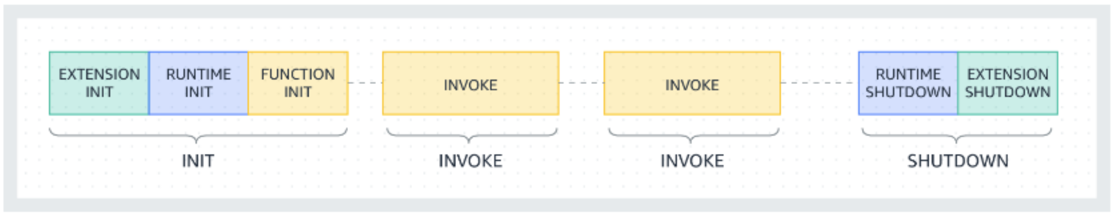
AWS Lambda

- AWS Compute service
- Virtual functions – no servers to manage!
- Runs code on-demand.
- Scales automatically
- you are responsible only for your code.
- Ideal for applications that need to scale up/down rapidly.
- A Lambda function always runs inside a VPC owned by the Lambda service.
- Lambda controls security, monitoring, and maintenance.
- Lambda offers built-in HTTP(S) endpoint support through *function URLs*.
- Use cases:
    - File Processing: use Amazon S3 to trigger lambda data processing.
    - Stream Processing: lambda + kinesis to process real time steaming data.
    - Web Apps: lambda + other AWS services
    - IoT backends
    - Mobile backends

Lambda Concepts

- Function: a resource that invoke to run code
- Trigger: a resource that invoke lambda function
- Event: JSON formatted document contains data that lambda processes
- Execution Environment: a secure & isolated runtime env for lambda
- Deployment package: lambda uses 2 types: - .zip file or container image.
- Layer: .zip file that can contain libs, custom runtime, data, config files.
- Extension: lambda uses extension to integrate with monitoring, observability, security, and governance tools.
- Concurrency: number of requests currently lambda is serving.
- Qualifier: is a version or alias
- Destination: a resource where lambda can send events from async invocations
- Cold start: When Lambda must initialize a new environment in order to carry out an invocation, this is known as a cold start.

Lambda execution environment

- While creating lambda function, you need to specify amount of memory available & maximum execution time allowed.
- Lambda uses this info to create execution environment.
  Lambda execution environment lifecycle
- 

Lambda Deployment Package

- Two types: .zip or container image
- Container Image:
    - includes the base operating system, the runtime, Lambda extensions, your application code, and its dependencies.

    - Upload container image to ECR.
    - To deploy, specify this ECR URL
    - Cannot use layers. Everything in container image
- .zip file
    - A .zip file archive includes your application code and its dependencies.
    - To deploy lambda function, you need to upload .zip file in Amazon S3
    - Can use layers to package libs, custom runtime, dependencies that independently from code.

IaC tools for Lambda

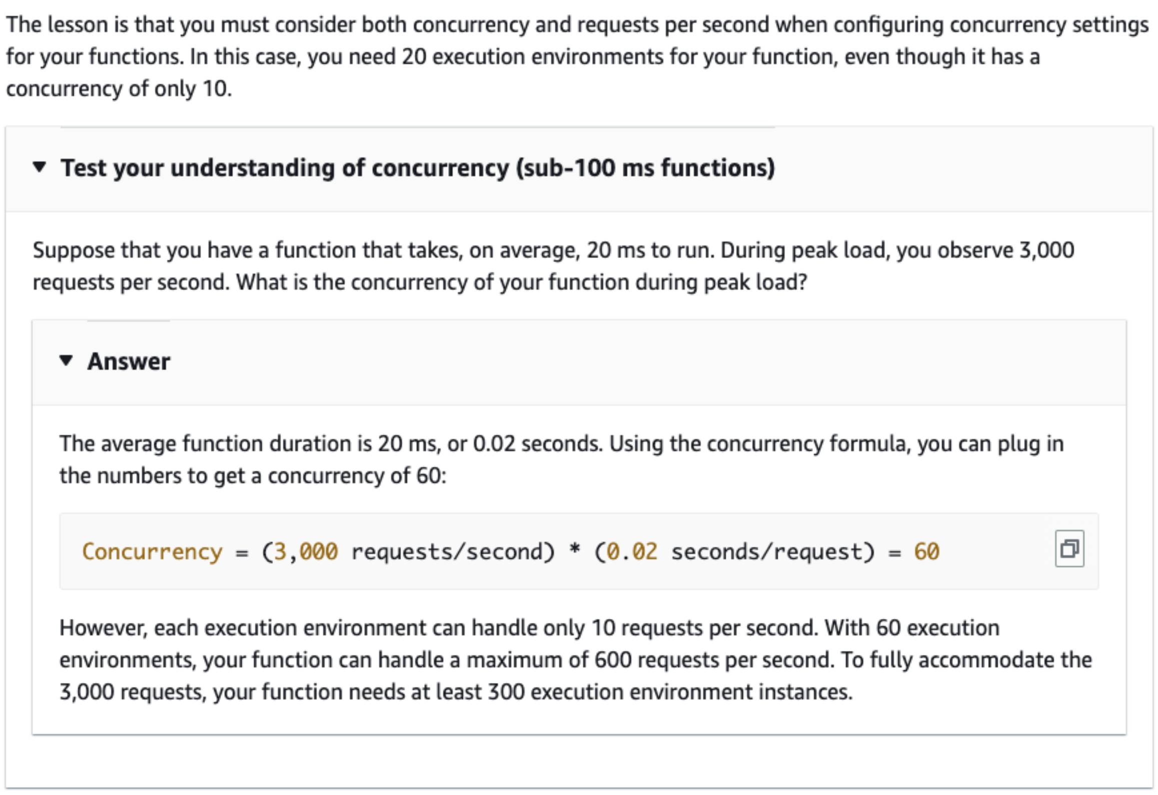
- Most of time lambda runs as a part of a serverless application with DB, queue, storage. To automate deployment process and make it easy. IaC tools are helpful.
- Tools are:
    - SAM
    - Cloud Development Kit - code-first approach
- Both AWS SAM & CDK uses CloudFormation to build and deploy infrastructure code.

Lambda VPC

- Lambda always runs inside a VPC owned by the Lambda service.
- By default, your Lambda function is launched outside your own VPC (in an AWS-owned VPC.
- Lambda applies network access and security rules to this VPC and Lambda maintains and monitors the VPC automatically.
- By default, lambda cannot access resources in your VPC.
- Hyperplane ENI is used by lambda to access resources in your VPC.
- VPC endpoints are used to connect to other AWS services privately.
- NAT gateways in public subnet are used to give lambda access to internet.

Concurrency Control

- Concurrency is number of requests lambda is currently serving at any given time.
- When function is invoked, lambda allocates an instance to process the event.
- When this processing is finished, it handles another request.
- But, if function is invoked while processing first request, lambda allocates another instance to new request, increasing function's concurrency.
- Default limit is **1000** concurrent executions **across all functions in an AWS Region.**
- **each execution environment can handle only 10 requests per second.**
- **Concurrency = (average requests per second) * (average request duration in seconds)**
    - 100 req. per second of duration 1 sec.
      Concurrency = (100 request /sec) * (1 sec/request) = 100
      If average duration is 500ms, then
      Concurrency = (100 request /sec) * (0.5 sec/request) = 50
      5000 requests takes 200 ms to run then,
      Concurrency = (5000 request /sec) * (0.2 sec/request) = 1000



The lesson is that you must consider both concurrency and requests per second when configuring concurrency settings for your functions. In this case, you need 20 execution environments for your function, even though it has a concurrency of only 10.

▼ **Test your understanding of concurrency (sub-100 ms functions)**

Suppose that you have a function that takes, on average, 20 ms to run. During peak load, you observe 3,000 requests per second. What is the concurrency of your function during peak load?

▼ **Answer**

The average function duration is 20 ms, or 0.02 seconds. Using the concurrency formula, you can plug in the numbers to get a concurrency of 60:

```
Concurrency = (3,000 requests/second) * (0.02 seconds/request) = 60
```

However, each execution environment can handle only 10 requests per second. With 60 execution environments, your function can handle a maximum of 600 requests per second. To fully accommodate the 3,000 requests, your function needs at least 300 execution environment instances.

- Function experience throttling (i.e. starts dropping requests), if run out of available concurrency.
- To ensure critical functions get concurrency what they need, use below :
  - Reserved concurrency: Reserve portion of account's concurrency for a critical function.only uses concurrency from its dedicated subset.
  - Provisioned Concurrency: pre-initialize env instances for a function and keep them always running. This reduces cold start latencies. Application Auto Scaling can manage concurrency (schedule or target utilization)
- maximum concurrency and reserved concurrency are different.
- make sure reserved concurrency is greater than or equal to the total maximum concurrency for all Amazon SQS event sources on the function. Otherwise lambda throttle messeges.
- There are two level of concurrency quotas : at account level (1000) & at function level (900)
- Lambda automatically scales up. However, to protect against over-scaling in response to sudden bursts of traffic, Lambda limits how fast your functions can scale. This is **concurrency scaling rate.**
- **In each AWS Region, and for each function, your concurrency scaling rate is 1,000 execution environment instances every 10 seconds**. In other words, every 10 seconds, Lambda can allocate at most 1,000 additional execution environment instances to each of your functions.
- Importantly, the concurrency scaling rate is a function-level limit. This means that each function in your account can scale independently of other functions.

Managing provisioned concurrency with Application Auto Scaling

- Application Auto Scaling can manage concurrency (schedule or target utilization)
- If predictable traffic use **scheduled scaling policy.**
- if unpredictable traffic and want function to maintain specific utilization percentage, use **target scaling policy.**
- Application Auto Scaling creates two alarms in CloudWatch.
- The first alarm - when the utilization of provisioned concurrency consistently exceeds 70% , Application Auto Scaling allocates more provisioned concurrency to reduce utilization.
- The second alarm - when utilization is consistently less than 63% (90 percent of the 70% target). When this happens, Application Auto Scaling reduces the alias's provisioned concurrency.

Invoking Lambda functions

- we can directly invoke lambda using console, API, CLI, SDK
- Other AWS services can invoke lambda in response to events, requests, schedules, by creating trigger.
- Event source mapping is created to invoke lambda from a stream or a queue.
- Event source mapping reads items from a stream or a queue and creates events containing batches of items to send to your Lambda function.
- We can choose to invoke a lambda function synchronously or asynchronously.

Lambda Synchronous Invocation:

- Lambda runs the function and waits for a response.
- CLI, SDK, API Gateway, Application Load Balancer
- Result is returned right away.
- Error handling must happen at client side. (retries, exponential backoffs, etc.)

  ```
  aws lambda invoke
  --function-name my-function
  --cli-binary-format raw-in-base64-out
  --payload '{ "key": "value" }' response.json
  ```

Lambda Asynchronous Invocation:

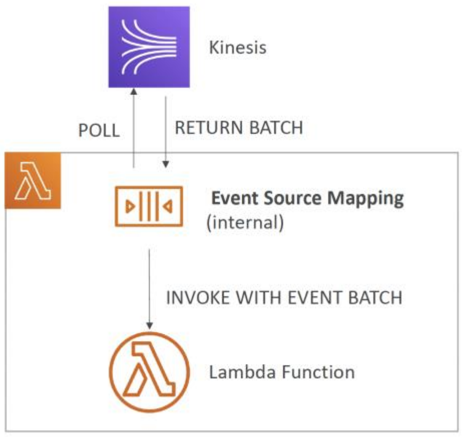
- Lambda queues the event for processing and returns a response immediately.
- The events are placed in an Event Queue
- Asynchronous invocations allow you to speed up the processing if you don't need to wait for the result (ex: you need 1000 files processed)
- Lambda handles retries if the function returns an error or is throttled.
- can configure error handling settings on a function, version, or alias.
- Can send events that failed processing to a dead-letter queue,
- Or send a record of any invocation to a destination.
- To invoke a function asynchronously, set the invocation type parameter to Event.

  ```
  aws lambda invoke \
    --function-name my-function \
    --invocation-type Event \
    --cli-binary-format raw-in-base64-out \
    --payload '{"key": "value"}' response.json
  ```
- The maximum amount of time Lambda retains an event in the asynchronous event queue, **up to 6 hours. After that lambda discards it.**
- **To save such events failed-event destination can be configured.**
- If the function returns an error, Lambda attempts to run it **two** more times.
- a one-minute wait between the first two attempts, two minutes between the second and third attempts.
- Function errors include errors returned by the function's code and errors returned by the function's runtime, such as timeouts.
- can define **destinations** for successful and failed event.
- ensure that your function's execution role also contains the relevant permissions.
  - Amazon SQS – A standard SQS queue.
  - Amazon SNS – A standard SNS topic.
  - AWS Lambda – A Lambda function.
  - Amazon EventBridge – An EventBridge event bus.
- **SQS FIFO queues and SNS FIFO topics are not supported.**
- If Lambda can't send a record to a destination you have configured, it sends a **DestinationDeliveryFailures** metric to Amazon CloudWatch.
- This can happen due to unsupported destination, size limits, permission errors.
- Lambda can be configured to send discarded events to a **dead-letter-queue (SNS or SQS)**

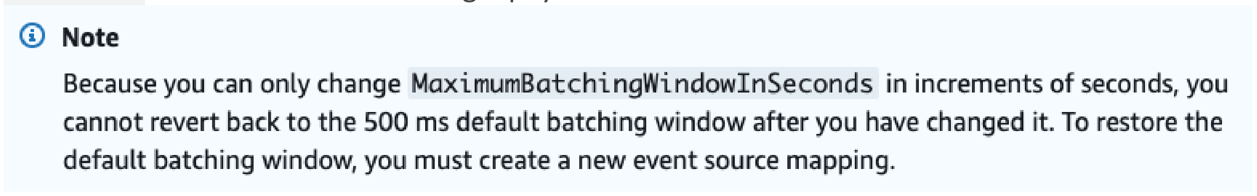
- For dead-letter queues, Lambda only sends the content of the event, without details about the response.
- If Lambda can't send a message to the dead-letter queue, it deletes the event and emits the DeadLetterErrors metric.
- AWS recommends you use destinations instead of DLQ now (but both can be used at the same time).
- **AWS X-Ray** can be used to see a connected view of each request as it's queued, processed by a Lambda function, and passed to the destination service.
- **If X-Ray is enabled, Lambda adds an X-Ray header to the request and passes the header to the destination service.**

Lambda event source mappings

- **Event source mappings** polls items from a stream or queue then batches records together into a single payload which then Lambda sends to your function.



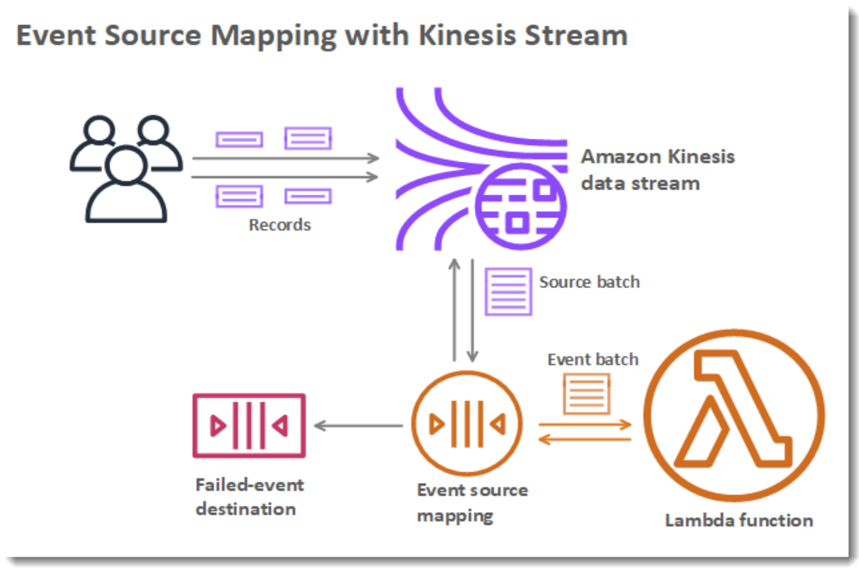
- **Sources can be: (**need **permissions** in the function's execution role to read items in source)
  - **Kinesis data stream**
  - **SQS, SQS FIFO**
  - **Amazon MQ**
  - **DynamoDB, DocumentDB**
  - **AWS managed Apache Kafka or Self-managed Apache Kafka**
- MaximumBatchingWindowInSeconds–amount of time to gather records into single payload
- BatchSize- number of records in single payload
-



Note

Because you can only change `MaximumBatchingWindowInSeconds` in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

The following example shows an event source mapping that reads from a Kinesis stream. If a batch of events fails all processing attempts, the event source mapping sends details about the batch to an SQS queue.



The event batch is the event that Lambda sends to the function. It is a batch of records or messages compiled from the items that the event source mapping reads up until the current batching window expires.

-
- For Kinesis and DynamoDB streams, An event source mapping creates an iterator for each shard, processes items **in order.**
- Processed items aren't removed from the stream (other consumers can read them)
- Low traffic: use batch window to accumulate records before processing.
- By default, if your function returns an error, the event source mapping **reprocesses the entire batch** until the function succeeds, or the items in the batch expire.
- To ensure in-order processing, the event source mapping pauses processing for the affected shard until the error is resolved.
- can process multiple batches in parallel.
- in-order processing is still guaranteed for each partition key
- Use **Event filtering** to control which records from a stream or queue Lambda sends to your function.
- can define up to five different filters for a single event source mapping.

Lambda maintains states of a function:

- **Pending**: After creating function, lambda sets status as Pending, during which it creates and configures resources for function such as VPC or EFS
- **Active**: Function can be invoked now
- **Failed**: Indicates that resource configuration or provisioning encountered an error.
- **Inactive:** function becomes inactive when it has been idle long enough

Recursive loop detection

- If a lambda function is configured to output to same service that invokes lambda, it's possible to create a recursive loop.
- It is turned on by default for SNS, SQS. (it's free). Doesn't detect for other services currently.
- This can cause unexpected charges being billing, lambda to scale and use all available capacity.
- Lambda Lambda uses AWS X-Ray tracing headers to detect recursive loops.
- Consider example where, SQS invokes lambda when item in added in queue, in response lambda writes to SQS, again SQS invokes lambda and so on. This is chain of request.

- If lambda function is invoked more than 16 times in the same chain of requests, then Lambda automatically stops the next function invocation in that request chain and notifies you.
- Responding to recursive loop detection:
  - reduce function's concurrency to Zero
  - disable or remove trigger that invokes lambda function
  - fix code
  - in case of SQS, make sure to configure dead letter queue on source queue.
  - In case of SNS, consider adding on-failure destination for lambda function.

Lambda – Function URL

- Dedicated HTTP(S) endpoint for your Lambda function
- A unique URL endpoint is generated for you (never changes)
  https://<url-id>. lambda-url. <region>. on.aws (dual-stack IPv4 & IPv6)
- Invoke via a web browser, curl, Postman, or any HTTP client, must have lambda:InvokeFunctionUrl permissions.
- Access your function URL through the public Internet only.
- Doesn't support PrivateLink (Lambda functions do support)
- Supports Resource-based Policies & CORS configurations.
- Can be applied to any function alias or to $LATEST (can't be applied to other function versions)
- Create and configure using AWS Console or AWS API
- You can **throttle** the rate of requests that your Lambda function processes through a function URL by configuring reserved concurrency.
- Reserved concurrency limits the number of maximum concurrent invocations for your function.
- maximum request rate per second (RPS) is equivalent to 10 times the configured reserved concurrency.
- E.g. if reserved concurrency = 100, then RPS is 1000.
- Whenever function concurrency exceeds reserved concurrency, functional URL returns 429
- If request exceeds 10 * RPS, then also **429 (throttle code).**
- In an emergency, you might want to reject all traffic to your function URL, in that case **deactivate functional URL.** (Set the reserved concurrency to zero.**).**
- AWS **CloudTrail** and Amazon **CloudWatch** to monitor your function URLs.

Lambda – Function URL Security

- using the AuthType parameter
- resource based policy – authorize other accounts / specific CIDR / IAM Principals
- Cross-Origin Resource Sharing (CORS) – call lambda function from another domain.
- The AuthType parameter determines how Lambda authenticates or authorizes requests to your function URL.
- AuthType options:
  - AWS_IAM - (IAM) to authenticate and authorize requests. Choose this option if you want only authenticated users and roles to invoke your function via the function URL. You must sign each HTTP request using **AWS Signature Version 4 (SigV4).**
  - NONE - doesn't perform any authentication before invoking function. However, resource-based policy is always in effect. Choose this option to allow public, unauthenticated access to your function URL. Don't have to sign your requests using SigV4.
- IAM Access Analyzer also monitors for new or updated permissions on your Lambda functions

Monitoring Lambda

1. **AWS CloudTrail**
   - Tracks call made by lambda and calls made to lambda.
   - **CloudTrail Event History** stores records of past 90 days.

  - Every event or log entry contains information about who generated the request.
  - Whether the request was made with root user or user credentials.
  - Whether the request was made on behalf of an IAM Identity Center user.
  - Whether the request was made with temporary security credentials for a role or federated user.
  - Whether the request was made by another AWS service.
  - A ***trail*** enables CloudTrail to deliver log files to an Amazon S3 bucket.
  - *CloudTrail Lake* lets you run SQL-based queries on your events.
  - CloudTrail Lake converts existing events in row-based JSON format to Apache ORC format.
  -

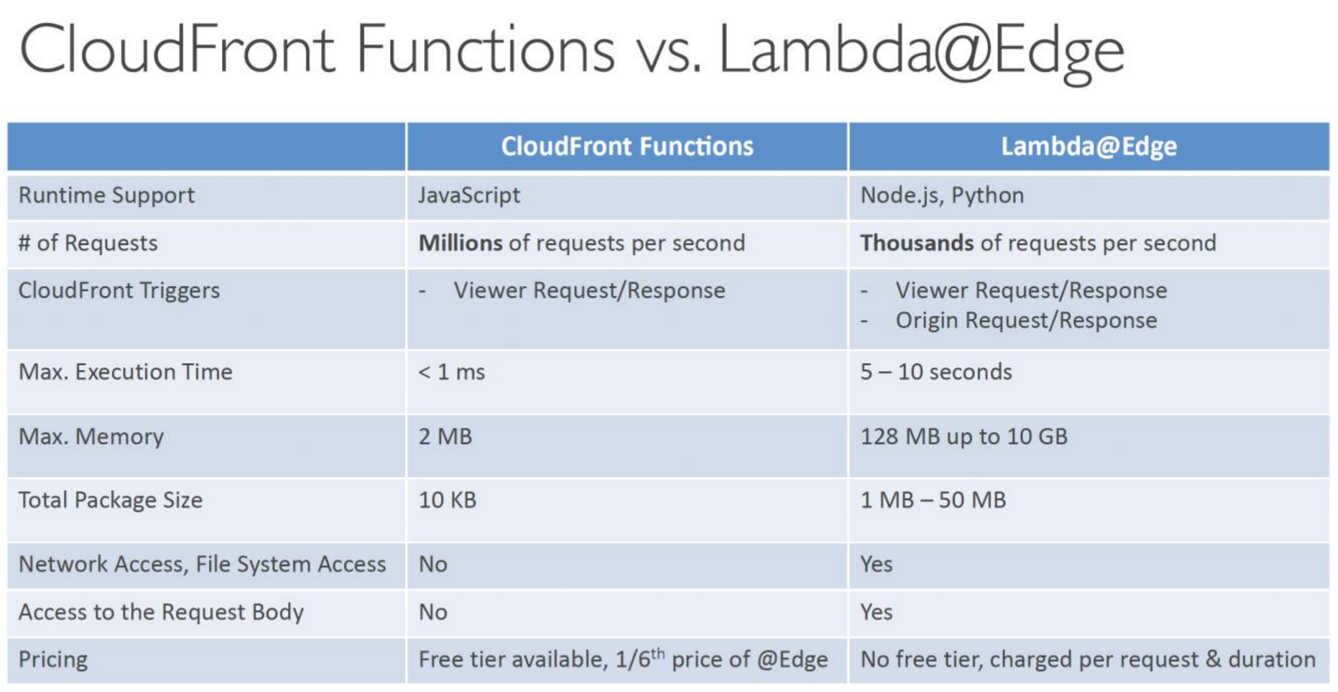
2. **CloudWatch**
   - AWS Lambda execution logs are stored in AWS CloudWatch Logs
   - Make sure your AWS Lambda function has an execution role with an IAM.
   - policy that authorizes writes to CloudWatch Logs
   - AWS Lambda metrics are displayed in AWS CloudWatch Metrics
   - Invocations, Durations, Concurrent Executions
   - Error count, Success Rates, Throttles
   - Async Delivery Failures
   - Iterator Age (Kinesis & DynamoDB Streams)
3. **AWS X-Ray**
   - Enable in Lambda configuration (Active Tracing)
   - Runs the X-Ray daemon for you.
   - Use AWS X-Ray SDK in Code
   - Ensure Lambda Function has a correct IAM Execution Role
   - The managed policy is called AWSXRayDaemonWriteAccess.
   - Environment variables to communicate with X-Ray:
     - _X_AMZN_TRACE_ID: contains the tracing header
     - AWS_XRAY_CONTEXT_MISSING: by default, LOG_ERROR
     - AWS_XRAY_DAEMON_ADDRESS: the X-Ray Daemon IP_ADDRESS:PORT

Using AWS Lambda with CloudFront Lambda@Edge

- Lambda@Edge is an extension of AWS Lambda that lets you deploy Python and Node.js functions at Amazon CloudFront edge locations.
- Up to 10,000 requests per second per Region
- Used to change CloudFront requests and responses:
  - Viewer Request – after CloudFront receives a request from a viewer.
  - Origin Request – before CloudFront forwards the request to the origin.
  - Origin Response – after CloudFront receives the response from the origin.
  - Viewer Response – before CloudFront forwards the response to the viewer.
- Author your functions in one AWS Region (us-east-1), then CloudFront replicates to its locations.

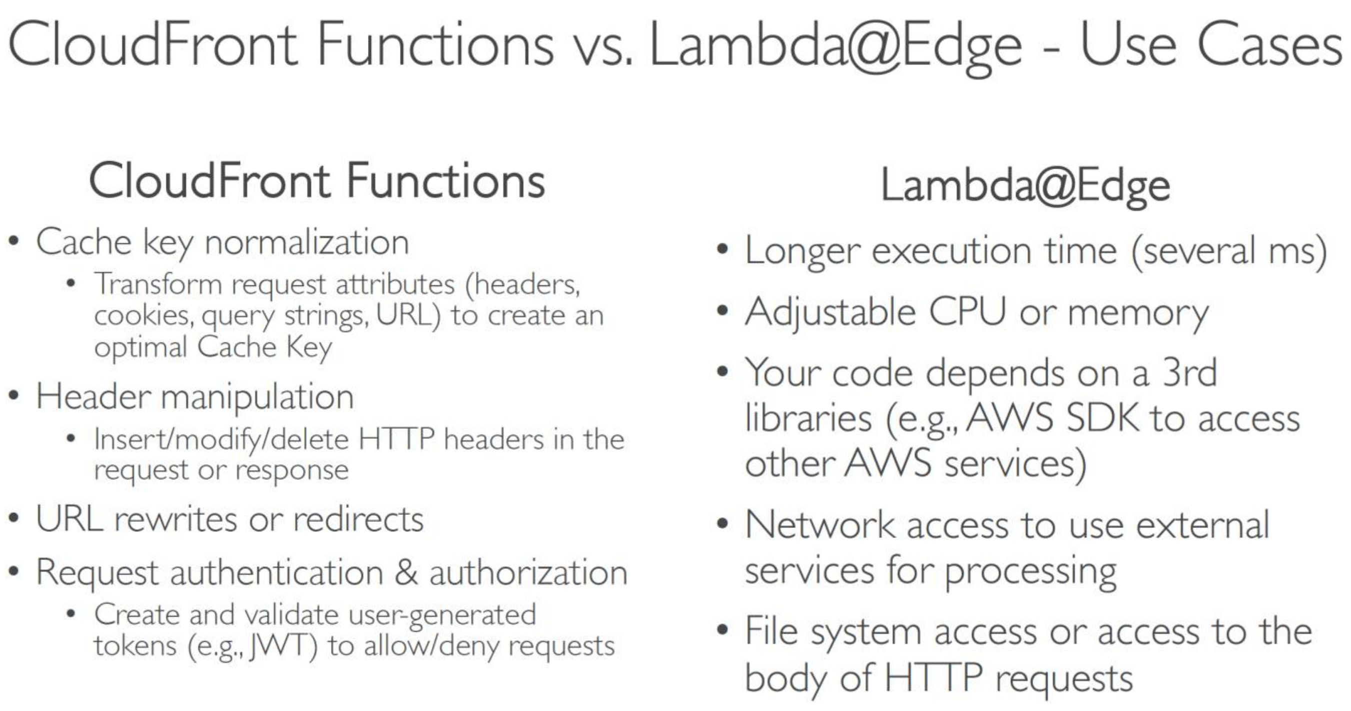
CloudFront Functions

- Lightweight functions written in JavaScript.
- For high-scale, latency-sensitive CDN customizations
- Sub-ms start-up times, millions of requests/second.
- Used to change Viewer requests and responses:
- Viewer Request: after CloudFront receives a request from a viewer.
- Viewer Response: before CloudFront forwards the response to the viewer.
- Native feature of CloudFront (manage code entirely within CloudFront)



CloudFront Functions vs. Lambda@Edge

	CloudFront Functions	Lambda@Edge
Runtime Support	JavaScript	Node.js, Python
# of Requests	**Millions** of requests per second	**Thousands** of requests per second
CloudFront Triggers	- Viewer Request/Response	- Viewer Request/Response - Origin Request/Response
Max. Execution Time	< 1 ms	5 – 10 seconds
Max. Memory	2 MB	128 MB up to 10 GB
Total Package Size	10 KB	1 MB – 50 MB
Network Access, File System Access	No	Yes
Access to the Request Body	No	Yes
Pricing	Free tier available, 1/6th price of @Edge	No free tier, charged per request & duration



CloudFront Functions vs. Lambda@Edge - Use Cases

CloudFront Functions

- Cache key normalization
  - Transform request attributes (headers, cookies, query strings, URL) to create an optimal Cache Key
- Header manipulation
  - Insert/modify/delete HTTP headers in the request or response
- URL rewrites or redirects
- Request authentication & authorization
  - Create and validate user-generated tokens (e.g., JWT) to allow/deny requests

Lambda@Edge

- Longer execution time (several ms)
- Adjustable CPU or memory
- Your code depends on a 3rd libraries (e.g., AWS SDK to access other AWS services)
- Network access to use external services for processing
- File system access or access to the body of HTTP requests

AWS Lambda Limits to Know - per region:

- **Execution:**
  - Memory allocation: 128 MB – 10GB (1 MB increments)
  - Maximum execution time: 900 seconds (15 minutes)
  - Environment variables (4 KB)
  - Disk capacity in the "function container" (in /tmp): 512 MB to 10GB.
  - Concurrency executions: 1000 (can be increased)
- **Deployment:**
  - Lambda function deployment size (compressed .zip): 50 MB.
  - Size of uncompressed deployment (code + dependencies): 250 MB
  - Can use the /tmp directory to load other files at startup
  - Size of environment variables: 4 KB

Read in detail :

https://docs.aws.amazon.com/lambda/latest/dg/with-kinesis.html

Best practices for working with AWS Lambda functions:

- Separate the Lambda handler from your core logic.
- Take advantage of execution environment reuse to improve the performance of your function.
- Initialize SDK clients and database connections outside of the function handler and cache static assets locally in the /tmp directory.
- To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications.
- If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.
- Use a keep-alive directive to maintain persistent connections because Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error.
- Use environment variables to pass operational parameters to your function.
- e.g. instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.
- Control the dependencies in your function's deployment package. package all of your dependencies with your deployment package.
- Minimize your deployment package size to its runtime necessities.
- Reduce the time it takes Lambda to unpack deployment packages authored in Java by putting your dependency .jar files in a separate /lib directory. This is faster than putting all your function's code in a single jar with a large number of .class files.
- Minimize the complexity of your dependencies. Prefer simpler frameworks that load quickly on execution environment start-up. E.g. use Dagger or Guice instead Spring Framework.
- Avoid using recursive code in your Lambda function,
- Do not use non-documented, non-public APIs in your Lambda function code.
- Write idempotent code. Your code should properly validate events and gracefully handle duplicate events.
- Avoid using the Java DNS cache. Lambda functions already cache DNS responses. If you use another DNS cache, then you might experience connection timeouts.
- Performance testing your Lambda function is a crucial. Increase in memory increases CPU.
- Load test your Lambda function to determine an optimum timeout value.
- Use most-restrictive permissions when setting IAM policies.
- Be familiar with Lambda quotas. Payload size, file descriptors and /tmp space are often overlooked when determining runtime resource limits.
- Be familiar with your upstream and downstream throughput constraints.
- Delete Lambda functions that you are no longer using.
- If you are using SQS Service as an event source, make sure the value of the function's expected invocation time does not exceed the Visibility Timeout value on the queue.
- This applies both to CreateFunction and UpdateFunctionConfiguration.
  - In case of **CreateFunction**, AWS Lambda will fail the function creation process.
  - In case of **UpdateFunctionConfiguration**, it could result in duplicate invocations of function.
- By default, Lambda invokes your function as soon as records are available. If the batch that Lambda reads from the event source has only one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a *batching window*. Before invoking the

function, Lambda continues to read records from the event source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB.
- Increase Kinesis stream processing throughput by adding shards.
- Use Amazon CloudWatch on IteratorAge to determine if your Kinesis stream is being processed.
- Monitor your usage of AWS Lambda by using AWS Security Hub.

Lambda resource access permissions:

- Every Lambda function has an IAM role called an execution role.
- To give other accounts and AWS services permission to use your Lambda resources, use a resource-based policy.
- Use identity-based policies to allow users to perform operations on Lambda functions.
- When a user tries to access a Lambda resource, Lambda considers both the user's identity-based policies and the resource's resource-based policy.
- When an AWS service such as Amazon Simple Storage Service (Amazon S3) calls your Lambda function, Lambda considers only the resource-based policy.

Lambda Layer

- A Lambda layer is a .zip file archive that contain library dependencies, a custom runtime, or configuration files.
- reduce the size of your deployment packages
- separate core function logic from dependencies.
- share dependencies across multiple functions
- use the Lambda console code editor.
- Lambda extracts the layer contents into the /opt directory in your function’s execution env.
- You can include up to five layers per function.
- The total unzipped size of the function and all layers cannot exceed the unzipped deployment package size quota of 250 MB.
- To create a layer, either upload the .zip file archive from your local machine or from S3
- Each layer has unique version
- To update the code or make other configuration changes, must create a new version.
- Can delete layer using layer version. Once deleted, it can no longer be used. Function that already uses it can have access to it.

Lambda Extensions

- uses extensions to integrate Lambda with their preferred tools for monitoring, observability, security, and governance.
- supports external and internal extensions
- external extension runs as an independent process in the execution environment and continues to run after the function invocation is fully processed.
- An internal extension runs as part of the runtime process
- You are charged for the execution time that the extension consumes
- The size of your function's extensions counts towards the deployment package size limit.
- total unzipped size of the function and all extensions cannot exceed 250MB
- Extensions can impact the performance of your function because they share function resources such as CPU, memory, and storage.
- Each extension must complete its initialization before Lambda invokes the function
- You can register up to 10 extensions for a function.
- This limit is enforced through the Register API call.

Lambda Telemetry API

- enables your extensions to receive telemetry data directly from Lambda.
- During function initialization and invocation, Lambda automatically captures telemetry, including logs, platform metrics, and platform traces.
- Extensions can use the Telemetry API to subscribe to three different telemetry streams:
  - **Platform telemetry** – Logs, metrics, and traces, which describe events and errors related to the execution environment runtime lifecycle, extension lifecycle, and function invocations.
  - **Function logs** – Custom logs that the Lambda function code generates.
  - **Extension logs** – Custom logs that the Lambda extension code generates.
-

Lambda quotas

Resource	**Quota**
Concurrent executions	1000
(.zip file archives) and layers.	75 GB
EFS	250
Function memory allocation	128 MB to 10,240 MB, in 1-MB increments.
Function timeout	900 seconds (15 minutes)
environment variables	4 KB
resource-based policy	20 KB
layers	5
concurrency scaling limit	for each function, 1000 execution every 10 seconds
invocatio payload	6 MB each for request and response (synchronous) 256 KB (asynchronous) 1 MB for the total combined size of request line and header values
Deploymwnt Package	50 MB (zipped, for direct upload) 250 MB (unzipped) This quota applies to all the files you upload, including layers and custom runtimes. 3 MB (console editor)
Container Image	16 KB
Container image settings size	16 KB
Container image code package size	10 GB (maximum uncompressed image size, including all layers)
Test events (console editor)	10
/tmp directory storage	Between 512 MB and 10,240 MB, in 1-MB increments
File descriptors	1024

Execution processes/threads	1024
Invocation requests per function per Region (synchronous)	Each instance of your execution environment can serve up to 10 requests per second. In other words, the total invocation limit is 10 times your concurrency limit. See Lambda function scaling.
Invocation requests per function per Region (asynchronous)	Each instance of your execution environment can serve an unlimited number of requests. In other words, the total invocation limit is based only on concurrency available to your function. See Lambda function scaling.
Invocation requests per function version or alias (requests per second)	10 x allocated provisioned concurrency Note This quota applies only to functions that use provisioned concurrency.
GetFunction API requests	100 requests per second. Cannot be increased.
GetPolicy API requests	15 requests per second. Cannot be increased.
Remainder of the control plane API requests (excludes invocation, GetFunction, and GetPolicy requests)	15 requests per second across all APIs (not 15 requests per second per API). Cannot be increased.

Remember :

- When you upload a deployment package or layer archive directly to Lambda, the size of the ZIP file is limited to 50 MB. To upload a larger file, store it in Amazon S3 and use the S3Bucket and S3Key parameters.
- Lambda SnapStart for Java can improve startup performance for latency-sensitive applications by up to 10x at no extra cost, typically with no changes to your function code.
- With SnapStart, Lambda initializes your function when you publish a function version. Lambda takes a Firecracker microVM snapshot of the memory and disk state of the initialized execution environment, encrypts the snapshot, and caches it for low-latency access. When you invoke the function version for the first time, and as the invocations scale up, Lambda resumes new execution environments from the cached snapshot instead of initializing them from scratch, improving startup latency.
- SnapStart helps you improve startup performance by up to 10x at no extra cost compared to Provisioned concurrency
- SnapStart works best when used with function invocations at scale.
- Configuring provisioned concurrency incurs charges to your AWS account.
- Use provisioned concurrency if your application has strict cold start latency requirements.