Amazon

(AWS Certified Developer - Associate DVA-C02)

AWS Certified Developer - Associate DVA-C02

Total: **551 Questions**

Link:

Question: 1

A company is implementing an application on Amazon EC2 instances. The application needs to process incoming transactions. When the application detects a transaction that is not valid, the application must send a chat message to the company's support team. To send the message, the application needs to retrieve the access token to authenticate by using the chat API.
A developer needs to implement a solution to store the access token. The access token must be encrypted at rest and in transit. The access token must also be accessible from other AWS accounts.
Which solution will meet these requirements with the LEAST management overhead?

A.Use an AWS Systems Manager Parameter Store SecureString parameter that uses an AWS Key Management Service (AWS KMS) AWS managed key to store the access token. Add a resource-based policy to the parameter to allow access from other accounts. Update the IAM role of the EC2 instances with permissions to access Parameter Store. Retrieve the token from Parameter Store with the decrypt flag enabled. Use the decrypted access token to send the message to the chat.

B.Encrypt the access token by using an AWS Key Management Service (AWS KMS) customer managed key. Store the access token in an Amazon DynamoDB table. Update the IAM role of the EC2 instances with permissions to access DynamoDB and AWS KMS. Retrieve the token from DynamoDDecrypt the token by using AWS KMS on the EC2 instances. Use the decrypted access token to send the message to the chat.

C.Use AWS Secrets Manager with an AWS Key Management Service (AWS KMS) customer managed key to store the access token. Add a resource-based policy to the secret to allow access from other accounts. Update the IAM role of the EC2 instances with permissions to access Secrets Manager. Retrieve the token from Secrets Manager. Use the decrypted access token to send the message to the chat.

D.Encrypt the access token by using an AWS Key Management Service (AWS KMS) AWS managed key. Store the access token in an Amazon S3 bucket. Add a bucket policy to the S3 bucket to allow access from other accounts. Update the IAM role of the EC2 instances with permissions to access Amazon S3 and AWS KMS. Retrieve the token from the S3 bucket. Decrypt the token by using AWS KMS on the EC2 instances. Use the decrypted access token to send the massage to the chat.

Answer: C

Explanation:

The best solution is **C. Use AWS Secrets Manager with an AWS Key Management Service (AWS KMS) customer managed key to store the access token.**

Here's a breakdown of why this is the most suitable option and why the others are less ideal:

AWS Secrets Manager Benefits: Secrets Manager is designed specifically for storing, rotating, and managing secrets like access tokens. It provides built-in encryption at rest using KMS and automatically handles retrieval and decryption. This reduces the developer's management overhead compared to other options.

KMS Customer Managed Key: Using a customer-managed KMS key gives the company more control over the encryption key lifecycle, including rotation and access policies. This is generally preferred for sensitive data.

Resource-Based Policy for Cross-Account Access: Secrets Manager allows you to attach a resource-based policy to a secret, granting access to other AWS accounts. This addresses the requirement of accessibility from other accounts.

IAM Role for EC2 Instances: Granting the EC2 instances an IAM role with permissions to access Secrets Manager enables them to retrieve the secrets securely.

Least Management Overhead: Secrets Manager handles the encryption, decryption, and rotation of the secret, minimizing the amount of code and infrastructure the developer needs to manage.

Why other options are not ideal:

A (Parameter Store): While Parameter Store SecureString can store encrypted data, Secrets Manager is better suited for secrets management as it offers features like automatic rotation. Parameter Store also has limits on size and throughput that Secrets Manager does not.

B (DynamoDB): Storing the encrypted token in DynamoDB requires manual encryption/decryption and handling of the KMS key, increasing management overhead. DynamoDB is also not designed specifically for secrets management.

D (S3): Storing the encrypted token in S3 requires manual encryption/decryption and handling of the KMS key. S3 is object storage rather than secrets management, increasing overhead and complexity.

In summary, Secrets Manager is the specialized service for storing and managing sensitive information. It provides built-in encryption, access control, and other features that simplify the process of storing and retrieving access tokens securely and efficiently.

Supporting Links:

AWS Secrets Manager:https://aws.amazon.com/secrets-manager/
AWS Key Management Service (KMS):https://aws.amazon.com/kms/
Resource-Based Policies:https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_identity-vs-resource.html

Question: 2

A company is running Amazon EC2 instances in multiple AWS accounts. A developer needs to implement an application that collects all the lifecycle events of the EC2 instances. The application needs to store the lifecycle events in a single Amazon Simple Queue Service (Amazon SQS) queue in the company's main AWS account for further processing.
Which solution will meet these requirements?

A.Configure Amazon EC2 to deliver the EC2 instance lifecycle events from all accounts to the Amazon EventBridge event bus of the main account. Add an EventBridge rule to the event bus of the main account that matches all EC2 instance lifecycle events. Add the SQS queue as a target of the rule.

B.Use the resource policies of the SQS queue in the main account to give each account permissions to write to that SQS queue. Add to the Amazon EventBridge event bus of each account an EventBridge rule that matches all EC2 instance lifecycle events. Add the SQS queue in the main account as a target of the rule.

C.Write an AWS Lambda function that scans through all EC2 instances in the company accounts to detect EC2 instance lifecycle changes. Configure the Lambda function to write a notification message to the SQS queue in the main account if the function detects an EC2 instance lifecycle change. Add an Amazon EventBridge scheduled rule that invokes the Lambda function every minute.

D.Configure the permissions on the main account event bus to receive events from all accounts. Create an Amazon EventBridge rule in each account to send all the EC2 instance lifecycle events to the main account event bus. Add an EventBridge rule to the main account event bus that matches all EC2 instance lifecycle events. Set the SQS queue as a target for the rule.

Answer: D

Explanation:

Here's a detailed justification for why option D is the best solution, along with supporting information and links:

Option D utilizes Amazon EventBridge, which is designed for event-driven architectures and cross-account event delivery, making it ideal for this scenario. First, it configures permissions on the main account's EventBridge event bus to receive events from other AWS accounts. This is a crucial step for allowing the central account to gather events from all the other accounts. Then, it configures each individual AWS account to send EC2 instance lifecycle events to the central account's event bus. This ensures that all relevant lifecycle events are captured and routed to the central account. Finally, within the main account's event bus, it creates a rule to match all incoming EC2 instance lifecycle events and targets the SQS queue. This effectively captures the events sent from other accounts and places them into the SQS queue for further processing.

Here's why the other options are less suitable:

Option A: While it centralizes events in the main account's EventBridge, it relies solely on EC2 to directly deliver events to the central EventBridge. This mechanism is less robust for cross-account scenarios compared to explicitly configuring event sending through EventBridge rules in each source account.

Option B: Directly granting cross-account permissions to the SQS queue can become a security management challenge as the number of accounts grows. Managing resource policies for direct write access to the SQS queue from multiple external accounts is less scalable and could introduce security risks. It also bypasses EventBridge's powerful routing and filtering capabilities.

Option C: This solution involves polling all EC2 instances using a Lambda function, which is inefficient, costly, and prone to missed events. This is because polling at a specific interval (e.g., every minute) might not capture all lifecycle events, especially short-lived ones. Event-driven approaches using EventBridge are significantly more effective for capturing real-time events.

Why Option D is best:

Option D leverages the intended use case of Amazon EventBridge for cross-account event delivery, which offers a decoupled, scalable, and reliable solution. By pushing events from individual accounts to the main account's EventBridge, it avoids direct access to the SQS queue, maintaining better security control. Furthermore, EventBridge allows for flexible filtering and routing, ensuring that only relevant EC2 lifecycle events are captured and delivered to the SQS queue. This is essential for clean architecture and efficient event processing.

Supporting Links:

Amazon EventBridge cross-account events:
https://docs.aws.amazon.com/eventbridge/latest/userguide/eventbridge-cross-account-event-delivery.html
Amazon SQS resource policies:
https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-writing-an-sqs-policy.html
Amazon EventBridge concepts:https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-what-is.html

Question: 3

An application is using Amazon Cognito user pools and identity pools for secure access. A developer wants to integrate the user-specific file upload and download features in the application with Amazon S3. The developer must ensure that the files are saved and retrieved in a secure manner and that users can access only their own files. The file sizes range from 3 KB to 300 MB.
Which option will meet these requirements with the HIGHEST level of security?

A.Use S3 Event Notifications to validate the file upload and download requests and update the user interface (UI).

B.Save the details of the uploaded files in a separate Amazon DynamoDB table. Filter the list of files in the user interface (UI) by comparing the current user ID with the user ID associated with the file in the table.

C.Use Amazon API Gateway and an AWS Lambda function to upload and download files. Validate each request in the Lambda function before performing the requested operation.

D.Use an IAM policy within the Amazon Cognito identity prefix to restrict users to use their own folders in Amazon S3.

Answer: D

Explanation:

Here's a detailed justification for why option D is the most secure and appropriate solution, along with supporting explanations and links:

Option D, "Use an IAM policy within the Amazon Cognito identity prefix to restrict users to use their own folders in Amazon S3," is the best approach because it directly leverages IAM to enforce granular, identity-based access control at the S3 bucket level. Cognito Identity Pools (Federated Identities) allow you to grant temporary AWS credentials to users, whether they authenticate through Cognito User Pools or other identity providers. By prefixing S3 object keys with a user's Cognito identity ID, you create a logical separation of user data within the bucket.

The IAM policy associated with the Cognito identity role can then be configured to allow users to only access objects within their specific prefix (folder). This approach implements the principle of least privilege, ensuring that users can only interact with their own data and not other users' data. This greatly reduces the risk of unauthorized access.

For instance, the IAM policy would grant permissions like s3:GetObject and s3:PutObject only for paths like arn:aws:s3:::your-bucket-name/$ cognito-identity.amazonaws.com:sub /*, where $ cognito-identity.amazonaws.com:sub resolves to the user's unique Cognito identity ID. This allows users to upload, download, and manage files within their designated folder, ensuring isolation and security.

The advantage of this solution is that access control is enforced directly at the AWS infrastructure level (IAM and S3). There's no need to rely on application-level checks (like options A, B, and C) that can be potentially bypassed or have vulnerabilities. It is also the most scalable and performant, since S3 natively handles access control.

Option A introduces complexity with S3 event notifications, which are more suitable for triggering workflows after uploads or downloads, not for primary access control. Option B requires maintaining a DynamoDB table, which adds overhead and introduces potential synchronization issues. It also relies on application logic for filtering, which is less secure than IAM. Option C, using API Gateway and Lambda, introduces unnecessary complexity and latency for simple file uploads and downloads. It also adds another layer where security vulnerabilities can occur, needing careful validation in the Lambda function. IAM policies directly integrated with S3 are far more efficient and secure.

Therefore, leveraging IAM policies with Cognito identity prefixes offers the highest level of security and is the most straightforward, scalable, and cost-effective solution. It directly integrates with AWS's identity and access management system, reducing the attack surface and simplifying access control.

Supporting Links:

Amazon Cognito Identity Pools:https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-identity.html
IAM Policies for Amazon S3:https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-access-control.html
IAM Policy Variables:https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_variables.html

Question: 4

A company is building a scalable data management solution by using AWS services to improve the speed and agility of development. The solution will ingest large volumes of data from various sources and will process this data through multiple business rules and transformations.
The solution requires business rules to run in sequence and to handle reprocessing of data if errors occur when the business rules run. The company needs the solution to be scalable and to require the least possible maintenance. Which AWS service should the company use to manage and automate the orchestration of the data flows to meet these requirements?

A.AWS Batch
B.AWS Step Functions

C.AWS Glue

D.AWS Lambda

Answer: B

Explanation:

The correct answer is **B. AWS Step Functions.** Here's why:

AWS Step Functions is designed specifically to orchestrate distributed applications and microservices using visual workflows. It allows you to define a series of steps (tasks) in a state machine definition that can be executed in a specific order. This directly addresses the requirement for business rules to run in sequence.

Step Functions includes built-in error handling and retry mechanisms, which are crucial for managing reprocessing of data when errors occur during business rule execution. You can define different retry policies and catch exceptions to handle failures gracefully, ensuring data processing continues even in the face of errors. This built-in fault tolerance reduces the operational overhead significantly.

The service is highly scalable. Step Functions can handle a large number of concurrent executions, making it suitable for processing large volumes of data ingested from various sources. The scalability is inherent in the service's architecture, relieving the company from managing the underlying infrastructure.

Step Functions is a fully managed service. AWS takes care of the underlying infrastructure and scaling. This greatly reduces the operational maintenance required, allowing the company to focus on developing and deploying business rules rather than managing infrastructure.

While AWS Batch (A) is designed for batch processing jobs, it does not inherently provide the orchestration and sequencing capabilities of Step Functions. AWS Glue (C) is primarily an ETL (Extract, Transform, Load) service focused on data cataloging and transformation, but lacks the robust orchestration and state management features for complex, sequenced business rules. AWS Lambda (D) can be used within Step Functions but by itself does not provide the workflow management capabilities.

Therefore, Step Functions directly addresses the requirements for sequential execution, error handling, scalability, and minimal maintenance, making it the most suitable service for the company's needs.

Supporting Links:

AWS Step Functions Documentation
Orchestrating Microservices with AWS Step Functions

Question: 5

A developer has created an AWS Lambda function that is written in Python. The Lambda function reads data from objects in Amazon S3 and writes data to an Amazon DynamoDB table. The function is successfully invoked from an S3 event notification when an object is created. However, the function fails when it attempts to write to the DynamoDB table.
What is the MOST likely cause of this issue?

A.The Lambda function's concurrency limit has been exceeded.

B.DynamoDB table requires a global secondary index (GSI) to support writes.

C.The Lambda function does not have IAM permissions to write to DynamoDB.

D.The DynamoDB table is not running in the same Availability Zone as the Lambda function.

Answer: C

Explanation:

The most likely reason the Lambda function is failing to write to the DynamoDB table, despite successfully being invoked by the S3 event notification, is due to insufficient IAM permissions. Lambda functions, by default, do not have permissions to access other AWS services like DynamoDB. Access to these services must be explicitly granted via an IAM role. Option A, the concurrency limit being exceeded, would likely manifest in throttling errors, but the question indicates the function fails to write, implying a permission issue rather than resource contention. Option B, needing a GSI, is irrelevant because the problem is an inability to write at all, not write efficiently or query effectively. GSIs are used for optimizing read queries based on attributes different from the primary key. Option D, the Availability Zone, is also incorrect. DynamoDB is a regional service and replicates data across multiple AZs within a region, so network latency or unavailability of a specific AZ would not typically prevent write operations completely. The Lambda function will be able to write to any DynamoDB table regardless of the AZ configuration as long as its configuration is on the same region as the DynamoDB. The error most likely is that the IAM role attached to the Lambda function doesn't have the necessary permissions to perform PutItem operations on the DynamoDB table. The IAM role will need permissions to dynamodb:PutItem on the specific DynamoDB table ARN or a wildcard to allow writing to all tables. To fix this, the developer must configure the Lambda function's execution role to include an IAM policy that grants the dynamodb:PutItem permission on the target DynamoDB table. This ensures the function has the authorization to perform write operations. Without this IAM policy, DynamoDB will deny the Lambda function's write requests, leading to the observed failure.

Relevant links:

AWS Lambda Permissions: https://docs.aws.amazon.com/lambda/latest/dg/lambda-intro-execution-role.html IAM Policies for DynamoDB: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/security-iam-id-based-policy-examples.html

Question: 6

A developer is creating an AWS CloudFormation template to deploy Amazon EC2 instances across multiple AWS accounts. The developer must choose the EC2 instances from a list of approved instance types.
How can the developer incorporate the list of approved instance types in the CloudFormation template?

A.Create a separate CloudFormation template for each EC2 instance type in the list.

B.In the Resources section of the CloudFormation template, create resources for each EC2 instance type in the list.

C.In the CloudFormation template, create a separate parameter for each EC2 instance type in the list.

D.In the CloudFormation template, create a parameter with the list of EC2 instance types as AllowedValues.

Answer: D

Explanation:

The most effective and manageable way for a developer to incorporate a list of approved EC2 instance types within a CloudFormation template is to utilize the AllowedValues property within a parameter declaration. This approach allows for centralized control and validation of user input.

Option D suggests creating a parameter within the CloudFormation template. This parameter would specifically define the list of approved EC2 instance types using the AllowedValues property. When a user deploys the CloudFormation stack, they are restricted to choosing only from the instance types listed in the AllowedValues. If the user attempts to provide a different instance type, CloudFormation will reject the input, preventing the creation of an unsupported EC2 instance. This method is clean, concise, and ensures that only approved instance types are launched. It promotes standardization and helps to avoid configuration errors.

Options A, B, and C are less ideal because they are either overly complex or do not provide the necessary validation. Creating separate CloudFormation templates for each instance type (A) would be cumbersome and difficult to maintain. Creating resources for each instance type (B) would also be inefficient and would not allow the user to choose an instance type. Creating separate parameters for each instance type (C) would be confusing for the user and would not provide a way to enforce a specific list of approved instance types.

Therefore, option D provides the most appropriate and efficient solution by providing a manageable way to enforce the list of approved EC2 instance types within the CloudFormation template. This method is consistent with the principles of infrastructure as code and helps ensure that the deployed infrastructure meets security and compliance requirements.

Further research can be conducted at the following links:

AWS CloudFormation Parameters:
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html AWS CloudFormation AllowedValues:
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html#parameters-section-structure-properties

Question: 7

A developer has an application that makes batch requests directly to Amazon DynamoDB by using the BatchGetItem low-level API operation. The responses frequently return values in the UnprocessedKeys element.
Which actions should the developer take to increase the resiliency of the application when the batch response includes values in UnprocessedKeys? (Choose two.)

A.Retry the batch operation immediately.

B.Retry the batch operation with exponential backoff and randomized delay.

C.Update the application to use an AWS software development kit (AWS SDK) to make the requests.

D.Increase the provisioned read capacity of the DynamoDB tables that the operation accesses.

E.Increase the provisioned write capacity of the DynamoDB tables that the operation accesses.

Answer: BD

Explanation:

The UnprocessedKeys element in a BatchGetItem response indicates that DynamoDB was unable to retrieve some items due to throttling or internal errors. To improve application resiliency, we need to address this issue effectively.

Option B is correct because retrying the batch operation with exponential backoff and randomized delay is a standard and best-practice approach for handling transient errors and throttling in distributed systems.
Exponential backoff gradually increases the wait time between retries, reducing the likelihood of overwhelming DynamoDB. The randomized delay further helps to prevent a "thundering herd" effect where multiple clients retry simultaneously, potentially exacerbating the throttling issue. This strategy adheres to principles of fault tolerance and retry mechanisms in cloud computing.
https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

Option D is also correct because UnprocessedKeys can indicate that the read capacity of the DynamoDB table is insufficient to handle the batch read request. Increasing the provisioned read capacity can reduce the frequency of throttling and thus the occurrence of UnprocessedKeys. It directly addresses the potential cause of the error by increasing the resources available to the application. This aligns with the concept of scaling resources to meet demand in cloud environments.
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html

Option A is incorrect because retrying immediately without any delay can actually worsen the situation, potentially leading to further throttling.

Option C is incorrect. While using the AWS SDK is generally recommended, it doesn't inherently solve the issue of UnprocessedKeys. The SDK provides helpful methods, including retry logic, but the application still needs to be configured to handle throttled requests appropriately using exponential backoff and random jitter.

Option E is incorrect because UnprocessedKeys in the context of BatchGetItem relate to read operations, not write operations. Increasing write capacity will not address the problem.

Question: 8

A company is running a custom application on a set of on-premises Linux servers that are accessed using Amazon API Gateway. AWS X-Ray tracing has been enabled on the API test stage.
How can a developer enable X-Ray tracing on the on-premises servers with the LEAST amount of configuration?

A.Install and run the X-Ray SDK on the on-premises servers to capture and relay the data to the X-Ray service.

B.Install and run the X-Ray daemon on the on-premises servers to capture and relay the data to the X-Ray service.

C.Capture incoming requests on-premises and configure an AWS Lambda function to pull, process, and relay relevant data to X-Ray using the PutTraceSegments API call.

D.Capture incoming requests on-premises and configure an AWS Lambda function to pull, process, and relay relevant data to X-Ray using the PutTelemetryRecords API call.

Answer: B

Explanation:

The correct answer is B because it provides the simplest and most efficient way to enable X-Ray tracing on on-premises servers with minimal configuration. The X-Ray daemon is specifically designed for this purpose. It listens for traffic on UDP port 2000, buffers segment documents, and uploads them to AWS X-Ray.

Here's a breakdown of why the other options are less suitable:

A. Install and run the X-Ray SDK on the on-premises servers to capture and relay the data to the X-Ray service: While the SDK is necessary to instrument the application code itself to create segments and subsegments, it doesn't automatically relay the data to X-Ray. The SDK typically requires integration with the X-Ray daemon or direct AWS API calls, increasing complexity.

C. Capture incoming requests on-premises and configure an AWS Lambda function to pull, process, and relay relevant data to X-Ray using the PutTraceSegments API call: This approach involves significant overhead. It requires capturing the necessary data, setting up and managing a Lambda function, and manually crafting the data into the format expected by the PutTraceSegments API. This increases complexity and management overhead substantially.

D. Capture incoming requests on-premises and configure an AWS Lambda function to pull, process, and relay relevant data to X-Ray using the PutTelemetryRecords API call: PutTelemetryRecords is used for metrics and operational data, not for tracing data like segments. It's not appropriate for sending X-Ray tracing information. Similar to option C, this method introduces significant overhead and complexity using Lambda.

The X-Ray daemon provides a streamlined, dedicated solution for collecting and forwarding trace data. By running the daemon on the on-premises servers, the application can send trace data to the daemon, which then efficiently batches and sends the data to X-Ray. This approach avoids the need for complex custom code or direct AWS API calls from the application itself.

The daemon automatically handles authentication and buffering, reducing the complexity of sending trace data to X-Ray. It's the recommended approach for offloading the task of sending trace data from your applications.

Authoritative Links:

AWS X-Ray Daemon: https://docs.aws.amazon.com/xray/latest/devguide/xray-daemon.html AWS X-Ray SDK: https://docs.aws.amazon.com/xray/latest/devguide/xray-sdk.html

Question: 9

A company wants to share information with a third party. The third party has an HTTP API endpoint that the company can use to share the information. The company has the required API key to access the HTTP API. The company needs a way to manage the API key by using code. The integration of the API key with the application code cannot affect application performance. Which solution will meet these requirements MOST securely?

A.Store the API credentials in AWS Secrets Manager. Retrieve the API credentials at runtime by using the AWS SDK. Use the credentials to make the API call.

B.Store the API credentials in a local code variable. Push the code to a secure Git repository. Use the local code variable at runtime to make the API call.

C.Store the API credentials as an object in a private Amazon S3 bucket. Restrict access to the S3 object by using IAM policies. Retrieve the API credentials at runtime by using the AWS SDK. Use the credentials to make the API call.

D.Store the API credentials in an Amazon DynamoDB table. Restrict access to the table by using resource-based policies. Retrieve the API credentials at runtime by using the AWS SDK. Use the credentials to make the API call.

Answer: A

Explanation:

The most secure solution is to store the API credentials in AWS Secrets Manager and retrieve them at runtime using the AWS SDK. Here's why:

Centralized Secret Management: AWS Secrets Manager is a dedicated service designed to securely store and manage secrets like API keys, passwords, and database credentials. This centralizes secret management, making it easier to control access and rotate secrets. https://aws.amazon.com/secrets-manager/
Encryption: Secrets Manager encrypts secrets both in transit and at rest. This provides a strong layer of protection against unauthorized access.

Access Control: Secrets Manager integrates with IAM, allowing you to define fine-grained access control policies specifying which users or services can access which secrets. This restricts access only to authorized entities.

Auditing: Secrets Manager integrates with AWS CloudTrail, providing an audit trail of all access and modification events. This helps in tracking and investigating any potential security breaches.

Rotation: Secrets Manager supports automatic rotation of secrets, which is a crucial security practice for minimizing the impact of compromised credentials.

On-Demand Retrieval: Retrieving secrets at runtime using the AWS SDK ensures that the API key is not hardcoded in the application code or configuration files.

Performance Impact: Retrieving secrets from Secrets Manager at runtime has minimal impact on application performance. The AWS SDK is optimized for efficient secret retrieval.

Options B, C, and D are less secure:

Option B: Storing the API key in a local code variable and pushing it to a Git repository exposes the secret to anyone with access to the repository. This is a severe security risk.

Option C: While storing the API key in an S3 bucket with restricted access is better than Option B, it's still not as secure as Secrets Manager. S3 is designed for object storage, not secret management. Secrets Manager offers additional features like encryption, rotation, and auditing that are specifically tailored for secret management.

Option D: Storing the API key in a DynamoDB table with resource-based policies is also less secure than Secrets Manager. Similar to S3, DynamoDB is a database service, not a secret management service, and lacks the specialized features of Secrets Manager.

Therefore, option A offers the most robust and secure solution for managing API keys. It leverages a dedicated secret management service, providing encryption, access control, auditing, and rotation capabilities. This minimizes the risk of compromised credentials and helps maintain the security of the company's data and systems.

Question: 10

A developer is deploying a new application to Amazon Elastic Container Service (Amazon ECS). The developer needs to securely store and retrieve different types of variables. These variables include authentication information for a remote API, the URL for the API, and credentials. The authentication information and API URL must be available to all current and future deployed versions of the application across development, testing, and production environments.
How should the developer retrieve the variables with the FEWEST application changes?

A.Update the application to retrieve the variables from AWS Systems Manager Parameter Store. Use unique paths in Parameter Store for each variable in each environment. Store the credentials in AWS Secrets Manager in each environment.

B.Update the application to retrieve the variables from AWS Key Management Service (AWS KMS). Store the API URL and credentials as unique keys for each environment.

C.Update the application to retrieve the variables from an encrypted file that is stored with the application. Store the API URL and credentials in unique files for each environment.

D.Update the application to retrieve the variables from each of the deployed environments. Define the authentication information and API URL in the ECS task definition as unique names during the deployment process.

Answer: A

Explanation:

The most appropriate solution is A: "Update the application to retrieve the variables from AWS Systems Manager Parameter Store. Use unique paths in Parameter Store for each variable in each environment. Store the credentials in AWS Secrets Manager in each environment."

Here's why:

AWS Systems Manager Parameter Store: Parameter Store provides centralized, encrypted storage for configuration data management. It allows storing parameters as key-value pairs and is ideal for storing the API URL and authentication information which are common across environments but might vary slightly (e.g., different URLs for dev, test, and prod). Using unique paths for each environment ensures segregation and prevents conflicts. This aligns with the requirement for making these variables available to all versions across environments.

AWS Secrets Manager: Secrets Manager is designed specifically for securely storing secrets like credentials (passwords, API keys, tokens, etc.). It offers features like automatic rotation, which enhances security and reduces the risk of compromised credentials. Storing credentials separately from other configuration data is a best practice.

Least Application Changes: This approach requires modifying the application to fetch values from Parameter Store and Secrets Manager. This is a manageable change and avoids hardcoding sensitive data within the application code or build artifacts. Option D, using ECS task definition, is more tedious for maintenance across multiple environments.

Security: Parameter Store supports encryption at rest using AWS KMS, and Secrets Manager offers automatic rotation and tighter access control, making option A the most secure approach.

Scalability and Maintainability: Using Parameter Store and Secrets Manager scales well with application growth and simplifies configuration management across multiple environments. Changes to variables can be made centrally without redeploying the application (after a short propagation delay).

Let's look at why the other options are not ideal:

B. AWS Key Management Service (AWS KMS): While KMS can encrypt data, it's primarily a key management service and less suitable for storing and retrieving configuration parameters or secrets directly. Using KMS directly would involve more complex encryption/decryption logic within the application.

C. Encrypted File Stored with the Application: This approach poses significant security risks. Storing secrets in a file, even if encrypted, makes them more vulnerable to compromise, especially if the application is compromised. It also requires managing encryption keys, adding complexity.

D. ECS Task Definition: While ECS task definitions can store environment variables, it is not appropriate for storing secrets like credentials. They are plainly visible in the AWS console and through the API and not intended for sensitive information. Also, managing these directly for multiple versions and environments would be cumbersome. This option also would need to be updated whenever we have to rotate the credentials, making it highly undesirable.

References:

AWS Systems Manager Parameter Store: https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-parameter-store.html
AWS Secrets Manager: https://aws.amazon.com/secrets-manager/

Question: 11

A company is migrating legacy internal applications to AWS. Leadership wants to rewrite the internal employee directory to use native AWS services. A developer needs to create a solution for storing employee contact details and high-resolution photos for use with the new application.
Which solution will enable the search and retrieval of each employee's individual details and high-resolution photos using AWS APIs?

A.Encode each employee's contact information and photos using Base64. Store the information in an Amazon DynamoDB table using a sort key.

B.Store each employee's contact information in an Amazon DynamoDB table along with the object keys for the photos stored in Amazon S3.

C.Use Amazon Cognito user pools to implement the employee directory in a fully managed software-as-a-service (SaaS) method.

D.Store employee contact information in an Amazon RDS DB instance with the photos stored in Amazon Elastic File System (Amazon EFS).

Answer: B

Explanation:

The correct solution involves leveraging the strengths of both Amazon DynamoDB and Amazon S3.

DynamoDB, a NoSQL database, is ideal for storing structured data like employee contact information due to its fast read and write performance and scalability. Amazon S3, on the other hand, is designed for storing objects, making it perfect for storing high-resolution photos.

Option B proposes storing employee contact information in DynamoDB and storing the photos in S3, with the DynamoDB entry including the S3 object key for each photo. This approach allows for efficient retrieval of both contact details and associated photos. The DynamoDB entry acts as a pointer to the S3 object, enabling applications to quickly locate and retrieve the photo using the S3 API. This architecture separates structured data (contact details) from unstructured data (photos), optimizing storage and retrieval for each data type.

Option A is less efficient because encoding photos into Base64 and storing them directly in DynamoDB bloats the database and negatively impacts performance. DynamoDB is not optimized for storing large binary files within table entries.

Option C, using Amazon Cognito user pools, is primarily for authentication and authorization, not for storing employee contact information and photos. While Cognito can manage user identities, it's not a suitable database solution for this scenario.

Option D, using Amazon RDS and Amazon EFS, is less cost-effective and more complex than using DynamoDB and S3. RDS is a relational database service, which is suitable for structured data, but storing employee data may not benefit significantly from relational features compared to DynamoDB's key-value structure. While EFS could store the photos, it introduces additional complexity for retrieval compared to S3's object storage model.

Therefore, Option B provides the most scalable, performant, and cost-effective solution for storing and retrieving both employee contact details and high-resolution photos using AWS APIs.

Relevant links:

Amazon DynamoDB:https://aws.amazon.com/dynamodb/
Amazon S3:https://aws.amazon.com/s3/

Question: 12

A developer is creating an application that will give users the ability to store photos from their cellphones in the cloud. The application needs to support tens of thousands of users. The application uses an Amazon API Gateway REST API that is integrated with AWS Lambda functions to process the photos. The application stores details about the photos in Amazon DynamoDB.
Users need to create an account to access the application. In the application, users must be able to upload photos and retrieve previously uploaded photos. The photos will range in size from 300 KB to 5 MB.
Which solution will meet these requirements with the LEAST operational overhead?

A.Use Amazon Cognito user pools to manage user accounts. Create an Amazon Cognito user pool authorizer in API Gateway to control access to the API. Use the Lambda function to store the photos and details in the DynamoDB table. Retrieve previously uploaded photos directly from the DynamoDB table.

B.Use Amazon Cognito user pools to manage user accounts. Create an Amazon Cognito user pool authorizer in API Gateway to control access to the API. Use the Lambda function to store the photos in Amazon S3. Store the object's S3 key as part of the photo details in the DynamoDB table. Retrieve previously uploaded photos by querying DynamoDB for the S3 key.

C.Create an IAM user for each user of the application during the sign-up process. Use IAM authentication to access the API Gateway API. Use the Lambda function to store the photos in Amazon S3. Store the object's S3 key as part of the photo details in the DynamoDB table. Retrieve previously uploaded photos by querying DynamoDB for the S3 key.

D.Create a users table in DynamoDB. Use the table to manage user accounts. Create a Lambda authorizer that validates user credentials against the users table. Integrate the Lambda authorizer with API Gateway to control access to the API. Use the Lambda function to store the photos in Amazon S3. Store the object's S3 key as par of the photo details in the DynamoDB table. Retrieve previously uploaded photos by querying DynamoDB for the S3 key.

Answer: B

Explanation:

Here's a detailed justification for why option B is the best solution, along with supporting concepts and links:

Option B provides the most efficient and scalable solution with the least operational overhead because it leverages managed AWS services specifically designed for user authentication, authorization, object storage, and data retrieval.

Amazon Cognito User Pools: Cognito handles user registration, authentication, and authorization. This removes the operational burden of managing user accounts, passwords, and security from the developer. It scales automatically to handle tens of thousands of users. https://aws.amazon.com/cognito/
Cognito User Pool Authorizer in API Gateway: This authorizer controls access to your API based on user identity managed by Cognito. API Gateway handles the authentication and authorization, relieving the Lambda functions from these tasks, which improves their performance.

https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-integrate-with-cognito.html **Storing Photos in Amazon S3:** S3 is designed for cost-effective and scalable object storage. Storing photos directly in S3 handles the large file sizes efficiently, offloading storage management from the Lambda function and avoiding potential Lambda timeout issues. https://aws.amazon.com/s3/
DynamoDB for Metadata: DynamoDB stores the metadata about the photos, including the S3 key. This provides a fast and efficient way to query and retrieve the S3 key for a specific photo based on user criteria.

https://aws.amazon.com/dynamodb/
Retrieving Photos by Querying DynamoDB for S3 Key: This approach allows the application to quickly identify the location of the photo in S3 and retrieve it. The Lambda function only needs to fetch the S3 object using the key instead of managing file storage directly.

Why other options are not optimal:

A: Storing photos directly in DynamoDB is not ideal for large binary data. DynamoDB is optimized for structured data. Storing large photos in DynamoDB can be costly and inefficient.

C: Creating IAM users for each application user is not scalable or manageable. IAM is designed for AWS administrators, not application end-users. This approach would create a large administrative burden.

D: Managing user accounts in a custom DynamoDB table requires developing and maintaining the authentication and authorization logic. This is less efficient than using Cognito and introduces security risks.

In summary, option B offers a combination of managed services designed for the specific requirements, resulting in the least operational overhead and best scalability. It delegates user management to Cognito, offloads storage to S3, and uses DynamoDB efficiently for metadata, making it the most suitable solution.

Question: 13

A company receives food orders from multiple partners. The company has a microservices application that uses Amazon API Gateway APIs with AWS Lambda integration. Each partner sends orders by calling a customized API that is exposed through API Gateway. The API call invokes a shared Lambda function to process the orders. Partners need to be notified after the Lambda function processes the orders. Each partner must receive updates for only the partner's own orders. The company wants to add new partners in the future with the fewest code changes possible.
Which solution will meet these requirements in the MOST scalable way?

A.Create a different Amazon Simple Notification Service (Amazon SNS) topic for each partner. Configure the Lambda function to publish messages for each partner to the partner's SNS topic.

B.Create a different Lambda function for each partner. Configure the Lambda function to notify each partner's service endpoint directly.

C.Create an Amazon Simple Notification Service (Amazon SNS) topic. Configure the Lambda function to publish messages with specific attributes to the SNS topic. Subscribe each partner to the SNS topic. Apply the appropriate filter policy to the topic subscriptions.

D.Create one Amazon Simple Notification Service (Amazon SNS) topic. Subscribe all partners to the SNS topic.

Answer: C

Explanation:

The most scalable solution is option C, which leverages a single SNS topic with message filtering based on attributes. Here's why:

Scalability: SNS is designed for high throughput and fan-out messaging. Using a single topic avoids managing and maintaining numerous SNS topics, simplifying infrastructure management and reducing the operational overhead as the number of partners grows.

Flexibility: Option C enables easy onboarding of new partners. The company only needs to add the new partner's subscription to the existing SNS topic with the appropriate filter policy. No code change is needed in the Lambda function.

Cost Efficiency: Managing one SNS topic is more cost-effective than managing multiple topics as it requires less resources.

Message Filtering: SNS filter policies allow subscribers (the partners) to receive only messages that match specific attributes. In this scenario, the Lambda function publishes messages with an attribute indicating the partner ID. Each partner subscribes to the SNS topic and uses a filter policy to only receive messages where the partner ID matches their own. This ensures that each partner only receives updates relevant to their orders.

Reduced Code Changes: This solution minimizes code changes in the Lambda function. It only needs to publish messages with the partner ID attribute. No changes are needed to the Lambda function when adding new partners, keeping the code maintainable.

Option A creates an SNS topic for each partner. This can become difficult to manage as the number of partners grows. It is less scalable.

Option B involves creating a different Lambda function for each partner, which duplicates code and increases maintenance overhead. Directly notifying each partner from the Lambda function complicates the logic and reduces flexibility. Also, Lambda functions are not primarily intended for direct notifications.

Option D doesn't provide message filtering, resulting in all partners receiving all order updates, violating the requirement that each partner only receive updates for their own orders.

In conclusion, option C offers the most scalable and manageable solution by leveraging SNS filter policies, minimizing code changes, and efficiently routing messages to the correct partners.

Supporting Links:

Amazon SNS Message Filtering
Amazon SNS FAQs

Question: 14

A financial company must store original customer records for 10 years for legal reasons. A complete record contains personally identifiable information (PII). According to local regulations, PII is available to only certain people in the company and must not be shared with third parties. The company needs to make the records available to third-party organizations for statistical analysis without sharing the PII.

A developer wants to store the original immutable record in Amazon S3. Depending on who accesses the S3 document, the document should be returned as is or with all the PII removed. The developer has written an AWS Lambda function to remove the PII from the document. The function is named removePii.
What should the developer do so that the company can meet the PII requirements while maintaining only one copy of the document?

A.Set up an S3 event notification that invokes the removePii function when an S3 GET request is made. Call Amazon S3 by using a GET request to access the object without PII.

B.Set up an S3 event notification that invokes the removePii function when an S3 PUT request is made. Call Amazon S3 by using a PUT request to access the object without PII.

C.Create an S3 Object Lambda access point from the S3 console. Select the removePii function. Use S3 Access Points to access the object without PII.

D.Create an S3 access point from the S3 console. Use the access point name to call the GetObjectLegalHold S3 API function. Pass in the removePii function name to access the object without PII.

Answer: C

Explanation:

The correct answer is **C. Create an S3 Object Lambda access point from the S3 console. Select the removePii function. Use S3 Access Points to access the object without PII.**

Here's why:

S3 Object Lambda allows you to add your own code to S3 GET requests to modify and process data as it is being retrieved from S3. This perfectly fits the requirement of conditionally removing PII based on the access context without storing multiple copies of the data. https://aws.amazon.com/s3/features/object-lambda/

By creating an S3 Object Lambda access point and configuring it to invoke the removePii Lambda function, the function will automatically be triggered whenever a GET request is made through that specific access point.

The Lambda function will then remove the PII from the object before returning it to the requester. For authorized users accessing the S3 bucket directly or through a different access point, they'll receive the original, unaltered record. For third parties accessing via Object Lambda Access point, PII will be removed.

S3 Access Points help you manage access to your data at scale. An access point is a named network endpoint that is attached to a bucket that you can use to perform S3 object operations, such as GetObject and PutObject. Object Lambda is a feature of S3 Access Points.

Why other options are incorrect:

A and B: S3 event notifications trigger Lambda functions based on events like PUT or GET requests, but they don't inherently modify the content returned by the GET request. They would require you to store a modified copy of the data somewhere, which violates the requirement of maintaining only one copy. Also, triggering on a GET request (option A) is too late because the data is already being sent to the requester.

D:GetObjectLegalHold is an S3 API function for managing object retention, not for dynamically modifying object content based on access. It is irrelevant to the problem.

Question: 15

A developer is deploying an AWS Lambda function The developer wants the ability to return to older versions of the function quickly and seamlessly.
How can the developer achieve this goal with the LEAST operational overhead?

A.Use AWS OpsWorks to perform blue/green deployments.

B.Use a function alias with different versions.

C.Maintain deployment packages for older versions in Amazon S3.

D.Use AWS CodePipeline for deployments and rollbacks.

Answer: B

Explanation:

The best approach for seamless and quick rollbacks of AWS Lambda functions with minimal operational overhead is using a function alias with different versions. Here's why:

Lambda versions are immutable snapshots of the function code and configuration. Creating versions is straightforward within Lambda. (https://docs.aws.amazon.com/lambda/latest/dg/configuration-versions.html)

Aliases act as pointers to specific Lambda versions. By directing an alias to a previous version, you effectively roll back your function's active code. (https://docs.aws.amazon.com/lambda/latest/dg/configuration-aliases.html)

This mechanism is significantly less complex than other approaches. OpsWorks (option A) is overkill for simple rollbacks and involves substantial configuration. CodePipeline (option D) introduces a CI/CD pipeline, which is suitable for more complex deployment scenarios but adds unnecessary overhead for simple rollbacks. Maintaining packages in S3 (option C) requires manual deployment processes, eliminating the "seamless" requirement and adding operational burden.

Aliases allow you to shift traffic between versions gradually, enabling canary deployments and A/B testing if needed, though that isn't the primary goal here. Rolling back with aliases only requires changing the alias's target version, which is a quick and easily automated operation. No new deployments are needed. The function's previous code is already deployed in its versioned form.

Moreover, aliases can be associated with different environment variables. This allows a rollback not only to the previous code but also to the previous environment configurations. This approach aligns perfectly with the problem's requirements of minimal operational overhead while enabling swift and straightforward rollbacks.

Question: 16

A developer has written an AWS Lambda function. The function is CPU-bound. The developer wants to ensure that the function returns responses quickly.
How can the developer improve the function's performance?

A.Increase the function's CPU core count.

B.Increase the function's memory.

C.Increase the function's reserved concurrency.

D.Increase the function's timeout.

Answer: B

Explanation:

The correct answer is **B. Increase the function's memory.**

Here's a detailed justification:

Lambda functions automatically allocate CPU power proportional to the amount of memory configured. A CPU-bound function's performance directly benefits from increased processing power. By increasing the memory allocated to the Lambda function, the underlying compute also increases, providing more CPU cores and processing power. This allows the function to execute its CPU-intensive tasks faster, leading to quicker response times.

Option A, "Increase the function's CPU core count," is not directly possible within the AWS Lambda configuration. You cannot manually adjust the CPU core count. Instead, you indirectly increase CPU power by increasing the memory.

Option C, "Increase the function's reserved concurrency," improves the function's ability to handle a larger number of concurrent requests, but it does not directly address the performance of a single CPU-bound function execution. Concurrency manages the number of simultaneous executions, not the speed of a single execution.

Option D, "Increase the function's timeout," only allows the function to run longer before being terminated; it doesn't improve its performance. It's a workaround for slow performance, not a solution to make it faster.

In summary, increasing the memory allocated to a CPU-bound Lambda function increases the available CPU power, resulting in faster execution and improved response times. This is the most direct and effective way to improve the function's performance in this scenario.

Further research:

AWS Lambda Pricing:https://aws.amazon.com/lambda/pricing/ - Understanding how memory affects the cost and performance.

Configuring Lambda function options:https://docs.aws.amazon.com/lambda/latest/dg/configuration-options.html - Detailed information on memory allocation and how it relates to underlying resources.

Question: 17

For a deployment using AWS Code Deploy, what is the run order of the hooks for in-place deployments?

A.BeforeInstall -> ApplicationStop -> ApplicationStart -> AfterInstall

B.ApplicationStop -> BeforeInstall -> AfterInstall -> ApplicationStart

C.BeforeInstall -> ApplicationStop -> ValidateService -> ApplicationStart

D.ApplicationStop -> BeforeInstall -> ValidateService -> ApplicationStart

Answer: B

Explanation:

The correct answer is B: ApplicationStop -> BeforeInstall -> AfterInstall -> ApplicationStart.

AWS CodeDeploy's in-place deployments follow a specific lifecycle event sequence for managing application updates on existing instances. The ApplicationStop hook is executed first. This is crucial because it halts the currently running version of the application, preventing conflicts during the update. After the existing application is stopped, the BeforeInstall hook is triggered. This allows for tasks like backing up the current application version or preparing the environment for the new deployment.

Next, the AfterInstall hook is executed. This hook facilitates actions needed immediately after the new application version is installed, such as configuring file permissions or setting up symbolic links. Finally, the ApplicationStart hook starts the newly deployed application. This makes the updated application accessible to users.

Option A is incorrect because it places BeforeInstall before ApplicationStop, potentially causing conflicts if the new installation interferes with the still-running application. Option C and D include ValidateService, which is not a standard hook for in-place deployments. While you can configure custom hooks, it doesn't replace nor disrupt the order of these core hooks. The prescribed order ensures a smooth transition from the old application version to the new one, minimizing downtime and preventing errors. The order prioritizes stopping the running application first, preparing for the new version, finalizing the installation, and then starting the updated application. This approach is fundamental to in-place deployments.

For more information, refer to the AWS CodeDeploy documentation on deployment lifecycles: https://docs.aws.amazon.com/codedeploy/latest/userguide/reference-appspec-file-structure-hooks.html and https://docs.aws.amazon.com/codedeploy/latest/userguide/deployments-lifecycle.html.

Question: 18

A company is building a serverless application on AWS. The application uses an AWS Lambda function to process customer orders 24 hours a day, 7 days a week. The Lambda function calls an external vendor's HTTP API to process payments. During load tests, a developer discovers that the external vendor payment processing API occasionally times out and returns errors. The company expects that some payment processing API calls will return errors.
The company wants the support team to receive notifications in near real time only when the payment processing external API error rate exceed 5% of the total number of transactions in an hour. Developers need to use an existing Amazon Simple Notification Service (Amazon SNS) topic that is configured to notify the support team. Which solution will meet these requirements?

A.Write the results of payment processing API calls to Amazon CloudWatch. Use Amazon CloudWatch Logs Insights to query the CloudWatch logs. Schedule the Lambda function to check the CloudWatch logs and notify the existing SNS topic.

B.Publish custom metrics to CloudWatch that record the failures of the external payment processing API calls. Configure a CloudWatch alarm to notify the existing SNS topic when error rate exceeds the specified rate.

C.Publish the results of the external payment processing API calls to a new Amazon SNS topic. Subscribe the support team members to the new SNS topic.

D.Write the results of the external payment processing API calls to Amazon S3. Schedule an Amazon Athena query to run at regular intervals. Configure Athena to send notifications to the existing SNS topic when the error rate exceeds the specified rate.

Answer: B

Explanation:

The best solution is **B: Publish custom metrics to CloudWatch that record the failures of the external payment processing API calls. Configure a CloudWatch alarm to notify the existing SNS topic when the error rate exceeds the specified rate.**

Here's why:

Real-time monitoring and alerting: CloudWatch is designed for real-time monitoring of metrics and setting up alarms based on thresholds. This aligns with the requirement for near real-time notifications when the error rate exceeds 5%.

Custom metrics: Since the standard CloudWatch metrics might not directly track the external API call failures, custom metrics are crucial. We can publish metrics specifically recording successful and failed API calls.

CloudWatch Alarms: CloudWatch alarms allow you to define a metric (in this case, the error rate calculated from the custom metrics), a threshold (5%), and an action (notify the existing SNS topic) when the threshold is breached.

Leveraging existing infrastructure: This solution reuses the existing SNS topic, minimizing the need for new configurations and integrations.

Why other options are less suitable:

A: CloudWatch Logs Insights: While CloudWatch Logs Insights can query logs, it's not the best option for real-time monitoring and alerting. Scheduled Lambda functions querying logs would introduce latency and complexity. It is not designed for continuous, real-time monitoring and alarming.

C: New SNS topic: Creating a new SNS topic would require managing an additional notification stream for the support team when the requirement is to reuse the existing one. It doesn't solve the metric aggregation and threshold monitoring problem.

D: Amazon S3 and Athena: This approach is geared towards analytical workloads, not real-time monitoring and alerting. Storing data in S3, running Athena queries at intervals, and then triggering notifications would introduce significant latency and operational overhead compared to using CloudWatch. It is not designed for real-time or near real-time alerting and monitoring.

In summary, Option B provides the most efficient, real-time, and cost-effective solution by directly leveraging CloudWatch's monitoring and alerting capabilities with custom metrics and alarms. It efficiently meets the requirement of notifying the support team through the existing SNS topic when the error rate exceeds the specified threshold.

Authoritative Links:

CloudWatch Custom Metrics:
https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/publishingMetrics.html
CloudWatch Alarms:
https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/AlarmThatSendsEmail.html

Question: 19

A company is offering APIs as a service over the internet to provide unauthenticated read access to statistical information that is updated daily. The company uses Amazon API Gateway and AWS Lambda to develop the APIs. The service has become popular, and the company wants to enhance the responsiveness of the APIs.
Which action can help the company achieve this goal?

A.Enable API caching in API Gateway.
B.Configure API Gateway to use an interface VPC endpoint.
C.Enable cross-origin resource sharing (CORS) for the APIs.
D.Configure usage plans and API keys in API Gateway.

Answer: A

Explanation:

The correct answer is A: Enable API caching in API Gateway.

Here's a detailed justification:

The primary goal is to improve API responsiveness for unauthenticated read access to frequently updated, but not continuously changing, statistical data. API caching directly addresses this by storing API responses in API Gateway for a specified time (TTL). Subsequent requests for the same data within the TTL are served from the cache, significantly reducing latency and the load on the backend Lambda functions. This avoids repeatedly executing the Lambda function and querying data sources for the same information.

Option B, configuring API Gateway to use an interface VPC endpoint, is more relevant for private APIs accessed within a VPC and not public APIs exposed over the internet. While it enhances security and internal network performance, it doesn't improve responsiveness for public internet-facing requests.

Option C, enabling CORS, is crucial for browser-based applications making cross-origin requests to the APIs. However, CORS primarily addresses security concerns related to JavaScript making requests from different domains and doesn't directly improve API response times. The service is already offered over the internet and becoming popular, implying the company has probably already configured CORS appropriately.

Option D, configuring usage plans and API keys, is primarily used for managing API access, rate limiting, and monetization. While useful for controlling access and preventing abuse, it doesn't inherently improve the speed with which API requests are served. It is more relevant for authenticated access to API's where usage is controlled.

Caching is ideally suited for read-heavy workloads where the underlying data changes periodically, as is the case here (daily updates). API Gateway caching is simple to implement and can provide a dramatic improvement in performance and responsiveness, as it offloads the Lambda functions, improves end user experience, and reduces latency.

Refer to the following AWS documentation for more details on API Gateway caching:

Caching API Responses in Amazon API Gateway
Amazon API Gateway Pricing - Caching

Question: 20

A developer wants to store information about movies. Each movie has a title, release year, and genre. The movie information also can include additional properties about the cast and production crew. This additional information is inconsistent across movies. For example, one movie might have an assistant director, and another movie might have an animal trainer.
The developer needs to implement a solution to support the following use cases:
For a given title and release year, get all details about the movie that has that title and release year.
For a given title, get all details about all movies that have that title.
For a given genre, get all details about all movies in that genre.
Which data store configuration will meet these requirements?

A.Create an Amazon DynamoDB table. Configure the table with a primary key that consists of the title as the partition key and the release year as the sort key. Create a global secondary index that uses the genre as the partition key and the title as the sort key.

B.Create an Amazon DynamoDB table. Configure the table with a primary key that consists of the genre as the partition key and the release year as the sort key. Create a global secondary index that uses the title as the partition key.

C.On an Amazon RDS DB instance, create a table that contains columns for title, release year, and genre. Configure the title as the primary key.

D.On an Amazon RDS DB instance, create a table where the primary key is the title and all other data is encoded into JSON format as one additional column.

Answer: A

Explanation:

The most appropriate data store configuration is option A, utilizing Amazon DynamoDB with a specific key design and a global secondary index (GSI). DynamoDB is well-suited for handling flexible schemas, which aligns with the inconsistent additional properties about cast and production crew.

The primary key configuration, with title as the partition key and release year as the sort key, efficiently supports the first use case: retrieving movie details by title and release year. The partition key enables quick data location across DynamoDB's distributed architecture, and the sort key allows for efficient querying within that partition based on release year.

The GSI, using genre as the partition key and title as the sort key, enables efficient querying for all movies within a specific genre (the third use case). This index copies the genre attribute and arranges it for fast retrieval of movies belonging to that genre. The title as the sort key allows to return data in certain order.

Finally, the second use case—retrieving all movies with a specific title—is inherently supported by the primary key structure, as all movies with the same title will reside within the same partition. While a filter expression might be needed to filter for different year values, it's still efficient given DynamoDB's high-performance capabilities.

Option B is incorrect because it prioritizes genre-based lookups over title-based lookups, making the primary query inefficient. Options C and D, which suggest using Amazon RDS, are less suitable for handling a flexible schema and would require more complex schema management or JSON parsing, impacting performance and increasing development overhead. DynamoDB's native ability to store flexible schemas and query efficiently makes it the most appropriate choice here.

Authoritative Links:

Amazon DynamoDB - Key Design
Amazon DynamoDB - Global Secondary Indexes

Question: 21

A developer maintains an Amazon API Gateway REST API. Customers use the API through a frontend UI and Amazon Cognito authentication.
The developer has a new version of the API that contains new endpoints and backward-incompatible interface changes. The developer needs to provide beta access to other developers on the team without affecting customers.
Which solution will meet these requirements with the LEAST operational overhead?

A.Define a development stage on the API Gateway API. Instruct the other developers to point the endpoints to the development stage.

B.Define a new API Gateway API that points to the new API application code. Instruct the other developers to point the endpoints to the new API.

C.Implement a query parameter in the API application code that determines which code version to call.

D.Specify new API Gateway endpoints for the API endpoints that the developer wants to add.

Answer: A

Explanation:

Option A, defining a development stage in API Gateway, is the most suitable approach because it leverages API Gateway's built-in features for managing multiple versions of an API. API Gateway stages are independent environments that allow developers to deploy and test API changes without impacting the production environment. This feature simplifies beta testing, allowing specific users (in this case, other developers on the team) to access the new API version through the development stage endpoint.

Option B, creating a new API Gateway API, would require replicating the entire API configuration, increasing operational overhead. It duplicates resources and adds complexity to management.

Option C, using a query parameter in the application code, introduces logic within the application to handle versioning. This can become complex and difficult to maintain, tightly coupling the application code to the API versioning scheme. It also bypasses API Gateway's features for stage management and traffic control.

Option D, defining new API Gateway endpoints for new features, would lead to an inconsistent API structure. This option is suitable for adding new features that do not break backwards compatibility, however, in this case there are backward-incompatible interface changes, so this will not fully resolve the problem.

Therefore, option A offers a cleaner, more manageable solution with the least operational overhead by utilizing API Gateway's built-in staging capabilities. This approach ensures customers are not impacted while allowing developers to test the beta version of the API.

https://docs.aws.amazon.com/apigateway/latest/developerguide/using-apigateway-stages.htmlhttps://aws.amazon.com/api-gateway/

Question: 22

A developer is creating an application that will store personal health information (PHI). The PHI needs to be encrypted at all times. An encrypted Amazon RDS for MySQL DB instance is storing the data. The developer wants to increase the performance of the application by caching frequently accessed data while adding the ability to sort or rank the cached datasets.
Which solution will meet these requirements?

A.Create an Amazon ElastiCache for Redis instance. Enable encryption of data in transit and at rest. Store frequently accessed data in the cache.

B.Create an Amazon ElastiCache for Memcached instance. Enable encryption of data in transit and at rest. Store frequently accessed data in the cache.

C.Create an Amazon RDS for MySQL read replica. Connect to the read replica by using SSL. Configure the read replica to store frequently accessed data.

D.Create an Amazon DynamoDB table and a DynamoDB Accelerator (DAX) cluster for the table. Store frequently accessed data in the DynamoDB table.

Answer: A

Explanation:

The correct answer is **A. Create an Amazon ElastiCache for Redis instance. Enable encryption of data in transit and at rest. Store frequently accessed data in the cache.**

Here's a detailed justification:

1. **PHI and Encryption:** The application deals with PHI (Personal Health Information), necessitating end-to-end encryption, both in transit and at rest, to comply with regulations like HIPAA. This requirement immediately rules out options that don't explicitly address encryption or rely solely on SSL.

2. **Caching for Performance:** Caching frequently accessed data is a standard technique to improve application performance by reducing the load on the database. ElastiCache is a managed in-memory data store service specifically designed for caching.

3. **Redis Capabilities:** ElastiCache for Redis is the better choice over Memcached because Redis offers advanced data structures (like sorted sets) which support sorting and ranking cached datasets. Memcached, on the other hand, is primarily a simple key-value store without these features.

4. **Encryption in ElastiCache for Redis:** ElastiCache for Redis supports encryption at rest using AWS Key Management Service (KMS) and encryption in transit using TLS (Transport Layer Security). This ensures that the PHI is protected at all times.

5. **ElastiCache vs. RDS Read Replica:** While an RDS read replica (option C) can offload read traffic from the primary database, it's still a relational database. It doesn't provide the fast, in-memory access that ElastiCache offers. Also, managing and scaling read replicas can be more complex than using a managed caching service.

6. **ElastiCache vs. DynamoDB with DAX:** DynamoDB with DAX (option D) is suitable for high-throughput, low-latency NoSQL workloads. However, for simply caching frequently accessed data from an existing RDS database and requiring sorting capabilities, Redis is a more appropriate and potentially cost-effective solution. DynamoDB primarily works when the main database itself is DynamoDB.

In summary, ElastiCache for Redis with encryption enabled fulfills all requirements: caching for performance, support for sorting/ranking (using Redis's sorted sets), and robust encryption for protecting PHI. It is a dedicated caching solution that integrates seamlessly with AWS services and is easier to manage than alternatives like read replicas.

Authoritative Links:

Amazon ElastiCache:https://aws.amazon.com/elasticache/
ElastiCache for Redis:https://aws.amazon.com/elasticache/redis/
ElastiCache Security:https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/security.html **HIPAA Compliance:**https://aws.amazon.com/compliance/hipaa/

Question: 23

A company has a multi-node Windows legacy application that runs on premises. The application uses a network shared folder as a centralized configuration repository to store configuration files in .xml format. The company is migrating the application to Amazon EC2 instances. As part of the migration to AWS, a developer must identify a solution that provides high availability for the repository.
Which solution will meet this requirement MOST cost-effectively?

A.Mount an Amazon Elastic Block Store (Amazon EBS) volume onto one of the EC2 instances. Deploy a file system on the EBS volume. Use the host operating system to share a folder. Update the application code to read and write configuration files from the shared folder.

B.Deploy a micro EC2 instance with an instance store volume. Use the host operating system to share a folder. Update the application code to read and write configuration files from the shared folder.

C.Create an Amazon S3 bucket to host the repository. Migrate the existing .xml files to the S3 bucket. Update the application code to use the AWS SDK to read and write configuration files from Amazon S3.

D.Create an Amazon S3 bucket to host the repository. Migrate the existing .xml files to the S3 bucket. Mount the S3 bucket to the EC2 instances as a local volume. Update the application code to read and write configuration files from the disk.

Answer: C

Explanation:

The correct answer is C, using Amazon S3 to host the configuration repository. Here's why:

Option A, using EBS and sharing a folder from an EC2 instance, creates a single point of failure. If that EC2 instance fails, the configuration repository becomes unavailable. EBS volumes, while durable, are tied to a specific Availability Zone, hindering high availability across multiple AZs.

Option B, utilizing a micro EC2 instance with instance store, is even less reliable. Instance store volumes are ephemeral, meaning they are lost when the instance is stopped or terminated. This makes it entirely unsuitable for a critical configuration repository.

Option D, using S3 with a mounted local volume, introduces unnecessary complexity and performance bottlenecks. While theoretically possible with solutions like s3fs, it's not a native feature of AWS and adds overhead, negating some of S3's performance advantages. Mounting S3 as a local drive can also introduce latency compared to accessing S3 directly via the AWS SDK. Additionally, potential compatibility issues may arise, and the "local" filesystem view doesn't provide true atomic consistency.

Option C, directly leveraging Amazon S3, provides the best combination of high availability, cost-effectiveness, and scalability. S3 is designed for 99.999999999% durability and is inherently highly available across multiple Availability Zones. Updating the application code to use the AWS SDK for S3 allows direct and efficient interaction with the configuration files. S3 offers a cost-effective storage solution, especially for infrequently accessed configuration files. There's no need to manage file servers or worry about replication, as S3 handles this automatically. The AWS SDKs are well-documented and provide robust features for reading and writing objects (configuration files). This approach simplifies the architecture and minimizes operational overhead.

Further Reading:

Amazon S3:https://aws.amazon.com/s3/
AWS SDKs:https://aws.amazon.com/developer/tools/

Question: 24

A company wants to deploy and maintain static websites on AWS. Each website's source code is hosted in one of several version control systems, including AWS CodeCommit, Bitbucket, and GitHub.
The company wants to implement phased releases by using development, staging, user acceptance testing, and production environments in the AWS Cloud. Deployments to each environment must be started by code merges on the relevant Git branch. The company wants to use HTTPS for all data exchange. The company needs a solution that does not require servers to run continuously.
Which solution will meet these requirements with the LEAST operational overhead?

A.Host each website by using AWS Amplify with a serverless backend. Conned the repository branches that correspond to each of the desired environments. Start deployments by merging code changes to a desired branch.

B.Host each website in AWS Elastic Beanstalk with multiple environments. Use the EB CLI to link each repository branch. Integrate AWS CodePipeline to automate deployments from version control code merges.

C.Host each website in different Amazon S3 buckets for each environment. Configure AWS CodePipeline to pull source code from version control. Add an AWS CodeBuild stage to copy source code to Amazon S3.

D.Host each website on its own Amazon EC2 instance. Write a custom deployment script to bundle each website's static assets. Copy the assets to Amazon EC2. Set up a workflow to run the script when code is merged.

Answer: A

Explanation:

The correct answer is A, using AWS Amplify. Here's why:

AWS Amplify is specifically designed for hosting static websites and single-page applications with serverless backends. It directly integrates with various version control systems (CodeCommit, Bitbucket, GitHub) making connecting to repository branches simple. Amplify automatically deploys changes when code is merged to a branch, enabling the desired phased release workflow (development, staging, UAT, production). Amplify provides built-in HTTPS support, fulfilling the security requirement. Moreover, Amplify is a serverless platform, meaning the company doesn't need to manage any underlying servers, minimizing operational overhead, and costs are only incurred during deployments or when the website is accessed. It's quick to set up using the AWS Amplify console.

Option B, using Elastic Beanstalk, is less suitable because Elastic Beanstalk is designed for more complex applications and requires managed server instances, violating the "no continuously running servers" requirement. While CodePipeline could be used for deployments, it adds complexity compared to Amplify's built-in integration.Option C, using S3 and CodePipeline/CodeBuild, requires more manual configuration.
You'd need to manage bucket policies, configure CodePipeline stages, and manually set up HTTPS via CloudFront. This increases operational overhead. S3 also requires extra configuration such as CloudFront to make the website accessible via HTTPS, while Amplify does it by default.Option D, using EC2, requires constant server management and manual deployment scripts. This contradicts the "least operational overhead" requirement and the need for serverless infrastructure.

In summary, AWS Amplify is the most efficient and straightforward solution for hosting static websites with version control integration, phased releases, HTTPS, and minimal operational overhead.Authoritative links:

AWS Amplify: https://aws.amazon.com/amplify/
AWS CodePipeline: https://aws.amazon.com/codepipeline/
Amazon S3: https://aws.amazon.com/s3/
AWS Elastic Beanstalk: https://aws.amazon.com/elasticbeanstalk/

Question: 25

A company is migrating an on-premises database to Amazon RDS for MySQL. The company has read-heavy workloads. The company wants to refactor the code to achieve optimum read performance for queries. Which solution will meet this requirement with LEAST current and future effort?

A.Use a multi-AZ Amazon RDS deployment. Increase the number of connections that the code makes to the database or increase the connection pool size if a connection pool is in use.

B.Use a multi-AZ Amazon RDS deployment. Modify the code so that queries access the secondary RDS instance.

C.Deploy Amazon RDS with one or more read replicas. Modify the application code so that queries use the URL for the read replicas.

D.Use open source replication software to create a copy of the MySQL database on an Amazon EC2 instance. Modify the application code so that queries use the IP address of the EC2 instance.

Answer: C

Explanation:

The correct answer is C because it directly addresses the requirement of achieving optimum read performance for read-heavy workloads with minimal effort. Amazon RDS Read Replicas are specifically designed for scaling read operations, and the change needed in the application is simply pointing read queries to the replica's endpoint.

Option A is incorrect because a multi-AZ deployment primarily provides high availability and failover capabilities, not specifically read scaling. Increasing the number of connections or the connection pool size might help to some degree, but it is not the optimal solution for read scaling and can potentially overload the primary database.

Option B is problematic as directly querying the secondary instance in a multi-AZ setup is not a supported or recommended practice. The secondary instance is for failover, and attempting to use it for read traffic can lead to instability during failovers and is not guaranteed to provide consistent data. AWS documentation advises against this approach.

Option D is an overly complex solution. It requires manually setting up and managing replication software on an EC2 instance, which increases operational overhead and management complexity. Amazon RDS Read Replicas are a managed service, meaning AWS handles the replication process, patching, and other administrative tasks, reducing the burden on the company. Choosing a managed service aligns with the goal of minimizing current and future effort. Furthermore, manual replication might not be as performant or reliable as the managed RDS Read Replicas.

Using Read Replicas provides a purpose-built, scalable, and managed solution for handling read-heavy workloads on Amazon RDS. It's the least effort approach as it utilizes a service directly designed for the specified use case, abstracting away the complexities of manually managing replication infrastructure.

Authoritative links:

Amazon RDS Read Replicas:
https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ReadRepl.html

Amazon RDS Multi-AZ Deployments:
https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html

Question: 26

A developer is creating an application that will be deployed on IoT devices. The application will send data to a RESTful API that is deployed as an AWS Lambda function. The application will assign each API request a unique identifier. The volume of API requests from the application can randomly increase at any given time of day. During periods of request throttling, the application might need to retry requests. The API must be able to handle duplicate requests without inconsistencies or data loss.
Which solution will meet these requirements?

A.Create an Amazon RDS for MySQL DB instance. Store the unique identifier for each request in a database table. Modify the Lambda function to check the table for the identifier before processing the request.

B.Create an Amazon DynamoDB table. Store the unique identifier for each request in the table. Modify the Lambda function to check the table for the identifier before processing the request.

C.Create an Amazon DynamoDB table. Store the unique identifier for each request in the table. Modify the Lambda function to return a client error response when the function receives a duplicate request.

D.Create an Amazon ElastiCache for Memcached instance. Store the unique identifier for each request in the cache. Modify the Lambda function to check the cache for the identifier before processing the request.

Answer: B

Explanation:

The most suitable solution is to use DynamoDB for handling potentially duplicate API requests from IoT devices routed through a Lambda function. Here's why:

DynamoDB excels at handling high-volume, bursty workloads, aligning perfectly with the application's requirement to handle random spikes in API requests. Its scalability and speed make it suitable for scenarios where request volume can fluctuate significantly, which is common with IoT device applications.
(https://aws.amazon.com/dynamodb/)

The core requirement is to handle duplicate requests idempotently, preventing inconsistencies and data loss. By storing a unique identifier for each request in DynamoDB, the Lambda function can efficiently check if a request has already been processed before executing the main logic. If the identifier exists in the DynamoDB table, the Lambda function knows it's a duplicate and can take appropriate action (e.g., return a cached response or simply acknowledge the request without reprocessing).
(https://aws.amazon.com/blogs/database/implementing-application-idempotency-on-aws/)

Option A (RDS MySQL) is less optimal. RDS, while reliable, is relational database, is better suited for consistent structured data, may not scale and perform as effectively as DynamoDB for high-volume writes and reads needed in this scenario, especially during peak load. Additionally, the overhead of managing an RDS instance is greater than managing a DynamoDB table.

Option C (DynamoDB with client error) fails to handle the retry requirement effectively. Returning a client error on duplicate requests would force the IoT devices to retry indefinitely, potentially exacerbating throttling issues. The goal is to process the request, even if it's a duplicate, rather than rejecting it.

Option D (ElastiCache Memcached) is not ideal for persistent data storage. Memcached is an in-memory caching service, and data can be lost if the cache is cleared or the instance fails. For idempotency, you need a durable store. The persistent check needed for preventing duplicate executions requires a durable store.

In summary, DynamoDB's scalability, low latency, and persistence make it the best choice for implementing idempotency in a serverless API application with a high potential for duplicate requests.

Question: 27

A developer wants to expand an application to run in multiple AWS Regions. The developer wants to copy Amazon Machine Images (AMIs) with the latest changes and create a new application stack in the destination Region. According to company requirements, all AMIs must be encrypted in all Regions. However, not all the AMIs that the company uses are encrypted. How can the developer expand the application to run in the destination Region while meeting the encryption requirement?

A.Create new AMIs, and specify encryption parameters. Copy the encrypted AMIs to the destination Region. Delete the unencrypted AMIs.

B.Use AWS Key Management Service (AWS KMS) to enable encryption on the unencrypted AMIs. Copy the encrypted AMIs to the destination Region.

C.Use AWS Certificate Manager (ACM) to enable encryption on the unencrypted AMIs. Copy the encrypted AMIs to the destination Region.

D.Copy the unencrypted AMIs to the destination Region. Enable encryption by default in the destination Region.

Answer: A

Explanation:

The correct answer is A. Here's why:

Option A is the most straightforward and reliable approach to ensure all AMIs are encrypted as required before deploying the application to the new region. You can create new AMIs from existing unencrypted AMIs and specify encryption during the AMI creation process. This ensures that the new AMIs are encrypted from the outset. Copying these newly encrypted AMIs to the destination region fulfills the requirement that all AMIs be encrypted in all regions. Deleting the unencrypted AMIs is a prudent step for security and compliance.

Option B is incorrect because while you can use KMS to encrypt EBS volumes associated with EC2 instances, you can't directly "enable encryption" on an existing unencrypted AMI using KMS as a single step. You would need to create a new AMI by backing the unencrypted AMI while also specifying encryption parameters (similar to option A).

Option C is incorrect. AWS Certificate Manager (ACM) is used for managing SSL/TLS certificates for securing network communications. It does not handle AMI encryption. Therefore, it's irrelevant in this scenario.

Option D is incorrect. While you can set encryption by default in a Region, this setting primarily affects new EBS volumes created in that region. Copying an unencrypted AMI will still result in an unencrypted AMI in the destination region, even if encryption by default is enabled. You will still need to create a new AMI from the unencrypted AMI to enable encryption.

Therefore, creating new AMIs with encryption enabled during creation is the correct approach to ensure all AMIs are encrypted and meet the stated company requirements.

Relevant AWS documentation:

Creating an AMI:https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/creating-an-ami.html
Encrypting AMIs and EBS snapshots:
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSEncryption.html
Copying an AMI:https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/CopyingAMIs.html

Question: 28

A company hosts a client-side web application for one of its subsidiaries on Amazon S3. The web application can be accessed through Amazon CloudFront from https://www.example.com. After a successful rollout, the company wants to host three more client-side web applications for its remaining subsidiaries on three separate S3 buckets. To achieve this goal, a developer moves all the common JavaScript files and web fonts to a central S3 bucket that serves the web applications. However, during testing, the developer notices that the browser blocks the JavaScript files and web fonts.
What should the developer do to prevent the browser from blocking the JavaScript files and web fonts?

A.Create four access points that allow access to the central S3 bucket. Assign an access point to each web application bucket.

B.Create a bucket policy that allows access to the central S3 bucket. Attach the bucket policy to the central S3 bucket

C.Create a cross-origin resource sharing (CORS) configuration that allows access to the central S3 bucket. Add the CORS configuration to the central S3 bucket.

D.Create a Content-MD5 header that provides a message integrity check for the central S3 bucket. Insert the Content-MD5 header for each web application request.

Answer: C

Explanation:

The problem arises because the JavaScript files and web fonts are now being served from a different origin (the central S3 bucket) than the web applications themselves (the individual S3 buckets for each subsidiary). This violates the browser's Same-Origin Policy, which prevents web pages from making requests to a different domain than the one which served the web page. Browsers implement this security feature to prevent malicious scripts from accessing sensitive data from other websites.

To overcome this, the correct solution is to configure Cross-Origin Resource Sharing (CORS) on the central S3 bucket. CORS is a mechanism that uses HTTP headers to tell browsers to give a web application running at one origin, access to selected resources from a different origin. By adding a CORS configuration to the central S3 bucket, the developer explicitly grants permission to the web applications hosted on the other S3 buckets (or example.com via CloudFront) to access the JavaScript files and web fonts. This tells the browser that it's safe to allow the cross-origin requests.

Option A is incorrect because S3 access points are primarily for managing access control on a more granular level within the bucket, not for addressing cross-origin issues. They don't directly handle CORS.

Option B is not sufficient. A bucket policy controls which AWS accounts and IAM principals can access the S3 bucket. While a bucket policy is essential for general security, it doesn't address the Same-Origin Policy enforced by web browsers. The browser checks for CORS headers, not the bucket policy, to determine if a cross-origin request is allowed.

Option D is incorrect because the Content-MD5 header is for verifying data integrity, not for addressing CORS issues. It verifies that the data received is the same as the data that was sent, and has nothing to do with the browser's security policy for cross-origin requests.

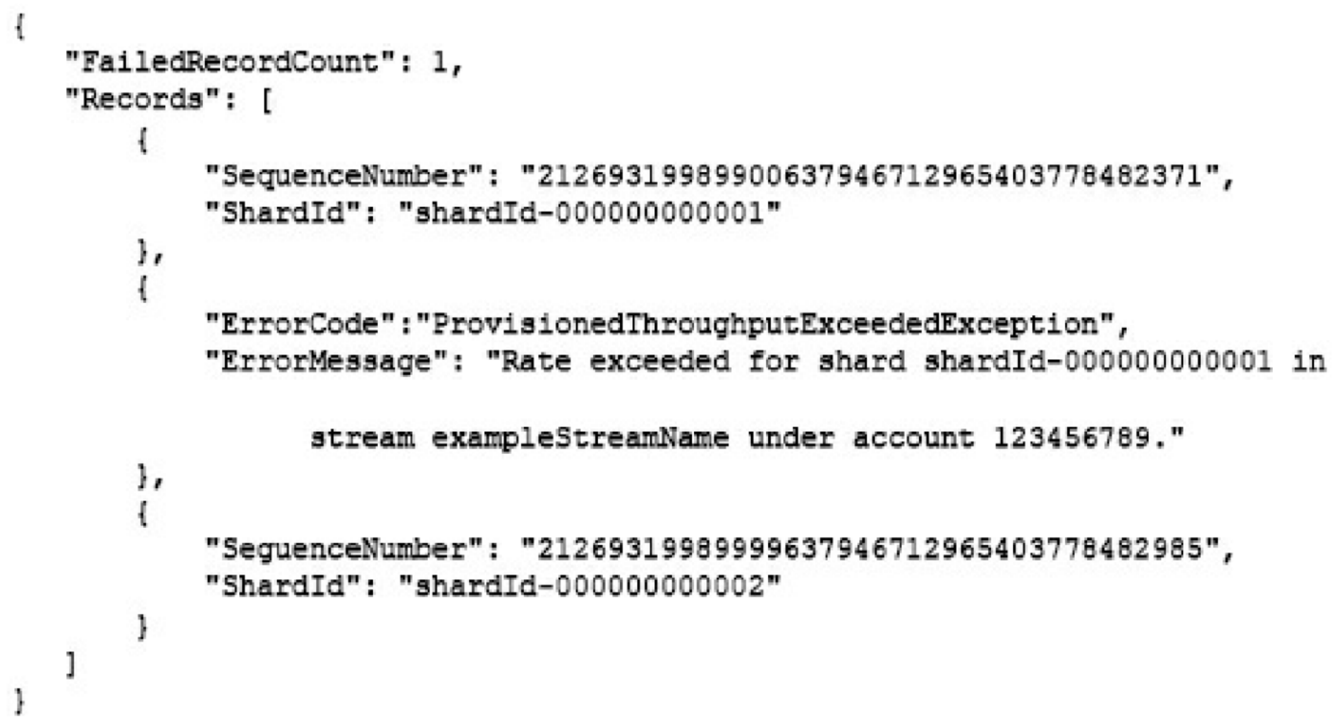
Therefore, CORS configuration on the central S3 bucket is the appropriate and standard method to solve this specific problem. It allows the browser to understand that the web applications are authorized to access the resources in the central bucket, thus preventing the blocking of JavaScript files and web fonts.

Further research:

CORS on AWS
Same-Origin Policy
CORS

Question: 29

An application is processing clickstream data using Amazon Kinesis. The clickstream data feed into Kinesis experiences periodic spikes. The PutRecords API call occasionally fails and the logs show that the failed call returns the response shown below:



```
{
    "FailedRecordCount": 1,
    "Records": [
        {
            "SequenceNumber": "21269319989900637946712965403778482371",
            "ShardId": "shardId-000000000001"
        },
        {
            "ErrorCode":"ProvisionedThroughputExceededException",
            "ErrorMessage": "Rate exceeded for shard shardId-000000000001 in

                  stream exampleStreamName under account 123456789."
        },
        {
            "SequenceNumber": "21269319989999637946712965403778482985",
            "ShardId": "shardId-000000000002"
        }
    ]
}
```

Which techniques will help mitigate this exception? (Choose two.)

A.Implement retries with exponential backoff.
B.Use a PutRecord API instead of PutRecords.
C.Reduce the frequency and/or size of the requests.
D.Use Amazon SNS instead of Kinesis.
E.Reduce the number of KCL consumers.

Answer: AC

Explanation:

AC as per AWS : Provisioned Throughput Exceeded Exception The request rate for the stream is too high, or the requested data is too large for the available throughput. Reduce the frequency or size of your requests. For more information, see Streams Limits in the Amazon Kinesis Data Streams Developer Guide, and Error Retries and Exponential Back off in AWS in the AWS General

Reference.

https://docs.aws.amazon.com/kinesis/latest/APIReference/API_PutRecords.html

Question: 30

A company has an application that uses Amazon Cognito user pools as an identity provider. The company must secure access to user records. The company has set up multi-factor authentication (MFA). The company also wants to send a login activity notification by email every time a user logs in.
What is the MOST operationally efficient solution that meets this requirement?

A.Create an AWS Lambda function that uses Amazon Simple Email Service (Amazon SES) to send the email notification. Add an Amazon API Gateway API to invoke the function. Call the API from the client side when login confirmation is received.

B.Create an AWS Lambda function that uses Amazon Simple Email Service (Amazon SES) to send the email notification. Add an Amazon Cognito post authentication Lambda trigger for the function.

C.Create an AWS Lambda function that uses Amazon Simple Email Service (Amazon SES) to send the email notification. Create an Amazon CloudWatch Logs log subscription filter to invoke the function based on the login status.

D.Configure Amazon Cognito to stream all logs to Amazon Kinesis Data Firehose. Create an AWS Lambda function to process the streamed logs and to send the email notification based on the login status of each user.

Answer: B

Explanation:

The correct answer is **B**. Here's a detailed justification:

Option B is the most operationally efficient solution because it leverages the built-in integration between Amazon Cognito and AWS Lambda through Cognito triggers. Specifically, the "Post authentication" trigger is designed to execute a Lambda function after a user successfully authenticates. This means the Lambda function automatically receives the event data related to the login, including user details and authentication context, making it easy to extract the necessary information for the email notification. This eliminates the need for any custom polling, API calls, or complex log parsing.

Using a Cognito trigger simplifies the implementation and maintenance. Cognito handles invoking the Lambda function at the appropriate point in the authentication flow. This reduces the operational overhead, as you don't have to manage the invocation logic yourself. Additionally, the trigger mechanism is reliable and scales automatically with the Cognito service.

Here's why other options are less efficient:

A: Calling an API Gateway endpoint from the client-side introduces complexity and potential unreliability. The client-side code would need to handle retries and error scenarios. Also, it relies on the client to properly trigger the notification, making it less reliable than a server-side trigger. It also introduces unnecessary costs and operational overhead through API Gateway.
C: Using CloudWatch Logs and a log subscription filter requires parsing log data to determine login events.
This adds complexity to the Lambda function and can be less reliable than direct triggers. Furthermore, relying on logs for real-time events can be slower due to log aggregation and processing latency.
D: Streaming logs to Kinesis Data Firehose and then processing them with Lambda adds significant complexity. It involves setting up and managing Kinesis Data Firehose, dealing with potential data delivery issues, and parsing complex log data. This approach is overly complex for a simple login notification. It introduces costs and operational complexity that are not justified.

Therefore, leveraging the "Post authentication" Lambda trigger in Cognito (option B) provides the most streamlined, efficient, and reliable solution for sending login activity notifications.

Supporting Links:

Amazon Cognito Lambda Triggers:https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools-working-with-aws-lambda-triggers.html
Amazon SES:https://aws.amazon.com/ses/

Question: 31

A developer has an application that stores data in an Amazon S3 bucket. The application uses an HTTP API to store and retrieve objects. When the PutObject API operation adds objects to the S3 bucket the developer must encrypt these objects at rest by using server-side encryption with Amazon S3 managed keys (SSE-S3).
Which solution will meet this requirement?

A.Create an AWS Key Management Service (AWS KMS) key. Assign the KMS key to the S3 bucket.

B.Set the x-amz-server-side-encryption header when invoking the PutObject API operation.

C.Provide the encryption key in the HTTP header of every request.

D.Apply TLS to encrypt the traffic to the S3 bucket.

Answer: B

Explanation:

The correct answer is B: Set the x-amz-server-side-encryption header when invoking the PutObject API operation. Here's why:

The requirement is to encrypt objects at rest in S3 using Server-Side Encryption with Amazon S3 Managed Keys (SSE-S3). SSE-S3 is automatically enabled at the bucket level without having to create new encryption keys. This encryption is done by S3 itself using its own keys, managed by AWS.

Option B directly addresses this requirement. The x-amz-server-side-encryption header, when included in the PutObject API request, instructs S3 to encrypt the object using SSE-S3. No key management on the developer's side is needed. S3 handles the key management completely, which is exactly what the requirement states.

Option A is incorrect because it suggests using AWS KMS keys. While using AWS KMS for server-side encryption (SSE-KMS) is valid, it's not SSE-S3, which is the specified requirement. It also involves more complexity with key creation and management.

Option C is also incorrect. Providing the encryption key in the HTTP header implies using Server-Side Encryption with Customer-Provided Keys (SSE-C), where the customer manages the encryption keys. Again, this is not SSE-S3.

Option D, applying TLS, encrypts the data in transit, protecting data as it moves between the application and S3. However, it does not address encryption at rest once the data is stored within the S3 bucket, thus not fulfilling the stated requirement.

In summary, the question explicitly asks for SSE-S3. Setting the x-amz-server-side-encryption header is the simplest and most direct way to enable SSE-S3 encryption on S3 objects during the PutObject operation, with S3 automatically handling the key management.

Reference links:

Amazon S3 Server-Side Encryption: https://docs.aws.amazon.com/AmazonS3/latest/userguide/serv-side-encryption.html
Protecting Data Using Server-Side Encryption:
https://docs.aws.amazon.com/AmazonS3/latest/userguide/protecting-data-server-side-encryption.html PUT Object - Server-Side Encryption: https://docs.aws.amazon.com/AmazonS3/latest/API/API_PutObject.html

Question: 32

A developer needs to perform geographic load testing of an API. The developer must deploy resources to multiple AWS Regions to support the load testing of the API.
How can the developer meet these requirements without additional application code?

A.Create and deploy an AWS Lambda function in each desired Region. Configure the Lambda function to create a stack from an AWS CloudFormation template in that Region when the function is invoked.

B.Create an AWS CloudFormation template that defines the load test resources. Use the AWS CLI create-stack-set command to create a stack set in the desired Regions.

C.Create an AWS Systems Manager document that defines the resources. Use the document to create the resources in the desired Regions.

D.Create an AWS CloudFormation template that defines the load test resources. Use the AWS CLI deploy command to create a stack from the template in each Region.

Answer: B

Explanation:

The correct answer is B: Create an AWS CloudFormation template that defines the load test resources. Use the AWS CLI create-stack-set command to create a stack set in the desired Regions. This is because AWS CloudFormation StackSets provide a convenient mechanism to provision stacks (and thus resources) across multiple AWS Regions from a single CloudFormation template. This eliminates the need to write custom code or scripts to handle the deployment in each region.

StackSets enable you to manage and update stacks across multiple accounts and regions. They treat a collection of stacks as a single entity. This simplifies the process of deploying identical resources to different regions, perfectly aligning with the requirement of geographic load testing.

Option A is incorrect because using Lambda to deploy CloudFormation stacks in multiple regions requires extra coding to handle Lambda invocation and region-specific deployments.

Option C is incorrect as Systems Manager Documents, while capable of managing resources, are more oriented towards configuration management and automation tasks within an existing environment, not the initial creation of infrastructure across multiple regions. It's not the most efficient approach for deploying resources to different regions, as the scope is geared towards instance specific operations more.

Option D is incorrect because using the AWS CLI deploy command would require manual execution for each region, making it less efficient and prone to errors compared to StackSets. Also this doesnt leverage a managed way to deal with the infrastructure accross the regions.

In conclusion, StackSets (option B) provides the most efficient and manageable way to provision resources across multiple AWS Regions without additional application code, by using a single template across multiple regions.

Relevant links for further research:

AWS CloudFormation StackSets:
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/stacksets-concepts.html
AWS CLI create-stack-set command:
https://awscli.amazonaws.com/v2/documentation/api/latest/reference/cloudformation/create-stack-set.html

Question: 33

A developer is creating an application that includes an Amazon API Gateway REST API in the us-east-2 Region. The developer wants to use Amazon CloudFront and a custom domain name for the API. The developer has acquired an SSL/TLS certificate for the domain from a third-party provider.
How should the developer configure the custom domain for the application?

A.Import the SSL/TLS certificate into AWS Certificate Manager (ACM) in the same Region as the API. Create a DNS A record for the custom domain.

B.Import the SSL/TLS certificate into CloudFront. Create a DNS CNAME record for the custom domain.

C.Import the SSL/TLS certificate into AWS Certificate Manager (ACM) in the same Region as the API. Create a DNS CNAME record for the custom domain.

D.Import the SSL/TLS certificate into AWS Certificate Manager (ACM) in the us-east-1 Region. Create a DNS CNAME record for the custom domain.

Answer: D

Explanation:

The correct answer is D. Here's why:

When using a custom domain with CloudFront and API Gateway, the SSL/TLS certificate must reside in AWS Certificate Manager (ACM) in the us-east-1 (N. Virginia) Region, regardless of where the API Gateway or CloudFront distribution is located. This is a specific requirement of CloudFront for custom SSL certificates.

CloudFront is a global service, and ACM certificates used with CloudFront custom domains must be in the US East (N. Virginia) Region because that's where the CloudFront infrastructure that handles SSL/TLS termination for custom domains is located. If you attempt to import a certificate for CloudFront in any other region, you will encounter an error.

Also, to map the custom domain to the CloudFront distribution, a CNAME record is required in your DNS configuration. A CNAME record creates an alias from your custom domain (e.g., api.example.com) to the CloudFront distribution's domain name (e.g., d111111abcdef8.cloudfront.net). This directs traffic from your custom domain to the CloudFront distribution, which then routes it to the underlying API Gateway.

Option A is incorrect because it places the certificate in the us-east-2 region, which is incorrect for CloudFront. It also uses a DNS A record, while a CNAME is needed.

Option B is incorrect because you don't directly import certificates into CloudFront. Certificates are managed within ACM.

Option C is incorrect because the certificate is stored in us-east-2, which is not supported for CloudFront.

In summary, for a custom domain and SSL/TLS certificate to work with CloudFront, you must: 1. Import the certificate into ACM in the us-east-1 Region. 2. Create a CNAME record in your DNS settings that points your custom domain to your CloudFront distribution's domain name.

References:

AWS Documentation - Using Alternate Domain Names (CNAMEs):
https://docs.aws.amazon.com/cloudfront/latest/developerguide/using-alternate-domain-names.html AWS Documentation - Regions where you can request or import ACM certificates:
https://docs.aws.amazon.com/acm/latest/userguide/acm-regions.html

Question: 34

A developer is creating a template that uses AWS CloudFormation to deploy an application. The application is serverless and uses Amazon API Gateway, Amazon DynamoDB, and AWS Lambda.
Which AWS service or tool should the developer use to define serverless resources in YAML?

A.CloudFormation serverless intrinsic functions
B.AWS Elastic Beanstalk
C.AWS Serverless Application Model (AWS SAM)
D.AWS Cloud Development Kit (AWS CDK)

Answer: C

Explanation:

The correct answer is C: AWS Serverless Application Model (AWS SAM).

Here's a detailed justification:

AWS SAM is a framework built on top of AWS CloudFormation specifically designed for building serverless applications. It simplifies the process of defining and deploying serverless resources such as Lambda functions, API Gateways, DynamoDB tables, and more. SAM achieves this by providing a simplified syntax using YAML or JSON to describe these resources. It uses shorthand declarations which are then translated into fully expanded CloudFormation templates during deployment.

Option A, CloudFormation intrinsic functions, are useful for general CloudFormation templates but don't provide the simplified resource definitions specifically tailored for serverless architectures that SAM offers.
Intrinsic functions are used within CloudFormation templates to perform tasks like referencing resource attributes or conditional logic. While SAM templates use CloudFormation intrinsic functions under the hood after SAM processing, they are not the primary tool for defining serverless resources in a concise manner.

Option B, AWS Elastic Beanstalk, is a platform-as-a-service (PaaS) offering that automates the deployment and management of web applications and services. While Elastic Beanstalk can be used with some serverless components, it's not primarily designed for defining and deploying a complete serverless architecture like the one described in the question (API Gateway, Lambda, DynamoDB).

Option D, AWS Cloud Development Kit (CDK), is a framework for defining cloud infrastructure in code using familiar programming languages like Python, TypeScript, or Java. While CDK can define serverless resources, AWS SAM is a more direct and concise way to create CloudFormation templates specifically tailored for serverless applications expressed in YAML or JSON. The CDK provides a higher level of abstraction and more flexibility but also can introduce more complexity than SAM for serverless projects. SAM is a much more lightweight option.

Therefore, because the question specifically asks for a service or tool to define serverless resources in YAML, AWS SAM is the most appropriate choice due to its focus on simplifying serverless deployments through declarative templates and serverless resource abstractions built directly on CloudFormation.

Here are some authoritative links for further research:

AWS SAM:https://aws.amazon.com/serverless/sam/
AWS CloudFormation:https://aws.amazon.com/cloudformation/
AWS CDK:https://aws.amazon.com/cdk/

Question: 35

A developer wants to insert a record into an Amazon DynamoDB table as soon as a new file is added to an Amazon S3 bucket.
Which set of steps would be necessary to achieve this?

A.Create an event with Amazon EventBridge that will monitor the S3 bucket and then insert the records into DynamoDB.

B.Configure an S3 event to invoke an AWS Lambda function that inserts records into DynamoDB.

C.Create an AWS Lambda function that will poll the S3 bucket and then insert the records into DynamoDB.

D.Create a cron job that will run at a scheduled time and insert the records into DynamoDB.

Answer: B

Explanation:

The best approach to automatically insert records into a DynamoDB table upon adding a new file to an S3 bucket is option B: configure an S3 event to invoke an AWS Lambda function that inserts records into DynamoDB.

Here's why: S3 events allow you to trigger actions automatically when specific events occur in your S3 bucket, such as object creation (adding a new file). By configuring an event notification for s3:ObjectCreated:*, you can ensure that whenever a new object is uploaded, the event will be triggered.

AWS Lambda provides serverless compute, allowing you to run code without provisioning or managing servers. You can create a Lambda function that receives the S3 event data (which includes information about the newly added file, such as its name, size, and bucket) and then extract relevant information to construct a record for the DynamoDB table. The Lambda function can then use the AWS SDK to insert this record into the DynamoDB table.

This approach is efficient, scalable, and cost-effective because Lambda only runs when triggered by the S3 event, and you only pay for the compute time used. It is a fully managed, event-driven architecture, which is ideal for this scenario.

Option A (Amazon EventBridge) is less direct. While EventBridge can monitor S3, it adds an unnecessary layer of complexity compared to S3 event notifications. EventBridge is better suited for complex event routing and transformation scenarios, which are not required here.

Option C (Lambda polling S3) is highly inefficient. Polling involves repeatedly checking for new files, which wastes compute resources and introduces latency. Event-driven approaches are always preferred over polling in such scenarios.

Option D (Cron job) is also unsuitable because it relies on a fixed schedule, which means that the record insertion may be delayed until the next scheduled execution. This defeats the requirement of inserting the record as soon as a new file is added.

Therefore, directly configuring an S3 event to trigger a Lambda function is the most efficient, responsive, and cost-effective solution for this use case.

Further research can be done on:

S3 Event Notifications:https://docs.aws.amazon.com/AmazonS3/latest/userguide/EventNotifications.html **AWS Lambda:**https://aws.amazon.com/lambda/
DynamoDB:https://aws.amazon.com/dynamodb/

Question: 36

A development team maintains a web application by using a single AWS CloudFormation template. The template defines web servers and an Amazon RDS database. The team uses the Cloud Formation template to deploy the Cloud Formation stack to different environments.
During a recent application deployment, a developer caused the primary development database to be dropped and recreated. The result of this incident was a loss of data. The team needs to avoid accidental database deletion in the future.
Which solutions will meet these requirements? (Choose two.)

A.Add a CloudFormation Deletion Policy attribute with the Retain value to the database resource.

B.Update the CloudFormation stack policy to prevent updates to the database.

C.Modify the database to use a Multi-AZ deployment.

D.Create a CloudFormation stack set for the web application and database deployments.

E.Add a Cloud Formation DeletionPolicy attribute with the Retain value to the stack.

Answer: AB

Explanation:

The correct answers are A and B. Here's why:

A. Add a CloudFormation DeletionPolicy attribute with the Retain value to the database resource.

The DeletionPolicy attribute in CloudFormation controls what happens to a resource when its stack is deleted or updated in a way that removes the resource. Setting DeletionPolicy: Retain on the database resource ensures that if the CloudFormation stack is deleted or if the database resource is removed from the template during an update, the database itself is not deleted. Instead, it's retained as-is outside of CloudFormation's control. This prevents accidental deletion and data loss.

Concept: CloudFormation resource lifecycle management.
Benefit: Data durability, preventing accidental database removal by CloudFormation.
Relevant Link:https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-attribute-deletionpolicy.html

B. Update the CloudFormation stack policy to prevent updates to the database.

A CloudFormation stack policy is a JSON document that defines the update actions that can be performed on specific resources in a stack. By creating a stack policy that explicitly denies updates to the database resource, the team can prevent any accidental or unintended modifications to the database schema, settings, or deletion via CloudFormation updates. This provides a strong safeguard against modifications of the database via CloudFormation.

Concept: CloudFormation stack policy based governance and control.
Benefit: Prevents accidental database modifications via stack updates.
Relevant Link:https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/protect-stack-resources.html

Why the other options are incorrect:

C. Modify the database to use a Multi-AZ deployment. Multi-AZ deployment enhances database availability by providing a standby replica in a different Availability Zone. It does not prevent accidental database deletion, which is the core issue.
D. Create a CloudFormation stack set for the web application and database deployments. Stack sets facilitate deploying CloudFormation stacks across multiple AWS accounts and regions. While useful for organization and consistency, it doesn't inherently prevent accidental database deletion within a stack. **E. Add a Cloud Formation DeletionPolicy attribute with the Retain value to the stack.** The DeletionPolicy applies to resources within a stack, not the stack itself. Adding the DeletionPolicy to the stack itself won't protect the database resource.

Question: 37

A company has an Amazon S3 bucket that contains sensitive data. The data must be encrypted in transit and at rest. The company encrypts the data in the S3 bucket by using an AWS Key Management Service (AWS KMS) key.
A developer needs to grant several other AWS accounts the permission to use the S3 GetObject operation to retrieve the data from the S3 bucket.
How can the developer enforce that all requests to retrieve the data provide encryption in transit?

A.Define a resource-based policy on the S3 bucket to deny access when a request meets the condition“aws:SecureTransport”: “false”.

B.Define a resource-based policy on the S3 bucket to allow access when a request meets the condition“aws:SecureTransport”: “false”.

C.Define a role-based policy on the other accounts' roles to deny access when a request meets the condition of“aws:SecureTransport”: “false”.

D.Define a resource-based policy on the KMS key to deny access when a request meets the condition of“aws:SecureTransport”: “false”.

Answer: A

Explanation:

The correct answer is A. Here's why:

The requirement is to enforce encryption in transit for all GetObject requests to an S3 bucket containing sensitive data. Encryption in transit means data is encrypted while being transferred between the client and the S3 bucket. HTTPS ensures this encryption using TLS/SSL.

Option A is correct because it leverages an S3 bucket policy (a resource-based policy) to deny access when a request does not use secure transport (HTTPS). The condition "aws:SecureTransport": "false" specifically checks if the request is not using HTTPS. By denying access under this condition, the policy effectively forces all successful requests to use HTTPS, thus fulfilling the requirement of encryption in transit.

Option B is incorrect because allowing access when "aws:SecureTransport": "false" would permit unencrypted traffic, directly contradicting the requirement.

Option C is incorrect because role-based policies are attached to IAM roles assumed by the requesting accounts, not directly to the S3 bucket resource. While you could theoretically manage it through roles, it's more complex and less direct than using a resource-based policy on the S3 bucket itself. Furthermore, this would require coordination and modification across all accounts needing access, making it less maintainable.

Option D is incorrect because the KMS key policy primarily controls who can use the KMS key for encryption and decryption, not whether the data transfer to and from S3 uses encryption in transit (HTTPS). While the data is encrypted at rest using KMS, this is a separate concern from the in-transit encryption enforced by HTTPS. The KMS key policy wouldn't be directly involved in enforcing HTTPS for GetObject requests.

In summary, S3 bucket policies offer a straightforward and centrally managed way to enforce encryption in transit using the aws:SecureTransport condition. By denying non-HTTPS requests, you effectively mandate the use of HTTPS for accessing the S3 bucket's contents.

Authoritative Links:

AWS S3 Bucket Policies:https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-iam-policies.html **AWS Condition Keys for S3:**https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-policy-keys.html **Protecting Data in Transit Using Encryption:**https://docs.aws.amazon.com/whitepapers/latest/aws-security-best-practices/protecting-data-in-transit-using-encryption.html

Question: 38

An application that is hosted on an Amazon EC2 instance needs access to files that are stored in an Amazon S3 bucket. The application lists the objects that are stored in the S3 bucket and displays a table to the user. During testing, a developer discovers that the application does not show any objects in the list.
What is the MOST secure way to resolve this issue?

A.Update the IAM instance profile that is attached to the EC2 instance to include the S3:* permission for the S3 bucket.

B.Update the IAM instance profile that is attached to the EC2 instance to include the S3:ListBucket permission for the S3 bucket.

C.Update the developer's user permissions to include the S3:ListBucket permission for the S3 bucket.

D.Update the S3 bucket policy by including the S3:ListBucket permission and by setting the Principal element to specify the account number of the EC2 instance.

Answer: B

Explanation:

The correct answer is **B: Update the IAM instance profile that is attached to the EC2 instance to include the S3:ListBucket permission for the S3 bucket.**

Here's a detailed justification:

The core issue is the EC2 instance, hosting the application, lacks the necessary permission to list the contents of the S3 bucket. Since the application running on the EC2 instance needs to interact with S3, the most secure and appropriate method is to grant the EC2 instance permissions via an IAM role. This avoids embedding credentials directly within the application or giving unnecessary permissions to individual developers.

Option B precisely addresses this by adding the S3:ListBucket permission to the IAM instance profile associated with the EC2 instance. This permission allows the EC2 instance to list the objects within the specified S3 bucket.

Option A, granting S3:* (all S3 permissions), violates the principle of least privilege. It gives the EC2 instance far more permissions than it needs, increasing the potential attack surface if the EC2 instance is compromised. It's best to grant only the specific permissions required.

Option C, updating the developer's user permissions, is incorrect. The application is running on the EC2 instance, not under the developer's credentials. The application's access to S3 should not rely on the developer's individual permissions.

Option D, updating the S3 bucket policy, is less secure and not the best practice for this scenario. While a bucket policy can grant access, using an IAM role attached to the EC2 instance is generally preferred for resources within the same AWS account because it simplifies management and aligns better with AWS security best practices. Also, the Principal element within the S3 bucket policy would need to be carefully crafted, potentially allowing broader access than intended if not configured correctly. Bucket policies are better suited for cross-account access. Using the EC2 instance role is more streamlined for application-to-S3 communication within the same account.

Therefore, granting the EC2 instance the least privileged permission needed (S3:ListBucket) through an IAM role attached to the instance is the most secure and efficient solution.

Relevant links for further research:

IAM Roles for EC2:https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_ec2.html **IAM Best Practices:**https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html **S3 Permissions:**https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-access-control.html **Principle of Least Privilege:**https://en.wikipedia.org/wiki/Principle_of_least_privilege

Question: 40

A company has deployed infrastructure on AWS. A development team wants to create an AWS Lambda function that will retrieve data from an Amazon Aurora database. The Amazon Aurora database is in a private subnet in company's VPC. The VPC is named VPC1. The data is relational in nature. The Lambda function needs to access the data securely.
Which solution will meet these requirements?

A.Create the Lambda function. Configure VPC1 access for the function. Attach a security group named SG1 to both the Lambda function and the database. Configure the security group inbound and outbound rules to allow TCP traffic on Port 3306.

B.Create and launch a Lambda function in a new public subnet that is in a new VPC named VPC2. Create a peering connection between VPC1 and VPC2.

C.Create the Lambda function. Configure VPC1 access for the function. Assign a security group named SG1 to the Lambda function. Assign a second security group named SG2 to the database. Add an inbound rule to SG1 to allow TCP traffic from Port 3306.

D.Export the data from the Aurora database to Amazon S3. Create and launch a Lambda function in VPC1. Configure the Lambda function query the data from Amazon S3.

Answer: A

Explanation:

Here's a detailed justification for why option A is the most suitable solution, along with explanations of why the other options are less ideal:

Why Option A is the Best Choice:

Option A provides a secure and efficient method for a Lambda function to access an Aurora database within a VPC:

1. **VPC Integration:** Placing the Lambda function within the same VPC (VPC1) as the Aurora database allows it to communicate directly without traversing the internet or requiring public IP addresses. This significantly enhances security by keeping traffic within the private network.

2. **Security Groups:** Security groups act as virtual firewalls, controlling inbound and outbound traffic at the instance level. By attaching a security group (SG1) to both the Lambda function and the Aurora database, you can precisely control the allowed traffic between them.

3. **Port 3306 (MySQL/Aurora Standard Port):** Configuring the security group to allow TCP traffic on port 3306 (the standard port for MySQL and Aurora) enables the Lambda function to connect to the database server.

4. **Least Privilege:** This solution allows you to restrict access to only the necessary port (3306) and only between the Lambda function and the database. This follows the principle of least privilege, minimizing the attack surface.

5. **Avoiding unnecessary complexity:** This solution avoids unnecessary network configurations (peering connections in Option B) or data movement (exporting data to S3 in Option D).

In summary, Option A leverages VPC integration and security groups to establish a secure and direct connection between the Lambda function and the Aurora database, adhering to best practices for security and network design.

Why Other Options are Less Suitable:

Option B (Peering and Public Subnet): Creating a new VPC (VPC2) with a public subnet and then peering it with VPC1 adds unnecessary complexity and potential security risks. Public subnets expose instances to the internet, which is undesirable for database access. Peering introduces additional configuration overhead.

Option C (Inbound Rule on Lambda SG1 only): Option C has critical security flaw. You should not configure inbound rules on SG1. Inbound rules are for connections coming IN to the resource, and outbound rules are for traffic initiated from the resource. Therefore, database SG2 needs to have the inbound rule that accepts traffic on port 3306 from the lambda's SG1.

Option D (Export to S3): Exporting the data to S3 introduces unnecessary complexity, latency, and cost. It also undermines the real-time nature of relational data. Reading from S3 is generally slower and less efficient than querying a database directly for relational data scenarios. It changes the type of workload to batch, which does not fulfill the requirement of the Lambda function retrieving data from the Aurora database.

Supporting Documentation:

AWS Lambda VPC Configuration:https://docs.aws.amazon.com/lambda/latest/dg/services-vpc.html **Amazon VPC Security Groups:**https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html **Connecting Lambda to RDS:**https://aws.amazon.com/blogs/compute/connecting-aws-lambda-functions-to-amazon-rds-using-data-api/ (Although this example uses Data API, the principles of VPC and security group configuration are the same.)

Question: 41

A developer is building a web application that uses Amazon API Gateway to expose an AWS Lambda function to process requests from clients. During testing, the developer notices that the API Gateway times out even though the Lambda function finishes under the set time limit.
Which of the following API Gateway metrics in Amazon CloudWatch can help the developer troubleshoot the issue? (Choose two.)

A.CacheHitCount

B.IntegrationLatency

C.CacheMissCount

D.Latency

E.Count

Answer: BD

Explanation:

The problem arises when the API Gateway times out despite the Lambda function completing within its allocated time. To effectively troubleshoot this, we need to investigate metrics related to the API Gateway's interaction with the backend Lambda function and the overall request processing time.

B. IntegrationLatency: This metric measures the time it takes for API Gateway to send a request to the backend Lambda function and receive a response. A high IntegrationLatency indicates a bottleneck or delay in the communication between the API Gateway and the Lambda function. This could be due to network issues, Lambda function cold starts, or the Lambda function's processing time nearing its limit. If IntegrationLatency is high, even though the Lambda function completes within its overall configured limit, the interaction time plus the Lambda execution time might be exceeding the API Gateway's timeout configuration.

D. Latency: This metric represents the total time taken for API Gateway to process a request, from the moment it receives the request to the moment it sends the response back to the client. A high Latency value suggests a problem within the API Gateway itself or in its interaction with the backend. By comparing Latency with IntegrationLatency, the developer can discern if the bottleneck lies within the API Gateway or in its communication with the Lambda function. If the difference between Latency and IntegrationLatency is significant, it indicates processing delays within the API Gateway, such as authorization or transformation steps.

Why other options are less relevant:

A. CacheHitCount and C. CacheMissCount: These metrics pertain to API Gateway's caching functionality. Since the problem arises even when the Lambda function completes within its time limit, the caching behavior is unlikely to be the root cause of the timeout. Caching is typically used to reduce Lambda invocations and improve response times for frequently accessed data. However, timeouts often indicate a fundamental issue with the API Gateway's processing or interaction with the backend service, not how frequently cache is hit or missed.

E. Count: This metric indicates the total number of requests received by API Gateway. While useful for understanding traffic volume, it does not provide insights into the performance issues causing the timeout. A high request count doesn't inherently explain why a request is timing out, only that many requests are being processed. It will not help in diagnosing the latency issue.

In summary, IntegrationLatency and Latency provide the most relevant information to pinpoint the cause of the API Gateway timeout. These metrics allow the developer to identify whether the delay occurs during the integration with the Lambda function or within the API Gateway itself.

Supporting Documentation:

Monitoring API Gateway API execution with Amazon CloudWatch metrics

Question: 42

A development team wants to build a continuous integration/continuous delivery (CI/CD) pipeline. The team is using AWS CodePipeline to automate the code build and deployment. The team wants to store the program code to prepare for the CI/CD pipeline.
Which AWS service should the team use to store the program code?

A.AWS CodeDeploy
B.AWS CodeArtifact
C.AWS CodeCommit
D.Amazon CodeGuru

Answer: C

Explanation:

The correct answer is C, AWS CodeCommit.

Here's why: A CI/CD pipeline starts with the code repository. AWS CodeCommit is a fully managed source control service that hosts secure Git-based repositories. It allows the team to privately store and version their code. CodeCommit integrates directly with AWS CodePipeline, making it a natural choice for storing the application code in this scenario. CodePipeline can be configured to automatically trigger builds and deployments whenever changes are pushed to the CodeCommit repository.

AWS CodeDeploy (A) is a deployment service, not a code repository. It automates the deployment of applications to various compute services, but it doesn't store the code itself. AWS CodeArtifact (B) is a package repository service used for storing and sharing software packages like dependencies (e.g., Maven artifacts, npm packages). While it's related to development, it's not the primary place to store the application's source code itself. Amazon CodeGuru (D) is a code review and performance profiling service; it analyzes existing code but doesn't store it.

Therefore, CodeCommit is the logical first step in creating the CI/CD pipeline using CodePipeline to automate the code build and deployment. It addresses the requirement to store program code, serving as the source for the pipeline.

Further research:

AWS CodeCommit: https://aws.amazon.com/codecommit/
AWS CodePipeline: https://aws.amazon.com/codepipeline/

Question: 43

A developer is designing an AWS Lambda function that creates temporary files that are less than 10 MB during invocation. The temporary files will be accessed and modified multiple times during invocation. The developer has no need to save or retrieve these files in the future.

Where should the temporary files be stored?

A.the /tmp directory
B.Amazon Elastic File System (Amazon EFS)
C.Amazon Elastic Block Store (Amazon EBS)
D.Amazon S3

Answer: A

Explanation:

The correct location for storing temporary files within an AWS Lambda function, specifically those smaller than 10 MB that don't require persistent storage, is the /tmp directory.

Here's why:

Ephemeral Storage: Lambda provides 512 MB of ephemeral disk space in the /tmp directory. This storage is only available during the invocation of the function. Once the function execution completes, the /tmp directory's contents are discarded.

Performance: Accessing files within the /tmp directory is significantly faster compared to using external storage services like Amazon S3 or Amazon EFS. The files are stored locally on the Lambda execution environment.

Cost-Effective: Using /tmp is free and does not incur additional charges. The cost is implicitly included in the Lambda execution cost, based on function duration and memory allocation.

Appropriate Use Case: The question states the temporary files are created and accessed multiple times during the invocation and have no need to be stored or retrieved. This perfectly aligns with the use of /tmp. **Amazon S3 Inappropriate:** Amazon S3 is designed for durable, object-based storage. Using S3 for temporary files would introduce unnecessary overhead (network latency, API calls, cost) since the files are only needed during the function execution.

Amazon EFS Inappropriate: Amazon EFS is a network file system ideal for shared persistent storage. Similar to S3, it's not designed for ephemeral, short-lived files generated during a single Lambda invocation. It would also introduce unnecessary network latency and cost.

Amazon EBS Inappropriate: Amazon EBS volumes are block storage devices typically attached to EC2 instances. They aren't directly compatible with Lambda functions in the same way as /tmp and adding them would involve additional configuration and complexity.

Therefore, the /tmp directory provides the optimal balance of performance, cost, and convenience for storing temporary files within the described Lambda function scenario.

Further reading:

Working with AWS Lambda layers and extensions in container images: https://aws.amazon.com/blogs/compute/working-with-aws-lambda-layers-and-extensions-in-container-images/ (discusses temporary file storage in container-based Lambda functions, which also use /tmp) **AWS Lambda execution environment:**https://docs.aws.amazon.com/lambda/latest/dg/lambda-environment-variables.html (mentions the /tmp directory and its limitations)

Question: 44

A developer is designing a serverless application with two AWS Lambda functions to process photos. One Lambda function stores objects in an Amazon S3 bucket and stores the associated metadata in an Amazon DynamoDB table. The other Lambda function fetches the objects from the S3 bucket by using the metadata from the DynamoDB table. Both Lambda functions use the same Python library to perform complex computations and are approaching the quota for the maximum size of zipped deployment packages.

What should the developer do to reduce the size of the Lambda deployment packages with the LEAST operational overhead?

A.Package each Python library in its own .zip file archive. Deploy each Lambda function with its own copy of the library.
B.Create a Lambda layer with the required Python library. Use the Lambda layer in both Lambda functions.
C.Combine the two Lambda functions into one Lambda function. Deploy the Lambda function as a single .zip file archive.
D.Download the Python library to an S3 bucket. Program the Lambda functions to reference the object URLs.

Answer: B

Explanation:

The correct answer is B: Create a Lambda layer with the required Python library. Use the Lambda layer in both Lambda functions.

Here's a detailed justification:

Lambda layers are a mechanism to package dependencies, such as libraries, separately from the Lambda function code. This addresses the problem of exceeding deployment package size limits, which is precisely the scenario described. Lambda functions are limited in the size of their deployment packages and uncompressed size after deployment. When multiple Lambda functions share the same dependency (in this case, the Python library), packaging that dependency separately as a layer avoids redundant duplication of the library in each function's deployment package.

Option B provides the least operational overhead for several reasons. Unlike option A, it avoids maintaining separate copies of the library. Option C, combining the Lambda functions, might not be feasible or desirable if the functions serve distinct purposes or need to scale independently. It also changes the architecture substantially. Option D introduces significant operational overhead and complexity because the Lambda functions would need to download the library from S3 every time they're invoked. This adds latency and increases the risk of failure if the S3 bucket is unavailable. Moreover, that creates unnecessary
ingress/egress costs to S3 on every invocation. Lambda layers, on the other hand, are readily available to Lambda functions and don't add overhead to each invocation's execution time once deployed.

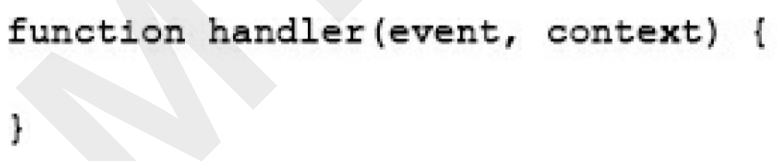
Using Lambda layers offers several benefits, including reduced deployment package sizes, faster deployment times, code reusability, and simplified dependency management. Layers are versioned, which allows you to update libraries and easily roll back to previous versions if needed. This provides control and resilience in your serverless application deployments. The operational overhead is minimal as AWS manages the layer's availability and integrity. By leveraging Lambda layers, the developer can efficiently address the size constraints while maintaining a clean and maintainable architecture.

For further research, refer to the AWS Lambda documentation:

AWS Lambda Layers:https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html

Question: 45

A developer is writing an AWS Lambda function. The developer wants to log key events that occur while the Lambda function runs. The developer wants to include a unique identifier to associate the events with a specific function invocation. The developer adds the following code to the Lambda function:



```
function handler(event, context) {

}
```

Which solution will meet this requirement?

A.Obtain the request identifier from the AWS request ID field in the context object. Configure the application to write logs to standard output.

B.Obtain the request identifier from the AWS request ID field in the event object. Configure the application to write logs to a file.

C.Obtain the request identifier from the AWS request ID field in the event object. Configure the application to write logs to standard output.

D.Obtain the request identifier from the AWS request ID field in the context object. Configure the application to write logs to a file.

Answer: A

Explanation:

Ahttps://docs.aws.amazon.com/lambda/latest/dg/nodejs-context.htmlhttps://docs.aws.amazon.com/lambda/latest/dg/nodejs-logging.htmlThere is no explicit information for the runtime, the code is written in Node.js.

Both A and D could work here, as both rely on the context object to get access to execution ID https://docs.aws.amazon.com/us_en/lambda/latest/dg/python-context.html While A uses stoud to send log to Cloud Watch Log, D writes to a file. D is less specific (where is the file stored? A single file for each execution?) and looks more complex (manage file(s), manage concurrency access to the file .), thus I'll go for A

Question: 46

A developer is working on a serverless application that needs to process any changes to an Amazon DynamoDB table with an AWS Lambda function.
How should the developer configure the Lambda function to detect changes to the DynamoDB table?

A.Create an Amazon Kinesis data stream, and attach it to the DynamoDB table. Create a trigger to connect the data stream to the Lambda function.

B.Create an Amazon EventBridge rule to invoke the Lambda function on a regular schedule. Conned to the DynamoDB table from the Lambda function to detect changes.

C.Enable DynamoDB Streams on the table. Create a trigger to connect the DynamoDB stream to the Lambda function.

D.Create an Amazon Kinesis Data Firehose delivery stream, and attach it to the DynamoDB table. Configure the delivery stream destination as the Lambda function.

Answer: C

Explanation:

The correct answer is C because it utilizes DynamoDB Streams, which is the direct and intended mechanism for capturing changes to a DynamoDB table and triggering a Lambda function. DynamoDB Streams records all data modifications (inserts, updates, deletes) to a DynamoDB table in near real-time and in the order they occurred. Enabling DynamoDB Streams automatically captures these events.

A Lambda function can then be configured as a trigger for the DynamoDB stream. When a change occurs in the DynamoDB table, the stream receives the event, and the trigger invokes the Lambda function, passing the event data as input. This enables the Lambda function to process the changes asynchronously and in response to real-time updates.

Option A is incorrect because using Kinesis Data Streams is not the standard method for capturing DynamoDB changes. While technically possible, it would require more complex and custom configuration, like writing to the Kinesis stream from application code, and isn't the intended use case. DynamoDB Streams provides a managed and integrated solution that's much simpler and more efficient.

Option B is incorrect because polling the DynamoDB table from the Lambda function on a schedule is inefficient, costly, and not real-time. It would consume unnecessary read capacity units and might miss changes if the polling interval is too long. EventBridge is generally used for schedule-based and event-driven invocations from other AWS services and applications, but it’s not a direct way to capture DynamoDB changes.

Option D is incorrect because Kinesis Data Firehose is primarily designed for loading streaming data into data lakes, data warehouses, and analytics services, such as Amazon S3, Amazon Redshift, and Splunk. It is not intended for real-time processing of DynamoDB changes by Lambda functions. Data Firehose focuses on batch delivery of data and doesn't inherently trigger functions for each individual change.

In summary, using DynamoDB Streams and a Lambda trigger provides the most efficient, cost-effective, and real-time method for processing DynamoDB changes with a serverless Lambda function.

Authoritative Links:

DynamoDB Streams:https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html **Using AWS Lambda with DynamoDB:**https://docs.aws.amazon.com/lambda/latest/dg/with-ddb.html

Question: 47

An application uses an Amazon EC2 Auto Scaling group. A developer notices that EC2 instances are taking a long time to become available during scale-out events. The UserData script is taking a long time to run.
The developer must implement a solution to decrease the time that elapses before an EC2 instance becomes available. The solution must make the most recent version of the application available at all times and must apply all available security updates. The solution also must minimize the number of images that are created. The images must be validated.
Which combination of steps should the developer take to meet these requirements? (Choose two.)

A.Use EC2 Image Builder to create an Amazon Machine Image (AMI). Install all the patches and agents that are needed to manage and run the application. Update the Auto Scaling group launch configuration to use the AMI.

B.Use EC2 Image Builder to create an Amazon Machine Image (AMI). Install the latest version of the application and all the patches and agents that are needed to manage and run the application. Update the Auto Scaling group launch configuration to use the AMI.

C.Set up AWS CodeDeploy to deploy the most recent version of the application at runtime.

D.Set up AWS CodePipeline to deploy the most recent version of the application at runtime.

E.Remove any commands that perform operating system patching from the UserData script.

Answer: AE

Explanation:

The problem is slow EC2 instance availability during Auto Scaling group scale-out events due to a lengthy UserData script. The goal is to minimize instance launch time, always have the latest application version, apply security updates, minimize image creation, and validate images.

Option A is correct because EC2 Image Builder automates the creation, patching, and testing of AMIs. By pre-installing all necessary agents and patches, instances boot ready for application deployment, significantly reducing UserData script execution time. This addresses the slow boot time problem directly.

Option E is correct because it complements option A. Moving OS patching out of the UserData script aligns with the goal of reducing the script's execution time. Instead, OS patching is handled by EC2 Image Builder when creating the AMI, ensuring instances start with the latest security updates without delaying the boot process.

Option B is similar to A but specifies including the latest version of the application in the AMI. While seemingly beneficial, baking the application into the AMI increases the frequency of AMI rebuilds to stay up-to-date, violating the requirement to minimize image creation. It also doesn't address how the latest version is determined and incorporated into the AMI build process.

Option C (CodeDeploy) is incorrect because, while it deploys the latest application version, it runs at runtime (i.e., after the instance is launched). This still contributes to the overall time it takes for an instance to become available, conflicting with the requirement to minimize instance launch time.

Option D (CodePipeline) is incorrect for similar reasons to C. CodePipeline is a CI/CD service that can automate application deployment, but it would still result in deployment at runtime, which is slower than having the instances pre-configured with an updated AMI.

Therefore, using EC2 Image Builder to create pre-baked AMIs with agents, patches (A), and moving patching out of the UserData script (E) effectively addresses the problem by minimizing UserData execution time and ensures validated, secure instances with minimal image creation.

Relevant Documentation:

EC2 Image Builder:https://aws.amazon.com/image-builder/
Auto Scaling Groups:https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html

Question: 48

A developer is creating an AWS Lambda function that needs credentials to connect to an Amazon RDS for MySQL database. An Amazon S3 bucket currently stores the credentials. The developer needs to improve the existing solution by implementing credential rotation and secure storage. The developer also needs to provide integration with the Lambda function.
Which solution should the developer use to store and retrieve the credentials with the LEAST management overhead?

A.Store the credentials in AWS Systems Manager Parameter Store. Select the database that the parameter will access. Use the default AWS Key Management Service (AWS KMS) key to encrypt the parameter. Enable automatic rotation for the parameter. Use the parameter from Parameter Store on the Lambda function to connect to the database.

B.Encrypt the credentials with the default AWS Key Management Service (AWS KMS) key. Store the credentials as environment variables for the Lambda function. Create a second Lambda function to generate new credentials and to rotate the credentials by updating the environment variables of the first Lambda function. Invoke the second Lambda function by using an Amazon EventBridge rule that runs on a schedule. Update the database to use the new credentials. On the first Lambda function, retrieve the credentials from the environment variables. Decrypt the credentials by using AWS KMS, Connect to the database.

C.Store the credentials in AWS Secrets Manager. Set the secret type to Credentials for Amazon RDS database. Select the database that the secret will access. Use the default AWS Key Management Service (AWS KMS) key to encrypt the secret. Enable automatic rotation for the secret. Use the secret from Secrets Manager on the Lambda function to connect to the database.

D.Encrypt the credentials by using AWS Key Management Service (AWS KMS). Store the credentials in an Amazon DynamoDB table. Create a second Lambda function to rotate the credentials. Invoke the second Lambda function by using an Amazon EventBridge rule that runs on a schedule. Update the DynamoDB table. Update the database to use the generated credentials. Retrieve the credentials from DynamoDB with the first Lambda function. Connect to the database.

Answer: C

Explanation:

Here's a detailed justification for why option C is the best solution, and why the other options are less optimal, considering the requirements of secure storage, credential rotation, minimal management overhead, and easy integration with a Lambda function:

Why Option C (AWS Secrets Manager) is the Best Solution:

Purpose-Built for Credentials: AWS Secrets Manager is specifically designed for securely storing and managing sensitive information like database credentials. It offers built-in features that simplify the process compared to generic storage solutions.

Automatic Rotation: Secrets Manager allows automated rotation of database credentials without requiring custom code. This drastically reduces the management overhead and improves security posture by regularly changing credentials.

Easy Integration: Secrets Manager has direct integration with Lambda functions. You can retrieve secrets with a simple API call, which reduces code complexity and improves security by centralizing credential management.

Encryption: Secrets Manager encrypts secrets at rest and in transit using AWS KMS. This provides an additional layer of security. The default KMS key simplifies setup, but you can use your own CMK for more control.

RDS Integration: Secrets Manager has a specific "Credentials for Amazon RDS database" secret type, which simplifies the configuration and management of RDS credentials, making the integration seamless.

Why Other Options Are Suboptimal:

Option A (Systems Manager Parameter Store): Parameter Store is a good general-purpose storage, but not specialized for credentials management or rotation. While secure strings can be encrypted, automatic rotation requires custom scripting and management, which increases overhead.

Option B (Lambda Function with KMS and Environment Variables): Storing credentials as environment variables is generally discouraged due to potential exposure through logs or function introspection. Creating a separate Lambda function for rotation adds unnecessary complexity.

Option D (Lambda Function with KMS and DynamoDB): Similar to option B, this approach involves more manual configuration and code for rotation and retrieval. Managing credentials in DynamoDB also lacks the specific security and rotation features offered by Secrets Manager.

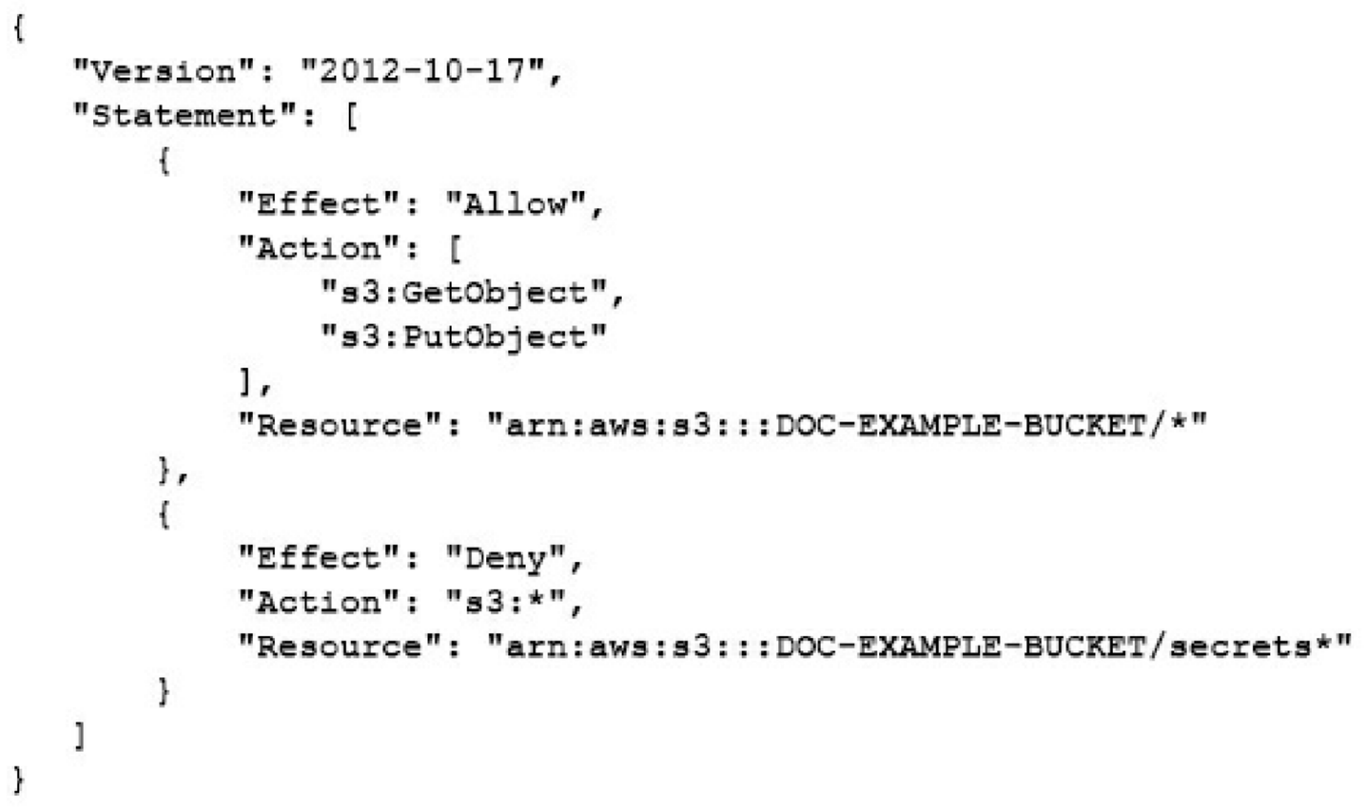
In summary: AWS Secrets Manager offers a complete, integrated solution for securely storing, retrieving, and rotating database credentials, with minimal management overhead. This makes it the best choice for the given scenario.

Authoritative Links:

AWS Secrets Manager:https://aws.amazon.com/secrets-manager/
Rotating AWS Secrets Manager secrets:
https://docs.aws.amazon.com/secretsmanager/latest/userguide/rotating-secrets.html
Accessing AWS Secrets Manager Secrets from AWS Lambda:
https://aws.amazon.com/blogs/security/access-secrets-stored-in-aws-secrets-manager-from-aws-lambda/

Question: 49

A developer has written the following IAM policy to provide access to an Amazon S3 bucket:



```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
        },
        {
            "Effect": "Deny",
            "Action": "s3:*",
            "Resource": "arn:aws:s3:::DOC-EXAMPLE-BUCKET/secrets*"
        }
    ]
}
```

Which access does the policy allow regarding the s3:GetObject and s3:PutObject actions?

A.Access on all buckets except the "DOC-EXAMPLE-BUCKET" bucket
B.Access on all buckets that start with "DOC-EXAMPLE-BUCKET" except the "DOC-EXAMPLE-BUCKET/secrets" bucket
C.Access on all objects in the "DOC-EXAMPLE-BUCKET" bucket along with access to all S3 actions for objects in the "DOC-EXAMPLE-BUCKET" bucket that start with "secrets"
D.Access on all objects in the "DOC-EXAMPLE-BUCKET" bucket except on objects that start with "secrets"

Answer: D

Explanation:

Access on all objects in the "DOC-EXAMPLE-BUCKET" bucket except on objects that start with "secrets"

Reference:

https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-with-s3-actions.html

Question: 50

A developer is creating a mobile app that calls a backend service by using an Amazon API Gateway REST API. For integration testing during the development phase, the developer wants to simulate different backend responses without invoking the backend service.
Which solution will meet these requirements with the LEAST operational overhead?

A.Create an AWS Lambda function. Use API Gateway proxy integration to return constant HTTP responses. B.Create an Amazon EC2 instance that serves the backend REST API by using an AWS CloudFormation template.
C.Customize the API Gateway stage to select a response type based on the request.
D.Use a request mapping template to select the mock integration response.

Answer: D

Explanation:

The most efficient solution is to utilize API Gateway's request mapping templates to generate mock integration responses (Option D). Request mapping templates allow you to transform the incoming request and construct a static response directly within API Gateway, bypassing the backend service entirely. This avoids the operational overhead associated with managing a Lambda function (Option A) or a full-fledged EC2 instance (Option B) just for simulation purposes. Customizing the API Gateway stage (Option C) is not the correct approach. Stages are meant to manage different deployment environments, not simulate responses.

Request mapping templates are serverless and easy to configure. You define the desired HTTP response code, headers, and body directly within the template. API Gateway handles serving the response based on the mapping, without invoking any downstream service. This significantly reduces latency compared to proxying to even a simple Lambda function. Furthermore, this solution is more cost-effective because it eliminates Lambda execution costs or EC2 instance costs incurred by Option A and B, respectively. It's a configuration-driven approach, making it easier to manage and update the mock responses as needed during testing. The learning curve for mapping templates is relatively low and can be reused for several API calls making this the superior solution. Because option D provides the means to directly craft a defined response in line with the testing needs of the developer, without additional backend services, it holds the least operational overhead.

For further research, refer to the official AWS documentation:

API Gateway Mapping Templates
API Gateway Integration Types

Question: 51

A developer has a legacy application that is hosted on-premises. Other applications hosted on AWS depend on the on-premises application for proper functioning. In case of any application errors, the developer wants to be able to use Amazon CloudWatch to monitor and troubleshoot all applications from one place.
How can the developer accomplish this?

A.Install an AWS SDK on the on-premises server to automatically send logs to CloudWatch.

B.Download the CloudWatch agent to the on-premises server. Configure the agent to use IAM user credentials with permissions for CloudWatch.

C.Upload log files from the on-premises server to Amazon S3 and have CloudWatch read the files.

D.Upload log files from the on-premises server to an Amazon EC2 instance and have the instance forward the logs to CloudWatch.

Answer: B

Explanation:

The correct answer is B because it provides the most direct and efficient way to send logs from an on-premises server to CloudWatch. The CloudWatch agent is specifically designed for this purpose, allowing collection of log data from various sources, including on-premises servers, and streaming it to CloudWatch Logs.

Option B suggests downloading the CloudWatch agent onto the on-premises server. This agent, once configured, monitors the specified log files and transmits the data to CloudWatch Logs. The agent's configuration involves setting up IAM user credentials with permissions to write to CloudWatch. This provides a secure way to authenticate and authorize the on-premises server to send log data to AWS.

Option A is incorrect because while AWS SDKs can interact with CloudWatch, they are more commonly used for application-level interactions rather than system-level log forwarding. Configuring an SDK to actively monitor and send logs would require custom scripting and development, which is less efficient than using the dedicated CloudWatch agent.

Option C is not ideal because uploading log files to Amazon S3 and then having CloudWatch read them introduces unnecessary complexity and delay. It requires writing code to periodically upload logs to S3 and configuring CloudWatch to poll S3. This is less real-time and more resource-intensive than using the CloudWatch agent.

Option D is less efficient than option B, because it requires spinning up an EC2 instance simply to act as an intermediary for forwarding logs. This increases operational overhead and cost compared to using the CloudWatch agent directly on the on-premises server.

The CloudWatch agent solution efficiently addresses the need for centralized monitoring by bringing on-premises logs into AWS. This eliminates the need to manually manage and upload files, and provides a single pane of glass for monitoring both on-premises and cloud-based applications, aiding in proactive problem identification and resolution.

For more information, refer to the official AWS documentation:

CloudWatch Agent:https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Install-CloudWatch-Agent.html

Question: 52

An Amazon Kinesis Data Firehose delivery stream is receiving customer data that contains personally identifiable information. A developer needs to remove pattern-based customer identifiers from the data and store the modified data in an Amazon S3 bucket.
What should the developer do to meet these requirements?

A.Implement Kinesis Data Firehose data transformation as an AWS Lambda function. Configure the function to remove the customer identifiers. Set an Amazon S3 bucket as the destination of the delivery stream.

B.Launch an Amazon EC2 instance. Set the EC2 instance as the destination of the delivery stream. Run an application on the EC2 instance to remove the customer identifiers. Store the transformed data in an Amazon S3 bucket.

C.Create an Amazon OpenSearch Service instance. Set the OpenSearch Service instance as the destination of the delivery stream. Use search and replace to remove the customer identifiers. Export the data to an Amazon S3 bucket.

D.Create an AWS Step Functions workflow to remove the customer identifiers. As the last step in the workflow, store the transformed data in an Amazon S3 bucket. Set the workflow as the destination of the delivery stream.

Answer: A

Explanation:

The correct answer is A because it leverages Kinesis Data Firehose's built-in data transformation capabilities using AWS Lambda, providing an efficient and cost-effective solution for removing PII before data lands in S3.
Kinesis Data Firehose allows for real-time data transformation using Lambda functions, enabling the scrubbing of sensitive information like customer identifiers directly within the data stream.

Option A directly addresses the problem by integrating data transformation into the Firehose delivery pipeline. The Lambda function can be configured to use regular expressions or other pattern-matching techniques to identify and remove the personally identifiable information (PII) before the data is delivered to S3. By doing this, the PII never persists in S3, fulfilling the requirement.

Option B, using an EC2 instance, introduces unnecessary complexity and operational overhead. Managing an EC2 instance for data transformation requires managing the instance itself, the application running on it, and ensuring its availability and scalability. This approach is more expensive and less efficient than using Lambda.

Option C, using OpenSearch Service, is not ideal for this scenario. OpenSearch Service is designed for search and analytics, not primarily for data transformation. While it could be used for this purpose, it would be an over-engineered solution. Moreover, exporting data from OpenSearch to S3 adds an extra step and potential latency.

Option D, using AWS Step Functions, is also an over-engineered solution. Step Functions are designed for orchestrating complex workflows, but in this case, the data transformation is relatively simple and can be handled more efficiently by a Lambda function within the Kinesis Data Firehose pipeline. Using Step Functions would introduce unnecessary complexity and cost.

Therefore, using a Lambda function within Kinesis Data Firehose, as outlined in option A, is the most efficient, cost-effective, and scalable solution for removing PII from the data before storing it in S3. This aligns with the best practices for data security and privacy in cloud computing.

Further research:

AWS Kinesis Data Firehose Data Transformation:https://docs.aws.amazon.com/firehose/latest/dev/data-transformation.html
AWS Lambda:https://aws.amazon.com/lambda/

Question: 53

A developer is using an AWS Lambda function to generate avatars for profile pictures that are uploaded to an Amazon S3 bucket. The Lambda function is automatically invoked for profile pictures that are saved under the /original/ S3 prefix. The developer notices that some pictures cause the Lambda function to time out. The developer wants to implement a fallback mechanism by using another Lambda function that resizes the profile picture.
Which solution will meet these requirements with the LEAST development effort?

A.Set the image resize Lambda function as a destination of the avatar generator Lambda function for the events that fail processing.

B.Create an Amazon Simple Queue Service (Amazon SQS) queue. Set the SQS queue as a destination with an on failure condition for the avatar generator Lambda function. Configure the image resize Lambda function to poll from the SQS queue.

C.Create an AWS Step Functions state machine that invokes the avatar generator Lambda function and uses the image resize Lambda function as a fallback. Create an Amazon EventBridge rule that matches events from the S3 bucket to invoke the state machine.

D.Create an Amazon Simple Notification Service (Amazon SNS) topic. Set the SNS topic as a destination with an on failure condition for the avatar generator Lambda function. Subscribe the image resize Lambda function to the SNS topic.

Answer: A

Explanation:

The best solution is A: Set the image resize Lambda function as a destination of the avatar generator Lambda function for events that fail processing.

Here's why:

Direct Integration & Least Effort: Lambda destinations provide a direct integration between functions. If the avatar generator Lambda function fails (times out or throws an error), the destination function (image resizer) is automatically invoked. This minimizes development effort as it avoids managing queues or state machines.

Built-in Retry Mechanism: Lambda destinations include a built-in retry mechanism that can be configured, giving the avatar generator function additional chances to succeed before invoking the fallback.

Automatic Error Handling: Lambda destinations provide a streamlined error handling mechanism. You don't need to write custom code to detect failures and trigger the fallback, as Lambda handles this automatically. **No Polling Overhead:** Option B requires the image resize function to continuously poll an SQS queue, adding overhead and potential latency.

Complexity Avoidance: Option C introduces a Step Functions state machine, adding unnecessary complexity for a simple fallback scenario. While Step Functions are powerful, they are overkill for this specific requirement.

SNS Fanout Avoidance: Option D utilizes SNS, which introduces a fanout pattern unnecessary for this scenario. SNS is better suited for broadcasting messages to multiple subscribers, not a single fallback function.

Lambda Destinations on Failure provides the most straightforward, efficient, and least effort approach for implementing a fallback mechanism for Lambda function failures.

Relevant Links:

AWS Lambda Destinations: https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#dlq AWS Lambda Error Handling: https://docs.aws.amazon.com/lambda/latest/dg/services-sns.html

Question: 54

A developer needs to migrate an online retail application to AWS to handle an anticipated increase in traffic. The application currently runs on two servers: one server for the web application and another server for the database. The web server renders webpages and manages session state in memory. The database server hosts a MySQL database that contains order details. When traffic to the application is heavy, the memory usage for the web server approaches 100% and the application slows down considerably.
The developer has found that most of the memory increase and performance decrease is related to the load of managing additional user sessions. For the web server migration, the developer will use Amazon EC2 instances with an Auto Scaling group behind an Application Load Balancer.
Which additional set of changes should the developer make to the application to improve the application's performance?

A.Use an EC2 instance to host the MySQL database. Store the session data and the application data in the MySQL database.

B.Use Amazon ElastiCache for Memcached to store and manage the session data. Use an Amazon RDS for MySQL DB instance to store the application data.

C.Use Amazon ElastiCache for Memcached to store and manage the session data and the application data.

D.Use the EC2 instance store to manage the session data. Use an Amazon RDS for MySQL DB instance to store the application data.

Answer: B

Explanation:

The correct answer is **B. Use Amazon ElastiCache for Memcached to store and manage the session data. Use an Amazon RDS for MySQL DB instance to store the application data.**

Here's a detailed justification:

The problem highlights that the web server's performance degrades due to memory pressure caused by session management. Storing session data in the web server's memory (in-memory session management) does not scale horizontally. Each web server needs its own copy of session data, and changes to a session are not automatically propagated across servers. This leads to inconsistent user experiences and inefficient resource utilization when scaling with an Auto Scaling group.

Offloading session management to a dedicated, scalable solution is crucial. Amazon ElastiCache for Memcached is designed for caching frequently accessed data, including session data. It offers high performance and low latency, alleviating the memory burden on the web servers. By using ElastiCache, the web servers in the Auto Scaling group can quickly retrieve and update session information without being constrained by local memory limits. This architecture facilitates horizontal scalability, allowing the application to handle increased traffic effectively.

Regarding application data (order details), using Amazon RDS for MySQL is a best practice. RDS provides managed database services with features like automated backups, patching, and scalability. It ensures the application data remains durable and readily available. Separating the application data from the session data allows you to use services optimized for each type of data.

Option A is incorrect because storing session data in the same MySQL database as the application data would likely create performance bottlenecks. MySQL is designed for persistent data storage and not for the low-latency requirements of session management. Placing both workloads on the same database instance increases the database's workload and can cause performance issues. Furthermore, using a single EC2 instance to host the MySQL database does not provide the scalability and availability benefits of RDS.

Option C is incorrect because Amazon ElastiCache for Memcached is not the right choice for persistent data. ElastiCache is optimized for caching and would not work well as a primary durable datastore. It's volatile; data can be lost. Order details should be stored in a persistent database like RDS.

Option D is incorrect because EC2 instance store is ephemeral (non-persistent). If the EC2 instance fails, the data stored in the instance store is lost. Also, managing session data locally on each instance, even with the instance store, doesn't scale effectively and negates the benefit of the Auto Scaling group.

Therefore, the best approach is to separate session data from application data, using a dedicated, scalable caching service (ElastiCache) for session management and a managed, persistent database service (RDS) for application data. This strategy addresses the memory pressure on the web servers and facilitates horizontal scalability.

Relevant links for further research:

Amazon ElastiCache:https://aws.amazon.com/elasticache/
Amazon RDS:https://aws.amazon.com/rds/
Application Load Balancer:https://aws.amazon.com/elasticloadbalancing/application-load-balancer/ **Auto Scaling:**https://aws.amazon.com/autoscaling/

Question: 55

An application uses Lambda functions to extract metadata from files uploaded to an S3 bucket; the metadata is stored in Amazon DynamoDB. The application starts behaving unexpectedly, and the developer wants to examine the logs of the Lambda function code for errors.
Based on this system configuration, where would the developer find the logs?

A.Amazon S3
B.AWS CloudTrail
C.Amazon CloudWatch
D.Amazon DynamoDB

Answer: C

Explanation:

The correct answer is C, Amazon CloudWatch. Lambda functions automatically integrate with CloudWatch Logs. When a Lambda function executes, it streams log data, including any print statements or errors encountered during execution, to a CloudWatch Logs group associated with the Lambda function.

Here's a detailed breakdown of why the other options are incorrect and why CloudWatch is the correct choice:

A. Amazon S3: S3 is object storage. While files trigger the Lambda function, the logs generated during the function's execution are not stored in S3. S3's primary purpose is to store data, not to capture application logs.

B. AWS CloudTrail: CloudTrail tracks API calls made to AWS services. It provides audit trails of who did what and when. While CloudTrail can record that a Lambda function was invoked, it does not contain the detailed application logs generated by the Lambda function's code, such as print statements or error messages.

D. Amazon DynamoDB: DynamoDB is a NoSQL database. It's used to store the extracted metadata in this scenario. DynamoDB would not store logs generated by a Lambda function. DynamoDB stores structured data, not unstructured log information.

CloudWatch Logs provides centralized logging for applications and services. Lambda functions are configured to send their logs to CloudWatch Logs by default, making it the ideal place to examine the errors or debug messages from the Lambda function. Developers can access these logs through the AWS Management Console, AWS CLI, or SDKs. CloudWatch offers various features to search, filter, and analyze log data to help troubleshoot application issues.

Therefore, the developer would find the Lambda function's logs in Amazon CloudWatch.

Further resources:

AWS Lambda Logging
Using AWS Lambda with Amazon CloudWatch Logs

Question: 56

A company is using an AWS Lambda function to process records from an Amazon Kinesis data stream. The company recently observed slow processing of the records. A developer notices that the iterator age metric for the function is increasing and that the Lambda run duration is constantly above normal.
Which actions should the developer take to increase the processing speed? (Choose two.)

A.Increase the number of shards of the Kinesis data stream.
B.Decrease the timeout of the Lambda function.
C.Increase the memory that is allocated to the Lambda function.
D.Decrease the number of shards of the Kinesis data stream.
E.Increase the timeout of the Lambda function.

Answer: AC

Explanation:

The correct answer is A and C. Here's why:

A. Increase the number of shards of the Kinesis data stream: The increasing iterator age suggests the Lambda function isn't keeping up with the incoming data. Kinesis shards are the base unit of throughput. Each shard provides a capacity of 1 MB/sec input and 2 MB/sec output. By increasing the number of shards, you increase the overall read capacity of the stream, allowing more concurrent Lambda function invocations to process data in parallel. Each Lambda function execution reads from a single shard at a time. More shards effectively parallelize the processing. This directly addresses the slow processing and increasing iterator age.
https://docs.aws.amazon.com/streams/latest/dev/kinesis-scaling.html

C. Increase the memory that is allocated to the Lambda function: Increasing the memory allocated to the Lambda function also increases its CPU allocation proportionally. Lambda functions allocate CPU power proportional to the memory allocated. If the Lambda's run duration is constantly above normal, it indicates

that the function might be CPU-bound or taking longer for processing. By increasing the memory, you give the function more CPU power, potentially speeding up the record processing. A more performant execution decreases run duration and assists in keeping up with incoming stream data.
https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html

Now, let's consider why the other options are incorrect:

B. Decrease the timeout of the Lambda function: Decreasing the timeout would only cause the function to terminate prematurely if it's already taking too long to process the records. It doesn't address the root cause of the slow processing.

D. Decrease the number of shards of the Kinesis data stream: Decreasing the number of shards would reduce the read capacity of the stream, further exacerbating the slow processing and increasing iterator age.

E. Increase the timeout of the Lambda function: While this prevents the Lambda from timing out, it does not speed up processing. It simply provides a longer window for the slow processing to complete, masking the underlying performance issue rather than fixing it.

Question: 57

A company needs to harden its container images before the images are in a running state. The company's application uses Amazon Elastic Container Registry (Amazon ECR) as an image registry. Amazon Elastic Kubernetes Service (Amazon EKS) for compute, and an AWS CodePipeline pipeline that orchestrates a continuous integration and continuous delivery (CI/CD) workflow.
Dynamic application security testing occurs in the final stage of the pipeline after a new image is deployed to a development namespace in the EKS cluster. A developer needs to place an analysis stage before this deployment to analyze the container image earlier in the CI/CD pipeline.
Which solution will meet these requirements with the MOST operational efficiency?

A.Build the container image and run the docker scan command locally. Mitigate any findings before pushing changes to the source code repository. Write a pre-commit hook that enforces the use of this workflow before commit.

B.Create a new CodePipeline stage that occurs after the container image is built. Configure ECR basic image scanning to scan on image push. Use an AWS Lambda function as the action provider. Configure the Lambda function to check the scan results and to fail the pipeline if there are findings.

C.Create a new CodePipeline stage that occurs after source code has been retrieved from its repository. Run a security scanner on the latest revision of the source code. Fail the pipeline if there are findings.

D.Add an action to the deployment stage of the pipeline so that the action occurs before the deployment to the EKS cluster. Configure ECR basic image scanning to scan on image push. Use an AWS Lambda function as the action provider. Configure the Lambda function to check the scan results and to fail the pipeline if there are findings.

Answer: B

Explanation:

Here's a detailed justification for why option B is the best solution, along with supporting concepts and links:

The core requirement is to analyze container images before deployment but after the image is built, integrating it seamlessly into the existing CI/CD pipeline with operational efficiency.

Option B directly addresses this by inserting a new CodePipeline stage after the image build. It leverages ECR's built-in basic image scanning on push, which is operationally efficient because it's a managed service feature requiring minimal custom infrastructure. The AWS Lambda function acts as an orchestrator, checking scan results and failing the pipeline based on findings. This creates a "gate" in the pipeline, preventing vulnerable images from being deployed. Lambda's event-driven, serverless nature ensures cost-effectiveness and scalability for this task.

Option A is less efficient. Relying on developers to run docker scan locally and enforce this with pre-commit hooks is prone to human error and inconsistent enforcement. It doesn't guarantee consistent security practices across the development team and lacks centralized control.

Option C focuses on scanning source code, which is valuable but doesn't directly address the vulnerabilities that might be introduced during the container image build process (e.g., outdated packages pulled from repositories during docker build). While source code analysis is important, it doesn't satisfy the requirement of analyzing the container image.

Option D attempts to insert the scanning logic into the deployment stage, which is after the image has already passed through earlier stages of the pipeline. This is less than ideal as it delays vulnerability detection until a later point and may require rolling back a deployed image if issues are found. Additionally, while leveraging ECR scanning and Lambda is good, adding it to the deployment stage doesn't truly analyze the image "earlier" in the pipeline, which is a key requirement.

Therefore, option B offers the best balance of operational efficiency, integration with the existing CI/CD pipeline, and the crucial timing of image analysis after the build but before deployment. ECR's integrated scanning and a simple Lambda function minimizes operational overhead.

Supporting concepts and links:

Amazon ECR Image Scanning:https://docs.aws.amazon.com/AmazonECR/latest/userguide/image-scanning.html
AWS Lambda:https://aws.amazon.com/lambda/
AWS CodePipeline:https://aws.amazon.com/codepipeline/
CI/CD Pipelines: Understanding the principles of CI/CD helps to understand where different security checks fit in the pipeline for maximum impact.

Question: 58

A developer is testing a new file storage application that uses an Amazon CloudFront distribution to serve content from an Amazon S3 bucket. The distribution accesses the S3 bucket by using an origin access identity (OAI). The S3 bucket's permissions explicitly deny access to all other users.
The application prompts users to authenticate on a login page and then uses signed cookies to allow users to access their personal storage directories. The developer has configured the distribution to use its default cache behavior with restricted viewer access and has set the origin to point to the S3 bucket. However, when the developer tries to navigate to the login page, the developer receives a 403 Forbidden error.
The developer needs to implement a solution to allow unauthenticated access to the login page. The solution also must keep all private content secure.
Which solution will meet these requirements?

A.Add a second cache behavior to the distribution with the same origin as the default cache behavior. Set the path pattern for the second cache behavior to the path of the login page, and make viewer access unrestricted. Keep the default cache behavior's settings unchanged.

B.Add a second cache behavior to the distribution with the same origin as the default cache behavior. Set the path pattern for the second cache behavior to *, and make viewer access restricted. Change the default cache behavior's path pattern to the path of the login page, and make viewer access unrestricted.

C.Add a second origin as a failover origin to the default cache behavior. Point the failover origin to the S3 bucket. Set the path pattern for the primary origin to *, and make viewer access restricted. Set the path pattern for the failover origin to the path of the login page, and make viewer access unrestricted.

D.Add a bucket policy to the S3 bucket to allow read access. Set the resource on the policy to the Amazon Resource Name (ARN) of the login page object in the S3 bucket. Add a CloudFront function to the default cache behavior to redirect unauthorized requests to the login page's S3 URL.

Answer: A

Explanation:

The correct solution is A. This approach leverages CloudFront's cache behavior functionality to selectively allow unauthenticated access to the login page while maintaining restricted access for other content.

Here's a detailed justification:

1. **Problem:** The default cache behavior restricts viewer access, meaning all requests to the CloudFront distribution are subject to the configured authentication (in this case, signed cookies). This includes the login page, which needs to be publicly accessible for unauthenticated users to initiate the login process.

2. **CloudFront Cache Behaviors:** CloudFront distributions can have multiple cache behaviors, each associated with a specific path pattern. The path pattern determines which cache behavior is applied to a given request.

   https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/distribution-web-values-general.html#d0E940

3. **Solution Rationale:** By adding a second cache behavior with the login page's path pattern and setting viewer access to unrestricted, we create an exception. When a user requests the login page, CloudFront will use this new cache behavior and serve the page directly from S3 without requiring signed cookies. The default cache behavior, still in place, continues to protect all other content behind the signed cookie authentication.

4. **Why other options are incorrect:**

   B: Changing the default cache behavior's path pattern to the login page would expose the login page but still restrict all other resources using signed cookies. Using "*" for restricted viewer access on the new behavior is incorrect.

   C: Failover origins are for high availability, not access control. It wouldn't restrict content behind signed cookies. The primary origin configuration will always attempt access with the OAI.

   D: Modifying the S3 bucket policy to directly expose the login page might work at the S3 level, but would bypass CloudFront and thus defeat the purpose of using CloudFront for caching and security. CloudFront functions redirecting unauthorized requests to the S3 URL will also bypass CloudFront protection for all other resources. It's better to manage access within CloudFront.

5. **Security:** By maintaining the OAI and bucket policy that only allows access from the CloudFront distribution, the private content remains secure. Unauthenticated access is granted only for the login page through CloudFront's cache behavior, not directly to the S3 bucket.

6. **Scalability and Performance:** CloudFront's caching capabilities ensure that the login page is served quickly and efficiently, even with a high volume of requests.

In summary, adding a cache behavior specifically for the login page with unrestricted access allows unauthenticated users to reach the login form while keeping the private content secure via signed cookies enforced by the default cache behavior.

Question: 59

A developer is using AWS Amplify Hosting to build and deploy an application. The developer is receiving an increased number of bug reports from users. The developer wants to add end-to-end testing to the application to eliminate as many bugs as possible before the bugs reach production.
Which solution should the developer implement to meet these requirements?

A.Run the amplify add test command in the Amplify CLI.

B.Create unit tests in the application. Deploy the unit tests by using the amplify push command in the Amplify

CLI.

C.Add a test phase to the amplify.yml build settings for the application.

D.Add a test phase to the aws-exports.js file for the application.

Answer: C

Explanation:

The correct answer is C: Add a test phase to the amplify.yml build settings for the application.

Here's why: AWS Amplify Hosting leverages the amplify.yml file to define the build and deployment process for your application. This file allows you to customize different phases like install, build, and test. Adding a test phase specifically allows you to incorporate end-to-end (E2E) testing directly into your deployment pipeline.

Option A, using amplify add test, is not a valid Amplify CLI command for directly integrating E2E testing into the build process. It is typically used to add backend functionality.

Option B, creating and deploying unit tests using amplify push, is important for code quality, but doesn't address the requirement of end-to-end testing. Unit tests verify individual components in isolation, while E2E tests simulate real user scenarios across the entire application. E2E tests are crucial for catching integration issues that unit tests might miss. While amplify push deploys backend changes, it doesn't execute frontend test suites.

Option D, adding a test phase to aws-exports.js, is incorrect. The aws-exports.js file primarily contains configuration information required for your application to interact with AWS services and is not the appropriate place to define build and test phases. Altering it for testing purposes could lead to unpredictable application behavior.

By modifying the amplify.yml file, the developer can define commands to execute E2E tests (using tools like Cypress, Selenium, or Playwright) during the build process. This ensures that before deploying to production, the application is thoroughly tested for user flows and integration with backend services. If the E2E tests fail, the build can be stopped, preventing faulty code from reaching users. This significantly reduces the number of bugs reported in production.

Relevant Documentation:

AWS Amplify Hosting Configuration (amplify.yml):https://docs.amplify.aws/gen2/build-compute/configuration/
AWS Amplify Hosting Build Specifications:https://docs.aws.amazon.com/amplify/latest/userguide/build-settings.html

Question: 60

An ecommerce company is using an AWS Lambda function behind Amazon API Gateway as its application tier. To process orders during checkout, the application calls a POST API from the frontend. The POST API invokes the Lambda function asynchronously. In rare situations, the application has not processed orders. The Lambda application logs show no errors or failures.
What should a developer do to solve this problem?

A.Inspect the frontend logs for API failures. Call the POST API manually by using the requests from the log file.

B.Create and inspect the Lambda dead-letter queue. Troubleshoot the failed functions. Reprocess the events.

C.Inspect the Lambda logs in Amazon CloudWatch for possible errors. Fix the errors.

D.Make sure that caching is disabled for the POST API in API Gateway.

Answer: B

Explanation:

The correct answer is **B. Create and inspect the Lambda dead-letter queue. Troubleshoot the failed functions. Reprocess the events.** Here's why:

When a Lambda function is invoked asynchronously, AWS handles retries if the function fails. However, after exhausting retry attempts, the event can be sent to a dead-letter queue (DLQ). The question states that the Lambda logs show no errors or failures. This strongly suggests the function initially succeeds, but some downstream operation is failing after the Lambda execution context has ended, making standard CloudWatch logs less useful. These failures are handled by retry mechanism, then by DLQ when retry fails

A DLQ acts as a repository for events that the Lambda function was unable to process successfully after all retries are exhausted. Inspecting the DLQ allows developers to identify the specific events that failed and understand the reasons for their failure (e.g., bad data, dependency issues, intermittent downstream failures).
This is crucial when standard logs show nothing, as DLQ give you a record of failed events that need to be inspected

By analyzing the failed events in the DLQ, the developer can troubleshoot the root cause of the processing failures. Once the issue is resolved (e.g., fixing a bug in the Lambda function or correcting data errors), the events can be reprocessed to ensure that the orders are processed correctly, solving the initial problem.

Here's why the other options are less ideal:

A. Inspect the frontend logs for API failures. Call the POST API manually by using the requests from the log file: While frontend logs can be useful, the question states the Lambda logs show no errors. Frontend errors might suggest the API call never happened, but this does not explain why events are not processing if they reach the Lambda.

C. Inspect the Lambda logs in Amazon CloudWatch for possible errors. Fix the errors: The question states that the Lambda application logs show no errors or failures. This makes inspecting CloudWatch logs less relevant.

D. Make sure that caching is disabled for the POST API in API Gateway: Caching in API Gateway could cause stale data, but it would more likely result in processing incorrect orders, not in orders not being processed.

Relevant Documentation:

AWS Lambda Dead-Letter Queues:https://docs.aws.amazon.com/lambda/latest/dg/services-sqs.html
Asynchronous Invocation:https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html

Question: 61

A company is building a web application on AWS. When a customer sends a request, the application will generate reports and then make the reports available to the customer within one hour. Reports should be accessible to the customer for 8 hours. Some reports are larger than 1 MB. Each report is unique to the customer. The application should delete all reports that are older than 2 days.
Which solution will meet these requirements with the LEAST operational overhead?

A.Generate the reports and then store the reports as Amazon DynamoDB items that have a specified TTL. Generate a URL that retrieves the reports from DynamoDB. Provide the URL to customers through the web application.

B.Generate the reports and then store the reports in an Amazon S3 bucket that uses server-side encryption. Attach the reports to an Amazon Simple Notification Service (Amazon SNS) message. Subscribe the customer to email notifications from Amazon SNS.

C.Generate the reports and then store the reports in an Amazon S3 bucket that uses server-side encryption. Generate a presigned URL that contains an expiration date Provide the URL to customers through the web application. Add S3 Lifecycle configuration rules to the S3 bucket to delete old reports.

D.Generate the reports and then store the reports in an Amazon RDS database with a date stamp. Generate an URL that retrieves the reports from the RDS database. Provide the URL to customers through the web application. Schedule an hourly AWS Lambda function to delete database records that have expired date stamps.

Answer: C

Explanation:

The best solution is option C because it leverages Amazon S3, presigned URLs, and lifecycle policies to efficiently manage report storage, access, and deletion with minimal operational overhead.

Here's why:

S3 for Storage: S3 is designed for storing objects (like reports) and offers scalability, durability, and cost-effectiveness. https://aws.amazon.com/s3/
Server-Side Encryption: Provides data security at rest, meeting security requirements.

Presigned URLs: Presigned URLs grant temporary access to S3 objects without requiring customers to have AWS credentials. This is a secure and convenient way to provide access to the reports for a limited time (8 hours, as required).

https://docs.aws.amazon.com/AmazonS3/latest/userguide/PresignedUrlUploadObject.html
S3 Lifecycle Policies: S3 lifecycle policies automate the deletion of objects based on age. This fulfills the requirement to delete reports older than 2 days without needing custom code or manual intervention.
https://docs.aws.amazon.com/AmazonS3/latest/userguide/lifecycle-configuration-examples.html

Why other options are less suitable:

A (DynamoDB): DynamoDB is a NoSQL database suitable for key-value or document data, not ideal for storing large reports (especially > 1MB) because its cost can be prohibitive for large binary data. Also, DynamoDB TTL (Time To Live) is more suited for eventual cleanup and not as precise as S3 Lifecycle policies, and serving the files through DynamoDB is less efficient than serving them directly from S3.

B (SNS): SNS is a messaging service, and attaching large reports directly to SNS messages is inefficient and potentially impractical. SNS is designed for notifications, not for distributing large files.

D (RDS): RDS is a relational database. Storing binary reports directly in an RDS database isn't generally recommended for large files. It can impact database performance and increase storage costs. Also, scheduling a Lambda function to delete records adds operational overhead compared to S3 lifecycle policies. Furthermore, retrieving the reports from a database rather than S3 is less efficient.

Question: 62

A company has deployed an application on AWS Elastic Beanstalk. The company has configured the Auto Scaling group that is associated with the Elastic Beanstalk environment to have five Amazon EC2 instances. If the capacity is fewer than four EC2 instances during the deployment, application performance degrades. The company is using the all-at-once deployment policy.
What is the MOST cost-effective way to solve the deployment issue?

A.Change the Auto Scaling group to six desired instances.

B.Change the deployment policy to traffic splitting. Specify an evaluation time of 1 hour.

C.Change the deployment policy to rolling with additional batch. Specify a batch size of 1.

D.Change the deployment policy to rolling. Specify a batch size of 2.

Answer: C

Explanation:

Here's a detailed justification for why option C is the most cost-effective solution, along with relevant links for further research:

The core problem is that the "all-at-once" deployment policy in Elastic Beanstalk causes application degradation because it can briefly reduce the instance count below the required threshold (4 instances). The goal is to maintain at least four running instances throughout the deployment process while minimizing costs.

Option A (Increase desired instances to six): This ensures capacity during deployment, but it's the least cost-effective. The company will permanently pay for an additional instance, even when deployments aren't happening. This is an over-provisioned and therefore, expensive solution.

Option B (Traffic splitting with evaluation): Traffic splitting (Canary deployments) is a valid strategy but focuses more on verifying the new version before a full rollout. The evaluation period adds overhead and doesn't directly address the instance count problem during the initial deployment phase. It is also more complex to implement and manage.

Option C (Rolling with additional batch, batch size 1): This is the most cost-effective. "Rolling with additional batch" temporarily increases the Auto Scaling group's desired capacity by the batch size during the deployment. A batch size of 1 means that one new instance is launched before one old instance is taken out of service. Since the Auto Scaling group starts with five instances, the instance count will briefly increase to six.

This guarantees that there will always be at least four healthy instances serving traffic during the deployment. Once the deployment to a batch is complete, that instance joins the original instance pool, allowing for another batch to be deployed. After the entire deployment is complete, the temporary instance is removed, returning the total running count back to the desired five. This only adds temporary cost for the length of the deployment, and only by one instance at a time.

Option D (Rolling, batch size 2): Rolling deployments update instances in batches, but without the "additional batch" functionality, it will take down two instances simultaneously, which can degrade performance and violate the requirement of keeping at least four instances running.

In summary: Option C strikes the optimal balance between maintaining capacity and minimizing cost. By adding only one additional instance at a time during the deployment, and then removing it later, it minimizes overhead while still fulfilling the requirement.

Authoritative Links for further research:

AWS Elastic Beanstalk Deployment Policies:https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features-deploy-policies.html
Elastic Beanstalk Rolling Deployments with Additional Batch:
https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features-rolling-version-deploy.html **AWS Auto Scaling:**https://aws.amazon.com/autoscaling/

Question: 63

A developer is incorporating AWS X-Ray into an application that handles personal identifiable information (PII). The application is hosted on Amazon EC2 instances. The application trace messages include encrypted PII and go to Amazon CloudWatch. The developer needs to ensure that no PII goes outside of the EC2 instances.
Which solution will meet these requirements?

A.Manually instrument the X-Ray SDK in the application code.
B.Use the X-Ray auto-instrumentation agent.
C.Use Amazon Macie to detect and hide PII. Call the X-Ray API from AWS Lambda.
D.Use AWS Distro for Open Telemetry.

Answer: A

Explanation:

The correct answer is A: Manually instrument the X-Ray SDK in the application code.

Here's a detailed justification:

The requirement is to prevent PII from leaving the EC2 instances while still using X-Ray for tracing. Manually instrumenting the X-Ray SDK gives the developer granular control over what data is captured and sent as trace data. This allows the developer to specifically exclude any PII before the data is passed to the X-Ray service. By selectively choosing what to trace, the developer can ensure no sensitive information leaves the application.

Option B, using the X-Ray auto-instrumentation agent, is less desirable because it automatically captures a broad range of data, potentially including PII, without explicit control from the developer. This increases the risk of sending PII outside the EC2 instance. Auto-instrumentation focuses on ease of use and might not provide the necessary level of control when dealing with sensitive data.

Option C, using Amazon Macie and AWS Lambda, is an overkill and introduces unnecessary complexity. Macie is designed for discovering and protecting sensitive data at rest and in transit across AWS, and utilizing Lambda adds extra points of failure and latency. It's more efficient to directly control what data is traced at the source rather than attempt to scrub it later. Moreover, the trace data is already encrypted, so the primary concern is preventing the PII from being collected and added to the traces in the first place, which Macie and Lambda do not address at this stage.

Option D, AWS Distro for OpenTelemetry (ADOT), provides a vendor-neutral way to collect telemetry data.
While ADOT offers flexibility, it still requires careful configuration to prevent PII from being captured. Using ADOT, you would need to configure processors to remove PII, which adds complexity similar to Option C and is less straightforward than directly controlling what data is traced using manual instrumentation. Manual instrumentation via the X-Ray SDK gives the developer direct control at the source of the trace information.

Therefore, manually instrumenting the X-Ray SDK within the application code ensures the greatest level of control over what data is traced and prevents PII from inadvertently leaving the EC2 instances, satisfying the requirements of the question.

Supporting links:

AWS X-Ray SDK: https://docs.aws.amazon.com/xray/latest/devguide/xray-sdk.html
AWS X-Ray Concepts: https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html

Question: 64

A developer is migrating some features from a legacy monolithic application to use AWS Lambda functions instead. The application currently stores data in an Amazon Aurora DB cluster that runs in private subnets in a VPC. The AWS account has one VPC deployed. The Lambda functions and the DB cluster are deployed in the same AWS Region in the same AWS account. The developer needs to ensure that the Lambda functions can securely access the DB cluster without crossing the public internet.
Which solution will meet these requirements?

A.Configure the DB cluster's public access setting to Yes.
B.Configure an Amazon RDS database proxy for he Lambda functions.
C.Configure a NAT gateway and a security group for the Lambda functions.
D.Configure the VPC, subnets, and a security group for the Lambda functions.

Answer: D

Explanation:

The correct answer is **D: Configure the VPC, subnets, and a security group for the Lambda functions.**

Here's why:

VPC Integration is Key: Lambda functions, by default, run in an AWS-managed VPC. To access resources in your private VPC (where the Aurora DB cluster resides), the Lambda functions must be configured to connect to your VPC. This establishes a secure, private network path.

Subnets and Security Groups Define Access: Deploying Lambda functions into the same VPC as the Aurora DB cluster and specifying subnets within that VPC lets the Lambda functions obtain private IP addresses within the VPC's CIDR block. Security groups then control the inbound and outbound traffic. We want to allow traffic from the Lambda function's security group to the Aurora DB cluster's security group on the appropriate port (typically 3306 for MySQL/Aurora).

Avoiding Public Internet: The requirement to avoid the public internet is directly met by keeping all communication within the VPC using private IP addresses. This is much more secure than exposing the database publicly.

Why other options are wrong:

A: Configuring the DB cluster's public access setting to Yes: This is fundamentally incorrect because it directly contradicts the requirement of not crossing the public internet. It exposes the database to the outside world, which is a major security risk.

B: Configuring an Amazon RDS database proxy for the Lambda functions: While RDS Proxy can improve connection management and security, it doesn't inherently solve the networking issue of the Lambda functions being isolated from the VPC. It needs VPC integration first. Furthermore, while helpful, it's not strictly necessary to meet the basic requirement of secure private access.

C: Configuring a NAT gateway and a security group for the Lambda functions: NAT gateway is required only for outbound internet traffic. Since we need the lambda functions to connect directly to aurora database inside private subnet, the lambda function will require to be placed inside private subnet and no need for NAT gateway.

Authoritative Links for Further Research:

Configuring Lambda functions to access resources in an Amazon VPC:
https://docs.aws.amazon.com/lambda/latest/dg/configuration-vpc.html
Security Groups:https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html
Amazon RDS Proxy:https://docs.aws.amazon.com/rds/proxy/latest/userguide/rds-proxy.html

Question: 65

A developer is building a new application on AWS. The application uses an AWS Lambda function that retrieves information from an Amazon DynamoDB table. The developer hard coded the DynamoDB table name into the Lambda function code. The table name might change over time. The developer does not want to modify the Lambda code if the table name changes.
Which solution will meet these requirements MOST efficiently?

A.Create a Lambda environment variable to store the table name. Use the standard method for the programming language to retrieve the variable.

B.Store the table name in a file. Store the file in the /tmp folder. Use the SDK for the programming language to retrieve the table name.

C.Create a file to store the table name. Zip the file and upload the file to the Lambda layer. Use the SDK for the programming language to retrieve the table name.

D.Create a global variable that is outside the handler in the Lambda function to store the table name.

Answer: A

Explanation:

The most efficient solution is to use Lambda environment variables. Here's why:

Option A is the most efficient because Lambda environment variables are designed specifically for storing configuration information that needs to be accessible to Lambda functions without being hardcoded into the function's code. This makes it easy to update the DynamoDB table name without redeploying the function. Lambda seamlessly integrates with these variables, providing a simple and direct way to retrieve them using standard programming language methods.

Option B is less efficient. While storing the table name in a file within the /tmp directory works, it adds unnecessary complexity. The function would need to read the file every time it needs the table name, introducing I/O overhead. Moreover, the /tmp directory is ephemeral and not suitable for persistent storage of configuration.

Option C, using Lambda Layers, is also less efficient than using environment variables. Layers are best used for code or dependencies shared across multiple Lambda functions, not for simple configuration values. Creating a layer for a single table name introduces unnecessary overhead in terms of deployment and maintenance. While layers can be used, it's overkill for this simple scenario.

Option D, using a global variable outside the handler, has severe drawbacks. The Lambda execution environment can be reused, so the global variable's value might persist across invocations. This can lead to unexpected behavior if the variable is modified during one invocation and then used in a subsequent invocation before being reset. Furthermore, this approach still requires modifying the Lambda function's code if the table name changes, defeating the purpose of the requirement.

In conclusion, Lambda environment variables provide the simplest, most efficient, and recommended approach for managing configuration data like DynamoDB table names in Lambda functions. They are designed for this purpose, avoid unnecessary overhead, and provide a clear separation of code and configuration.

Relevant Links:

AWS Lambda Environment Variables: https://docs.aws.amazon.com/lambda/latest/dg/configuration-envvars.html

Question: 66

A company has a critical application on AWS. The application exposes an HTTP API by using Amazon API Gateway. The API is integrated with an AWS Lambda function. The application stores data in an Amazon RDS for MySQL DB instance with 2 virtual CPUs (vCPUs) and 64 GB of RAM.

Customers have reported that some of the API calls return HTTP 500 Internal Server Error responses. Amazon CloudWatch Logs shows errors for “too many connections.” The errors occur during peak usage times that are unpredictable.

The company needs to make the application resilient. The database cannot be down outside of scheduled maintenance hours.

Which solution will meet these requirements?

A.Decrease the number of vCPUs for the DB instance. Increase the max_connections setting.

B.Use Amazon RDS Proxy to create a proxy that connects to the DB instance. Update the Lambda function to connect to the proxy.

C.Add a CloudWatch alarm that changes the DB instance class when the number of connections increases to more than 1,000.

D.Add an Amazon EventBridge rule that increases the max_connections setting of the DB instance when CPU utilization is above 75%.

Answer: B

Explanation:

The best solution is **B. Use Amazon RDS Proxy to create a proxy that connects to the DB instance. Update the Lambda function to connect to the proxy.**

Here's why:

The problem is "too many connections" to the RDS database, leading to HTTP 500 errors during peak usage. RDS Proxy is specifically designed to manage database connections and improve application scalability and resilience in such scenarios. It sits between the application (Lambda function) and the RDS database, pooling and sharing database connections. This reduces the number of direct connections to the database, preventing it from being overwhelmed, even during unpredictable peak usage times. https://aws.amazon.com/rds/proxy/

By updating the Lambda function to connect to the RDS Proxy endpoint, the function no longer directly establishes connections with the database. Instead, it uses the proxy as an intermediary. The proxy then manages the connection lifecycle, efficiently reusing connections and preventing connection exhaustion.
https://docs.aws.amazon.com/rds/proxy/latest/userguide/what-is.html

Let's examine why the other options are less suitable:

A. Decrease the number of vCPUs for the DB instance. Increase the max_connections setting: Decreasing vCPUs will likely degrade performance. Simply increasing max_connections might postpone the issue but doesn't solve the underlying problem of inefficient connection management and could lead to resource contention.

C. Add a CloudWatch alarm that changes the DB instance class when the number of connections increases to more than 1,000: Scaling up the instance class is a reactive measure and takes time. It doesn't prevent the "too many connections" error from happening in the first place, leading to temporary downtime. This is also less efficient and cost-effective than connection pooling. Furthermore, changing the instance class requires a database restart which leads to downtime.

D. Add an Amazon EventBridge rule that increases the max_connections setting of the DB instance when CPU utilization is above 75%: Dynamically adjusting max_connections based on CPU utilization is complex and potentially risky. Incorrectly configured rules could lead to excessive resource consumption or instability.

It also doesn't address the core issue of inefficient connection management. Similar to option A, it also requires a database restart.

RDS Proxy provides a managed and reliable solution specifically designed to handle the "too many connections" problem, making it the most appropriate choice for ensuring application resilience without significant downtime.

Question: 67

A company has installed smart meters in all its customer locations. The smart meters measure power usage at 1-minute intervals and send the usage readings to a remote endpoint for collection. The company needs to create an endpoint that will receive the smart meter readings and store the readings in a database. The company wants to store the location ID and timestamp information.

The company wants to give its customers low-latency access to their current usage and historical usage on demand. The company expects demand to increase significantly. The solution must not impact performance or include downtime while scaling.

Which solution will meet these requirements MOST cost-effectively?

A.Store the smart meter readings in an Amazon RDS database. Create an index on the location ID and timestamp columns. Use the columns to filter on the customers' data.

B.Store the smart meter readings in an Amazon DynamoDB table. Create a composite key by using the location ID and timestamp columns. Use the columns to filter on the customers' data.

C.Store the smart meter readings in Amazon ElastiCache for Redis. Create a SortedSet key by using the location ID and timestamp columns. Use the columns to filter on the customers' data.

D.Store the smart meter readings in Amazon S3. Partition the data by using the location ID and timestamp columns. Use Amazon Athena to filter on the customers' data.

Answer: B

Explanation:

Here's a detailed justification for why option B is the most cost-effective solution for storing and querying smart meter data, considering the requirements of low-latency access, scalability, and cost-effectiveness:

DynamoDB for Scalability and Low Latency: DynamoDB is a fully managed NoSQL database service specifically designed for high-performance applications that require consistent, single-digit millisecond latency at any scale. The question mentions the expectation of significant demand increase, making DynamoDB's scalability a crucial factor.

Composite Key Design: Using a composite key (partition key and sort key) consisting of the location ID and timestamp is an optimal strategy for DynamoDB. The location ID acts as the partition key, distributing the data across multiple partitions for horizontal scalability. The timestamp acts as the sort key, allowing for efficient querying of data within each partition based on time ranges. This directly supports the requirement for historical usage data access.

Filtering with Keys: DynamoDB allows for efficient filtering using the key attributes. Customers' data can be easily retrieved by querying based on their location ID (partition key) and specific time ranges (using the timestamp sort key). This fulfills the requirement of low-latency access to customer data.

Cost-Effectiveness: DynamoDB's pay-per-request and provisioned capacity modes offer cost optimization opportunities. Given the high volume of incoming meter readings, analyzing usage patterns can help determine the most cost-effective billing model.

RDS Drawbacks: While RDS (option A) can handle structured data, it typically requires more management overhead and might not scale as seamlessly and cost-effectively as DynamoDB for the expected high volume of data and requests. Indexing helps but might not provide the same low latency at scale.

ElastiCache Drawbacks: ElastiCache (option C) is an in-memory data store primarily used for caching. It's not designed for durable storage of large volumes of historical data, as required by the question. Storing all readings in ElastiCache would be costly and impractical.

S3/Athena Drawbacks: S3/Athena (option D) is suitable for analytical queries on large datasets, but the query latency is significantly higher compared to DynamoDB, making it unsuitable for low-latency, on-demand access to current and historical usage data. The data would need to be scanned, making it a slower process.

In summary, DynamoDB provides the best balance of scalability, low latency, cost-effectiveness, and ease of use for the given requirements. The composite key design enables efficient data retrieval and filtering, directly addressing the need for low-latency access to current and historical usage data for each customer.

Relevant links:

Amazon DynamoDB
DynamoDB Key Design
Choosing the Right Database

Question: 68

A company is building a serverless application that uses AWS Lambda functions. The company needs to create a set of test events to test Lambda functions in a development environment. The test events will be created once and then will be used by all the developers in an IAM developer group. The test events must be editable by any of the IAM users in the IAM developer group.

Which solution will meet these requirements?

A.Create and store the test events in Amazon S3 as JSON objects. Allow S3 bucket access to all IAM users.
B.Create the test events. Configure the event sharing settings to make the test events shareable.
C.Create and store the test events in Amazon DynamoDB. Allow access to DynamoDB by using IAM roles.
D.Create the test events. Configure the event sharing settings to make the test events private.

Answer: B

Explanation:

The correct answer is **B**. Here's why:

The requirement is to create and share editable test events for Lambda functions among developers in an IAM group.

Option A: While storing events in S3 and granting access might seem feasible, it involves managing JSON files directly. This isn't how Lambda typically handles test events. Lambda has a built-in mechanism for managing and sharing them. S3 is also not as streamlined for managing test events, making it difficult to modify and trigger from the Lambda console.

Option B: Lambda supports sharing test events. This is a direct, built-in feature designed for this exact purpose. Configuring event sharing makes events accessible and editable by authorized users (in this case, members of the IAM developer group). This allows developers to easily use and modify test events directly within the Lambda console/AWS CLI.

Option C: Similar to S3, DynamoDB isn't the intended way to manage Lambda test events. It would require complex application logic to translate DynamoDB entries into usable test events within the Lambda environment.

Option D: Creating private test events contradicts the requirement that all developers in the group should have access to and be able to edit the events.

Therefore, option B offers the simplest, most direct, and intended way to achieve the stated requirements using Lambda's built-in test event sharing functionality. It avoids the complexity of managing event data in external storage like S3 or DynamoDB.

Lambda Test Event Documentation: https://docs.aws.amazon.com/lambda/latest/dg/testing-functions.html (While this document doesn't explicitly mention "sharing," it implicitly discusses creation/management which leads to shared configurations.)

Question: 69

A developer is configuring an application's deployment environment in AWS CodePipeline. The application code is stored in a GitHub repository. The developer wants to ensure that the repository package's unit tests run in the new deployment environment. The developer has already set the pipeline's source provider to GitHub and has specified the repository and branch to use in the deployment.

Which combination of steps should the developer take next to meet these requirements with the LEAST overhead? (Choose two.)

A.Create an AWS CodeCommit project. Add the repository package's build and test commands to the project's buildspec.

B.Create an AWS CodeBuild project. Add the repository package's build and test commands to the project's buildspec.

C.Create an AWS CodeDeploy project. Add the repository package's build and test commands to the project's buildspec.

D.Add an action to the source stage. Specify the newly created project as the action provider. Specify the build artifact as the action's input artifact.

E.Add a new stage to the pipeline after the source stage. Add an action to the new stage. Specify the newly created project as the action provider. Specify the source artifact as the action's input artifact.

Answer: BE

Explanation:

The correct answer is BE. Here's why:

B. Create an AWS CodeBuild project. Add the repository package's build and test commands to the project's buildspec. AWS CodeBuild is a fully managed continuous integration service that compiles source code, runs tests, and produces software packages that are ready to deploy. A buildspec.yml file is used to define the build commands, including the necessary commands for running unit tests. This is exactly what the developer needs to accomplish. CodeBuild integrates directly with CodePipeline.
https://aws.amazon.com/codebuild/

E. Add a new stage to the pipeline after the source stage. Add an action to the new stage. Specify the newly created project as the action provider. Specify the source artifact as the action's input artifact. This step connects the CodeBuild project to the CodePipeline. After the "Source" stage (where the code is retrieved from GitHub), a new "Build" stage is needed to execute the unit tests. The "action provider" is set to the CodeBuild project, and the "input artifact" is the source code retrieved from GitHub (the "source artifact"). This ensures that the CodeBuild project receives the code from the source stage, runs the tests defined in the buildspec.yml, and produces an artifact for the next stage (if any).
https://docs.aws.amazon.com/codepipeline/latest/userguide/tutorial-create-simple-pipeline.html

Why the other options are incorrect:

A. Create an AWS CodeCommit project. Add the repository package's build and test commands to the project's buildspec. The code is already in a GitHub repository. Creating a CodeCommit project and duplicating the code is unnecessary overhead. CodePipeline integrates directly with GitHub, so there's no need to migrate the code.

C. Create an AWS CodeDeploy project. Add the repository package's build and test commands to the project's buildspec. AWS CodeDeploy is a deployment service that automates application deployments to various compute services such as EC2, ECS, Lambda, and on-premises servers. While CodeDeploy can run tests as part of deployment, it's not the right tool for running unit tests before deployment. CodeBuild is specifically designed for build and test phases in a pipeline. https://aws.amazon.com/codedeploy/

D. Add an action to the source stage. Specify the newly created project as the action provider. Specify the build artifact as the action's input artifact. The source stage's purpose is to retrieve the source code. Adding a build project to the source stage is illogical. The source stage should only handle fetching the code from the repository. Furthermore, there is no build artifact at the source stage, because the code hasn't been built or tested yet.

Question: 70

An engineer created an A/B test of a new feature on an Amazon CloudWatch Evidently project. The engineer configured two variations of the feature (Variation A and Variation B) for the test. The engineer wants to work exclusively with Variation A. The engineer needs to make updates so that Variation A is the only variation that appears when the engineer hits the application's endpoint.

Which solution will meet this requirement?

A.Add an override to the feature. Set the identifier of the override to the engineer's user ID. Set the variation to Variation A.

B.Add an override to the feature. Set the identifier of the override to Variation A. Set the variation to 100%.

C.Add an experiment to the project. Set the identifier of the experiment to Variation B. Set the variation to 0%.

D.Add an experiment to the project. Set the identifier of the experiment to the AWS account's account ISet the variation to Variation A.

Answer: A

Explanation:

The correct answer is A: Add an override to the feature. Set the identifier of the override to the engineer's user ID. Set the variation to Variation A.

Here's a detailed justification:

CloudWatch Evidently allows for feature flagging and A/B testing. When working on a specific feature, developers often need to isolate themselves to a particular variation for development and testing purposes without impacting other users or skewing the overall A/B test results. Evidently provides overrides precisely for this scenario. Overrides allow you to force a specific user (identified by a unique identifier, typically their user ID) to always receive a particular variation of a feature.

Option A directly addresses this need. By adding an override, the engineer can specify their user ID as the identifier. This ensures that any time the Evidently evaluation context includes this user ID, they will be served Variation A, regardless of the ongoing A/B test configuration or any other factors. This solution is isolated to the engineer's user, so it does not affect the larger A/B test running for other users. It is specific and targeted at the user who needs only the variation.

Option B is incorrect because overrides are meant to be associated with specific user identifiers, not variations themselves. Setting the identifier to "Variation A" wouldn't achieve the desired user-specific isolation. It might cause unexpected behavior or errors.

Options C and D are also incorrect because they suggest using experiments instead of overrides. Experiments are used for measuring impact of features across different groups, while overrides are for testing.
Experiments can be costly and can affect the production environment. Overrides are not meant to run production deployments.

In summary, Evidently's override functionality enables developers to control the feature variation they experience, providing a valuable tool for development and debugging without affecting the broader A/B test results or other users.

Refer to the official AWS CloudWatch Evidently documentation for more in-depth information:

AWS CloudWatch Evidently Features

CloudWatch Evidently Overrides

Question: 71

A developer is working on an existing application that uses Amazon DynamoDB as its data store. The DynamoDB table has the following attributes: partNumber (partition key), vendor (sort key), description, productFamily, and productType. When the developer analyzes the usage patterns, the developer notices that there are application modules that frequently look for a list of products based on the productFamily and productType attributes.

The developer wants to make changes to the application to improve performance of the query operations.

Which solution will meet these requirements?

A.Create a global secondary index (GSI) with productFamily as the partition key and productType as the sort key.

B.Create a local secondary index (LSI) with productFamily as the partition key and productType as the sort key.

C.Recreate the table. Add partNumber as the partition key and vendor as the sort key. During table creation, add a local secondary index (LSI) with productFamily as the partition key and productType as the sort key.

D.Update the queries to use Scan operations with productFamily as the partition key and productType as the sort key.

Answer: A

Explanation:

Here's a detailed justification for why option A is the correct solution:

The problem describes a scenario where frequent queries are being made based on productFamily and productType attributes, but the existing table structure (partition key: partNumber, sort key: vendor) is not optimized for these queries. This leads to inefficient scans. The goal is to improve query performance based on productFamily and productType.

Option A suggests creating a Global Secondary Index (GSI) with productFamily as the partition key and productType as the sort key. This is the best approach because GSIs allow you to query the DynamoDB table using attributes other than the table's primary key. By using productFamily and productType as the GSI's keys, the application can efficiently query for products belonging to a specific family and type without scanning the entire table. GSIs are "global" because they can be created and modified at any time, and they don't need to have the same partition key as the base table.

Option B, creating a Local Secondary Index (LSI), is incorrect because LSIs must share the same partition key as the base table (in this case, partNumber). The question specifically states the need to query based on productFamily and productType, which are distinct from the base table's partition key. Furthermore, LSIs are limited to 5 per table and are defined at table creation.

Option C is incorrect as recreating the table is an invasive and unnecessary change. Restructuring the primary key around partNumber and vendor doesn't solve the problem of querying on productFamily and productType. Furthermore, adding an LSI still necessitates using the base table's partition key, which defeats the purpose of efficient querying on the desired attributes.

Option D, using Scan operations, is the least efficient solution. Scans read through the entire table, filtering on productFamily and productType. This is resource-intensive, slow, and expensive, especially as the table grows. The objective is to avoid scans and instead use targeted queries.

In summary, a GSI with productFamily as the partition key and productType as the sort key provides the most efficient way to query the DynamoDB table based on those attributes, without requiring table recreation or inefficient scans.

Further research:

DynamoDB Indexes
Choosing the Right Index

Question: 72

A developer creates a VPC named VPC-A that has public and private subnets. The developer also creates an Amazon RDS database inside the private subnet of VPC-A. To perform some queries, the developer creates an AWS Lambda function in the default VPC. The Lambda function has code to access the RDS database. When the Lambda function runs, an error message indicates that the function cannot connect to the RDS database.

How can the developer solve this problem?

A.Modify the RDS security group. Add a rule to allow traffic from all the ports from the VPC CIDR block.

B.Redeploy the Lambda function in the same subnet as the RDS instance. Ensure that the RDS security group allows traffic from the Lambda function.

C.Create a security group for the Lambda function. Add a new rule in the RDS security group to allow traffic from the new Lambda security group.

D.Create an IAM role. Attach a policy that allows access to the RDS database. Attach the role to the Lambda function.

Answer: B

Explanation:

The problem is that the Lambda function, running in the default VPC, cannot connect to the RDS database in VPC-A's private subnet. This is because Lambda functions in a default VPC generally lack the necessary network configuration to reach resources in a separate, custom VPC.

Option B, redeploying the Lambda function into the same subnet as the RDS instance and adjusting the security group, directly addresses this issue. By placing the Lambda function within the same private subnet (or another subnet within VPC-A), it gains network access to the RDS database via internal IP addresses. The security group adjustment allows the Lambda function (or its associated ENI) to communicate with the RDS instance on the database port (e.g., 3306 for MySQL, 5432 for PostgreSQL). This removes the need for complex VPC peering or routing configurations. Lambda functions deployed within a VPC automatically inherit the VPC's network settings.

Option A is not optimal because opening all ports from the VPC CIDR block is a security risk. It violates the principle of least privilege by granting overly broad access.

Option C is partially correct in that it advocates for using security groups for Lambda, a best practice.
However, it doesn't address the fundamental networking issue of the Lambda function being in a different VPC. While it makes sense to have a security group associated with the Lambda function, without moving the function into VPC-A, the RDS instance will not be reachable.

Option D focuses on IAM roles, which are crucial for authorization (what actions the Lambda function can perform) but not network connectivity (how the Lambda function reaches the RDS database). The Lambda function already needs an IAM role to interact with AWS services, but that role doesn't solve the network connectivity problem. IAM cannot override networking limitations imposed by VPC boundaries.

Therefore, the most direct and secure solution is to place the Lambda function within VPC-A and configure the security groups appropriately, making Option B the correct answer.

Further research:

AWS Lambda VPC Networking: https://docs.aws.amazon.com/lambda/latest/dg/services-vpc.html AWS Security Groups: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html

Question: 73

A company runs an application on AWS. The company deployed the application on Amazon EC2 instances. The application stores data on Amazon Aurora.

The application recently logged multiple application-specific custom DECRYP_ERROR errors to Amazon CloudWatch logs. The company did not detect the issue until the automated tests that run every 30 minutes failed.
A developer must implement a solution that will monitor for the custom errors and alert a development team in real time when these errors occur in the production environment.

Which solution will meet these requirements with the LEAST operational overhead?

A.Configure the application to create a custom metric and to push the metric to CloudWatch. Create an AWS CloudTrail alarm. Configure the CloudTrail alarm to use an Amazon Simple Notification Service (Amazon SNS) topic to send notifications.

B.Create an AWS Lambda function to run every 5 minutes to scan the CloudWatch logs for the keyword DECRYP_ERROR. Configure the Lambda function to use Amazon Simple Notification Service (Amazon SNS) to send a notification.

C.Use Amazon CloudWatch Logs to create a metric filter that has a filter pattern for DECRYP_ERROR. Create a CloudWatch alarm on this metric for a threshold >=1. Configure the alarm to send Amazon Simple Notification Service (Amazon SNS) notifications.

D.Install the CloudWatch unified agent on the EC2 instance. Configure the application to generate a metric for the keyword DECRYP_ERROR errors. Configure the agent to send Amazon Simple Notification Service (Amazon SNS) notifications.

Answer: C

Explanation:

The correct answer is C because it leverages built-in CloudWatch features for log monitoring and alerting with minimal operational overhead. CloudWatch Logs metric filters allow you to define patterns to search for in logs and increment a metric when those patterns are found. In this scenario, the filter pattern is "DECRYP_ERROR". Creating a CloudWatch alarm based on this metric allows you to trigger an SNS notification when the error count (metric value) reaches a threshold (>=1), providing near real-time alerting to the development team.

Option A is incorrect because CloudTrail monitors AWS API calls, not application logs. It's not suitable for application-specific errors like DECRYP_ERROR. Creating a custom metric within the application and pushing it to CloudWatch (as suggested in A and D) is more complex than using built-in metric filters. While it works, it requires modifying application code and managing metric creation and pushing logic within the application itself.

Option B involves creating a Lambda function that polls CloudWatch Logs. This is less efficient than using metric filters because it requires more code to manage and execute and incurs costs for Lambda execution. Polling introduces latency and is less responsive than CloudWatch Logs' near real-time filtering capabilities.

Option D is less desirable because while it uses CloudWatch, it necessitates installing and configuring the CloudWatch agent on each EC2 instance. The problem can be solved without installing any agent to EC2 and creating custom metrics in the application. This increases operational overhead.

In summary, CloudWatch Logs metric filters with alarms offer a straightforward and efficient way to monitor logs for specific patterns and trigger notifications in real-time, fulfilling the requirements with the least operational burden.

Relevant Documentation:

CloudWatch Logs Metric Filters
CloudWatch Alarms
Amazon SNS

Question: 74

A developer created an AWS Lambda function that accesses resources in a VPC. The Lambda function polls an Amazon Simple Queue Service (Amazon SQS) queue for new messages through a VPC endpoint. Then the function calculates a rolling average of the numeric values that are contained in the messages. After initial tests of the Lambda function, the developer found that the value of the rolling average that the function returned was not accurate.

How can the developer ensure that the function calculates an accurate rolling average?

A.Set the function's reserved concurrency to 1. Calculate the rolling average in the function. Store the calculated rolling average in Amazon ElastiCache.

B.Modify the function to store the values in Amazon ElastiCache. When the function initializes, use the previous values from the cache to calculate the rolling average.

C.Set the function's provisioned concurrency to 1. Calculate the rolling average in the function. Store the calculated rolling average in Amazon ElastiCache.

D.Modify the function to store the values in the function's layers. When the function initializes, use the previously stored values to calculate the rolling average.

Answer: B

Explanation:

The correct answer is B. The inaccuracy in the rolling average calculation suggests that multiple Lambda function invocations are running concurrently and interfering with each other's state if the average is calculated and stored locally within the function's execution environment.

Option B addresses this concurrency issue directly. By using Amazon ElastiCache as a shared, external storage for the rolling average values, each Lambda invocation can access the most up-to-date state of the average, regardless of which invocation calculated it. ElastiCache ensures consistent and atomic updates to the average, preventing race conditions and ensuring accuracy across concurrent executions. The Lambda function initializes by fetching the last calculated average from ElastiCache and then uses that to calculate the new rolling average with the new message value. This ensures that each invocation contributes correctly to the overall calculation.

Option A is partially correct in setting the reserved concurrency to 1, which forces the function to execute sequentially. This would prevent concurrency issues. However, storing the rolling average in ElastiCache after each calculation is still beneficial for persistence and fault tolerance. If the Lambda function fails, the last saved average is maintained. Option A is also less efficient; limiting concurrency restricts throughput unnecessarily in many cases. ElastiCache is designed for low-latency data access, making it suitable for this type of state management.

Option C uses provisioned concurrency instead of reserved concurrency. While provisioned concurrency helps with cold starts, it doesn't guarantee sequential execution in the same way as reserved concurrency set to 1, so the same concurrency issues persist without the shared state management of ElastiCache.

Option D is incorrect because Lambda layers are meant for sharing code and dependencies, not for persistent data storage across invocations. Data stored in the function's execution environment or layers is lost when the Lambda function's environment is recycled or a new instance is invoked. Layers are also immutable and designed for read-only access during execution.

Therefore, the most robust solution is to modify the function to leverage Amazon ElastiCache for storing and retrieving the rolling average values, ensuring accuracy and persistence even with concurrent invocations. This approach allows the function to scale efficiently while maintaining data consistency.

Relevant Resources:

AWS Lambda Concurrency:https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html
Amazon ElastiCache:https://aws.amazon.com/elasticache/
AWS Lambda Layers:https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html

Question: 75

A developer is writing unit tests for a new application that will be deployed on AWS. The developer wants to validate all pull requests with unit tests and merge the code with the main branch only when all tests pass.

The developer stores the code in AWS CodeCommit and sets up AWS CodeBuild to run the unit tests. The developer creates an AWS Lambda function to start the CodeBuild task. The developer needs to identify the CodeCommit events in an Amazon EventBridge event that can invoke the Lambda function when a pull request is created or updated.

Which CodeCommit event will meet these requirements?

A.

```
{
    "source": ["aws.codecommit"],
    "detail": {
        "event": ["pullRequestMergeStatusUpdated"],
    }
}
```

B.

```
{
    "source": ["aws.codecommit"],
    "detail": {
        "event": ["pullRequestApprovalRuleCreated"]
    }
}
```

C.

```
{
    "source": ["aws.codecommit"],
    "detail": {
        "event": ["pullRequestSourceBranchUpdated", "pullRequestCreated"]
    }
}
```

D.

```
{
    "source": ["aws.codecommit"],
    "detail": {
        "event": ["pullRequestUpdated", "pullRequestSourceBranchCreated"]
    }
}
```

Answer: C

Explanation:

C. Events in answer D are not real. A & B are clearly wrong since two events are required.

Question: 76

A developer deployed an application to an Amazon EC2 instance. The application needs to know the public IPv4 address of the instance.

How can the application find this information?

A.Query the instance metadata from http://169.254.169.254/latest/meta-data/.
B.Query the instance user data from http://169.254.169.254/latest/user-data/.
C.Query the Amazon Machine Image (AMI) information from http://169.254.169.254/latest/meta-data/ami/.
D.Check the hosts file of the operating system.

Answer: A

Explanation:

The correct answer is A because instance metadata is specifically designed to provide information about the EC2 instance itself, including its public IPv4 address, without requiring any external API calls or configuration files. This information is accessible from within the instance via a specific, non-routable IP address (169.254.169.254). This address is consistent across all AWS regions and instance types, making it a reliable and standardized way to access instance metadata. The metadata includes details such as instance ID, public and private IP addresses, security groups, IAM role, and more.

Option B is incorrect because user data allows you to pass scripts or configuration data to the instance during launch. It is not meant for retrieving dynamic instance information like the public IP address. While you could store the public IP in user data during instance creation, this information would not update if the IP changes.

Option C is incorrect because it targets AMI information. While the AMI used to launch the instance is part of the metadata, it does not directly provide the public IP address. The AMI contains the base operating system and software configuration, not dynamic runtime information.

Option D is incorrect. The hosts file maps hostnames to IP addresses. While it might contain an entry for a local hostname, it does not typically contain the public IPv4 address assigned by AWS. Moreover, relying on the hosts file would make the application dependent on manual configuration, defeating the purpose of retrieving this information programmatically. Querying the hosts file is also generally unreliable in dynamic cloud environments.

Using instance metadata is the most efficient, reliable, and secure way for an EC2 instance to discover information about itself within the AWS environment.

Refer to these AWS documentation links for more information:

Instance Metadata and User Data
Accessing Instance Metadata

Question: 77

An application under development is required to store hundreds of video files. The data must be encrypted within the application prior to storage, with a unique key for each video file.

How should the developer code the application?

A.Use the KMS Encrypt API to encrypt the data. Store the encrypted data key and data.

B.Use a cryptography library to generate an encryption key for the application. Use the encryption key to encrypt the data. Store the encrypted data.

C.Use the KMS GenerateDataKey API to get a data key. Encrypt the data with the data key. Store the encrypted data key and data.

D.Upload the data to an S3 bucket using server side-encryption with an AWS KMS key.

Answer: C

Explanation:

The best approach for encrypting hundreds of video files client-side with unique keys using AWS KMS is option C. Here's why:

Requirement Alignment: The requirement is to encrypt data before storing it, with a unique key per file. This eliminates server-side encryption options directly.

KMS for Key Management: AWS KMS is designed for secure key management. Instead of creating and managing keys manually (Option B), KMS offers a secure, managed service to create, store, and control the encryption keys used to encrypt your data.

GenerateDataKey API: The GenerateDataKey API specifically addresses the scenario where you need a unique key for encrypting a large number of data objects. It generates a data key (used for encrypting the video file) and encrypts it using a KMS key you control.

Envelope Encryption: This approach utilizes envelope encryption, a standard security practice. You use a data key generated and encrypted by KMS to actually encrypt the data. The encrypted data key is then stored alongside the encrypted data. This separation prevents direct exposure of the KMS key.

Security Benefits: KMS provides audited key usage, rotation, and centralized control. It removes the burden of key generation and storage from the application.

Why other options are less suitable:

A (KMS Encrypt): While Encrypt can encrypt small amounts of data directly, it's inefficient and not recommended for encrypting large files like video files directly. It's better to encrypt a symmetric key. **B (Cryptography Library):** Managing keys yourself using a cryptography library is a security risk. You'd have to handle key generation, storage, rotation, and access control, which KMS handles for you.
D (S3 Server-Side Encryption): Server-side encryption happens after the data reaches S3. The requirement is to encrypt before storage in the application.

In summary, using GenerateDataKey allows you to offload key management to AWS KMS, providing a more secure and scalable solution while adhering to the requirement of unique keys for each video file and client-side encryption.

Further research:

AWS KMS Best Practices: https://docs.aws.amazon.com/kms/latest/developerguide/best-practices.html AWS KMS GenerateDataKey API:
https://docs.aws.amazon.com/kms/latest/APIReference/API_GenerateDataKey.html
Envelope Encryption: https://aws.amazon.com/blogs/security/how-to-protect-data-at-rest-with-aws-key-management-service-and-envelope-encryption/

Question: 78

A company is planning to deploy an application on AWS behind an Elastic Load Balancer. The application uses an HTTP/HTTPS listener and must access the client IP addresses.

Which load-balancing solution meets these requirements?

A.Use an Application Load Balancer and the X-Forwarded-For headers.
B.Use a Network Load Balancer (NLB). Enable proxy protocol support on the NLB and the target application.
C.Use an Application Load Balancer. Register the targets by the instance ID.
D.Use a Network Load Balancer and the X-Forwarded-For headers.

Answer: A

Explanation:

The correct answer is A: Use an Application Load Balancer and the X-Forwarded-For headers.

Here's why:

Application Load Balancer (ALB): ALBs operate at the application layer (Layer 7) of the OSI model, enabling them to inspect HTTP headers. This capability is crucial for accessing the client's IP address in HTTP/HTTPS traffic.

X-Forwarded-For Header: When a client connects to an ALB, the ALB adds an X-Forwarded-For header to the HTTP request. This header contains the client's IP address, allowing the application to retrieve it.

HTTP/HTTPS Listener: The question explicitly states that the application uses an HTTP/HTTPS listener, making an ALB the appropriate choice for handling this type of traffic.

Let's examine why the other options are incorrect:

B. Network Load Balancer (NLB) with Proxy Protocol: NLBs operate at the transport layer (Layer 4). While they can forward the client's connection to the target, using the proxy protocol necessitates that the backend application must be able to interpret the proxy protocol header, adding complexity. An ALB's X-Forwarded-For header approach is simpler and more standard for HTTP/HTTPS traffic.

C. Application Load Balancer with Instance ID: Registering targets by instance ID is a valid configuration but doesn't directly address accessing the client's IP address. It only affects how the ALB routes traffic to the instances.

D. Network Load Balancer with X-Forwarded-For: NLBs do not automatically add or process X-Forwarded-For headers. They forward the TCP connection, leaving it to the backend application to manage headers. Therefore, this option is incorrect.

In summary, the most straightforward and appropriate solution for accessing client IP addresses in an HTTP/HTTPS application behind a load balancer on AWS is to use an Application Load Balancer and read the X-Forwarded-For header.

Supporting documentation:

Application Load Balancers - Retrieving Client IP Address:
Network Load Balancers - Proxy Protocol:

Question: 79

A developer wants to debug an application by searching and filtering log data. The application logs are stored in Amazon CloudWatch Logs. The developer creates a new metric filter to count exceptions in the application logs. However, no results are returned from the logs.

What is the reason that no filtered results are being returned?

A.A setup of the Amazon CloudWatch interface VPC endpoint is required for filtering the CloudWatch Logs in the VPC.

B.CloudWatch Logs only publishes metric data for events that happen after the filter is created.

C.The log group for CloudWatch Logs should be first streamed to Amazon OpenSearch Service before metric filtering returns the results.

D.Metric data points for logs groups can be filtered only after they are exported to an Amazon S3 bucket.

Answer: B

Explanation:

The correct answer is B: CloudWatch Logs only publishes metric data for events that happen after the filter is created.

Here's a detailed justification:

CloudWatch Logs operates on a real-time, forward-looking basis for metric filters. When a metric filter is created in CloudWatch Logs, it begins monitoring incoming log events from that point forward. It does not retroactively process existing log data. The filter is applied to newly ingested logs. This behavior is by design to ensure efficient and scalable log processing.

If a developer creates a metric filter and immediately expects to see results from past log events, they will be disappointed. The filter will only begin to collect data once new log entries that match the filter criteria are written to the log group.

Option A is incorrect because a VPC endpoint for CloudWatch Logs is relevant when an application within a VPC needs private access to CloudWatch Logs, bypassing the public internet. It doesn't directly impact the functionality of metric filters.

Option C is incorrect because while streaming logs to Amazon OpenSearch Service (formerly Elasticsearch Service) is a valid way to perform log analysis and search, it's not a prerequisite for using metric filters within CloudWatch Logs itself. Metric filters are a built-in feature of CloudWatch Logs that operate independently of OpenSearch Service.

Option D is incorrect because exporting log groups to Amazon S3 is primarily for long-term archival and more complex analytical processing of log data. While S3-exported logs can subsequently be analyzed, it is not required for metric filters to function. The metric filter results are independent of log export to S3. Metric filters operate directly on logs stored in CloudWatch Logs.

Therefore, the absence of results strongly indicates that the newly created metric filter has not yet encountered any new log events that match its criteria since its creation. The solution would involve waiting for new exception logs to be generated and then checking if the metric filter is correctly processing them.

Authoritative links:

Using Metric Filters - Amazon CloudWatch Logs (Specifically, note how filters monitor incoming logs). Create Metric Filters from Log Groups - Amazon CloudWatch Logs

Question: 80

A company is planning to use AWS CodeDeploy to deploy an application to Amazon Elastic Container Service (Amazon ECS). During the deployment of a new version of the application, the company initially must expose only 10% of live traffic to the new version of the deployed application. Then, after 15 minutes elapse, the company must route all the remaining live traffic to the new version of the deployed application.

Which CodeDeploy predefined configuration will meet these requirements?

A.CodeDeployDefault.ECSCanary10Percent15Minutes
B.CodeDeployDefault.LambdaCanary10Percent5Minutes
C.CodeDeployDefault.LambdaCanary10Percentl15Minutes
D.CodeDeployDefault.ECSLinear10PercentEvery1Minutes

Answer: A

Explanation:

The correct answer is **A. CodeDeployDefault.ECSCanary10Percent15Minutes**.

Here's a detailed justification:

CodeDeploy offers predefined deployment configurations to manage traffic shifting during application updates. The scenario requires a canary deployment strategy for an application running on Amazon ECS (Elastic Container Service). A canary deployment involves initially routing a small percentage of traffic to the new version of the application, monitoring its performance, and then gradually shifting more traffic if everything is stable.

Option A, CodeDeployDefault.ECSCanary10Percent15Minutes, is specifically designed for canary deployments to ECS. It first routes 10% of the traffic to the new version. After a waiting period of 15 minutes, it automatically routes the remaining 90% of the traffic to the new version. This perfectly aligns with the requirement of initially exposing only 10% of live traffic for 15 minutes and then shifting the remaining traffic.

Option B, CodeDeployDefault.LambdaCanary10Percent5Minutes, is for AWS Lambda deployments, not ECS. Furthermore, it only waits for 5 minutes, which does not satisfy the 15-minute waiting period.

Option C, CodeDeployDefault.LambdaCanary10Percentl15Minutes, has an invalid configuration name. Even if it were a valid Lambda configuration, the same issues as Option B applies: it's for Lambda and not ECS.

Option D, CodeDeployDefault.ECSLinear10PercentEvery1Minutes, uses a linear deployment strategy where traffic is shifted in increments (10% every 1 minute). This is not a canary deployment, and it doesn't directly fulfill the specific requirement of 10% for 15 minutes followed by 100%.

Therefore, only option A precisely meets the given requirements. It's an ECS-specific predefined configuration that implements a canary deployment with the desired 10% initial traffic exposure and a 15-minute waiting period before shifting all the remaining traffic.

Authoritative links for further research:

AWS CodeDeploy Documentation:https://docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html
CodeDeploy Deployment Configurations:
https://docs.aws.amazon.com/codedeploy/latest/userguide/deployment-configurations.html
Canary Deployments with CodeDeploy:https://aws.amazon.com/blogs/devops/performing-canary-deployments-with-aws-codedeploy/