



## Question 1: Can you explain what AWS CodePipeline is and how you've used it in your projects?

Answer:

- **AWS CodePipeline** is a CI/CD service that automates the software release process.
- **Experience:**
  - Used for a microservices architecture with 15+ services.
  - Triggered by GitHub pushes using webhooks.
  - **Pipeline Stages:**
    - **Source Stage:** GitHub webhook.
    - **Build Stage:** CodeBuild for compilation and unit tests.
    - **Test Stage:** Deployed to staging for integration tests.
    - **Deploy Stage:** Manual approval → Production deployment.
  - **Impact:** Reduced deployment time from **4 hours to 45 minutes**, and **deployment errors by 80%**.

---

## Question 2: How do you handle parallel execution in CodePipeline? Can you share a specific use case?

Answer:

- CodePipeline supports **parallel execution within stages** using the same `runOrder`.
- **Use Case:**
  - **E-commerce platform** testing stage used parallel actions:
    - Unit Tests (CodeBuild)
    - Security Scans (3rd party)
    - Code Quality (SonarQube)
    - Performance Tests (Lambda)

- **Result:** Testing time reduced from **30 minutes to 8 minutes**.

---

## Question 3: What are pipeline artifacts and how do you manage them effectively?

**Answer:**

- Artifacts = **files passed between pipeline stages** (code, builds, dependencies).
- **Stored in S3**, managed through .gitignore, caching, and lifecycle policies.
- **Use Case:**
  - Node.js app pipeline:
    - **Source Artifact:** Raw GitHub code
    - **Build Artifact:** Compiled app + node\_modules
    - **Test Artifact:** Reports, coverage
    - **Deploy Artifact:** Final bundle
  - **Optimization:** Reduced artifact size, used caching, applied S3 lifecycle rules.

---

## Question 4: How do you handle pipeline execution modes? What's your experience with SUPERSEDED mode?

In my projects, especially where multiple developers are pushing changes frequently to the same branch, managing how pipeline executions behave became critical. That's where understanding and configuring execution modes helped us maintain stability and avoid deployment chaos.

---

### Understanding the Execution Modes in AWS CodePipeline

**AWS CodePipeline supports three pipeline execution modes:**

Mode	Description	Use Case
SUPERSEDED (default)	Automatically cancels ongoing executions when a newer revision arrives	Best for rapid iteration environments
PARALLEL	Allows multiple executions to run at the same time	Good when executions don't conflict (e.g., deploying to dev environments)
QUEUED	Queues executions and runs them one after another	Best for production pipelines with critical changes

---

### My Experience with SUPERSEDED Mode

"Initially, we used the default SUPERSEDED mode across all environments. It looked fine until we hit a critical situation."

Real Scenario:

- Developer A pushes code to the develop branch → Pipeline starts executing and deploys to Staging.
- Just 5 minutes later, Developer B pushes a hotfix.
- The pipeline cancels A's execution and starts B's.

✅ While this saved time,

❌ it caused confusion:

The QA team didn't realize the first deployment was stopped, and they began testing a partially deployed state.

---

### Solution & Best Practices We Implemented

1. Environment-based Pipeline Splitting
  - We isolated critical environments (like staging and production) into separate pipelines.
  - This prevented unrelated commits from interrupting the same pipeline.
2. Execution Mode Customization
  - For Staging, we switched to QUEUED mode:
    - Ensures that one execution completes fully before the next starts.
  - For Dev or feature branches, we retained SUPERSEDED to save time during fast iterations.
3. Monitoring & Alerts
  - Enabled SNS notifications for Superseded executions.
  - This helped developers and QA know exactly when a pipeline was skipped.

---

### Takeaway

"Choosing the right execution mode based on environment and branch criticality significantly improved our deployment reliability. The key lesson: Don't rely on the default – customize execution modes strategically to match your workflow."

---

#### 1. Use QUEUED Mode

- Benefit: Executes every change in sequence, ensuring no commit is skipped.
- Use Case: Ideal for production pipelines where every commit matters.

#### 2. Use PARALLEL Mode

- Benefit: Runs multiple executions concurrently.
- Use Case: Safe only if your infrastructure supports isolated deployments per execution.

---

### Additional Considerations and Best Practices

#### Version Control Strategy

- Feature Branches: Encourage developers to use feature/\* branches.
- Pull Requests: Merge into develop or main via PRs to trigger pipelines only for reviewed changes.
- GitFlow / Trunk-Based Development: Choose a branching model suited to your pipeline throughput.


**Which of the following actions can be executed in parallel in a CodePipeline stage (same runOrder)?**

- A. Only deployment actions
- B. Only CodeBuild actions
- C. Unit tests, security scans, and performance tests
- D. Only source actions

**Correct:** C. Unit tests, security scans, and performance tests

**What are pipeline artifacts in AWS CodePipeline?**

- A. Test results saved in RDS
- B. Scripts used for deployment
- C. Files passed between pipeline stages
- D. Lambda function logs

 **Correct Answer:** C. Files passed between pipeline stages


**What does SUPERSEDED execution mode in CodePipeline do?**

- A. Runs all executions simultaneously
- B. Cancels ongoing pipeline executions if a new revision arrives
- C. Queues executions for later
- D. Triggers manual approval on each commit

 **Correct Answer:** B. Cancels ongoing pipeline executions if a new revision arrives

**Which execution mode was adopted for staging environments to ensure reliable testing?**

- A. SUPERSEDED
- B. PARALLEL
- C. QUEUED
- D. RANDOMIZED

 **Correct Answer:** C. QUEUED  
(or Sent SNS notifications for superseded executions)

---