How do you implement lifecycle policies in ECR?

Answer:
I implement **ECR lifecycle policies** to automatically delete old or unused images and manage storage cost.

🔷 **Steps:**

- Navigate to the ECR repository in the AWS Console.
- Go to the **“Lifecycle Policy”** tab.
- Define JSON rules such as:

json

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

- Save the policy.

✅ This helps in cleaning up stale images automatically.

What are ECR image scanning capabilities and how do you enable them?

Answer:
ECR supports **image vulnerability scanning** to detect CVEs in container images.

🔷 **Types of Scanning:**

- **Basic Scanning (default)**: Uses **Common Vulnerabilities and Exposures (CVE)** database from Clair.
- **Enhanced Scanning (optional)**: Uses **Amazon Inspector** with more detailed findings.

🔷 **To Enable:**

- When creating the repo, enable **"Scan on push"**.
- Or update an existing repo using:

bash

```
aws ecr put-image-scanning-configuration \
 --repository-name my-repo \
 --image-scanning-configuration scanOnPush=true
```

🔷 View scan results in the AWS Console or via CLI.

What are the IAM permissions required for ECR operations?

Answer:
Key IAM permissions are:

🔶 **For Push/Pull:**

- ecr:GetAuthorizationToken
- ecr:BatchCheckLayerAvailability
- ecr:PutImage

🔶 **For Repo Management:**

- ecr:CreateRepository
- ecr:DeleteRepository
- ecr:PutLifecyclePolicy
- ecr:SetRepositoryPolicy

🔶 **Example Policy Snippet:**

json

```
{
 "Effect": "Allow",
 "Action": [
  "ecr:*"
 ],
 "Resource": "*"
}
```

✅ For cross-account access, use **resource-based policies** on the ECR repo.

How do you implement image tagging strategies in ECR?

Answer:

1. latest Tag

- **Use:** Points to the most recently built image.
- **Best for:** Development environments or non-production use.
- ⚠️ **Caution:** Overwrites frequently—**not recommended for production**.

bash
docker tag my-app my-repo:latest

---

2. Semantic Versioning Tags

- **Format:** vMAJOR.MINOR.PATCH (e.g., v1.0.0)
- **Use:** Helps track versions across releases.
- **Best for:** Production-grade releases.

bash
docker tag my-app my-repo:v1.2.3

---

3. Git-Based Tags

- **Types:**
  - **Commit-SHA:** commit-a1b2c3d
  - **Branch name:** main, feature-login
- **Use:** Trace exactly **which code commit** created the image.
- **Best for:** Debugging and tracking builds in CI/CD.

bash
docker tag my-app my-repo:commit-9f28bc7

---

4. Environment Tags

- **Examples:** dev, qa, staging, prod
- **Use:** Indicate where the image is deployed.
- **Best for:** Promoting builds between environments.

bash
docker tag my-app my-repo:staging

Benefits of a Good Tagging Strategy

- 🔄 Enables **rollback** to previous versions
- 🔍 Improves **traceability** of code → image → deployment
- 📊 Ensures **clear visibility** across teams and environments
- 🔒 Helps **secure production** by avoiding "latest" image usage
- 🤝 Supports **automated promotion** from dev → QA → prod

Best Practices

- Never use latest in production.
- Always tag with **commit SHA** or **build number** in CI.
- Maintain **consistent conventions** across teams.

- Automate tagging logic in your CI/CD scripts.
- Document the tagging convention in your project's README or wiki.

What is ECR image immutability and when would you use it?

Answer:

Image immutability **prevents overwriting** of existing tags once pushed.

🔷 **When to Use:**

- To ensure consistency and **prevent accidental overwrite**.
- Useful in **production pipelines** for auditability and traceability.

🔷 **Enable with:**

bash

```
aws ecr put-image-tag-mutability \
 --repository-name my-repo \
 --image-tag-mutability IMMUTABLE
```

✅ Recommended for production images with fixed tags like v1.0.0.

How do you integrate ECR with CI/CD pipelines?

Answer:

I integrate ECR with tools like **Jenkins, GitHub Actions, GitLab CI, or CodePipeline**.

🔷 **Steps:**

1. **Login to ECR:**

bash

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

2. **Build & Tag Image:**

bash

```
--------
docker build -t my-app .
docker tag my-app <account>.dkr.ecr.<region>.amazonaws.com/my-app:latest
```

3. **Push to ECR:**

bash

```
--------
docker push <account>.dkr.ecr.<region>.amazonaws.com/my-app:latest
```

4. **Deploy:** Use ECS, EKS, or other services pulling from ECR.

✅ In GitHub Actions, I use aws-actions/amazon-ecr-login and in CodePipeline, I configure a build stage with ECR push logic.

What does enabling image immutability in ECR help prevent?

a. Image scan failures
b. IAM misconfigurations
c. Overwriting tags like v1.0.0 once they're pushed
d. Pushing images to S3 instead of ECR

✅ **Correct Answer: c.** Overwriting tags like v1.0.0 once they're pushed

Which of the following is a good ECR image tagging strategy?

a. Use repository names to indicate versions like my-repo-v1.0.0
b. Use the tag latest for all environments
c. Use semantic versioning and commit-SHA for traceability
d. Use IP addresses of build servers as tags

✅ **Correct Answer: c.** Use semantic versioning and commit-SHA for traceability