Question 10: How do you handle security and access control in CodePipeline?

Security is non-negotiable in any CI/CD pipeline, especially in enterprise environments where multiple teams, accounts, and environments are involved. In my DevOps implementations, I place strong focus on **IAM roles**, **least privilege**, and **secure access delegation** across accounts.

---

1. IAM Roles in CodePipeline: Types & Use Cases

🛠️ **a) Service Roles (Execution Roles)**

- These are the roles **assumed by CodePipeline** or **CodeBuild** when performing actions like:
  - Storing artifacts in S3
  - Deploying using CodeDeploy
- We use **fine-grained permissions** to follow **least privilege principle**.

📍 Example: CodePipeline role can access only:

```
json
"Action": [
 "s3:GetObject",
 "codebuild:StartBuild",
 "codedeploy:CreateDeployment"
],
"Resource": [
 "arn:aws:s3:::my-app-artifacts/*",
 "arn:aws:codebuild:region:account:project/my-project"
]
```

---

b) Cross-Account Roles

- When we had **multiple AWS accounts** (e.g., dev, prod, security), we used **cross-account IAM roles** so pipelines in one account can trigger deployments in another securely.

Real Use Case:
Pipeline in **Account A** needed to deploy to ECS in **Account B**.

So we created a **trusted role in Account B** that allowed CodePipeline from Account A to assume it.

Trust Policy in Account B:

json

```
"Statement": [
 {
  "Effect": "Allow",
  "Principal": {
   "AWS": "arn:aws:iam::ACCOUNT-A:role/CodePipelineRole"
  },
  "Action": "sts:AssumeRole"
 }
]
```

In CodePipeline (Account A), we used assume-role action to take on this temporary role for deploying.

---

c) Developer IAM Access Roles

We restricted developers using:

- **View-only IAM policy** (can view pipelines, logs, execution history)
- **Approval-only role** (can only approve/reject manual approval actions)

🔐 This helped us implement **Separation of Duties (SoD)**:

- Dev can't directly deploy to production
- Only release managers can approve

---

2. Additional Security Best Practices

Best Practice	Description
✅ **S3 Encryption**	Artifacts stored in S3 are encrypted using either SSE-S3 or SSE-KMS
✅ **VPC Endpoints**	CodeBuild/CodeDeploy access resources securely over private AWS network
✅ **Secrets Manager**	API tokens, GitHub credentials, DB secrets stored securely and rotated
✅ **CloudTrail Logs**	Every IAM role assumption and pipeline action is logged for audit
✅ **SSM Parameter Store**	Used for non-secret config values (e.g., ENV name, region)

📌 **Example: GitHub Token Handling**

We needed to authenticate CodePipeline with GitHub. Instead of hardcoding credentials, we:

- Stored token in **AWS Secrets Manager**
- Used **Secrets Manager integration in CodeBuild**
- Set up an IAM policy that only allowed **read access to that specific secret**

Takeaway

By using scoped IAM roles, cross-account delegation, and best practices like encryption, secrets management, and VPC isolation, we made our pipelines **secure, auditable, and compliant** — without slowing down development."

Question 11: How do you optimize CodePipeline performance and cost?

As our team scaled CI/CD across multiple microservices, we saw our build times increasing and monthly costs creeping up. That's when we did a full audit and implemented several strategies to **boost performance and reduce cost**, without compromising on quality or security.

Performance Optimization Strategies

1️⃣ Enable Caching in CodeBuild

Caching significantly reduces build times by avoiding redundant work.

Type	Benefit
Docker Layer Caching	Speeds up builds by reusing unchanged layers – **60% faster builds**
NPM or Yarn Caching	Avoids re-downloading packages every time – **~70% improvement**
Maven/Gradle Caching	Reduces time for dependency resolution – **50% gain**

✅ **Example: Enabling NPM cache in buildspec.yml**

yaml

```
cache:
  paths:
    - '/root/.npm/**/*'
```

For Docker image builds:

- Use --cache-from strategy with ECR
- Or use **CodeBuild Docker layer cache** (via privileged: true and local cache settings)

2️⃣ Parallel Execution

We split the pipeline into parallel branches for independent modules like:

- Frontend
- Backend
- Unit Tests
- Integration Tests

🔍 **Result:**

- Reduced pipeline time from **45 minutes to 18 minutes**

📌 Pro Tip: In CodePipeline, use **multiple CodeBuild actions in parallel** under one stage.

💸 Cost Optimization Strategies

1️⃣ Right-size CodeBuild Instance Types

We audited our CodeBuild jobs and selected instance types based on build workload:

Instance Type	Specs	Use Case
build.general1.small	2 vCPU, 3 GB RAM	Lightweight builds, unit tests
build.general1.medium	4 vCPU, 7 GB RAM	Docker builds, integration tests
build.general1.large/2XL	8+ vCPU	ML/long-running builds only if justified

📌 You're billed per second, so even saving **30 seconds per build** adds up when you have 50+ daily builds.

2️⃣ S3 Lifecycle Policies for Artifacts

Artifacts stored in S3 can cost a lot if not cleaned up.

Strategy:

- Set **S3 lifecycle rule**:
  - Move to **infrequent access (IA)** after 7 days
  - **Delete** after 30 days

Example Rule:

json

```
{
 "ID": "ArtifactLifecycle",
 "Prefix": "artifacts/",
 "Status": "Enabled",
 "Transitions": [{
  "Days": 7,
  "StorageClass": "STANDARD_IA"
 }],
 "Expiration": {
  "Days": 30
 }
}
```

📊 Monitoring & Benchmarking

1️⃣ CloudWatch Metrics

Tracked:

- **Average build duration**
- **Cache hit/miss ratio**
- **Stage failure rates**

2️⃣ Cost Alerts (AWS Budgets)

Set:

- Daily and monthly budgets
- Alert thresholds (e.g., notify if CodeBuild > ₹500/day)

3️⃣ Custom Benchmark Dashboard

We used **CloudWatch Dashboards** + **Lambda + DynamoDB** to:

- Track performance improvement per microservice
- Identify build jobs with abnormal duration or failure spikes

---

✅ **Summary of Optimizations**

Area	Optimization	Result
Build Time	Caching, parallel stages	2x–3x faster
Cost	Smaller instance types	~30% monthly savings
Storage	S3 lifecycle	60–70% reduction in artifact costs
Observability	Dashboards & alerts	Proactive cost & time management

---

🧠 **Takeaway**

"CI/CD performance isn't just about speed—it's about **smart execution**. With caching, parallelism, right-sized infra, and tight monitoring, we were able to cut **build time by 60%** and save **thousands per month**, while improving release frequency and confidence."

What financial control was implemented to alert the team about budget overruns?

A. EC2 Reserved Instances
B. IAM policy triggers
C. AWS Budgets with alert thresholds
D. CloudTrail auditing

✅ **Correct Answer:** C