Interview Preparation Q&A (Level Wise)

Easy interview questions

1. What is AWS Lambda?
Answer: AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS) that allows developers to run code without provisioning or managing servers. Lambda automatically scales the application by running code in response to each trigger. The triggers can be modifications made to data in an Amazon S3 bucket, Amazon DynamoDB table updates, or HTTP requests sent to an Amazon API Gateway, among others.

2. How does AWS Lambda handle scaling?
Answer: AWS Lambda handles scaling automatically by adjusting its computing capacity in response to the incoming request rate. It runs each trigger independently, in parallel, and processes each trigger immediately, without requiring manual intervention or configuration. This allows the service to handle anything from a few requests per day to thousands per second.

3. What are the supported programming languages in AWS Lambda?
Answer: As of my last update, AWS Lambda supports multiple programming languages including Node.js, Python, Java, Go, C#, PowerShell, and Ruby. This support allows developers to write functions in a language they are comfortable with or that fits the requirements of the project.

4. Can AWS Lambda functions access resources in a VPC?
Answer: Yes, AWS Lambda functions can access resources within a Virtual Private Cloud (VPC) by setting up VPC access in the Lambda function's configuration. This allows Lambda functions to interact with resources like databases, cache instances, or internal services hosted within a VPC, providing enhanced security and flexibility.

5. How is AWS Lambda priced?
Answer: AWS Lambda pricing is based on the number of requests for your functions and the duration, the time it takes for your code to execute. AWS charges per 1 million requests and per 100ms of execution time at the memory size you choose for your functions.

Scenario-Based Questions:

6. Scenario: You need to process uploaded images to an S3 bucket using AWS Lambda. Describe how you would set up this process.
Answer: To process images uploaded to an S3 bucket, you would:

Create an AWS Lambda function with the required image processing logic, using a supported programming language.
Set up an S3 trigger in the Lambda console, specifying the bucket and event type (e.g., PUT or POST for uploads).
Grant the Lambda function the necessary permissions to read from and write to the S3 bucket through an execution role.
Once configured, every time an image is uploaded to the specified bucket, S3 will automatically trigger the Lambda function to process the image according to the defined logic.

7. Scenario: How would you use AWS Lambda to automatically resize images uploaded to an S3 bucket?
Answer:

Create a Lambda Function: Write a Lambda function in a supported language (e.g., Python) using a library like PIL (Python Imaging Library) for image resizing.
Set Up Triggers: Configure an S3 trigger on the Lambda function for ObjectCreated events. This trigger will invoke the Lambda function every time a new image is uploaded to the S3 bucket.
Execution Role Permissions: Ensure the Lambda function's execution role has permissions to read the original image from the S3 bucket and write the resized image back to S3.
Function Logic: The Lambda function will read the uploaded image, resize it as needed, and then save the resized image back to the S3 bucket, either replacing the original image or saving as a new file.
Test and Deploy: Test the Lambda function with sample image uploads to ensure it resizes images correctly, then deploy it for production use.

8. How do you secure a Lambda function?
Answer: Securing a Lambda function involves several practices:

Execution Role: Assign the least privilege necessary to the execution role to limit access to AWS resources.
Environment Variables: Use environment variables to store sensitive information, and encrypt them using AWS KMS.
VPC Configuration: If accessing resources within a VPC, configure VPC settings for Lambda to ensure it operates within a secure network environment.
Function Policies: Use resource-based policies to control who can invoke your Lambda function.
Monitoring and Logging: Enable CloudWatch Logs for monitoring and auditing Lambda function executions.

9. What is a cold start in AWS Lambda, and how does it affect performance?
Answer: A cold start occurs when an AWS Lambda function is invoked after not being used for an extended period. In such cases, AWS has to allocate an instance and load the function's code and dependencies, which takes additional time. This initial invocation delay can affect performance, especially for latency-sensitive applications. Subsequent requests are served by "warm" instances with lower latency. Optimizations include keeping the function's footprint small, using provisioned concurrency to keep instances warm, or deploying Lambda in a container image for potentially faster start times.

10. Explain the use of environment variables in AWS Lambda.
Answer: Environment variables in AWS Lambda are key-value pairs that you can define within your Lambda function's configuration and access within your function code. They are useful for:

Storing configuration settings and secrets, allowing you to separate your code from your configuration.
Customizing function behavior without changing the code, making your Lambda function more flexible and easier to manage across different environments (e.g., development, testing, production).
Securely storing sensitive information, such as database passwords or API keys, especially when used in conjunction with AWS KMS for encryption.

Medium level

1. How does AWS Lambda integrate with Amazon API Gateway?
Answer: AWS Lambda integrates with Amazon API Gateway to create RESTful APIs that trigger Lambda functions. This setup allows developers to build serverless applications that can execute Lambda functions in response to HTTP requests. API Gateway serves as the HTTP endpoint that proxies requests to Lambda, and it can handle tasks like request validation, user authentication, and rate limiting. This integration enables developers to build scalable, serverless web applications.

2. Explain the concept of Provisioned Concurrency in AWS Lambda.
Answer: Provisioned Concurrency is a feature in AWS Lambda that allows developers to prepare a specified number of Lambda function instances ready to respond immediately, eliminating cold starts. This feature is crucial for applications with strict latency requirements. By allocating provisioned concurrency, AWS ensures that there is always a pool of warm instances that can serve requests, improving the response time of Lambda functions.

3. What is AWS Lambda@Edge, and how is it used?
Answer: AWS Lambda@Edge allows developers to run Lambda functions at AWS Edge locations, closer to the end-users, by integrating with Amazon CloudFront. This setup is used to customize the content delivered through CloudFront distributions, enabling use

cases such as website personalization, SEO optimization, real-time image transformation, and security enhancements. By executing functions at the edge, applications can reduce latency and improve the user experience.

4. Describe how you can monitor and debug AWS Lambda functions.
Answer: Monitoring and debugging AWS Lambda functions can be achieved through AWS CloudWatch and AWS X-Ray. CloudWatch provides logs, metrics, and alarms to monitor function executions, performance statistics, and operational health. AWS X-Ray, on the other hand, offers insights into the function's behavior, allowing developers to trace and debug function execution paths, analyze latencies, and identify errors and performance bottlenecks.

5. How can you manage dependencies in AWS Lambda?
Answer: Managing dependencies in AWS Lambda involves packaging them along with your function code or using Lambda Layers. For packaging dependencies, developers include the library files in the deployment package uploaded to Lambda. Alternatively, Lambda Layers allow sharing common components across multiple functions, enabling easier management of libraries, custom runtimes, and other dependencies. This approach reduces the size of deployment packages and simplifies updates and maintenance.

Scenario-Based Questions:
6. Scenario: You are developing a serverless application with high traffic during specific hours. How would you optimize AWS Lambda costs while maintaining performance?
Answer: To optimize costs while maintaining performance for a high-traffic serverless application, consider the following strategies:

Use Provisioned Concurrency: Allocate provisioned concurrency during peak hours to ensure warm instances are available, reducing latency and improving user experience.
Fine-Tune Memory Allocation: Adjust the memory size based on the function's requirement to avoid over-provisioning. Use AWS Lambda's Power Tuning tool to identify the optimal memory configuration.
Implement Request Throttling: Use API Gateway to throttle requests and prevent the Lambda function from over-provisioning resources during unexpected spikes.
Utilize Reserved Instances: For predictable workload patterns, consider purchasing Savings Plans for a committed usage amount, which can lead to significant cost savings.

7. Scenario: How would you handle dependencies that require native binaries in AWS Lambda?
Answer: Handling dependencies with native binaries in AWS Lambda involves:

Compiling Binaries: Compile the binaries on an Amazon Linux environment (e.g., using an EC2 instance or a Docker container that mimics the Lambda execution environment) to ensure compatibility.
Package with Deployment: Include the compiled binaries in the Lambda deployment package or use Lambda Layers to manage and deploy these binaries across multiple Lambda functions.
Set Environment Variables: Configure necessary environment variables or system paths within Lambda to ensure the function can locate and use the binaries properly.

8. Explain the difference between synchronous and asynchronous invocation of a Lambda function.
Answer: In synchronous invocation, the caller waits for the Lambda function to process the event and return a response, suitable for real-time applications. API Gateway integration is a common use case. Asynchronous invocation does not wait for the function to complete; instead, the event is placed in a queue for Lambda to process when it can. This is ideal for tasks where immediate response to the user is not critical, like processing log files.

9. How do you secure sensitive data within a Lambda function?
Answer: Securing sensitive data within a Lambda function involves:

Environment Variables Encryption: Use AWS KMS to encrypt environment variables. Access the decrypted values programmatically within your Lambda function.
Use AWS Secrets Manager: Store sensitive data like database credentials in Secrets Manager and retrieve them dynamically within your Lambda function. This minimizes exposure and allows for centralized management of secrets.

10. Scenario: Describe a strategy to automate the deployment of AWS Lambda functions across multiple environments (e.g., development, testing, production).
Answer: To automate the deployment of AWS Lambda functions across environments, implement a CI/CD pipeline using AWS CodePipeline and AWS CodeBuild. The pipeline triggers on code changes, automatically builds the Lambda deployment package, and deploys it to the respective environment. Use AWS CloudFormation or the AWS Serverless Application Model (SAM) to define your Lambda functions and resources in templates, enabling consistent deployments across environments. Environment variables or separate configuration files can manage environment-specific settings, ensuring that each deployment is configured correctly for its target environment.

Hard level

1. Discuss the implications of Lambda's execution context reuse on secure coding practices.
Answer: Lambda's execution context reuse can significantly impact secure coding practices. When a Lambda function is invoked, AWS may keep the execution environment alive for some time after the function execution completes. This environment, including the global variables and file system (/tmp space), can be reused for subsequent invocations. Secure coding practices must ensure that sensitive data is not inadvertently exposed across invocations. Developers should avoid storing sensitive information in global or static variables without proper cleanup and should use temporary storage (/tmp) cautiously, ensuring sensitive data is deleted after use. Understanding and properly handling the execution context is crucial for maintaining data security and privacy in Lambda functions.

2. How can you minimize cold start times for AWS Lambda functions in a VPC?
Answer: Minimizing cold start times for Lambda functions in a VPC involves several strategies:

Provisioned Concurrency: Allocate provisioned concurrency to keep a specified number of instances warm and ready to execute.
Optimize VPC Configuration: Ensure that the Lambda function's VPC configuration is optimized by minimizing the number of ENIs (Elastic Network Interfaces) and NAT Gateways, as they can add latency to the initialization process.
Reduce Package Size: Smaller deployment packages load faster. Keep the deployment package size minimal by removing unnecessary dependencies.
Optimize Startup Code: Minimize the amount of code that runs at initialization by lazy loading dependencies and deferring the initialization of heavy resources until they are needed.

3. Explain how AWS Lambda's concurrency model works and how it affects the scalability of serverless applications.
Answer: AWS Lambda's concurrency model controls the number of instances processing events at any given time. When a Lambda function is invoked, AWS Lambda ensures that an instance of the function is available to process the event. If the function is already processing the maximum number of events as defined by its concurrency limit, additional invocations are throttled. The concurrency model affects scalability by:

Limiting the number of concurrent executions: This can prevent a Lambda function from overloading downstream resources but can also lead to throttling if the limit is too low.
Allowing for Reserved Concurrency: This reserves a portion of the AWS account's total concurrency limit exclusively for a function, ensuring that critical functions have the necessary resources.
Providing Provisioned Concurrency: This warms up the specified number of function instances, ensuring immediate response to events, which is critical for applications requiring consistent performance.

4. Discuss strategies for managing state in serverless architectures using AWS Lambda.
Answer: Managing state in serverless architectures, where Lambda functions are stateless, involves external state management solutions:

DynamoDB: A fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. It can be used to store and retrieve state information between Lambda function invocations.
ElastiCache: An in-memory data store service, compatible with Redis or Memcached, ideal for storing transient, non-persistent state data that can be accessed quickly by Lambda functions.
S3: An object storage service that can be used for storing state information that is infrequently accessed or requires long-term persistence.
Step Functions: A serverless function orchestrator that makes it easy to sequence AWS Lambda functions and multiple AWS services into business-critical applications through visual workflows.

5. How does Lambda's custom runtime feature work, and what are its use cases?
Answer: Lambda's custom runtime feature allows developers to run functions in languages or specific language versions not natively supported by AWS Lambda. It works by including a runtime API bootstrap file in the deployment package that interacts with the Lambda Runtime API. This bootstrap file is responsible for managing the lifecycle of the function's invocation. Use cases for custom runtimes include:

Running Functions in Unsupported Languages: Enables the use of languages not directly supported by AWS Lambda.
Specific Language Versions: Allows running functions in specific or newer versions of a language before official support is rolled out by AWS Lambda.
Custom Execution Environments: Developers can customize the execution environment to include specific libraries or security settings.
Scenario-Based Questions:

6. Scenario: Design a serverless architecture for processing real-time streaming data using AWS Lambda. How would you ensure minimal latency and high availability?
Answer: To design a serverless architecture for processing real-time streaming data with minimal latency and high availability, consider the following:

Use AWS Kinesis: Capture and process streaming data in real-time. Lambda functions can be triggered by Kinesis to process data as it arrives.
Lambda Function Optimization: Optimize Lambda functions for quick execution. Keep the code lean, use Provisioned Concurrency to avoid cold starts, and adjust memory allocation based on performance testing.
Multi-Region Deployment: Deploy the Lambda functions and Kinesis streams in multiple regions to provide high availability and disaster recovery capabilities.
Monitor and Scale: Use CloudWatch to monitor the performance and throughput of Kinesis streams and Lambda functions. Set up alarms to scale the processing power and concurrency based on demand.

7. Scenario: You're tasked with optimizing an AWS Lambda function that interacts with DynamoDB to reduce latency. What steps would you take?
Answer: To optimize latency for a Lambda function interacting with DynamoDB:

Enable DynamoDB Accelerator (DAX): Implement DAX for caching read operations, significantly reducing read latency.
Optimize Lambda Execution: Review and optimize the Lambda function's code to ensure efficient interaction with DynamoDB, including using batch operations and avoiding unnecessary read/write operations.
Adjust Provisioned Throughput: Adjust the provisioned read/write capacity units for DynamoDB to match the workload or enable DynamoDB Auto Scaling to adjust capacity automatically.
Use Provisioned Concurrency for Lambda: To avoid cold starts, use provisioned concurrency for the Lambda function, ensuring that function instances are always warm and ready to execute.

8. Describe how to implement transactional workflows involving multiple AWS Lambda functions.
Answer: Implementing transactional workflows involving multiple AWS Lambda functions can be achieved using AWS Step Functions. Step Functions allow you to orchestrate multiple AWS services into serverless workflows. For transactional workflows:

Define a State Machine: Use Step Functions to define a state machine that represents the workflow, including decision logic, parallel execution paths, and error handling.
Lambda Integration: Incorporate Lambda functions as tasks within the state machine. Step Functions can pass data between functions and control execution based on the functions' output.
Implement Compensation Transactions: For rollback capabilities in case of errors, design Lambda functions that can reverse the workflow's actions, ensuring data integrity across distributed transactions.
Error Handling: Use Step Functions' built-in error handling to manage retries, catch exceptions, and execute fallback logic as necessary.

9. How do you implement blue/green deployment for AWS Lambda functions?

Answer: Implementing blue/green deployment for AWS Lambda functions can involve several approaches:

Using Aliases and Versions: Publish a new version of the Lambda function (green) while the current version remains active (blue). Use aliases to point to the blue and green versions and gradually shift traffic from blue to green by adjusting the alias.
AWS CodeDeploy: Utilize AWS CodeDeploy to automate the deployment process, including traffic shifting between the blue and green environments. CodeDeploy can manage the rollout, monitor health during deployment, and automatically roll back if issues are detected.
SAM/CloudFormation: Use AWS SAM or CloudFormation templates to manage Lambda deployments, defining the blue and green environments as separate resources and manually shifting traffic by updating the stack.

10. Scenario: Explain how to secure a serverless application that uses AWS Lambda to process sensitive data, focusing on encryption, access control, and auditing.

Answer: To secure a serverless application processing sensitive data:

Encryption:
Use AWS KMS to encrypt sensitive data before storing it in S3 or DynamoDB. Ensure Lambda functions have the necessary permissions to decrypt the data for processing.
Enable encryption at rest and in transit for all AWS services involved in the application.
Access Control:
Implement least privilege access by carefully crafting IAM roles for Lambda functions, ensuring they have only the permissions needed to perform their tasks.
Use resource-based policies to restrict who can invoke the Lambda function.
Auditing:
Enable AWS CloudTrail to log API calls and Lambda execution activity, providing an audit trail of function invocations and data access.
Use AWS Config to monitor and record compliance of AWS resources, including Lambda, with desired configuration policies.
Network Security:
Place Lambda functions within a VPC if they need to access internal resources, ensuring secure network isolation.
Utilize API Gateway for HTTP endpoints, enabling TLS encryption and optionally integrating with AWS WAF for additional security measures against common web exploits.