Serverless

Serverless.

- **Debunking the myths:**

  Serverless does not mean there are no servers

- **You, the developer:**
  1. **Don’t need to care about servers when coding.**
  2. **Don’t have to manage physical capacity.**

Without Serverless.

- **What is the right size for my server?**
- **What capacity is left on my server?**
- **How many servers shoul l provision?**
- **How do l handle hardware failures?**

- **What OS should my server run?**
- **Whom should have access to my servers?**
- **How do l detect a compromised server?**
- **How do l keep my server patched?**

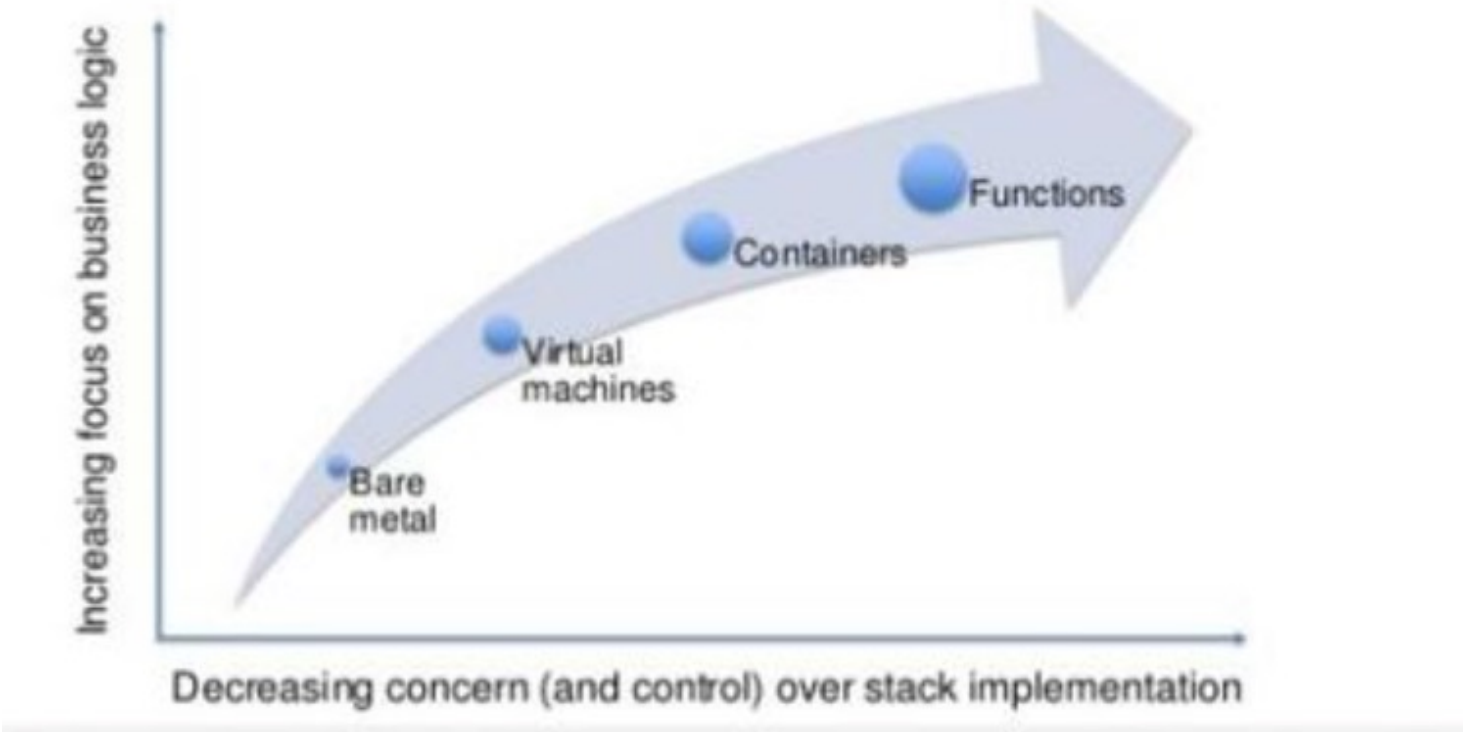
Serverless means ….

1. **No servers to provision or manage.**
2. **Scales with usage.**
3. **Pay for value.**
4. **Availability and fault tolerance built-in.**

- **Benefits:**
  1. **Greater agility.**
  2. **Less overhead.**
  3. **Increased scale.**
  4. **Better focus.**
  5. **Faster time to market.**

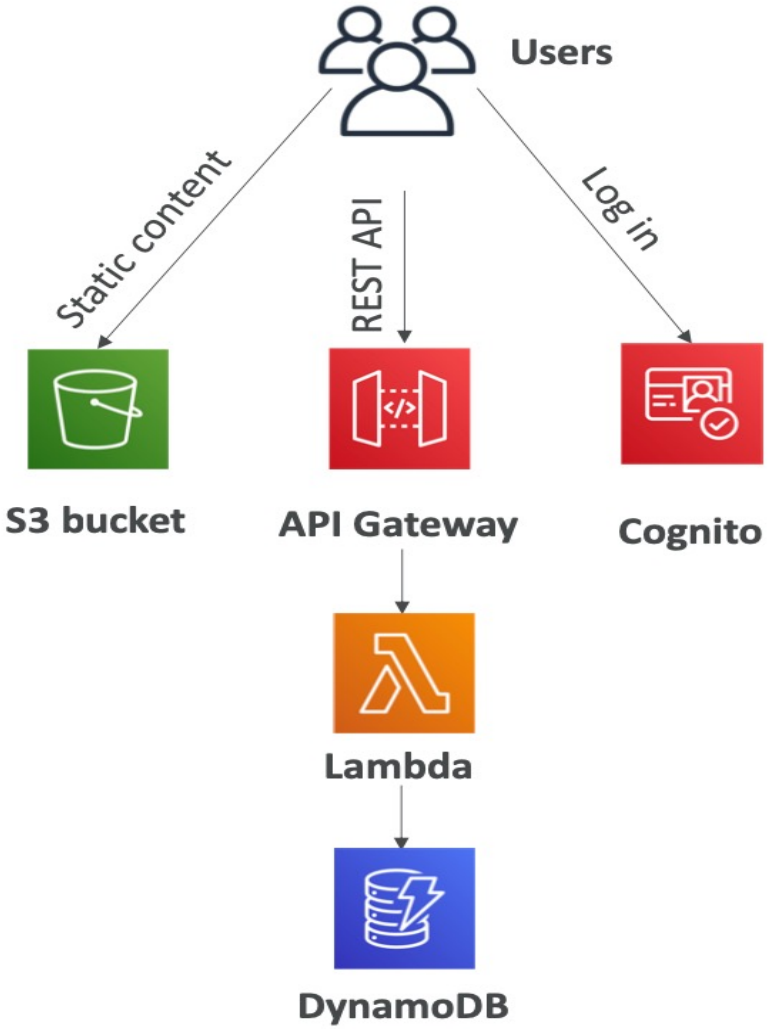
Developer focus.

- **Developers should focus on the product, not infrastructure.**



Serverless Services on AWS

- AWS Lambda. (Compute)
- DynamoDB. (NoSQL)
- AWS Cognito. (User Mgt.)
- API Gateway (HTTP/REST endpoints)
- S3 (Storage)
- SNS & SQS. (Messaging)
- AWS Kinesis Data Firehose
- Aurora Serverless (RDB)
- Step Functions (Orchestration)
- Fargate (Containers)
- **And more**



AWS Lambda Service

(Serverless compute)

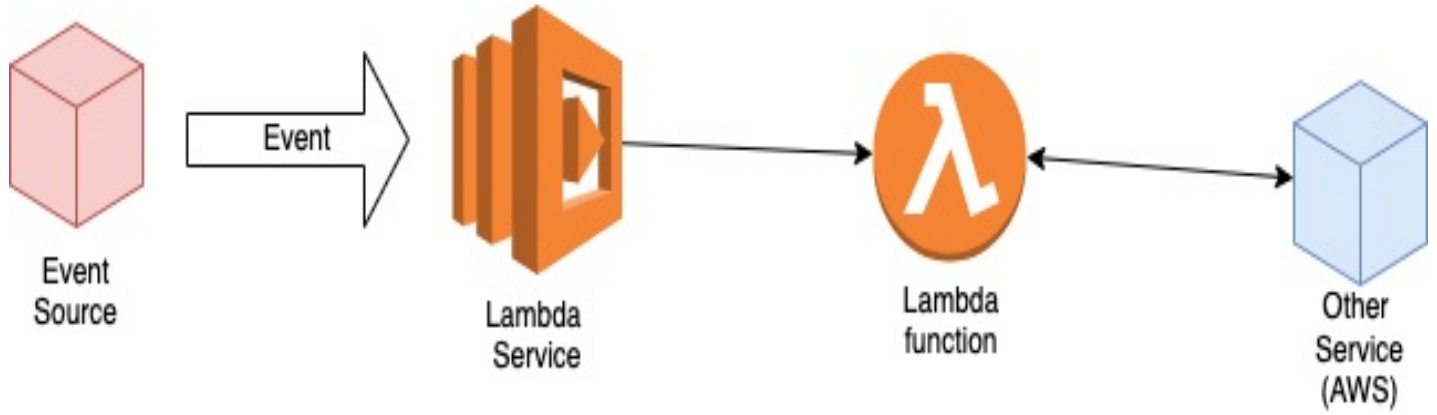
AWS Lambda

- **“Lambda is an event-driven, serverless computing platform provided by AWS. It is a computing service that runs code (a function) in response to events and automatically manages the computing resources (CPU, memory, networking) required by that code. It was introduced on November 13, 2014.” Wikipedia**

- **“Custom code that runs in an ephemeral container” Mike Robins**

- **FaaS (Functions as a Service)**
  - **IaaS, PaaS, SaaS**

Serverless application



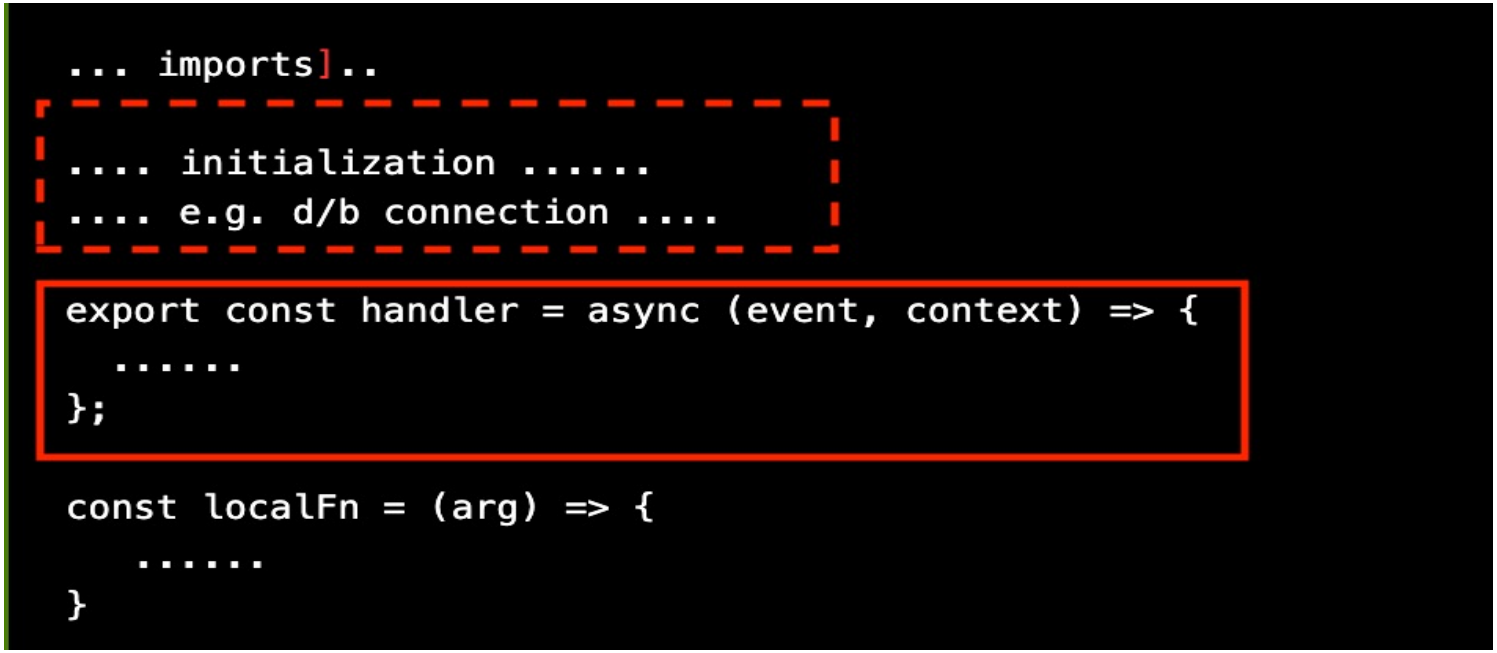
- **Event Source (Trigger): HTTP request; Change in data store, e.g.database, S3; Change in resource state, e.g. EC2.**
- **Lambda function: Python, Node, Java, Go, C#, etc**

AWS Lambda

- **The Lambda service manages:**
  - **Auto scaling (horizontal)**
  - **Load balancing.**
  - **OS maintenance.**
  - **Security isolation.**
  - **Utilization (Memory, CPU, Networking)**

- **Characteristics:**
  - **Function as a unit of scale.**
  - **Function as a unit of deployment.**
  - **Stateless nature.**
  - **Limited by time - short executions.**
  - **Run on-demand.**
  - **Pay per execution and compute time – generous free tier.**
  - **Do not pay for idle time.**

Anatomy of a Lambda function



- **Handler() – function to be executed upon invocation.**
- **Event object – the data sent during lambda function invocation**
- **Context object – access to runtime information.**
- **Initialization code executes before the handler for cold starts. only.**

Lambda Configuration

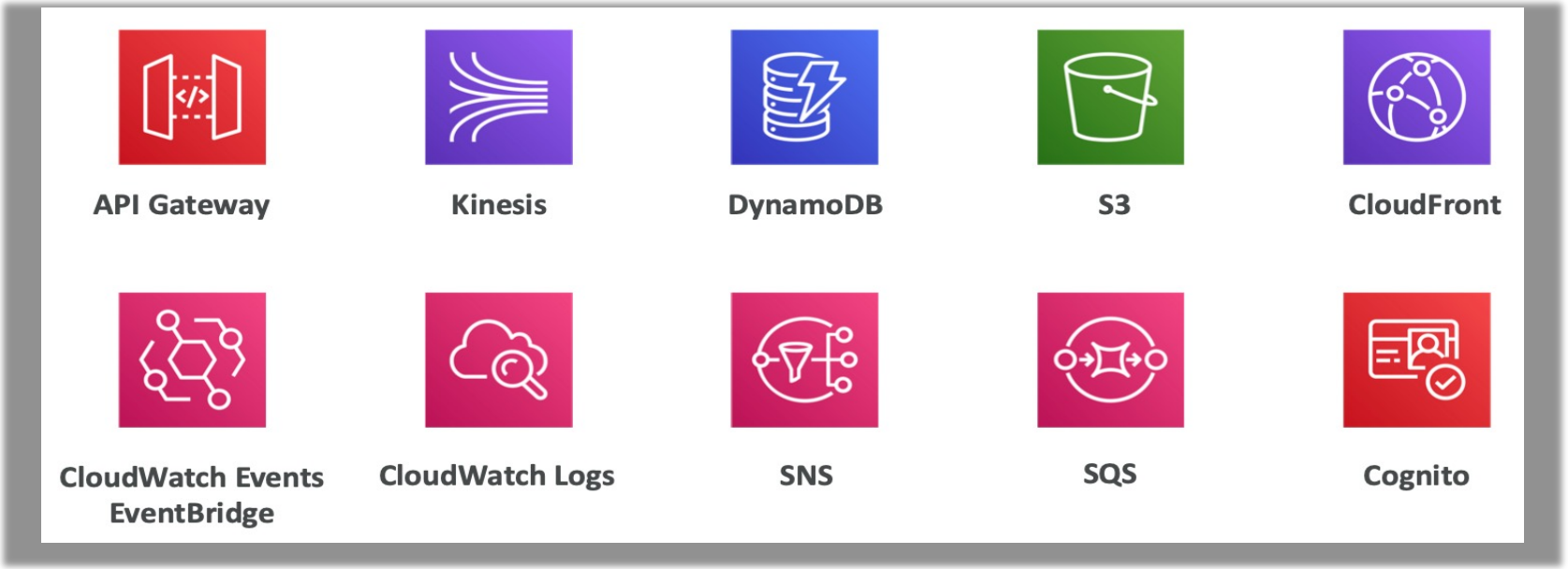
- **Lambda service exposes only a memory control to configure a function's computer power.**
  - **The % of CPU core and network capacity are computed proportionally.**
- **RAM:**
  - **From 128MB to 3,008MB in 64MB increments**
  - **The more RAM you add, the more vCPU credits you get**
  - **At 1,792 MB, a function has the equivalent of one full vCPU**
  - **After 1,792 MB, you get more than one CPU, and need to use multi-threading in your code to benefit from it.**
- **For CPU-bound processing, increase the RAM allocation.**
- **Timeout: default is 3 seconds, maximum is 900 seconds (15 minutes).**

Demo

- **Objective:**
  1. **Use the CDK to provision a ‘Hello World’ lambda function.**
  2. **invoked it from the AWS CLI.**
  3. **See console.log() statement output in Cloudwatch Logs**

Lambda integration

- **Main ones.**

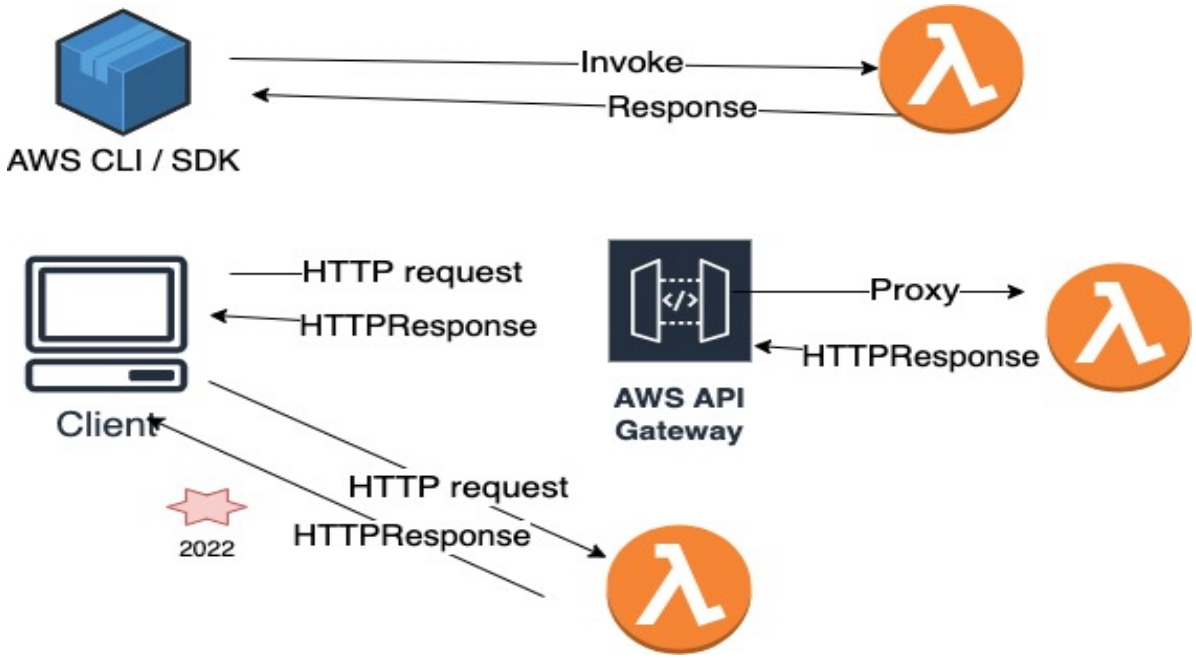


- **Integration models:**

  1. Synchronous. 2. Asynchronous. 3. Poll-based

Lambda – Synchronous Invocations

- **Event Source (Trigger): CLI, SDK, API Gateway, Load Balancers, Function URLs**
  - **Client waits for the results.**
  - **Error handling must happen client-side (retries, exponential backoff, etc...)**

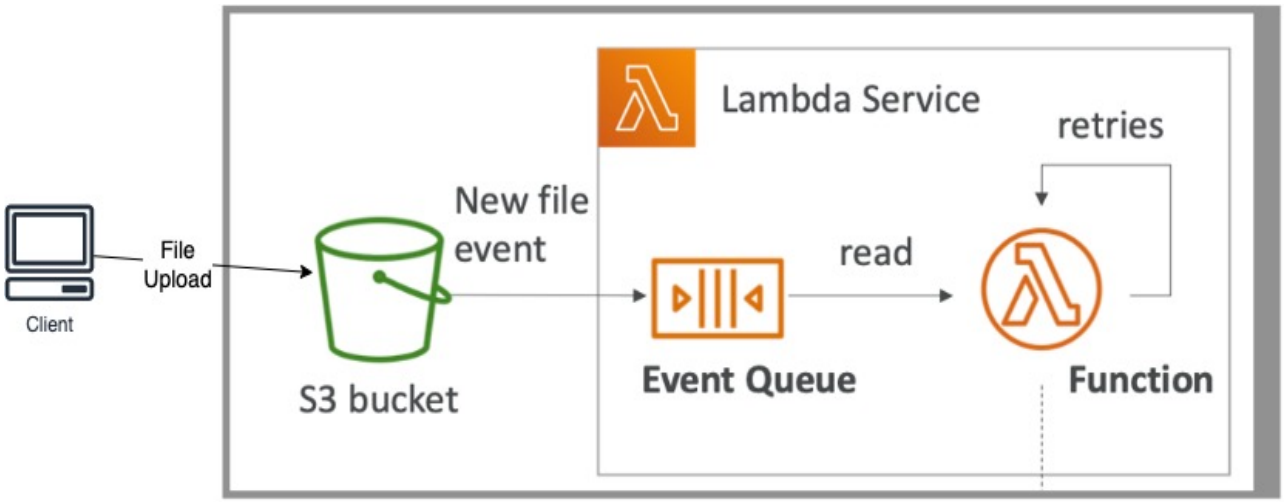


Demo

- **Objective:**
  1. **Use the CDK to provision a lambda function and generate a URL endpoint for invoking it.**
  2. **Test with Postman**
  3. **Make the endpoint private.**

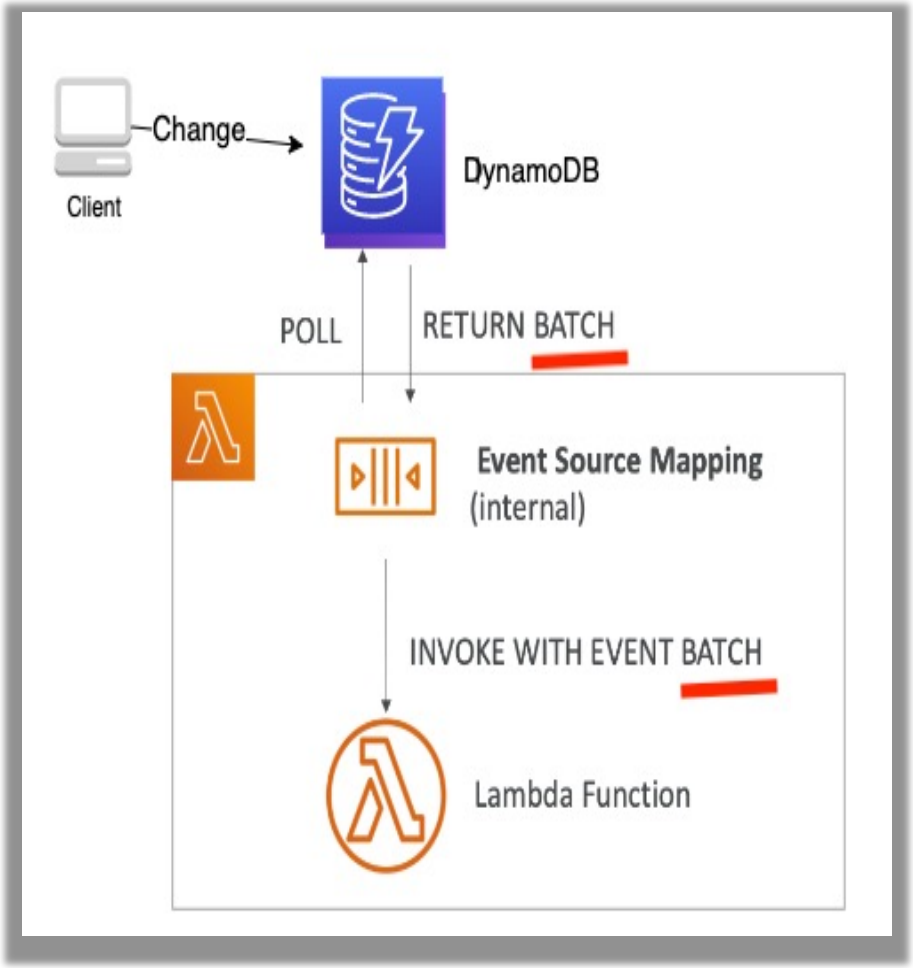
Lambda – Asynchronous Invocations

- **Trigger: S3 event, SNS, Cloudwatch Events.**
- **Lambda service places the events in a queue.**
- **Lambda service attempts to retry on errors – 3 retries, using exponential backoff.**
- **Make sure the processing is idempotent (in case of retries)**
- **Async invocations allow you to speed up the processing if you don't need to wait for a result.**



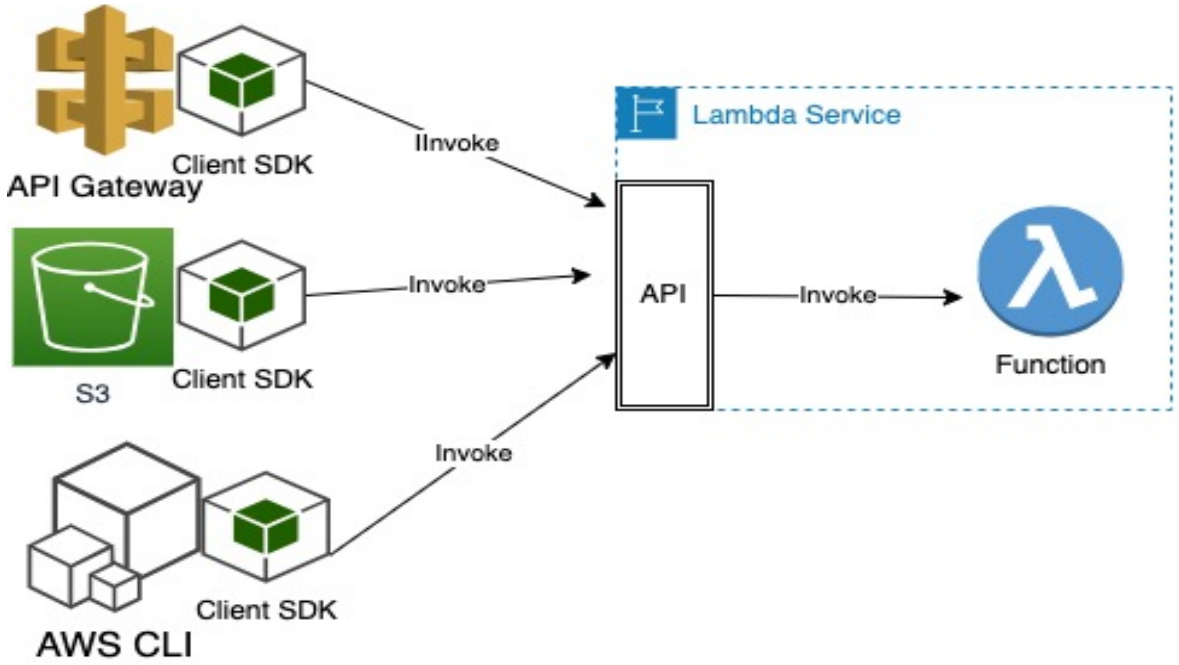
Lambda – Event source mapping. (Poll-based)

- **Sources: DynamoDB streams, SQS, Kinesis streams.**
- **Common denominator: records need to be polled from the source.**
- **Your Lambda function is invoked synchronously.**



Lambda API

- **Lambda Service provides an API**
- **Used by all other services that invoke Lambda functions across all models.**
- **Can pass any event payload structure you want.**



Lambda Execution Role (IAM Role).

- **Grants the Lambda function permissions to access AWS services / resources.**
- **Many Mmanaged policies for Lambda, e.g.**
  - *AWSLambdaBasicExecutionRole* **– Upload logs to CloudWatch.**
  - *AWSLambdaDynamoDBExecutionRole* **– Read from DynamoDB Streams.**
  - *AWSLambdaSQSQueueExecutionRole* **– Read from SQS**
  - *AWSLambdaVPCAccessExecutionRole* **– Deploy Lambda function in VPC.**

- **Best practice: create one Lambda Execution Role per function.**

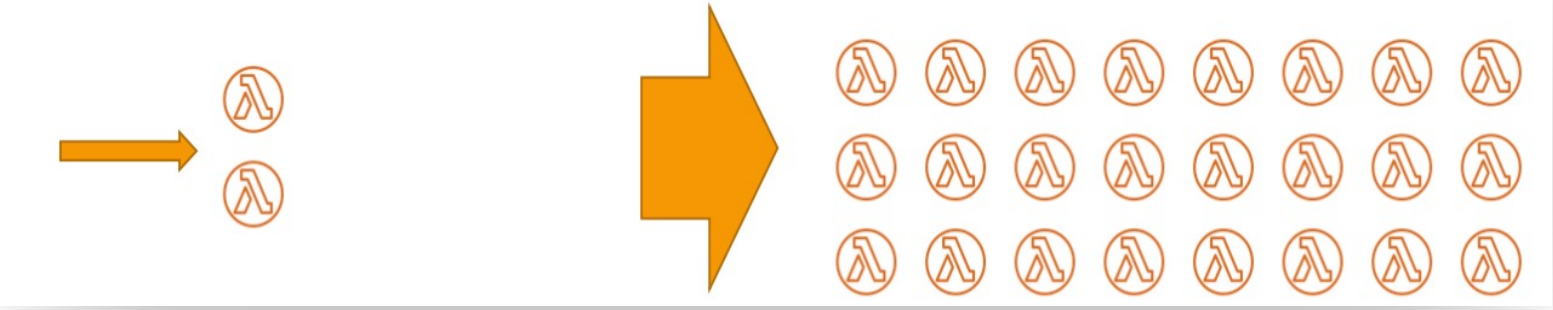
Lambda Resource based Policies.

- **Use resource-based policies to give other AWS services (and accounts) permission to use your Lambda resources.**

- **An IAM principal (e.g. user, service) can access a Lambda resource:**
  - **if the IAM policy attached to the principal authorizes it.**
  - **OR if the resource-based policy authorizes (e.g. service access).**

- **Ex.: An AWS service like Amazon S3 can call your Lambda function if its resource-based policy permits it.**

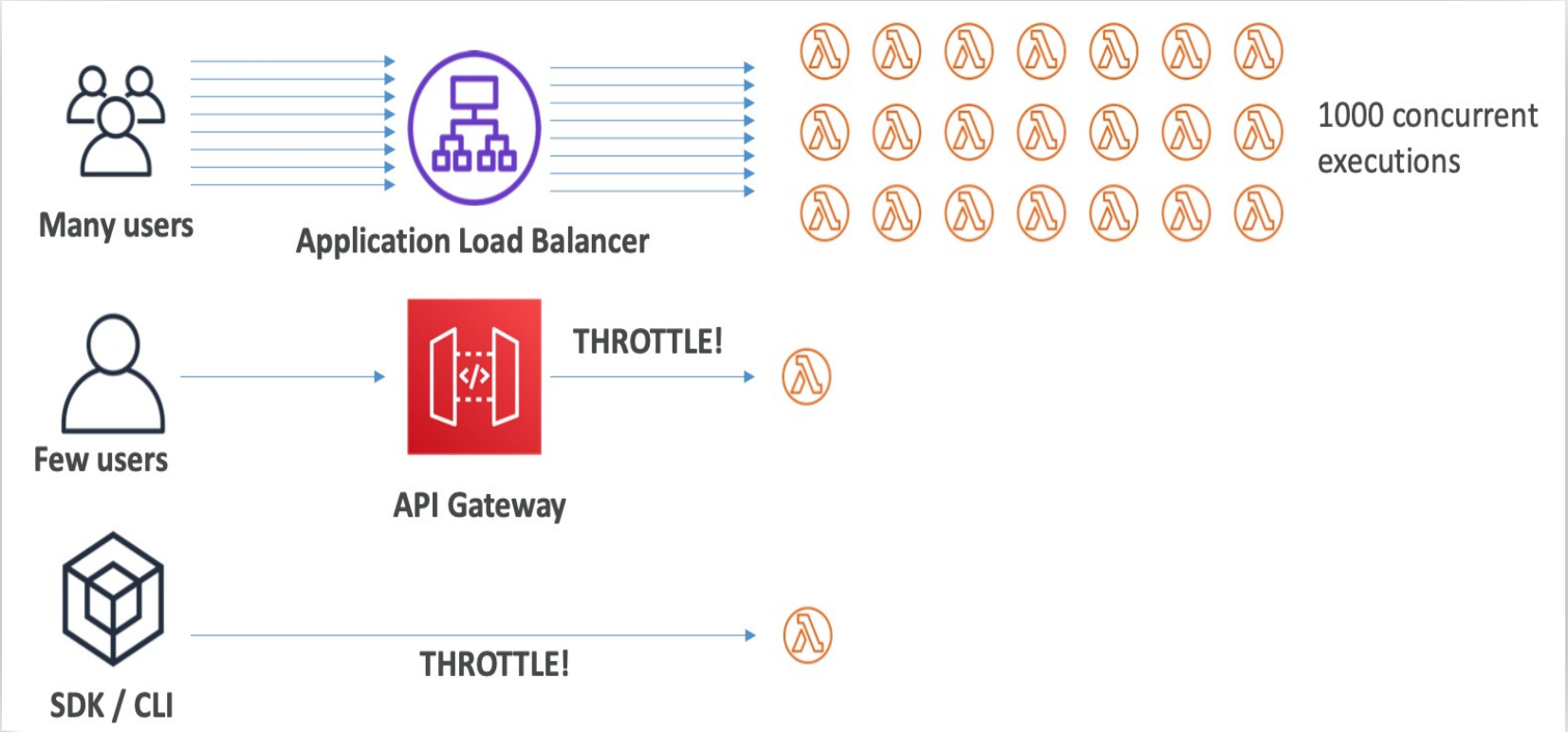
Lambda Concurrency and Throttling



- AWS account concurrency limit: set to 1000 concurrent executions
- Can set a "reserved concurrency" at the function level (= limit).
- Each invocation over the reserved limit will trigger a "Throttle" error.
- Throttle behaviour:
  - If synchronous invocation => return ThrottleError – HTTP status 429
  - If asynchronous invocation => retry automatically and then go to a DLQ (Dead Letter Queue).

Lambda Concurrency Issue

- If you don't reserve (=limit) concurrency, the following can happen:



Cold Start & Provisioned Concurrency.

- **Cold Start:**

  Event → New micro VM instance created → function's initialization code (init) executes → the handler executes.

  - **If the init code is large (LoC, dependencies, SDK), this processing can take time.**
  - **First event processed by a new instances has a higher latency than the rest.**
- **Provisioned Concurrency:**
  - **Micro VMs are pre-allocated before the function is invoked (in advance). Init code executed during pre-allocation.**
  - **So the cold start never happens and all invocations have a lower latency.**

DynamoDB

DynamoDB

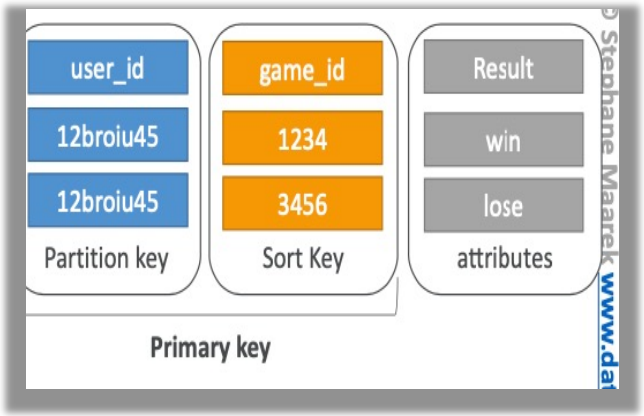
- **NoSQL database - not a relational database.**
- **Fully Managed, Highly available with replication across 3 AZ.**
  - **Serverless**
- **Schema-less (outside the primary key).**
  - **Records in the same table can have different attributes**
- **Scales to massive workloads.**
- **Integrated with IAM for security, authorization and administration.**
- **Enables event driven programming with DynamoDB Streams**

DynamoDB - Basics

- **DynamoDB is made up of tables.**
- **Each table has a primary key (must be decided at creation time)**
- **Each table can have an infinite number of items (= rows)**
- **Each item has attributes (can be added over time – can be null)**
- **Maximum size of a item is 400KB**
- **Data types supported are:**
  - **Scalar Types: String, Number, Binary, Boolean, Null**
  - **Document Types: List, Map**
  - **Set Types: String Set, Number Set, Binary Set**

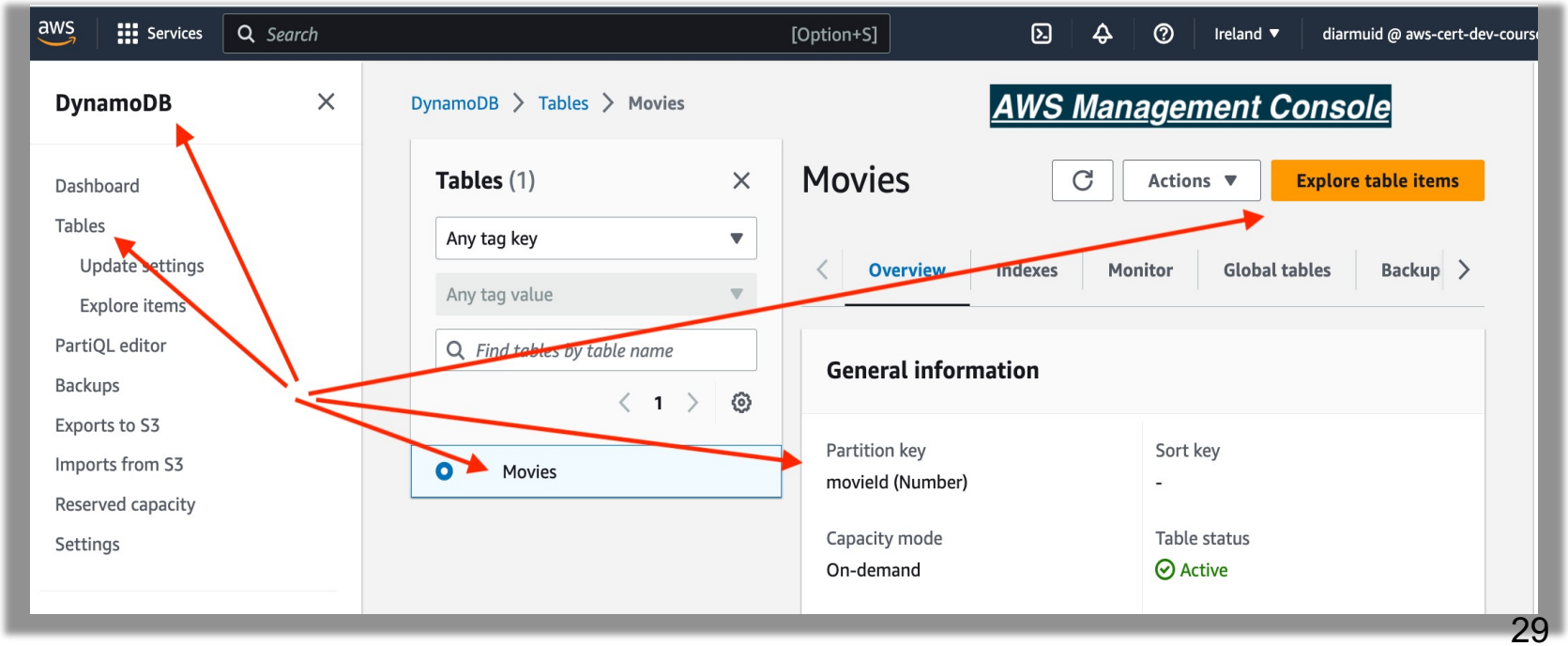
DynamoDB - Primary key

- *Option 1*: Simple - Partition key (aka Hash key) only.
  - Partition key must be unique for each item.
  - Partition key value range must be "diverse" so that the data is distributed evenly.
  - Example: user_id for a users table.
- *Option 2*: Composite - Partition key + Sort Key (aka Range key)
  - The combination must be unique
  - Item collections - Items are:
    - grouped by partition key
    - ordered by the sort key
- Example: users-games table
  - user_id for the partition key
  - game_id for the sort key



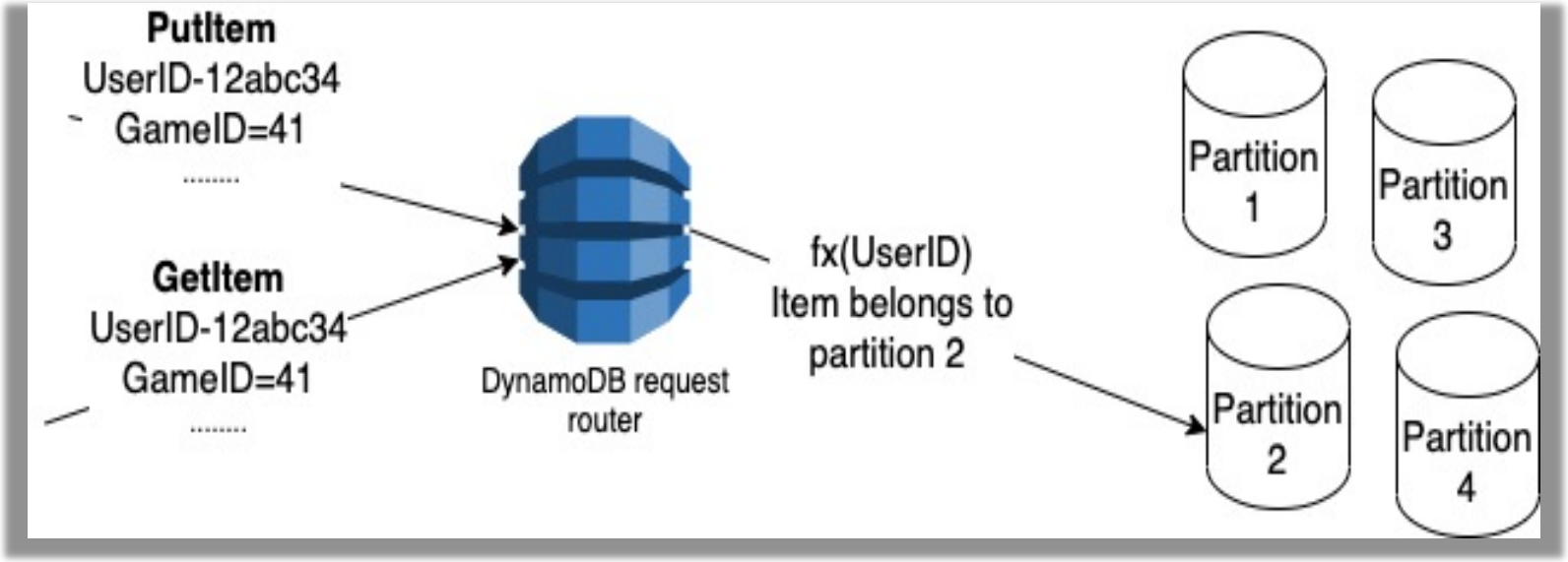
DynamoDB – CDK table definition

```
import { RemovalPolicy } from "aws-cdk-lib";
import { AttributeType, BillingMode, Table } from "aws-cdk-lib/aws-dynamodb";

// CDK code for DynamoDB table

const moviesTable = new Table(this, "MoviesTable", {
  billingMode: BillingMode.PAY_PER_REQUEST,
  partitionKey: { name: "movieId", type: AttributeType.NUMBER },
  removalPolicy: RemovalPolicy.DESTROY,
  tableName: "Movies",
});
```



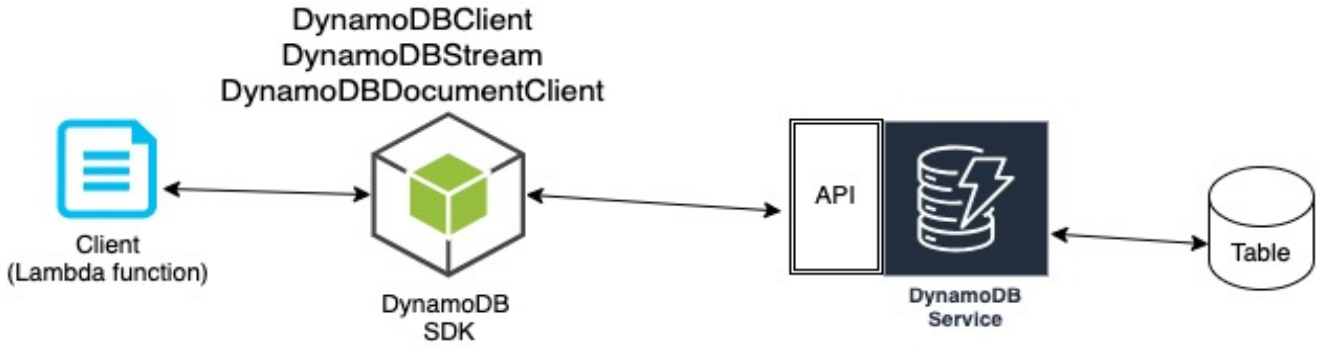
DynamoDB – Horizontal Scaling

- **Data is distributed across a fleet of computers, where a single computer holds a subset of your table's data, called 'partitions' (10GB).**
- **Adv – Infinite scalability; Consistent performance.**



DynamoDB API & SDK

- **DynamoDB SDK provides three classes for clients:** DynamoDBClient, DynamoDBStreams, **and** DynamoDB.DocumentClien**t.**




```
"dependencies": {
  "@aws-sdk/client-dynamodb": "^3.67.0",
  "@aws-sdk/lib-dynamodb": "^3.79.0",
  "@aws-sdk/util-dynamodb": "^3.303.0",
  "aws-cdk-lib": "2.71.0",      You, now • Unc
```

DynamoDB API

- **Three main types of actions:**

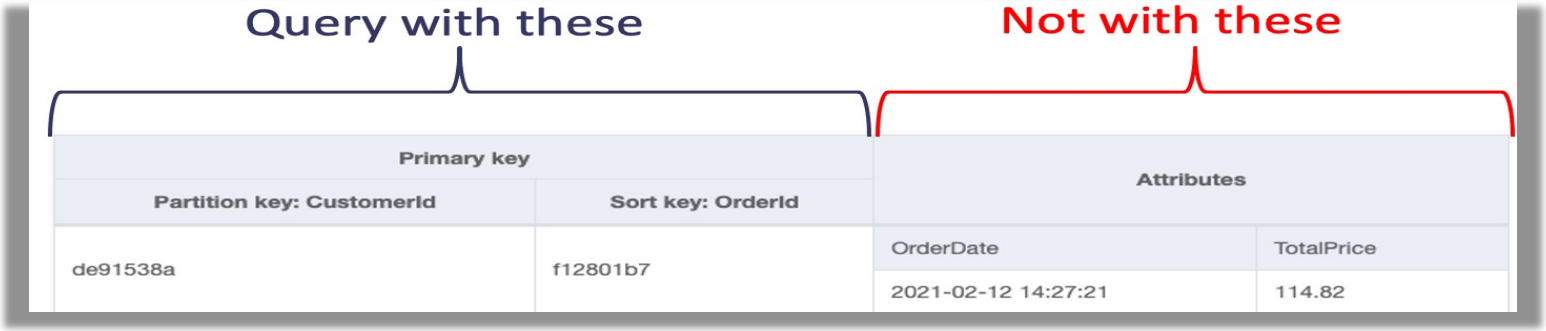
1. **Single-item requests - acts on a single, specific table item and requires the full primary key.**
   - **PutItem, GetItem, UpdateItem, DeleteItem.**
2. **Query - reads a range of items; request must include the partition key.**
   - **Only suitable for tables with composite primary key**
   - **Query result are always from the same partition.**
3. **Scan - reads a range of items but searches across the entire table; an inefficient operation.**

DynamoDB API - DynamoDB.DocumentClient

```
 7  import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
 8  import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";
 9
10  // Native DDB client
11  const ddbClient = new DynamoDBClient({ region: process.env.REGION });
12  // Abstracted DDB client (Document client)
13  const ddbDocClient = DynamoDBDocumentClient.from(
14      ddbClient,
15      ... marshalling/unmarshalling options ....
16    );
17
18  const commandOutput = await ddbDocClient.send(
19      new GetCommand({
20        TableName: process.env.TABLE_NAME,
21        Key: { movieId: 1234 },
22      })
23    );
24
```

- Marshalling **and** Unmarshalling **Data - The document client allows the use of JS types instead of DynamoDB's native AttributeValues. JS objects passed as parameters are marshalled into AttributeValue shapes required by DDB. Responses from DDB are unmarshalled into plain JS objects.**

DynamoDB API – Query action.



- **Query requests return items based on:**
  - **Partition Key value (equals (=) operator only).**
  - **Sort Key value (=, <, <=, >, >=, Between, Begin) – optional.**
  - **Can include a Filter Expression for further filtering (performed on the client side).**
- **Query response:**
  - **Up to 1MB of data,**
  - **Or the number of items specified in Limit parameter.**
- **Pagination on the response is supported.**

DynamoDB API – Query example

- **This table stores the cast for each movie, with one item per movie role.**

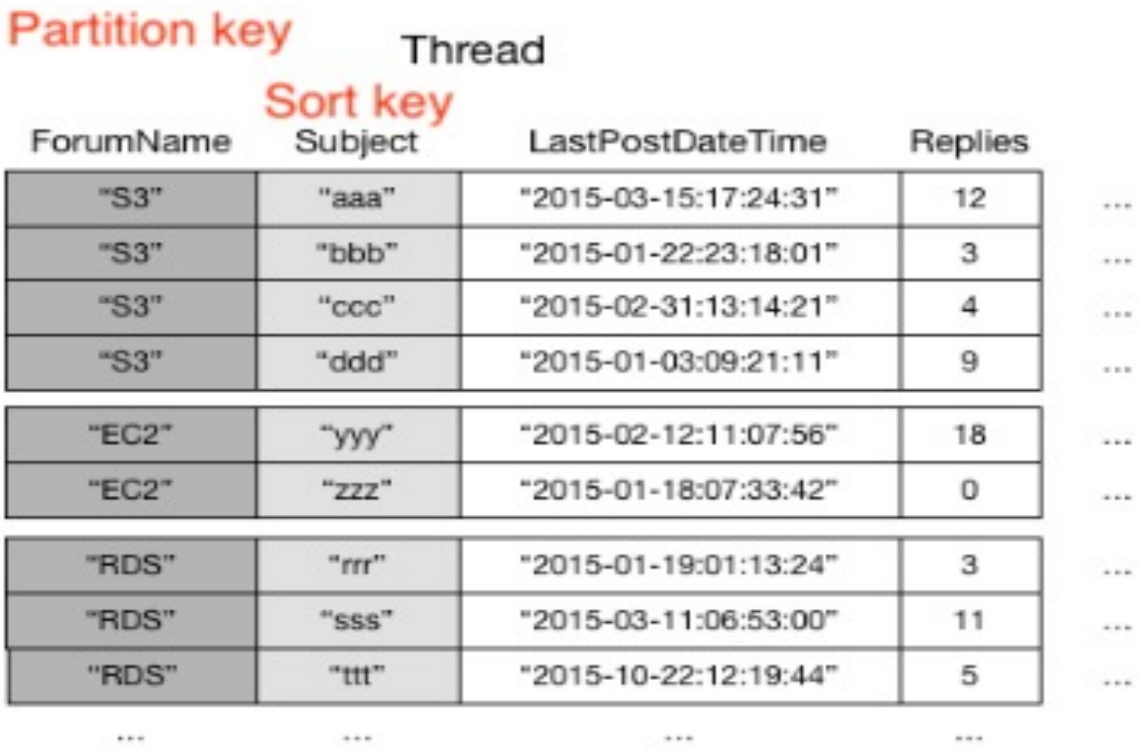
```
24
25  // CDK code for DynamoDB table
26  const movieCastsTable = new Table(this, "MovieCastTable", {
27      billingMode: BillingMode.PAY_PER_REQUEST,
28      partitionKey: { name: "movieId", type: AttributeType.NUMBER },
29      sortKey: { name: "actorName", type: AttributeType.STRING },
30      removalPolicy: RemovalPolicy.DESTROY,
31      tableName: "MovieCast",
32  });
33  //============================
34  // SDK code for DynamoDB query
35  // Find actors whose name begins with Bob on the movie with ID 1234
36  const commandOutput = await ddbDocClient.send(
37      new QueryCommand({
38          TableName: process.env.TABLE_NAME,
39          KeyConditionExpression: "movieId = :m and begins_with(actorName, :a) ",
40          ExpressionAttributeValues: {
41              ":m": 1234,
42              ":a": 'Bob',
43          },
44      })
45  )
```

DynamoDB – LSI (Local secondary index).

- **LSIs are based on alternate sort/range key for a table.**
- **An LSI is stored local to the partition key.**
- **Max of five LSIs per table.**
- **Used with Query actions.**
- **The attribute chosen for LSI sort key must be a scalar String, Number, or Binary.**
- **LSI must be defined at table creation time.**
- **Each LSI entry is a projection of the table item.**
- **A table has:**
  - **a primary / main index,**
  - **(optionally) multiple LSIs, and**
  - **(optionally) multiple GSIs (global secondary indices)**

DynamoDB – Table indices.

- **Example: Table of Discussion threads for a social media app.**
  - **The main / primary index**



DynamoDB –LSI (Local secondary index).

- **Example: Table of Discussion threads for a social media app.**
  - **A Local secondary index**

LastPostIndex

Partition key	Sortkey	
ForumName	*LastPostDateTime*	*Subject*
"S3"	*"2015-01-03:09:21:11"*	*"ddd"*
"S3"	*"2015-01-22:23:18:01"*	*"bbb"*
"S3"	*"2015-02-31:13:14:21"*	*"ccc"*
"S3"	*"2015-03-15:17:24:31"*	*"aaa"*
"EC2"	*"2015-01-18:07:33:42"*	*"zzz"*
"EC2"	*"2015-02-12:11:07:56"*	*"yyy"*
"RDS"	*"2015-01-19:01:13:24"*	*"rrr"*
"RDS"	*"2015-02-22:12:19:44"*	*"ttt"*
"RDS"	*"2015-03-11:06:53:00"*	*"sss"*
...	...	...

To be continued …..