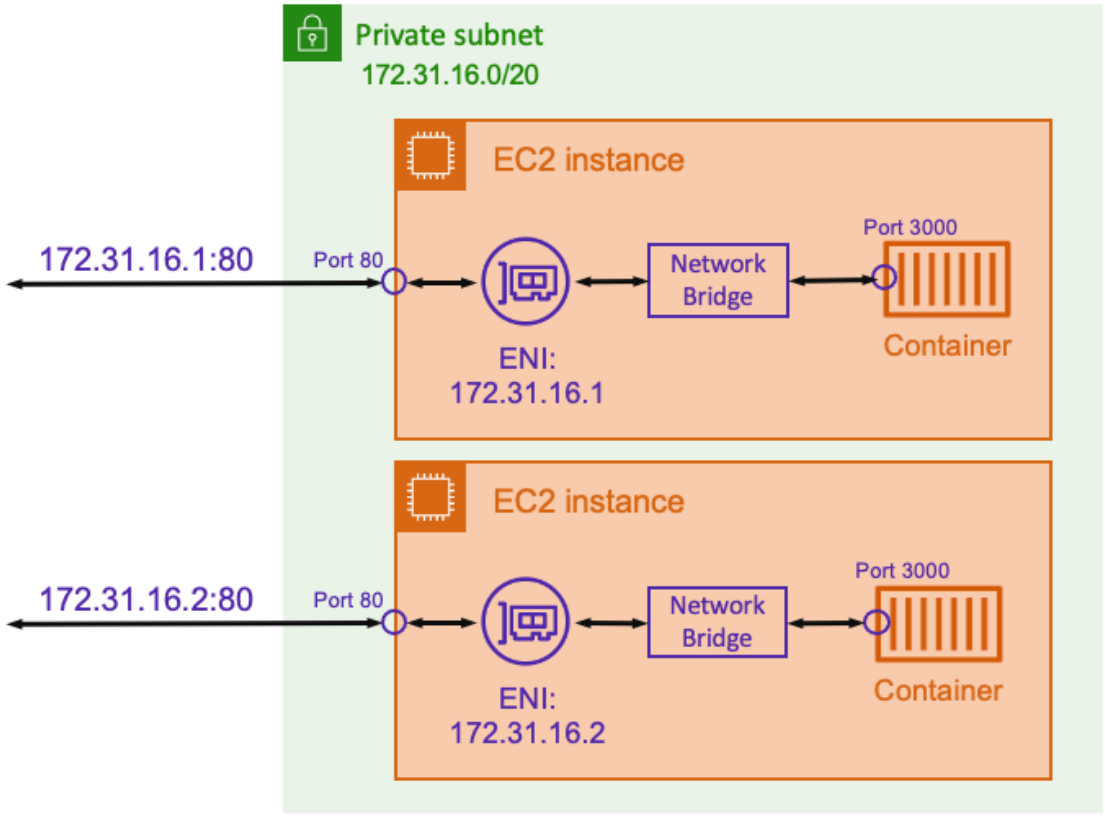
While this approach is simple, there are significant drawbacks to using this network mode. You can't run more than a single instantiation of a task on each host, as only the first task will be able to bind to its required port on the Amazon EC2 instance. Also there is no way to remap a container port when using `host` network mode. If the application the container is running wants to listen on a particular port number, you can't remap that port number. As a result, any port conflicts must be managed by changing the configuration of the application.

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 is not supported when using Amazon ECS on Fargate.

Bridge mode

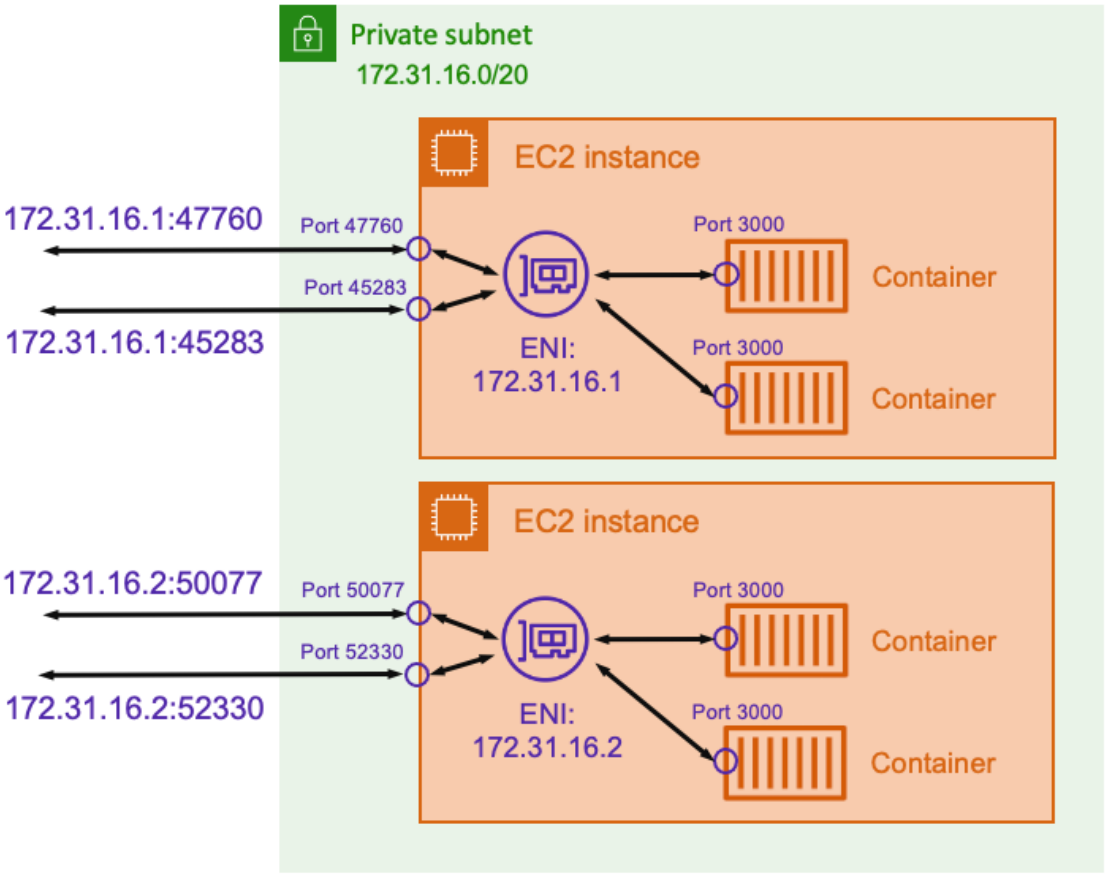
Using `bridge` mode, you are using a virtual network bridge to create a layer between the host and the container's networking. This allows you to create port mappings that remap a host port to a container port. The mappings can be either static or dynamic.



With a static port mapping, you 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`.

Static port mappings work well if you just want to change the traffic port, but they still have the same downside that using the `host` network mode does. You can't run more than a single instantiation of a task on each host, as a static port mapping only allows a single container to be mapped to port 80.

To solve this problem, many people use 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, we can have Docker choose a random, unused port from the ephemeral port range and assign that 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 like `47760` on the Amazon EC2 host. This allows you to run multiple copies of that container on the host and each container will be assigned its own port on the host. From the containerized application perspective each copy of the container is receiving traffic on port `3000` as usual, but clients that are sending traffic to these containers are using the randomly assigned host ports.

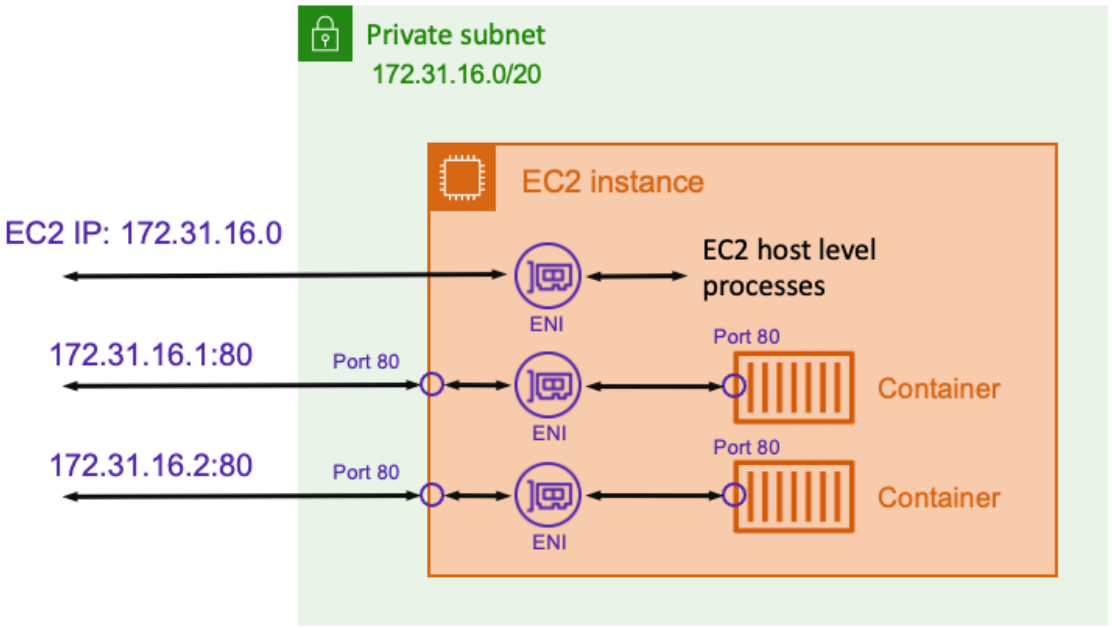
Amazon ECS helps you to keep track of the randomly assigned ports for each task, by automatically updating load balancer target groups and AWS Cloud Map service discovery to have the list of task IP addresses and ports. This makes it easier to use services operating using `bridge` mode with dynamic ports.

However, one downside of using the `bridge` network mode is that it's much harder to lock down service to service communications. Because services may be assigned to any random port, it is necessary to open broad port ranges between hosts and there is no easy way to create specific rules so that a particular service can only talk 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 example above, the Amazon EC2 instance has its own ENI, which represents that EC2 instance's IP address that is used for network communications at the host level. Each task has its own ENI as well, and its own private IP address. Because each ENI is separate, each container can bind to port `80` on the task ENI. Rather than having to keep track of port numbers, you can send traffic to port `80` at the IP address of the task ENI.

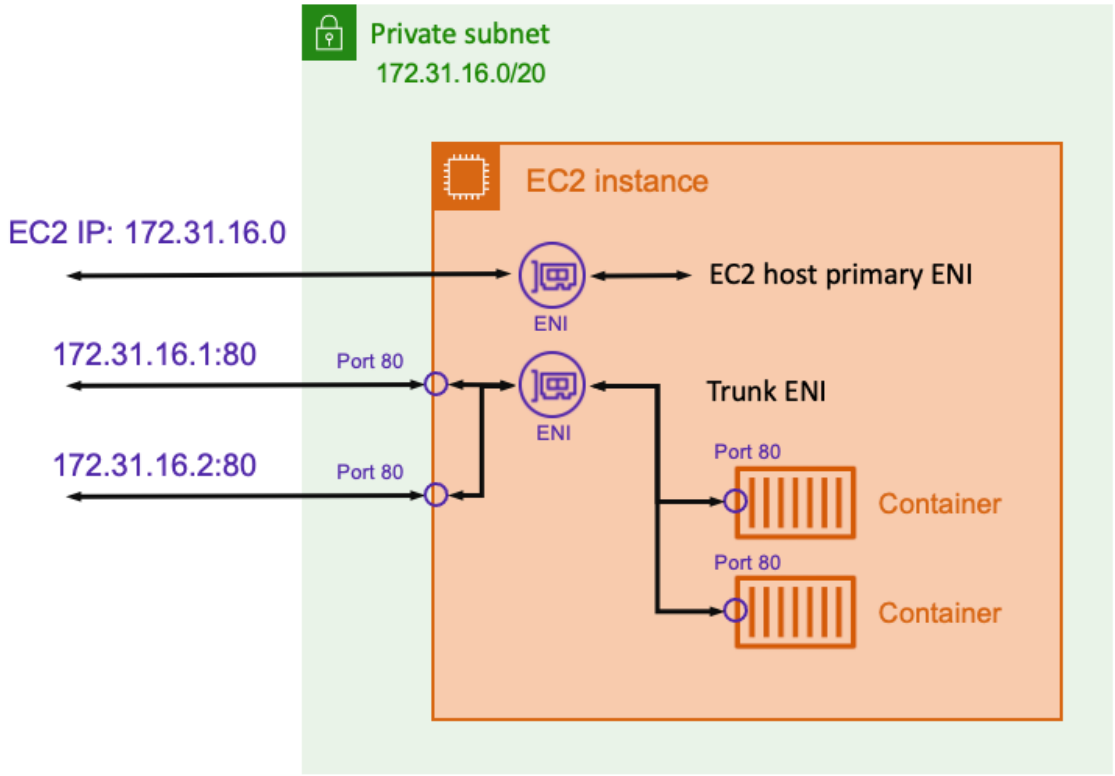
The advantage of using the `awsvpc` network mode is that each task can now have its own security group to allow or deny traffic. This gives you the ability to control communication between tasks and services at a more granular level. You can even configure a task to deny incoming traffic from another task that is located on the same host.

The `awsvpc` network mode is supported for Amazon ECS tasks hosted on both Amazon EC2 and Fargate. 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 challenge when using the `awsvpc` network mode with tasks hosted on Amazon EC2 instances is that EC2 instances have a limit on the number of ENI's that can be attached to them. This imposes a limit on 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 consumed 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.

As an example, without ENI trunking a `c5.large` instance that has 2 vCPU's can only host 2 tasks. With ENI trunking, a `c5.large` instance that has 2 vCPU's can host 10 tasks, each with its own IP address and security group. For more information on the 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, but it does increase task startup time. If ENI trunking is used, you should ensure that your autoscaling rules and other workloads that depend on task startup time still behave as you expect.

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

Preventing IP address exhaustion

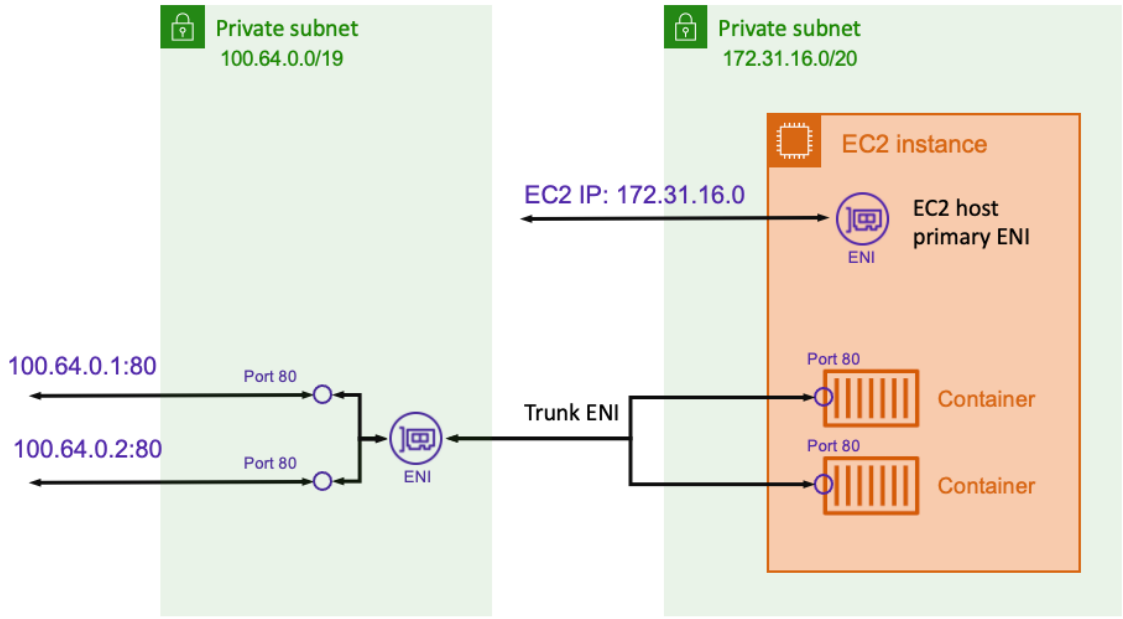
Each task having its own IP address makes your infrastructure easier to understand and allows you to make more secure security groups. However this can lead to IP exhaustion quickly.

The default VPC on your AWS account has pre-provisioned subnets that have a `/20` CIDR range, which means each subnet has 4,091 available IP addresses. (Several IP addresses within the `/20` range are reserved for AWS specific usage.) So if you are distributing your applications across three subnets in three Availability Zones for high availability you would have around 12,000 available IP addresses across those three subnets.

Using ENI trunking, each Amazon EC2 instance you launch requires two IP addresses: one for the primary ENI and another for the trunk ENI. Additionally, each Amazon ECS task on the instance requires an IP

address. If you are launching an extremely large workload you may run out of IP addresses at some point. This would show up as Amazon EC2 launch or task launch failures, as the ENI's would be unable to get IP addresses inside the VPC.

When using the `awsvpc` network mode, you should evaluate your IP address needs and ensure that your subnet CIDR ranges meet your needs. However, if you start with a VPC that has small subnets and begin to run out of address space you can add a secondary subnet to give yourself more room.



Using ENI trunking, the Amazon VPC CNI is configured to use ENIs in a different IP address space than the host. This allows you to give your Amazon EC2 host and your tasks different IP address ranges that don't overlap. For example, in the diagram above, the EC2 host IP address is in subnet that has the `172.31.16.0/20` IP range. The tasks that are running on the host are actually being assigned IP addresses from the `100.64.0.0/19` range. With two independent IP ranges you don't have to worry about tasks consuming too many IP addresses and not leaving enough IP addresses for instances, or vice versa.

Using IPv6 dual stack mode

The `awsvpc` network mode is compatible with VPCs 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 as well as an IPv6 CIDR range. For more information, see IP addressing in your VPC in the *Amazon VPC User Guide*.

You cannot disable IPv4 support for your VPC and subnets, thus it will not address IPv4 exhaustion issues. However, the IPv6 support does allow you to use some new capabilities, in particular 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 does not allow inbound connections from the public internet. For more information, see Egress-only internet gateways in the *Amazon VPC User Guide*.

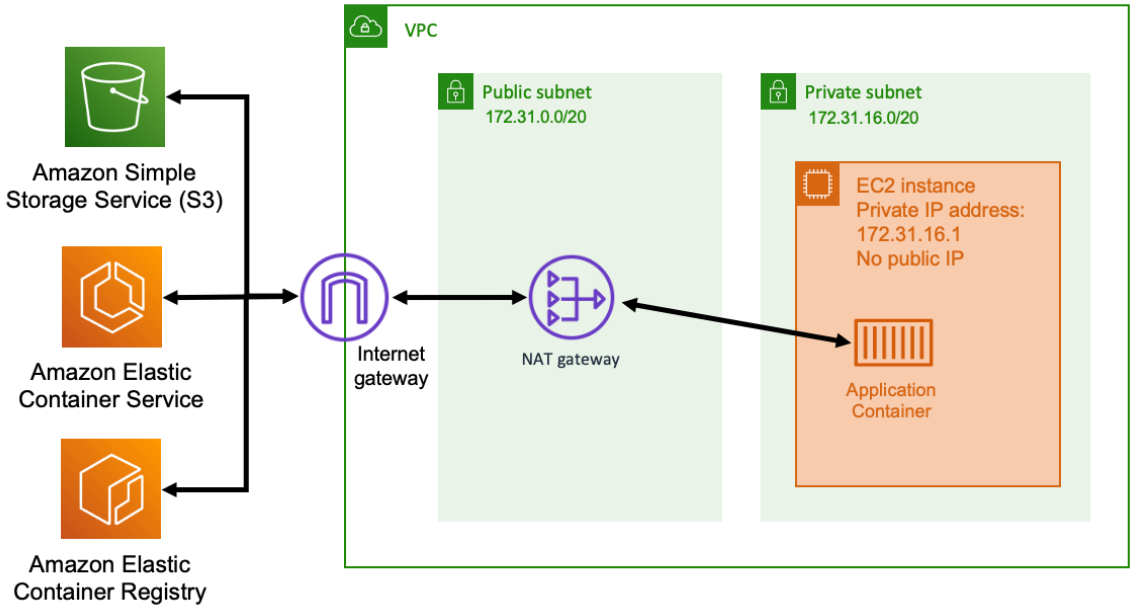
Connecting to AWS services from inside your VPC

In order for Amazon ECS to function, the ECS container agent that runs on each host needs to be able to communicate with the Amazon ECS control plane. If you are storing your container images in Amazon ECR, the Amazon EC2 hosts need to be able to communicate to the Amazon ECR service endpoint, as well as to Amazon S3, where the image layers are stored. Additionally, you may be using other AWS services for your containerized application. For example, you might be persisting information in DynamoDB. Therefore, it is important to ensure that you have networking to the AWS services that you depend on.

Topics

NAT gateway

The NAT gateway networking approach discussed earlier is the easiest way to ensure that your Amazon ECS tasks are able to access other AWS services.



The following are the downsides to using this approach:

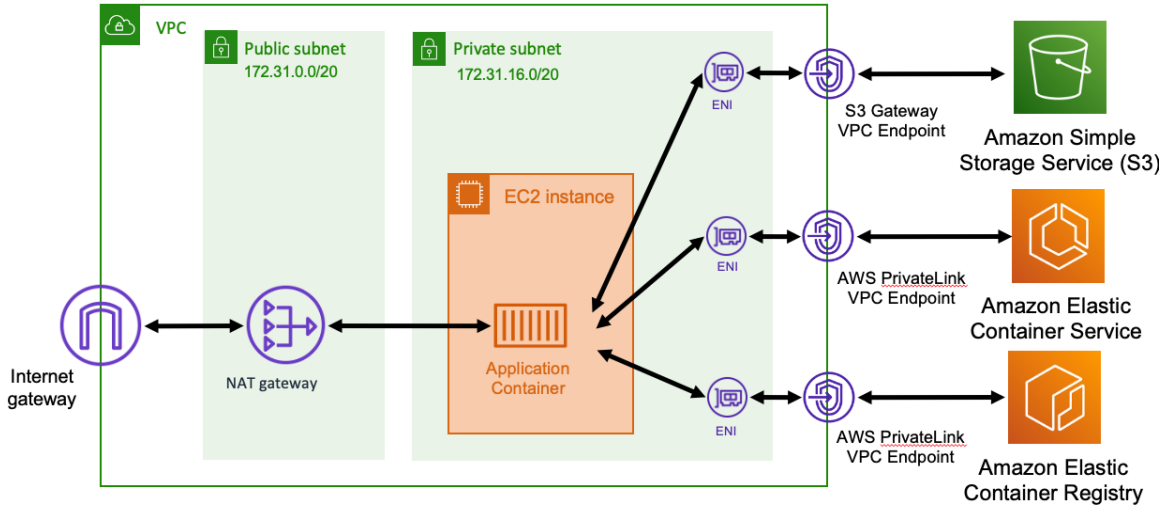
- It isn't possible to limit what destinations the NAT gateway can communicate with. If you want to limit what destinations your backend tier is able to communicate out to, for example to make data exfiltration harder if an attacker manages to gain access, A NAT gateway can't achieve that as it will allow outbound communications to anything outside your VPC.
- NAT gateways charge per 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 will be paying for every GB of bandwidth. Additionally, NAT gateways support 5 Gbps of bandwidth and automatically scale up to 45 Gbps. Therefore applications that require very high bandwidth connections may see networking constraints from trying to route all traffic through a single NAT gateway. To work around this, 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 does not 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 demonstrates 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

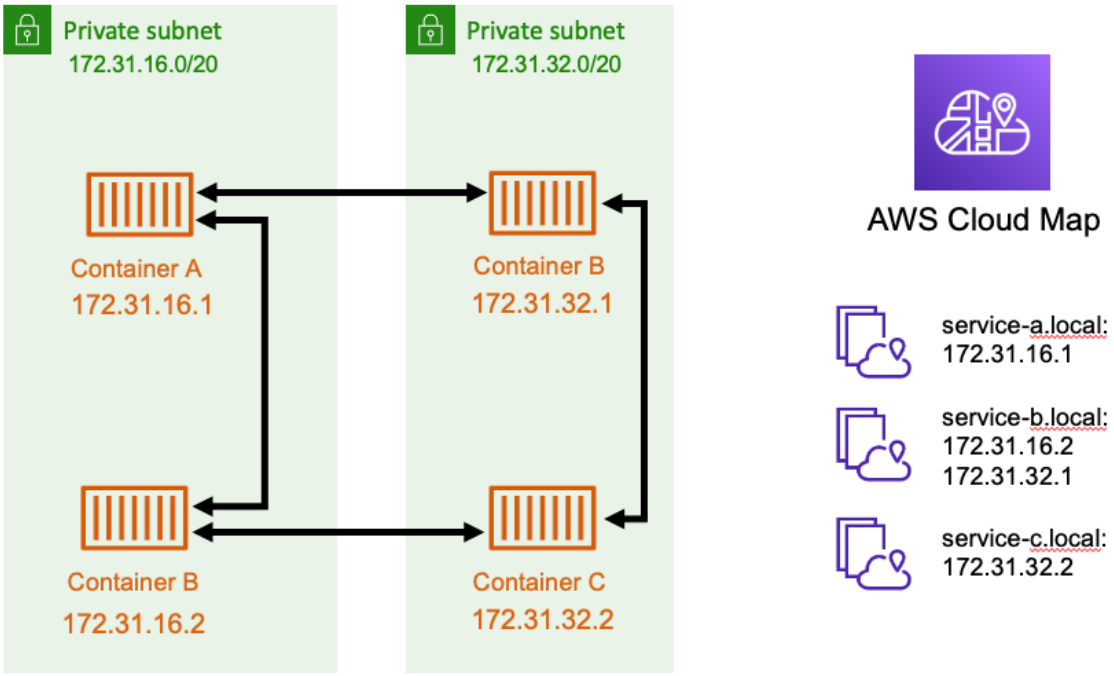
Applications often start out as monolithic codebases, but over time various business functions need to be split out into their own deployments that can be deployed or scaled independently. At that point

you need your Amazon ECS services to be able to network with each other. For example, you might have a customer facing API service that makes use of an internal user service that stores your customers personal identifying information in a compliant manner. You would need to ensure that the API service can locate and communicate with the user service.

There are multiple ways to solve this problem, depending on how complex your architecture is and how many services you have.

Using service discovery

One method 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 diagram above there are three services. `serviceA` has one container and needs to talk to `serviceB` that has two containers. `serviceB` needs to talk to `serviceC` which has one container. Each container in these services are able to 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 for service to service communication has low latency, and at first glance it is also simple as there are no extra components between the containers. Traffic just travels directly from one container to the other container.

This approach works best 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 would be an `A` record. If you are using `bridge` network mode, multiple containers may 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 would no longer be enough for service discovery. You would need to

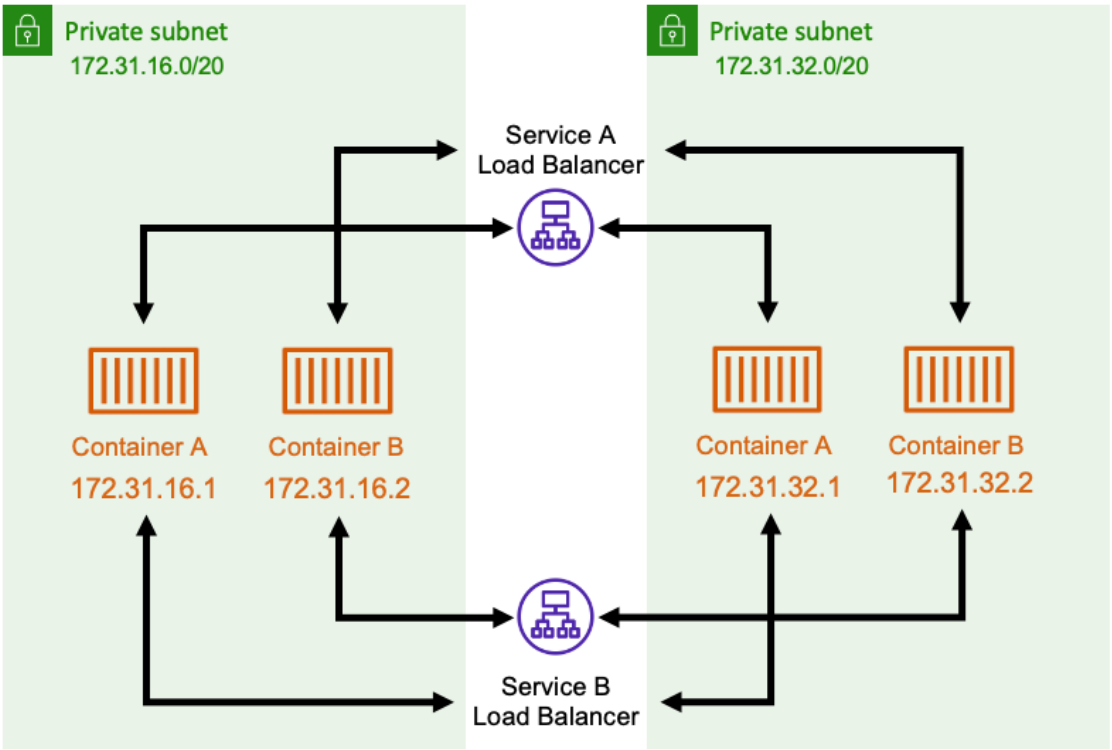
use an `SRV` record. This type of record can keep track of both IP addresses and port numbers, but will usually require some extra effort for your application to make use of. Some prebuilt applications that you use may not support `SRV` records.

Additionally, the `awsvpc` network mode allows you to have a unique security group for each service. This security group can be configured to allow incoming connections from only the specific upstream services that need to talk to that service.

The main downside of direct service to service communication using service discovery is that you will need to 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 may end up resolving the DNS record to point at another container that is no longer there. Your application needs to handle retries and have logic to ignore bad backends.

Using an internal load balancer

Another method for service to service communication is to use an internal load balancer. An internal load balancer lives 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` wants to talk to 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 is able to 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.

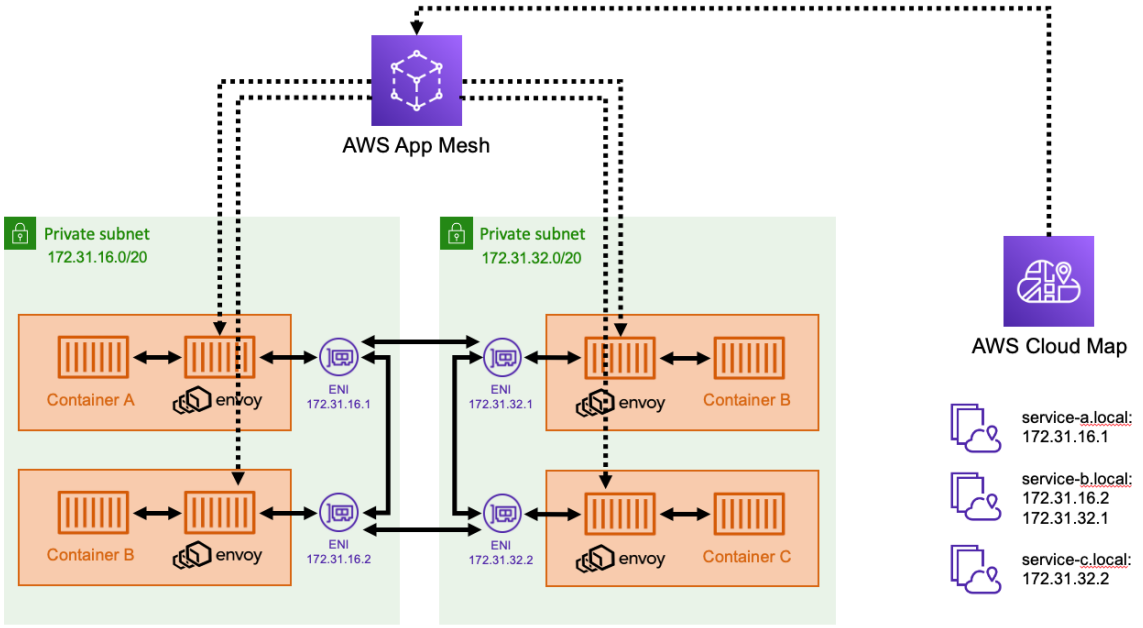
This approach works well with all network modes. The load balancer is capable of keeping track of task IP addresses when using the `awsvpc` network mode, as well as more advanced combinations of IP address and port when using the `bridge` network mode. It will evenly distribute traffic across all the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

The one downside of using an internal load balancer is cost. In order 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.

You can reduce the cost overhead by having multiple services share a load balancer. This works especially well with REST style 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 go to a container that is part of the `user` service, while `/api/order/*` might go to the associated `order` service. With this approach, you only pay for one Application Load Balancer, and have one consistent URL for your API, but you can split the traffic off to various microservices on the backend.

Using a service mesh

AWS App Mesh is a service mesh, which helps you when you have a large number of services and want more control over how traffic gets routed between multiple services. App Mesh could be considered a halfway point between basic service discovery, and load balancing. Applications no longer talk directly to each other, but they also don't use a centralized load balancer. 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 diagram above, 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 will also do its own healthchecks against each target to ensure that they are available.

This approach combines the power of service discovery, with the ease of the managed load balancer. Applications do not have to implement as much load balancing logic within their application code,

because the Envoy proxy sidecar handles that load balancing. The Envoy proxy can be configured to detect failures and retry failed requests. Additionally, the Envoy proxy can be configured to use mTLS to encrypt traffic in transit, as well as ensure that your applications are communicating to a verified destination.

There are a few differences between an Envoy proxy and a load balancer like an Application Load Balancer and Network Load Balancer. Rather than paying for a fully managed load balancer, you are responsible for deploying and managing your own Envoy proxy sidecar which uses some of the CPU and memory that you allocate to the Amazon ECS task. This adds a little bit of extra overhead to the task resource consumption, as well as additional operational workload to maintain and update the proxy when needed.

App Mesh and an Envoy proxy enables extremely low latency between tasks because the Envoy proxy runs collocated to each task. There is only one instance to instance network jump, between one Envoy proxy and another Envoy proxy. This means there is less network overhead compared to the external load balancer approach, where there are two network jumps: from the upstream task to the load balancer, then from the load balancer to the downstream task.

Networking services across AWS accounts and VPCs

Large organizations generally have multiple teams and divisions that deploy services independently into separate VPCs inside a shared AWS account, or their own AWS accounts with their own VPCs. In these scenarios, you need some extra networking components to help you route traffic between these separate VPCs. There are multiple AWS services that can help provide a solution.

- AWS Transit Gateway is the first place to start. It 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 + VPN support helps you to create site to site VPN connections, for connecting on-premise 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 peering helps you to connect multiple VPCs, either in the same account, or cross account. For more information, see What is VPC peering? in the *Amazon VPC Peering Guide*.
- Shared VPCs allow you to use a VPC and VPC subnets across multiple AWS accounts. For more information, see Working with shared VPCs in the *Amazon VPC User Guide*.

Refer to the documentation for the services listed above for guidance on setting up cross-account networking.

Optimizing and troubleshooting

When developing an architecture that has significant networking needs it can be hard to troubleshoot networking issues, and figure out what to optimize. Let's look at some of the tools that are available for understanding your networking and then some configurations and general tips for optimizing your networking.

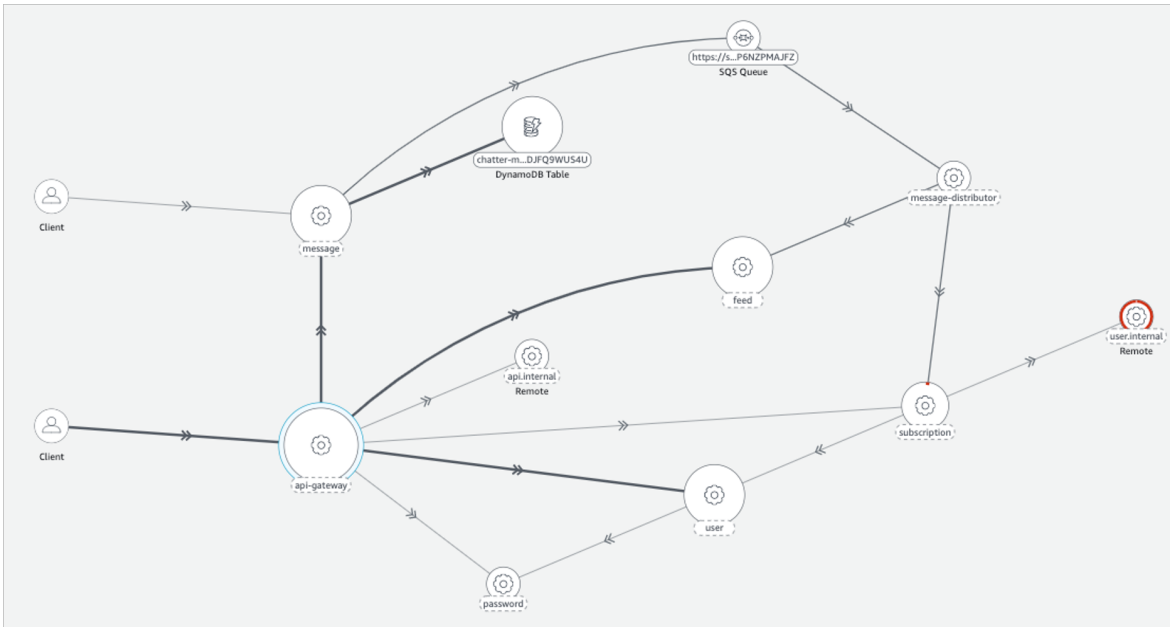
CloudWatch Container Insights

CloudWatch Container Insights collects, aggregates, and summarizes metrics and logs from your containerized applications and microservices. The metrics include utilization for resources such as CPU,

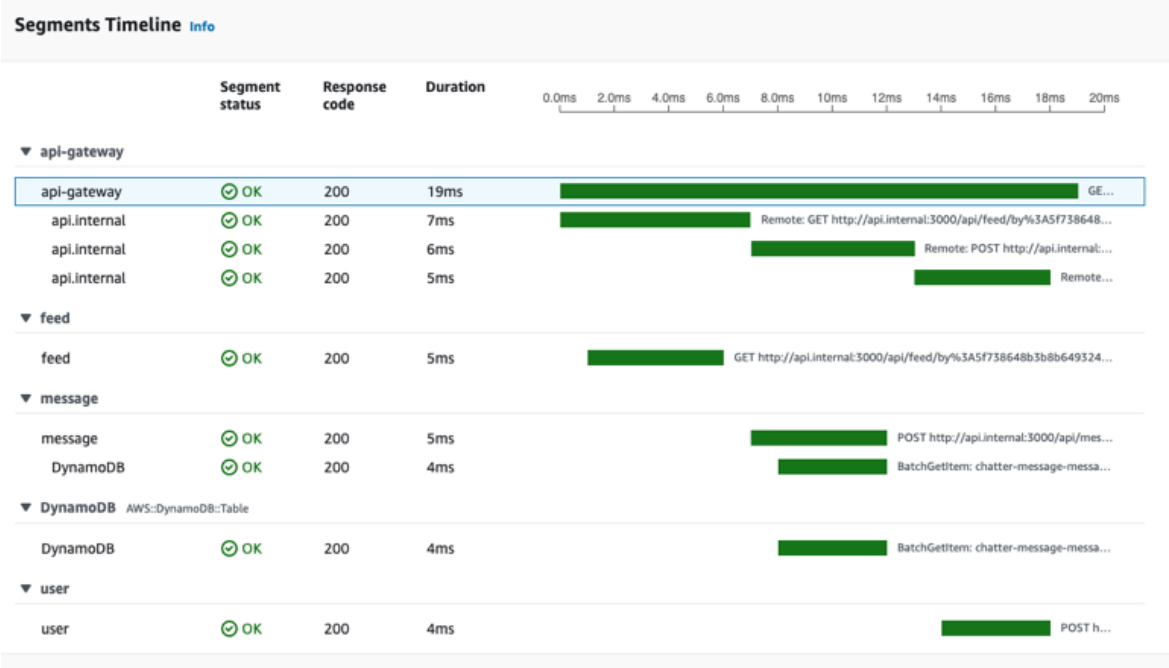
memory, disk, and network. The metrics are available in CloudWatch automatic dashboards. For more information, see Setting up Container Insights on Amazon ECS in the *Amazon CloudWatch User Guide*.

AWS X-Ray

AWS X-Ray is a tracing service that helps you 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 own 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 graphs of how your services network with each other, and explore aggregate stats about how each service to service link is performing. You can also dive deeper into a specific transaction to see segments representing network calls associated with that particular transaction.



These tools can make it much easier to identify when there is a networking bottleneck, or when a specific service within your network is not performing as expected.

VPC Flow Logs

Amazon VPC flow logs are another powerful tool that helps you to analyze your network performance and debug connectivity issues. With VPC flow logs enabled, you can capture a log of all the connections in your VPC, including connections to networking interfaces associated with Elastic Load Balancing, Amazon RDS, NAT gateways, and other key AWS services 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, then you will need to consider the system limit on number of files. When there are a lot of network sockets open each one needs to be represented by a file descriptor. If your file descriptor limit is too low then it will have the unintended effect of limiting your network sockets, resulting in failed connections or errors. You can update the container specific ulimit on number of files in the Amazon ECS task definition. If running on Amazon EC2 instead of AWS Fargate then you may also need to adjust these limits on your underlying Amazon EC2 instance as well.

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, but many of these settings adjust the size of read and write buffers which can help in some situations when running really large Amazon EC2 instances that have a lot of containers on them.

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
Initial release (p. 23)	Initial release of the Amazon ECS Best Practices Guide	April 6, 2021