# Serverless Overview

## What’s serverless?

- Serverless is a new paradigm in which the developers don’t have to manage servers anymore.
- They just deploy code.
- They just deploy… functions!
- Initially, Serverless == FaaS (Function as a Service).
- Serverless was pioneered by AWS Lambda but now also includes anything that’s managed:
  - databases
  - messaging
  - storage
  - etc.
- Serverless does not mean there are no servers…
  - it means you just don’t manage / provision / see them.

## Serverless in AWS

- AWS Lambda
- DynamoDB
- AWS Cognito
- AWS API Gateway
- Amazon S3
- AWS SNS & SQS
- AWS Kinesis Data Firehose
- Aurora Serverless
- Step Functions
- Fargate

### Example Architecture Description (from diagram)

1. **Static content** is hosted in an S3 bucket.
2. **Users** log in via AWS Cognito.
3. **REST API** is served via API Gateway.
4. The API Gateway triggers **Lambda** functions.
5. Lambda functions interact with **DynamoDB** for persistence.

## Why AWS Lambda

### Amazon EC2

- Virtual servers in the cloud
- Limited by RAM and CPU
- Continuously running
- Scaling requires manual intervention to add/remove servers

### Amazon Lambda

- Virtual functions – no servers to manage!
- Limited by time – short executions
- Run on-demand
- Scaling is automated!

## Benefits of AWS Lambda

- **Easy Pricing**:
  - Pay per request and compute time
  - Free tier: 1,000,000 requests and 400,000 GB-seconds of compute time
- Integrated with the whole AWS suite of services
- Supports many programming languages
- Easy monitoring via AWS CloudWatch
- Simple to allocate more resources per function (up to 10 GB of RAM)
- Increasing RAM also boosts CPU and network performance

## AWS Lambda Language Support

- Node.js (JavaScript)
- Python
- Java
- C# (.NET Core) / Powershell
- Ruby
- Custom Runtime API (community supported, e.g., Rust or Golang)

### Lambda Container Image

- The container image must implement the Lambda Runtime API
- For running arbitrary Docker images, **ECS / Fargate** is preferred

## AWS Lambda Integrations

### Main Integrations

- **CloudWatch Logs**
- **SNS (Simple Notification Service)**
- **Cognito**
- **SQS (Simple Queue Service)**
- **Amazon S3**
- **Kinesis**
- **API Gateway**
- **DynamoDB**
- **CloudFront**
- **CloudWatch Events**
- **EventBridge**

## Example: Serverless Thumbnail Creation

### Workflow

1. A **new image** is uploaded to an **S3 bucket**.
2. This **triggers an AWS Lambda function**.
3. The Lambda function **creates a thumbnail** of the image.
4. The **new thumbnail** is saved back to **S3**.

5. **Metadata** about the image is stored in **DynamoDB**, such as:
    - Image name
    - Image size
    - Creation date
    - etc.

## Example: Serverless CRON Job

### Workflow

1. **CloudWatch Events** or **EventBridge** is configured to trigger an event.
2. The event triggers an **AWS Lambda function**.
3. The Lambda function **performs a scheduled task**.

### Example Schedule

- Trigger: **Every 1 hour**

## AWS Lambda Pricing: Example

You can find pricing details at: https://aws.amazon.com/lambda/pricing/

### Pay per calls

- First 1,000,000 requests are **free**
- $0.20 per 1 million requests thereafter
    - ($0.0000002 per request)

### Pay per duration (billed in 1 ms increments)

- 400,000 GB-seconds of compute time per month **free**
    - Equals 400,000 seconds for a function with 1 GB RAM
    - Equals 3,200,000 seconds for a function with 128 MB RAM
- After that: **$1.00 for 600,000 GB-seconds**

> *AWS Lambda is generally **very cheap** to run, making it highly popular.*

## AWS Lambda Limits to Know – Per Region

### Execution

- **Memory allocation**: from 128 MB to 10 GB (in 1 MB increments)
- **Maximum execution time**: 900 seconds (15 minutes)
- **Environment variables size**: 4 KB
- **Disk capacity** in the function container ( `/tmp` ): from 512 MB to 10 GB
- **Concurrency executions**: 1000 (default, can be increased)

### Deployment

- **Deployment package size (compressed .zip)**: up to 50 MB
- **Uncompressed size (code + dependencies)**: up to 250 MB
- **Use of `/tmp` directory** allowed to load additional files at startup
- **Environment variables size**: 4 KB (reiterated)

## Lambda Concurrency and Throttling

- **Default concurrency limit**: up to 1000 concurrent executions
- You can set a **reserved concurrency** at the function level (acts as a limit)

### Throttling Behavior

- If the number of invocations exceeds the concurrency limit:
  - **Synchronous invocation** → returns `ThrottleError` (HTTP 429)
  - **Asynchronous invocation** → AWS **automatically retries**, and if it still fails, the event is sent to a **Dead Letter Queue (DLQ)**
- To increase the concurrency limit, you must **open a support ticket**

## Lambda Concurrency Issue

- If you don't **reserve (limit) concurrency**, issues may occur under high load.

### Scenario

- Services like **Application Load Balancer**, **API Gateway**, or **SDK/CLI** can trigger Lambda functions.
- If many users are active at the same time:
  - The number of **concurrent executions** can reach 1000 (default limit).
  - New invocations beyond the limit will be **throttled**.
  - This results in **Throttle errors** even for users with lower usage.

> *Setting a **reserved concurrency** per function helps protect critical functions from being throttled due to traffic spikes elsewhere.*

## Concurrency and Asynchronous Invocations

### Example Scenario

- A new file is uploaded to an **S3 bucket**, triggering multiple **Lambda invocations**.

### Behavior

- If the function **doesn't have enough concurrency** to process all events:
  - Additional requests are **throttled** (HTTP 429).
- For **throttling errors (429)** and **system errors (5xx)**:
  - Lambda **returns the event to the queue**.
  - Lambda **automatically retries** the invocation for **up to 6 hours**.

### Retry Strategy

- Retry interval follows **exponential backoff**:
  - Starts at 1 second after the first attempt
  - Increases up to a maximum interval of 5 minutes

## Cold Starts & Provisioned Concurrency

### Cold Start

- A **cold start** occurs when a new Lambda instance is created:

    - The code is **loaded** and **code outside the handler** is run (initialization phase).
    - If the initialization is large (e.g., many dependencies or SDKs), it can add latency.
- The **first request** served by a new instance experiences **higher latency**.

### Provisioned Concurrency

- Concurrency is **allocated in advance**, before the function is invoked.
- This **eliminates cold starts**, ensuring **low latency** for all invocations.
- You can manage provisioned concurrency with **Application Auto Scaling** (based on schedule or utilization).

### Note

- Cold start times for **Lambda in VPC** were significantly reduced in Oct & Nov 2019.
- AWS Blog on VPC improvements

## Reserved and Provisioned Concurrency

- **Reserved Concurrency**:

    - Guarantees a maximum number of concurrent executions for a specific function.
    - Prevents other functions from using those reserved slots.
    - Can be used to limit or isolate resource usage per function.

- **Provisioned Concurrency**:

    - Pre-warms the Lambda function by initializing instances ahead of time.
    - Ensures zero latency for all requests (no cold starts).
    - Managed via **Application Auto Scaling** based on schedule or utilization.

AWS Docs - Lambda Concurrency

## Lambda SnapStart

- Improves Lambda function **performance up to 10x** at **no extra cost**
- Available for **Java**, **Python**, and **.NET**

### How It Works

- When **SnapStart is enabled**:
    - The function is invoked from a **pre-initialized state**
    - There's **no cold initialization** on each invocation

### On Publishing a New Version

1. Lambda **initializes** your function
2. Takes a **snapshot** of the memory and disk state after initialization
3. The **snapshot is cached** for low-latency access on future invocations

### Invocation Lifecycle Comparison

#### SnapStart Disabled

- `Invoke` → `Init` → `Invoke` → `Shutdown`

#### SnapStart Enabled

- `Invoke` → `Invoke` → `Shutdown`
  *(Function is pre-initialized)*

> *SnapStart significantly reduces latency by skipping the init phase on each call.*

## Customization At The Edge

- Modern applications often run part of their logic **at the edge** to reduce latency.
- **Edge Function**:
    - A code snippet that runs at the edge, attached to **CloudFront distributions**
    - Executes **close to users** for minimal response time

### Options Provided by CloudFront

- **CloudFront Functions**
- **Lambda@Edge**

### Key Characteristics

- **No servers to manage**
- **Globally deployed**
- **Use case**: Customize CDN content on the fly
- **Pricing**: Pay only for what you use
- **Architecture**: Fully serverless

## CloudFront Functions & Lambda@Edge – Use Cases

- **Website Security and Privacy**
- **Dynamic Web Application at the Edge**
- **Search Engine Optimization (SEO)**
- **Intelligent Routing Across Origins and Data Centers**
- **Bot Mitigation at the Edge**
- **Real-time Image Transformation**
- **A/B Testing**
- **User Authentication and Authorization**
- **User Prioritization**
- **User Tracking and Analytics**

## CloudFront Functions

- **Lightweight functions** written in **JavaScript**
- Designed for **high-scale, latency-sensitive** CDN customizations
- Capable of **sub-millisecond startup times** and **millions of requests per second**

### Typical Use Cases

- Modify **Viewer Request**: after CloudFront receives a request from the client
- Modify **Viewer Response**: before CloudFront sends the response back to the client

### Architecture Notes

- Native feature of **CloudFront**
- Code is managed **entirely within CloudFront**, no external service required

### Lifecycle Events Handled

- **Viewer Request**
- **Viewer Response**
- (**Origin Request** and **Origin Response** are handled by Lambda@Edge)

## Lambda@Edge

- **Lambda functions** written in **Node.js** or **Python**
- Automatically **scales** to handle **thousands of requests per second**

### Supported CloudFront Lifecycle Events

- **Viewer Request**: after CloudFront receives the request from the 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

### Deployment Model

- Author your Lambda@Edge function in the **us-east-1** region
- **CloudFront replicates** the function to its edge locations

> *Lambda@Edge enables deeper customization compared to CloudFront Functions, as it can also modify origin-related requests and responses.*

## CloudFront Functions vs. Lambda@Edge

| Feature | 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 & File System Access** | No | Yes |
| **Access to the Request Body** | No | Yes |
| **Pricing** | Free tier available, ~1/6th cost of @Edge | No free tier, charged per request & duration |

> *Use **CloudFront Functions** for lightweight, high-speed operations at the edge. Use **Lambda@Edge** for more advanced use cases that need network, file system access, or body inspection.*

## CloudFront Functions vs. Lambda@Edge – Use Cases

### CloudFront Functions

- **Cache key normalization**:
  - Transform request attributes (headers, cookies, query strings, URL) to optimize cache key

generation

- **Header manipulation**:
  - Insert, modify, or 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

- Suitable for use cases requiring:
  - **Longer execution times** (several milliseconds)
  - **Adjustable CPU or memory allocation**
  - **Third-party libraries** (e.g., AWS SDK for accessing AWS services)
  - **Network access** to external services
  - **File system access** or access to the **body of HTTP requests**

## Lambda by Default

### Default Deployment Behavior

- By default, a **Lambda function is launched outside of your VPC**, in an **AWS-owned VPC**.
- As a result, the function **cannot access private resources in your VPC**, such as:
  - RDS (Relational Database Service)
  - ElastiCache
  - Internal Load Balancers (ELB)
  - Any other service restricted to **private subnets**

### What Works by Default

- **Public internet access** (e.g., public web APIs) works without issues.

> *To access private resources, you need to explicitly connect the Lambda function to your VPC.*

## Lambda in VPC

To allow a Lambda function to access private resources (e.g., RDS) within your VPC:

### Configuration Requirements

- You must specify:
  - **VPC ID**
  - One or more **Subnets**
  - One or more **Security Groups**

### How It Works

- Lambda creates an **Elastic Network Interface (ENI)** in the specified subnets.
- This ENI enables **network access** to other VPC resources.

### Example

- Lambda function can communicate with an **Amazon RDS** instance inside the same VPC, provided that:
  - The **Lambda security group** allows outbound traffic
  - The **RDS security group** allows inbound traffic from the Lambda security group

## Lambda with RDS Proxy

### Problem

- When Lambda functions **directly connect** to an RDS database, they may open **too many connections** under high load, causing scalability issues.

### Solution: RDS Proxy

- **Improves scalability** by **pooling and sharing** database connections across Lambda invocations.
- **Improves availability**:
  - Reduces **failover time by 66%**
  - **Preserves existing connections** during failovers
- **Improves security**:
  - Enforces **IAM authentication**
  - Stores database credentials securely in **AWS Secrets Manager**

### Requirements

- Lambda must be **deployed in a VPC**, because **RDS Proxy is never publicly accessible**.

### Typical Architecture

- Lambda functions → RDS Proxy → RDS Database (in a private subnet within a VPC)

## Invoking Lambda from RDS & Aurora

### Overview

- You can **invoke Lambda functions from within your DB instance**.
- This allows you to **process data events directly from the database**.

### Supported Engines

- **RDS for PostgreSQL**
- **Aurora MySQL**

### Use Case Example

- A user performs an **INSERT** in the database (e.g., registration)
- The database **invokes a Lambda function**
- The Lambda function sends an **email via Amazon SES**

### Requirements

- The **DB instance must allow outbound traffic** to the Lambda function via:
  - Public internet
  - NAT Gateway
  - VPC Endpoints
- The **DB instance must have permissions** to invoke the Lambda:
  - Define a **Lambda Resource-based Policy**
  - Attach the appropriate **IAM Policy** to the DB instance role

## RDS Event Notifications

### What It Does

- Sends **notifications about the DB instance state**, such as:
  - Created
  - Started
  - Stopped
- **Does not provide** any information about the actual **data** in the database

### Subscribable Event Categories

- DB Instance
- DB Snapshot
- DB Parameter Group
- DB Security Group
- RDS Proxy
- Custom Engine Version

### Characteristics

- **Near real-time** delivery (within ~5 minutes)
- Notifications can be sent to:
  - **Amazon SNS**
  - **Amazon EventBridge**
  - These can then **trigger Lambda functions**, push to **SQS**, etc.

## Amazon DynamoDB

- **Fully managed** NoSQL database service
- **Highly available**, with replication across **multiple Availability Zones**
- Not a relational database, but supports **transactions**
- **Massively scalable**:
  - Handles **millions of requests per second**
  - Stores **trillions of rows**, **hundreds of TB**
- **Consistent low latency** performance (single-digit milliseconds)
- **Integrated with IAM** for:
  - Security
  - Authorization
  - Administration
- **Cost-effective** with **auto-scaling** capabilities
- **No maintenance or patching** required
- Supports:
  - **Standard Table Class**
  - **Infrequent Access (IA) Table Class**

## DynamoDB – Basics

- A **DynamoDB database** is composed of **Tables**
- Each table requires a **Primary Key**, defined at creation time
- Tables can store an **unlimited number of items** (rows)
- Each item contains **attributes**, which:
  - Can be **added over time**
  - Can contain **null** values
- **Maximum item size**: 400 KB

### Supported Data Types

- **Scalar Types**:
  - String
  - Number
  - Binary
  - Boolean
  - Null
- **Document Types**:
  - List
  - Map
- **Set Types**:
  - String Set
  - Number Set
  - Binary Set

> *DynamoDB enables **rapid schema evolution**, allowing flexible and dynamic data structures.*

## DynamoDB – Table Example

### Table Structure

| User_ID | Game_ID | Score | Result |
|---|---|---|---|
| 7791a3d6-... | 873e0634-... | 4421 | Win |
| 7791a3d6-... | 873e0634-... | 4521 | Win |
| 7791a3d6-... | 873e0634-... | 1894 | Lose |

### Key Concepts

- **Primary Key** = Partition Key + (optional) Sort Key
- In this example:
  - `User_ID` is the **Partition Key**
  - `Game_ID` is the **Sort Key**
- `Score` and `Result` are **attributes** (non-key fields)

> *This schema design allows multiple games per user, uniquely identified by `Game_ID`.*

## DynamoDB – Read/Write Capacity Modes

Control how you manage a table's **read/write throughput**.

### Provisioned Mode (default)

- You **specify** the number of reads/writes per second
- Requires **capacity planning** ahead of time
- You are charged based on:
  - **Read Capacity Units (RCU)**

- **Write Capacity Units (WCU)**

- Supports **Auto Scaling** for RCU and WCU

### On-Demand Mode

- **Automatically scales** read/write capacity with workload
- **No need for planning** capacity
- **Pay-per-request model**, typically **more expensive**
- Ideal for:
  - **Unpredictable workloads**
  - **Sudden spikes in traffic**

> *Choose the mode based on workload characteristics: predictable vs. unpredictable.*

## DynamoDB Accelerator (DAX)

- **Fully-managed**, **highly available**, in-memory cache for DynamoDB
- Designed to **reduce read congestion** and improve performance
- Provides **microsecond latency** for cached data

### Key Features

- **Seamless integration** with existing DynamoDB APIs
  - No need to modify application logic
- Default **TTL (Time to Live)** for cache entries: **5 minutes**
- Suitable for **read-intensive workloads**

### Architecture Overview

- Application → DAX Cluster → DynamoDB Tables

## DynamoDB Accelerator (DAX) vs. ElastiCache

### DynamoDB Accelerator (DAX)

- Purpose-built **in-memory cache** for **Amazon DynamoDB**
- **Integrated caching** at the **API level**
- Caches:
  - **Individual objects**
  - **Query and scan results**
- Requires **no application logic changes**
- Used to **accelerate read performance** directly from DynamoDB tables

### Amazon ElastiCache

- **General-purpose cache** for any application
- Can be used to store:
  - **Aggregation results**
  - **Pre-computed values**
- Requires **explicit cache management** in your application logic

### Use Case Summary

- **Use DAX** when working directly with DynamoDB and want seamless acceleration
- **Use ElastiCache** when caching data from multiple sources or custom logic

# DynamoDB – Stream Processing

## What It Is

- An **ordered stream** of **item-level modifications** (create, update, delete) in a DynamoDB table

## Use Cases

- React to changes in real-time (e.g., send a welcome email when a new user is created)
- Real-time analytics
- Insert into derivative tables
- Cross-region replication
- Trigger **AWS Lambda** functions on data changes

## Stream Options

### DynamoDB Streams

- **24 hours** data retention
- **Limited number of consumers**
- Can be processed using:
    - **AWS Lambda Triggers**
    - **DynamoDB Stream Kinesis Adapter**

### Kinesis Data Streams (newer alternative)

- **1 year** data retention
- **High number of consumers**
- Integration with:
    - **AWS Lambda**
    - **Kinesis Data Analytics**
    - **Kinesis Data Firehose**
    - **AWS Glue Streaming ETL**

> *Use DynamoDB Streams for basic triggers, and Kinesis for more complex streaming pipelines.*

# DynamoDB Streams – Architecture and Integrations

## Flow Overview

- **DynamoDB Table** emits **Streams** on item-level operations (create/update/delete)
- Streamed data flows into a **Processing Layer** that can integrate with various services

## Integration Options

- **AWS Lambda**:
    - Trigger functions directly from DynamoDB Streams
- **DynamoDB KCL Adapter**:
    - Use with **Kinesis Client Library** for stream processing
- **Kinesis Data Streams**:
    - Enhanced throughput and long-term stream processing
- **Kinesis Data Firehose**:
    - Send data to:
        - **Amazon Redshift** (analytics)

        - **Amazon S3** (archiving)
        - **Amazon OpenSearch** (indexing)
- **Amazon SNS**:
    - Notifications and messaging

### Processing Possibilities

- Data **filtering**
- **Transformation**
- **Fan-out** to multiple destinations

> *DynamoDB Streams enable flexible and powerful integration pipelines for real-time and near-real-time processing.*

## DynamoDB Global Tables

- Enable **low-latency access** to a DynamoDB table from **multiple AWS regions**
- Implements **Active-Active replication** across regions
- Applications can **READ and WRITE** to the table from **any region**
- Ensures **global data availability** and improved **fault tolerance**

### Requirements

- **DynamoDB Streams** must be **enabled** to use Global Tables

### Example

- A table in `us-east-1` replicates to a table in `ap-southeast-2`
- **Two-way replication** keeps data consistent across regions

> *Ideal for global applications needing fast, local access to shared data.*

## DynamoDB – Time To Live (TTL)

- **Automatically deletes** items from a table after a specified **expiration timestamp**
- Helps reduce storage costs and manage data lifecycle

### Use Cases

- Retain only **current or relevant items**
- **Web session management**
- **Regulatory compliance** (e.g., data retention limits)

### Example Table (SessionData)

| User_ID | Session_ID | ExpTime (TTL) |
|---|---|---|
| 7791a3d6-... | 74686572652 | 1631188571 |
| 873e0634-... | 6e6f7468696 | 1631274971 |
| a80f73a1-... | 746f2073656 | 1631102171 |

- The **ExpTime** is a UNIX **epoch timestamp**
- When the current time exceeds `ExpTime`, DynamoDB:
    - **Scans for expired items**

- **Deletes them automatically**

> *TTL is best-effort and items are not removed immediately at expiration.*

## DynamoDB – Backups for Disaster Recovery

### Continuous Backups (Point-in-Time Recovery - PITR)

- **Automatically** backs up data continuously
- Can be enabled for up to **35 days**
- Allows restoring the table to **any point in time** within the retention window
- **Recovery** process creates a **new table**

### On-Demand Backups

- **Full table backups** for **long-term retention**
- Persist until **explicitly deleted**
- **No impact** on table performance or latency
- Can be managed via **AWS Backup**:
  - Supports **cross-region copy**
- Recovery process also creates a **new table**

> *Use PITR for operational recovery, and On-Demand backups for compliance and archiving.*

## DynamoDB – Integration with Amazon S3

### Export to S3 (requires PITR enabled)

- Export data from **any point in time** within the **last 35 days**
- **Does not affect read capacity**
- Use cases:
  - Perform **data analysis** on DynamoDB data
  - **Snapshot retention** for auditing
  - Run **ETL pipelines** before re-importing into DynamoDB
- Export formats:
  - **DynamoDB JSON**
  - **ION**

### Import from S3

- Supported formats:
  - **CSV**
  - **DynamoDB JSON**
  - **ION**
- **No write capacity** is consumed
- A **new table** is created during import
- **Import errors** are logged in **CloudWatch Logs**

### Integration Example

- Export to **S3**, query with **Athena**, or re-import to **DynamoDB**

## Example: Building a Serverless API

### Architecture Overview

- A **Client** makes HTTP requests to an **API Gateway**.
- The API Gateway **proxies** the request to an **AWS Lambda function**.
- The Lambda function performs **CRUD operations** on a **DynamoDB** table.

### Components

- **API Gateway**: Exposes the REST API to clients
- **Lambda**: Contains business logic
- **DynamoDB**: Stores persistent data

> *This is a classic serverless architecture pattern for building scalable, low-maintenance APIs.*

## AWS API Gateway

- Used in combination with **AWS Lambda** for fully serverless APIs
- **No infrastructure** to manage

### Features

- Support for **WebSocket Protocol**
- Manage **API versioning** (e.g., `/v1`, `/v2`)
- Handle **multiple environments**: dev, test, prod
- Integrated **authentication and authorization**
- Ability to create **API keys** and enforce **request throttling**
- Support for **Swagger/OpenAPI** import to define APIs quickly
- **Request and response transformation & validation**
- **SDK and API specification generation**
- **API response caching** to improve performance

> *API Gateway is a powerful tool for exposing and managing REST and WebSocket APIs in a serverless architecture.*

## API Gateway – Integrations High Level

### Lambda Function

- Invoke Lambda function
- Easy way to expose REST APIs backed by AWS Lambda

### HTTP

- Expose HTTP endpoints in the backend
- Examples: internal HTTP API on-premises, Application Load Balancer
- **Why?** Add rate limiting, caching, user authentication, API keys, etc.

### AWS Service

- Expose any AWS API through the API Gateway
- Examples: start an AWS Step Function workflow, post a message to SQS
- **Why?** Add authentication, deploy publicly, rate control

## API Gateway – AWS Service Integration

### Kinesis Data Streams Example

**Flow:**

1. **Client** sends requests via API Gateway.
2. API Gateway forwards records to **Kinesis Data Streams**.
3. Data is then pushed to **Kinesis Data Firehose**.
4. Firehose delivers and stores the data as `.json` files into **Amazon S3**.

## API Gateway - Endpoint Types

### Edge-Optimized (default)

- Designed for global clients
- Requests are routed through CloudFront Edge locations to improve latency
- The API Gateway itself is deployed in a single region

### Regional

- Suitable for clients within the same AWS region
- Can be manually combined with CloudFront for custom caching strategies and distribution control

### Private

- Accessible only from within your VPC via an interface VPC endpoint (ENI)
- Requires a resource policy to define access permissions

## API Gateway – Security

### User Authentication Options

- **IAM Roles**: suitable for internal applications
- **Cognito**: provides identity for external users (e.g., mobile apps)
- **Custom Authorizer**: allows implementation of custom logic for authentication

### Custom Domain Name & HTTPS Security

- Integrates with **AWS Certificate Manager (ACM)**
- **Edge-Optimized endpoint**: certificate must be in `us-east-1`
- **Regional endpoint**: certificate must be in the same region as the API Gateway
- DNS setup required via **CNAME** or **A-alias** record in **Route 53**

## AWS Step Functions

- Build serverless visual workflows to orchestrate your Lambda functions

### Features

- Sequence, parallel execution
- Conditional logic
- Timeouts
- Error handling

### Integrations

- Can interact with:
  - EC2
  - ECS
  - On-premises servers
  - API Gateway
  - SQS queues

### Additional Capabilities

- Supports implementation of human approval steps

### Use Cases

- Order fulfillment
- Data processing
- Web applications
- Any custom workflow

## Amazon Cognito

- Assign identities to users interacting with your web or mobile application

### Cognito User Pools

- Provide sign-in functionality for application users
- Can be integrated with:
    - API Gateway
    - Application Load Balancer

### Cognito Identity Pools (Federated Identity)

- Provide AWS credentials to users for direct access to AWS resources
- Can use Cognito User Pools as an identity provider

### Cognito vs IAM

- Cognito is more appropriate for:
    - Hundreds of users
    - Mobile users
    - Authentication with SAML

## Cognito User Pools (CUP) – User Features

- Create a serverless user database for web and mobile applications

### Key Features

- Simple login using username (or email) and password
- Password reset capability
- Email and phone number verification
- Multi-Factor Authentication (MFA)
- Federated identities support (e.g., Facebook, Google, SAML)

## Cognito User Pools (CUP) - Integrations

- Cognito User Pools integrate with:
    - API Gateway
    - Application Load Balancer (ALB)

### API Gateway Integration

1. REST API receives a request with a Cognito token
2. API Gateway evaluates the token
3. If valid, the request is authenticated and forwarded to the backend

### ALB Integration

1. ALB listener receives request
2. Authentication is handled via Cognito User Pools
3. If authenticated, the request is routed to the appropriate target group and backend

## Cognito Identity Pools (Federated Identities)

- Provide identities to users so they can obtain **temporary AWS credentials**

### Identity Sources

- Cognito User Pools
- Third-party logins (e.g., Facebook, Google, SAML)

### Access Capabilities

- Users can access AWS services:
  - Directly
  - Through API Gateway

### Permissions

- IAM policies applied to credentials are defined within Cognito
- Policies can be customized per `user_id` for fine-grained access control
- Supports default IAM roles for:
  - Authenticated users
  - Guest users

## Cognito Identity Pools – Diagram

### Flow Explanation

1. **Web & Mobile Applications** authenticate users using:

   - Cognito User Pools
   - Social Identity Providers (e.g., Facebook, Google)

2. The application receives a token upon login.

3. The token is exchanged for **temporary AWS credentials** via Cognito Identity Pools.

4. These credentials allow **direct access to AWS services** such as:

   - Private S3 Buckets
   - DynamoDB Tables

5. The credentials are validated and authorized using IAM policies associated with the identity.

This mechanism enables secure and controlled access to AWS resources from client applications.

## Cognito Identity Pools – Row Level Security in DynamoDB

- When using Cognito Identity Pools, you can enforce **row-level security** in DynamoDB by customizing IAM policies.

### Example Use Case

- Each user should only access their own records in a DynamoDB table.

### How It Works

- IAM policies can include conditions that reference the `user_id` or `sub` (subject) from the Cognito identity.
- These conditions restrict access to specific items in the DynamoDB table based on the identity of the authenticated user.

This setup ensures that users can only read or write records that belong to them.