aws

User Guide

Amazon ECR

API Version 2015-09-21

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon ECR: User Guide

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is Amazon ECR 1
Concepts and components 1
Common use cases 3
Features of Amazon ECR 5
How to get started with Amazon ECR 6
Pricing for Amazon ECR 6
Moving an image through its lifecycle 7
Prerequisites 7
Install the AWS CLI 7
Install Docker 7
Step 1: Create a Docker image 9
Step 2: Create a repository 11
Step 3: Authenticate to your default registry 11
Step 4: Push an image to Amazon ECR 12
Step 5: Pull an image from Amazon ECR 13
Step 6: Delete an image 14
Step 7: Delete a repository 14
Optimizing performance 15
Making requests 17
Getting started with IPv6 17
Testing IP address compatibility 18
Making requests using dual-stack endpoints 19
Using Amazon ECR endpoints from the docker CLI 19
Using IPv6 addresses in IAM policies 20
Private registry 22
Registry concepts 22
Registry authentication 22
Using the Amazon ECR credential helper 23
Using an authorization token 23
Using HTTP API authentication 24
Registry settings 25
Registry permissions 26
Registry policy examples 27
Switching to the extended registry policy scope 30

Granting permissions for cross account replication ... 32
Granting permissions for pull through cache ... 33
Private repositories ... 35
Repository concepts ... 35
Creating a repository to store images ... 36
Next steps ... 37
Viewing repository details ... 38
Deleting a repository ... 39
Repository policies ... 39
Repository policies vs IAM policies ... 40
Repository policy examples ... 41
Setting a repository policy statement ... 44
Tagging a repository ... 46
Tag basics ... 46
Tagging your resources for billing ... 46
Adding tags ... 47
Deleting tags ... 48
Private images ... 50
Pushing an image ... 51
Required IAM permissions ... 51
Pushing a Docker image ... 52
Pushing a multi-architecture image ... 54
Pushing a Helm chart ... 55
Signing an image ... 58
Considerations ... 58
Prerequisites ... 58
Configure authentication for the Notary client ... 58
Signing an image ... 59
Next steps ... 60
Deleting artifacts ... 60
Viewing image details ... 63
Pulling an image ... 64
Pulling the Amazon Linux container image ... 65
Deleting an image ... 67
Retagging an image ... 68
Preventing image tags from being overwritten ... 71

Setting image tag mutability (AWS Management Console) ... 71
Setting image tag mutability (AWS CLI) ... 72
Container image manifest formats ... 73
Amazon ECR image manifest conversion ... 74
Using Amazon ECR images with Amazon ECS ... 75
Required IAM permissions ... 75
Specifying an Amazon ECR image in a task definition ... 77
Using Amazon ECR Images with Amazon EKS ... 77
Required IAM permissions ... 78
Installing a Helm chart on an Amazon EKS cluster ... 78
Scan images for vulnerabilities ... 81
Filters for repositories ... 82
Filter wildcards ... 82
Enhanced scanning ... 83
Considerations for enhanced scanning ... 83
Changing the enhanced scanning duration ... 84
Required IAM permissions ... 85
Configuring enhanced scanning ... 86
EventBridge events ... 88
Retrieving findings ... 93
Basic scanning ... 94
Operating system support for basic scanning and improved basic scanning ... 95
Configuring basic scanning ... 98
Switching to the improved basic scanning ... 98
Manually scanning an image ... 100
Retrieving findings ... 101
Troubleshooting image scanning ... 103
Understanding scan status `SCAN_ELIGIBILITY_EXPIRED` ... 103
Sync an upstream registry ... 105
Repository creation templates ... 106
Considerations for using pull through cache rules ... 106
Required IAM permissions ... 108
Using registry permissions ... 108
Next steps ... 110
Setting up permissions for cross-account ECR to ECR PTC ... 111
IAM policies required for cross-account ECR to ECR pull through cache ... 111

Creating a pull through cache rule ... 113
Prerequisites ... 113
Using the AWS Management Console ... 114
Using the AWS CLI ... 121
Next steps ... 124
Validating pull through cache rule ... 125
Pulling an image with a pull through cache rule ... 126
Storing your upstream repository credentials ... 128
Customizing repository prefixes ... 134
Troubleshooting pull through cache issues ... 135
Replicate images ... 138
Considerations for private image replication ... 138
Replication examples ... 139
Example: Configuring cross-Region replication to a single destination Region ... 140
Example: Configuring cross-Region replication using a repository filter ... 140
Example: Configuring cross-Region replication to multiple destination Regions ... 141
Example: Configuring cross-account replication ... 141
Example: Specifying multiple rules in a configuration ... 142
Configuring replication ... 143
Repository creation templates ... 145
How it works ... 145
Creating a repository creation template ... 149
IAM permissions for creating repository creation templates ... 149
Create a custom policy ... 150
Create an IAM role ... 151
Create a repository creation template ... 152
Updating repository creation templates ... 157
Deleting a repository creation template ... 158
Automate the cleanup of images ... 160
How lifecycle policies work ... 160
Lifecycle policy evaluation rules ... 161
Creating a lifecycle policy preview ... 162
Creating a lifecycle policy ... 164
Prerequisite ... 164
Examples of lifecycle policies ... 166
Lifecycle policy template ... 166

Filtering on image age ... 167
Filtering on image count ... 167
Filtering on multiple rules ... 168
Filtering on multiple tags in a single rule ... 171
Filtering on all images ... 173
Lifecycle policy properties ... 175
Rule priority ... 175
Description ... 176
Tag status ... 176
Tag pattern list ... 176
Tag prefix list ... 177
Count type ... 177
Count unit ... 178
Count number ... 178
Action ... 178
Security ... 179
Identity and Access Management ... 179
Audience ... 180
Authenticating with identities ... 181
Managing access using policies ... 184
How Amazon Elastic Container Registry works with IAM ... 186
Identity-based policy examples ... 191
Using Tag-Based Access Control ... 195
AWS managed policies for Amazon ECR ... 197
Using service-linked roles ... 207
Troubleshooting ... 216
Data protection ... 218
Encryption at rest ... 219
Compliance validation ... 226
Infrastructure Security ... 227
Interface VPC Endpoints (AWS PrivateLink) ... 228
Cross-service confused deputy prevention ... 236
Monitoring ... 239
Visualizing Your Service Quotas and Setting Alarms ... 240
Usage Metrics ... 241
Usage Reports ... 242

Repository metrics 243
Enabling CloudWatch metrics 243
Available metrics and dimensions 243
Viewing metrics with CloudWatch 244
Events and EventBridge 244
Sample events from Amazon ECR 245
..... 248
Logging Actions with AWS CloudTrail 249
Amazon ECR information in CloudTrail 250
Understanding Amazon ECR log file entries 251
Working with AWS SDKs 264
Code examples 266
Basics 271
Hello Amazon ECR 271
Learn the basics 275
Actions 331
Service quotas 382
Managing your Amazon ECR service quotas in the AWS Management Console 387
Creating a CloudWatch alarm to monitor API usage metrics 388
Troubleshooting 389
Troubleshooting Docker 389
Docker logs do not contain expected error messages 389
Error: "Filesystem Verification Failed" or "404: Image Not Found" when pulling an image from an Amazon ECR repository 390
Error: "Filesystem Layer Verification Failed" when pulling images from Amazon ECR 390
HTTP 403 Errors or "no basic auth credentials" error when pushing to repository 391
Troubleshooting Amazon ECR error messages 392
HTTP 429: Too Many Requests or ThrottleException 392
HTTP 403: "User [arn] is not authorized to perform [operation]" 393
HTTP 404: "Repository Does Not Exist" error 393
Error: Cannot perform an interactive login from a non TTY device 393
Using Podman with Amazon ECR 395
Using Podman to authenticate with Amazon ECR 395
Using the Amazon ECR credential helper with Podman 395
Pulling images from Amazon ECR with Podman 396
Running containers for Amazon ECR with Podman 396

Pushing images to Amazon ECR with Podman 396
Document history 398

What is Amazon Elastic Container Registry?

Amazon Elastic Container Registry (Amazon ECR) is an AWS managed container image registry service that is secure, scalable, and reliable. Amazon ECR supports private repositories with resource-based permissions using AWS IAM. This is so that specified users or Amazon EC2 instances can access your container repositories and images. You can use your preferred CLI to push, pull, and manage Docker images, Open Container Initiative (OCI) images, and OCI compatible artifacts.

> **Note**
>
> Amazon ECR supports public container image repositories as well. For more information, see What is Amazon ECR Public in the *Amazon ECR Public User Guide*.

The AWS container services team maintains a public roadmap on GitHub. It contains information about what the teams are working on and allows all AWS customers the ability to give direct feedback. For more information, see AWS Containers Roadmap.

Concepts and components of Amazon ECR

Amazon ECR is a fully managed Docker container registry service provided by AWS. It allows you to store, manage, and deploy Docker container images securely and reliably. These concepts and components work together to provide a secure, scalable, and reliable Docker container registry service within the AWS, enabling you to efficiently manage and deploy your containerized applications.

Here are some key concepts and components of Amazon ECR:

Registry

An Amazon ECR registry is a private repository provided to each AWS account, where you can create one or more repositories. These repositories allow you to store and distribute Docker images, Open Container Initiative (OCI) images, and other OCI-compatible artifacts within your AWS environment. For more information, see Amazon ECR private registry.

Authorization token

Your client must authenticate to an Amazon ECR private registry as an AWS user before it can push and pull images. For more information, see Private registry authentication in Amazon ECR.

Repository

A repository in Amazon ECR is a logical collection where you can store your Docker images, Open Container Initiative (OCI) images, and other OCI-compatible artifacts. Within a single Amazon ECR registry, you can have multiple repositories to organize your container images. For more information, see Amazon ECR private repositories.

Repository policy

You can control access to your repositories and the contents within them with repository policies. For more information, see Private repository policies in Amazon ECR.

Image

You can push and pull container images to your repositories. You can use these images locally on your development system, or you can use them in Amazon ECS task definitions and Amazon EKS pod specifications. For more information, see Using Amazon ECR images with Amazon ECS and Using Amazon ECR Images with Amazon EKS.

Lifecycle Policy

Amazon ECR lifecycle policies allow you to manage the lifecycle of your images by defining rules for pruning and expiring old or unused images. For more information, see Automate the cleanup of images by using lifecycle policies in Amazon ECR.

Image Scanning

Amazon ECR provides an integrated image scanning capability that helps identify software vulnerabilities in your container images. For more information, see Scan images for software vulnerabilities in Amazon ECR.

Access Control

Amazon ECR uses IAM to control access to your repositories. You can create IAM users, groups, and roles with specific permissions to push, pull, or manage Amazon ECR repositories. For more information, see Security in Amazon Elastic Container Registry.

Cross-account and Cross-region Replication

Amazon ECR supports replicating images across multiple AWS accounts and regions for increased availability and reduced latency. For more information, see Private image replication in Amazon ECR.

Encryption

Amazon ECR supports server-side encryption of your Docker images at rest using AWS KMS. For more information, see Data protection in Amazon ECR.

AWS Command Line Interface Integration

The AWS CLI provides commands to interact with Amazon ECR repositories, such as creating, listing, pushing, and pulling images.

AWS Management Console

Amazon ECR can also be managed through the AWS Management Console, providing a user-friendly web interface for working with your repositories and images.

AWS CloudTrail

Amazon ECR integrates with AWS CloudTrail, allowing you to log and audit API calls made to Amazon ECR for security and compliance purposes. For more information, see Logging Amazon ECR actions with AWS CloudTrail.

Amazon CloudWatch

Amazon ECR provides metrics and logs that can be monitored using Amazon CloudWatch, enabling you to track the performance and usage of your Amazon ECR repositories. For more information, see Amazon ECR repository metrics.

Common use cases in Amazon ECR

Amazon ECR is a fully-managed Docker container registry service offered by AWS. It provides a secure and scalable repository for storing and distributing Docker container images, making it an essential component in containerized application deployments. Amazon ECR simplifies the process of building, distributing, and running containerized applications across various AWS services and on-premises environments.

Here are some key use cases for Amazon ECR:

Container Image Storage and Distribution

Amazon ECR serves as a centralized repository for storing and distributing Docker container images within an organization or for public consumption. Developers can push their container images to Amazon ECR and then pull them from any compute environment within AWS, such

as Amazon EC2, AWS Fargate, or Amazon EKS. For more information, see Amazon ECR private repositories.

Continuous Integration and Continuous Deployment (CI/CD)

Amazon ECR integrates seamlessly with AWS CodeBuild, AWS CodePipeline, and other CI/CD tools, enabling automated building, testing, and deployment of containerized applications. Container images can be automatically pushed to Amazon ECR as part of the CI/CD pipeline, ensuring consistent and reliable deployment across different environments.

Microservices Architecture

Amazon ECR is well suited for microservices architectures, where applications are broken down into smaller, decoupled services packaged as containers. Each microservice can have its own container image stored in Amazon ECR, enabling independent development, deployment, and scaling of individual services.

Hybrid and Multi-Cloud Deployments

Amazon ECR supports the ability to pull container images from other container registries, such as Docker Hub or third party registries. This allows organizations to maintain a consistent deployment model across hybrid or multi-cloud environments, using Amazon ECR as the central repository for container images.

Access Control and Security

Amazon ECR provides fine-grained access control mechanisms, allowing organizations to control who can push or pull container images from the registry. It also integrates with AWS Identity and Access Management for authentication and authorization, ensuring secure access to container images. For more information, see Security in Amazon Elastic Container Registry.

Image Vulnerability Scanning

Amazon ECR offers automatic scanning of container images for software vulnerabilities and potential misconfiguration, helping to maintain a secure and compliant container environment. For more information, see Scan images for software vulnerabilities in Amazon ECR.

Private Container Registry

For organizations with strict security or compliance requirements, Amazon ECR can be used as a private container registry, ensuring that sensitive container images are not exposed to public registries and are accessible only within the organization's AWS environment. For more information, see Amazon ECR private registry.

Globally Distributed Application Deployment with Amazon ECR Replication

Leveraging Amazon ECR replication capability, you can centralize your containerized web application images in a primary repository, enabling automated distribution across multiple AWS regions, ensuring consistent global deployments with low latency worldwide and reducing operational burden. For more information, see Private image replication in Amazon ECR

Automated Cleanup of Stale Container Images

Amazon ECR lifecycle policies enable automated cleanup of stale container images based on defined rules such as age, count, or tags, optimizing storage costs, maintaining an organized registry, enhancing security and compliance, and streamlining development workflows through automation. For more information, see Automate the cleanup of images by using lifecycle policies in Amazon ECR

Features of Amazon ECR

Amazon ECR provides the following features:

- Lifecycle policies help with managing the lifecycle of the images in your repositories. You define rules that result in the cleaning up of unused images. You can test rules before applying them to your repository. For more information, see Automate the cleanup of images by using lifecycle policies in Amazon ECR.
- Image scanning helps in identifying software vulnerabilities in your container images. Each repository can be configured to **scan on push**. This ensures that each new image pushed to the repository is scanned. You can then retrieve the results of the image scan. For more information, see Scan images for software vulnerabilities in Amazon ECR.
- Cross-Region and cross-account replication makes it easier for you to have your images where you need them. This is configured as a registry setting and is on a per-Region basis. For more information, see Private registry settings in Amazon ECR.
- Pull through cache rules provide a way to cache repositories in an upstream registry in your private Amazon ECR registry. Using a pull through cache rule, Amazon ECR will periodically reach out to the upstream registry to ensure the cached image in your Amazon ECR private registry is up to date. For more information, see Sync an upstream registry with an Amazon ECR private registry.

How to get started with Amazon ECR

If you are using Amazon Elastic Container Service (Amazon ECS) or Amazon Elastic Kubernetes Service (Amazon EKS), note that the setup for those two services is similar to the setup for Amazon ECR because Amazon ECR is an extension of both services.

When using the AWS Command Line Interface with Amazon ECR, use a version of the AWS CLI that supports the latest Amazon ECR features. If you don't see support for an Amazon ECR feature in the AWS CLI, upgrade to the latest version of the AWS CLI. For information about installing the latest version of the AWS CLI, see Install or update to the latest version of the AWS CLI in the *AWS Command Line Interface User Guide*.

To learn how to push a container image to a private Amazon ECR repository using the AWS CLI and Docker, see Moving an image through its lifecycle in Amazon ECR.

Pricing for Amazon ECR

With Amazon ECR, you only pay for the amount of data you store in your repositories and for the data transfer from your image pushes and pulls. For more information, see Amazon ECR pricing.

Moving an image through its lifecycle in Amazon ECR

If you are using Amazon ECR for the first time, use the following steps with the Docker CLI and the AWS CLI to create a sample image, authenticate to the default registry, and create a private repository. Then push an image to and pull an image from the private repository. When you are finished with the sample image, delete the sample image and the repository.

To use the AWS Management Console instead of the AWS CLI, see the section called "Creating a repository to store images".

For more information on the other tools available for managing your AWS resources, including the different AWS SDKs, IDE toolkits, and the Windows PowerShell command line tools, see http://aws.amazon.com/tools/.

Prerequisites

If you do not have the latest AWS CLI and Docker installed and ready to use, use the following steps to install both of these tools.

Install the AWS CLI

To use the AWS CLI with Amazon ECR, install the latest AWS CLI version. For information, see Installing the AWS Command Line Interface in the *AWS Command Line Interface User Guide*.

Install Docker

Docker is available on many different operating systems, including most modern Linux distributions, like Ubuntu, and even macOS and Windows. For more information about how to install Docker on your particular operating system, go to the Docker installation guide.

You do not need a local development system to use Docker. If you are using Amazon EC2 already, you can launch an Amazon Linux 2023 instance and install Docker to get started.

If you already have Docker installed, skip to Step 1: Create a Docker image.

To install Docker on an Amazon EC2 instance using an Amazon Linux 2023 AMI

1. Launch an instance with the latest Amazon Linux 2023 AMI. For more information, see Launching an instance in the *Amazon EC2 User Guide*.

2. Connect to your instance. For more information, see Connect to Your Linux Instance in the *Amazon EC2 User Guide*.
3. Update the installed packages and package cache on your instance.

```
sudo yum update -y
```

4. Install the most recent Docker Community Edition package.

```
sudo yum install docker
```

5. Start the Docker service.

```
sudo service docker start
```

6. Add the `ec2-user` to the `docker` group so you can execute Docker commands without using `sudo`.

```
sudo usermod -a -G docker ec2-user
```

7. Log out and log back in again to pick up the new `docker` group permissions. You can accomplish this by closing your current SSH terminal window and reconnecting to your instance in a new one. Your new SSH session will have the appropriate `docker` group permissions.
8. Verify that the `ec2-user` can run Docker commands without `sudo`.

```
docker info
```

> **Note**
>
> In some cases, you may need to reboot your instance to provide permissions for the `ec2-user` to access the Docker daemon. Try rebooting your instance if you see the following error:
>
> ```
> Cannot connect to the Docker daemon. Is the docker daemon running on this
>  host?
> ```

Step 1: Create a Docker image

In this step, you create a Docker image of a simple web application, and test it on your local system or Amazon EC2 instance.

To create a Docker image of a simple web application

1. Create a file called `Dockerfile`. A Dockerfile is a manifest that describes the base image to use for your Docker image and what you want installed and running on it. For more information about Dockerfiles, go to the Dockerfile Reference.

   ```
   touch Dockerfile
   ```

2. Edit the `Dockerfile` you just created and add the following content.

   ```
   FROM public.ecr.aws/amazonlinux/amazonlinux:latest

   # Install dependencies
   RUN yum update -y && \
    yum install -y httpd

   # Install apache and write hello world message
   RUN echo 'Hello World!' > /var/www/html/index.html

   # Configure apache
   RUN echo 'mkdir -p /var/run/httpd' >> /root/run_apache.sh && \
    echo 'mkdir -p /var/lock/httpd' >> /root/run_apache.sh && \
    echo '/usr/sbin/httpd -D FOREGROUND' >> /root/run_apache.sh && \
    chmod 755 /root/run_apache.sh

   EXPOSE 80

   CMD /root/run_apache.sh
   ```

   This Dockerfile uses the public Amazon Linux 2 image hosted on Amazon ECR Public. The `RUN` instructions update the package caches, installs some software packages for the web server, and then write the "Hello World!" content to the web servers document root. The `EXPOSE` instruction exposes port 80 on the container, and the `CMD` instruction starts the web server.

3. Build the Docker image from your Dockerfile.

> **Note**
>
> Some versions of Docker may require the full path to your Dockerfile in the following command, instead of the relative path shown below.

```
docker build -t hello-world .
```

4. List your container image.

```
docker images --filter reference=hello-world
```

Output:

```
REPOSITORY          TAG                 IMAGE ID            CREATED
 SIZE
hello-world         latest              e9ffedc8c286        4 minutes ago
 194MB
```

5. Run the newly built image. The `-p 80:80` option maps the exposed port 80 on the container to port 80 on the host system. For more information about **docker run**, go to the Docker run reference.

```
docker run -t -i -p 80:80 hello-world
```

> **Note**
>
> Output from the Apache web server is displayed in the terminal window. You can ignore the "`Could not reliably determine the fully qualified domain name`" message.

6. Open a browser and point to the server that is running Docker and hosting your container.

   - If you are using an EC2 instance, this is the **Public DNS** value for the server, which is the same address you use to connect to the instance with SSH. Make sure that the security group for your instance allows inbound traffic on port 80.
   - If you are running Docker locally, point your browser to http://localhost/.

- If you are using **docker-machine** on a Windows or Mac computer, find the IP address of the VirtualBox VM that is hosting Docker with the **docker-machine ip** command, substituting *machine-name* with the name of the docker machine you are using.

  ```
  docker-machine ip machine-name
  ```

  You should see a web page with your "Hello World!" statement.

7. Stop the Docker container by typing **Ctrl + c**.

Step 2: Create a repository

Now that you have an image to push to Amazon ECR, you must create a repository to hold it. In this example, you create a repository called `hello-repository` to which you later push the `hello-world:latest` image. To create a repository, run the following command:

```
aws ecr create-repository \
    --repository-name hello-repository \
    --region region
```

Step 3: Authenticate to your default registry

After you have installed and configured the AWS CLI, authenticate the Docker CLI to your default registry. That way, the **docker** command can push and pull images with Amazon ECR. The AWS CLI provides a **get-login-password** command to simplify the authentication process.

To authenticate Docker to an Amazon ECR registry with get-login-password, run the **aws ecr get-login-password** command. When passing the authentication token to the **docker login** command, use the value `AWS` for the username and specify the Amazon ECR registry URI you want to authenticate to. If authenticating to multiple registries, you must repeat the command for each registry.

> **Important**
>
> If you receive an error, install or upgrade to the latest version of the AWS CLI. For more information, see Installing the AWS Command Line Interface in the *AWS Command Line Interface User Guide*.

- get-login-password (AWS CLI)

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

- Get-ECRLoginCommand (AWS Tools for Windows PowerShell)

```
(Get-ECRLoginCommand).Password | docker login --username AWS --password-
stdin aws_account_id.dkr.ecr.region.amazonaws.com
```

Step 4: Push an image to Amazon ECR

Now you can push your image to the Amazon ECR repository you created in the previous section. Use the **docker** CLI to push images after the following prerequisites are met:

- The minimum version of **docker** is installed: 1.7.
- The Amazon ECR authorization token was configured with **docker login**.
- The Amazon ECR repository exists and the user has access to push to the repository.

After those prerequisites are met, you can push your image to your newly created repository in the default registry for your account.

To tag and push an image to Amazon ECR

1. List the images you have stored locally to identify the image to tag and push.

   ```
   docker images
   ```

   Output:

   ```
   REPOSITORY          TAG                 IMAGE ID            CREATED
    VIRTUAL SIZE
   hello-world         latest              e9ffedc8c286        4 minutes ago
    241MB
   ```

2. Tag the image to push to your repository.

```
docker tag hello-world:latest aws_account_id.dkr.ecr.region.amazonaws.com/hello-
repository
```

3. Push the image.

```
docker push aws_account_id.dkr.ecr.region.amazonaws.com/hello-repository:latest
```

Output:

```
The push refers to a repository [aws_account_id.dkr.ecr.region.amazonaws.com/hello-
repository] (len: 1)
e9ae3c220b23: Pushed
a6785352b25c: Pushed
0998bf8fb9e9: Pushed
0a85502c06c9: Pushed
latest: digest: sha256:215d7e4121b30157d8839e81c4e0912606fca105775bb0636EXAMPLE
 size: 6774
```

Step 5: Pull an image from Amazon ECR

After your image is pushed to your Amazon ECR repository, you can pull it from other locations. Use the **docker** CLI to pull images after the following prerequisites are met:

- The minimum version of **docker** is installed: 1.7.
- The Amazon ECR authorization token was configured with **docker login**.
- The Amazon ECR repository exists and the user has access to pull from the repository.

After those prerequisites are met, you can pull your image. To pull your example image from Amazon ECR, run the following command:

```
docker pull aws_account_id.dkr.ecr.region.amazonaws.com/hello-repository:latest
```

Output:

```
latest: Pulling from hello-repository
0a85502c06c9: Pull complete
0998bf8fb9e9: Pull complete
```

```
a6785352b25c: Pull complete
e9ae3c220b23: Pull complete
Digest: sha256:215d7e4121b30157d8839e81c4e0912606fca105775bb0636EXAMPLE
Status: Downloaded newer image for aws_account_id.dkr.region.amazonaws.com/hello-
repository:latest
```

Step 6: Delete an image

If you no longer need an image in one of your repositories, you can delete the image. To delete an image, specify the repository that it's in and either an `imageTag` or `imageDigest` value for the image. The following example deletes an image in the `hello-repository` repository with the image tag `latest`. To delete your example image from the repository, run the following command:

```
aws ecr batch-delete-image \
      --repository-name hello-repository \
      --image-ids imageTag=latest \
      --region region
```

Step 7: Delete a repository

If you no longer need an entire repository of images, you can delete the repository. The following example uses the `--force` flag to delete a repository that contains images. To delete a repository that contains images (and all the images within it), run the following command:

```
aws ecr delete-repository \
      --repository-name hello-repository \
      --force \
      --region region
```

Optimizing performance for Amazon ECR

You can use the following recommendations about settings and strategies to optimize performance when using Amazon ECR.

Use Docker 1.10 and above to take advantage of simultaneous layer uploads

Docker images are composed of layers, which are intermediate build stages of the image. Each line in a Dockerfile results in the creation of a new layer. When you use Docker 1.10 and above, Docker defaults to pushing as many layers as possible as simultaneous uploads to Amazon ECR, resulting in faster upload times.

Use a smaller base image

The default images available through Docker Hub may contain many dependencies that your application doesn't require. Consider using a smaller image created and maintained by others in the Docker community, or build your own base image using Docker's minimal scratch image. For more information, see Create a base image in the Docker documentation.

Place the dependencies that change the least earlier in your Dockerfile

Docker caches layers, and that speeds up build times. If nothing on a layer has changed since the last build, Docker uses the cached version instead of rebuilding the layer. However, each layer is dependent on the layers that came before it. If a layer changes, Docker recompiles not only that layer, but any layers that come after that layer as well.

To minimize the time required to rebuild a Dockerfile and to re-upload layers, consider placing the dependencies that change the least frequently earlier in your Dockerfile. Place rapidly changing dependencies (such as your application's source code) later in the stack.

Chain commands to avoid unnecessary file storage

Intermediate files created on a layer remain a part of that layer even if they are deleted in a subsequent layer. Consider the following example:

```
WORKDIR /tmp
RUN wget http://example.com/software.tar.gz
RUN wget tar -xvf software.tar.gz
RUN mv software/binary /opt/bin/myapp
RUN rm software.tar.gz
```

In this example, the layers created by the first and second RUN commands contain the original .tar.gz file and all of its unzipped contents. This is even though the .tar.gz file is deleted by the fourth RUN command. These commands can be chained together into a single RUN statement to ensure that these unnecessary files aren't part of the final Docker image:

```
WORKDIR /tmp
RUN wget http://example.com/software.tar.gz &&\
    wget tar -xvf software.tar.gz &&\
    mv software/binary /opt/bin/myapp &&\
    rm software.tar.gz
```

Use the closest regional endpoint

You can reduce latency in pulling images from Amazon ECR by ensuring that you are using the regional endpoint closest to where your application is running. If your application is running on an Amazon EC2 instance, you can use the following shell code to obtain the region from the Availability Zone of the instance:

```
REGION=$(curl -s http://169.254.169.254/latest/meta-data/placement/availability-zone
 |\
    sed -n 's/\(\d*\)[a-zA-Z]*$/\1/p')
```

The region can be passed to AWS CLI commands using the **--region** parameter, or set as the default region for a profile using the **aws configure** command. You can also set the region when making calls using the AWS SDK. For more information, see the documentation for the SDK for your specific programming language.

Making requests to Amazon ECR registries

You can push, pull, delete, view, and manage OCI images, Docker images, and OCI-compatible artifacts in Amazon ECR private registries using either IPv4-only endpoints or dual-stack (IPv4 and IPv6) endpoints. For making requests from IPv4 networks, you can use either dual-stack or IPv4 endpoints. For making requests from an IPv6 network, use a dual-stack endpoint. For more information about making requests to Amazon ECR Public registries using IPv4 and dual-stack endpoints, see Making requests to Amazon ECR Public registries. There are no additional charges for accessing Amazon ECR over IPv6. For more information about pricing, see Amazon Elastic Container Registry pricing.

> **Note**
>
> Amazon ECR does not support AWS PrivateLink traffic over dual-stack endpoints. You must use IPv4-only Amazon ECR endpoints if you need AWS PrivateLink support.

Amazon ECR endpoints are designated by attributes beyond IPv4-only endpoint or dual-stack endpoints support. These attributes can include:

- **Region** – Each endpoint is specific to a Region.
- **Type** – Endpoint selection depends on whether you're using the AWS SDK or OCI-compatible and Docker command line interfaces.
- **Security** – In select Regions Amazon ECR offers FIPS-compliant endpoints. For more information about a list of FIPS-compliant Amazon ECR endpoints, see Federal Information Processing Standard (FIPS) 140-3.

For more information about service endpoints supported by IPv4, dual-stack, Docker, and OCI client, which handles Amazon ECR API calls from AWS CLI and AWS SDKs see, Service endpoints.

Getting started with making requests over IPv6

To make a request to an Amazon ECR registry over IPv6, you need to use a dual-stack endpoint. Before accessing an Amazon ECR registry over IPv6, verify the following requirements:

- Your client and network must support IPv6.

- Amazon ECR supports the following request types over IPv6:
  - OCI and Docker client requests:

    `<registry-id>.dkr-ecr.<aws-region>.on.aws`
  - AWS API requests:

    `ecr.<aws-region>.api.aws`
- You must update any AWS Identity and Access Management (IAM) or registry policies that use source IP address filtering to include IPv6 address ranges. For more information, see Using IPv6 addresses in IAM policies.
- When you use IPv6, server access logs display `Remote IP` addresses in IPv6 format. Update your existing tools, scripts, and software to parse these IPv6-formatted IP addresses.

> **Note**
>
> If you experience issues related to the presence of IPv6 addresses in log files, contact AWS Support.

Testing IP address compatibility

If you are using use Linux/Unix or Mac OS X, you can test whether you can access a dual-stack endpoint over IPv6 by using the `curl` command as shown in the following example:

Example

```
curl --verbose https://ecr.us-west-2.api.aws
```

You get back information similar to the following example. If you are connected over IPv6 the connected IP address will be an IPv6 address.

```
* About to connect() to ecr.us-west-2.api.aws port 443 (#0)
* Trying IPv6 address... connected
* Connected to ecr.us-west-2.api.aws (IPv6 address) port 443 (#0)
> Host: ecr.us-west-2.api.aws
* Request completely sent off
```

If you are using Microsoft Windows 7 or Windows 10, you can test whether you can access a dual-stack endpoint over IPv4 or IPv6 by using the `ping` command as shown in the following example.

```
ping ecr.us-west-2.api.aws
```

Making requests over IPv6 by using dual-stack endpoints

You can make Amazon ECR API calls over IPv6 using dual-stack endpoints. The functionality and performance of Amazon ECR API operations remain consistent whether you use IPv4 or IPv6.

When you use the AWS Command Line Interface (AWS CLI) and AWS SDKs, you can enable IPv6 either by using a parameter or flag to switch to a dual-stack endpoint, or by directly specifying the dual-stack endpoint in your config file to override the default Amazon ECR endpoint. You can also make configuration changes by using a command, which sets `use_dualstack_endpoint` to true in the default profile. For more information about `use_dualstack_endpoint`, see Dual-stack and FIPS endpoints.

Example Making configuration changes by using a command

```
aws configure set default.ecr.use_dualstack_endpoint true
```

Example Making requests over IPv6 using AWS CLI

```
aws ecr describe-repositories --region us-west-2 --endpoint-url https://
ecr.us-west-2.api.aws
```

Using Amazon ECR endpoints from the docker CLI

After you sign in to your Amazon ECR repository and tag your image, you can push and pull OCI images and Docker images to and from Amazon ECR registries. The following examples demonstrate docker push and docker pull commands with both dual-stack endpoints.

Example Pushing docker images using IPv4 endpoint

```
docker push <registry-id>.dkr.ecr.us-west-1.amazonaws.com/my-repository:tag
```

Example Pushing docker images using dual-stack endpoint

```
docker push <registry-id>.dkr-ecr.us-west-1.on.aws/my-repository:tag
```

Example Pulling docker images using IPv4 endpoint

```
docker pull <registry-id>.dkr.ecr.us-west-1.amazonaws.com/my-repository:tag
```

Example Pulling docker images using dual-stack endpoint

```
docker pull <registry-id>.dkr-ecr.us-west-1.on.aws/my-repository:tag
```

Using IPv6 addresses in IAM policies

Before you access a registry using IPv6, ensure that your IAM user and Amazon ECR registry policies that use IP address filtering include IPv6 address ranges. If IP address filtering policies aren't updated to handle IPv6 addresses, clients might incorrectly lose or gain access to the registry when they start using IPv6. For more information about managing access permissions with IAM, see Identity and Access Management for Amazon Elastic Container Registry.

IAM policies that filter IP addresses use IP Address Condition Operators. The following registry policy example shows how to identify the `54.240.143.*` range of allowed IPv4 addresses by using IP address condition operators. Any IP addresses outside of this range are denied access to the registry (`exampleregistry`). Because all IPv6 addresses are outside of the allowed range, this policy prevents IPv6 addresses from accessing `exampleregistry`.

To allow both IPv4 (`54.240.143.0/24`) and IPv6 (`2001:DB8:1234:5678::/64`) address ranges, modify the registry policy's Condition element as shown in the following example. You can use this `Condition` block format to update both your IAM user and registry policies.

```
      "Condition": {
        "IpAddress": {
           "aws:SourceIp": [
             "54.240.143.0/24",
              "2001:DB8:1234:5678::/64"
            ]
        }
      }
```

Important

Before using IPv6 you must update all relevant IAM user and registry policies that use IP address filtering. We don't recommend using IP address filtering in registry policies.

You can review your IAM user policies using the IAM console at https://console.aws.amazon.com/iam/. For more information about IAM, see the IAM User Guide.

Amazon ECR private registry

An Amazon ECR private registry hosts your container images in a highly available and scalable architecture. You can use your private registry to manage private image repositories consisting of Docker and Open Container Initiative (OCI) images and artifacts. Each AWS account is provided with a default private Amazon ECR registry. For more information about Amazon ECR public registries, see Public registries in the *Amazon Elastic Container Registry Public User Guide*.

Private registry concepts

- The URL for your default private registry is `https://aws_account_id.dkr.ecr.region.amazonaws.com`.
- By default, your account has read and write access to the repositories in your private registry. However, users require permissions to make calls to the Amazon ECR APIs and to push or pull images to and from your private repositories. Amazon ECR provides several managed policies to control user access at varying levels. For more information, see Amazon Elastic Container Registry Identity-based policy examples.
- You must authenticate your Docker client to your private registry so that you can use the **docker push** and **docker pull** commands to push and pull images to and from the repositories in that registry. For more information, see Private registry authentication in Amazon ECR.
- Private repositories can be controlled with both user access policies and repository policies. For more information about repository policies, see Private repository policies in Amazon ECR.
- The repositories in your private registry can be replicated across AWS Regions in your own private registry and across separate accounts by configuring replication for your private registry. For more information, see Private image replication in Amazon ECR.

Private registry authentication in Amazon ECR

You can use the AWS Management Console, the AWS CLI, or the AWS SDKs to create and manage private repositories. You can also use those methods to perform some actions on images, such as listing or deleting them. These clients use standard AWS authentication methods. Even though you can use the Amazon ECR API to push and pull images, you're more likely to use the Docker CLI or a language-specific Docker library.

The Docker CLI doesn't support native IAM authentication methods. Additional steps must be taken so that Amazon ECR can authenticate and authorize Docker push and pull requests.

The registry authentication methods that are detailed in the following sections are available.

Using the Amazon ECR credential helper

Amazon ECR provides a Docker credential helper which makes it easier to store and use Docker credentials when pushing and pulling images to Amazon ECR. For installation and configuration steps, see Amazon ECR Docker Credential Helper.

> **Note**
>
> The Amazon ECR Docker credential helper doesn't support multi-factor authentication (MFA) currently.

Using an authorization token

An authorization token's permission scope matches that of the IAM principal used to retrieve the authentication token. An authentication token is used to access any Amazon ECR registry that your IAM principal has access to and is valid for 12 hours. To obtain an authorization token, you must use the GetAuthorizationToken API operation to retrieve a base64-encoded authorization token containing the username `AWS` and an encoded password. The AWS CLI `get-login-password` command simplifies this by retrieving and decoding the authorization token which you can then pipe into a **docker login** command to authenticate.

To authenticate Docker to an Amazon ECR private registry with get-login

- To authenticate Docker to an Amazon ECR registry with get-login-password, run the **aws ecr get-login-password** command. When passing the authentication token to the **docker login** command, use the value `AWS` for the username and specify the Amazon ECR registry URI you want to authenticate to. If authenticating to multiple registries, you must repeat the command for each registry.

> **Important**
>
> If you receive an error, install or upgrade to the latest version of the AWS CLI. For more information, see Installing the AWS Command Line Interface in the *AWS Command Line Interface User Guide*.

- get-login-password (AWS CLI)

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

- Get-ECRLoginCommand (AWS Tools for Windows PowerShell)

```
(Get-ECRLoginCommand).Password | docker login --username AWS --password-
stdin aws_account_id.dkr.ecr.region.amazonaws.com
```

Using HTTP API authentication

Amazon ECR supports the Docker Registry HTTP API. However, because Amazon ECR is a private registry, you must provide an authorization token with every HTTP request. You can add an HTTP authorization header using the `-H` option for **curl** and pass the authorization token provided by the **get-authorization-token** AWS CLI command.

To authenticate with the Amazon ECR HTTP API

1. Retrieve an authorization token with the AWS CLI and set it to an environment variable.

```
TOKEN=$(aws ecr get-authorization-token --output text --query
 'authorizationData[].authorizationToken')
```

2. To authenticate to the API, pass the `$TOKEN` variable to the `-H` option of **curl**. For example, the following command lists the image tags in an Amazon ECR repository. For more information, see the Docker Registry HTTP API reference documentation.

```
curl -i -H "Authorization: Basic $TOKEN"
 https://aws_account_id.dkr.ecr.region.amazonaws.com/v2/amazonlinux/tags/list
```

The output is as follows:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Date: Thu, 04 Jan 2018 16:06:59 GMT
Docker-Distribution-Api-Version: registry/2.0
Content-Length: 50
Connection: keep-alive

{"name":"amazonlinux","tags":["2017.09","latest"]}
```

Private registry settings in Amazon ECR

Amazon ECR uses private registry settings to configure features at the registry level. The private registry settings are configured separately for each Region. You can use private registry settings to configure the following features.

- **Registry permissions** – A registry permissions policy provides control over the replication and pull through cache permissions. For more information, see Private registry permissions in Amazon ECR.
- **Pull through cache rules** – A pull through cache rule is used to cache images from an upstream registry in your Amazon ECR private registry. For more information, see Sync an upstream registry with an Amazon ECR private registry.
- **Replication configuration** – The replication configuration is used to control whether your repositories are copied across AWS Regions or accounts. For more information, see Private image replication in Amazon ECR
- **Repository creation templates** – A repository creation template is used to define the standard settings to apply when new repositories are created by Amazon ECR on your behalf. For example, repositories created by a pull through cache action. For more information, see Templates to control repositories created during a pull through cache or replication action.
- **Scanning configuration** – By default, your registry is enabled for basic scanning. You may enable enhanced scanning which provides an automated, continuous scanning mode that scans for both operating system and programming language package vulnerabilities. For more information, see Scan images for software vulnerabilities in Amazon ECR.

Private registry permissions in Amazon ECR

Amazon ECR uses a **registry policy** to grant permissions to an AWS principal at the private registry level.

The scope is set by choosing the registry policy version. There are two versions with different registry policy scope: version 1 (V1) and version 2 (V2). V2 is the expanded registry policy scope that includes all ECR permissions. For the full list of API actions, see the *Amazon ECR API Guide*. The V2 version is the default registry policy scope. For more information about viewing or setting your registry policy scope, see Switching to the extended registry policy scope. For information about general settings for your Amazon ECR private registry, see Private registry settings in Amazon ECR.

The versions are detailed as follows.

- **V1** – For version 1, Amazon ECR only enforces the following permissions at the private registry level.
  - `ecr:ReplicateImage` – Grants permission to another account, referred to as the source registry, to replicate its images to your registry. This is only used for cross-account replication.
  - `ecr:BatchImportUpstreamImage` – Grants permission to retrieve the external image and import it to your private registry.
  - `ecr:CreateRepository` – Grants permission to create a repository in a private registry. This permission is required if the repository storing either the replicated or cached images doesn't already exist in the private registry.
- **V2** – For version 2, Amazon ECR allows all ECR actions in the policy and enforces the registry policy in all ECR requests.

You can use the console or the CLI to view or change your registry policy scope.

> **Note**
>
> While it is possible to add the `ecr:*` action to a private registry policy, it is considered best practice to only add the specific actions required based on the feature you're using rather than use a wildcard.

Topics

- Private registry policy examples for Amazon ECR

- Switching to the extended registry policy scope
- Granting registry permissions for cross account replication in Amazon ECR
- Granting registry permissions for pull through cache in Amazon ECR

Private registry policy examples for Amazon ECR

The following examples show registry permissions policy statements that you could use to control the permissions that users have to your Amazon ECR registry.

> **Note**
>
> In each example, if the `ecr:CreateRepository` action is removed from your registry policy, replication can still occur. However, for successful replication, you need to create repositories with the same name within your account.

Example: Allow the root user of a source account to replicate all repositories

The following registry permissions policy allows the root user of a source account to replicate all repositories.

JSON

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"ReplicationAccessCrossAccount",
            "Effect":"Allow",
            "Principal":{
                "AWS":"arn:aws:iam::111122223333:root"
            },
            "Action":[
                "ecr:CreateRepository",
                "ecr:ReplicateImage"
            ],
            "Resource": [
                "arn:aws:ecr:us-west-2:444455556666:repository/*"
            ]
```

```
        }
    ]
}
```

Example: Allow root users from multiple accounts

The following registry permissions policy has two statements. Each statement allows the root user of a source account to replicate all repositories.

JSON

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"ReplicationAccessCrossAccount1",
            "Effect":"Allow",
            "Principal":{
                "AWS":"arn:aws:iam::111122223333:root"
            },
            "Action":[
                "ecr:CreateRepository",
                "ecr:ReplicateImage"
            ],
            "Resource": [
                "arn:aws:ecr:us-west-2:123456789012:repository/*"
            ]
        },
        {
            "Sid":"ReplicationAccessCrossAccount2",
            "Effect":"Allow",
            "Principal":{
                "AWS":"arn:aws:iam::444455556666:root"
            },
            "Action":[
                "ecr:CreateRepository",
                "ecr:ReplicateImage"
            ],
            "Resource": [
                "arn:aws:ecr:us-west-2:123456789012:repository/*"
            ]
```

```
        }
    ]
}
```

Example: Allow the root user of a source account to replicate all repositories with prefix `prod-`.

The following registry permissions policy allows the root user of a source account to replicate all repositories that start with `prod-`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"ReplicationAccessCrossAccount",
            "Effect":"Allow",
            "Principal":{
                "AWS":"arn:aws:iam::111122223333:root"
            },
            "Action":[
                "ecr:CreateRepository",
                "ecr:ReplicateImage"
            ],
            "Resource": [
                "arn:aws:ecr:us-west-2:444455556666:repository/prod-*"
            ]
        }
    ]
}
```

Switching to the extended registry policy scope

Important

For new users, your registries are automatically configured to use the V2 registry policy upon creation. There is no action for you to take. Amazon ECR doesn't recommend reverting to the previous registry policy V1.

You can use the console or the CLI to view or change your registry policy scope.

AWS Management Console

Use the following steps to view your account settings. To view or update the registry policy scope, see the CLI procedure on this page.

Turn on the enhanced registry policy for your private registry

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/private-registry/repositories
2. From the navigation bar, choose the Region.
3. In the navigation pane, choose **Private registry**, **Feature & Settings**, and then choose **Permissions**.
4. On the **Permissions** page, for **Registry policy** view your policy JSON. If you have the V1 policy, a banner displays with instructions to update to V2. Choose **Enable**.

   A banner displays indicating that the registry policy scope has been updated to V2.
5. You can also optionally configure permissions with the CLI. For more information, see Private registry settings in Amazon ECR.

Note

To view or update the registry policy scope, see the CLI procedure on this page.

AWS CLI

Amazon ECR generates the V2 registry policy. Use the following steps to view or update the registry policy scope. You cannot view or change the registry policy scope in the console

- To retrieve the registry policy you are currently using.

```
aws ecr get-account-setting --name REGISTRY_POLICY_SCOPE
```

The parameter name is a required field. If you don't provide the name you will receive the following error:

```
 aws: error: the following arguments are required: --name
```

View the output for your registry policy command. In the following example output, the registry policy version is V1.

```
{
    "name": "REGISTRY_POLICY_SCOPE",
    "value": "V1"
}
```

You can change your registry policy version from `V1` to `V2`. V1 is not the recommended registry policy scope.

```
aws ecr put-account-setting --name REGISTRY_POLICY_SCOPE --value value
```

For example, use the following command to update to V2.

```
aws ecr put-account-setting --name REGISTRY_POLICY_SCOPE --value V2
```

View the output for your registry policy command. In the following example output, the registry policy version was updated to V2.

```
{
    "name": "REGISTRY_POLICY_SCOPE",
    "value": "V2"
}
```

Granting registry permissions for cross account replication in Amazon ECR

The cross account policy type is used to grant permissions to an AWS principal, allowing the replication of the repositories from a source registry to your registry. By default, you have permission to configure cross-Region replication within your own registry. You only need to configure the registry policy if you're granting another account permission to replicate contents to your registry.

A registry policy must grant permission for the `ecr:ReplicateImage` API action. This API is an internal Amazon ECR API that can replicate images between Regions or accounts. You can also grant permission for the `ecr:CreateRepository` permission, which allows Amazon ECR to create repositories in your registry if they don't exist already. If the `ecr:CreateRepository` permission isn't provided, a repository with the same name as the source repository must be created manually in your registry. If neither is done, replication fails. Any failed `CreateRepository` or `ReplicateImage` API actions show up in CloudTrail.

To configure a permissions policy for replication (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region to configure your registry policy in.
3. In the navigation pane, choose **Private registry**, choose **Features & Settings**, and then choose **Permissions**.
4. On the **Registry permissions** page, choose **Generate statement**.
5. Complete the following steps to define your policy statement using the policy generator.
    a. For **Policy type**, choose **Replication - cross account**.
    b. For **Statement id**, enter a unique statement ID. This field is used as the `Sid` on the registry policy.
    c. For **Accounts**, enter the account IDs for each account you want to grant permissions to. When specifying multiple account IDs, separate them with a comma.
6. Choose **Save**.

To configure a permissions policy for replication (AWS CLI)

1. Create a file named `registry_policy.json` and populate it with a registry policy.

JSON

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"ReplicationAccessCrossAccount",
            "Effect":"Allow",
            "Principal":{
                "AWS":"arn:aws:iam::111122223333:root"
            },
            "Action":[
                "ecr:CreateRepository",
                "ecr:ReplicateImage"
            ],
            "Resource": [
                "arn:aws:ecr:us-west-2:444455556666:repository/*"
            ]
        }
    ]
}
```

2. Create the registry policy using the policy file.

```
aws ecr put-registry-policy \
      --policy-text file://registry_policy.json \
      --region us-west-2
```

3. Retrieve the policy for your registry to confirm.

```
aws ecr get-registry-policy \
      --region us-west-2
```

Granting registry permissions for pull through cache in Amazon ECR

Amazon ECR private registry permissions may be used to scope the permissions of individual IAM entities to use pull through cache. If an IAM entity has more permissions granted by an IAM policy than the registry permissions policy is granting, the IAM policy takes precedence.

To create a private registry permissions policy (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region to configure your private registry permissions statement in.
3. In the navigation pane, choose **Private registry**, choose **Features & Settings**, and then choose **Permissions**.
4. On the **Registry permissions** page, choose **Generate statement**.
5. For each pull through cache permissions policy statement you want to create, do the following.

    a. For **Policy type**, choose **Pull through cache policy**.

    b. For **Statement id**, provide a name for the pull through cache statement policy.

    c. For **IAM entities**, specify the users, groups, or roles to include in the policy.

    d. For **Cache namespace**, select the pull through cache rule to associate the policy with.

    e. For **Repository names**, specify the repository base name to apply the rule for. For example, if you want to specify the Amazon Linux repository on Amazon ECR Public, the repository name would be `amazonlinux`.

Amazon ECR private repositories

An Amazon ECR private repository contains your Docker images, Open Container Initiative (OCI) images, and OCI compatible artifacts. You can create, monitor, and delete image repositories and set permissions that control who can access them by using Amazon ECR API operations or the **Repositories** section of the Amazon ECR console. Amazon ECR also integrates with the Docker CLI, so that you can push and pull images from your development environments to your repositories.

Topics

- Private repository concepts
- Creating an Amazon ECR private repository to store images
- Viewing the contents and details of a private repository in Amazon ECR
- Deleting a private repository in Amazon ECR
- Private repository policies in Amazon ECR
- Tagging a private repository in Amazon ECR

Private repository concepts

- By default, your account has read and write access to the repositories in your default registry (*`aws_account_id`*`.dkr.ecr.`*`region`*`.amazonaws.com`). However, users require permissions to make calls to the Amazon ECR APIs and to push or pull images to and from your repositories. Amazon ECR provides several managed policies to control user access at varying levels. For more information, see Amazon Elastic Container Registry Identity-based policy examples.
- Repositories can be controlled with both user access policies and individual repository policies. For more information, see Private repository policies in Amazon ECR.
- Repository names can support namespaces, which you can use to group similar repositories. For example, if there are several teams using the same registry, Team A can use the `team-a` namespace, and Team B can use the `team-b` namespace. By doing this, each team has their own image called `web-app` with each image prefaced with the team namespace. This configuration allows these images on each team to be used simultaneously without interference. Team A's image is `team-a/web-app`, and Team B's image is `team-b/web-app`.
- Your images can be replicated to other repositories across Regions in your own registry and across accounts. You can do this by specifying a replication configuration in your registry settings. For more information, see Private registry settings in Amazon ECR.

Creating an Amazon ECR private repository to store images

> **Important**
>
> Dual-layer server-side encryption with AWS KMS (DSSE-KMS) is only available in the AWS GovCloud (US) Regions.

Create an Amazon ECR private repository, and then use the repository to store your container images. Use the following steps to create a private repository using the AWS Management Console. For steps to create a repository using the AWS CLI, see Step 2: Create a repository.

To create a repository (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region to create your repository in.
3. Choose **Private repositories**, and then choose **Create repository**.
4. For **Repository name**, enter a unique name for your repository. The repository name can be specified on its own (for example `nginx-web-app`). Alternatively, it can be prepended with a namespace to group the repository into a category (for example `project-a/nginx-web-app`).

   > **Note**
   >
   > The repository name may container a maximum of 256 characters. The name must start with a letter and can only contain lowercase letters, numbers, hyphens, underscores, periods and forward slashes. Using a double hyphen, double underscore, or double forward slash isn't supported.

5. For **Image tag immutability**, choose one of the following tag mutability settings for the repository.

   - **Mutable** – Choose this option if you want image tags to be overwritten. Recommended for repositories using pull through cache actions to ensure Amazon ECR can update cached images. Additionally, to disable tag updates for a few mutable tags, enter tag names or use wildcards (*) to match multiple similar tags in the **Mutable tag exclusion** text box.

- **Immutable** – Choose this option if you want to prevent image tags from being overwritten, and it applies to all tags and exclusions in the repository when pushing an image with existing tag. Amazon ECR returns an `ImageTagAlreadyExistsException` if you attempt to push an image with an existing tag. Additionally, to enable tag updates for a few immutable tags, enter tag names or use wildcards (*) to match multiple similar tags in the **Immutable tag exclusion** text box.

> **Note**
>
> Individual tag mutability settings aren't supported.

6. For **Encryption configuration**, choose between **AES-256** or **AWS KMS**. For more information, see Encryption at rest.

   a. If AWS KMS is chosen, choose between Single-layer encryption and Dual-layer encryption. There are additional charges for using AWS KMS or Dual-layer encryption. For more information, see Amazon ECR Service Pricing.

   b. By default, AWS managed key with the alias `aws/ecr` is chosen. This key is created in your account the first time that you create a repository with AWS KMS encryption enabled. Select **Customer managed key (advanced)** to choose your own AWS KMS key. The AWS KMS key must be in the same Region as the cluster. Select **Create an AWS KMS key** to navigate to the AWS KMS console to create your own key.

7. For **Image scanning settings**, while you can specify the scan settings at the repository level for basic scanning, it is a best practice to specify the scan configuration at the private registry level. Configuring the scanning settings at the private registry level enables you to choose between enhanced scanning or basic scanning, and also allows you to define filters to specify which repositories should be scanned.

8. Choose **Create**.

Next steps

To view the steps to push an image to your repository, select the repository and choose **View push commands**. For more information about pushing an image to your repository, see Pushing an image to an Amazon ECR private repository.

Viewing the contents and details of a private repository in Amazon ECR

After you created a private repository, you can view details about the repository in the AWS Management Console:

- Which images are stored in a repository
- Details about each image stored in the repository, including the size and SHA digest for each image
- The scan frequency specified for the contents of the repository
- Whether the repository has an active pull through cache rule associated with it
- The encryption setting for the repository

> **Note**
>
> Starting with Docker version 1.9, the Docker client compresses image layers before pushing them to a V2 Docker registry. The output of the **docker images** command shows the uncompressed image size. Therefore, keep in mind that Docker might return a larger image than the image shown in the AWS Management Console.

To view repository information (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region that contains the repository to view.
3. In the navigation pane, choose **Repositories**.
4. On the **Repositories** page, choose the **Private** tab and then the repository to view.
5. On the repository detail page, the console defaults to the **Images** view. Use the navigation menu to view other information about the repository.
   - Choose **Summary** to view the repository details and pull count data for the repository.
   - Choose **Images** to view information about the image tags in the repository. To view more information about the image, select the image tag. For more information, see Viewing image details in Amazon ECR.

If there are untagged images that you want to delete, you can select the box to the left of the repositories to delete and choose **Delete**. For more information, see Deleting an image in Amazon ECR.

- Choose **Permissions** to view the repository policies that are applied to the repository. For more information, see Private repository policies in Amazon ECR.
- Choose **Lifecycle Policy** to view the lifecycle policy rules that are applied to the repository. The lifecycle events history is also viewed here. For more information, see Automate the cleanup of images by using lifecycle policies in Amazon ECR.
- Choose **Tags** to view the metadata tags that are applied to the repository.

Deleting a private repository in Amazon ECR

If you're finished using a repository, you can delete it. When you delete a repository in the AWS Management Console, all of the images contained in the repository are also deleted; this cannot be undone.

> **Important**
>
> Images in the deleted repositories are also deleted. You cannot undo this operation.

To delete a repository (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region that contains the repository to delete.
3. In the navigation pane, choose **Repositories**.
4. On the **Repositories** page, choose the **Private** tab and then select the repository to delete and choose **Delete**.
5. In the **Delete *repository_name*** window, verify that the selected repositories should be deleted and choose **Delete**.

Private repository policies in Amazon ECR

Amazon ECR uses resource-based permissions to control access to repositories. Resource-based permissions let you specify which users or roles have access to a repository and what actions they

can perform on the repository. By default, only the AWS account that created the repository has access to the repository. You can apply a repository policy that allows additional access to your repository.

Topics

- Repository policies vs IAM policies
- Private repository policy examples in Amazon ECR
- Setting a private repository policy statement in Amazon ECR

Repository policies vs IAM policies

Amazon ECR repository policies are a subset of IAM policies that are scoped for, and specifically used for, controlling access to individual Amazon ECR repositories. IAM policies are generally used to apply permissions for the entire Amazon ECR service but can also be used to control access to specific resources as well.

Both Amazon ECR repository policies and IAM policies are used when determining which actions a specific user or role may perform on a repository. If a user or role is allowed to perform an action through a repository policy but is denied permission through an IAM policy (or vice versa) then the action will be denied. A user or role only needs to be allowed permission for an action through either a repository policy or an IAM policy but not both for the action to be allowed.

Important

Amazon ECR requires that users have permission to make calls to the `ecr:GetAuthorizationToken` API through an IAM policy before they can authenticate to a registry and push or pull any images from any Amazon ECR repository. Amazon ECR provides several managed IAM policies to control user access at varying levels. For more information, see Amazon Elastic Container Registry Identity-based policy examples.

You can use either of these policy types to control access to your repositories, as shown in the following examples.

This example shows an Amazon ECR repository policy, which allows for a specific user to describe the repository and the images within the repository.

This example shows an IAM policy that achieves the same goal as above, by scoping the policy to a repository (specified by the full ARN of the repository) using the resource parameter. For more information about Amazon Resource Name (ARN) format, see Resources.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowDescribeRepoImage",
            "Effect": "Allow",
            "Action": [
                "ecr:DescribeImages",
                "ecr:DescribeRepositories"
            ],
            "Resource": ["arn:aws:ecr:us-
east-1:111122223333:repository/repository-name"]
        }
    ]
}
```

Private repository policy examples in Amazon ECR

> **Important**
>
> The repository policy examples on this page are meant to be applied to Amazon ECR private repositories. They will not work properly if used with an IAM principal directly unless modified to specify the Amazon ECR repository as the resource. For more information on setting repository policies, see Setting a private repository policy statement in Amazon ECR.

Amazon ECR repository policies are a subset of IAM policies that are scoped for, and specifically used for, controlling access to individual Amazon ECR repositories. IAM policies are generally used to apply permissions for the entire Amazon ECR service but can also be used to control access to specific resources as well. For more information, see Repository policies vs IAM policies.

The following repository policy examples show permission statements that you could use to control access to your Amazon ECR private repositories.

> **Important**
>
> Amazon ECR requires that users have permission to make calls to the `ecr:GetAuthorizationToken` API through an IAM policy before they can authenticate to a registry and push or pull any images from any Amazon ECR repository. Amazon ECR provides several managed IAM policies to control user access at varying levels. For more information, see Amazon Elastic Container Registry Identity-based policy examples.

Example: Allow one or more users

The following repository policy allows one or more users to push and pull images to and from a repository.

Example: Allow another account

The following repository policy allows a specific account to push images.

> **Important**
>
> The account you are granting permissions to must have the Region you are creating the repository policy in enabled, otherwise an error will occur.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCrossAccountPush",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
            },
            "Action": [
                "ecr:BatchCheckLayerAvailability",
```

```
                "ecr:CompleteLayerUpload",
                "ecr:InitiateLayerUpload",
                "ecr:PutImage",
                "ecr:UploadLayerPart"
            ],
            "Resource": "*"
        }
    ]
}
```

The following repository policy allows some users to pull images (*pull-user-1* and *pull-user-2*) while providing full access to another (*admin-user*).

> **Note**
>
> For more complicated repository policies that are not currently supported in the AWS Management Console, you can apply the policy with the **set-repository-policy** AWS CLI command.

Example: Deny all

The following repository policy denies all users in all accounts the ability to pull images.

Example: Restricting access to specific IP addresses

The following example denies permissions to any user to perform any Amazon ECR operations when applied to a repository from a specific range of addresses.

The condition in this statement identifies the `54.240.143.*` range of allowed Internet Protocol version 4 (IPv4) IP addresses.

The `Condition` block uses the `NotIpAddress` conditions and the `aws:SourceIp` condition key, which is an AWS-wide condition key. For more information about these condition keys, see AWS Global Condition Context Keys. The `aws:sourceIp` IPv4 values use the standard CIDR notation. For more information, see IP Address Condition Operators in the *IAM User Guide*.

JSON

```
{
```

```
    "Version": "2012-10-17",
    "Id": "ECRPolicyId1",
    "Statement": [
        {
            "Sid": "IPAllow",
            "Effect": "Deny",
            "Principal": "*",
            "Action": "ecr:*",
            "Resource": "*",
            "Condition": {
                "NotIpAddress": {
                    "aws:SourceIp": "54.240.143.0/24"
                }
            }
        }
    ]
}
```

Example: Allow an AWS service

The following repository policy allows AWS CodeBuild access to the Amazon ECR API actions necessary for integration with that service. When using the following example, you should use the `aws:SourceArn` and `aws:SourceAccount` condition keys to scope which resources can assume these permissions. For more information, see Amazon ECR sample for CodeBuild in the *AWS CodeBuild User Guide*.

Setting a private repository policy statement in Amazon ECR

You can add an access policy statement to a repository in the AWS Management Console by following the steps below. You can add multiple policy statements per repository. For example policies, see Private repository policy examples in Amazon ECR.

> **Important**
>
> Amazon ECR requires that users have permission to make calls to the `ecr:GetAuthorizationToken` API through an IAM policy before they can authenticate to a registry and push or pull any images from any Amazon ECR repository. Amazon ECR provides several managed IAM policies to control user access at varying levels. For more information, see Amazon Elastic Container Registry Identity-based policy examples.

To set a repository policy statement

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region that contains the repository to set a policy statement on.
3. In the navigation pane, choose **Repositories**.
4. On the **Repositories** page, choose the repository to set a policy statement on to view the contents of the repository.
5. From the repository image list view, in the navigation pane, choose **Permissions**, **Edit**.

> **Note**
>
> If you don't see the **Permissions** option in the navigation pane, ensure that you are in the repository image list view.

6. On the **Edit permissions** page, choose **Add statement**.
7. For **Statement name**, enter a name for the statement.
8. For **Effect**, choose whether the policy statement will result in an allow or an explicit deny.
9. For **Principal**, choose the scope to apply the policy statement to. For more information, see AWS JSON Policy Elements: Principal in the *IAM User Guide*.
   - You can apply the statement to all authenticated AWS users by selecting the **Everyone (*)** check box.
   - For **Service principal**, specify the service principal name (for example, `ecs.amazonaws.com`) to apply the statement to a specific service.
   - For **AWS Account IDs**, specify an AWS account number (for example, 111122223333) to apply the statement to all users under a specific AWS account. Multiple accounts can be specified by using a comma delimited list.

     > **Important**
     >
     > The account you are granting permissions to must have the Region you are creating the repository policy in enabled, otherwise an error will occur.

   - For **IAM Entities**, select the roles or users under your AWS account to apply the statement to.

> **Note**
>
> For more complicated repository policies that are not currently supported in the AWS Management Console, you can apply the policy with the **set-repository-policy** AWS CLI command.

10. For **Actions**, choose the scope of the Amazon ECR API operations that the policy statement should apply to from the list of individual API operations.
11. When you are finished, choose **Save** to set the policy.
12. Repeat the previous step for each repository policy to add.

Tagging a private repository in Amazon ECR

To help you manage your Amazon ECR repositories, you can assign your own metadata to new or existing Amazon ECR repositories by using AWS resource *tags*. For example, you could define a set of tags for your account's Amazon ECR repositories that helps you track the owner of each repository.

Tag basics

Tags don't have any semantic meaning to Amazon ECR and are interpreted strictly as a string of characters. Tags are not automatically assigned to your resources. You can edit tag keys and values, and you can remove tags from a resource at any time. You can set the value of a tag to an empty string, but you can't set the value of a tag to null. If you add a tag that has the same key as an existing tag on that resource, the new value overwrites the old value. If you delete a resource, any tags for the resource are also deleted.

You can work with tags using the Amazon ECR console, the AWS CLI, and the Amazon ECR API.

Using AWS Identity and Access Management (IAM), you can control which users in your AWS account have permission to create, edit, or delete tags. For information about tags in IAM policies, see the section called “Using Tag-Based Access Control”.

Tagging your resources for billing

The tags you add to your Amazon ECR repositories are helpful when reviewing cost allocation after enabling them in your Cost & Usage Report. For more information, see Amazon ECR usage reports.

To see the cost of your combined resources, you can organize your billing information based on resources that have the same tag key values. For example, you can tag several resources with a specific application name, and then organize your billing information to see the total cost of that application across several services. For more information about setting up a cost allocation report with tags, see The Monthly Cost Allocation Report in the *AWS Billing User Guide*.

> **Note**
>
> If you've just enabled reporting, data for the current month is available for viewing after 24 hours.

Adding tags to a private repository in Amazon ECR

You can add tags to a private repository.

For information about names and best practices for tags, see Tag naming limits and requirements and Best practices in the *Tagging AWS Resources User Guide*.

Adding tags to a repository (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, select the region to use.
3. In the navigation pane, choose **Repositories**.
4. On the **Repositories** page, select the check box next to the repository you want to tag.
5. From the **Action** menu, select **Repository tags**.
6. On the **Repository tags** page, select **Add tags**, **Add tag**.
7. On the **Edit repository tags** page, specify the key and value for each tag, and then choose **Save**.

Adding tags to a repository (AWS CLI or API)

You can add or overwrite one or more tags by using the AWS CLI or an API.

- AWS CLI - tag-resource
- API action - TagResource

The following examples show how to add tags using the AWS CLI.

Example 1: Tag a repository

The following command tags a repository.

```
aws ecr tag-resource \
     --resource-arn arn:aws:ecr:region:account_id:repository/repository_name \
     --tags Key=stack,Value=dev
```

Example 2: Tag a repository with multiple tags

The following command adds three tags to a repository.

```
aws ecr tag-resource \
     --resource-arn arn:aws:ecr:region:account_id:repository/repository_name \
     --tags Key=key1,Value=value1 Key=key2,Value=value2 Key=key3,Value=value3
```

Example 3: List tags for a repository

The following command lists the tags associated with a repository.

```
aws ecr list-tags-for-resource \
     --resource-arn arn:aws:ecr:region:account_id:repository/repository_name
```

Example 4: Create a repository and add a tag

The following command creates a repository named `test-repo` and adds a tag with key `team` and value `devs`.

```
aws ecr create-repository \
     --repository-name test-repo \
     --tags Key=team,Value=devs
```

Deleting tags from a private repository in Amazon ECR

You can delete tags from a private repository.

To delete a tag from a private repository (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.

2. From the navigation bar, select the region to use.
3. On the **Repositories** page, select the check box next to the repository you want to remove a tag from.
4. From the **Action** menu, select **Repository tags**.
5. On the **Repository tags** page, select **Edit**.
6. On the **Edit repository tags** page, select **Remove** for each tag you want to delete, and choose **Save**.

To delete a tag from a private repository (AWS CLI)

You can delete one or more tags by using the AWS CLI or an API.

- AWS CLI - untag-resource
- API action - UntagResource

The following example shows how to delete a tag from a repository using the AWS CLI.

```
aws ecr untag-resource \
     --resource-arn arn:aws:ecr:region:account_id:repository/repository_name \
     --tag-keys tag_key
```

Private images in Amazon ECR

Amazon ECR stores Docker images, Open Container Initiative (OCI) images, and OCI compatible artifacts in private repositories. You can use the Docker CLI, or your preferred client, to push and pull images to and from your repositories.

With Amazon ECR support for OCI v1.1, you can store and manage reference artifacts that are defined by the OCI Referrers API. Artifacts include signatures, Software Bill of Materials (SBoMs), Helm charts, scan results, and attestations. A set of artifacts for a container image is transferred with that container and stored as a separate image that counts as an image consumed for your repository.

The Signing an image stored in an Amazon ECR private repository and Deleting signatures and other artifacts from an Amazon ECR private repository pages provide examples of how to use signature-related artifacts. For more information on signing container images, see Signing container images in the *AWS Signer Developer Guide*.

Topics

- Pushing an image to an Amazon ECR private repository
- Signing an image stored in an Amazon ECR private repository
- Deleting signatures and other artifacts from an Amazon ECR private repository
- Viewing image details in Amazon ECR
- Pulling an image to your local environment from an Amazon ECR private repository
- Pulling the Amazon Linux container image
- Deleting an image in Amazon ECR
- Retagging an image in Amazon ECR
- Preventing image tags from being overwritten in Amazon ECR
- Container image manifest format support in Amazon ECR
- Using Amazon ECR images with Amazon ECS
- Using Amazon ECR Images with Amazon EKS

Pushing an image to an Amazon ECR private repository

You can push your Docker images, manifest lists, and Open Container Initiative (OCI) images and compatible artifacts to your private repositories.

Amazon ECR also provides a way to replicate your images to other repositories. By specifying a replication configuration in your private registry settings, you can replicate across Regions in your own registry and across different accounts. For more information, see Private registry settings in Amazon ECR.

Topics

- IAM permissions for pushing an image to an Amazon ECR private repository
- Pushing a Docker image to an Amazon ECR private repository
- Pushing a multi-architecture image to an Amazon ECR private repository
- Pushing a Helm chart to an Amazon ECR private repository

IAM permissions for pushing an image to an Amazon ECR private repository

Users need IAM permissions to push images to Amazon ECR private repositories. Following the best practice of granting least privilege, you can grant access to a specific repository. You can also grant access to all repositories.

A user must authenticate to each Amazon ECR registry they want to push images to by requesting an authorization token. Amazon ECR provides several AWS managed policies to control user access at varying levels. For more information, see AWS managed policies for Amazon Elastic Container Registry.

You can also create a your own IAM policies. The following IAM policy grants the required permissions for pushing an image to a specific repository. To limit the permissions for a specific repository, use the full Amazon Resource Name (ARN) of the repository.

The following IAM policy grants the required permissions for pushing an image to all repositories.

JSON

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecr:CompleteLayerUpload",
                "ecr:GetAuthorizationToken",
                "ecr:UploadLayerPart",
                "ecr:InitiateLayerUpload",
                "ecr:BatchCheckLayerAvailability",
                "ecr:PutImage"
            ],
             "Resource": "arn:aws:ecr:us-west-2:111122223333:repository/*"
        }
    ]
}
```

Pushing a Docker image to an Amazon ECR private repository

You can push your container images to an Amazon ECR repository with the **docker push** command.

Amazon ECR also supports creating and pushing Docker manifest lists that are used for multi-architecture images. For information, see Pushing a multi-architecture image to an Amazon ECR private repository.

To push a Docker image to an Amazon ECR repository

The Amazon ECR repository must exist before you push the image. For more information, see the section called “Creating a repository to store images”.

1. Authenticate your Docker client to the Amazon ECR registry to which you intend to push your image. Authentication tokens must be obtained for each registry used, and the tokens are valid for 12 hours. For more information, see Private registry authentication in Amazon ECR.

   To authenticate Docker to an Amazon ECR registry, run the **aws ecr get-login-password** command. When passing the authentication token to the **docker login** command, use the value `AWS` for the username and specify the Amazon ECR registry URI you want to authenticate to. If authenticating to multiple registries, you must repeat the command for each registry.

> **Important**
>
> If you receive an error, install or upgrade to the latest version of the AWS CLI. For more information, see Installing the AWS Command Line Interface in the *AWS Command Line Interface User Guide*.

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

2. If your image repository doesn't exist in the registry you intend to push to yet, create it. For more information, see Creating an Amazon ECR private repository to store images.
3. Identify the local image to push. Run the **docker images** command to list the container images on your system.

   ```
   docker images
   ```

   You can identify an image with the *repository:tag* value or the image ID in the resulting command output.
4. Tag your image with the Amazon ECR registry, repository, and optional image tag name combination to use. The registry format is *aws_account_id*.dkr.ecr.*region*.amazonaws.com. The repository name should match the repository that you created for your image. If you omit the image tag, we assume that the tag is `latest`.

   The following example tags a local image with the ID *e9ae3c220b23* as *aws_account_id*.dkr.ecr.*region*.amazonaws.com/my-repository:tag.

   ```
   docker tag e9ae3c220b23 aws_account_id.dkr.ecr.region.amazonaws.com/my-
   repository:tag
   ```
5. Push the image using the **docker push** command:

   ```
   docker push aws_account_id.dkr.ecr.region.amazonaws.com/my-repository:tag
   ```
6. (Optional) Apply any additional tags to your image and push those tags to Amazon ECR by repeating Step 4 and Step 5.

Pushing a multi-architecture image to an Amazon ECR private repository

You can push multi-architecture images to an Amazon ECR repository by creating and pushing Docker manifest lists. A *manifest list* is a list of images that is created by specifying one or more image names. In most cases, the manifest list is created from images that serve the same function but are for different operating systems or architectures. The manifest list isn't required. For more information, see docker manifest.

A manifest list can be pulled or referenced in an Amazon ECS task definition or Amazon EKS pod spec like other Amazon ECR images.

Prerequisites

- In your Docker CLI, turn on experimental features. For information about experimental features, see Experimental features in the Docker documentation.
- The Amazon ECR repository must exist before you push the image. For more information, see the section called “Creating a repository to store images”.
- Images must be pushed to your repository before you create the Docker manifest. For information about how to push an image, see Pushing a Docker image to an Amazon ECR private repository.

To push a multi-architecture Docker image to an Amazon ECR repository

1. Authenticate your Docker client to the Amazon ECR registry to which you intend to push your image. Authentication tokens must be obtained for each registry used, and the tokens are valid for 12 hours. For more information, see Private registry authentication in Amazon ECR.

   To authenticate Docker to an Amazon ECR registry, run the **aws ecr get-login-password** command. When passing the authentication token to the **docker login** command, use the value `AWS` for the username and specify the Amazon ECR registry URI you want to authenticate to. If authenticating to multiple registries, you must repeat the command for each registry.

> **Important**
>
> If you receive an error, install or upgrade to the latest version of the AWS CLI. For more information, see Installing the AWS Command Line Interface in the *AWS Command Line Interface User Guide*.

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

2. List the images in your repository, confirming the image tags.

```
aws ecr describe-images --repository-name my-repository
```

3. Create the Docker manifest list. The `manifest create` command verifies that the referenced images are already in your repository and creates the manifest locally.

```
docker manifest create aws_account_id.dkr.ecr.region.amazonaws.com/my-
repository aws_account_id.dkr.ecr.region.amazonaws.com/my-
repository:image_one_tag aws_account_id.dkr.ecr.region.amazonaws.com/my-
repository:image_two
```

4. (Optional) Inspect the Docker manifest list. This enables you to confirm the size and digest for each image manifest referenced in the manifest list.

```
docker manifest inspect aws_account_id.dkr.ecr.region.amazonaws.com/my-repository
```

5. Push the Docker manifest list to your Amazon ECR repository.

```
docker manifest push aws_account_id.dkr.ecr.region.amazonaws.com/my-repository
```

Pushing a Helm chart to an Amazon ECR private repository

You can push Open Container Initiative (OCI) artifacts to an Amazon ECR repository. To see an example of this functionality, use the following steps to push a Helm chart to Amazon ECR.

For information about using your Amazon ECR hosted Helm charts with Amazon EKS, see Installing a Helm chart on an Amazon EKS cluster.

To push a Helm chart to an Amazon ECR repository

1. Install the latest version of the Helm client. These steps were written using Helm version `3.8.2`. For more information, see Installing Helm.
2. Use the following steps to create a test Helm chart. For more information, see Helm Docs - Getting Started.

   a. Create a Helm chart named `helm-test-chart` and clear the contents of the `templates` directory.

   ```
   helm create helm-test-chart
   rm -rf ./helm-test-chart/templates/*
   ```

   b. Create a `ConfigMap` in the `templates` folder.

   ```
   cd helm-test-chart/templates
   cat <<EOF > configmap.yaml
   apiVersion: v1
   kind: ConfigMap
   metadata:
     name: helm-test-chart-configmap
   data:
     myvalue: "Hello World"
   EOF
   ```

3. Package the chart. The output will contain the filename of the packaged chart which you use when pushing the Helm chart.

   ```
   cd ../..
   helm package helm-test-chart
   ```

   Output

   ```
   Successfully packaged chart and saved it to: /Users/username/helm-test-chart-0.1.0.tgz
   ```

4. Create a repository to store your Helm chart. The name of your repository should match the name you used when creating the Helm chart in step 2. For more information, see Creating an Amazon ECR private repository to store images.

```
aws ecr create-repository \
     --repository-name helm-test-chart \
     --region us-west-2
```

5. Authenticate your Helm client to the Amazon ECR registry to which you intend to push your Helm chart. Authentication tokens must be obtained for each registry used, and the tokens are valid for 12 hours. For more information, see Private registry authentication in Amazon ECR.

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

6. Push the Helm chart using the **helm push** command. The output should include the Amazon ECR repository URI and SHA digest.

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

7. Describe your Helm chart.

```
aws ecr describe-images \
     --repository-name helm-test-chart \
     --region us-west-2
```

In the output, verify that the `artifactMediaType` parameter indicates the proper artifact type.

```
{
    "imageDetails": [
        {
            "registryId": "aws_account_id",
            "repositoryName": "helm-test-chart",
            "imageDigest":
 "sha256:dd8aebdda7df991a0ffe0b3d6c0cf315fd582cd26f9755a347a52adEXAMPLE",
            "imageTags": [
                "0.1.0"
            ],
            "imageSizeInBytes": 1620,
            "imagePushedAt": "2021-09-23T11:39:30-05:00",
```

```
            "imageManifestMediaType": "application/vnd.oci.image.manifest.v1+json",
            "artifactMediaType": "application/vnd.cncf.helm.config.v1+json"
        }
    ]
}
```

8. (Optional) For additional steps, install the Helm `ConfigMap` and get started with Amazon EKS. For more information, see Installing a Helm chart on an Amazon EKS cluster.

Signing an image stored in an Amazon ECR private repository

Amazon ECR integrates with AWS Signer to provide a way for you to sign your container images. You can store both your container images and the signatures in your private repositories.

Considerations

The following should be considered when using Amazon ECR image signing.

- Signatures stored in your repository count against the service quota for the maximum number of images per repository. For more information, see Amazon ECR service quotas.
- When reference artifacts are present in a repository, Amazon ECR lifecycle policies will automatically clean up those artifacts within 24 hours of the deletion of the subject image.

Prerequisites

Before you begin, The following prerequisites must be met.

- Install and configure the latest version of the AWS CLI. For more information, see Installing or updating the latest version of the AWS CLI in the *AWS Command Line Interface User Guide*.
- Install the Notation CLI and the AWS Signer plugin for Notation. For more information, see Prerequisites for signing container images in the *AWS Signer Developer Guide*.
- Have a container image stored in an Amazon ECR private repository to sign. For more information, see Pushing an image to an Amazon ECR private repository.

Configure authentication for the Notary client

Before you can create a signature using the Notation CLI, you must configure the client so it can authenticate to Amazon ECR. If you have Docker installed on the same host where you install

the Notation client, then Notation will reuse the same authentication method you use for the Docker client. The Docker `login` and `logout` commands will allow the Notation `sign` and `verify` commands to use those same credentials, and you don't have to separately authenticate Notation. For more information on configuring your Notation client for authentication, see Authenticate with OCI-compliant registries in the Notary Project documentation.

If you are not using Docker or another tool that uses Docker credentials, then we recommend using the Amazon ECR Docker Credential Helper as your credential store. For more information on how to install and configure the Amazon ECR Credential Helper, see Amazon ECR Docker Credential Helper.

Signing an image

The following steps can be used to create the resources necessary to sign a container image and store the signature in an Amazon ECR private repository. Notation signs images using the digest.

To sign an image

1. Create an AWS Signer signing profile using the `Notation-OCI-SHA384-ECDSA` signing platform. You can optionally specify a signature validity period using the `--signature-validity-period` parameter. This value may be specified using `DAYS`, `MONTHS`, or `YEARS`. If no validity period is specified, the default value of 135 months is used.

   ```
   aws signer put-signing-profile --profile-name ecr_signing_profile --platform-id
    Notation-OCI-SHA384-ECDSA
   ```

   > **Note**
   >
   > The signing profile name only supports alphanumeric characters and the underscore (_).

2. Authenticate the Notation client to your default registry. The following example uses the AWS CLI to authenticate the Notation CLI to an Amazon ECR private registry.

   ```
   aws ecr get-login-password --region region | notation login --username AWS --
   password-stdin 111122223333.dkr.ecr.region.amazonaws.com
   ```

3. Use the Notation CLI to sign the image, specifying the image using the repository name and the SHA digest. This creates the signature and pushes it to the same Amazon ECR private repository that the image being signed is in.

   In the following example, we are signing an image in the `curl` repository with SHA digest `sha256:ca78e5f730f9a789ef8c63bb55275ac12dfb9e8099e6EXAMPLE`.

   ```
   notation
    sign 111122223333.dkr.ecr.region.amazonaws.com/
   curl@sha256:ca78e5f730f9a789ef8c63bb55275ac12dfb9e8099e6EXAMPLE --plugin
    "com.amazonaws.signer.notation.plugin" --id "arn:aws:signer:region:111122223333:/
   signing-profiles/ecrSigningProfileName"
   ```

Next steps

After you sign your container image, you can verify the signature locally. For instructions about verifying an image, see Verify an image locally after signing in the *AWS Signer Developer Guide*.

Deleting signatures and other artifacts from an Amazon ECR private repository

You can use the ORAS client to list and delete signatures and other reference type artifacts from an Amazon ECR private repository. Deleting signatures and other reference artifacts is similar to how an image is deleted (see Deleting an image in Amazon ECR). Here is how to list artifacts and delete signatures:

To manage image artifacts using the ORAS CLI

1. Install and configure the ORAS client.

   For information about installing and configuring the ORAS client, see Installation in the ORAS documentation.

2. To list available artifacts for an Amazon ECR image, use `oras discover`, followed by an image name:

   ```
   oras discover 111222333444.dkr.ecr.us-east-1.amazonaws.com/oci:helloworld
   ```

The output should look similar to this:

```
111222333444.dkr.ecr.us-east-1.amazonaws.com/
oci@sha256:88c0c54329bfdc1d94d6f58cd3fcb1226d46f58670f44a8c689cb3c9b37b6925
### application/vnd.cncf.notary.signature
    ### sha256:387c10c1598ee18aae81dcfc86d0d06d116e46461d1c3cda8927e69c48108c42
    ### sha256:6527bcec87adf1d55460666183b9d0968b3cd4e4bc34602d485206a219851171
```

3. To delete a signature using the ORAS CLI, given the previous example, run the following command:

```
oras manifest delete 111222333444.dkr.ecr.us-east-1.amazonaws.com/
oci@sha256:387c10c1598ee18aae81dcfc86d0d06d116e46461d1c3cda8927e69c48108c42
```

The output should look similar to this:

```
Are you sure you want to delete the manifest "111222333444.dkr.ecr.us-
east-1.amazonaws.com/
oci@sha256:387c10c1598ee18aae81dcfc86d0d06d116e46461d1c3cda8927e69c48108c42" and
 all tags associated with it? [y/N] y
```

4. Press y. The artifact should be deleted.

To troubleshoot artifact deletion

If a signature deletion, such as the one just shown, should fail, output similar to the following appears.

```
Error response from registry: failed to delete 111222333444.dkr.ecr.us-
east-1.amazonaws.com/
oci@sha256:387c10c1598ee18aae81dcfc86d0d06d116e46461d1c3cda8927e69c48108c42:
unsupported: Requested image referenced by manifest list:
 [sha256:005e2c97a6373e483799fa4ff29ac64a42dd10f08efcc166d6775f9b74943b5b]
```

This failure can happen when deleting an image pushed before the OCI 1.1 launch. As noted in the error, you must delete the manifest referencing the image before you can delete the image as follows:

1. To delete the manifest associated with the signature you want to delete, type:

```
 oras manifest delete 111222333444.dkr.ecr.us-east-1.amazonaws.com/
oci@sha256:005e2c97a6373e483799fa4ff29ac64a42dd10f08efcc166d6775f9b74943b5b
```

   The output should look similar to this:

```

Are you sure you want to delete the manifest
 "sha256:005e2c97a6373e483799fa4ff29ac64a42dd10f08efcc166d6775f9b74943b5b" and all
 tags associated with it? [y/N] y
```

2. Press y. The manifest should be deleted.
3. With the manifest gone, you can delete the signature:

```
 oras manifest delete 111222333444.dkr.ecr.us-east-1.amazonaws.com/
oci@sha256:387c10c1598ee18aae81dcfc86d0d06d116e46461d1c3cda8927e69c48108c42
```

   The output should look similar to this. Press y.

```

Are you sure you want to delete the manifest
 "sha256:387c10c1598ee18aae81dcfc86d0d06d116e46461d1c3cda8927e69c48108c42" and all
 tags associated with it? [y/N] y
Deleted [registry] 111222333444.dkr.ecr.us-east-1.amazonaws.com/
oci@sha256:387c10c1598ee18aae81dcfc86d0d06d116e46461d1c3cda8927e69c48108c42
```

4. To see that the signature was deleted, type:

```
oras discover 111222333444.dkr.ecr.us-east-1.amazonaws.com/oci:helloworld
```

   The output should look similar to this:

```
111222333444.dkr.ecr.us-east-1.amazonaws.com/
oci@sha256:88c0c54329bfdc1d94d6f58cd3fcb1226d46f58670f44a8c689cb3c9b37b6925
### application/vnd.cncf.notary.signature
    ### sha256:6527bcec87adf1d55460666183b9d0968b3cd4e4bc34602d485206a219851171
```

Viewing image details in Amazon ECR

After you push an image to your repository, you can view information about it. The details included are as follows:

- Image URI
- Image tags
- Artifact media type
- Image manifest type
- Scanning status
- The size of the image in MB
- When the image was pushed to the repository
- The replication status

To view image details (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region that contains the repository containing your image.
3. In the navigation pane, choose **Repositories**.
4. On the **Repositories** page, choose the repository to view.
5. On the **Repositories : *repository_name*** page, choose the image to view the details of.

Pulling an image to your local environment from an Amazon ECR private repository

If you want to run a Docker image that is available in Amazon ECR, you can pull it to your local environment with the **docker pull** command. You can do this from either your default registry or from a registry associated with another AWS account.

To use an Amazon ECR image in an Amazon ECS task definition, see Using Amazon ECR images with Amazon ECS.

> **Important**
>
> Amazon ECR requires that users have permission to make calls to the `ecr:GetAuthorizationToken` API through an IAM policy before they can authenticate to a registry and push or pull any images from any Amazon ECR repository. Amazon ECR provides several AWS managed policies to control user access at varying levels. For information about the AWS managed policies for Amazon ECR, see AWS managed policies for Amazon Elastic Container Registry.

To pull a Docker image from an Amazon ECR repository

1. Authenticate your Docker client to the Amazon ECR registry that you intend to pull your image from. Authentication tokens must be obtained for each registry used, and the tokens are valid for 12 hours. For more information, see Private registry authentication in Amazon ECR.
2. (Optional) Identify the image to pull.
   - You can list the repositories in a registry with the **aws ecr describe-repositories** command:

     ```
     aws ecr describe-repositories
     ```

     The example registry above has a repository called `amazonlinux`.
   - You can describe the images within a repository with the **aws ecr describe-images** command:

     ```
     aws ecr describe-images --repository-name amazonlinux
     ```

The example repository above has an image tagged as `latest` and `2016.09`, with the image digest `sha256:f1d4ae3f7261a72e98c6ebefe9985cf10a0ea5bd762585a43e0700ed99863807`.

3. Pull the image using the **docker pull** command. The image name format should be *registry*/*repository*[:*tag*] to pull by tag, or *registry*/*repository*[@*digest*] to pull by digest.

```
docker pull aws_account_id.dkr.ecr.us-west-2.amazonaws.com/amazonlinux:latest
```

> **Important**
>
> If you receive a *repository-url* `not found: does not exist or no pull access` error, you might need to authenticate your Docker client with Amazon ECR. For more information, see Private registry authentication in Amazon ECR.

Pulling the Amazon Linux container image

The Amazon Linux container image is built from the same software components that are included in the Amazon Linux AMI. The Amazon Linux container image is available for use in any environment as a base image for Docker workloads. If you use the Amazon Linux AMI for applications in Amazon EC2, you can containerize your applications with the Amazon Linux container image.

You can use the Amazon Linux container image in your local development environment and then push your application to AWS using Amazon ECS. For more information, see Using Amazon ECR images with Amazon ECS.

The Amazon Linux container image is available on Amazon ECR Public and on Docker Hub. For support for the Amazon Linux container image, go to the AWS developer forums.

To pull the Amazon Linux container image from Amazon ECR Public

1. Authenticate your Docker client to the Amazon Linux Public registry. Authentication tokens are valid for 12 hours. For more information, see Private registry authentication in Amazon ECR.

> **Note**
>
> The **ecr-public** commands are available in the AWS CLI starting with version `1.18.1.187`, however we recommend using the latest version of the AWS CLI. For more information, see Installing the AWS Command Line Interface in the *AWS Command Line Interface User Guide*.

```
aws ecr-public get-login-password --region us-east-1 | docker login --username AWS
 --password-stdin public.ecr.aws
```

The output is as follows:

```
Login succeeded
```

2. Pull the Amazon Linux container image using the **docker pull** command. To view the Amazon Linux container image on the Amazon ECR Public Gallery, see Amazon ECR Public Gallery - amazonlinux.

```
docker pull public.ecr.aws/amazonlinux/amazonlinux:latest
```

3. (Optional) Run the container locally.

```
docker run -it public.ecr.aws/amazonlinux/amazonlinux /bin/bash
```

To pull the Amazon Linux container image from Docker Hub

1. Pull the Amazon Linux container image using the **docker pull** command.

```
docker pull amazonlinux
```

2. (Optional) Run the container locally.

```
docker run -it amazonlinux:latest /bin/bash
```

Deleting an image in Amazon ECR

If you're finished using an image, you can delete it from your repository. If you're finished with a repository, you can delete the entire repository and all of the images within it. For more information, see Deleting a private repository in Amazon ECR.

As an alternative to deleting images manually, you can create repository lifecycle policies which provide more control over the lifecycle management of images in your repositories. Lifecycle policies automate this process for you. For more information, see Automate the cleanup of images by using lifecycle policies in Amazon ECR.

> **Note**
>
> If your repository has a mix of images, some of which were pushed before Amazon ECR supported OCI v1.1, some signatures will have image indexes or manifest lists pointing to them. As a result, when you delete a pre-OCI v1.1 image, you might need to manually delete the manifest list that references the image in order to delete the artifact.

To delete an image (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region that contains the image to delete.
3. In the navigation pane, choose **Repositories**.
4. On the **Repositories** page, choose the repository that contains the image to delete.
5. On the **Repositories: *repository_name*** page, select the box to the left of the image to delete and choose **Delete**.
6. In the **Delete image(s)** dialog box, verify that the selected images should be deleted and choose **Delete**.

To delete an image (AWS CLI)

1. List the images in your repository. Tagged images will have both an image digest as well as a list of associated tags. Untagged images will only have an image digest.

```
aws ecr list-images \
```

```
    --repository-name my-repo
```

2. (Optional) Delete any unwanted tags for the image by specifying the tag associated with the image you want to delete. When the last tag is deleted from an image, the image is also deleted.

```
aws ecr batch-delete-image \
    --repository-name my-repo \
    --image-ids imageTag=tag1 imageTag=tag2
```

3. Delete a tagged or untagged image by specifying the image digest. When you delete an image by referencing its digest, the image and all of its tags are deleted.

```
aws ecr batch-delete-image \
    --repository-name my-repo \
    --image-ids imageDigest=sha256:4f70ef7a4d29e8c0c302b13e25962d8f7a0bd304EXAMPLE
```

   To delete multiple images, you can specify multiple image tags or image digests in the request.

```
aws ecr batch-delete-image \
    --repository-name my-repo \
    --image-ids imageDigest=sha256:4f70ef7a4d29e8c0c302b13e25962d8f7a0bd304EXAMPLE
 imageDigest=sha256:f5t0e245ssffc302b13e25962d8f7a0bd304EXAMPLE
```

Retagging an image in Amazon ECR

With Docker Image Manifest V2 Schema 2 images, you can use the `--image-tag` option of the **put-image** command to retag an existing image. You can retag without pulling or pushing the image with Docker. For larger images, this process saves a considerable amount of network bandwidth and time required to retag an image.

To retag an image (AWS CLI)

To retag an image with the AWS CLI

1. Use the **batch-get-image** command to get the image manifest for the image to retag and write it to a file. In this example, the manifest for an image with the tag, *latest*, in the repository, *amazonlinux*, is written to an environment variable named *MANIFEST*.

```
MANIFEST=$(aws ecr batch-get-image --repository-name amazonlinux --image-ids
 imageTag=latest --output text --query 'images[].imageManifest')
```

2. Use the `--image-tag` option of the **put-image** command to put the image manifest to Amazon ECR with a new tag. In this example, the image is tagged as *2017.03*.

> **Note**
>
> If the `--image-tag` option isn't available in your version of the AWS CLI, upgrade to the latest version. For more information, see Installing the AWS Command Line Interface in the *AWS Command Line Interface User Guide*.

```
aws ecr put-image --repository-name amazonlinux --image-tag 2017.03 --image-
manifest "$MANIFEST"
```

3. Verify that your new image tag is attached to your image. In the following output, the image has the tags `latest` and `2017.03`.

```
aws ecr describe-images --repository-name amazonlinux
```

The output is as follows:

```
{
    "imageDetails": [
        {
            "imageSizeInBytes": 98755613,
            "imageDigest":
 "sha256:8d00af8f076eb15a33019c2a3e7f1f655375681c4e5be157a26EXAMPLE",
            "imageTags": [
                "latest",
                "2017.03"
            ],
            "registryId": "aws_account_id",
            "repositoryName": "amazonlinux",
            "imagePushedAt": 1499287667.0
        }
    ]
}
```

To retag an image (AWS Tools for Windows PowerShell)

To retag an image with the AWS Tools for Windows PowerShell

1. Use the **Get-ECRImageBatch cmdlet** to obtain the description of the image to retag and write it to an environment variable. In this example, an image with the tag, *latest*, in the repository, *amazonlinux*, is written to the environment variable, *$Image*.

   > **Note**
   >
   > If you don't have the **Get-ECRImageBatch cmdlet** available on your system, see Setting up the AWS Tools for Windows PowerShell in the *AWS Tools for PowerShell User Guide*.

   ```
   $Image = Get-ECRImageBatch -ImageId @{ imageTag="latest" } -
   RepositoryName amazonlinux
   ```

2. Write the manifest of the image to the *$Manifest* environment variable.

   ```
   $Manifest = $Image.Images[0].ImageManifest
   ```

3. Use the `-ImageTag` option of the **Write-ECRImage cmdlet** to put the image manifest to Amazon ECR with a new tag. In this example, the image is tagged as *2017.09*.

   ```
   Write-ECRImage -RepositoryName amazonlinux -ImageManifest $Manifest -
   ImageTag 2017.09
   ```

4. Verify that your new image tag is attached to your image. In the following output, the image has the tags `latest` and `2017.09`.

   ```
   Get-ECRImage -RepositoryName amazonlinux
   ```

   The output is as follows:

   ```
   ImageDigest                                                             ImageTag
   -----------                                                             --------
   sha256:359b948ea8866817e94765822787cd482279eed0c17bc674a7707f4256d5d497 latest
   sha256:359b948ea8866817e94765822787cd482279eed0c17bc674a7707f4256d5d497 2017.09
   ```

Preventing image tags from being overwritten in Amazon ECR

You can prevent image tags from being overwritten by turning on tag immutability in a repository. After tag immutability is turned on, the `ImageTagAlreadyExistsException` error is returned if you push an image with a tag that is already in the repository. Tag immutability affects all tags. You cannot make some tags immutable while others aren't.

You can use the AWS Management Console and AWS CLI tools to set image tag mutability for a new repository or for an existing repository. To create a repository using console steps, see Creating an Amazon ECR private repository to store images.

Setting image tag mutability (AWS Management Console)

To set image tag mutability

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region that contains the repository to edit.
3. In the navigation pane, choose **Repositories** under **Private registry**.

   If you don't see **Repositories**, choose **Private registry** to expand the menu and then choose **Repositories**.
4. On the **Private repositories** page, choose the radio button before the repository name for which you want to set the image tag mutability settings.
5. Choose **Actions** and then choose **Repository** under **Edit**.
6. For **Image tag immutability**, choose one of the following tag mutability settings for the repository.

   - **Mutable** – Choose this option if you want image tags to be overwritten. Recommended for repositories using pull through cache actions to ensure Amazon ECR can update cached images. Additionally, to disable tag updates for a few mutable tags, enter tag names or use wildcards (*) to match multiple similar tags in the **Mutable tag exclusion** text box.
   - **Immutable** – Choose this option if you want to prevent image tags from being overwritten, and it applies to all tags and exclusions in the repository when pushing an image with existing tag. Amazon ECR returns an `ImageTagAlreadyExistsException` if you attempt to push an image with an existing tag. Additionally, to enable tag updates for a few immutable tags, enter tag names or use wildcards (*) to match multiple similar tags in the **Immutable tag exclusion** text box.

7. For **Image scan settings**, while you can specify the scan settings at the repository level for basic scanning, it is best practice to specify the scan configuration at the private registry level. Specify the scanning settings at the private registry allow you to enable either enhanced scanning or basic scanning as well as define filters to specify which repositories are scanned. For more information, see Scan images for software vulnerabilities in Amazon ECR.
8. For **Encryption settings**, this is a view only field as the encryption settings for a repository can't be changed once the repository is created.
9. Choose **Save** to update the repository settings.

Setting image tag mutability (AWS CLI)

To create a repository with immutable tags configured

Use one of the following commands to create a new image repository with immutable tags configured.

- create-repository (AWS CLI) with image tag mutability

```
aws ecr create-repository --repository-name name --image-tag-mutability IMMUTABLE --
region us-east-2
```

- create-repository (AWS CLI) with image tag mutability exclusion filters

```
aws ecr create-repository --repository-name name --image-tag-
mutability IMMUTABLE_WITH_EXCLUSION --image-tag-mutability-exclusion-filters
 filterType=WILDCARD,filter=filter-text --region us-east-2
```

- New-ECRRepository (AWS Tools for Windows PowerShell) with image tag mutability

```
New-ECRRepository -RepositoryName name -ImageTagMutability IMMUTABLE -Region us-
east-2 -Force
```

- New-ECRRepository (AWS Tools for Windows PowerShell) with image tag mutability exclusion filters

```
New-ECRRepository -RepositoryName name -ImageTagMutability IMMUTABLE_WITH_EXCLUSION
 -ImageTagMutabilityExclusionFilter @{FilterType=WILDCARD Filter=filter-text} -
Region us-east-2 -Force
```

To update the image tag mutability settings for a repository

Use one of the following commands to update the image tag mutability settings for an existing repository.

- put-image-tag-mutability (AWS CLI) with image tag mutability

```
aws ecr put-image-tag-mutability --repository-name name --image-tag-
mutability IMMUTABLE --region us-east-2
```

- put-image-tag-mutability (AWS CLI) with image tag mutability exclusion filters

```
aws ecr put-image-tag-mutability --repository-name name --image-tag-
mutability IMMUTABLE_WITH_EXCLUSION --image-tag-mutability-exclusion-filters
 filterType=WILDCARD,filter=latest --region us-east-2
```

- Write-ECRImageTagMutability (AWS Tools for Windows PowerShell) with image tag mutability

```
Write-ECRImageTagMutability -RepositoryName name -ImageTagMutability IMMUTABLE -
Region us-east-2 -Force
```

- Write-ECRImageTagMutability (AWS Tools for Windows PowerShell) with image tag mutability exclusion filters

```
Write-ECRImageTagMutability -RepositoryName name -
ImageTagMutability IMMUTABLE_WITH_EXCLUSION -ImageTagMutabilityExclusionFilter
 @{FilterType=WILDCARD Filter=latest}
```

Container image manifest format support in Amazon ECR

Amazon ECR supports the following container image manifest formats:

- Docker Image Manifest V2 Schema 1 (used with Docker version 1.9 and older)
- Docker Image Manifest V2 Schema 2 (used with Docker version 1.10 and newer)
- Open Container Initiative (OCI) Specifications (v1.0 and v1.1)

Support for Docker Image Manifest V2 Schema 2 provides the following functionality:

- The ability to use multiple tags for a singular image.

- Support for storing Windows container images.

Amazon ECR image manifest conversion

When you push and pull images to and from Amazon ECR, your container engine client (for example, Docker) communicates with the registry to agree on a manifest format that is understood by the client and the registry to use for the image.

When you push an image to Amazon ECR with Docker version 1.9 or earlier, the image manifest format is stored as Docker Image Manifest V2 Schema 1. When you push an image to Amazon ECR with Docker version 1.10 or later, the image manifest format is stored as Docker Image Manifest V2 Schema 2.

When you pull an image from Amazon ECR *by tag*, Amazon ECR returns the image manifest format that is stored in the repository. The format is returned only if that format is understood by the client. If the stored image manifest format isn't understood by the client, Amazon ECR converts the image manifest into a format that is understood. For example, if a Docker 1.9 client requests an image manifest that is stored as Docker Image Manifest V2 Schema 2, Amazon ECR returns the manifest in the Docker Image Manifest V2 Schema 1 format. The following table describes the available conversions supported by Amazon ECR when an image is pulled *by tag*:

Schema requested by client	Pushed to ECR as V2, schema 1	Pushed to ECR as V2, schema 2	Pushed to ECR as OCI
V2, schema 1	No translation required	Translated to V2, schema 1	No translation available
V2, schema 2	No translation available, client falls back to V2, schema 1	No translation required	Translated to V2, schema 2
OCI	No translation available	Translated to OCI	No translation required

Important

If you pull an image *by digest*, there is no translation available. Your client must understand the image manifest format that is stored in Amazon ECR. If you request a Docker Image Manifest V2 Schema 2 image by digest on a Docker 1.9 or older client, the image pull fails. For more information, see Registry compatibility in the Docker documentation.
In this example, if you request the same image *by tag*, Amazon ECR translates the image manifest into a format that the client can understand. The image pull succeeds.

Using Amazon ECR images with Amazon ECS

You can use your Amazon ECR private repositories to host container images and artifacts that your Amazon ECS tasks may pull from. For this to work, the Amazon ECS, or Fargate, container agent must have permissions to make the `ecr:BatchGetImage`, `ecr:GetDownloadUrlForLayer`, and `ecr:GetAuthorizationToken` APIs.

Required IAM permissions

The following table shows the IAM role to use, for each launch type, that provides the required permissions for your tasks to pull from an Amazon ECR private repository. Amazon ECS provides managed IAM policies that include the required permissions.

Launch type	IAM role	AWS managed IAM policy
Amazon ECS on Amazon EC2 instances	Use the container instance IAM role, which is associated with the Amazon EC2 instance registered to your Amazon ECS cluster. For more information, see Container instance IAM role in the *Amazon Elastic Container Service Developer Guide*.	`AmazonEC2ContainerServiceforEC2Role` For more information, see AmazonEC2ContainerServiceforEC2Role in the *Amazon Elastic Container Service Developer Guide*
Amazon ECS on Fargate	Use the task execution IAM role that you reference	`AmazonECSTaskExecutionRolePolicy`

Launch type	IAM role	AWS managed IAM policy
	in your Amazon ECS task definition. For more information, see Task execution IAM role in the *Amazon Elastic Container Service Developer Guide*.	For more information, see AmazonECSTaskExecutionRolePolicy in the *Amazon Elastic Container Service Developer Guide*.
Amazon ECS on external instances	Use the container instance IAM role, which is associated with the on-premises server or virtual machine (VM) registered to your Amazon ECS cluster. For more information, see Container instance Amazon ECS role in the *Amazon Elastic Container Service Developer Guide*.	`AmazonEC2ContainerServiceforEC2Role` For more information, see AmazonEC2ContainerServiceforEC2Role in the *Amazon Elastic Container Service Developer Guide*.

Important

The AWS managed IAM policies contain additional permissions that you may not require for your use. In this case, these are the minimum required permissions to pull from an Amazon ECR private repository.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecr:BatchGetImage",
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetAuthorizationToken"
            ],
```

```
            "Resource": "*"
        }
    ]
}
```

Specifying an Amazon ECR image in an Amazon ECS task definition

When creating an Amazon ECS task definition, you can specify a container image hosted in an Amazon ECR private repository. In the task definition, ensure that you use the full `registry/repository:tag` naming for your Amazon ECR images. For example, *aws_account_id*.dkr.ecr.*region*.amazonaws.com/*my-repository*:*latest*.

The following task definition snippet shows the syntax you would use to specify a container image hosted in Amazon ECR in your Amazon ECS task definition.

```
{
    "family": "task-definition-name",
    ...
    "containerDefinitions": [
        {
            "name": "container-name",
            "image": "aws_account_id.dkr.ecr.region.amazonaws.com/my-
repository:latest",
            ...
        }
    ],
    ...
}
```

Using Amazon ECR Images with Amazon EKS

You can use your Amazon ECR images with Amazon EKS.

When referencing an image from Amazon ECR, you must use the full `registry/repository:tag` naming for the image. For example, *aws_account_id*.dkr.ecr.*region*.amazonaws.com/*my-repository*:*latest*.

Required IAM permissions

If you have Amazon EKS workloads hosted on managed nodes, self-managed nodes, or AWS Fargate, review the following:

- Amazon EKS workloads hosted on managed or self-managed nodes: The Amazon EKS worker node IAM role (`NodeInstanceRole`) is required. The Amazon EKS worker node IAM role must contain the following IAM policy permissions for Amazon ECR.

  JSON

  ```
  {
      "Version": "2012-10-17",
      "Statement": [
          {
              "Effect": "Allow",
              "Action": [
                  "ecr:BatchCheckLayerAvailability",
                  "ecr:BatchGetImage",
                  "ecr:GetDownloadUrlForLayer",
                  "ecr:GetAuthorizationToken"
              ],
              "Resource": "*"
          }
      ]
  }
  ```

  Note

  If you used `eksctl` or the AWS CloudFormation templates in Getting Started with Amazon EKS to create your cluster and worker node groups, these IAM permissions are applied to your worker node IAM role by default.

- Amazon EKS workloads hosted on AWS Fargate: Use the Fargate pod execution role, which provides your pods permission to pull images from private Amazon ECR repositories. For more information, see Create a Fargate pod execution role.

Installing a Helm chart on an Amazon EKS cluster

Helm charts hosted in Amazon ECR can be installed on your Amazon EKS clusters.

Prerequisites

- Install the latest version of the Helm client. These steps were written using Helm version `3.9.0`. For more information, see Installing Helm.
- You have at least version `1.23.9` or `2.6.3` of the AWS CLI installed on your computer. For more information, see Installing or updating the latest version of the AWS CLI.
- You have pushed a Helm chart to your Amazon ECR repository. For more information, see Pushing a Helm chart to an Amazon ECR private repository.
- You have configured `kubectl` to work with Amazon EKS. For more information, see Create a `kubeconfig` for Amazon EKS in the Amazon EKS User Guide. If the following commands succeeds for your cluster, you're properly configured.

```
kubectl get svc
```

To install a Helm chart on an Amazon EKS cluster

1. Authenticate your Helm client to the Amazon ECR registry that your Helm chart is hosted. Authentication tokens must be obtained for each registry used, and the tokens are valid for 12 hours. For more information, see Private registry authentication in Amazon ECR.

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

2. Install the chart. Replace *helm-test-chart* with your repository and *0.1.0* with your Helm chart's tag.

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

   The output should look similar to this:

   ```
   NAME: ecr-chart-demo
   LAST DEPLOYED: Tue May 31 17:38:56 2022
   NAMESPACE: default
   STATUS: deployed
   REVISION: 1
   ```

```
TEST SUITE: None
```

3. Verify the chart installation.

```
helm list -n default
```

Example output:

```
NAME            NAMESPACE       REVISION        UPDATED
     STATUS          CHART                   APP VERSION
ecr-chart-demo  default         1               2022-06-01 15:56:40.128669157 +0000
 UTC deployed        helm-test-chart-0.1.0   1.16.0
```

4. (Optional) See the installed Helm chart `ConfigMap`.

```
kubectl describe configmap helm-test-chart-configmap
```

5. When you are finished, you can remove the chart release from your cluster.

```
helm uninstall ecr-chart-demo
```

Scan images for software vulnerabilities in Amazon ECR

Amazon ECR image scanning helps to identify software vulnerabilities in your container images. The following scanning types are offered.

> **Important**
>
> Switching between the **Enhanced scanning**, **Basic scanning**, and the **Improved basic scanning** versions will cause previously established scans to no longer be available. You will have to set up your scans again. However, if you switch back to your previous scanning version the established scans will be available.

- **Enhanced scanning** – Amazon ECR integrates with Amazon Inspector to provide automated, continuous scanning of your repositories. Your container images are scanned for both operating systems and programming language package vulnerabilities. As new vulnerabilities appear, the scan results are updated and Amazon Inspector emits an event to EventBridge to notify you. Enhanced scanning provides the following:
  - OS and programming languages package vulnerabilities
  - Two scanning frequencies: Scan on push and continuous scan
- **Basic scanning** – Amazon ECR provides two versions of basic scanning which use the Common Vulnerabilities and Exposures (CVEs) database:.
  - **AWS native basic scanning** – Uses AWS native technology, which is now GA and recommended. All new customer registries are opted into this improved version by default.
  - **Clair basic scanning** – Uses the open-source Clair project and is deprecated.

  With basic scanning, you configure your repositories to scan on push or you can perform manual scans and Amazon ECR provides a list of scan findings. Basic scanning provides the following:
  - OS scans
  - Two scanning frequencies: Manual and scan on push

  > **Important**
  >
  > The new version of Amazon ECR Basic Scanning doesn't use the `imageScanFindingsSummary` and `imageScanStatus` attributes from the `DescribeImages` API response to return scan results. Use the

`DescribeImageScanFindings` API instead. For more information, see `DescribeImageScanFindings`.

Filters to choose which repositories are scanned in Amazon ECR

When you configure image scanning for your private registry, you can use filters to choose which repositories are scanned.

When **basic** scanning is used, you may specify scan on push filters to specify which repositories are set to do an image scan when new images are pushed. Any repositories not matching a basic scanning scan on push filter will be set to the **manual** scan frequency which means to perform a scan, you must manually trigger the scan.

When **enhanced** scanning is used, you may specify separate filters for scan on push and continuous scanning. Any repositories not matching an enhanced scanning filter will have scanning disabled. If you are using enhanced scanning and specify separate filters for scan on push and continuous scanning where multiple filters match the same repository, then Amazon ECR enforces the continuous scanning filter over the scan on push filter for that repository.

Filter wildcards

When a filter is specified, a filter with no wildcard will match all repository names that contain the filter. A filter with a wildcard (*) matches on any repository name where the wildcard replaces zero or more characters in the repository name.

The following table provides examples where repository names are expressed on the horizontal axis and example filters are specified on the vertical axis.

	prod	**repo-prod**	**prod-repo**	**repo-prod-repo**	**prodrepo**
prod	Yes	Yes	Yes	Yes	Yes
*prod	Yes	Yes	No	No	No
prod*	Yes	No	Yes	No	Yes
prod	Yes	Yes	Yes	Yes	Yes

	prod	repo-prod	prod-repo	repo-prod-repo	prodrepo
prod*repo	No	No	Yes	No	Yes

Scan images for OS and programming language package vulnerabilities in Amazon ECR

Amazon ECR enhanced scanning is an integration with Amazon Inspector which provides vulnerability scanning for your container images. Your container images are scanned for both operating systems and programming language package vulnerabilities. You can view the scan findings with both Amazon ECR and with Amazon Inspector directly. For more information about Amazon Inspector, see Scanning container images with Amazon Inspector in the *Amazon Inspector User Guide*.

With enhanced scanning, you can choose which repositories are configured for automatic, continuous scanning and which are configured for scan on push. This is done by setting scan filters.

Considerations for enhanced scanning

Consider the following before enabling Amazon ECR enhanced scanning.

- There is no additional cost from Amazon ECR to use this feature, however there is a cost from Amazon Inspector to scan your images. This feature is available in Regions where Amazon Inspector is supported. For more information, see:
  - Amazon Inspector pricing – Amazon Inspector pricing.
  - Amazon Inspector supported Regions – Regions and endpoints.
- Amazon ECR enhanced scanning shows how images are used on Amazon EKS and Amazon ECS. You can see when images were last used and identify how many clusters use each image. This information helps you prioritize vulnerability remediation for actively used images. You can quickly determine which clusters might be affected by newly discovered vulnerabilities. For more information about how to request these information and view the response, see `DescribeImageScanFindings`.
- Amazon Inspector supports scanning for specific operating systems. For a full list, see Supported operating systems - Amazon ECR scanning in the *Amazon Inspector User Guide*.

- Amazon Inspector uses a service-linked IAM role, which provides the permissions needed to provide enhanced scanning for your repositories. The service-linked IAM role is created automatically by Amazon Inspector when enhanced scanning is turned on for your private registry. For more information, see Using service-linked roles for Amazon Inspector in the *Amazon Inspector User Guide*.
- When you initially turn on enhanced scanning for your private registry, Amazon Inspector only recognizes images pushed to Amazon ECR in the last 14 days, based on the image push timestamp. Older images will have the `SCAN_ELIGIBILITY_EXPIRED` scan status. If you'd like these images to be scanned by Amazon Inspector you should push them again to your repository.
- When enhanced scanning is turned on for your Amazon ECR private registry, repositories matching the scan filters are scanned using enhanced scanning only. Any repositories that don't match a filter will have an `Off` scan frequency and won't be scanned. Manual scans using enhanced scanning aren't supported. For more information, see Filters to choose which repositories are scanned in Amazon ECR.
- If you specify separate filters for scan on push and continuous scanning where multiple filters match the same repository, then Amazon ECR enforces the continuous scanning filter over the scan on push filter for that repository.
- When enhanced scanning is turned on, Amazon ECR sends an event to EventBridge when the scan frequency for a repository is changed. Amazon Inspector emits events to EventBridge when an initial scan is completed and when an image scan finding is created, updated, or closed.

Changing the enhanced scanning duration for images in Amazon Inspector

After enabling enhanced scanning, Amazon ECR continually scans newly pushed images for the configured duration. By default, Amazon Inspector monitors your repositories until images are deleted or enhanced scanning is disabled. You can configure both push date duration (up to Lifetime) and re-scan duration in the Amazon Inspector console to suit your environment's needs. When the scan duration for a repository elapses, the scan status shows as `SCAN_ELIGIBILITY_EXPIRED`. For more information about configuring re-scan duration settings for Amazon ECR in Amazon Inspector, see Configuring the Amazon ECR re-scan duration in the *Amazon Inspector User Guide*.

IAM permissions required for enhanced scanning in Amazon ECR

Amazon ECR enhanced scanning requires an Amazon Inspector service-linked IAM role and that the IAM principal enabling and using enhanced scanning has permissions to call the Amazon Inspector APIs needed for scanning. The Amazon Inspector service-linked IAM role is created automatically by Amazon Inspector when enhanced scanning is turned on for your private registry. For more information, see Using service-linked roles for Amazon Inspector in the *Amazon Inspector User Guide*.

The following IAM policy grants the required permissions for enabling and using enhanced scanning. It includes the permission needed for Amazon Inspector to create the service-linked IAM role as well as the Amazon Inspector API permissions needed to turned on and off enhanced scanning and retrieve the scan findings.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "inspector2:Enable",
                "inspector2:Disable",
                "inspector2:ListFindings",
                "inspector2:ListAccountPermissions",
                "inspector2:ListCoverage"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": [
                        "inspector2.amazonaws.com"
                    ]
                }
            }
```

```
        }
    ]
}
```

Configuring enhanced scanning for images in Amazon ECR

Configure enhanced scanning per Region for your private registry.

Verify that you have the proper IAM permissions to configure enhanced scanning. For information, see IAM permissions required for enhanced scanning in Amazon ECR.

AWS Management Console

To turn on enhanced scanning for your private registry

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region to set the scanning configuration for.
3. In the navigation pane, choose **Private registry**, and then choose **Settings**.
4. On the **Scanning configuration** page, for **Scan type** choose **Enhanced scanning**.

   By default, when **Enhanced scanning** is selected, all of your repositories are continuously scanned.
5. To choose specific repositories to continuously scan, clear the **Continuously scan all repositories** box, and then define your filters:

   > **Important**
   >
   > Filters with no wildcard will match all repository names that contain the filter. Filters with wildcards (*) match on a repository name where the wildcard replaces zero or more characters in the repository name. To see examples of how filters behave, see the section called “Filter wildcards”.

   a. Enter a filter based on repository names, and then choose **Add filter**.

   b. Decide which repositories to scan when an image is pushed:

   - To scan all repositories on push, select **Scan on push all repositories**.

- To choose specific repositories to scan on push, enter a filter based on repository names, and then choose **Add filter**.

6. Choose **Save**.
7. Repeat these steps in each Region in which you want to turn on enhanced scanning.

AWS CLI

Use the following AWS CLI command to turn on enhanced scanning for your private registry using the AWS CLI. You can specify scan filters using the `rules` object.

- put-registry-scanning-configuration (AWS CLI)

  The following example turns on enhanced scanning for your private registry. By default, when no `rules` are specified, Amazon ECR sets the scanning configuration to continuous scanning for all repositories.

```
aws ecr put-registry-scanning-configuration \
     --scan-type ENHANCED \
     --region us-east-2
```

  The following example turns on enhanced scanning for your private registry and specifies a scan filter. The scan filter in the example turns on continuous scanning for all repositories with `prod` in its name.

```
aws ecr put-registry-scanning-configuration \
     --scan-type ENHANCED \
     --rules '[{"repositoryFilters" : [{"filter":"prod","filterType" :
 "WILDCARD"}],"scanFrequency" : "CONTINUOUS_SCAN"}]' \
     --region us-east-2
```

  The following example turns on enhanced scanning for your private registry and specifies multiple scan filters. The scan filters in the example turns on continuous scanning for all repositories with `prod` in its name and turns on scan on push only for all other repositories.

```
aws ecr put-registry-scanning-configuration \
     --scan-type ENHANCED \
```

```
    --rules '[{"repositoryFilters" : [{"filter":"prod","filterType" :
"WILDCARD"}],"scanFrequency" : "CONTINUOUS_SCAN"},{"repositoryFilters" :
[{"filter":"*","filterType" : "WILDCARD"}],"scanFrequency" : "SCAN_ON_PUSH"}]' \
    --region us-west-2
```

EventBridge events sent for enhanced scanning in Amazon ECR

When enhanced scanning is turned on, Amazon ECR sends an event to EventBridge when the scan frequency for a repository is changed. Amazon Inspector sends events to EventBridge when an initial scan is completed and when an image scan finding is created, updated, or closed.

Event for a repository scan frequency change

When enhanced scanning is turned on for your registry, the following event is sent by Amazon ECR when there is a change with a resource that has enhanced scanning turned on. This includes new repositories being created, the scan frequency for a repository being changed, or when images are created or deleted in repositories with enhanced scanning turned on. For more information, see Scan images for software vulnerabilities in Amazon ECR.

```
{
 "version": "0",
 "id": "0c18352a-a4d4-6853-ef53-0abEXAMPLE",
 "detail-type": "ECR Scan Resource Change",
 "source": "aws.ecr",
 "account": "123456789012",
 "time": "2021-10-14T20:53:46Z",
 "region": "us-east-1",
 "resources": [],
 "detail": {
  "action-type": "SCAN_FREQUENCY_CHANGE",
  "repositories": [{
    "repository-name": "repository-1",
    "repository-arn": "arn:aws:ecr:us-east-1:123456789012:repository/repository-1",
    "scan-frequency": "SCAN_ON_PUSH",
    "previous-scan-frequency": "MANUAL"
   },
   {
    "repository-name": "repository-2",
    "repository-arn": "arn:aws:ecr:us-east-1:123456789012:repository/repository-2",
    "scan-frequency": "CONTINUOUS_SCAN",
    "previous-scan-frequency": "SCAN_ON_PUSH"
```

```
    },
    {
     "repository-name": "repository-3",
     "repository-arn": "arn:aws:ecr:us-east-1:123456789012:repository/repository-3",
     "scan-frequency": "CONTINUOUS_SCAN",
     "previous-scan-frequency": "SCAN_ON_PUSH"
    }
   ],
   "resource-type": "REPOSITORY",
   "scan-type": "ENHANCED"
  }
}
```

Event for an initial image scan (enhanced scanning)

When enhanced scanning is turned on for your registry, the following event is sent by Amazon Inspector when the initial image scan is completed. The `finding-severity-counts` parameter will only return a value for a severity level if one exists. For example, if the image contains no findings at `CRITICAL` level, then no critical count is returned. For more information, see Scan images for OS and programming language package vulnerabilities in Amazon ECR.

Event pattern:

```
{
  "source": ["aws.inspector2"],
  "detail-type": ["Inspector2 Scan"]
}
```

Example output:

```
{
    "version": "0",
    "id": "739c0d3c-4f02-85c7-5a88-94a9EXAMPLE",
    "detail-type": "Inspector2 Scan",
    "source": "aws.inspector2",
    "account": "123456789012",
    "time": "2021-12-03T18:03:16Z",
    "region": "us-east-2",
    "resources": [
        "arn:aws:ecr:us-east-2:123456789012:repository/amazon/amazon-ecs-sample"
    ],
    "detail": {
```

```
        "scan-status": "INITIAL_SCAN_COMPLETE",
        "repository-name": "arn:aws:ecr:us-east-2:123456789012:repository/amazon/
amazon-ecs-sample",
        "finding-severity-counts": {
            "CRITICAL": 7,
            "HIGH": 61,
            "MEDIUM": 62,
            "TOTAL": 158
        },
        "image-digest":
 "sha256:36c7b282abd0186e01419f2e58743e1bf635808231049bbc9d77e5EXAMPLE",
        "image-tags": [
            "latest"
        ]
    }
}
```

Event for an image scan finding update (enhanced scanning)

When enhanced scanning is turned on for your registry, the following event is sent by Amazon Inspector when the image scan finding is created, updated, or closed. For more information, see Scan images for OS and programming language package vulnerabilities in Amazon ECR.

Event pattern:

```
{
  "source": ["aws.inspector2"],
  "detail-type": ["Inspector2 Finding"]
}
```

Example output:

```
{
    "version": "0",
    "id": "42dbea55-45ad-b2b4-87a8-afaEXAMPLE",
    "detail-type": "Inspector2 Finding",
    "source": "aws.inspector2",
    "account": "123456789012",
    "time": "2021-12-03T18:02:30Z",
    "region": "us-east-2",
    "resources": [
        "arn:aws:ecr:us-east-2:123456789012:repository/amazon/amazon-ecs-sample/
sha256:36c7b282abd0186e01419f2e58743e1bf635808231049bbc9d77eEXAMPLE"
```

```
    ],
    "detail": {
        "awsAccountId": "123456789012",
        "description": "In libssh2 v1.9.0 and earlier versions, the SSH_MSG_DISCONNECT
 logic in packet.c has an integer overflow in a bounds check, enabling an attacker to
 specify an arbitrary (out-of-bounds) offset for a subsequent memory read. A crafted
 SSH server may be able to disclose sensitive information or cause a denial of service
 condition on the client system when a user connects to the server.",
        "findingArn": "arn:aws:inspector2:us-east-2:123456789012:finding/
be674aaddd0f75ac632055EXAMPLE",
        "firstObservedAt": "Dec 3, 2021, 6:02:30 PM",
        "inspectorScore": 6.5,
        "inspectorScoreDetails": {
            "adjustedCvss": {
                "adjustments": [],
                "cvssSource": "REDHAT_CVE",
                "score": 6.5,
                "scoreSource": "REDHAT_CVE",
                "scoringVector": "CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:N/A:N",
                "version": "3.0"
            }
        },
        "lastObservedAt": "Dec 3, 2021, 6:02:30 PM",
        "packageVulnerabilityDetails": {
            "cvss": [
                {
                    "baseScore": 6.5,
                    "scoringVector": "CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:N/A:N",
                    "source": "REDHAT_CVE",
                    "version": "3.0"
                },
                {
                    "baseScore": 5.8,
                    "scoringVector": "AV:N/AC:M/Au:N/C:P/I:N/A:P",
                    "source": "NVD",
                    "version": "2.0"
                },
                {
                    "baseScore": 8.1,
                    "scoringVector": "CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:N/A:H",
                    "source": "NVD",
                    "version": "3.1"
                }
            ],
```

```
            "referenceUrls": [
                "https://access.redhat.com/errata/RHSA-2020:3915"
            ],
            "source": "REDHAT_CVE",
            "sourceUrl": "https://access.redhat.com/security/cve/CVE-2019-17498",
            "vendorCreatedAt": "Oct 16, 2019, 12:00:00 AM",
            "vendorSeverity": "Moderate",
            "vulnerabilityId": "CVE-2019-17498",
            "vulnerablePackages": [
                {
                    "arch": "X86_64",
                    "epoch": 0,
                    "name": "libssh2",
                    "packageManager": "OS",
                    "release": "12.amzn2.2",
                    "sourceLayerHash":
"sha256:72d97abdfae3b3c933ff41e39779cc72853d7bd9dc1e4800c5294dEXAMPLE",
                    "version": "1.4.3"
                }
            ]
        },
        "remediation": {
            "recommendation": {
                "text": "Update all packages in the vulnerable packages section to
their latest versions."
            }
        },
        "resources": [
            {
                "details": {
                    "awsEcrContainerImage": {
                        "architecture": "amd64",
                        "imageHash":
"sha256:36c7b282abd0186e01419f2e58743e1bf635808231049bbc9d77e5EXAMPLE",
                        "imageTags": [
                            "latest"
                        ],
                        "platform": "AMAZON_LINUX_2",
                        "pushedAt": "Dec 3, 2021, 6:02:13 PM",
                        "lastInUseAt": "Dec 3, 2021, 6:02:13 PM",
                        "inUseCount": 1,
                        "registry": "123456789012",
                        "repositoryName": "amazon/amazon-ecs-sample"
                    }
```

```
                },
                "id": "arn:aws:ecr:us-east-2:123456789012:repository/amazon/amazon-ecs-
sample/sha256:36c7b282abd0186e01419f2e58743e1bf635808231049bbc9d77EXAMPLE",
                "partition": "N/A",
                "region": "N/A",
                "type": "AWS_ECR_CONTAINER_IMAGE"
            }
        ],
        "severity": "MEDIUM",
        "status": "ACTIVE",
        "title": "CVE-2019-17498 - libssh2",
        "type": "PACKAGE_VULNERABILITY",
        "updatedAt": "Dec 3, 2021, 6:02:30 PM"
    }
}
```

Retrieving the findings for enhanced scans in Amazon ECR

You can retrieve the scan findings for the last completed enhanced image scan, and then open the findings in Amazon Inspector to see more detail. The software vulnerabilities that were discovered are listed by severity based on the Common Vulnerabilities and Exposures (CVEs) database.

For troubleshooting details for some common issues when scanning images, see Troubleshooting image scanning in Amazon ECR.

AWS Management Console

Use the following steps to retrieve image scan findings using the AWS Management Console.

To retrieve image scan findings

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region where your repository exists.
3. In the navigation pane, choose **Repositories**.
4. On the **Repositories** page, choose the repository that contains the image to retrieve the scan findings for.
5. On the **Images** page, under the **Image tag** column, select the image tag to retrieve the scan findings.
6. To view more details in the Amazon Inspector console, choose the vulnerability name in the **Name** column.

AWS CLI

Use the following AWS CLI command to retrieve image scan findings using the AWS CLI. You can specify an image using the `imageTag` or `imageDigest`, both of which can be obtained using the list-images CLI command.

- describe-image-scan-findings (AWS CLI)

  The following example uses an image tag.

```
aws ecr describe-image-scan-findings \
     --repository-name name \
     --image-id imageTag=tag_name \
     --region us-east-2
```

  The following example uses an image digest.

```
aws ecr describe-image-scan-findings \
     --repository-name name \
     --image-id imageDigest=sha256_hash \
     --region us-east-2
```

Scan images for OS vulnerabilities in Amazon ECR

Amazon ECR provides two versions of basic scanning that use the Common Vulnerabilities and Exposures (CVEs) database:

- **AWS native basic scanning** – Uses AWS native technology, which is now GA and recommended. This improved basic scanning is designed to provide customers with better scanning results and vulnerability detection across a broad set of popular operating systems. This allows customers to further strengthen the security of their container images. All new customer registries are opted into this improved version by default.
- **Clair basic scanning** – The previous basic scanning version which uses the open-source Clair project and is deprecated. For more information about Clair, see Clair on GitHub.

Both AWS native and Clair basic scanning are supported in all regions listed in AWS Services by Region, except for those that were added after September, 2024. Because Clair support is

deprecated, Clair will not be supported in new regions as they are added and will no longer be supported in all regions as of October 1, 2025.

Amazon ECR uses the severity for a CVE from the upstream distribution source if available. Otherwise, the Common Vulnerability Scoring System (CVSS) score is used. The CVSS score can be used to obtain the NVD vulnerability severity rating. For more information, see NVD Vulnerability Severity Ratings.

Both versions of Amazon ECR basic scanning support filters to specify which repositories to scan on push. Any repositories that don't match a scan on push filter are set to the **manual** scan frequency which means you must manually start the scan. An image can be scanned once per 24 hours. The 24 hours includes the initial scan on push, if configured, and any manual scans. With basic scanning, you can scan up to 100,000 images per 24 hours in a given registry. The 100,000 limit includes both initial scan on push and manual scans, across both Clair and improved version of basic scanning.

The last completed image scan findings can be retrieved for each image. When an image scan is completed, Amazon ECR sends an event to Amazon EventBridge. For more information, see Amazon ECR events and EventBridge.

Operating system support for basic scanning and improved basic scanning

As a security best practice and for continued coverage, we recommend that you continue to use supported versions of an operating system. In accordance with vendor policy, discontinued operating systems are no longer updated with patches and, in many cases, new security advisories are no longer released for them. In addition, some vendors remove existing security advisories and detections from their feeds when an affected operating system reaches the end of standard support. After a distribution loses support from its vendor, Amazon ECR may no longer support scanning it for vulnerabilities. Any findings that Amazon ECR does generate for a discontinued operating system should be used for informational purposes only. Listed below are the current supported operating systems and versions.

Operating System	Version	AWS native basic	Clair basic
Alpine Linux (Alpine)	3.18	Yes	Yes

Operating System	Version	AWS native basic	Clair basic
Alpine Linux (Alpine)	3.19	Yes	Yes
Alpine Linux (Alpine)	3.20	Yes	Yes
Alpine Linux (Alpine)	3.21	Yes	No
AlmaLinux	8	Yes	No
AlmaLinux	9	Yes	No
Amazon Linux 2 (AL2)	AL2	Yes	Yes
Amazon Linux 2023(AL2023)	AL2023	Yes	Yes
Debian Server (Bookworm)	12	Yes	Yes
Debian Server (Bullseye)	11	Yes	Yes
Fedora	40	Yes	No
Fedora	41	Yes	No
OpenSUSE Leap	15.6	Yes	No
Oracle Linux (Oracle)	9	Yes	Yes

Operating System	Version	AWS native basic	Clair basic
Oracle Linux (Oracle)	8	Yes	Yes
Photon OS	4	Yes	No
Photon OS	5	Yes	No
Red Hat Enterprise Linux (RHEL)	8	Yes	Yes
Red Hat Enterprise Linux (RHEL)	9	Yes	Yes
Rocky Linux	8	Yes	No
Rocky Linux	9	Yes	No
SUSE Linux Enterprise Server (SLES)	15.6	Yes	No
Ubuntu (Xenial)	16.04 (ESM)	Yes	Yes
Ubuntu (Bionic)	18.04 (ESM)	Yes	Yes
Ubuntu (Focal)	20.04 (LTS)	Yes	Yes
Ubuntu (Jammy)	22.04 (LTS)	Yes	Yes
Ubuntu (Noble Numbat)	24.04	Yes	No

Operating System	Version	AWS native basic	Clair basic
Ubuntu (Oracular Oriole))	24.10	Yes	No

Configuring basic scanning for images in Amazon ECR

By default, Amazon ECR turns on basic scanning for all private registries. As a result, unless you've changed the scanning settings on your private registry there is no need to turn on basic scanning. Basic scanning uses the open-source Clair project.

You can use the following steps to define one or more scan on push filters.

To turn on basic scanning for your private registry

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/private-registry/repositories
2. From the navigation bar, choose the Region to set the scanning configuration for.
3. In the navigation pane, choose **Private registry**, **Scanning**.
4. On the **Scanning configuration** page, For **Scan type** choose **Basic scanning**.
5. By default all of your repositories are set for **Manual** scanning. You can optionally configure scan on push by specifying **Scan on push filters**. You can set scan on push for all repositories or individual repositories. For more information, see Filters to choose which repositories are scanned in Amazon ECR.

Switching to the improved basic scanning for images in Amazon ECR

Amazon ECR provides enhanced container image scanning capabilities through improved version of basic scanning that uses AWS native technology. This feature helps you identify software vulnerabilities in your container images. The following procedure helps you to switch to this improved version of basic scanning if you are using previous version of basic scanning that uses `CLAIR` technology.

> **Important**
>
> For new users, your registries are automatically configured to use the `AWS_NATIVE` scanning technology upon creation. There is no action for you to take. Amazon ECR doesn't recommend reverting to the previous scanning technology `CLAIR`.

AWS Management Console

To turn on improved basic scanning for your private registry

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/private-registry/repositories
2. From the navigation bar, choose the Region to set the scanning configuration for.
3. In the navigation pane, choose **Private registry**, **Features & Settings**, **Scanning**.
4. On the **Scanning configuration** page, choose **Opt in (recommended)** to select improved version of basic scanning.
5. By default all of your repositories are set for **Manual** scanning. You can optionally configure scan on push by specifying **Scan on push filters**. You can set scan on push for all repositories or individual repositories. For more information, see Filters to choose which repositories are scanned in Amazon ECR.

AWS CLI

Amazon ECR has basic scanning enabled for all private registries. Use the following commands below to view your current basic scan type and to change your basic scan type.

- To retrieve the basic scan type version you are currently using.

```
aws ecr get-account-setting --name BASIC_SCAN_TYPE_VERSION
```

The parameter name is a required field. If you don't provide the name you will receive the following error:

```
aws: error: the following arguments are required: --name
```

To change your basic scan type version from `CLAIR` to `AWS_NATIVE`. Once you change your basic scan type version from `CLAIR` to `AWS_NATIVE` it's not recommended that you revert back to `CLAIR`.

```
aws ecr put-account-setting --name BASIC_SCAN_TYPE_VERSION --value value
```

Manually scanning an image for OS vulnerabilities in Amazon ECR

If your repositories aren't configured to **scan on push**, you can manually start image scans. An image can be scanned once per 24 hours. The 24 hours includes the initial scan on push, if configured, and any manual scans.

For troubleshooting details for some common issues when scanning images, see Troubleshooting image scanning in Amazon ECR.

AWS Management Console

Use the following steps to start a manual image scan using the AWS Management Console.

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/private-registry/repositories
2. From the navigation bar, choose the Region to create your repository in.
3. In the navigation pane, choose **Repositories**.
4. On the **Repositories** page, choose the repository that contains the image to scan.
5. On the **Images** page, select the image to scan and then choose **Scan**.

AWS CLI

- start-image-scan (AWS CLI)

  The following example uses an image tag.

  ```
  aws ecr start-image-scan --repository-name name --image-id imageTag=tag_name --
  region us-east-2
  ```

  The following example uses an image digest.

```
aws ecr start-image-scan --repository-name name --image-id imageDigest=sha256_hash
 --region us-east-2
```

AWS Tools for Windows PowerShell

- Get-ECRImageScanFinding (AWS Tools for Windows PowerShell)

  The following example uses an image tag.

  ```
  Start-ECRImageScan -RepositoryName name -ImageId_ImageTag tag_name -Region us-
  east-2 -Force
  ```

  The following example uses an image digest.

  ```
  Start-ECRImageScan -RepositoryName name -ImageId_ImageDigest sha256_hash -
  Region us-east-2 -Force
  ```

Retrieving the findings for basic scans in Amazon ECR

You can retrieve the scan findings for the last completed basic image scan. The software vulnerabilities that were discovered are listed by severity based on the Common Vulnerabilities and Exposures (CVEs) database.

For troubleshooting details for some common issues when scanning images, see Troubleshooting image scanning in Amazon ECR.

AWS Management Console

Use the following steps to retrieve image scan findings using the AWS Management Console.

To retrieve image scan findings

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/private-registry/repositories
2. From the navigation bar, choose the Region to create your repository in.
3. In the navigation pane, choose **Repositories**.

4. On the **Repositories** page, choose the repository that contains the image to retrieve the scan findings for.
5. On the **Images** page, under the **Image tag** column, select the image tag to retrieve the scan findings.

AWS CLI

Use the following AWS CLI command to retrieve image scan findings using the AWS CLI. You can specify an image using the `imageTag` or `imageDigest`, both of which can be obtained using the list-images CLI command.

- describe-image-scan-findings (AWS CLI)

  The following example uses an image tag.

```
aws ecr describe-image-scan-findings --repository-name name --image-id
 imageTag=tag_name --region us-east-2
```

  The following example uses an image digest.

```
aws ecr describe-image-scan-findings --repository-name name --image-id
 imageDigest=sha256_hash --region us-east-2
```

AWS Tools for Windows PowerShell

- Get-ECRImageScanFinding (AWS Tools for Windows PowerShell)

  The following example uses an image tag.

```
Get-ECRImageScanFinding -RepositoryName name -ImageId_ImageTag tag_name -
Region us-east-2
```

  The following example uses an image digest.

```
Get-ECRImageScanFinding -RepositoryName name -ImageId_ImageDigest sha256_hash -
Region us-east-2
```

Troubleshooting image scanning in Amazon ECR

The following are common image scan failures. You can view errors like this in the Amazon ECR console by displaying the image details or through the API or AWS CLI by using the `DescribeImageScanFindings` API.

UnsupportedImageError

You may get an `UnsupportedImageError` error when attempting to perform a basic scan on an image that was built using an operating system that Amazon ECR doesn't support basic image scanning for. Amazon ECR supports package vulnerability scanning for major versions of Amazon Linux, Amazon Linux 2, Debian, Ubuntu, CentOS, Oracle Linux, Alpine, and RHEL Linux distributions. Once a distribution loses support from its vendor, Amazon ECR may no longer support scanning it for vulnerabilities. Amazon ECR does not support scanning images built from the Docker scratch image.

> **Important**
>
> When using enhanced scanning, Amazon Inspector supports scanning for specific operating systems and media types. For a full list, see Supported operating systems and media types in the *Amazon Inspector User Guide*.

An `UNDEFINED` severity level is returned

You may receive a scan finding that has a severity level of `UNDEFINED`. The following are the common causes for this:

- The vulnerability was not assigned a priority by the CVE source.
- The vulnerability was assigned a priority that Amazon ECR did not recognize.

To determine the severity and description of a vulnerability, you can view the CVE directly from the source.

Understanding scan status `SCAN_ELIGIBILITY_EXPIRED`

When enhanced scanning using Amazon Inspector is enabled for your private registry and you are viewing your scan vulnerabilities, you may see a scan status of `SCAN_ELIGIBILITY_EXPIRED`. The following are the most common causes of this.

- When you initially turn on enhanced scanning for your private registry, Amazon Inspector only recognizes images pushed to Amazon ECR in the last 30 days, based on the image push timestamp. Older images will have the `SCAN_ELIGIBILITY_EXPIRED` scan status. If you'd like these images to be scanned by Amazon Inspector you should push them again to your repository.
- If the **ECR re-scan duration** is changed in the Amazon Inspector console and that time elapses, the scan status of the image is changed to `inactive` with a reason code of `expired`, and all associated findings for the image are scheduled to be closed. This results in the Amazon ECR console listing the scan status as `SCAN_ELIGIBILITY_EXPIRED`.

Sync an upstream registry with an Amazon ECR private registry

Using pull through cache rules, you can sync the contents of an upstream registry with your Amazon ECR private registry.

Amazon ECR currently supports creating pull through cache rules for the following upstream registries:

- Amazon ECR Public, Kubernetes container image registry, and Quay (doesn't require authentication)
- Docker Hub, Microsoft Azure Container Registry, GitHub Container Registry, and GitLab Container Registry (requires authentication with AWS Secrets Manager secret)
- Amazon ECR (requires authentication with AWS IAM role)

For GitLab Container Registry, Amazon ECR supports pull through cache only with GitLab's Software as a Service (SaaS) offering. For more information about using GitLab's SaaS offering, see GitLab.com.

For upstream registries that require authentication with secrets (such as Docker Hub), you must store your credentials in an AWS Secrets Manager secret. You can use the Amazon ECR console to create Secrets Manager secrets for each authenticated upstream registry. For more information about creating a Secrets Manager secret using the Secrets Manager console, see Storing your upstream repository credentials in an AWS Secrets Manager secret.

For Amazon ECR, you must create an IAM role if the upstream and downstream Amazon ECR registries belong to different AWS account. For more information about creating an IAM role, see IAM policies required for cross-account ECR to ECR pull through cache.

After you've created a pull through cache rule for the upstream registry, pull an image from that upstream registry using your Amazon ECR private registry URI. Amazon ECR then creates a repository and caches that image in your private registry. For subsequent pull requests of the cached image with a given tag, Amazon ECR checks the upstream registry for a new version of the image with that specific tag and attempts to update the image in your private registry at least once every 24 hours.

Repository creation templates

Amazon ECR has added support for repository creation templates, which gives you the control to specify initial configurations for new repositories created by Amazon ECR on your behalf using pull through cache rules. Each template contains a repository namespace prefix which is used to match new repositories to a specific template. Templates can specify the configuration for all repository settings including resource-based access policies, tag immutability, encryption, and lifecycle policies. The settings in a repository creation template are only applied during repository creation and don't have any effect on existing repositories or repositories created using any other method. For more information, see Templates to control repositories created during a pull through cache or replication action.

Considerations for using pull through cache rules

Consider the following when using Amazon ECR pull through cache rules.

- Creating pull through cache rules isn't supported in the following Regions.
  - China (Beijing) (`cn-north-1`)
  - China (Ningxia) (`cn-northwest-1`)
  - AWS GovCloud (US-East) (`us-gov-east-1`)
  - AWS GovCloud (US-West) (`us-gov-west-1`)
- AWS Lambda doesn't support pulling container images from Amazon ECR using a pull through cache rule.
- When pulling images using pull through cache, the Amazon ECR FIPS service endpoints aren't supported the first time an image is pulled. Using the Amazon ECR FIPS service endpoints work on subsequent pulls though.
- When a cached image is pulled through the Amazon ECR private registry URI, the image pulls are initiated by AWS IP addresses. This ensures that the image pull doesn't count against any pull rate quotas implemented by the upstream registry.
- When a cached image is pulled through the Amazon ECR private registry URI, Amazon ECR checks the upstream repository at least once every 24 hours to verify whether the cached image is the latest version. If there is a newer image in the upstream registry, Amazon ECR attempts to update the cached image. This timer is based off the last pull of the cached image.
- If Amazon ECR is unable to update the image from the upstream registry for any reason and the image is pulled, the last cached image will still be pulled.

- When creating the Secrets Manager secret that contains the upstream registry credentials, the secret name must use the `ecr-pullthroughcache/` prefix. The secret must also be in the same account and Region that the pull through cache rule is created in.
- When a multi-architecture image is pulled using a pull through cache rule, the manifest list and each image referenced in the manifest list are pulled to the Amazon ECR repository. If you only want to pull a specific architecture, you can pull the image using the image digest or tag associated with the architecture rather than the tag associated with the manifest list.
- Amazon ECR uses a service-linked IAM role, which provides the permissions needed for Amazon ECR to create the repository, retrieve the Secrets Manager secret value for authentication, and push the cached image on your behalf. The service-linked IAM role is created automatically when a pull through cache rule is created. For more information, see Amazon ECR service-linked role for pull through cache.
- By default, the IAM principal pulling the cached image has the permissions granted to them through their IAM policy. You may use the Amazon ECR private registry permissions policy to further scope the permissions of an IAM entity. For more information, see Using registry permissions.
- Amazon ECR repositories created using the pull through cache workflow are treated like any other Amazon ECR repository. All repository features, such as replication and image scanning are supported.
- When Amazon ECR creates a new repository on your behalf using a pull through cache action, the following default settings are applied to the repository unless there is a matching repository creation template. You can use a repository creation template to define the settings applied to repositories created by Amazon ECR on your behalf. For more information, see Templates to control repositories created during a pull through cache or replication action.
  - Tag immutability – Tag immutability specifies whether image tags can be overwritten. By default, image tags are mutable (can be overwritten). You can modify tag behavior by configuring tag exclusion filters in either the **Mutable tag exclusion** text box when **Mutable** is selected, or the **Immutable tag exclusion** text box when **Immutable** is selected.
  - Encryption – The default AES256 encryption is used.
  - Repository permissions – Omitted, no repository permissions policy is applied.
  - Lifecycle policy – Omitted, no lifecycle policy is applied.
  - Resource tags – Omitted, no resource tags are applied.
- Turning on image tag immutability for repositories using a pull through cache rule will prevent Amazon ECR from updating images using the same tag.

- When an image is pulled using the pull through cache rule for the first time a route to the internet may be required. There are certain circumstances in which a route to the internet is required so it's best to set up a route to avoid any failures. Thus, if you've configured Amazon ECR to use an interface VPC endpoint using AWS PrivateLink then you need to ensure the first pull has a route to the internet. One way to do this is to create a public subnet in the same VPC, with an internet gateway, and then route all outbound traffic to the internet from their private subnet to the public subnet. Subsequent image pulls using the pull through cache rule don't require this. For more information, see Example routing options in the *Amazon Virtual Private Cloud User Guide*.

IAM permissions required to sync an upstream registry with an Amazon ECR private registry

In addition to the Amazon ECR API permissions needed to authenticate to a private registry and to push and pull images, the following additional permissions are needed to use pull through cache rules effectively.

- `ecr:CreatePullThroughCacheRule` – Grants permission to create a pull through cache rule. This permission must be granted via an identity-based IAM policy.
- `ecr:BatchImportUpstreamImage` – Grants permission to retrieve the external image and import it to your private registry. This permission can be granted by using the private registry permissions policy, an identity-based IAM policy, or by using the resource-based repository permissions policy. For more information about using repository permissions, see Private repository policies in Amazon ECR.
- `ecr:CreateRepository` – Grants permission to create a repository in a private registry. This permission is required if the repository storing the cached images doesn't already exist. This permission can be granted by either an identity-based IAM policy or the private registry permissions policy.

Using registry permissions

Amazon ECR private registry permissions may be used to scope the permissions of individual IAM entities to use pull through cache. If an IAM entity has more permissions granted by an IAM policy than the registry permissions policy is granting, the IAM policy takes precedence. For example, if user has `ecr:*` permissions granted, no additional permissions are needed at the registry level.

To create a private registry permissions policy (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region to configure your private registry permissions statement in.
3. In the navigation pane, choose **Private registry**, **Registry permissions**.
4. On the **Registry permissions** page, choose **Generate statement**.
5. For each pull through cache permissions policy statement you want to create, do the following.

    a. For **Policy type**, choose **Pull through cache policy**.

    b. For **Statement id**, provide a name for the pull through cache statement policy.

    c. For **IAM entities**, specify the users, groups, or roles to include in the policy.

    d. For **Repository namespace**, select the pull through cache rule to associate the policy with.

    e. For **Repository names**, specify the repository base name to apply the rule for. For example, if you want to specify the Amazon Linux repository on Amazon ECR Public, the repository name would be `amazonlinux`.

To create a private registry permissions policy (AWS CLI)

Use the following AWS CLI command to specify the private registry permissions using the AWS CLI.

1. Create a local file named `ptc-registry-policy.json` with the contents of your registry policy. The following example grants the `ecr-pull-through-cache-user` permission to create a repository and pull an image from Amazon ECR Public, which is the upstream source associated with the previously created pull through cache rule.

    JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PullThroughCacheFromReadOnlyRole",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:user/ecr-pull-through-cache-user"
```

```
      },
      "Action": [
        "ecr:CreateRepository",
        "ecr:BatchImportUpstreamImage"
      ],
      "Resource": "arn:aws:ecr:us-east-1:111122223333:repository/ecr-public/
*"
    }
  ]
}
```

> **Important**
>
> The `ecr-CreateRepository` permission is only required if the repository storing the cached images doesn't already exist. For example, if the repository creation action and the image pull actions are being done by separate IAM principals such as an administrator and a developer.

2. Use the put-registry-policy command to set the registry policy.

```
aws ecr put-registry-policy \
     --policy-text file://ptc-registry.policy.json
```

Next steps

Once you are ready to start using pull through cache rules, the following are the next steps.

- Create a pull through cache rule. For more information, see Creating a pull through cache rule in Amazon ECR.
- Create a repository creation template. A repository creation template gives you control to define the settings to use for new repositories created by Amazon ECR on your behalf during a pull through cache action. For more information, see Templates to control repositories created during a pull through cache or replication action.

Setting up permissions for cross-account ECR to ECR PTC

The Amazon ECR to Amazon ECR (ECR to ECR) pull through cache feature enables automatic synchronization of images between Regions, AWS accounts, or both. With ECR to ECR PTC, you can push images to your primary Amazon ECR registry and configure a pull through cache rule to cache images in downstream Amazon ECR registries.

IAM policies required for cross-account ECR to ECR pull through cache

To cache images between Amazon ECR registries across different AWS accounts, create an IAM role in the downstream account and configure the policies in this section to provide the following permissions:

- Amazon ECR needs permissions to pull images from the upstream Amazon ECR registry on your behalf. You can grant these permissions by creating an IAM role and then specifying it in your pull through cache rule.
- The upstream registry owner must also grant the cache registry owner with the required permissions to pull the images in to the resource policies.

Policies

- Creating an IAM role to define the pull through cache permissions
- Creating a Trust policy for the IAM role
- Creating a resource policy in the upstream Amazon ECR registry

Creating an IAM role to define the pull through cache permissions

The following example shows a permissions policy that grants an IAM role permission to pull images from the upstream Amazon ECR registry on your behalf. When Amazon ECR assumes the role, it receives the permissions specified in this policy.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetAuthorizationToken",
                "ecr:BatchImportUpstreamImage",
                "ecr:BatchGetImage",
                "ecr:GetImageCopyStatus",
                "ecr:InitiateLayerUpload",
                "ecr:UploadLayerPart",
                "ecr:CompleteLayerUpload",
                "ecr:PutImage"
            ],
            "Resource": "*"
        }
    ]
}
```

Creating a Trust policy for the IAM role

The following example shows a trust policy that identifies Amazon ECR pull through cache as the AWS service principal that can assume the role.

JSON

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

Creating a resource policy in the upstream Amazon ECR registry

The upstream Amazon ECR registry owner must also add a registry policy or a repository policy to grant the downstream registry owner the required permissions to perform the following actions.

```
{
    "Effect": "Allow",
    "Principal": {
        "AWS": "arn:aws:iam::444455556666:root"
    },
    "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchImportUpstreamImage",
        "ecr:GetImageCopyStatus"
    ],
    "Resource": "arn:aws:ecr:region:111122223333:repository/*"
}
```

Creating a pull through cache rule in Amazon ECR

For each upstream registry containing images that you want to cache in your Amazon ECR private registry, you must create a pull through cache rule.

For upstream registries that require authentication with secrets, you must store the credentials in a Secrets Manager secret. You can use an existing secret or create a new secret. You can create the Secrets Manager secret in either the Amazon ECR console or the Secrets Manager console. To create a Secrets Manager secret using the Secrets Manager console instead of the Amazon ECR console, see Storing your upstream repository credentials in an AWS Secrets Manager secret.

Prerequisites

- Verify that you have the proper IAM permissions to create pull through cache rules. For information, see IAM permissions required to sync an upstream registry with an Amazon ECR private registry.
- For upstream registries that require authentication with secrets: If you want to use an existing secret, verify that the Secrets Manager secret meets the following requirements:
  - The name of the secret begins with `ecr-pullthroughcache/`. The AWS Management Console only displays Secrets Manager secrets with the `ecr-pullthroughcache/` prefix.

- The account and Region that the secret is in must match the account and Region that the pull through cache rule is in.

To create a pull through cache rule (AWS Management Console)

The following steps show how to create a pull through cache rule and a Secrets Manager secret using the Amazon ECR console. To create a secret using the Secrets Manager console, see Storing your upstream repository credentials in an AWS Secrets Manager secret.

For Amazon ECR Public, Kubernetes container registry, or Quay

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region to configure your private registry settings in.
3. In the navigation pane, choose **Private registry**, **Pull through cache**.
4. On the **Pull through cache configuration** page, choose **Add rule**.
5. On the **Step 1: Specify a source** page, for **Registry**, choose either Amazon ECR Public, Kubernetes, or Quay from the list of upstream registries and then choose **Next**.
6. On the **Step 2: Specify a destination** page, for **Amazon ECR repository prefix**, specify the repository namespace prefix to use when caching images pulled from the source public registry and then choose **Next**. By default, a namespace is populated but a custom namespace can be specified as well.
7. On the **Step 3: Review and create** page, review the pull through cache rule configuration and then choose **Create**.
8. Repeat the previous step for each pull through cache you want to create. The pull through cache rules are created separately for each Region.

For Docker Hub

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region to configure your private registry settings in.
3. In the navigation pane, choose **Private registry**, **Pull through cache**.
4. On the **Pull through cache configuration** page, choose **Add rule**.
5. On the **Step 1: Specify a source** page, for **Registry**, choose **Docker Hub**, **Next**.

6. On the **Step 2: Configure authentication** page, for **Upstream credentials**, you must store your authentication credentials for Docker Hub in an AWS Secrets Manager secret. You can specify an existing secret or use the Amazon ECR console to create a new secret.

   a. To use an existing secret, choose **Use an existing AWS secret**. For **Secret name** use the drop down to select your existing secret, and then choose **Next**.

   > **Note**
   >
   > The AWS Management Console only displays Secrets Manager secrets with names using the `ecr-pullthroughcache/` prefix. The secret must also be in the same account and Region that the pull through cache rule is created in.

   b. To create a new secret, choose **Create an AWS secret**, do the following, then choose **Next**.

      i. For **Secret name**, specify a descriptive name for the secret. Secret names must contain 1-512 Unicode characters.

      ii. For **Docker Hub email**, specify your Docker Hub email.

      iii. For **Docker Hub access token**, specify your Docker Hub access token. For more information on creating a Docker Hub access token, see Create and manage access tokens in the Docker documentation.

7. On the **Step 3: Specify a destination** page, for **Amazon ECR repository prefix**, specify the repository namespace to use when caching images pulled from the source public registry and then choose **Next**.

   By default, a namespace is populated but a custom namespace can be specified as well.

8. On the **Step 4: Review and create** page, review the pull through cache rule configuration and then choose **Create**.

9. Repeat the previous step for each pull through cache you want to create. The pull through cache rules are created separately for each Region.

For GitHub Container Registry

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.

2. From the navigation bar, choose the Region to configure your private registry settings in.

3. In the navigation pane, choose **Private registry**, **Pull through cache**.

4. On the **Pull through cache configuration** page, choose **Add rule**.
5. On the **Step 1: Specify a source** page, for **Registry**, choose **GitHub Container Registry**, **Next**.
6. On the **Step 2: Configure authentication** page, for **Upstream credentials**, you must store your authentication credentials for GitHub Container Registry in an AWS Secrets Manager secret. You can specify an existing secret or use the Amazon ECR console to create a new secret.

   a. To use an existing secret, choose **Use an existing AWS secret**. For **Secret name** use the drop down to select your existing secret, and then choose **Next**.

   > **Note**
   >
   > The AWS Management Console only displays Secrets Manager secrets with names using the `ecr-pullthroughcache/` prefix. The secret must also be in the same account and Region that the pull through cache rule is created in.

   b. To create a new secret, choose **Create an AWS secret**, do the following, then choose **Next**.

      i. For **Secret name**, specify a descriptive name for the secret. Secret names must contain 1-512 Unicode characters.

      ii. For **GitHub Container Registry username**, specify your GitHub Container Registry username.

      iii. For **GitHub Container Registry access token**, specify your GitHub Container Registry access token. For more information on creating a GitHub access token, see Managing your personal access tokens in the GitHub documentation.
7. On the **Step 3: Specify a destination** page, for **Amazon ECR repository prefix**, specify the repository namespace to use when caching images pulled from the source public registry and then choose **Next**.

   By default, a namespace is populated but a custom namespace can be specified as well.
8. On the **Step 4: Review and create** page, review the pull through cache rule configuration and then choose **Create**.
9. Repeat the previous step for each pull through cache you want to create. The pull through cache rules are created separately for each Region.

For Microsoft Azure Container Registry

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.

2. From the navigation bar, choose the Region to configure your private registry settings in.
3. In the navigation pane, choose **Private registry**, **Pull through cache**.
4. On the **Pull through cache configuration** page, choose **Add rule**.
5. On the **Step 1: Specify a source** page, do the following.

   a. For **Registry**, choose **Microsoft Azure Container Registry**

   b. For **Source registry URL**, specify the name of your Microsoft Azure container registry and then choose **Next**.

   > **Important**
   >
   > You only need to specify the prefix, as the `.azurecr.io` suffix is populated on your behalf.

6. On the **Step 2: Configure authentication** page, for **Upstream credentials**, you must store your authentication credentials for Microsoft Azure Container Registry in an AWS Secrets Manager secret. You can specify an existing secret or use the Amazon ECR console to create a new secret.

   a. To use an existing secret, choose **Use an existing AWS secret**. For **Secret name** use the drop down to select your existing secret, and then choose **Next**.

   > **Note**
   >
   > The AWS Management Console only displays Secrets Manager secrets with names using the `ecr-pullthroughcache/` prefix. The secret must also be in the same account and Region that the pull through cache rule is created in.

   b. To create a new secret, choose **Create an AWS secret**, do the following, then choose **Next**.

      i. For **Secret name**, specify a descriptive name for the secret. Secret names must contain 1-512 Unicode characters.

      ii. For **Microsoft Azure Container Registry username**, specify your Microsoft Azure Container Registry username.

      iii. For **Microsoft Azure Container Registry access token**, specify your Microsoft Azure Container Registry access token. For more information on creating an Microsoft Azure

Container Registry access token, see Create token - portal in the Microsoft Azure documentation.

7. On the **Step 3: Specify a destination** page, for **Amazon ECR repository prefix**, specify the repository namespace to use when caching images pulled from the source public registry and then choose **Next**.

   By default, a namespace is populated but a custom namespace can be specified as well.

8. On the **Step 4: Review and create** page, review the pull through cache rule configuration and then choose **Create**.

9. Repeat the previous step for each pull through cache you want to create. The pull through cache rules are created separately for each Region.

For GitLab Container Registry

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.

2. From the navigation bar, choose the Region to configure your private registry settings in.

3. In the navigation pane, choose **Private registry**, **Pull through cache**.

4. On the **Pull through cache configuration** page, choose **Add rule**.

5. On the **Step 1: Specify a source** page, for Registry, choose GitLab Container Registry, Next.

6. On the **Step 2: Configure authentication** page, for **Upstream credentials**, you must store your authentication credentials for GitLab Container Registry in an AWS Secrets Manager secret. You can specify an existing secret or use the Amazon ECR console to create a new secret.

   a. To use an existing secret, choose **Use an existing AWS secret**. For **Secret name** use the drop down to select your existing secret, and then choose **Next**. For more information on creating a Secrets Manager secret using the Secrets Manager console, see Storing your upstream repository credentials in an AWS Secrets Manager secret.

   > **Note**
   >
   > The AWS Management Console only displays Secrets Manager secrets with names using the `ecr-pullthroughcache/` prefix. The secret must also be in the same account and Region that the pull through cache rule is created in.

   b. To create a new secret, choose **Create an AWS secret**, do the following, then choose **Next**.

i. For **Secret name**, specify a descriptive name for the secret. Secret names must contain 1-512 Unicode characters.

ii. For **GitLab Container Registry username**, specify your GitLab Container Registry username.

iii. For **GitLab Container Registry access token**, specify your GitLab Container Registry access token. For more information on creating a GitLab Container Registry access token, see Personal access tokens, Group access tokens, or Project access tokens, in the GitLab documentation.

7. On the **Step 3: Specify a destination** page, for **Amazon ECR repository prefix**, specify the repository namespace to use when caching images pulled from the source public registry and then choose **Next**.

   By default, a namespace is populated but a custom namespace can be specified as well.

8. On the **Step 4: Review and create** page, review the pull through cache rule configuration and then choose **Create**.

9. Repeat the previous step for each pull through cache you want to create. The pull through cache rules are created separately for each Region.

For Amazon ECR private registry within your AWS account

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.

2. From the navigation bar, choose the Region in which you want to configure your private registry settings.

3. In the navigation pane, choose **Private registry**, **Pull through cache**.

4. On the **Pull through cache configuration** page, choose **Add rule**.

5. On the **Step 1: Specify upstream** page, for **Registry**, choose **Amazon ECR Private** and **This account**. For **Region**, select the Region for the upstream Amazon ECR registry, and then choose **Next**.

6. On the **Step 2: Specify namespaces** page, for **Cache namespace**, choose whether to create pull through cache repositories with **A specific prefix** or **no prefix**. If you select **A specific prefix**, you must specify a prefix name to be used as part of the namespace for caching images from the upstream registry.

7. For **Upstream namespace**, choose whether to pull from **A specific prefix** that exists in the upstream registry. If you select **no prefix**, you can pull from any repository in the upstream registry. Specify the upstream repository prefix if prompted, and then choose **Next**.

   > **Note**
   >
   > To learn more about customizing cache and upstream namespaces, see Customizing repository prefixes for ECR to ECR pull through cache.

8. On the **Step 3: Review and create** page, review the pull through cache rule configuration and then choose **Create**.
9. Repeat these steps for each pull through cache you want to create. The pull through cache rules are created separately for each Region.

For Amazon ECR private registry from another AWS account

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region to configure your private registry settings in.
3. In the navigation pane, choose **Private registry**, **Pull through cache**.
4. On the **Pull through cache configuration** page, choose **Add rule**.
5. On the **Step 1: Specify upstream** page, for **Registry**, choose **Amazon ECR Private** and **Cross account**. For **Region**, select the Region for the upstream Amazon ECR registry. For **Account**, specify the AWS account ID for the upstream Amazon ECR registry, and then choose **Next**.
6. On the **Step 2: Specify permissions** page, for **IAM role**, select a role to be used for cross account pull through cache access and then choose **Create**.

   > **Note**
   >
   > Make sure that you select the IAM role which uses the permissions created in IAM policies required for cross-account ECR to ECR pull through cache.

7. On the **Step 3: Specify namespaces** page, for **Cache namespace**, choose whether to create pull through cache repositories with **A specific prefix** or **no prefix**. If you select **A specific prefix**, you must specify a prefix name to be used as part of the namespace for caching images from the upstream registry.

8. For **Upstream namespace**, choose whether to pull from **A specific prefix** that exists in the upstream registry. If you select **no prefix**, you can pull from any repository in the upstream registry. Specify the upstream repository prefix if prompted, and then choose **Next**.

   > **Note**
   >
   > To learn more about customizing cache and upstream namespaces, see Customizing repository prefixes for ECR to ECR pull through cache.

9. On the **Step 4: Review and create** page, review the pull through cache rule configuration and then choose **Create**.
10. Repeat these steps for each pull through cache you want to create. The pull through cache rules are created separately for each Region.

To create a pull through cache rule (AWS CLI)

Use the create-pull-through-cache-rule AWS CLI command to create a pull through cache rule for an Amazon ECR private registry. For upstream registries that require authentication with secrets, you must store the credentials in an Secrets Manager secret. To create a secret using the Secrets Manager console, see Storing your upstream repository credentials in an AWS Secrets Manager secret.

The following examples are provided for each supported upstream registry.

For Amazon ECR Public

The following example creates a pull through cache rule for the Amazon ECR Public registry. It specifies a repository prefix of `ecr-public`, which results in each repository created using the pull through cache rule to have the naming scheme of `ecr-public/`*`upstream-repository-name`*.

```
aws ecr create-pull-through-cache-rule \
     --ecr-repository-prefix ecr-public \
     --upstream-registry-url public.ecr.aws \
     --region us-east-2
```

For Kubernetes Container Registry

The following example creates a pull through cache rule for the Kubernetes public registry. It specifies a repository prefix of `kubernetes`, which results in each repository created using the pull through cache rule to have the naming scheme of `kubernetes/`*`upstream-repository-name`*.

```
aws ecr create-pull-through-cache-rule \
     --ecr-repository-prefix kubernetes \
     --upstream-registry-url registry.k8s.io \
     --region us-east-2
```

For Quay

The following example creates a pull through cache rule for the Quay public registry. It specifies a repository prefix of `quay`, which results in each repository created using the pull through cache rule to have the naming scheme of `quay/`*`upstream-repository-name`*.

```
aws ecr create-pull-through-cache-rule \
     --ecr-repository-prefix quay \
     --upstream-registry-url quay.io \
     --region us-east-2
```

For Docker Hub

The following example creates a pull through cache rule for the Docker Hub registry. It specifies a repository prefix of `docker-hub`, which results in each repository created using the pull through cache rule to have the naming scheme of `docker-hub/`*`upstream-repository-name`*. You must specify the full Amazon Resource Name (ARN) of the secret containing your Docker Hub credentials.

```
aws ecr create-pull-through-cache-rule \
     --ecr-repository-prefix docker-hub \
     --upstream-registry-url registry-1.docker.io \
     --credential-arn arn:aws:secretsmanager:us-east-2:111122223333:secret:ecr-
pullthroughcache/example1234 \
     --region us-east-2
```

For GitHub Container Registry

The following example creates a pull through cache rule for the GitHub Container Registry. It specifies a repository prefix of `github`, which results in each repository created using the pull

through cache rule to have the naming scheme of `github/`*`upstream-repository-name`*. You must specify the full Amazon Resource Name (ARN) of the secret containing your GitHub Container Registry credentials.

```
aws ecr create-pull-through-cache-rule \
     --ecr-repository-prefix github \
     --upstream-registry-url ghcr.io \
     --credential-arn arn:aws:secretsmanager:us-east-2:111122223333:secret:ecr-
pullthroughcache/example1234 \
     --region us-east-2
```

For Microsoft Azure Container Registry

The following example creates a pull through cache rule for the Microsoft Azure Container Registry. It specifies a repository prefix of `azure`, which results in each repository created using the pull through cache rule to have the naming scheme of `azure/`*`upstream-repository-name`*. You must specify the full Amazon Resource Name (ARN) of the secret containing your Microsoft Azure Container Registry credentials.

```
aws ecr create-pull-through-cache-rule \
     --ecr-repository-prefix azure \
     --upstream-registry-url myregistry.azurecr.io \
     --credential-arn arn:aws:secretsmanager:us-east-2:111122223333:secret:ecr-
pullthroughcache/example1234 \
     --region us-east-2
```

For GitLab Container Registry

The following example creates a pull through cache rule for the GitLab Container Registry. It specifies a repository prefix of `gitlab`, which results in each repository created using the pull through cache rule to have the naming scheme of `gitlab/`*`upstream-repository-name`*. You must specify the full Amazon Resource Name (ARN) of the secret containing your GitLab Container Registry credentials.

```
aws ecr create-pull-through-cache-rule \
     --ecr-repository-prefix gitlab \
     --upstream-registry-url registry.gitlab.com \
     --credential-arn arn:aws:secretsmanager:us-east-2:111122223333:secret:ecr-
pullthroughcache/example1234 \
     --region us-east-2
```

For Amazon ECR private registry within your AWS account

The following example creates a pull through cache rule for the Amazon ECR private registry for cross-Region within the same AWS account. It specifies a repository prefix of `ecr`, which results in each repository created using the pull through cache rule to have the naming scheme of `ecr/`*`upstream-repository-name`*.

```
aws ecr create-pull-through-cache-rule \
     --ecr-repository-prefix ecr \
     --upstream-registry-url aws_account_id.dkr.ecr.region.amazonaws.com \
     --region us-east-2
```

For Amazon ECR private registry from another AWS account

The following example creates a pull through cache rule for the Amazon ECR private registry for cross-Region within the same AWS account. It specifies a repository prefix of `ecr`, which results in each repository created using the pull through cache rule to have the naming scheme of `ecr/`*`upstream-repository-name`*. You must specify the full Amazon Resource Name (ARN) of the IAM role with the permissions created in Creating a pull through cache rule in Amazon ECR.

```
aws ecr create-pull-through-cache-rule \
 --ecr-repository-prefix ecr \
 --upstream-registry-url aws_account_id.dkr.ecr.region.amazonaws.com \
 --custom-role-arn arn:aws:iam::aws_account_id:role/example-role \
 --region us-east-2
```

Next steps

After you create your pull through cache rules, the following are the next steps:

- Create a repository creation template. A repository creation template gives you control to define the settings to use for new repositories created by Amazon ECR on your behalf during a pull through cache action. For more information, see Templates to control repositories created during a pull through cache or replication action.
- Validate your pull through cache rules. When validating a pull through cache rule, Amazon ECR makes a network connection with the upstream registry, verifies that it can access the Secrets Manager secret containing the credentials for the upstream registry, and that authentication was successful. For more information, see Validating pull through cache rules in Amazon ECR.

- Start using your pull through cache rules. For more information, see Pulling an image with a pull through cache rule in Amazon ECR.

Validating pull through cache rules in Amazon ECR

After you create a pull through cache rule, for upstream registries that require authentication you can validate that the rule works properly. When validating a pull through cache rule, Amazon ECR makes a network connection with the upstream registry, verifies that it can access the Secrets Manager secret containing the credentials for the upstream registry, and verifies that authentication was successful.

Before you start working with your pull through cache rules, verify that you have the proper IAM permissions. For more information, see IAM permissions required to sync an upstream registry with an Amazon ECR private registry.

To validate a pull through cache rule (AWS Management Console)

The following steps show how to validate a pull through cache rule using the Amazon ECR console.

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region containing the pull through cache rule to validate.
3. In the navigation pane, choose **Private registry**, **Pull through cache**.
4. On the **Pull through cache configuration** page, select the pull through cache rule to validate. Then, use the **Actions** drop down menu and choose **View details**.
5. On the pull through cache rule detail page, use the **Actions** drop down menu and choose **Verify authentication**. Amazon ECR will display a banner with the result.
6. Repeat these steps for each pull through cache rule you want to validate.

To validate a pull through cache rule (AWS CLI)

The validate-pull-through-cache-rule AWS CLI command is used to validate a pull through cache rule for an Amazon ECR private registry. The following example uses the `ecr-public` namespace prefix. Replace that value with the prefix value for the pull through cache rule to validate.

```
aws ecr validate-pull-through-cache-rule \
     --ecr-repository-prefix ecr-public \
```

```
    --region us-east-2
```

In the response, the `isValid` parameter indicates whether the validation was successful or not. If `true`, Amazon ECR was able to reach the upstream registry and authentication was successful. If `false`, there was an issue and validation failed. The `failure` parameter indicates the cause.

Pulling an image with a pull through cache rule in Amazon ECR

The following examples show the command syntax to use when pulling an image using a pull through cache rule. If you receive an error pulling an upstream image using a pull through cache rule, see Troubleshooting pull through cache issues in Amazon ECR for the most common errors and how to resolve them.

Before you start working with your pull through cache rules, verify that you have the proper IAM permissions. For more information, see IAM permissions required to sync an upstream registry with an Amazon ECR private registry.

> **Note**
>
> The following examples use the default Amazon ECR repository namespace values that the AWS Management Console uses. Ensure that you use the Amazon ECR private repository URI that you've configured.

For Amazon ECR Public

```
docker pull aws_account_id.dkr.ecr.region.amazonaws.com/ecr-public/repository_name/
image_name:tag
```

Kubernetes container registry

```
docker pull aws_account_id.dkr.ecr.region.amazonaws.com/kubernetes/repository_name/
image_name:tag
```

Quay

```
docker pull aws_account_id.dkr.ecr.region.amazonaws.com/quay/repository_name/
image_name:tag
```

Docker Hub

For Docker Hub official images:

```
docker pull aws_account_id.dkr.ecr.region.amazonaws.com/docker-hub/
library/image_name:tag
```

> **Note**
>
> For Docker Hub official images, the `/library` prefix must be included. For all other Docker Hub repositories, you should omit the `/library` prefix.

For all other Docker Hub images:

```
docker pull aws_account_id.dkr.ecr.region.amazonaws.com/docker-hub/repository_name/
image_name:tag
```

GitHub Container Registry

```
docker pull aws_account_id.dkr.ecr.region.amazonaws.com/github/repository_name/
image_name:tag
```

Microsoft Azure Container Registry

```
docker pull aws_account_id.dkr.ecr.region.amazonaws.com/azure/repository_name/
image_name:tag
```

GitLab Container Registry

```
docker pull aws_account_id.dkr.ecr.region.amazonaws.com/gitlab/repository_name/
image_name:tag
```

Storing your upstream repository credentials in an AWS Secrets Manager secret

When creating a pull through cache rule for an upstream repository that requires authentication, you must store the credentials in a Secrets Manager secret. There may be a cost for using an Secrets Manager secret. For more information, see AWS Secrets Manager pricing.

The following procedures walk you through how to create an Secrets Manager secret for each supported upstream repository. You can optionally use the create pull through cache rule workflow in the Amazon ECR console to create the secret instead of creating the secret using the Secrets Manager console. For more information, see Creating a pull through cache rule in Amazon ECR.

Docker Hub

To create a Secrets Manager secret for your Docker Hub credentials (AWS Management Console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Store a new secret**.
3. On the **Choose secret type** page, do the following.
   a. For **Secret type**, choose **Other type of secret**.
   b. In **Key/value pairs**, create two rows for your Docker Hub credentials. You can store up to 65536 bytes in the secret.
      i. For the first key/value pair, specify `username` as the key and your Docker Hub username as the value.
      ii. For the second key/value pair, specify `accessToken` as the key and your Docker Hub access token as the value. For more information on creating a Docker Hub access token, see Create and manage access tokens in the Docker documentation.
   c. For **Encryption key**, keep the default **aws/secretsmanager** AWS KMS key value and then choose **Next**. There is no cost for using this key. For more information, see Secret encryption and decryption in Secrets Manager in the *AWS Secrets Manager User Guide*.

> **Important**
>
> You must use the default `aws/secretsmanager` encryption key to encrypt your secret. Amazon ECR doesn't support using a customer managed key (CMK) for this.

4. On the **Configure secret** page, do the following.

   a. Enter a descriptive **Secret name** and **Description**. Secret names must contain 1-512 Unicode characters and be prefixed with `ecr-pullthroughcache/`.

   > **Important**
   >
   > The Amazon ECR AWS Management Console only displays Secrets Manager secrets with names using the `ecr-pullthroughcache/` prefix.

   b. (Optional) In the **Tags** section, add tags to your secret. For tagging strategies, see Tag Secrets Manager secrets in the *AWS Secrets Manager User Guide*. Don't store sensitive information in tags because they aren't encrypted.

   c. (Optional) In **Resource permissions**, to add a resource policy to your secret, choose **Edit permissions**. For more information, see Attach a permissions policy to an Secrets Manager secret in the *AWS Secrets Manager User Guide*.

   d. (Optional) In **Replicate secret**, to replicate your secret to another AWS Region, choose **Replicate secret**. You can replicate your secret now or come back and replicate it later. For more information, see Replicate a secret to other Regions in the *AWS Secrets Manager User Guide*.

   e. Choose **Next**.

5. (Optional) On the **Configure rotation** page, you can turn on automatic rotation. You can also keep rotation off for now and then turn it on later. For more information, see Rotate Secrets Manager secrets in the *AWS Secrets Manager User Guide*. Choose **Next**.

6. On the **Review** page, review your secret details, and then choose **Store**.

   Secrets Manager returns to the list of secrets. If your new secret doesn't appear, choose the refresh button.

GitHub Container Registry

To create an Secrets Manager secret for your GitHub Container Registry credentials (AWS Management Console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Store a new secret**.
3. On the **Choose secret type** page, do the following.
   a. For **Secret type**, choose **Other type of secret**.
   b. In **Key/value pairs**, create two rows for your GitHub credentials. You can store up to 65536 bytes in the secret.
      i. For the first key/value pair, specify `username` as the key and your GitHub username as the value.
      ii. For the second key/value pair, specify `accessToken` as the key and your GitHub access token as the value. For more information on creating a GitHub access token, see Managing your personal access tokens in the GitHub documentation.
   c. For **Encryption key**, keep the default **aws/secretsmanager** AWS KMS key value and then choose **Next**. There is no cost for using this key. For more information, see Secret encryption and decryption in Secrets Manager in the *AWS Secrets Manager User Guide*.

   > **Important**
   >
   > You must use the default `aws/secretsmanager` encryption key to encrypt your secret. Amazon ECR doesn't support using a customer managed key (CMK) for this.

4. On the **Configure secret** page, do the following:
   a. Enter a descriptive **Secret name** and **Description**. Secret names must contain 1-512 Unicode characters and be prefixed with `ecr-pullthroughcache/`.

   > **Important**
   >
   > The Amazon ECR AWS Management Console only displays Secrets Manager secrets with names using the `ecr-pullthroughcache/` prefix.

b. (Optional) In the **Tags** section, add tags to your secret. For tagging strategies, see Tag Secrets Manager secrets in the *AWS Secrets Manager User Guide*. Don't store sensitive information in tags because they aren't encrypted.

c. (Optional) In **Resource permissions**, to add a resource policy to your secret, choose **Edit permissions**. For more information, see Attach a permissions policy to an Secrets Manager secret in the *AWS Secrets Manager User Guide*.

d. (Optional) In **Replicate secret**, to replicate your secret to another AWS Region, choose **Replicate secret**. You can replicate your secret now or come back and replicate it later. For more information, see Replicate a secret to other Regions in the *AWS Secrets Manager User Guide*.

e. Choose **Next**.

5. (Optional) On the **Configure rotation** page, you can turn on automatic rotation. You can also keep rotation off for now and then turn it on later. For more information, see Rotate Secrets Manager secrets in the *AWS Secrets Manager User Guide*. Choose **Next**.

6. On the **Review** page, review your secret details, and then choose **Store**.

   Secrets Manager returns to the list of secrets. If your new secret doesn't appear, choose the refresh button.

Microsoft Azure Container Registry

To create an Secrets Manager secret for your Microsoft Azure Container Registry credentials (AWS Management Console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Store a new secret**.
3. On the **Choose secret type** page, do the following.

   a. For **Secret type**, choose **Other type of secret**.

   b. In **Key/value pairs**, create two rows for your Microsoft Azure credentials. You can store up to 65536 bytes in the secret.

      i. For the first key/value pair, specify `username` as the key and your Microsoft Azure Container Registry username as the value.

      ii. For the second key/value pair, specify `accessToken` as the key and your Microsoft Azure Container Registry access token as the value. For more information

on creating an Microsoft Azure access token, see Create token - portal in the Microsoft Azure documentation.

c. For **Encryption key**, keep the default **aws/secretsmanager** AWS KMS key value and then choose **Next**. There is no cost for using this key. For more information, see Secret encryption and decryption in Secrets Manager in the *AWS Secrets Manager User Guide*.

> **Important**
>
> You must use the default `aws/secretsmanager` encryption key to encrypt your secret. Amazon ECR doesn't support using a customer managed key (CMK) for this.

4. On the **Configure secret** page, do the following:

a. Enter a descriptive **Secret name** and **Description**. Secret names must contain 1-512 Unicode characters and be prefixed with `ecr-pullthroughcache/`.

> **Important**
>
> The Amazon ECR AWS Management Console only displays Secrets Manager secrets with names using the `ecr-pullthroughcache/` prefix.

b. (Optional) In the **Tags** section, add tags to your secret. For tagging strategies, see Tag Secrets Manager secrets in the *AWS Secrets Manager User Guide*. Don't store sensitive information in tags because they aren't encrypted.

c. (Optional) In **Resource permissions**, to add a resource policy to your secret, choose **Edit permissions**. For more information, see Attach a permissions policy to an Secrets Manager secret in the *AWS Secrets Manager User Guide*.

d. (Optional) In **Replicate secret**, to replicate your secret to another AWS Region, choose **Replicate secret**. You can replicate your secret now or come back and replicate it later. For more information, see Replicate a secret to other Regions in the *AWS Secrets Manager User Guide*.

e. Choose **Next**.

5. (Optional) On the **Configure rotation** page, you can turn on automatic rotation. You can also keep rotation off for now and then turn it on later. For more information, see Rotate Secrets Manager secrets in the *AWS Secrets Manager User Guide*. Choose **Next**.

6. On the **Review** page, review your secret details, and then choose **Store**.

   Secrets Manager returns to the list of secrets. If your new secret doesn't appear, choose the refresh button.

GitLab Container Registry

To create an Secrets Manager secret for your GitLab Container Registry credentials (AWS Management Console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Store a new secret**.
3. On the **Choose secret type** page, do the following.

   a. For **Secret type**, choose **Other type of secret**.

   b. In **Key/value pairs**, create two rows for your GitLab credentials. You can store up to 65536 bytes in the secret.

      i. For the first key/value pair, specify `username` as the key and your GitLab Container Registry username as the value.

      ii. For the second key/value pair, specify `accessToken` as the key and your GitLab Container Registry access token as the value. For more information on creating a GitLab Container Registry access token, see Personal access tokens, Group access tokens, or Project access tokens, in the GitLab documentation.

   c. For **Encryption key**, keep the default **aws/secretsmanager** AWS KMS key value and then choose **Next**. There is no cost for using this key. For more information, see Secret encryption and decryption in Secrets Manager in the *AWS Secrets Manager User Guide*.

   > **Important**
   >
   > You must use the default `aws/secretsmanager` encryption key to encrypt your secret. Amazon ECR doesn't support using a customer managed key (CMK) for this.

4. On the **Configure secret** page, do the following:

   a. Enter a descriptive **Secret name** and **Description**. Secret names must contain 1-512 Unicode characters and be prefixed with `ecr-pullthroughcache/`.

> **Important**
>
> The Amazon ECR AWS Management Console only displays Secrets Manager secrets with names using the `ecr-pullthroughcache/` prefix.

b. (Optional) In the **Tags** section, add tags to your secret. For tagging strategies, see Tag Secrets Manager secrets in the *AWS Secrets Manager User Guide*. Don't store sensitive information in tags because they aren't encrypted.

c. (Optional) In **Resource permissions**, to add a resource policy to your secret, choose **Edit permissions**. For more information, see Attach a permissions policy to an Secrets Manager secret in the *AWS Secrets Manager User Guide*.

d. (Optional) In **Replicate secret**, to replicate your secret to another AWS Region, choose **Replicate secret**. You can replicate your secret now or come back and replicate it later. For more information, see Replicate a secret to other Regions in the *AWS Secrets Manager User Guide*.

e. Choose **Next**.

5. (Optional) On the **Configure rotation** page, you can turn on automatic rotation. You can also keep rotation off for now and then turn it on later. For more information, see Rotate Secrets Manager secrets in the *AWS Secrets Manager User Guide*. Choose **Next**.

6. On the **Review** page, review your secret details, and then choose **Store**.

   Secrets Manager returns to the list of secrets. If your new secret doesn't appear, choose the refresh button.

Customizing repository prefixes for ECR to ECR pull through cache

Pull through cache rules support both the **ecr repository prefix** and the **upstream repository prefix**. The **ecr repository prefix** is the repository namespace prefix in Amazon ECR cache registry that's associated with the rule. All repositories using this prefix become pull through cache-enabled repositories for the upstream registry defined in the rule. For example, a prefix of `prod` applies to all repositories beginning with `prod/`. To apply a template to all repositories in your registry that don't have an associated pull through cache rule, use R00T as the prefix.

> **Important**
>
> There is always an assumed / applied to the end of the prefix. If you specify `ecr-public` as the prefix, Amazon ECR treats that as `ecr-public/`.

The **upstream repository prefix** matches the upstream repository name. By default, it's set to ROOT, which allows matching with any upstream repository. You can set the **upstream repository prefix** only when the Amazon ECR repository prefix has a non-ROOT value.

The following table shows the mapping between cache repository names and upstream repository names based on their prefix configurations in pull through cache rules.

Cache namespace	Upstream namespace	Mapping relationship (cache repository → upstream repository)
ecr-public	ROOT (default)	`ecr-public/my-app/image1` → `my-app/image1` `ecr-public/my-app/image2` → `my-app/image2`
ROOT	ROOT	`my-app/image1` → `my-app/image1`
team-a	team-a	`team-a/myapp/image1` → `team-a/myapp/image1`
my-app	upstream-app	`my-app/image1` → `upstream-app/image1`

Troubleshooting pull through cache issues in Amazon ECR

When pulling an upstream image using a pull through cache rule, the following are the most common errors you may receive.

Repository does not exist

An error indicating that the repository doesn't exist is most often caused by either the repository not existing in your Amazon ECR private registry or the `ecr:CreateRepository` permission not being granted to the IAM principal pulling the upstream image. To resolve this error, you should verify that the repository URI in your pull command is correct, the required IAM permissions are granted to the IAM principal pulling the upstream image, or that the repository for the upstream image to be pushed to is created in your Amazon ECR private registry before doing the upstream image pull. For more information about the required IAM permissions, see IAM permissions required to sync an upstream registry with an Amazon ECR private registry

The following is an example of this error.

```
Error response from daemon: repository 111122223333.dkr.ecr.us-east-1.amazonaws.com/
ecr-public/amazonlinux/amazonlinux not found: name unknown: The repository with
 name 'ecr-public/amazonlinux/amazonlinux' does not exist in the registry with id
 '111122223333'
```

Requested image not found

An error indicating that the image can't be found is most often caused by either the image not existing in the upstream registry or the `ecr:BatchImportUpstreamImage` permission not being granted to the IAM principal pulling the upstream image but the repository already being created in your Amazon ECR private registry. To resolve this error, you should verify the upstream image and image tag name is correct and that it exists and the required IAM permissions are granted to the IAM principal pulling the upstream image. For more information about the required IAM permissions, see IAM permissions required to sync an upstream registry with an Amazon ECR private registry.

The following is an example of this error.

```
Error response from daemon: manifest for 111122223333.dkr.ecr.us-
east-1.amazonaws.com/ecr-public/amazonlinux/amazonlinux:latest not found: manifest
 unknown: Requested image not found
```

403 Forbidden when pulling from a Docker Hub repository

When pulling from a Docker Hub repository that is tagged as a **Docker Official Image**, you must include the `/library/` in the URI you use. For example,

aws_account_id.dkr.ecr.*region*.amazonaws.com/docker-hub/**library/***image_name:tag*. If you omit the /library/ for Docker Hub Official images, a 403 Forbidden error will be returned when you attempt to pull the image using a pull through cache rule. For more information, see Pulling an image with a pull through cache rule in Amazon ECR.

The following is an example of this error.

```
Error response from daemon: failed to resolve reference "111122223333.dkr.ecr.us-
west-2.amazonaws.com/docker-hub/amazonlinux:2023": pulling from host
 111122223333.dkr.ecr.us-west-2.amazonaws.com failed with status code [manifests
 2023]: 403 Forbidden
```

Private image replication in Amazon ECR

You can configure your Amazon ECR private registry to support the replication of your repositories. Amazon ECR supports both cross-Region and cross-account replication. For cross-account replication to occur, the destination account must configure a registry permissions policy to allow replication from the source registry to occur. For more information, see Private registry permissions in Amazon ECR.

Topics

- Considerations for private image replication
- Private image replication examples for Amazon ECR
- Configuring private image replication in Amazon ECR

Considerations for private image replication

The following should be considered when using private image replication.

- Only repository content pushed to a repository after replication is configured is replicated. Any preexisting content in a repository isn't replicated. Once replication is configured for a repository, Amazon ECR keeps the destination and source synchronized.
- The repository name will remain the same across Regions and accounts when replication has occurred. Amazon ECR doesn't support changing the repository name during replication.
- The first time you configure your private registry for replication, Amazon ECR creates a service-linked IAM role on your behalf. The service-linked IAM role grants the Amazon ECR replication service the permission it needs to create repositories and replicate images in your registry. For more information, see Using service-linked roles for Amazon ECR.
- For cross-account replication to occur, the private registry destination must grant permission to allow the source registry to replicate its images. This is done by setting a private registry permissions policy. For more information, see Private registry permissions in Amazon ECR.
- If the permission policy for a private registry are changed to remove a permission, any in-progress replications previously granted may complete.
- For cross-Region replication to occur, both the source and destination accounts must be opted-in to the Region prior to any replication actions occurring within or to that Region. For more information, see Managing AWS Regions in the *Amazon Web Services General Reference*.

- Cross-Region replication is not supported between AWS partitions. For example, a repository in `us-west-2` can't be replicated to `cn-north-1`. For more information about AWS partitions, see ARN format in the *AWS General Reference*.
- The replication configuration for a private registry may contain up to 25 unique destinations across all rules, with a maximum of 10 rules total. Each rule may contain up to 100 filters. This allows for specifying separate rules for repositories containing images used for production and testing, for example.
- The replication configuration supports filtering which repositories in a private registry are replicated by specifying a repository prefix. For an example, see Example: Configuring cross-Region replication using a repository filter.
- A replication action only occurs once per image push. For example, if you configured cross-Region replication from `us-west-2` to `us-east-1` and from `us-east-1` to `us-east-2`, an image pushed to `us-west-2` replicates to only `us-east-1`, it doesn't replicate again to `us-east-2`. This behavior applies to both cross-Region and cross-account replication.
- The majority of images replicate in less than 30 minutes, but in rare cases the replication might take longer.
- Registry replication doesn't perform any delete actions. Replicated images and repositories can be manually deleted when they are no longer being used.
- Repository policies, including IAM policies, and lifecycle policies aren't replicated and don't have any effect other than on the repository they are defined for.
- Repository settings aren't replicated by default, you can replicate the repository settings using repository creation templates. These settings include tag mutability, encryption, repository permissions, and lifecycle policies. For more information about repository creation templates, see Templates to control repositories created during a pull through cache or replication action.
- If tag immutability is enabled on a repository and an image is replicated that uses the same tag as an existing image, the image is replicated but won't contain the duplicated tag. This might result in the image being untagged.

Private image replication examples for Amazon ECR

The following examples show common use cases for private image replication. If you configure replication by using the AWS CLI, you can use the JSON examples as a starting point when you create your JSON file. If you configure replication by using the AWS Management Console, you will see similar JSON when you review your replication rule on the **Review and submit** page.

Example: Configuring cross-Region replication to a single destination Region

The following shows an example for configuring cross-Region replication within a single registry. This example assumes that your account ID is 111122223333 and that you're specifying this replication configuration in a Region other than `us-west-2`.

```
{
    "rules": [
        {
            "destinations": [
                {
                    "region": "us-west-2",
                    "registryId": "111122223333"
                }
            ]
        }
    ]
}
```

Example: Configuring cross-Region replication using a repository filter

The following shows an example for configuring cross-Region replication for repositories that match a prefix name value. This example assumes your account ID is 111122223333 and that you're specifying this replication configuration in a Region other than `us-west-1` and have repositories with a prefix of `prod`.

```
{
 "rules": [{
  "destinations": [{
   "region": "us-west-1",
   "registryId": "111122223333"
  }],
  "repositoryFilters": [{
   "filter": "prod",
   "filterType": "PREFIX_MATCH"
  }]
 }]
}
```

Example: Configuring cross-Region replication to multiple destination Regions

The following shows an example for configuring cross-Region replication within a single registry. This example assumes your account ID is 111122223333 and that you're specifying this replication configuration in a Region other than `us-west-1` or `us-west-2`.

```
{
    "rules": [
        {
            "destinations": [
                {
                    "region": "us-west-1",
                    "registryId": "111122223333"
                },
                {
                    "region": "us-west-2",
                    "registryId": "111122223333"
                }
            ]
        }
    ]
}
```

Example: Configuring cross-account replication

The following shows an example for configuring cross-account replication for your registry. This example configures replication to the 444455556666 account and to the `us-west-2` Region.

> **Important**
>
> For cross-account replication to occur, the destination account must configure a registry permissions policy to allow replication to occur. For more information, see Private registry permissions in Amazon ECR.

```
{
    "rules": [
        {
```

```
            "destinations": [
                {
                    "region": "us-west-2",
                    "registryId": "444455556666"
                }
            ]
        }
    ]
}
```

Example: Specifying multiple rules in a configuration

The following shows an example for configuring multiple replication rules for your registry. This example configures replication for the *111122223333* account with one rule that replicates repositories with a prefix of `prod` to the `us-west-2` Region and repositories with a prefix of `test` to the `us-east-2` Region. A replication configuration may contain up to 10 rules, with each rule specifying up to 25 destinations.

```
{
 "rules": [{
   "destinations": [{
    "region": "us-west-2",
    "registryId": "111122223333"
   }],
   "repositoryFilters": [{
    "filter": "prod",
    "filterType": "PREFIX_MATCH"
   }]
  },
  {
   "destinations": [{
    "region": "us-east-2",
    "registryId": "111122223333"
   }],
   "repositoryFilters": [{
    "filter": "test",
    "filterType": "PREFIX_MATCH"
   }]
  }
 ]
}
```

Configuring private image replication in Amazon ECR

Configure replication per Region for your private registry. You can configure cross-Region replication or cross-account replication.

For examples of how replication is commonly used, see Private image replication examples for Amazon ECR.

To configure registry replication settings (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region to configure your registry replication settings for.
3. In the navigation pane, choose **Private registry**.
4. On the **Private registry** page, choose **Settings** and then choose **Edit** under **Replication configuration**.
5. On the **Replication** page, choose **Add replication rule**.
6. On the **Destination types** page, choose whether to enable cross-Region replication, cross-account replication, or both and then choose **Next**.
7. If cross-Region replication is enabled, then for **Configure destination regions**, choose one or more **Destination regions** and then choose **Next**.
8. If cross-account replication is enabled, then for **Cross-account replication**, choose the cross-account replication setting for the registry. For **Destination account**, enter the account ID for the destination account and one or more **Destination regions** to replicate to. Choose **Destination account +** to configure additional accounts as replication destinations.

   > **Important**
   >
   > For cross-account replication to occur, the destination account must configure a registry permissions policy to allow replication to occur. For more information, see Private registry permissions in Amazon ECR.

9. (Optional) On the **Add filters** page, specify one or more filters for the replication rule and then choose **Add**. Repeat this step for each filter you want to associate with the replication action. A filter must be specified as a repository name prefix. If no filters are added, the contents of all repositories are replicated. Choose **Next** once all filters have been added.

10. On the **Review and submit** page, review the replication rule configuration and then choose **Submit rule**.

To configure registry replication settings (AWS CLI)

1. Create a JSON file containing the replication rules to define for your registry. A replication configuration may contain up to 10 rules, with up to 25 unique destinations across all rules and 100 filters per each rule. To configure cross-Region replication within your own account, you specify your own account ID. For more examples, see Private image replication examples for Amazon ECR.

```
{
 "rules": [{
  "destinations": [{
   "region": "destination_region",
   "registryId": "destination_accountId"
  }],
  "repositoryFilters": [{
   "filter": "repository_prefix_name",
   "filterType": "PREFIX_MATCH"
  }]
 }]
}
```

2. Create a replication configuration for your registry.

```
aws ecr put-replication-configuration \
     --replication-configuration file://replication-settings.json \
     --region us-west-2
```

3. Confirm your registry settings.

```
aws ecr describe-registry \
     --region us-west-2
```

Templates to control repositories created during a pull through cache or replication action

Use Amazon ECR repository creation templates to define the settings for repositories created by Amazon ECR on your behalf. The settings in a repository creation template are only applied during repository creation and don't have any effect on existing repositories or repositories created using any other method. Currently, repository creation templates can be used to apply settings during repository creation for these features:

- Pull through cache
- Replication

Repository creation templates aren't supported in the following Regions:

- China (Beijing) (`cn-north-1`)
- China (Ningxia) (`cn-northwest-1`)
- AWS GovCloud (US-East) (`us-gov-east-1`)
- AWS GovCloud (US-West) (`us-gov-west-1`)

How repository creation templates work

There are times when Amazon ECR needs to create a new private repository on your behalf. For example:

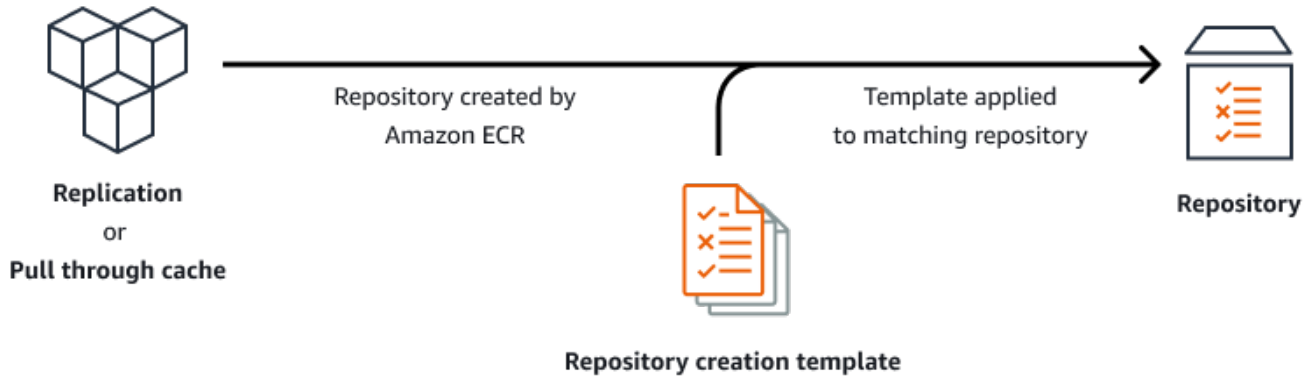
- The first time you use a pull through cache rule to retrieve the contents of an upstream repository and store it in your Amazon ECR private registry.
- When you want Amazon ECR to replicate a repository to another region or account.

When there isn't a repository creation template that matches your pull through cache rule or replicated repository, Amazon ECR uses the default settings for the new repository. These default settings include turning off tag immutability, using `AES-256` encryption, and not applying any repository or lifecycle policies.

Using a repository creation template gives you the ability to define the settings Amazon ECR applies to new repositories created through the pull through cache and replication actions. You can

define the tag immutability, encryption configuration, repository permissions, lifecycle policy, and resource tags for the new repositories.

The following diagram shows the workflow that Amazon ECR uses when a repository creation template is used with a pull through cache action.



The following describes each parameter in a repository creation template in detail.

Prefix

The **Prefix** is the repository namespace prefix to associate with the template. All repositories created using this prefix will have the settings applied that are defined in this template. For example, a prefix of `prod` would apply to all repositories beginning with `prod/`. Similarly, a prefix of `prod/team` would apply to all repositories beginning with `prod/team/`. In a registry containing two templates, if one template has the prefix "prod" and the other has the prefix "prod/team", the template with the prefix "prod/team" will be applied to all repositories whose names start with "prod/team/".

To apply a template to all repositories in your registry that don't have an associated creation template, you can use `ROOT` as the prefix.

> **Important**
>
> There is always an assumed `/` applied to the end of the prefix. If you specify `ecr-public` as the prefix, Amazon ECR treats that as `ecr-public/`. When using a pull through cache rule, the repository prefix you specify during rule creation is what you should specify as your repository creation template prefix as well.

Description

This **template description** is optional and is used to describe the purpose for the repository creation template.

Applied For

The **applied for** setting determines which Amazon ECR-created repositories will be created with this template. The valid values are `PULL_THROUGH_CACHE` and `REPLICATION`. For example, the first time you use a pull through cache rule to retrieve the contents of an upstream repository and store it in your Amazon ECR private registry. When there isn't a repository creation template that matches your pull through cache rule, Amazon ECR uses the default settings for the new repository.

Repository creation role

The **repository creation role** is an IAM role ARN that will be assumed by Amazon ECR when creating and configuring repositories via repository creation templates. This role must be provided when using repository tags and/or KMS in the template, otherwise the repository creation will fail.

Image tag mutability

The **tag mutability** setting to use for repositories created using the template. If this parameter is omitted, the default setting of **MUTABLE** will be used which will allow image tags to be overwritten. This is the recommended setting to use for templates used for repositories created by pull through cache actions. This ensures that Amazon ECR can update the cached images when the tags are the same.

If **IMMUTABLE** is specified, all image tags within the repository will be immutable which will prevent them from being overwritten.

Encryption configuration

> **Important**
>
> Dual-layer server-side encryption with AWS KMS (DSSE-KMS) is only available in the AWS GovCloud (US) Regions.

The **encryption configuration** to use for repositories created using the template.

If you use the **KMS** encryption type, the contents of the repository will be encrypted using server-side encryption with an AWS Key Management Service key stored in AWS KMS. When you use AWS KMS to encrypt your data, you can either use the default AWS managed AWS KMS key for Amazon ECR, or specify your own AWS KMS key, which you already created. You can further choose to use Single-layer or Dual-layer encryption with AWS KMS. For more information, see Encryption at rest . If you're using the **KMS** encryption type and using it with cross region replication, you may need additional permissions. For more information, see Creating a KMS key policy for replication.

If you use the **AES256** encryption type, Amazon ECR uses server-side encryption with Amazon S3-managed encryption keys which encrypts the images in the repository using an AES-256 encryption algorithm. For more information, see Protecting data using server-side encryption with Amazon S3-managed encryption keys (SSE-S3) in the *Amazon Simple Storage Service User Guide*.

Repository permissions

The **repository policy** to apply to repositories created using the template. A repository policy uses resource-based permissions to control access to a repository. Resource-based permissions let you specify which IAM users or roles have access to a repository and what actions they can perform on it. By default, only the AWS account that created the repository has access to a repository. You can apply a policy document to grant or deny additional permissions to your repository. For more information, see Private repository policies in Amazon ECR.

Repository lifecycle policy

The **lifecycle policy** to use for repositories created using the template. A lifecycle policy provides more control over the lifecycle management of images in a private repository. A lifecycle policy contains one or more rules, where each rule defines an action for Amazon ECR. This provides a way to automate the cleaning up of your container images by expiring images based on age or count. For more information, see Automate the cleanup of images by using lifecycle policies in Amazon ECR.

Resource tags

The **resource tags** are metadata to apply to the repository to help you categorize and organize them. Each tag consists of a key and an optional value, both of which you define. This permission needs to be applied on the destination registry policy if you are using repository creation templates with cross region replication.

Creating a repository creation template in Amazon ECR

You can create a repository creation template to define the settings to use for repositories created by Amazon ECR on your behalf during pull through cache or replication actions. Once the repository creation template is created, all new repositories created will have the settings applied. This doesn't have any effect on any previously created repositories.

When setting up a repository with templates, you have the option to specify KMS keys and resource tags. If you intend to use KMS keys, resource tags, or a combination of both in one or more templates, you need to:

- Create a custom policy for repository creation templates.
- Create an IAM role for repository creation templates.

Once configured, you can attach the custom role to specific templates in your registry.

IAM permissions for creating repository creation templates

The following permissions are needed for an IAM principal to manage repository creation templates. These permission must be granted using an identity-based IAM policy.

- `ecr:CreateRepositoryCreationTemplate` – Grants permission to create a repository creation template.
- `ecr:UpdateRepositoryCreationTemplate` – Grants permission to update a repository creation template.
- `ecr:DescribeRepositoryCreationTemplates` – Grants permission to list repository creation templates in a registry.
- `ecr:DeleteRepositoryCreationTemplate` – Grants permission to delete a repository creation template.
- `ecr:CreateRepository` – Grants permission to create an Amazon ECR repository.
- `ecr:PutLifecyclePolicy` – Grants permission to create a lifecycle policy and apply it to a repository. This permission is only required if the repository creation template includes a lifecycle policy.
- `ecr:SetRepositoryPolicy` – Grants permission to create a permissions policy for a repository. This permission is only required if the repository creation template includes a repository policy.

- `iam:PassRole` – Grants permission to allow an entity to pass a role to a service or application. This permission is necessary for services and applications that need to assume a role to perform actions on your behalf.

Create a custom policy for repository creation templates

You can use the AWS Management Console to define a policy that will be subsequently associated with an IAM role. This IAM role can then be utilized as a repository creation role when configuring a repository creation template.

AWS Management Console

To use the JSON policy editor to create a custom policy for repository creation templates.

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane on the left, choose **Policies**.
3. Choose **Create policy**.
4. In the **Policy editor** section, choose the **JSON** option.
5. Enter the following policy in the **JSON** field.

   JSON

```
{
        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Action": [
                    "ecr:CreateRepository",
                    "ecr:ReplicateImage",
                    "ecr:TagResource"
                ],
                "Resource": "*"
            },
            {
                "Effect": "Allow",
                "Action": [
                    "kms:CreateGrant",
```

```
                    "kms:RetireGrant",
                    "kms:DescribeKey"
                ],
                "Resource": "*"
            }
        ]
}
```

6. Resolve any security warnings, errors, or general warnings generated during policy validation, and then choose **Next**.
7. When you are finished adding permissions to the policy, choose **Next**.
8. On the **Review and create** page, type a **Policy Name** and a **Description** (optional) for the policy that you are creating. Review **Permissions defined in this policy** to see the permissions that are granted by your policy.
9. Choose **Create policy** to save your new policy.
10. Create a role to assign this policy for the creation template, see Create an IAM role for repository creation templates.

Create an IAM role for repository creation templates

You can use the AWS Management Console to create a role that can be used by Amazon ECR when you specify the repository creation role in a repository creation template that is using repository tags or KMS in a template.

AWS Management Console

To create a role.

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane of the console, choose **Roles** and then choose **Create role**.
3. Choose **Custom trust policy** role type.
4. In the **Custom trust policy** section, paste the custom trust policy listed below:

   JSON

```
{
```

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

5. Choose **Next**.
6. From the **Add permissions** page, select the check box next to the custom policy you created earlier from the list of Permissions policies and choose **Next**.
7. For **Role name**, enter a name for your role. Role names must be unique within your AWS account. When a role name is used in a policy or as part of an ARN, the role name is case sensitive. When a role name appears to customers in the console, such as during the sign-in process, the role name is case insensitive. Because various entities might reference the role, you can't edit the name of the role after it is created.
8. (Optional) For **Description**, enter a description for the new role.
9. Review the role and then choose **Create role**.

Create a repository creation template

Once you've completed the necessary prerequisites for your templates, you can proceed to create the repository creation templates.

AWS Management Console

To create a repository creation template (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region to create the repository creation template in.
3. In the navigation pane, choose **Private registry**, **Repository creation templates**.
4. On the **Repository creation templates** page, choose **Create template**.
5. On the **Step 1: Define template** page, for **Template details**, choose **A specific prefix** to apply the template to a specific repository namespace prefix or choose **Any prefix in your**

ECR registry to apply the template to all repositories that don't match any other template in the Region.

a. If you choose **A specific prefix**, for **Prefix** specify the repository namespace prefix to apply the template to. There is always an assumed / applied to the end of the prefix. For example, a prefix of `prod` would apply to all repositories beginning with `prod/`. Similarly, a prefix of `prod/team` would apply to all repositories beginning with `prod/team/`.

b. If you choose **Any prefix in your ECR registry**, the **Prefix** will be set to `ROOT`.

6. For **Applied for**, specify which Amazon ECR workflows this template will apply to. The options are `PULL_THROUGH_CACHE` and `REPLICATION`.

7. For **Template description**, specify an optional description for the template and then choose **Next**.

8. On the **Step 2: Add repository creation configuration** page, specify the repository setting configuration to apply to repositories created using the template.

a. For **Image tag mutability**, choose the tag mutability setting to use. For more information, see Preventing image tags from being overwritten in Amazon ECR.

- **Mutable** – Choose this option if you want image tags to be overwritten. Recommended for repositories using pull through cache actions to ensure Amazon ECR can update cached images. Additionally, to disable tag updates for a few mutable tags, enter tag names or use wildcards (*) to match multiple similar tags in the **Mutable tag exclusion** text box.
- **Immutable** – Choose this option if you want to prevent image tags from being overwritten, and it applies to all tags and exclusions in the repository when pushing an image with existing tag. Amazon ECR returns an `ImageTagAlreadyExistsException` if you attempt to push an image with an existing tag. Additionally, to enable tag updates for a few immutable tags, enter tag names or use wildcards (*) to match multiple similar tags in the **Immutable tag exclusion** text box.

b. For **Encryption configuration**, choose the encryption setting to use. For more information, see Encryption at rest.

When **AES-256** is selected, Amazon ECR uses server-side encryption with Amazon Simple Storage Service-managed encryption keys which encrypts your data at rest

using an industry standard AES-256 encryption algorithm. This is offered at no additional cost.

When **AWS KMS** is selected, Amazon ECR uses server-side encryption with keys stored in AWS Key Management Service (AWS KMS). When you use AWS KMS to encrypt your data, you can either use the default AWS managed key, which is managed by Amazon ECR, or specify your own AWS KMS key, which is referred to as a *customer managed key*.

> **Note**
>
> The encryption settings for a repository can't be changed once the repository is created.

c. For **Repository permissions**, specify the repository permissions policy to apply to repositories created using this template. You can optionally use the drop down to select one of the JSON samples for the most common use cases. For more information, see Private repository policies in Amazon ECR.

d. For **Repository lifecycle policy**, specify the repository lifecycle policy to apply to repositories created using this template. You can optionally use the drop down to select one of the JSON samples for the most common use cases. For more information, see Automate the cleanup of images by using lifecycle policies in Amazon ECR.

e. For **Repository AWS tags**, specify the metadata, in the form of key-value pairs, to associate with the repositories created using this template and then choose **Next**. For more information, see Tagging a private repository in Amazon ECR.

f. For **Repository creation role**, select a custom IAM role from the drop-down menu to be used for repository creation templates when using repository tags or KMS in the template (see Create an IAM role for repository creation templates for details).Then choose **Next**.

9. On the **Step 3: Review and create** page, review the settings you specified for the repository creation template. Choose the **Edit** option to make changes. Choose **Create** once you're done.

AWS CLI

The create-repository-creation-template AWS CLI command is used to create a repository creation template for your private registry.

To create a repository creation template (AWS CLI)

1. Use the AWS CLI to generate a skeleton for the create-repository-creation-template command.

```
aws ecr create-repository-creation-template \
    --generate-cli-skeleton
```

The output of the command displays the full syntax of the repository creation template.

```
{
"appliedFor":[""], // string array, but valid are PULL_THROUGH_CACHE and
 REPLICATION
"prefix": "string",
    "description": "string",
    "imageTagMutability":
 "MUTABLE"|"IMMUTABLE"|"IMMUTABLE_WITH_EXCLUSION"|"MUTABLE_WITH_EXCLUSION",
    "imageTagMutabilityExclusionFilters": [
        "filterType": "WILDCARD",
        "filter": "string"
    ],
    "repositoryPolicy": "string",
    "lifecyclePolicy": "string"
"encryptionConfiguration": {
"encryptionType": "AES256"|"KMS",
        "kmsKey": "string"
    },
    "resourceTags": [
        {
"Key": "string",
            "Value": "string"
        }
    ],
    "customRoleArn": "string", // must be a valid IAM Role ARN
}
```

2. Create a file named `repository-creation-template.json` with the output of the previous step. This template sets a KMS encryption key for any repository created under `prod/*` with a repository policy that enables pushing and pulling images to future repositories, sets a lifecycle policy that will expire images older than two weeks and sets a

custom role that will let ECR access the KMS key and assign the resource tag `examplekey` to future repositories.

```
{
"prefix": "prod",
    "description": "For repositories cached from my PTC rule and in my
 replication configuration that start with 'prod/'",
    "appliedFor": ["PULL_THROUGH_CACHE","REPLICATION"],
    "encryptionConfiguration": {
"encryptionType": "KMS",
        "kmsKey": "arn:aws:kms:us-west-2:111122223333:key/a1b2c3d4-5678-90ab-
cdef-example11111"
    },
    "resourceTags": [
        {
"Key": "examplekey",
            "Value": "examplevalue"
        }
    ],
    "imageTagMutability": "IMMUTABLE_WITH_EXCLUSION",
    "imageTagMutabilityExclusionFilters": [
      {
      "filterType": "WILDCARD",
      "filter": "latest"
      },
      {
      "filterType": "WILDCARD",
      "filter": "beta*"
      }
    ]
    "repositoryPolicy": "{\"Version\":\"2012-10-17\",\"Statement\":[{\"Sid
\":\"AllowPushPullIAMRole\",\"Effect\":\"Allow\",\"Principal\":{\"AWS\":
\"arn:aws:iam::111122223333:user\/IAMusername\"},\"Action\":[\"ecr:BatchGetImage
\",\"ecr:BatchCheckLayerAvailability\",\"ecr:CompleteLayerUpload\",
\"ecr:GetDownloadUrlForLayer\",\"ecr:InitiateLayerUpload\",\"ecr:PutImage\",
\"ecr:UploadLayerPart\"]}]}",
    "lifecyclePolicy": "{\"rules\":[{\"rulePriority\":1,\"description\":\"Expire
 images older than 14 days\",\"selection\":{\"tagStatus\":\"any\",\"countType
\":\"sinceImagePushed\",\"countUnit\":\"days\",\"countNumber\":14},\"action\":
{\"type\":\"expire\"}}]}",
    "customRoleArn": "arn:aws:iam::111122223333:role/myRole"
}
```

3. Use the following command to create a repository creation template. Ensure that you specify the name of the configuration file created in the previous step in place of the `repository-creation-template.json` in the following example.

```
aws ecr create-repository-creation-template \
    --cli-input-json file://repository-creation-template.json
```

Updating a repository creation template

You can edit a repository creation template if you need to change its configurations. Once the repository creation template is edited, the new configurations will apply to the existing template.

> **Important**
>
> This doesn't have any effect on any previously created repositories.

AWS Management Console

To edit a repository creation template (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region the repository creation template to edit is in.
3. In the navigation pane, choose **Private registry**, then choose **Settings**.
4. From the navigation bar, choose the **Repository creation templates**.
5. On the **Repository creation templates** page, select the repository creation template to edit.
6. From the **Actions** dropdown menu, choose **Edit**.
7. Review and update the configuration settings.
8. Choose update to apply the new creation template configurations.

AWS CLI

To edit a repository creation template (AWS CLI)

- Use the update-repository-creation-template command to update an existing repository creation template. You must specify the `prefix` value of the template. The following example updates a repository creation template with the `prod` prefix.

```
aws ecr update-repository-creation-template \
     --prefix prod \
     --image-tag-mutability="IMMUTABLE_WITH_EXCLUSION" \
     --image-tag-mutability-exclusion-filters filterType=WILDCARD, filter=latest
```

  The output of the command displays the details of the updated repository creation template.

Deleting a repository creation template in Amazon ECR

You can delete a repository creation template if you are finished using it. Once a repository creation template is deleted, any newly created repositories under the associated prefix during a pull through cache or replication action will inherit the default settings, unless another matching template is found, see How repository creation templates work.

> **Important**
>
> This doesn't have any effect on any previously created repositories.

AWS Management Console

To delete a repository creation template (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region the repository creation template to delete is in.
3. In the navigation pane, choose **Private registry**, **Repository creation templates**.
4. On the **Repository creation templates** page, select the repository creation template to delete.
5. From the **Actions** dropdown menu, choose **Delete**.

AWS CLI

To delete a repository creation template (AWS CLI)

- Use the delete-repository-creation-template.html command to delete an existing repository creation template. You must specify the `prefix` value of the template. The following example deletes a repository creation template with the `prod` prefix.

```
aws ecr delete-repository-creation-template \
     --prefix prod
```

The output of the command displays the details of the deleted repository creation template.

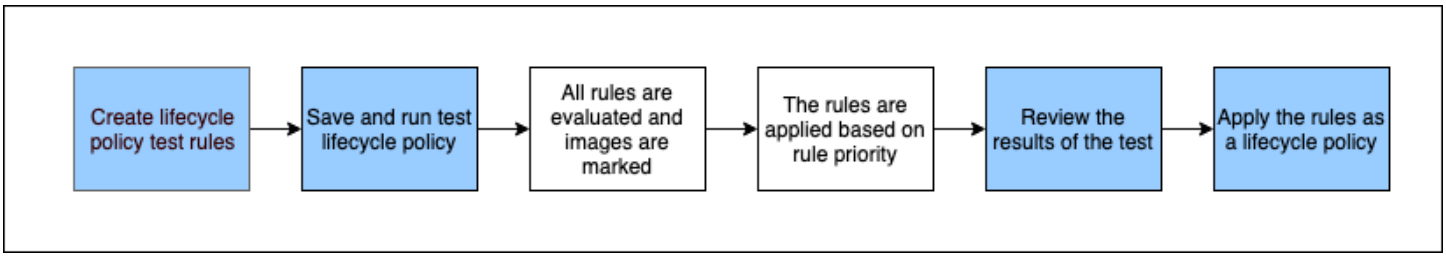
Automate the cleanup of images by using lifecycle policies in Amazon ECR

Amazon ECR lifecycle policies provide more control over the lifecycle management of images in a private repository. A lifecycle policy contains one or more rules, and each rule defines an action for Amazon ECR. Based on the expiration criteria in the lifecycle policy, images expire based on age or count within 24 hours. When Amazon ECR performs an action based on a lifecycle policy, this action is captured as an event in AWS CloudTrail. For more information, see Logging Amazon ECR actions with AWS CloudTrail.

How lifecycle policies work

A lifecycle policy consists of one or more rules that determine which images in a repository should be expired. When considering the use of lifecycle policies, it's important to use the lifecycle policy preview to confirm which images the lifecycle policy expires before applying it to a repository. Once a lifecycle policy is applied to a repository, you should expect that images become expired within 24 hours after they meet the expiration criteria. When Amazon ECR performs an action based on a lifecycle policy, this is captured as an event in AWS CloudTrail. For more information, see Logging Amazon ECR actions with AWS CloudTrail.

The following diagram shows the lifecycle policy workflow.



1. Create one or more test rules.
2. Save the test rules and run the preview.
3. The lifecycle policy evaluator goes through all of the rules and marks the images that each rule affects.
4. The lifecycle policy evaluator then applies the rules, based on rule priority, and displays which images in the repository are set to be expired. A lower rule priority number means higher priority. For example, a rule with priority 1 takes precedence over a rule with priority 2.

5. Review the results of the test, ensuring that the images that are marked to be expired are what you intended.
6. Apply the test rules as the lifecycle policy for the repository.
7. Once the lifecycle policy is created, you should expect that images become expired within 24 hours after they meet the expiration criteria.

Lifecycle policy evaluation rules

The lifecycle policy evaluator is responsible for parsing the plaintext JSON of the lifecycle policy, evaluating all rules, and then applying those rules based on rule priority to the images in the repository. The following explains the logic of the lifecycle policy evaluator in more detail. For examples, see Examples of lifecycle policies in Amazon ECR.

- When reference artifacts are present in a repository, Amazon ECR lifecycle policies automatically clean up those artifacts within 24 hours of the deletion of the subject image.
- All rules are evaluated at the same time, regardless of rule priority. After all rules are evaluated, they are then applied based on rule priority.
- An image is expired by exactly one or zero rules.
- An image that matches the tagging requirements of a rule cannot be expired by a rule with a lower priority.
- Rules can never mark images that are marked by higher priority rules, but can still identify them as if they haven't been expired.
- The set of rules must contain a unique set of tag prefixes.
- Only one rule is allowed to select untagged images.
- If an image is referenced by a manifest list, it cannot be expired without the manifest list being deleted first.
- Expiration is always ordered by `pushed_at_time`, and always expires older images before newer ones.
- A lifecycle policy rule may specify either `tagPatternList` or `tagPrefixList`, but not both. However, a lifecycle policy may contain multiple rules where different rules use both pattern and prefix lists. An image is successfully matched if all of the tags in the `tagPatternList` or `tagPrefixList` value are matched against any of the image's tags.
- The `tagPatternList` or `tagPrefixList` parameters may only used if the `tagStatus` is `tagged`.

- When using `tagPatternList`, an image is successfully matched if it matches the wildcard filter. For example, if a filter of `prod*` is applied, it would match image-tags whose name begins with `prod` such as `prod`, `prod1`, or `production-team1`. Similarly, if a filter of `*prod*` is applied, it would match image-tags whose name contains `prod` such as `repo-production` or `prod-team`.

  Important

  There is a maximum limit of four wildcards (*) per string. For example, `["*test*1*2*3", "test*1*2*3*"]` is valid but `["test*1*2*3*4*5*6"]` is invalid.

- When using `tagPrefixList`, an image is successfully matched if ***all*** of the wildcard filters in the `tagPrefixList` value are matched against any of the image's tags.
- The `countUnit` parameter is only used if `countType` is `sinceImagePushed`.
- With `countType = imageCountMoreThan`, images are sorted from youngest to oldest based on `pushed_at_time` and then all images greater than the specified count are expired.
- With `countType = sinceImagePushed`, all images whose `pushed_at_time` is older than the specified number of days based on `countNumber` are expired.

Creating a lifecycle policy preview in Amazon ECR

You can use a lifecycle policy preview to see the impact of a lifecycle policy on an image repository before you apply it. It is considered best practice to do a preview before applying a lifecycle policy to a repository.

Note

If you are using Amazon ECR replication to make copies of a repository across different Regions or accounts, note that a lifecycle policy can only take an action on repositories in the Region it was created in. Therefore, if you have replication turned on you may want to create a lifecycle policy in each Region and account you are replicating your repositories to.

To create a lifecycle policy preview (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region that contains the repository on which to perform a lifecycle policy preview.

3. In the navigation pane, under **Private registry**, choose **Repositories**.
4. On the **Private repositories** page, select a repository and that use the **Actions** drop down to choose **Lifecycle policies**.
5. On the lifecycle policy rules page for the repository, choose **Edit test rules**, **Create rule**.
6. Specify the following details for each test lifecycle policy rule.

   a. For **Rule priority**, type a number for the rule priority. The rule priority determines in what order the lifecycle policy rules are applied. A lower number means higher priority. For example, a rule with priority 1 takes precedence over a rule with priority 2.

   b. For **Rule description**, type a description for the lifecycle policy rule.

   c. For **Image status**, choose **Tagged (wildcard matching)**, **Tagged (prefix matching)**, **Untagged**, or **Any**.

   > **Important**
   >
   > If you specify multiple tags, only the images with all specified tags are selected.

   d. If you chose **Tagged (wildcard matching)** for **Image status**, then for **Specify tags for wildcard matching**, you can specify a list of image tags with a wildcard (*) on which to take action with your lifecycle policy. For example, if your images are tagged as `prod`, `prod1`, `prod2`, and so on, you would specify `prod*` to take action on all of them. If you specify multiple tags, only the images with all specified tags are selected.

   > **Important**
   >
   > There is a maximum limit of four wildcards (*) per string. For example, `["*test*1*2*3", "test*1*2*3*"]` is valid but `["test*1*2*3*4*5*6"]` is invalid.

   e. If you chose **Tagged (prefix matching)** for **Image status**, then for **Specify tags for prefix matching**, you can specify a list of image tags on which to take action with your lifecycle policy.

   f. For **Match criteria**, choose either **Since image pushed** or **Image count more than** and then specify a value.

   g. Choose **Save**.

7. Create additional test lifecycle policy rules by repeating steps 5–7.

8. To run the lifecycle policy preview, choose **Save and run test**.
9. Under **Image matches for test lifecycle rules**, review the impact of your lifecycle policy preview.
10. If you are satisfied with the preview results, choose **Apply as lifecycle policy** to create a lifecycle policy with the specified rules. You should expect that after applying a lifecycle policy, the affected images are expired within 24 hours.
11. If you aren't satisfied with the preview results, you may delete one or more test lifecycle rules and create one or more rules to replace them and then repeat the test.

Creating a lifecycle policy for a repository in Amazon ECR

Use a lifecycle policy to create a set of rules that expire unused repository images. After creating a lifecycle policy, the affected images are expired within 24 hours.

> **Note**
>
> If you are using Amazon ECR replication to make copies of a repository across different Regions or accounts, note that a lifecycle policy can only take an action on repositories in the Region it was created in. Therefore, if you have replication turned on you may want to create a lifecycle policy in each Region and account you are replicating your repositories to.

Prerequisite

Best practice: Create a lifecycle policy preview to verify that the images expired by your lifecycle policy rules are what you intend. For instructions, see Creating a lifecycle policy preview in Amazon ECR.

To create a lifecycle policy (AWS Management Console)

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/repositories.
2. From the navigation bar, choose the Region that contains the repository for which to create a lifecycle policy.
3. In the navigation pane, under **Private registry**, choose **Repositories**.
4. On the **Private repositories** page, select a repository and that use the **Actions** drop down to choose **Lifecycle policies**.

5. On the lifecycle policy rules page for the repository, choose **Create rule**.
6. Enter the following details for your lifecycle policy rule.

   a. For **Rule priority**, type a number for the rule priority. The rule priority determines in what order the lifecycle policy rules are applied. A lower rule priority number means higher priority. For example, a rule with priority 1 takes precedence over a rule with priority 2.

   b. For **Rule description**, type a description for the lifecycle policy rule.

   c. For **Image status**, choose **Tagged (wildcard matching)**, **Tagged (prefix matching)**, **Untagged**, or **Any**.

   > **Important**
   >
   > If you specify multiple tags, only the images with all specified tags are selected.

   d. If you chose **Tagged (wildcard matching)** for **Image status**, then for **Specify tags for wildcard matching**, you can specify a list of image tags with a wildcard (*) on which to take action with your lifecycle policy. For example, if your images are tagged as `prod`, `prod1`, `prod2`, and so on, you would specify `prod*` to take action on all of them. If you specify multiple tags, only the images with all specified tags are selected.

   > **Important**
   >
   > There is a maximum limit of four wildcards (*) per string. For example, `["*test*1*2*3", "test*1*2*3*"]` is valid but `["test*1*2*3*4*5*6"]` is invalid.

   e. If you chose **Tagged (prefix matching)**for **Image status**, then for **Specify tags for prefix matching**, you can specify a list of image tags on which to take action with your lifecycle policy.

   f. For **Match criteria**, choose either **Since image pushed** or **Image count more than** and then specify a value.

   g. Choose **Save**.

7. Create additional lifecycle policy rules by repeating steps 5–7.

To create a lifecycle policy (AWS CLI)

1. Obtain the name of the repository for which to create the lifecycle policy.

```
aws ecr describe-repositories
```

2. Create a local file named `policy.json` with the contents of the lifecycle policy. For lifecycle policy examples, see Examples of lifecycle policies in Amazon ECR.
3. Create a lifecycle policy by specifying the repository name and reference the lifecycle policy JSON file you created.

```
aws ecr put-lifecycle-policy \
      --repository-name repository-name \
      --lifecycle-policy-text file://policy.json
```

Examples of lifecycle policies in Amazon ECR

The following are example lifecycle policies showing the syntax.

To see more information about policy properties, see Lifecycle policy properties in Amazon ECR. For instructions about creating a lifecycle policy by using the AWS CLI, see To create a lifecycle policy (AWS CLI).

Lifecycle policy template

The contents of your lifecycle policy are evaluated before being associated with a repository. The following is the JSON syntax template for the lifecycle policy.

```
{
        "rules": [
            {
                "rulePriority": integer,
                "description": "string",
                "selection": {
                    "tagStatus": "tagged"|"untagged"|"any",
                    "tagPatternList": list<string>,
                    "tagPrefixList": list<string>,
                    "countType": "imageCountMoreThan"|"sinceImagePushed",
                    "countUnit": "string",
                    "countNumber": integer
```

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

Filtering on image age

The following example shows the lifecycle policy syntax for a policy that expires images with a tag starting with `prod` by using a `tagPatternList` of `prod*` that are also older than 14 days.

```
{
    "rules": [
        {
            "rulePriority": 1,
            "description": "Expire images older than 14 days",
            "selection": {
                "tagStatus": "tagged",
                "tagPatternList": ["prod*"],
                "countType": "sinceImagePushed",
                "countUnit": "days",
                "countNumber": 14
            },
            "action": {
                "type": "expire"
            }
        }
    ]
}
```

Filtering on image count

The following example shows the lifecycle policy syntax for a policy that keeps only one untagged image and expires all others.

```
{
    "rules": [
        {
            "rulePriority": 1,
            "description": "Keep only one untagged image, expire all others",
```

```
            "selection": {
                "tagStatus": "untagged",
                "countType": "imageCountMoreThan",
                "countNumber": 1
            },
            "action": {
                "type": "expire"
            }
        }
    ]
}
```

Filtering on multiple rules

The following examples use multiple rules in a lifecycle policy. An example repository and lifecycle policy are given along with an explanation of the outcome.

Example A

Repository contents:

- Image A, Taglist: ["beta-1", "prod-1"], Pushed: 10 days ago
- Image B, Taglist: ["beta-2", "prod-2"], Pushed: 9 days ago
- Image C, Taglist: ["beta-3"], Pushed: 8 days ago

Lifecycle policy text:

```
{
    "rules": [
        {
            "rulePriority": 1,
            "description": "Rule 1",
            "selection": {
                "tagStatus": "tagged",
                "tagPatternList": ["prod*"],
                "countType": "imageCountMoreThan",
                "countNumber": 1
            },
            "action": {
                "type": "expire"
            }
```

```
        },
        {
            "rulePriority": 2,
            "description": "Rule 2",
            "selection": {
                "tagStatus": "tagged",
                "tagPatternList": ["beta*"],
                "countType": "imageCountMoreThan",
                "countNumber": 1
            },
            "action": {
                "type": "expire"
            }
        }
    ]
}
```

The logic of this lifecycle policy would be:

- Rule 1 identifies images tagged with prefix `prod`. It should mark images, starting with the oldest, until there is one or fewer images remaining that match. It marks Image A for expiration.
- Rule 2 identifies images tagged with prefix `beta`. It should mark images, starting with the oldest, until there is one or fewer images remaining that match. It marks both Image A and Image B for expiration. However, Image A has already been seen by Rule 1 and if Image B were expired it would violate Rule 1 and thus is skipped.
- Result: Image A is expired.

Example B

This is the same repository as the previous example but the rule priority order is changed to illustrate the outcome.

Repository contents:

- Image A, Taglist: ["beta-1", "prod-1"], Pushed: 10 days ago
- Image B, Taglist: ["beta-2", "prod-2"], Pushed: 9 days ago
- Image C, Taglist: ["beta-3"], Pushed: 8 days ago

Lifecycle policy text:

```
{
    "rules": [
        {
            "rulePriority": 1,
            "description": "Rule 1",
            "selection": {
                "tagStatus": "tagged",
                "tagPatternList": ["beta*"],
                "countType": "imageCountMoreThan",
                "countNumber": 1
            },
            "action": {
                "type": "expire"
            }
        },
        {
            "rulePriority": 2,
            "description": "Rule 2",
            "selection": {
                "tagStatus": "tagged",
                "tagPatternList": ["prod*"],
                "countType": "imageCountMoreThan",
                "countNumber": 1
            },
            "action": {
                "type": "expire"
            }
        }
    ]
}
```

The logic of this lifecycle policy would be:

- Rule 1 identifies images tagged with prefix `beta`. It should mark images, starting with the oldest, until there is one or fewer images remaining that match. It sees all three images and would mark Image A and Image B for expiration.
- Rule 2 identifies images tagged with prefix `prod`. It should mark images, starting with the oldest, until there is one or fewer images remaining that match. It would see no images because all available images were already seen by Rule 1 and thus would mark no additional images.
- Result: Images A and B are expired.

Filtering on multiple tags in a single rule

The following examples specify the lifecycle policy syntax for multiple tag patterns in a single rule. An example repository and lifecycle policy are given along with an explanation of the outcome.

Example A

When multiple tag patterns are specified on a single rule, images must match all listed tag patterns.

Repository contents:

- Image A, Taglist: ["alpha-1"], Pushed: 12 days ago
- Image B, Taglist: ["beta-1"], Pushed: 11 days ago
- Image C, Taglist: ["alpha-2", "beta-2"], Pushed: 10 days ago
- Image D, Taglist: ["alpha-3"], Pushed: 4 days ago
- Image E, Taglist: ["beta-3"], Pushed: 3 days ago
- Image F, Taglist: ["alpha-4", "beta-4"], Pushed: 2 days ago

```
{
    "rules": [
        {
            "rulePriority": 1,
            "description": "Rule 1",
            "selection": {
                "tagStatus": "tagged",
                "tagPatternList": ["alpha*", "beta*"],
                "countType": "sinceImagePushed",
                "countNumber": 5,
                "countUnit": "days"
            },
            "action": {
                "type": "expire"
            }
        }
    ]
}
```

The logic of this lifecycle policy would be:

- Rule 1 identifies images tagged with prefix `alpha` and `beta`. It sees images C and F. It should mark images that are older than five days, which would be Image C.
- Result: Image C is expired.

Example B

The following example illustrates that tags are not exclusive.

Repository contents:

- Image A, Taglist: ["alpha-1", "beta-1", "gamma-1"], Pushed: 10 days ago
- Image B, Taglist: ["alpha-2", "beta-2"], Pushed: 9 days ago
- Image C, Taglist: ["alpha-3", "beta-3", "gamma-2"], Pushed: 8 days ago

```
{
    "rules": [
        {
            "rulePriority": 1,
            "description": "Rule 1",
            "selection": {
                "tagStatus": "tagged",
                "tagPatternList": ["alpha*", "beta*"],
                "countType": "imageCountMoreThan",
                "countNumber": 1
            },
            "action": {
                "type": "expire"
            }
        }
    ]
}
```

The logic of this lifecycle policy would be:

- Rule 1 identifies images tagged with prefix `alpha` and `beta`. It sees all images. It should mark images, starting with the oldest, until there is one or fewer images remaining that match. It marks image A and B for expiration.
- Result: Images A and B are expired.

Filtering on all images

The following lifecycle policy examples specify all images with different filters. An example repository and lifecycle policy are given along with an explanation of the outcome.

Example A

The following shows the lifecycle policy syntax for a policy that applies to all rules but keeps only one image and expires all others.

Repository contents:

- Image A, Taglist: ["alpha-1"], Pushed: 4 days ago
- Image B, Taglist: ["beta-1"], Pushed: 3 days ago
- Image C, Taglist: [], Pushed: 2 days ago
- Image D, Taglist: ["alpha-2"], Pushed: 1 day ago

```
{
    "rules": [
        {
            "rulePriority": 1,
            "description": "Rule 1",
            "selection": {
                "tagStatus": "any",
                "countType": "imageCountMoreThan",
                "countNumber": 1
            },
            "action": {
                "type": "expire"
            }
        }
    ]
}
```

The logic of this lifecycle policy would be:

- Rule 1 identifies all images. It sees images A, B, C, and D. It should expire all images other than the newest one. It marks images A, B, and C for expiration.
- Result: Images A, B, and C are expired.

Example B

The following example illustrates a lifecycle policy that combines all the rule types in a single policy.

Repository contents:

- Image A, Taglist: ["alpha-1", "beta-1"], Pushed: 4 days ago
- Image B, Taglist: [], Pushed: 3 days ago
- Image C, Taglist: ["alpha-2"], Pushed: 2 days ago
- Image D, Taglist: ["git hash"], Pushed: 1 day ago
- Image E, Taglist: [], Pushed: 1 day ago

```
{
    "rules": [
        {
            "rulePriority": 1,
            "description": "Rule 1",
            "selection": {
                "tagStatus": "tagged",
                "tagPatternList": ["alpha*"],
                "countType": "imageCountMoreThan",
                "countNumber": 1
            },
            "action": {
                "type": "expire"
            }
        },
        {
            "rulePriority": 2,
            "description": "Rule 2",
            "selection": {
                "tagStatus": "untagged",
                "countType": "sinceImagePushed",
                "countUnit": "days",
                "countNumber": 1
            },
            "action": {
                "type": "expire"
            }
        },
```

```
        {
            "rulePriority": 3,
            "description": "Rule 3",
            "selection": {
                "tagStatus": "any",
                "countType": "imageCountMoreThan",
                "countNumber": 1
            },
            "action": {
                "type": "expire"
            }
        }
    ]
}
```

The logic of this lifecycle policy would be:

- Rule 1 identifies images tagged with prefix `alpha`. It identifies images A and C. It should keep the newest image and mark the rest for expiration. It marks image A for expiration.
- Rule 2 identifies untagged images. It identifies images B and E. It should mark all images older than one day for expiration. It marks image B for expiration.
- Rule 3 identifies all images. It identifies images A, B, C, D, and E. It should keep the newest image and mark the rest for expiration. However, it can't mark images A, B, C, or E because they were identified by higher priority rules. It marks image D for expiration.
- Result: Images A, B, and D are expired.

Lifecycle policy properties in Amazon ECR

Lifecycle policies have the following properties.

To see examples of lifecycle policies, see Examples of lifecycle policies in Amazon ECR. For instructions about creating a lifecycle policy by using the AWS CLI, see To create a lifecycle policy (AWS CLI).

Rule priority

`rulePriority`

Type: integer

Required: yes

Sets the order in which rules are applied, lowest to highest. A lifecycle policy rule with a priority of 1 is applied first, a rule with priority of 2 is next, and so on. When you add rules to a lifecycle policy, you must give them each a unique value for `rulePriority`. Values don't need to be sequential across rules in a policy. A rule with a `tagStatus` value of `any` must have the highest value for `rulePriority` and be evaluated last.

Description

`description`

Type: string

Required: no

(Optional) Describes the purpose of a rule within a lifecycle policy.

Tag status

`tagStatus`

Type: string

Required: yes

Determines whether the lifecycle policy rule that you are adding specifies a tag for an image. Acceptable options are `tagged`, `untagged`, or `any`. If you specify `any`, then all images have the rule evaluated against them. If you specify `tagged`, then you must also specify a `tagPrefixList` value. If you specify `untagged`, then you must omit `tagPrefixList`.

Tag pattern list

`tagPatternList`

Type: list[string]

Required: yes, if `tagStatus` is set to tagged and `tagPrefixList` isn't specified

When creating a lifecycle policy for tagged images, it's best practice to use a `tagPatternList` to specify the tags to expire. You specify a comma-separated list of image tag patterns that may contain wildcards (*) on which to take action with your lifecycle policy. For example, if your images are tagged as `prod`, `prod1`, `prod2`, and so on, you would use the tag pattern list `prod*` to specify all of them. If you specify multiple tags, only the images with all specified tags are selected.

> **Important**
>
> There is a maximum limit of four wildcards (*) per string. For example, `["*test*1*2*3", "test*1*2*3*"]` is valid but `["test*1*2*3*4*5*6"]` is invalid.

Tag prefix list

`tagPrefixList`

Type: list[string]

Required: yes, if `tagStatus` is set to tagged and `tagPatternList` isn't specified

Only used if you specified `"tagStatus": "tagged"` and you aren't specifying a `tagPatternList`. You must specify a comma-separated list of image tag prefixes on which to take action with your lifecycle policy. For example, if your images are tagged as `prod`, `prod1`, `prod2`, and so on, you would use the tag prefix `prod` to specify all of them. If you specify multiple tags, only the images with all specified tags are selected.

Count type

`countType`

Type: string

Required: yes

Specify a count type to apply to the images.

If `countType` is set to `imageCountMoreThan`, you also specify `countNumber` to create a rule that sets a limit on the number of images that exist in your repository. If `countType` is set to

`sinceImagePushed`, you also specify `countUnit` and `countNumber` to specify a time limit on the images that exist in your repository.

Count unit

`countUnit`

Type: string

Required: yes, only if `countType` is set to `sinceImagePushed`

Specify a count unit of `days` to indicate that as the unit of time, in addition to `countNumber`, which is the number of days.

This should only be specified when `countType` is `sinceImagePushed`; an error will occur if you specify a count unit when `countType` is any other value.

Count number

`countNumber`

Type: integer

Required: yes

Specify a count number. Acceptable values are positive integers (`0` is not an accepted value).

If the `countType` used is `imageCountMoreThan`, then the value is the maximum number of images that you want to retain in your repository. If the `countType` used is `sinceImagePushed`, then the value is the maximum age limit for your images.

Action

`type`

Type: string

Required: yes

Specify an action type. The supported value is `expire`.

Security in Amazon Elastic Container Registry

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to Amazon ECR, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Amazon ECR. The following topics show you how to configure Amazon ECR to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Amazon ECR resources.

Topics

- Identity and Access Management for Amazon Elastic Container Registry
- Data protection in Amazon ECR
- Compliance validation for Amazon Elastic Container Registry
- Infrastructure Security in Amazon Elastic Container Registry
- Cross-service confused deputy prevention

Identity and Access Management for Amazon Elastic Container Registry

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in)

and *authorized* (have permissions) to use Amazon ECR resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How Amazon Elastic Container Registry works with IAM
- Amazon Elastic Container Registry Identity-based policy examples
- Using Tag-Based Access Control
- AWS managed policies for Amazon Elastic Container Registry
- Using service-linked roles for Amazon ECR
- Troubleshooting Amazon Elastic Container Registry Identity and Access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Amazon ECR.

Service user – If you use the Amazon ECR service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Amazon ECR features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Amazon ECR, see Troubleshooting Amazon Elastic Container Registry Identity and Access.

Service administrator – If you're in charge of Amazon ECR resources at your company, you probably have full access to Amazon ECR. It's your job to determine which Amazon ECR features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Amazon ECR, see How Amazon Elastic Container Registry works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Amazon ECR. To view example Amazon ECR identity-based policies that you can use in IAM, see Amazon Elastic Container Registry Identity-based policy examples.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.

- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.
  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific

resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session

policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon Elastic Container Registry works with IAM

Before you use IAM to manage access to Amazon ECR, you should understand what IAM features are available to use with Amazon ECR. To get a high-level view of how Amazon ECR and other AWS services work with IAM, see AWS Services That Work with IAM in the *IAM User Guide*.

Topics

- Amazon ECR Identity-based policies
- Amazon ECR resource-based policies
- Authorization based on Amazon ECR tags
- Amazon ECR IAM roles

Amazon ECR Identity-based policies

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. Amazon ECR supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a JSON policy, see IAM JSON Policy Elements Reference in the *IAM User Guide*.

Actions

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

Policy actions in Amazon ECR use the following prefix before the action: `ecr:`. For example, to grant someone permission to create an Amazon ECR repository with the Amazon ECR `CreateRepository` API operation, you include the `ecr:CreateRepository` action in their policy. Policy statements must include either an `Action` or `NotAction` element. Amazon ECR defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
      "ecr:action1",
      "ecr:action2"
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Describe`, include the following action:

```
"Action": "ecr:Describe*"
```

To see a list of Amazon ECR actions, see Actions, Resources, and Condition Keys for Amazon Elastic Container Registry in the *IAM User Guide*.

Resources

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

An Amazon ECR repository resource has the following ARN:

```
arn:${Partition}:ecr:${Region}:${Account}:repository/${Repository-name}
```

For more information about the format of ARNs, see Amazon Resource Names (ARNs) and AWS Service Namespaces.

For example, to specify the `my-repo` repository in the `us-east-1` Region in your statement, use the following ARN:

```
"Resource": "arn:aws:ecr:us-east-1:123456789012:repository/my-repo"
```

To specify all repositories that belong to a specific account, use the wildcard (*):

```
"Resource": "arn:aws:ecr:us-east-1:123456789012:repository/*"
```

To specify multiple resources in a single statement, separate the ARNs with commas.

```
"Resource": [
      "resource1",
      "resource2"
```

To see a list of Amazon ECR resource types and their ARNs, see Resources Defined by Amazon Elastic Container Registry in the *IAM User Guide*. To learn with which actions you can specify the ARN of each resource, see Actions Defined by Amazon Elastic Container Registry.

Condition keys

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

Amazon ECR defines its own set of condition keys and also supports using some global condition keys. To see all AWS global condition keys, see AWS Global Condition Context Keys in the *IAM User Guide*.

Most Amazon ECR actions support the `aws:ResourceTag` and `ecr:ResourceTag` condition keys. For more information, see Using Tag-Based Access Control.

To see a list of Amazon ECR condition keys, see Condition Keys Defined by Amazon Elastic Container Registry in the *IAM User Guide*. To learn with which actions and resources you can use a condition key, see Actions Defined by Amazon Elastic Container Registry.

Examples

To view examples of Amazon ECR identity-based policies, see Amazon Elastic Container Registry Identity-based policy examples.

Amazon ECR resource-based policies

Resource-based policies are JSON policy documents that specify what actions a specified principal can perform on an Amazon ECR resource and under what conditions. Amazon ECR supports resource-based permissions policies for Amazon ECR repositories. Resource-based policies let you grant usage permission to other accounts on a per-resource basis. You can also use a resource-based policy to allow an AWS service to access your Amazon ECR repositories.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, you must also grant the principal entity permission to access the resource. Grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, you don't need additional Amazon ECR repository permissions in the identity-based policy. For more information, see How IAM Roles Differ from Resource-based Policies in the *IAM User Guide*.

The Amazon ECR service supports only one type of resource-based policy called a *repository policy*, which is attached to a *repository*. This policy defines which principal entities (accounts, users, roles, and federated users) can perform actions on the repository. To learn how to attach a resource-based policy to a repository, see Private repository policies in Amazon ECR.

> **Note**
>
> In an Amazon ECR repository policy, the policy element `Sid` supports additional characters and spacing not supported in IAM policies.

Examples

To view examples of Amazon ECR resource-based policies, see Private repository policy examples in Amazon ECR,

Authorization based on Amazon ECR tags

You can attach tags to Amazon ECR resources or pass tags in a request to Amazon ECR. To control access based on tags, you provide tag information in the condition element of a policy using the `ecr:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`*, or `aws:TagKeys` condition keys. For more information about tagging Amazon ECR resources, see Tagging a private repository in Amazon ECR.

To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see Using Tag-Based Access Control.

Amazon ECR IAM roles

An IAM role is an entity within your AWS account that has specific permissions.

Using Temporary Credentials with Amazon ECR

You can use temporary credentials to sign in with federation, assume an IAM role, or to assume a cross-account role. You obtain temporary security credentials by calling AWS STS API operations such as AssumeRole or GetFederationToken.

Amazon ECR supports using temporary credentials.

Service-linked roles

Service-linked roles allow AWS services to access resources in other services to complete an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view but not edit the permissions for service-linked roles.

Amazon ECR supports service-linked roles. For more information, see Using service-linked roles for Amazon ECR.

Amazon Elastic Container Registry Identity-based policy examples

By default, users and roles don't have permission to create or modify Amazon ECR resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Amazon ECR, including the format of the ARNs for each of the resource types, see Actions, resources, and condition keys for Amazon Elastic Container Registry in the *Service Authorization Reference*.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating Policies on the JSON Tab in the *IAM User Guide*.

Topics

- Policy Best Practices
- Using the Amazon ECR console
- Allow Users to View Their Own Permissions
- Accessing One Amazon ECR Repository

Policy Best Practices

Identity-based policies determine whether someone can create, access, or delete Amazon ECR resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Amazon ECR console

To access the Amazon Elastic Container Registry console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Amazon ECR resources in your AWS account. If you create an identity-based policy that is more restrictive than

the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

To ensure that those entities can still use the Amazon ECR console, add the `AmazonEC2ContainerRegistryReadOnly` AWS managed policy to the entities. For more information, see Adding Permissions to a User in the *IAM User Guide*:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetRepositoryPolicy",
                "ecr:DescribeRepositories",
                "ecr:ListImages",
                "ecr:DescribeImages",
                "ecr:BatchGetImage",
                "ecr:GetLifecyclePolicy",
                "ecr:GetLifecyclePolicyPreview",
                "ecr:ListTagsForResource",
                "ecr:DescribeImageScanFindings"
            ],
            "Resource": "*"
        }
    ]
}
```

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that you're trying to perform.

Allow Users to View Their Own Permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Accessing One Amazon ECR Repository

In this example, you want to grant a user in your AWS account access to one of your Amazon ECR repositories, `my-repo`. You also want to allow the user to push, pull, and list images.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
```

```
      {
         "Sid":"GetAuthorizationToken",
         "Effect":"Allow",
         "Action":[
            "ecr:GetAuthorizationToken"
         ],
         "Resource":"*"
      },
      {
         "Sid":"ManageRepositoryContents",
         "Effect":"Allow",
         "Action":[
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetRepositoryPolicy",
                "ecr:DescribeRepositories",
                "ecr:ListImages",
                "ecr:DescribeImages",
                "ecr:BatchGetImage",
                "ecr:InitiateLayerUpload",
                "ecr:UploadLayerPart",
                "ecr:CompleteLayerUpload",
                "ecr:PutImage"
         ],
         "Resource":"arn:aws:ecr:us-east-1:123456789012:repository/my-repo"
      }
   ]
}
```

Using Tag-Based Access Control

The Amazon ECR `CreateRepository` API action enables you to specify tags when you create the repository. For more information, see Tagging a private repository in Amazon ECR.

To enable users to tag repositories on creation, they must have permissions to use the action that creates the resource (for example, `ecr:CreateRepository`). If tags are specified in the resource-creating action, Amazon performs additional authorization on the `ecr:CreateRepository` action to verify if users have permissions to create tags.

You can use tag-based access control through IAM policies. The following are examples.

The following policy would only allow a user to create or tag a repository as `key=environment,value=dev`.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateTaggedRepository",
            "Effect": "Allow",
            "Action": [
                "ecr:CreateRepository"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/environment": "dev"
                }
            }
        },
        {
            "Sid": "AllowTagRepository",
            "Effect": "Allow",
            "Action": [
                "ecr:TagResource"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/environment": "dev"
                }
            }
        }
    ]
}
```

The following policy would allow a user to pull images from all repositories unless they were tagged as `key=environment,value=prod`.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [{
            "Effect": "Allow",
            "Action": [
                "ecr:BatchGetImage",
                "ecr:GetDownloadUrlForLayer"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": [
                "ecr:BatchGetImage",
                "ecr:GetDownloadUrlForLayer"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "ecr:ResourceTag/environment": "prod"
                }
            }
        }
    ]
}
```

AWS managed policies for Amazon Elastic Container Registry

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

Amazon ECR provides several managed policies that you can attach to IAM identities or to Amazon EC2 instances. These managed policies allow differing levels of control over access to Amazon ECR resources and API operations. For more information about each API operation mentioned in these policies, see Actions in the *Amazon Elastic Container Registry API Reference*.

Topics

- AmazonEC2ContainerRegistryFullAccess
- AmazonEC2ContainerRegistryPowerUser
- AmazonEC2ContainerRegistryPullOnly
- AmazonEC2ContainerRegistryReadOnly
- AWSECRPullThroughCache_ServiceRolePolicy
- ECRReplicationServiceRolePolicy
- ECRTemplateServiceRolePolicy
- Amazon ECR updates to AWS managed policies

`AmazonEC2ContainerRegistryFullAccess`

You can attach the `AmazonEC2ContainerRegistryFullAccess` policy to your IAM identities.

You can use this managed policy as a starting point to create your own IAM policy based on your specific requirements. For example, you can create a policy specifically for providing a user or role with full administrator access to manage the use of Amazon ECR. With the Amazon ECR Lifecycle Policies feature, you can specify the lifecycle management of images in a repository. Lifecycle policy events are reported as CloudTrail events. Amazon ECR is integrated with AWS CloudTrail so it can display your lifecycle policy events directly in the Amazon ECR console. The `AmazonEC2ContainerRegistryFullAccess` managed IAM policy includes the `cloudtrail:LookupEvents` permission to facilitate this behavior.

Permissions details

This policy includes the following permissions:

- `ecr` – Allows principals full access to all Amazon ECR APIs.
- `cloudtrail` – Allows principals to looks up management events or AWS CloudTrail Insights events that are captured by CloudTrail.

The `AmazonEC2ContainerRegistryFullAccess` policy is as follows.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecr:*",
                "cloudtrail:LookupEvents"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": [
                        "replication.ecr.amazonaws.com"
                    ]
                }
            }
        }
    ]
}
```

`AmazonEC2ContainerRegistryPowerUser`

You can attach the `AmazonEC2ContainerRegistryPowerUser` policy to your IAM identities.

This policy grants administrative permissions that allow IAM users to read and write to repositories, but doesn't allow them to delete repositories or change the policy documents that are applied to them.

Permissions details

This policy includes the following permissions:

- `ecr` – Allows principals to read and write to repositories, as well as read lifecycle policies. Principals aren't granted permission to delete repositories or change the lifecycle policies that are applied to them.

The `AmazonEC2ContainerRegistryPowerUser` policy is as follows.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetRepositoryPolicy",
                "ecr:DescribeRepositories",
                "ecr:ListImages",
                "ecr:DescribeImages",
                "ecr:BatchGetImage",
                "ecr:GetLifecyclePolicy",
                "ecr:GetLifecyclePolicyPreview",
                "ecr:ListTagsForResource",
                "ecr:DescribeImageScanFindings",
                "ecr:InitiateLayerUpload",
                "ecr:UploadLayerPart",
                "ecr:CompleteLayerUpload",
                "ecr:PutImage"
            ],
            "Resource": "*"
        }
    ]
}
```

AmazonEC2ContainerRegistryPullOnly

You can attach the AmazonEC2ContainerRegistryPullOnly policy to your IAM identities.

This policy grants permission to pull container images from Amazon ECR. If the registry is enabled for pull-through cache, it will also allow pulls to import an image from an upstream registry.

Permissions details

This policy includes the following permissions:

- ecr – Allows principals to read repositories and their respective lifecycle policies.

The AmazonEC2ContainerRegistryPullOnly policy is as follows.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecr:GetAuthorizationToken",
                "ecr:BatchGetImage",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchImportUpstreamImage"
            ],
            "Resource": "*"
        }
    ]
}
```

AmazonEC2ContainerRegistryReadOnly

You can attach the AmazonEC2ContainerRegistryReadOnly policy to your IAM identities.

This policy grants read-only permissions to Amazon ECR. This includes the ability to list repositories and images within the repositories. It also includes the ability to pull images from Amazon ECR with the Docker CLI.

Permissions details

This policy includes the following permissions:

- `ecr` – Allows principals to read repositories and their respective lifecycle policies.

The `AmazonEC2ContainerRegistryReadOnly` policy is as follows.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetRepositoryPolicy",
                "ecr:DescribeRepositories",
                "ecr:ListImages",
                "ecr:DescribeImages",
                "ecr:BatchGetImage",
                "ecr:GetLifecyclePolicy",
                "ecr:GetLifecyclePolicyPreview",
                "ecr:ListTagsForResource",
                "ecr:DescribeImageScanFindings"
            ],
            "Resource": "*"
        }
    ]
}
```

`AWSECRPullThroughCache_ServiceRolePolicy`

You can't attach the `AWSECRPullThroughCache_ServiceRolePolicy` managed IAM policy to your IAM entities. This policy is attached to a service-linked role that allows Amazon ECR to push images to your repositories through the pull through cache workflow. For more information, see Amazon ECR service-linked role for pull through cache.

`ECRReplicationServiceRolePolicy`

You can't attach the `ECRReplicationServiceRolePolicy` managed IAM policy to your IAM entities. This policy is attached to a service-linked role that allows Amazon ECR to perform actions on your behalf. For more information, see Using service-linked roles for Amazon ECR.

`ECRTemplateServiceRolePolicy`

You can't attach the `ECRTemplateServiceRolePolicy` managed IAM policy to your IAM entities. This policy is attached to a service-linked role that allows Amazon ECR to perform actions on your behalf. For more information, see Using service-linked roles for Amazon ECR.

Amazon ECR updates to AWS managed policies

View details about updates to AWS managed policies for Amazon ECR since the time that this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Amazon ECR Document history page.

Change	Description	Date
Amazon ECR service-linked role for pull through cache – Update to an existing policy	Amazon ECR added new permissions to the `AWSECRPullThroughCache_ServiceRolePolicy` policy. These permissions allow Amazon ECR to pull images from ECR private registry. This is required when using a pull through cache rule to cache images from another Amazon ECR private registry.	March 12, 2025
AmazonEC2ContainerRegistryPullOnly – New policy	Amazon ECR added a new policy which grants pull-only permissions to Amazon ECR.	October 10, 2024
ECRTemplateServiceRolePolicy – New policy	Amazon ECR added a new policy. This policy is associated with the `ECRTemplateServiceRolePolicy` service-linked role for the	June 20, 2024

Change	Description	Date
	repository creation template feature.	
AWSECRPullThroughCache_ServiceRolePolicy – Update to an existing policy	Amazon ECR added new permissions to the `AWSECRPullThroughCache_ServiceRolePolicy` policy. These permissions allow Amazon ECR to retrieve the encrypted contents of a Secrets Manager secret. This is required when using a pull through cache rule to cache images from an upstream registry that requires authentication.	November 15, 2023
AWSECRPullThroughCache_ServiceRolePolicy – New policy	Amazon ECR added a new policy. This policy is associated with the `AWSServiceRoleForECRPullThroughCache` service-linked role for the pull through cache feature.	November 29, 2021
ECRReplicationServiceRolePolicy – New policy	Amazon ECR added a new policy. This policy is associated with the `AWSServiceRoleForECRReplication` service-linked role for the replication feature.	December 4, 2020

Change	Description	Date
AmazonEC2ContainerRegistryFullAccess – Update to an existing policy	Amazon ECR added new permissions to the `AmazonEC2ContainerRegistryFullAccess` policy. These permissions allow principals to create the Amazon ECR service-linked role.	December 4, 2020
AmazonEC2ContainerRegistryReadOnly – Update to an existing policy	Amazon ECR added new permissions to the `AmazonEC2ContainerRegistryReadOnly` policy which allow principals to read lifecycle policies, list tags, and describe the scan findings for images.	December 10, 2019
AmazonEC2ContainerRegistryPowerUser – Update to an existing policy	Amazon ECR added new permissions to the `AmazonEC2ContainerRegistryPowerUser` policy. They allow principals to read lifecycle policies, list tags, and describe the scan findings for images.	December 10, 2019

Change	Description	Date
AmazonEC2ContainerRegistryFullAccess – Update to an existing policy	Amazon ECR added new permissions to the `AmazonEC2ContainerRegistryFullAccess` policy. They allow principals to look up management events or AWS CloudTrail Insights events that are captured by CloudTrail.	November 10, 2017
AmazonEC2ContainerRegistryReadOnly – Update to an existing policy	Amazon ECR added new permissions to the `AmazonEC2ContainerRegistryReadOnly` policy. They allow principals to describe Amazon ECR images.	October 11, 2016
AmazonEC2ContainerRegistryPowerUser – Update to an existing policy	Amazon ECR added new permissions to the `AmazonEC2ContainerRegistryPowerUser` policy. They allow principals to describe Amazon ECR images.	October 11, 2016

Change	Description	Date
AmazonEC2ContainerRegistryReadOnly – New policy	Amazon ECR added a new policy which grants read-only permissions to Amazon ECR. These permissions include the ability to list repositories and images within the repositories. They also include the ability to pull images from Amazon ECR with the Docker CLI.	December 21, 2015
AmazonEC2ContainerRegistryPowerUser – New policy	Amazon ECR added a new policy which grants administrative permissions that allow users to read and write to repositories but doesn't allow them to delete repositories or change the policy documents that are applied to them.	December 21, 2015
AmazonEC2ContainerRegistryFullAccess – New policy	Amazon ECR added a new policy. This policy grants full access to Amazon ECR.	December 21, 2015
Amazon ECR started tracking changes	Amazon ECR started tracking changes for AWS managed policies.	June 24, 2021

Using service-linked roles for Amazon ECR

Amazon Elastic Container Registry (Amazon ECR) uses AWS Identity and Access Management (IAM) service-linked roles to provide the permissions necessary to use the replication and pull through cache features. A service-linked role is a unique type of IAM role that is linked directly to Amazon ECR. The service-linked role is predefined by Amazon ECR. It includes all of the permissions that the service requires to support the replication and pull through cache features for your private

registry. After you configure replication or pull through cache for your registry, a service-linked role is created automatically on your behalf. For more information, see Private registry settings in Amazon ECR.

A service-linked role makes setting up replication and pull through cache with Amazon ECR easier. This is because, by using it, you don't have to manually add all the necessary permissions. Amazon ECR defines the permissions of its service-linked roles, and unless defined otherwise, only Amazon ECR can assume its roles. The defined permissions include the trust policy and the permissions policy. The permissions policy can't be attached to any other IAM entity.

You can delete the corresponding service-linked role only after disabling either replication or pull through cache on your registry. This ensures that you don't inadvertently remove the permissions Amazon ECR requires for these features.

For information about other services that support service-linked roles, see AWS services that work with IAM. On this linked-to page, look for the services that have **Yes** in the **Service-linked role** column. Choose a **Yes** with a link to view the relevant service-linked role documentation for that service.

Topics

- Supported Regions for Amazon ECR service-linked roles
- Amazon ECR service-linked role for replication
- Amazon ECR service-linked role for pull through cache
- Amazon ECR service-linked role for repository creation templates

Supported Regions for Amazon ECR service-linked roles

Amazon ECR supports using service-linked roles in all of the Regions where the Amazon ECR service is available. For more information about Amazon ECR Region availability, see AWS Regions and Endpoints.

Amazon ECR service-linked role for replication

Amazon ECR uses a service-linked role named **AWSServiceRoleForECRReplication** that allows Amazon ECR to replicate images across multiple accounts.

Service-linked role permissions for Amazon ECR

The AWSServiceRoleForECRReplication service-linked role trusts the following services to assume the role:

- `replication.ecr.amazonaws.com`

The following `ECRReplicationServiceRolePolicy` role permissions policy allows Amazon ECR to use the following actions on resources:

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecr:CreateRepository",
                "ecr:ReplicateImage"
            ],
            "Resource": "*"
        }
    ]
}
```

Note

The `ReplicateImage` is an internal API that Amazon ECR uses for replication and can't be called directly.

You must configure permissions to allow an IAM entity (for example a user, group, or role) to create, edit, or delete a service-linked role. For more information, see Service-Linked Role Permissions in the *IAM User Guide*.

Creating a service-linked role for Amazon ECR

You don't need to manually create the Amazon ECR service-linked role. When you configure replication settings for your registry in the AWS Management Console, the AWS CLI, or the AWS API, Amazon ECR creates the service-linked role for you.

If you delete this service-linked role and need to create it again, you can use the same process to recreate the role in your account. When you configure replication settings for your registry, Amazon ECR creates the service-linked role for you again.

Editing a service-linked role for Amazon ECR

Amazon ECR doesn't allow manually editing the AWSServiceRoleForECRReplication service-linked role. After you create a service-linked role, you can't change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting the service-linked role for Amazon ECR

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way, you don’t have an unused entity that isn't actively monitored or maintained. However, you must remove the replication configuration for your registry in every Region before you can manually delete the service-linked role.

> **Note**
>
> If you try to delete resources while the Amazon ECR service is still using the roles, your delete action might fail. If that happens, wait for a few minutes and try again.

To delete Amazon ECR resources used by the AWSServiceRoleForECRReplication

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region your replication configuration is set on.
3. In the navigation pane, choose **Private registry**.
4. On the **Private registry** page, on the **Replication configuration** section, choose **Edit**.
5. To delete all of your replication rules, choose **Delete all**. This step requires confirmation.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the **AWSServiceRoleForECRReplication** service-linked role. For more information, see Deleting a Service-Linked Role in the *IAM User Guide*.

Amazon ECR service-linked role for pull through cache

Amazon ECR uses a service-linked role named **AWSServiceRoleForECRPullThroughCache** which gives permission for Amazon ECR to perform actions on your behalf to complete pull through cache actions. For more information about pull through cache, see Templates to control repositories created during a pull through cache or replication action.

Service-linked role permissions for Amazon ECR

The **AWSServiceRoleForECRPullThroughCache** service-linked role trusts the following service to assume the role.

- `pullthroughcache.ecr.amazonaws.com`

Permissions details

The `AWSECRPullThroughCache_ServiceRolePolicy` permissions policy is attached to the service-linked role. This managed policy grants Amazon ECR permission to perform the following actions. For more information, see `AWSECRPullThroughCache_ServiceRolePolicy`.

- `ecr` – Allows the Amazon ECR service to pull and push images to a private repository.
- `secretsmanager:GetSecretValue` – Allows the Amazon ECR service to retrieve the encrypted contents of an AWS Secrets Manager secret. This is required when using a pull through cache rule to cache images from an upstream registry that requires authentication in your private registry. This permission applies only to secrets with the `ecr-pullthroughcache/` name prefix.

The `AWSECRPullThroughCache_ServiceRolePolicy` policy contains the following JSON.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Sid": "ECR",
            "Effect": "Allow",
            "Action": [
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:InitiateLayerUpload",
                "ecr:UploadLayerPart",
                "ecr:CompleteLayerUpload",
                "ecr:PutImage",
                "ecr:BatchGetImage",
                "ecr:BatchImportUpstreamImage",
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetImageCopyStatus"
            ],
            "Resource": "*"
        },
        {
            "Sid": "SecretsManager",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "arn:aws:secretsmanager:*:*:secret:ecr-pullthroughcache/
*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceAccount": "${aws:PrincipalAccount}"
                }
            }
        }
    ]
}
```

You must configure permissions to allow an IAM entity (for example a user, group, or role) to create, edit, or delete a service-linked role. For more information, see Service-linked role permissions in the *IAM User Guide*.

Creating a service-linked role for Amazon ECR

You don't need to manually create the Amazon ECR service-linked role for pull through cache. When you create a pull through cache rule for your private registry in the AWS Management Console, the AWS CLI, or the AWS API, Amazon ECR creates the service-linked role for you.

If you delete this service-linked role and need to create it again, you can use the same process to recreate the role in your account. When you create a pull through cache rule for your private registry, Amazon ECR creates the service-linked role for you again if it doesn't already exist.

Editing a service-linked role for Amazon ECR

Amazon ECR doesn't allow manually editing the **AWSServiceRoleForECRPullThroughCache** service-linked role. After the service-linked role is created, you can't change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting the service-linked role for Amazon ECR

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way, you don't have an unused entity that isn't actively monitored or maintained. However, you must delete the pull through cache rules for your registry in every Region before you can manually delete the service-linked role.

> **Note**
>
> If you try to delete resources while the Amazon ECR service is still using the role, your delete action might fail. If that happens, wait for a few minutes and try again.

To delete Amazon ECR resources used by the AWSServiceRoleForECRPullThroughCache service-linked role

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region where your pull through cache rules are created.
3. In the navigation pane, choose **Private registry**.
4. On the **Private registry** page, on the **Pull through cache configuration** section, choose **Edit**.
5. For each pull through cache rule you have created, select the rule and then choose **Delete rule**.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the **AWSServiceRoleForECRPullThroughCache** service-linked role. For more information, see Deleting a Service-Linked Role in the *IAM User Guide*.

Amazon ECR service-linked role for repository creation templates

Amazon ECR uses a service-linked role named **AWSServiceRoleForECRTemplate** which gives permission for Amazon ECR to perform actions on your behalf to complete repository creation template actions.

Service-linked role permissions for Amazon ECR

The **AWSServiceRoleForECRTemplate** service-linked role trusts the following service to assume the role.

- `ecr.amazonaws.com`

Permissions details

The ECRTemplateServiceRolePolicy permissions policy is attached to the service-linked role. This managed policy grants Amazon ECR permission to perform repository creation actions on your behalf.

The `ECRTemplateServiceRolePolicy` policy contains the following JSON.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
     "Sid": "CreateRepositoryWithTemplate",
            "Effect": "Allow",
            "Action": [
                "ecr:CreateRepository"
            ],
            "Resource": "*"
        }
    ]
}
```

You must configure permissions to allow an IAM entity (for example a user, group, or role) to create, edit, or delete a service-linked role. For more information, see Service-linked role permissions in the *IAM User Guide*.

Creating a service-linked role for Amazon ECR

You don't need to manually create the Amazon ECR service-linked role for repository creation template. When you create a repository creation template rule for your private registry in the AWS Management Console, the AWS CLI, or the AWS API, Amazon ECR creates the service-linked role for you.

If you delete this service-linked role and need to create it again, you can use the same process to recreate the role in your account. When you create a repository creation rule for your private registry, Amazon ECR creates the service-linked role for you again if it doesn't already exist.

Editing a service-linked role for Amazon ECR

Amazon ECR doesn't allow manually editing the **AWSServiceRoleForECRTemplate** service-linked role. After the service-linked role is created, you can't change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting the service-linked role for Amazon ECR

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way, you don’t have an unused entity that isn't actively monitored or maintained. However, you must delete the repository creation rules for your registry in every Region before you can manually delete the service-linked role.

> **Note**
>
> If you try to delete resources while the Amazon ECR service is still using the role, your delete action might fail. If that happens, wait for a few minutes and try again.

To delete Amazon ECR resources used by the AWSServiceRoleForECRTemplate service-linked role

1. Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.
2. From the navigation bar, choose the Region where your repository creation rules are created.
3. In the navigation pane, choose **Private registry**.
4. On the **Private registry** page, on the **Repository creation templates** section, choose **Edit**.

5. For each repository creation rule you have created, select the rule and then choose **Delete rule**.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the **AWSServiceRoleForECRTemplate** service-linked role. For more information, see Deleting a Service-Linked Role in the *IAM User Guide*.

Troubleshooting Amazon Elastic Container Registry Identity and Access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Amazon ECR and IAM.

Topics

- I Am Not Authorized to Perform an Action in Amazon ECR
- I Am Not Authorized to Perform iam:PassRole
- I want to allow people outside of my AWS account to access my Amazon ECR resources

I Am Not Authorized to Perform an Action in Amazon ECR

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `ecr:GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 ecr:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `ecr:GetWidget` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I Am Not Authorized to Perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Amazon ECR.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Amazon ECR. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Amazon ECR resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Amazon ECR supports these features, see How Amazon Elastic Container Registry works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Data protection in Amazon ECR

The AWS shared responsibility model applies to data protection in Amazon Elastic Container Service. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Amazon ECS or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Topics

- Encryption at rest

Encryption at rest

> **Important**
>
> Dual-layer server-side encryption with AWS KMS (DSSE-KMS) is only available in the AWS GovCloud (US) Regions.

Amazon ECR stores images in Amazon S3 buckets that Amazon ECR manages. By default, Amazon ECR uses server-side encryption with Amazon S3-managed encryption keys which encrypts your data at rest using an AES-256 encryption algorithm. This does not require any action on your part and is offered at no additional charge. For more information, see Protecting Data Using Server-Side Encryption with Amazon S3-Managed Encryption Keys (SSE-S3) in the *Amazon Simple Storage Service User Guide*.

For more control over the encryption for your Amazon ECR repositories, you can use server-side encryption with KMS keys stored in AWS Key Management Service (AWS KMS). When you use AWS KMS to encrypt your data, you can either use the default AWS managed key, which is managed by Amazon ECR, or specify your own KMS key (referred to as a customer managed key). For more information, see Protecting Data Using Server-Side Encryption with KMS keys Stored in AWS KMS (SSE-KMS) in the *Amazon Simple Storage Service User Guide*.

You can choose to apply two layers of encryption to your Amazon ECR images using dual-layer server-side encryption with AWS KMS (DSSE-KMS). DSSE-KMS option is similar to SSE-KMS, but applies two individual layers of encryption instead of one layer. For more information, see Using dual-layer server-side encryption with AWS KMS keys (DSSE-KMS).

Each Amazon ECR repository has an encryption configuration, which is set when the repository is created. You can use different encryption configurations on each repository. For more information, see Creating an Amazon ECR private repository to store images.

When a repository is created with AWS KMS encryption enabled, a KMS key is used to encrypt the contents of the repository. Moreover, Amazon ECR adds an AWS KMS grant to the KMS key with the Amazon ECR repository as the grantee principal.

The following provides a high-level understanding of how Amazon ECR is integrated with AWS KMS to encrypt and decrypt your repositories:

1. When creating a repository, Amazon ECR sends a DescribeKey call to AWS KMS to validate and retrieve the Amazon Resource Name (ARN) of the KMS key specified in the encryption configuration.
2. Amazon ECR sends two CreateGrant requests to AWS KMS to create grants on the KMS key to allow Amazon ECR to encrypt and decrypt data using the data key.
3. When pushing an image, a GenerateDataKey request is made to AWS KMS that specifies the KMS key to use for encrypting the image layer and manifest.
4. AWS KMS generates a new data key, encrypts it under the specified KMS key, and sends the encrypted data key to be stored with the image layer metadata and the image manifest.
5. When pulling an image, a Decrypt request is made to AWS KMS, specifying the encrypted data key.
6. AWS KMS decrypts the encrypted data key and sends the decrypted data key to Amazon S3.
7. The data key is used to decrypt the image layer before the image layer being pulled.
8. When a repository is deleted, Amazon ECR sends two RetireGrant requests to AWS KMS to retire the grants created for the repository.

Considerations

The following points should be considered when using AWS KMS based encryption (SSE-KMS or DSSE-KMS) with Amazon ECR.

- If you create your Amazon ECR repository with KMS encryption and you do not specify a KMS key, Amazon ECR uses an AWS managed key with the alias `aws/ecr` by default. This KMS key is created in your account the first time that you create a repository with KMS encryption enabled.
- Repository Encryption Configuration can't be changed after a repository is created.
- When you use KMS encryption with your own KMS key, the key must exist in the same Region as your repository.
- The grants that Amazon ECR creates on your behalf should not be revoked. If you revoke the grant that gives Amazon ECR permission to use the AWS KMS keys in your account, Amazon ECR cannot access this data, encrypt new images pushed to the repository, or decrypt them when they are pulled. When you revoke a grant for Amazon ECR, the change occurs immediately. To revoke access rights, you should delete the repository rather than revoking the grant. When a repository is deleted, Amazon ECR retires the grants on your behalf.
- There is a cost associated with using AWS KMS keys. For more information, see AWS Key Management Service pricing.

- There is a cost associated with using dual-layer server-side encryption. For more information, see Amazon ECR pricing

Required IAM permissions

When creating or deleting an Amazon ECR repository with server-side encryption using AWS KMS, the permissions required depend on the specific KMS key you are using.

Required IAM permissions when using the AWS managed key for Amazon ECR

By default, when AWS KMS encryption is enabled for an Amazon ECR repository but no KMS key is specified, the AWS managed key for Amazon ECR is used. When the AWS-managed KMS key for Amazon ECR is used to encrypt a repository, any principal that has permission to create a repository can also enable AWS KMS encryption on the repository. However, the IAM principal that deletes the repository must have the `kms:RetireGrant` permission. This enables the retirement of the grants that were added to the AWS KMS key when the repository was created.

The following example IAM policy can be added as an inline policy to a user to ensure they have the minimum permissions needed to delete a repository that has encryption enabled. The KMS key used to encrypt the repository can be specified using the resource parameter.

JSON

```
{
    "Version": "2012-10-17",
    "Id": "ecr-kms-permissions",
    "Statement": [
        {
            "Sid": "AllowAccessToRetireTheGrantsAssociatedWithTheKey",
            "Effect": "Allow",
            "Action": [
                "kms:RetireGrant"
            ],
            "Resource": "arn:aws:kms:us-
west-2:111122223333:key/b8d9ae76-080c-4043-92EXAMPLE"
        }
    ]
}
```

Required IAM permissions when using a customer managed key

When creating a repository with AWS KMS encryption enabled using a customer managed key, there are required permissions for both the KMS key policy and the IAM policy for the user or role creating the repository.

When creating your own KMS key, you can either use the default key policy AWS KMS creates or you can specify your own. To ensure that the customer managed key remains manageable by the account owner, the key policy for the KMS key should allow all AWS KMS actions for the root user of the account. Additional scoped permissions may be added to the key policy but at minimum the root user should be given permissions to manage the KMS key. To allow the KMS key to be used only for requests that originate in Amazon ECR, you can use the kms:ViaService condition key with the `ecr.<region>.amazonaws.com` value.

The following example key policy gives the AWS account (root user) that owns the KMS key full access to the KMS key. For more information about this example key policy, see Allows access to the AWS account and enables IAM policies in the *AWS Key Management Service Developer Guide*.

JSON

```
{
    "Version": "2012-10-17",
    "Id": "ecr-key-policy",
    "Statement": [
        {
            "Sid": "EnableIAMUserPermissions",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
            },
            "Action": "kms:*",
            "Resource": "*"
        }
    ]
}
```

The IAM user, IAM role, or AWS account creating your repositories must have the `kms:CreateGrant`, `kms:RetireGrant`, and `kms:DescribeKey` permission in addition to the necessary Amazon ECR permissions.

Note

The `kms:RetireGrant` permission must be added to the IAM policy of the user or role creating the repository. The `kms:CreateGrant` and `kms:DescribeKey` permissions can be added to either the key policy for the KMS key or the IAM policy of user or role creating the repository. For more information on how AWS KMS permissions work, see AWS KMS API permissions: Actions and resources reference in the *AWS Key Management Service Developer Guide*.

The following example IAM policy can be added as an inline policy to a user to ensure they have the minimum permissions needed to create a repository with encryption enabled and delete the repository when they are finished with it. The AWS KMS key used to encrypt the repository can be specified using the resource parameter.

JSON

```
{
    "Version": "2012-10-17",
    "Id": "ecr-kms-permissions",
    "Statement": [
        {
            "Sid":
 "AllowAccessToCreateAndRetireTheGrantsAssociatedWithTheKeyAsWellAsDescribeTheKey",
            "Effect": "Allow",
            "Action": [
                "kms:CreateGrant",
                "kms:RetireGrant",
                "kms:DescribeKey"
            ],
            "Resource": "arn:aws:kms:us-
west-2:111122223333:key/b8d9ae76-080c-4043-92EXAMPLE"
        }
    ]
}
```

Allow a user to list KMS keys in the console when creating a repository

When using the Amazon ECR console to create a repository, you can grant permissions to enable a user to list the customer managed KMS keys in the Region when enabling encryption for the repository. The following IAM policy example shows the permissions needed to list your KMS keys and aliases when using the console.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": [
      "kms:ListKeys",
      "kms:ListAliases",
      "kms:DescribeKey"
    ],
    "Resource": "*"
  }
}
```

Monitoring Amazon ECR interaction with AWS KMS

You can use AWS CloudTrail to track the requests that Amazon ECR sends to AWS KMS on your behalf. The log entries in the CloudTrail log contain an encryption context key to make them more easily identifiable.

Amazon ECR encryption context

An *encryption context* is a set of key–value pairs that contains arbitrary nonsecret data. When you include an encryption context in a request to encrypt data, AWS KMS cryptographically binds the encryption context to the encrypted data. To decrypt the data, you must pass in the same encryption context.

In its GenerateDataKey and Decrypt requests to AWS KMS, Amazon ECR uses an encryption context with two name–value pairs that identify the repository and Amazon S3 bucket being used. This is shown in the following example. The names do not vary, but combined encryption context values will be different for each value.

```
"encryptionContext": {
    "aws:s3:arn": "arn:aws:s3:::us-west-2-starport-manifest-bucket/EXAMPLE1-90ab-cdef-fedc-ba987BUCKET1/
sha256:a7766145a775d39e53a713c75b6fd6d318740e70327aaa3ed5d09e0ef33fc3df",
    "aws:ecr:arn": "arn:aws:ecr:us-west-2:111122223333:repository/repository-name"
}
```

You can use the encryption context to identify these cryptographic operation in audit records and logs, such as AWS CloudTrail and Amazon CloudWatch Logs, and as a condition for authorization in policies and grants.

The Amazon ECR encryption context consists of two name-value pairs.

- **aws:s3:arn** – The first name–value pair identifies the bucket. The key is `aws:s3:arn`. The value is the Amazon Resource Name (ARN) of the Amazon S3 bucket.

  ```
  "aws:s3:arn": "ARN of an Amazon S3 bucket"
  ```

  For example, if the ARN of the bucket is `arn:aws:s3:::us-west-2-starport-manifest-bucket/EXAMPLE1-90ab-cdef-fedc-ba987BUCKET1/sha256:a7766145a775d39e53a713c75b6fd6d318740e70327aaa3ed5d09e0ef33fc3df`, the encryption context would include the following pair.

  ```
  "arn:aws:s3:::us-west-2-starport-manifest-bucket/EXAMPLE1-90ab-cdef-fedc-ba987BUCKET1/sha256:a7766145a775d39e53a713c75b6fd6d318740e70327aaa3ed5d09e0ef33fc3df"
  ```

- **aws:ecr:arn** – The second name–value pair identifies the Amazon Resource Name (ARN) of the repository. The key is `aws:ecr:arn`. The value is the ARN of the repository.

  ```
  "aws:ecr:arn": "ARN of an Amazon ECR repository"
  ```

  For example, if the ARN of the repository is `arn:aws:ecr:us-west-2:111122223333:repository/repository-name`, the encryption context would include the following pair.

  ```
  "aws:ecr:arn": "arn:aws:ecr:us-west-2:111122223333:repository/repository-name"
  ```

Troubleshooting

When deleting an Amazon ECR repository with the console, if the repository is successfully deleted but Amazon ECR is unable to retire the grants added to your KMS key for your repository, you will receive the following error.

```
The repository [{repository-name}] has been deleted successfully but the grants created
 by the kmsKey [{kms_key}] failed to be retired
```

When this occurs, you can retire the AWS KMS grants for the repository yourself.

To retire AWS KMS grants for a repository manually

1. List the grants for the AWS KMS key used for the repository. The `key-id` value is included in the error you receive from the console. You can also use the `list-keys` command to list both the AWS managed keys and customer managed KMS keys in a specific Region in your account.

   ```
   aws kms list-grants \
        --key-id b8d9ae76-080c-4043-9237-c815bfc21dfc
        --region us-west-2
   ```

   The output include an `EncryptionContextSubset` with the Amazon Resource Name (ARN) of your repository. This can be used to determine which grant added to the key is the one you want to retire. The `GrantId` value will be used when retiring the grant in the next step.

2. Retire each grant for the AWS KMS key added for the repository. Replace the value for *`GrantId`* with the ID of the grant from the output of the previous step.

   ```
   aws kms retire-grant \
        --key-id b8d9ae76-080c-4043-9237-c815bfc21dfc \
        --grant-id GrantId \
        --region us-west-2
   ```

Compliance validation for Amazon Elastic Container Registry

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security Compliance & Governance – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- HIPAA Eligible Services Reference – Lists HIPAA eligible services. Not all AWS services are HIPAA eligible.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.
- Amazon GuardDuty – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Infrastructure Security in Amazon Elastic Container Registry

As a managed service, Amazon Elastic Container Registry is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see

AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access Amazon ECR through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

You can call these API operations from any network location, but Amazon ECR does support resource-based access policies, which can include restrictions based on the source IP address. You can also use Amazon ECR policies to control access from specific Amazon Virtual Private Cloud (Amazon VPC) endpoints or specific VPCs. Effectively, this isolates network access to a given Amazon ECR resource from only the specific VPC within the AWS network. For more information, see Amazon ECR interface VPC endpoints (AWS PrivateLink).

Amazon ECR interface VPC endpoints (AWS PrivateLink)

You can improve the security posture of your VPC by configuring Amazon ECR to use an interface VPC endpoint. VPC endpoints are powered by AWS PrivateLink, a technology that enables you to privately access Amazon ECR APIs through private IP addresses. AWS PrivateLink restricts all network traffic between your VPC and Amazon ECR to the Amazon network. You don't need an internet gateway, a NAT device, or a virtual private gateway.

For more information about AWS PrivateLink and VPC endpoints, see VPC Endpoints in the *Amazon VPC User Guide*.

Considerations for Amazon ECR VPC endpoints

Before you configure VPC endpoints for Amazon ECR, be aware of the following considerations.

- To allow your Amazon ECS tasks hosted on Amazon EC2 instances to pull private images from Amazon ECR, create the interface VPC endpoints for Amazon ECS. For more information, see

Interface VPC Endpoints (AWS PrivateLink) in the *Amazon Elastic Container Service Developer Guide*.

- Amazon ECS tasks hosted on Fargate that pull container images from Amazon ECR can restrict access to the specific VPC their tasks use and to the VPC endpoint the service uses by adding condition keys to the task execution IAM role for the task. For more information, see Optional IAM Permissions for Fargate Tasks Pulling Amazon ECR Images over Interface Endpoints in the *Amazon Elastic Container Service Developer Guide*.
- The security group attached to the VPC endpoint must allow incoming connections on port 443 from the private subnet of the VPC.
- VPC endpoints currently don't support Amazon ECR Public repositories. Consider using a pull through cache rule to host the public image in a private repository in the same Region as the VPC endpoint. For more information, see Sync an upstream registry with an Amazon ECR private registry.
- VPC endpoints only support AWS provided DNS through Amazon Route 53. If you want to use your own DNS, you can use conditional DNS forwarding. For more information, see DHCP Options Sets in the *Amazon VPC User Guide*.
- If your containers have existing connections to Amazon S3, their connections might be briefly interrupted when you add the Amazon S3 gateway endpoint. If you want to avoid this interruption, create a new VPC that uses the Amazon S3 gateway endpoint and then migrate your Amazon ECS cluster and its containers into the new VPC.
- When an image is pulled using a pull through cache rule for the first time, if you've configured Amazon ECR to use an interface VPC endpoint using AWS PrivateLink then you need to create a public subnet in the same VPC, with a NAT gateway, and then route all outbound traffic to the internet from their private subnet to the NAT gateway in order for the pull to work. Subsequent image pulls don't require this. For more information, see Scenario: Access the internet from a private subnet in the *Amazon Virtual Private Cloud User Guide*.

Considerations for Windows images

Images based on the Windows operating system include artifacts that are restricted by license from being distributed. By default, when you push Windows images to an Amazon ECR repository, the layers that include these artifacts are not pushed as they are considered *foreign layers*. When the artifacts are provided by Microsoft, the foreign layers are retrieved from Microsoft Azure infrastructure. For this reason, to enable your containers to pull these foreign layers from Azure additional steps are needed beyond creating the VPC endpoints.

It is possible to override this behavior when pushing Windows images to Amazon ECR by using the `--allow-nondistributable-artifacts` flag in the Docker daemon. When enabled, this flag will push the licensed layers to Amazon ECR which enables these images to be pulled from Amazon ECR via the VPC endpoint without additional access to Azure being required.

> **Important**
>
> Using the `--allow-nondistributable-artifacts` flag does not preclude your obligation to comply with the terms of the Windows container base image license; you cannot post Windows content for public or third-party redistribution. Usage within your own environment is allowed.

To enable the use of this flag for your Docker installation, you must modify the Docker daemon configuration file which, depending on your Docker installation, can typically be configured in settings or preferences menu under the **Docker Engine** section or by editing the `C:\ProgramData\docker\config\daemon.json` file directly.

The following is an example of the required configuration. Replace the value with the repository URI you are pushing images to.

```
{
    "allow-nondistributable-artifacts": [
        "111122223333.dkr.ecr.us-west-2.amazonaws.com"
    ]
}
```

After modifying the Docker daemon configuration file, you must restart the Docker daemon before attempting to push your image. Confirm the push worked by verifying that the base layer was pushed to your repository.

> **Note**
>
> The base layers for Windows images are large. The layer size will result in a longer time to push and additional storage costs in Amazon ECR for these images. For these reasons, we recommend only using this option when it is strictly required to reduce build times and ongoing storage costs. For example, the `mcr.microsoft.com/windows/servercore` image is approximately 1.7 GiB in size when compressed in Amazon ECR.

Create the VPC endpoints for Amazon ECR

To create the VPC endpoints for the Amazon ECR service, use the Creating an Interface Endpoint procedure in the *Amazon VPC User Guide*.

Amazon ECS tasks hosted on Amazon EC2 instances require both Amazon ECR endpoints and the Amazon S3 gateway endpoint.

Amazon ECS tasks hosted on Fargate using platform version `1.4.0` or later require both Amazon ECR VPC endpoints and the Amazon S3 gateway endpoints.

Amazon ECS tasks hosted on Fargate that use platform version `1.3.0` or earlier only require the **com.amazonaws.*region*.ecr.dkr** Amazon ECR VPC endpoint and the Amazon S3 gateway endpoints.

> **Note**
>
> The order that the endpoints are created in doesn't matter.

com.amazonaws.*region*.ecr.dkr

This endpoint is used for the Docker Registry APIs. Docker client commands such as `push` and `pull` use this endpoint.

When you create this endpoint, you must enable a private DNS hostname. To do this, ensure that the **Enable Private DNS Name** option is selected in the Amazon VPC console when you create the VPC endpoint.

com.amazonaws.*region*.ecr.api

> **Note**
>
> The specified *region* represents the Region identifier for an AWS Region supported by Amazon ECR, such as `us-east-2` for the US East (Ohio) Region.

This endpoint is used for calls to the Amazon ECR API. API actions such as `DescribeImages` and `CreateRepository` go to this endpoint.

When this endpoint is created, you have the option to enable a private DNS hostname. Enable this setting by selecting **Enable Private DNS Name** in the VPC console when you create the VPC endpoint. If you enable a private DNS hostname for the VPC endpoint, update your SDK or AWS CLI to the latest version so that specifying an endpoint URL when using the SDK or AWS CLI isn't necessary.

If you enable a private DNS hostname and are using an SDK or AWS CLI version released before January 24, 2019, you must use the `--endpoint-url` parameter to specify the interface endpoints. The following example shows the format for the endpoint URL.

```
aws ecr create-repository --repository-name name --endpoint-url https://
api.ecr.region.amazonaws.com
```

If you don't enable a private DNS hostname for the VPC endpoint, you must use the `--endpoint-url` parameter specifying the VPC endpoint ID for the interface endpoint. The following example shows the format for the endpoint URL.

```
aws ecr create-repository --repository-name name --endpoint-url
 https://VPC_endpoint_ID.api.ecr.region.vpce.amazonaws.com
```

Create the Amazon S3 gateway endpoint

For your Amazon ECS tasks to pull private images from Amazon ECR, you must create a gateway endpoint for Amazon S3. The gateway endpoint is required because Amazon ECR uses Amazon S3 to store your image layers. When your containers download images from Amazon ECR, they must access Amazon ECR to get the image manifest and then Amazon S3 to download the actual image layers. The following is the Amazon Resource Name (ARN) of the Amazon S3 bucket containing the layers for each Docker image.

```
arn:aws:s3:::prod-region-starport-layer-bucket/*
```

Use the Creating a gateway endpoint procedure in the *Amazon VPC User Guide* to create the following Amazon S3 gateway endpoint for Amazon ECR. When creating the endpoint, be sure to select the route tables for your VPC.

com.amazonaws.*region*.s3

The Amazon S3 gateway endpoint uses an IAM policy document to limit access to the service. The **Full Access** policy can be used because any restrictions that you have put in your task IAM roles or other IAM user policies still apply on top of this policy. If you want to limit Amazon S3 bucket access to the minimum required permissions for using Amazon ECR, see Minimum Amazon S3 Bucket Permissions for Amazon ECR.

Minimum Amazon S3 Bucket Permissions for Amazon ECR

The Amazon S3 gateway endpoint uses an IAM policy document to limit access to the service. To allow only the minimum Amazon S3 bucket permissions for Amazon ECR, restrict access to the Amazon S3 bucket that Amazon ECR uses when you create the IAM policy document for the endpoint.

The following table describes the Amazon S3 bucket policy permissions needed by Amazon ECR.

Permission	Description
`arn:aws:s3:::prod-` *`region`*`-starport-layer-bucket/*`	Provides access to the Amazon S3 bucket containing the layers for each Docker image. Represents the Region identifier for an AWS Region supported by Amazon ECR, such as `us-east-2` for the US East (Ohio) Region.

Example

The following example illustrates how to provide access to the Amazon S3 buckets required for Amazon ECR operations.

```
{
  "Statement": [
    {
      "Sid": "Access-to-specific-bucket-only",
      "Principal": "*",
      "Action": [
        "s3:GetObject"
      ],
```

```
      "Effect": "Allow",
      "Resource": ["arn:aws:s3:::prod-region-starport-layer-bucket/*"]
    }
  ]
}
```

Create the CloudWatch Logs endpoint

Amazon ECS tasks using the Fargate launch type that use a VPC without an internet gateway that also use the `awslogs` log driver to send log information to CloudWatch Logs require that you create the **com.amazonaws.*region*.logs** interface VPC endpoint for CloudWatch Logs. For more information, see Using CloudWatch Logs with interface VPC endpoints in the *Amazon CloudWatch Logs User Guide*.

Create an endpoint policy for your Amazon ECR VPC endpoints

A VPC endpoint policy is an IAM resource policy that you attach to an endpoint when you create or modify the endpoint. If you don't attach a policy when you create an endpoint, AWS attaches a default policy for you that allows full access to the service. An endpoint policy doesn't override or replace user policies or service-specific policies. It's a separate policy for controlling access from the endpoint to the specified service. Endpoint policies must be written in JSON format. For more information, see Controlling Access to Services with VPC Endpoints in the *Amazon VPC User Guide*.

We recommend creating a single IAM resource policy and attaching it to both of the Amazon ECR VPC endpoints.

The following is an example of an endpoint policy for Amazon ECR. This policy enables a specific IAM role to pull images from Amazon ECR.

```
{
 "Statement": [{
  "Sid": "AllowPull",
  "Principal": {
   "AWS": "arn:aws:iam::1234567890:role/role_name"
  },
  "Action": [
   "ecr:BatchGetImage",
   "ecr:GetDownloadUrlForLayer",
                  "ecr:GetAuthorizationToken"
  ],
  "Effect": "Allow",
```

```
  "Resource": "*"
 }]
}
```

The following endpoint policy example prevents a specified repository from being deleted.

```
{
 "Statement": [{
   "Sid": "AllowAll",
   "Principal": "*",
   "Action": "*",
   "Effect": "Allow",
   "Resource": "*"
  },
  {
   "Sid": "PreventDelete",
   "Principal": "*",
   "Action": "ecr:DeleteRepository",
   "Effect": "Deny",
   "Resource": "arn:aws:ecr:region:1234567890:repository/repository_name"
  }
 ]
}
```

The following endpoint policy example combines the two previous examples into a single policy.

```
{
 "Statement": [{
   "Sid": "AllowAll",
   "Effect": "Allow",
   "Principal": "*",
   "Action": "*",
   "Resource": "*"
  },
  {
   "Sid": "PreventDelete",
   "Effect": "Deny",
   "Principal": "*",
   "Action": "ecr:DeleteRepository",
   "Resource": "arn:aws:ecr:region:1234567890:repository/repository_name"
  },
  {
   "Sid": "AllowPull",
```

```
    "Effect": "Allow",
    "Principal": {
     "AWS": "arn:aws:iam::1234567890:role/role_name"
    },
    "Action": [
     "ecr:BatchGetImage",
     "ecr:GetDownloadUrlForLayer",
                          "ecr:GetAuthorizationToken"
    ],
    "Resource": "*"
   }
  ]
}
```

To modify the VPC endpoint policy for Amazon ECR

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Endpoints**.
3. If you have not already created the VPC endpoints for Amazon ECR, see Create the VPC endpoints for Amazon ECR.
4. Select the Amazon ECR VPC endpoint to add a policy to, and choose the **Policy** tab in the lower half of the screen.
5. Choose **Edit Policy** and make the changes to the policy.
6. Choose **Save** to save the policy.

Shared subnets

You can't create, describe, modify, or delete VPC endpoints in subnets that are shared with you. However, you can use the VPC endpoints in subnets that are shared with you.

Cross-service confused deputy prevention

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect

your data for all services with service principals that have been given access to resources in your account.

We recommend using the aws:SourceArn or aws:SourceAccount global condition context keys in resource policies to limit the permissions that Amazon ECR gives another service to the resource. Use aws:SourceArn if you want only one resource to be associated with the cross-service access. Use aws:SourceAccount if you want to allow any resource in that account to be associated with the cross-service use.

The most effective way to protect against the confused deputy problem is to use the aws:SourceArn global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the aws:SourceArn global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, arn:aws:*servicename*:*region*:*123456789012*:*.

If the aws:SourceArn value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both global condition context keys to limit permissions.

The value of aws:SourceArn must be ResourceDescription.

The following example shows how you can use the aws:SourceArn and aws:SourceAccount global condition context keys in an Amazon ECR repository policy to allow AWS CodeBuild access to the Amazon ECR API actions necessary for integration with that service while also preventing the confused deputy problem.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"CodeBuildAccess",
         "Effect":"Allow",
         "Principal":{
            "Service":"codebuild.amazonaws.com"
         },
         "Action":[
            "ecr:BatchGetImage",
            "ecr:GetDownloadUrlForLayer"
         ],
```

```
         "Resource": "*",
         "Condition":{
            "ArnLike":{
               "aws:SourceArn":"arn:aws:codebuild:us-
east-1:123456789012:project/project-name"
            },
            "StringEquals":{
               "aws:SourceAccount":"123456789012"
            }
         }
      }
   ]
}
```

Amazon ECR monitoring

You can monitor your Amazon ECR API usage with Amazon CloudWatch, which collects and processes raw data from Amazon ECR into readable, near real-time metrics. These statistics are recorded for a period of two weeks so that you can access historical information and gain perspective on your API usage. Amazon ECR metric data is automatically sent to CloudWatch in one-minute periods. For more information about CloudWatch, see the Amazon CloudWatch User Guide.

Amazon ECR provides metrics based on your API usage for authorization, image push, and image pull actions.

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon ECR and your AWS solutions. We recommend that you collect monitoring data from the resources that make up your AWS solution so that you can more easily debug a multi-point failure if one occurs. Before you start monitoring Amazon ECR, however, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

The next step is to establish a baseline for normal Amazon ECR performance in your environment by measuring performance at various times and under different load conditions. As you monitor Amazon ECR, store historical monitoring data so that you can compare it with new performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

Topics

- Visualizing your service quotas and setting alarms
- Amazon ECR usage metrics
- Amazon ECR usage reports

- Amazon ECR repository metrics
- Amazon ECR events and EventBridge
- Logging Amazon ECR actions with AWS CloudTrail

Visualizing your service quotas and setting alarms

You can use the CloudWatch console to visualize your service quotas and see how your current usage compares to service quotas. You can also set alarms so that you will be notified when you approach a quota.

To visualize a service quota and optionally set an alarm

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Metrics**.
3. On the **All metrics** tab, choose **Usage**, then choose **By AWS Resource**.

   The list of service quota usage metrics appears.
4. Select the check box next to one of the metrics.

   The graph displays your current usage of that AWS resource.
5. To add your service quota to the graph, do the following:

   a. Choose the **Graphed metrics** tab.

   b. Choose **Math expression**, **Start with an empty expression**. Then in the new row, under **Details**, enter **`SERVICE_QUOTA(m1)`**.

      A new line is added to the graph, displaying the service quota for the resource represented in the metric.
6. To see your current usage as a percentage of the quota, add a new expression or change the current **SERVICE_QUOTA** expression. For the new expression, use **`m1/60/SERVICE_QUOTA(m1)*100`**.
7. (Optional) To set an alarm that notifies you if you approach the service quota, do the following:

   a. On the **`m1/60/SERVICE_QUOTA(m1)*100`** row, under **Actions**, choose the alarm icon. It looks like a bell.

The alarm creation page appears.

b. Under **Conditions**, ensure that **Threshold type** is **Static** and **Whenever Expression1 is** is set to **Greater**. Under **than**, enter **80**. This creates an alarm that goes into ALARM state when your usage exceeds 80 percent of the quota.

c. Choose **Next**.

d. On the next page, select an Amazon SNS topic or create a new one. This topic is notified when the alarm goes to ALARM state. Then choose **Next**.

e. On the next page, enter a name and description for the alarm, and then choose **Next**.

f. Choose **Create alarm**.

Amazon ECR usage metrics

You can use CloudWatch usage metrics to provide visibility into your account's usage of resources. Use these metrics to visualize your current service usage on CloudWatch graphs and dashboards.

Amazon ECR usage metrics correspond to AWS service quotas. You can configure alarms that alert you when your usage approaches a service quota. For more information about Amazon ECR service quotas, see Amazon ECR service quotas.

Amazon ECR publishes the following metrics in the `AWS/Usage` namespace.

Metric	Description
`CallCount`	The number of API action calls from your account. The resources are defined by the dimensions associated with the metric. The most useful statistic for this metric is `SUM`, which represents the sum of the values from all contributors during the period defined.

The following dimensions are used to refine the usage metrics that are published by Amazon ECR.

Dimension	Description
`Service`	The name of the AWS service containing the resource. For Amazon ECR usage metrics, the value for this dimension is `ECR`.
`Type`	The type of entity that is being reported. Currently, the only valid value for Amazon ECR usage metrics is `API`.
`Resource`	The type of resource that is running. Currently, Amazon ECR returns information on your API usage for the following API actions. • `GetAuthorizationToken` • `BatchCheckLayerAvailability` • `InitiateLayerUpload` • `UploadLayerPart` • `CompleteLayerUpload` • `PutImage` • `BatchGetImage` • `GetDownloadUrlForLayer`
`Class`	The class of resource being tracked. Currently, Amazon ECR does not use the class dimension.

Amazon ECR usage reports

AWS provides a free reporting tool called Cost Explorer that enables you to analyze the cost and usage of your Amazon ECR resources.

Use Cost Explorer to view charts of your usage and costs. You can view data from the previous 13 months and forecast how much you are likely to spend for the next three months. You can use Cost Explorer to see patterns in how much you spend on AWS resources over time, identify areas that need further inquiry, and see trends that you can use to understand your costs. You also can specify time ranges for the data and view time data by day or by month.

The metering data in your Cost and Usage Reports shows usage across all of your Amazon ECR repositories. For more information, see Tagging your resources for billing.

For more information about creating an AWS Cost and Usage Report, see AWS Cost and Usage Report in the *AWS Billing User Guide*.

Amazon ECR repository metrics

Amazon ECR sends repository pull count metrics to Amazon CloudWatch. Amazon ECR metric data is automatically sent to CloudWatch in 1-minute periods. For more information about CloudWatch, see the Amazon CloudWatch User Guide.

Topics

- Enabling CloudWatch metrics
- Available metrics and dimensions
- Viewing Amazon ECR metrics using the CloudWatch console

Enabling CloudWatch metrics

Amazon ECR sends repository metrics automatically for all repositories. There is no need to take any manual steps.

Available metrics and dimensions

The following sections list the metrics and dimensions that Amazon ECR sends to Amazon CloudWatch.

Amazon ECR metrics

Amazon ECR provides metrics for you to monitor your repositories. You can measure the pull count.

The `AWS/ECR` namespace includes the following metrics.

`RepositoryPullCount`

The total number of pulls for the images in the repository.

Valid dimensions: `RepositoryName`.

Valid statistics: Average, Minimum, Maximum, Sum, Sample Count. The most useful statistic is Sum.

Unit: Integer.

Dimensions for Amazon ECR metrics

Amazon ECR metrics use the `AWS/ECR` namespace and provide metrics for the following dimensions.

`RepositoryName`

This dimension filters the data that you request for all container images in a specified repository.

Viewing Amazon ECR metrics using the CloudWatch console

You can view Amazon ECR repository metrics on the CloudWatch console. The CloudWatch console provides a fine-grained and customizable display of your resources. For more information, see the Amazon CloudWatch User Guide.

Amazon ECR events and EventBridge

Amazon EventBridge enables you to automate your AWS services and to respond automatically to system events such as application availability issues or resource changes. Events from AWS services are delivered to EventBridge in near real time. You can write simple rules to indicate which events are of interest to you and include automated actions to take when an event matches a rule. The actions that can be automatically triggered include the following:

- Adding events to log groups in CloudWatch Logs
- Invoking an AWS Lambda function
- Invoking Amazon EC2 Run Command
- Relaying the event to Amazon Kinesis Data Streams
- Activating an AWS Step Functions state machine
- Notifying an Amazon SNS topic or an Amazon SQS queue

For more information, see Getting Started with Amazon EventBridge in the *Amazon EventBridge User Guide*.

Sample events from Amazon ECR

The following are example events from Amazon ECR. Events are emitted on a best effort basis.

Event for a completed image push

The following event is sent when each image push is completed. For more information, see Pushing a Docker image to an Amazon ECR private repository.

```
{
    "version": "0",
    "id": "13cde686-328b-6117-af20-0e5566167482",
    "detail-type": "ECR Image Action",
    "source": "aws.ecr",
    "account": "123456789012",
    "time": "2019-11-16T01:54:34Z",
    "region": "us-west-2",
    "resources": [],
    "detail": {
        "result": "SUCCESS",
        "repository-name": "my-repository-name",
        "image-digest":
 "sha256:7f5b2640fe6fb4f46592dfd3410c4a79dac4f89e4782432e0378abcd1234",
        "action-type": "PUSH",
        "image-tag": "latest"
    }
}
```

Event for a pull through cache action

The following event is sent when a pull through cache action is attempted. For more information, see Sync an upstream registry with an Amazon ECR private registry.

```
{
    "version": "0",
    "id": "85fc3613-e913-7fc4-a80c-a3753e4aa9ae",
    "detail-type": "ECR Pull Through Cache Action",
    "source": "aws.ecr",
    "account": "123456789012",
    "time": "2023-02-29T02:36:48Z",
```

```
    "region": "us-west-2",
    "resources": [
        "arn:aws:ecr:us-west-2:123456789012:repository/docker-hub/alpine"
    ],
    "detail": {
        "rule-version": "1",
        "sync-status": "SUCCESS",
        "ecr-repository-prefix": "docker-hub",
        "repository-name": "docker-hub/alpine",
        "upstream-registry-url": "public.ecr.aws",
        "image-tag": "3.17.2",
        "image-digest":
 "sha256:4aa08ef415aecc80814cb42fa41b658480779d80c77ab15EXAMPLE",
    }
}
```

Event for a completed image scan (basic scanning)

When basic scanning is enabled for your registry, the following event is sent when each image scan is completed. The `finding-severity-counts` parameter will only return a value for a severity level if one exists. For example, if the image contains no findings at `CRITICAL` level, then no critical count is returned. For more information, see Scan images for OS vulnerabilities in Amazon ECR.

> **Note**
>
> For details about events that Amazon Inspector emits when enhanced scanning is enabled, see EventBridge events sent for enhanced scanning in Amazon ECR.

```
{
    "version": "0",
    "id": "85fc3613-e913-7fc4-a80c-a3753e4aa9ae",
    "detail-type": "ECR Image Scan",
    "source": "aws.ecr",
    "account": "123456789012",
    "time": "2019-10-29T02:36:48Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:ecr:us-east-1:123456789012:repository/my-repository-name"
    ],
    "detail": {
```

```
        "scan-status": "COMPLETE",
        "repository-name": "my-repository-name",
        "finding-severity-counts": {
        "CRITICAL": 10,
        "MEDIUM": 9
      },
        "image-digest":
 "sha256:7f5b2640fe6fb4f46592dfd3410c4a79dac4f89e4782432e0378abcd1234",
        "image-tags": []
    }
}
```

Event for a change notification on a resource with enhanced scanning enabled (enhanced scanning)

When enhanced scanning is enabled for your registry, the following event is sent by Amazon ECR when there is a change with a resource that has enhanced scanning enabled. This includes new repositories being created, the scan frequency for a repository being changed, or when images are created or deleted in repositories with enhanced scanning enabled. For more information, see Scan images for software vulnerabilities in Amazon ECR.

```
{
 "version": "0",
 "id": "0c18352a-a4d4-6853-ef53-0ab8638973bf",
 "detail-type": "ECR Scan Resource Change",
 "source": "aws.ecr",
 "account": "123456789012",
 "time": "2021-10-14T20:53:46Z",
 "region": "us-east-1",
 "resources": [],
 "detail": {
  "action-type": "SCAN_FREQUENCY_CHANGE",
  "repositories": [{
    "repository-name": "repository-1",
    "repository-arn": "arn:aws:ecr:us-east-1:123456789012:repository/repository-1",
    "scan-frequency": "SCAN_ON_PUSH",
    "previous-scan-frequency": "MANUAL"
   },
   {
    "repository-name": "repository-2",
    "repository-arn": "arn:aws:ecr:us-east-1:123456789012:repository/repository-2",
    "scan-frequency": "CONTINUOUS_SCAN",
    "previous-scan-frequency": "SCAN_ON_PUSH"
```

```
    },
    {
     "repository-name": "repository-3",
     "repository-arn": "arn:aws:ecr:us-east-1:123456789012:repository/repository-3",
     "scan-frequency": "CONTINUOUS_SCAN",
     "previous-scan-frequency": "SCAN_ON_PUSH"
    }
   ],
   "resource-type": "REPOSITORY",
   "scan-type": "ENHANCED"
 }
}
```

Event for an image deletion

The following event is sent when an image is deleted. For more information, see Deleting an image in Amazon ECR.

```
{
    "version": "0",
    "id": "dd3b46cb-2c74-f49e-393b-28286b67279d",
    "detail-type": "ECR Image Action",
    "source": "aws.ecr",
    "account": "123456789012",
    "time": "2019-11-16T02:01:05Z",
    "region": "us-west-2",
    "resources": [],
    "detail": {
        "result": "SUCCESS",
        "repository-name": "my-repository-name",
        "image-digest":
 "sha256:7f5b2640fe6fb4f46592dfd3410c4a79dac4f89e4782432e0378abcd1234",
        "action-type": "DELETE",
        "image-tag": "latest"
    }
}
```

Event for a completed image replication

The following event is sent when each image replication is completed. For more information, see Private image replication in Amazon ECR.

```
{
  "version": "0",
  "id": "c8b133b1-6029-ee73-e2a1-4f466b8ba999",
  "detail-type": "ECR Replication Action",
  "source": "aws.ecr",
  "account": "123456789012",
  "time": "2024-05-08T20:44:54Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:ecr:us-east-1:123456789012:repository/docker-hub/alpine"
  ],
  "detail": {
    "result": "SUCCESS",
    "repository-name": "docker-hub/alpine",
    "image-digest":
 "sha256:7f5b2640fe6fb4f46592dfd3410c4a79dac4f89e4782432e0378abcd1234",
    "source-account": "123456789012",
    "action-type": "REPLICATE",
    "source-region": "us-west-2",
    "image-tag": "3.17.2"
  }
}
```


Logging Amazon ECR actions with AWS CloudTrail

Amazon ECR is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, a role, or an AWS service in Amazon ECR. CloudTrail captures the following Amazon ECR actions as events:

- All API calls, including calls from the Amazon ECR console
- All actions taken due to the encryption settings on your repositories
- All actions taken due to lifecycle policy rules, including both successful and unsuccessful actions

> **Important**
>
> Due to the size limitations of individual CloudTrail events, for lifecycle policy actions where 10 or more images are expired Amazon ECR sends multiple events to CloudTrail. Additionally, Amazon ECR includes a maximum of 100 tags per image.

When a trail is created, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Amazon ECR. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using this information, you can determine the request that was made to Amazon ECR, the originating IP address, who made the request, when it was made, and additional details.

For more information, see the AWS CloudTrail User Guide.

Amazon ECR information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Amazon ECR, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for Amazon ECR, create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. When you create a trail in the console, you can apply the trail to a single Region or to all Regions. The trail logs events in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to analyze and act upon the event data collected in CloudTrail logs. For more information, see:

- Creating a trail for your AWS account
- AWS service integrations with CloudTrail logs
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple Regions and Receiving CloudTrail log files from multiple accounts

All Amazon ECR API actions are logged by CloudTrail and are documented in the Amazon Elastic Container Registry API Reference. When you perform common tasks, sections are generated in the CloudTrail log files for each API action that is part of that task. For example, when you create a repository, `GetAuthorizationToken`, `CreateRepository` and `SetRepositoryPolicy` sections are generated in the CloudTrail log files. When you push an image to a repository, `InitiateLayerUpload`, `UploadLayerPart`, `CompleteLayerUpload`, and `PutImage` sections are generated. When you pull an image, `GetDownloadUrlForLayer` and `BatchGetImage` sections are generated. When OCI clients that support the OCI 1.1 specification fetch the list of referrers, or reference artifacts, for an image using the Referrers API, a `ListImageReferrers`

CloudTrail event is emitted. For examples of these common tasks, see CloudTrail log entry examples.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or user credentials
- Whether the request was made with temporary security credentials for a role or federated user
- Whether the request was made by another AWS service

For more information, see the CloudTrail `userIdentity` Element.

Understanding Amazon ECR log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and other information. CloudTrail log files are not an ordered stack trace of the public API calls, so they do not appear in any specific order.

CloudTrail log entry examples

The following are CloudTrail log entry examples for a few common Amazon ECR tasks.

These examples have been formatted for improved readability. In a CloudTrail log file, all entries and events are concatenated into a single line. In addition, this example has been limited to a single Amazon ECR entry. In a real CloudTrail log file, you see entries and events from multiple AWS services.

> **Important**
>
> The **sourceIPAddress** is the IP address that the request was made from. For actions that originate from the service console, the address reported is for your underlying resource, not the console web server. For services in AWS, only the DNS name is displayed. We still evaluate the auth with the client source IP even if it's redacted to AWS service DNS name.

Topics

- Example: Create repository action

- Example: AWS KMSCreateGrant API action when creating an Amazon ECR repository
- Example: Image push action
- Example: Image pull action
- Example: Image lifecycle policy action
- Example: Image referrers action

Example: Create repository action

The following example shows a CloudTrail log entry that demonstrates the `CreateRepository` action.

```
{
    "eventVersion": "1.04",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE:account_name",
        "arn": "arn:aws:sts::123456789012:user/Mary_Major",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2018-07-11T21:54:07Z"
            },
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AIDACKCEVSQ6C2EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/Admin",
                "accountId": "123456789012",
                "userName": "Admin"
            }
        }
    },
    "eventTime": "2018-07-11T22:17:43Z",
    "eventSource": "ecr.amazonaws.com",
    "eventName": "CreateRepository",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "203.0.113.12",
    "userAgent": "console.amazonaws.com",
    "requestParameters": {
        "repositoryName": "testrepo"
```

```
    },
    "responseElements": {
        "repository": {
            "repositoryArn": "arn:aws:ecr:us-east-2:123456789012:repository/testrepo",
            "repositoryName": "testrepo",
            "repositoryUri": "123456789012.dkr.ecr.us-east-2.amazonaws.com/testrepo",
            "createdAt": "Jul 11, 2018 10:17:44 PM",
            "registryId": "123456789012"
        }
    },
    "requestID": "cb8c167e-EXAMPLE",
    "eventID": "e3c6f4ce-EXAMPLE",
    "resources": [
        {
            "ARN": "arn:aws:ecr:us-east-2:123456789012:repository/testrepo",
            "accountId": "123456789012"
        }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

Example: AWS KMS`CreateGrant` API action when creating an Amazon ECR repository

The following example shows a CloudTrail log entry that demonstrates the AWS KMS `CreateGrant` action when creating an Amazon ECR repository with KMS encryption enabled. For each repository that is created with KMS encryption is enabled, you should see two `CreateGrant` log entries in CloudTrail.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDAIEP6W46J43IG7LXAQ",
        "arn": "arn:aws:iam::123456789012:user/Mary_Major",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "Mary_Major",
        "sessionContext": {
            "sessionIssuer": {

            },
            "webIdFederationData": {
```

```
            },
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2020-06-10T19:22:10Z"
            }
        },
        "invokedBy": "AWS Internal"
    },
    "eventTime": "2020-06-10T19:22:10Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "CreateGrant",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.12",
    "userAgent": "console.amazonaws.com",
    "requestParameters": {
        "keyId": "4b55e5bf-39c8-41ad-b589-18464af7758a",
        "granteePrincipal": "ecr.us-west-2.amazonaws.com",
        "operations": [
            "GenerateDataKey",
            "Decrypt"
        ],
        "retiringPrincipal": "ecr.us-west-2.amazonaws.com",
        "constraints": {
            "encryptionContextSubset": {
                "aws:ecr:arn": "arn:aws:ecr:us-west-2:123456789012:repository/testrepo"
            }
        }
    },
    "responseElements": {
        "grantId": "3636af9adfee1accb67b83941087dcd45e7fadc4e74ff0103bb338422b5055f3"
    },
    "requestID": "047b7dea-b56b-4013-87e9-a089f0f6602b",
    "eventID": "af4c9573-c56a-4886-baca-a77526544469",
    "readOnly": false,
    "resources": [
        {
            "accountId": "123456789012",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-west-2:123456789012:key/4b55e5bf-39c8-41ad-
b589-18464af7758a"
        }
    ],
    "eventType": "AwsApiCall",
```

```
    "recipientAccountId": "123456789012"
}
```

Example: Image push action

The following example shows a CloudTrail log entry that demonstrates an image push which uses the `PutImage` action.

> **Note**
>
> When pushing an image, you will also see `InitiateLayerUpload`, `UploadLayerPart`, and `CompleteLayerUpload` references in the CloudTrail logs.

```
{
    "eventVersion": "1.04",
    "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDACKCEVSQ6C2EXAMPLE:account_name",
    "arn": "arn:aws:sts::123456789012:user/Mary_Major",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
  "userName": "Mary_Major",
  "sessionContext": {
   "attributes": {
    "mfaAuthenticated": "false",
    "creationDate": "2019-04-15T16:42:14Z"
   }
  }
 },
 "eventTime": "2019-04-15T16:45:00Z",
 "eventSource": "ecr.amazonaws.com",
 "eventName": "PutImage",
 "awsRegion": "us-east-2",
 "sourceIPAddress": "AWS Internal",
 "userAgent": "AWS Internal",
 "requestParameters": {
  "repositoryName": "testrepo",
  "imageTag": "latest",
  "registryId": "123456789012",
  "imageManifest": "{\n   \"schemaVersion\": 2,\n   \"mediaType\": \"application/
vnd.docker.distribution.manifest.v2+json\",\n   \"config\": {\n      \"mediaType\":
```

```
 \"application/vnd.docker.container.image.v1+json\",\n      \"size\": 5543,\n
 \"digest\": \"sha256:000b9b805af1cdb60628898c9f411996301a1c13afd3dbef1d8a16ac6dbf503a
\"\n   },\n   \"layers\": [\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 43252507,\n
 \"digest\": \"sha256:3b37166ec61459e76e33282dda08f2a9cd698ca7e3d6bc44e6a6e7580cdeff8e
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 846,\n         \"digest
\": \"sha256:504facff238fde83f1ca8f9f54520b4219c5b8f80be9616ddc52d31448a044bd
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 615,\n         \"digest
\": \"sha256:ebbcacd28e101968415b0c812b2d2dc60f969e36b0b08c073bf796e12b1bb449\"\n
      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 850,\n         \"digest
\": \"sha256:c7fb3351ecad291a88b92b600037e2435c84a347683d540042086fe72c902b8a
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 168,\n         \"digest\":
 \"sha256:2e3debadcbf7e542e2aefbce1b64a358b1931fb403b3e4aeca27cb4d809d56c2\"\n      },
\n      {\n         \"mediaType\": \"application/vnd.docker.image.rootfs.diff.tar.gzip
\",\n         \"size\": 37720774,\n         \"digest\":
 \"sha256:f8c9f51ad524d8ae9bf4db69cd3e720ba92373ec265f5c390ffb21bb0c277941\"\n
      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 30432107,\n
 \"digest\": \"sha256:813a50b13f61cf1f8d25f19fa96ad3aa5b552896c83e86ce413b48b091d7f01b
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 197,\n         \"digest
\": \"sha256:7ab043301a6187ea3293d80b30ba06c7bf1a0c3cd4c43d10353b31bc0cecfe7d
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 154,\n         \"digest
\": \"sha256:67012cca8f31dc3b8ee2305e7762fee20c250513effdedb38a1c37784a5a2e71\"\n
      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 176,\n         \"digest
\": \"sha256:3bc892145603fffc9b1c97c94e2985b4cb19ca508750b15845a5d97becbd1a0e
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 183,\n         \"digest
\": \"sha256:6f1c79518f18251d35977e7e46bfa6c6b9cf50df2a79d4194941d95c54258d18\"\n
      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 212,\n         \"digest
\": \"sha256:b7bcfbc2e2888afebede4dd1cd5eebf029bb6315feeaf0b56e425e11a50afe42\"\n
      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 212,\n         \"digest\":
 \"sha256:2b220f8b0f32b7c2ed8eaafe1c802633bbd94849b9ab73926f0ba46cdae91629\"\n      }\n
   ]\n}"
 },
 "responseElements": {
```

```
  "image": {
   "repositoryName": "testrepo",
   "imageManifest": "{\n   \"schemaVersion\": 2,\n   \"mediaType\": \"application/
vnd.docker.distribution.manifest.v2+json\",\n   \"config\": {\n      \"mediaType\":
 \"application/vnd.docker.container.image.v1+json\",\n      \"size\": 5543,\n
 \"digest\": \"sha256:000b9b805af1cdb60628898c9f411996301a1c13afd3dbef1d8a16ac6dbf503a
\"\n   },\n   \"layers\": [\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 43252507,\n
 \"digest\": \"sha256:3b37166ec61459e76e33282dda08f2a9cd698ca7e3d6bc44e6a6e7580cdeff8e
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 846,\n         \"digest
\": \"sha256:504facff238fde83f1ca8f9f54520b4219c5b8f80be9616ddc52d31448a044bd
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 615,\n         \"digest
\": \"sha256:ebbcacd28e101968415b0c812b2d2dc60f969e36b0b08c073bf796e12b1bb449\"\n
      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 850,\n         \"digest
\": \"sha256:c7fb3351ecad291a88b92b600037e2435c84a347683d540042086fe72c902b8a
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 168,\n         \"digest\":
 \"sha256:2e3debadcbf7e542e2aefbce1b64a358b1931fb403b3e4aeca27cb4d809d56c2\"\n      },
\n      {\n         \"mediaType\": \"application/vnd.docker.image.rootfs.diff.tar.gzip
\",\n         \"size\": 37720774,\n         \"digest\":
 \"sha256:f8c9f51ad524d8ae9bf4db69cd3e720ba92373ec265f5c390ffb21bb0c277941\"\n
      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 30432107,\n
 \"digest\": \"sha256:813a50b13f61cf1f8d25f19fa96ad3aa5b552896c83e86ce413b48b091d7f01b
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 197,\n         \"digest
\": \"sha256:7ab043301a6187ea3293d80b30ba06c7bf1a0c3cd4c43d10353b31bc0cecfe7d
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 154,\n         \"digest
\": \"sha256:67012cca8f31dc3b8ee2305e7762fee20c250513effdedb38a1c37784a5a2e71\"\n
      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 176,\n         \"digest
\": \"sha256:3bc892145603fffc9b1c97c94e2985b4cb19ca508750b15845a5d97becbd1a0e
\"\n      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 183,\n         \"digest
\": \"sha256:6f1c79518f18251d35977e7e46bfa6c6b9cf50df2a79d4194941d95c54258d18\"\n
      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 212,\n         \"digest
\": \"sha256:b7bcfbc2e2888afebede4dd1cd5eebf029bb6315feeaf0b56e425e11a50afe42\"\n
      },\n      {\n         \"mediaType\": \"application/
vnd.docker.image.rootfs.diff.tar.gzip\",\n         \"size\": 212,\n         \"digest\":
```

```
 \"sha256:2b220f8b0f32b7c2ed8eaafe1c802633bbd94849b9ab73926f0ba46cdae91629\"\n      }\n
   ]\n}",
   "registryId": "123456789012",
   "imageId": {
    "imageDigest":
 "sha256:98c8b060c21d9adbb6b8c41b916e95e6307102786973ab93a41e8b86d1fc6d3e",
    "imageTag": "latest"
   }
  }
 },
 "requestID": "cf044b7d-5f9d-11e9-9b2a-95983139cc57",
 "eventID": "2bfd4ee2-2178-4a82-a27d-b12939923f0f",
 "resources": [{
  "ARN": "arn:aws:ecr:us-east-2:123456789012:repository/testrepo",
  "accountId": "123456789012"
 }],
 "eventType": "AwsApiCall",
 "recipientAccountId": "123456789012"
}
```

Example: Image pull action

The following example shows a CloudTrail log entry that demonstrates an image pull which uses the `BatchGetImage` action.

> **Note**
>
> When pulling an image, if you don't already have the image locally, you will also see `GetDownloadUrlForLayer` references in the CloudTrail logs.

```
{
    "eventVersion": "1.04",
    "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDACKCEVSQ6C2EXAMPLE:account_name",
    "arn": "arn:aws:sts::123456789012:user/Mary_Major",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
  "userName": "Mary_Major",
  "sessionContext": {
   "attributes": {
```

```
    "mfaAuthenticated": "false",
    "creationDate": "2019-04-15T16:42:14Z"
   }
  }
 },
 "eventTime": "2019-04-15T17:23:20Z",
 "eventSource": "ecr.amazonaws.com",
 "eventName": "BatchGetImage",
 "awsRegion": "us-east-2",
 "sourceIPAddress": "ecr.amazonaws.com",
 "userAgent": "ecr.amazonaws.com",
 "requestParameters": {
  "imageIds": [{
   "imageTag": "latest"
  }],
  "acceptedMediaTypes": [
   "application/json",
   "application/vnd.oci.image.manifest.v1+json",
   "application/vnd.oci.image.index.v1+json",
   "application/vnd.docker.distribution.manifest.v2+json",
   "application/vnd.docker.distribution.manifest.list.v2+json",
   "application/vnd.docker.distribution.manifest.v1+prettyjws"
  ],
  "repositoryName": "testrepo",
  "registryId": "123456789012"
 },
 "responseElements": null,
 "requestID": "2a1b97ee-5fa3-11e9-a8cd-cd2391aeda93",
 "eventID": "c84f5880-c2f9-4585-9757-28fa5c1065df",
 "resources": [{
  "ARN": "arn:aws:ecr:us-east-2:123456789012:repository/testrepo",
  "accountId": "123456789012"
 }],
 "eventType": "AwsApiCall",
 "recipientAccountId": "123456789012"
}
```


Example: Image lifecycle policy action

The following example shows a CloudTrail log entry that demonstrates when an image is expired due to a lifecycle policy rule. This event type can be located by filtering for `PolicyExecutionEvent` for the event name field.

When you test a lifecycle policy preview, Amazon ECR generates a CloudTrail log entry with the event name field of `DryRunEvent`, with the exact same structure as the `PolicyExecutionEvent`. By changing the event name to `DryRunEvent`, you can filter on dry run events instead.

Important

Due to the size limitations of individual CloudTrail events, for lifecycle policy actions where 10 or more images are expired Amazon ECR sends multiple events to CloudTrail. Additionally, Amazon ECR includes a maximum of 100 tags per image.


```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "accountId": "123456789012",
        "invokedBy": "AWS Internal"
    },
    "eventTime": "2020-03-12T20:22:12Z",
    "eventSource": "ecr.amazonaws.com",
    "eventName": "PolicyExecutionEvent",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "AWS Internal",
    "userAgent": "AWS Internal",
    "requestParameters": null,
    "responseElements": null,
    "eventID": "9354dd7f-9aac-4e9d-956d-12561a4923aa",
    "readOnly": true,
    "resources": [
        {
            "ARN": "arn:aws:ecr:us-west-2:123456789012:repository/testrepo",
            "accountId": "123456789012",
            "type": "AWS::ECR::Repository"
        }
    ],
    "eventType": "AwsServiceEvent",
    "recipientAccountId": "123456789012",
    "serviceEventDetails": {
        "repositoryName": "testrepo",
        "lifecycleEventPolicy": {
            "lifecycleEventRules": [
                {
```

```
                    "rulePriority": 1,
                    "description": "remove all images > 2",
                    "lifecycleEventSelection": {
                        "tagStatus": "Any",
                        "tagPrefixList": [],
                        "countType": "Image count more than",
                        "countNumber": 2
                    },
                    "action": "expire"
                }
            ],
            "lastEvaluatedAt": 0,
            "policyVersion": 1,
            "policyId": "ceb86829-58e7-9498-920c-aa042e33037b"
        },
        "lifecycleEventImageActions": [
            {
                "lifecycleEventImage": {
                    "digest":
"sha256:ddba4d27a7ffc3f86dd6c2f92041af252a1f23a8e742c90e6e1297bfa1bc0c45",
                    "tagStatus": "Tagged",
                    "tagList": [
                        "alpine"
                    ],
                    "pushedAt": 1584042813000
                },
                "rulePriority": 1
            },
            {
                "lifecycleEventImage": {
                    "digest":
"sha256:6ab380c5a5acf71c1b6660d645d2cd79cc8ce91b38e0352cbf9561e050427baf",
                    "tagStatus": "Tagged",
                    "tagList": [
                        "centos"
                    ],
                    "pushedAt": 1584042842000
                },
                "rulePriority": 1
            }
        ],
        "lifecycleEventFailureDetails": [
            {
                "lifecycleEventImage": {
```

```
                    "digest":
 "sha256:9117e1bc28cd20751e584b4ccd19b1178d14cf02d134b04ce6be0cc51bff762a",
                    "tagStatus": "Untagged",
                    "tagList": [],
                    "pushedAt": 1584042844000
                },
                "rulePriority": 1,
                "failureCode": "ImageReferencedByManifestList",
                "failureReason": "Requested image referenced by manifest list:
 [sha256:4b27c83d44a18c31543039d9e8b2786043ec6c8d00804d5800c5148d6b6f65bc]"
            }
        ]
    }
}
```

Example: Image referrers action

The following example shows a AWS CloudTrail log entry that demonstrates when an OCI 1.1 compliant client fetches a list of referrers, or reference artifacts, for an image using the `Referrers` API.

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE:account_name",
        "arn": "arn:aws:sts::123456789012:user/Mary_Major",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AIDACKCEVSQ6C2EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/Admin",
                "accountId": "123456789012",
                "userName": "Admin"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2024-10-08T16:38:39Z",
                "mfaAuthenticated": "false"
            },
            "ec2RoleDelivery": "2.0"
```

```
        },
        "invokedBy": "ecr.amazonaws.com"
    },
    "eventTime": "2024-10-08T17:22:51Z",
    "eventSource": "ecr.amazonaws.com",
    "eventName": "ListImageReferrers",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "ecr.amazonaws.com",
    "userAgent": "ecr.amazonaws.com",
    "requestParameters": {
        "registryId": "123456789012",
        "repositoryName": "testrepo",
        "subjectId": {
            "imageDigest":
 "sha256:000b9b805af1cdb60628898c9f411996301a1c13afd3dbef1d8a16ac6dbf503a"
        },
        "nextToken": "urD72mdD/mC8b5-EXAMPLE"
    },
    "responseElements": null,
    "requestID": "cb8c167e-EXAMPLE",
    "eventID": "e3c6f4ce-EXAMPLE",
    "readOnly": true,
    "resources": [
        {
            "accountId": "123456789012",
            "ARN": "arn:aws:ecr:us-east-2:123456789012:repository/testrepo"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management"
}
```

Using Amazon ECR with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
AWS SDK for C++	AWS SDK for C++ code examples
AWS CLI	AWS CLI code examples
AWS SDK for Go	AWS SDK for Go code examples
AWS SDK for Java	AWS SDK for Java code examples
AWS SDK for JavaScript	AWS SDK for JavaScript code examples
AWS SDK for Kotlin	AWS SDK for Kotlin code examples
AWS SDK for .NET	AWS SDK for .NET code examples
AWS SDK for PHP	AWS SDK for PHP code examples
AWS Tools for PowerShell	AWS Tools for PowerShell code examples
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples
AWS SDK for Ruby	AWS SDK for Ruby code examples
AWS SDK for Rust	AWS SDK for Rust code examples
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples
AWS SDK for Swift	AWS SDK for Swift code examples

For examples specific to Amazon ECR, see Code examples for Amazon ECR using AWS SDKs.

Example availability

Can't find what you need? Request a code example by using the **Provide feedback** link at the bottom of this page.

Code examples for Amazon ECR using AWS SDKs

The following code examples show how to use Amazon ECR with an AWS software development kit (SDK).

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Get started

Hello Amazon ECR

The following code examples show how to get started using Amazon ECR.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.ecr.EcrClient;
import software.amazon.awssdk.services.ecr.model.EcrException;
import software.amazon.awssdk.services.ecr.model.ListImagesRequest;
import software.amazon.awssdk.services.ecr.paginators.ListImagesIterable;

public class HelloECR {

    public static void main(String[] args) {
        final String usage = """
            Usage:    <repositoryName>
```

```
            Where:
               repositoryName - The name of the Amazon ECR repository.
            """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String repoName = args[0];
        EcrClient ecrClient = EcrClient.builder()
            .region(Region.US_EAST_1)
            .build();

        listImageTags(ecrClient, repoName);
    }
    public static void listImageTags(EcrClient ecrClient, String repoName){
        ListImagesRequest listImagesPaginator = ListImagesRequest.builder()
            .repositoryName(repoName)
            .build();

        ListImagesIterable imagesIterable =
 ecrClient.listImagesPaginator(listImagesPaginator);
        imagesIterable.stream()
            .flatMap(r -> r.imageIds().stream())
            .forEach(image -> System.out.println("The docker image tag is: "
 +image.imageTag()));
    }
}
```

- For API details, see listImages in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import aws.sdk.kotlin.services.ecr.EcrClient
import aws.sdk.kotlin.services.ecr.model.ListImagesRequest
import kotlin.system.exitProcess

suspend fun main(args: Array<String>) {
    val usage = """
            Usage: <repositoryName>

            Where:
               repositoryName - The name of the Amazon ECR repository.

    """.trimIndent()

    if (args.size != 1) {
        println(usage)
        exitProcess(1)
    }

    val repoName = args[0]
    listImageTags(repoName)
}

suspend fun listImageTags(repoName: String?) {
    val listImages =
        ListImagesRequest {
            repositoryName = repoName
        }

    EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
        val imageResponse = ecrClient.listImages(listImages)
        imageResponse.imageIds?.forEach { imageId ->
            println("Image tag: ${imageId.imageTag}")
        }
    }
}
```

- For API details, see listImages in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import boto3
import argparse
from boto3 import client


def hello_ecr(ecr_client: client, repository_name: str) -> None:
    """
    Use the AWS SDK for Python (Boto3) to create an Amazon Elastic Container
 Registry (Amazon ECR)
    client and list the images in a repository.
    This example uses the default settings specified in your shared credentials
    and config files.

    :param ecr_client: A Boto3 Amazon ECR Client object. This object wraps
                           the low-level Amazon ECR service API.
    :param repository_name: The name of an Amazon ECR repository in your account.
    """
    print(
        f"Hello, Amazon ECR! Let's list some images in the repository
 '{repository_name}':\n"
    )
    paginator = ecr_client.get_paginator("list_images")
    page_iterator = paginator.paginate(
        repositoryName=repository_name, PaginationConfig={"MaxItems": 10}
    )

    image_names: [str] = []
    for page in page_iterator:
        for schedule in page["imageIds"]:
            image_names.append(schedule["imageTag"])

    print(f"{len(image_names)} image(s) retrieved.")
```

```
    for schedule_name in image_names:
        print(f"\t{schedule_name}")


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Run hello Amazon ECR.")
    parser.add_argument(
        "--repository-name",
        type=str,
        help="the name of an Amazon ECR repository in your account.",
        required=True,
    )
    args = parser.parse_args()

    hello_ecr(boto3.client("ecr"), args.repository_name)
```

- For API details, see listImages in *AWS SDK for Python (Boto3) API Reference*.

Code examples

- Basic examples for Amazon ECR using AWS SDKs
  - Hello Amazon ECR
  - Learn the basics of Amazon ECR with an AWS SDK
  - Actions for Amazon ECR using AWS SDKs
    - Use CreateRepository with an AWS SDK or CLI
    - Use DeleteRepository with an AWS SDK or CLI
    - Use DescribeImages with an AWS SDK or CLI
    - Use DescribeRepositories with an AWS SDK or CLI
    - Use GetAuthorizationToken with an AWS SDK or CLI
    - Use GetRepositoryPolicy with an AWS SDK or CLI
    - Use ListImages with an AWS SDK or CLI
    - Use PushImageCmd with an AWS SDK
    - Use PutLifeCyclePolicy with an AWS SDK or CLI
    - Use SetRepositoryPolicy with an AWS SDK or CLI
    - Use StartLifecyclePolicyPreview with an AWS SDK or CLI

Basic examples for Amazon ECR using AWS SDKs

The following code examples show how to use the basics of Amazon Elastic Container Registry with AWS SDKs.

Examples

- Hello Amazon ECR
- Learn the basics of Amazon ECR with an AWS SDK
- Actions for Amazon ECR using AWS SDKs
  - Use CreateRepository with an AWS SDK or CLI
  - Use DeleteRepository with an AWS SDK or CLI
  - Use DescribeImages with an AWS SDK or CLI
  - Use DescribeRepositories with an AWS SDK or CLI
  - Use GetAuthorizationToken with an AWS SDK or CLI
  - Use GetRepositoryPolicy with an AWS SDK or CLI
  - Use ListImages with an AWS SDK or CLI
  - Use PushImageCmd with an AWS SDK
  - Use PutLifeCyclePolicy with an AWS SDK or CLI
  - Use SetRepositoryPolicy with an AWS SDK or CLI
  - Use StartLifecyclePolicyPreview with an AWS SDK or CLI

Hello Amazon ECR

The following code examples show how to get started using Amazon ECR.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.ecr.EcrClient;
import software.amazon.awssdk.services.ecr.model.EcrException;
import software.amazon.awssdk.services.ecr.model.ListImagesRequest;
import software.amazon.awssdk.services.ecr.paginators.ListImagesIterable;

public class HelloECR {

    public static void main(String[] args) {
        final String usage = """
            Usage:    <repositoryName>

            Where:
               repositoryName - The name of the Amazon ECR repository.
            """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String repoName = args[0];
        EcrClient ecrClient = EcrClient.builder()
            .region(Region.US_EAST_1)
            .build();

        listImageTags(ecrClient, repoName);
    }
    public static void listImageTags(EcrClient ecrClient, String repoName){
        ListImagesRequest listImagesPaginator = ListImagesRequest.builder()
            .repositoryName(repoName)
            .build();

        ListImagesIterable imagesIterable =
 ecrClient.listImagesPaginator(listImagesPaginator);
        imagesIterable.stream()
            .flatMap(r -> r.imageIds().stream())
            .forEach(image -> System.out.println("The docker image tag is: "
 +image.imageTag()));
    }
}
```

- For API details, see listImages in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import aws.sdk.kotlin.services.ecr.EcrClient
import aws.sdk.kotlin.services.ecr.model.ListImagesRequest
import kotlin.system.exitProcess

suspend fun main(args: Array<String>) {
    val usage = """
            Usage: <repositoryName>

            Where:
               repositoryName - The name of the Amazon ECR repository.

    """.trimIndent()

    if (args.size != 1) {
        println(usage)
        exitProcess(1)
    }

    val repoName = args[0]
    listImageTags(repoName)
}

suspend fun listImageTags(repoName: String?) {
    val listImages =
        ListImagesRequest {
            repositoryName = repoName
        }

    EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
        val imageResponse = ecrClient.listImages(listImages)
```

```
        imageResponse.imageIds?.forEach { imageId ->
            println("Image tag: ${imageId.imageTag}")
        }
    }
}
```

- For API details, see listImages in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import boto3
import argparse
from boto3 import client


def hello_ecr(ecr_client: client, repository_name: str) -> None:
    """
    Use the AWS SDK for Python (Boto3) to create an Amazon Elastic Container
 Registry (Amazon ECR)
    client and list the images in a repository.
    This example uses the default settings specified in your shared credentials
    and config files.

    :param ecr_client: A Boto3 Amazon ECR Client object. This object wraps
                           the low-level Amazon ECR service API.
    :param repository_name: The name of an Amazon ECR repository in your account.
    """
    print(
        f"Hello, Amazon ECR! Let's list some images in the repository
 '{repository_name}':\n"
    )
    paginator = ecr_client.get_paginator("list_images")
    page_iterator = paginator.paginate(
```

```
        repositoryName=repository_name, PaginationConfig={"MaxItems": 10}
    )

    image_names: [str] = []
    for page in page_iterator:
        for schedule in page["imageIds"]:
            image_names.append(schedule["imageTag"])

    print(f"{len(image_names)} image(s) retrieved.")
    for schedule_name in image_names:
        print(f"\t{schedule_name}")


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Run hello Amazon ECR.")
    parser.add_argument(
        "--repository-name",
        type=str,
        help="the name of an Amazon ECR repository in your account.",
        required=True,
    )
    args = parser.parse_args()

    hello_ecr(boto3.client("ecr"), args.repository_name)
```

- For API details, see listImages in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Learn the basics of Amazon ECR with an AWS SDK

The following code examples show how to:

- Create an Amazon ECR repository.
- Set repository policies.
- Retrieve repository URIs.
- Get Amazon ECR authorization tokens.
- Set lifecycle policies for Amazon ECR repositories.

- Push a Docker image to an Amazon ECR repository.
- Verify the existence of an image in an Amazon ECR repository.
- List Amazon ECR repositories for your account and get details about them.
- Delete Amazon ECR repositories.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario demonstrating Amazon ECR features.

```
import software.amazon.awssdk.services.ecr.model.EcrException;
import
 software.amazon.awssdk.services.ecr.model.RepositoryPolicyNotFoundException;

import java.util.Scanner;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * This Java code example requires an IAM Role that has permissions to interact
 with the Amazon ECR service.
 *
 * To create an IAM role, see:
 *
 * https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_create.html
 *
 * This Java scenario example requires a local docker image named echo-text.
 Without a local image,
```

```
 * this Java program will not successfully run. For more information including
 how to create the local
 * image, see:
 *
 * /scenarios/basics/ecr/README
 *
 */
public class ECRScenario {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");
    public static void main(String[] args) {
        final String usage = """
            Usage: <iamRoleARN> <accountId>

            Where:
               iamRoleARN - The IAM role ARN that has the necessary permissions
 to access and manage the Amazon ECR repository.
               accountId - Your AWS account number.
            """;

        if (args.length != 2) {
            System.out.println(usage);
            return;
        }

        ECRActions ecrActions = new ECRActions();
        String iamRole = args[0];
        String accountId = args[1];
        String localImageName;

        Scanner scanner = new Scanner(System.in);
        System.out.println("""
             The Amazon Elastic Container Registry (ECR) is a fully-managed
 Docker container registry
             service provided by AWS. It allows developers and organizations to
 securely
             store, manage, and deploy Docker container images.
             ECR provides a simple and scalable way to manage container images
 throughout their lifecycle,
             from building and testing to production deployment.\s

             The `EcrAsyncClient` interface in the AWS SDK for Java 2.x provides
 a set of methods to
```

```
            programmatically interact with the Amazon ECR service. This allows
developers to
            automate the storage, retrieval, and management of container images
as part of their application
            deployment pipelines. With ECR, teams can focus on building and
deploying their
            applications without having to worry about the underlying
infrastructure required to
            host and manage a container registry.

           This scenario walks you through how to perform key operations for
this service.
           Let's get started...

           You have two choices:
           1 - Run the entire program.
           2 - Delete an existing Amazon ECR repository named echo-text (created
from a previous execution of
           this program that did not complete).
           """);

       while (true) {
           String input = scanner.nextLine();
           if (input.trim().equalsIgnoreCase("1")) {
               System.out.println("Continuing with the program...");
               System.out.println("");
               break;
           } else if (input.trim().equalsIgnoreCase("2")) {
               String repoName = "echo-text";
               ecrActions.deleteECRRepository(repoName);
               return;
           } else {
               // Handle invalid input.
               System.out.println("Invalid input. Please try again.");
           }
       }

       waitForInputToContinue(scanner);
       System.out.println(DASHES);

       System.out.println("""
          1. Create an ECR repository.
```

```
            The first task is to ensure we have a local Docker image named echo-
text.
            If this image exists, then an Amazon ECR repository is created.

            An ECR repository is a private Docker container repository provided
            by Amazon Web Services (AWS). It is a managed service that makes it
 easy
            to store, manage, and deploy Docker container images.\s
            """ );

        // Ensure that a local docker image named echo-text exists.
        boolean doesExist = ecrActions.isEchoTextImagePresent();
        String repoName;
        if (!doesExist){
            System.out.println("The local image named echo-text does not exist");
            return;
        } else {
            localImageName = "echo-text";
            repoName = "echo-text";
        }

        try {
            String repoArn = ecrActions.createECRRepository(repoName);
            System.out.println("The ARN of the ECR repository is " + repoArn);

        } catch (IllegalArgumentException e) {
            System.err.println("Invalid repository name: " + e.getMessage());
            return;
        } catch (RuntimeException e) {
            System.err.println("An error occurred while creating the ECR
 repository: " + e.getMessage());
            e.printStackTrace();
            return;
        }
        waitForInputToContinue(scanner);

        System.out.println(DASHES);
        System.out.println("""
        2. Set an ECR repository policy.

        Setting an ECR repository policy using the `setRepositoryPolicy` function
 is crucial for maintaining
        the security and integrity of your container images. The repository
 policy allows you to
```

```
        define specific rules and restrictions for accessing and managing the
images stored within your ECR
        repository.
        """);
        waitForInputToContinue(scanner);
        try {
            ecrActions.setRepoPolicy(repoName, iamRole);

        } catch (RepositoryPolicyNotFoundException e) {
            System.err.println("Invalid repository name: " + e.getMessage());
            return;
        } catch (EcrException e) {
            System.err.println("An ECR exception occurred: " + e.getMessage());
            return;
        } catch (RuntimeException e) {
            System.err.println("An error occurred while creating the ECR
repository: " + e.getMessage());
            return;
        }
        waitForInputToContinue(scanner);

        System.out.println(DASHES);
        System.out.println("""
        3. Display ECR repository policy.

        Now we will retrieve the ECR policy to ensure it was successfully set.
        """);
        waitForInputToContinue(scanner);
        try {
            String policyText = ecrActions.getRepoPolicy(repoName);
            System.out.println("Policy Text:");
            System.out.println(policyText);

        } catch (EcrException e) {
            System.err.println("An ECR exception occurred: " + e.getMessage());
            return;
        } catch (RuntimeException e) {
            System.err.println("An error occurred while creating the ECR
repository: " + e.getMessage());
            return;
        }

        waitForInputToContinue(scanner);
```

```
        System.out.println(DASHES);
        System.out.println("""
        4. Retrieve an ECR authorization token.

        You need an authorization token to securely access and interact with the
Amazon ECR registry.
        The `getAuthorizationToken` method of the `EcrAsyncClient` is responsible
for securely accessing
        and interacting with an Amazon ECR repository. This operation is
responsible for obtaining a
        valid authorization token, which is required to authenticate your
requests to the ECR service.

        Without a valid authorization token, you would not be able to perform any
operations on the
        ECR repository, such as pushing, pulling, or managing your Docker images.

        """);
        waitForInputToContinue(scanner);
        try {
             ecrActions.getAuthToken();

        } catch (EcrException e) {
            System.err.println("An ECR exception occurred: " + e.getMessage());
            return;
        } catch (RuntimeException e) {
            System.err.println("An error occurred while retrieving the
authorization token: " + e.getMessage());
            return;
        }
        waitForInputToContinue(scanner);

        System.out.println(DASHES);
        System.out.println("""
        5. Get the ECR Repository URI.

        The URI  of an Amazon ECR repository is important. When you want to
deploy a container image to
        a container orchestration platform like Amazon Elastic Kubernetes Service
(EKS)
        or Amazon Elastic Container Service (ECS), you need to specify the full
image URI,
        which includes the ECR repository URI. This allows the container runtime
to pull the
```

```
        correct container image from the ECR repository.
       """);
        waitForInputToContinue(scanner);

        try {
            ecrActions.getRepositoryURI(repoName);

        } catch (EcrException e) {
            System.err.println("An ECR exception occurred: " + e.getMessage());
            return;

        } catch (RuntimeException e) {
            System.err.println("An error occurred while retrieving the URI: " +
e.getMessage());
            return;
        }
        waitForInputToContinue(scanner);

        System.out.println(DASHES);
        System.out.println("""
            6. Set an ECR Lifecycle Policy.

            An ECR Lifecycle Policy is used to manage the lifecycle of Docker
images stored in your ECR repositories.
            These policies allow you to automatically remove old or unused Docker
images from your repositories,
            freeing up storage space and reducing costs.

            This example policy helps to maintain the size and efficiency of the
container registry
            by automatically removing older and potentially unused images,
ensuring that the
            storage is optimized and the registry remains up-to-date.
            """);
        waitForInputToContinue(scanner);
        try {
            ecrActions.setLifeCyclePolicy(repoName);

        } catch (RuntimeException e) {
            System.err.println("An error occurred while setting the lifecycle
policy: " + e.getMessage());
            e.printStackTrace();
            return;
        }
```

```
        waitForInputToContinue(scanner);

        System.out.println(DASHES);
        System.out.println("""
        7. Push a docker image to the Amazon ECR Repository.

        The `pushImageCmd()` method pushes a local Docker image to an Amazon ECR
repository.
        It sets up the Docker client by connecting to the local Docker host using
the default port.
        It then retrieves the authorization token for the ECR repository by
making a call to the AWS SDK.

        The method uses the authorization token to create an `AuthConfig` object,
which is used to authenticate
        the Docker client when pushing the image. Finally, the method tags the
Docker image with the specified
        repository name and image tag, and then pushes the image to the ECR
repository using the Docker client.
        If the push operation is successful, the method prints a message
indicating that the image was pushed to ECR.
        """);
        waitForInputToContinue(scanner);

        try {
            ecrActions.pushDockerImage(repoName, localImageName);

        } catch (RuntimeException e) {
            System.err.println("An error occurred while pushing a local Docker
image to Amazon ECR: " + e.getMessage());
            e.printStackTrace();
            return;
        }
        waitForInputToContinue(scanner);

        System.out.println(DASHES);
        System.out.println("8. Verify if the image is in the ECR Repository.");
        waitForInputToContinue(scanner);
        try {
            ecrActions.verifyImage(repoName, localImageName);

        } catch (EcrException e) {
            System.err.println("An ECR exception occurred: " + e.getMessage());
            return;
```

```
        } catch (RuntimeException e) {
            System.err.println("An error occurred " + e.getMessage());
            e.printStackTrace();
            return;
        }
        waitForInputToContinue(scanner);

        System.out.println(DASHES);
        System.out.println("9. As an optional step, you can interact with the
 image in Amazon ECR by using the CLI.");
        System.out.println("Would you like to view instructions on how to use the
 CLI to run the image? (y/n)");
        String ans = scanner.nextLine().trim();
        if (ans.equalsIgnoreCase("y")) {
            String instructions = """
            1. Authenticate with ECR - Before you can pull the image from Amazon
 ECR, you need to authenticate with the registry. You can do this using the AWS
 CLI:

                aws ecr get-login-password --region us-east-1 | docker login --
username AWS --password-stdin %s.dkr.ecr.us-east-1.amazonaws.com

            2. Describe the image using this command:

               aws ecr describe-images --repository-name %s --image-ids imageTag=
%s

            3. Run the Docker container and view the output using this command:

               docker run --rm %s.dkr.ecr.us-east-1.amazonaws.com/%s:%s
            """;

            instructions = String.format(instructions, accountId, repoName,
 localImageName, accountId, repoName, localImageName);
            System.out.println(instructions);
        }
        waitForInputToContinue(scanner);

        System.out.println(DASHES);
        System.out.println("10. Delete the ECR Repository.");
        System.out.println(
        """
        If the repository isn't empty, you must either delete the contents of the
 repository
```

```
       or use the force option (used in this scenario) to delete the repository
and have Amazon ECR delete all of its contents
       on your behalf.
       """);
       System.out.println("Would you like to delete the Amazon ECR Repository?
(y/n)");
       String delAns = scanner.nextLine().trim();
       if (delAns.equalsIgnoreCase("y")) {
           System.out.println("You selected to delete the AWS ECR resources.");

           try {
               ecrActions.deleteECRRepository(repoName);

           } catch (EcrException e) {
               System.err.println("An ECR exception occurred: " +
e.getMessage());
               return;
           } catch (RuntimeException e) {
               System.err.println("An error occurred while deleting the Docker
image: " + e.getMessage());
               e.printStackTrace();
               return;
           }
       }

       System.out.println(DASHES);
       System.out.println("This concludes the Amazon ECR SDK scenario");
       System.out.println(DASHES);
   }

  private static void waitForInputToContinue(Scanner scanner) {
      while (true) {
          System.out.println("");
          System.out.println("Enter 'c' followed by <ENTER> to continue:");
          String input = scanner.nextLine();

          if (input.trim().equalsIgnoreCase("c")) {
              System.out.println("Continuing with the program...");
              System.out.println("");
              break;
          } else {
              // Handle invalid input.
              System.out.println("Invalid input. Please try again.");
          }
```

```
        }
    }
}
```

A wrapper class for Amazon ECR SDK methods.

```
import com.github.dockerjava.api.DockerClient;
import com.github.dockerjava.api.exception.DockerClientException;
import com.github.dockerjava.api.model.AuthConfig;
import com.github.dockerjava.api.model.Image;
import com.github.dockerjava.core.DockerClientBuilder;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.http.async.SdkAsyncHttpClient;
import software.amazon.awssdk.http.nio.netty.NettyNioAsyncHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.ecr.EcrAsyncClient;
import software.amazon.awssdk.services.ecr.model.AuthorizationData;
import software.amazon.awssdk.services.ecr.model.CreateRepositoryRequest;
import software.amazon.awssdk.services.ecr.model.CreateRepositoryResponse;
import software.amazon.awssdk.services.ecr.model.DeleteRepositoryRequest;
import software.amazon.awssdk.services.ecr.model.DeleteRepositoryResponse;
import software.amazon.awssdk.services.ecr.model.DescribeImagesRequest;
import software.amazon.awssdk.services.ecr.model.DescribeImagesResponse;
import software.amazon.awssdk.services.ecr.model.DescribeRepositoriesRequest;
import software.amazon.awssdk.services.ecr.model.DescribeRepositoriesResponse;
import software.amazon.awssdk.services.ecr.model.EcrException;
import software.amazon.awssdk.services.ecr.model.GetAuthorizationTokenResponse;
import software.amazon.awssdk.services.ecr.model.GetRepositoryPolicyRequest;
import software.amazon.awssdk.services.ecr.model.GetRepositoryPolicyResponse;
import software.amazon.awssdk.services.ecr.model.ImageIdentifier;
import software.amazon.awssdk.services.ecr.model.Repository;
import
 software.amazon.awssdk.services.ecr.model.RepositoryPolicyNotFoundException;
import software.amazon.awssdk.services.ecr.model.SetRepositoryPolicyRequest;
import software.amazon.awssdk.services.ecr.model.SetRepositoryPolicyResponse;
import
 software.amazon.awssdk.services.ecr.model.StartLifecyclePolicyPreviewRequest;
import
 software.amazon.awssdk.services.ecr.model.StartLifecyclePolicyPreviewResponse;
import com.github.dockerjava.api.command.DockerCmdExecFactory;
```

```
import com.github.dockerjava.netty.NettyDockerCmdExecFactory;
import java.time.Duration;
import java.util.Base64;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionException;

public class ECRActions {
    private static EcrAsyncClient ecrClient;

    private static DockerClient dockerClient;

    private static Logger logger = LoggerFactory.getLogger(ECRActions.class);

    /**
     * Creates an Amazon Elastic Container Registry (Amazon ECR) repository.
     *
     * @param repoName the name of the repository to create.
     * @return the Amazon Resource Name (ARN) of the created repository, or an
 empty string if the operation failed.
     * @throws IllegalArgumentException     If repository name is invalid.
     * @throws RuntimeException             if an error occurs while creating the
 repository.
     */
    public String createECRRepository(String repoName) {
        if (repoName == null || repoName.isEmpty()) {
            throw new IllegalArgumentException("Repository name cannot be null or
 empty");
        }

        CreateRepositoryRequest request = CreateRepositoryRequest.builder()
            .repositoryName(repoName)
            .build();

        CompletableFuture<CreateRepositoryResponse> response =
 getAsyncClient().createRepository(request);
        try {
            CreateRepositoryResponse result = response.join();
            if (result != null) {
                System.out.println("The " + repoName + " repository was created
 successfully.");
                return result.repository().repositoryArn();
            } else {
                throw new RuntimeException("Unexpected response type");
```

```
            }
        } catch (CompletionException e) {
            Throwable cause = e.getCause();
            if (cause instanceof EcrException ex) {
                if
("RepositoryAlreadyExistsException".equals(ex.awsErrorDetails().errorCode())) {
                    System.out.println("The Amazon ECR repository already exists,
moving on...");
                    DescribeRepositoriesRequest describeRequest =
DescribeRepositoriesRequest.builder()
                        .repositoryNames(repoName)
                        .build();
                    DescribeRepositoriesResponse describeResponse =
getAsyncClient().describeRepositories(describeRequest).join();
                    return
describeResponse.repositories().get(0).repositoryArn();
                } else {
                    throw new RuntimeException(ex);
                }
            } else {
                throw new RuntimeException(e);
            }
        }
    }

    /**
     * Deletes an ECR (Elastic Container Registry) repository.
     *
     * @param repoName the name of the repository to delete.
     * @throws IllegalArgumentException if the repository name is null or empty.
     * @throws EcrException if there is an error deleting the repository.
     * @throws RuntimeException if an unexpected error occurs during the deletion
process.
     */
    public void deleteECRRepository(String repoName) {
        if (repoName == null || repoName.isEmpty()) {
            throw new IllegalArgumentException("Repository name cannot be null or
empty");
        }

        DeleteRepositoryRequest repositoryRequest =
DeleteRepositoryRequest.builder()
            .force(true)
            .repositoryName(repoName)
```

```
            .build();

        CompletableFuture<DeleteRepositoryResponse> response =
getAsyncClient().deleteRepository(repositoryRequest);
        response.whenComplete((deleteRepositoryResponse, ex) -> {
            if (deleteRepositoryResponse != null) {
                System.out.println("You have successfully deleted the " +
repoName + " repository");
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof EcrException) {
                    throw (EcrException) cause;
                } else {
                    throw new RuntimeException("Unexpected error: " +
cause.getMessage(), cause);
                }
            }
        });

        // Wait for the CompletableFuture to complete
        response.join();
    }



    private static DockerClient getDockerClient() {
        String osName = System.getProperty("os.name");
        if (osName.startsWith("Windows")) {
            // Make sure Docker Desktop is running.
            String dockerHost = "tcp://localhost:2375"; // Use the Docker Desktop
default port.
            DockerCmdExecFactory dockerCmdExecFactory = new
NettyDockerCmdExecFactory().withReadTimeout(20000).withConnectTimeout(20000);
            dockerClient =
DockerClientBuilder.getInstance(dockerHost).withDockerCmdExecFactory(dockerCmdExecFactor
        } else {
            dockerClient = DockerClientBuilder.getInstance().build();
        }
        return dockerClient;
    }

    /**
     * Retrieves an asynchronous Amazon Elastic Container Registry (ECR) client.
     *
```

```
     * @return the configured ECR asynchronous client.
     */
    private static EcrAsyncClient getAsyncClient() {

        /*
         The `NettyNioAsyncHttpClient` class is part of the AWS SDK for Java,
version 2,
         and it is designed to provide a high-performance, asynchronous HTTP
client for interacting with AWS services.
         It uses the Netty framework to handle the underlying network
communication and the Java NIO API to
         provide a non-blocking, event-driven approach to HTTP requests and
responses.
         */
        SdkAsyncHttpClient httpClient = NettyNioAsyncHttpClient.builder()
            .maxConcurrency(50)  // Adjust as needed.
            .connectionTimeout(Duration.ofSeconds(60))  // Set the connection
timeout.
            .readTimeout(Duration.ofSeconds(60))  // Set the read timeout.
            .writeTimeout(Duration.ofSeconds(60))  // Set the write timeout.
            .build();

        ClientOverrideConfiguration overrideConfig =
ClientOverrideConfiguration.builder()
            .apiCallTimeout(Duration.ofMinutes(2))  // Set the overall API call
timeout.
            .apiCallAttemptTimeout(Duration.ofSeconds(90))  // Set the individual
call attempt timeout.
            .build();

        if (ecrClient == null) {
            ecrClient = EcrAsyncClient.builder()
                .region(Region.US_EAST_1)
                .httpClient(httpClient)
                .overrideConfiguration(overrideConfig)
                .build();
        }
        return ecrClient;
    }

    /**
     * Sets the lifecycle policy for the specified repository.
     *
```

```
     * @param repoName the name of the repository for which to set the lifecycle
 policy.
     */
    public void setLifeCyclePolicy(String repoName) {
        /*
           This policy helps to maintain the size and efficiency of the container
 registry
           by automatically removing older and potentially unused images,
           ensuring that the storage is optimized and the registry remains up-to-
date.
         */
        String polText = """
             {
             "rules": [
                 {
                     "rulePriority": 1,
                     "description": "Expire images older than 14 days",
                     "selection": {
                         "tagStatus": "any",
                         "countType": "sinceImagePushed",
                         "countUnit": "days",
                         "countNumber": 14
                     },
                     "action": {
                         "type": "expire"
                     }
                 }
             ]
             }
             """;

        StartLifecyclePolicyPreviewRequest lifecyclePolicyPreviewRequest =
 StartLifecyclePolicyPreviewRequest.builder()
             .lifecyclePolicyText(polText)
             .repositoryName(repoName)
             .build();

        CompletableFuture<StartLifecyclePolicyPreviewResponse> response =
 getAsyncClient().startLifecyclePolicyPreview(lifecyclePolicyPreviewRequest);
        response.whenComplete((lifecyclePolicyPreviewResponse, ex) -> {
            if (lifecyclePolicyPreviewResponse != null) {
                System.out.println("Lifecycle policy preview started
 successfully.");
            } else {
```

```
                if (ex.getCause() instanceof EcrException) {
                    throw (EcrException) ex.getCause();
                } else {
                    String errorMessage = "Unexpected error occurred: " +
ex.getMessage();
                    throw new RuntimeException(errorMessage, ex);
                }
            }
        });
        // Wait for the CompletableFuture to complete.
        response.join();
    }

    /**
     * Verifies the existence of an image in an Amazon Elastic Container Registry
(Amazon ECR) repository asynchronously.
     *
     * @param repositoryName The name of the Amazon ECR repository.
     * @param imageTag       The tag of the image to verify.
     * @throws EcrException             if there is an error retrieving the image
information from Amazon ECR.
     * @throws CompletionException      if the asynchronous operation completes
exceptionally.
     */
    public void verifyImage(String repositoryName, String imageTag) {
        DescribeImagesRequest request = DescribeImagesRequest.builder()
            .repositoryName(repositoryName)
            .imageIds(ImageIdentifier.builder().imageTag(imageTag).build())
            .build();

        CompletableFuture<DescribeImagesResponse> response =
getAsyncClient().describeImages(request);
        response.whenComplete((describeImagesResponse, ex) -> {
            if (ex != null) {
                if (ex instanceof CompletionException) {
                    Throwable cause = ex.getCause();
                    if (cause instanceof EcrException) {
                        throw (EcrException) cause;
                    } else {
                        throw new RuntimeException("Unexpected error: " +
cause.getMessage(), cause);
                    }
                } else {
```

```
                    throw new RuntimeException("Unexpected error: " +
 ex.getCause());
                }
            } else if (describeImagesResponse != null && !
describeImagesResponse.imageDetails().isEmpty()) {
                System.out.println("Image is present in the repository.");
            } else {
                System.out.println("Image is not present in the repository.");
            }
        });

        // Wait for the CompletableFuture to complete.
        response.join();
    }

    /**
     * Retrieves the repository URI for the specified repository name.
     *
     * @param repoName the name of the repository to retrieve the URI for.
     * @return the repository URI for the specified repository name.
     * @throws EcrException        if there is an error retrieving the repository
 information.
     * @throws CompletionException if the asynchronous operation completes
 exceptionally.
     */
    public void getRepositoryURI(String repoName) {
        DescribeRepositoriesRequest request =
 DescribeRepositoriesRequest.builder()
            .repositoryNames(repoName)
            .build();

        CompletableFuture<DescribeRepositoriesResponse> response =
 getAsyncClient().describeRepositories(request);
        response.whenComplete((describeRepositoriesResponse, ex) -> {
            if (ex != null) {
                Throwable cause = ex.getCause();
                if (cause instanceof InterruptedException) {
                    Thread.currentThread().interrupt();
                    String errorMessage = "Thread interrupted while waiting for
 asynchronous operation: " + cause.getMessage();
                    throw new RuntimeException(errorMessage, cause);
                } else if (cause instanceof EcrException) {
                    throw (EcrException) cause;
                } else {
```

```
                    String errorMessage = "Unexpected error: " +
cause.getMessage();
                    throw new RuntimeException(errorMessage, cause);
                }
            } else {
                if (describeRepositoriesResponse != null) {
                    if (!describeRepositoriesResponse.repositories().isEmpty()) {
                        String repositoryUri =
describeRepositoriesResponse.repositories().get(0).repositoryUri();
                        System.out.println("Repository URI found: " +
repositoryUri);
                    } else {
                        System.out.println("No repositories found for the given
name.");
                    }
                } else {
                    System.err.println("No response received from
describeRepositories.");
                }
            }
        });
        response.join();
    }

    /**
     * Retrieves the authorization token for Amazon Elastic Container Registry
(ECR).
     * This method makes an asynchronous call to the ECR client to retrieve the
authorization token.
     * If the operation is successful, the method prints the token to the
console.
     * If an exception occurs, the method handles the exception and prints the
error message.
     *
     * @throws EcrException     if there is an error retrieving the authorization
token from ECR.
     * @throws RuntimeException if there is an unexpected error during the
operation.
     */
    public void getAuthToken() {
        CompletableFuture<GetAuthorizationTokenResponse> response =
getAsyncClient().getAuthorizationToken();
        response.whenComplete((authorizationTokenResponse, ex) -> {
            if (authorizationTokenResponse != null) {
```

```
                AuthorizationData authorizationData =
authorizationTokenResponse.authorizationData().get(0);
                String token = authorizationData.authorizationToken();
                if (!token.isEmpty()) {
                    System.out.println("The token was successfully retrieved.");
                }
            } else {
                if (ex.getCause() instanceof EcrException) {
                    throw (EcrException) ex.getCause();
                } else {
                    String errorMessage = "Unexpected error occurred: " +
ex.getMessage();
                    throw new RuntimeException(errorMessage, ex); // Rethrow the
exception
                }
            }
        });
        response.join();
    }

    /**
     * Gets the repository policy for the specified repository.
     *
     * @param repoName the name of the repository.
     * @throws EcrException if an AWS error occurs while getting the repository
policy.
     */
    public String getRepoPolicy(String repoName) {
        if (repoName == null || repoName.isEmpty()) {
            throw new IllegalArgumentException("Repository name cannot be null or
empty");
        }

        GetRepositoryPolicyRequest getRepositoryPolicyRequest =
GetRepositoryPolicyRequest.builder()
            .repositoryName(repoName)
            .build();

        CompletableFuture<GetRepositoryPolicyResponse> response =
getAsyncClient().getRepositoryPolicy(getRepositoryPolicyRequest);
        response.whenComplete((resp, ex) -> {
            if (resp != null) {
                System.out.println("Repository policy retrieved successfully.");
            } else {
```

```
                if (ex.getCause() instanceof EcrException) {
                    throw (EcrException) ex.getCause();
                } else {
                    String errorMessage = "Unexpected error occurred: " +
ex.getMessage();
                    throw new RuntimeException(errorMessage, ex);
                }
            }
        });

        GetRepositoryPolicyResponse result = response.join();
        return result != null ? result.policyText() : null;
    }

    /**
     * Sets the repository policy for the specified ECR repository.
     *
     * @param repoName the name of the ECR repository.
     * @param iamRole  the IAM role to be granted access to the repository.
     * @throws RepositoryPolicyNotFoundException if the repository policy does
not exist.
     * @throws EcrException                      if there is an unexpected error
setting the repository policy.
     */
    public void setRepoPolicy(String repoName, String iamRole) {
        /*
          This example policy document grants the specified AWS principal the
permission to perform the
          `ecr:BatchGetImage` action. This policy is designed to allow the
specified principal
          to retrieve Docker images from the ECR repository.
         */
        String policyDocumentTemplate = """
            {
             "Version" : "2012-10-17",
             "Statement" : [ {
               "Sid" : "new statement",
               "Effect" : "Allow",
               "Principal" : {
                 "AWS" : "%s"
               },
               "Action" : "ecr:BatchGetImage"
             } ]
           }
```

```
            """;

       String policyDocument = String.format(policyDocumentTemplate, iamRole);
       SetRepositoryPolicyRequest setRepositoryPolicyRequest =
SetRepositoryPolicyRequest.builder()
           .repositoryName(repoName)
           .policyText(policyDocument)
           .build();

       CompletableFuture<SetRepositoryPolicyResponse> response =
getAsyncClient().setRepositoryPolicy(setRepositoryPolicyRequest);
       response.whenComplete((resp, ex) -> {
           if (resp != null) {
               System.out.println("Repository policy set successfully.");
           } else {
               Throwable cause = ex.getCause();
               if (cause instanceof RepositoryPolicyNotFoundException) {
                   throw (RepositoryPolicyNotFoundException) cause;
               } else if (cause instanceof EcrException) {
                   throw (EcrException) cause;
               } else {
                   String errorMessage = "Unexpected error: " +
cause.getMessage();
                   throw new RuntimeException(errorMessage, cause);
               }
           }
       });
       response.join();
   }

   /**
    * Pushes a Docker image to an Amazon Elastic Container Registry (ECR)
repository.
    *
    * @param repoName  the name of the ECR repository to push the image to.
    * @param imageName the name of the Docker image.
    */
   public void pushDockerImage(String repoName, String imageName) {
       System.out.println("Pushing " + imageName + " to Amazon ECR will take a
few seconds.");
       CompletableFuture<AuthConfig> authResponseFuture =
getAsyncClient().getAuthorizationToken()
           .thenApply(response -> {
```

```
                String token =
 response.authorizationData().get(0).authorizationToken();
                String decodedToken = new
 String(Base64.getDecoder().decode(token));
                String password = decodedToken.substring(4);

                DescribeRepositoriesResponse descrRepoResponse =
 getAsyncClient().describeRepositories(b -> b.repositoryNames(repoName)).join();
                Repository repoData =
 descrRepoResponse.repositories().stream().filter(r ->
 r.repositoryName().equals(repoName)).findFirst().orElse(null);
                assert repoData != null;
                String registryURL = repoData.repositoryUri().split("/")[0];

                AuthConfig authConfig = new AuthConfig()
                    .withUsername("AWS")
                    .withPassword(password)
                    .withRegistryAddress(registryURL);
                return authConfig;
            })
            .thenCompose(authConfig -> {
                DescribeRepositoriesResponse descrRepoResponse =
 getAsyncClient().describeRepositories(b -> b.repositoryNames(repoName)).join();
                Repository repoData =
 descrRepoResponse.repositories().stream().filter(r ->
 r.repositoryName().equals(repoName)).findFirst().orElse(null);
                getDockerClient().tagImageCmd(imageName + ":latest",
 repoData.repositoryUri() + ":latest", imageName).exec();
                try {

 getDockerClient().pushImageCmd(repoData.repositoryUri()).withTag("echo-
text").withAuthConfig(authConfig).start().awaitCompletion();
                    System.out.println("The " + imageName + " was pushed to
 ECR");

                } catch (InterruptedException e) {
                    throw (RuntimeException) e.getCause();
                }
                return CompletableFuture.completedFuture(authConfig);
            });

        authResponseFuture.join();
    }
```

```
    // Make sure local image echo-text exists.
    public boolean isEchoTextImagePresent() {
        try {
            List<Image> images = getDockerClient().listImagesCmd().exec();
            boolean helloWorldFound = false;
            for (Image image : images) {
                String[] repoTags = image.getRepoTags();
                if (repoTags != null) {
                    for (String tag : repoTags) {
                        if (tag.startsWith("echo-text")) {
                            System.out.println(tag);
                            helloWorldFound = true;
                        }
                    }
                }
            }
            if (helloWorldFound) {
                System.out.println("The local image named echo-text exists.");
                return true;
            } else {
                System.out.println("The local image named echo-text does not
 exist.");
                return false;
            }
        } catch (DockerClientException ex) {
            logger.error("ERROR: " + ex.getMessage());
            return false;
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - CreateRepository
  - DeleteRepository
  - DescribeImages
  - DescribeRepositories
  - GetAuthorizationToken
  - GetRepositoryPolicy
  - SetRepositoryPolicy

- StartLifecyclePolicyPreview

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario demonstrating Amazon ECR features.

```
import java.util.Scanner

/**
 * Before running this Kotlin code example, set up your development environment,
 including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-kotlin/latest/developer-guide/setup.html
 *
 * This code example requires an IAM Role that has permissions to interact with
 the Amazon ECR service.
 *
 * To create an IAM role, see:
 *
 * https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_create.html
 *
 * This code example requires a local docker image named echo-text. Without a
 local image,
 * this program will not successfully run. For more information including how to
 create the local
 * image, see:
 *
 * /scenarios/basics/ecr/README
 *
 */

val DASHES = String(CharArray(80)).replace("\u0000", "-")
```

```
suspend fun main(args: Array<String>) {
    val usage =
        """
        Usage: <iamRoleARN> <accountId>

        Where:
           iamRoleARN - The IAM role ARN that has the necessary permissions to
 access and manage the Amazon ECR repository.
           accountId - Your AWS account number.

        """.trimIndent()

    if (args.size != 2) {
        println(usage)
        return
    }

    var iamRole = args[0]
    var localImageName: String
    var accountId = args[1]
    val ecrActions = ECRActions()
    val scanner = Scanner(System.`in`)

    println(
        """
        The Amazon Elastic Container Registry (ECR) is a fully-managed Docker
 container registry
        service provided by AWS. It allows developers and organizations to
 securely
        store, manage, and deploy Docker container images.
        ECR provides a simple and scalable way to manage container images
 throughout their lifecycle,
        from building and testing to production deployment.

        The `EcrClient` service client that is part of the AWS SDK for Kotlin
 provides a set of methods to
        programmatically interact with the Amazon ECR service. This allows
 developers to
        automate the storage, retrieval, and management of container images as
 part of their application
        deployment pipelines. With ECR, teams can focus on building and deploying
 their
```

```
        applications without having to worry about the underlying infrastructure
 required to
        host and manage a container registry.

        This scenario walks you through how to perform key operations for this
 service.
        Let's get started...

         You have two choices:
            1 - Run the entire program.
            2 - Delete an existing Amazon ECR repository named echo-text (created
 from a previous execution of
            this program that did not complete).

        """.trimIndent(),
    )

    while (true) {
        val input = scanner.nextLine()
        if (input.trim { it <= ' ' }.equals("1", ignoreCase = true)) {
            println("Continuing with the program...")
            println("")
            break
        } else if (input.trim { it <= ' ' }.equals("2", ignoreCase = true)) {
            val repoName = "echo-text"
            ecrActions.deleteECRRepository(repoName)
            return
        } else {
            // Handle invalid input.
            println("Invalid input. Please try again.")
        }
    }

    waitForInputToContinue(scanner)
    println(DASHES)
    println(
        """
        1. Create an ECR repository.

        The first task is to ensure we have a local Docker image named echo-
text.
        If this image exists, then an Amazon ECR repository is created.

        An ECR repository is a private Docker container repository provided
```

```
        by Amazon Web Services (AWS). It is a managed service that makes it easy
        to store, manage, and deploy Docker container images.

        """.trimIndent(),
    )

    // Ensure that a local docker image named echo-text exists.
    val doesExist = ecrActions.listLocalImages()
    val repoName: String
    if (!doesExist) {
        println("The local image named echo-text does not exist")
        return
    } else {
        localImageName = "echo-text"
        repoName = "echo-text"
    }

    val repoArn = ecrActions.createECRRepository(repoName).toString()
    println("The ARN of the ECR repository is $repoArn")
    waitForInputToContinue(scanner)

    println(DASHES)
    println(
        """
        2. Set an ECR repository policy.

        Setting an ECR repository policy using the `setRepositoryPolicy` function
is crucial for maintaining
        the security and integrity of your container images. The repository
policy allows you to
        define specific rules and restrictions for accessing and managing the
images stored within your ECR
        repository.

        """.trimIndent(),
    )
    waitForInputToContinue(scanner)
    ecrActions.setRepoPolicy(repoName, iamRole)
    waitForInputToContinue(scanner)

    println(DASHES)
    println(
        """
        3. Display ECR repository policy.
```

```
        Now we will retrieve the ECR policy to ensure it was successfully set.
        """.trimIndent(),
    )
    waitForInputToContinue(scanner)
    val policyText = ecrActions.getRepoPolicy(repoName)
    println("Policy Text:")
    println(policyText)
    waitForInputToContinue(scanner)

    println(DASHES)
    println(
        """
        4. Retrieve an ECR authorization token.

        You need an authorization token to securely access and interact with the
Amazon ECR registry.
        The `getAuthorizationToken` method of the `EcrAsyncClient` is responsible
for securely accessing
        and interacting with an Amazon ECR repository. This operation is
responsible for obtaining a
        valid authorization token, which is required to authenticate your
requests to the ECR service.

        Without a valid authorization token, you would not be able to perform any
operations on the
        ECR repository, such as pushing, pulling, or managing your Docker images.


        """.trimIndent(),
    )
    waitForInputToContinue(scanner)
    ecrActions.getAuthToken()
    waitForInputToContinue(scanner)

    println(DASHES)
    println(
        """
        5. Get the ECR Repository URI.

        The URI  of an Amazon ECR repository is important. When you want to
deploy a container image to
        a container orchestration platform like Amazon Elastic Kubernetes Service
(EKS)
```

```
        or Amazon Elastic Container Service (ECS), you need to specify the full
image URI,
        which includes the ECR repository URI. This allows the container runtime
to pull the
        correct container image from the ECR repository.

        """.trimIndent(),
    )
    waitForInputToContinue(scanner)
    val repositoryURI: String? = ecrActions.getRepositoryURI(repoName)
    println("The repository URI is $repositoryURI")
    waitForInputToContinue(scanner)

    println(DASHES)
    println(
        """
        6. Set an ECR Lifecycle Policy.

        An ECR Lifecycle Policy is used to manage the lifecycle of Docker images
stored in your ECR repositories.
        These policies allow you to automatically remove old or unused Docker
images from your repositories,
        freeing up storage space and reducing costs.

        """.trimIndent(),
    )
    waitForInputToContinue(scanner)
    val pol = ecrActions.setLifeCyclePolicy(repoName)
    println(pol)
    waitForInputToContinue(scanner)

    println(DASHES)
    println(
        """
        7. Push a docker image to the Amazon ECR Repository.

        The `pushImageCmd()` method pushes a local Docker image to an Amazon ECR
repository.
        It sets up the Docker client by connecting to the local Docker host using
the default port.
        It then retrieves the authorization token for the ECR repository by
making a call to the AWS SDK.
```

```
        The method uses the authorization token to create an `AuthConfig` object,
 which is used to authenticate
        the Docker client when pushing the image. Finally, the method tags the
 Docker image with the specified
        repository name and image tag, and then pushes the image to the ECR
 repository using the Docker client.
        If the push operation is successful, the method prints a message
 indicating that the image was pushed to ECR.

        """.trimIndent(),
    )

    waitForInputToContinue(scanner)
    ecrActions.pushDockerImage(repoName, localImageName)
    waitForInputToContinue(scanner)

    println(DASHES)
    println("8. Verify if the image is in the ECR Repository.")
    waitForInputToContinue(scanner)
    ecrActions.verifyImage(repoName, localImageName)
    waitForInputToContinue(scanner)

    println(DASHES)
    println("9. As an optional step, you can interact with the image in Amazon
 ECR by using the CLI.")
    println("Would you like to view instructions on how to use the CLI to run the
 image? (y/n)")
    val ans = scanner.nextLine().trim()
    if (ans.equals("y", true)) {
        val instructions = """
        1. Authenticate with ECR - Before you can pull the image from Amazon ECR,
 you need to authenticate with the registry. You can do this using the AWS CLI:

            aws ecr get-login-password --region us-east-1 | docker login --
username AWS --password-stdin $accountId.dkr.ecr.us-east-1.amazonaws.com

        2. Describe the image using this command:

           aws ecr describe-images --repository-name $repoName --image-ids
 imageTag=$localImageName

        3. Run the Docker container and view the output using this command:
```

```
            docker run --rm $accountId.dkr.ecr.us-east-1.amazonaws.com/$repoName:
$localImageName
        """
        println(instructions)
    }
    waitForInputToContinue(scanner)

    println(DASHES)
    println("10. Delete the ECR Repository.")
    println(
        """
        If the repository isn't empty, you must either delete the contents of the
 repository
        or use the force option (used in this scenario) to delete the repository
 and have Amazon ECR delete all of its contents
        on your behalf.

        """.trimIndent(),
    )
    println("Would you like to delete the Amazon ECR Repository? (y/n)")
    val delAns = scanner.nextLine().trim { it <= ' ' }
    if (delAns.equals("y", ignoreCase = true)) {
        println("You selected to delete the AWS ECR resources.")
        waitForInputToContinue(scanner)
        ecrActions.deleteECRRepository(repoName)
    }

    println(DASHES)
    println("This concludes the Amazon ECR SDK scenario")
    println(DASHES)
}

private fun waitForInputToContinue(scanner: Scanner) {
    while (true) {
        println("")
        println("Enter 'c' followed by <ENTER> to continue:")
        val input = scanner.nextLine()
        if (input.trim { it <= ' ' }.equals("c", ignoreCase = true)) {
            println("Continuing with the program...")
            println("")
            break
        } else {
            // Handle invalid input.
            println("Invalid input. Please try again.")
```

```
        }
    }
}
```

A wrapper class for Amazon ECR SDK methods.

```
import aws.sdk.kotlin.services.ecr.EcrClient
import aws.sdk.kotlin.services.ecr.model.CreateRepositoryRequest
import aws.sdk.kotlin.services.ecr.model.DeleteRepositoryRequest
import aws.sdk.kotlin.services.ecr.model.DescribeImagesRequest
import aws.sdk.kotlin.services.ecr.model.DescribeRepositoriesRequest
import aws.sdk.kotlin.services.ecr.model.EcrException
import aws.sdk.kotlin.services.ecr.model.GetRepositoryPolicyRequest
import aws.sdk.kotlin.services.ecr.model.ImageIdentifier
import aws.sdk.kotlin.services.ecr.model.RepositoryAlreadyExistsException
import aws.sdk.kotlin.services.ecr.model.SetRepositoryPolicyRequest
import aws.sdk.kotlin.services.ecr.model.StartLifecyclePolicyPreviewRequest
import com.github.dockerjava.api.DockerClient
import com.github.dockerjava.api.command.DockerCmdExecFactory
import com.github.dockerjava.api.model.AuthConfig
import com.github.dockerjava.core.DockerClientBuilder
import com.github.dockerjava.netty.NettyDockerCmdExecFactory
import java.io.IOException
import java.util.Base64

class ECRActions {
    private var dockerClient: DockerClient? = null

    private fun getDockerClient(): DockerClient? {
        val osName = System.getProperty("os.name")
        if (osName.startsWith("Windows")) {
            // Make sure Docker Desktop is running.
            val dockerHost = "tcp://localhost:2375" // Use the Docker Desktop
 default port.
            val dockerCmdExecFactory: DockerCmdExecFactory =

 NettyDockerCmdExecFactory().withReadTimeout(20000).withConnectTimeout(20000)
            dockerClient =
 DockerClientBuilder.getInstance(dockerHost).withDockerCmdExecFactory(dockerCmdExecFactor
        } else {
            dockerClient = DockerClientBuilder.getInstance().build()
        }
```

```
        return dockerClient
    }


    /**
     * Sets the lifecycle policy for the specified repository.
     *
     * @param repoName the name of the repository for which to set the lifecycle
policy.
     */
    suspend fun setLifeCyclePolicy(repoName: String): String? {
        val polText =
            """
             {
             "rules": [
                 {
                     "rulePriority": 1,
                     "description": "Expire images older than 14 days",
                     "selection": {
                         "tagStatus": "any",
                         "countType": "sinceImagePushed",
                         "countUnit": "days",
                         "countNumber": 14
                     },
                     "action": {
                         "type": "expire"
                     }
                 }
            ]
            }

            """.trimIndent()
        val lifecyclePolicyPreviewRequest =
            StartLifecyclePolicyPreviewRequest {
                lifecyclePolicyText = polText
                repositoryName = repoName
            }

        // Execute the request asynchronously.
        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val response =
ecrClient.startLifecyclePolicyPreview(lifecyclePolicyPreviewRequest)
            return response.lifecyclePolicyText
        }
```

```
    }


    /**
     * Retrieves the repository URI for the specified repository name.
     *
     * @param repoName the name of the repository to retrieve the URI for.
     * @return the repository URI for the specified repository name.
     */
    suspend fun getRepositoryURI(repoName: String?): String? {
        require(!(repoName == null || repoName.isEmpty())) { "Repository name
cannot be null or empty" }
        val request =
            DescribeRepositoriesRequest {
                repositoryNames = listOf(repoName)
            }

        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val describeRepositoriesResponse =
ecrClient.describeRepositories(request)
            if (!describeRepositoriesResponse.repositories?.isEmpty()!!) {
                return
describeRepositoriesResponse?.repositories?.get(0)?.repositoryUri
            } else {
                println("No repositories found for the given name.")
                return ""
            }
        }
    }


    /**
     * Retrieves the authorization token for Amazon Elastic Container Registry
(ECR).
     *
     */
    suspend fun getAuthToken() {
        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            // Retrieve the authorization token for ECR.
            val response = ecrClient.getAuthorizationToken()
            val authorizationData = response.authorizationData?.get(0)
            val token = authorizationData?.authorizationToken
            if (token != null) {
                println("The token was successfully retrieved.")
```

```
            }
        }
    }


    /**
     * Gets the repository policy for the specified repository.
     *
     * @param repoName the name of the repository.
     */
    suspend fun getRepoPolicy(repoName: String?): String? {
        require(!(repoName == null || repoName.isEmpty())) { "Repository name
cannot be null or empty" }

        // Create the request
        val getRepositoryPolicyRequest =
            GetRepositoryPolicyRequest {
                repositoryName = repoName
            }
        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val response =
ecrClient.getRepositoryPolicy(getRepositoryPolicyRequest)
            val responseText = response.policyText
            return responseText
        }
    }


    /**
     * Sets the repository policy for the specified ECR repository.
     *
     * @param repoName the name of the ECR repository.
     * @param iamRole the IAM role to be granted access to the repository.
     */
    suspend fun setRepoPolicy(
        repoName: String?,
        iamRole: String?,
    ) {
        val policyDocumentTemplate =
            """
             {
              "Version" : "2012-10-17",
              "Statement" : [ {
                "Sid" : "new statement",
```

```
                "Effect" : "Allow",
                "Principal" : {
                  "AWS" : "$iamRole"
                },
                "Action" : "ecr:BatchGetImage"
              } ]
            }

            """.trimIndent()
        val setRepositoryPolicyRequest =
            SetRepositoryPolicyRequest {
                repositoryName = repoName
                policyText = policyDocumentTemplate
            }

        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val response =
ecrClient.setRepositoryPolicy(setRepositoryPolicyRequest)
            if (response != null) {
                println("Repository policy set successfully.")
            }
        }
    }


    /**
     * Creates an Amazon Elastic Container Registry (Amazon ECR) repository.
     *
     * @param repoName the name of the repository to create.
     * @return the Amazon Resource Name (ARN) of the created repository, or an
empty string if the operation failed.
     * @throws RepositoryAlreadyExistsException if the repository exists.
     * @throws EcrException         if an error occurs while creating the
repository.
     */
    suspend fun createECRRepository(repoName: String?): String? {
        val request =
            CreateRepositoryRequest {
                repositoryName = repoName
            }

        return try {
            EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
                val response = ecrClient.createRepository(request)
```

```
                response.repository?.repositoryArn
            }
        } catch (e: RepositoryAlreadyExistsException) {
            println("Repository already exists: $repoName")
            repoName?.let { getRepoARN(it) }
        } catch (e: EcrException) {
            println("An error occurred: ${e.message}")
            null
        }
    }

    suspend fun getRepoARN(repoName: String): String? {
        // Fetch the existing repository's ARN.
        val describeRequest =
            DescribeRepositoriesRequest {
                repositoryNames = listOf(repoName)
            }
        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val describeResponse =
ecrClient.describeRepositories(describeRequest)
            return describeResponse.repositories?.get(0)?.repositoryArn
        }
    }

    fun listLocalImages(): Boolean = try {
        val images = getDockerClient()?.listImagesCmd()?.exec()
        images?.any { image ->
            image.repoTags?.any { tag -> tag.startsWith("echo-text") } ?: false
        } ?: false
    } catch (ex: Exception) {
        println("ERROR: ${ex.message}")
        false
    }


    /**
     * Pushes a Docker image to an Amazon Elastic Container Registry (ECR)
repository.
     *
     * @param repoName the name of the ECR repository to push the image to.
     * @param imageName the name of the Docker image.
     */
    suspend fun pushDockerImage(
        repoName: String,
```

```
        imageName: String,
    ) {
        println("Pushing $imageName to $repoName will take a few seconds")
        val authConfig = getAuthConfig(repoName)

        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val desRequest =
                DescribeRepositoriesRequest {
                    repositoryNames = listOf(repoName)
                }

            val describeRepoResponse = ecrClient.describeRepositories(desRequest)
            val repoData =
                describeRepoResponse.repositories?.firstOrNull
{ it.repositoryName == repoName }
                    ?: throw RuntimeException("Repository not found: $repoName")

            val tagImageCmd = getDockerClient()?.tagImageCmd("$imageName",
"${repoData.repositoryUri}", imageName)
            if (tagImageCmd != null) {
                tagImageCmd.exec()
            }
            val pushImageCmd =
                repoData.repositoryUri?.let {
                    dockerClient?.pushImageCmd(it)
                        // ?.withTag("latest")
                        ?.withAuthConfig(authConfig)
                }

            try {
                if (pushImageCmd != null) {
                    pushImageCmd.start().awaitCompletion()
                }
                println("The $imageName was pushed to Amazon ECR")
            } catch (e: IOException) {
                throw RuntimeException(e)
            }
        }
    }


    /**
     * Verifies the existence of an image in an Amazon Elastic Container Registry
(Amazon ECR) repository asynchronously.
```

```
     *
     * @param repositoryName The name of the Amazon ECR repository.
     * @param imageTag       The tag of the image to verify.
     */
    suspend fun verifyImage(
        repoName: String?,
        imageTagVal: String?,
    ) {
        require(!(repoName == null || repoName.isEmpty())) { "Repository name
 cannot be null or empty" }
        require(!(imageTagVal == null || imageTagVal.isEmpty())) { "Image tag
 cannot be null or empty" }

        val imageId =
            ImageIdentifier {
                imageTag = imageTagVal
            }
        val request =
            DescribeImagesRequest {
                repositoryName = repoName
                imageIds = listOf(imageId)
            }

        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val describeImagesResponse = ecrClient.describeImages(request)
            if (describeImagesResponse != null && !
describeImagesResponse.imageDetails?.isEmpty()!!) {
                println("Image is present in the repository.")
            } else {
                println("Image is not present in the repository.")
            }
        }
    }


    /**
     * Deletes an ECR (Elastic Container Registry) repository.
     *
     * @param repoName the name of the repository to delete.
     */
    suspend fun deleteECRRepository(repoName: String) {
        if (repoName.isNullOrEmpty()) {
            throw IllegalArgumentException("Repository name cannot be null or
 empty")
```

```
        }

        val repositoryRequest =
            DeleteRepositoryRequest {
                force = true
                repositoryName = repoName
            }

        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            ecrClient.deleteRepository(repositoryRequest)
            println("You have successfully deleted the $repoName repository")
        }
    }

    // Return an AuthConfig.
    private suspend fun getAuthConfig(repoName: String): AuthConfig {
        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            // Retrieve the authorization token for ECR.
            val response = ecrClient.getAuthorizationToken()
            val authorizationData = response.authorizationData?.get(0)
            val token = authorizationData?.authorizationToken
            val decodedToken = String(Base64.getDecoder().decode(token))
            val password = decodedToken.substring(4)

            val request =
                DescribeRepositoriesRequest {
                    repositoryNames = listOf(repoName)
                }

            val descrRepoResponse = ecrClient.describeRepositories(request)
            val repoData = descrRepoResponse.repositories?.firstOrNull
 { it.repositoryName == repoName }
            val registryURL: String =
 repoData?.repositoryUri?.split("/")?.get(0) ?: ""

            return AuthConfig()
                .withUsername("AWS")
                .withPassword(password)
                .withRegistryAddress(registryURL)
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Kotlin API reference*.
  - CreateRepository
  - DeleteRepository
  - DescribeImages
  - DescribeRepositories
  - GetAuthorizationToken
  - GetRepositoryPolicy
  - SetRepositoryPolicy
  - StartLifecyclePolicyPreview

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario at a command prompt.

```
class ECRGettingStarted:
    """
    A scenario that demonstrates how to use Boto3 to perform basic operations
 using
    Amazon ECR.
    """

    def __init__(
        self,
        ecr_wrapper: ECRWrapper,
        docker_client: docker.DockerClient,
    ):
        self.ecr_wrapper = ecr_wrapper
        self.docker_client = docker_client
        self.tag = "echo-text"
        self.repository_name = "ecr-basics"
```

```
        self.docker_image = None
        self.full_tag_name = None
        self.repository = None

    def run(self, role_arn: str) -> None:
        """
        Runs the scenario.
        """
        print(
            """
The Amazon Elastic Container Registry (ECR) is a fully-managed Docker container
 registry
service provided by AWS. It allows developers and organizations to securely
store, manage, and deploy Docker container images.
ECR provides a simple and scalable way to manage container images throughout
 their lifecycle,
from building and testing to production deployment.

The `ECRWrapper' class is a wrapper for the Boto3 'ecr' client. The 'ecr' client
 provides a set of methods to
programmatically interact with the Amazon ECR service. This allows developers to
automate the storage, retrieval, and management of container images as part of
 their application
deployment pipelines. With ECR, teams can focus on building and deploying their
applications without having to worry about the underlying infrastructure required
 to
host and manage a container registry.

This scenario walks you through how to perform key operations for this service.
Let's get started...
        """
        )
        press_enter_to_continue()
        print_dashes()
        print(
            f"""
* Create an ECR repository.

An ECR repository is a private Docker container repository provided
by Amazon Web Services (AWS). It is a managed service that makes it easy
to store, manage, and deploy Docker container images.
        """
        )
        print(f"Creating a repository named {self.repository_name}")
```

```
        self.repository =
 self.ecr_wrapper.create_repository(self.repository_name)
        print(f"The ARN of the ECR repository is
 {self.repository['repositoryArn']}")
        repository_uri = self.repository["repositoryUri"]
        press_enter_to_continue()
        print_dashes()

        print(
            f"""
* Build a Docker image.

Create a local Docker image if it does not already exist.
A Python Docker client is used to execute Docker commands.
You must have Docker installed and running.
            """
        )
        print(f"Building a docker image from 'docker_files/Dockerfile'")
        self.full_tag_name = f"{repository_uri}:{self.tag}"
        self.docker_image = self.docker_client.images.build(
            path="docker_files", tag=self.full_tag_name
        )[0]
        print(f"Docker image {self.full_tag_name} successfully built.")
        press_enter_to_continue()
        print_dashes()

        if role_arn is None:
            print(
                """
* Because an IAM role ARN was not provided, a role policy will not be set for
 this repository.
            """
            )
        else:
            print(
                """
* Set an ECR repository policy.

Setting an ECR repository policy using the `setRepositoryPolicy` function is
 crucial for maintaining
the security and integrity of your container images. The repository policy allows
 you to
define specific rules and restrictions for accessing and managing the images
 stored within your ECR
```

```
repository.
        """
            )

            self.grant_role_download_access(role_arn)
            print(f"Download access granted to the IAM role ARN {role_arn}")
            press_enter_to_continue()
            print_dashes()

            print(
                """
* Display ECR repository policy.

Now we will retrieve the ECR policy to ensure it was successfully set.
            """
            )

            policy_text =
 self.ecr_wrapper.get_repository_policy(self.repository_name)
            print("Policy Text:")
            print(f"{policy_text}")
            press_enter_to_continue()
            print_dashes()

        print(
            """
* Retrieve an ECR authorization token.

You need an authorization token to securely access and interact with the Amazon
 ECR registry.
The `get_authorization_token` method of the `ecr` client is responsible for
 securely accessing
and interacting with an Amazon ECR repository. This operation is responsible for
 obtaining a
valid authorization token, which is required to authenticate your requests to the
 ECR service.

Without a valid authorization token, you would not be able to perform any
 operations on the
ECR repository, such as pushing, pulling, or managing your Docker images.
        """
        )

        authorization_token = self.ecr_wrapper.get_authorization_token()
```

```
        print("Authorization token retrieved.")
        press_enter_to_continue()
        print_dashes()
        print(
            """
* Get the ECR Repository URI.

The URI  of an Amazon ECR repository is important. When you want to deploy a
 container image to
a container orchestration platform like Amazon Elastic Kubernetes Service (EKS)
or Amazon Elastic Container Service (ECS), you need to specify the full image
 URI,
which includes the ECR repository URI. This allows the container runtime to pull
 the
correct container image from the ECR repository.
        """
        )
        repository_descriptions = self.ecr_wrapper.describe_repositories(
            [self.repository_name]
        )
        repository_uri = repository_descriptions[0]["repositoryUri"]
        print(f"Repository URI found: {repository_uri}")
        press_enter_to_continue()
        print_dashes()

        print(
            """
* Set an ECR Lifecycle Policy.

An ECR Lifecycle Policy is used to manage the lifecycle of Docker images stored
 in your ECR repositories.
These policies allow you to automatically remove old or unused Docker images from
 your repositories,
freeing up storage space and reducing costs.

This example policy helps to maintain the size and efficiency of the container
 registry
by automatically removing older and potentially unused images, ensuring that the
storage is optimized and the registry remains up-to-date.
            """
        )
        press_enter_to_continue()
        self.put_expiration_policy()
        print(f"An expiration policy was added to the repository.")
```

```
        print_dashes()

        print(
            """
* Push a docker image to the Amazon ECR Repository.

The Docker client uses the authorization token is used to authenticate the when
 pushing the image to the
ECR repository.
        """
        )
        decoded_authorization =
 base64.b64decode(authorization_token).decode("utf-8")
        username, password = decoded_authorization.split(":")

        resp = self.docker_client.api.push(
            repository=repository_uri,
            auth_config={"username": username, "password": password},
            tag=self.tag,
            stream=True,
            decode=True,
        )
        for line in resp:
            print(line)

        print_dashes()

        print("* Verify if the image is in the ECR Repository.")
        image_descriptions = self.ecr_wrapper.describe_images(
            self.repository_name, [self.tag]
        )
        if len(image_descriptions) > 0:
            print("Image found in ECR Repository.")
        else:
            print("Image not found in ECR Repository.")
        press_enter_to_continue()
        print_dashes()

        print(
            "* As an optional step, you can interact with the image in Amazon ECR
 by using the CLI."
        )
        if q.ask(
```

```
            "Would you like to view instructions on how to use the CLI to run the
 image? (y/n)",
            q.is_yesno,
        ):
            print(
                f"""
1. Authenticate with ECR - Before you can pull the image from Amazon ECR, you
 need to authenticate with the registry. You can do this using the AWS CLI:

   aws ecr get-login-password --region us-east-1 | docker login --username AWS
 --password-stdin {repository_uri.split("/")[0]}

2. Describe the image using this command:

  aws ecr describe-images --repository-name {self.repository_name} --image-ids
 imageTag={self.tag}

3. Run the Docker container and view the output using this command:

  docker run --rm {self.full_tag_name}
"""
            )

        self.cleanup(True)

    def cleanup(self, ask: bool):
        """
        Deletes the resources created in this scenario.
        :param ask: If True, prompts the user to confirm before deleting the
 resources.
        """
        if self.repository is not None and (
            not ask
            or q.ask(
                f"Would you like to delete the ECR repository
 '{self.repository_name}? (y/n) "
            )
        ):
            print(f"Deleting the ECR repository '{self.repository_name}'.")
            self.ecr_wrapper.delete_repository(self.repository_name)

        if self.full_tag_name is not None and (
            not ask
            or q.ask(
```

```
                f"Would you like to delete the local Docker image
'{self.full_tag_name}? (y/n) "
            )
        ):
            print(f"Deleting the docker image '{self.full_tag_name}'.")
            self.docker_client.images.remove(self.full_tag_name)

    def grant_role_download_access(self, role_arn: str):
        """
        Grants the specified role access to download images from the ECR
repository.

        :param role_arn: The ARN of the role to grant access to.
        """
        policy_json = {
            "Version": "2008-10-17",
            "Statement": [
                {
                    "Sid": "AllowDownload",
                    "Effect": "Allow",
                    "Principal": {"AWS": role_arn},
                    "Action": ["ecr:BatchGetImage"],
                }
            ],
        }

        self.ecr_wrapper.set_repository_policy(
            self.repository_name, json.dumps(policy_json)
        )


    def put_expiration_policy(self):
        """
        Puts an expiration policy on the ECR repository.
        """
        policy_json = {
            "rules": [
                {
                    "rulePriority": 1,
                    "description": "Expire images older than 14 days",
                    "selection": {
                        "tagStatus": "any",
                        "countType": "sinceImagePushed",
                        "countUnit": "days",
```

```
                        "countNumber": 14,
                    },
                    "action": {"type": "expire"},
                }
            ]
        }

        self.ecr_wrapper.put_lifecycle_policy(
            self.repository_name, json.dumps(policy_json)
        )



if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Run Amazon ECR getting started scenario."
    )
    parser.add_argument(
        "--iam-role-arn",
        type=str,
        default=None,
        help="an optional IAM role ARN that will be granted access to download
 images from a repository.",
        required=False,
    )
    parser.add_argument(
        "--no-art",
        action="store_true",
        help="accessibility setting that suppresses art in the console output.",
    )
    args = parser.parse_args()
    no_art = args.no_art
    iam_role_arn = args.iam_role_arn
    demo = None
    a_docker_client = None
    try:
        a_docker_client = docker.from_env()
        if not a_docker_client.ping():
            raise docker.errors.DockerException("Docker is not running.")
    except docker.errors.DockerException as err:
        logging.error(
            """
        The Python Docker client could not be created.
        Do you have Docker installed and running?
```

```
        Here is the error message:
        %s
        """,
            err,
        )
        sys.exit("Error with Docker.")
    try:
        an_ecr_wrapper = ECRWrapper.from_client()
        demo = ECRGettingStarted(an_ecr_wrapper, a_docker_client)
        demo.run(iam_role_arn)

    except Exception as exception:
        logging.exception("Something went wrong with the demo!")
        if demo is not None:
            demo.cleanup(False)
```

ECRWrapper class that wraps Amazon ECR actions.

```
class ECRWrapper:
    def __init__(self, ecr_client: client):
        self.ecr_client = ecr_client

    @classmethod
    def from_client(cls) -> "ECRWrapper":
        """
        Creates a ECRWrapper instance with a default Amazon ECR client.

        :return: An instance of ECRWrapper initialized with the default Amazon
 ECR client.
        """
        ecr_client = boto3.client("ecr")
        return cls(ecr_client)


    def create_repository(self, repository_name: str) -> dict[str, any]:
        """
        Creates an ECR repository.

        :param repository_name: The name of the repository to create.
        :return: A dictionary of the created repository.
        """
```

```
        try:
            response =
self.ecr_client.create_repository(repositoryName=repository_name)
            return response["repository"]
        except ClientError as err:
            if err.response["Error"]["Code"] ==
"RepositoryAlreadyExistsException":
                print(f"Repository {repository_name} already exists.")
                response = self.ecr_client.describe_repositories(
                    repositoryNames=[repository_name]
                )
                return self.describe_repositories([repository_name])[0]
            else:
                logger.error(
                    "Error creating repository %s. Here's why %s",
                    repository_name,
                    err.response["Error"]["Message"],
                )
                raise


    def delete_repository(self, repository_name: str):
        """
        Deletes an ECR repository.

        :param repository_name: The name of the repository to delete.
        """
        try:
            self.ecr_client.delete_repository(
                repositoryName=repository_name, force=True
            )
            print(f"Deleted repository {repository_name}.")
        except ClientError as err:
            logger.error(
                "Couldn't delete repository %s.. Here's why %s",
                repository_name,
                err.response["Error"]["Message"],
            )
            raise


    def set_repository_policy(self, repository_name: str, policy_text: str):
        """
        Sets the policy for an ECR repository.
```

```
        :param repository_name: The name of the repository to set the policy for.
        :param policy_text: The policy text to set.
        """
        try:
            self.ecr_client.set_repository_policy(
                repositoryName=repository_name, policyText=policy_text
            )
            print(f"Set repository policy for repository {repository_name}.")
        except ClientError as err:
            if err.response["Error"]["Code"] ==
"RepositoryPolicyNotFoundException":
                logger.error("Repository does not exist. %s.", repository_name)
                raise
            else:
                logger.error(
                    "Couldn't set repository policy for repository %s. Here's why
%s",
                    repository_name,
                    err.response["Error"]["Message"],
                )
                raise


    def get_repository_policy(self, repository_name: str) -> str:
        """
        Gets the policy for an ECR repository.

        :param repository_name: The name of the repository to get the policy for.
        :return: The policy text.
        """
        try:
            response = self.ecr_client.get_repository_policy(
                repositoryName=repository_name
            )
            return response["policyText"]
        except ClientError as err:
            if err.response["Error"]["Code"] ==
"RepositoryPolicyNotFoundException":
                logger.error("Repository does not exist. %s.", repository_name)
                raise
            else:
                logger.error(
```

```
                    "Couldn't get repository policy for repository %s. Here's why
%s",
                    repository_name,
                    err.response["Error"]["Message"],
                )
                raise


    def get_authorization_token(self) -> str:
        """
        Gets an authorization token for an ECR repository.

        :return: The authorization token.
        """
        try:
            response = self.ecr_client.get_authorization_token()
            return response["authorizationData"][0]["authorizationToken"]
        except ClientError as err:
            logger.error(
                "Couldn't get authorization token. Here's why %s",
                err.response["Error"]["Message"],
            )
            raise


    def describe_repositories(self, repository_names: list[str]) -> list[dict]:
        """
        Describes ECR repositories.

        :param repository_names: The names of the repositories to describe.
        :return: The list of repository descriptions.
        """
        try:
            response = self.ecr_client.describe_repositories(
                repositoryNames=repository_names
            )
            return response["repositories"]
        except ClientError as err:
            logger.error(
                "Couldn't describe repositories. Here's why %s",
                err.response["Error"]["Message"],
            )
            raise
```

```
   def put_lifecycle_policy(self, repository_name: str, lifecycle_policy_text:
str):
       """
       Puts a lifecycle policy for an ECR repository.

       :param repository_name: The name of the repository to put the lifecycle
policy for.
       :param lifecycle_policy_text: The lifecycle policy text to put.
       """
       try:
           self.ecr_client.put_lifecycle_policy(
               repositoryName=repository_name,
               lifecyclePolicyText=lifecycle_policy_text,
           )
           print(f"Put lifecycle policy for repository {repository_name}.")
       except ClientError as err:
           logger.error(
               "Couldn't put lifecycle policy for repository %s. Here's why %s",
               repository_name,
               err.response["Error"]["Message"],
           )
           raise


   def describe_images(
       self, repository_name: str, image_ids: list[str] = None
   ) -> list[dict]:
       """
       Describes ECR images.

       :param repository_name: The name of the repository to describe images
for.
       :param image_ids: The optional IDs of images to describe.
       :return: The list of image descriptions.
       """
       try:
           params = {
               "repositoryName": repository_name,
           }
           if image_ids is not None:
               params["imageIds"] = [{"imageTag": tag} for tag in image_ids]

           paginator = self.ecr_client.get_paginator("describe_images")
```

```
            image_descriptions = []
            for page in paginator.paginate(**params):
                image_descriptions.extend(page["imageDetails"])
            return image_descriptions
        except ClientError as err:
            logger.error(
                "Couldn't describe images. Here's why %s",
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
  - CreateRepository
  - DeleteRepository
  - DescribeImages
  - DescribeRepositories
  - GetAuthorizationToken
  - GetRepositoryPolicy
  - SetRepositoryPolicy
  - StartLifecyclePolicyPreview

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Actions for Amazon ECR using AWS SDKs

The following code examples demonstrate how to perform individual Amazon ECR actions with AWS SDKs. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

The following examples include only the most commonly used actions. For a complete list, see the Amazon Elastic Container Registry API Reference.

Examples

- Use CreateRepository with an AWS SDK or CLI
- Use DeleteRepository with an AWS SDK or CLI
- Use DescribeImages with an AWS SDK or CLI
- Use DescribeRepositories with an AWS SDK or CLI
- Use GetAuthorizationToken with an AWS SDK or CLI
- Use GetRepositoryPolicy with an AWS SDK or CLI
- Use ListImages with an AWS SDK or CLI
- Use PushImageCmd with an AWS SDK
- Use PutLifeCyclePolicy with an AWS SDK or CLI
- Use SetRepositoryPolicy with an AWS SDK or CLI
- Use StartLifecyclePolicyPreview with an AWS SDK or CLI

Use `CreateRepository` with an AWS SDK or CLI

The following code examples show how to use `CreateRepository`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

CLI

AWS CLI

Example 1: To create a repository

The following `create-repository` example creates a repository inside the specified namespace in the default registry for an account.

```
aws ecr create-repository \
    --repository-name project-a/sample-repo
```

Output:

```
{
```

```
    "repository": {
        "registryId": "123456789012",
        "repositoryName": "project-a/sample-repo",
        "repositoryArn": "arn:aws:ecr:us-west-2:123456789012:repository/project-
a/sample-repo"
    }
}
```

For more information, see Creating a Repository in the *Amazon ECR User Guide*.

Example 2: To create a repository configured with image tag immutability

The following `create-repository` example creates a repository configured for tag immutability in the default registry for an account.

```
aws ecr create-repository \
    --repository-name project-a/sample-repo \
    --image-tag-mutability IMMUTABLE
```

Output:

```
{
    "repository": {
        "registryId": "123456789012",
        "repositoryName": "project-a/sample-repo",
        "repositoryArn": "arn:aws:ecr:us-west-2:123456789012:repository/project-
a/sample-repo",
        "imageTagMutability": "IMMUTABLE"
    }
}
```

For more information, see Image Tag Mutability in the *Amazon ECR User Guide*.

Example 3: To create a repository configured with a scanning configuration

The following `create-repository` example creates a repository configured to perform a vulnerability scan on image push in the default registry for an account.

```
aws ecr create-repository \
    --repository-name project-a/sample-repo \
    --image-scanning-configuration scanOnPush=true
```

Output:

```
{
    "repository": {
        "registryId": "123456789012",
        "repositoryName": "project-a/sample-repo",
        "repositoryArn": "arn:aws:ecr:us-west-2:123456789012:repository/project-
a/sample-repo",
        "imageScanningConfiguration": {
            "scanOnPush": true
        }
    }
}
```

For more information, see Image Scanning in the *Amazon ECR User Guide*.

- For API details, see CreateRepository in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Creates an Amazon Elastic Container Registry (Amazon ECR) repository.
     *
     * @param repoName the name of the repository to create.
     * @return the Amazon Resource Name (ARN) of the created repository, or an
 empty string if the operation failed.
     * @throws IllegalArgumentException     If repository name is invalid.
     * @throws RuntimeException             if an error occurs while creating the
 repository.
     */
    public String createECRRepository(String repoName) {
        if (repoName == null || repoName.isEmpty()) {
            throw new IllegalArgumentException("Repository name cannot be null or
 empty");
```

```
        }

        CreateRepositoryRequest request = CreateRepositoryRequest.builder()
            .repositoryName(repoName)
            .build();

        CompletableFuture<CreateRepositoryResponse> response =
getAsyncClient().createRepository(request);
        try {
            CreateRepositoryResponse result = response.join();
            if (result != null) {
                System.out.println("The " + repoName + " repository was created
successfully.");
                return result.repository().repositoryArn();
            } else {
                throw new RuntimeException("Unexpected response type");
            }
        } catch (CompletionException e) {
            Throwable cause = e.getCause();
            if (cause instanceof EcrException ex) {
                if
("RepositoryAlreadyExistsException".equals(ex.awsErrorDetails().errorCode())) {
                    System.out.println("The Amazon ECR repository already exists,
moving on...");
                    DescribeRepositoriesRequest describeRequest =
DescribeRepositoriesRequest.builder()
                        .repositoryNames(repoName)
                        .build();
                    DescribeRepositoriesResponse describeResponse =
getAsyncClient().describeRepositories(describeRequest).join();
                    return
describeResponse.repositories().get(0).repositoryArn();
                } else {
                    throw new RuntimeException(ex);
                }
            } else {
                throw new RuntimeException(e);
            }
        }
    }
```

- For API details, see [CreateRepository](#) in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Creates an Amazon Elastic Container Registry (Amazon ECR) repository.
     *
     * @param repoName the name of the repository to create.
     * @return the Amazon Resource Name (ARN) of the created repository, or an
empty string if the operation failed.
     * @throws RepositoryAlreadyExistsException if the repository exists.
     * @throws EcrException         if an error occurs while creating the
repository.
     */
    suspend fun createECRRepository(repoName: String?): String? {
        val request =
            CreateRepositoryRequest {
                repositoryName = repoName
            }

        return try {
            EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
                val response = ecrClient.createRepository(request)
                response.repository?.repositoryArn
            }
        } catch (e: RepositoryAlreadyExistsException) {
            println("Repository already exists: $repoName")
            repoName?.let { getRepoARN(it) }
        } catch (e: EcrException) {
            println("An error occurred: ${e.message}")
            null
        }
    }
```

- For API details, see CreateRepository in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class ECRWrapper:
    def __init__(self, ecr_client: client):
        self.ecr_client = ecr_client

    @classmethod
    def from_client(cls) -> "ECRWrapper":
        """
        Creates a ECRWrapper instance with a default Amazon ECR client.

        :return: An instance of ECRWrapper initialized with the default Amazon
 ECR client.
        """
        ecr_client = boto3.client("ecr")
        return cls(ecr_client)


    def create_repository(self, repository_name: str) -> dict[str, any]:
        """
        Creates an ECR repository.

        :param repository_name: The name of the repository to create.
        :return: A dictionary of the created repository.
        """
        try:
            response =
 self.ecr_client.create_repository(repositoryName=repository_name)
            return response["repository"]
        except ClientError as err:
            if err.response["Error"]["Code"] ==
 "RepositoryAlreadyExistsException":
```

```
                print(f"Repository {repository_name} already exists.")
                response = self.ecr_client.describe_repositories(
                    repositoryNames=[repository_name]
                )
                return self.describe_repositories([repository_name])[0]
            else:
                logger.error(
                    "Error creating repository %s. Here's why %s",
                    repository_name,
                    err.response["Error"]["Message"],
                )
                raise
```

- For API details, see CreateRepository in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteRepository` with an AWS SDK or CLI

The following code examples show how to use `DeleteRepository`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

CLI

AWS CLI

To delete a repository

The following `delete-repository` example command force deletes the specified repository in the default registry for an account. The `--force` flag is required if the repository contains images.

```
aws ecr delete-repository \
```

```
    --repository-name ubuntu \
    --force
```

Output:

```
{
    "repository": {
        "registryId": "123456789012",
        "repositoryName": "ubuntu",
        "repositoryArn": "arn:aws:ecr:us-west-2:123456789012:repository/ubuntu"
    }
}
```

For more information, see Deleting a Repository in the *Amazon ECR User Guide*.

- For API details, see DeleteRepository in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Deletes an ECR (Elastic Container Registry) repository.
     *
     * @param repoName the name of the repository to delete.
     * @throws IllegalArgumentException if the repository name is null or empty.
     * @throws EcrException if there is an error deleting the repository.
     * @throws RuntimeException if an unexpected error occurs during the deletion
process.
     */
    public void deleteECRRepository(String repoName) {
        if (repoName == null || repoName.isEmpty()) {
            throw new IllegalArgumentException("Repository name cannot be null or
empty");
        }
```

```
        DeleteRepositoryRequest repositoryRequest =
DeleteRepositoryRequest.builder()
            .force(true)
            .repositoryName(repoName)
            .build();

        CompletableFuture<DeleteRepositoryResponse> response =
getAsyncClient().deleteRepository(repositoryRequest);
        response.whenComplete((deleteRepositoryResponse, ex) -> {
            if (deleteRepositoryResponse != null) {
                System.out.println("You have successfully deleted the " +
repoName + " repository");
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof EcrException) {
                    throw (EcrException) cause;
                } else {
                    throw new RuntimeException("Unexpected error: " +
cause.getMessage(), cause);
                }
            }
        });

        // Wait for the CompletableFuture to complete
        response.join();
    }
```

- For API details, see DeleteRepository in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/**
```

```
     * Deletes an ECR (Elastic Container Registry) repository.
     *
     * @param repoName the name of the repository to delete.
     */
    suspend fun deleteECRRepository(repoName: String) {
        if (repoName.isNullOrEmpty()) {
            throw IllegalArgumentException("Repository name cannot be null or
 empty")
        }

        val repositoryRequest =
            DeleteRepositoryRequest {
                force = true
                repositoryName = repoName
            }

        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            ecrClient.deleteRepository(repositoryRequest)
            println("You have successfully deleted the $repoName repository")
        }
    }
```

- For API details, see DeleteRepository in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class ECRWrapper:
    def __init__(self, ecr_client: client):
        self.ecr_client = ecr_client

    @classmethod
    def from_client(cls) -> "ECRWrapper":
        """
```

```
        Creates a ECRWrapper instance with a default Amazon ECR client.

        :return: An instance of ECRWrapper initialized with the default Amazon
ECR client.
        """
        ecr_client = boto3.client("ecr")
        return cls(ecr_client)


    def delete_repository(self, repository_name: str):
        """
        Deletes an ECR repository.

        :param repository_name: The name of the repository to delete.
        """
        try:
            self.ecr_client.delete_repository(
                repositoryName=repository_name, force=True
            )
            print(f"Deleted repository {repository_name}.")
        except ClientError as err:
            logger.error(
                "Couldn't delete repository %s.. Here's why %s",
                repository_name,
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see DeleteRepository in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeImages` with an AWS SDK or CLI

The following code examples show how to use `DescribeImages`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

CLI

AWS CLI

To describe an image in a repository

The folowing `describe-images` example displays details about an image in the `cluster-autoscaler` repository with the tag `v1.13.6`.

```
aws ecr describe-images \
    --repository-name cluster-autoscaler \
    --image-ids imageTag=v1.13.6
```

Output:

```
{
    "imageDetails": [
        {
            "registryId": "012345678910",
            "repositoryName": "cluster-autoscaler",
            "imageDigest":
 "sha256:4a1c6567c38904384ebc64e35b7eeddd8451110c299e3368d2210066487d97e5",
            "imageTags": [
                "v1.13.6"
            ],
            "imageSizeInBytes": 48318255,
            "imagePushedAt": 1565128275.0
        }
    ]
}
```

- For API details, see DescribeImages in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /**
    * Verifies the existence of an image in an Amazon Elastic Container Registry
(Amazon ECR) repository asynchronously.
    *
    * @param repositoryName The name of the Amazon ECR repository.
    * @param imageTag       The tag of the image to verify.
    * @throws EcrException             if there is an error retrieving the image
information from Amazon ECR.
    * @throws CompletionException      if the asynchronous operation completes
exceptionally.
    */
   public void verifyImage(String repositoryName, String imageTag) {
       DescribeImagesRequest request = DescribeImagesRequest.builder()
           .repositoryName(repositoryName)
           .imageIds(ImageIdentifier.builder().imageTag(imageTag).build())
           .build();

       CompletableFuture<DescribeImagesResponse> response =
getAsyncClient().describeImages(request);
       response.whenComplete((describeImagesResponse, ex) -> {
           if (ex != null) {
               if (ex instanceof CompletionException) {
                   Throwable cause = ex.getCause();
                   if (cause instanceof EcrException) {
                       throw (EcrException) cause;
                   } else {
                       throw new RuntimeException("Unexpected error: " +
cause.getMessage(), cause);
                   }
               } else {
                   throw new RuntimeException("Unexpected error: " +
ex.getCause());
```

```
                }
            } else if (describeImagesResponse != null && !
describeImagesResponse.imageDetails().isEmpty()) {
                System.out.println("Image is present in the repository.");
            } else {
                System.out.println("Image is not present in the repository.");
            }
        });

        // Wait for the CompletableFuture to complete.
        response.join();
    }
```

- For API details, see DescribeImages in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Verifies the existence of an image in an Amazon Elastic Container Registry
(Amazon ECR) repository asynchronously.
     *
     * @param repositoryName The name of the Amazon ECR repository.
     * @param imageTag       The tag of the image to verify.
     */
    suspend fun verifyImage(
        repoName: String?,
        imageTagVal: String?,
    ) {
        require(!(repoName == null || repoName.isEmpty())) { "Repository name
cannot be null or empty" }
        require(!(imageTagVal == null || imageTagVal.isEmpty())) { "Image tag
cannot be null or empty" }
```

```
        val imageId =
            ImageIdentifier {
                imageTag = imageTagVal
            }
        val request =
            DescribeImagesRequest {
                repositoryName = repoName
                imageIds = listOf(imageId)
            }

        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val describeImagesResponse = ecrClient.describeImages(request)
            if (describeImagesResponse != null && !
describeImagesResponse.imageDetails?.isEmpty()!!) {
                println("Image is present in the repository.")
            } else {
                println("Image is not present in the repository.")
            }
        }
    }
```

- For API details, see DescribeImages in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class ECRWrapper:
    def __init__(self, ecr_client: client):
        self.ecr_client = ecr_client

    @classmethod
    def from_client(cls) -> "ECRWrapper":
        """
```

```
        Creates a ECRWrapper instance with a default Amazon ECR client.

        :return: An instance of ECRWrapper initialized with the default Amazon
ECR client.
        """
        ecr_client = boto3.client("ecr")
        return cls(ecr_client)


    def describe_images(
        self, repository_name: str, image_ids: list[str] = None
    ) -> list[dict]:
        """
        Describes ECR images.

        :param repository_name: The name of the repository to describe images
for.
        :param image_ids: The optional IDs of images to describe.
        :return: The list of image descriptions.
        """
        try:
            params = {
                "repositoryName": repository_name,
            }
            if image_ids is not None:
                params["imageIds"] = [{"imageTag": tag} for tag in image_ids]

            paginator = self.ecr_client.get_paginator("describe_images")
            image_descriptions = []
            for page in paginator.paginate(**params):
                image_descriptions.extend(page["imageDetails"])
            return image_descriptions
        except ClientError as err:
            logger.error(
                "Couldn't describe images. Here's why %s",
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see DescribeImages in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeRepositories` with an AWS SDK or CLI

The following code examples show how to use `DescribeRepositories`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

CLI

AWS CLI

To describe the repositories in a registry

This example describes the repositories in the default registry for an account.

Command:

```
aws ecr describe-repositories
```

Output:

```
{
    "repositories": [
        {
            "registryId": "012345678910",
            "repositoryName": "ubuntu",
            "repositoryArn": "arn:aws:ecr:us-west-2:012345678910:repository/
ubuntu"
        },
        {
            "registryId": "012345678910",
            "repositoryName": "test",
            "repositoryArn": "arn:aws:ecr:us-west-2:012345678910:repository/test"
        }
    ]
```

```
}
```

- For API details, see DescribeRepositories in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /**
    * Retrieves the repository URI for the specified repository name.
    *
    * @param repoName the name of the repository to retrieve the URI for.
    * @return the repository URI for the specified repository name.
    * @throws EcrException        if there is an error retrieving the repository
information.
    * @throws CompletionException if the asynchronous operation completes
exceptionally.
    */
   public void getRepositoryURI(String repoName) {
       DescribeRepositoriesRequest request =
DescribeRepositoriesRequest.builder()
           .repositoryNames(repoName)
           .build();

       CompletableFuture<DescribeRepositoriesResponse> response =
getAsyncClient().describeRepositories(request);
       response.whenComplete((describeRepositoriesResponse, ex) -> {
           if (ex != null) {
               Throwable cause = ex.getCause();
               if (cause instanceof InterruptedException) {
                   Thread.currentThread().interrupt();
                   String errorMessage = "Thread interrupted while waiting for
asynchronous operation: " + cause.getMessage();
                   throw new RuntimeException(errorMessage, cause);
               } else if (cause instanceof EcrException) {
                   throw (EcrException) cause;
```

```
                } else {
                    String errorMessage = "Unexpected error: " +
cause.getMessage();
                    throw new RuntimeException(errorMessage, cause);
                }
            } else {
                if (describeRepositoriesResponse != null) {
                    if (!describeRepositoriesResponse.repositories().isEmpty()) {
                        String repositoryUri =
describeRepositoriesResponse.repositories().get(0).repositoryUri();
                        System.out.println("Repository URI found: " +
repositoryUri);
                    } else {
                        System.out.println("No repositories found for the given
name.");
                    }
                } else {
                    System.err.println("No response received from
describeRepositories.");
                }
            }
        });
        response.join();
    }
```

- For API details, see DescribeRepositories in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/**
 * Retrieves the repository URI for the specified repository name.
 *
```

```
     * @param repoName the name of the repository to retrieve the URI for.
     * @return the repository URI for the specified repository name.
     */
    suspend fun getRepositoryURI(repoName: String?): String? {
        require(!(repoName == null || repoName.isEmpty())) { "Repository name
 cannot be null or empty" }
        val request =
            DescribeRepositoriesRequest {
                repositoryNames = listOf(repoName)
            }

        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val describeRepositoriesResponse =
 ecrClient.describeRepositories(request)
            if (!describeRepositoriesResponse.repositories?.isEmpty()!!) {
                return
 describeRepositoriesResponse?.repositories?.get(0)?.repositoryUri
            } else {
                println("No repositories found for the given name.")
                return ""
            }
        }
    }
```

- For API details, see [DescribeRepositories](#) in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class ECRWrapper:
    def __init__(self, ecr_client: client):
        self.ecr_client = ecr_client

    @classmethod
```

```
    def from_client(cls) -> "ECRWrapper":
        """
        Creates a ECRWrapper instance with a default Amazon ECR client.

        :return: An instance of ECRWrapper initialized with the default Amazon
ECR client.
        """
        ecr_client = boto3.client("ecr")
        return cls(ecr_client)


    def describe_repositories(self, repository_names: list[str]) -> list[dict]:
        """
        Describes ECR repositories.

        :param repository_names: The names of the repositories to describe.
        :return: The list of repository descriptions.
        """
        try:
            response = self.ecr_client.describe_repositories(
                repositoryNames=repository_names
            )
            return response["repositories"]
        except ClientError as err:
            logger.error(
                "Couldn't describe repositories. Here's why %s",
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see DescribeRepositories in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
async fn show_repos(client: &aws_sdk_ecr::Client) -> Result<(),
 aws_sdk_ecr::Error> {
    let rsp = client.describe_repositories().send().await?;

    let repos = rsp.repositories();

    println!("Found {} repositories:", repos.len());

    for repo in repos {
        println!("  ARN:  {}", repo.repository_arn().unwrap());
        println!("  Name: {}", repo.repository_name().unwrap());
    }

    Ok(())
}
```

- For API details, see DescribeRepositories in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetAuthorizationToken` with an AWS SDK or CLI

The following code examples show how to use `GetAuthorizationToken`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

CLI

AWS CLI

To get an authorization token for your default registry

The following `get-authorization-token` example command gets an authorization token for your default registry.

```
aws ecr get-authorization-token
```

Output:

```
{
    "authorizationData": [
        {
            "authorizationToken": "QVdTOkN...",
            "expiresAt": 1448875853.241,
            "proxyEndpoint": "https://123456789012.dkr.ecr.us-
west-2.amazonaws.com"
        }
    ]
}
```

- For API details, see GetAuthorizationToken in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Retrieves the authorization token for Amazon Elastic Container Registry
(ECR).
     * This method makes an asynchronous call to the ECR client to retrieve the
authorization token.
     * If the operation is successful, the method prints the token to the
console.
     * If an exception occurs, the method handles the exception and prints the
error message.
     *
     * @throws EcrException     if there is an error retrieving the authorization
token from ECR.
     * @throws RuntimeException if there is an unexpected error during the
operation.
```

```
     */
    public void getAuthToken() {
        CompletableFuture<GetAuthorizationTokenResponse> response =
getAsyncClient().getAuthorizationToken();
        response.whenComplete((authorizationTokenResponse, ex) -> {
            if (authorizationTokenResponse != null) {
                AuthorizationData authorizationData =
authorizationTokenResponse.authorizationData().get(0);
                String token = authorizationData.authorizationToken();
                if (!token.isEmpty()) {
                    System.out.println("The token was successfully retrieved.");
                }
            } else {
                if (ex.getCause() instanceof EcrException) {
                    throw (EcrException) ex.getCause();
                } else {
                    String errorMessage = "Unexpected error occurred: " +
ex.getMessage();
                    throw new RuntimeException(errorMessage, ex); // Rethrow the
exception
                }
            }
        });
        response.join();
    }
```

- For API details, see GetAuthorizationToken in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
```

```
     * Retrieves the authorization token for Amazon Elastic Container Registry
 (ECR).
     *
     */
    suspend fun getAuthToken() {
        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            // Retrieve the authorization token for ECR.
            val response = ecrClient.getAuthorizationToken()
            val authorizationData = response.authorizationData?.get(0)
            val token = authorizationData?.authorizationToken
            if (token != null) {
                println("The token was successfully retrieved.")
            }
        }
    }
```

- For API details, see GetAuthorizationToken in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class ECRWrapper:
    def __init__(self, ecr_client: client):
        self.ecr_client = ecr_client

    @classmethod
    def from_client(cls) -> "ECRWrapper":
        """
        Creates a ECRWrapper instance with a default Amazon ECR client.

        :return: An instance of ECRWrapper initialized with the default Amazon
 ECR client.
        """
        ecr_client = boto3.client("ecr")
```

```
        return cls(ecr_client)


    def get_authorization_token(self) -> str:
        """
        Gets an authorization token for an ECR repository.

        :return: The authorization token.
        """
        try:
            response = self.ecr_client.get_authorization_token()
            return response["authorizationData"][0]["authorizationToken"]
        except ClientError as err:
            logger.error(
                "Couldn't get authorization token. Here's why %s",
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see GetAuthorizationToken in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetRepositoryPolicy` with an AWS SDK or CLI

The following code examples show how to use `GetRepositoryPolicy`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

CLI

AWS CLI

To retrieve the repository policy for a repository

The following `get-repository-policy` example displays details about the repository policy for the `cluster-autoscaler` repository.

```
aws ecr get-repository-policy \
    --repository-name cluster-autoscaler
```

Output:

```
{
    "registryId": "012345678910",
    "repositoryName": "cluster-autoscaler",
    "policyText": "{\n  \"Version\" : \"2008-10-17\",\n  \"Statement\" :
 [ {\n    \"Sid\" : \"allow public pull\",\n    \"Effect\" : \"Allow\",\n
 \"Principal\" : \"*\",\n    \"Action\" : [ \"ecr:BatchCheckLayerAvailability\",
 \"ecr:BatchGetImage\", \"ecr:GetDownloadUrlForLayer\" ]\n  } ]\n}"
}
```

- For API details, see GetRepositoryPolicy in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Gets the repository policy for the specified repository.
     *
     * @param repoName the name of the repository.
     * @throws EcrException if an AWS error occurs while getting the repository
policy.
     */
    public String getRepoPolicy(String repoName) {
        if (repoName == null || repoName.isEmpty()) {
            throw new IllegalArgumentException("Repository name cannot be null or
empty");
```

```
        }

        GetRepositoryPolicyRequest getRepositoryPolicyRequest =
GetRepositoryPolicyRequest.builder()
            .repositoryName(repoName)
            .build();

        CompletableFuture<GetRepositoryPolicyResponse> response =
getAsyncClient().getRepositoryPolicy(getRepositoryPolicyRequest);
        response.whenComplete((resp, ex) -> {
            if (resp != null) {
                System.out.println("Repository policy retrieved successfully.");
            } else {
                if (ex.getCause() instanceof EcrException) {
                    throw (EcrException) ex.getCause();
                } else {
                    String errorMessage = "Unexpected error occurred: " +
ex.getMessage();
                    throw new RuntimeException(errorMessage, ex);
                }
            }
        });

        GetRepositoryPolicyResponse result = response.join();
        return result != null ? result.policyText() : null;
    }
```

- For API details, see [GetRepositoryPolicy](#) in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/**
```

```
     * Gets the repository policy for the specified repository.
     *
     * @param repoName the name of the repository.
     */
    suspend fun getRepoPolicy(repoName: String?): String? {
        require(!(repoName == null || repoName.isEmpty())) { "Repository name
 cannot be null or empty" }

        // Create the request
        val getRepositoryPolicyRequest =
            GetRepositoryPolicyRequest {
                repositoryName = repoName
            }
        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val response =
ecrClient.getRepositoryPolicy(getRepositoryPolicyRequest)
            val responseText = response.policyText
            return responseText
        }
    }
```

- For API details, see GetRepositoryPolicy in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class ECRWrapper:
    def __init__(self, ecr_client: client):
        self.ecr_client = ecr_client

    @classmethod
    def from_client(cls) -> "ECRWrapper":
        """
        Creates a ECRWrapper instance with a default Amazon ECR client.
```

```
        :return: An instance of ECRWrapper initialized with the default Amazon
ECR client.
        """
        ecr_client = boto3.client("ecr")
        return cls(ecr_client)


    def get_repository_policy(self, repository_name: str) -> str:
        """
        Gets the policy for an ECR repository.

        :param repository_name: The name of the repository to get the policy for.
        :return: The policy text.
        """
        try:
            response = self.ecr_client.get_repository_policy(
                repositoryName=repository_name
            )
            return response["policyText"]
        except ClientError as err:
            if err.response["Error"]["Code"] ==
"RepositoryPolicyNotFoundException":
                logger.error("Repository does not exist. %s.", repository_name)
                raise
            else:
                logger.error(
                    "Couldn't get repository policy for repository %s. Here's why
%s",
                    repository_name,
                    err.response["Error"]["Message"],
                )
                raise
```

- For API details, see GetRepositoryPolicy in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListImages` with an AWS SDK or CLI

The following code examples show how to use `ListImages`.

CLI

AWS CLI

To list the images in a repository

The following `list-images` example displays a list of the images in the `cluster-autoscaler` repository.

```
aws ecr list-images \
    --repository-name cluster-autoscaler
```

Output:

```
{
    "imageIds": [
        {
            "imageDigest":
 "sha256:99c6fb4377e9a420a1eb3b410a951c9f464eff3b7dbc76c65e434e39b94b6570",
            "imageTag": "v1.13.8"
        },
        {
            "imageDigest":
 "sha256:99c6fb4377e9a420a1eb3b410a951c9f464eff3b7dbc76c65e434e39b94b6570",
            "imageTag": "v1.13.7"
        },
        {
            "imageDigest":
 "sha256:4a1c6567c38904384ebc64e35b7eeddd8451110c299e3368d2210066487d97e5",
            "imageTag": "v1.13.6"
        }
    ]
}
```

- For API details, see ListImages in *AWS CLI Command Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
async fn show_images(
    client: &aws_sdk_ecr::Client,
    repository: &str,
) -> Result<(), aws_sdk_ecr::Error> {
    let rsp = client
        .list_images()
        .repository_name(repository)
        .send()
        .await?;

    let images = rsp.image_ids();

    println!("found {} images", images.len());

    for image in images {
        println!(
            "image: {}:{}",
            image.image_tag().unwrap(),
            image.image_digest().unwrap()
        );
    }

    Ok(())
}
```

- For API details, see ListImages in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use PushImageCmd with an AWS SDK

The following code examples show how to use PushImageCmd.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /**
    * Pushes a Docker image to an Amazon Elastic Container Registry (ECR)
repository.
    *
    * @param repoName  the name of the ECR repository to push the image to.
    * @param imageName the name of the Docker image.
    */
   public void pushDockerImage(String repoName, String imageName) {
       System.out.println("Pushing " + imageName + " to Amazon ECR will take a
few seconds.");
       CompletableFuture<AuthConfig> authResponseFuture =
getAsyncClient().getAuthorizationToken()
           .thenApply(response -> {
               String token =
response.authorizationData().get(0).authorizationToken();
               String decodedToken = new
String(Base64.getDecoder().decode(token));
               String password = decodedToken.substring(4);

               DescribeRepositoriesResponse descrRepoResponse =
getAsyncClient().describeRepositories(b -> b.repositoryNames(repoName)).join();
               Repository repoData =
descrRepoResponse.repositories().stream().filter(r ->
r.repositoryName().equals(repoName)).findFirst().orElse(null);
               assert repoData != null;
               String registryURL = repoData.repositoryUri().split("/")[0];

               AuthConfig authConfig = new AuthConfig()
```

```
                    .withUsername("AWS")
                    .withPassword(password)
                    .withRegistryAddress(registryURL);
                return authConfig;
            })
            .thenCompose(authConfig -> {
                DescribeRepositoriesResponse descrRepoResponse =
 getAsyncClient().describeRepositories(b -> b.repositoryNames(repoName)).join();
                Repository repoData =
 descrRepoResponse.repositories().stream().filter(r ->
 r.repositoryName().equals(repoName)).findFirst().orElse(null);
                getDockerClient().tagImageCmd(imageName + ":latest",
 repoData.repositoryUri() + ":latest", imageName).exec();
                try {

 getDockerClient().pushImageCmd(repoData.repositoryUri()).withTag("echo-
text").withAuthConfig(authConfig).start().awaitCompletion();
                    System.out.println("The " + imageName + " was pushed to
 ECR");

                } catch (InterruptedException e) {
                    throw (RuntimeException) e.getCause();
                }
                return CompletableFuture.completedFuture(authConfig);
            });

        authResponseFuture.join();
    }
```

- For API details, see PushImageCmd in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /**
    * Pushes a Docker image to an Amazon Elastic Container Registry (ECR)
repository.
    *
    * @param repoName the name of the ECR repository to push the image to.
    * @param imageName the name of the Docker image.
    */
   suspend fun pushDockerImage(
       repoName: String,
       imageName: String,
   ) {
       println("Pushing $imageName to $repoName will take a few seconds")
       val authConfig = getAuthConfig(repoName)

       EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
           val desRequest =
               DescribeRepositoriesRequest {
                   repositoryNames = listOf(repoName)
               }

           val describeRepoResponse = ecrClient.describeRepositories(desRequest)
           val repoData =
               describeRepoResponse.repositories?.firstOrNull
{ it.repositoryName == repoName }
                   ?: throw RuntimeException("Repository not found: $repoName")

           val tagImageCmd = getDockerClient()?.tagImageCmd("$imageName",
"${repoData.repositoryUri}", imageName)
           if (tagImageCmd != null) {
               tagImageCmd.exec()
           }
           val pushImageCmd =
               repoData.repositoryUri?.let {
                   dockerClient?.pushImageCmd(it)
                       // ?.withTag("latest")
                       ?.withAuthConfig(authConfig)
               }

           try {
               if (pushImageCmd != null) {
                   pushImageCmd.start().awaitCompletion()
               }
```

```
                println("The $imageName was pushed to Amazon ECR")
            } catch (e: IOException) {
                throw RuntimeException(e)
            }
        }
    }
```

- For API details, see PushImageCmd in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `PutLifeCyclePolicy` with an AWS SDK or CLI

The following code examples show how to use `PutLifeCyclePolicy`.

CLI

AWS CLI

To create a lifecycle policy

The following `put-lifecycle-policy` example creates a lifecycle policy for the specified repository in the default registry for an account.

```
aws ecr put-lifecycle-policy \
    --repository-name "project-a/amazon-ecs-sample" \
    --lifecycle-policy-text "file://policy.json"
```

Contents of `policy.json`:

```
{
   "rules": [
       {
           "rulePriority": 1,
           "description": "Expire images older than 14 days",
           "selection": {
               "tagStatus": "untagged",
               "countType": "sinceImagePushed",
```

```
                "countUnit": "days",
                "countNumber": 14
            },
            "action": {
                "type": "expire"
            }
        }
    ]
}
```

Output:

```
{
   "registryId": "<aws_account_id>",
   "repositoryName": "project-a/amazon-ecs-sample",
   "lifecyclePolicyText": "{\"rules\":[{\"rulePriority\":1,\"description\":
\"Expire images older than 14 days\",\"selection\":{\"tagStatus\":\"untagged\",
\"countType\":\"sinceImagePushed\",\"countUnit\":\"days\",\"countNumber\":14},
\"action\":{\"type\":\"expire\"}}]}"
}
```

For more information, see Lifecycle Policies in the *Amazon ECR User Guide*.

- For API details, see PutLifeCyclePolicy in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class ECRWrapper:
    def __init__(self, ecr_client: client):
        self.ecr_client = ecr_client

    @classmethod
    def from_client(cls) -> "ECRWrapper":
```

```
        """
        Creates a ECRWrapper instance with a default Amazon ECR client.

        :return: An instance of ECRWrapper initialized with the default Amazon
ECR client.
        """
        ecr_client = boto3.client("ecr")
        return cls(ecr_client)


    def put_lifecycle_policy(self, repository_name: str, lifecycle_policy_text:
str):
        """
        Puts a lifecycle policy for an ECR repository.

        :param repository_name: The name of the repository to put the lifecycle
policy for.
        :param lifecycle_policy_text: The lifecycle policy text to put.
        """
        try:
            self.ecr_client.put_lifecycle_policy(
                repositoryName=repository_name,
                lifecyclePolicyText=lifecycle_policy_text,
            )
            print(f"Put lifecycle policy for repository {repository_name}.")
        except ClientError as err:
            logger.error(
                "Couldn't put lifecycle policy for repository %s. Here's why %s",
                repository_name,
                err.response["Error"]["Message"],
            )
            raise
```

Example that puts an expiration date policy.

```
    def put_expiration_policy(self):
        """
        Puts an expiration policy on the ECR repository.
        """
        policy_json = {
            "rules": [
```

```
                {
                    "rulePriority": 1,
                    "description": "Expire images older than 14 days",
                    "selection": {
                        "tagStatus": "any",
                        "countType": "sinceImagePushed",
                        "countUnit": "days",
                        "countNumber": 14,
                    },
                    "action": {"type": "expire"},
                }
            ]
        }

        self.ecr_wrapper.put_lifecycle_policy(
            self.repository_name, json.dumps(policy_json)
        )
```

- For API details, see PutLifeCyclePolicy in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `SetRepositoryPolicy` with an AWS SDK or CLI

The following code examples show how to use `SetRepositoryPolicy`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

CLI

AWS CLI

To set the repository policy for a repository

The following `set-repository-policy` example attaches a repository policy contained in a file to the `cluster-autoscaler` repository.

```
aws ecr set-repository-policy \
    --repository-name cluster-autoscaler \
    --policy-text file://my-policy.json
```

Contents of `my-policy.json`:

```
{
    "Version" : "2008-10-17",
    "Statement" : [
        {
            "Sid" : "allow public pull",
            "Effect" : "Allow",
            "Principal" : "*",
            "Action" : [
                "ecr:BatchCheckLayerAvailability",
                "ecr:BatchGetImage",
                "ecr:GetDownloadUrlForLayer"
            ]
        }
    ]
}
```

Output:

```
{
    "registryId": "012345678910",
    "repositoryName": "cluster-autoscaler",
    "policyText": "{\n  \"Version\" : \"2008-10-17\",\n  \"Statement\" :
 [ {\n    \"Sid\" : \"allow public pull\",\n    \"Effect\" : \"Allow\",\n
 \"Principal\" : \"*\",\n    \"Action\" : [ \"ecr:BatchCheckLayerAvailability\",
 \"ecr:BatchGetImage\", \"ecr:GetDownloadUrlForLayer\" ]\n  } ]\n}"
}
```

- For API details, see SetRepositoryPolicy in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /**
    * Sets the repository policy for the specified ECR repository.
    *
    * @param repoName the name of the ECR repository.
    * @param iamRole  the IAM role to be granted access to the repository.
    * @throws RepositoryPolicyNotFoundException if the repository policy does
not exist.
    * @throws EcrException                      if there is an unexpected error
setting the repository policy.
    */
   public void setRepoPolicy(String repoName, String iamRole) {
       /*
         This example policy document grants the specified AWS principal the
permission to perform the
         `ecr:BatchGetImage` action. This policy is designed to allow the
specified principal
         to retrieve Docker images from the ECR repository.
        */
       String policyDocumentTemplate = """
           {
            "Version" : "2012-10-17",
            "Statement" : [ {
              "Sid" : "new statement",
              "Effect" : "Allow",
              "Principal" : {
                "AWS" : "%s"
              },
              "Action" : "ecr:BatchGetImage"
            } ]
          }
           """;
```

```
        String policyDocument = String.format(policyDocumentTemplate, iamRole);
        SetRepositoryPolicyRequest setRepositoryPolicyRequest =
SetRepositoryPolicyRequest.builder()
            .repositoryName(repoName)
            .policyText(policyDocument)
            .build();

        CompletableFuture<SetRepositoryPolicyResponse> response =
getAsyncClient().setRepositoryPolicy(setRepositoryPolicyRequest);
        response.whenComplete((resp, ex) -> {
            if (resp != null) {
                System.out.println("Repository policy set successfully.");
            } else {
                Throwable cause = ex.getCause();
                if (cause instanceof RepositoryPolicyNotFoundException) {
                    throw (RepositoryPolicyNotFoundException) cause;
                } else if (cause instanceof EcrException) {
                    throw (EcrException) cause;
                } else {
                    String errorMessage = "Unexpected error: " +
cause.getMessage();
                    throw new RuntimeException(errorMessage, cause);
                }
            }
        });
        response.join();
    }
```

- For API details, see SetRepositoryPolicy in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Sets the repository policy for the specified ECR repository.
     *
     * @param repoName the name of the ECR repository.
     * @param iamRole the IAM role to be granted access to the repository.
     */
    suspend fun setRepoPolicy(
        repoName: String?,
        iamRole: String?,
    ) {
        val policyDocumentTemplate =
            """
             {
              "Version" : "2012-10-17",
              "Statement" : [ {
                "Sid" : "new statement",
                "Effect" : "Allow",
                "Principal" : {
                  "AWS" : "$iamRole"
                },
                "Action" : "ecr:BatchGetImage"
              } ]
            }

            """.trimIndent()
        val setRepositoryPolicyRequest =
            SetRepositoryPolicyRequest {
                repositoryName = repoName
                policyText = policyDocumentTemplate
            }

        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val response =
ecrClient.setRepositoryPolicy(setRepositoryPolicyRequest)
            if (response != null) {
                println("Repository policy set successfully.")
            }
        }
    }
```

- For API details, see SetRepositoryPolicy in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class ECRWrapper:
    def __init__(self, ecr_client: client):
        self.ecr_client = ecr_client

    @classmethod
    def from_client(cls) -> "ECRWrapper":
        """
        Creates a ECRWrapper instance with a default Amazon ECR client.

        :return: An instance of ECRWrapper initialized with the default Amazon
 ECR client.
        """
        ecr_client = boto3.client("ecr")
        return cls(ecr_client)


    def set_repository_policy(self, repository_name: str, policy_text: str):
        """
        Sets the policy for an ECR repository.

        :param repository_name: The name of the repository to set the policy for.
        :param policy_text: The policy text to set.
        """
        try:
            self.ecr_client.set_repository_policy(
                repositoryName=repository_name, policyText=policy_text
            )
            print(f"Set repository policy for repository {repository_name}.")
        except ClientError as err:
            if err.response["Error"]["Code"] ==
 "RepositoryPolicyNotFoundException":
                logger.error("Repository does not exist. %s.", repository_name)
```

```
                raise
            else:
                logger.error(
                    "Couldn't set repository policy for repository %s. Here's why
%s",
                    repository_name,
                    err.response["Error"]["Message"],
                )
                raise
```

Example that grants an IAM role download access.

```
    def grant_role_download_access(self, role_arn: str):
        """
        Grants the specified role access to download images from the ECR
repository.

        :param role_arn: The ARN of the role to grant access to.
        """
        policy_json = {
            "Version": "2008-10-17",
            "Statement": [
                {
                    "Sid": "AllowDownload",
                    "Effect": "Allow",
                    "Principal": {"AWS": role_arn},
                    "Action": ["ecr:BatchGetImage"],
                }
            ],
        }

        self.ecr_wrapper.set_repository_policy(
            self.repository_name, json.dumps(policy_json)
        )
```

- For API details, see SetRepositoryPolicy in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `StartLifecyclePolicyPreview` with an AWS SDK or CLI

The following code examples show how to use `StartLifecyclePolicyPreview`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

CLI

AWS CLI

To create a lifecycle policy preview

The following `start-lifecycle-policy-preview` example creates a lifecycle policy preview defined by a JSON file for the specified repository.

```
aws ecr start-lifecycle-policy-preview \
    --repository-name "project-a/amazon-ecs-sample" \
    --lifecycle-policy-text "file://policy.json"
```

Contents of `policy.json`:

```
{
   "rules": [
       {
           "rulePriority": 1,
           "description": "Expire images older than 14 days",
           "selection": {
               "tagStatus": "untagged",
               "countType": "sinceImagePushed",
               "countUnit": "days",
               "countNumber": 14
           },
           "action": {
               "type": "expire"
           }
```

```
        }
    ]
}
```

Output:

```
{
    "registryId": "012345678910",
    "repositoryName": "project-a/amazon-ecs-sample",
    "lifecyclePolicyText": "{\n    \"rules\": [\n        {\n
 \"rulePriority\": 1,\n           \"description\": \"Expire images older than 14
 days\",\n           \"selection\": {\n               \"tagStatus\": \"untagged
\",\n               \"countType\": \"sinceImagePushed\",\n
 \"countUnit\": \"days\",\n               \"countNumber\": 14\n           },\n
          \"action\": {\n               \"type\": \"expire\"\n           }\n
      }\n    ]\n}\n",
    "status": "IN_PROGRESS"
}
```

- For API details, see StartLifecyclePolicyPreview in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Verifies the existence of an image in an Amazon Elastic Container Registry
 (Amazon ECR) repository asynchronously.
     *
     * @param repositoryName The name of the Amazon ECR repository.
     * @param imageTag       The tag of the image to verify.
     * @throws EcrException             if there is an error retrieving the image
 information from Amazon ECR.
     * @throws CompletionException      if the asynchronous operation completes
 exceptionally.
```

```
     */
    public void verifyImage(String repositoryName, String imageTag) {
        DescribeImagesRequest request = DescribeImagesRequest.builder()
            .repositoryName(repositoryName)
            .imageIds(ImageIdentifier.builder().imageTag(imageTag).build())
            .build();

        CompletableFuture<DescribeImagesResponse> response =
 getAsyncClient().describeImages(request);
        response.whenComplete((describeImagesResponse, ex) -> {
            if (ex != null) {
                if (ex instanceof CompletionException) {
                    Throwable cause = ex.getCause();
                    if (cause instanceof EcrException) {
                        throw (EcrException) cause;
                    } else {
                        throw new RuntimeException("Unexpected error: " +
 cause.getMessage(), cause);
                    }
                } else {
                    throw new RuntimeException("Unexpected error: " +
 ex.getCause());
                }
            } else if (describeImagesResponse != null &&
describeImagesResponse.imageDetails().isEmpty()) {
                System.out.println("Image is present in the repository.");
            } else {
                System.out.println("Image is not present in the repository.");
            }
        });

        // Wait for the CompletableFuture to complete.
        response.join();
    }
```

- For API details, see StartLifecyclePolicyPreview in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Verifies the existence of an image in an Amazon Elastic Container Registry
 (Amazon ECR) repository asynchronously.
     *
     * @param repositoryName The name of the Amazon ECR repository.
     * @param imageTag       The tag of the image to verify.
     */
    suspend fun verifyImage(
        repoName: String?,
        imageTagVal: String?,
    ) {
        require(!(repoName == null || repoName.isEmpty())) { "Repository name
 cannot be null or empty" }
        require(!(imageTagVal == null || imageTagVal.isEmpty())) { "Image tag
 cannot be null or empty" }

        val imageId =
            ImageIdentifier {
                imageTag = imageTagVal
            }
        val request =
            DescribeImagesRequest {
                repositoryName = repoName
                imageIds = listOf(imageId)
            }

        EcrClient.fromEnvironment { region = "us-east-1" }.use { ecrClient ->
            val describeImagesResponse = ecrClient.describeImages(request)
            if (describeImagesResponse != null && !
describeImagesResponse.imageDetails?.isEmpty()!!) {
                println("Image is present in the repository.")
```

```
            } else {
                println("Image is not present in the repository.")
            }
        }
    }
```

- For API details, see StartLifecyclePolicyPreview in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon ECR with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Amazon ECR service quotas

The following table provides the default service quotas for Amazon Elastic Container Registry (Amazon ECR).

Name	Default	Adjustable	Description
Basic image scans per 24 hours	Each supported Region: 100,000	No	The maximum number of images that can be scanned within a 24 hour period in the current account and region using basic scanning. This limit includes both scan on push and manual scans.
Filters per rule in a replication configuration	Each supported Region: 100	No	The maximum number of filters per rule in a replication configuration.
Images per repository	Each supported Region: 100,000	Yes	The maximum number of images per repository.
Layer parts	Each supported Region: 4,200	No	The maximum number of layer parts. This is only applicable if you are using Amazon ECR API actions directly to initiate multipart uploads for image push operations.
Lifecycle policy length	Each supported Region: 30,720	No	The maximum number of characters in a lifecycle policy.
Maximum layer part size	Each supported Region: 10	No	The maximum size (MiB) of a layer part. This is

Name	Default	Adjustable	Description
			only applicable if you are using Amazon ECR API actions directly to initiate multipart uploads for image push operations.
Maximum layer size	Each supported Region: 52,000	No	The maximum size (MiB) of a layer.
Minimum layer part size	Each supported Region: 5	No	The minimum size (MiB) of a layer part. This is only applicable if you are using Amazon ECR API actions directly to initiate multipart uploads for image push operations.
Pull through cache rules per registry	Each supported Region: 50	No	The maximum number of pull-through cache rules.
Rate of BatchCheckLayerAvailability requests	Each supported Region: 1,000 per second	Yes	The maximum number of BatchCheckLayerAvailability requests that you can make per second in the current Region. When an image is pushed to a repository, each image layer is checked to verify if it has been uploaded before. If it has been uploaded, then the image layer is skipped.

Name	Default	Adjustable	Description
Rate of BatchGetImage requests	Each supported Region: 2,000 per second	Yes	The maximum number of BatchGetImage requests that you can make per second in the current Region. When an image is pulled, the BatchGetImage API is called once to retrieve the image manifest. If you request a quota increase for this API, review your GetDownloadUrlForLayer usage as well.
Rate of CompleteLayerUpload requests	Each supported Region: 100 per second	Yes	The maximum number of CompleteLayerUpload requests that you can make per second in the current Region. When an image is pushed, the CompleteLayerUpload API is called once per each new image layer to verify that the upload has completed.
Rate of GetAuthorizationToken requests	Each supported Region: 500 per second	Yes	The maximum number of GetAuthorizationToken requests that you can make per second in the current Region.

Name	Default	Adjustable	Description
Rate of GetDownloadUrlForLayer requests	Each supported Region: 3,000 per second	Yes	The maximum number of GetDownloadUrlForLayer requests that you can make per second in the current Region. When an image is pulled, the GetDownloadUrlForLayer API is called once per image layer that is not already cached. If you request a quota increase for this API, review your BatchGetImage usage as well.
Rate of InitiateLayerUpload requests	Each supported Region: 100 per second	Yes	The maximum number of InitiateLayerUpload requests that you can make per second in the current Region. When an image is pushed, the InitiateLayerUpload API is called once per image layer that has not already been uploaded. Whether or not an image layer has been uploaded is determined by the BatchCheckLayerAvailability API action.

Name	Default	Adjustable	Description
Rate of PutImage requests	Each supported Region: 10 per second	Yes	The maximum number of PutImage requests that you can make per second in the current Region. When an image is pushed and all new image layers have been uploaded, the PutImage API is called once to create or update the image manifest and the tags associated with the image.
Rate of UploadLayerPart requests	Each supported Region: 500 per second	Yes	The maximum number of UploadLayerPart requests that you can make per second in the current Region. When an image is pushed, each new image layer is uploaded in parts and the UploadLayerPart API is called once per each new image layer part.
Rate of image scans	Each supported Region: 1	No	The maximum number of image scans per image, per 24 hours.
Registered repositories	Each supported Region: 100,000	Yes	The maximum number of repositories that you can create in this account in the current Region.

Name	Default	Adjuste	Description
Rules per lifecycle policy	Each supported Region: 50	No	The maximum number of rules in a lifecycle policy
Rules per replication configuration	Each supported Region: 10	No	The maximum number of rules in a replication configuration.
Tags per image	Each supported Region: 1,000	No	The maximum number of tags per image.
Unique destinations across all rules in a replication configuration	Each supported Region: 25	No	The maximum number of unique destinations across all rules in a replication configuration.

Managing your Amazon ECR service quotas in the AWS Management Console

Amazon ECR has integrated with Service Quotas, an AWS service that enables you to view and manage your quotas from a central location. For more information, see What Is Service Quotas? in the *Service Quotas User Guide*.

Service Quotas makes it easy to look up the value of all Amazon ECR service quotas.

To view Amazon ECR service quotas (AWS Management Console)

1. Open the Service Quotas console at https://console.aws.amazon.com/servicequotas/.
2. In the navigation pane, choose **AWS services**.
3. From the **AWS services** list, search for and select **Amazon Elastic Container Registry (Amazon ECR)**.

   In the **Service quotas** list, you can see the service quota name, applied value (if it is available), AWS default quota, and whether the quota value is adjustable.

4. To view additional information about a service quota, such as the description, choose the quota name.

To request a quota increase, see Requesting a quota increase in the *Service Quotas User Guide*.

Creating a CloudWatch alarm to monitor API usage metrics

Amazon ECR provides CloudWatch usage metrics that correspond to the AWS service quotas for each of the APIs involved with the registry authentication, image push, and image pull actions. In the Service Quotas console, you can visualize your usage on a graph and configure alarms that alert you when your usage approaches a service quota. For more information, see Amazon ECR usage metrics.

Use the following steps to create a CloudWatch alarm based on one of the Amazon ECR API usage metrics.

To create an alarm based on your Amazon ECR usage quotas (AWS Management Console)

1. Open the Service Quotas console at https://console.aws.amazon.com/servicequotas/.
2. In the navigation pane, choose **AWS services**.
3. From the **AWS services** list, search for and select **Amazon Elastic Container Registry (Amazon ECR)**.
4. In the **Service quotas** list, select the Amazon ECR usage quota you want to create an alarm for.
5. In the Amazon CloudWatch Events alarms section, choose **Create**.
6. For **Alarm threshold**, choose the percentage of your applied quota value that you want to set as the alarm value.
7. For **Alarm name**, enter a name for the alarm and then choose **Create**.

Amazon ECR troubleshooting

This chapter helps you find diagnostic information for Amazon ECR, and provides troubleshooting steps for common issues and error messages.

Topics

- Troubleshooting Docker commands and issues when using Amazon ECR
- Troubleshooting Amazon ECR error messages

Troubleshooting Docker commands and issues when using Amazon ECR

In some cases, running a Docker command against Amazon ECR might result in an error message. Some common error messages and potential solutions are explained below.

Topics

- Docker logs do not contain expected error messages
- Error: "Filesystem Verification Failed" or "404: Image Not Found" when pulling an image from an Amazon ECR repository
- Error: "Filesystem Layer Verification Failed" when pulling images from Amazon ECR
- HTTP 403 Errors or "no basic auth credentials" error when pushing to repository

Docker logs do not contain expected error messages

To begin debugging any Docker-related issue, start by turning on Docker debugging output on the Docker daemon running on your host instances. If you are using images pulled from Amazon ECR on Amazon ECS container instances, see Configuring verbose output from the Docker daemon in the *Amazon Elastic Container Service Developer Guide*.

Error: "Filesystem Verification Failed" or "404: Image Not Found" when pulling an image from an Amazon ECR repository

You may receive the error `Filesystem verification failed` when using the **docker pull** command to pull an image from an Amazon ECR repository with Docker 1.9 or above. You may receive the error `404: Image not found` when you are using Docker versions before 1.9.

Some possible reasons and their explanations are given below.

The local disk is full

If the local disk on which you're running **docker pull** is full, then the SHA-1 hash calculated on the local file may be different than the one calculated by Amazon ECR. Check that your local disk has enough remaining free space to store the Docker image you are pulling. You can also delete old images to make room for new ones. Use the **docker images** command to see a list of all locally downloaded Docker images, along with their sizes.

Client cannot connect to the remote repository due to network error

Calls to an Amazon ECR repository require a functioning connection to the internet. Verify your network settings, and verify that other tools and applications can access resources on the internet. If you are running **docker pull** on an Amazon EC2 instance in a private subnet, verify that the subnet has a route to the internet. Use a network address translation (NAT) server or a managed NAT gateway.

Currently, calls to an Amazon ECR repository also require network access through your corporate firewall to Amazon Simple Storage Service (Amazon S3). If your organization uses firewall software or a NAT device that allows service endpoints, ensure that the Amazon S3 service endpoints for your current Region are allowed.

If you are using Docker behind an HTTP proxy, you can configure Docker with the appropriate proxy settings. For more information, see HTTP proxy in the Docker documentation.

Error: "Filesystem Layer Verification Failed" when pulling images from Amazon ECR

You may receive the error `image image-name not found` when pulling images using the **docker pull** command. If you inspect the Docker logs, you may see an error like the following:

```
filesystem layer verification failed for digest sha256:2b96f...
```

This error indicates that one or more of the layers for your image has failed to download. Some possible reasons and their explanations are given below.

You are using an older version of Docker

This error can occur in a small percentage of cases when using a Docker version less than 1.10. Upgrade your Docker client to 1.10 or greater.

Your client has encountered a network or disk error

A full disk or a network issue may prevent one or more layers from downloading, as discussed earlier about the `Filesystem verification failed` message. Follow the recommendations above to ensure that your filesystem is not full, and that you have enabled access to Amazon S3 from within your network.

HTTP 403 Errors or "no basic auth credentials" error when pushing to repository

There are times when you may receive an `HTTP 403 (Forbidden)` error, or the error message `no basic auth credentials` from the **docker push** or **docker pull** commands, even if you have successfully authenticated to Docker using the **aws ecr get-login-password** command. The following are some known causes of this issue:

You have authenticated to a different region

Authentication requests are tied to specific regions, and cannot be used across regions. For example, if you obtain an authorization token from US West (Oregon), you cannot use it to authenticate against your repositories in US East (N. Virginia). To resolve the issue, ensure that you have retrieved an authentication token from the same Region your repository exists in. For more information, see the section called “Registry authentication”.

You have authenticated to push to a repository you don't have permissions for

You do not have the necessary permissions to push to the repository. For more information, see Private repository policies in Amazon ECR.

Your token has expired

The default authorization token expiration period for tokens obtained using the `GetAuthorizationToken` operation is 12 hours.

Bug in `wincred` credential manager

Some versions of Docker for Windows use a credential manager called `wincred`, which does not properly handle the Docker login command produced by **aws ecr get-login-password** (for more information, see `CredsStore` fails with private repositories). You can run the Docker login command that is output, but when you try to push or pull images, those commands fail. You can work around this bug by removing the `https://` scheme from the registry argument in the Docker login command that is output from **aws ecr get-login-password**. An example Docker login command without the HTTPS scheme is shown below.

```
docker login -u AWS -p <password> <aws_account_id>.dkr.ecr.<region>.amazonaws.com
```

Troubleshooting Amazon ECR error messages

In some cases, an API call that you have initiated through the Amazon ECR console or the AWS CLI exits with an error message. Some common error messages and potential solutions are explained below.

HTTP 429: Too Many Requests or ThrottleException

You may receive a `429: Too Many Requests` error or a `ThrottleException` error from one or more Amazon ECR actions or API calls. This indicates that you are calling a single endpoint in Amazon ECR repeatedly over a short interval, and that your requests are getting throttled. Throttling occurs when calls to a single endpoint from a single user exceed a certain threshold over a period of time.

Each API operations in Amazon ECR has a rate throttles associated with it. For example, the throttle for the `GetAuthorizationToken` action is 20 transaction per second (TPS), with up to a 200 TPS burst allowed. In each region, each account receives a bucket that can store up to 200 `GetAuthorizationToken` credits. These credits are replenished at a rate of 20 per second. If your bucket has 200 credits, you could achieve 200 `GetAuthorizationToken` API transactions per second for one second, and then sustain 20 transactions per second indefinitely. For more information on the rate limits for Amazon ECR APIs, see Amazon ECR service quotas.

To handle throttling errors, implement a retry function with incremental backoff into your code. For more information, see Retry behavior in the *AWS SDKs and Tools Reference Guide*. Another option is to request a rate limit increase, which you can do using the Service Quotas console. For more information, see Managing your Amazon ECR service quotas in the AWS Management Console..

HTTP 403: "User [arn] is not authorized to perform [operation]"

You may receive the following error when attempting to perform an action with Amazon ECR:

```
$ aws ecr get-login-password
A client error (AccessDeniedException) occurred when calling the GetAuthorizationToken
 operation:
    User: arn:aws:iam::account-number:user/username is not authorized to perform:
    ecr:GetAuthorizationToken on resource: *
```

This indicates that your user does not have permissions granted to use Amazon ECR, or that those permissions are not set up correctly. In particular, if you are performing actions against an Amazon ECR repository, verify that the user has been granted permissions to access that repository. For more information about creating and verifying permissions for Amazon ECR, see Identity and Access Management for Amazon Elastic Container Registry.

HTTP 404: "Repository Does Not Exist" error

If you specify a Docker Hub repository that does not currently exist, Docker Hub creates it automatically. With Amazon ECR, new repositories must be explicitly created before they can be used. This prevents new repositories from being created accidentally (for example, due to typos), and it also ensures that an appropriate security access policy is explicitly assigned to any new repositories. For more information about creating repositories, see Amazon ECR private repositories.

Error: Cannot perform an interactive login from a non TTY device

If you receive the error `Cannot perform an interactive login from a non TTY device`, the following troubleshooting steps should help.

- Verify that you're using AWS CLI version 2 and that you don't have a conflicting version of AWS CLI version 1 on your system. For more information, see Installing or updating the latest version of the AWS CLI.

- Verify that you've configured your AWS CLI with valid credentials. For more information, see Installing or updating the latest version of the AWS CLI.
- Verify that the syntax of your AWS CLI command is correct.

Using Podman with Amazon ECR

Using Podman with Amazon ECR enables organizations to leverage the security and simplicity of Podman while benefiting from the scalability and reliability of Amazon ECR for container image management. By following the outlined steps and commands, developers and administrators can streamline their container workflows, enhance security, and optimize resource utilization. As containerization continues to gain momentum, using Podman and Amazon ECR provides a robust and flexible solution for managing and deploying containerized applications.

Using Podman to authenticate with Amazon ECR

Before interacting with Amazon ECR using Podman, authentication is required. This can be achieved by running the `aws ecr get-login-password` command to retrieve an authentication token, and then using that token with the `podman login` command to authenticate with Amazon ECR.

```
aws ecr get-login-password --region region | podman login --username AWS --password-
stdin aws_account_id.dkr.ecr.region.amazonaws.com
```

Using the Amazon ECR credential helper with Podman

Amazon ECR provides a Docker credential helper that works with Podman. The credential helper makes it easier to store and use Docker credentials when pushing and pulling images to Amazon ECR. For installation and configuration steps, see Amazon ECR Docker Credential Helper.

Important

Podman only partially supports the docker-creds-helper specification. Podman supports the `credHelpers` keyword in Docker configuration but does not support the `credsStore` keyword.

To use the Amazon ECR credential helper with Podman, configure your Docker configuration file with the `credHelpers` format:

```
{
    "credHelpers": {
        "public.ecr.aws": "ecr-login",
```

```
        "<aws_account_id>.dkr.ecr.<region>.amazonaws.com": "ecr-login"
    }
}
```

The following `credsStore` configuration is not supported by Podman:

```
{
    "credsStore": "ecr-login"
}
```

Note

The Amazon ECR Docker credential helper does not support multi-factor authentication (MFA) currently.

Pulling images from Amazon ECR with Podman

After successful authentication, container images can be pulled from Amazon ECR using the `podman pull` command with the full Amazon ECR repository URI.

```
podman pull aws_account_id.dkr.ecr.region.amazonaws.com/repository_name:tag
```

Running containers for Amazon ECR with Podman

Once the desired image has been pulled, a container can be instantiated using the `podman run` command.

```
podman run -d aws_account_id.dkr.ecr.region.amazonaws.com/repository_name:tag
```

Pushing images to Amazon ECR with Podman

To push a local image to Amazon ECR, the image must first be tagged with the Amazon ECR repository URI using `podman tag`, and then the `podman push` command can be used to upload the image to Amazon ECR.

```
podman
 tag local_image:tag aws_account_id.dkr.ecr.region.amazonaws.com/repository_name:tag
podman push aws_account_id.dkr.ecr.region.amazonaws.com/repository_name:tag
```

Document history

The following table describes the important changes to the documentation since the last release of Amazon ECR. We also update the documentation frequently to address the feedback that you send us.

Change	Description	Date
Updated to include support for image tag immutability exclusion patterns	Amazon ECR has updated updated image tagging abilities to include image tag immutability exclusion patterns while creating and updating repositories. You can now to specify tags that can be updated even when tag immutability is enabled on a repository by defining wildcard patterns (such as, `latest`", `v*.beta`, `dev-*`) to exclude specific tags from immutability rules while maintaining immutability for all other tags. For more information, see Creating an Amazon ECR private repository to store images.	23 July 2025
Updated Enhanced Image Scanning to provide image usage insights	Amazon ECR has updated updated Enhanced Image Scanning abilities to include visibility into how images are used on Amazon EKS and Amazon ECS. For more information, see Scan images for OS and programming language package vulnerabilities in Amazon ECR.	16 June 2025
IPv6 support	Added support for making requests to Amazon ECR registries using both IPv4-only and dual-stack (IPv4 and IPv6) endpoints. For more information, see Making requests to Amazon ECR registries.	30 April 2025
Added Amazon ECR private registry support to pull through cache	Amazon ECR added support for creating pull through cache rules for the Amazon ECR private registry. For more information, see Sync an upstream registry with an Amazon ECR private registry and Amazon ECR service-linked role for pull through cache.	12 March 2025

Change	Description	Date
Added support for setting registry policy scope	Amazon ECR added support for configuring registry policy scope for your private registry. For more information, see Private registry permissions in Amazon ECR and Amazon ECR private registry.	23 December 2024
AmazonEC2ContainerRegistryPullOnly – New policy	Amazon ECR added a new policy that grants pull-only permissions to Amazon ECR.	10 October 2024
Docker/OCI Client-proxied operations in CloudTrail events now point to `ecr.amazonaws.com`	The value `ecr.amazonaws.com` replaces `AWS Internal` in User Agent (`userAgent`) and Source IP Address (`sourceIPAddress`) fields for CloudTrail events associated with Docker/OCI Client endpoints. For examples, see Example: Image pull action and Example: Image push action.	1 July 2024
Added description of new Amazon ECR service-linked role for repository creation templates.	Amazon ECR uses a service-linked role named **AWSServiceRoleForECRTemplate** which gives permission for Amazon ECR to perform actions on your behalf to complete repository creation template actions. For more information, see Amazon ECR service-linked role for repository creation templates.	20 June 2024
Added the `ECRTemplateServiceRolePolicy` service-linked role.	Added the `ECRTemplateServiceRolePolicy` service-linked role. For more information, see `ECRTemplateServiceRolePolicy`	20 June 2024
Added cross-Region and cross-account replication to China Regions.	Amazon ECR added support to China Region for filtering which repositories are replicated. For more information, see Private image replication in Amazon ECR	15 May 2024

Change	Description	Date
Added GitLab container registry to pull through cache rules	Amazon ECR added support for creating pull through cache rules for the GitLab container registry. For more information, see Sync an upstream registry with an Amazon ECR private registry.	8 May 2024
Amazon ECR lifecycle policy update to add support for using wildcards	Amazon ECR added support for wildcards in a lifecycle policy through the use of the `tagPatternList` parameter in a lifecycle policy rule. For more information, see Automate the cleanup of images by using lifecycle policies in Amazon ECR.	18 December 2023
Amazon ECR repository creation templates	Amazon ECR added support for repository creation templates. For more information, see Templates to control repositories created during a pull through cache or replication action.	15 November 2023
Amazon ECR pull through cache added supported for authenticated upstream registries	Amazon ECR added support for using upstream registries that require authentication for your pull through cache rules. For more information, see Sync an upstream registry with an Amazon ECR private registry.	15 November 2023
AWSECRPullThroughCache_ServiceRolePolicy – Update to an existing policy	Amazon ECR added new permissions to the `AWSECRPullThroughCache_ServiceRolePolicy` policy. These permissions allow Amazon ECR to retrieve the encrypted contents of a Secrets Manager secret. This is required when using a pull through cache rule to cache images from an upstream registry that requires authentication.	November 15, 2023
Amazon ECR image signing	Amazon ECR and AWS Signer added support for creating and pushing container image signatures using the Notary client. For more information, see Signing an image stored in an Amazon ECR private repository.	6 June 2023

Change	Description	Date
Added Kubernetes container registry to pull through cache rules	Amazon ECR added support for creating pull through cache rules for the Kubernetes container registry. For more information, see Sync an upstream registry with an Amazon ECR private registry.	1 June 2023
Amazon ECR enhanced scanning duration support	Amazon Inspector added support for setting the duration that your repositories are monitored for when enhanced scanning is enabled. For more information, see Changing the enhanced scanning duration for images in Amazon Inspector.	28 June 2022
Amazon ECR sends repository pull count metrics to Amazon CloudWatch	Amazon ECR sends repository pull count metrics to Amazon CloudWatch. For more information, see Amazon ECR repository metrics.	6 January 2022
Expanded replication support	Amazon ECR added support for filtering which repositories are replicated. For more information, see Private image replication in Amazon ECR.	21 September 2021
AWS managed policies for Amazon ECR	Amazon ECR added documentation of AWS managed policies. For more information, see AWS managed policies for Amazon Elastic Container Registry.	24 June 2021
Cross-Region and cross-account replication	Amazon ECR added support for configuring replicati on settings for your private registry. For more information, see Private registry settings in Amazon ECR.	8 December 2020

Change	Description	Date
OCI artifact support	Amazon ECR added support for pushing and pulling Open Container Initiative (OCI) artifacts. A new parameter `artifactMediaType` was added to the `DescribeImages` API response to indicate the type of artifact. For more information, see Pushing a Helm chart to an Amazon ECR private repository.	24 August 2020
Encryption at rest	Amazon ECR added support for configuring encryption for your repositories using server-side encryption with customer managed keys stored in AWS Key Management Service (AWS KMS). For more information, see Encryption at rest.	29 July 2020
Multi-architecture images	Amazon ECR added support for creating and pushing Docker manifest lists which are used for multi-architecture images. For more information, see Pushing a multi-architecture image to an Amazon ECR private repository.	28 April 2020
Amazon ECR Usage Metrics	Amazon ECR added CloudWatch usage metrics which provides visibility into your account's resource usage. You also have the ability to create CloudWatch alarms from both the CloudWatch and Service Quotas consoles to get alerts when your usage approaches your applied service quota. For more information, see Amazon ECR usage metrics.	28 Feb 2020

Change	Description	Date
Updated Amazon ECR service quotas	Updated the Amazon ECR service quotas to include per-API quotas. For more information, see Amazon ECR service quotas.	19 Feb 2020
Added `get-login-password` command	Added support for **get-login-password**, which provides a simple and secure method for retrieving an authorization token. For more information, see Using an authorization token.	4 Feb 2020
Image Scanning	Added support for image scanning, which helps in identifying software vulnerabilities in your container images. Amazon ECR uses the Common Vulnerabilities and Exposures (CVEs) database from the open source CoreOS Clair project and provides you with a list of scan findings. For more information, see Scan images for software vulnerabilities in Amazon ECR.	24 Oct 2019
VPC Endpoint Policy	Added support for setting an IAM policy on the Amazon ECR interface VPC endpoints. For more information, see Create an endpoint policy for your Amazon ECR VPC endpoints.	26 Sept 2019
Image Tag Mutability	Added support for configuring a repository to be immutable to prevent image tags from being overwritten. For more information, see Preventing image tags from being overwritten in Amazon ECR.	25 July 2019

Change	Description	Date
Interface VPC Endpoints (AWS PrivateLink)	Added support for configuring interface VPC endpoints powered by AWS PrivateLink. This allows you to create a private connection between your VPC and Amazon ECR without requiring access over the internet, through a NAT instance, a VPN connection, or AWS Direct Connect. For more information, see Amazon ECR interface VPC endpoints (AWS PrivateLink).	25 Jan 2019
Resource tagging	Amazon ECR added support for adding metadata tags to your repositories. For more information, see Tagging a private repository in Amazon ECR.	18 Dec 2018
Amazon ECR Name Change	Amazon Elastic Container Registry is renamed (previously Amazon EC2 Container Registry).	21 Nov 2017
Lifecycle Policies	Amazon ECR lifecycle policies enable you to specify the lifecycle management of images in a repository. For more information, see Automate the cleanup of images by using lifecycle policies in Amazon ECR.	11 Oct 2017
Amazon ECR support for Docker image manifest 2, schema 2	Amazon ECR now supports Docker Image Manifest V2 Schema 2 (used with Docker version 1.10 and newer). For more information, see Container image manifest format support in Amazon ECR.	27 Jan 2017
Amazon ECR General Availability	Amazon Elastic Container Registry (Amazon ECR) is a managed AWS Docker registry service that is secure, scalable, and reliable.	21 Dec 2015