

An ecommerce company is receiving reports that its order history page is experiencing delays in reflecting the processing status of orders. The order processing system consists of an AWS Lambda function that uses reserved concurrency. The Lambda function processes order messages from an Amazon Simple Queue Service (Amazon SQS) queue and inserts processed orders into an Amazon DynamoDB table. The DynamoDB table has auto scaling enabled for read and write capacity.

- A. Check the ApproximateAgeOfOldestMessage metric for the SQS queue. Increase the Lambda function concurrency limit.
- B. Check the ApproximateAgeOfOldestMessage metnc for the SQS queue. Configure a redrive policy on the SQS queue.
- C. Check the NumberOfMessagesSent metric for the SQS queue. Increase the SQS queue visibility timeout.
- D. Check the WriteThrottleEvents metric for the DynamoDB table. Increase the maximum write capacity units (WCUs) for the table's scaling policy.
- E. Check the Throttles metric for the Lambda function. Increase the Lambda function Timeout.

To address the delays in the order history page reflecting the processing status of orders, we need to identify and resolve bottlenecks in the system. The setup involves an AWS Lambda function with reserved concurrency pulling messages from an Amazon SQS queue and writing processed orders to an Amazon DynamoDB table with auto-scaling enabled. Delays suggest that either messages are piling up in the queue or the downstream components (Lambda or DynamoDB) aren't keeping up. Let's evaluate the options and determine the two most effective actions.

The ApproximateAgeOfOldestMessage metric in SQS indicates how long the oldest unprocessed message has been waiting in the queue. If this value is high, it means messages aren't being processed quickly enough, pointing to a bottleneck in the Lambda function's ability to consume them. Lambda's reserved concurrency limits how many instances can run simultaneously, so if this limit is too low, it could starve the system of processing power, causing a backlog. Increasing the concurrency limit would allow more Lambda instances to process messages in parallel, reducing the queue delay. This makes option A a strong candidate.

Next, consider the DynamoDB table. Since it has auto-scaling enabled, it should adjust its read and write capacity based on demand. However, if the auto-scaling policy's maximum write capacity units (WCUs) are capped too low, the table might throttle write operations under heavy load, slowing down the Lambda function's ability to insert processed orders. The WriteThrottleEvents metric tracks how often write requests are throttled. If this metric is elevated, it confirms DynamoDB is a bottleneck. Increasing the maximum WCUs in the scaling policy would allow DynamoDB to handle more writes, alleviating delays. This supports option D.

Now, let's assess the other options. Option B suggests checking ApproximateAgeOfOldestMessage and configuring a redrive policy for the SQS queue. A redrive policy moves unprocessable messages to a dead-letter queue (DLQ) after a set number of retries, which is useful for handling poison-pill messages (e.g., malformed orders). However, the problem is about delays, not message failures, and there's no indication of messages failing to process—just that they're slow. A redrive policy wouldn't address the throughput issue, so B is less relevant.

Option C involves checking the NumberOfMessagesSent metric (how many messages are added to the queue) and increasing the SQS visibility timeout. The visibility timeout determines how long a message is hidden from other consumers after being picked up by Lambda. If it's too short, Lambda might not finish processing before the timeout expires, causing messages to reappear in the queue and leading to duplicate

processing. However, this would manifest as duplicate entries or errors, not delays in status updates, and the scenario doesn't suggest this issue. Plus, `NumberOfMessagesSent` reflects input rate, not processing delay, making C a weaker fit.

Option E suggests checking the `Throttles` metric for Lambda (how often invocations are throttled due to hitting the concurrency limit) and increasing the Lambda timeout. If `Throttles` is high, it confirms concurrency is exhausted, aligning with option A's fix of increasing concurrency. However, increasing the timeout (how long Lambda waits before giving up) only helps if individual executions are timing out, which would result in failed orders, not delays in status updates. The scenario implies orders are processed eventually, just slowly, so E's timeout adjustment doesn't directly address the issue.

Given the goal is to resolve delays, the most direct actions target throughput bottlenecks. Option A (increase Lambda concurrency) addresses slow queue consumption if `ApproximateAgeOfOldestMessage` is high, and option D (increase DynamoDB max WCUs) tackles write throttling if `WriteThrottleEvents` is elevated. These two cover the primary components---Lambda and DynamoDB---without overlap, as concurrency affects processing rate while WCUs affect write capacity.

Final Answer: A. Check the `ApproximateAgeOfOldestMessage` metric for the SQS queue. Increase the Lambda function concurrency limit. D. Check the `WriteThrottleEvents` metric for the DynamoDB table. Increase the maximum write capacity units (WCUs) for the table's scaling policy.

To run an application, a DevOps engineer launches an Amazon EC2 instance with public IP addresses in a public subnet. A user data script obtains the application artifacts and installs them on the instances upon launch. A change to the security classification of the application now requires the instances to run with no access to the internet. While the instances launch successfully and show as healthy, the application does not seem to be installed.

Which of the following should successfully install the application while complying with the new rule?

- A. Launch the instances in a public subnet with Elastic IP addresses attached. Once the application is installed and running, run a script to disassociate the Elastic IP addresses afterwards.
- B. Set up a NAT gateway. Deploy the EC2 instances to a private subnet. Update the private subnet's route table to use the NAT gateway as the default route.
- C. Publish the application artifacts to an Amazon S3 bucket and create a VPC endpoint for S3. Assign an IAM instance profile to the EC2 instances so they can read the application artifacts from the S3 bucket.
- D. Create a security group for the application instances and allow only outbound traffic to the artifact repository. Remove the security group rule once the install is complete.

Let's dive into this problem and figure out whether your choice, B, is correct, while explaining why each option works or doesn't in the context of the new security requirement---no internet access for the EC2 instances---while ensuring the application gets installed.

The scenario starts with an EC2 instance in a public subnet, using a public IP and a user data script to fetch application artifacts from the internet during launch. The instances launch and are healthy, but the application isn't installed because the new rule cuts off internet access, breaking the artifact retrieval. We need a solution that lets the instances access the artifacts without exposing them to the public internet, all while keeping the setup functional.

Option A: Launch in a public subnet with Elastic IPs, then disassociate them after install

This approach keeps the instances in a public subnet with Elastic IPs (public IPs) during launch, allowing the user data script to fetch artifacts from the internet. After installation, a script removes the Elastic IPs, theoretically cutting off internet access. However, this has flaws. First, instances in a public subnet with a route to an internet gateway (IGW) retain internet access via their subnet's routing, even without Elastic IPs, unless you modify the route table post-launch---which isn't mentioned. Second, "no access to the internet" implies a stricter requirement than just removing public IPs; it suggests no outbound connectivity at all. This solution relies on temporary internet access, which might violate the intent of the new rule. Plus, coordinating the disassociation after install adds complexity and potential failure points. This doesn't fully comply with the no-internet rule, so A is incorrect.

Option B: Set up a NAT gateway, deploy to a private subnet, route via NAT

Your choice! Here, the instances move to a private subnet (no public IPs, no direct internet ingress), and a NAT gateway is added in a public subnet with an IGW. The private subnet's route table is updated to send outbound traffic (0.0.0.0/0) to the NAT gateway. The NAT allows instances to initiate outbound connections to the internet (e.g., to fetch artifacts) while blocking inbound internet traffic. During launch, the user data script can still pull artifacts because the NAT provides outbound access, and once installed, the instances remain isolated from inbound internet traffic. However, there's a catch: the instances retain outbound internet access via the NAT, which might conflict with a strict interpretation of "no access to the internet." In AWS, private subnets with a NAT are common for "no public internet" setups, but if the requirement means absolutely no internet connectivity (inbound or outbound), this could be questioned. For the exam, though, NAT in a private subnet is often an acceptable compromise for controlled outbound access. This is likely correct, but let's check the others to confirm.

Option C: Use an S3 bucket with a VPC endpoint and IAM instance profile

This option shifts the artifacts to an S3 bucket and adds an S3 VPC endpoint (a gateway endpoint) in the VPC. The instances, presumably in a private subnet (implied but not stated), get an IAM instance profile to access the S3 bucket. The user data script is updated to fetch artifacts from S3 instead of the internet. The S3 endpoint allows private access to S3 without traversing the internet---no NAT or IGW needed. This fully eliminates internet dependency: the instances have no public IPs, no outbound internet route, and only communicate with S3 via AWS's internal network. It's secure, compliant with "no internet access," and leverages AWS-native features. The catch? It requires modifying the script and infrastructure (S3 bucket, endpoint, IAM role), but that's feasible and aligns with DevOps best practices. This also seems correct---maybe even more so than B.

Option D: Use a security group to allow outbound traffic to the artifact repo, then remove the rule

Here, the instances stay in their current setup (likely public subnet, given the original scenario), and a security group allows outbound traffic only to the artifact repository. The user data script fetches the artifacts, then the rule is removed to block further internet access. This is creative but problematic. First, if the instances are in a public subnet with an IGW route, they still have internet access unless the route table changes---security groups don't override routing. Second, dynamically removing the rule after install is tricky to automate reliably at launch (user data runs once; how do you trigger the removal?). Third, it assumes the instances start with internet access, which might already violate the new rule. This is impractical and doesn't fully comply, so D is incorrect.

Analysis: Is B Correct?

Your pick, B, works because it moves instances to a private subnet and uses a NAT gateway to enable outbound artifact retrieval during launch. The application installs, and the instances lack public IPs or inbound internet access, aligning with many AWS scenarios where "no internet" means "no public exposure." However, C stands out as potentially more correct. It eliminates internet access entirely (no NAT, no outbound internet route) by keeping traffic within the VPC via the S3 endpoint. If the exam interprets "no access to the internet" as zero internet connectivity (inbound and outbound), C wins. AWS exams often favor VPC endpoints for private access over NAT-based solutions when S3 is involved, as it's more secure and cost-effective (no NAT gateway fees).

That said, B isn't wrong in a practical sense---NAT in a private subnet is a standard pattern for controlled outbound access, and the question doesn't specify the artifact source (it could be an external HTTP server, not S3). Without that detail, B remains viable. If this were a real exam question with two correct answers required, B and C might both be intended, but here it's one choice, and C edges out slightly for strict compliance.

Explanation for Your Answer (B)

You're mostly correct with B because it solves the installation issue while reducing internet exposure. The NAT gateway ensures the user data script can fetch artifacts, and the private subnet prevents inbound internet access. The reason it's not perfect is the lingering outbound internet capability, which C avoids. If you assumed the artifacts aren't in S3 or that NAT is sufficient for "no internet" in this context, B is a solid pick. For exam purposes, watch for clues like S3 mentions---here, there's none, so B holds up.

Correct Answer (likely intended): C, for full internet isolation.

Your Answer (B): Very reasonable, just less strict on "no internet."

What do you think---does that clarify why B works but C might be the exam's preference? Ready for another question?

****A company wants to use AWS CloudFormation for infrastructure deployment. The company has strict tagging and resource requirements and wants to limit the deployment to two Regions. Developers will need to deploy multiple versions of the same application.**

Which solution ensures resources are deployed in accordance with company policy?**

- A. Create AWS Trusted Advisor checks to find and remediate unapproved CloudFormation StackSets.
- B. Create a Cloud Formation drift detection operation to find and remediate unapproved CloudFormation StackSets.
- C. Create CloudFormation StackSets with approved CloudFormation templates.
- D. Create AWS Service Catalog products with approved CloudFormation templates.

Let's tackle this CloudFormation question and see if your pick, C, holds up. The company wants to use AWS CloudFormation for infrastructure deployment with strict tagging and resource requirements, limit deployments to two specific Regions, and allow developers to deploy multiple versions of the same application. We need a solution that enforces policy compliance while meeting these needs. Let's break it down option by option.

Option A: AWS Trusted Advisor checks to find and remediate unapproved CloudFormation StackSets

Trusted Advisor is a tool that provides best-practice recommendations, like cost optimization or security checks, based on your AWS resources. It can flag untagged resources or other policy violations, but it's a reactive tool---it inspects what's already deployed and doesn't actively control or deploy resources. Using it to "find and remediate unapproved CloudFormation StackSets" implies a manual or scripted post-deployment cleanup process. This doesn't prevent non-compliant deployments upfront, nor does it inherently limit Regions or support multiple application versions. It's a monitoring fix, not a deployment solution. A is incorrect---it's too late in the process and lacks proactive control.

Option B: CloudFormation drift detection to find and remediate unapproved CloudFormation StackSets

CloudFormation drift detection checks if resources managed by a stack have deviated from their defined templates (e.g., someone manually edited a resource). It's useful for ensuring compliance after deployment, but like Trusted Advisor, it's reactive---you detect drift after the fact, then remediate. It doesn't deploy resources, enforce Region limits, or manage multiple application versions. The mention of "unapproved CloudFormation StackSets" suggests policing existing deployments, not setting up a compliant system from the start. B is incorrect---it's a drift fix, not a deployment enforcer.

Option C: CloudFormation StackSets with approved CloudFormation templates

Your choice! CloudFormation StackSets extend CloudFormation to deploy stacks across multiple accounts and Regions from a single operation. You define a template (with strict tagging and resource configs baked in) and specify target Regions---here, the two allowed ones. StackSets can be restricted to approved templates via IAM permissions, ensuring developers only deploy what complies with company policy. For multiple application versions, developers can use different StackSets or parameterize the same template (e.g., version-specific inputs). The centralized control and Region targeting make this a strong fit. The catch? It's not as developer-friendly as a self-service catalog, but it still works if developers are given access to approved StackSets. C looks promising---let's confirm against D.

Option D: AWS Service Catalog products with approved CloudFormation templates

AWS Service Catalog lets you create a portfolio of "products"---pre-approved CloudFormation templates---that developers can launch via a self-service portal. You embed tagging rules, resource configs, and constraints (like allowed Regions) into the templates and portfolio settings. Developers pick a product, specify parameters (e.g., app version), and deploy, all while staying within policy guardrails. You can limit deployments to the two Regions via IAM policies or launch constraints tied to the portfolio. For multiple versions, you can offer different products or use parameterized templates. This adds a user-friendly layer over raw CloudFormation, making compliance seamless. D seems very strong---maybe stronger than C.

Analysis: Is C Correct?

Your pick, C (CloudFormation StackSets), ensures compliance by using approved templates with strict tagging and resources, and it can limit deployments to two Regions via its Region targeting feature. Developers can deploy multiple versions by managing different StackSets or tweaking parameters. It's a solid AWS-native solution focused on multi-Region stack management, and with proper IAM controls, it enforces policy.

However, D (AWS Service Catalog) might be the exam's darling. It wraps CloudFormation in a governance layer, explicitly designed for companies with strict requirements. It's more developer-friendly (self-service UI) and naturally supports versioning via multiple products or parameters, while Region limits can be enforced

through portfolio constraints or IAM. Service Catalog is often the go-to for "policy enforcement + developer flexibility" scenarios in AWS cert questions, aligning with DevOps principles of controlled autonomy.

The key difference is intent: StackSets (C) is about deploying stacks across Regions/accounts efficiently, while Service Catalog (D) is about curating and governing what developers can deploy. The question's emphasis on "strict requirements" and developer-driven deployments leans toward D, but C isn't wrong---it just requires more setup (e.g., IAM) to lock it down.

For the exam, if it's one answer, D often edges out because it's purpose-built for this use case. If it were "choose two," C and D could both apply (StackSets for deployment, Service Catalog for governance). Given "which solution," D's governance focus likely wins.

Explanation for Your Answer (C)

You're almost right with C---StackSets can enforce policy via approved templates and Region targeting, and it supports multiple versions with some flexibility. It's a practical choice, especially if you pictured admins managing deployments. The reason it's not the best is that Service Catalog (D) adds a governance layer tailored to strict policies and developer self-service, which aligns more precisely with the scenario. C works but lacks the curated control D offers.

Correct Answer (likely intended): D. Create AWS Service Catalog products with approved CloudFormation templates.

Your Answer (C): Very close, just missing that extra governance polish.

A company's application is currently deployed to a single AWS Region. Recently, the company opened a new office on a different continent. The users in the new office are experiencing high latency. The company's application runs on Amazon EC2 instances behind an Application Load Balancer (ALB) and uses Amazon DynamoDB as the database layer. The instances run in an EC2 Auto Scaling group across multiple Availability Zones. A DevOps engineer is tasked with minimizing application response times and improving availability for users in both Regions.

Which combination of actions should be taken to address the latency issues? (Choose three.)

- A. Create a new DynamoDB table in the new Region with cross-Region replication enabled.
- B. Create new ALB and Auto Scaling group global resources and configure the new ALB to direct traffic to the new Auto Scaling group.
- C. Create new ALB and Auto Scaling group resources in the new Region and configure the new ALB to direct traffic to the new Auto Scaling group.
- D. Create Amazon Route 53 records, health checks, and latency-based routing policies to route to the ALB.
- E. Create Amazon Route 53 aliases, health checks, and failover routing policies to route to the ALB.
- F. Convert the DynamoDB table to a global table.

Understanding the Problem

The application is in one Region (let's call it Region A), and users in the new office on another continent (Region B) face high latency due to the distance. Latency in this context likely stems from:

1. Network distance to the ALB and EC2 instances in Region A.

2. Database access---DynamoDB reads/writes from Region A are slow for Region B users.

To improve availability, we need to handle failures (e.g., Region outages) and ensure the app is responsive globally. A multi-Region architecture with intelligent routing and data replication is the likely solution. Let's evaluate each option.

Option A: Create a new DynamoDB table in the new Region with cross-Region replication enabled

This involves creating a second DynamoDB table in Region B and setting up cross-Region replication (e.g., using DynamoDB Streams and Lambda to sync data). This reduces read latency for Region B users by serving data locally, but writes still go to the primary table (typically in Region A) unless the app is updated to write to both. It's a manual setup---DynamoDB doesn't natively do cross-Region replication this way without custom code. It also risks consistency issues (eventual consistency across Regions). This helps with database latency but isn't the most efficient approach compared to a native solution. A is viable but not ideal.

Option B: Create new ALB and Auto Scaling group global resources and configure the new ALB to direct traffic to the new Auto Scaling group

The phrase "global resources" is odd---ALBs and Auto Scaling groups are Region-specific, not global. This might mean deploying them in a second Region (like C), but "global" could imply confusion. Assuming it's a miswording for "in the new Region," it's similar to C: deploy an ALB and Auto Scaling group in Region B to serve local users. However, without routing (e.g., Route 53), users won't automatically hit the closer ALB. If "global" means something else (like a single ALB somehow spanning Regions), that's not how AWS works---ALBs are Regional. B is unclear but likely redundant with C.

Option C: Create new ALB and Auto Scaling group resources in the new Region and configure the new ALB to direct traffic to the new Auto Scaling group

Your first pick! This deploys a new ALB and Auto Scaling group with EC2 instances in Region B. The new ALB routes traffic to these local instances, reducing latency for Region B users by serving them from a closer Region. This is a core piece of a multi-Region architecture---local compute resources mean faster responses. However, we need a way to route users to the right Region (not addressed here). C is a strong choice for compute latency.

Option D: Create Amazon Route 53 records, health checks, and latency-based routing policies to route to the ALB

Your second pick! Route 53 latency-based routing directs users to the closest healthy ALB based on network latency. You'd set up DNS records pointing to both ALBs (Region A and Region B), with health checks ensuring traffic only goes to an ALB if its Region is up. This pairs perfectly with C---Region B users hit the new ALB, minimizing latency, while Region A users stay with the original. It improves availability by avoiding failed Regions (via health checks). D is a must for multi-Region routing.

Option E: Create Amazon Route 53 aliases, health checks, and failover routing policies to route to the ALB

This sets up Route 53 with failover routing: one ALB is primary (Region A), and if it fails (detected via health checks), traffic switches to the secondary (Region B). Aliases simplify DNS management. Failover improves availability but doesn't address latency---Region B users still hit Region A unless it fails, which doesn't solve the primary issue (high latency). E helps availability but not latency, so it's less relevant here.

Option F: Convert the DynamoDB table to a global table

Your third pick! DynamoDB global tables automatically replicate data across multiple Regions with eventual consistency. Converting the table to a global table (Region A + Region B) means Region B users read from a local replica, slashing database latency. Writes can go to either Region and replicate globally, simplifying app logic (no custom replication needed, unlike A). This doesn't affect compute latency but fixes the database side. F is excellent for database latency.

Analysis: Are C, D, and F Correct?

Your picks:

- C: New ALB and Auto Scaling group in Region B. This brings compute closer to Region B users, cutting network latency for app requests. It's a foundational step for multi-Region HA.
- D: Route 53 latency-based routing with health checks. This ensures users hit the closest ALB (Region B for new users), directly addressing latency, and improves availability by avoiding unhealthy Regions.
- F: DynamoDB global table. This replicates data to Region B, reducing database read/write latency for those users, complementing the compute fix.

These three together form a complete solution:

- Compute latency (C): Local ALB + EC2 in Region B.
- Routing (D): Route 53 directs users to the lowest-latency ALB.
- Database latency (F): Global table serves data locally.

They address both goals:

- Minimize response times: C and D reduce app latency; F reduces DB latency.
- Improve availability: C adds a second Region; D ensures routing to healthy Regions; F keeps data available multi-Region.

Alternatives:

- A: Cross-Region replication for DynamoDB is manual and less efficient than F (global tables). F replaces A.
- B: Likely a misworded version of C. If it means something else, it's unclear and doesn't add value over C.
- E: Failover routing helps availability but not latency---Region B users still suffer until Region A fails. D's latency-based routing is better.

Your combo is spot-on: C and D handle the app and routing, while F tackles the database. No overlap, and all bases covered.

Why You Picked C, D, and F

You likely recognized:

- C: The need for local compute in Region B to reduce app latency---a multi-Region setup is key.

- D: Route 53 latency-based routing as the best way to direct users to the closest ALB, addressing the core latency issue.
- F: DynamoDB global tables as the simplest way to cut database latency, leveraging AWS's native multi-Region feature.

You nailed the balance of compute, routing, and data---exactly what a DevOps engineer should prioritize for global HA and low latency.

Correct Answer: C, D, and F.

A development team is using AWS CodeCommit to version control application code and AWS CodePipeline to orchestrate software deployments. The team has decided to use a remote main branch as the trigger for the pipeline to integrate code changes. A developer has pushed code changes to the CodeCommit repository, but noticed that the pipeline had no reaction, even after 10 minutes.

Which of the following actions should be taken to troubleshoot this issue?

- A. Check that an Amazon EventBridge rule has been created for the main branch to trigger the pipeline.
- B. Check that the CodePipeline service role has permission to access the CodeCommit repository.
- C. Check that the developer's IAM role has permission to push to the CodeCommit repository.
- D. Check to see if the pipeline failed to start because of CodeCommit errors in Amazon CloudWatch Logs.

Understanding the Problem

The pipeline is configured to trigger off the main branch, meaning it should detect commits to that branch and start automatically. The developer pushed changes to the CodeCommit repository, but nothing happened. Possible reasons include:

1. The trigger mechanism (e.g., event notification) isn't set up correctly.
2. Permissions issues are preventing the pipeline from detecting or accessing the changes.
3. The developer's push didn't actually succeed, or the branch isn't what the pipeline expects.
4. The pipeline itself has a failure or misconfiguration.

Since the pipeline didn't react at all, we're likely dealing with a failure in the trigger mechanism or permissions, rather than an execution error (which would show the pipeline starting and failing). Let's evaluate the options.

Option A: Check that an Amazon EventBridge rule has been created for the main branch to trigger the pipeline

CodePipeline often uses Amazon EventBridge (formerly CloudWatch Events) to detect changes in CodeCommit repositories. When a commit is pushed to a specified branch (here, main), an EventBridge rule should capture that event and trigger the pipeline. If this rule is missing, misconfigured, or doesn't match the main branch, the pipeline won't start. The 10-minute delay with no reaction fits this scenario---EventBridge rules trigger near-instantly when working, so their absence would cause exactly this issue. Checking for the rule (and its branch filter) is a logical first step. A looks promising.

Option B: Check that the CodePipeline service role has permission to access the CodeCommit repository

CodePipeline uses a service role to interact with its source (CodeCommit). This role needs permissions to read from the repository (e.g., `codecommit:Get*` actions). If the role lacks these permissions, the pipeline might fail to detect changes or pull the source code. However, if permissions were the issue, you'd often see the pipeline start and then fail with an error (visible in the pipeline's status or logs), rather than not reacting at all. Still, it's worth checking, as some setups might silently fail to trigger due to access issues. B is plausible but less likely to cause a total lack of reaction.

Option C: Check that the developer's IAM role has permission to push to the CodeCommit repository

Your pick! The developer pushed changes to CodeCommit, but if their IAM role lacks permissions (e.g., `codecommit:Put*` or `codecommit:GitPush`), the push might have failed silently from CodePipeline's perspective. However, the developer likely used a Git client, which would error out visibly if permissions were lacking (e.g., "access denied"). The question implies the push succeeded ("developer has pushed code changes"), and 10 minutes have passed, so the commit should be in CodeCommit. Checking the developer's permissions is useful if we suspect the push didn't actually happen, but if the commit is in the repository, this isn't the issue. We'd need to verify the commit exists in the main branch to rule this out. C is a good thought but likely not the core issue if the push succeeded.

Option D: Check to see if the pipeline failed to start because of CodeCommit errors in Amazon CloudWatch Logs

CloudWatch Logs can show errors from CodePipeline or CodeCommit (e.g., if the pipeline tried to start but failed due to a misconfiguration). However, the pipeline "had no reaction," meaning it didn't start at all---no failed stages, no error status. If the pipeline isn't starting, there won't be logs for a failed execution. CodeCommit might log push events separately, but the question points us to pipeline-related errors. This option is more relevant if the pipeline started and then errored out, which isn't the case here. D isn't the best first step.

Analysis: Is C the Best Action?

Your pick, C, focuses on the developer's ability to push to CodeCommit. It's a reasonable troubleshooting step if you're unsure whether the commit actually made it to the repository. If the developer lacked permissions, the push would fail, and the pipeline wouldn't trigger because there's no new commit. However, the question states the developer "pushed code changes to the CodeCommit repository," implying the push succeeded. If the commit is in the main branch (which we'd confirm in the repo's history), the issue lies downstream---either the pipeline isn't detecting the change (trigger issue) or can't access it (permissions).

- Why C might not be the best: If the push succeeded, the developer's permissions aren't the problem. The pipeline's lack of reaction points to a trigger failure (EventBridge rule) or a pipeline permission issue.
- What's more likely: Option A jumps out as the most direct troubleshooting step. CodePipeline relies on EventBridge to detect CodeCommit changes. If the rule is missing, points to the wrong branch (e.g., master instead of main), or isn't linked to the pipeline, nothing will happen---exactly what we're seeing. Checking this first would confirm whether the trigger mechanism is even in place.

Let's double-check the others:

- B: Pipeline permissions could cause issues, but typically you'd see the pipeline attempt to start and fail, not sit idle. Still, it's worth investigating after A.
- D: No pipeline activity means no logs to check for a failed start. This is more of a follow-up step.

The best first action is A---verify the EventBridge rule. If the rule exists and is correct, then we'd check the pipeline's permissions (B) and confirm the commit is in the right branch, possibly revisiting the developer's permissions (C) if the commit isn't there.

Why You Picked C

You likely picked C because you wanted to ensure the developer's push actually happened. If their IAM role lacked permissions, the commit wouldn't be in CodeCommit, explaining why the pipeline didn't trigger. It's a logical step to rule out user error, especially if you weren't 100% sure the push succeeded. However, the question's phrasing suggests the push went through, shifting the focus to why the pipeline isn't detecting it. That's where the EventBridge rule (A) becomes critical.

Correct Answer (likely intended): A. Check that an Amazon EventBridge rule has been created for the main branch to trigger the pipeline.

Your Answer (C): A good troubleshooting step, but not the most direct for this specific failure.

A company's developers use Amazon EC2 instances as remote workstations. The company is concerned that users can create or modify EC2 security groups to allow unrestricted inbound access. A DevOps engineer needs to develop a solution to detect when users create unrestricted security group rules. The solution must detect changes to security group rules in near real time, remove unrestricted rules, and send email notifications to the security team. The DevOps engineer has created an AWS Lambda function that checks for security group ID from input, removes rules that grant unrestricted access, and sends notifications through Amazon Simple Notification Service (Amazon SNS).

What should the DevOps engineer do next to meet the requirements?

- A. Configure the Lambda function to be invoked by the SNS topic. Create an AWS CloudTrail subscription for the SNS topic. Configure a subscription filter for security group modification events.
- B. Create an Amazon EventBridge scheduled rule to invoke the Lambda function. Define a schedule pattern that runs the Lambda function every hour.
- C. Create an Amazon EventBridge event rule that has the default event bus as the source. Define the rule's event pattern to match EC2 security group creation and modification events. Configure the rule to invoke the Lambda function.
- D Create an Amazon EventBridge custom event bus that subscribes to events from all AWS services. Configure the Lambda function to be invoked by the custom event bus.

Understanding the Requirements

We need to:

1. Detect security group rule changes in near real time: Catch when rules are created or modified (e.g., via `AuthorizeSecurityGroupIngress` or `ModifySecurityGroupRules` API calls).
2. Remove unrestricted rules: The Lambda function already does this.

3. Send email notifications: The Lambda function handles this via SNS.

4. Trigger mechanism: We need to invoke the Lambda function whenever a security group rule changes.

The key is a triggering system that captures security group changes in near real time and invokes the Lambda function. Let's evaluate the options.

Option A: Configure the Lambda function to be invoked by the SNS topic. Create an AWS CloudTrail subscription for the SNS topic. Configure a subscription filter for security group modification events

This suggests using SNS as the trigger for Lambda, with CloudTrail feeding events to SNS. CloudTrail logs API calls (like security group changes), and it can integrate with Amazon EventBridge (not SNS directly) to capture events. However, "CloudTrail subscription for the SNS topic" is incorrect--CloudTrail doesn't subscribe to SNS. You'd typically use CloudTrail to log events, then use EventBridge to match those events and trigger SNS or Lambda. Even if we interpret this as CloudTrail → EventBridge → SNS → Lambda, it adds unnecessary complexity (SNS in the middle) when EventBridge can directly invoke Lambda. The subscription filter for security group events is feasible in EventBridge, but the SNS setup here is off. A is incorrect due to the flawed SNS-CloudTrail linkage.

Option B: Create an Amazon EventBridge scheduled rule to invoke the Lambda function. Define a schedule pattern that runs the Lambda function every hour

This sets up an EventBridge rule to run the Lambda function hourly. The Lambda would then check all security groups for unrestricted rules. This isn't near real time---changes could sit undetected for up to an hour, violating the requirement. It's also inefficient, as it scans all security groups repeatedly, even if no changes occurred. Scheduled polling is better for periodic cleanup, not event-driven detection. B fails the "near real time" requirement.

Option C: Create an Amazon EventBridge event rule that has the default event bus as the source. Define the rule's event pattern to match EC2 security group creation and modification events. Configure the rule to invoke the Lambda function

Your pick! EventBridge can capture AWS API events via CloudTrail integration. The default event bus receives events from AWS services like EC2. We can define an event pattern to match security group changes, such as `AuthorizeSecurityGroupIngress`, `AuthorizeSecurityGroupEgress`, `RevokeSecurityGroupIngress`, or `ModifySecurityGroupRules`. For example, the pattern might look like:

json

```
{  
  "source": ["aws.ec2"],  
  "detail-type": ["AWS API Call via CloudTrail"],  
  "detail": {  
    "eventSource": ["ec2.amazonaws.com"],  
    "eventName": [  
      "AuthorizeSecurityGroupIngress",
```

```
"AuthorizeSecurityGroupEgress",  
"RevokeSecurityGroupIngress",  
"ModifySecurityGroupRules"  
]  
}  
}
```

When a matching event occurs (e.g., a developer adds a 0.0.0.0/0 rule), EventBridge invokes the Lambda function, passing the security group ID. The Lambda then removes the rule and sends an SNS notification. EventBridge events are near real time (typically seconds), meeting the requirement. This is a clean, event-driven architecture. C looks like a winner.

Option D: Create an Amazon EventBridge custom event bus that subscribes to events from all AWS services. Configure the Lambda function to be invoked by the custom event bus

A custom event bus in EventBridge is for user-defined or third-party events, not AWS service events like EC2 API calls. The default event bus (as in C) already captures AWS service events via CloudTrail. Creating a custom event bus to "subscribe to events from all AWS services" isn't how EventBridge works---AWS services publish to the default bus, not custom ones, unless explicitly configured (e.g., via cross-account setups). Even if we forced this setup, it's overcomplicating things compared to using the default bus with a rule (C). D is incorrect and unnecessarily complex.

Analysis: Is C the Best Action?

Your pick, C, nails the requirements:

- Near real time: EventBridge captures security group changes within seconds via CloudTrail.
- Event-driven: The rule matches creation/modification events (e.g., AuthorizeSecurityGroupIngress), triggering Lambda immediately.
- Simplicity: It uses the default event bus, which is designed for AWS service events, and directly invokes Lambda.

The Lambda function handles the rest (removing rules, sending SNS notifications). This setup is a textbook AWS pattern for real-time monitoring and remediation:

- CloudTrail logs the API call → EventBridge matches the event → Lambda acts.

Alternatives:

- A: Misunderstands CloudTrail-SNS integration and adds unnecessary hops.
- B: Fails "near real time" with hourly checks.
- D: Custom event bus is overkill and incorrect for AWS service events.

C is the clear winner. It's event-driven, fast, and leverages AWS best practices.

Why You Picked C

You likely picked C because you recognized EventBridge as the go-to service for capturing AWS API events in real time. You saw that security group changes (via EC2 API calls) can be matched with an event pattern, and triggering Lambda directly is the most efficient way to act on those changes. You also understood the "near real time" requirement, ruling out scheduled approaches like B. Your instinct to use EventBridge's default bus for AWS events was spot-on---nice work!

Correct Answer: C. Create an Amazon EventBridge event rule that has the default event bus as the source. Define the rule's event pattern to match EC2 security group creation and modification events. Configure the rule to invoke the Lambda function.

A DevOps engineer is creating an AWS CloudFormation template to deploy a web service. The web service will run on Amazon EC2 instances in a private subnet behind an Application Load Balancer (ALB). The DevOps engineer must ensure that the service can accept requests from clients that have IPv6 addresses.

What should the DevOps engineer do with the CloudFormation template so that IPv6 clients can access the web service?

- A. Add an IPv6 CIDR block to the VPC and the private subnet for the EC2 instances. Create route table entries for the IPv6 network, use EC2 instance types that support IPv6, and assign IPv6 addresses to each EC2 instance.
- B. Assign each EC2 instance an IPv6 Elastic IP address. Create a target group, and add the EC2 instances as targets. Create a listener on port 443 of the ALB, and associate the target group with the ALB.
- C. Replace the ALB with a Network Load Balancer (NLB). Add an IPv6 CIDR block to the VPC and subnets for the NLB, and assign the NLB an IPv6 Elastic IP address.
- D. Add an IPv6 CIDR block to the VPC and subnets for the ALB. Create a listener on port 443. and specify the dualstack IP address type on the ALB. Create a target group, and add the EC2 instances as targets. Associate the target group with the ALB.

Understanding the Requirements

The web service architecture:

- EC2 instances in a private subnet (no public IPs, isolated from direct internet access).
- An Application Load Balancer (ALB) in front of the EC2 instances, likely in a public subnet to accept client traffic.
- Clients with IPv6 addresses need to access the service.

To support IPv6:

1. The ALB must accept IPv6 traffic (since it faces the clients).
2. The VPC and relevant subnets must support IPv6.
3. Traffic must flow from the ALB to the EC2 instances (IPv4 or IPv6, depending on setup).
4. CloudFormation must define these resources with IPv6 support.

The key is ensuring the ALB can handle IPv6 client requests, as it's the entry point. Let's evaluate the options.

Option A: Add an IPv6 CIDR block to the VPC and the private subnet for the EC2 instances. Create route table entries for the IPv6 network, use EC2 instance types that support IPv6, and assign IPv6 addresses to each EC2 instance

This focuses on the EC2 instances:

- Adds an IPv6 CIDR block to the VPC and private subnet (good start---VPC needs IPv6 support).
- Adds route table entries for IPv6 (e.g., to an internet gateway for outbound traffic, but private subnets don't need this for inbound).
- Uses IPv6-compatible EC2 instance types (all modern types support IPv6, so this is trivial).
- Assigns IPv6 addresses to each EC2 instance.

This ensures the EC2 instances can communicate over IPv6, but the question is about clients accessing the service via the ALB. The ALB is the internet-facing component, and this option doesn't configure the ALB for IPv6. If the ALB can't accept IPv6 requests, the EC2 instances' IPv6 support is irrelevant for inbound client traffic. The ALB-to-EC2 traffic can use IPv4 (common in dual-stack setups), so the EC2 instances don't strictly need IPv6 addresses. A misses the ALB configuration, making it incomplete.

Option B: Assign each EC2 instance an IPv6 Elastic IP address. Create a target group, and add the EC2 instances as targets. Create a listener on port 443 of the ALB, and associate the target group with the ALB

This suggests:

- Assigning IPv6 Elastic IP addresses to EC2 instances.
- Setting up a target group and listener on the ALB.

Elastic IPs (EIPs) are public IPv4 or IPv6 addresses, but EC2 instances in a private subnet shouldn't have public IPs (EIPs are public). This violates the architecture---private subnets are meant to be non-internet-facing, and the ALB handles public traffic. Even if we ignore that, EIPs on EC2 instances aren't needed for ALB-to-EC2 communication; the ALB can route to instances via private IPs (IPv4 or IPv6). The listener on port 443 and target group setup are fine but don't address IPv6 on the ALB itself. The ALB needs to accept IPv6 client requests, which this option doesn't configure. B is incorrect due to the misuse of EIPs and lack of ALB IPv6 support.

Option C: Replace the ALB with a Network Load Balancer (NLB). Add an IPv6 CIDR block to the VPC and subnets for the NLB, and assign the NLB an IPv6 Elastic IP address

This proposes:

- Swapping the ALB for an NLB.
- Adding an IPv6 CIDR block to the VPC and NLB subnets (likely public subnets).
- Assigning the NLB an IPv6 EIP.

NLBs support IPv6 via dual-stack mode (IPv4 + IPv6), and adding an IPv6 CIDR block to the VPC and subnets enables this. However, NLBs don't use EIPs---EIPs are for EC2 instances or NAT gateways, not NLBs. NLBs get

IPv6 support by enabling dual-stack on the load balancer itself, not via EIPs. Additionally, the question implies the ALB is intentional (it's a web service, likely needing Layer 7 features like path-based routing or SSL termination, which ALBs handle better). Switching to an NLB loses these features and isn't necessary---ALBs support IPv6 too. C is incorrect due to the EIP misuse and unnecessary ALB-to-NLB swap.

Option D: Add an IPv6 CIDR block to the VPC and subnets for the ALB. Create a listener on port 443, and specify the dualstack IP address type on the ALB. Create a target group, and add the EC2 instances as targets. Associate the target group with the ALB

Your pick! Let's break it down:

- Add an IPv6 CIDR block to the VPC and subnets for the ALB: ALBs are typically in public subnets (to face the internet). Adding an IPv6 CIDR block to the VPC (e.g., an Amazon-provided /56 block) and associating it with the ALB's subnets enables IPv6 support. In CloudFormation, this uses `IPv6CidrBlock` for the VPC and `AssignIpv6AddressOnCreation` for the subnets.
- Specify the dualstack IP address type on the ALB: ALBs support `ipv4` or `dualstack` modes. Setting `ip-address-type: dualstack` in the CloudFormation template (via the `IpAddressType` attribute of the `AWS::ElasticLoadBalancingV2::LoadBalancer` resource) allows the ALB to accept both IPv4 and IPv6 traffic. This means IPv6 clients can connect directly.
- Create a listener on port 443, create a target group, add EC2 instances, and associate the target group: This sets up the ALB to route traffic to the EC2 instances. The listener on port 443 (HTTPS) is standard for web services, and the target group defines the EC2 instances as targets (using their private IPs, typically IPv4).

This setup ensures:

- The ALB accepts IPv6 client requests (dual-stack).
- Traffic flows to the EC2 instances (which can use IPv4 in the private subnet---no IPv6 needed on the instances, as the ALB handles IPv6 termination).
- The CloudFormation template defines all resources correctly.

The EC2 instances don't need IPv6 addresses because the ALB-to-EC2 communication can use IPv4 (dual-stack ALBs support this). The private subnet doesn't need an IPv6 CIDR block unless the instances need to initiate outbound IPv6 traffic (not required here). D meets all requirements elegantly.

Analysis: Is D the Best Action?

Your pick, D, directly addresses the goal: enabling IPv6 clients to access the web service. It configures the ALB (the client-facing component) for IPv6 by:

- Adding IPv6 support to the VPC and ALB subnets.
- Setting the ALB to dualstack mode.
- Setting up standard ALB components (listener, target group).

This ensures IPv6 clients can connect, and the EC2 instances in the private subnet remain isolated (no public IPs needed). It's the most straightforward and AWS-recommended approach for this scenario.

Alternatives:

- A: Focuses on EC2 instances, which don't need IPv6 for inbound traffic from the ALB. Misses ALB configuration.
- B: Misuses EIPs on private instances and doesn't enable IPv6 on the ALB.
- C: NLB swap is unnecessary, and EIPs on NLBs aren't correct.

D is the clear winner. It leverages ALB's dual-stack capability, keeps the architecture intact, and meets the IPv6 requirement without overcomplicating things.

Why You Picked D

You likely picked D because you recognized that the ALB, as the internet-facing component, needs to handle IPv6 client requests. You understood that enabling dualstack on the ALB and adding IPv6 CIDR blocks to the VPC and ALB subnets would allow IPv6 traffic, while the EC2 instances could remain IPv4-only in the private subnet. You also saw that the listener and target group setup is standard for ALB operation, ensuring the full traffic flow. Your choice aligns perfectly with AWS best practices for IPv6-enabled web services---great instinct!

Correct Answer: D. Add an IPv6 CIDR block to the VPC and subnets for the ALB. Create a listener on port 443, and specify the dualstack IP address type on the ALB. Create a target group, and add the EC2 instances as targets. Associate the target group with the ALB.

A company uses AWS Organizations and AWS Control Tower to manage all the company's AWS accounts. The company uses the Enterprise Support plan. A DevOps engineer is using Account Factory for Terraform (AFT) to provision new accounts. When new accounts are provisioned, the DevOps engineer notices that the support plan for the new accounts is set to the Basic Support plan. The DevOps engineer needs to implement a solution to provision the new accounts with the Enterprise Support Plan.

Which solution will meet these requirements?

- A. Use an AWS Config conformance pack to deploy the account-part-of-organizations AWS Config rule and to automatically remediate any noncompliant accounts.
- B. Create an AWS Lambda function to create a ticket for AWS Support to add the account to the Enterprise Support plan. Grant the Lambda function the support:ResolveCase permission.
- C. Add an additional value to the control_tower_parameters input to set the AWSEnterpriseSupport parameter as the organization's management account number.
- D. Set the aft_feature_enterprise_support feature flag to True in the AFT deployment input configuration. Redeploy AFT and apply the changes.

Understanding the Problem

- AWS Organizations and Control Tower: The company uses AWS Organizations to manage multiple accounts, with Control Tower providing governance and account provisioning through a landing zone.
- Enterprise Support Plan: The organization is enrolled in the Enterprise Support plan, which should apply to all accounts for consistent support levels (e.g., 24/7 access to support, technical account

managers).

- Account Factory for Terraform (AFT): AFT is a Terraform module that automates account provisioning in Control Tower, following a GitOps model. It provisions accounts and applies customizations via a pipeline.
- Issue: New accounts provisioned via AFT default to the Basic Support plan (AWS's default for new accounts) instead of inheriting the organization's Enterprise Support plan.
- Goal: Ensure new AFT-provisioned accounts are enrolled in the Enterprise Support plan automatically during provisioning.

The root issue is that AWS accounts, even under an organization with Enterprise Support, don't automatically inherit the support plan unless explicitly configured. AFT, as a provisioning tool, needs to be set up to enroll new accounts into Enterprise Support during creation. Let's explore how AFT and AWS handle support plans and evaluate the options.

Option A: Use an AWS Config conformance pack to deploy the account-part-of-organizations AWS Config rule and automatically remediate noncompliant accounts

- What this does: AWS Config conformance packs are collections of AWS Config rules that check for compliance. The account-part-of-organizations rule verifies that an account is part of an AWS Organization. If an account isn't compliant (e.g., not in the organization), a remediation action can be triggered.
- Analysis: This option focuses on ensuring accounts are part of the organization, but the accounts provisioned by AFT are already part of AWS Organizations (since AFT uses Control Tower, which operates under Organizations). The issue isn't about organizational membership---it's about the support plan. AWS Config rules don't directly manage support plans, and there's no Config rule to check or set the support plan (e.g., Enterprise vs. Basic). Even if we stretched this to assume a custom Config rule for support plans, remediation would be reactive (after provisioning), not part of the provisioning process, and it's not clear how Config would change the support plan.
- Conclusion: This doesn't address the support plan issue directly. It's about organizational compliance, not support enrollment. A is incorrect.

Option B: Create an AWS Lambda function to create a ticket for AWS Support to add the account to the Enterprise Support plan. Grant the Lambda function the support:ResolveCase permission

- What this does: This proposes writing a Lambda function that opens a support ticket (via the AWS Support API) to request adding the new account to the Enterprise Support plan. The support:ResolveCase permission allows the Lambda to resolve cases, but to create a case, it needs support:CreateCase (not mentioned, but let's assume it's implied).
- Analysis: This is a creative approach. AWS Support API can be used to open a ticket requesting Enterprise Support enrollment for a new account. In practice, when an organization has Enterprise Support, adding new accounts often involves a support ticket (especially if done manually). AFT itself uses this method when configured to enable Enterprise Support---it opens a ticket during provisioning. The Lambda could be triggered post-provisioning (e.g., via an EventBridge rule on account creation). However:

- Timing: Support tickets aren't instant; AWS Support processes them on their timeline, which could delay the account's enrollment in Enterprise Support.
 - Automation: This adds complexity---writing and maintaining a Lambda function, managing permissions, and handling ticket responses.
 - AFT capability: AFT already has a built-in feature to handle this (we'll see in D), so this feels like reinventing the wheel.
 - Permissions: `support:ResolveCase` lets Lambda close cases, but creating one requires `support:CreateCase`. The option doesn't mention how Lambda gets the account ID or is triggered, adding ambiguity.
- Conclusion: This could work but is overly complex and manual compared to a native AFT solution. It's a fallback if AFT didn't have a built-in feature, but we'll check that next. B is not the best choice.

Option C: Add an additional value to the `control_tower_parameters` input to set the `AWSEnterpriseSupport` parameter as the organization's management account number

- What this does: In AFT, the `control_tower_parameters` block in an account request Terraform file (e.g., in the `aft-account-request` repository) defines parameters for Control Tower account provisioning. This option suggests adding an `AWSEnterpriseSupport` parameter set to the management account number.
- Analysis: Let's break this down:
 - AFT account requests: When provisioning an account via AFT, you create a Terraform file in the `aft-account-request` repository. The `control_tower_parameters` block includes fields like `AccountEmail`, `AccountName`, and `ManagedOrganizationalUnit`. For example:

hcl

```
module "new_account" {
  source = "github.com/aws-ia/terraform-aws-
control_tower_account_factory//modules/aft-account-request"
  control_tower_parameters = {
    AccountEmail = "new-account@example.com"
    AccountName  = "NewAccount"
    ManagedOrganizationalUnit = "Sandbox"
  }
}
```

- Does `AWSEnterpriseSupport` exist?: There's no documented `AWSEnterpriseSupport` parameter in AFT's `control_tower_parameters`. AFT documentation (e.g., AWS Control Tower docs) lists supported fields, and this isn't one of them. The management account number might be used in support tickets, but it's not a parameter for enabling Enterprise Support in AFT.
- Support plan enrollment: Support plans are managed at the account level, but in Organizations, the payer account (management account) governs the plan for all member accounts. AFT handles this via a specific feature flag, not a `control_tower_parameters` field.

- Conclusion: This option invents a parameter that doesn't exist in AFT. It's trying to mimic a manual support ticket process (like B) but through an incorrect mechanism. C is incorrect.

Option D: Set the `aft_feature_enterprise_support` feature flag to `True` in the AFT deployment input configuration. Redeploy AFT and apply the changes

- What this does: AFT has optional feature flags that can be enabled during deployment. The `aft_feature_enterprise_support` flag, when set to `True`, automatically enrolls new accounts provisioned by AFT into the Enterprise Support plan. This requires redeploying AFT with the updated configuration.
- Analysis:
 - AFT feature flags: AFT offers several optional features, enabled via flags in the Terraform module configuration. According to AWS documentation (e.g., "Enable feature options - AWS Control Tower"), the `aft_feature_enterprise_support` flag is one such option. When enabled, AFT automatically opens a support ticket during provisioning to enroll the new account in Enterprise Support, assuming the payer account (management account) is already on Enterprise Support.
 - How it works: In the AFT module configuration (e.g., `main.tf` or `terraform.tfvars`), you set:

hcl

```
module "aft" {
  source = "github.com/aws-ia/terraform-aws-
control_tower_account_factory"
  # Required parameters
  ct_management_account_id = "123412341234"
  log_archive_account_id   = "234523452345"
  audit_account_id         = "345634563456"
  aft_management_account_id = "456745674567"
  ct_home_region           = "us-east-1"
  tf_backend_secondary_region = "us-west-2"
  # Feature flag
  aft_feature_enterprise_support = true
}
```

After setting this flag to `True`, you redeploy AFT by running `terraform apply`. This updates the AFT pipeline to include the Enterprise Support enrollment step for all new accounts.

- Requirements: The payer account (management account) must already be enrolled in Enterprise Support, which it is (per the question). AFT then ensures new accounts inherit this plan.
- Timing: This is proactive---new accounts get Enterprise Support during provisioning, not after. Redeploying AFT applies this to all future accounts provisioned via AFT.
- Effort: Minimal change---just a flag update and redeployment. No custom code or Lambda needed.

- Conclusion: This leverages AFT's built-in capability for Enterprise Support enrollment, directly addressing the requirement. It's the simplest, most AWS-native solution, aligning with AFT's design. D looks like the winner.

Double-Checking the Options

- A: AWS Config can't manage support plans, and the rule doesn't address the issue.
- B: Lambda could work but is overkill---writing custom code to replicate what AFT already does is inefficient.
- C: `AWSEnterpriseSupport` isn't a real parameter in AFT, making this a red herring.
- D: Matches AFT's documented feature (`aft_feature_enterprise_support`), solves the problem during provisioning, and requires minimal effort.

One potential concern: existing accounts already provisioned won't be retroactively updated by D---they'd need manual enrollment (e.g., via a support ticket). But the question focuses on new accounts ("provision the new accounts with the Enterprise Support plan"), so D fits perfectly.

Why This Matters for the Exam

This question tests your understanding of AFT and its integration with AWS Organizations and Control Tower. It highlights:

- AFT feature flags: Knowing AFT's built-in options (like `aft_feature_enterprise_support`) is key for automation tasks.
- AWS support plans: Understanding that support plans don't automatically propagate in Organizations unless configured.
- Proactive vs. reactive solutions: D is proactive (during provisioning), while B is reactive (post-provisioning).

Final Answer

D. Set the `aft_feature_enterprise_support` feature flag to True in the AFT deployment input configuration. Redeploy AFT and apply the changes.

A company's DevOps engineer uses AWS Systems Manager to perform maintenance tasks during maintenance windows. The company has a few Amazon EC2 instances that require a restart after notifications from AWS Health. The DevOps engineer needs to implement an automated solution to remediate these notifications. The DevOps engineer creates an Amazon EventBridge rule.

How should the DevOps engineer configure the EventBridge rule to meet these requirements?

- A. Configure an event source of AWS Health, a service of EC2, and an event type that indicates instance maintenance. Target a Systems Manager document to restart the EC2 instance.
- B. Configure an event source of Systems Manager and an event type that indicates a maintenance window. Target a Systems Manager document to restart the EC2 instance.
- C. Configure an event source of AWS Health, a service of EC2, and an event type that indicates instance

maintenance. Target a newly created AWS Lambda function that registers an automation task to restart the EC2 instance during a maintenance window.

D. Configure an event source of EC2 and an event type that indicates instance maintenance. Target a newly created AWS Lambda function that registers an automation task to restart the EC2 instance during a maintenance window.

Understanding the Problem

- AWS Systems Manager (SSM): The engineer uses SSM for maintenance tasks, likely leveraging SSM Automation to execute tasks during maintenance windows (predefined time periods for performing actions like patching or restarting).
- AWS Health Notifications: AWS Health (part of AWS Personal Health Dashboard) sends notifications about events affecting AWS resources, such as scheduled maintenance requiring an EC2 instance restart (e.g., underlying host maintenance).
- EC2 Instances: A few instances need restarting after these notifications.
- Goal: Automate restarting the EC2 instances in response to AWS Health notifications, ideally during a maintenance window, using an EventBridge rule.
- Requirements:
 1. Detect AWS Health notifications about EC2 instance maintenance.
 2. Trigger an action to restart the EC2 instance.
 3. Ensure the restart happens during a maintenance window (since the engineer uses SSM maintenance windows).

The key is configuring an EventBridge rule to catch the right event (AWS Health notification for EC2 maintenance) and trigger an action (restart via SSM) that respects the maintenance window constraint.

Option A: Configure an event source of AWS Health, a service of EC2, and an event type that indicates instance maintenance. Target a Systems Manager document to restart the EC2 instance

Your pick! Let's break this down:

- Event Source: AWS Health as the source, with EC2 as the service, and an event type for instance maintenance. AWS Health emits events to EventBridge when there's an issue or scheduled maintenance (e.g., `AWS_EC2_INSTANCE_SCHEDULED_REBOOT` or `AWS_EC2_INSTANCE_REBOOT_REQUIRED_BY_OS`). The event pattern might look like:

json

```
{
  "source": ["aws.health"],
  "detail-type": ["AWS Health Event"],
  "detail": {
    "service": ["EC2"],
    "eventTypeCategory": ["scheduledChange"],
    "eventTypeCode": ["AWS_EC2_INSTANCE_SCHEDULED_REBOOT"]
  }
}
```

```
}  
}
```

This correctly captures EC2-specific maintenance events, like a required reboot.

- Target: A Systems Manager document (SSM document) to restart the EC2 instance. SSM Automation documents (e.g., AWS-RestartEC2Instance) can restart instances. EventBridge can invoke an SSM Automation execution as a target, passing the instance ID from the AWS Health event (found in the resources field of the event payload).
- Maintenance Window: The option doesn't explicitly mention the maintenance window, but since the engineer uses SSM maintenance windows, we can assume the targeted SSM document is associated with a maintenance window (or the restart task is queued to run during one). In SSM, maintenance windows define schedules and tasks (via Automation documents), so the restart would execute during the next window.

Analysis:

- Pros: This directly ties AWS Health events to an SSM Automation action, which is a clean architecture. AWS Health → EventBridge → SSM Automation is a standard pattern for reacting to maintenance events. The AWS-RestartEC2Instance document (or a custom one) can handle the restart, and SSM ensures it aligns with maintenance window schedules if configured properly.
- Cons: The maintenance window isn't explicitly configured in the EventBridge rule. If the SSM document isn't tied to a maintenance window, the restart might happen immediately, not during the desired time. However, the question implies the engineer already uses SSM maintenance windows, so we can assume the document or SSM setup enforces this timing.
- Fit: This option correctly identifies the event source (AWS Health, EC2) and uses SSM to restart, aligning with the engineer's existing workflow.

A looks promising, but let's check the others.

Option B: Configure an event source of Systems Manager and an event type that indicates a maintenance window. Target a Systems Manager document to restart the EC2 instance

- Event Source: Systems Manager as the source, with an event type for a maintenance window. SSM can emit events to EventBridge, such as when a maintenance window starts (AWS_SSM_MAINTENANCE_WINDOW_EXECUTION_STARTED).
- Target: An SSM document to restart the EC2 instance, same as A.
- Analysis:
 - Issue with Source: The trigger here is the maintenance window itself, not the AWS Health notification. This means the rule would restart instances when a maintenance window opens, not in response to a specific AWS Health event. The question requires remediation triggered by AWS Health notifications, not just during a maintenance window.
 - Mismatch: This approach doesn't detect the need for a restart (from AWS Health); it assumes restarts happen on a schedule. It could restart instances unnecessarily if there's no AWS Health

event.

- Maintenance Window: While it ensures restarts happen during a maintenance window, it misses the core requirement of reacting to AWS Health notifications.
- Conclusion: This option focuses on the wrong trigger (SSM maintenance window instead of AWS Health). B is incorrect.

Option C: Configure an event source of AWS Health, a service of EC2, and an event type that indicates instance maintenance. Target a newly created AWS Lambda function that registers an automation task to restart the EC2 instance during a maintenance window

- Event Source: Same as A---AWS Health, EC2 service, instance maintenance event. This part is correct and matches the requirement to detect AWS Health notifications.
- Target: A new Lambda function that "registers an automation task to restart the EC2 instance during a maintenance window."
- Analysis:
 - Lambda Role: The Lambda would parse the AWS Health event, extract the instance ID, and use the SSM API (e.g., StartAutomationExecution) to trigger an SSM Automation document (like AWS-RestartEC2Instance). It would also need to interact with SSM to schedule this task during a maintenance window.
 - Maintenance Window Handling: Lambda would need logic to:
 1. Query SSM for the next maintenance window (e.g., via DescribeMaintenanceWindows).
 2. Schedule the automation task to run during that window (e.g., using SSM Maintenance Window tasks or a scheduled automation). This is possible but complex---Lambda would need to manage scheduling, which isn't its strength.
 - Comparison to A: Option A directly invokes the SSM document, assuming the maintenance window is already part of the SSM setup. C adds Lambda as a middleman to explicitly enforce the maintenance window, which adds control but also complexity.
 - Effort: Writing a Lambda function, granting it permissions (e.g., ssm:StartAutomationExecution, ssm:DescribeMaintenanceWindows), and handling scheduling logic is more work than using SSM directly.
- Conclusion: This works but is overkill. It achieves the same result as A (AWS Health → restart during maintenance window) but with an extra layer (Lambda). AWS Health → EventBridge → SSM is a simpler pattern unless the maintenance window scheduling needs explicit code, which isn't implied here. C is viable but less efficient than A.

Option D: Configure an event source of EC2 and an event type that indicates instance maintenance. Target a newly created AWS Lambda function that registers an automation task to restart the EC2 instance during a maintenance window

- Event Source: EC2 as the source, with an event type for instance maintenance.

- Target: Same Lambda as C, handling the restart during a maintenance window.
 - Analysis:
 - Issue with Source: EC2 itself doesn't emit "instance maintenance" events to EventBridge for this purpose. Events like instance state changes (e.g., EC2 Instance State-change Notification) come from EC2, but maintenance notifications (e.g., scheduled reboots) come from AWS Health. For example, an AWS Health event might have eventTypeCode: `AWS_EC2_INSTANCE_SCHEDULED_REBOOT`, while EC2 events are more like state: running or state: stopped. The question specifically mentions "notifications from AWS Health," so the source should be AWS Health, not EC2.
 - Target: The Lambda is the same as C---overly complex compared to direct SSM invocation.
 - Conclusion: The event source is wrong---EC2 doesn't emit the maintenance events we need; AWS Health does. This makes the whole setup invalid. D is incorrect.
-

Analysis: Is A the Best Choice?

Your pick, A, configures the EventBridge rule to:

- Detect AWS Health events for EC2 instance maintenance (correct source and event type).
- Target an SSM document to restart the instance.

Why A Works:

- Event Detection: AWS Health is the right source for maintenance notifications. The event pattern matches EC2-specific events like scheduled reboots, fulfilling the requirement to "remediate these notifications."
- Action: Targeting an SSM Automation document (e.g., `AWS-RestartEC2Instance`) directly restarts the instance. EventBridge can pass the instance ID from the AWS Health event to the SSM target.
- Maintenance Window: The question states the engineer "uses AWS Systems Manager to perform maintenance tasks during maintenance windows." This implies an existing SSM setup with maintenance windows. While A doesn't explicitly mention scheduling, SSM Automation executions can be tied to maintenance windows (e.g., via an SSM Maintenance Window task). In practice, you'd configure the SSM document or a maintenance window to ensure the restart happens at the right time, which aligns with the engineer's workflow.
- Simplicity: This avoids custom code (like Lambda in C and D), using a native EventBridge-to-SSM integration.

Potential Concern:

- If the SSM document isn't tied to a maintenance window, the restart might happen immediately, not during the desired time. However, the context ("uses SSM for maintenance windows") suggests the SSM setup handles this. In a real-world setup, you'd configure an SSM Maintenance Window with a task that uses the document, but for exam purposes, A assumes this is in place.

Comparison to Alternatives:

- B: Wrong source---SSM maintenance window events don't tie to AWS Health notifications.
- C: Adds Lambda for explicit maintenance window scheduling, which is more precise but unnecessary if SSM is already set up for maintenance windows. It's also more complex.
- D: Wrong source (EC2 instead of AWS Health), making it invalid.

A is the simplest, most direct solution that meets the requirements, assuming the SSM maintenance window setup is already in place (which the question implies). C is a close contender if we needed explicit control over the maintenance window timing, but the added complexity makes it less ideal for an exam answer unless the question demanded it.

****A company has containerized all of its in-house quality control applications. The company is running Jenkins on Amazon EC2 instances, which require patching and upgrading. The compliance officer has requested a DevOps engineer begin encrypting build artifacts since they contain company intellectual property. **** What should the DevOps engineer do to accomplish this in the MOST maintainable Manner?

- A. Automate patching and upgrading using AWS Systems Manager on EC2 instances and encrypt Amazon EBS volumes by default.
- B. Deploy Jenkins to an Amazon ECS cluster and copy build artifacts to an Amazon S3 bucket with default encryption enabled.
- C. Leverage AWS CodePipeline with a build action and encrypt the artifacts using AWS Secrets Manager.
- D. Use AWS CodeBuild with artifact encryption to replace the Jenkins instance running on EC2 instances.

Understanding the Problem

- Setup:
 - Jenkins on EC2: Jenkins runs on EC2 instances, which require patching and upgrading (e.g., OS updates, Jenkins updates).
 - Containerized Apps: The quality control apps are containerized, implying they could run in containers (e.g., Docker), but Jenkins itself is on EC2, not necessarily containerized.
 - Build Artifacts: Jenkins builds produce artifacts (e.g., compiled binaries, test reports) containing sensitive intellectual property.
- Requirements:
 - Encrypt Build Artifacts: Compliance demands encryption of these artifacts to protect intellectual property.
 - Most Maintainable: The solution should minimize operational overhead (e.g., patching EC2 instances) and ensure long-term manageability.
- Implicit Goals:
 - Address the patching/upgrading burden of Jenkins on EC2, as this is highlighted as a concern.
 - Ensure the encryption mechanism is sustainable and integrates well with AWS services.

The key is to find a solution that encrypts artifacts securely while reducing the maintenance burden of managing Jenkins on EC2. Let's evaluate each option.

Option A: Automate patching and upgrading using AWS Systems Manager on EC2 instances and encrypt Amazon EBS volumes by default

- What This Does:
 - AWS Systems Manager (SSM): Use SSM to automate patching (via Patch Manager) and upgrading of the EC2 instances running Jenkins. This handles OS updates and potentially Jenkins updates (if scripted).
 - Encrypt EBS Volumes: Enable default encryption for EBS volumes attached to the EC2 instances, ensuring data at rest is encrypted.
- Analysis:
 - Patching/Upgrading: SSM Patch Manager can apply OS patches to EC2 instances during maintenance windows, reducing manual effort. Upgrading Jenkins itself would require additional scripting (e.g., via SSM Run Command), but it's feasible.
 - Artifact Encryption: Build artifacts are typically stored on the Jenkins instance's filesystem (on EBS) before being moved elsewhere (e.g., S3). Encrypting EBS volumes ensures that artifacts are encrypted at rest while on the EC2 instance. However:
 - Artifacts are often ephemeral---Jenkins builds generate them, then they're archived (e.g., to S3 or another storage solution). EBS encryption doesn't cover artifacts once they leave the instance.
 - The question focuses on encrypting the artifacts themselves, not just the storage they temporarily reside on. EBS encryption is a good practice but doesn't directly address artifact encryption in transit or in their final storage location.
 - Maintainability: SSM reduces the patching burden, which is a plus. However, you still manage EC2 instances (e.g., scaling, monitoring, security groups), which isn't the "most maintainable" compared to fully managed services.
- Conclusion: This addresses patching but doesn't fully solve artifact encryption (only on EBS, not in final storage or transit). It also keeps the EC2 maintenance burden. A is incomplete and not the most maintainable.

Option B: Deploy Jenkins to an Amazon ECS cluster and copy build artifacts to an Amazon S3 bucket with default encryption enabled

- What This Does:
 - Jenkins on ECS: Move Jenkins from EC2 to an Amazon ECS cluster (e.g., using Fargate for serverless compute or EC2-based ECS with managed nodes).
 - S3 with Default Encryption: Copy build artifacts to an S3 bucket with default encryption enabled (e.g., using SSE-S3 or SSE-KMS).

- Analysis:
 - Jenkins on ECS:
 - ECS (especially Fargate) reduces maintenance---no need to patch EC2 instances, as AWS manages the underlying infrastructure. For EC2-based ECS, you'd still patch the nodes, but tools like ECS-optimized AMIs simplify this.
 - Jenkins runs in a container, aligning with the company's containerized apps. You'd need to configure Jenkins with persistent storage (e.g., EFS) for build history, but this is manageable.
 - Artifact Encryption:
 - Copying artifacts to S3 is standard for Jenkins (e.g., using the S3 plugin). Enabling default encryption on the S3 bucket (via BucketEncryption in the S3 configuration) ensures artifacts are encrypted at rest with SSE-S3 or SSE-KMS.
 - This fully addresses the encryption requirement---artifacts are encrypted in their final storage location (S3), and S3 encryption is seamless (no extra steps in the build process).
 - Maintainability:
 - ECS (Fargate) eliminates EC2 patching, a big win for maintainability. Even with EC2-based ECS, updates are streamlined.
 - S3 encryption is a set-it-and-forget-it feature---no ongoing management needed.
 - Challenges: Setting up Jenkins on ECS requires initial effort (e.g., configuring the ECS task definition, networking, and storage). You'd also need to ensure Jenkins agents (if used) run in ECS.
- Conclusion: This reduces EC2 maintenance by moving to ECS and encrypts artifacts in S3, meeting both goals. It's highly maintainable with Fargate, though the initial setup is more involved than using a managed build service. B is a strong contender.

Option C: Leverage AWS CodePipeline with a build action and encrypt the artifacts using AWS Secrets Manager

Your pick! Let's break this down:

- What This Does:
 - AWS CodePipeline: Use CodePipeline to orchestrate the build process, replacing or supplementing Jenkins. It includes a build action (e.g., using AWS CodeBuild).
 - Encrypt Artifacts with Secrets Manager: Encrypt the build artifacts using AWS Secrets Manager.
- Analysis:
 - CodePipeline:

- CodePipeline is a CI/CD service that can replace Jenkins for orchestrating builds. It integrates with a source (e.g., CodeCommit, GitHub), a build stage (e.g., CodeBuild), and deployment stages.
- A build action in CodePipeline typically uses CodeBuild, a fully managed build service. CodeBuild eliminates the need to manage Jenkins on EC2---no patching or upgrading required.
- Artifact Encryption:
 - CodePipeline stores build artifacts in S3 (in a pipeline-specific bucket). Artifacts can be encrypted by enabling encryption on the S3 bucket (e.g., SSE-KMS), which CodePipeline supports natively.
 - Secrets Manager for Encryption: This part is confusing. Secrets Manager is for managing secrets (e.g., API keys, passwords), not for encrypting artifacts. You might use Secrets Manager to store a key for manual encryption, but:
 - CodePipeline/CodeBuild doesn't integrate with Secrets Manager for artifact encryption.
 - Artifacts in CodePipeline are already encrypted in S3 (by default with SSE-S3, or SSE-KMS if configured). There's no need to use Secrets Manager to encrypt them separately.
 - If the intent is to encrypt sensitive data within artifacts (e.g., embedding a secret in a build output), Secrets Manager could provide a key, but this isn't standard for "encrypting artifacts" and adds complexity (manual encryption/decryption logic).
- Maintainability:
 - CodePipeline + CodeBuild is highly maintainable---no EC2/Jenkins management, as CodeBuild is fully managed. AWS handles scaling, patching, and upgrades.
 - However, the Secrets Manager approach is a red herring. It's not the right tool for artifact encryption, and it adds unnecessary complexity compared to S3 encryption.
- Replacing Jenkins:
 - The option assumes CodePipeline replaces Jenkins entirely, which requires migrating pipelines to CodePipeline/CodeBuild. This is feasible but involves reconfiguring build jobs.
- Conclusion: CodePipeline + CodeBuild reduces maintenance by eliminating EC2, but the Secrets Manager part is incorrect for artifact encryption. If it said "encrypt artifacts in S3 with KMS," this would be a strong choice. As written, the encryption method is flawed. C has a major flaw in the encryption approach.

Option D: Use AWS CodeBuild with artifact encryption to replace the Jenkins instance running on EC2 instances

- What This Does:

- AWS CodeBuild: Replace Jenkins with CodeBuild, a managed build service, for running builds.
 - Artifact Encryption: Enable encryption for CodeBuild artifacts (stored in S3).
 - Analysis:
 - CodeBuild:
 - CodeBuild is a fully managed build service---no EC2 instances to patch or upgrade. It runs builds in ephemeral containers, pulling source code (e.g., from CodeCommit or S3) and producing artifacts.
 - Replacing Jenkins with CodeBuild eliminates the maintenance burden entirely. You'd need to migrate Jenkins jobs to CodeBuild buildspecs (YAML files defining build steps), but this aligns with the containerized app approach.
 - Artifact Encryption:
 - CodeBuild stores build artifacts in an S3 bucket (specified in the build project). You can enable encryption for these artifacts by configuring the S3 bucket with encryption (e.g., SSE-KMS). In the CodeBuild project configuration, you specify the artifact location and encryption settings:
- json
- ```
artifacts {
 location = "my-build-artifacts-bucket"
 encryption = true
 type = "S3"
}
```
- CodeBuild also encrypts artifacts in transit (using HTTPS to S3). This fully addresses the requirement to encrypt artifacts.
  - Maintainability:
    - CodeBuild is serverless---AWS handles scaling, patching, and infrastructure. No EC2 management, no Jenkins upgrades.
    - S3 encryption for artifacts is automatic once configured, requiring no ongoing effort.
    - Migration effort: Moving from Jenkins to CodeBuild requires rewriting build jobs as buildspecs, but this is a one-time task, and CodeBuild's simplicity pays off long-term.
  - Integration: CodeBuild integrates with CodePipeline (for CI/CD pipelines) or can be used standalone (e.g., triggered by EventBridge or CLI).
- Conclusion: This eliminates EC2 maintenance, uses a managed service, and encrypts artifacts in S3---the most maintainable approach. It directly addresses both requirements without the encryption confusion of C. D looks like the winner.

## Analysis: Is C the Best Choice?

Your pick, C, aims to:

- Use CodePipeline (with a build action, likely CodeBuild) to reduce EC2 maintenance.
- Encrypt artifacts with Secrets Manager.

Why C Falls Short:

- Encryption Issue: Secrets Manager isn't the right tool for artifact encryption. CodePipeline stores artifacts in S3, which supports encryption natively (SSE-S3 or SSE-KMS). Using Secrets Manager to encrypt artifacts would require custom logic (e.g., in CodeBuild) to encrypt/decrypt files, which is complex and not maintainable.
- Maintainability: CodePipeline + CodeBuild is maintainable---no EC2 management, which is great. But the encryption method undermines this.

Comparison to Alternatives:

- A: Addresses patching but not artifact encryption in the final storage (only on EBS). Keeps EC2 maintenance burden.
- B: Moves Jenkins to ECS (reduces maintenance, especially with Fargate) and correctly encrypts artifacts in S3. Very maintainable, though ECS setup is more involved than CodeBuild.
- D: Replaces Jenkins with CodeBuild (fully managed, no EC2), encrypts artifacts in S3 (correctly), and is the simplest long-term solution.

D is the most maintainable:

- CodeBuild eliminates EC2/Jenkins management entirely.
- S3 encryption for artifacts is seamless and AWS-native.
- No custom encryption logic (unlike C's Secrets Manager approach).

B is a close second---ECS (Fargate) also reduces maintenance, and S3 encryption is correct. However, CodeBuild (D) is simpler than managing an ECS cluster, especially for builds, making D the "most maintainable."

Why You Picked C

You likely chose C because:

- You saw CodePipeline as a way to reduce the maintenance burden of Jenkins on EC2, replacing it with a managed CI/CD service.
- You thought Secrets Manager could handle encryption, which makes sense if you interpreted "encrypt artifacts" as needing a key management service. However, Secrets Manager is for secrets, not file encryption---S3 encryption is the better fit.

Your instinct to move away from EC2 to a managed service was spot-on, and CodePipeline/CodeBuild is a great direction. The encryption piece just threw a curveball---Secrets Manager sounds secure, but it's not the

right tool here.

Correct Answer: D. Use AWS CodeBuild with artifact encryption to replace the Jenkins instance running on EC2 instances.

---

**An IT team has built an AWS CloudFormation template so others in the company can quickly and reliably deploy and terminate an application. The template creates an Amazon EC2 instance with a user data script to install the application and an Amazon S3 bucket that the application uses to serve static webpages while it is running. All resources should be removed when the CloudFormation stack is deleted. However, the team observes that CloudFormation reports an error during stack deletion, and the S3 bucket created by the stack is not deleted.**

How can the team resolve the error in the MOST efficient manner to ensure that all resources are deleted without errors?

- A. Add a DeletionPolicy attribute to the S3 bucket resource, with the value Delete forcing the bucket to be removed when the stack is deleted.
- B. Add a custom resource with an AWS Lambda function with the DependsOn attribute specifying the S3 bucket, and an IAM role. Write the Lambda function to delete all objects from the bucket when RequestType is Delete.
- C. Identify the resource that was not deleted. Manually empty the S3 bucket and then delete it.
- D. Replace the EC2 and S3 bucket resources with a single AWS OpsWorks Stacks resource. Define a custom recipe for the stack to create and delete the EC2 instance and the S3 bucket.

Understanding the Problem

- Setup:
  - CloudFormation Template: Creates an EC2 instance and an S3 bucket.
  - EC2 Instance: Uses a user data script to install the application.
  - S3 Bucket: Stores static webpages for the application.
- Requirement: All resources (EC2 instance and S3 bucket) should be deleted when the stack is deleted.
- Issue: During stack deletion, CloudFormation reports an error, and the S3 bucket isn't deleted.
- Goal: Fix the error in the most efficient way, ensuring all resources are deleted cleanly.

Why the S3 Bucket Isn't Deleting:

- CloudFormation deletes resources it creates during stack deletion, but S3 buckets have a specific behavior: they must be empty (no objects) before they can be deleted. If the bucket contains objects (e.g., static webpages uploaded by the application or user), CloudFormation's default deletion fails with an error like "BucketNotEmpty."
- We need to ensure the bucket is empty before CloudFormation attempts to delete it, or force its deletion regardless of contents.

Let's evaluate each option to find the most efficient solution.



Option A: Add a DeletionPolicy attribute to the S3 bucket resource, with the value Delete forcing the bucket to be removed when the stack is deleted

- What This Does:
  - In CloudFormation, the DeletionPolicy attribute controls what happens to a resource during stack deletion. For an S3 bucket (AWS::S3::Bucket), setting DeletionPolicy: Delete seems intuitive, but let's clarify its behavior.
  - By default, CloudFormation tries to delete resources when a stack is deleted (implicit DeletionPolicy: Delete). However, for S3 buckets, this fails if the bucket isn't empty.
  - A related property, ForceDelete, doesn't exist for S3 buckets in CloudFormation. However, there's a common misunderstanding: some think DeletionPolicy: Delete forces deletion, but it doesn't---it's the default behavior.
- Analysis:
  - The actual solution for S3 buckets is the EmptyBucketOnDelete behavior. In CloudFormation, you can use a combination of DeletionPolicy and custom logic (like a custom resource) to empty the bucket, but DeletionPolicy: Delete alone doesn't empty the bucket---it just instructs CloudFormation to delete the resource, which it already tries to do.
  - There's a feature in AWS called "Empty bucket on stack deletion" (introduced in late 2022), where you can enable automatic bucket emptying by opting into a new S3 bucket property in CloudFormation. As of March 2025 (the current date), this is available: you can set LifecycleConfiguration or use a custom resource to empty the bucket, but the option here misrepresents DeletionPolicy: Delete as "forcing" the deletion, which it doesn't do alone.
  - If we interpret this as adding a lifecycle rule to empty the bucket (not explicitly stated), that would work, but the option's wording is misleading.
- Efficiency:
  - If DeletionPolicy: Delete worked as described (forcing deletion by emptying the bucket), this would be efficient---a simple template change. However, it doesn't address the "bucket not empty" error without additional configuration (e.g., a lifecycle rule or custom resource).
- Conclusion: The option's description is incorrect or misleading. DeletionPolicy: Delete doesn't force deletion of a non-empty bucket. A lifecycle rule to empty the bucket would work, but that's not what this says. A is incorrect as written.

Option B: Add a custom resource with an AWS Lambda function with the DependsOn attribute specifying the S3 bucket, and an IAM role. Write the Lambda function to delete all objects from the bucket when RequestType is Delete

Your pick! Let's break this down:

- What This Does:
  - Custom Resource: CloudFormation custom resources allow you to extend functionality using Lambda. The custom resource triggers a Lambda function during stack operations (CREATE,

UPDATE, DELETE).

- DependsOn: The custom resource depends on the S3 bucket, ensuring the bucket exists before the custom resource is processed during creation. During deletion, CloudFormation processes the custom resource before attempting to delete the bucket.
  - Lambda Function: The Lambda function checks the RequestType. When it's Delete (during stack deletion), the function deletes all objects in the S3 bucket.
  - IAM Role: The Lambda function needs an IAM role with permissions to list and delete objects in the S3 bucket (e.g., s3:ListBucket, s3:DeleteObject).
- Analysis:
    - How It Works:
      - In the CloudFormation template, you define a custom resource:

yaml

```
Resources:
 S3Bucket:
 Type: AWS::S3::Bucket
 Properties:
 BucketName: my-app-bucket
 BucketCleanup:
 Type: AWS::CloudFormation::CustomResource
 DependsOn: S3Bucket
 Properties:
 ServiceToken: !GetAtt BucketCleanupLambda.Arn
 BucketName: !Ref S3Bucket
 BucketCleanupLambda:
 Type: AWS::Lambda::Function
 Properties:
 Handler: index.handler
 Role: !GetAtt LambdaExecutionRole.Arn
 Code:
 ZipFile: |
 import json
 import boto3
 import cfnresponse

 def lambda_handler(event, context):
 try:
 request_type = event['RequestType']
 bucket_name = event['ResourceProperties']
['BucketName']
 s3 = boto3.client('s3')

 if request_type == 'Delete':
 # Delete all objects in the bucket
 response =
s3.list_objects_v2(Bucket=bucket_name)
```

```

 if 'Contents' in response:
 objects = [{'Key': obj['Key']} for
obj in response['Contents']]

s3.delete_objects(Bucket=bucket_name, Delete={'Objects':
objects})

 cfnresponse.send(event, context,
cfnresponse.SUCCESS, {})
 else:
 cfnresponse.send(event, context,
cfnresponse.SUCCESS, {})
 except Exception as e:
 cfnresponse.send(event, context,
cfnresponse.FAILED, {'Error': str(e)})

```

- During stack deletion:
  1. CloudFormation processes the custom resource first (before deleting the S3 bucket, due to dependency ordering).
  2. The Lambda function deletes all objects in the bucket.
  3. CloudFormation then deletes the empty bucket successfully.
- Does It Solve the Issue?: Yes! By emptying the bucket before CloudFormation attempts to delete it, this ensures the deletion succeeds without errors.
- Efficiency:
  - Initial Setup: Writing the Lambda function and adding the custom resource to the template takes some effort (a few lines of code and IAM setup).
  - Ongoing: Once added, this solution is automatic---no manual intervention needed for future stack deletions. It fixes the root cause (non-empty bucket) in a CloudFormation-native way.
  - Scalability: Works for any stack deployment, not just a one-time fix.
- Maintainability: The Lambda function is simple (list and delete objects), and custom resources are a standard CloudFormation pattern.
- Conclusion: This directly addresses the deletion error by ensuring the bucket is empty before CloudFormation deletes it. It's efficient in the long term (automatic for all stack deletions), though it requires initial setup. B looks strong.

Option C: Identify the resource that was not deleted. Manually empty the S3 bucket and then delete it

- What This Does:
  - Manually troubleshoot the deletion error by identifying the S3 bucket that failed to delete.
  - Empty the bucket (e.g., via the AWS Console, CLI, or SDK) by deleting all objects.

- Delete the bucket manually.
- Analysis:
  - Does It Solve the Issue?: Yes, for this specific instance. Emptying the bucket allows CloudFormation to retry the deletion (or you can delete the bucket manually, then tell CloudFormation to skip the resource).
  - Efficiency:
    - One-Time Fix: This resolves the current error but doesn't fix the template for future deployments. The next time the stack is deployed and deleted, the same error will occur if the bucket has objects.
    - Manual Effort: Requires manual intervention, which isn't efficient for a DevOps workflow--it defeats the purpose of automation via CloudFormation.
  - Maintainability: This isn't a sustainable solution. The question asks for a fix that ensures "all resources are deleted without errors" in general, not just this one time.
- Conclusion: This is a quick fix for the immediate problem but doesn't address the root cause in the template. It's not efficient for ongoing use. C is incorrect.

Option D: Replace the EC2 and S3 bucket resources with a single AWS OpsWorks Stacks resource. Define a custom recipe for the stack to create and delete the EC2 instance and the S3 bucket

- What This Does:
  - Replace the EC2 and S3 resources in the CloudFormation template with an AWS OpsWorks Stacks resource.
  - Use a custom OpsWorks recipe (e.g., Chef recipes) to create and delete the EC2 instance and S3 bucket.
- Analysis:
  - AWS OpsWorks Stacks: OpsWorks is a configuration management service (using Chef or Puppet) to manage applications and infrastructure. It can manage EC2 instances and run custom recipes during lifecycle events (e.g., setup, shutdown).
  - Creating/Deleting Resources:
    - OpsWorks can launch EC2 instances as part of a stack, and a custom recipe could create an S3 bucket (e.g., via AWS CLI commands in the recipe).
    - During stack deletion, OpsWorks terminates its managed instances, and a custom recipe could delete the S3 bucket (e.g., by emptying it and calling `aws s3 rb`).
  - Does It Solve the Issue?: Potentially. If the recipe empties and deletes the S3 bucket during stack shutdown, it could avoid the deletion error. However:
    - Complexity: OpsWorks is overkill for this simple setup. The template only needs an EC2 instance and S3 bucket---introducing OpsWorks adds a layer of complexity (Chef recipes,

OpsWorks layers, stacks) that's unnecessary for this use case.

- Scope: OpsWorks manages the lifecycle of its own resources (e.g., instances in its stack). Managing an S3 bucket via recipes is possible but not native---OpsWorks isn't designed to manage S3 buckets directly, so you're shoehorning it in.
- Error Handling: If the recipe fails to empty the bucket (e.g., due to permissions or errors), the deletion could still fail, and troubleshooting OpsWorks is harder than a simple Lambda custom resource.
- Efficiency:
  - Initial Effort: Rewriting the entire template to use OpsWorks, learning Chef recipes, and setting up an OpsWorks stack is a massive overhaul compared to a small template change.
  - Ongoing: OpsWorks adds operational overhead (managing stacks, recipes, Chef server) compared to a native CloudFormation solution.
- Conclusion: This solves the problem but in a highly inefficient way. It's a sledgehammer approach for a problem that needs a scalpel. D is incorrect.

Analysis: Is B the Most Efficient?

Your pick, B, uses a custom resource with a Lambda function to empty the S3 bucket during stack deletion:

- Does It Meet Requirements?: Yes. It ensures the bucket is empty before CloudFormation deletes it, preventing the "BucketNotEmpty" error. All resources (EC2 instance and S3 bucket) are deleted cleanly.
- Efficiency:
  - Initial Setup: Writing a small Lambda function (list and delete objects) and adding a custom resource to the template takes a bit of effort---about 20-30 lines of code and IAM setup.
  - Ongoing: Once added, it's automatic for all stack deletions. No manual intervention, no recurring errors.
  - Scalability: Works for any future deployments of the stack.
- Maintainability: The Lambda function is simple and reusable. Custom resources are a standard CloudFormation pattern for handling edge cases like this.
- Comparison to Alternatives:
  - A: Misrepresents DeletionPolicy: Delete. If it meant adding a lifecycle rule to empty the bucket, that'd be efficient too, but the wording is wrong.
  - C: A manual fix---quick for this instance but not a solution for the template. Inefficient for ongoing use.
  - D: OpsWorks is a massive overcomplication, requiring a rewrite of the entire template and adding operational overhead.

B is the most efficient:

- It's a one-time template change that fixes the issue permanently.
- It's CloudFormation-native (custom resource + Lambda is a common pattern).
- It requires minimal effort compared to overhauls (D) or manual fixes (C).

A Potential Alternative (Not Listed):

- As of 2022, CloudFormation supports an S3 bucket property to empty the bucket on deletion. You can add a lifecycle rule to the bucket to delete all objects when the stack is deleted:

yaml

```
Resources:
 S3Bucket:
 Type: AWS::S3::Bucket
 Properties:
 BucketName: my-app-bucket
 LifecycleConfiguration:
 Rules:
 - Id: EmptyBucketOnDelete
 Status: Enabled
 ExpirationInDays: 1 # Ensures objects are deleted quickly
```

- This would be the simplest fix---just a few lines in the template---but it's not an option here. B achieves the same result via a custom resource.

Why You Picked B

You likely chose B because:

- You recognized that the S3 bucket deletion fails because it's not empty---a common CloudFormation issue.
- You knew CloudFormation can't empty buckets by default, so you need to intervene before deletion.
- You saw that a custom resource with Lambda can empty the bucket during stack deletion, ensuring a clean removal. The DependsOn attribute ensures proper ordering (empty bucket before deletion attempt).
- You understood that this is a template-level fix, making it efficient for all future stack deletions, not just a one-time manual fix.

Your reasoning aligns perfectly with AWS best practices---using a custom resource to handle S3 bucket cleanup is a standard solution for this exact problem.

Correct Answer: B. Add a custom resource with an AWS Lambda function with the DependsOn attribute specifying the S3 bucket, and an IAM role. Write the Lambda function to delete all objects from the bucket when RequestType is Delete.

---

**A company runs an application on one Amazon EC2 instance. Application metadata is stored in Amazon S3 and must be retrieved if the instance is restarted. The instance must restart or relaunch automatically if the instance becomes unresponsive.**

Which solution will meet these requirements?

- A. Create an Amazon CloudWatch alarm for the StatusCheckFailed metric. Use the recover action to stop and start the instance. Use an S3 event notification to push the metadata to the instance when the instance is back up and running.
- B. Configure AWS OpsWorks, and use the auto healing feature to stop and start the instance. Use a lifecycle event in OpsWorks to pull the metadata from Amazon S3 and update it on the instance.
- C. Use EC2 Auto Recovery to automatically stop and start the instance in case of a failure. Use an S3 event notification to push the metadata to the instance when the instance is back up and running.
- D. Use AWS CloudFormation to create an EC2 instance that includes the UserData property for the EC2 resource. Add a command in UserData to retrieve the application metadata from Amazon S3.

### Understanding the Problem

- Setup:
  - Single EC2 Instance: The application runs on one EC2 instance.
  - Metadata in S3: Application metadata is stored in an S3 bucket and needs to be retrieved if the instance restarts.
  - Unresponsiveness: The instance must automatically restart or relaunch if it becomes unresponsive.
- Requirements:
  1. Automatic Recovery: If the instance becomes unresponsive, it should automatically restart or relaunch.
  2. Metadata Retrieval: After a restart or relaunch, the instance must retrieve the metadata from S3.
- Key Considerations:
  - Unresponsiveness could mean the instance fails health checks (e.g., Status Check failures in AWS, like a system failure or loss of network connectivity).
  - Metadata retrieval implies the instance needs to pull the data from S3 post-restart, likely during startup.
  - The solution should be simple and reliable for a single instance, without overcomplicating the architecture.

Let's evaluate each option to see which one meets both requirements effectively.

Option A: Create an Amazon CloudWatch alarm for the StatusCheckFailed metric. Use the recover action to stop and start the instance. Use an S3 event notification to push the metadata to the instance when the

instance is back up and running

- What This Does:
  - CloudWatch Alarm for StatusCheckFailed:
    - EC2 instances have two status checks: System Status (checks the underlying hardware/network) and Instance Status (checks the instance's OS/software).
    - The StatusCheckFailed metric triggers if either check fails (e.g., StatusCheckFailed\_System or StatusCheckFailed\_Instance).
    - A CloudWatch alarm can monitor this metric and take an action when it fails.
  - Recover Action:
    - The "recover" action for EC2 (available since 2017) automatically stops and restarts the instance on the same underlying hardware (if possible). If the hardware is the issue, it might move the instance to new hardware.
    - This is a lightweight recovery mechanism for EC2 instances, designed for scenarios like this where you want to recover a single instance.
  - S3 Event Notification:
    - S3 event notifications can trigger actions (e.g., Lambda, SQS, SNS) when objects are created/updated in a bucket.
    - The idea here is to "push" metadata to the instance after it restarts, using an S3 event notification.
- Analysis:
  - Automatic Recovery:
    - The CloudWatch alarm with the recover action meets the requirement for automatic restart/relaunch. If the instance becomes unresponsive (e.g., fails system or instance status checks), the alarm triggers, and AWS attempts to recover the instance.
    - Recovery typically takes a few minutes and preserves the instance's ID, IP addresses, and EBS volumes.
  - Metadata Retrieval:
    - The S3 event notification part is problematic. S3 events are triggered by bucket actions (e.g., object creation), not by instance state changes. There's no direct way for an S3 event to "push" metadata to an instance when it restarts.
    - Possible interpretation:
      - After the instance restarts, a script on the instance could upload a "restart marker" object to S3, triggering an S3 event.



- The S3 event could invoke a Lambda function to "push" the metadata to the instance (e.g., via SSM Run Command or by writing to a known location the instance polls).
- This is convoluted and unreliable:
  - The instance would need to know when to upload the marker object (requires user data scripting).
  - "Pushing" metadata to an instance isn't straightforward---SSM Run Command could work, but it's complex and assumes the instance is reachable.
  - A simpler approach would be for the instance to pull the metadata from S3 during startup (e.g., via user data), not rely on S3 events to push.
- Does It Meet Requirements?:
  - Recovery: Yes, the recover action handles unresponsiveness.
  - Metadata: No, S3 event notifications aren't a practical way to push metadata to an instance. It overcomplicates what should be a simple pull operation.
- Conclusion: The recovery part works, but the metadata retrieval is flawed. A is not the best solution.

Option B: Configure AWS OpsWorks, and use the auto healing feature to stop and start the instance. Use a lifecycle event in OpsWorks to pull the metadata from Amazon S3 and update it on the instance

Your pick! Since you're not familiar with OpsWorks, I'll explain it thoroughly.

- What This Does:
  - AWS OpsWorks:
    - AWS OpsWorks Stacks is a configuration management service that uses Chef or Puppet to manage instances and applications. It organizes instances into stacks and layers (e.g., a web server layer).
    - You define a stack (e.g., "App Stack"), a layer (e.g., "Web Layer"), and add the EC2 instance to the layer. OpsWorks manages the instance's lifecycle.
  - Auto Healing:
    - OpsWorks has an "auto healing" feature: if an instance fails health checks (e.g., it stops responding to OpsWorks agent pings), OpsWorks automatically terminates and relaunches the instance.
    - This is similar to EC2 Auto Scaling's health checks but managed by OpsWorks.
  - Lifecycle Event:
    - OpsWorks supports lifecycle events (e.g., setup, configure, deploy, undeploy, shutdown) that trigger Chef recipes on the instance.
    - The setup event runs after an instance boots, making it ideal for initialization tasks.

- You can write a Chef recipe to pull metadata from S3 (e.g., using the AWS CLI: `aws s3 cp s3://my-bucket/metadata.json /path/to/metadata`) and update the instance (e.g., place the file in the right directory or update app config).
- Analysis:
  - Automatic Recovery:
    - OpsWorks auto healing meets the requirement. If the instance becomes unresponsive, OpsWorks detects the failure (via its agent) and relaunches the instance. This is a straightforward way to handle unresponsiveness for a single instance.
    - Note: OpsWorks relaunches the instance (new instance ID), unlike EC2 recovery (which tries to preserve the ID). This still meets the "restart or relaunch" requirement.
  - Metadata Retrieval:
    - Using a lifecycle event (e.g., setup) to pull metadata from S3 is a clean approach. The Chef recipe runs automatically after the instance boots:

ruby

```
Chef recipe (e.g., recipes/setup.rb)
execute 'download_metadata' do
 command 'aws s3 cp s3://my-bucket/metadata.json
/app/metadata.json'
 action :run
end
```

- This ensures the metadata is retrieved post-restart, fulfilling the requirement. The instance pulls the data directly---no need for complex event notifications.
- Does It Meet Requirements?:
  - Recovery: Yes, auto healing handles unresponsiveness.
  - Metadata: Yes, the lifecycle event pulls metadata from S3 during startup.
- Drawbacks:
  - Complexity: OpsWorks is a heavyweight solution for a single EC2 instance. It requires setting up a stack, layer, and Chef recipes, which is overkill for this simple use case. OpsWorks is better suited for managing fleets of instances or complex applications with multiple layers (e.g., web, app, DB).
  - Learning Curve: Since you're not familiar with OpsWorks, it's worth noting there's a learning curve---understanding stacks, layers, and Chef recipes takes time.
  - Overhead: OpsWorks adds operational complexity (e.g., managing the OpsWorks agent, Chef server) compared to simpler EC2-native solutions.

- Conclusion: This meets both requirements but isn't the simplest approach. OpsWorks works, but it's more complex than necessary for a single instance. B is viable but not the most efficient.

Option C: Use EC2 Auto Recovery to automatically stop and start the instance in case of a failure. Use an S3 event notification to push the metadata to the instance when the instance is back up and running

- What This Does:
  - EC2 Auto Recovery:
    - This refers to the same mechanism as A's "recover" action but via a different setup. You can enable EC2 Auto Recovery by creating a CloudWatch alarm on the `StatusCheckFailed_System` metric with the recover action.
    - It automatically stops and restarts the instance if it fails system status checks (e.g., hardware failure, network issues).
  - S3 Event Notification: Same as A---use an S3 event to "push" metadata to the instance after it restarts.
- Analysis:
  - Automatic Recovery:
    - EC2 Auto Recovery works for unresponsiveness. It's effectively the same as A's CloudWatch alarm with the recover action, just described differently.
    - It restarts the instance on the same hardware (or new hardware if needed), preserving the instance ID and IPs.
  - Metadata Retrieval:
    - As in A, the S3 event notification approach is flawed. S3 events are triggered by bucket actions, not instance restarts. "Pushing" metadata to an instance via an S3 event is impractical---it would require the instance to trigger the event (e.g., by uploading an object) and then a mechanism (e.g., Lambda + SSM) to push data back, which is overly complex.
    - A simpler approach is for the instance to pull the metadata itself during startup.
  - Does It Meet Requirements?:
    - Recovery: Yes, EC2 Auto Recovery handles unresponsiveness.
    - Metadata: No, the S3 event notification method is impractical.
- Conclusion: Same as A---the recovery part works, but the metadata retrieval is flawed. C is not the best solution.

Option D: Use AWS CloudFormation to create an EC2 instance that includes the `UserData` property for the EC2 resource. Add a command in `UserData` to retrieve the application metadata from Amazon S3

- What This Does:

- CloudFormation Template:
  - Define an EC2 instance in CloudFormation with a UserData script.
  - The UserData script runs on instance boot and includes a command to pull metadata from S3.
- UserData:

- Example script in the CloudFormation template:

yaml

```
Resources:
 EC2Instance:
 Type: AWS::EC2::Instance
 Properties:
 ImageId: ami-12345678
 InstanceType: t3.micro
 UserData:
 Fn::Base64: |
 #!/bin/bash
 aws s3 cp s3://my-bucket/metadata.json
/app/metadata.json
 # Restart the application if needed
 systemctl restart my-app
```

- UserData runs on every instance boot, including after a restart or relaunch.
- Analysis:
  - Automatic Recovery:
    - This option doesn't address the recovery requirement. CloudFormation creates the instance but doesn't handle unresponsiveness. You'd need to add a recovery mechanism (e.g., a CloudWatch alarm as in A/C, or use Auto Scaling with a single instance).
    - Without recovery, this fails the "restart or relaunch automatically" requirement.
  - Metadata Retrieval:
    - The UserData script is a perfect way to pull metadata from S3. On boot (including after a restart), the instance runs the script, downloads the metadata, and updates the application.
    - This is simple and reliable---no need for S3 event notifications or external triggers.
  - Does It Meet Requirements?:
    - Recovery: No, CloudFormation alone doesn't handle unresponsiveness.
    - Metadata: Yes, UserData handles metadata retrieval cleanly.

- Conclusion: This only solves half the problem. While the metadata retrieval is spot-on, it misses the automatic recovery requirement. D is incomplete.

Analysis: Is B the Best Choice?

Your pick, B, uses AWS OpsWorks with auto healing and a lifecycle event:

- Does It Meet Requirements?:
  - Recovery: Yes, OpsWorks auto healing stops and starts the instance if it becomes unresponsive.
  - Metadata: Yes, the setup lifecycle event pulls metadata from S3 post-restart.
- Is It the Most Logical?:
  - You chose B because it seemed logical---it addresses both requirements in one solution. OpsWorks handles recovery (via auto healing) and metadata retrieval (via lifecycle event), making it a cohesive approach.
  - However, OpsWorks is overkill for a single instance. It's designed for managing fleets of instances with complex configurations, not a simple single-instance app.

Comparison to Alternatives:

- A and C: Both use EC2 Auto Recovery (via CloudWatch alarm), which is a simpler recovery mechanism for a single instance. However, their S3 event notification approach for metadata is impractical.
- D: Nails metadata retrieval with UserData but misses recovery entirely.

What's Missing?:

- The ideal solution would combine the simplicity of EC2 Auto Recovery (from A/C) with the metadata retrieval of UserData (from D). For example:
  - Use CloudFormation to create the EC2 instance with a UserData script to pull metadata (like D).
  - Add a CloudWatch alarm with the recover action to handle unresponsiveness (like A/C).
- This isn't an option, so we must pick the best available.

Why B Isn't Optimal:

- OpsWorks adds unnecessary complexity for a single instance. Setting up a stack, layer, and Chef recipes is more effort than needed compared to EC2 Auto Recovery + UserData.
- A simpler solution (not listed) would be better, but among the options, B is the only one that fully meets both requirements.

Best Available Choice:

- B is the only option that addresses both requirements, even if it's overkill. A and C fail on metadata retrieval, and D fails on recovery.

Correct Answer: B. Configure AWS OpsWorks, and use the auto healing feature to stop and start the instance. Use a lifecycle event in OpsWorks to pull the metadata from Amazon S3 and update it on the instance.

### Why You Picked B

You chose B because it logically addresses both requirements:

- Recovery: OpsWorks auto healing restarts the instance if unresponsive.
- Metadata: The lifecycle event pulls metadata from S3 post-restart.

Your reasoning makes sense---B is the only option that fully meets the requirements in one package. You weren't familiar with OpsWorks, but you recognized it as a tool that could handle both recovery and metadata retrieval, which shows great intuition! OpsWorks does work here, but in a real-world scenario, you'd likely use a simpler approach (e.g., EC2 Auto Recovery + UserData) to avoid the overhead of OpsWorks for a single instance.

### More on AWS OpsWorks (Since You're Unfamiliar)

- What Is OpsWorks?:
  - OpsWorks Stacks is a configuration management service that helps you manage applications and infrastructure using Chef or Puppet.
  - You organize resources into stacks (a collection of resources) and layers (e.g., a web layer, app layer).
  - Each layer can manage EC2 instances, and OpsWorks runs a Chef agent on those instances to execute recipes during lifecycle events.
- Key Features:
  - Auto Healing: If an instance fails health checks (e.g., the OpsWorks agent stops responding), OpsWorks relaunches it.
  - Lifecycle Events: Events like setup (post-boot), configure (config changes), and deploy (app deployment) trigger Chef recipes to automate tasks.
- Use Case in This Question:
  - You'd create a stack with a single layer (e.g., "App Layer"), add the EC2 instance to it, enable auto healing, and write a Chef recipe to pull metadata from S3 during the setup event.
- Why It's Overkill:
  - OpsWorks is designed for complex apps with multiple instances/layers (e.g., web servers, app servers, databases). For a single EC2 instance, simpler tools like EC2 Auto Recovery and CloudFormation are more appropriate.

---

**A developer is maintaining a fleet of 50 Amazon EC2 Linux servers. The servers are part of an Amazon EC2 Auto Scaling group, and also use Elastic Load Balancing for load balancing. Occasionally, some**

**application servers are being terminated after failing ELB HTTP health checks. The developer would like to perform a root cause analysis on the issue, but before being able to access application logs, the server is terminated.**

How can log collection be automated?

A. Use Auto Scaling lifecycle hooks to put instances in a Pending:Wait state. Create an Amazon CloudWatch alarm for EC2 Instance Terminate Successful and trigger an AWS Lambda function that invokes an SSM Run Command script to collect logs, push them to Amazon S3, and complete the lifecycle action once logs are collected.

B. Use Auto Scaling lifecycle hooks to put instances in a Terminating:Wait state. Create an AWS Config rule for EC2 Instance-terminate Lifecycle Action and trigger a step function that invokes a script to collect logs, push them to Amazon S3, and complete the lifecycle action once logs are collected.

C. Use Auto Scaling lifecycle hooks to put instances in a Terminating:Wait state. Create an Amazon CloudWatch subscription filter for EC2 Instance Terminate Successful and trigger a CloudWatch agent that invokes a script to collect logs, push them to Amazon S3, and complete the lifecycle action once logs are collected.

D. Use Auto Scaling lifecycle hooks to put instances in a Terminating:Wait state. Create an Amazon EventBridge rule for EC2 Instance-terminate Lifecycle Action and trigger an AWS Lambda function that invokes an SSM Run Command script to collect logs, push them to Amazon S3, and complete the lifecycle action once logs are collected.

Understanding the Problem

- Setup:
  - EC2 Instances: 50 Linux servers in an Auto Scaling group.
  - Elastic Load Balancing (ELB): The ASG integrates with ELB, which performs HTTP health checks on the instances.
  - Health Check Failures: Some instances fail ELB HTTP health checks, causing the ASG to mark them as unhealthy and terminate them.
  - Log Collection Issue: Instances are terminated before the developer can access application logs for root cause analysis.
- Requirements:
  - Automate log collection before the instance is terminated.
  - Ensure logs are preserved (e.g., by pushing them to a durable location like S3).
  - Allow the termination to proceed after log collection.
- Key Mechanism:
  - When an instance fails ELB health checks, the ASG marks it unhealthy and initiates termination.
  - We need to intercept the termination process, collect logs, and then allow the termination to complete.

### Auto Scaling Termination Process:

- When an instance is marked unhealthy (e.g., by ELB), the ASG starts the termination process.
- By default, the instance is terminated immediately, and logs on the instance (e.g., in /var/log) are lost unless preserved.

### Solution Approach:

- Use Auto Scaling lifecycle hooks to pause the termination process, giving time to collect logs.
- Trigger an action during this pause to collect logs, push them to S3, and then complete the termination.

Let's evaluate each option to find the best approach.

Option A: Use Auto Scaling lifecycle hooks to put instances in a Pending:Wait state. Create an Amazon CloudWatch alarm for EC2 Instance Terminate Successful and trigger an AWS Lambda function that invokes an SSM Run Command script to collect logs, push them to Amazon S3, and complete the lifecycle action once logs are collected

- What This Does:
  - Lifecycle Hooks:
    - Lifecycle hooks allow you to pause an ASG instance during state transitions (e.g., launching or terminating).
    - Pending:Wait is a state during instance launch (not termination). The correct state for termination is Terminating:Wait.
    - This option incorrectly uses Pending:Wait, which applies when an instance is launching, not terminating. We need to catch the instance before it terminates, so this is a red flag.
  - CloudWatch Alarm:
    - The alarm triggers on the EC2 Instance Terminate Successful metric, which indicates an instance has already terminated.
    - If the instance is already terminated, it's too late to collect logs---the instance is gone, and its EBS volumes (if not set to persist) are deleted.
  - Lambda and SSM Run Command:
    - The Lambda function invokes an SSM Run Command script to collect logs (e.g., `aws s3 cp /var/log/app.log s3://my-bucket/logs/`) and push them to S3.
    - SSM Run Command requires the instance to be running and the SSM Agent to be active, which isn't possible if the instance is already terminated.
  - Completing the Lifecycle Action:
    - The Lambda would complete the lifecycle action (e.g., using `CompleteLifecycleAction`), but since the hook state is wrong, this step is irrelevant.



- Analysis:
  - Lifecycle Hook State: Pending:Wait is incorrect for termination---it's for launching instances. The correct state is Terminating:Wait.
  - Trigger Timing: EC2 Instance Terminate Successful happens after termination, so logs can't be collected at that point.
  - Does It Meet Requirements?: No. The hook state is wrong, and the trigger is too late.
- Conclusion: This option fails due to the incorrect lifecycle state and late trigger. A is incorrect.

Option B: Use Auto Scaling lifecycle hooks to put instances in a Terminating:Wait state. Create an AWS Config rule for EC2 Instance-terminate Lifecycle Action and trigger a step function that invokes a script to collect logs, push them to Amazon S3, and complete the lifecycle action once logs are collected

Your pick! Let's break this down:

- What This Does:
  - Lifecycle Hooks:
    - Terminating:Wait is the correct state. When an instance is marked for termination (e.g., due to failing ELB health checks), the lifecycle hook pauses the instance in the Terminating:Wait state.
    - During this state, the instance is still running, giving us time to collect logs.
    - You configure the hook to send a notification (e.g., to EventBridge or an SNS topic) when the instance enters this state.
  - AWS Config Rule:
    - AWS Config rules evaluate the compliance of resources against desired configurations.
    - The rule here is for EC2 Instance-terminate Lifecycle Action, implying it triggers when a lifecycle action occurs (e.g., termination).
    - AWS Config rules can integrate with EventBridge to react to events, but Config rules are typically for compliance checks (e.g., "is an instance tagged correctly?"), not for reacting to lifecycle events.
    - Auto Scaling lifecycle hooks emit events to EventBridge (e.g., autoscaling:EC2\_INSTANCE\_TERMINATING), but AWS Config isn't the right mechanism to catch these events---EventBridge or CloudWatch Events rules are.
  - Step Function:
    - The Config rule triggers an AWS Step Function, which orchestrates a workflow to:
      1. Invoke a script to collect logs (e.g., via SSM Run Command or a Lambda invoking a script).
      2. Push logs to S3.

### 3. Complete the lifecycle action (e.g., using CompleteLifecycleAction API).

- Analysis:
  - Lifecycle Hook: Terminating:Wait is correct. The instance pauses before termination, giving us a window to act.
  - Trigger Mechanism:
    - AWS Config rules aren't designed to react to lifecycle events like EC2\_INSTANCE\_TERMINATING. Config rules evaluate resource state changes (e.g., via CloudTrail logs), but lifecycle hooks emit events directly to EventBridge.
    - The correct approach is to use an EventBridge rule to catch the lifecycle event (autoscaling:EC2\_INSTANCE\_TERMINATING), not AWS Config.
    - If we stretch this, AWS Config could theoretically detect a state change (via CloudTrail), but it's not the idiomatic way to handle lifecycle events, and Config rules don't directly "trigger" on lifecycle actions---they evaluate compliance.
  - Step Function:
    - A Step Function is a good orchestrator for this workflow:
      - Step 1: Invoke SSM Run Command to collect logs (e.g., /var/log/app.log).
      - Step 2: Push logs to S3.
      - Step 3: Complete the lifecycle action.
    - Step Functions add reliability (e.g., retries, error handling), but they're overkill for a simple task that a single Lambda can handle.
  - Does It Meet Requirements?:
    - Log Collection: Yes, if we assume the script collects logs while the instance is in Terminating:Wait.
    - Preservation: Yes, logs are pushed to S3.
    - Termination: Yes, the lifecycle action is completed.
  - Issue: The AWS Config rule is the wrong trigger. Lifecycle hooks emit events to EventBridge, not AWS Config. This makes the solution technically incorrect as written.
- Conclusion: The lifecycle hook state is correct, and the Step Function could work, but AWS Config isn't the right way to detect lifecycle events. B is incorrect due to the Config rule.

Option C: Use Auto Scaling lifecycle hooks to put instances in a Terminating:Wait state. Create an Amazon CloudWatch subscription filter for EC2 Instance Terminate Successful and trigger a CloudWatch agent that invokes a script to collect logs, push them to Amazon S3, and complete the lifecycle action once logs are collected

- What This Does:

- Lifecycle Hooks: Terminating:Wait is correct, pausing the instance before termination.
- CloudWatch Subscription Filter:
  - A subscription filter in CloudWatch Logs typically filters log events and streams them to a destination (e.g., Lambda).
  - EC2 Instance Terminate Successful isn't a log event---it's a CloudWatch metric (from A), not something you'd find in CloudWatch Logs.
  - Even if we interpret this as a CloudWatch Logs event (e.g., from CloudTrail logging ASG actions), the "Terminate Successful" event happens after termination---too late to collect logs.
- CloudWatch Agent:
  - The CloudWatch agent (Unified CloudWatch Agent) collects logs and metrics from instances and sends them to CloudWatch Logs.
  - Here, it's supposed to invoke a script to collect logs, but if the instance is already terminated (per the trigger), the agent can't run.
- Analysis:
  - Trigger Timing: EC2 Instance Terminate Successful is too late---logs are gone.
  - Subscription Filter: This is the wrong mechanism. Subscription filters are for log streams, not lifecycle events or metrics.
  - CloudWatch Agent: The agent can't collect logs from a terminated instance.
- Conclusion: The lifecycle hook state is correct, but the trigger and agent usage are wrong. C is incorrect.

Option D: Use Auto Scaling lifecycle hooks to put instances in a Terminating:Wait state. Create an Amazon EventBridge rule for EC2 Instance-terminate Lifecycle Action and trigger an AWS Lambda function that invokes an SSM Run Command script to collect logs, push them to Amazon S3, and complete the lifecycle action once logs are collected

- What This Does:
  - Lifecycle Hooks: Terminating:Wait is correct, pausing the instance before termination.
  - EventBridge Rule:
    - When an instance enters Terminating:Wait, the lifecycle hook emits an event to EventBridge (or CloudWatch Events, its predecessor).
    - The event is autoscaling:EC2\_INSTANCE\_TERMINATING, indicating the instance is about to terminate.
    - An EventBridge rule can match this event:

json

```
{
 "source": ["aws.autoscaling"],
 "detail-type": ["EC2 Instance-terminate Lifecycle Action"]
}
```

- Lambda Function:
  - The Lambda is triggered by the EventBridge rule.
  - It extracts the instance ID from the event payload.
  - It invokes an SSM Run Command script (e.g., AWS-RunShellScript) to:
    1. Collect logs (e.g., `cp /var/log/app.log /tmp/`).
    2. Push logs to S3 (e.g., `aws s3 cp /tmp/app.log s3://my-bucket/logs/`).
  - It completes the lifecycle action using the Auto Scaling API (`CompleteLifecycleAction`), allowing termination to proceed.
- Analysis:
  - Lifecycle Hook: Correct state (`Terminating:Wait`).
  - Trigger: EventBridge is the right tool. Lifecycle hooks emit events to EventBridge, and the EC2 Instance-terminate Lifecycle Action event is exactly what we need.
  - Log Collection:
    - SSM Run Command runs a script on the instance while it's still in `Terminating:Wait` (instance is running, SSM Agent is active).
    - The script collects logs and pushes them to S3.
  - Completing the Action: Lambda calls `CompleteLifecycleAction`, allowing the ASG to proceed with termination.
  - Does It Meet Requirements?:
    - Log Collection: Yes, SSM Run Command collects logs before termination.
    - Preservation: Yes, logs are pushed to S3.
    - Termination: Yes, the lifecycle action is completed.
  - Efficiency:
    - EventBridge → Lambda → SSM Run Command is a standard AWS pattern for this use case.
    - Minimal setup: create a lifecycle hook, an EventBridge rule, and a Lambda function (with IAM permissions for SSM and ASG).

- Conclusion: This is the cleanest, most AWS-native solution. It uses the right trigger (EventBridge), a reliable execution mechanism (SSM Run Command), and completes the lifecycle action. D looks like the winner.

Analysis: Is B the Best Choice?

Your pick, B, uses:

- Auto Scaling lifecycle hooks (Terminating:Wait)---correct.
- AWS Config rule to trigger a Step Function---incorrect trigger mechanism.
- Step Function to collect logs and complete the action---works but overkill.

Why B Isn't the Best:

- Trigger Issue: AWS Config isn't the right way to catch lifecycle events. Auto Scaling emits EC2\_INSTANCE\_TERMINATING events to EventBridge, not AWS Config. Config rules are for compliance (e.g., "is this resource compliant?"), not for reacting to lifecycle actions.
- Complexity: Step Functions are great for orchestration but unnecessary here. A single Lambda (as in D) can handle this task more simply.

Comparison to Alternatives:

- A: Wrong lifecycle state (Pending:Wait) and wrong trigger timing (post-termination).
- C: Wrong trigger (CloudWatch subscription filter) and timing (post-termination).
- D: Correct lifecycle state, correct trigger (EventBridge), and efficient execution (Lambda + SSM).

D is the best choice:

- It uses EventBridge to catch the lifecycle event, which is the standard approach.
- Lambda + SSM Run Command is a simple, reliable way to collect logs.
- It minimizes complexity while meeting all requirements.

Why You Picked B

You chose B because it seemed the most logical:

- Lifecycle Hook: You correctly identified that Terminating:Wait is the right state to pause termination and collect logs.
- AWS Config Rule: You might have thought Config could monitor lifecycle actions, which is a reasonable guess if you're not familiar with how lifecycle hooks integrate with EventBridge. Config rules do evaluate resource changes, but lifecycle hooks emit events directly to EventBridge, making D the better fit.
- Step Function: You likely saw this as a way to orchestrate the log collection process, which is a good instinct for reliability, but a Lambda (as in D) is simpler for this task.

Your reasoning was solid---you got the lifecycle hook right, which is the core mechanism. The trigger (AWS Config vs. EventBridge) was the tricky part, and now you know EventBridge is the go-to for lifecycle events.

Correct Answer: D. Use Auto Scaling lifecycle hooks to put instances in a Terminating:Wait state. Create an Amazon EventBridge rule for EC2 Instance-terminate Lifecycle Action and trigger an AWS Lambda function that invokes an SSM Run Command script to collect logs, push them to Amazon S3, and complete the lifecycle action once logs are collected.

---

**A company has migrated its container-based applications to Amazon EKS and want to establish automated email notifications. The notifications sent to each email address are for specific activities related to EKS components. The solution will include Amazon SNS topics and an AWS Lambda function to evaluate incoming log events and publish messages to the correct SNS topic.**

Which logging solution will support these requirements?

- A. Enable Amazon CloudWatch Logs to log the EKS components. Create a CloudWatch subscription filter for each component with Lambda as the subscription feed destination.
- B. Enable Amazon CloudWatch Logs to log the EKS components. Create CloudWatch Logs Insights queries linked to Amazon EventBridge events that invoke Lambda.
- C. Enable Amazon S3 logging for the EKS components. Configure an Amazon CloudWatch subscription filter for each component with Lambda as the subscription feed destination.
- D. Enable Amazon S3 logging for the EKS components. Configure S3 PUT Object event notifications with AWS Lambda as the destination.

Understanding the Problem

- Setup:
  - Amazon EKS: The company's applications run on Amazon EKS (Elastic Kubernetes Service), a managed Kubernetes service.
  - EKS Components: These likely include EKS cluster components like the control plane, worker nodes, kube-apiserver, kube-controller-manager, or application-level logs (e.g., pod logs).
  - Notifications: Email notifications are sent to specific email addresses based on activities related to EKS components.
  - SNS and Lambda:
    - SNS topics will send notifications (e.g., one topic per component or activity type, with email subscriptions).
    - A Lambda function will evaluate incoming log events and publish messages to the appropriate SNS topic.
- Requirements:
  - Log EKS component activities.
  - Process these logs to identify specific events.

- Route events to a Lambda function for evaluation.
- Lambda publishes messages to the correct SNS topic for email notifications.
- Key Considerations:
  - EKS logging: What logs are available, and where are they stored?
  - How do we get logs to the Lambda function for processing?
  - The solution must support filtering logs per component and triggering Lambda in near real time.

Let's evaluate each option to find the best logging solution.

Option A: Enable Amazon CloudWatch Logs to log the EKS components. Create a CloudWatch subscription filter for each component with Lambda as the subscription feed destination

Your pick! Let's break this down:

- What This Does:
  - CloudWatch Logs for EKS:
    - EKS integrates with Amazon CloudWatch Logs to capture logs from various components:
      - Control Plane Logs: EKS can send control plane logs (e.g., API server, scheduler, controller manager) to CloudWatch Logs. You enable this in the EKS cluster settings (e.g., `enabledClusterLogTypes: ["api", "scheduler"]`).
      - Pod Logs: Pods running on EKS worker nodes can send logs to CloudWatch Logs using a logging agent like Fluentd or Fluent Bit (via the `aws-observability/aws-for-fluent-bit` Helm chart or a DaemonSet).
      - Worker Node Logs: System logs (e.g., kubelet, container runtime) can also be sent to CloudWatch Logs via the CloudWatch Agent.
    - Logs are organized into log groups (e.g., `/aws/eks//cluster` for control plane logs, or custom log groups for pod logs).
  - CloudWatch Subscription Filter:
    - A subscription filter in CloudWatch Logs allows you to stream log events in near real time to a destination (e.g., Lambda, Kinesis, or another service).
    - You create a filter for each EKS component (e.g., one for API server logs, one for scheduler logs, one for pod logs of a specific app).
    - The filter pattern matches log events for that component (e.g., `{"component": "kube-apiserver"}` or error keywords like `Failed`).
    - The destination is the Lambda function, which receives the log events, evaluates them, and publishes to the appropriate SNS topic.

- Lambda and SNS:
  - Lambda processes each log event, determines the activity (e.g., "pod failed to start"), and sends a message to the corresponding SNS topic.
  - SNS topics are subscribed to email addresses, sending notifications.
- Analysis:
  - Logging EKS Components:
    - CloudWatch Logs is the standard logging solution for EKS. Control plane logs are natively supported, and pod/worker logs can be sent via agents (Fluentd/Fluent Bit or CloudWatch Agent).
    - This meets the requirement to log EKS components.
  - Filtering per Component:
    - Subscription filters allow you to define patterns for each component (e.g., filter API server logs separately from scheduler logs).
    - You can create multiple filters within the same log group (e.g., /aws/eks//cluster) or across different log groups (e.g., one per app).
  - Triggering Lambda:
    - Subscription filters stream logs to Lambda in near real time (typically within seconds), which is ideal for automated notifications.
    - Lambda can process the events, decide which SNS topic to publish to, and send the notification.
  - Does It Meet Requirements?:
    - Log Collection: Yes, CloudWatch Logs captures EKS component logs.
    - Processing: Yes, subscription filters route logs to Lambda for evaluation.
    - Notifications: Yes, Lambda publishes to SNS topics for email notifications.
  - Efficiency:
    - This is a standard AWS pattern: CloudWatch Logs → Subscription Filter → Lambda → SNS.
    - It's scalable (handles large log volumes) and near real-time (seconds delay).
- Conclusion: This looks like a strong solution. It leverages CloudWatch Logs for EKS logging, uses subscription filters to route logs per component, and integrates with Lambda and SNS as required. A is promising.

Option B: Enable Amazon CloudWatch Logs to log the EKS components. Create CloudWatch Logs Insights queries linked to Amazon EventBridge events that invoke Lambda



- What This Does:
  - CloudWatch Logs for EKS: Same as A---enable CloudWatch Logs to capture EKS component logs.
  - CloudWatch Logs Insights:
    - Logs Insights is a query tool for analyzing logs in CloudWatch Log groups. You write queries (e.g., `fields @message | filter component = "kube-apiserver" | filter @message like /Failed/`) to find specific events.
  - Linked to EventBridge Events:
    - Logs Insights queries can be scheduled (e.g., via CloudWatch Metric Filters or Contributor Insights) to detect patterns and emit metrics or events.
    - You'd need to set up a mechanism to trigger EventBridge events based on query results (e.g., using a CloudWatch Alarm on a metric derived from the query).
  - Invoke Lambda:
    - EventBridge events trigger the Lambda function, which evaluates the event and publishes to SNS.
- Analysis:
  - Logging: CloudWatch Logs works for EKS, as in A.
  - Logs Insights:
    - Logs Insights is great for ad-hoc analysis or scheduled queries, but it's not designed for real-time streaming.
    - To link to EventBridge, you'd:
      1. Create a Metric Filter to detect patterns in logs (e.g., "Failed" messages).
      2. Emit a CloudWatch Metric based on the filter.
      3. Set a CloudWatch Alarm on the metric to trigger an EventBridge event.
      4. EventBridge invokes Lambda.
    - This process is delayed---Metric Filters and Alarms aren't near real-time (they aggregate over minutes, e.g., 1-minute periods).
  - Per Component:
    - You'd need a separate Metric Filter and Alarm for each component, which is feasible but cumbersome.
  - Does It Meet Requirements?:
    - Log Collection: Yes, CloudWatch Logs works.

- Processing: Yes, but with delays---Logs Insights and Alarms aren't real-time (minutes delay vs. seconds in A).
- Notifications: Yes, Lambda can publish to SNS, but the delay impacts responsiveness.
- Efficiency:
  - This is more complex than A: Metric Filters → Metrics → Alarms → EventBridge → Lambda.
  - It's not near real-time, which is a drawback for "automated notifications."
- Conclusion: This works but is less efficient than A due to delays and complexity. Subscription filters (A) are the better choice for real-time log streaming. B is not the best.

Option C: Enable Amazon S3 logging for the EKS components. Configure an Amazon CloudWatch subscription filter for each component with Lambda as the subscription feed destination

- What This Does:
  - S3 Logging for EKS:
    - This implies storing EKS component logs in S3 (e.g., as log files).
    - Issue: EKS doesn't natively send logs to S3. EKS integrates with CloudWatch Logs for control plane and pod logs (via agents like Fluentd). You could configure a custom solution to write logs to S3 (e.g., Fluentd with an S3 output plugin), but this isn't standard for EKS.
  - CloudWatch Subscription Filter:
    - Subscription filters apply to CloudWatch Logs, not S3. If logs are in S3, you can't directly apply a CloudWatch subscription filter.
    - To use subscription filters, logs must first be in CloudWatch Logs, contradicting the "S3 logging" premise.
- Analysis:
  - Logging: EKS doesn't natively send logs to S3. You'd need a custom pipeline (e.g., Fluentd → S3), which adds complexity and isn't mentioned.
  - Subscription Filter: This only works with CloudWatch Logs, not S3. The option is internally inconsistent.
  - Does It Meet Requirements?: No---S3 isn't the right logging destination for EKS, and subscription filters don't apply to S3.
- Conclusion: This is incorrect due to the mismatch between S3 logging and CloudWatch subscription filters. C is incorrect.

Option D: Enable Amazon S3 logging for the EKS components. Configure S3 PUT Object event notifications with AWS Lambda as the destination

- What This Does:
  - S3 Logging for EKS: Same as C---implies storing EKS logs in S3, which isn't native for EKS.
  - S3 PUT Object Event Notifications:
    - S3 can trigger events (e.g., Lambda, SQS, SNS) when objects are created (PUT operations).
    - If logs are written to S3 as objects, an S3 event notification can invoke the Lambda function for each new log file.
- Analysis:
  - Logging: As in C, EKS doesn't natively send logs to S3. You'd need a custom setup (e.g., Fluentd → S3), which isn't standard and adds complexity.
  - S3 Event Notifications:
    - If logs are in S3, this works---each new log file triggers Lambda, which processes the file and publishes to SNS.
    - However, this is file-based, not event-based. Logs are batched into files, so notifications are delayed until the file is written (e.g., every 5 minutes with Fluentd's buffering).
    - Per Component: S3 events don't easily filter "per component"---you'd need to organize logs into separate prefixes (e.g., logs/kube-apiserver/, logs/scheduler/) and set up separate event notifications, which is cumbersome.
  - Does It Meet Requirements?:
    - Log Collection: No, EKS doesn't natively log to S3.
    - Processing: Yes, if logs were in S3, but with delays (file-based, not real-time).
    - Notifications: Yes, Lambda can publish to SNS.
  - Efficiency:
    - Requires a custom logging pipeline to get EKS logs to S3.
    - S3 events are less real-time than CloudWatch subscription filters (A).
- Conclusion: This is impractical due to EKS's lack of native S3 logging and the delayed, file-based nature of S3 events. D is incorrect.

Analysis: Is A the Best Choice?

Your pick, A, uses:

- CloudWatch Logs to capture EKS component logs.
- Subscription filters per component to stream logs to Lambda.
- Lambda to evaluate logs and publish to SNS.

### Why A Works:

- Logging: CloudWatch Logs is the native logging solution for EKS---control plane logs, pod logs, and worker node logs can all be sent here.
- Per Component: Subscription filters allow filtering by component (e.g., via log patterns or separate log groups), meeting the requirement for component-specific notifications.
- Real-Time: Subscription filters stream logs to Lambda in near real time (seconds), ideal for automated notifications.
- Integration: CloudWatch Logs → Lambda → SNS is a standard AWS pattern for log-driven notifications.

### Comparison to Alternatives:

- B: CloudWatch Logs Insights with EventBridge is slower (minutes delay) and more complex (Metric Filters → Alarms → EventBridge).
- C: S3 logging isn't native for EKS, and subscription filters don't apply to S3.
- D: S3 logging isn't native, and S3 events are file-based (delayed, harder to filter per component).

### A is the best choice:

- It uses the right logging destination (CloudWatch Logs).
- It provides real-time streaming (subscription filters).
- It supports per-component filtering and integrates seamlessly with Lambda and SNS.

### Why You Picked A

#### You chose A because:

- You recognized that CloudWatch Logs is the standard way to capture EKS logs (control plane, pods, etc.).
- You understood that subscription filters can stream logs to Lambda in real time, allowing for quick evaluation and notification.
- You saw that this approach aligns with the architecture (Lambda evaluating logs, publishing to SNS for emails).

Your reasoning is spot-on! CloudWatch Logs with subscription filters is exactly the right pattern for this use case---real-time, component-specific, and AWS-native.

Correct Answer: A. Enable Amazon CloudWatch Logs to log the EKS components. Create a CloudWatch subscription filter for each component with Lambda as the subscription feed destination.

---

**A company is implementing an Amazon Elastic Container Service (Amazon ECS) cluster to run its workload. The company architecture will run multiple ECS services on the cluster. The architecture includes an Application Load Balancer on the front end and uses multiple target groups to route traffic.**

**A DevOps engineer must collect application and access logs. The DevOps engineer then needs to send the logs to an Amazon S3 bucket for near-real-time analysis.**

Which combination of steps must the DevOps engineer take to meet these requirements? (Choose three.)

- A. Download the Amazon CloudWatch Logs container instance from AWS. Configure this instance as a task. Update the application service definitions to include the logging task.
- B. Install the Amazon CloudWatch Logs agent on the ECS instances. Change the logging driver in the ECS task definition to awslogs.
- C. Use Amazon EventBridge to schedule an AWS Lambda function that will run every 60 seconds and will run the Amazon CloudWatch Logs create-export-task command. Then point the output to the logging S3 bucket.
- D. Activate access logging on the ALB. Then point the ALB directly to the logging S3 bucket.
- E. Activate access logging on the target groups that the ECS services use. Then send the logs directly to the logging S3 bucket.
- F. Create an Amazon Kinesis Data Firehose delivery stream that has a destination of the logging S3 bucket. Then create an Amazon CloudWatch Logs subscription filter for Kinesis Data Firehose.

### Understanding the Problem

- Setup:
  - Amazon ECS Cluster: Runs multiple ECS services (likely using tasks on EC2 or Fargate launch types).
  - Application Load Balancer (ALB): Fronts the ECS services, routing traffic to multiple target groups (one per service).
  - Logs to Collect:
    - Application Logs: Logs generated by the containers in the ECS tasks (e.g., app-specific logs like `/var/log/app.log`).
    - Access Logs: Logs from the ALB capturing HTTP requests (e.g., client IPs, request paths, response codes).
  - Destination: An S3 bucket for near-real-time analysis.
- Requirements:
  - Collect application logs from ECS services.
  - Collect access logs from the ALB.
  - Send logs to an S3 bucket in near real time (implies minimal delay, ideally seconds).
- Key Considerations:
  - ECS application logs typically go to a logging driver (e.g., awslogs for CloudWatch Logs), but need to end up in S3.
  - ALB access logs are natively supported and can be sent to S3 directly.

- "Near real-time" suggests streaming or frequent exports, not batch processes with long delays (e.g., hourly exports).

Let's evaluate each option to find the three steps that meet these requirements.

Option A: Download the Amazon CloudWatch Logs container instance from AWS. Configure this instance as a task. Update the application service definitions to include the logging task

- What This Does:
  - CloudWatch Logs Container Instance:
    - This likely refers to a containerized version of the CloudWatch Logs Agent (e.g., amazon/cloudwatch-agent or aws-observability/aws-for-fluent-bit).
    - You run this as a sidecar container in your ECS task to collect logs from other containers and send them to CloudWatch Logs.
  - Configure as a Task:
    - Add the CloudWatch Logs container as a sidecar to each ECS task definition, alongside the application container.
  - Update Service Definitions:
    - Update the ECS service to use the new task definition with the logging sidecar.
- Analysis:
  - Application Logs:
    - A sidecar container (e.g., Fluent Bit or CloudWatch Agent) can collect logs from the application container (e.g., by mounting a shared volume or reading stdout/stderr) and send them to CloudWatch Logs.
    - This works for ECS, especially on Fargate, where you can't install agents directly on the host (unlike EC2).
  - Path to S3:
    - Logs go to CloudWatch Logs first, but the requirement is to send them to S3. You'd need an additional step (e.g., a subscription filter to Kinesis Firehose, then to S3) to get logs from CloudWatch to S3.
    - This option doesn't address the S3 step, leaving the solution incomplete.
  - Access Logs:
    - This doesn't address ALB access logs at all---it's only for application logs.
  - Near Real-Time:
    - Sidecar logging to CloudWatch Logs is near real-time (seconds delay), but getting logs to S3 requires another mechanism.

- Complexity:
  - Using a sidecar is a valid approach but more complex than using the awslogs driver (B), which is built into ECS and achieves the same result (logs to CloudWatch) without a separate container.
- Conclusion: This collects application logs but doesn't address S3 delivery or ALB access logs. It's also less efficient than the awslogs driver. A is not the best choice.

Option B: Install the Amazon CloudWatch Logs agent on the ECS instances. Change the logging driver in the ECS task definition to awslogs

Your first pick! Let's break this down:

- What This Does:
  - CloudWatch Logs Agent on ECS Instances:
    - If the ECS cluster uses the EC2 launch type, the "ECS instances" are EC2 instances running the ECS agent.
    - Installing the CloudWatch Logs Agent on these instances prepares them to send logs to CloudWatch Logs.
    - However, for container logs, the agent isn't strictly necessary if you use the awslogs logging driver (more on this below).
  - Logging Driver to awslogs:
    - In an ECS task definition, the logConfiguration specifies the logging driver for containers:

json

```
"logConfiguration": {
 "logDriver": "awslogs",
 "options": {
 "awslogs-group": "/ecs/my-app",
 "awslogs-region": "us-east-1",
 "awslogs-stream-prefix": "ecs"
 }
}
```

- The awslogs driver sends container logs (stdout/stderr) directly to CloudWatch Logs, without needing a separate agent.
    - If using Fargate, the awslogs driver is supported natively---no agent installation required.
- Analysis:
  - Application Logs:

- The awslogs driver sends container logs to CloudWatch Logs, which is the standard logging solution for ECS.
- For EC2 launch type, the ECS instances need the CloudWatch Logs Agent (or proper IAM permissions) to send logs, but modern ECS setups often rely on the awslogs driver directly, which uses the ECS agent's permissions.
- For Fargate, no agent installation is needed---awslogs works out of the box.
- Path to S3:
  - Logs go to CloudWatch Logs first. To get them to S3, we need another step (e.g., a subscription filter to Kinesis Firehose, then to S3).
  - This option doesn't address the S3 delivery, but it's a foundational step for application logs.
- Access Logs:
  - This doesn't address ALB access logs---it's only for ECS application logs.
- Near Real-Time:
  - awslogs sends logs to CloudWatch in near real time (seconds delay), but S3 delivery requires another step.
- Agent Installation:
  - If using EC2 launch type, installing the CloudWatch Logs Agent might be necessary for older setups, but the awslogs driver typically doesn't require it---the ECS agent handles log forwarding (as long as the instance role has CloudWatch permissions).
  - For Fargate, this step is irrelevant---no agent can be installed.
- Conclusion: This is a key step for collecting application logs from ECS tasks. It's the most straightforward way to get container logs to CloudWatch Logs, which can then be routed to S3 (via another step). The agent installation part is questionable (often unnecessary), but the awslogs driver is spot-on. B is a good choice for application logs.

Option C: Use Amazon EventBridge to schedule an AWS Lambda function that will run every 60 seconds and will run the Amazon CloudWatch Logs create-export-task command. Then point the output to the logging S3 bucket

- What This Does:
  - EventBridge Schedule:
    - Schedule a Lambda function to run every 60 seconds using an EventBridge rule.
  - Lambda Function:
    - The Lambda runs the create-export-task command for CloudWatch Logs, which exports logs from a log group to an S3 bucket.



- Output to S3:
  - The exported logs are written to the specified S3 bucket.
- Analysis:
  - Application Logs:
    - If application logs are in CloudWatch Logs (e.g., via awslogs driver from B), this exports them to S3.
  - Access Logs:
    - ALB access logs aren't in CloudWatch Logs---they go directly to S3 (via D). This doesn't address access logs.
  - Near Real-Time:
    - create-export-task is a batch operation---it exports a range of logs (e.g., from a start time to an end time) to S3.
    - Running every 60 seconds helps, but it's not truly near real-time:
      - Export tasks take time to complete (minutes for large log volumes).
      - Logs are batched, not streamed---there's a delay between log generation and S3 availability.
    - Near real-time typically means seconds, not minutes. A streaming solution (e.g., subscription filter to Kinesis Firehose) is better.
  - Efficiency:
    - Exporting every 60 seconds creates many S3 objects (e.g., one per export task), which can complicate analysis.
    - It's less efficient than streaming logs directly to S3.
- Conclusion: This gets logs from CloudWatch to S3 but isn't near real-time (batch exports take minutes) and doesn't address ALB access logs. It's a partial solution for application logs but not the best approach. C is not the best choice.

Option D: Activate access logging on the ALB. Then point the ALB directly to the logging S3 bucket

Your second pick! Let's break this down:

- What This Does:
  - ALB Access Logging:
    - ALBs support access logs, which capture HTTP request details (e.g., client IP, request URL, response code).
    - You enable access logging on the ALB and specify an S3 bucket as the destination:

json

Attributes:

```
access_logs.enabled: true
access_logs.s3.enabled: true
access_logs.s3.bucket: my-logging-bucket
access_logs.s3.prefix: alb-logs
```

- Direct to S3:
  - ALB writes access logs directly to the specified S3 bucket (e.g., as files like alb-logs/2025/03/01/logfile.gz).
- Analysis:
  - Access Logs:
    - This directly addresses the requirement to collect ALB access logs.
    - ALB access logs are written to S3 natively---no intermediate steps needed.
  - Application Logs:
    - This doesn't address ECS application logs---it's only for ALB access logs.
  - Near Real-Time:
    - ALB access logs are written to S3 in near real time---logs are typically available within seconds to a few minutes (AWS batches them into files every 5 minutes or so, but this is still considered near real-time for most use cases).
  - Efficiency:
    - This is the simplest, most direct way to get ALB access logs to S3---no additional services or processing required.
- Conclusion: This is a necessary step for ALB access logs. It meets the requirement for access logs and delivers them to S3 in near real time. D is a great choice.

Option E: Activate access logging on the target groups that the ECS services use. Then send the logs directly to the logging S3 bucket

Your third pick! Let's break this down:

- What This Does:
  - Access Logging on Target Groups:
    - Target groups are part of the ALB configuration---they route traffic to ECS tasks.
    - However, target groups do not support access logging. Access logging is a feature of the ALB itself, not the target groups.

- Send Logs to S3:
  - If target groups supported logging, this would send logs to S3.
- Analysis:
  - Access Logs:
    - This is factually incorrect---target groups don't have access logging. ALB access logging (D) captures all relevant request data, including which target group handled the request (via the `target_group_arn` field in the log).
    - There's no need to log at the target group level---ALB logging already covers this.
  - Does It Meet Requirements?:
    - No, because target groups don't support access logging.
- Conclusion: This option is incorrect due to a misunderstanding of ALB and target group capabilities. ALB access logging (D) is the correct approach. E is incorrect.

Option F: Create an Amazon Kinesis Data Firehose delivery stream that has a destination of the logging S3 bucket. Then create an Amazon CloudWatch Logs subscription filter for Kinesis Data Firehose

- What This Does:
  - Kinesis Data Firehose:
    - Firehose is a streaming service that can deliver data to destinations like S3.
    - You create a Firehose delivery stream with the logging S3 bucket as the destination.
  - CloudWatch Logs Subscription Filter:
    - A subscription filter in CloudWatch Logs (e.g., on the ECS log group) streams logs to Firehose.
    - Firehose buffers the logs (e.g., for 60 seconds or 1 MB) and writes them to S3.
- Analysis:
  - Application Logs:
    - If ECS logs are in CloudWatch Logs (e.g., via `awslogs` driver from B), this subscription filter streams them to Firehose, which delivers them to S3.
    - This completes the path for application logs: ECS → CloudWatch Logs → Firehose → S3.
  - Access Logs:
    - This doesn't address ALB access logs, which aren't in CloudWatch Logs---ALB logs go directly to S3 (via D).
  - Near Real-Time:

- Firehose streams logs to S3 in near real time---buffering delays are typically 60 seconds (configurable), which qualifies as near real-time for this use case.
- This is much better than batch exports (e.g., create-export-task in C, which takes minutes).
- Efficiency:
  - Firehose is a managed service, handling scaling and buffering automatically.
  - It's a standard pattern for streaming CloudWatch Logs to S3.
- Conclusion: This is a critical step for getting application logs from CloudWatch Logs to S3 in near real time. It complements B (collecting logs) by providing the delivery mechanism. F is a great choice.

Analysis: Are B, D, and E the Right Combination?

Your picks:

- B: Install the CloudWatch Logs agent on ECS instances and use the awslogs driver in the task definition.
- D: Activate ALB access logging and send logs to S3.
- E: Activate access logging on target groups and send logs to S3.

Evaluation:

- B:
  - Correct for application logs---awslogs sends ECS container logs to CloudWatch Logs.
  - The agent installation part is unnecessary for Fargate and often not needed for EC2 (the ECS agent handles awslogs with proper IAM permissions), but the awslogs driver is the key piece.
  - This is a necessary step, but it needs a way to get logs to S3 (which E doesn't provide).
- D:
  - Correct for ALB access logs---enables logging on the ALB and sends logs directly to S3.
  - Meets the access logs requirement in near real time.
- E:
  - Incorrect---target groups don't support access logging. ALB logging (D) already covers this.

What's Missing?:

- Application Logs Path to S3: B gets logs to CloudWatch, but we need a step to stream them to S3. F (Kinesis Firehose with a subscription filter) completes this path.
- Correct Combination:
  - B: Collect application logs (awslogs driver → CloudWatch Logs).

- D: Collect ALB access logs (ALB → S3).
- F: Stream application logs to S3 (CloudWatch Logs → Firehose → S3).

Why E Doesn't Fit:

- Target group logging isn't a thing---D already handles ALB access logs.

Correct Combination: B, D, and F.

- B: Sets up application log collection to CloudWatch Logs.
- D: Handles ALB access logs to S3.
- F: Streams application logs from CloudWatch to S3 in near real time.

Why You Picked B, D, and E

- B: You recognized that ECS uses the awslogs driver to send container logs to CloudWatch Logs, a standard practice for application logs.
- D: You correctly identified that ALB access logs can be sent directly to S3, meeting the access logs requirement.
- E: You might have thought target groups could log separately to capture service-specific traffic, but ALB logging (D) already includes target group details in its logs (e.g., `target_group_arn` field).

You were very close---B and D are spot-on, and E was a reasonable guess if you weren't sure about target group logging. Now you know that ALB logging covers it, and F completes the application logs path to S3.

Correct Answer: B, D, and F.

- B. Install the Amazon CloudWatch Logs agent on the ECS instances. Change the logging driver in the ECS task definition to `awslogs`.
- D. Activate access logging on the ALB. Then point the ALB directly to the logging S3 bucket.
- F. Create an Amazon Kinesis Data Firehose delivery stream that has a destination of the logging S3 bucket. Then create an Amazon CloudWatch Logs subscription filter for Kinesis Data Firehose.

---

**A company that uses electronic health records is running a fleet of Amazon EC2 instances with an Amazon Linux operating system. As part of patient privacy requirements, the company must ensure continuous compliance for patches for operating system and applications running on the EC2 instances.**

How can the deployments of the operating system and application patches be automated using a default and custom repository?

- A. Use AWS Systems Manager to create a new patch baseline including the custom repository. Run the `AWS-RunPatchBaseline` document using the `run` command to verify and install patches.
- B. Use AWS Direct Connect to integrate the corporate repository and deploy the patches using Amazon CloudWatch scheduled events, then use the CloudWatch dashboard to create reports.
- C. Use `yum-config-manager` to add the custom repository under `/etc/yum.repos.d` and run `yum-config-`

manager-enable to activate the repository.

D. Use AWS Systems Manager to create a new patch baseline including the corporate repository. Run the AWS-AmazonLinuxDefaultPatchBaseline document using the run command to verify and install patches.

### Understanding the Problem

- Setup:
  - EC2 Instances: A fleet of instances running Amazon Linux (likely Amazon Linux 2, as it's common for such scenarios).
  - Patient Privacy: Likely requires compliance with standards like HIPAA, meaning OS and application patches must be applied consistently to mitigate vulnerabilities.
- Requirements:
  - Automate OS and application patch deployments.
  - Use a default repository (e.g., Amazon Linux repositories for OS patches).
  - Use a custom repository (e.g., a corporate repo for application patches).
  - Ensure continuous compliance (implies regular, automated patching).
- Key Considerations:
  - Amazon Linux uses yum (or dnf in Amazon Linux 2023) for package management, pulling from default repos (e.g., amzn2-core).
  - A custom repo would host application-specific packages (e.g., .rpm files for the electronic health record app).
  - We need a solution that integrates both repos, automates patching, and ensures compliance (e.g., via scheduling and reporting).

Let's evaluate each option to find the best solution.

Option A: Use AWS Systems Manager to create a new patch baseline including the custom repository. Run the AWS-RunPatchBaseline document using the run command to verify and install patches

Your pick! Let's break this down:

- What This Does:
  - AWS Systems Manager (SSM):
    - SSM provides Patch Manager, a feature to automate patching for EC2 instances.
    - Patch Manager uses patch baselines to define which patches are approved for installation.
  - New Patch Baseline with Custom Repository:
    - You create a custom patch baseline in SSM Patch Manager.

- A patch baseline specifies:
  - Rules for approving patches (e.g., auto-approve critical OS patches after 7 days).
  - Sources for patches, including default repos (Amazon Linux repos) and custom repos.
- For Amazon Linux, the default repo is the Amazon Linux repository (amzn2-core, etc.).
- You add the custom repository as a patch source:

json

```
{
 "Name": "CustomAppRepo",
 "Products": ["AmazonLinux2"],
 "Configuration": "https://custom-
repo.example.com/amazonlinux2/",
 "Architecture": ["x86_64"]
}
```

- This allows SSM to pull OS patches from the default Amazon Linux repo and application patches from the custom repo.
- AWS-RunPatchBaseline Document:
  - AWS-RunPatchBaseline is an SSM Automation document that:
    - Scans the instance against the patch baseline to verify compliance (which patches are missing).
    - Installs approved patches from the baseline's sources (default + custom repos).
  - You run this document using SSM Run Command:

bash

```
aws ssm send-command\
 --document-name "AWS-RunPatchBaseline"\
 --targets Key=tag:Environment,Values=Production\
 --parameters '{"Operation":"Install"}'
```

- Operation Types:
  - Scan: Checks for missing patches and reports compliance.
  - Install: Installs missing patches.
- Analysis:

- OS and Application Patching:
  - The custom patch baseline includes both the default Amazon Linux repo (for OS patches) and the custom repo (for application patches).
  - AWS-RunPatchBaseline installs both types of patches, meeting the requirement to use default and custom repositories.
- Automation:
  - You can automate this by:
    - Creating a State Manager association to run AWS-RunPatchBaseline on a schedule (e.g., every week).
    - Using a Patch Group to apply the custom baseline to specific instances (e.g., tag-based targeting).
  - This ensures continuous compliance through regular, automated patching.
- Compliance:
  - Patch Manager provides compliance reporting in the AWS Console or via API (e.g., DescribeInstancePatchStates), showing which instances are patched and which aren't.
  - This aligns with patient privacy requirements (e.g., HIPAA) by ensuring vulnerabilities are addressed.
- Does It Meet Requirements?:
  - Default and Custom Repo: Yes, the patch baseline includes both.
  - Automation: Yes, via State Manager or scheduled Run Command.
  - Continuous Compliance: Yes, with scheduled patching and reporting.
- Conclusion: This is a strong solution. It leverages SSM Patch Manager, a purpose-built tool for automated patching, and supports both default and custom repos in the patch baseline. A looks promising.

Option B: Use AWS Direct Connect to integrate the corporate repository and deploy the patches using Amazon CloudWatch scheduled events, then use the CloudWatch dashboard to create reports

- What This Does:
  - AWS Direct Connect:
    - Direct Connect provides a dedicated network connection between your on-premises network and AWS.
    - Here, it's used to integrate the corporate repository (assumed to be on-premises) with AWS.
  - CloudWatch Scheduled Events:



- Schedule an event (e.g., via EventBridge) to trigger a patching action periodically.
- The event could invoke a Lambda function or SSM Run Command to deploy patches.
- CloudWatch Dashboard for Reports:
  - Use CloudWatch to create dashboards for patching reports.
- Analysis:
  - Corporate Repository:
    - Direct Connect ensures low-latency access to an on-premises corporate repo, which could host application patches.
    - However, Direct Connect isn't necessary---custom repos in SSM Patch Manager (A) can be HTTP/HTTPS-based (e.g., an S3 bucket or an HTTP server), and don't require Direct Connect unless the repo is only accessible via a private network.
  - Patching:
    - CloudWatch Events (now EventBridge) can schedule a Lambda or SSM Run Command to deploy patches.
    - However, this option doesn't specify how patches are deployed:
      - It doesn't mention default repos (Amazon Linux repos for OS patches).
      - It doesn't specify a mechanism like SSM Patch Manager to manage both OS and application patches.
      - You'd need to write custom scripts to pull patches from both repos (e.g., yum install from the corporate repo and default repo), which is less automated than SSM Patch Manager.
  - Reporting:
    - CloudWatch Dashboards can display metrics, but patching reports are better handled by SSM Patch Manager's compliance views.
    - You'd need to log patching results to CloudWatch Logs and build custom metrics, which is more work than SSM's built-in reporting.
  - Does It Meet Requirements?:
    - Default and Custom Repo: Partially---custom repo via Direct Connect, but default repo isn't addressed.
    - Automation: Yes, via scheduled events, but the patching mechanism is unclear.
    - Continuous Compliance: No---lacks a robust patching framework and compliance reporting.

- Conclusion: This is overly complex (Direct Connect isn't needed) and lacks specificity on patching both OS and applications. SSM Patch Manager (A) is a better fit. B is incorrect.

Option C: Use yum-config-manager to add the custom repository under /etc/yum.repos.d and run yum-config-manager-enable to activate the repository

- What This Does:
  - yum-config-manager:
    - This is a command-line tool to manage yum repositories on Amazon Linux.
    - Add the custom repo to /etc/yum.repos.d:

bash

```
sudo yum-config-manager --add-repo https://custom-repo.example.com/amazonlinux2/
```

- Enable the repo:

bash

```
sudo yum-config-manager --enable custom-repo
```

- This allows yum to pull packages from both the default Amazon Linux repo and the custom repo.
- Analysis:
  - OS and Application Patching:
    - This sets up the custom repo, ensuring yum can access both default and custom repos for OS and application patches.
    - However, it doesn't address automation---you'd still need to run yum update to install patches.
  - Automation:
    - This is a manual step (or requires scripting to run on each instance).
    - There's no mechanism for scheduling or continuous compliance (e.g., no automatic patching or reporting).
    - You could use SSM Run Command to automate running these commands and yum update, but that's not specified here.
  - Compliance:

- No built-in reporting or compliance tracking---you'd need to build this separately.
- Does It Meet Requirements?:
  - Default and Custom Repo: Yes, it adds the custom repo while keeping the default.
  - Automation: No, this is a manual setup step, not an automated patching solution.
  - Continuous Compliance: No, lacks scheduling and reporting.
- Conclusion: This sets up the repos but doesn't automate patching or ensure continuous compliance. It's a foundational step but incomplete. C is incorrect.

Option D: Use AWS Systems Manager to create a new patch baseline including the corporate repository. Run the AWS-AmazonLinuxDefaultPatchBaseline document using the run command to verify and install patches

- What This Does:
  - Patch Baseline with Corporate Repository:
    - Same as A---create a new patch baseline in SSM Patch Manager, including the corporate (custom) repo for application patches.
    - The baseline also includes default Amazon Linux repos for OS patches.
  - AWS-AmazonLinuxDefaultPatchBaseline Document:
    - This is an SSM document specific to Amazon Linux, designed to apply the default Amazon Linux patch baseline.
    - It installs OS patches from Amazon Linux repos but ignores custom patch baselines.
    - Running this via Run Command:

bash

```
aws ssm send-command\
 --document-name "AWS-AmazonLinuxDefaultPatchBaseline"\
 --targets Key=tag:Environment,Values=Production\
 --parameters '{"Operation":"Install"}'
```

- Analysis:
  - OS and Application Patching:
    - The AWS-AmazonLinuxDefaultPatchBaseline document applies the default Amazon Linux patch baseline, which only includes OS patches from Amazon Linux repos.
    - It does not use the custom patch baseline you created, so application patches from the corporate repo won't be installed.
  - Automation:

- You can automate this with State Manager (as in A), but it only applies OS patches.
- Compliance:
  - Provides compliance reporting for OS patches but misses application patches.
- Does It Meet Requirements?:
  - Default and Custom Repo: No---only uses the default repo (via the default baseline).
  - Automation: Yes, but incomplete (only OS patches).
  - Continuous Compliance: No, misses application patches.
- Conclusion: This is close to A but fails because the AWS-AmazonLinuxDefaultPatchBaseline document ignores the custom patch baseline, so application patches aren't applied. D is incorrect.

Analysis: Is A the Best Choice?

Your pick, A, uses:

- SSM Patch Manager to create a custom patch baseline with both default and custom repos.
- AWS-RunPatchBaseline document to apply the custom baseline, installing both OS and application patches.

Why A Works:

- Default and Custom Repos:
  - The custom patch baseline includes both the Amazon Linux repo (default) for OS patches and the corporate repo for application patches.
- Automation:
  - AWS-RunPatchBaseline can be scheduled via State Manager (e.g., weekly), automating the patching process.
  - Run Command allows manual or scripted execution for ad-hoc patching.
- Continuous Compliance:
  - Patch Manager provides compliance reporting, ensuring patches are applied consistently (critical for HIPAA).
  - Scheduling ensures continuous compliance through regular updates.
- Comparison to Alternatives:
  - B: Overcomplicates with Direct Connect and lacks a clear patching mechanism.
  - C: Sets up repos but doesn't automate or ensure compliance.
  - D: Similar to A but uses the default baseline document, missing application patches.

A vs. D:

- Both use SSM Patch Manager, but:
  - A uses AWS-RunPatchBaseline with the custom baseline, applying both OS and application patches.
  - D uses AWS-AmazonLinuxDefaultPatchBaseline, which only applies the default baseline (OS patches), ignoring the custom repo.

A is the best choice:

- It fully addresses both repos, automates patching, and ensures compliance with reporting and scheduling.

Why You Picked A

You chose A because:

- You recognized that SSM Patch Manager is the go-to tool for automated patching on EC2 instances.
- You understood that a custom patch baseline can include both default and custom repos, meeting the requirement for OS and application patches.
- You saw that AWS-RunPatchBaseline applies the custom baseline, ensuring both types of patches are installed, and can be automated for continuous compliance.

Your reasoning is spot-on! SSM Patch Manager with a custom baseline is exactly the right approach for this scenario, and AWS-RunPatchBaseline ensures both repos are used.

Correct Answer: A. Use AWS Systems Manager to create a new patch baseline including the custom repository. Run the AWS-RunPatchBaseline document using the run command to verify and install patches.

---

**A company uses AWS Storage Gateway in file gateway mode in front of an Amazon S3 bucket that is used by multiple resources. In the morning when business begins, users do not see the objects processed by a third party the previous evening. When a DevOps engineer looks directly at the S3 bucket, the data is there, but it is missing in Storage Gateway.**

Which solution ensures that all the updated third-party files are available in the morning?

- A. Configure a nightly Amazon EventBridge event to invoke an AWS Lambda function to run the RefreshCache command for Storage Gateway.
- B. Instruct the third party to put data into the S3 bucket using AWS Transfer for SFTP.
- C. Modify Storage Gateway to run in volume gateway mode.
- D. Use S3 Same-Region Replication to replicate any changes made directly in the S3 bucket to Storage Gateway.

Understanding the Problem

- Setup:

- AWS Storage Gateway in File Gateway Mode: Storage Gateway is a hybrid cloud storage service that connects on-premises environments to AWS storage. In file gateway mode, it provides a file system interface (NFS or SMB) to store and retrieve files as objects in an S3 bucket. Clients (e.g., users) access the S3 bucket via the file gateway, seeing S3 objects as files in a mounted file share.
- S3 Bucket: The underlying storage where files are stored as objects. Multiple resources, including a third party, write to this bucket.
- Third-Party Updates: The third party processes data in the evening, writing objects directly to the S3 bucket (not through the file gateway).
- Issue:
  - In the morning, users accessing the file gateway don't see the third-party objects, but a DevOps engineer confirms the data is in the S3 bucket.
- Requirements:
  - Ensure that third-party updates (objects written directly to S3) are visible to users via the file gateway in the morning.
  - The solution should be automated and reliable.

#### Why the Files Are Missing:

- In file gateway mode, Storage Gateway maintains a local cache on the gateway to provide low-latency access to recently used data. When users access files through the gateway, it serves them from the cache if available, or fetches them from S3 if not.
- When files are written through the file gateway (e.g., via NFS/SMB), the gateway updates its cache and uploads the files to S3 as objects.
- However, when a third party writes directly to the S3 bucket (bypassing the gateway), the file gateway's cache isn't aware of these changes. The gateway doesn't automatically sync its cache with S3 unless explicitly instructed.
- As a result, users accessing the file share in the morning don't see the new objects because the gateway's cache hasn't been updated to reflect the third-party changes.

#### Goal:

- We need a mechanism to sync the file gateway's cache with the S3 bucket's latest state, ensuring third-party updates are visible to users in the morning.

Let's evaluate each option.

Option A: Configure a nightly Amazon EventBridge event to invoke an AWS Lambda function to run the RefreshCache command for Storage Gateway

Your pick! Let's break this down:

- What This Does:

- Amazon EventBridge:
  - EventBridge can schedule events on a cron-like schedule (e.g., nightly at 2 AM).
  - You create a rule to trigger a Lambda function every night.
- AWS Lambda Function:
  - The Lambda function calls the RefreshCache command for Storage Gateway.
  - The RefreshCache operation forces the file gateway to refresh its local cache by re-inventorying the associated S3 bucket. This ensures the gateway's cache reflects the latest objects in S3, including those written by the third party.
  - Using the AWS CLI, the command looks like:

bash

```
aws storagegateway refresh-cache --file-share-arn
arn:aws:storagegateway:region:account-id:share/share-id
```

- Nightly Schedule:
  - Running this after the third party's evening processing (e.g., at 2 AM) ensures the cache is updated before users start work in the morning.
- Analysis:
  - Does It Fix the Cache Issue?:
    - Yes! The core problem is that the file gateway's cache is out of sync with the S3 bucket because the third party bypassed the gateway. RefreshCache explicitly updates the cache, making the third-party objects visible to users via the file share.
  - Automation:
    - EventBridge provides a reliable, automated schedule. A nightly run ensures the cache is always updated before business hours.
  - Timing:
    - The third party processes data in the evening, so a refresh at 2 AM (for example) would capture all changes and make them available by morning.
  - Scalability:
    - This solution scales well---RefreshCache works for any number of objects in the bucket. For large buckets, the refresh might take time, but it's a one-time operation per night.
  - Does It Meet Requirements?:
    - Updated Files Available: Yes, RefreshCache ensures the third-party objects are visible.

- Morning Availability: Yes, a nightly run ensures users see the updates when they start work.
- Conclusion: This directly addresses the cache sync issue in a simple, automated way. It's a standard approach for file gateway scenarios where S3 is updated outside the gateway. A looks like a winner.

Option B: Instruct the third party to put data into the S3 bucket using AWS Transfer for SFTP

- What This Does:
  - AWS Transfer for SFTP:
    - AWS Transfer Family provides a managed SFTP service to transfer files into and out of S3.
    - The third party would use SFTP to upload files to the S3 bucket instead of writing directly (e.g., via S3 API or SDK).
- Analysis:
  - Does It Fix the Cache Issue?:
    - No. The method of upload (SFTP vs. S3 API) doesn't change the core issue: the file gateway's cache isn't updated when files are written directly to S3, regardless of how they get there.
    - AWS Transfer for SFTP uploads files to S3, bypassing the file gateway, just like the third party's current method. The gateway's cache still won't reflect these changes without a refresh.
  - Automation:
    - This requires the third party to change their process (switch to SFTP), which isn't an automated solution on the company's side---it shifts the burden to the third party.
  - Practicality:
    - Changing the third party's workflow might be difficult or unnecessary if they already have a working S3 integration.
    - SFTP might add security (e.g., SSH-based access), but the question doesn't indicate a security issue---just a visibility issue.
  - Does It Meet Requirements?:
    - Updated Files Available: No, the cache sync problem persists.
    - Morning Availability: No, users still won't see the files.
- Conclusion: This doesn't solve the cache sync issue. It just changes how the third party uploads files, which doesn't address the gateway's visibility problem. B is incorrect.

Option C: Modify Storage Gateway to run in volume gateway mode

- What This Does:



- Volume Gateway Mode:
  - Storage Gateway supports multiple modes: file gateway, volume gateway, and tape gateway.
  - Volume gateway provides block storage (iSCSI volumes) backed by S3, not a file system interface like file gateway.
  - There are two sub-modes:
    - Cached Mode: Primary data in S3, frequently accessed data cached locally.
    - Stored Mode: Primary data on-premises, asynchronously backed up to S3.
- Switching to Volume Gateway:
  - This would replace the file gateway (NFS/SMB file shares) with iSCSI block volumes.
- Analysis:
  - Does It Fix the Cache Issue?:
    - No, it changes the entire architecture. Volume gateway doesn't use a file system interface---it provides block storage (like a virtual disk).
    - The third party is writing to an S3 bucket, and volume gateway would store data as EBS snapshots in S3, not as objects in the same bucket. This doesn't align with the current setup (multiple resources accessing the same S3 bucket).
  - Impact on Users:
    - Users currently access files via NFS/SMB through the file gateway. Switching to volume gateway means they'd need to mount iSCSI volumes instead, which changes how the application accesses data (block storage vs. file storage).
    - This is a major architectural change, likely breaking existing workflows.
  - Does It Meet Requirements?:
    - Updated Files Available: No, it doesn't address the cache sync issue and changes the access method entirely.
    - Morning Availability: No, it's a different solution that doesn't fit the current setup.
  - Practicality:
    - Switching modes is overkill and unnecessary---the problem is a cache sync issue, not a fundamental flaw in file gateway mode.
- Conclusion: This is a drastic, irrelevant change that doesn't solve the problem and introduces new complexities. C is incorrect.

Option D: Use S3 Same-Region Replication to replicate any changes made directly in the S3 bucket to Storage Gateway

- What This Does:
  - S3 Same-Region Replication (SRR):
    - SRR replicates objects within the same region to another S3 bucket.
    - The idea here is to replicate changes from the primary S3 bucket (where the third party writes) to another S3 bucket associated with Storage Gateway.
  - Storage Gateway:
    - File gateway is associated with a specific S3 bucket (the one it writes to and reads from).
- Analysis:
  - Does It Fix the Cache Issue?:
    - No. SRR replicates objects between two S3 buckets, but Storage Gateway's cache sync issue isn't about replication---it's about the gateway's local cache not reflecting updates in its own S3 bucket.
    - File gateway doesn't "receive" data via replication---it reads from its configured S3 bucket. Replicating to another bucket doesn't help unless the gateway is reconfigured to use the new bucket, which doesn't solve the core issue (the cache still needs refreshing).
  - Architecture:
    - File gateway is tied to one S3 bucket. If you replicate to a new bucket, the gateway won't see those objects unless you change its configuration, defeating the purpose.
    - Even if the gateway could read from the replicated bucket, the cache sync problem persists---RefreshCache is still needed.
  - Does It Meet Requirements?:
    - Updated Files Available: No, replication doesn't update the gateway's cache.
    - Morning Availability: No, users still won't see the files.
  - Practicality:
    - SRR is useful for backup or redundancy, but it's irrelevant here---the data is already in the right S3 bucket; the gateway just needs to refresh its cache.
- Conclusion: This misunderstands the problem. SRR doesn't address the cache sync issue and adds unnecessary complexity. D is incorrect.

Analysis: Is A the Best Choice?

Your pick, A, configures a nightly EventBridge event to invoke a Lambda function that runs the RefreshCache command:

- Does It Meet Requirements?:

- Updated Files Available: Yes, RefreshCache syncs the file gateway's cache with the S3 bucket, making third-party objects visible.
- Morning Availability: Yes, a nightly run ensures the cache is updated before users start work.
- Automation:
  - EventBridge and Lambda provide a fully automated solution---no manual intervention needed.
- Simplicity:
  - This is a straightforward fix: a single Lambda function with a scheduled trigger. The RefreshCache operation is designed for this exact scenario (S3 updates outside the gateway).
- Comparison to Alternatives:
  - B: Changes the third party's upload method but doesn't solve the cache issue.
  - C: Switches to volume gateway, which is irrelevant and breaks the architecture.
  - D: SRR doesn't help with cache sync and adds complexity.

#### Potential Concerns:

- Timing: If the third party finishes processing very late (e.g., 4 AM), a 2 AM refresh might miss some objects. However, the question implies evening processing, so a late-night refresh (e.g., 2 AM) should suffice. You could adjust the schedule if needed.
- Large Buckets: RefreshCache might take time for very large buckets, but it's a one-time operation per night and should complete well before morning.

#### A is the best choice:

- It directly addresses the cache sync issue with a purpose-built command (RefreshCache).
- It's automated, simple, and aligns with AWS best practices for file gateway.

#### More on AWS Storage Gateway (Since You're Unfamiliar)

- What Is File Gateway?:
  - AWS Storage Gateway connects on-premises environments to AWS storage. In file gateway mode, it provides NFS/SMB file shares backed by an S3 bucket.
  - Files written to the share are stored as S3 objects, and existing S3 objects appear as files in the share.
  - The gateway maintains a local cache for low-latency access, but this cache doesn't automatically update when S3 is modified outside the gateway (e.g., by a third party).
- Key Concept: Cache Sync:
  - The RefreshCache operation is the standard way to sync the gateway's cache with S3. AWS documentation (e.g., Troubleshooting: File Share Issues) recommends this for scenarios where

S3 is updated directly, as in this case.

- Use Case:
  - File gateway is ideal for hybrid setups where on-premises apps need file access to S3, but it requires careful handling when multiple parties access the same bucket.

Why You Picked A

You chose A because it sounded logical:

- You recognized that the issue is likely a sync problem---objects are in S3 but not visible via the gateway.
- You saw that a scheduled refresh (via EventBridge and Lambda) could update the gateway's view of S3, making the files available in the morning.
- The RefreshCache command intuitively fits the problem of missing files.

Your reasoning is spot-on! Even without deep familiarity with Storage Gateway, you identified the core issue (cache sync) and picked a solution that directly addresses it. RefreshCache is exactly what's needed here, and scheduling it nightly ensures the files are ready for users.

Correct Answer: A. Configure a nightly Amazon EventBridge event to invoke an AWS Lambda function to run the RefreshCache command for Storage Gateway.

---

**A space exploration company receives telemetry data from multiple satellites. Small packets of data are received through Amazon API Gateway and are placed directly into an Amazon Simple Queue Service (Amazon SQS) standard queue. A custom application is subscribed to the queue and transforms the data into a standard format. Because of inconsistencies in the data that the satellites produce, the application is occasionally unable to transform the data. In these cases, the messages remain in the SQS queue. A DevOps engineer must develop a solution that retains the failed messages and makes them available to scientists for review and future processing.**

Which solution will meet these requirements?

- A. Configure AWS Lambda to poll the SQS queue and invoke a Lambda function to check whether the queue messages are valid. If validation fails, send a copy of the data that is not valid to an Amazon S3 bucket so that the scientists can review and correct the data. When the data is corrected, amend the message in the SQS queue by using a replay Lambda function with the corrected data.
- B. Convert the SQS standard queue to an SQS FIFO queue. Configure AWS Lambda to poll the SQS queue every 10 minutes by using an Amazon EventBridge schedule. Invoke the Lambda function to identify any messages with a SentTimestamp value that is older than 5 minutes, push the data to the same location as the application's output location, and remove the messages from the queue.
- C. Create an SQS dead-letter queue. Modify the existing queue by including a redrive policy that sets the Maximum Receives setting to 1 and sets the dead-letter queue ARN to the ARN of the newly created queue. Instruct the scientists to use the dead-letter queue to review the data that is not valid. Reprocess this data at a later time.
- D. Configure API Gateway to send messages to different SQS virtual queues that are named for each of the satellites. Update the application to use a new virtual queue for any data that it cannot transform, and send

the message to the new virtual queue. Instruct the scientists to use the virtual queue to review the data that is not valid. Reprocess this data at a later time.

## Understanding the Problem

- Setup:
  - Amazon API Gateway: Receives small packets of telemetry data from satellites.
  - Amazon SQS Standard Queue: API Gateway places messages directly into an SQS standard queue.
  - Custom Application: Subscribed to the queue, transforms the data into a standard format, and (presumably) consumes the messages (deleting them from the queue if successful).
  - Inconsistencies in Data: The application fails to transform some messages due to inconsistent data, leaving those messages in the queue.
- Issue:
  - Failed messages remain in the SQS queue because the application can't process them.
  - SQS standard queues retry messages until they're successfully processed or manually removed (no automatic expiration unless a retention period is set).
- Requirements:
  - Retain Failed Messages: Keep messages that the application can't process.
  - Make Available to Scientists: Scientists need to review and potentially correct the failed data.
  - Future Reprocessing: Allow reprocessing of the failed data later.
- Key Considerations:
  - SQS standard queues don't have a built-in mechanism to automatically handle failed messages after a certain number of retries.
  - We need a way to identify failed messages, isolate them, and make them accessible for review and reprocessing.
  - The solution should be automated and integrate with SQS's native features where possible.

Let's evaluate each option to find the best solution.

Option A: Configure AWS Lambda to poll the SQS queue and invoke a Lambda function to check whether the queue messages are valid. If validation fails, send a copy of the data that is not valid to an Amazon S3 bucket so that the scientists can review and correct the data. When the data is corrected, amend the message in the SQS queue by using a replay Lambda function with the corrected data

- What This Does:
  - Lambda Polling SQS:

- Configure a Lambda function to poll the SQS queue (e.g., using an SQS event source mapping).
- Lambda processes each message, checking if the data is "valid" (presumably using the same logic the custom application uses to transform data).
- Validation Failure:
  - If validation fails, Lambda sends a copy of the invalid data to an S3 bucket.
  - Scientists can access the S3 bucket to review and correct the data.
- Replay Lambda Function:
  - After correction, a "replay Lambda" updates the message in the SQS queue with the corrected data.
  - Lambda deletes the original message from the queue (implicitly, by completing the processing).
- Analysis:
  - Retain Failed Messages:
    - Copying failed messages to S3 retains them for review.
  - Make Available to Scientists:
    - Storing failed messages in S3 allows scientists to access them (e.g., via the S3 Console or a custom app).
  - Future Reprocessing:
    - The replay Lambda updates the original message in the SQS queue with corrected data, allowing the application to reprocess it.
    - However, this assumes scientists manually correct the data in S3 and trigger the replay Lambda, which isn't specified (e.g., how does the Lambda know the data is corrected?).
  - Issues:
    - Lambda Polling: Lambda polling duplicates the custom application's role. The application already processes messages and fails on invalid ones. Adding a Lambda to do validation upfront means reimplementing the application's logic, which is inefficient.
    - Message Lifecycle:
      - If Lambda validates and fails, it would delete the message from the queue (or leave it if it doesn't ack), but the application might still process it, leading to race conditions.
      - Updating messages in SQS (via the replay Lambda) requires deleting the old message and adding a new one (SQS doesn't support in-place updates), which adds complexity.

- Automation:
  - The solution isn't fully automated---scientists must manually correct data in S3 and trigger the replay Lambda (not specified how).
- Does It Meet Requirements?:
  - Retain Failed Messages: Yes, via S3.
  - Make Available to Scientists: Yes, S3 is accessible.
  - Future Reprocessing: Yes, but with manual steps (replay Lambda isn't automated).
- Conclusion: This works but is inefficient. It duplicates the application's logic in Lambda, introduces race conditions, and requires manual intervention for reprocessing. There's a better SQS-native approach. A is not the best choice.

Option B: Convert the SQS standard queue to an SQS FIFO queue. Configure AWS Lambda to poll the SQS queue every 10 minutes by using an Amazon EventBridge schedule. Invoke the Lambda function to identify any messages with a `SentTimestamp` value that is older than 5 minutes, push the data to the same location as the application's output location, and remove the messages from the queue

- What This Does:
  - Convert to SQS FIFO Queue:
    - FIFO (First-In-First-Out) queues ensure strict message ordering and exactly-once delivery (deduplication via message group IDs).
    - Converting from a standard to a FIFO queue requires recreating the queue with a `.fifo` suffix (e.g., `my-queue.fifo`).
  - EventBridge Schedule:
    - Schedule a Lambda function to run every 10 minutes.
  - Lambda Function:
    - Lambda polls the SQS queue, checking for messages with a `SentTimestamp` older than 5 minutes (indicating they've been in the queue too long, likely due to processing failures).
    - Push these messages to the "application's output location" (assumed to be S3 or another storage, though not specified).
    - Remove the messages from the queue.
- Analysis:
  - Retain Failed Messages:
    - Messages older than 5 minutes are pushed to the application's output location (e.g., S3).
  - Make Available to Scientists:

- If the output location is accessible (e.g., S3), scientists can review the failed messages.
- Future Reprocessing:
  - Messages are removed from the queue after being copied, so there's no direct reprocessing path in SQS.
  - Scientists would need to manually reintroduce corrected messages to the queue (not specified how).
- Issues:
  - FIFO Queue:
    - Converting to a FIFO queue isn't necessary. The problem isn't about ordering or deduplication---it's about handling failed messages. Standard queues are sufficient.
    - FIFO queues have lower throughput (3,000 messages/sec with batching vs. nearly unlimited for standard queues), which might impact performance for telemetry data.
  - Timing-Based Approach:
    - Using `SentTimestamp` to identify failed messages (older than 5 minutes) is unreliable:
      - Messages might be delayed due to slow processing, not failure.
      - The application might still be retrying the message when Lambda removes it, causing race conditions.
    - Polling every 10 minutes isn't near real-time---failed messages sit in the queue longer than necessary.
  - Reprocessing:
    - Removing messages from the queue means they're no longer in SQS for reprocessing. Scientists must manually re-add corrected messages, which isn't automated.
- Does It Meet Requirements?:
  - Retain Failed Messages: Yes, via the output location.
  - Make Available to Scientists: Yes, if the location is accessible.
  - Future Reprocessing: No, messages are removed, and reprocessing isn't streamlined.
- Conclusion: This approach is flawed---FIFO isn't needed, the timing-based logic is unreliable, and reprocessing isn't well-supported. B is incorrect.

Option C: Create an SQS dead-letter queue. Modify the existing queue by including a `redrive` policy that sets the `Maximum Receives` setting to 1 and sets the dead-letter queue ARN to the ARN of the newly created



queue. Instruct the scientists to use the dead-letter queue to review the data that is not valid. Reprocess this data at a later time

Your pick! Let's break this down:

- What This Does:
    - SQS Dead-Letter Queue (DLQ):
      - A dead-letter queue is a secondary SQS queue where messages are sent after failing processing in the primary queue.
      - You create a new SQS queue to act as the DLQ.
    - Redrive Policy:
      - Modify the existing SQS queue to include a redrive policy:
- json
- ```
{
  "deadLetterTargetArn": "arn:aws:sqs:region:account-id:dead-letter-queue",
  "maxReceiveCount": 1
}
```
- Maximum Receives (maxReceiveCount): Set to 1, meaning a message is sent to the DLQ after being received once without successful processing (i.e., the application receives it, fails, and doesn't delete it).
 - Dead-Letter Queue ARN: Points to the new DLQ.
- Scientists Review:
 - Scientists access the DLQ to review failed messages.
- Reprocessing:
 - Messages in the DLQ can be reprocessed later (e.g., by moving them back to the primary queue after correction).
- Analysis:
 - How It Works:
 - The custom application receives a message from the SQS queue.
 - If it fails to transform the data (due to inconsistencies), it doesn't delete the message (either explicitly or by throwing an error, depending on how it's coded).
 - SQS increments the message's ApproximateReceiveCount. After 1 receive (maxReceiveCount = 1), the message is moved to the DLQ.

- Scientists can view messages in the DLQ via the SQS Console, CLI, or a custom app.
- Retain Failed Messages:
 - The DLQ retains failed messages, preventing them from clogging the primary queue.
- Make Available to Scientists:
 - Scientists can poll the DLQ to review failed messages. SQS messages contain the telemetry data (or a reference to it), which scientists can analyze.
- Future Reprocessing:
 - Messages in the DLQ can be reprocessed by:
 - Manually moving them back to the primary queue after correction (e.g., using the SQS Console's "Redrive" feature or a script).
 - Updating the application to handle the inconsistencies and replaying the messages.
 - SQS supports a redrive policy to move messages back (as of 2023, SQS added a "Start DLQ Redrive" feature), making reprocessing straightforward.
- MaxReceiveCount = 1:
 - Setting maxReceiveCount to 1 is aggressive---it moves messages to the DLQ after a single failed attempt. In practice, you might set it higher (e.g., 3-5) to allow retries, but for this scenario (where failures are due to data inconsistencies), 1 is acceptable since retries are unlikely to succeed without correction.
- Does It Meet Requirements?:
 - Retain Failed Messages: Yes, via the DLQ.
 - Make Available to Scientists: Yes, scientists can access the DLQ.
 - Future Reprocessing: Yes, messages can be moved back to the primary queue or reprocessed after correction.
- Automation:
 - The DLQ setup is fully automated---messages are moved automatically after failing.
 - Reviewing and reprocessing may involve manual steps (e.g., scientists correcting data), but the retention and availability are automated.
- Conclusion: This is a clean, SQS-native solution. The DLQ is designed for exactly this use case---handling failed messages in a queue. It's simple, automated, and supports reprocessing. C looks like the best choice.

Option D: Configure API Gateway to send messages to different SQS virtual queues that are named for each of the satellites. Update the application to use a new virtual queue for any data that it cannot transform, and

send the message to the new virtual queue. Instruct the scientists to use the virtual queue to review the data that is not valid. Reprocess this data at a later time

- What This Does:
 - Virtual Queues per Satellite:
 - "Virtual queues" isn't an SQS term---it likely means separate SQS queues (e.g., one per satellite: queue-satellite1, queue-satellite2).
 - API Gateway would route messages to the appropriate queue based on the satellite (e.g., using a mapping template or Lambda).
 - New Virtual Queue for Failed Messages:
 - The application, upon failing to transform a message, sends it to a new "failed" queue.
 - Scientists Review:
 - Scientists access the failed queue to review invalid messages.
 - Reprocessing:
 - Messages in the failed queue can be reprocessed later.
- Analysis:
 - Retain Failed Messages:
 - The failed queue acts like a DLQ, retaining messages the application can't process.
 - Make Available to Scientists:
 - Scientists can access the failed queue to review messages.
 - Future Reprocessing:
 - Messages in the failed queue can be reprocessed by moving them back to the primary queue(s) after correction.
 - Issues:
 - Per-Satellite Queues:
 - Creating separate queues per satellite isn't necessary for the requirement---it adds complexity without solving the core issue (handling failed messages).
 - The problem isn't about routing messages by satellite; it's about handling failures.
 - Application Changes:
 - The application must be modified to send failed messages to a new queue. This requires code changes (e.g., adding logic to publish to the failed queue), which is more invasive than configuring a DLQ.

- Automation:
 - Moving failed messages to the new queue is automated (via the application), but it requires app changes, whereas a DLQ (C) is a queue-level configuration.
- Does It Meet Requirements?:
 - Retain Failed Messages: Yes, via the failed queue.
 - Make Available to Scientists: Yes, scientists can access the queue.
 - Future Reprocessing: Yes, messages can be reprocessed.
- Conclusion: This works but is less efficient than a DLQ. It requires application changes and adds unnecessary complexity with per-satellite queues. A DLQ (C) is a simpler, SQS-native solution. D is not the best choice.

Analysis: Is C the Best Choice?

Your pick, C, uses an SQS dead-letter queue with a redrive policy:

- Does It Meet Requirements?:
 - Retain Failed Messages: Yes, the DLQ stores failed messages.
 - Make Available to Scientists: Yes, scientists can access the DLQ.
 - Future Reprocessing: Yes, messages can be reprocessed by moving them back to the primary queue.
- Automation:
 - The DLQ setup is fully automated---messages are moved to the DLQ after failing (maxReceiveCount = 1).
 - Reprocessing may involve manual steps (e.g., scientists correcting data), but the core requirement (retention and availability) is automated.
- Simplicity:
 - This leverages SQS's built-in DLQ feature, requiring no application changes---just a queue configuration.
 - It's a standard AWS pattern for handling failed messages.
- MaxReceiveCount:
 - Setting to 1 ensures immediate movement to the DLQ, which fits the scenario since retries are unlikely to succeed without data correction.
 - In practice, you might use a higher value (e.g., 3) to allow retries, but 1 is acceptable here.

Comparison to Alternatives:

- A: Duplicates the application's logic in Lambda, introduces race conditions, and requires manual reprocessing steps.
- B: FIFO queue is unnecessary, timing-based logic is unreliable, and reprocessing isn't streamlined.
- D: Works but requires application changes and adds unnecessary complexity with per-satellite queues.

C is the best choice:

- It uses SQS's native DLQ feature, designed for this exact scenario.
- It's simple (no app changes), automated, and supports reprocessing.

Why You Picked C

You chose C because:

- You recognized that failed messages need to be isolated from the primary queue to avoid clogging it, and a dead-letter queue is a logical fit.
- You saw that the DLQ makes messages available for scientists to review.
- You understood that messages in the DLQ can be reprocessed later, meeting the future reprocessing requirement.

Your reasoning is spot-on! The DLQ is the AWS-recommended way to handle failed messages in SQS---it's simple, automated, and aligns perfectly with the requirements. Even without deep familiarity with SQS, you picked the best solution.

Correct Answer: C. Create an SQS dead-letter queue. Modify the existing queue by including a redrive policy that sets the Maximum Receives setting to 1 and sets the dead-letter queue ARN to the ARN of the newly created queue. Instruct the scientists to use the dead-letter queue to review the data that is not valid. Reprocess this data at a later time.

A company requires that its internally facing web application be highly available. The architecture is made up of one Amazon EC2 web server instance and one NAT instance that provides outbound internet access for updates and accessing public data.

Which combination of architecture adjustments should the company implement to achieve high availability? (Choose two.)

- A. Add the NAT instance to an EC2 Auto Scaling group that spans multiple Availability Zones. Update the route tables.
- B. Create additional EC2 instances spanning multiple Availability Zones. Add an Application Load Balancer to split the load between them.
- C. Configure an Application Load Balancer in front of the EC2 instance. Configure Amazon CloudWatch alarms to recover the EC2 instance upon host failure.
- D. Replace the NAT instance with a NAT gateway in each Availability Zone. Update the route tables.
- E. Replace the NAT instance with a NAT gateway that spans multiple Availability Zones. Update the route tables.

The company has an internally facing web application running on one Amazon EC2 web server instance, with a NAT instance providing outbound internet access (e.g., for updates or public data). They want high availability (HA), meaning the system should survive failures like an instance crashing or an Availability Zone (AZ) outage. We need to pick two architecture adjustments to achieve this.

Option A: Add the NAT instance to an EC2 Auto Scaling group spanning multiple AZs, update route tables

This puts the NAT instance in an Auto Scaling group (ASG) across AZs, so if it fails, a new one launches in another AZ. You'd update private subnet route tables to point to the new NAT instance. It improves NAT availability, but it's clunky---NAT instances don't natively "span" AZs, and route table updates after a failover need scripting (e.g., via Lambda). Plus, it doesn't address the app instance's availability. A helps NAT resilience but not the app, and it's operationally messy.

Option B: Create additional EC2 instances spanning multiple AZs, add an Application Load Balancer (ALB)

Your first pick! This adds more EC2 instances for the app across AZs (e.g., in private subnets) and fronts them with an ALB to distribute traffic. It boosts app availability---if one instance or AZ fails, others handle the load. The ALB sits in public subnets, accepting inbound traffic and routing it to the private instances. However, it assumes the app receives inbound traffic (not stated but plausible), and it doesn't fix the NAT instance's single-point-of-failure issue for outbound traffic. B is solid for app HA but ignores the NAT problem.

Option C: Configure an ALB in front of the EC2 instance, configure CloudWatch alarms to recover on failure

This puts an ALB in front of the single EC2 app instance and uses CloudWatch alarms to trigger recovery (relaunch) if the instance fails. It improves resilience for that one instance, but it doesn't scale across AZs---recovery still means downtime, and a single AZ failure takes it all out. Like B, it assumes inbound traffic, and it leaves the NAT instance vulnerable. C is a half-measure---not enough for true HA.

Option D: Replace the NAT instance with a NAT gateway in each AZ, update route tables

Your second pick! This swaps the NAT instance for NAT gateways, one per AZ, and updates private subnet route tables to use them (e.g., subnet-1 routes to NAT-GW-1, subnet-2 to NAT-GW-2). NAT gateways are AWS-managed, highly available within an AZ, and eliminate the NAT instance's failure risk. Multiple NAT gateways ensure outbound traffic survives an AZ failure, assuming private subnets are split across AZs. It doesn't touch the app instance, but it fixes the NAT bottleneck. D is great for outbound HA and pairs well with app-focused fixes.

Option E: Replace the NAT instance with a NAT gateway spanning multiple AZs, update route tables

This suggests one NAT gateway "spanning" AZs, which isn't how NAT gateways work---they're AZ-specific. You place a NAT gateway in one AZ, and if that AZ fails, it's down (unlike an ASG'd NAT instance). The phrasing is odd---maybe it means one NAT gateway total, but then it's not multi-AZ resilient. Route tables point to this single NAT gateway, so an AZ failure kills outbound traffic. E is misleading or incorrect; NAT gateways don't span AZs.

Analysis: Are B and D Correct?

Your picks:

- B: Scales the app with multiple EC2 instances across AZs and an ALB. It's a classic HA move for inbound traffic, making the app resilient to instance or AZ failures. It assumes the app serves requests

(reasonable for an exam), but it doesn't touch the NAT issue.

- D: Replaces the NAT instance with per-AZ NAT gateways, ensuring outbound traffic (e.g., app updates, API calls) stays up even if an AZ fails. It's a best-practice upgrade from a NAT instance, complementing an app-focused fix.

These two together cover both sides of the system:

- App availability (B): Multi-AZ EC2 instances + ALB handle inbound load and failures.
- Outbound resilience (D): Multi-AZ NAT gateways ensure private instances can reach the internet (e.g., for patches or external services).

The catch? The question's intent isn't explicit. If it's all about the NAT instance's availability, D alone (or with A) might suffice, but B+D screams "full system HA," which AWS exams love. A lone NAT fix (D) leaves the app vulnerable; a lone app fix (B) risks outbound failure. Together, they're a dream team.

Alternatives:

- A: NAT-focused but ignores the app and is less elegant than D.
- C: Weak---single-instance recovery isn't HA.
- E: Misworded or impractical.

Given "choose two" and a holistic HA goal, B and D align perfectly: app HA (B) + outbound HA (D). Without the full question, I'd bet this is right---B upgrades the app, D upgrades the NAT, and both leverage multi-AZ resilience.

Explanation for Your Answer (B and D)

You're likely correct!

- B: You saw the need for app-level HA across AZs with an ALB---smart, especially if the app serves traffic. It's a DevOps staple.
- D: You caught that NAT instances are fragile and swapped them for NAT gateways per AZ, boosting outbound reliability. Spot-on upgrade.

Why they work: B ensures the app survives AZ failures, D ensures outbound connectivity does too. They don't overlap---B is app-focused, D is network-focused---making them a killer combo. If the intent was narrower (e.g., just NAT), D might pair with A, but B+D is the full-package answer exams often reward.

Probable Correct Answer: B and D.

Let me know the full question if I guessed the context wrong, or if you've got the official answer---otherwise, ready for the next one? You're crushing it!

A company's application development team uses Linux-based Amazon EC2 instances as bastion hosts. Inbound SSH access to the bastion hosts is restricted to specific IP addresses, as defined in the associated security groups. The company's security team wants to receive a notification if the security group rules are modified to allow SSH access from any IP address.

What should a DevOps engineer do to meet this requirement?

- A. Create an Amazon EventBridge rule with a source of `aws.cloudtrail` and the event name `AuthorizeSecurityGroupIngress`. Define an Amazon Simple Notification Service (Amazon SNS) topic as the target.
- B. Enable Amazon GuardDuty and check the findings for security groups in AWS Security Hub. Configure an Amazon EventBridge rule with a custom pattern that matches GuardDuty events with an output of `NON_COMPLIANT`. Define an Amazon Simple Notification Service (Amazon SNS) topic as the target.
- C. Create an AWS Config rule by using the `restricted-ssh` managed rule to check whether security groups disallow unrestricted incoming SSH traffic. Configure automatic remediation to publish a message to an Amazon Simple Notification Service (Amazon SNS) topic.
- D. Enable Amazon Inspector. Include the `Common Vulnerabilities and Exposures-1.1` rules package to check the security groups that are associated with the bastion hosts. Configure Amazon Inspector to publish a message to an Amazon Simple Notification Service (Amazon SNS) topic.

Understanding the Problem

- Setup:
 - EC2 Instances as Bastion Hosts: Linux-based EC2 instances used for secure access (likely to other resources in a VPC).
 - Security Groups: Restrict inbound SSH access (port 22) to specific IP addresses (e.g., `203.0.113.0/24`).
 - Security Team Requirement: Get notified if a security group rule is modified to allow SSH from any IP (e.g., `0.0.0.0/0` on port 22).
- Requirements:
 - Detect when a security group rule allows unrestricted SSH access (port 22 from `0.0.0.0/0`).
 - Send a notification to the security team (implied to be via Amazon SNS, based on the options).
- Key Considerations:
 - We need to monitor security group rule changes or configurations.
 - The detection should be specific to SSH (port 22) and unrestricted IPs (`0.0.0.0/0`).
 - The solution should trigger a notification (e.g., via SNS) when the condition is met.
 - AWS offers multiple services for monitoring and alerting (e.g., EventBridge, AWS Config, GuardDuty, Inspector), so we need to pick the most appropriate one.

Let's evaluate each option.

Option A: Create an Amazon EventBridge rule with a source of `aws.cloudtrail` and the event name `AuthorizeSecurityGroupIngress`. Define an Amazon Simple Notification Service (Amazon SNS) topic as the target

- What This Does:

- Amazon EventBridge:
 - EventBridge (formerly CloudWatch Events) can capture AWS API calls logged by CloudTrail.
 - The source `aws.cloudtrail` indicates we're looking for CloudTrail events.
- Event Name `AuthorizeSecurityGroupIngress`:
 - This is an EC2 API call that adds an inbound rule to a security group (e.g., `AuthorizeSecurityGroupIngress` to allow SSH from `0.0.0.0/0`).
 - The EventBridge rule pattern might look like:

json

```
{
  "source": ["aws.cloudtrail"],
  "detail-type": ["AWS API Call via CloudTrail"],
  "detail": {
    "eventSource": ["ec2.amazonaws.com"],
    "eventName": ["AuthorizeSecurityGroupIngress"]
  }
}
```

- SNS Topic as Target:
 - When the rule matches, it sends a notification to an SNS topic, which can email the security team.
- Analysis:
 - Detecting Rule Changes:
 - This rule triggers whenever any inbound rule is added to a security group (`AuthorizeSecurityGroupIngress`), not just SSH rules or unrestricted ones.
 - It catches rule additions but doesn't evaluate the rule's content (e.g., port 22, `0.0.0.0/0`).
 - Specificity:
 - The requirement is to detect SSH (port 22) from any IP (`0.0.0.0/0`). This rule would trigger for all inbound rules (e.g., port 80, specific IPs), generating many false positives.
 - To filter for SSH and `0.0.0.0/0`, you'd need to:
 - Add a detailed event pattern to match the rule's parameters (e.g., `detail.requestParameters.ipPermissions.items` contains port 22 and `0.0.0.0/0`).
 - Or, use a Lambda function as the target to inspect the event and filter before notifying SNS.

- The option doesn't specify this filtering, so it's too broad as written.
- Other Operations:
 - AuthorizeSecurityGroupIngress covers adding new rules, but modifying existing rules uses ModifySecurityGroupRules or UpdateSecurityGroupRuleDescriptionsIngress. This rule misses those operations.
- Notification:
 - SNS notification works, but without filtering, the security team gets too many alerts.
- Does It Meet Requirements?:
 - Detect SSH from Any IP: Partially---it catches rule additions but isn't specific to SSH or 0.0.0.0/0.
 - Notification: Yes, via SNS, but with false positives.
- Conclusion: This catches security group changes but lacks specificity for SSH and unrestricted IPs, leading to noise for the security team. It also misses some rule modification operations. A is not the best choice.

Option B: Enable Amazon GuardDuty and check the findings for security groups in AWS Security Hub. Configure an Amazon EventBridge rule with a custom pattern that matches GuardDuty events with an output of NON_COMPLIANT. Define an Amazon Simple Notification Service (Amazon SNS) topic as the target

- What This Does:
 - Amazon GuardDuty:
 - GuardDuty is a threat detection service that monitors for malicious activity and unauthorized behavior.
 - It can detect security group changes that indicate potential threats (e.g., overly permissive rules).
 - AWS Security Hub:
 - Security Hub aggregates findings from GuardDuty and other services (e.g., AWS Config, Inspector).
 - GuardDuty findings related to security groups (e.g., "Port 22 open to 0.0.0.0/0") can appear in Security Hub.
 - EventBridge Rule:
 - Match GuardDuty findings with a result of NON_COMPLIANT (though GuardDuty findings don't use this exact terminology---Security Hub does).
 - Example pattern:

json

```
{
  "source": ["aws.guardduty"],
  "detail-type": ["GuardDuty Finding"],
  "detail": {
    "type": ["Recon:EC2/PortProbeUnprotectedPort"],
    "resource.resourceType": ["Instance"]
  }
}
```

- Target an SNS topic for notifications.
- Analysis:
 - Detecting SSH from Any IP:
 - GuardDuty can detect overly permissive security groups, but its focus is on threat detection, not compliance.
 - GuardDuty findings like Recon:EC2/PortProbeUnprotectedPort might flag port 22 open to 0.0.0.0/0, but this is triggered by probe activity (e.g., an external IP scanning port 22), not the rule change itself.
 - If no malicious activity occurs, GuardDuty might not generate a finding, even if the rule is added.
 - Security Hub:
 - Security Hub aggregates GuardDuty findings but doesn't directly detect security group rule changes for compliance.
 - Security Hub uses AWS Config rules for compliance checks (e.g., restricted-ssh), which we'll see in C.
 - EventBridge Rule:
 - GuardDuty findings don't use NON_COMPLIANT---that's an AWS Config term. GuardDuty findings have types like Recon:EC2/PortProbeUnprotectedPort and severities (e.g., High, Medium).
 - You could match GuardDuty findings related to port 22, but it's indirect and depends on external activity.
 - Does It Meet Requirements?:
 - Detect SSH from Any IP: No---GuardDuty might miss the rule change if there's no threat activity.
 - Notification: Yes, via SNS, but only if GuardDuty triggers a finding.
 - Practicality:

- GuardDuty is better for threat detection (e.g., detecting actual SSH brute-force attacks) than for compliance monitoring.
- This approach introduces unnecessary complexity (GuardDuty + Security Hub) for a simple compliance check.
- Conclusion: GuardDuty isn't the right tool for this---it's threat-focused, not compliance-focused, and might miss the rule change. B is incorrect.

Option C: Create an AWS Config rule by using the restricted-ssh managed rule to check whether security groups disallow unrestricted incoming SSH traffic. Configure automatic remediation to publish a message to an Amazon Simple Notification Service (Amazon SNS) topic

Your pick! Let's break this down:

- What This Does:
 - AWS Config Rule:
 - AWS Config tracks the configuration of AWS resources and evaluates them against desired states using rules.
 - The restricted-ssh managed rule checks if security groups allow unrestricted SSH access (port 22 from 0.0.0.0/0 or ::/0 for IPv6).
 - If a security group rule permits SSH from any IP, the rule marks it as NON_COMPLIANT.
 - Automatic Remediation:
 - AWS Config supports automatic remediation for non-compliant resources.
 - Configure a remediation action to publish a message to an SNS topic when the rule evaluates to NON_COMPLIANT.
 - Example remediation setup:
 - Use an SSM Automation document (e.g., AWS-PublishSNSNotification).
 - Specify the SNS topic ARN and a message (e.g., "Security group SG-123 allows unrestricted SSH access").
- Analysis:
 - Detecting SSH from Any IP:
 - The restricted-ssh rule is designed for this exact use case---it checks all security groups for inbound rules allowing SSH (port 22) from 0.0.0.0/0 or ::/0.
 - It evaluates rules whenever a security group is created or modified (via CloudTrail events).
 - Example evaluation:
 - Security group rule: port 22, 0.0.0.0/0 → NON_COMPLIANT.

- Security group rule: port 22, 203.0.113.0/24 → COMPLIANT.
- Notification:
 - The remediation action publishes a message to an SNS topic when the rule fails, notifying the security team.
 - SNS can send emails or trigger other actions (e.g., a Lambda for custom handling).
- Timing:
 - AWS Config evaluates rules periodically (e.g., every 10 minutes) or on configuration changes (triggered by CloudTrail). This is near real-time for most use cases, sufficient for the requirement.
- Additional Features:
 - The remediation action could also remove the offending rule (e.g., using AWS-DeleteSecurityGroupRule), but the question only asks for notification, not remediation.
 - AWS Config provides a compliance dashboard, giving the security team visibility into non-compliant resources.
- Does It Meet Requirements?:
 - Detect SSH from Any IP: Yes, restricted-ssh is purpose-built for this.
 - Notification: Yes, via SNS.
- Simplicity:
 - This is a straightforward solution: one Config rule + remediation action to notify SNS.
 - It's a standard AWS pattern for compliance monitoring and alerting.
- Conclusion: This is the cleanest, most targeted solution. The restricted-ssh rule directly addresses the requirement, and SNS notification ensures the security team is alerted. C looks like the best choice.

Option D: Enable Amazon Inspector. Include the Common Vulnerabilities and Exposures-1.1 rules package to check the security groups that are associated with the bastion hosts. Configure Amazon Inspector to publish a message to an Amazon Simple Notification Service (Amazon SNS) topic

- What This Does:
 - Amazon Inspector:
 - Inspector is a security assessment service that checks for vulnerabilities and misconfigurations in EC2 instances and container workloads.
 - It uses rules packages (e.g., Common Vulnerabilities and Exposures, or CVE) to scan for issues.
 - CVE-1.1 Rules Package:

- The CVE rules package checks for known vulnerabilities in software installed on the instance (e.g., outdated SSH versions with CVEs).
- SNS Notification:
 - Inspector can publish findings to an SNS topic.
- Analysis:
 - Detecting SSH from Any IP:
 - Inspector doesn't check security group rules---it focuses on the instance's software and configuration (e.g., installed packages, open ports on the host).
 - The CVE-1.1 package looks for vulnerabilities in software (e.g., an outdated OpenSSH version), not security group rules.
 - Inspector has a "Network Reachability" rules package (not mentioned here) that can check for overly permissive security groups, but:
 - It's not part of the CVE package.
 - It's more about assessing network exposure risks (e.g., "instance is reachable from the internet on port 22"), not specifically SSH from 0.0.0.0/0.
 - Notification:
 - Inspector can notify via SNS, but the findings won't match the requirement (SSH from any IP).
 - Does It Meet Requirements?:
 - Detect SSH from Any IP: No---Inspector focuses on software vulnerabilities, not security group rules.
 - Notification: Yes, but for the wrong thing.
 - Practicality:
 - Inspector is overkill for this simple compliance check. It's better for vulnerability scanning (e.g., finding CVEs in the OS) than for security group monitoring.
- Conclusion: Inspector isn't the right tool for this---it can't detect security group rule changes for SSH access. D is incorrect.

Analysis: Is C the Best Choice?

Your pick, C, uses the AWS Config restricted-ssh rule with automatic remediation to notify via SNS:

- Does It Meet Requirements?:
 - Detect SSH from Any IP: Yes, restricted-ssh checks for exactly this condition (port 22, 0.0.0.0/0).
 - Notification: Yes, the remediation action publishes to SNS.

- Accuracy:
 - The rule is highly specific---no false positives (only triggers for SSH from unrestricted IPs).
 - It catches both new rules and modifications (via `ModifySecurityGroupRules`, `AuthorizeSecurityGroupIngress`, etc.).
- Timing:
 - Config evaluations are near real-time (triggered by CloudTrail events, typically within minutes), sufficient for the requirement.
- Comparison to Alternatives:
 - A: Catches rule additions but isn't specific to SSH or 0.0.0.0/0, and misses some modification operations.
 - B: GuardDuty focuses on threats, not compliance, and might miss the rule change.
 - D: Inspector doesn't check security group rules for this scenario.

C is the best choice:

- It uses a purpose-built AWS Config rule (restricted-ssh) for this exact scenario.
- It's simple, specific, and ensures notifications via SNS.
- It aligns with AWS best practices for compliance monitoring.

Why You Picked C

You chose C because:

- You recognized that the requirement is about compliance (checking security group rules for a specific condition).
- You saw that AWS Config is designed for this type of configuration monitoring, and the restricted-ssh rule matches the need exactly (SSH from any IP).
- You understood that automatic remediation can trigger an SNS notification, meeting the notification requirement.

Your reasoning is spot-on! The restricted-ssh rule is the AWS-recommended approach for this use case---it's precise, automated, and integrates seamlessly with SNS for notifications. You picked the best solution!

Correct Answer: C. Create an AWS Config rule by using the restricted-ssh managed rule to check whether security groups disallow unrestricted incoming SSH traffic. Configure automatic remediation to publish a message to an Amazon Simple Notification Service (Amazon SNS) topic.

A company hosts its staging website using an Amazon EC2 instance backed with Amazon EBS storage. The company wants to recover quickly with minimal data losses in the event of network connectivity issues or power failures on the EC2 instance.

Which solution will meet these requirements?

- A. Add the instance to an EC2 Auto Scaling group with the minimum, maximum, and desired capacity set to 1.
- B. Add the instance to an EC2 Auto Scaling group with a lifecycle hook to detach the EBS volume when the EC2 instance shuts down or terminates.
- C. Create an Amazon CloudWatch alarm for the StatusCheckFailed System metric and select the EC2 action to recover the instance.
- D. Create an Amazon CloudWatch alarm for the StatusCheckFailed Instance metric and select the EC2 action to reboot the instance.

Understanding the Problem

- Setup:
 - EC2 Instance: A single instance hosting a staging website.
 - Amazon EBS Storage: The instance uses EBS volumes for storage (e.g., root volume and possibly additional volumes for data).
 - Failure Scenarios:
 - Network connectivity issues: Could cause the instance to become unreachable (e.g., Status Check failures).
 - Power failures: Could cause the instance to stop or crash (e.g., underlying host failure).
- Requirements:
 - Recover Quickly: Minimize downtime after a failure (e.g., network or power issue).
 - Minimal Data Losses: Ensure data on the EBS volume is preserved and the application resumes with little to no data loss.
- Key Considerations:
 - Network connectivity or power failures can cause EC2 Status Check failures:
 - System Status Check: Fails if the underlying host has issues (e.g., power failure, network loss).
 - Instance Status Check: Fails if the instance OS or software is unresponsive.
 - EBS volumes are durable---data persists even if the instance fails, as long as the volume isn't deleted.
 - A single instance without redundancy (e.g., Auto Scaling) means recovery depends on restarting or replacing the instance.
 - "Minimal data losses" implies preserving EBS data and ensuring the application resumes where it left off (e.g., no corruption or loss of in-flight data).

Let's evaluate each option to find the best solution.

Option A: Add the instance to an EC2 Auto Scaling group with the minimum, maximum, and desired capacity set to 1

- What This Does:
 - EC2 Auto Scaling Group (ASG):
 - Places the EC2 instance in an ASG with min=1, max=1, and desired=1, meaning exactly one instance runs at all times.
 - ASG monitors instance health (e.g., via EC2 Status Checks or ELB health checks if configured).
 - Recovery Behavior:
 - If the instance fails a health check (e.g., due to network or power failure), ASG terminates it and launches a replacement instance.
 - The replacement instance is a new instance (new instance ID, new IP unless using an Elastic IP).
- Analysis:
 - Recover Quickly:
 - ASG can recover quickly---when the instance fails, ASG launches a new one within minutes (typically 2-5 minutes, depending on the AMI and startup scripts).
 - This meets the "recover quickly" requirement.
 - Minimal Data Losses:
 - The original instance's EBS volume is preserved by default (unless DeleteOnTermination is true, which is the default for root volumes but can be disabled).
 - However, ASG launches a new instance from the launch template, which typically creates a new root volume from the AMI snapshot---not the original EBS volume.
 - To preserve data:
 - You'd need to attach the original EBS volume to the new instance (e.g., using a lifecycle hook or user data script to reattach the volume).
 - Or, use a separate EBS volume for data (not the root volume) and ensure it's reattached to the new instance.
 - Without this configuration, the new instance starts fresh from the AMI, potentially losing data (e.g., application state, logs, or in-flight data not yet written to a durable store).
 - Challenges:
 - The question doesn't specify if the EBS volume is the root volume or a separate data volume. If it's the root volume, ASG's default behavior (launching a new instance) might not automatically reattach the original volume, leading to data loss.

- Network connectivity issues might not always trigger a health check failure (e.g., if using EC2 Status Checks, it depends on the failure type).
- Does It Meet Requirements?:
 - Recover Quickly: Yes, ASG replaces the instance within minutes.
 - Minimal Data Losses: No, unless additional configuration ensures the original EBS volume is reattached, which isn't specified.
- Conclusion: ASG ensures quick recovery but doesn't guarantee minimal data loss without extra steps to preserve the EBS volume. A is not the best choice.

Option B: Add the instance to an EC2 Auto Scaling group with a lifecycle hook to detach the EBS volume when the EC2 instance shuts down or terminates

Your pick! Let's break this down:

- What This Does:
 - EC2 Auto Scaling Group:
 - Same as A---places the instance in an ASG with min=1, max=1, desired=1, ensuring one instance runs.
 - If the instance fails (e.g., network or power failure), ASG terminates and replaces it.
 - Lifecycle Hook:
 - Adds a lifecycle hook to the ASG for the EC2_INSTANCE_TERMINATING lifecycle event.
 - When the instance is about to terminate (e.g., due to a failed health check), the hook pauses the termination in the Terminating:Wait state.
 - During this pause, a lifecycle action (e.g., a Lambda function) detaches the EBS volume from the instance.
 - After the action completes, the lifecycle hook allows the termination to proceed, and ASG launches a new instance.
- Analysis:
 - Recover Quickly:
 - ASG replaces the failed instance within minutes (slightly delayed by the lifecycle hook, e.g., 5-10 minutes depending on the hook timeout).
 - This meets the "recover quickly" requirement, though the lifecycle hook adds a small delay.
 - Minimal Data Losses:
 - Detaching the EBS volume ensures it's preserved when the instance terminates.

- However, the new instance launched by ASG doesn't automatically reattach the detached volume:
 - The new instance starts with a fresh root volume (from the AMI snapshot).
 - You'd need another lifecycle hook (e.g., on EC2_INSTANCE_LAUNCHING) or user data script to reattach the detached volume to the new instance.
- Without reattachment, the new instance starts fresh, losing access to the data on the original EBS volume (e.g., application data, state).
- The lifecycle hook protects the volume from deletion but doesn't ensure the new instance uses it, which could lead to data loss in the context of the application.
- Intent of Detaching:
 - Detaching the volume might be useful for backup or forensic analysis (e.g., taking a snapshot of the volume before termination), but the question focuses on recovery with minimal data loss, implying the application should resume using the same data.
- Does It Meet Requirements?:
 - Recover Quickly: Yes, ASG replaces the instance, though the lifecycle hook adds a small delay.
 - Minimal Data Losses: No, detaching the volume preserves it, but the new instance doesn't use it without additional configuration.
- Conclusion: This improves on A by preserving the EBS volume, but it doesn't ensure the new instance uses the volume, failing the "minimal data losses" requirement. B is not the best choice.

Option C: Create an Amazon CloudWatch alarm for the StatusCheckFailed System metric and select the EC2 action to recover the instance

- What This Does:
 - CloudWatch Alarm:
 - Monitors the StatusCheckFailed_System metric for the EC2 instance.
 - This metric triggers if the System Status Check fails (e.g., due to network connectivity issues, power failures, or underlying host problems).
 - EC2 Recover Action:
 - The "recover" action stops and restarts the instance on the same or new hardware (if the original host is unavailable).
 - Recovery preserves:
 - Instance ID
 - Private and public IP addresses (if not using an Elastic IP)

- EBS volumes (all attached volumes remain attached)
 - Instance metadata and tags
 - The instance is rebooted, typically within a few minutes (e.g., 2-5 minutes).
- Analysis:
 - Recover Quickly:
 - The recover action is fast---AWS stops and restarts the instance within minutes, minimizing downtime.
 - This meets the "recover quickly" requirement.
 - Minimal Data Losses:
 - Since the instance ID and EBS volumes are preserved, the instance restarts with all its data intact.
 - EBS volumes are durable---they persist through the recovery process, and data written to disk (e.g., application data, logs) is preserved.
 - The only potential data loss is in-flight data (e.g., data in memory not yet written to disk at the time of failure). However:
 - Network connectivity issues typically don't cause data corruption on EBS.
 - Power failures (host-level) might cause in-memory data loss, but EBS data remains intact, and the application can resume from the last consistent state.
 - This minimizes data loss compared to launching a new instance (A, B), as the same instance and volumes are reused.
 - Applicability:
 - StatusCheckFailed_System covers both network connectivity and power failure scenarios:
 - Network issues: Can cause System Status Check failures (e.g., loss of connectivity to the host).
 - Power failures: Cause System Status Check failures (e.g., host failure).
 - Recovery works for most instance types (except some older ones like T1).
 - Does It Meet Requirements?:
 - Recover Quickly: Yes, recovery happens within minutes.
 - Minimal Data Losses: Yes, EBS data is preserved, and the instance resumes with the same state (minus in-memory data).
- Conclusion: This directly addresses both requirements---quick recovery via the EC2 recover action, and minimal data loss by preserving the instance and EBS volumes. C looks like the best choice.

Option D: Create an Amazon CloudWatch alarm for the `StatusCheckFailed_Instance` metric and select the EC2 action to reboot the instance

- What This Does:
 - CloudWatch Alarm:
 - Monitors the `StatusCheckFailed_Instance` metric.
 - This metric triggers if the Instance Status Check fails (e.g., OS-level issues like kernel panics, software crashes, or resource exhaustion).
 - EC2 Reboot Action:
 - The "reboot" action restarts the instance without changing its underlying hardware.
 - The instance retains its ID, IPs, and EBS volumes.
- Analysis:
 - Recover Quickly:
 - A reboot is fast---typically completes in 1-2 minutes.
 - This meets the "recover quickly" requirement.
 - Minimal Data Losses:
 - Rebooting preserves the instance and EBS volumes, minimizing data loss (same as C).
 - In-memory data might be lost, but EBS data remains intact.
 - Applicability:
 - `StatusCheckFailed_Instance` covers OS-level failures, but:
 - Network connectivity issues typically cause System Status Check failures (`StatusCheckFailed_System`), not Instance Status Check failures, unless the network issue causes the OS to fail (e.g., resource starvation).
 - Power failures (host-level) cause System Status Check failures, not Instance Status Check failures.
 - This option doesn't cover the specified failure scenarios (network or power issues) as effectively as C.
 - Does It Meet Requirements?:
 - Recover Quickly: Yes, reboot is quick.
 - Minimal Data Losses: Yes, EBS data is preserved.
 - Covering Scenarios: No, it misses network and power failures (System Status issues).

- Conclusion: This recovers quickly with minimal data loss but doesn't address the specified failure scenarios (network and power), which are System Status issues. D is not the best choice.

Analysis: Is B the Best Choice?

Your pick, B, uses an Auto Scaling group with a lifecycle hook to detach the EBS volume on termination:

- Does It Meet Requirements?:
 - Recover Quickly: Yes, ASG replaces the instance, though the lifecycle hook adds a small delay.
 - Minimal Data Losses: No---detaching the volume preserves it, but the new instance doesn't automatically use it, potentially leading to data loss in the application context.
- Comparison to Alternatives:
 - A: Similar to B but without the lifecycle hook---same issue with data loss due to new volume creation.
 - C: Uses EC2 recovery, preserving the instance and EBS volumes, ensuring both quick recovery and minimal data loss. Also directly addresses network and power failures.
 - D: Reboots the instance, which is quick and preserves data, but doesn't cover the specified failure scenarios.

Why B Falls Short:

- While B ensures the EBS volume is preserved (detached), the new instance launched by ASG doesn't automatically reattach it. The application would start fresh on a new volume (from the AMI), losing access to the data on the original volume unless additional steps are taken (e.g., reattaching the volume via user data or another lifecycle hook). This doesn't meet the "minimal data losses" requirement, as the application needs to resume with its data intact.

C is the best choice:

- EC2 recovery (StatusCheckFailed_System) directly addresses network and power failures.
- It recovers the same instance, preserving EBS volumes and minimizing data loss.
- It's faster than ASG (no lifecycle hook delay) and simpler (no need to manage volume reattachment).

Why You Picked B

You chose B because:

- You recognized that network or power failures could cause the instance to fail, and an Auto Scaling group can replace it (quick recovery).
- You saw the lifecycle hook as a way to preserve the EBS volume (detaching it), aiming to minimize data loss.

Your reasoning is logical---ASG ensures recovery, and detaching the volume protects the data from deletion. However, the key miss is that the new instance doesn't automatically use the detached volume, which could lead to data loss in the application context (e.g., the website can't access its previous state). C is better

because EC2 recovery preserves the instance and its volumes, ensuring the application resumes with minimal disruption.

Correct Answer: C. Create an Amazon CloudWatch alarm for the StatusCheckFailed System metric and select the EC2 action to recover the instance.

A rapidly growing company wants to scale for developer demand for AWS development environments. Development environments are created manually in the AWS Management Console. The networking team uses AWS CloudFormation to manage the networking infrastructure, exporting stack output values for the Amazon VPC and all subnets. The development environments have common standards, such as Application Load Balancers, Amazon EC2 Auto Scaling groups, security groups, and Amazon DynamoDB tables. To keep up with demand, the DevOps engineer wants to automate the creation of development environments. Because the infrastructure required to support the application is expected to grow, there must be a way to easily update the deployed infrastructure. CloudFormation will be used to create a template for the development Environments.

Which approach will meet these requirements and quickly provide consistent AWS environments for developers?

- A. Use Fn::ImportValue intrinsic functions in the Resources section of the template to retrieve Virtual Private Cloud (VPC) and subnet values. Use CloudFormation StackSets for the development environments, using the Count input parameter to indicate the number of environments needed. Use the UpdateStackSet command to update existing development environments.
- B. Use nested stacks to define common infrastructure components. To access the exported values, use TemplateURL to reference the networking team's template. To retrieve Virtual Private Cloud (VPC) and subnet values, use Fn::ImportValue intrinsic functions in the Parameters section of the root template. Use the CreateChangeSet and ExecuteChangeSet commands to update existing development environments.
- C. Use nested stacks to define common infrastructure components. Use Fn::ImportValue intrinsic functions with the resources of the nested stack to retrieve Virtual Private Cloud (VPC) and subnet values. Use the CreateChangeSet and ExecuteChangeSet commands to update existing development environments.
- D. Use Fn::ImportValue intrinsic functions in the Parameters section of the root template to retrieve Virtual Private Cloud (VPC) and subnet values. Define the development resources in the order they need to be created in the CloudFormation nested stacks. Use the CreateChangeSet. and ExecuteChangeSet commands to update existing development environments.

Understanding the Problem

- Setup:
 - Current Process: Development environments are created manually via the AWS Management Console, which is time-consuming and error-prone.
 - Networking Infrastructure: Managed by the networking team using CloudFormation, with stack outputs for the VPC ID and subnet IDs (e.g., exported values like VPCId, PublicSubnet1, PrivateSubnet1).
 - Development Environments:

- Include common components: ALBs, EC2 Auto Scaling groups, security groups, DynamoDB tables.
- Need to be consistent across environments (e.g., same architecture, configurations).
- Requirements:
 - Automate Creation: Use CloudFormation to create development environments automatically.
 - Consistency: Ensure all environments follow the same standards.
 - Scalability: Support growing demand (e.g., easily create new environments).
 - Easy Updates: Allow updates to existing environments as infrastructure requirements evolve.
- Key Considerations:
 - CloudFormation must integrate with the networking team's stack outputs (VPC and subnet IDs).
 - The solution should modularize the infrastructure for maintainability (e.g., separate concerns like networking vs. app resources).
 - Updates to existing environments should be seamless (e.g., using CloudFormation's update capabilities).
 - We need to ensure the approach scales (e.g., supports multiple environments) and maintains consistency.

Let's evaluate each option to find the best approach.

Option A: Use `Fn::ImportValue` intrinsic functions in the Resources section of the template to retrieve Virtual Private Cloud (VPC) and subnet values. Use CloudFormation StackSets for the development environments, using the Count input parameter to indicate the number of environments needed. Use the `UpdateStackSet` command to update existing development environments

- What This Does:
 - `Fn::ImportValue`:
 - Retrieves the exported VPC and subnet values (e.g., `VPCId`, `PublicSubnet1`) from the networking team's stack in the Resources section of the template.
 - Example:

yaml

```
Resources:
  ALB:
    Type: AWS::ElasticLoadBalancingV2::LoadBalancer
    Properties:
      Subnets:
        - Fn::ImportValue: PublicSubnet1
        - Fn::ImportValue: PublicSubnet2
```



```
SecurityGroups:  
- !Ref ALBSecurityGroup
```

- CloudFormation StackSets:
 - StackSets allow you to deploy a CloudFormation stack across multiple accounts and regions.
 - Here, it's used to create multiple development environments (e.g., one stack instance per environment).
- Count Input Parameter:
 - Pass a Count parameter to StackSets to specify the number of environments (e.g., Count: 3 creates three stack instances).
- UpdateStackSet Command:
 - Use UpdateStackSet to update all stack instances (development environments) when the template changes.
- Analysis:
 - Automation:
 - StackSets automate the creation of multiple environments across accounts/regions.
 - However, the question implies environments are in the same account/region (since networking is a single stack), so StackSets might be overkill for this use case.
 - Consistency:
 - StackSets ensure all environments are created from the same template, maintaining consistency.
 - Scalability:
 - The Count parameter allows scaling the number of environments, but:
 - StackSets don't natively support a Count parameter---you'd need to script the creation of stack instances (e.g., via AWS CLI or SDK), passing different parameters for each environment (e.g., unique subnet sets).
 - Each environment needs unique resources (e.g., ALB, Auto Scaling group), which requires parameterization (e.g., different names, subnets), but the option doesn't specify how this is handled.
 - Easy Updates:
 - UpdateStackSet updates all stack instances, meeting the update requirement.
 - Issues:

- StackSets Complexity:
 - StackSets are designed for multi-account/region deployments. For a single account/region, they add overhead (e.g., managing stack instances, OperationPreferences).
 - The question doesn't indicate multi-account/region needs, so StackSets might be unnecessary.
- Resource Uniqueness:
 - Each environment needs unique resources (e.g., ALB, DynamoDB table names). Without parameterization or nested stacks, a single template might create conflicts (e.g., duplicate resource names).
- Networking Integration:
 - Fn::ImportValue in the Resources section is correct---it retrieves VPC/subnet values for use in ALB, Auto Scaling groups, etc.
- Does It Meet Requirements?:
 - Automate Creation: Yes, via StackSets.
 - Consistency: Yes, all environments use the same template.
 - Scalability: Yes, but Count is misleading---requires scripting to create multiple stack instances.
 - Easy Updates: Yes, via UpdateStackSet.
- Conclusion: This works but overcomplicates the solution with StackSets. A simpler approach (e.g., nested stacks in a single account) would suffice unless multi-account/region deployment is required, which isn't specified. A is not the best choice.

Option B: Use nested stacks to define common infrastructure components. To access the exported values, use TemplateURL to reference the networking team's template. To retrieve Virtual Private Cloud (VPC) and subnet values, use Fn::ImportValue intrinsic functions in the Parameters section of the root template. Use the CreateChangeSet and ExecuteChangeSet commands to update existing development environments

- What This Does:
 - Nested Stacks:
 - Nested stacks break the CloudFormation template into smaller templates (e.g., one for ALB, one for Auto Scaling, etc.).
 - A root template references child templates via the AWS::CloudFormation::Stack resource.
 - TemplateURL for Networking:
 - Suggests referencing the networking team's template directly as a nested stack using TemplateURL.

- Fn::ImportValue in Parameters:

- Retrieves VPC and subnet values in the Parameters section of the root template:

yaml

```
Parameters:
  VPCId:
    Type: String
    Default: !ImportValue VPCId
  PublicSubnet1:
    Type: String
    Default: !ImportValue PublicSubnet1
```

- These parameters are passed to child stacks.

- CreateChangeSet and ExecuteChangeSet:

- Use change sets to update existing environments (preview changes and apply them).

- Analysis:

- Nested Stacks:

- Good for modularity---separates concerns (e.g., ALB stack, Auto Scaling stack), making the template maintainable.

- TemplateURL Issue:

- TemplateURL is used to reference a nested stack template (e.g., an S3 URL for a child template).
 - The networking team's template is an existing stack, not a template to be deployed as part of this stack.
 - Using TemplateURL to reference the networking stack implies redeploying it, which isn't the intent---the networking stack already exists, and we need its outputs (VPCId, subnets), not to redeploy it.
 - This is a misunderstanding of how to access existing stack outputs.

- Fn::ImportValue in Parameters:

- Using Fn::ImportValue in the Parameters section with Default is correct---it retrieves the exported values and makes them available to the root template.
 - However, this is a stylistic choice---Fn::ImportValue can also be used directly in the Resources section (as in other options).

- Updates:

- CreateChangeSet and ExecuteChangeSet allow updating existing stacks, meeting the update requirement.
- Scalability:
 - Nested stacks help with modularity but don't address creating multiple environments. You'd need to deploy the root stack multiple times with different parameters (e.g., unique names for ALBs, tables), which isn't specified.
- Does It Meet Requirements?:
 - Automate Creation: Yes, via CloudFormation.
 - Consistency: Yes, nested stacks ensure consistent components.
 - Scalability: Not addressed---requires manual stack creation for each environment.
 - Easy Updates: Yes, via change sets.
 - Networking Integration: No, TemplateURL is incorrect for accessing existing stack outputs.
- Conclusion: The TemplateURL mistake makes this option incorrect---it misuses the networking team's stack. Otherwise, nested stacks and change sets are good practices, but this doesn't fully meet the requirements. B is incorrect.

Option C: Use nested stacks to define common infrastructure components. Use Fn::ImportValue intrinsic functions with the resources of the nested stack to retrieve Virtual Private Cloud (VPC) and subnet values. Use the CreateChangeSet and ExecuteChangeSet commands to update existing development environments

Your pick! Let's break this down:

- What This Does:
 - Nested Stacks:
 - Define common components (ALB, Auto Scaling, security groups, DynamoDB) as nested stacks.
 - Example structure:
 - root-template.yaml: The main template.
 - alb-stack.yaml: Defines the ALB.
 - asg-stack.yaml: Defines the Auto Scaling group.
 - dynamo-stack.yaml: Defines the DynamoDB table.
 - In root-template.yaml:

yaml

```
Resources:
  ALBStack:
    Type: AWS::CloudFormation::Stack
    Properties:
      TemplateURL: alb-stack.yaml
      Parameters:
        VPCId: !ImportValue VPCId
        Subnet1: !ImportValue PublicSubnet1
        Subnet2: !ImportValue PublicSubnet2
  ASGStack:
    Type: AWS::CloudFormation::Stack
    Properties:
      TemplateURL: asg-stack.yaml
      Parameters:
        VPCId: !ImportValue VPCId
        Subnet1: !ImportValue PrivateSubnet1
```

- Fn::ImportValue in Nested Stack Resources:
 - Each nested stack retrieves VPC and subnet values directly in its Resources section:

yaml

```
# alb-stack.yaml
Resources:
  ALB:
    Type: AWS::ElasticLoadBalancingV2::LoadBalancer
    Properties:
      Subnets:
        - Fn::ImportValue: PublicSubnet1
        - Fn::ImportValue: PublicSubnet2
```

- CreateChangeSet and ExecuteChangeSet:
 - Use change sets to update existing environments.
- Analysis:
 - Automation:
 - CloudFormation automates the creation of development environments via templates.
 - Consistency:
 - Nested stacks ensure all environments use the same component definitions (e.g., ALB, Auto Scaling), maintaining consistency.
 - Scalability:

- To create multiple environments, you deploy the root stack multiple times with different parameters (e.g., unique names for resources like ALB-Dev1, ALB-Dev2).
- This isn't explicitly addressed, but nested stacks make the template reusable---you can script the deployment of multiple stacks (e.g., via AWS CLI or SDK).
- Easy Updates:
 - CreateChangeSet and ExecuteChangeSet allow updating existing stacks:
 - Preview changes (e.g., adding a new DynamoDB table).
 - Apply changes with minimal downtime.
 - Nested stacks make updates modular---if the ALB configuration changes, you update alb-stack.yaml, and the root stack propagates the change.
- Networking Integration:
 - Fn::ImportValue in the Resources section of nested stacks is correct---it directly retrieves the VPC and subnet values where needed (e.g., for ALB subnets, Auto Scaling subnets).
- Does It Meet Requirements?:
 - Automate Creation: Yes, via CloudFormation.
 - Consistency: Yes, nested stacks ensure consistent components.
 - Scalability: Yes, deploy the root stack multiple times (requires scripting for full automation).
 - Easy Updates: Yes, via change sets.
 - Networking Integration: Yes, Fn::ImportValue correctly retrieves VPC/subnet values.
- Conclusion: This meets all requirements. Nested stacks provide modularity, Fn::ImportValue integrates with the networking stack, and change sets enable updates. Scalability requires deploying multiple stacks, but the approach is solid. C looks like the best choice.

Option D: Use Fn::ImportValue intrinsic functions in the Parameters section of the root template to retrieve Virtual Private Cloud (VPC) and subnet values. Define the development resources in the order they need to be created in the CloudFormation nested stacks. Use the CreateChangeSet and ExecuteChangeSet commands to update existing development environments

- What This Does:
 - Fn::ImportValue in Parameters:
 - Same as B---retrieves VPC/subnet values in the root template's Parameters section.
 - Nested Stacks:
 - Defines resources in nested stacks, similar to C.

- Order of Creation:
 - Specifies that resources in nested stacks are defined in the order they need to be created (e.g., security groups before ALB, ALB before Auto Scaling).
- CreateChangeSet and ExecuteChangeSet:
 - Same as C---use change sets for updates.
- Analysis:
 - Automation, Consistency, Updates:
 - Same as C---nested stacks, change sets, etc., meet these requirements.
 - Order of Creation:
 - CloudFormation automatically handles resource dependencies via DependsOn or intrinsic references (e.g., !Ref ALBSecurityGroup in the Auto Scaling group).
 - Explicitly ordering resources in the template isn't necessary---CloudFormation resolves dependencies regardless of order.
 - This part of the option is redundant and doesn't add value.
 - Fn::ImportValue in Parameters:
 - Works, but requires passing parameters to nested stacks (e.g., Parameters: VPCId: !ImportValue VPCId in the root, then passed to child stacks).
 - This is a stylistic difference compared to C (which uses Fn::ImportValue directly in nested stack resources). C is more direct---nested stacks fetch values where needed, reducing parameter passing.
 - Does It Meet Requirements?:
 - Same as C---meets all requirements.
 - C vs. D:
 - Both are correct, but:
 - C is more modular---nested stacks directly fetch VPC/subnet values, minimizing parameter passing.
 - D's "order of creation" is unnecessary since CloudFormation handles dependencies.
- Conclusion: This works but is slightly less elegant than C due to parameter passing and the redundant ordering note. D is a close second but not the best.

Analysis: Is C the Best Choice?

Your pick, C, uses nested stacks, Fn::ImportValue in nested stack resources, and change sets for updates:

- Does It Meet Requirements?:

- Automate Creation: Yes, CloudFormation automates environment creation.
- Consistency: Yes, nested stacks ensure consistent components.
- Scalability: Yes, deploy the root stack multiple times (requires scripting for full automation).
- Easy Updates: Yes, change sets allow updates.
- Networking Integration: Yes, Fn::ImportValue retrieves VPC/subnet values.
- Advantages:
 - Nested stacks provide modularity---each component (ALB, Auto Scaling) is self-contained, making the template maintainable as infrastructure grows.
 - Fn::ImportValue in nested stack resources is direct---each stack fetches what it needs, reducing parameter passing overhead.
 - Change sets enable safe updates to existing environments.
- Scalability:
 - To create multiple environments, deploy the root stack multiple times with unique parameters (e.g., EnvironmentName: Dev1, Dev2).
 - This can be automated via scripting (e.g., AWS CLI loop), which isn't specified but is implied by the need to "keep up with demand."
- Comparison to Alternatives:
 - A: StackSets are overkill for a single account/region, and Count is misleading.
 - B: Incorrectly uses TemplateURL for the networking stack, breaking the solution.
 - D: Works but is less modular (parameter passing) and includes unnecessary ordering.

C is the best choice:

- It balances modularity (nested stacks), integration (Fn::ImportValue), and updatability (change sets).
- It's the most practical for creating and managing multiple development environments in a single account/region.

Why You Picked C

You chose C because:

- You recognized that nested stacks provide a modular way to define common components (ALB, Auto Scaling, etc.), ensuring consistency.
- You saw that Fn::ImportValue in the nested stack resources correctly retrieves VPC/subnet values from the networking team's stack.
- You understood that CreateChangeSet and ExecuteChangeSet allow updating existing environments, meeting the update requirement.

Your reasoning is spot-on! Nested stacks are a best practice for modular CloudFormation templates, and C provides the most elegant, maintainable solution for this scenario. You picked the best approach!

Correct Answer: C. Use nested stacks to define common infrastructure components. Use `Fn::ImportValue` intrinsic functions with the resources of the nested stack to retrieve Virtual Private Cloud (VPC) and subnet values. Use the `CreateChangeSet` and `ExecuteChangeSet` commands to update existing development environments.

A development team uses AWS CodeCommit for version control for applications. The development team uses AWS CodePipeline, AWS CodeBuild, and AWS CodeDeploy for CI/CD infrastructure. In CodeCommit, the development team recently merged pull requests that did not pass long-running tests in the code base. The development team needed to perform rollbacks to branches in the codebase, resulting in lost time and wasted effort. A DevOps engineer must automate testing of pull requests in CodeCommit to ensure that reviewers more easily see the results of automated tests as part of the pull request review.

What should the DevOps engineer do to meet this requirement?

- A. Create an Amazon EventBridge rule that reacts to the `pullRequestStatusChanged` event. Create an AWS Lambda function that invokes a CodePipeline pipeline with a CodeBuild action that runs the tests for the application. Program the Lambda function to post the CodeBuild badge as a comment on the pull request so that developers will see the badge in their code review.
- B. Create an Amazon EventBridge rule that reacts to the `pullRequestCreated` event. Create an AWS Lambda function that invokes a CodePipeline pipeline with a CodeBuild action that runs the tests for the application. Program the Lambda function to post the CodeBuild test results as a comment on the pull request when the test results are complete.
- C. Create an Amazon EventBridge rule that reacts to `pullRequestCreated` and `pullRequestSourceBranchUpdated` events. Create an AWS Lambda function that invokes a CodePipeline pipeline with a CodeBuild action that runs the tests for the application. Program the Lambda function to post the CodeBuild badge as a comment on the pull request so that developers will see the badge in their code review.
- D. Create an Amazon EventBridge rule that reacts to the `pullRequestStatusChanged` event. Create an AWS Lambda function that invokes a CodePipeline pipeline with a CodeBuild action that runs the tests for the application. Program the Lambda function to post the CodeBuild test results as a comment on the pull request when the test results are complete.

Understanding the Problem

- Setup:
 - AWS CodeCommit: Used for version control, where developers create pull requests (PRs) to merge code changes.
 - CI/CD Pipeline: Uses CodePipeline, CodeBuild, and CodeDeploy for building, testing, and deploying applications.
 - Issue: Pull requests were merged without passing long-running tests, leading to rollbacks and wasted effort.

- Requirements:
 - Automate Testing: Run tests automatically when a pull request is created or updated.
 - Visibility for Reviewers: Show test results in the pull request review process (e.g., as a comment or badge).
 - Prevent Bad Merges: Ensure tests are run before merging, reducing the need for rollbacks.
- Key Considerations:
 - CodeCommit emits events (e.g., via Amazon EventBridge) when pull requests are created or updated.
 - We need to trigger a testing pipeline (e.g., using CodePipeline and CodeBuild) based on these events.
 - Test results must be posted back to the pull request in CodeCommit for reviewers to see.
 - The solution should handle both new pull requests (pullRequestCreated) and updates to existing pull requests (pullRequestSourceBranchUpdated).

Let's evaluate each option to find the best solution.

Option A: Create an Amazon EventBridge rule that reacts to the pullRequestStatusChanged event. Create an AWS Lambda function that invokes a CodePipeline pipeline with a CodeBuild action that runs the tests for the application. Program the Lambda function to post the CodeBuild badge as a comment on the pull request so that developers will see the badge in their code review

- What This Does:
 - EventBridge Rule:
 - Triggers on the pullRequestStatusChanged event from CodeCommit.
 - This event occurs when the pull request's status changes (e.g., from OPEN to CLOSED or MERGED).
 - Lambda Function:
 - Invokes a CodePipeline pipeline with a CodeBuild action to run tests.
 - Posts a CodeBuild badge (e.g., a visual indicator of pass/fail status) as a comment on the pull request.
 - CodePipeline/CodeBuild:
 - Runs the long-running tests for the application.
- Analysis:
 - Event Trigger:

- pullRequestStatusChanged fires when the PR status changes (e.g., when it's merged or closed).
- This is too late for the requirement---tests need to run before the merge, so reviewers can see the results during the review process. If the PR is already merged (status changed to CLOSED), running tests after the fact doesn't prevent bad merges.
- Test Results Visibility:
 - Posting a CodeBuild badge as a comment is a good way to show results in the PR, but since the trigger is after the status change, reviewers won't see it in time.
- Preventing Bad Merges:
 - This doesn't help---tests run after the status changes, so bad merges (like the ones that caused rollbacks) still happen.
- Does It Meet Requirements?:
 - Automate Testing: Yes, but at the wrong time.
 - Visibility for Reviewers: No, results are posted too late (after status change).
- Conclusion: The pullRequestStatusChanged trigger is incorrect---it runs tests after the PR status changes, not during the review process. A is incorrect.

Option B: Create an Amazon EventBridge rule that reacts to the pullRequestCreated event. Create an AWS Lambda function that invokes a CodePipeline pipeline with a CodeBuild action that runs the tests for the application. Program the Lambda function to post the CodeBuild test results as a comment on the pull request when the test results are complete

- What This Does:
 - EventBridge Rule:
 - Triggers on the pullRequestCreated event from CodeCommit.
 - This event fires when a new pull request is created.
 - Lambda Function:
 - Invokes a CodePipeline pipeline with a CodeBuild action to run tests.
 - Posts the CodeBuild test results (e.g., pass/fail details) as a comment on the pull request once tests are complete.
 - CodePipeline/CodeBuild:
 - Runs the tests for the application.
- Analysis:
 - Event Trigger:

- pullRequestCreated fires when a PR is created, which is a good time to run tests---reviewers can see the results before merging.
- Test Results Visibility:
 - Posting test results as a comment (e.g., "Tests passed: 100/100" or "Tests failed: 2/100, see details...") directly in the PR makes them visible during the review process.
- Preventing Bad Merges:
 - Tests run when the PR is created, so reviewers see the results before merging, reducing the chance of bad merges.
- Limitation:
 - pullRequestCreated only triggers when the PR is created. If the developer updates the source branch (e.g., pushes new commits to fix issues), this event doesn't fire again.
 - Without re-running tests on updates, reviewers might approve a PR based on outdated test results, allowing bad merges (e.g., new commits break the tests).
- Does It Meet Requirements?:
 - Automate Testing: Yes, for new PRs.
 - Visibility for Reviewers: Yes, results are posted as a comment.
 - Preventing Bad Merges: Partially---covers initial PR creation but misses updates.
- Conclusion: This is a good start but incomplete---it misses pull request updates, which are critical for ensuring tests are always current. B is not the best choice.

Option C: Create an Amazon EventBridge rule that reacts to pullRequestCreated and pullRequestSourceBranchUpdated events. Create an AWS Lambda function that invokes a CodePipeline pipeline with a CodeBuild action that runs the tests for the application. Program the Lambda function to post the CodeBuild badge as a comment on the pull request so that developers will see the badge in their code review

Your pick! Let's break this down:

- What This Does:
 - EventBridge Rule:
 - Triggers on two events:
 - pullRequestCreated: Fires when a new pull request is created.
 - pullRequestSourceBranchUpdated: Fires when the source branch of an existing pull request is updated (e.g., new commits are pushed).
 - Example rule pattern:

json

```
{
  "source": ["aws.codecommit"],
  "detail-type": ["CodeCommit Pull Request State Change"],
  "detail": {
    "event": ["pullRequestCreated",
"pullRequestSourceBranchUpdated"]
  }
}
```

- Lambda Function:
 - Invokes a CodePipeline pipeline with a CodeBuild action to run tests.
 - Posts a CodeBuild badge (e.g., a pass/fail indicator) as a comment on the pull request.
- CodePipeline/CodeBuild:
 - Runs the tests for the application.
- Analysis:
 - Event Triggers:
 - pullRequestCreated: Runs tests when a PR is created, ensuring initial test results are available.
 - pullRequestSourceBranchUpdated: Runs tests whenever the source branch is updated (e.g., new commits), ensuring test results stay current.
 - Together, these cover the full lifecycle of a pull request---creation and updates---ensuring tests are always run before merging.
 - Test Results Visibility:
 - Posting a CodeBuild badge as a comment makes the test status visible in the PR review process (e.g., a green "Tests Passed" badge or red "Tests Failed" badge).
 - Reviewers see the badge directly in CodeCommit's PR interface, making it easy to check test status.
 - Preventing Bad Merges:
 - Tests run on both PR creation and updates, ensuring reviewers always have up-to-date test results before merging.
 - This directly addresses the issue of bad merges causing rollbacks---reviewers can't approve a PR without seeing the latest test status.
 - Does It Meet Requirements?:
 - Automate Testing: Yes, tests run automatically on PR creation and updates.


- Visibility for Reviewers: Yes, the badge is posted as a comment in the PR.
- Preventing Bad Merges: Yes, tests are always current, reducing the risk of bad merges.
- Conclusion: This is a comprehensive solution---it triggers tests at the right times (creation and updates) and ensures visibility for reviewers via a badge. C looks like the best choice.

Option D: Create an Amazon EventBridge rule that reacts to the pullRequestStatusChanged event. Create an AWS Lambda function that invokes a CodePipeline pipeline with a CodeBuild action that runs the tests for the application. Program the Lambda function to post the CodeBuild test results as a comment on the pull request when the test results are complete

- What This Does:
 - EventBridge Rule:
 - Triggers on the pullRequestStatusChanged event (same as A).
 - Lambda Function:
 - Invokes a CodePipeline pipeline with a CodeBuild action to run tests.
 - Posts the CodeBuild test results (e.g., detailed pass/fail output) as a comment on the pull request.
 - CodePipeline/CodeBuild:
 - Runs the tests for the application.
- Analysis:
 - Event Trigger:
 - Same as A---pullRequestStatusChanged triggers after the PR status changes (e.g., merged or closed).
 - This is too late for reviewers to see the results during the review process.
 - Test Results Visibility:
 - Posting detailed test results is good, but since the trigger is after the status change, it doesn't help reviewers during the approval process.
 - Preventing Bad Merges:
 - Tests run after the fact, so bad merges still happen (as in the original issue).
 - Does It Meet Requirements?:
 - Automate Testing: Yes, but at the wrong time.
 - Visibility for Reviewers: No, results are posted too late.
- Conclusion: Same issue as A---the pullRequestStatusChanged trigger is incorrect for this use case. D is incorrect.

Analysis: Is C the Best Choice?

Your pick, C, uses EventBridge rules for `pullRequestCreated` and `pullRequestSourceBranchUpdated`, with a Lambda to trigger CodePipeline/CodeBuild and post a badge:

- Does It Meet Requirements?:
 - Automate Testing: Yes, tests run on PR creation and updates.
 - Visibility for Reviewers: Yes, CodeBuild badge is posted as a comment in the PR.
 - Preventing Bad Merges: Yes, tests run before merging, ensuring reviewers see the latest results.
- Completeness:
 - Triggering on both `pullRequestCreated` and `pullRequestSourceBranchUpdated` ensures tests are run at all relevant points in the PR lifecycle.
 - This prevents the issue of outdated test results leading to bad merges.
- Visibility:
 - The CodeBuild badge provides a clear, visual indicator of test status (e.g., pass/fail), which is ideal for reviewers.
 - A badge is often a URL or markdown (e.g.,  Build Status) that shows up in the PR comments, making it easy to see.
- Comparison to Alternatives:
 - A and D: Use `pullRequestStatusChanged`, which triggers too late (after merging), failing to help reviewers during the process.
 - B: Uses `pullRequestCreated`, which is good but misses updates to the source branch, risking outdated test results.

C is the best choice:

- It covers the full PR lifecycle (creation and updates).
- It ensures test results are visible to reviewers before merging, preventing bad merges.
- It integrates CodeCommit events with CodePipeline/CodeBuild, a standard AWS CI/CD pattern.

Why You Picked C

You chose C because:

- You recognized that tests need to run when a pull request is created (`pullRequestCreated`) and when it's updated (`pullRequestSourceBranchUpdated`) to ensure results are always current.
- You saw that posting a CodeBuild badge as a comment makes the test status visible to reviewers in the PR interface.

- You understood that this approach prevents bad merges by ensuring tests are run and visible before approval.

Your reasoning is spot-on! Triggering on both PR creation and updates is critical for a robust solution, and the CodeBuild badge ensures reviewers can easily see the test status. C is the best solution for this scenario, directly addressing the team's issue with failed merges.

Correct Answer: C. Create an Amazon EventBridge rule that reacts to pullRequestCreated and pullRequestSourceBranchUpdated events. Create an AWS Lambda function that invokes a CodePipeline pipeline with a CodeBuild action that runs the tests for the application. Program the Lambda function to post the CodeBuild badge as a comment on the pull request so that developers will see the badge in their code review.

****A DevOps engineer has automated a web service deployment by using AWS CodePipeline with the following steps:**

1. An AWS CodeBuild project compiles the deployment artifact and runs unit tests.
2. An AWS CodeDeploy deployment group deploys the web service to Amazon EC2 instances in the staging environment.
3. A CodeDeploy deployment group deploys the web service to EC2 instances in the production environment. The quality assurance (QA) team requests permission to inspect the build artifact before the deployment to the production environment occurs. The QA team wants to run an internal penetration testing tool to conduct manual tests. The tool will be invoked by a REST API call.**

Which combination of actions should the DevOps engineer take to fulfill this request? (Choose two.)

A. Insert a manual approval action between the test actions and deployment actions of the pipeline. B. Modify the buildspec.yml file for the compilation stage to require manual approval before completion. C. Update the CodeDeploy deployment groups so that they require manual approval to proceed. D. Update the pipeline to directly call the REST API for the penetration testing tool. E. Update the pipeline to invoke an AWS Lambda function that calls the REST API for the penetration testing tool.

Understanding the Problem

- Setup:
 - AWS CodePipeline:
 - Step 1: CodeBuild compiles the deployment artifact and runs unit tests.
 - Step 2: CodeDeploy deploys to the staging environment (EC2 instances).
 - Step 3: CodeDeploy deploys to the production environment (EC2 instances).
 - Deployment Artifact:
 - The artifact (e.g., compiled code, binaries) is generated by CodeBuild in Step 1, passed to CodeDeploy in Step 2 (staging), and then to Step 3 (production).
 - QA Team Request:

- Inspect Build Artifact: QA wants to review the artifact before production deployment.
- Penetration Testing Tool: QA wants to run a tool (via REST API) to conduct manual tests.
- Requirements:
 - Pause for Inspection: Allow QA to inspect the artifact after staging but before production deployment.
 - Run Penetration Tests: Invoke the penetration testing tool via a REST API call as part of the process.
 - Manual Testing: The penetration tests are manual, meaning QA will initiate the tests themselves after inspecting the artifact.
- Key Considerations:
 - CodePipeline supports manual approval actions to pause the pipeline for human review.
 - The penetration testing tool's REST API call can be automated (e.g., via a pipeline action), but since the tests are manual, QA likely needs to trigger the tool themselves after approval.
 - We need to balance automation (invoking the tool) with manual control (QA inspection and testing).
 - The solution must fit into the pipeline's flow: CodeBuild → Staging → [QA Step] → Production.

Since this is a "choose two" question, we need to identify two actions that together meet both aspects of the request (inspection and penetration testing).

Option A: Insert a manual approval action between the test actions and deployment actions of the pipeline

Your first pick! Let's break this down:

- What This Does:
 - Manual Approval Action:
 - Adds a manual approval step in CodePipeline between the staging deployment (Step 2) and production deployment (Step 3).
 - Example pipeline structure after modification:
 1. CodeBuild: Compile and test (produces artifact).
 2. CodeDeploy: Deploy to staging.
 3. Manual Approval: QA reviews and approves.
 4. CodeDeploy: Deploy to production.
 - How It Works:
 - After the staging deployment succeeds, the pipeline pauses at the manual approval action.

- QA receives a notification (e.g., via Amazon SNS, if configured) to review the deployment.
- QA can inspect the build artifact (stored in S3 by CodePipeline) and the staging environment (running the artifact).
- QA approves or rejects the action in the CodePipeline console.
- Analysis:
 - Inspection Requirement:
 - The manual approval action allows QA to pause the pipeline and inspect the build artifact before production deployment.
 - CodePipeline stores the artifact in S3 (e.g., in the pipeline's artifact bucket), and QA can access it via the console or S3 URL provided in the approval notification.
 - QA can also test the staging environment (e.g., via the ALB URL) to verify the deployment.
 - Penetration Testing:
 - This action doesn't directly invoke the penetration testing tool, but it gives QA the opportunity to run manual tests during the approval window.
 - QA can invoke the REST API for the penetration testing tool themselves (e.g., via a curl command or a custom UI) while the pipeline is paused.
 - Does It Meet Requirements?:
 - Inspect Build Artifact: Yes, the pause allows QA to review the artifact and staging deployment.
 - Penetration Testing: Partially---it enables QA to run the tests manually, but doesn't automate the REST API call.
 - Pipeline Flow:
 - This fits perfectly between staging and production, ensuring QA has time to review before the production deployment proceeds.
- Conclusion: This addresses the inspection requirement and provides a window for QA to run manual penetration tests. It's a key part of the solution but needs a second action to handle the REST API call for penetration testing. A is a good choice.

Option B: Modify the buildspec.yml file for the compilation stage to require manual approval before completion

- What This Does:
 - buildspec.yml:
 - The buildspec.yml file defines the build phase in CodeBuild (Step 1), including commands for compilation, testing, and artifact generation.

- Manual Approval in buildspec.yml:
 - Suggests adding a step in buildspec.yml to pause for manual approval.
 - However, CodeBuild doesn't natively support manual approval actions within the build process---you can't pause a build for human input.
- Analysis:
 - Feasibility:
 - CodeBuild is a fully automated build service. It doesn't support pausing for manual approval within the buildspec.yml.
 - You could simulate a pause by invoking an external process (e.g., a Lambda that triggers a manual approval in CodePipeline), but that's not what this option describes---it suggests the approval is inside the build stage, which isn't possible.
 - Inspection Requirement:
 - If this were feasible, it would pause before the build artifact is created (since it's in the compilation stage), meaning QA couldn't inspect the artifact (it doesn't exist yet).
 - QA needs to inspect the artifact after staging (Step 2), not during the build (Step 1).
 - Penetration Testing:
 - This doesn't address the penetration testing tool at all.
 - Does It Meet Requirements?:
 - Inspect Build Artifact: No, QA can't inspect the artifact if the build is paused, and the timing is wrong (before staging).
 - Penetration Testing: No, doesn't involve the REST API.
 - Pipeline Flow:
 - This would pause at the wrong point in the pipeline (during build, not between staging and production).
- Conclusion: This is not feasible---CodeBuild doesn't support manual approval in buildspec.yml, and even if it did, it's the wrong stage for QA inspection. B is incorrect.

Option C: Update the CodeDeploy deployment groups so that they require manual approval to proceed

- What This Does:
 - CodeDeploy Deployment Groups:
 - The staging deployment group (Step 2) deploys to staging EC2 instances.
 - The production deployment group (Step 3) deploys to production EC2 instances.
 - Manual Approval in CodeDeploy:

- Suggests adding a manual approval step within the CodeDeploy deployment process (e.g., in the deployment group settings).
- Analysis:
 - Feasibility:
 - CodeDeploy doesn't natively support manual approvals within a deployment group. Manual approvals are a CodePipeline feature, not a CodeDeploy feature.
 - CodeDeploy has deployment lifecycle events (e.g., BeforeInstall, AfterInstall), but these don't include pausing for human approval.
 - To pause, you'd need to modify the pipeline (e.g., add a manual approval action in CodePipeline, as in A), not the deployment group.
 - Inspection Requirement:
 - If this were possible, it could pause the production deployment (Step 3) to allow QA inspection.
 - However, since it's not supported, it doesn't work.
 - Penetration Testing:
 - Doesn't address the REST API call for the penetration testing tool.
 - Does It Meet Requirements?:
 - Inspect Build Artifact: No, CodeDeploy doesn't support this.
 - Penetration Testing: No, doesn't involve the REST API.
 - Pipeline Flow:
 - The intent (pausing before production) is correct, but CodeDeploy isn't the right place for this---CodePipeline is.
- Conclusion: This misunderstands CodeDeploy's capabilities---manual approvals belong in CodePipeline, not CodeDeploy. C is incorrect.

Option D: Update the pipeline to directly call the REST API for the penetration testing tool

- What This Does:
 - Direct REST API Call:
 - Adds a pipeline action to call the REST API of the penetration testing tool directly.
 - This could be done using a CodePipeline action (e.g., a Lambda action or a webhook action, but CodePipeline doesn't natively support direct HTTP calls).
- Analysis:
 - Penetration Testing:

- This automates the REST API call to invoke the penetration testing tool, which aligns with the requirement.
- However, the question specifies that the tests are manual---QA wants to conduct the tests themselves, meaning the tool should be invoked by QA, not automatically by the pipeline.
- Inspection Requirement:
 - This doesn't provide a pause for QA to inspect the artifact---it immediately runs the penetration tests, bypassing QA's manual inspection.
- Feasibility:
 - CodePipeline doesn't have a native "HTTP call" action. You'd need a Lambda function to make the REST API call (as in E), which this option doesn't specify.
- Does It Meet Requirements?:
 - Inspect Build Artifact: No, there's no pause for QA to inspect.
 - Penetration Testing: Yes, it invokes the REST API, but contradicts the "manual tests" requirement.
- Pipeline Flow:
 - This would add an automated step (e.g., after staging), but QA wants to manually run the tests after inspection, so this isn't the right approach.
- Conclusion: This automates the penetration testing tool, but QA wants to run manual tests, and there's no pause for inspection. D is incorrect.

Option E: Update the pipeline to invoke an AWS Lambda function that calls the REST API for the penetration testing tool

Your second pick! Let's break this down:

- What This Does:
 - Lambda Function:
 - Adds a pipeline action to invoke a Lambda function.
 - The Lambda makes the REST API call to the penetration testing tool.
 - Example Lambda code (simplified):

python

```
import json
import requests

def lambda_handler(event, context):
```

```
response = requests.post("https://pentest-  
tool.example.com/api/run", json={"artifact":  
"s3://bucket/artifact.zip"})  
return {  
    'statusCode': 200,  
    'body': json.dumps(response.json())  
}
```

- Pipeline Integration:
 - Add a Lambda action between staging (Step 2) and production (Step 3):
 1. CodeBuild: Compile and test.
 2. CodeDeploy: Deploy to staging.
 3. Lambda: Call penetration testing tool API.
 4. CodeDeploy: Deploy to production.
- Analysis:
 - Penetration Testing:
 - This automates the REST API call to invoke the penetration testing tool, which technically meets the requirement to "run the tool."
 - However, the question states the penetration tests are manual---QA wants to conduct the tests themselves, meaning they need to trigger the tool after inspecting the artifact.
 - Automating the API call in the pipeline bypasses QA's manual process, which isn't what they requested.
 - Inspection Requirement:
 - This doesn't provide a pause for QA to inspect the artifact---it immediately invokes the tool after staging.
 - Pipeline Flow:
 - This adds an automated step after staging, but QA needs a manual pause to inspect and then run the tests themselves.
 - Does It Meet Requirements?:
 - Inspect Build Artifact: No, there's no pause for inspection.
 - Penetration Testing: Yes, it invokes the REST API, but contradicts the "manual tests" requirement.
- Conclusion: This automates the penetration testing tool, but QA wants to run the tests manually after inspecting the artifact. It also misses the inspection requirement. E is not the best choice on its own.

Analysis: Are A and E the Best Combination?

Your picks:

- A: Insert a manual approval action between test actions and deployment actions.
- E: Update the pipeline to invoke a Lambda function that calls the REST API for the penetration testing tool.

Evaluation:

- A:
 - Correctly adds a manual approval action between staging (Step 2) and production (Step 3).
 - This allows QA to inspect the build artifact (in S3) and the staging environment before approving the production deployment.
 - It also provides a window for QA to manually run the penetration testing tool (e.g., by invoking the REST API themselves during the approval process).
- E:
 - Automates the REST API call via a Lambda function, which invokes the penetration testing tool.
 - However, QA wants to conduct manual tests---they should trigger the tool themselves after inspection, not have the pipeline do it automatically.

QA's Request:

- Inspection: QA needs to pause the pipeline to inspect the artifact and staging environment.
- Penetration Testing: QA wants to run the tool manually via a REST API call, meaning they need to invoke it themselves during the inspection window.

Why A and E Don't Fully Fit:

- A meets the inspection requirement by pausing the pipeline, allowing QA to review the artifact and manually run the penetration tests.
- E automates the penetration testing tool, which contradicts the "manual tests" requirement---QA wants to control the test execution.

Correct Combination:

- A is necessary---it provides the manual approval step for QA to inspect the artifact and run tests.
- E is incorrect because it automates the REST API call, which QA doesn't want (they want to run the tests manually).
- We need a second action to support the penetration testing, but none of the options provide a way to automate the REST API call in a manual context (e.g., a step that provides a UI for QA to trigger the API).

Best Interpretation:

- A alone is sufficient for the core requirement:
 - It pauses the pipeline for QA to inspect the artifact.
 - QA can manually invoke the REST API for the penetration testing tool during this pause (e.g., via a curl command, Postman, or a custom UI).
- The "choose two" aspect is tricky---none of the other options directly support manual invocation of the REST API by QA:
 - B and C are incorrect (CodeBuild and CodeDeploy don't support manual approvals).
 - D and E automate the API call, which doesn't align with "manual tests."

Alternative Interpretation:

- If we stretch the requirement and assume the REST API call can be automated (e.g., the tool runs tests and QA reviews the results manually), then A and E could work:
 - A: Pause for QA to inspect the artifact.
 - E: Automate the REST API call to run the tool, and QA reviews the results during the approval window.
- However, the question explicitly says "manual tests," meaning QA wants to initiate the tests themselves, not have the pipeline do it.

Correct Combination:

- The best approach is A (manual approval) as the primary action, allowing QA to inspect and manually run the tests.
- Since "choose two" is required, and no option directly supports manual invocation, the second action is less clear. The closest fit might be E if we interpret it as providing a mechanism for QA to trigger the API manually (e.g., Lambda could be a placeholder for a custom action), but that's a stretch.

Given the strict interpretation of "manual tests," A is the only fully correct action, but for "choose two," let's consider if another option could complement it. None do perfectly, so A paired with a non-invasive action (e.g., E reinterpreted) is the closest fit.

Final Answer: A is correct, but E doesn't align with "manual tests." Since no second action perfectly fits, A alone meets the core requirement, and the "choose two" might be a question design oversight. However, if forced to pick two, A and E are the closest, assuming E could be adjusted for manual invocation (not specified).

Correct Combination (Best Fit): A and E (with the caveat that E should support manual invocation, which it doesn't as written).

- A. Insert a manual approval action between the test actions and deployment actions of the pipeline.

- E: Update the pipeline to invoke an AWS Lambda function that calls the REST API for the penetration testing tool. (Interpreted as a mechanism QA could use manually, though not ideal.)

Why You Picked A and E

- A: You recognized that a manual approval action in CodePipeline allows QA to pause and inspect the build artifact before production deployment, which is exactly what they need.
- E: You likely thought the REST API call for the penetration testing tool should be part of the pipeline, automating the invocation of the tool.

Your reasoning for A is spot-on---it's the core action needed for QA to inspect the artifact and run manual tests. For E, you were thinking about integrating the penetration testing tool, but the "manual tests" requirement means QA wants to trigger the tool themselves, not have the pipeline do it automatically. This is a subtle distinction, but now you see the nuance!

A company is hosting a web application in an AWS Region. For disaster recovery purposes, a second region is being used as a standby. Disaster recovery requirements state that session data must be replicated between regions in near-real time and 1% of requests should route to the secondary region to continuously verify system functionality. Additionally, if there is a disruption in service in the main region, traffic should be automatically routed to the secondary region, and the secondary region must be able to scale up to handle all traffic.

How should a DevOps engineer meet these requirements?

- A. In both regions, deploy the application on AWS Elastic Beanstalk and use Amazon DynamoDB global tables for session data. Use an Amazon Route 53 weighted routing policy with health checks to distribute the traffic across the regions.
- B. In both regions, launch the application in Auto Scaling groups and use DynamoDB for session data. Use a Route 53 failover routing policy with health checks to distribute the traffic across the regions.
- C. In both regions, deploy the application in AWS Lambda, exposed by Amazon API Gateway, and use Amazon RDS for PostgreSQL with cross-region replication for session data. Deploy the web application with client-side logic to call the API Gateway directly.
- D. In both regions, launch the application in Auto Scaling groups and use DynamoDB global tables for session data. Enable an Amazon CloudFront weighted distribution across regions. Point the Amazon Route 53 DNS record at the CloudFront distribution.

Understanding the Problem

- Setup:
 - Primary Region: Hosts the web application.
 - Secondary Region: Acts as a standby for disaster recovery.
- Requirements:
 - Session Data Replication: Replicate session data between regions in near-real time (e.g., within seconds).

- Verify Secondary Region: Route 1% of traffic to the secondary region to continuously verify functionality.
- Automatic Failover: If the primary region fails, automatically route all traffic to the secondary region.
- Scalability: The secondary region must scale to handle 100% of traffic during failover.
- Key Considerations:
 - Session Data: Needs a multi-region data store with near-real-time replication (e.g., DynamoDB global tables).
 - Traffic Routing:
 - Route 1% to the secondary region for verification (e.g., Route 53 weighted routing).
 - Automatic failover requires health checks to detect primary region failure (e.g., Route 53 health checks).
 - Scalability: The secondary region's infrastructure must auto-scale (e.g., using Auto Scaling groups or serverless).
 - Disaster recovery often favors a pilot light or warm standby approach---the secondary region is active but minimally loaded (1% traffic) until failover.

Let's evaluate each option.

Option A: In both regions, deploy the application on AWS Elastic Beanstalk and use Amazon DynamoDB global tables for session data. Use an Amazon Route 53 weighted routing policy with health checks to distribute the traffic across the regions

Your pick! Let's break this down:

- What This Does:
 - AWS Elastic Beanstalk:
 - Deploy the application in both regions using Elastic Beanstalk, a platform-as-a-service (PaaS) that manages EC2 instances, Auto Scaling groups, Application Load Balancers (ALBs), and monitoring.
 - In the primary region, Beanstalk runs the full workload.
 - In the secondary region, Beanstalk runs a minimal setup (e.g., scaled down to handle 1% traffic) but can scale up during failover.
 - DynamoDB Global Tables:
 - Use DynamoDB global tables to replicate session data across regions with near-real-time consistency (typically seconds).
 - Route 53 Weighted Routing Policy with Health Checks:

- Weighted routing distributes traffic based on weights:
 - Primary region: 99% (weight 99).
 - Secondary region: 1% (weight 1).
 - Health checks monitor the health of each region (e.g., via the ALB's health endpoint).
 - If the primary region fails its health check, Route 53 stops routing traffic to it, sending 100% to the secondary region.
- Analysis:
 - Session Data Replication:
 - DynamoDB global tables replicate data across regions in near-real time (typically 1-2 seconds), meeting the requirement.
 - Eventual consistency is acceptable for session data (e.g., a user might briefly see an outdated session state during replication).
 - Verify Secondary Region (1% Traffic):
 - Route 53 weighted routing with weights of 99 (primary) and 1 (secondary) routes 1% of traffic to the secondary region, continuously verifying functionality.
 - Automatic Failover:
 - Health checks ensure that if the primary region fails (e.g., ALB fails health check), Route 53 adjusts routing:
 - Primary weight becomes 0 (failed health check).
 - Secondary weight becomes 1 (all traffic goes to the secondary region).
 - This provides automatic failover, meeting the requirement.
 - Scalability:
 - Elastic Beanstalk uses Auto Scaling groups under the hood---it automatically scales the secondary region's instances to handle 100% traffic during failover.
 - Beanstalk adjusts scaling policies dynamically (e.g., based on CPU, requests), ensuring the secondary region can handle the full load.
 - Disaster Recovery:
 - The secondary region is a warm standby---it's active (handling 1% traffic) and can scale up during failover, aligning with disaster recovery best practices.
 - Does It Meet Requirements?:
 - Session Data Replication: Yes, DynamoDB global tables (near-real time).
 - 1% Traffic to Secondary: Yes, Route 53 weighted routing (1% weight).

- Automatic Failover: Yes, health checks ensure failover.
- Scalability: Yes, Elastic Beanstalk auto-scales.
- Conclusion: This solution meets all requirements. Elastic Beanstalk simplifies management, DynamoDB global tables handle session data, and Route 53 weighted routing with health checks ensures traffic distribution, failover, and verification. A looks like the best choice.

Option B: In both regions, launch the application in Auto Scaling groups and use DynamoDB for session data. Use a Route 53 failover routing policy with health checks to distribute the traffic across the regions

- What This Does:
 - Auto Scaling Groups:
 - Deploy the application in two regions using EC2 Auto Scaling groups (likely with ALBs).
 - Auto Scaling ensures the application scales and replaces failed instances.
 - DynamoDB for Session Data:
 - Use DynamoDB, but it doesn't specify global tables.
 - Route 53 Failover Routing Policy with Health Checks:
 - Failover routing designates a primary region (100% traffic) and a secondary region (0% traffic unless failover).
 - Health checks monitor the primary region; if it fails, traffic switches to the secondary region.
- Analysis:
 - Session Data Replication:
 - DynamoDB without "global tables" implies a single-region table, which doesn't replicate data in near-real time.
 - Let's assume it's meant to be global tables (common in such questions), making this comparable to A.
 - Verify Secondary Region (1% Traffic):
 - Failover routing sends 100% of traffic to the primary region and 0% to the secondary region unless the primary fails.
 - This does not route 1% of traffic to the secondary region, failing the requirement to continuously verify functionality.
 - Automatic Failover:
 - Failover routing with health checks ensures that if the primary region fails, traffic switches to the secondary region, meeting this requirement.

- Scalability:
 - Auto Scaling groups in the secondary region can scale to handle 100% traffic during failover, meeting this requirement.
- Disaster Recovery:
 - The secondary region acts as a pilot light---it's running but not handling traffic (0%) until failover. This doesn't verify functionality (no traffic until failure).
- Does It Meet Requirements?:
 - Session Data Replication: Yes, if we assume global tables.
 - 1% Traffic to Secondary: No, failover routing sends 0% traffic to the secondary region.
 - Automatic Failover: Yes, failover routing with health checks.
 - Scalability: Yes, Auto Scaling groups.
- Conclusion: This fails the 1% traffic requirement---failover routing doesn't distribute traffic; it's all-or-nothing. B is incorrect.

Option C: In both regions, deploy the application in AWS Lambda, exposed by Amazon API Gateway, and use Amazon RDS for PostgreSQL with cross-region replication for session data. Deploy the web application with client-side logic to call the API Gateway directly

- What This Does:
 - AWS Lambda and API Gateway:
 - Deploy the application as a serverless Lambda function in two regions, exposed via API Gateway.
 - The web application uses client-side logic (e.g., JavaScript) to call API Gateway directly.
 - RDS for PostgreSQL with Cross-Region Replication:
 - Use RDS PostgreSQL with a read replica in the secondary region for session data.
 - Traffic Distribution:
 - Not specified---client-side logic must handle region selection (e.g., via AWS SDK or Route 53).
- Analysis:
 - Session Data Replication:
 - RDS cross-region replication creates a read replica in the secondary region, but:
 - Replication lag can be seconds to minutes, which doesn't meet the "near-real-time" requirement (DynamoDB global tables replicate in 1-2 seconds).

- Session data might be inconsistent during failover (e.g., a user logs in the primary region, but the secondary region doesn't see the session yet).
- Verify Secondary Region (1% Traffic):
 - No traffic distribution mechanism is specified.
 - Client-side logic could split traffic (e.g., 1% of requests to the secondary region API Gateway), but this isn't automated and requires complex application logic (e.g., handling retries, failover).
- Automatic Failover:
 - Without Route 53, there's no automatic failover---client-side logic must detect primary region failure and switch to the secondary, which isn't reliable or automated.
- Scalability:
 - Lambda scales automatically to handle 100% traffic, meeting this requirement.
- Disaster Recovery:
 - The secondary region is active (Lambda running), but without traffic distribution or failover, it doesn't verify functionality or switch automatically.
- Does It Meet Requirements?:
 - Session Data Replication: No, RDS cross-region replication isn't near-real-time (lag can be minutes).
 - 1% Traffic to Secondary: No, no routing policy specified.
 - Automatic Failover: No, client-side logic isn't automatic.
 - Scalability: Yes, Lambda scales.
- Conclusion: This fails multiple requirements---RDS replication isn't near-real-time, and there's no traffic distribution or automatic failover. C is incorrect.

Option D: In both regions, launch the application in Auto Scaling groups and use DynamoDB global tables for session data. Enable an Amazon CloudFront weighted distribution across regions. Point the Amazon Route 53 DNS record at the CloudFront distribution

- What This Does:
 - Auto Scaling Groups:
 - Deploy the application in two regions using Auto Scaling groups (likely with ALBs).
 - DynamoDB Global Tables:
 - Use DynamoDB global tables for session data, replicating across regions.
 - CloudFront Weighted Distribution:

- CloudFront is a CDN for caching static content.
- "Weighted distribution" isn't a CloudFront feature---CloudFront uses origin groups for failover or geo-routing, but let's assume this means Route 53 weighted routing with CloudFront as the entry point.
- Route 53 DNS Record:
 - Points to the CloudFront distribution, which routes to regional ALBs.
- Analysis:
 - Session Data Replication:
 - DynamoDB global tables meet the near-real-time requirement (1-2 seconds replication).
 - Verify Secondary Region (1% Traffic):
 - CloudFront doesn't support "weighted distribution"---it can use Route 53 weighted routing to distribute traffic to origins.
 - If interpreted as Route 53 weighted routing (weights 99/1) with CloudFront, it could route 1% traffic to the secondary region.
 - However, CloudFront is problematic:
 - CloudFront is designed for caching static content (e.g., HTML, CSS), not dynamic, session-based traffic.
 - Caching session data (e.g., cookies, tokens) can break the application (e.g., serving stale session responses).
 - Dynamic apps typically bypass CloudFront for session-based requests, using it only for static assets.
 - Automatic Failover:
 - CloudFront with Route 53 can use health checks to failover if the primary region fails, but caching issues might disrupt session consistency.
 - Scalability:
 - Auto Scaling groups scale to handle 100% traffic during failover.
 - Disaster Recovery:
 - The secondary region handles 1% traffic (if interpreted as Route 53 weighted routing), but CloudFront introduces complexity for a dynamic app.
 - Does It Meet Requirements?:
 - Session Data Replication: Yes, DynamoDB global tables.
 - 1% Traffic to Secondary: Yes, if interpreted as Route 53 weighted routing.

- Automatic Failover: Yes, with health checks.
 - Scalability: Yes, Auto Scaling groups.
 - Suitability: No, CloudFront isn't ideal for dynamic, session-based apps.
- Conclusion: CloudFront is a poor fit for a session-based web app---it's better for static content. A direct Route 53 weighted routing policy (A) avoids these issues. D is not the best choice.

Analysis: Is A the Best Choice?

Your pick, A, uses Elastic Beanstalk, DynamoDB global tables, and Route 53 weighted routing with health checks:

- Does It Meet Requirements?:
 - Session Data Replication: Yes, DynamoDB global tables replicate data in near-real time (1-2 seconds).
 - 1% Traffic to Secondary: Yes, Route 53 weighted routing (e.g., weight 99 for primary, 1 for secondary) sends 1% traffic to the secondary region.
 - Automatic Failover: Yes, health checks ensure traffic shifts to the secondary region if the primary fails.
 - Scalability: Yes, Elastic Beanstalk's Auto Scaling groups scale the secondary region to handle 100% traffic.
- Disaster Recovery:
 - The secondary region is a warm standby---it handles 1% traffic, verifying functionality, and scales during failover.
 - This aligns perfectly with the disaster recovery requirements.
- Comparison to Alternatives:
 - B: Failover routing sends 0% traffic to the secondary region (no verification) until failover, failing the 1% requirement.
 - C: RDS cross-region replication isn't near-real-time, and there's no traffic distribution or failover mechanism.
 - D: CloudFront is inappropriate for a dynamic, session-based app, and the "weighted distribution" terminology is incorrect.

A is the best choice:

- It meets all requirements with a managed, scalable architecture.
- DynamoDB global tables ensure near-real-time session replication.
- Route 53 weighted routing with health checks provides the 1% traffic split, automatic failover, and verification.

- Elastic Beanstalk simplifies management, and its Auto Scaling ensures the secondary region can handle full traffic.

Why You Picked A

You chose A because:

- You recognized that DynamoDB global tables meet the near-real-time session replication requirement.
- You saw that Route 53 weighted routing with health checks can send 1% traffic to the secondary region while enabling automatic failover.
- You understood that Elastic Beanstalk's Auto Scaling ensures the secondary region can scale to handle all traffic during failover.
- You appreciated the managed nature of Elastic Beanstalk, aligning with DevOps best practices for disaster recovery.

Your reasoning is spot-on! A is the best solution for this scenario---it perfectly balances disaster recovery, session management, traffic routing, and scalability. You nailed the requirements with a practical, AWS-recommended architecture.

Correct Answer: A. In both regions, deploy the application on AWS Elastic Beanstalk and use Amazon DynamoDB global tables for session data. Use an Amazon Route 53 weighted routing policy with health checks to distribute the traffic across the regions.

A development team wants to use AWS CloudFormation stacks to deploy an application. However, the developer IAM role does not have the required permissions to provision the resources that are specified in the AWS CloudFormation template. A DevOps engineer needs to implement a solution that allows the developers to deploy the stacks. The solution must follow the principle of least privilege.

Which solution will meet these requirements?

A. Create an IAM policy that allows the developers to provision the required resources. Attach the policy to the developer IAM role. B. Create an IAM policy that allows full access to AWS CloudFormation. Attach the policy to the developer IAM role. C. Create an AWS CloudFormation service role that has the required permissions. Grant the developer IAM role a cloudformation:* action. Use the new service role during stack deployments. D. Create an AWS CloudFormation service role that has the required permissions. Grant the developer IAM role the iam:PassRole permission. Use the new service role during stack deployments.

Understanding the Problem

- Setup:
 - AWS CloudFormation Stacks: Developers want to deploy an application using CloudFormation templates.
 - Developer IAM Role: Lacks the permissions to provision the resources defined in the template (e.g., EC2 instances, S3 buckets, DynamoDB tables).

- Issue:
 - When developers run `aws cloudformation create-stack` (or `update/delete`), the operation fails because their IAM role doesn't have permissions for the underlying resources (e.g., `ec2:RunInstances`, `s3:CreateBucket`).
- Requirements:
 - Enable Deployment: Allow developers to deploy CloudFormation stacks.
 - Principle of Least Privilege: Grant only the minimal permissions necessary for the task, avoiding overly broad access.
- Key Considerations:
 - CloudFormation needs permissions to provision resources on behalf of the user:
 - When a user deploys a stack, CloudFormation assumes the user's IAM role to create resources.
 - If the user's role lacks permissions (e.g., `ec2:RunInstances`), the operation fails.
 - Options to solve this:
 - Grant the developer role direct permissions to provision resources.
 - Use a CloudFormation service role to provision resources, limiting the developer role's permissions to CloudFormation actions only.
 - Least privilege prefers the most scoped-down permissions that still allow the task.

Let's evaluate each option.

Option A: Create an IAM policy that allows the developers to provision the required resources. Attach the policy to the developer IAM role

Your pick! Let's break this down:

- What This Does:
 - IAM Policy:
 - Create a policy that grants permissions for the specific resources in the CloudFormation template.
 - Example: If the template creates an EC2 instance, S3 bucket, and DynamoDB table, the policy might include:

json

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

```

    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances",
        "ec2:DescribeInstances",
        "ec2:CreateTags",
        "s3:CreateBucket",
        "s3:PutBucketPolicy",
        "dynamodb:CreateTable",
        "dynamodb:DescribeTable"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "cloudformation:*",
      "Resource": "*"
    }
  ]
}

```

- Attach to Developer Role:
 - Attach this policy to the developer IAM role, allowing the role to both manage CloudFormation stacks (cloudformation:*) and provision the underlying resources.
- Analysis:
 - Enable Deployment:
 - This allows developers to deploy the stack---CloudFormation uses the developer role's permissions to provision resources, and the policy grants the necessary actions (e.g., ec2:RunInstances, s3:CreateBucket).
 - Principle of Least Privilege:
 - The policy can be scoped to the exact actions and resources needed (e.g., specific EC2 instance types, specific S3 buckets), but the example above uses "Resource": "*", which isn't least privilege---it grants access to all resources of those types.
 - In practice, you'd need to:
 - Scope actions to specific resources (e.g., "Resource": "arn:aws:s3:::my-app-bucket").
 - Include only the actions required by the template (e.g., avoid granting ec2:DeleteInstance if the template only creates instances).
 - Without scoping, this grants more permissions than necessary, violating least privilege.
 - Operational Overhead:
 - The policy must be updated whenever the template changes (e.g., adding a new resource type like an RDS instance requires adding rds:CreateDBInstance to the policy).

- This can become cumbersome for complex templates or evolving applications, as the developer role needs permissions for every resource type in every stack.
- Security Risks:
 - Developers can use these permissions outside CloudFormation (e.g., manually run `aws ec2 run-instances`), potentially creating resources without oversight.
- Does It Meet Requirements?:
 - Enable Deployment: Yes, developers can deploy the stack.
 - Principle of Least Privilege: No, unless the policy is tightly scoped (not specified in the option).
- Conclusion: This works but doesn't fully adhere to least privilege unless the policy is meticulously scoped. It also introduces operational overhead and security risks by granting developers direct resource provisioning permissions. A is not the best choice.

Option B: Create an IAM policy that allows full access to AWS CloudFormation. Attach the policy to the developer IAM role

- What This Does:
 - IAM Policy:
 - Grants full access to CloudFormation:

json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "cloudformation:*",
      "Resource": "*"
    }
  ]
}
```

- Attach to Developer Role:
 - Attaches this policy to the developer IAM role.
- Analysis:
 - Enable Deployment:
 - This allows developers to manage CloudFormation stacks (`cloudformation:*`), but CloudFormation still needs permissions to provision the underlying resources.

- When CloudFormation creates resources, it uses the developer role's permissions---since this policy only grants cloudformation:*, it lacks permissions for resources (e.g., ec2:RunInstances), so the deployment fails.
- Principle of Least Privilege:
 - The policy grants full CloudFormation access, which is broader than necessary (e.g., developers can delete unrelated stacks, create stacks in other regions).
 - It doesn't address resource provisioning permissions, so it's incomplete.
- Does It Meet Requirements?:
 - Enable Deployment: No, deployment fails due to missing resource permissions.
 - Principle of Least Privilege: No, cloudformation:* is too broad.
- Conclusion: This doesn't enable deployment (lacks resource permissions) and violates least privilege by granting overly broad CloudFormation access. B is incorrect.

Option C: Create an AWS CloudFormation service role that has the required permissions. Grant the developer IAM role a cloudformation:* action. Use the new service role during stack deployments

- What This Does:
 - CloudFormation Service Role:
 - Create an IAM role (e.g., CloudFormationServiceRole) with permissions to provision the resources in the template:

json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances",
        "ec2:DescribeInstances",
        "ec2:CreateTags",
        "s3:CreateBucket",
        "s3:PutBucketPolicy",
        "dynamodb:CreateTable",
        "dynamodb:DescribeTable"
      ],
      "Resource": "*"
    }
  ]
}
```

- CloudFormation assumes this role to provision resources during stack operations.

- Developer Role Permissions:

- Grant the developer role `cloudformation:*`:

json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "cloudformation:*",
      "Resource": "*"
    }
  ]
}
```

- Use Service Role:

- When deploying the stack, specify the service role ARN (e.g., via `--role-arn` in the AWS CLI):

bash

```
aws cloudformation create-stack --stack-name MyStack --
template-body file://template.yaml --role-arn
arn:aws:iam::account-id:role/CloudFormationServiceRole
```

- Analysis:

- Enable Deployment:

- CloudFormation uses the service role to provision resources, so the deployment succeeds---the service role has the necessary permissions (e.g., `ec2:RunInstances`).

- Principle of Least Privilege:

- The developer role only needs `cloudformation:*` to manage stacks---it doesn't need direct resource permissions (e.g., `ec2:RunInstances`), reducing its privilege scope.
 - However, `cloudformation:*` is still broad---developers can perform any CloudFormation action (e.g., delete unrelated stacks).
 - The service role can be scoped to the exact resources and actions needed (e.g., specific S3 buckets), but the developer role's permissions are too permissive.
 - A key permission is missing: developers need `iam:PassRole` to pass the service role to CloudFormation (e.g., `iam:PassRole` for `CloudFormationServiceRole`).
 - Without `iam:PassRole`, the deployment fails when developers try to use the service role.

- Does It Meet Requirements?:
 - Enable Deployment: No, fails due to missing iam:PassRole permission.
 - Principle of Least Privilege: Partially---developer role is limited to CloudFormation actions, but cloudformation:* is too broad, and iam:PassRole is missing.
- Conclusion: This is close but incorrect---it misses the critical iam:PassRole permission, so developers can't deploy the stack. The cloudformation:* permission also violates least privilege. C is incorrect.

Option D: Create an AWS CloudFormation service role that has the required permissions. Grant the developer IAM role the iam:PassRole permission. Use the new service role during stack deployments

- What This Does:
 - CloudFormation Service Role:
 - Same as C---create a role (CloudFormationServiceRole) with permissions to provision the template's resources (e.g., ec2:RunInstances, s3:CreateBucket).
 - Developer Role Permissions:
 - Grant the developer role iam:PassRole for the service role:

json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": "arn:aws:iam::account-id:role/CloudFormationServiceRole"
    },
    {
      "Effect": "Allow",
      "Action": [
        "cloudformation:CreateStack",
        "cloudformation:UpdateStack",
        "cloudformation:DeleteStack",
        "cloudformation:DescribeStacks",
        "cloudformation:ListStacks"
      ],
      "Resource": "*"
    }
  ]
}
```

- The policy includes CloudFormation actions to manage stacks (not specified in the option, but implied).

- Use Service Role:
 - Same as C---specify the service role during stack deployment.
- Analysis:
 - Enable Deployment:
 - CloudFormation uses the service role to provision resources, ensuring the deployment succeeds.
 - iam:PassRole allows developers to pass the service role to CloudFormation, fixing the issue in C.
 - Principle of Least Privilege:
 - Developer role permissions:
 - iam:PassRole is scoped to the specific service role (CloudFormationServiceRole), adhering to least privilege.
 - CloudFormation actions (implied) can be scoped to specific actions (e.g., CreateStack, UpdateStack) and stacks (e.g., "Resource": "arn:aws:cloudformation:region:account-id:stack/MyStack/*"), ensuring minimal permissions.
 - The service role can be scoped to the exact resources and actions needed by the template, further adhering to least privilege.
 - Developers can't provision resources directly (e.g., no ec2:RunInstances), reducing risk---they can only manage CloudFormation stacks.
 - Operational Benefits:
 - The service role is managed by the DevOps engineer, centralizing resource provisioning permissions.
 - Developers only need permissions to manage stacks and pass the role, simplifying policy management.
 - If the template changes (e.g., adds an RDS instance), only the service role needs updating---not the developer role.
 - Does It Meet Requirements?:
 - Enable Deployment: Yes, the service role provisions resources, and iam:PassRole allows developers to deploy.
 - Principle of Least Privilege: Yes, developer permissions are minimal (iam:PassRole and scoped CloudFormation actions).
- Conclusion: This is the best solution---it enables deployment, follows least privilege, and reduces operational overhead. The service role approach is an AWS best practice for this scenario. D is the

correct choice.

Analysis: Is A the Best Choice?

Your pick, A, creates an IAM policy with resource permissions and attaches it to the developer role:

- Does It Meet Requirements?:
 - Enable Deployment: Yes, developers can deploy the stack with the added permissions.
 - Principle of Least Privilege: No, unless the policy is tightly scoped (not specified), and it grants developers direct resource access, increasing risk.
- Comparison to Alternatives:
 - B: Fails to enable deployment (lacks resource permissions) and violates least privilege (cloudformation:*)).
 - C: Fails due to missing iam:PassRole, and cloudformation:* is too broad.
 - D: Uses a service role, minimizing developer permissions to iam:PassRole and scoped CloudFormation actions, adhering to least privilege.

Why A Falls Short:

- Least Privilege: Granting developers direct permissions (e.g., ec2:RunInstances, s3:CreateBucket) allows them to provision resources outside CloudFormation, violating least privilege. The policy must be scoped to exact resources and actions, which isn't guaranteed in the option.
- Security: Developers can misuse permissions (e.g., manually create EC2 instances), increasing risk.
- Operational Overhead: The policy must be updated for every template change, making it harder to manage as the application evolves.

D is the best choice:

- It uses a CloudFormation service role to provision resources, reducing the developer role's permissions to iam:PassRole and CloudFormation actions.
- It adheres to least privilege by scoping permissions and centralizing resource provisioning in the service role.
- It's more maintainable---template changes only require updating the service role, not the developer role.

Why You Picked A

You chose A because:

- You recognized that the developer role needs permissions to provision the resources in the template (e.g., ec2:RunInstances).
- You thought attaching a policy with those permissions directly to the developer role would enable deployment.

Your reasoning is logical---A does enable deployment by granting the necessary permissions. However, it doesn't fully adhere to the principle of least privilege, as it gives developers direct resource access, which they can use outside CloudFormation. D is better because it uses a CloudFormation service role, limiting the developer role's permissions to iam:PassRole and CloudFormation actions, aligning with least privilege and reducing risk.

Correct Answer: D. Create an AWS CloudFormation service role that has the required permissions. Grant the developer IAM role the iam:PassRole permission. Use the new service role during stack deployments.

A production account has a requirement that any Amazon EC2 instance that has been logged in to manually must be terminated within 24 hours. All applications in the production account are using Auto Scaling groups with the Amazon CloudWatch Logs agent configured.

How can this process be automated?

A. Create a CloudWatch Logs subscription to an AWS Step Functions application. Configure an AWS Lambda function to add a tag to the EC2 instance that produced the login event and mark the instance to be decommissioned. Create an Amazon EventBridge rule to invoke a second Lambda function once a day that will terminate all instances with this tag. B. Create an Amazon CloudWatch alarm that will be invoked by the login event. Send the notification to an Amazon Simple Notification Service (Amazon SNS) topic that the operations team is subscribed to, and have them terminate the EC2 instance within 24 hours. C. Create an Amazon CloudWatch alarm that will be invoked by the login event. Configure the alarm to send to an Amazon Simple Queue Service (Amazon SQS) queue. Use a group of worker instances to process messages from the queue, which then schedules an Amazon EventBridge rule to be invoked. D. Create a CloudWatch Logs subscription to an AWS Lambda function. Configure the function to add a tag to the EC2 instance that produced the login event and mark the instance to be decommissioned. Create an Amazon EventBridge rule to invoke a daily Lambda function that terminates all instances with this tag.

Understanding the Problem

- Setup:
 - Production Account: Hosts applications using Auto Scaling groups.
 - CloudWatch Logs Agent: Configured on all EC2 instances, meaning system logs (e.g., /var/log/secure for Amazon Linux) are sent to CloudWatch Logs.
 - Manual Login Detection: Need to detect when someone logs into an EC2 instance (e.g., via SSH).
 - Termination Requirement: Terminate the instance within 24 hours of the login.
- Requirements:
 - Automate the Process: Detect manual logins, mark the instance, and terminate it within 24 hours.
 - Use Existing Setup: Leverage Auto Scaling groups and CloudWatch Logs.
- Key Considerations:
 - Detecting Logins:

- Manual logins (e.g., SSH) are logged in system logs like `/var/log/secure` (Amazon Linux) or `/var/log/auth.log` (Ubuntu).
- Example log entry for a successful SSH login:

```
Mar 01 12:00:01 ip-10-0-0-1 sshd[1234]: Accepted publickey  
for ec2-user from 203.0.113.1 port 54321 ssh2
```

- The CloudWatch Logs agent sends these logs to a CloudWatch Log group.
- Tagging and Termination:
 - Need to identify the instance that produced the login event, tag it, and terminate it within 24 hours.
- Auto Scaling:
 - Instances are in Auto Scaling groups, so termination must work with Auto Scaling (e.g., ASG will launch a replacement instance after termination).
- Automation:
 - The process must be fully automated---no manual intervention.

Let's evaluate each option to find the best solution.

Option A: Create a CloudWatch Logs subscription to an AWS Step Functions application. Configure an AWS Lambda function to add a tag to the EC2 instance that produced the login event and mark the instance to be decommissioned. Create an Amazon EventBridge rule to invoke a second Lambda function once a day that will terminate all instances with this tag

- What This Does:
 - CloudWatch Logs Subscription:
 - Create a subscription filter on the CloudWatch Log group containing EC2 instance logs (e.g., `/var/log/secure`).
 - Filter for login events (e.g., `Accepted publickey for ec2-user`) and stream matching events to a destination.
 - AWS Step Functions Application:
 - The subscription sends log events to AWS Step Functions, a workflow orchestration service.
 - Lambda Function (Tag Instance):
 - A Lambda function (invoked by Step Functions) processes the log event, identifies the EC2 instance that produced the login event, and adds a tag (e.g., `Decommission: true`).
 - EventBridge Rule:

- Schedules a second Lambda function to run daily (e.g., at 2 AM).
- Second Lambda Function (Terminate Instances):
 - Scans for EC2 instances with the Decommission tag and terminates them.
- Analysis:
 - Detecting Logins:
 - The CloudWatch Logs subscription filter can detect login events (e.g., by matching Accepted publickey in /var/log/secure).
 - The log event includes the instance's hostname (e.g., ip-10-0-0-1), which can be mapped to the instance ID using EC2 metadata (e.g., via `aws ec2 describe-instances`).
 - Tagging:
 - The first Lambda tags the instance (e.g., `Decommission: true`), marking it for termination.
 - Termination Within 24 Hours:
 - The second Lambda runs daily, terminating tagged instances.
 - If the login happens at 3 AM and the Lambda runs at 2 AM, the instance isn't terminated until the next day (up to 47 hours later), violating the 24-hour requirement.
 - Auto Scaling:
 - Terminating the instance via the second Lambda works with Auto Scaling---the ASG will launch a replacement instance.
 - Step Functions:
 - Step Functions is overkill for this use case---it's designed for complex workflows (e.g., with multiple steps, retries, branching).
 - A simple CloudWatch Logs subscription to Lambda (as in D) is sufficient---no need for orchestration.
 - Does It Meet Requirements?:
 - Automate the Process: Yes, the process is automated.
 - Within 24 Hours: No, a daily schedule could delay termination beyond 24 hours (e.g., up to 47 hours).
 - Use Existing Setup: Yes, leverages CloudWatch Logs and Auto Scaling.
- Conclusion: This solution automates the process but fails the 24-hour termination requirement due to the daily schedule. Step Functions adds unnecessary complexity. A is not the best choice.

Option B: Create an Amazon CloudWatch alarm that will be invoked by the login event. Send the notification to an Amazon Simple Notification Service (Amazon SNS) topic that the operations team is subscribed to, and

have them terminate the EC2 instance within 24 hours

- What This Does:
 - CloudWatch Alarm:
 - Create an alarm that triggers on a login event in CloudWatch Logs.
 - SNS Notification:
 - Send a notification to an SNS topic, alerting the operations team.
 - Manual Termination:
 - The operations team manually terminates the instance within 24 hours.
- Analysis:
 - Detecting Logins:
 - CloudWatch Alarms can be triggered by log events using a Metric Filter:
 - Create a Metric Filter on the log group to match login events (e.g., Accepted publickey).
 - Emit a metric (e.g., LoginEvents) when a match occurs.
 - Set an alarm on the metric (e.g., LoginEvents >= 1) to trigger the SNS notification.
 - Termination Within 24 Hours:
 - The operations team is notified and terminates the instance manually within 24 hours.
 - This meets the 24-hour requirement if the team acts promptly.
 - Automation:
 - The process isn't fully automated---it relies on human intervention (operations team terminating the instance).
 - The requirement asks for automation, which this fails to provide.
 - Does It Meet Requirements?:
 - Automate the Process: No, manual termination by the operations team.
 - Within 24 Hours: Yes, if the team acts within 24 hours.
 - Use Existing Setup: Yes, leverages CloudWatch Logs.
- Conclusion: This fails the automation requirement---manual termination isn't acceptable when the question explicitly asks for an automated process. B is incorrect.

Option C: Create an Amazon CloudWatch alarm that will be invoked by the login event. Configure the alarm to send to an Amazon Simple Queue Service (Amazon SQS) queue. Use a group of worker instances to

process messages from the queue, which then schedules an Amazon EventBridge rule to be invoked

- What This Does:
 - CloudWatch Alarm:
 - Same as B---use a Metric Filter to detect login events and trigger an alarm.
 - SQS Queue:
 - The alarm sends a message to an SQS queue when a login event is detected.
 - Worker Instances:
 - A group of EC2 instances (or other compute resources) poll the SQS queue, process messages, and schedule an EventBridge rule.
 - EventBridge Rule:
 - The rule is "scheduled" by the worker instances (presumably to terminate the instance).
- Analysis:
 - Detecting Logins:
 - The Metric Filter and alarm can detect login events, similar to B.
 - SQS Queue:
 - The alarm sends a message to SQS with details of the login event (e.g., instance ID).
 - Worker Instances:
 - Worker instances process SQS messages, but the option is vague about what they do:
 - Presumably, they extract the instance ID and "schedule" an EventBridge rule.
 - EventBridge Rule:
 - The phrasing "schedules an EventBridge rule to be invoked" is unclear:
 - EventBridge rules are event-driven or scheduled (e.g., cron-like).
 - "Scheduling" a rule might mean creating a one-time event (e.g., using EventBridge Scheduler to invoke a target after a delay), but this isn't standard terminology.
 - Let's assume the worker instances tag the instance (e.g., Decommission: true) and create a scheduled EventBridge rule to terminate it within 24 hours.
 - Termination Within 24 Hours:
 - If the worker schedules a termination event (e.g., 23 hours after the login), this could meet the 24-hour requirement.

- However, the setup is overly complex---worker instances, SQS, and dynamic EventBridge scheduling for each login event introduce unnecessary overhead.
- Automation:
 - This is automated, but the architecture is convoluted.
- Does It Meet Requirements?:
 - Automate the Process: Yes, it's automated.
 - Within 24 Hours: Yes, if the scheduling ensures termination within 24 hours.
 - Use Existing Setup: Yes, leverages CloudWatch Logs and Auto Scaling.
- Complexity:
 - This is far more complex than necessary:
 - Worker instances to process SQS messages add operational overhead (e.g., managing the instances, scaling).
 - Dynamic EventBridge rule scheduling for each login event is cumbersome (e.g., creating a one-time schedule per event).
 - A simpler solution (e.g., tag and terminate via Lambda) achieves the same result.
- Conclusion: This works but is overly complicated---worker instances and dynamic EventBridge scheduling add unnecessary complexity. A simpler solution is better. C is not the best choice.

Option D: Create a CloudWatch Logs subscription to an AWS Lambda function. Configure the function to add a tag to the EC2 instance that produced the login event and mark the instance to be decommissioned. Create an Amazon EventBridge rule to invoke a daily Lambda function that terminates all instances with this tag

Your pick! Let's break this down:

- What This Does:
 - CloudWatch Logs Subscription:
 - Create a subscription filter on the CloudWatch Log group (e.g., /var/log/secure) to match login events (e.g., Accepted publickey).
 - Stream matching log events to a Lambda function.
 - First Lambda Function (Tag Instance):
 - Processes the log event, extracts the instance ID (e.g., from the hostname ip-10-0-0-1 using `aws ec2 describe-instances`), and adds a tag (e.g., `Decommission: true`).
 - EventBridge Rule:
 - Schedules a second Lambda function to run daily (e.g., every 24 hours at 2 AM).

- Second Lambda Function (Terminate Instances):
 - Scans for EC2 instances with the Decommission tag and terminates them.
- Analysis:
 - Detecting Logins:
 - The CloudWatch Logs subscription filter detects login events in near real-time (seconds delay).
 - The first Lambda extracts the instance ID and tags it.
 - Tagging:
 - Tagging the instance (e.g., Decommission: true) marks it for termination.
 - Termination Within 24 Hours:
 - The second Lambda runs daily, terminating tagged instances.
 - Same issue as A: If a login happens at 3 AM and the Lambda runs at 2 AM, the instance isn't terminated until the next day (up to 47 hours later), violating the 24-hour requirement.
 - Auto Scaling:
 - Terminating the instance works with Auto Scaling---the ASG launches a replacement instance.
 - Automation:
 - The process is fully automated---log detection, tagging, and termination happen without manual intervention.
 - Does It Meet Requirements?:
 - Automate the Process: Yes, fully automated.
 - Within 24 Hours: No, a daily schedule could delay termination beyond 24 hours (up to 47 hours).
 - Use Existing Setup: Yes, leverages CloudWatch Logs and Auto Scaling.
- Conclusion: This is very close---it automates the process efficiently using CloudWatch Logs subscriptions and Lambda. However, the daily termination schedule fails the 24-hour requirement, as instances could persist longer than 24 hours. D is not the best choice as written.

Analysis: Is D the Best Choice?

Your pick, D, uses a CloudWatch Logs subscription to tag instances and a daily Lambda to terminate them:

- Does It Meet Requirements?:
 - Automate the Process: Yes, the process is automated.

- Within 24 Hours: No, the daily schedule could delay termination beyond 24 hours.
- Use Existing Setup: Yes, leverages CloudWatch Logs and Auto Scaling.
- Comparison to Alternatives:
 - A: Similar to D but uses Step Functions, which is overkill. It also fails the 24-hour requirement (daily schedule).
 - B: Fails automation due to manual termination.
 - C: Overly complex (SQS, worker instances, dynamic EventBridge scheduling) but could meet the 24-hour requirement if scheduling is precise.

Why D Falls Short:

- The daily termination schedule (via EventBridge) doesn't guarantee termination within 24 hours:
 - Worst case: A login at 3 AM, with the Lambda running at 2 AM, means the instance persists until 2 AM the next day (47 hours).
 - Best case: A login at 1 AM, with the Lambda at 2 AM, terminates the instance after 1 hour.
- The requirement is strict---"within 24 hours" means the maximum delay must be 24 hours.

Ideal Solution:

- D is the closest architecturally---it's simple and leverages AWS services effectively. However, to meet the 24-hour requirement:
 - The first Lambda could tag the instance with a timestamp (e.g., DecommissionAt: 2025-03-02T12:00:00Z, 24 hours after the login).
 - The second Lambda (running daily or more frequently, e.g., hourly) terminates instances where DecommissionAt is in the past.
- Alternatively:
 - Use EventBridge Scheduler (introduced in 2022) to schedule a one-time event 24 hours after the login:
 - The first Lambda tags the instance and creates a scheduled event to terminate it after 24 hours.
 - This ensures exact 24-hour termination but isn't an option here.

Best Available Choice:

- D is the closest but fails the 24-hour requirement due to the daily schedule.
- C could work if the worker instances schedule precise termination (e.g., 23 hours after login), but the architecture is overly complex and vague.
- None of the options perfectly meet the 24-hour requirement due to scheduling limitations:

- A daily schedule (A, D) can exceed 24 hours.
- C's dynamic scheduling is unclear but could theoretically work if interpreted as scheduling within 24 hours.

Given the options, D is the most practical architecture (simpler than A and C, automated unlike B), but it needs adjustment for the 24-hour requirement. Since the question doesn't offer a perfect solution, D is the best choice if we assume the daily schedule is frequent enough in practice (e.g., running every 12 hours to reduce the max delay to 35 hours, still not ideal).

Correct Answer (Best Fit): D. Create a CloudWatch Logs subscription to an AWS Lambda function. Configure the function to add a tag to the EC2 instance that produced the login event and mark the instance to be decommissioned. Create an Amazon EventBridge rule to invoke a daily Lambda function that terminates all instances with this tag.

Why You Picked D

You chose D because:

- You recognized that CloudWatch Logs subscriptions can detect login events in near real-time, leveraging the existing setup.
- You saw that tagging the instance and terminating it via a scheduled Lambda is a straightforward, automated process.
- You understood that this approach works with Auto Scaling groups---the ASG replaces terminated instances.

Your reasoning is strong---D is the most practical and automated solution among the options, and it aligns well with AWS best practices (e.g., CloudWatch Logs subscriptions, Lambda, EventBridge). The only issue is the daily schedule, which doesn't strictly meet the 24-hour requirement. In a real-world scenario, you'd adjust the schedule (e.g., run hourly) or use EventBridge Scheduler for precise timing, but among the given options, D is the closest fit.
