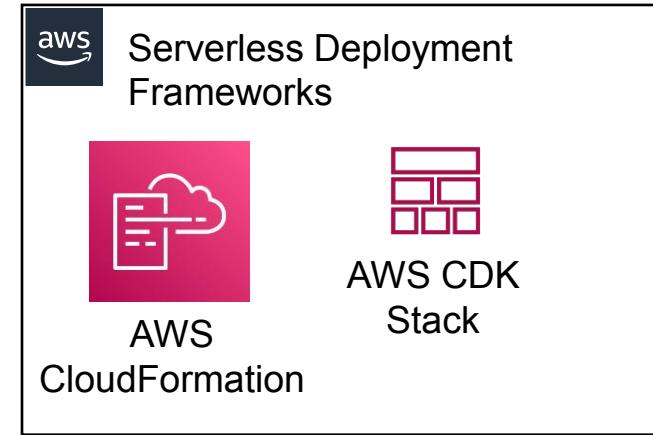
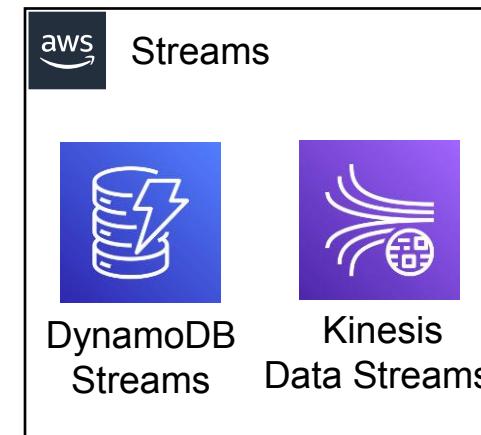
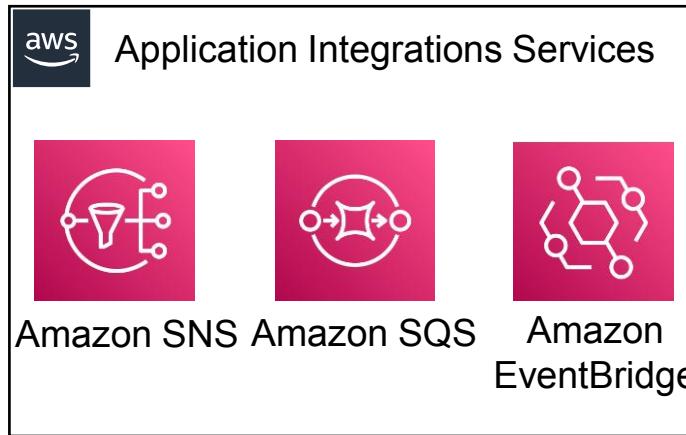
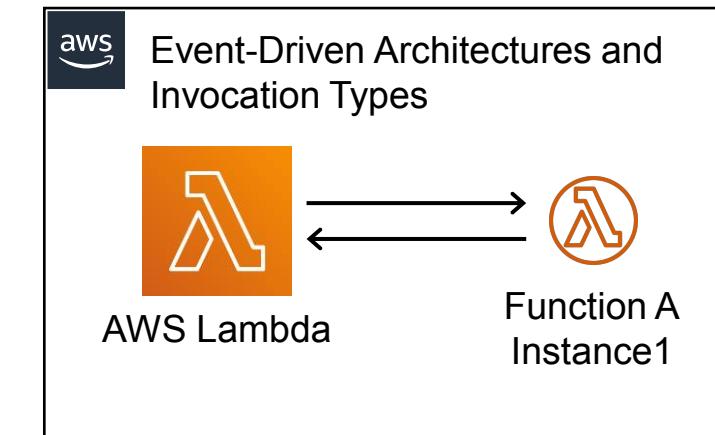
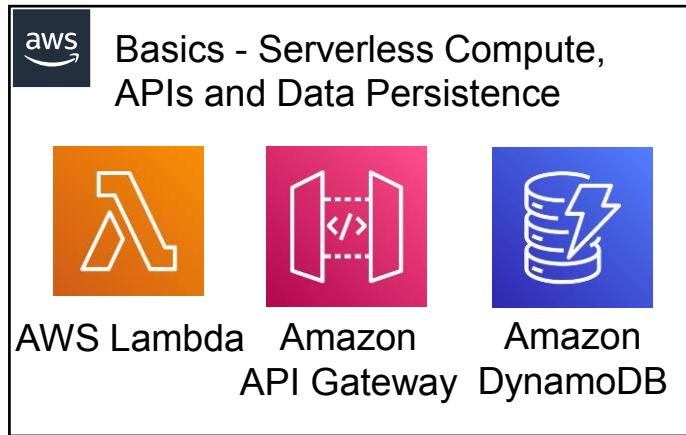


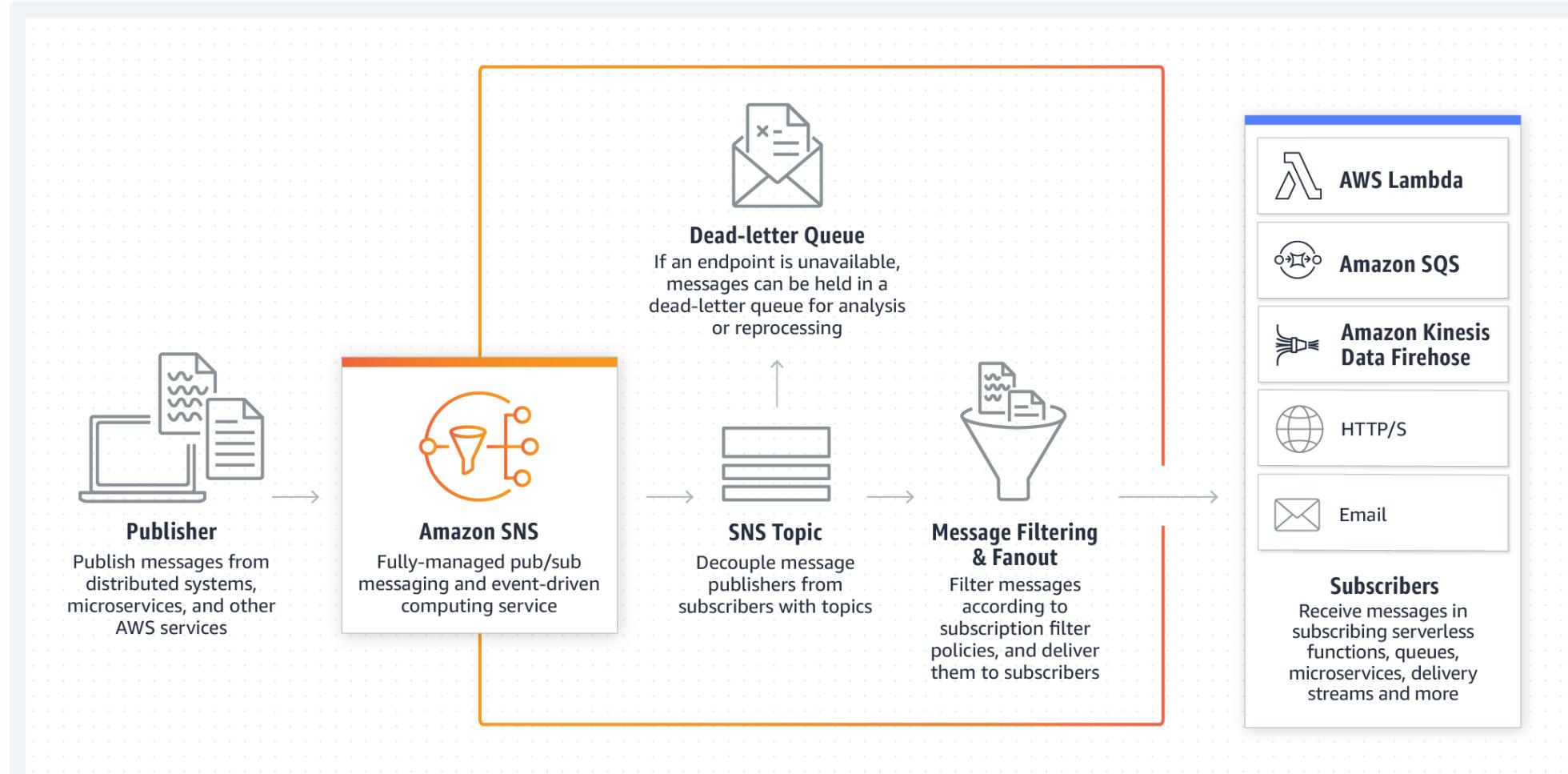
AWS Serverless Services During the Course by Grouping



Amazon SNS: Fully Managed Pub/Sub Messaging

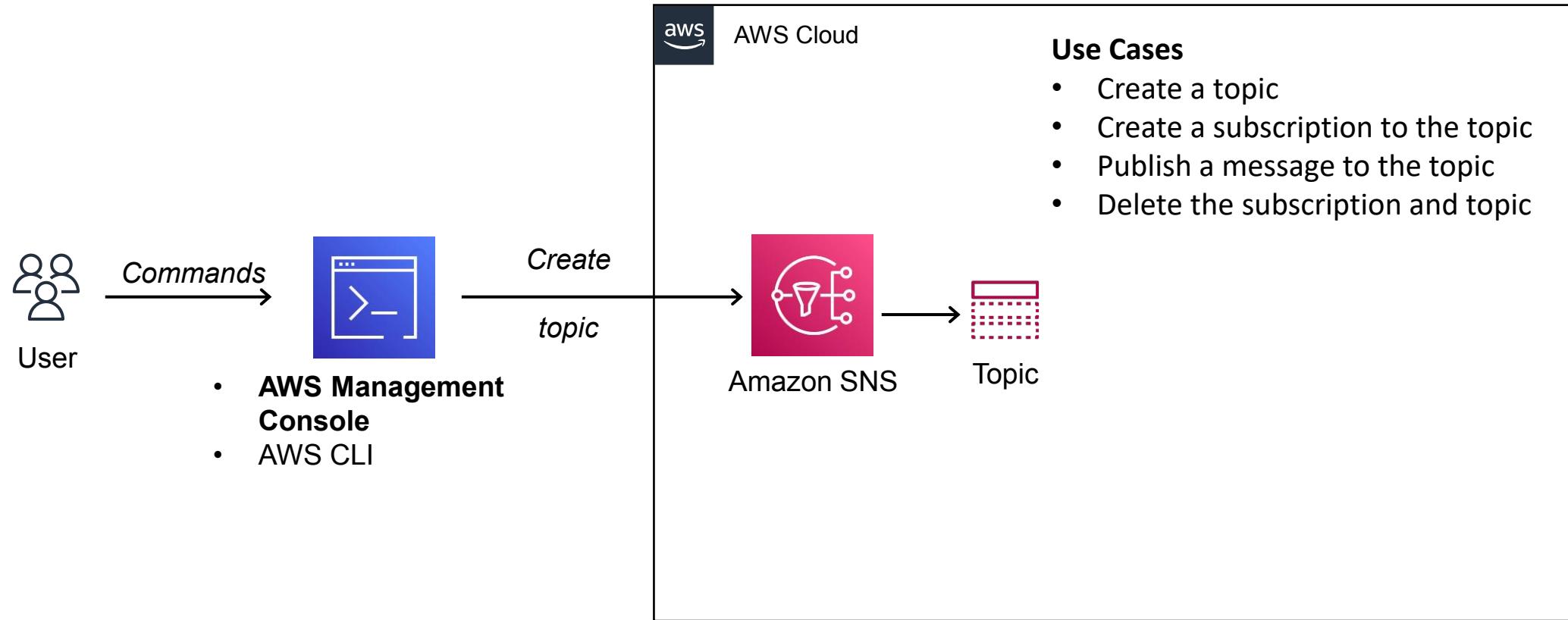


Application
Integration

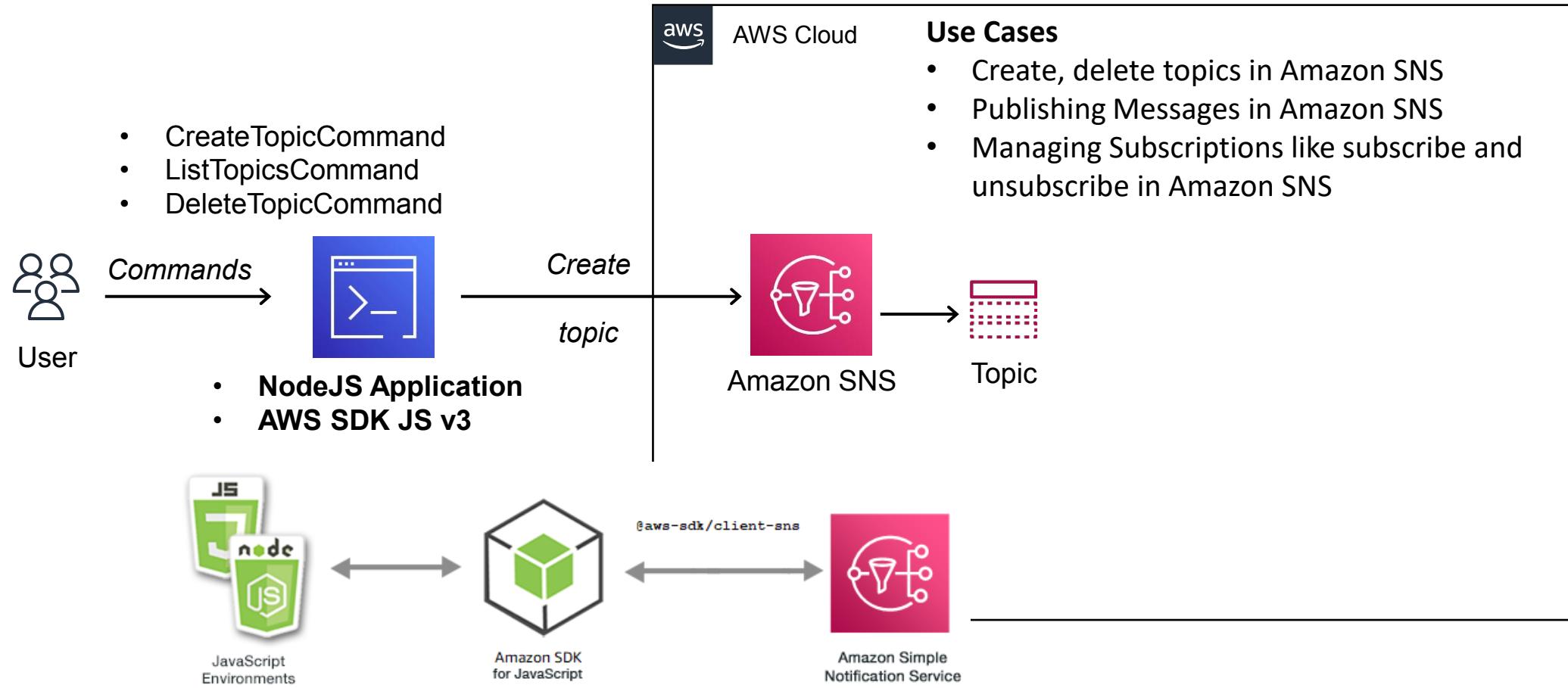


<https://aws.amazon.com/sns/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=deschttps%3A%2F%2Fdocs.aws.amazon.com%2Fsns%2Flatest%2Fdg%2Fwelcome.html>

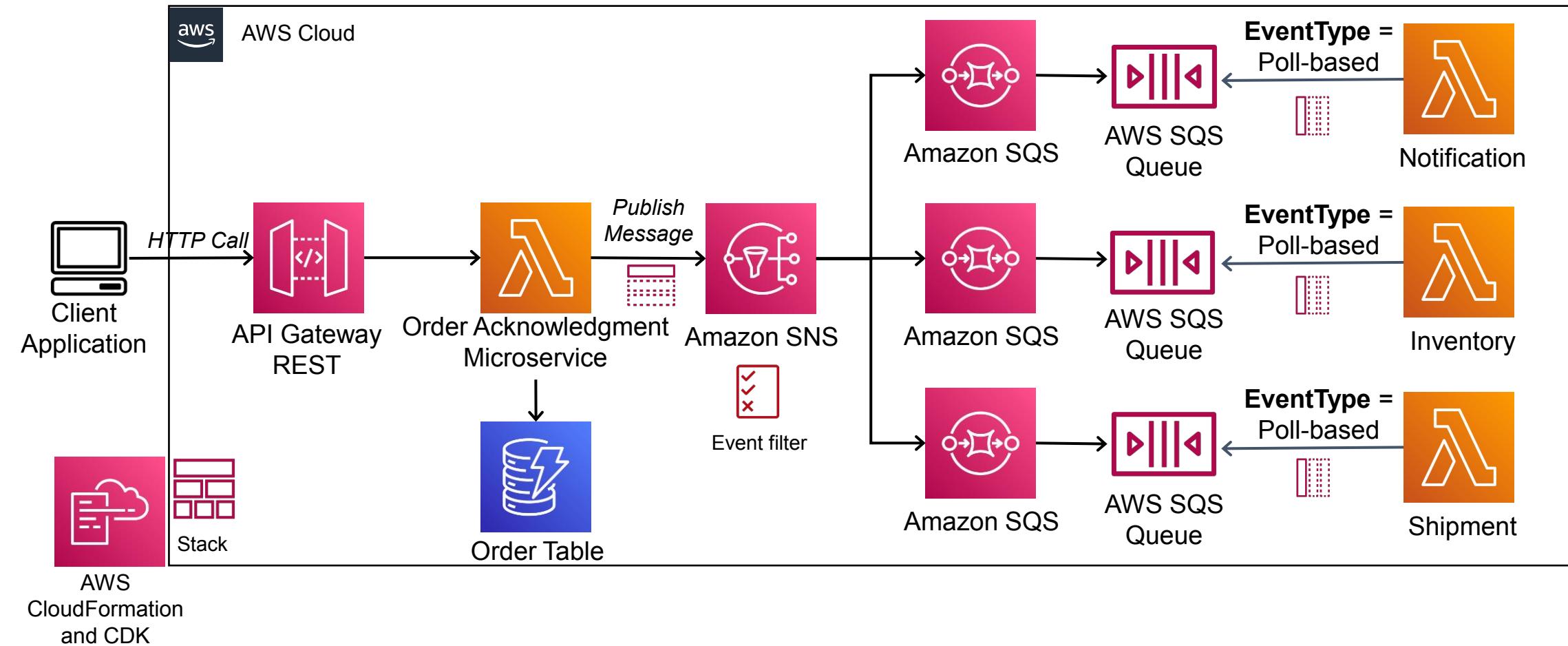
Getting started with Amazon SNS with AWS Management Console



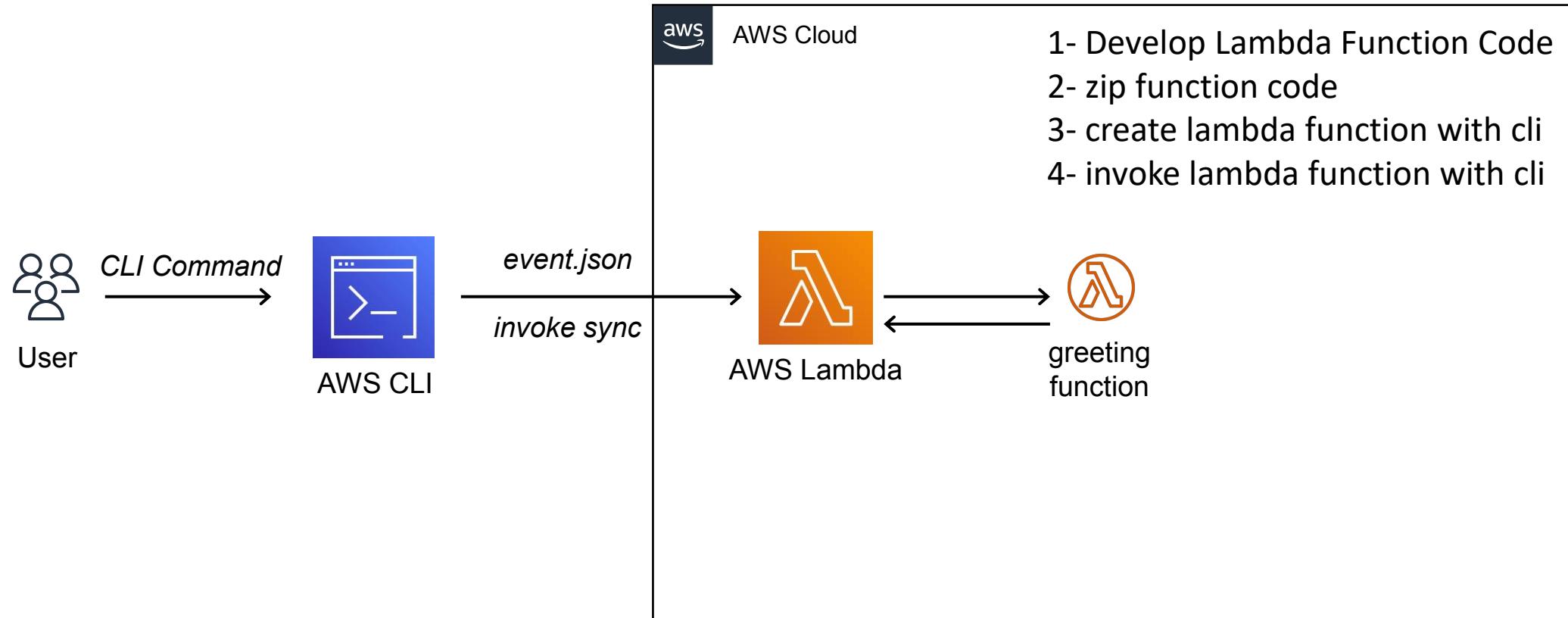
Amazon SNS SDK Examples using AWS SDK Javascript v3



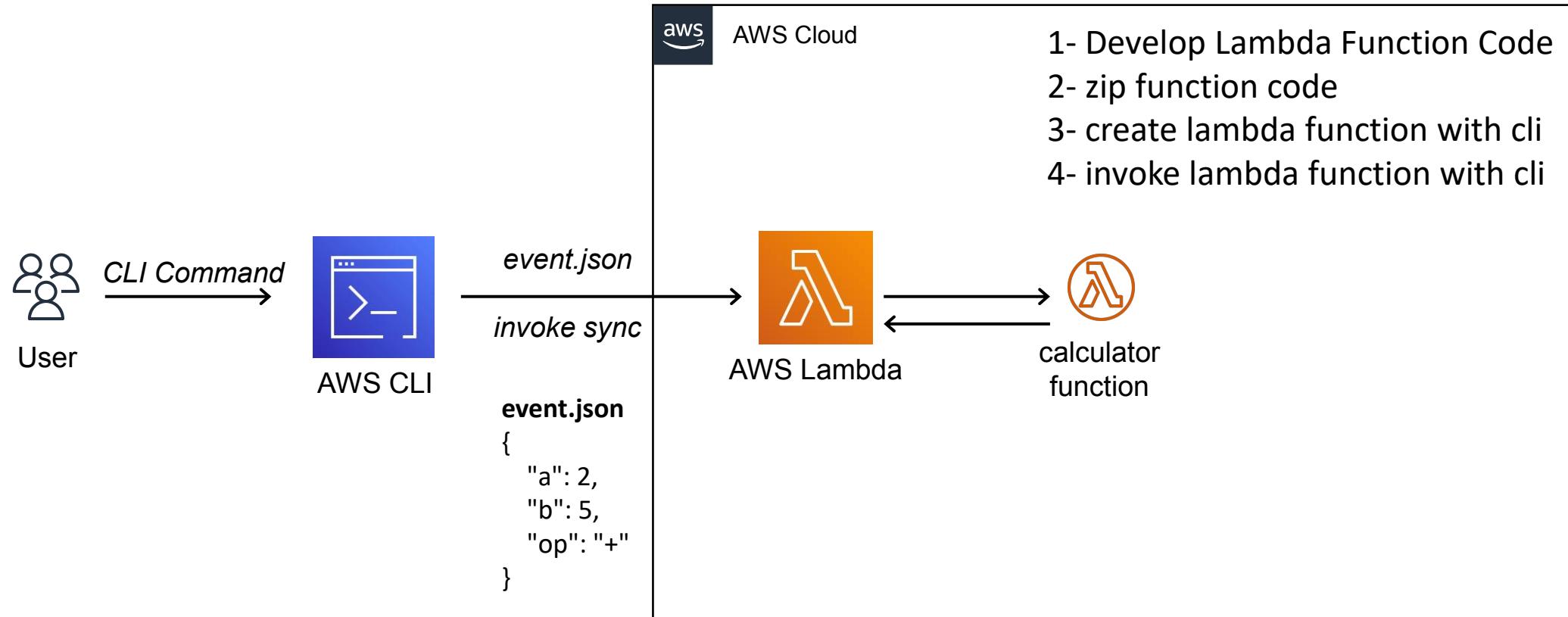
Hands-on Lab: Fan-Out Serverless Architectures Using SNS, SQS and Lambda



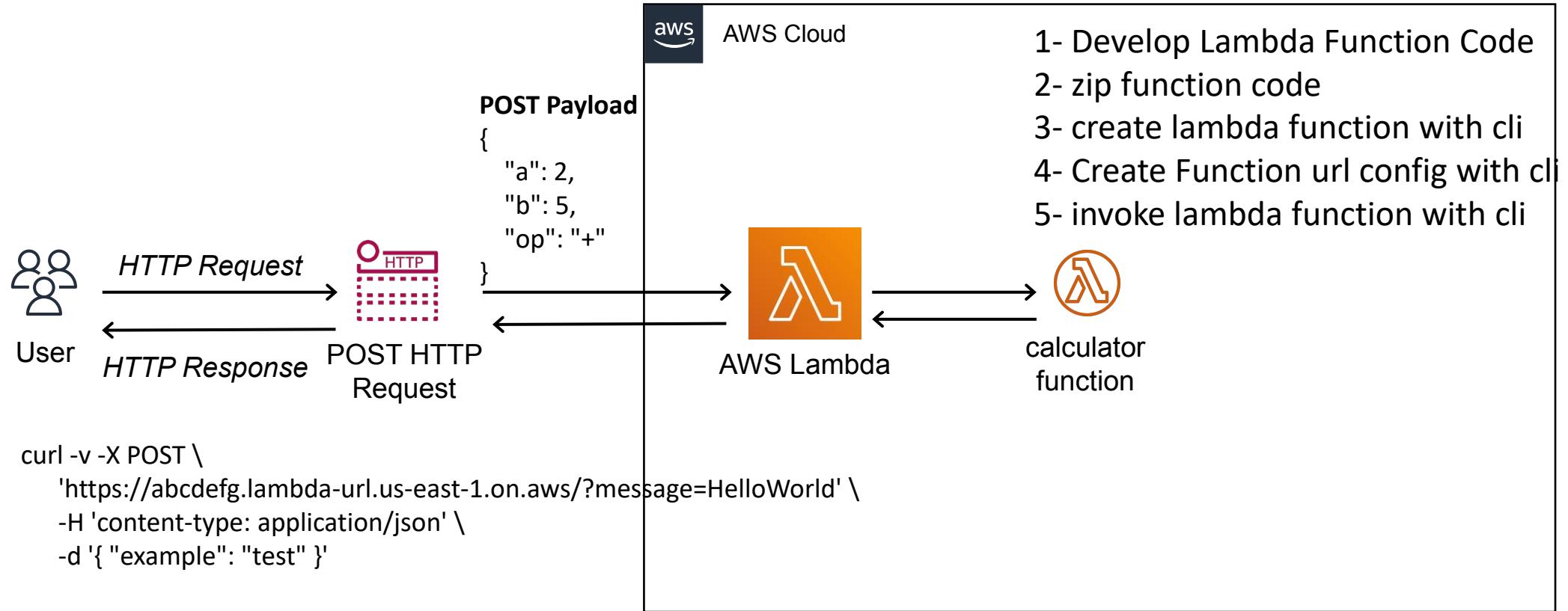
Hands-on Lab: Greeting Project



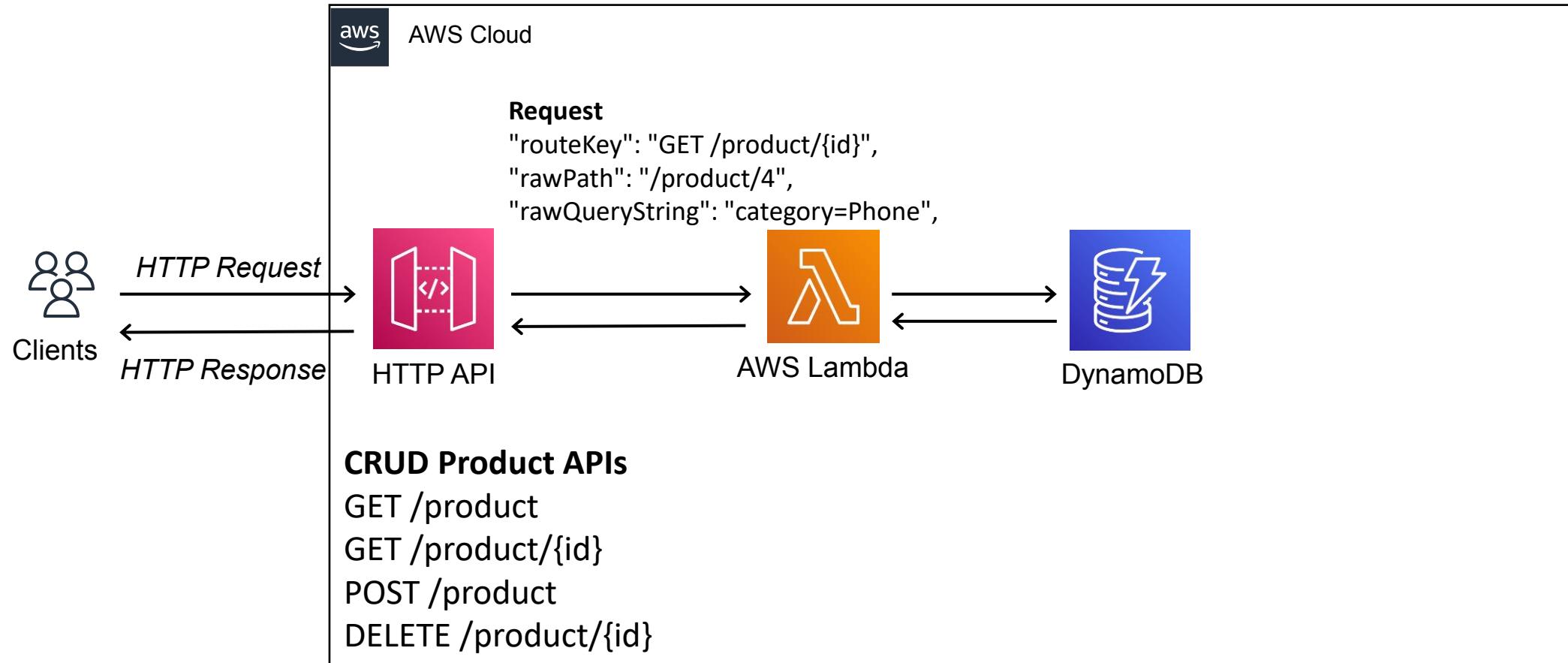
Hands-on Lab: Calculator Project



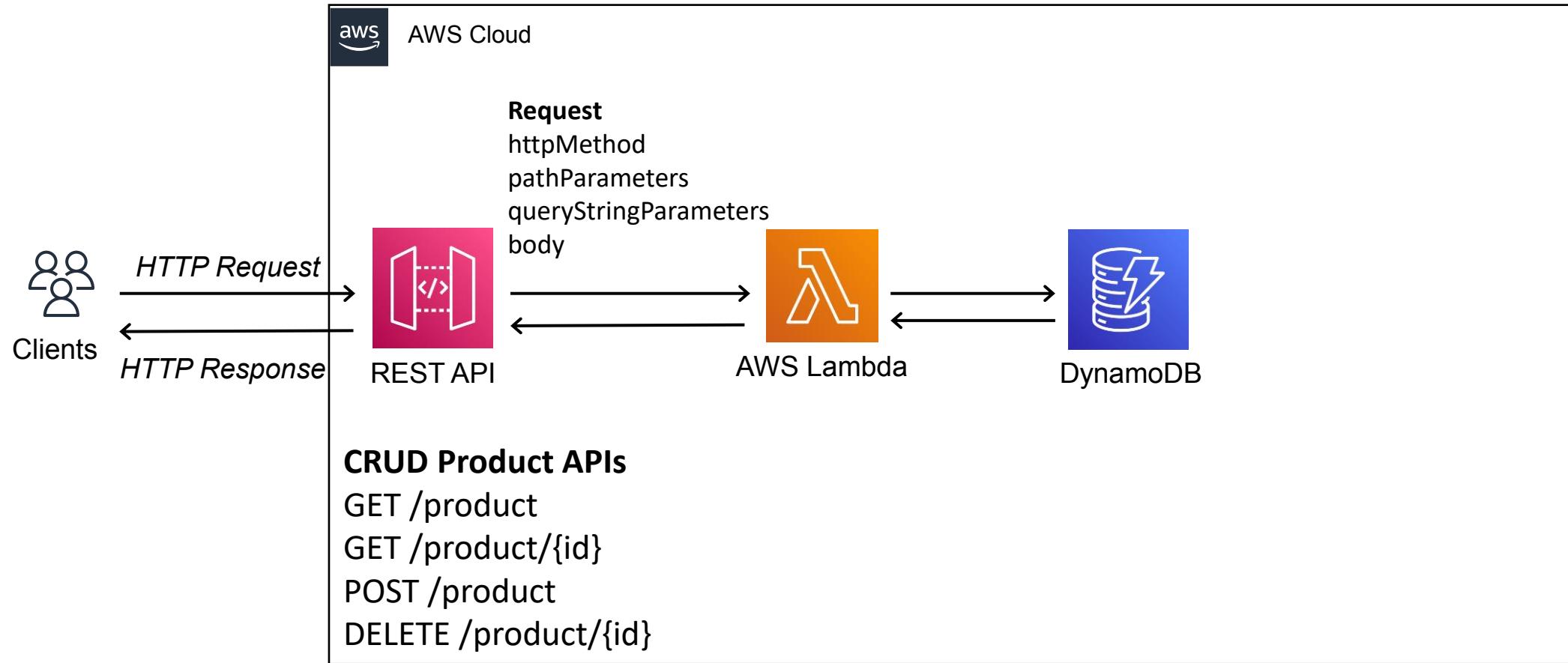
Hands-on Lab: Single Calculator Microservice Expose https methods with Lambda Function Url



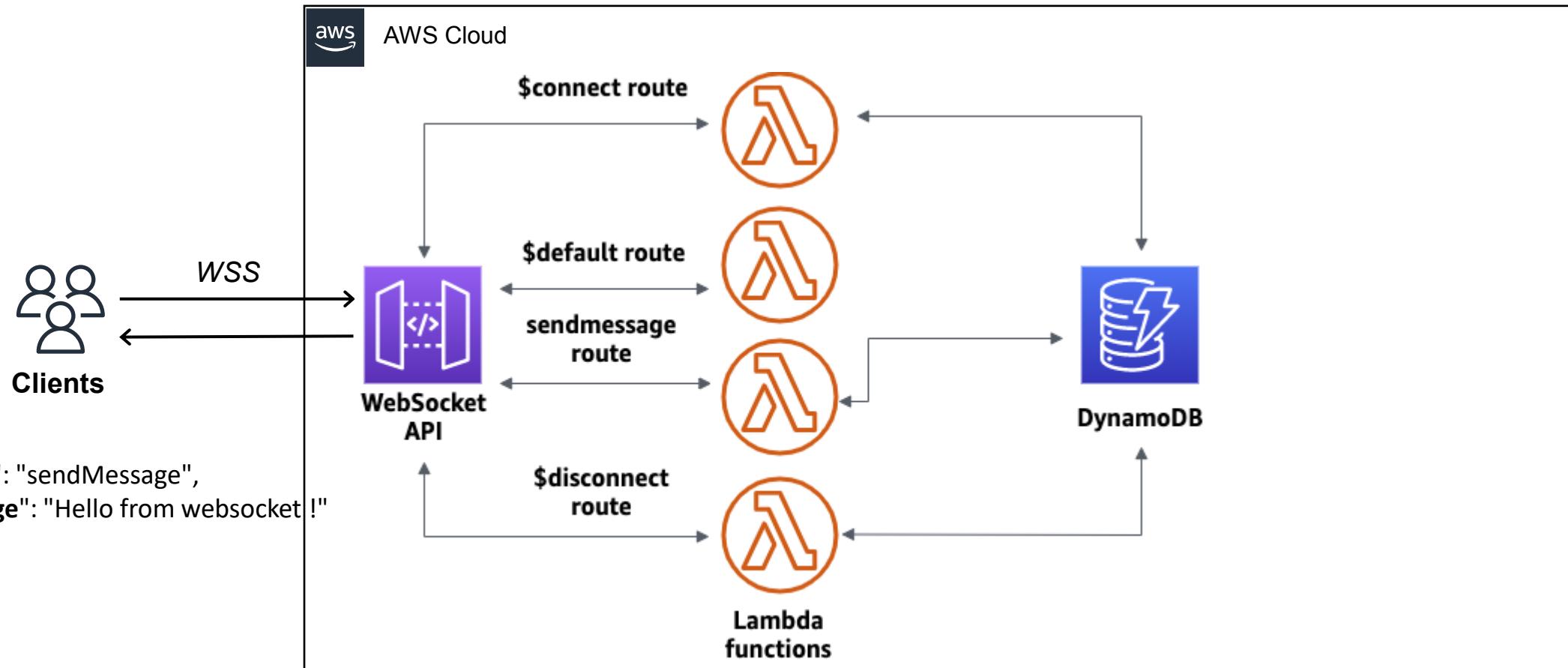
Hands-on Lab: Build CRUD Microservice with HTTP API and Lambda



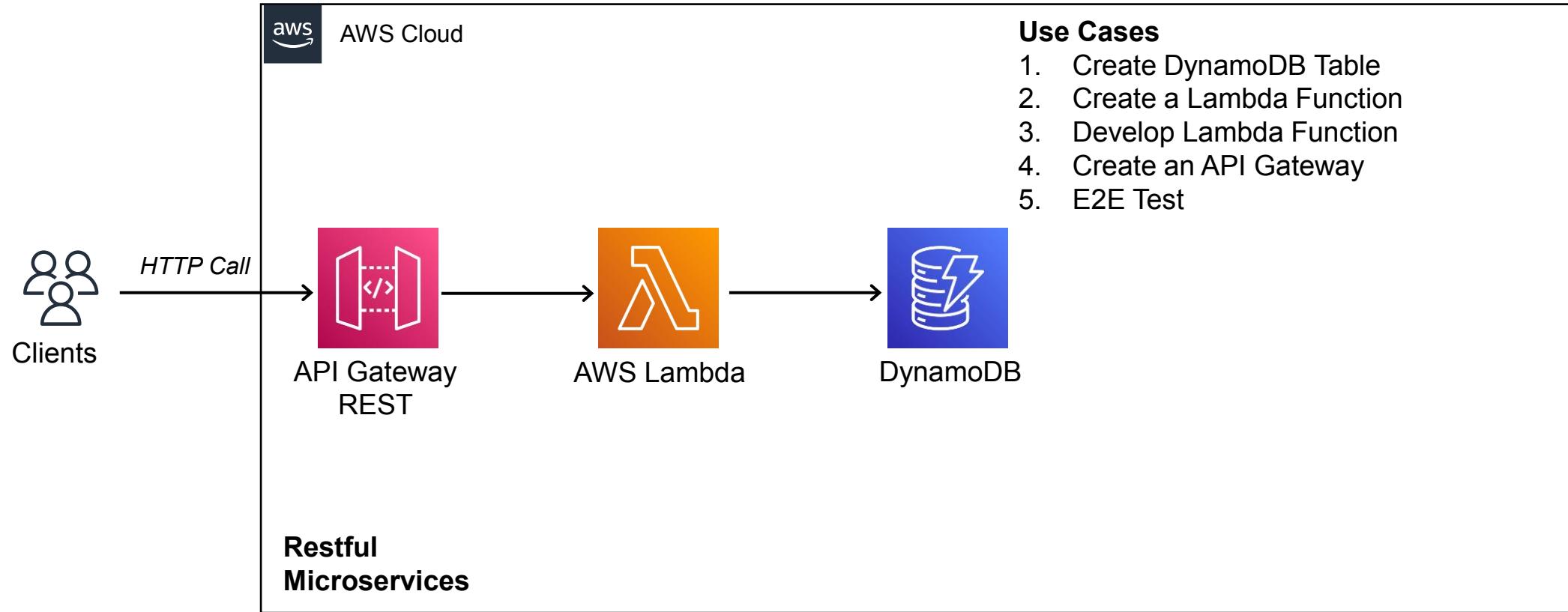
Hands-on Lab: Build CRUD Microservice with REST API and Lambda



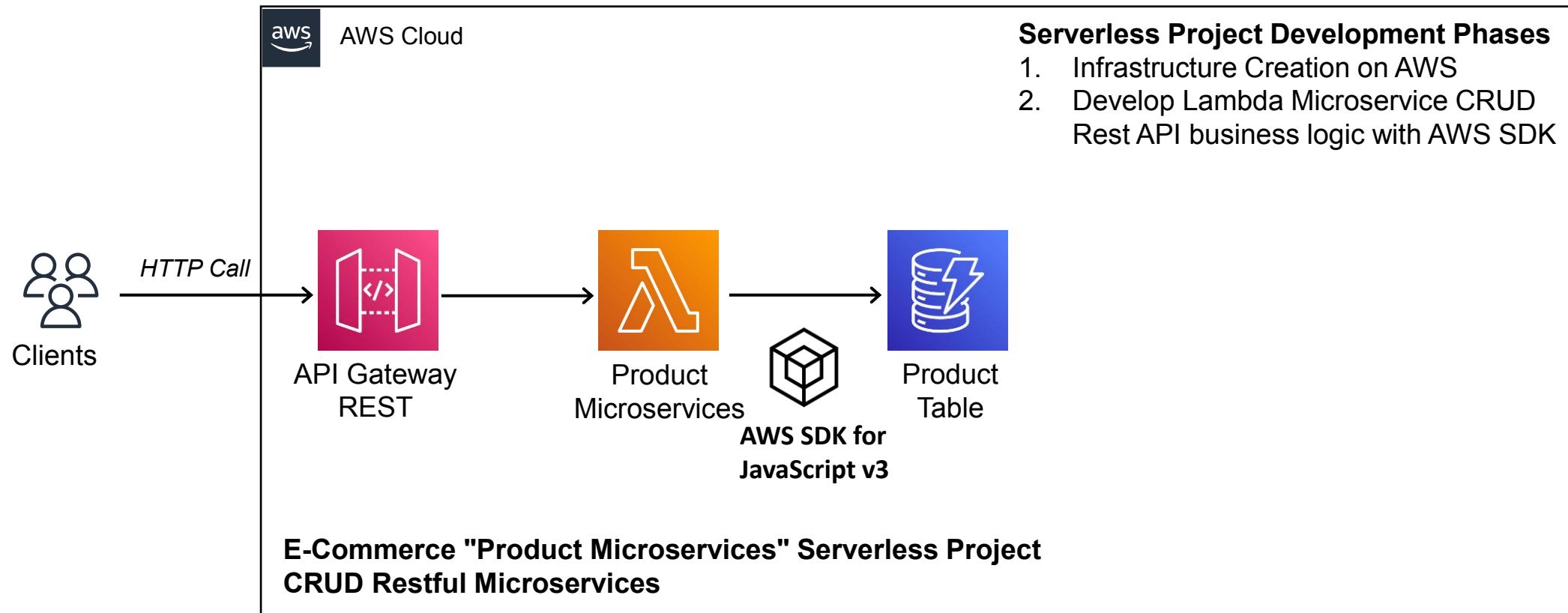
Hands-on Lab: Build Serverless Chat App with a WebSocket API and Lambda



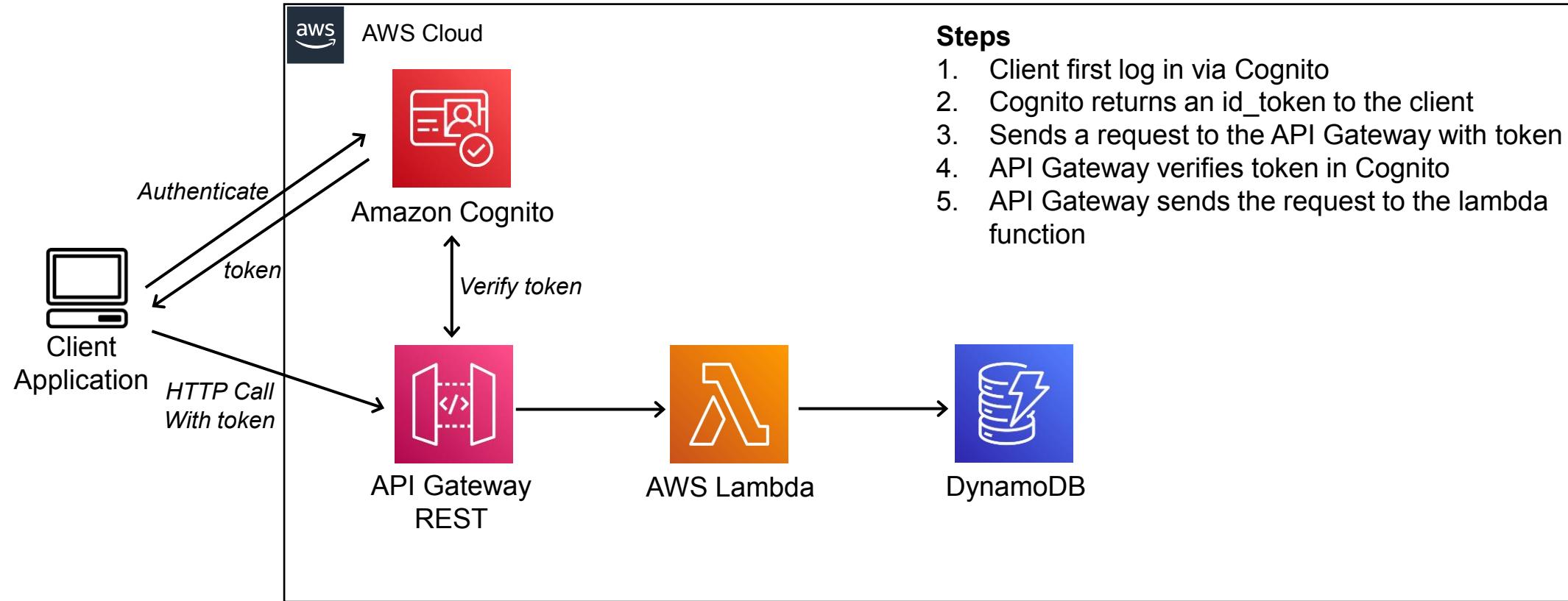
Hands-on Lab: Building RESTful Microservices with AWS Lambda, API Gateway and DynamoDB



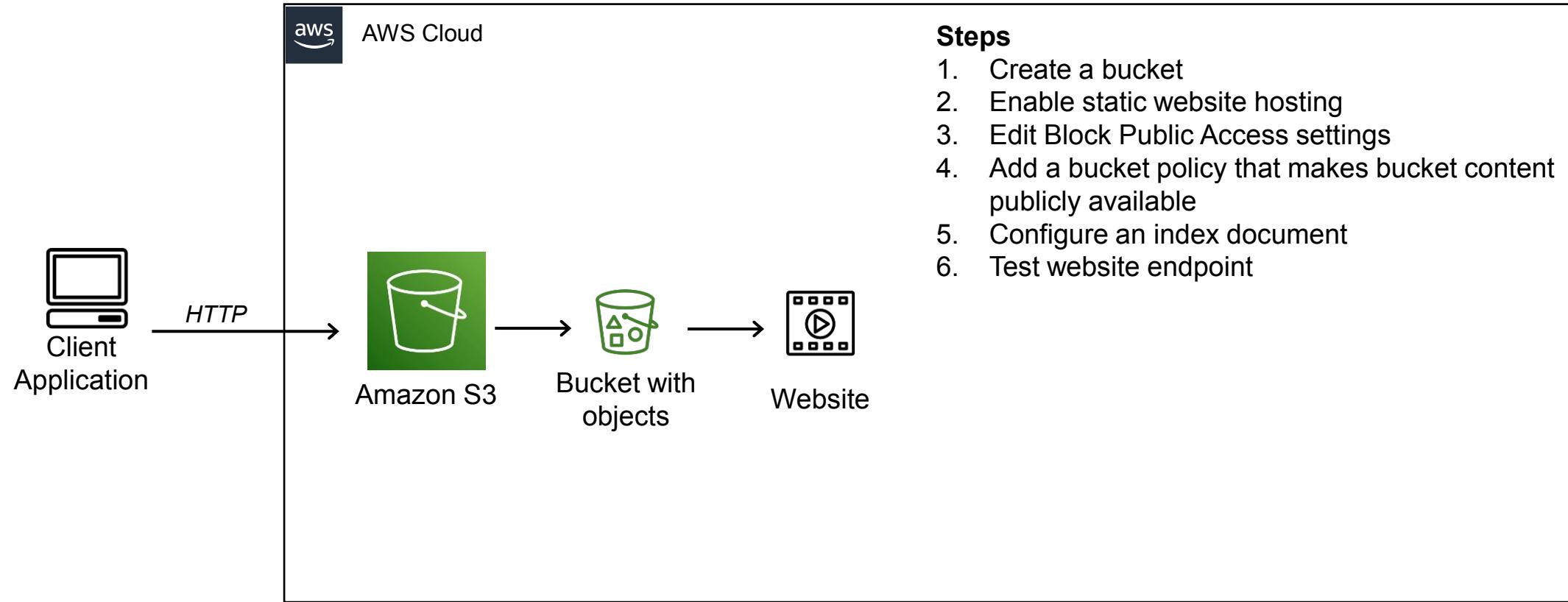
Hands-on Lab: Building RESTful Microservices with AWS Lambda, API Gateway and DynamoDB



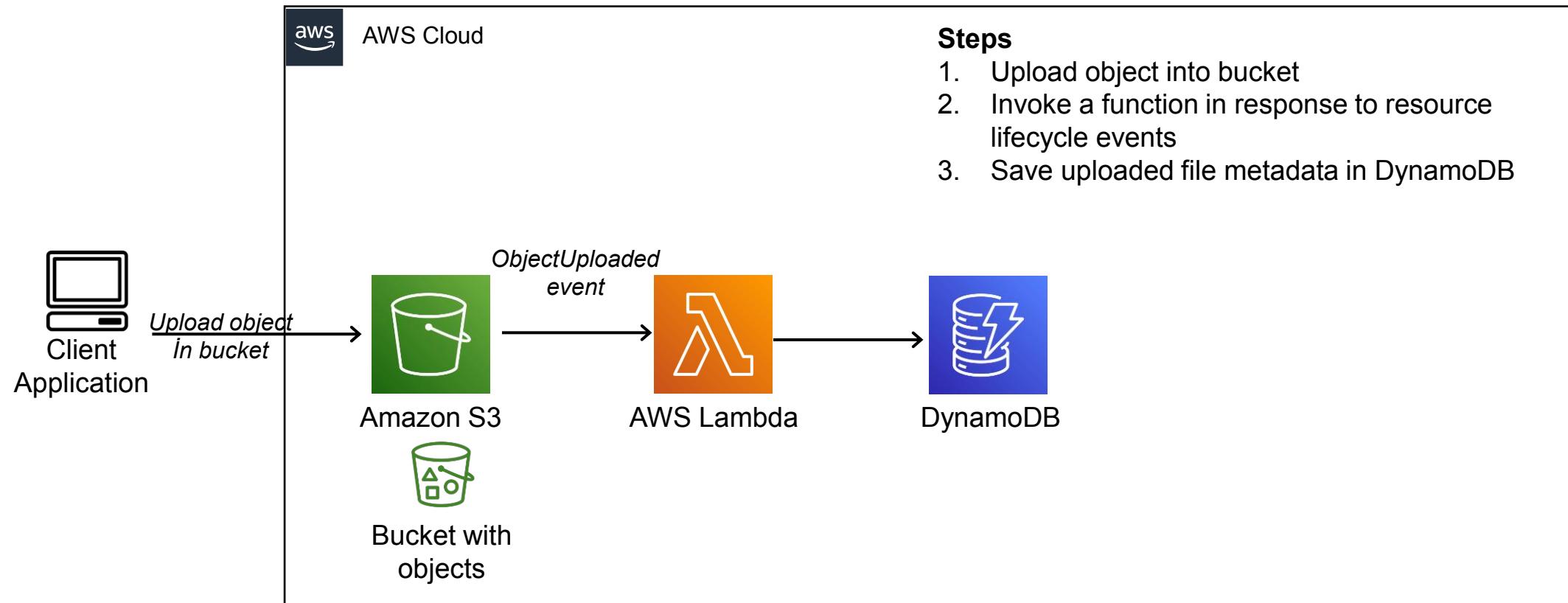
Hands-on Labs: Secure your API Gateway with Amazon Cognito User Pools



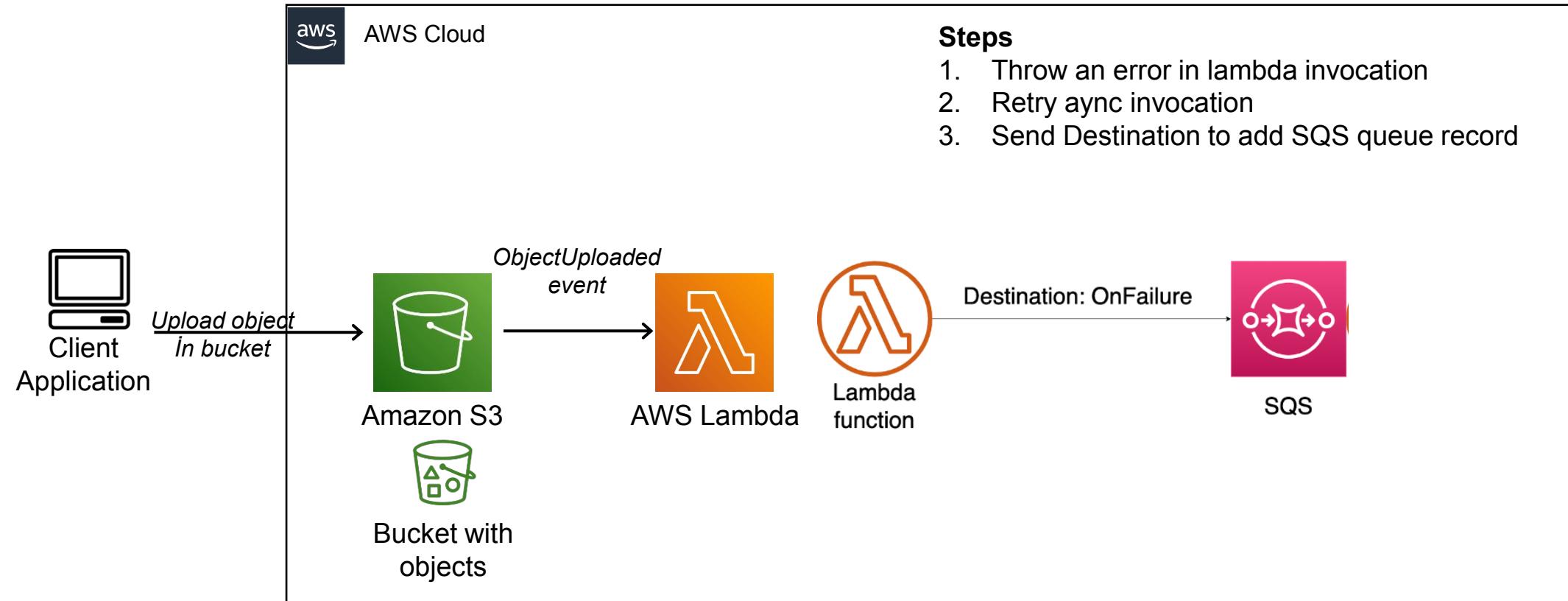
Hands-on Lab: Static Website Hosting on Amazon S3 using AWS Management Console



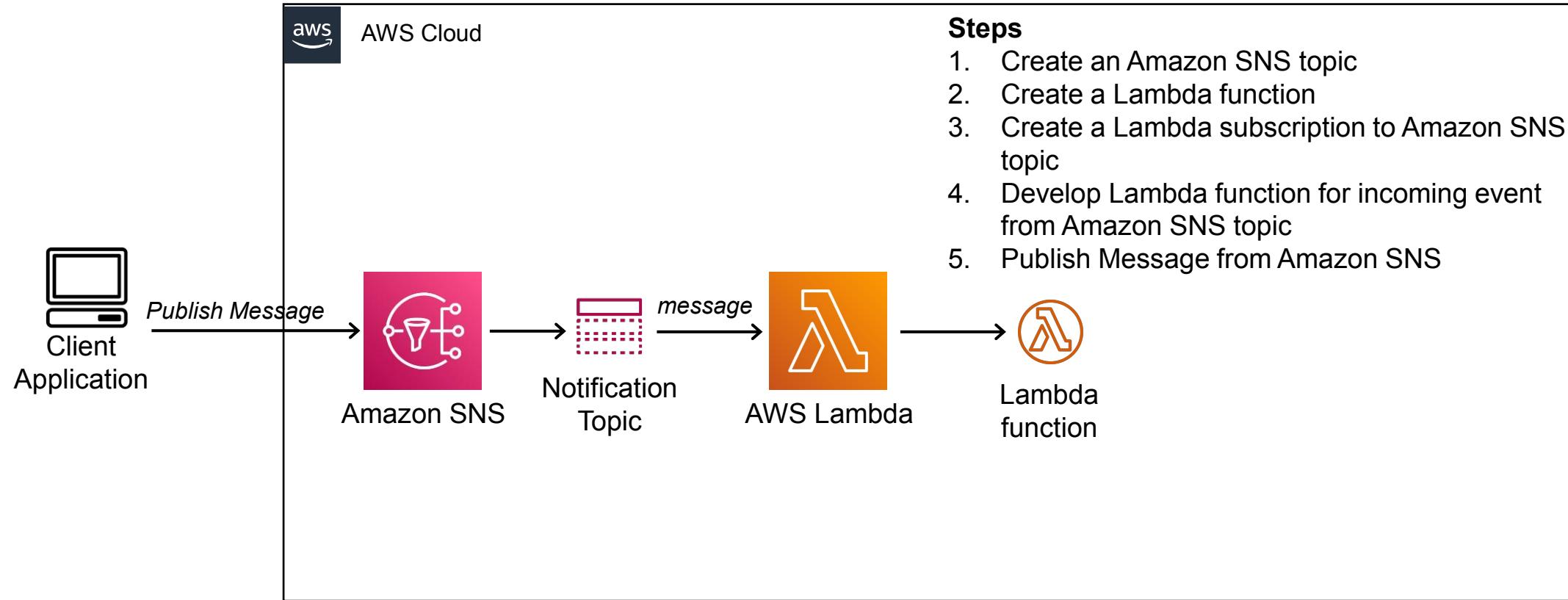
Hands-on Lab: Using an Amazon S3 trigger to invoke a Lambda function and persist on DynamoDB



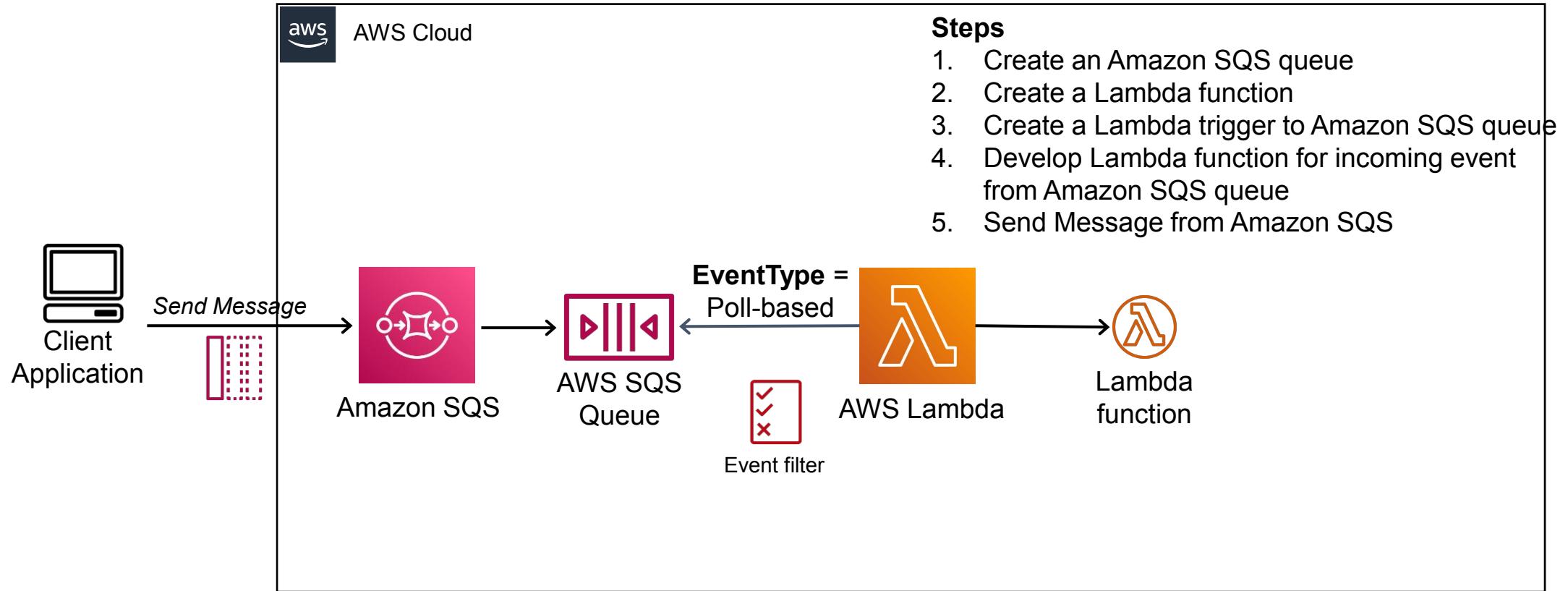
Hands-on Lab: AWS Lambda Destination to SQS - DLQ Case



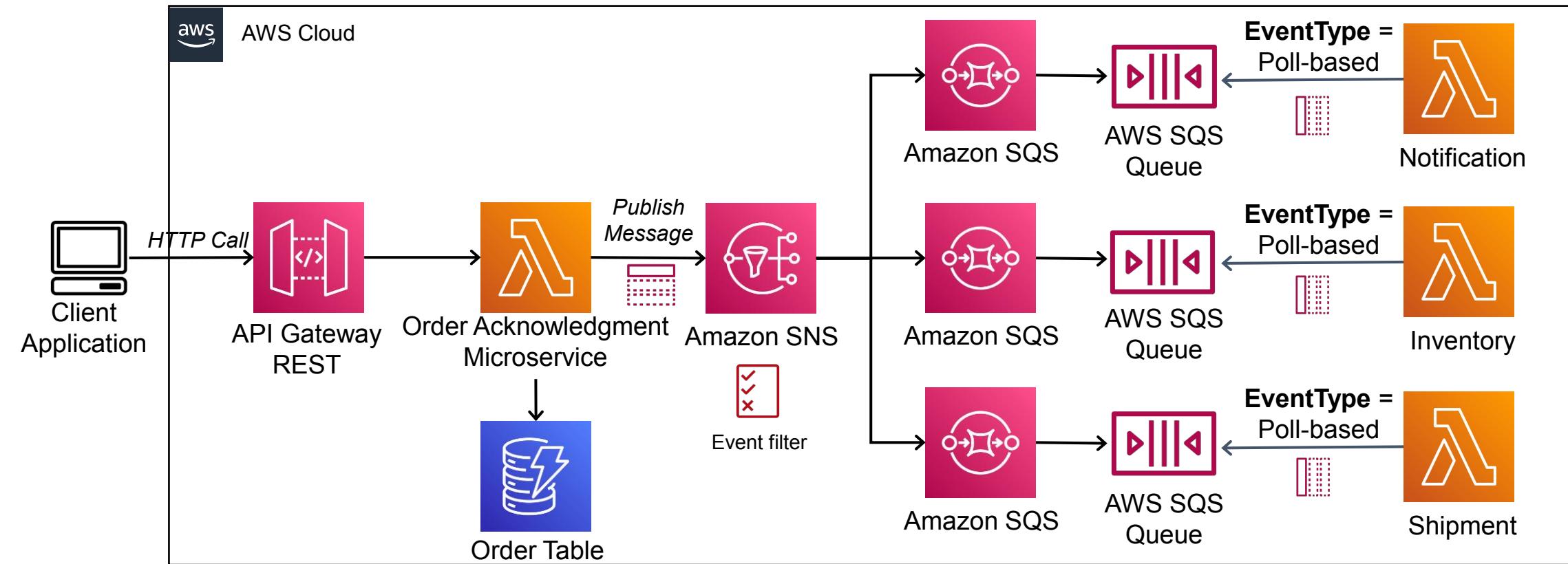
Hands-on Lab: Amazon SNS Notifications Subscribe From AWS Lambda



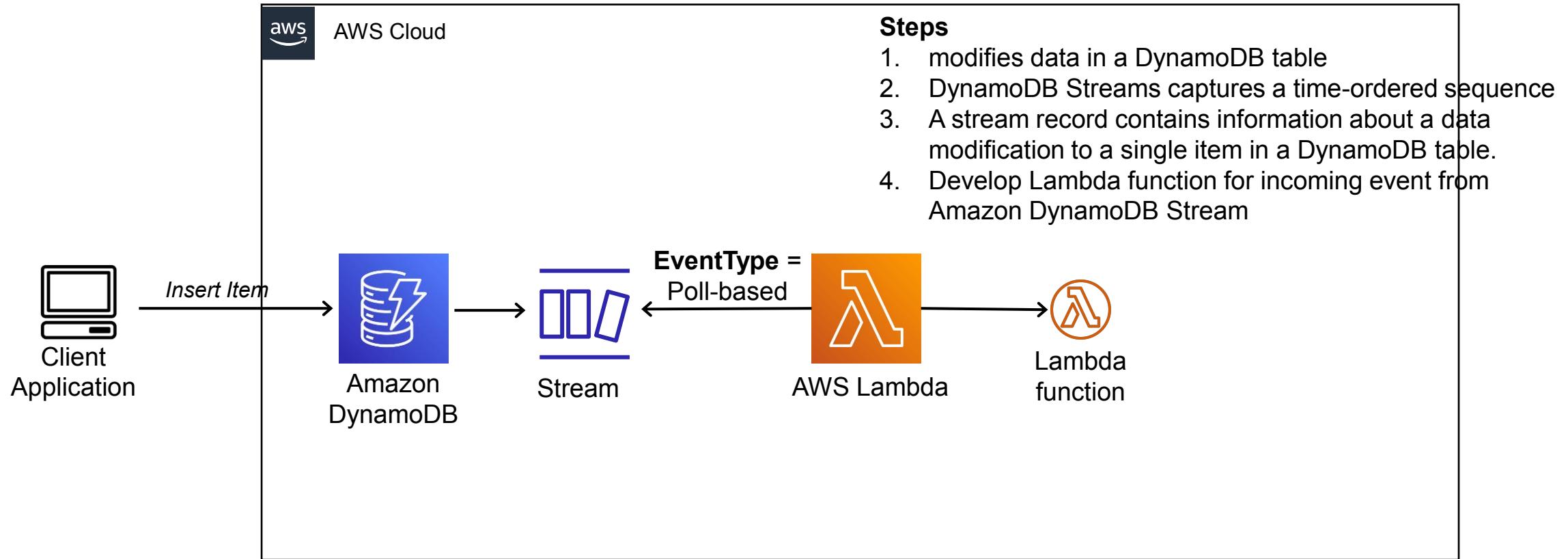
Hands-on Lab: Amazon SQS Queue Polling From AWS Lambda



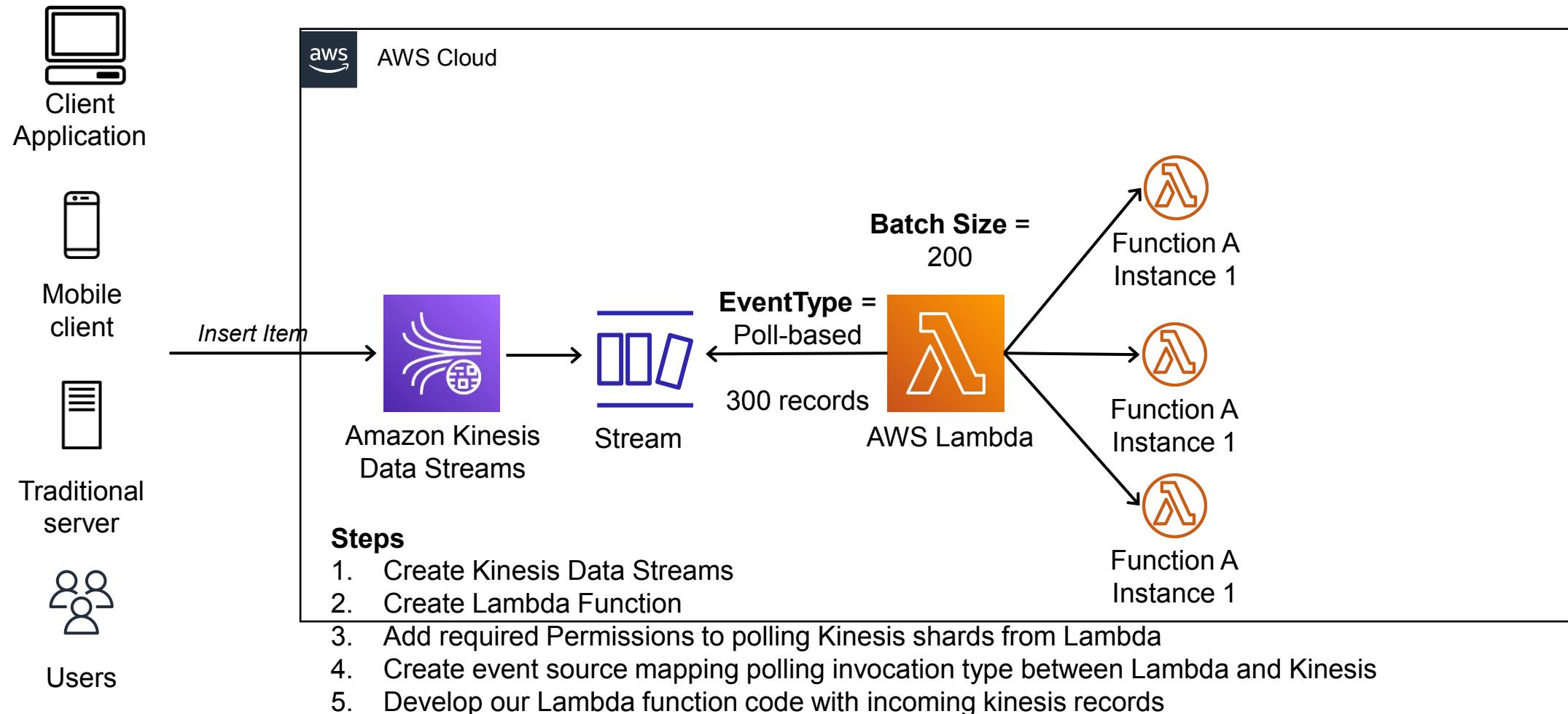
Hands-on Lab: Fan-Out Serverless Architectures Using SNS, SQS and Lambda



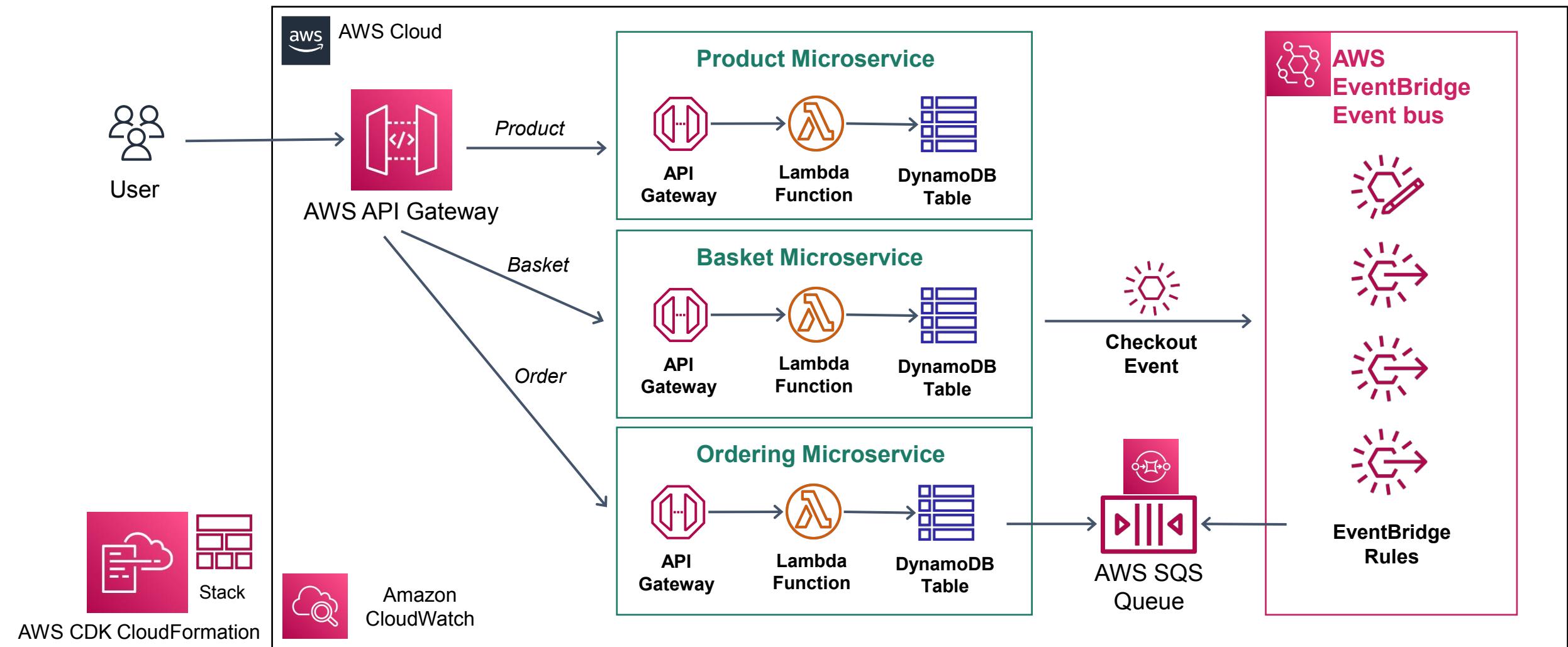
Hands-on Lab: Process DynamoDB Streams using AWS Lambda for Change Data Capture of DynamoDB Tables



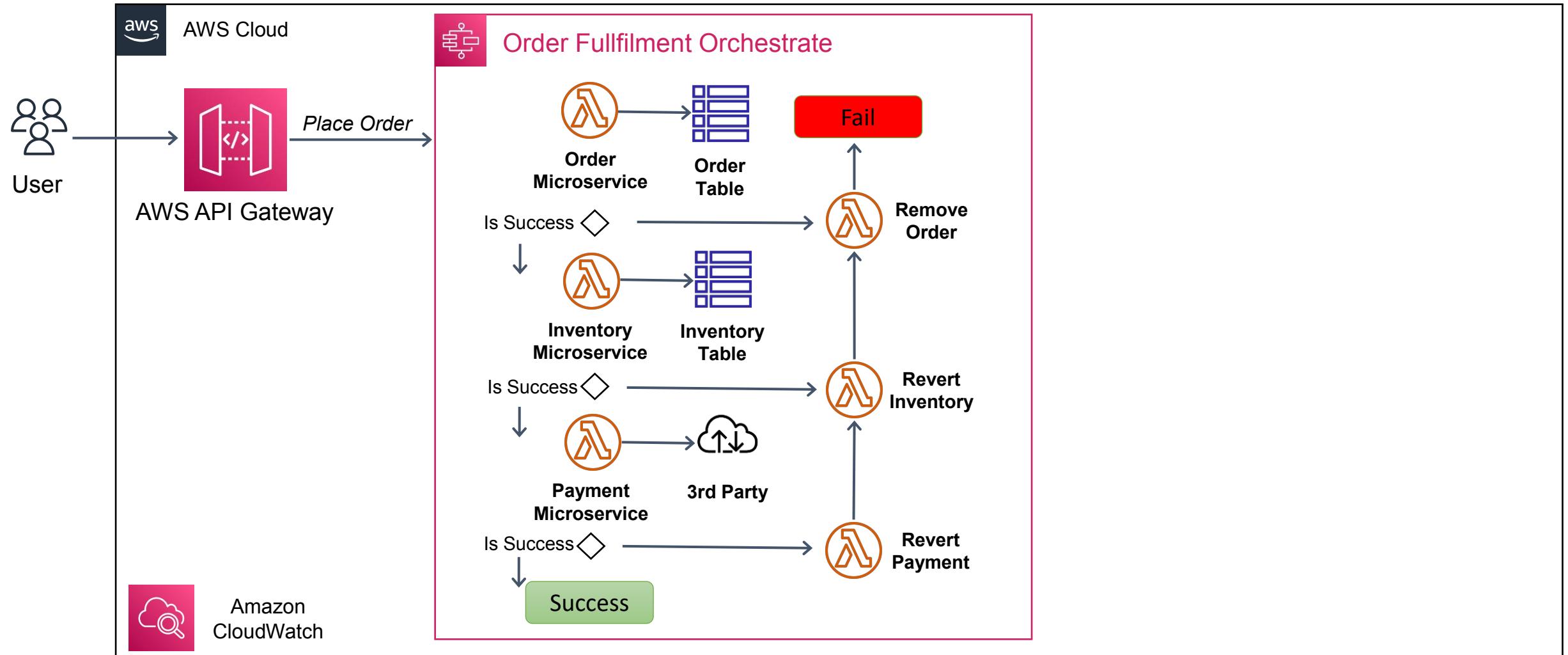
Hands-on Lab: Process Kinesis Streams using AWS Lambda



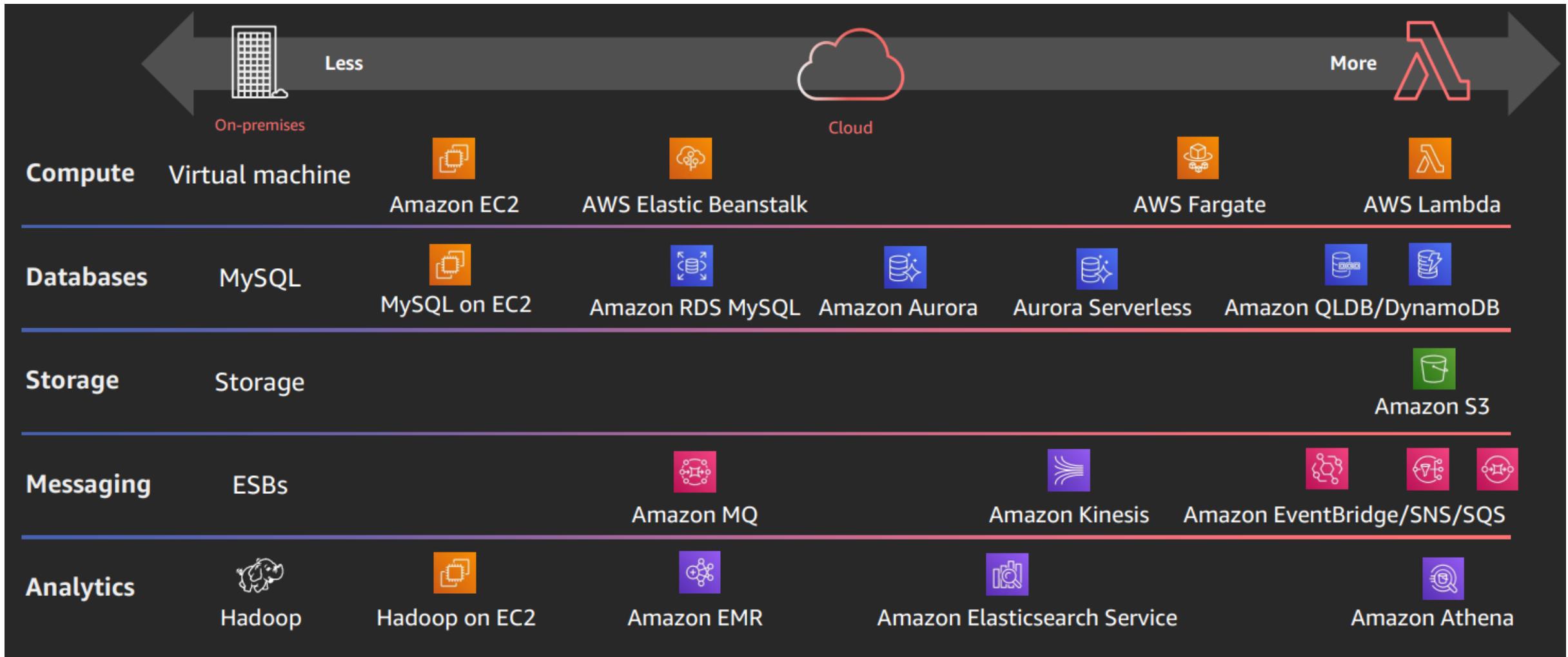
Hands-on Lab: AWS Serverless Microservices for Ecommerce using Amazon EventBridge



Hands-on Lab: Saga Pattern for Orchestrate Distributed Transactions using AWS Step Functions



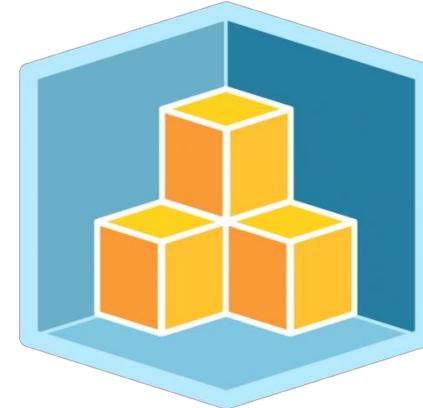
Serverless Explained : AWS Operational Responsibility Model



https://d1.awsstatic.com/events/reinvent/2019/REPEAT_3_Serverless_architectural_patterns_and_best_practices_ARC307-R3.pdf

Serverless Deployment Frameworks - IaC with AWS CDK

- **Cloud stack development** - IaC with Serverless Deployment Frameworks
- **Develop infrastructure** for all AWS services and perform interactions with coding interactions.
- **AWS CloudFormation, SAM, CDK** - Develop IaC Serverless Framework
- Develop combination of **AWS Serverless** services both infrastructure code and also actual lambda function code which interacts with other AWS services using **AWS SDK libraries**.
- This course is **%100 hand-on course** and include lots of development activities.



Project Code

- **Github Repository**
Find full source code on Github repository
- **Section by Section Github Repository**
- [awsrun organization](#) and created [aws-serverless repository](#)
- Clone or download both the repositories on GitHub
- Shared the links in the videos resources
- Ask questions from Q&A section

AWS Access Types - Programmatic / Management Console Access

- When creating a user, we have an option about **AWS Access Types**
 - Programmatic Access
 - AWS Management Console Access
- **Programmatic Access**
Enables **access key ID** and **secret access key** for the **AWS API, CLI, SDK**, and other development tools.
- During the course we will almost use all of these programmatic access types and of course use AWS Console every time

Add user

Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name*

[Add another user](#)

Select AWS access type

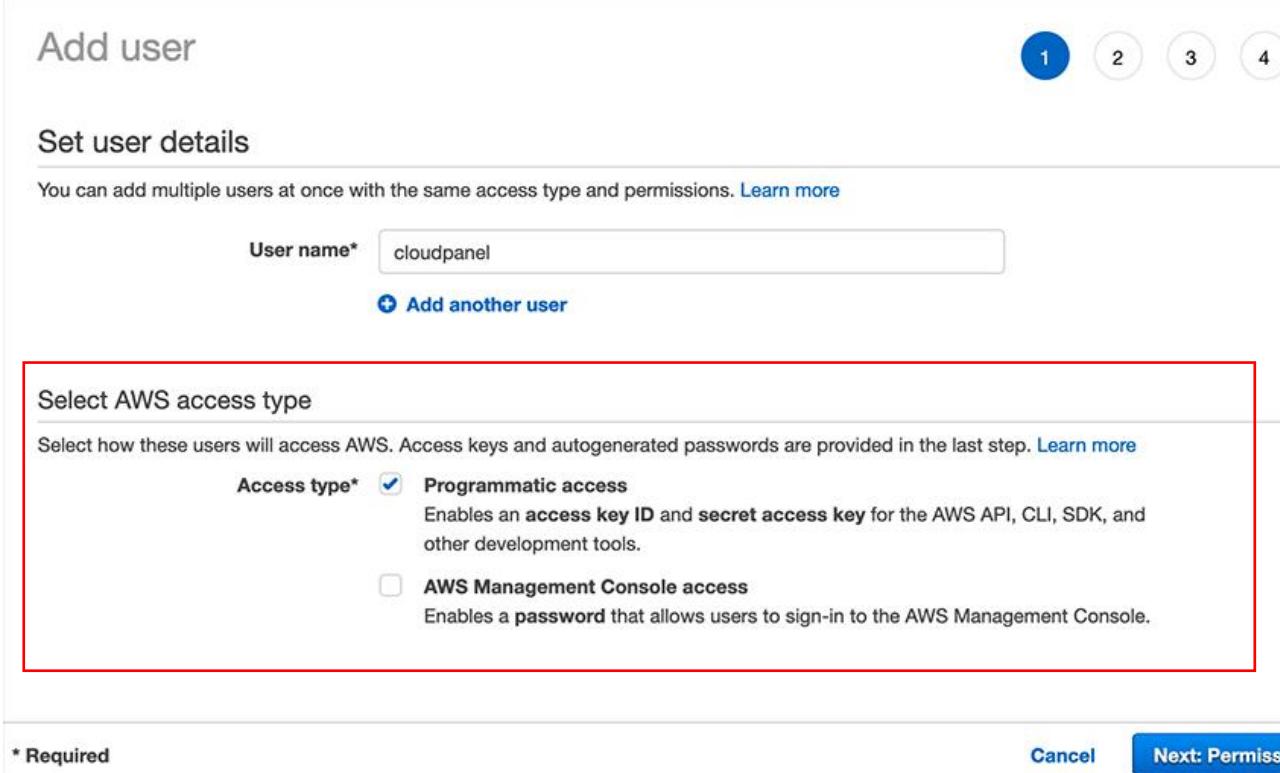
Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

Access type* **Programmatic access**
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

AWS Management Console access
Enables a password that allows users to sign-in to the AWS Management Console.

* Required

[Cancel](#) [Next: Permissions](#)



Programmatic Access

- **Programmatic Access**

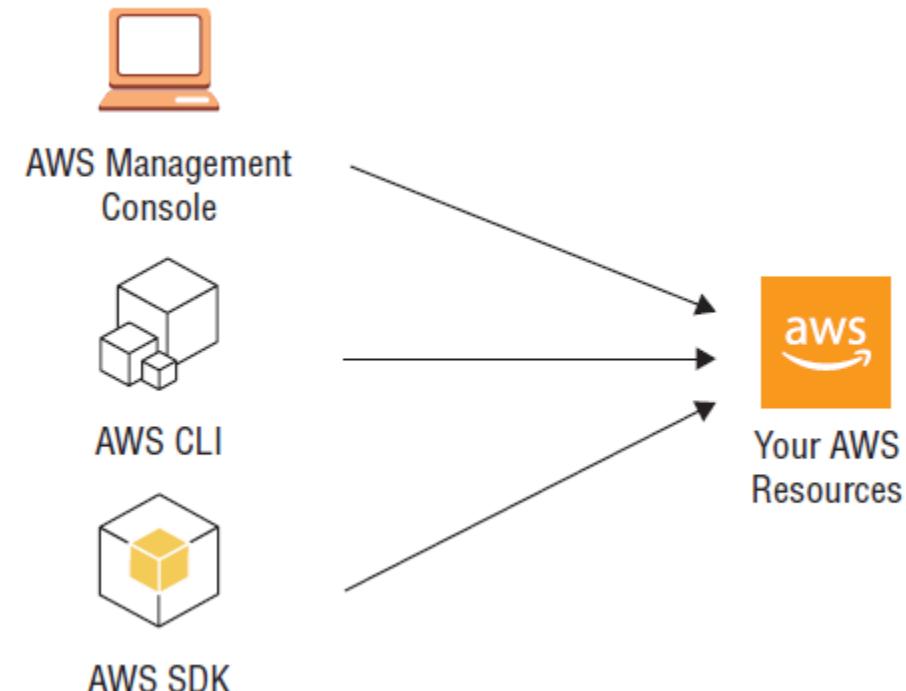
Gives us to manage AWS resources from our development environments and manage by writing codes.

- AWS CLI
- AWS SDK
- AWS Cloud Formation - IaC
 - AWS SAM
 - AWS CDK

- **AWS Command Line Interface (CLI)**

Unified tool to manage your AWS services.

- Control multiple AWS services from the command line and automate them through scripts.
- `$ aws ec2 describe-instances`
- `$ aws ec2 start-instances --instance-ids i-1348636c`



Programmatic Access - AWS SDK

- **AWS SDK - Software Development Kit**

Simplifies use of AWS Services by providing a set of libraries that are consistent and familiar for developers.

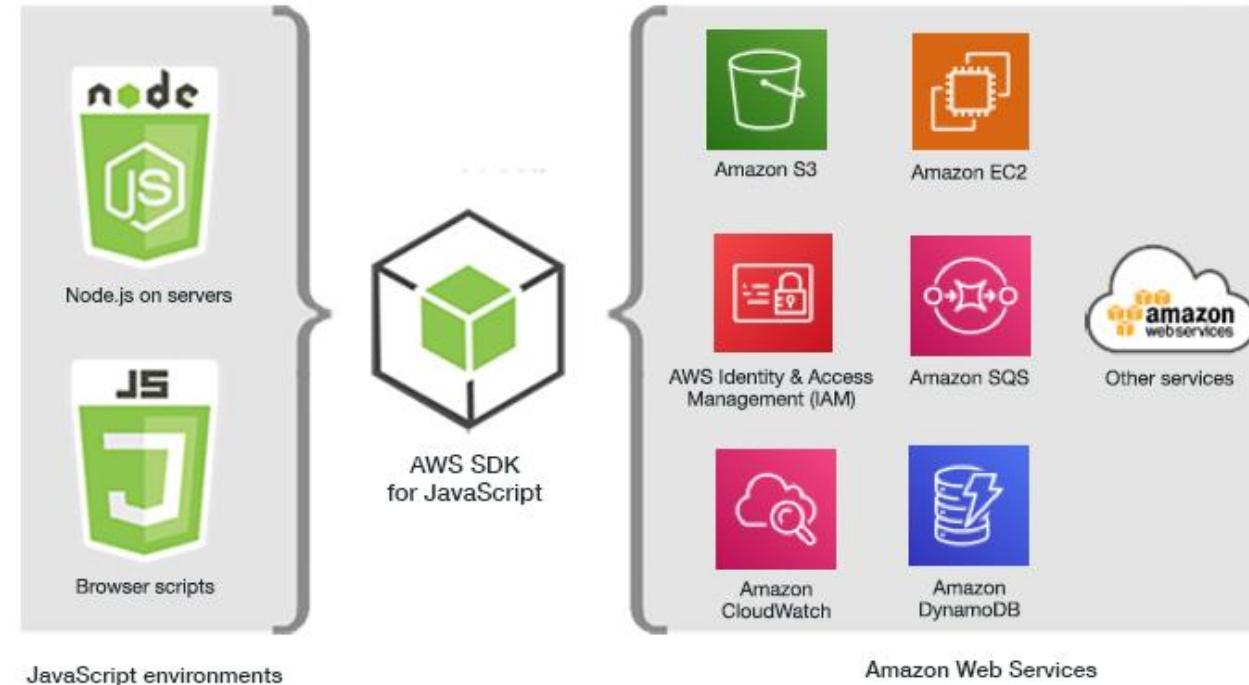
- Several programming languages AWS SDK packages that you can use, like Java, NodeJS, Javascript, .Net, Go and so on.

- Tools to Build on AWS

Tools for developing and managing applications on AWS

- Most common use cases, perform crud operations on DynamoDB table in your application code with using AWS SDK libraries

- Microservices codes when interacting with AWS DDynamoDB, EventBridge and SQS.



Programmatic Access - AWS CloudFormation and AWS CDK

- **AWS CloudFormation**

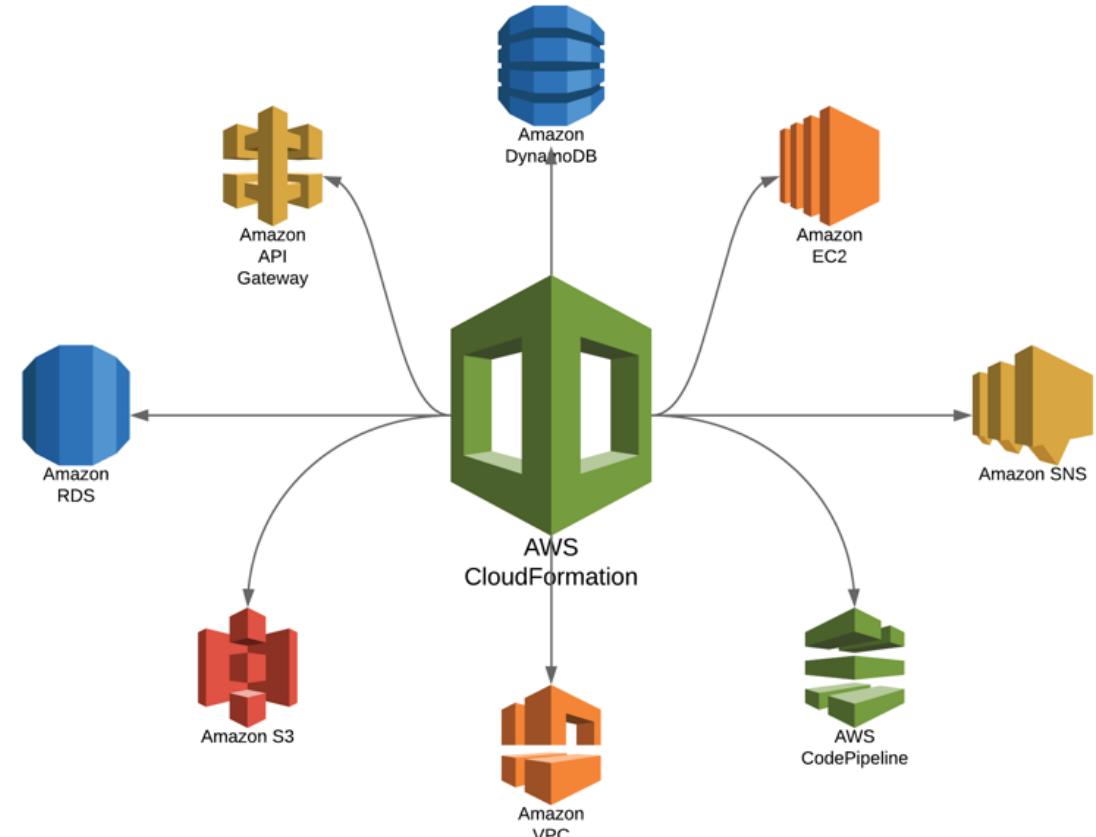
Infrastructure as code (IaC) service that allows you to easily model, provision, and manage AWS resources.

- AWS SAM
- AWS CDK

- **AWS CDK - Cloud Development Kit**

Open-source software development framework to define your cloud application resources using familiar programming languages.

- Provisioning cloud infrastructure with using Java, Typescript, Javascript, .Net, go and so on.
- During the course we will install and use all of these access types.



<https://aws.amazon.com/cloudformation/>

AWS Lambda - Serverless, Event-driven Compute Service

- AWS Lambda Main Features, Event Sources and Destinations, Invocation Types, Function Code, Execution Environment and Configurations.

What is AWS Lambda ? - Summarized

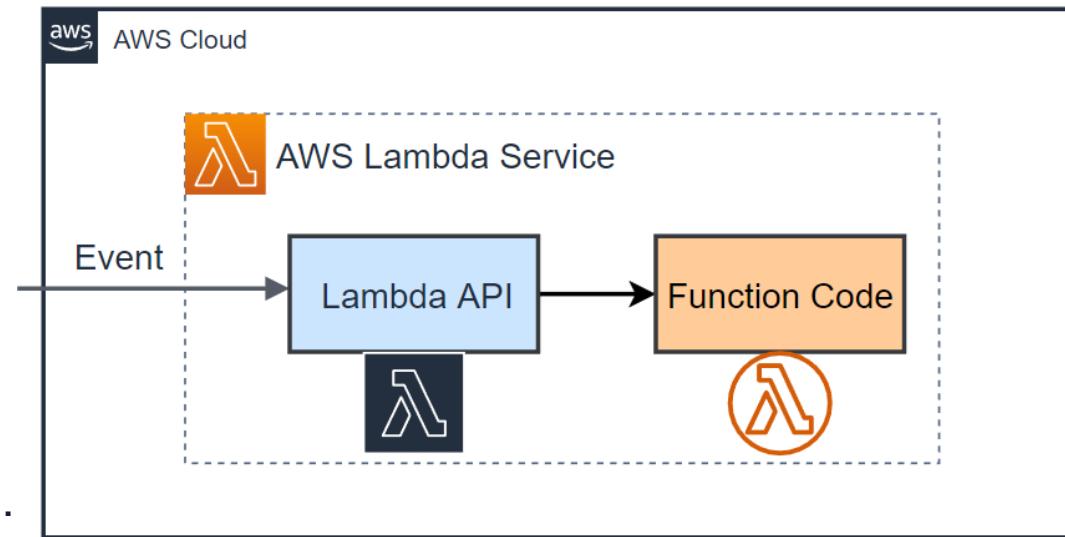
- Serverless, **Event-driven** compute service
- Trigger Lambda from over **200 AWS** services
- Run code without provisioning or managing infrastructure. Simply **write** and **upload** code as a .zip file or container image.
- Code execution requests at any **scale**
- **Pay-as-you-go**; Save costs by paying only for the compute time you use—by per-millisecond—instead of provisioning infrastructure
- Optimize **code execution** time and **performance** with the right function memory size.
- Respond to **high demand** in double-digit milliseconds with Provisioned Concurrency.



AWS Lambda

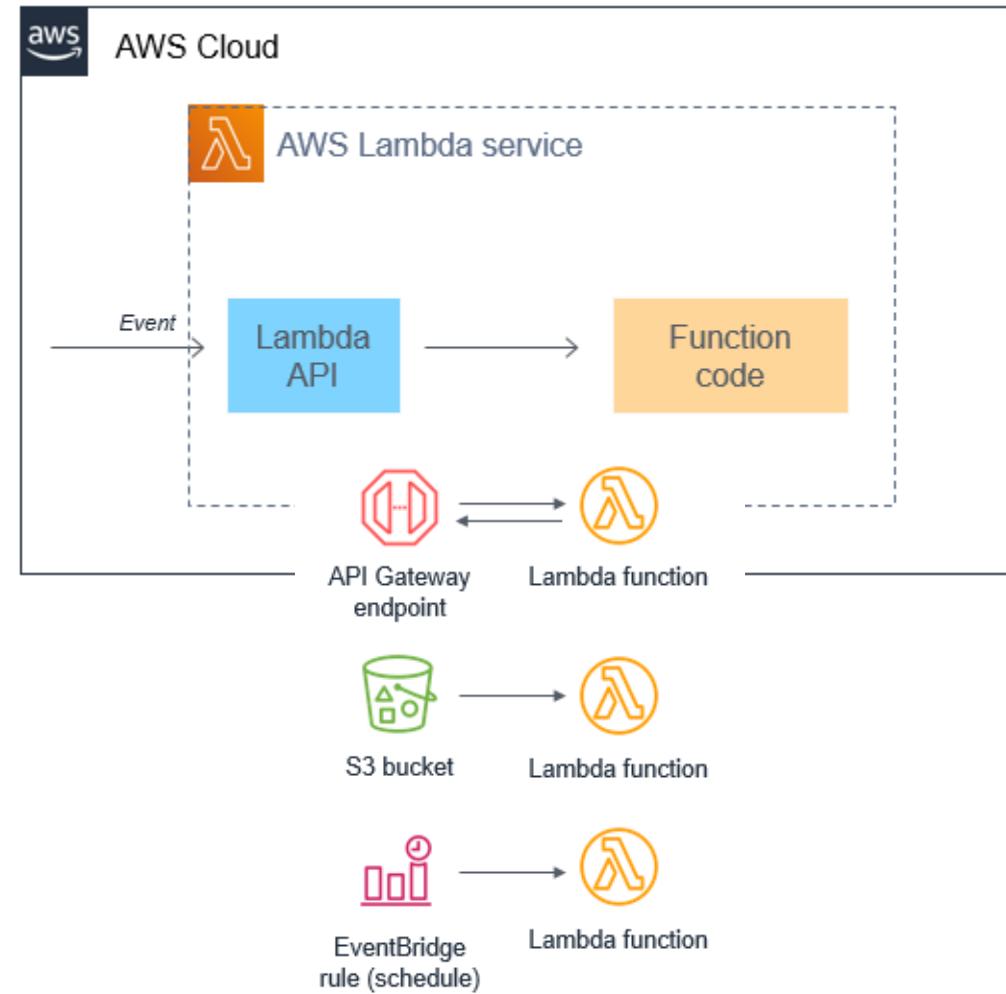
How does AWS Lambda work?

- Each Lambda function runs in its **own container**. You can think every lambda function as a **standalone docker containers**.
- When a function is created, Lambda **packages** it into a new **container** and then **executes that container** on a multi-region cloud clusters of servers managed by AWS.
- Each function's container is **allocated its necessary RAM** and **CPU capacity** that are **configurable** in AWS Lambda.
- **Charged based on the allocated memory** and the amount of **execution time** the function finished.
- AWS Lambda's entire infrastructure layer is **managed by AWS**.
- There is no infrastructure to maintain, you can spend more time on **application code** and your **actual business logics**.



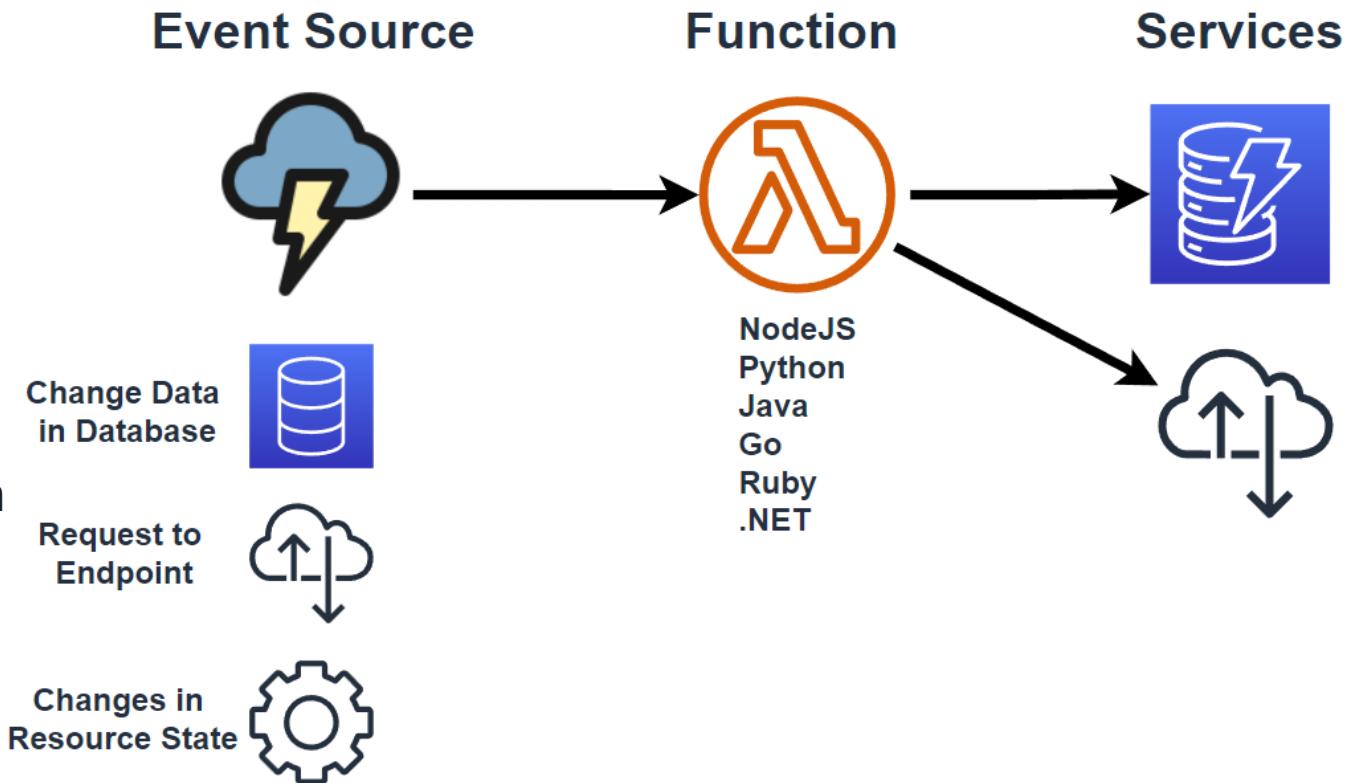
AWS Lambda Key Features

- There are several key features help you develop Lambda applications that are scalable, secure, and easily extensible
- Concurrency and scaling controls
- Functions defined as container images
- Code signing
- Lambda extensions
- Function plans
- Database access
- File systems access



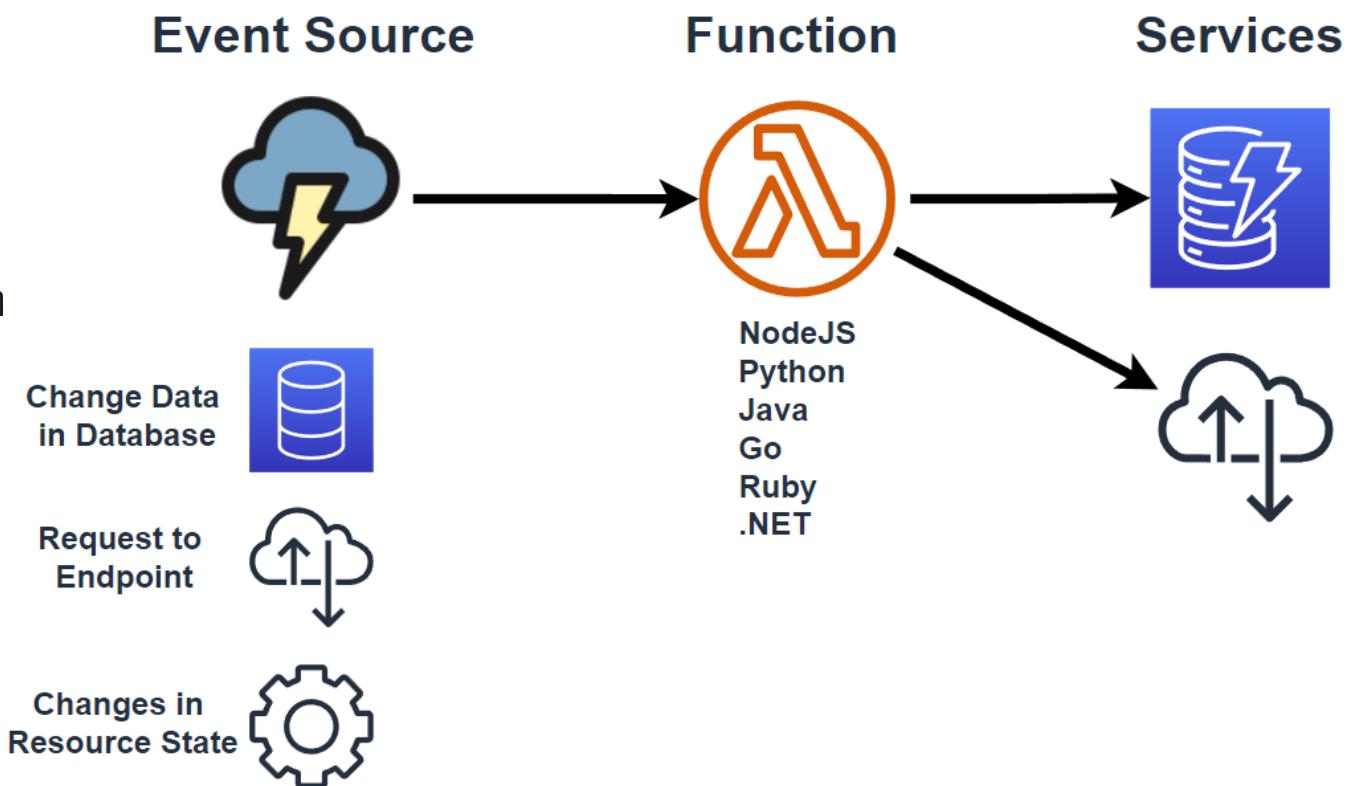
Lambda Event Sources and Destination Trigger Services

- AWS Lambda integrates with other AWS services to invoke functions or take other actions.
- There is an event source or trigger, and actual Function code and then there is the destination.
- Event source has a number of services; they can be a http call, cron job, uploading an object into S3 bucket, third party call like payment done through stripe
- Triggering event to lambda function, lambda launch the execution environment with different language and runtimes
- Lambda has destinations that can be interaction with your function code



Use Cases Lambda Event Sources and Destination Trigger Services

- Invoke a function in response to resource lifecycle events, such as with Amazon Simple Storage Service (Amazon S3)
- Respond to incoming HTTP requests. Using Lambda with API Gateway.
- Consume events from a queue. Using Lambda with Amazon SQS. Lambda poll queue records from Amazon SQS.
- Run a function on a schedule. Using AWS Lambda with Amazon EventBridge (CloudWatch Events).



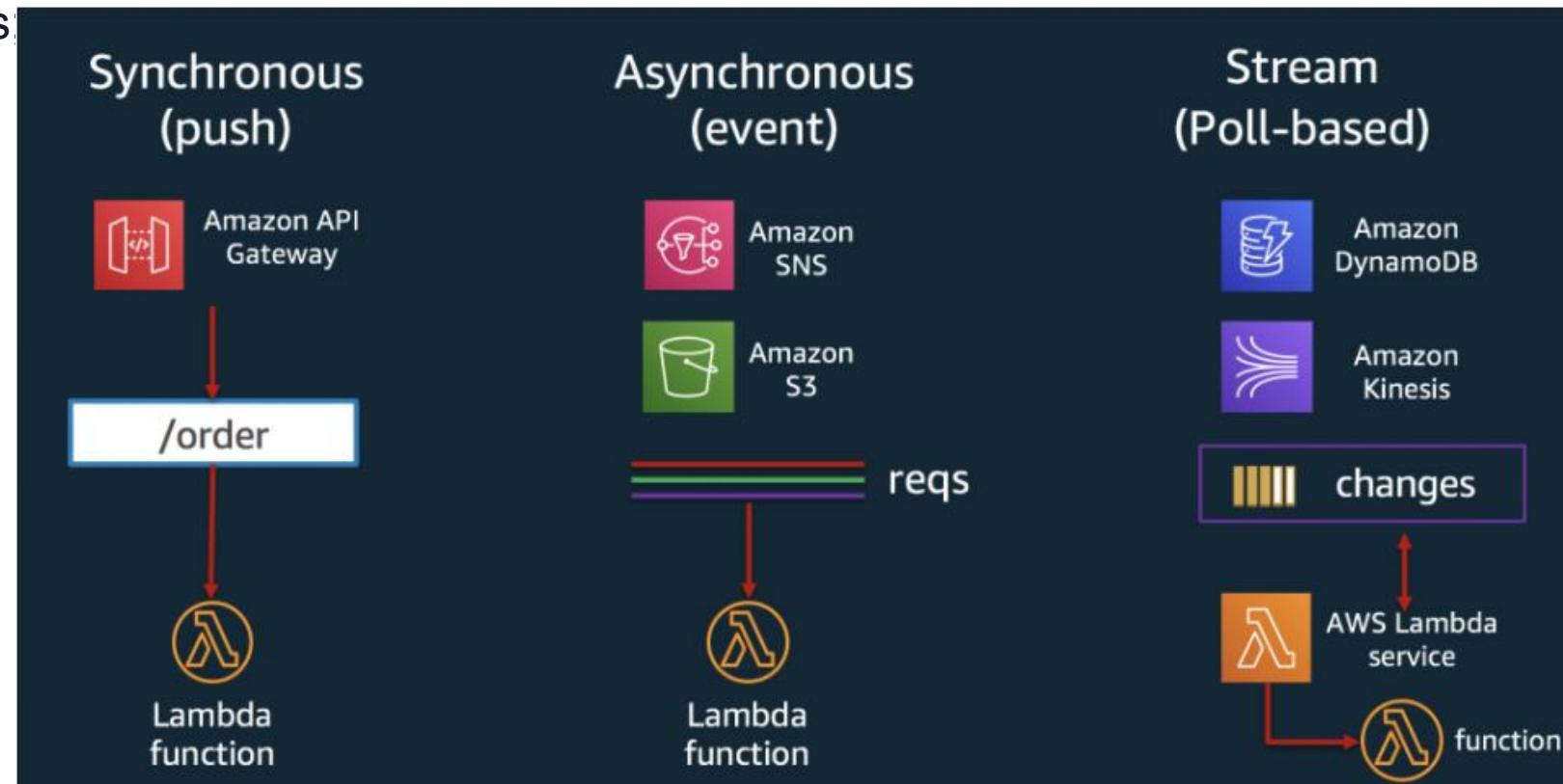
List of Services Lambda Event Sources

Service	Method of invocation
Amazon API Gateway	Event-driven; synchronous invocation
AWS CloudFormation	Event-driven; asynchronous invocation
Amazon CloudFront (Lambda@Edge)	Event-driven; synchronous invocation
Amazon EventBridge (CloudWatch Events)	Event-driven; asynchronous invocation
Amazon CloudWatch Logs	Event-driven; asynchronous invocation
AWS CodeCommit	Event-driven; asynchronous invocation
AWS CodePipeline	Event-driven; asynchronous invocation
Amazon Cognito	Event-driven; synchronous invocation
AWS Config	Event-driven; asynchronous invocation
Amazon Connect	Event-driven; synchronous invocation
Amazon DynamoDB	Lambda polling
Amazon Elastic File System	Special integration
Amazon Lambda (Application and API gateway)	Event-driven; asynchronous invocation

<https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html#eventsources-sqs>

AWS Lambda Invocation Types

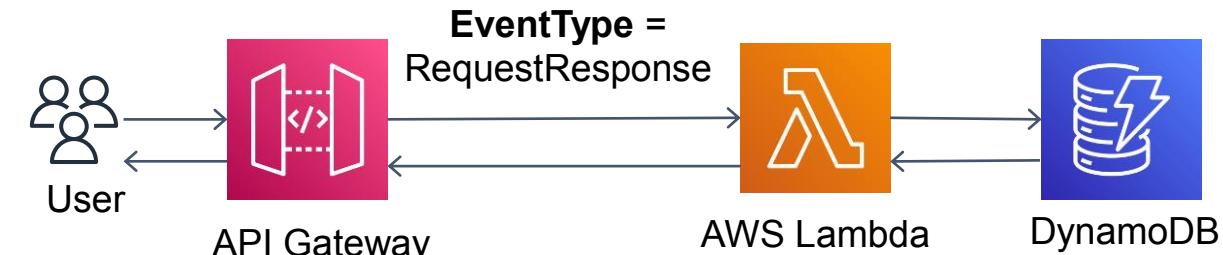
- Triggered lambda functions with different AWS Lambda Invocation Types
- AWS Lambda has 3 Invocation Types
- **Lambda Synchronous invocation**
- **Lambda Asynchronous invocation**
- **Lambda Event Source Mapping with polling invocation**



<https://aws.amazon.com/blogs/architecture/understanding-the-different-ways-to-invoke-lambda-functions/>

AWS Lambda Synchronous Invocation

- Execute immediately when you perform the Lambda Invoke API call.
- Wait for the function to process the function and return back to response.
- API Gateway + Lambda + DynamoDB
- Invocation-type flag should be “RequestResponse”
- Responsible for inspecting the response and determining if there was an error and decide to retry the invocation
- Example of synchronous invocation using the AWS CLI:
`aws lambda invoke —function-name MyLambdaFunction —invocation-type RequestResponse —payload '{ "key": "value" }'`
- Triggered AWS services of synchronous invocation; ELB (Application Load Balancer), Cognito, Lex, Alexa, API Gateway, CloudFront, Kinesis Data Firehose



AWS Lambda Asynchronous Invocation

- Lambda **sends the event** to a **internal queue** and returns a **success response** without any additional information
- Separate process **reads events** from the **queue** and **runs** our lambda function
- **S3 / SNS + Lambda + DynamoDB**
- **Invocation-type** flag should be “**Event**”
- AWS Lambda sets a **retry policy**

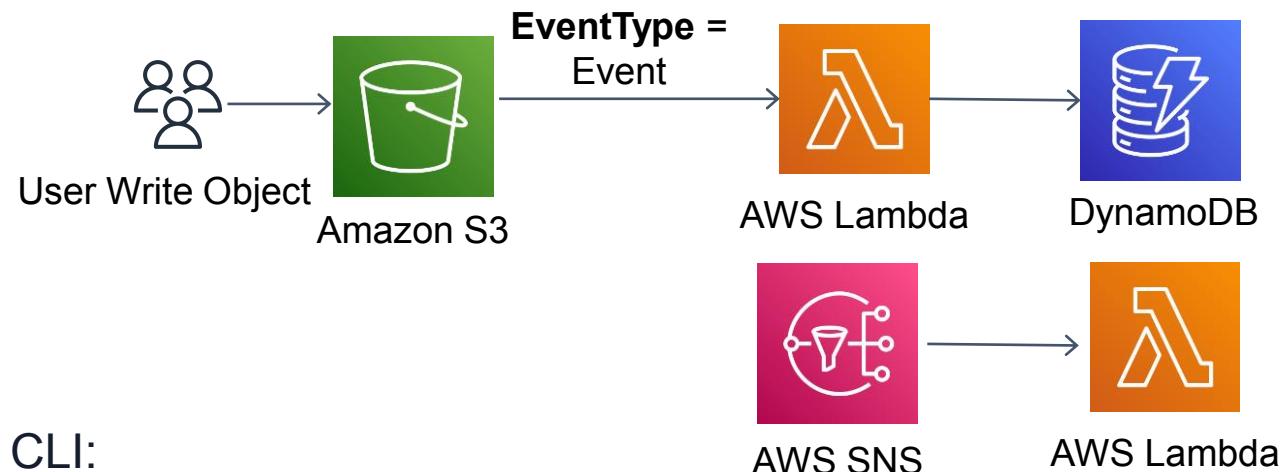
Retry Count = 2

Attach a Dead-Letter Queue (DLQ)

- Example of asynchronous invocation using the AWS CLI:

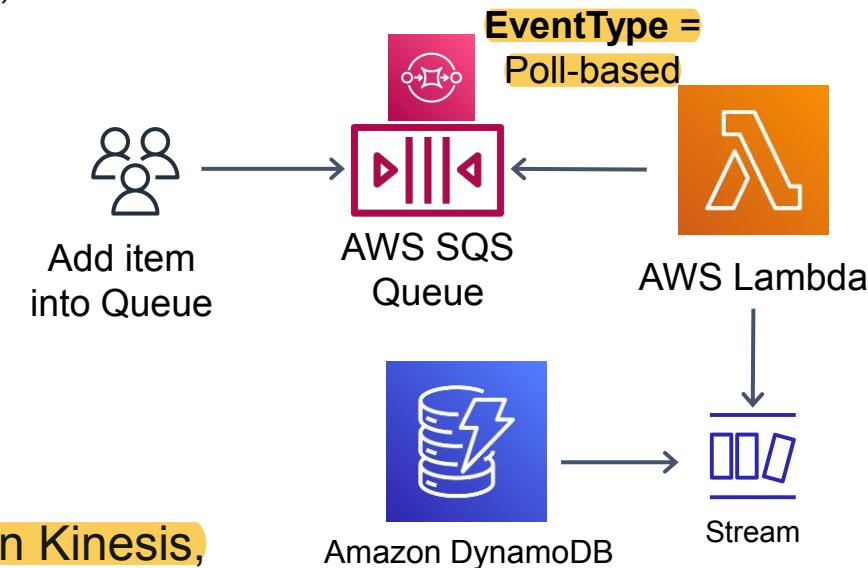
```
aws lambda invoke --function-name MyLambdaFunction --invocation-type Event --payload '{ "key": "value" }'
```

- Triggered **AWS services of asynchronous invocation**; S3, EventBridge, SNS, SES, CloudFormation, CloudWatch Logs, CloudWatch Events, CodeCommit



AWS Lambda Event Source Mapping with Polling Invocation

- **Pool-Based invocation** model allows us to **integrate** with **AWS Stream** and **Queue based services**.
- Lambda will **poll** from the AWS SQS or Kinesis streams, retrieve records, and invoke functions.
- Data stream or queue are **read in batches**,
- The function receives multiple items when execute function.
- **Batch sizes** can configure according to service types
- **SQS + Lambda**
- **Stream based processing** with **DynamoDB Streams + Lambda**
- Triggered **AWS services** of **Event Source Mapping invocation**; Amazon Kinesis, DynamoDB, Simple Queue Service (SQS)



Lambda Function Code

- AWS Lambda runs instances of your function to process events. Invoke function directly using the Lambda API, or configure an AWS service or resource to invoke your function.
- Lambda function has code to **process the events** that you pass into the function or that other AWS services send to the function with event json object.
- The **event object** contains all the information about the event that triggered this Lambda.
- The **context object** contains info about the runtime our Lambda function
- Return the **callback function** with the results

```
export const handler = async (event, context, callback) => {
  console.log("event:", JSON.stringify(event, undefined, 2));

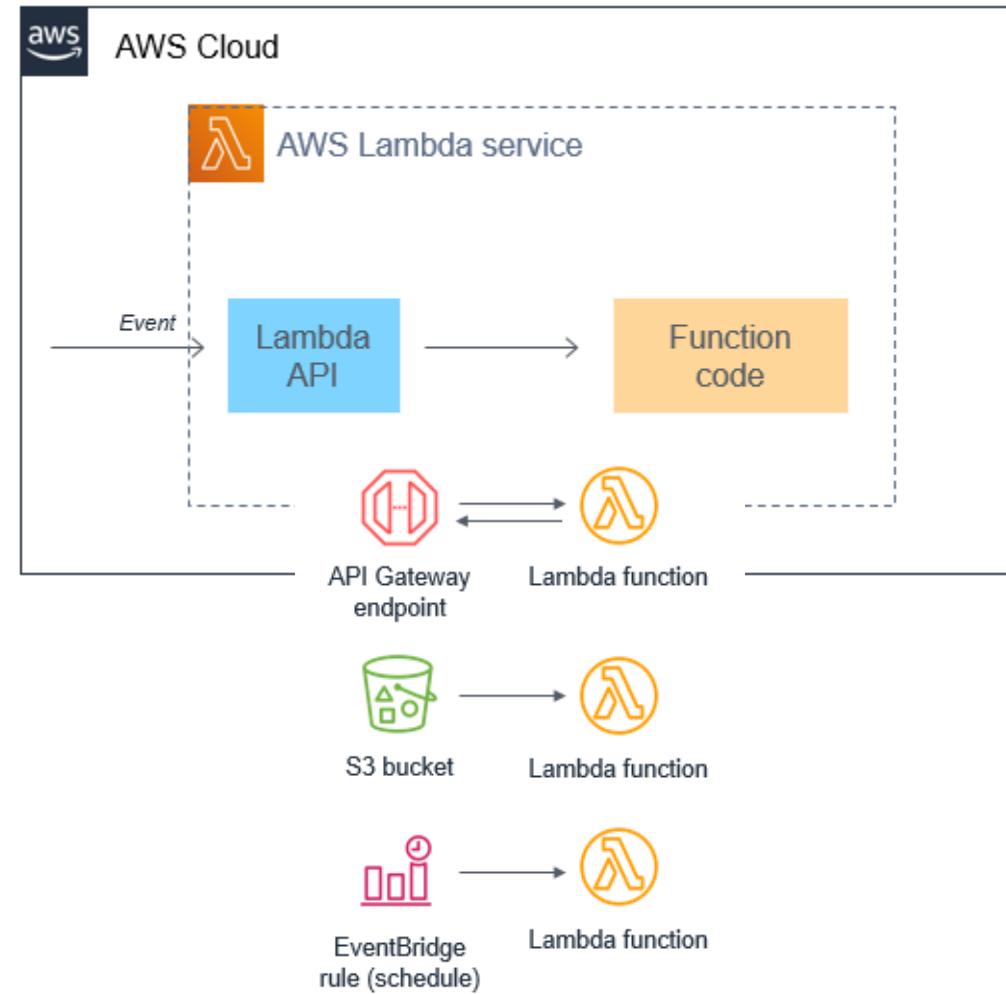
  // do stuff

  const response = {
    statusCode: 200,
    body: JSON.stringify({ "message": "success" })
  }

  callback(null, response);
}
```

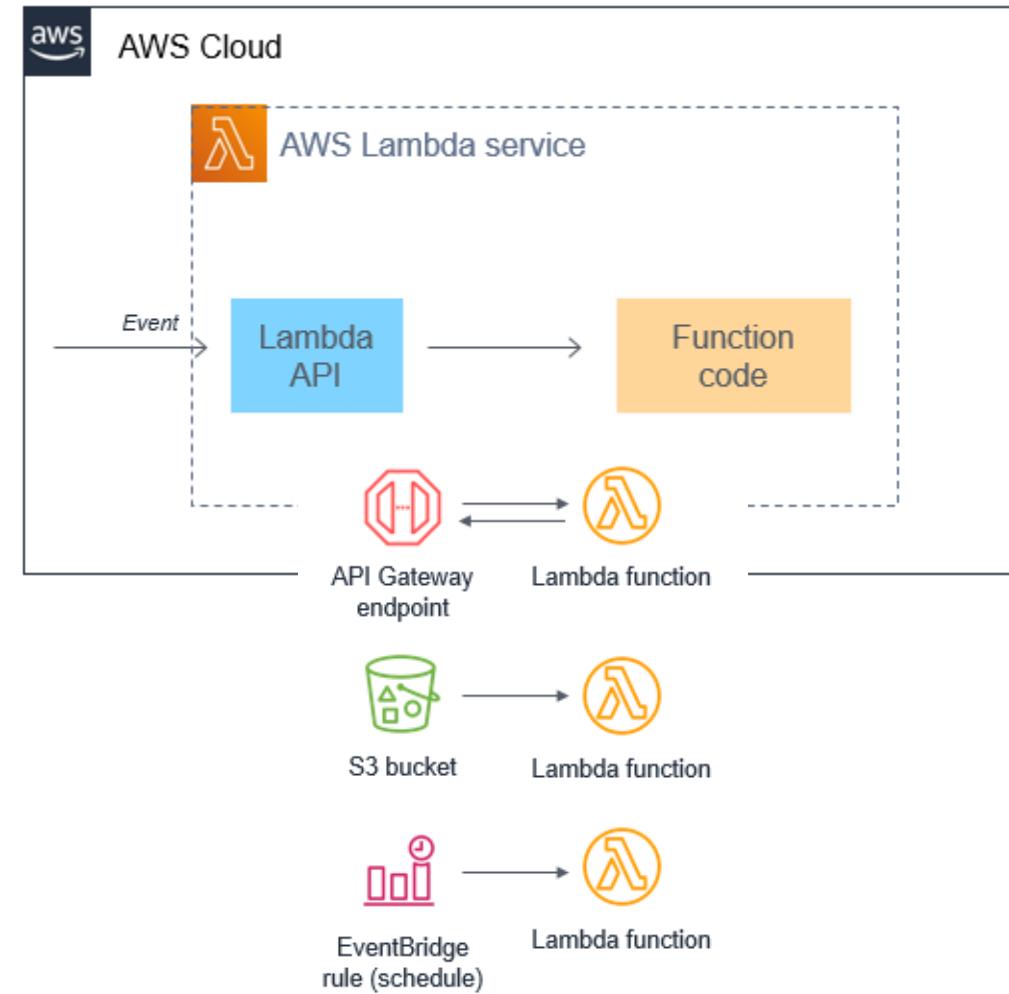
Key concepts of Lambda Function Code

- **Runtime:** select the runtime as part of configuring the function, and Lambda loads that runtime when initializing the environment.
- **Handler:** function runs starting at the handler method.
- **Function:** is a resource that you can invoke to run your code in Lambda.
- **Trigger:** is a resource or configuration that invokes a Lambda function.
- **Event:** is a JSON-formatted document that contains data for a Lambda function to process.
- **Execution environment:** provides a secure and isolated runtime environment for your Lambda function.



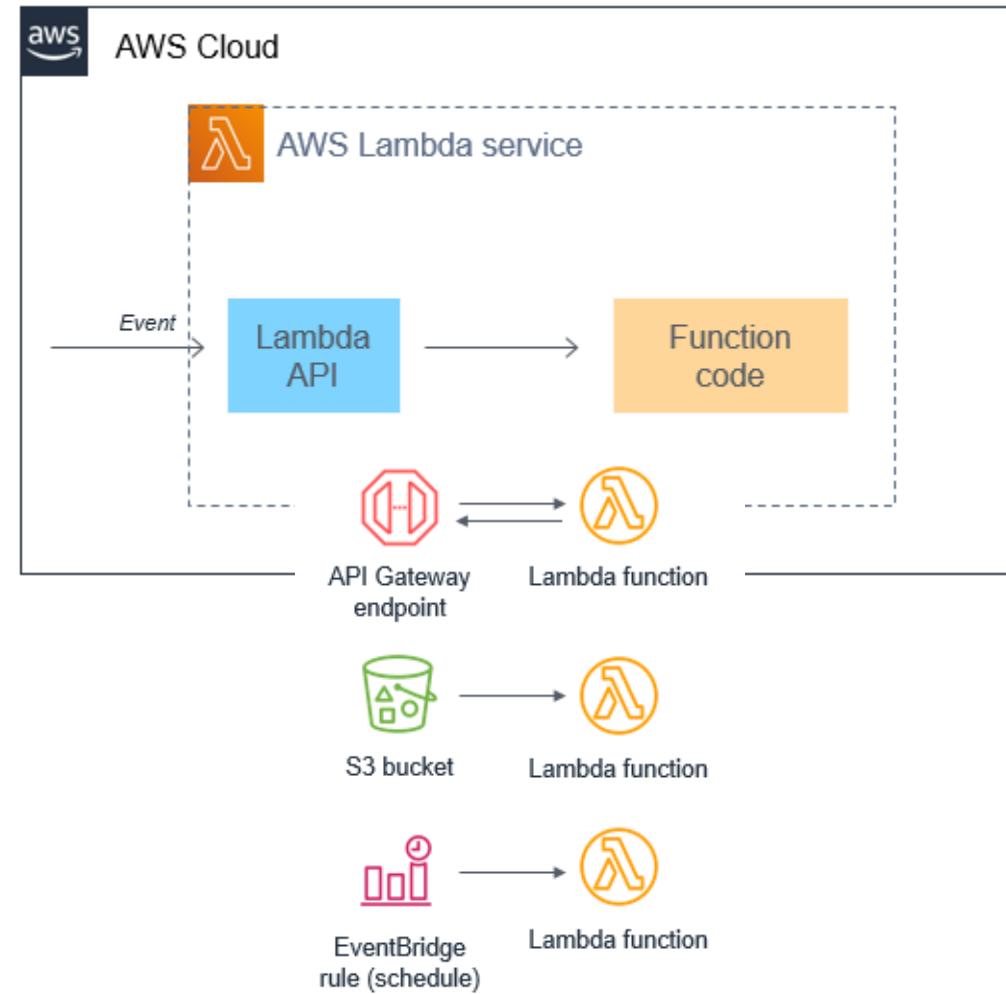
Key concepts of Lambda Function Code

- **Layer:** can contain libraries, a custom runtime, data, or configuration files. Using layers reduces the size of uploaded deployment archives and makes it faster to deploy your code.
- **Concurrency:** is the number of requests that your function is serving at any given time. When your function is invoked, Lambda provisions an instance of it to process the event. When the function code finishes running, it can handle another request.
- **Destination:** is an AWS resource where Lambda can send events from an asynchronous invocation. configure a destination for events that fail processing like setting DLQ for Lambda fails.



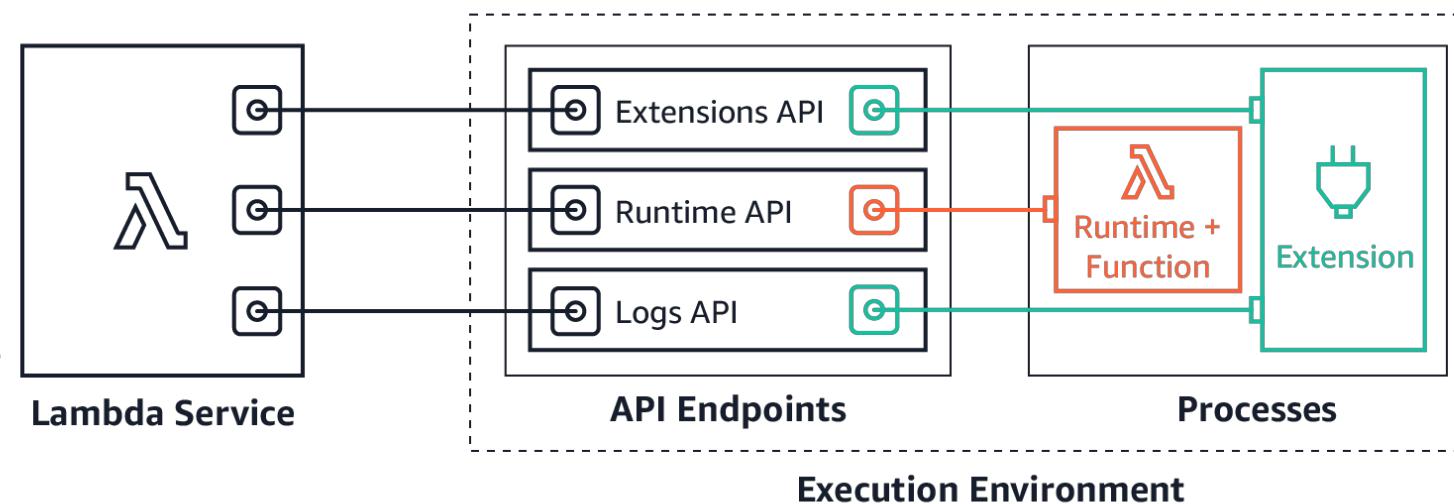
Best Practices of Lambda Function Code

- Take advantage of **environment reuse**, and check that background processes have completed.
- Manage **database connection pooling** with a database proxy. Persist state data externally.
- Configure Function with **Resource-based policy**. Resource-based policy grants permissions to invoke your lambda function. Execution role defines a function's permission to interact with resources.
- Use **Environment variables** to store secrets securely and adjust your function's behavior without updating code. An environment variable is a pair of strings that are stored in a function's version-specific configuration.



AWS Lambda Execution Environment

- When AWS Lambda invokes our function, it creates an **execution environment** to **isolated** runtime environment.
- The **execution environment** provides the required **resources** to run our function. Also provides **lifecycle support**.
- The function communicates with Lambda Runtime using the **Runtime API**. Extensions communicate with Lambda using the **Extensions API**.
- Lambda configurations like amount of **Memory**, **CPU usage** and **maximum execution time** settings. With these settings Lambda creates execution environments.



<https://docs.aws.amazon.com/lambda/latest/dg/runtime-environment.html>

Lambda Execution Environment Lifecycle

- **Init Phase**

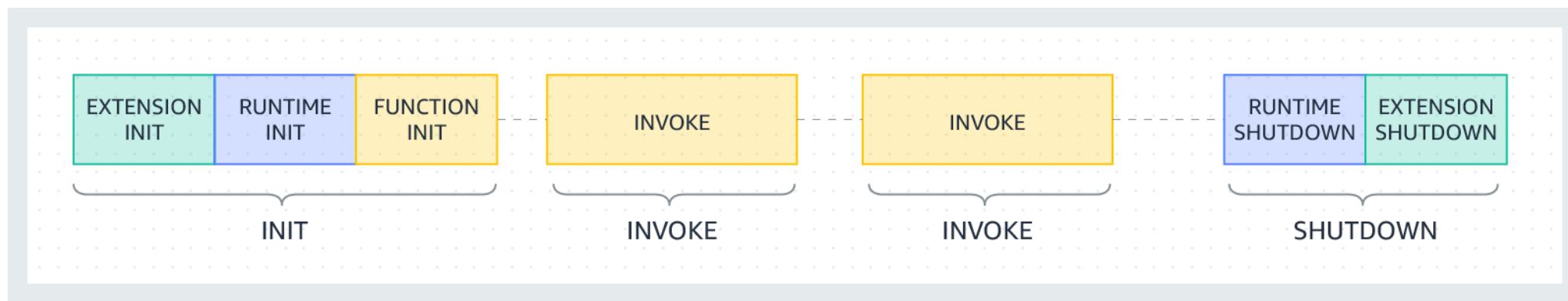
3 main tasks; Extension, Runtime and Function init. Ends when the runtime and all extensions are ready. Limited to 10 seconds. If not complete, Lambda retries the Init phase.

- **Invoke Phase**

Lambda invokes the function handler. Lambda prepares to handle another function invocation. Timeout setting limits the duration of the entire Invoke phase. Next API request invokes another function.

- **Shutdown Phase**

Happens if the Lambda function doesn't receive any invocations. Lambda shuts down the runtime, removes the environment. Lambda sends a Shutdown event to each extension.

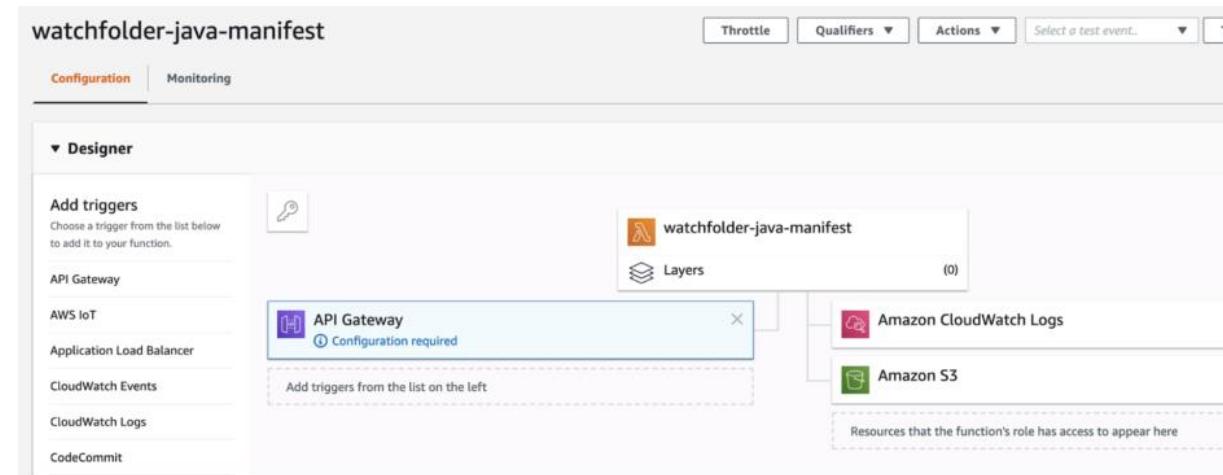


<https://docs.aws.amazon.com/lambda/latest/dg/runtime-environment.html>

AWS Lambda Configurations

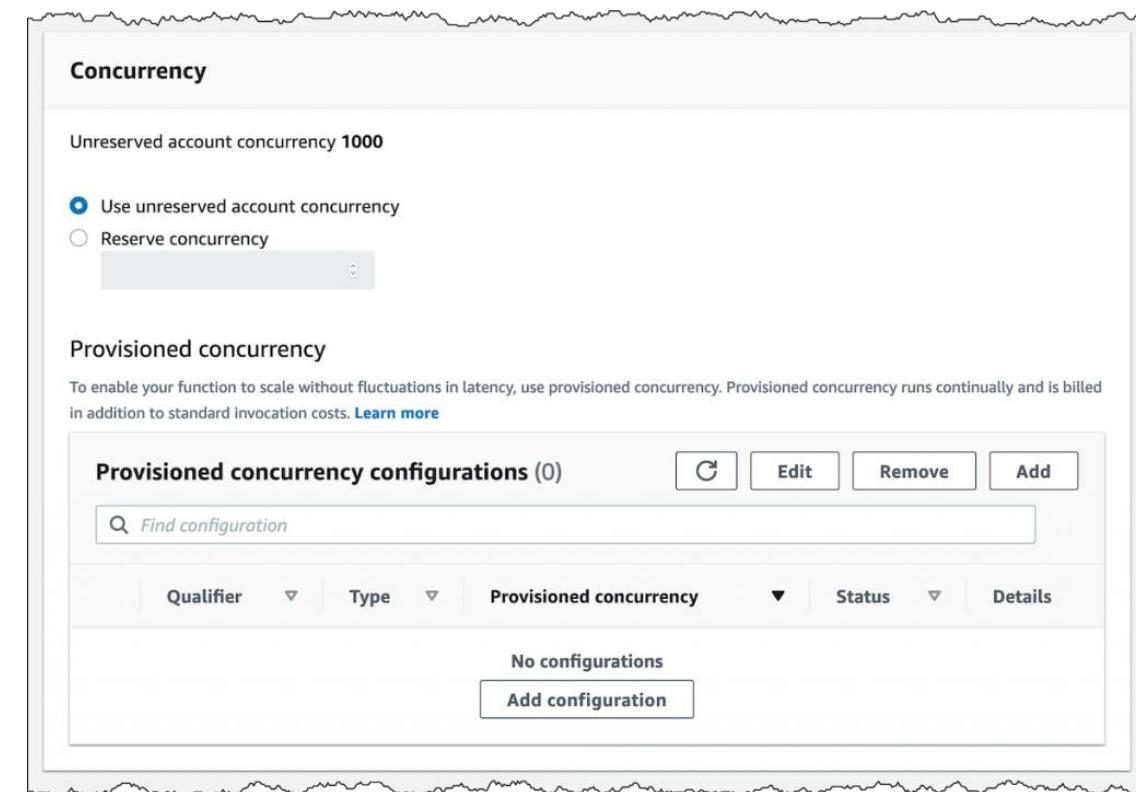
- AWS Lambda has several option to configure. The **runtime** and **type of deployment package** configurations can't change after function created.
- After function created, **Configure Settings** like permissions, environment variables, tags, and layers.
- **General Configuration:** Configure memory settings. Provide concurrency capacity of lambda function.
- **Permissions:** Configure the execution role and other permissions.
- **Environment variables:** Configure Key-value pairs for Lambda function in the execution environment.
- **Tags:** Use tags to organize Lambda functions into groups for cost reporting and filtering functions.

NOTE: Lambda execution summary details such as Execution time, Function version, Init duration, Duration, Billed duration, Resources configured, Max memory used, etc. can be found under cloud watch logs or test execution summary.



AWS Lambda Configurations – Part 2

- **Virtual private cloud (VPC):** Configure Virtual private cloud (VPC) if you need to set network access to AWS resources.
- **Concurrency:** Configure Reserve concurrency for a function to set the maximum number of simultaneous executions for a function.
- Provision concurrency to ensure that a function can scale without fluctuations in latency.
- **Function URL:** Configure a function URL to add a unique HTTP(S) endpoint to our Lambda function.
- Triggers, Destinations, Asynchronous invocation, Code signing, Database proxies, File systems, State machines



Lambda function Url: A simple HTTPS endpoint directly linked to a Lambda function without the complexity of API Gateway. But it Lacks advanced API management features (e.g., versioning, throttling, or request validation like cognito).

Concurrency on AWS Lambda: Reserved and Provisioned

- **Memory and Timeout Configuration Optimization**

The memory setting determines how much CPU power will receive for our lambda function. And higher CPU power decrease the function execution time.

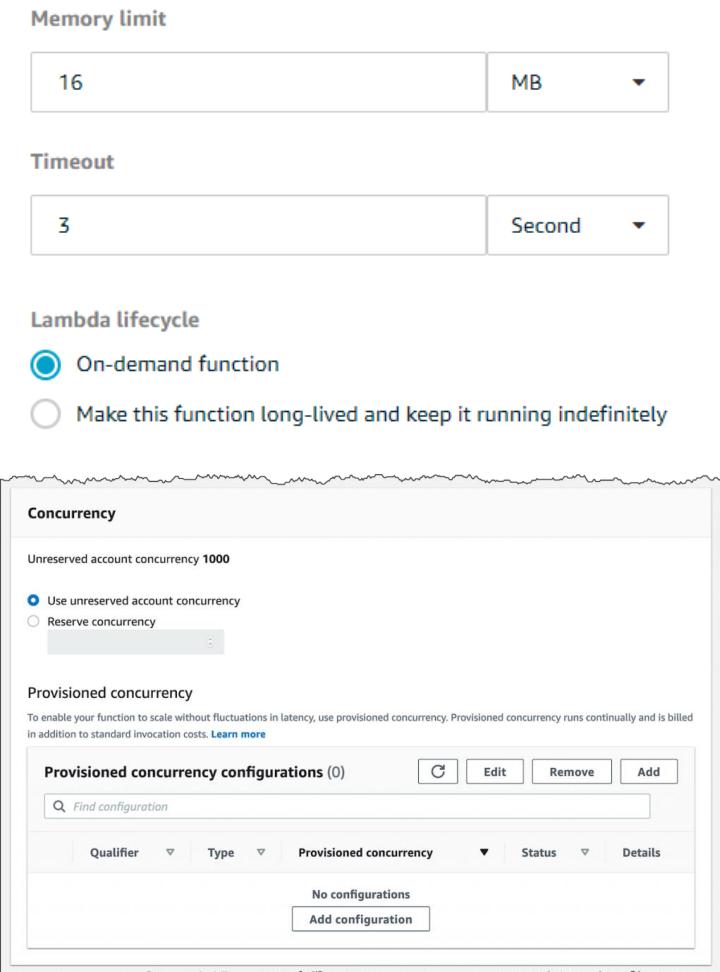
- When function **reaches to timeout value** and not finished, lambda forcibly stop the execution.

- **Concurrency:** The number of instances of your function that are active.

- Reserved Concurrency
 - Provisioned Concurrency

- When your function is invoked, Lambda **allocates** an **instance** of it to process the incoming event.

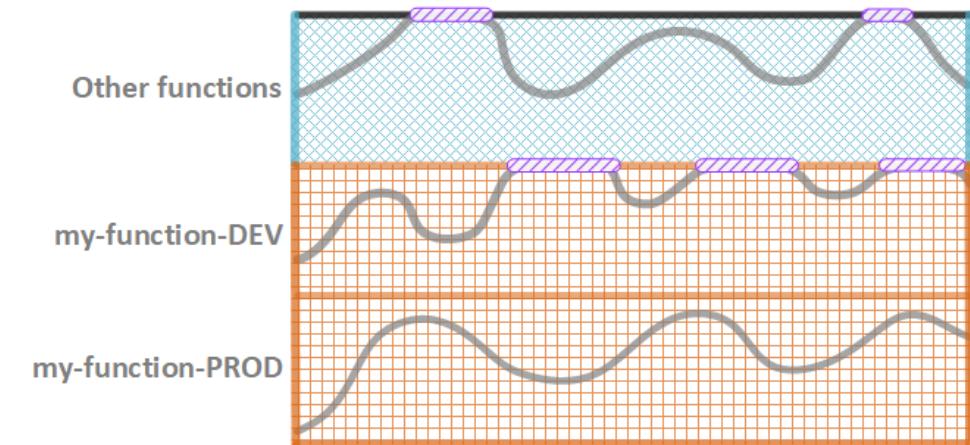
- If the function is invoked again when the request is still being processed, another instance is allocated, which **increases** the function's **concurrency**.



AWS Lambda Reserved Concurrency

- **Reserved concurrency**
Guarantees the maximum number of concurrent instances for the function.
- When a function has **reserved concurrency**, no other function can use that concurrency.
- We can configure **reserved concurrency** from AWS management console.
- **To throttle a function, we can set the reserved concurrency to zero.** This is a good way to suddenly stop your executions on any environment.
- Set reserved concurrency configuration with the AWS CLI;
 - `aws lambda put-function-concurrency --function-name my-function --reserved-concurrent-executions 100`
 - `"ReservedConcurrentExecutions": 100`

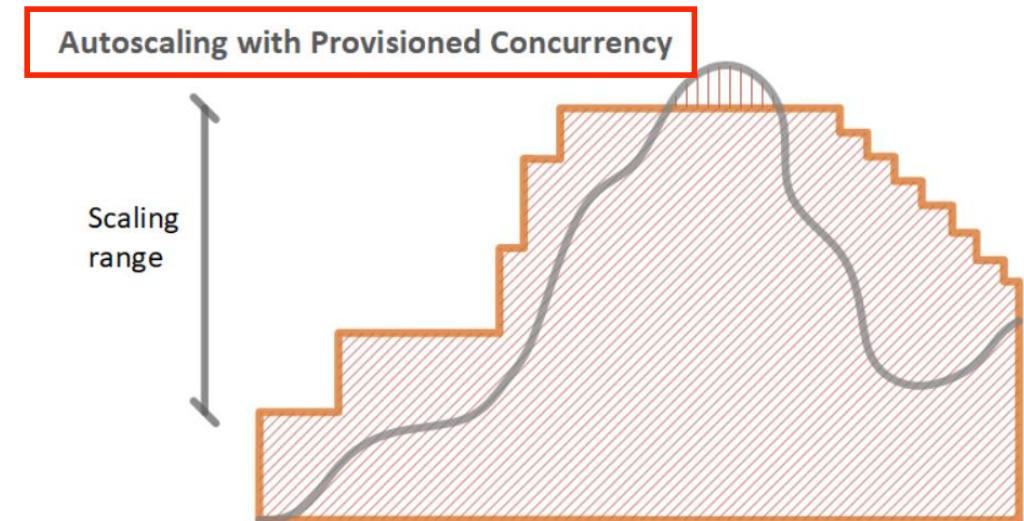
Reserved Concurrency



<https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html>

AWS Lambda Provisioned Concurrency

- **Provisioned concurrency**
Initializes a requested number of execution environments so that they are prepared to respond immediately to your function's invocations.
- When a function has **provisioned concurrency**, the runtime loads your function's code and runs initialization code.
- **Cold Start**
If your code and dependencies are large, or you create SDK clients during initialization, this process can take some time. Takes some time to spin up lambda function again and it makes new instances to have higher latency.
- **Allocate provisioned concurrency** before an increase in invocations, we can provide that requests are executed by initialized instances with low latency.



<https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html>

AWS Lambda Permissions; Execution Role and Resource-based Policies

- **AWS Identity and Access Management (IAM)**

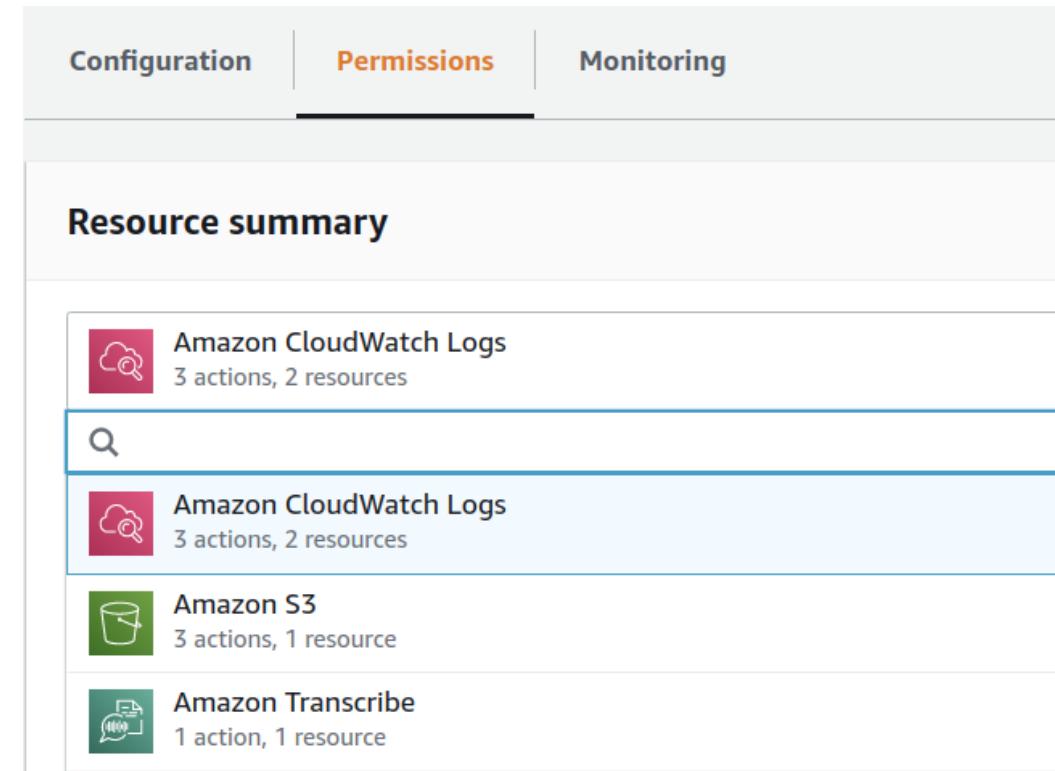
Handle permissions and manage access to the Lambda functions. Manage permissions in a permissions policy that you can apply to IAM users, groups, or roles.

- **Lambda Execution Role**

Grants permission to access AWS services and resources. By default Lambda function needs access to Amazon CloudWatch Logs for log streaming.

- If our lambda function access to DynamoDB to perform crud operations, we should give required permission in the execution role of the lambda function.

- **Resource-based policy**, to give other AWS services permission to use our Lambda function. AWS service like Amazon S3 calls our Lambda function, the resource-based policy gives it access.



AWS Lambda Execution Role

- **AWS Lambda Permissions**

- Lambda Execution Role
- Resource-based policy

- **Lambda Execution Role**

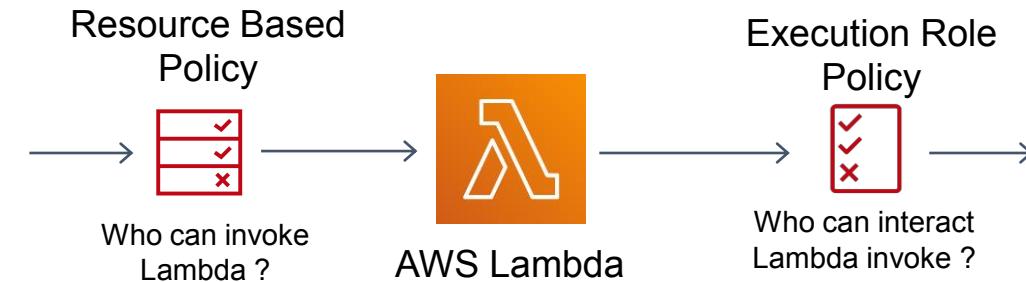
IAM role that Lambda has permissions to assume when invoking lambda function. Create an execution role when creating a new lambda function, and after that we can also modify the policies associated with the IAM role.

- If you have additional targets from your lambda function,

- performing crud operations on DynamoDB table
- sending notification to SNS,
- retrieve messages from queue or streams

- Lambda function's execution role required permissions to interact with those AWS services

- Grant least privilege access to your Lambda execution role



AWS managed policies for Lambda

The following AWS managed policies provide permissions t

Change

[AWSLambdaBasicExecutionRole](#) – Lambda started tracking changes to this policy.

[AWSLambdaDynamoDBExecutionRole](#) – Lambda started tracking changes to this policy.

[AWSLambdaKinesisExecutionRole](#) – Lambda started tracking changes to this policy.

[AWSLambdaMSKExecutionRole](#) – Lambda started tracking changes to this policy.

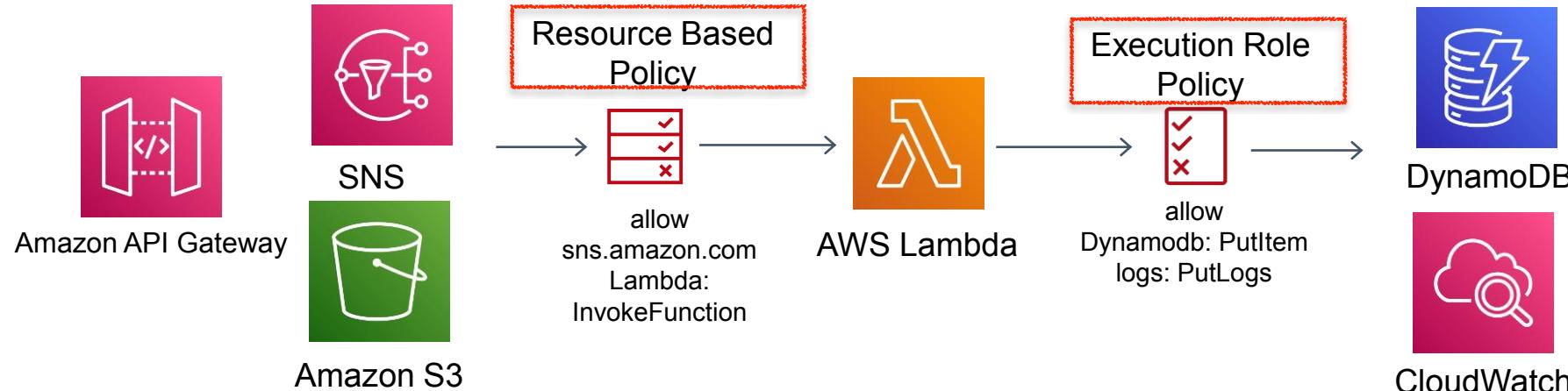
[AWSLambdaSQSQueueExecutionRole](#) – Lambda started tracking changes to this policy.

AWS Lambda Resource-based policy

- **Lambda Resource-based policy**

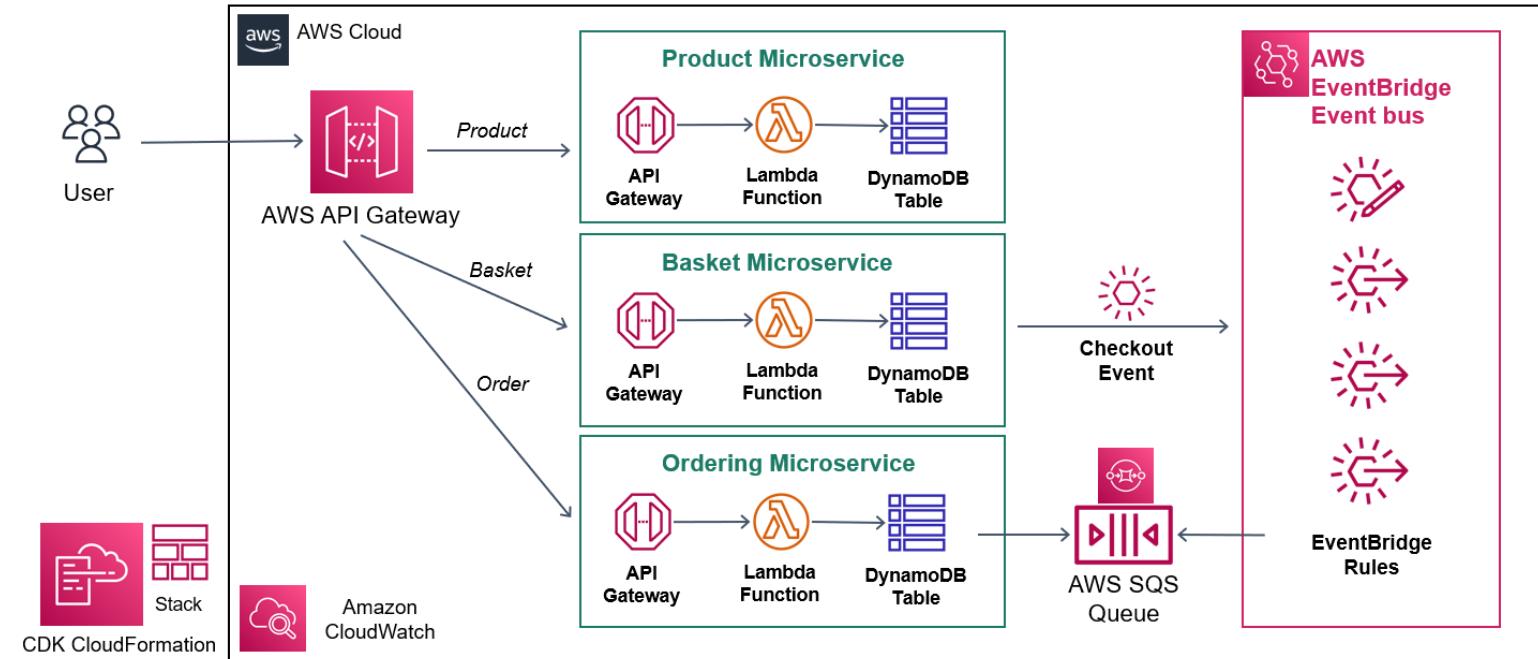
When any AWS service invokes Lambda function sync or async way. It lets you grant usage permission to other AWS accounts or organizations on a per-resource basis. Also use a resource-based policy to allow an AWS service to invoke your function on your behalf.

- API Gateway that targets to Lambda function, we should add resource-based policy permission to invoke lambda function from API gateway.
- Amazon S3 upload event triggers to lambda function asynchronously, so we should also add Resource-based policy into our Lambda function grants S3 invocation.



Prerequisites and Tools

- **5 main Prerequisites;**
- AWS Account and User
- AWS CLI
- Node.js
- IDE for your programming language
= Visual Studio Code
- Postman



AWS Lambda - Developing with AWS CLI

→ AWS Lambda - Developing with AWS CLI - Programmatic Access w/
Serverless APIs

Invoke AWS APIs with Different Ways

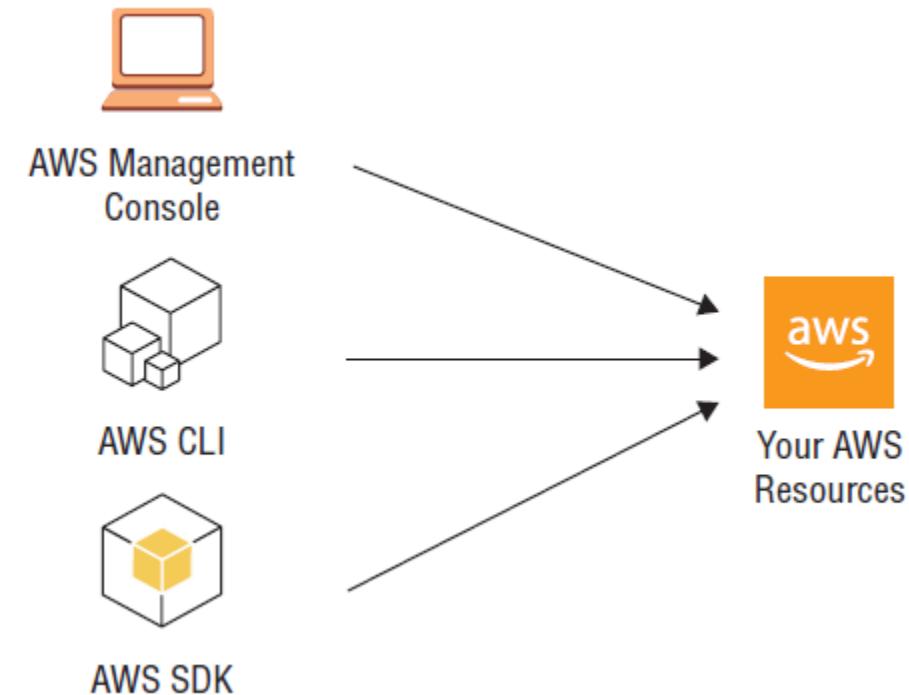
- **AWS Management Console Access**

You can think as a web application allows us to manage AWS resources for particular AWS accounts.

- **Programmatic Access**

Gives us to manage AWS resources from our development environments and manage by writing codes.

- AWS CLI
- AWS SDK
- AWS Cloud Formation - IaC
 - AWS SAM
 - AWS CDK



Programmatic Access

- **Programmatic Access**

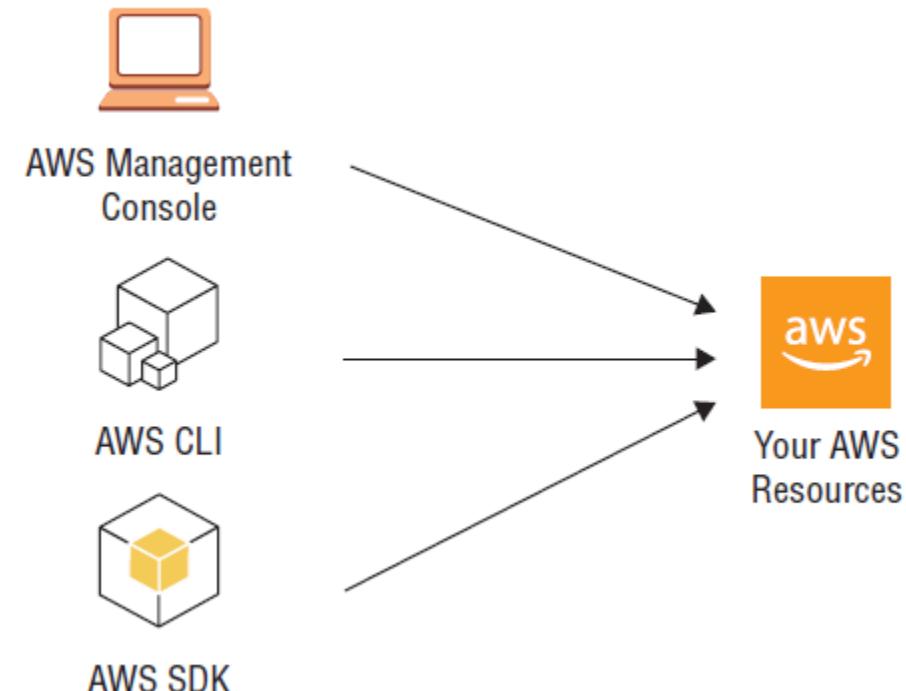
Gives us to manage AWS resources from our development environments and manage by writing codes.

- AWS CLI
- AWS SDK
- AWS Cloud Formation - IaC
 - AWS SAM
 - AWS CDK

- **AWS Command Line Interface (CLI)**

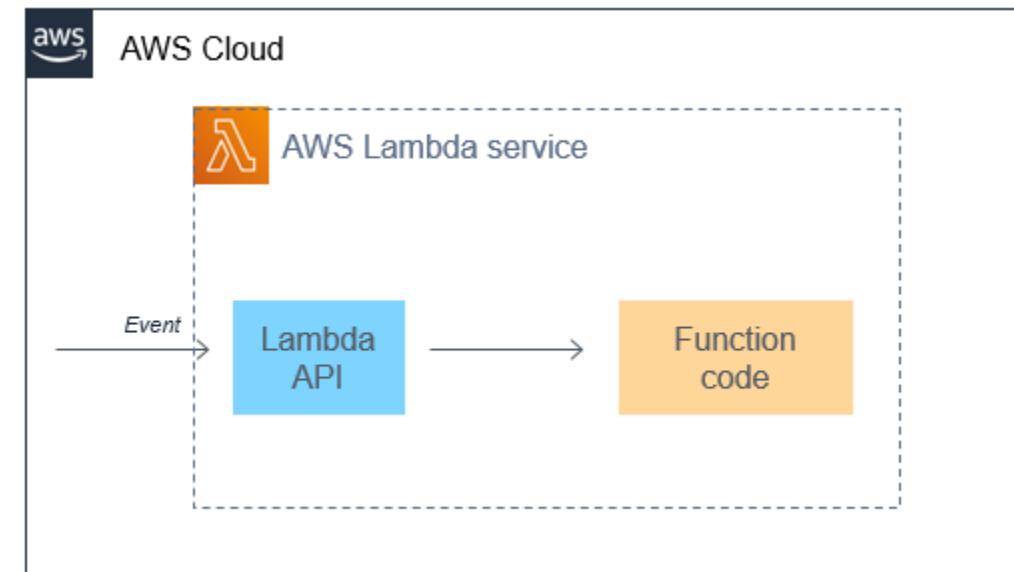
Unified tool to manage your AWS services.

- Control multiple AWS services from the command line and automate them through scripts.
- `$ aws ec2 describe-instances`
- `$ aws ec2 start-instances --instance-ids i-1348636c`



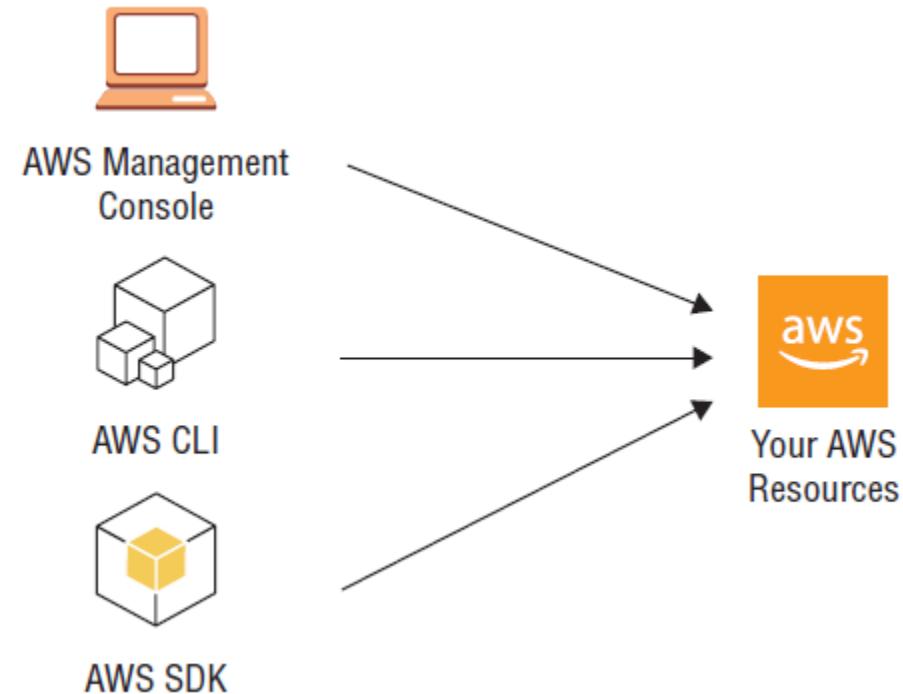
AWS CLI Lambda Interactions

- Listing lambda functions
- Create lambda functions
- Develop lambda functions
- Zip and deploy function
- Invoke and test function
- Update function
- Delete function



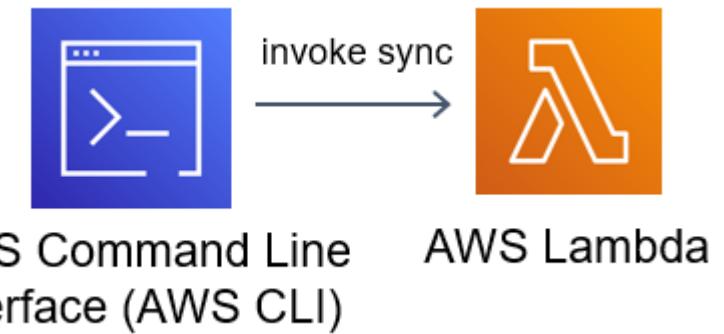
Using AWS Lambda with the AWS CLI

- We have 2 options to use AWS CLI
 - AWS CloudShell
 - Command Window
- **AWS CloudShell**
AWS CloudShell which establish on aws management console.
- **Command Window**
Windows PowerShell Command Windows or any local command windows for your operating system



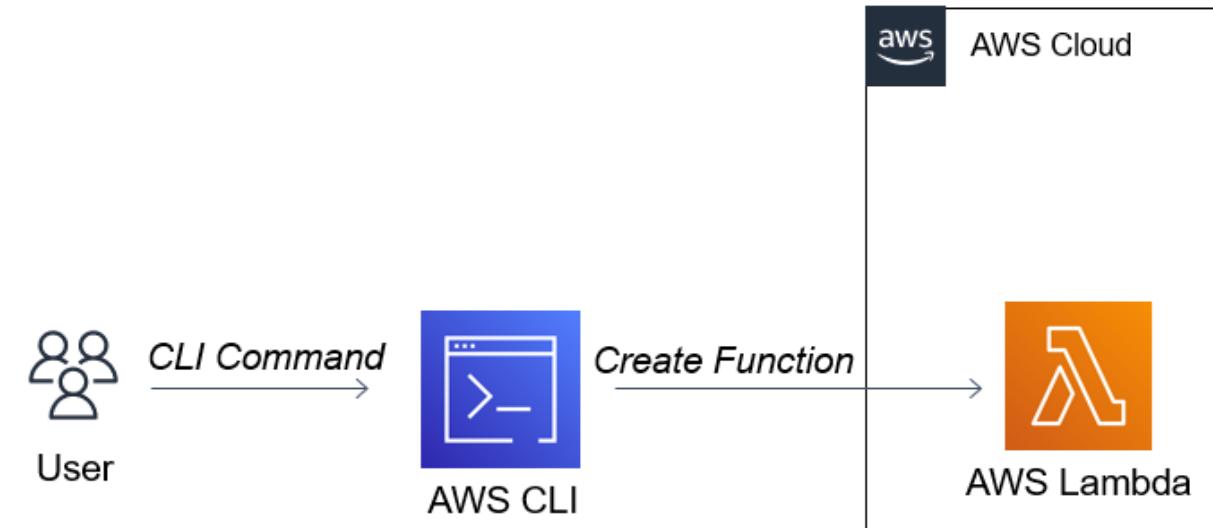
Invoke Lambda functions with the AWS CLI

- We can invoke Lambda functions directly using the;
 - Lambda console
 - Lambda API
 - AWS SDK
 - AWS Command Line Interface (AWS CLI)
- **Understand the Lambda Invoke Types**
When you invoke a function, we can choose to invoke it **synchronously or asynchronously**.
- **Synchronous invocation**, you wait for the function to process the event and return a response.
- **Asynchronous invocation**, Lambda queues the event for processing and returns a response immediately.
- First part of course, I will mainly invoke with synchronously way to lambda function.



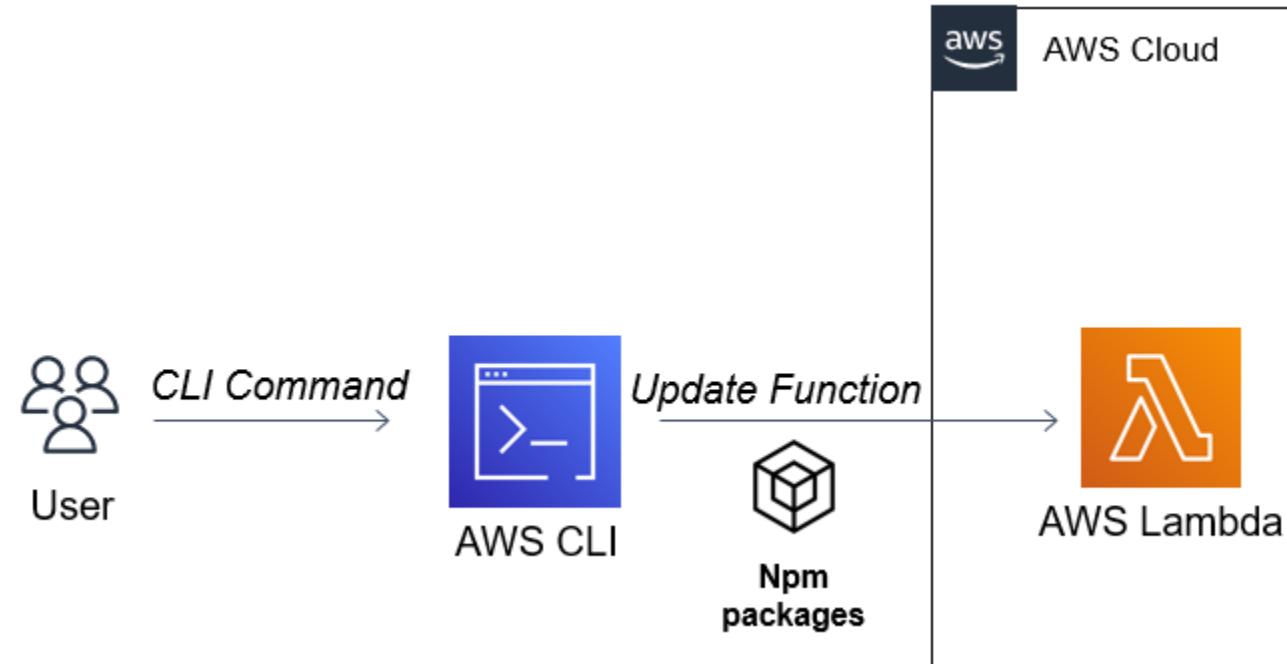
Create AWS Lambda functions with AWS CLI

- Steps of Create AWS Lambda functions with AWS CLI;
 - Create the execution role
 - Create function code
 - Create a deployment package with zip function code
 - Create Lambda Function with AWS CLI
 - Invoke Lambda Function
 - Check logs
- We will Check Lambda Function details with AWS CLI.
- We will Clean up resources.



Update AWS Lambda Function with Dependencies using AWS CLI

- Lambda function depends on libraries, we can use **npm** to include them in our deployment package.
- Add any external package with using "**npm install**" command
- Steps of Update AWS Lambda functions with AWS CLI;
 - 1- Install required Dependencies into node_modules
 - 2- zip function code
 - 3- update lambda function with cli
 - 4- invoke updated lambda function with cli
- We will Clean up resources.



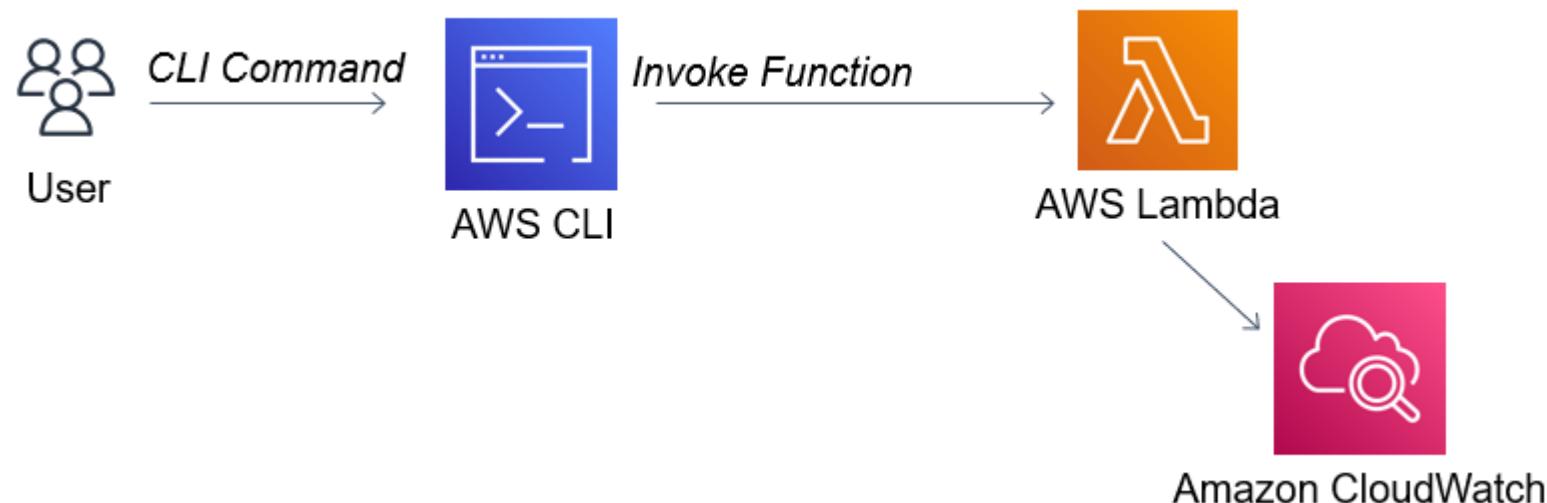
AWS Lambda Context Object in Node.js Function

- When Lambda runs our function, it passes a context object to the handler.
- Context object provides methods and properties that provide information about the invocation, function, and execution environment.
- Context methods
 - `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.
- Context properties
 - `functionName` – The name of the Lambda function.
 - `functionVersion` – The version of the function.

```
exports.handler = async function(event, context) {  
    console.log('Remaining time: ', context.getRemainingTimeInMillis())  
    console.log('Function name: ', context.functionName)  
    return context.logStreamName  
}
```

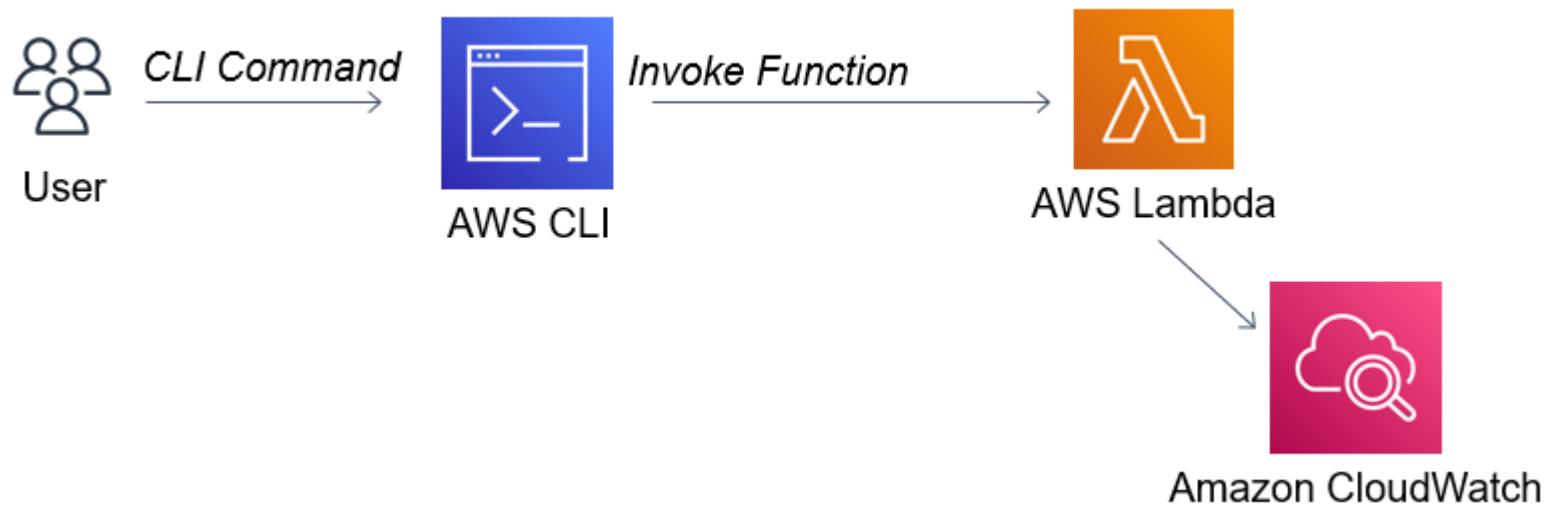
AWS Lambda Function Logging in Node.js Example

- AWS Lambda automatically monitors Lambda functions on your behalf and sends function metrics to Amazon CloudWatch.
- AWS Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function.
- The Lambda runtime environment sends details about each invocation to the log stream.



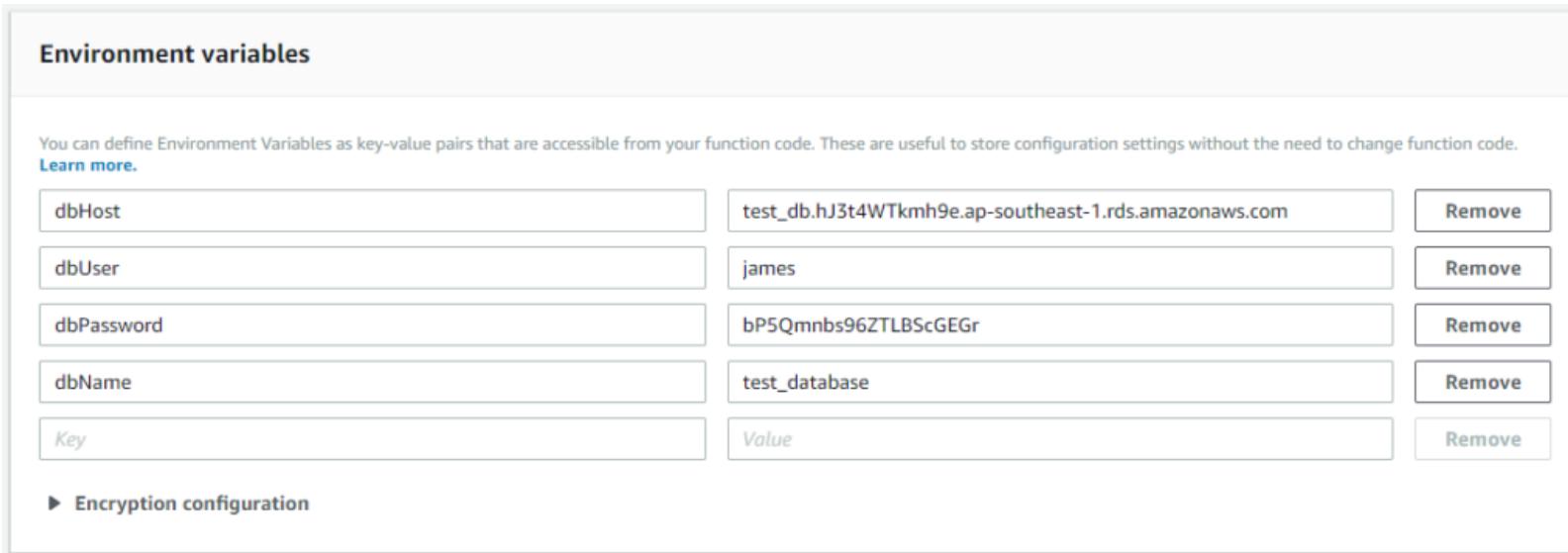
AWS Lambda Function Errors in Node.js Example

- When Lambda function code raises an error, Lambda generates a JSON structure of error.
- Error appears in invocation logs and see in output to Amazon CloudWatch.



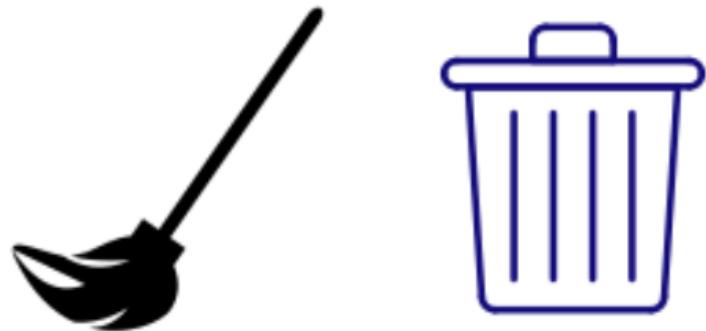
Using AWS Lambda Environment Variables

- Use environment variables to adjust your function's behavior without updating code.
- Environment variable is a pair of strings that is stored in a function's version-specific configuration.
- Configure environment variables with using AWS Management Console.
- Configure environment variables with the Lambda API with cli and sdk libraries.



Clean up Resources

- Delete AWS Resources that we create during the section.

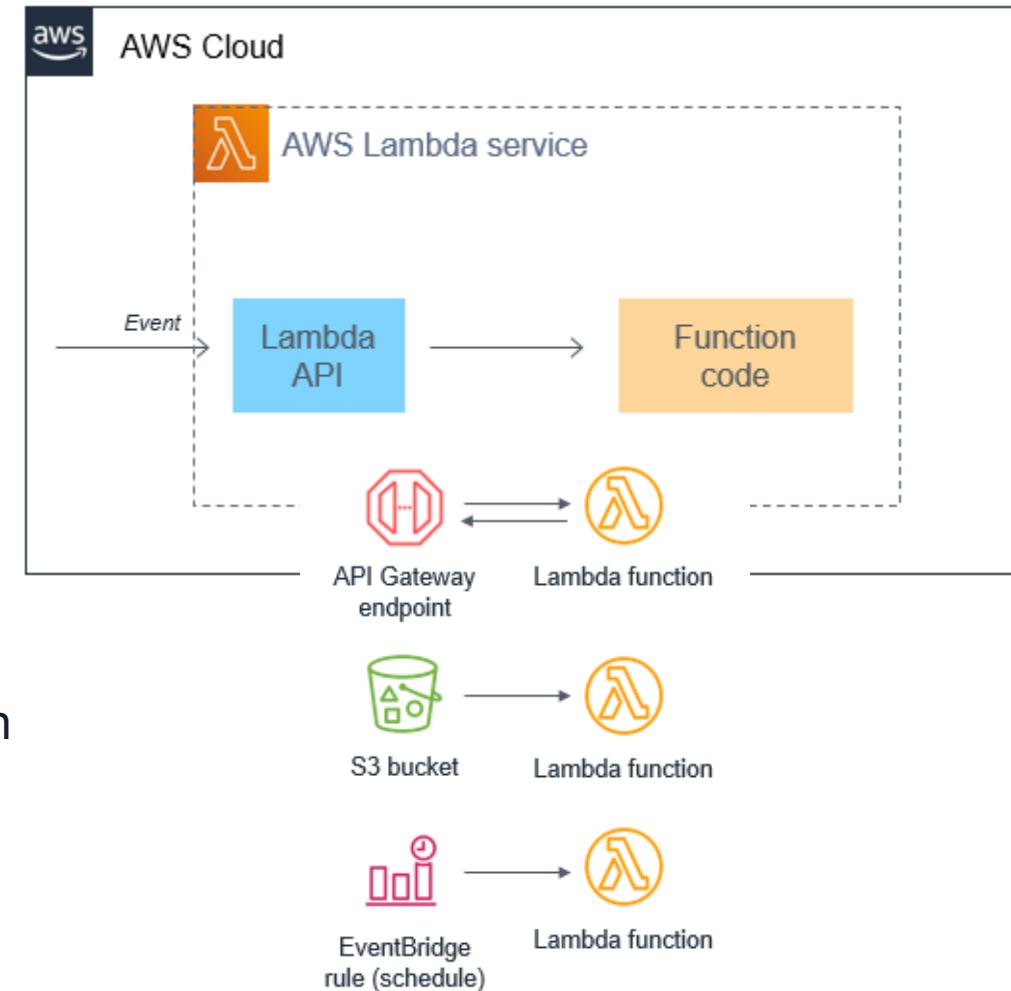


AWS Lambda - Working with Events

→ AWS Lambda Best Practices and Event-driven Architectures

AWS Lambda Best Practices and Event-driven Architecture

- AWS Lambda **design principles** and the **best practices** when developing our Lambda-based Event-driven Serverless e-commerce applications.
- Lambda is very **good fit** with **Event-driven Architectures**.
- AWS services generate events for **communicating** each other, most of AWS services are **event sources** for Lambda.
- Lambda always **handle all interactions** with the **Lambda API** and there is no direct invocation of functions from outside the service.
- The main purpose of lambda functions is to **handle events**. Even the simplest Lambda-based application uses at least **one event**.
- Lambda functions are limited to **15 minutes** in duration.
- An event triggering a Lambda function could be **almost anything**.



AWS Lambda Events

- The event is a **JSON object** that contains all information about what happened.
 - Represents a change in the system state.
- The first parameter of every **Lambda handler function** contains the event json object.
 - With using this event json object, we can access the event parameters into lambda function.
- An event could be custom-generated from another microservice,
 - New order generated in an ecommerce application.
- The event also can be generated from **existing AWS service**
 - Amazon SQS when a new queue message is available in a queue
- **Event-driven architectures** rely on creating events into all application state changes that are observable by other services
 - Loosely coupled services.

The screenshot shows the AWS Lambda function configuration and the code editor for a Lambda function named "index.js".

Event name: NewOrderEvent

Event JSON:

```
1  [ {  
2      "source": "myApplication",  
3      "detail": "submitOrder",  
4      "customerId": "customer123",  
5      "orderId": "order-A1234B56",  
6      "paymentStatus": "open",  
7      "cart": [  
8          {  
9              "product12": {  
10                 "qty": 2,  
11                 "itemPx": 35.22,  
12                 "currency": "USD"  
13             },  
14             "product44": {  
15                 "qty": 5,  
16                 "itemPx": 71.57,  
17                 "currency": "USD"  
18             }  
19         ],  
20         "timestamp": 1607774286  
21     ]  
22 } ]
```

index.js:

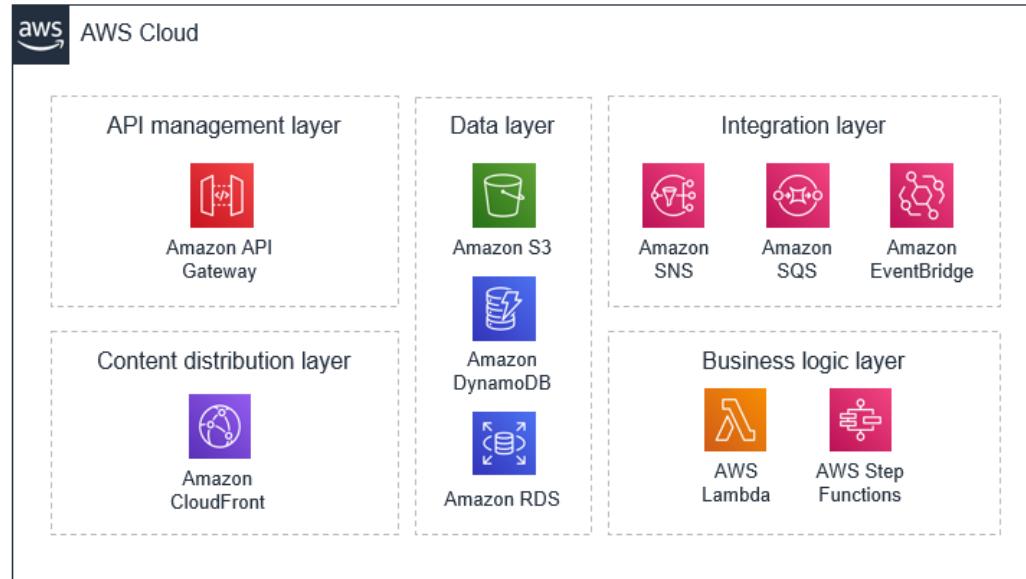
```
i 1 const AWS = require('aws-sdk')  
i 2 AWS.config.region = process.env.AWS_REGION  
i 3  
i 4 const s3 = new AWS.S3()  
i 5  
i 6 exports.handler = async (event) => {  
i 7  
i 8     console.log(JSON.stringify(event,0, null))  
i 9  
i 10    const Bucket = event.Records[0].s3.bucket.name  
i 11    const Key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, ' '))  
i 12  
i 13    const result = await s3.getObject({  
i 14        Bucket,  
i 15        Key  
i 16    }).promise()  
i 17  
i 18    const data = result.Body  
i 19    console.log('PDF text length: ', data.length)  
i 20  
i 21 }
```

Annotations:

- 1**: Points to the `event.source` and `event.detail` fields in the event JSON.
- 2**: Points to the `event.Records` array in the Lambda function code.

AWS Lambda Best Practices and Event-driven Architecture

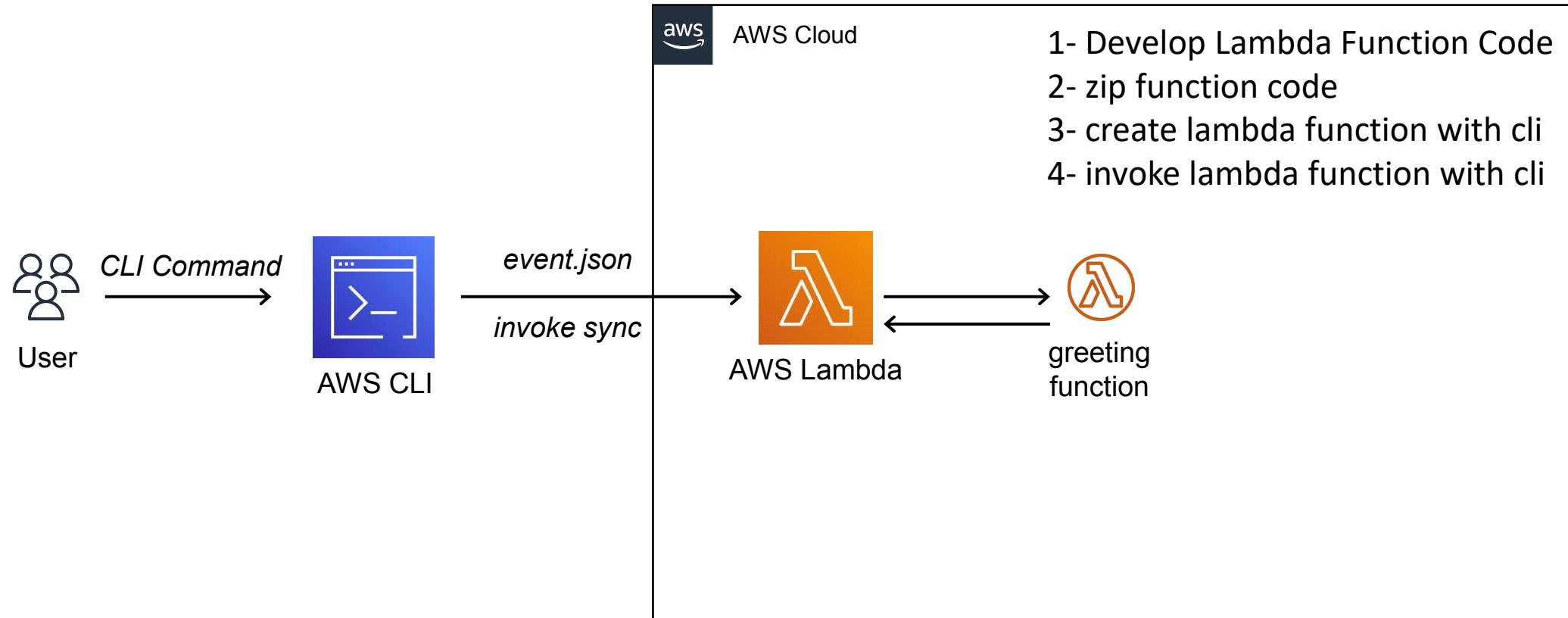
- Most Lambda-based applications use a **combination of AWS services** for different requirements about **Storage, API Management** and **integrating** with other system and services.
- Lambda is **connecting between services**, providing business logic to transform data that moves between services.
- You can find mostly **integrated AWS Services** which using Lambda functions.
- **Design patterns in Distributed architectures** with AWS Lambda
- When your application needs one of these patterns, we can use the corresponding AWS service.
- These services and patterns are **designed to integrate** with AWS Lambda functions



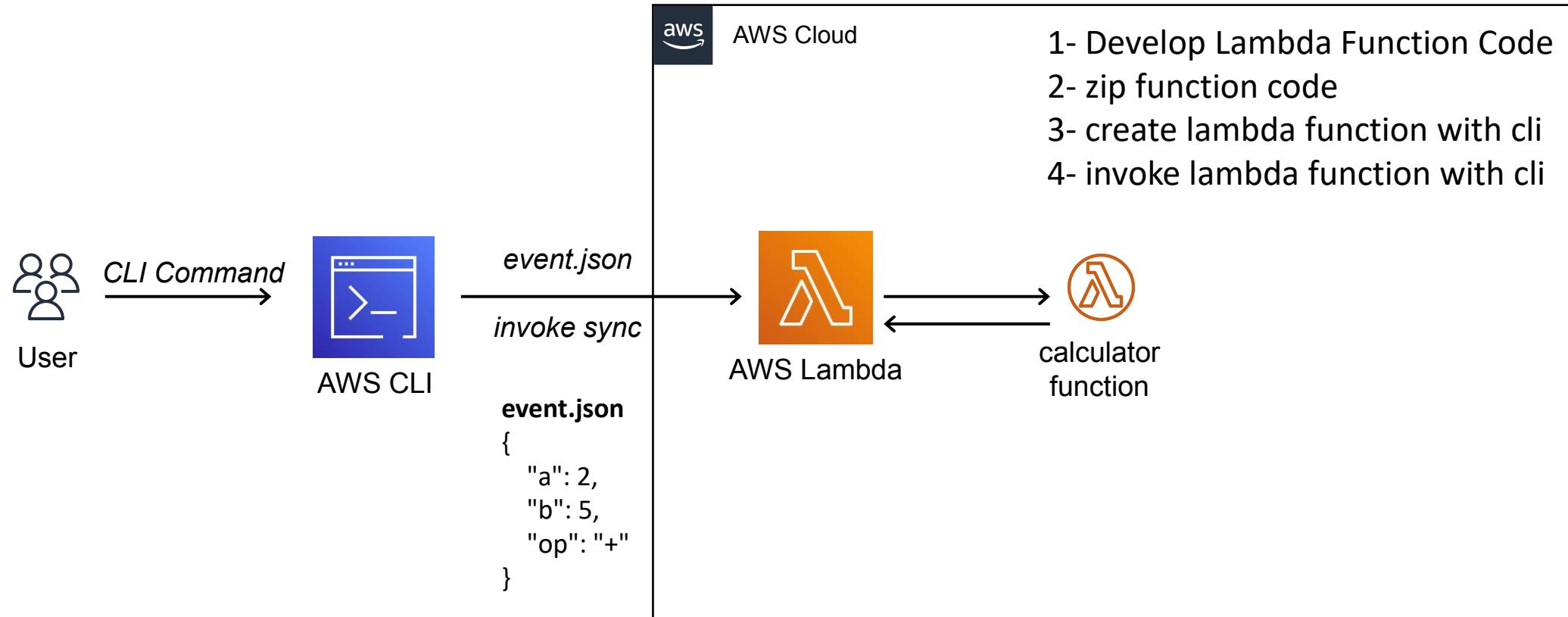
Pattern	AWS service
Queue	Amazon SQS
Event bus	Amazon EventBridge
Publish/subscribe (fan-out)	Amazon SNS
Orchestration	AWS Step Functions
API	Amazon API Gateway
Event streams	Amazon Kinesis

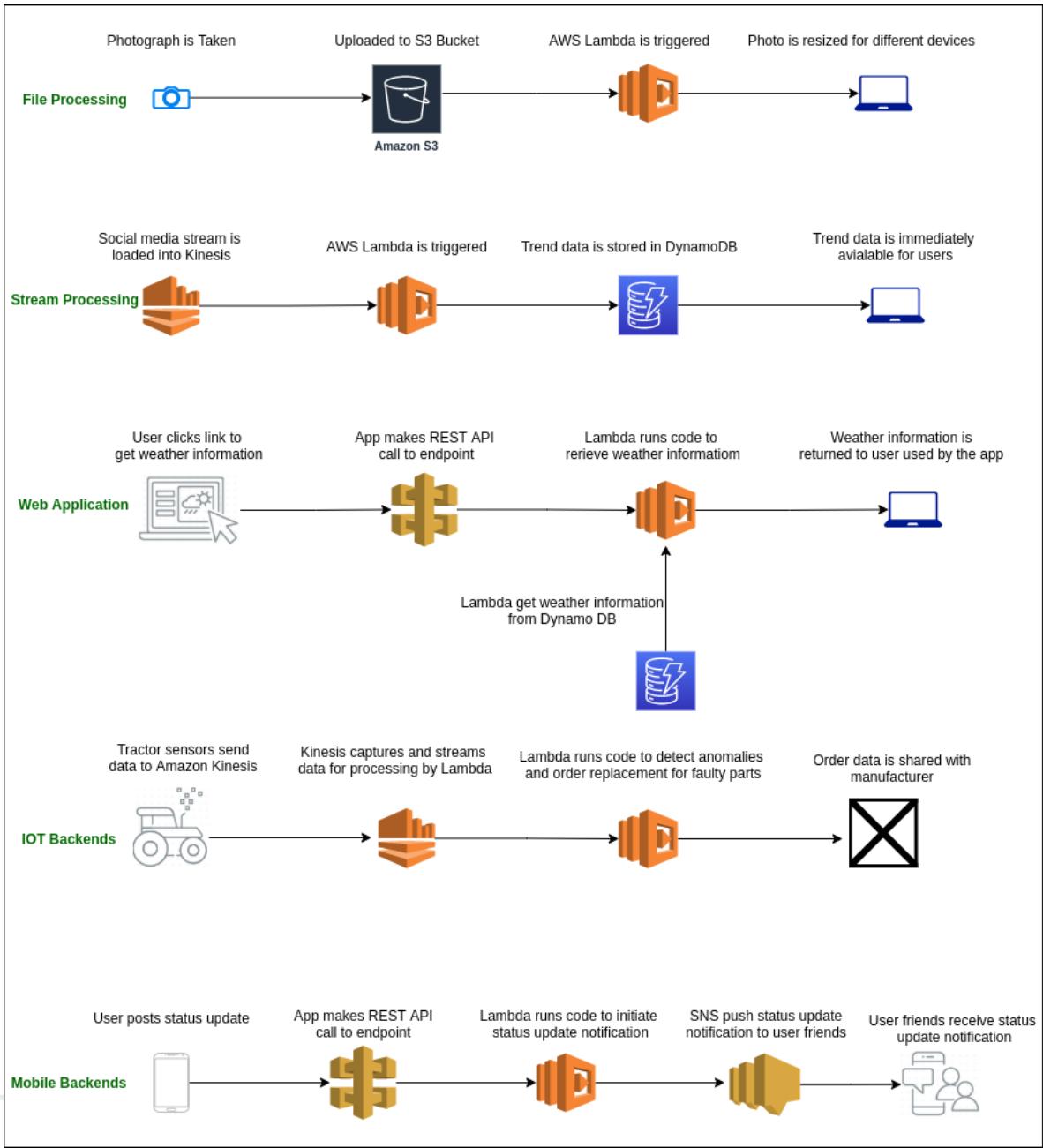
<https://aws.amazon.com/blogs/compute/operating-lambda-design-principles-in-event-driven-architecture/>

Hands-on Lab: Greeting Project



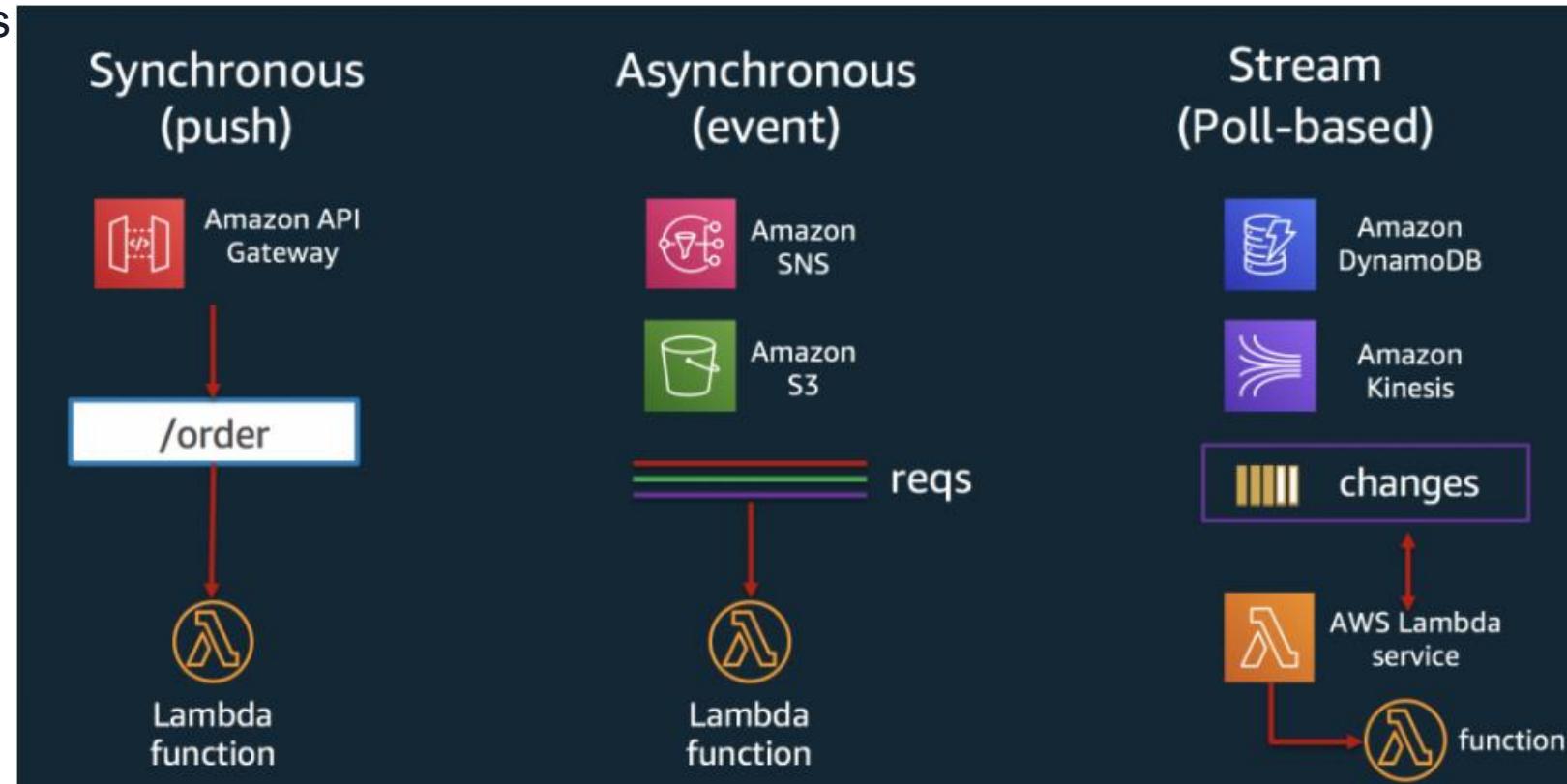
Hands-on Lab: Calculator Project





AWS Lambda Invocation Types

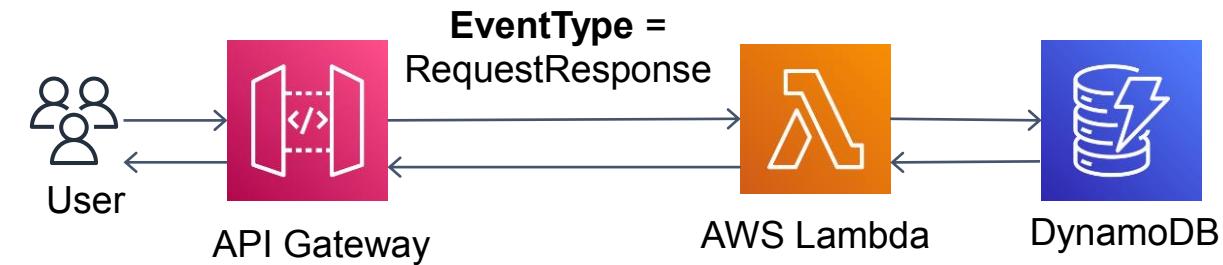
- Triggered lambda functions with different AWS Lambda Invocation Types
- AWS Lambda has 3 Invocation Types
- **Lambda Synchronous invocation**
- **Lambda Asynchronous invocation**
- **Lambda Event Source Mapping with polling invocation**



<https://aws.amazon.com/blogs/architecture/understanding-the-different-ways-to-invoke-lambda-functions/>

AWS Lambda Synchronous Invocation

- Execute immediately when you perform the Lambda Invoke API call.
- Wait for the function to process the function and return back to response.
- API Gateway + Lambda + DynamoDB
- Invocation-type flag should be “RequestResponse”
- Responsible for inspecting the response and determining if there was an error and decide to retry the invocation



- Example of synchronous invocation using the AWS CLI:
aws lambda invoke —function-name MyLambdaFunction —invocation-type RequestResponse —payload '{ "key": "value" }'
- Triggered AWS services of synchronous invocation; ELB (Application Load Balancer), Cognito, Lex, Alexa, API Gateway, CloudFront, Kinesis Data Firehose

AWS Lambda Asynchronous Invocation

- Lambda **sends the event** to a **internal queue** and returns a **success response** without any additional information
- Separate process **reads events** from the **queue** and **runs** our lambda function
- **S3 / SNS + Lambda + DynamoDB**
- **Invocation-type** flag should be “**Event**”
- AWS Lambda sets a **retry policy**

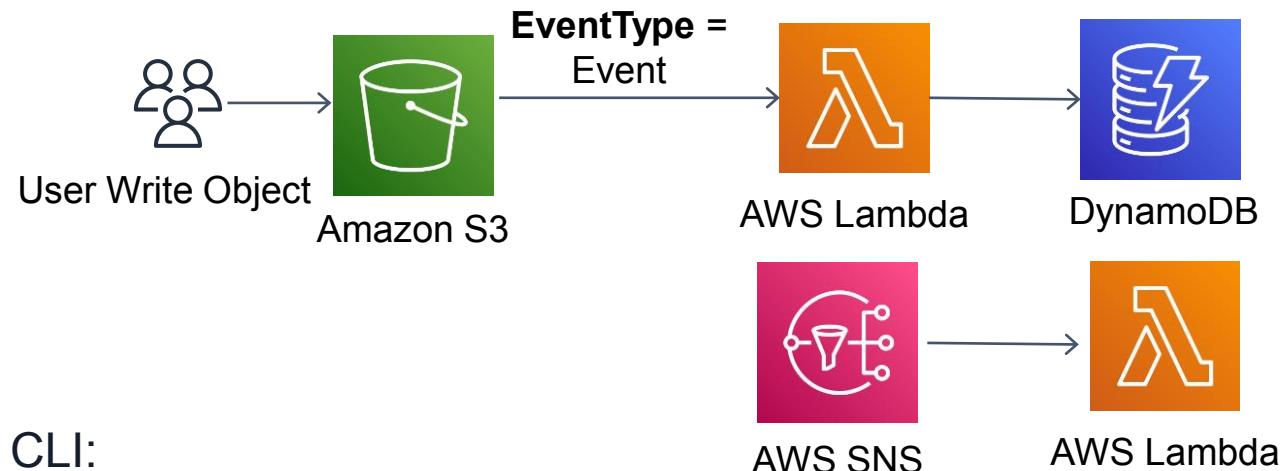
Retry Count = 2

Attach a Dead-Letter Queue (DLQ)

- Example of asynchronous invocation using the AWS CLI:

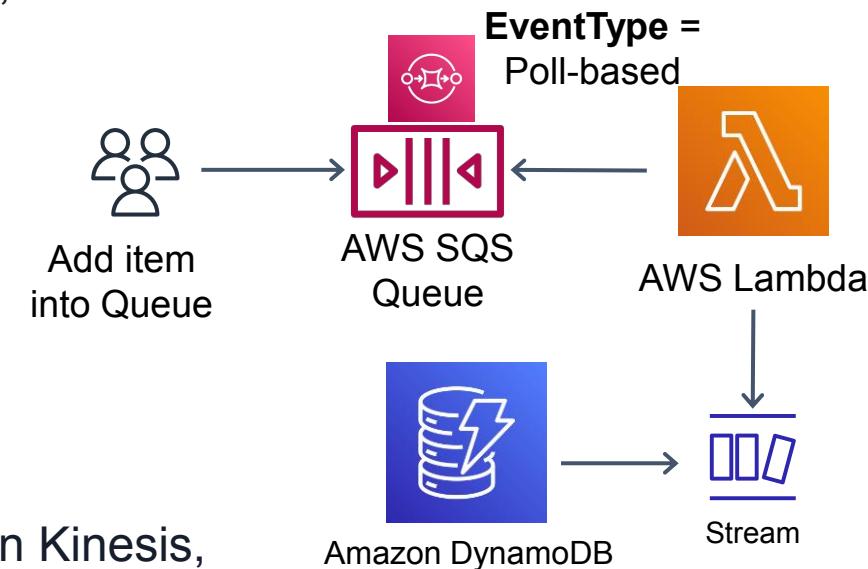
```
aws lambda invoke --function-name MyLambdaFunction --invocation-type Event --payload '{ "key": "value" }'
```

- Triggered **AWS services of asynchronous invocation**; S3, EventBridge, SNS, SES, CloudFormation, CloudWatch Logs, CloudWatch Events, CodeCommit



AWS Lambda Event Source Mapping with Polling Invocation

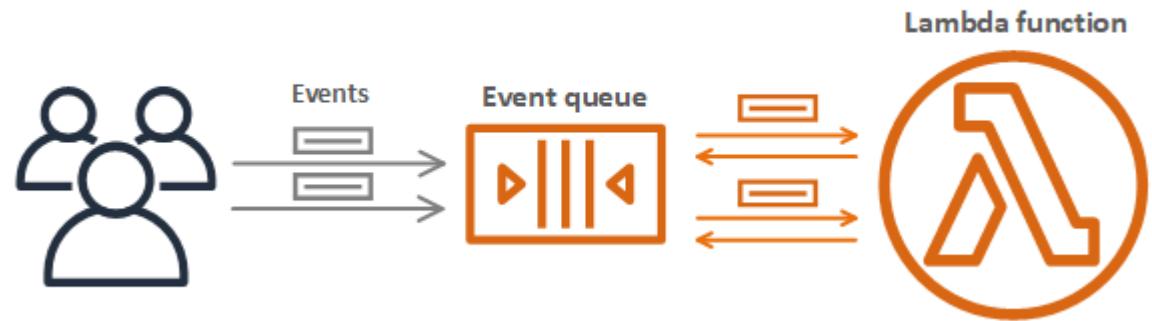
- **Pool-Based invocation** model allows us to **integrate** with **AWS Stream** and **Queue based services**.
- Lambda will **poll** from the AWS SQS or Kinesis streams, retrieve records, and invoke functions.
- Data stream or queue are **read in batches**,
- The function receives multiple items when execute function.
- **Batch sizes** can configure according to service types
- **SQS + Lambda**
- **Stream based processing** with **DynamoDB Streams + Lambda**
- Triggered **AWS services** of **Event Source Mapping invocation**; Amazon Kinesis, DynamoDB, Simple Queue Service (SQS)



AWS Lambda Asynchronous Invocation using AWS CLI

- AWS Lambda queues the events before sending them to the function.
- Lambda places the event in a queue and returns a success response without additional information.
- Separate process reads events from the queue and sends them to your function.
- ```
aws lambda invoke \
--function-name calculator \
--invocation-type Event \
--cli-binary-format raw-in-base64-out \
--payload '{ "key": "value" }' response.json
```
- RESPONSE:
  - {
  - "StatusCode": 202

Asynchronous Invocation



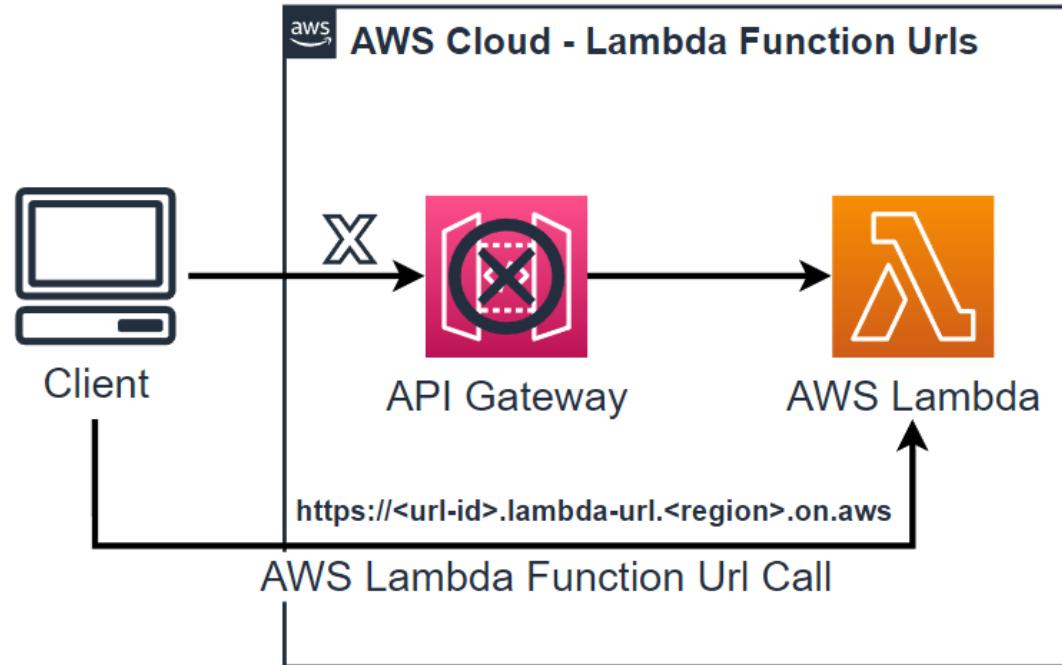
<https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html>

# AWS Lambda Function URLs: Built-in HTTPS Endpoints

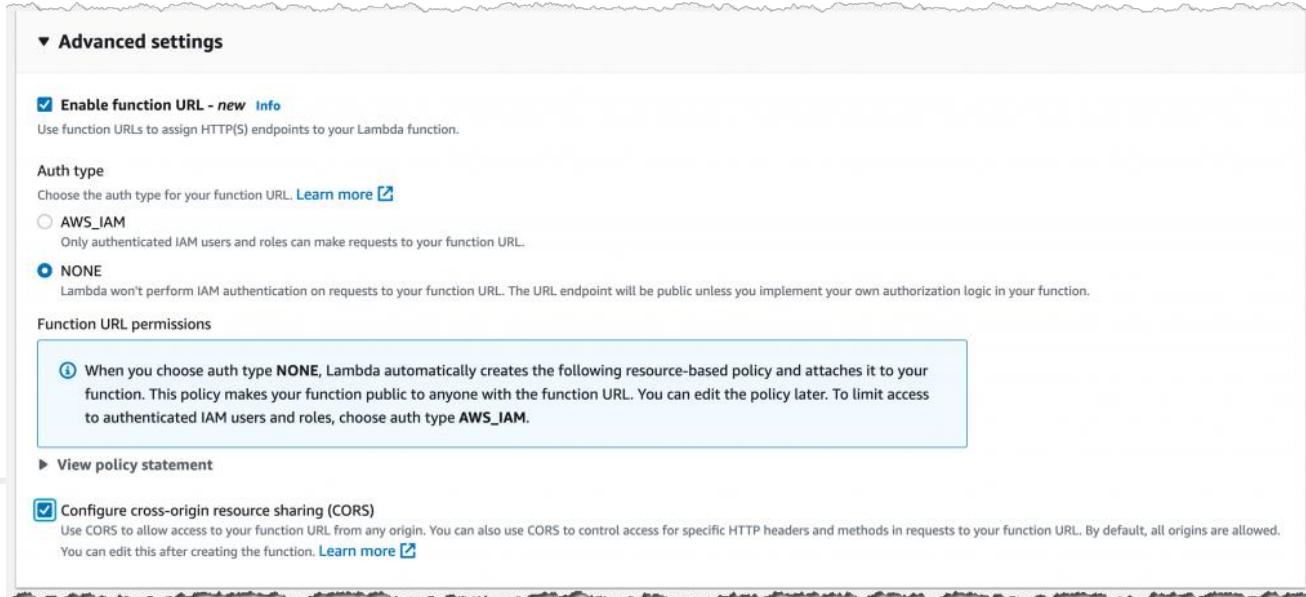
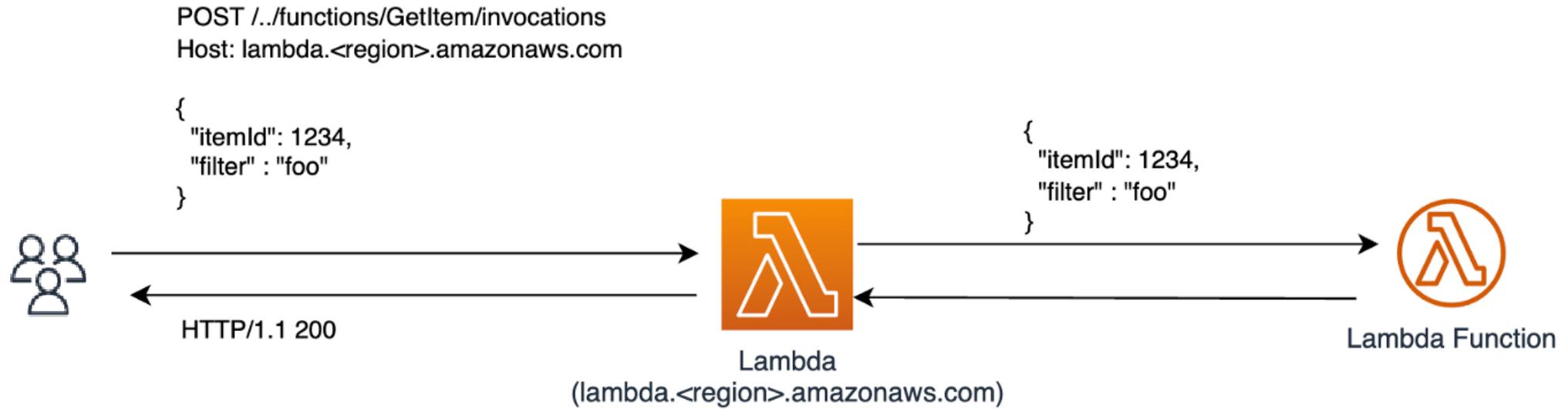
→ AWS Lambda Function URLs: Built-in HTTPS Endpoints for Single-Function Microservices

# AWS Lambda Function URLs: Built-in HTTPS Endpoints

- Organizations are **adopting microservices architectures** to build resilient and scalable applications using AWS Lambda.
- Each function is mapped to **API endpoints**, methods, and resources using services such as **Amazon API Gateway** and **Application Load Balancer**.
- Simple way to configure an **HTTPS endpoint** in front of your function
- AWS announce the general availability of **Lambda Function URLs**, This is a new feature that lets you add **HTTPS endpoints to any Lambda function** and optionally configure Cross-Origin Resource Sharing (**CORS**) headers.
- Lets you focus on business cases and AWS take care of configuring and monitoring a highly available, scalable, and secure HTTPS service.

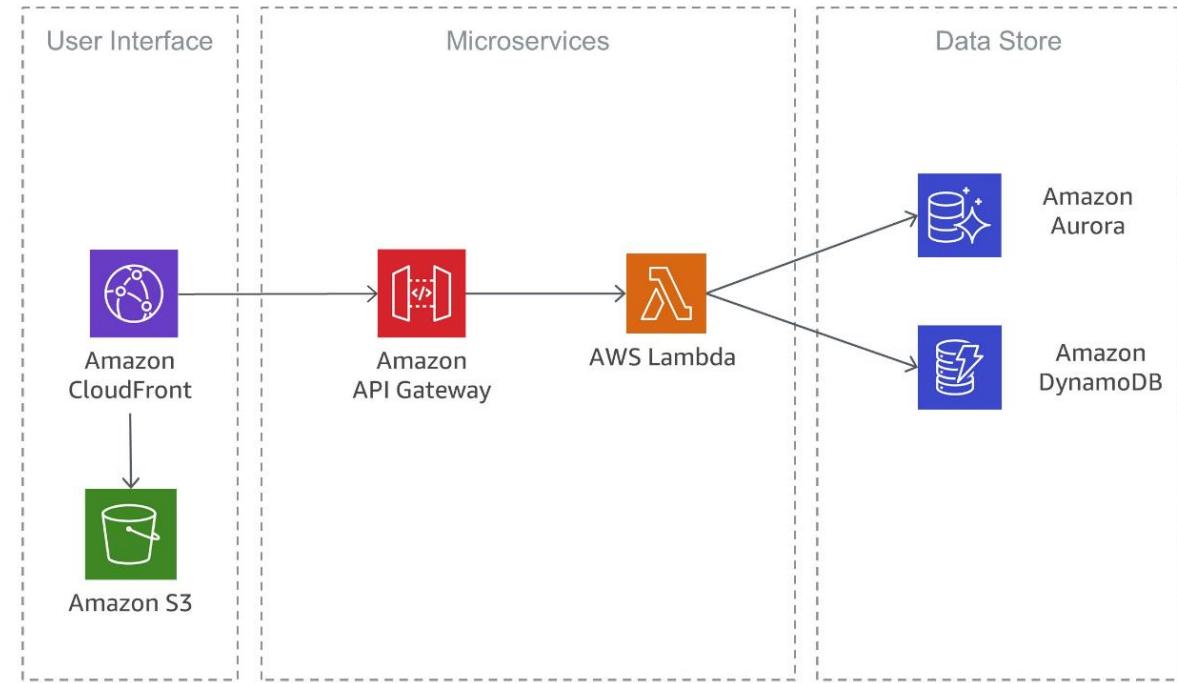


# How Lambda Function URLs Work ?



# AWS Lambda as a Microservice

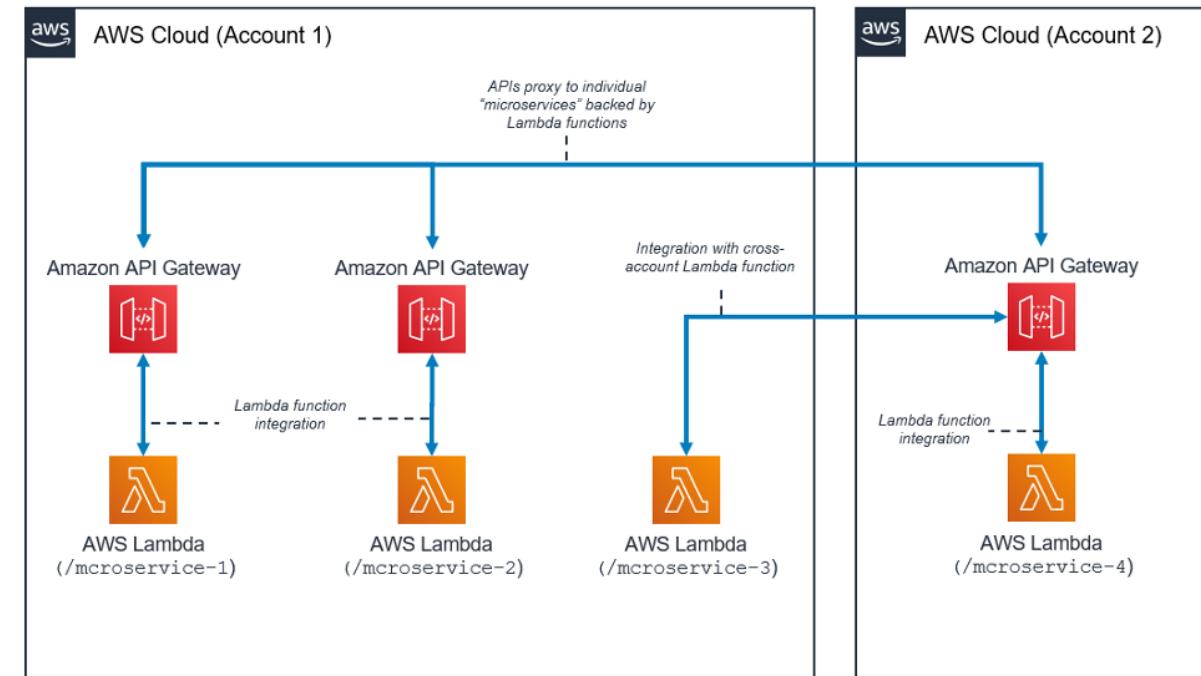
- Microservice are **small business services** that can work together and can be deployed **autonomously / independently**.
- **Lambda** is a service that allows you to run your functions in the cloud **completely serverless** and eliminates the operational **complexity**.
- It **integrates** with the **API gateway**, allows you to invoke functions with the API calls, and makes your architecture completely serverless.
- Microservice with **AWS Lambda**, which removing the architectural overhead of designing for **scaling** and high **availability**,
- Eliminating the **operational efforts** of operating and monitoring the microservice's underlying infrastructure.



<https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/serverless-microservices.html>

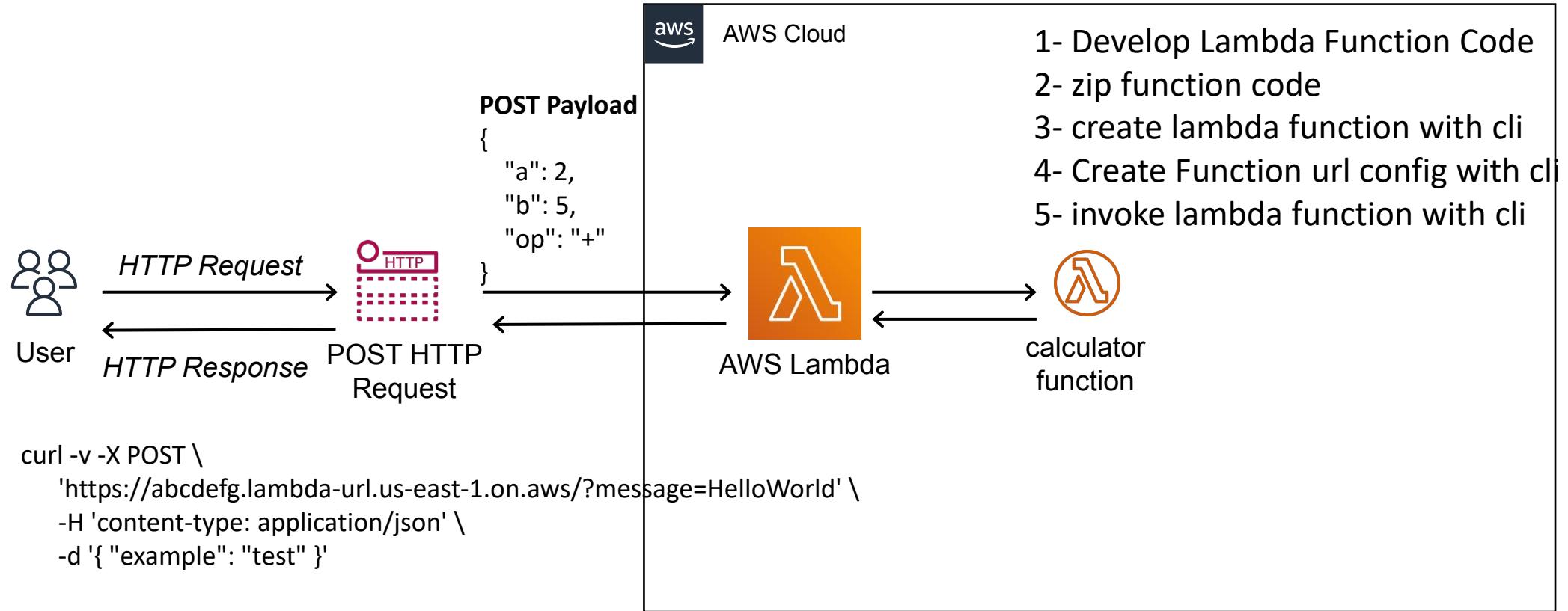
# Serverless Microservices with Lambda

- Each of the application components is **decoupled** and **deployed and run independently**.
- AWS **Lambda-initiated functions** is all you need to build a microservice.
- A microservices environment can introduce
  - **repeated overhead** for create each new microservice,
  - **problems optimizing** server usage,
  - **complexity** of running multiple versions of microservices,
  - client-side code **requirements to integrate** with many services.
- **Serverless microservices pattern** reduces the barrier for the creation of each subsequent microservice
- Optimizing **server utilization** is no longer relevant with this pattern.



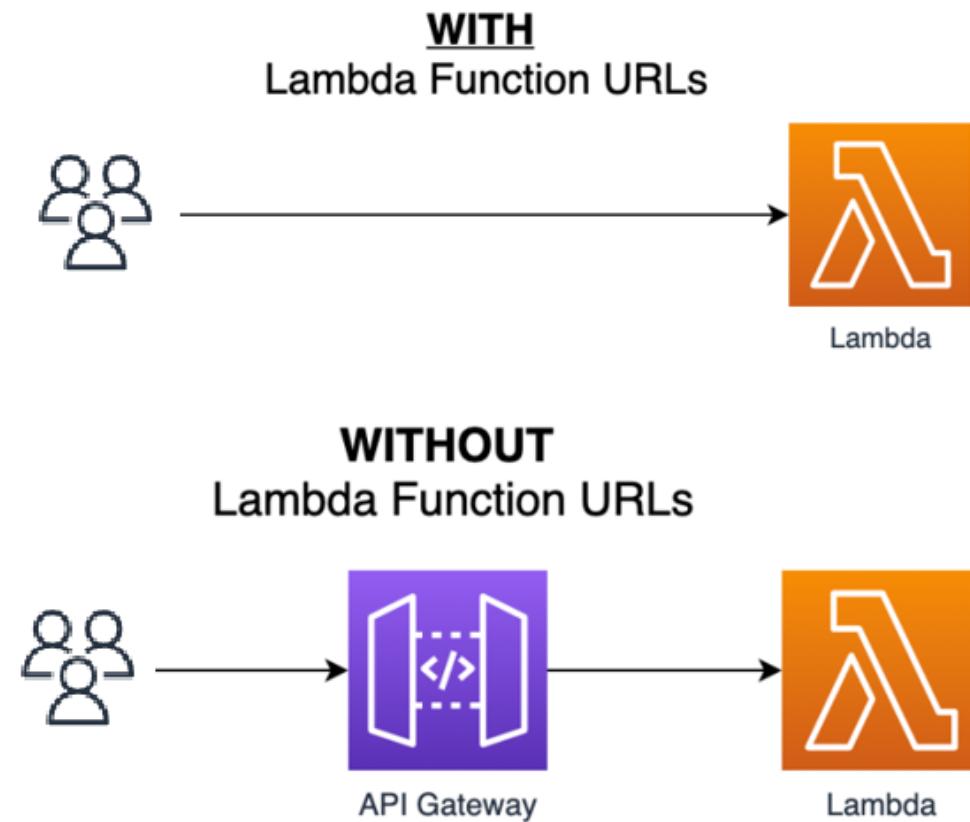
<https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/serverless-microservices.html>

# Hands-on Lab: Single Calculator Microservice Expose https methods with Lambda Function Url



# When to use Function URLs vs. Amazon API Gateway

- **API Gateway** and **Function URLs** offer similar feature. API Gateway or Function URLs, the idea is **exposing endpoints** with different accesses.
- **Function URLs** are best for use cases where you must implement a **single-function microservice** with **public endpoint** doesn't require the advanced functionality of API Gateway.
- Implementing **webhook handlers**, **form validators**, **mobile payment processing** and so on.
- **Amazon API Gateway** is a fully managed service to create, publish, maintain, monitor, and secure APIs.
- Use **Function URLs** for just needed a quick way to **expose your Lambda function** to the public internet. Use **API Gateway** for a product more robust with a whole suite of extra features.

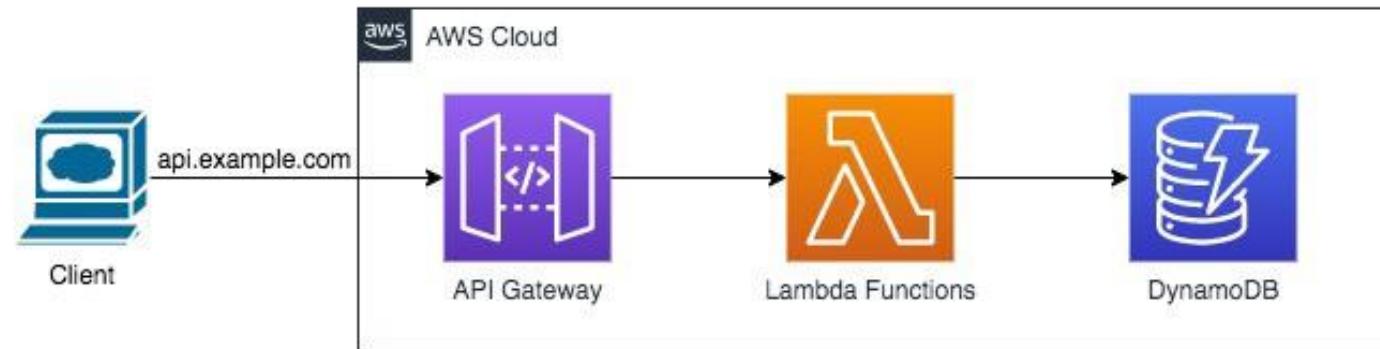


# Amazon API Gateway - API-Driven Development

→ Amazon API Gateway - API-Driven Development for Synchronous Event Sources

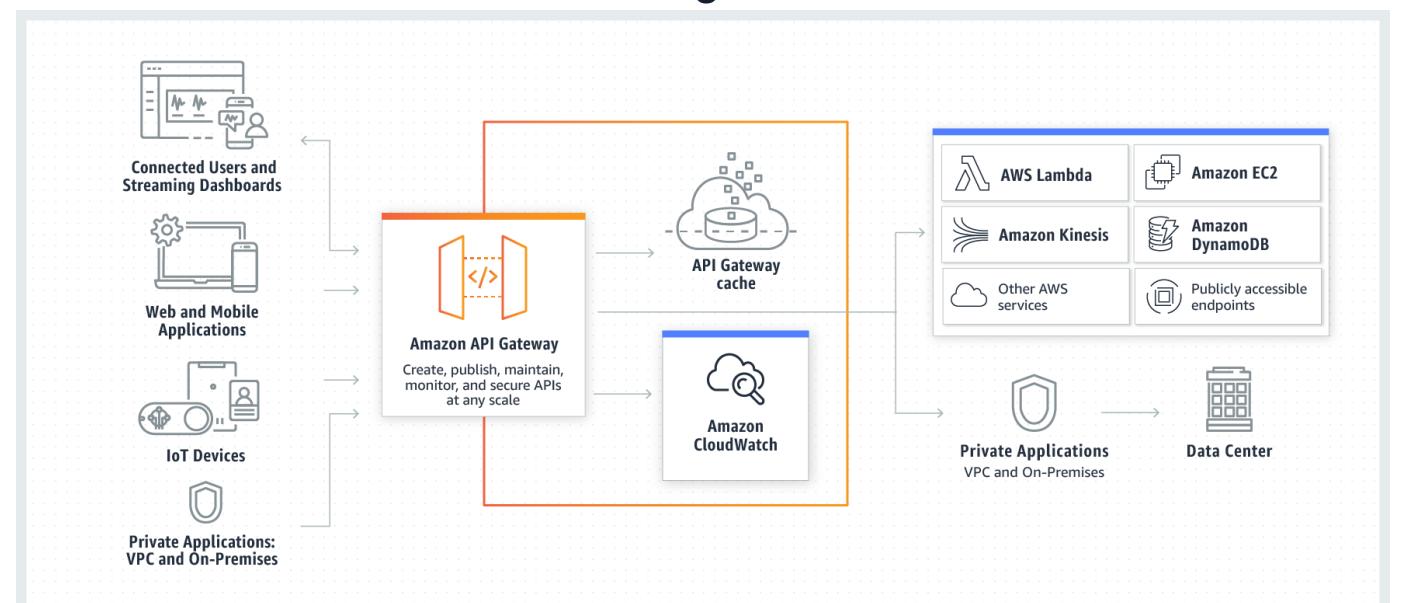
# API Gateway Restful API Development with Synchronous Lambda Event Sources

- What is Amazon API Gateway?
- Architecture of API Gateway
- Main Features of API Gateway
- Amazon API Gateway Use Cases
- API Gateway as a Lambda Synchronous Event Sources
- Amazon API Gateway Core Concepts
- Amazon API Gateway -Differences between REST - HTTP API
- Amazon API Gateway Walkthrough with AWS Management Console



# What is Amazon API Gateway?

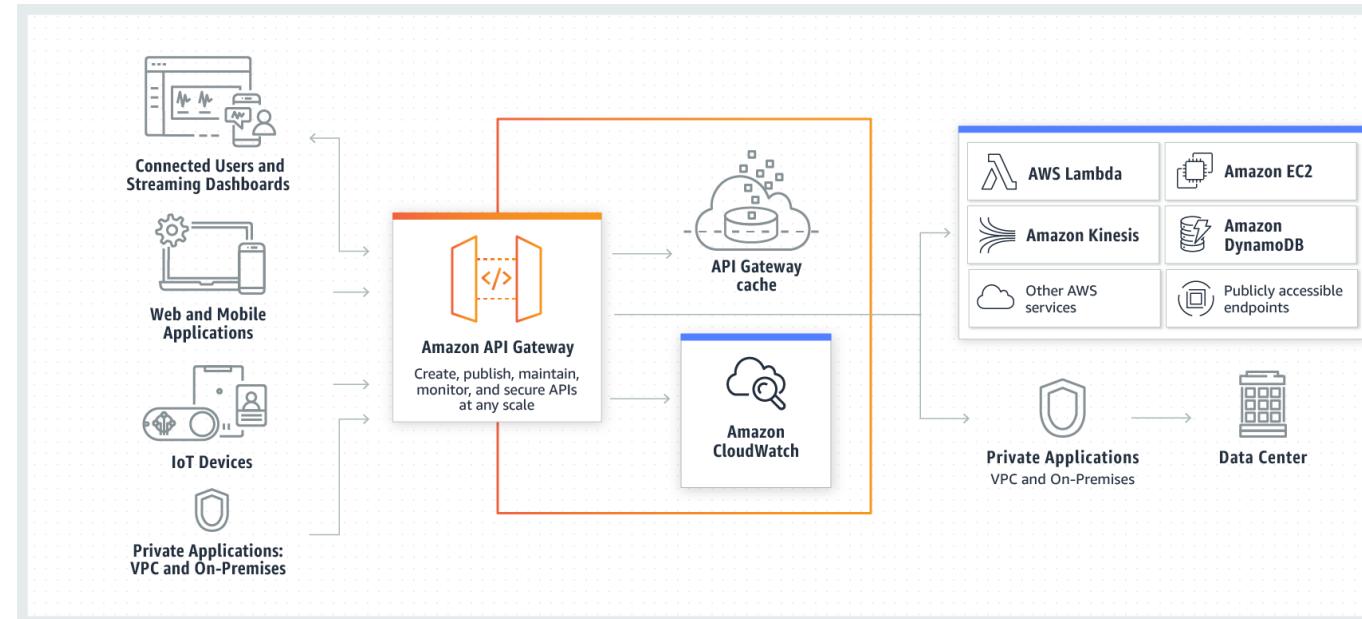
- **Fully managed** service for developers to create, publish, maintain, monitor and secure APIs at any scale.
- **Front door** for applications to access data, business logic from your backend services.
- Create **RESTful APIs** and **WebSocket APIs**
- **RESTful APIs** expose backend HTTP endpoints, AWS Lambda functions, or other AWS services.
- **RESTful APIs** optimized for serverless workloads and HTTP backends using HTTP APIs.
- **WebSocket APIs** are real-time two-way communication
- **Expose microservices** with RESTful APIs



<https://aws.amazon.com/api-gateway/>

# Architecture of API Gateway

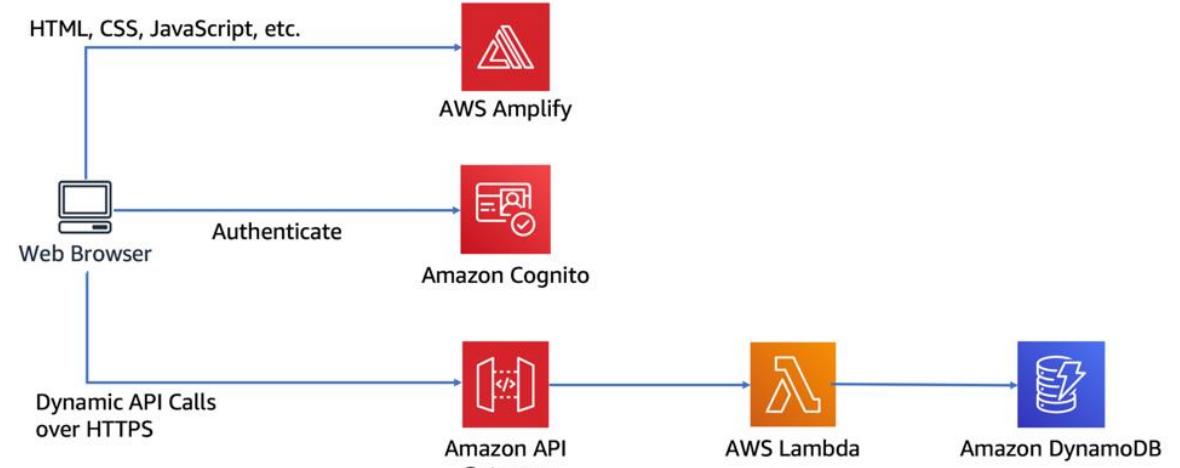
- Provide your customers with an **integrated** and **consistent** developer experience.
- Handles tasks in **accepting** and **processing** hundreds of thousands of **concurrent** API calls.
- Tasks; **traffic management**, **authorization** and access control, **monitoring**, and **API version** management.
- **Front door** for applications to access data, business logic, or functionality from your backend services.
- **Expose microservices** with RESTful APIs



<https://aws.amazon.com/api-gateway/>

# Main Features of API Gateway

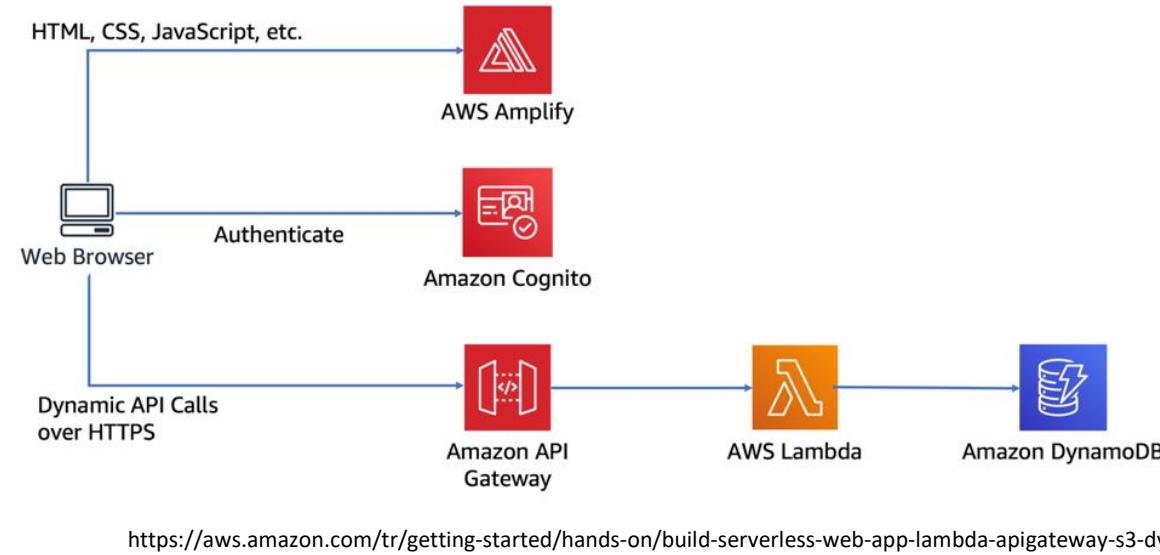
- Support for stateful (**WebSocket**) and stateless (**HTTP** and **REST**) APIs.
- **Flexible authentication** mechanisms, supports **OAuth2** and **OpenID** protocols.
- Provide **Developer portal** for publishing your APIs.
- **Canary release** deployments for safely rolling out changes.
- **CloudTrail logging** and monitoring of API usage and API changes.
- **CloudWatch access logging** and execution logging
- Ability to **use AWS CloudFormation** templates to enable API creation.
- Support for custom domain names. Integration with **AWS X-Ray**.



<https://aws.amazon.com/tr/getting-started/hands-on/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/>

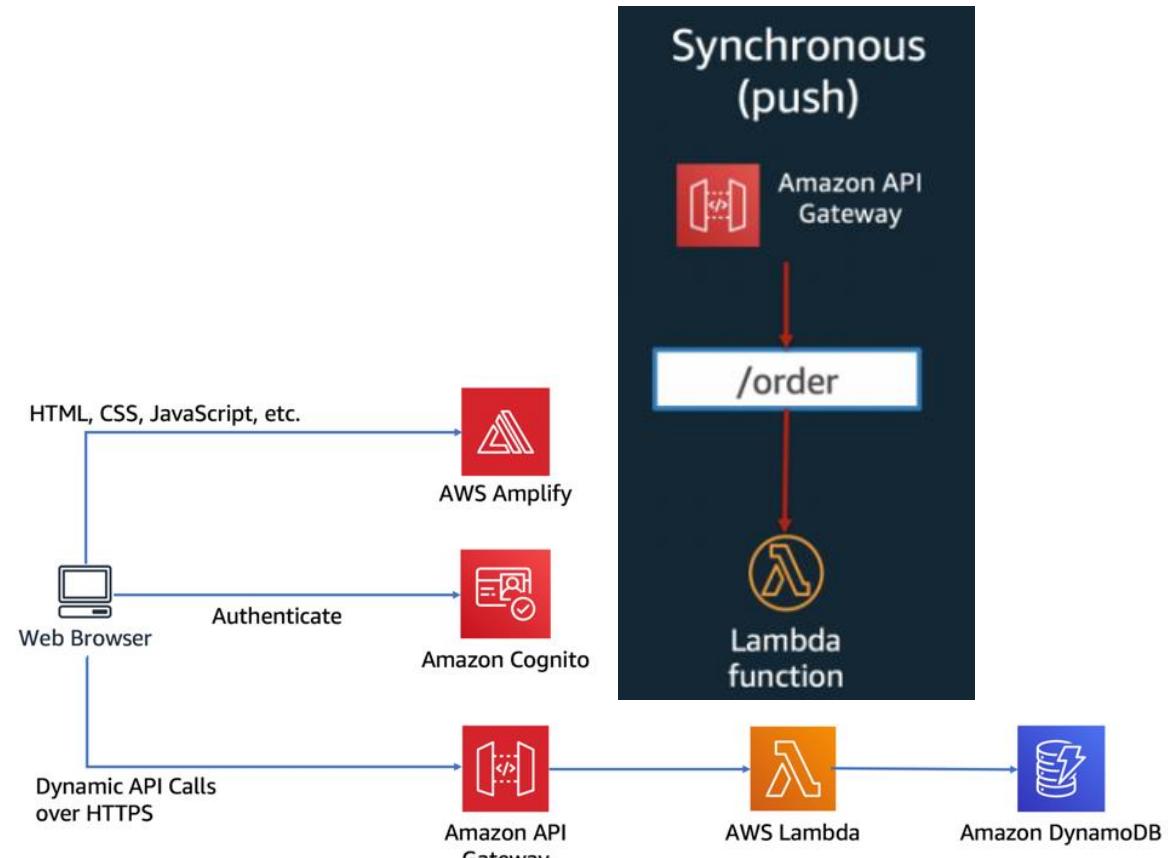
# Amazon API Gateway Use Cases

- There are 3 main use cases for Amazon API Gateway
  - Use API Gateway to create **HTTP APIs**
  - Use API Gateway to create **REST APIs**
  - Use API Gateway to create **WebSocket APIs**
- **HTTP APIs** enable you to create **RESTful APIs** with lower latency and lower cost than REST APIs.
  - Use HTTP APIs to **send requests** to AWS Lambda functions
  - Create an **HTTP API** that integrates with a Lambda function the backend
- **REST API** is made up of resources and methods.
  - A resource is a logical entity, a method corresponds to a **REST API** request
  - HTTP verbs such as **GET**, **POST**, **PUT**, **PATCH**, and **DELETE**
  - **POST /product** method can create a new product, and a **GET /basket** method can query for basket data.
- **WebSocket APIs** is client – server both send message each other.



# API Gateway as a Lambda Synchronous Event Sources

- **Synchronous** commands are request/response.
- **API Gateway** is a synchronous event source and provides a serverless **API proxy to Lambda**.
- Simple interaction common to create, read, update, and delete (**CRUD**) **API** actions.
- **Immediate response** to your API call.
- The disadvantage is that if something goes wrong or takes a long time, the whole **process is blocked**.
- API Gateway is **Synchronous trigger** of AWS Lambda
- Developing our Serverless **Product Microservices** CRUD operations.



<https://aws.amazon.com/tr/getting-started/hands-on/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/>

# Amazon API Gateway Core Concepts

- **API Gateway**

Creating, deploying, and managing a RESTful application programming interface (API) to expose backend HTTP endpoints. API Gateway has 3 main API types;

- API Gateway REST API
- API Gateway HTTP API
- API Gateway WebSocket API

- **API Gateway HTTP API**

A set of paths and methods integrated with backend HTTP endpoints or Lambda functions.

- **API Gateway REST API**

A collection of HTTP resources and methods integrated with backend HTTP endpoints, Lambda functions.

- **API Gateway WebSocket API**

A collection of WebSocket routes and route keys that are integrated with backend HTTP endpoints, Lambda functions.

# Amazon API Gateway Core Concepts

- **API Deployment**

A snapshot of your API Gateway API. The deployment must be associated with one or more API stages for clients to use it.

- **API Endpoint**

Hostname for an API deployed to a specific Region in API Gateway. The hostname is in the form {api-id}.execute-api.{region}.amazonaws.com. 3 API endpoint types;

- Edge-optimized API endpoint
- Private API endpoint
- Regional API endpoint (**default**)

- **Proxy Integration**

A simplified API Gateway integration configuration. We can set a proxy integration as HTTP proxy integration or Lambda proxy integration.

- API Gateway for HTTP proxy integration forwards all request and response between frontend and an HTTP backend. API Gateway sends the entire request as input to a backend Lambda function for Lambda proxy integration.

# Amazon API Gateway -Differences between REST - HTTP API

- **HTTP API** type is **lightweight** version of Restful apis in order to be more cost efficient.
- **HTTP API** has **less feature** than **REST API**.
- **HTTP APIs** are designed for **low-latency, cost-effective** integrations with AWS services, including AWS Lambda, and HTTP endpoints.
- See the **Development feature** comparison of both **HTTP** and **REST APIs**.
- During the course we will use **REST APIs**.

| Development                      | HTTP API | REST API |
|----------------------------------|----------|----------|
| API caching                      |          | ✓        |
| Request parameter transformation | ✓        | ✓        |
| Request body transformation      |          | ✓        |
| Request / response validation    |          | ✓        |
| Test invocation                  |          | ✓        |
| CORS configuration               | ✓        | ✓ *      |
| Automatic deployments            | ✓        |          |
| Default stage                    | ✓        |          |
| Default route                    | ✓        |          |
| Custom gateway responses         |          | ✓        |
| Canary release deployment        |          | ✓        |

<https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html>

# API Gateway - HTTP API - Walkthrough with AWS Management Console

→ DEMO - Amazon API Gateway – HTTP API – Walkthrough with AWS Management Console

# API Gateway - HTTP API

- **API Gateway**

Creating, deploying, and managing a RESTful application programming interface (API) to expose backend HTTP endpoints. API Gateway has 3 main API types;

- API Gateway REST API
- **API Gateway HTTP API**
- API Gateway WebSocket API

- **API Gateway HTTP API**

HTTP APIs enable you to create RESTful APIs with lower latency and lower cost than REST APIs. Use HTTP APIs to send requests to AWS Lambda functions or to any routable HTTP endpoint.

- Create an HTTP API that integrates with a Lambda function on the backend. When a client calls your API, API Gateway sends the request to the Lambda function and returns the function's response to the client.
- Core Concepts for API Gateway - HTTP API like
  - Routes
  - Integrations
  - Stages

# API Gateway - HTTP API - Core Concepts; Routes

- **Routes**

API Gateway uses routes to expose integrations to consumers of your API. Routes direct incoming API requests to backend resources.

- **Path variables**

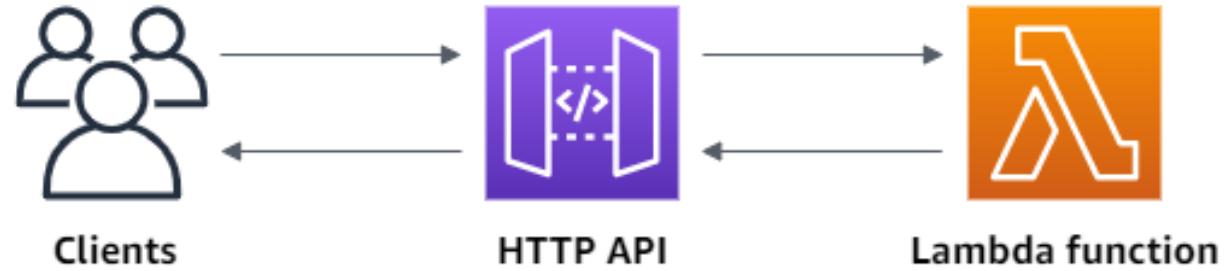
Use path variables in HTTP API routes. For example, the GET /product/{productId} route catches a GET request.

Greedy path variable catches all child resources of a route.  
{proxy+}.

- **Query string parameters**

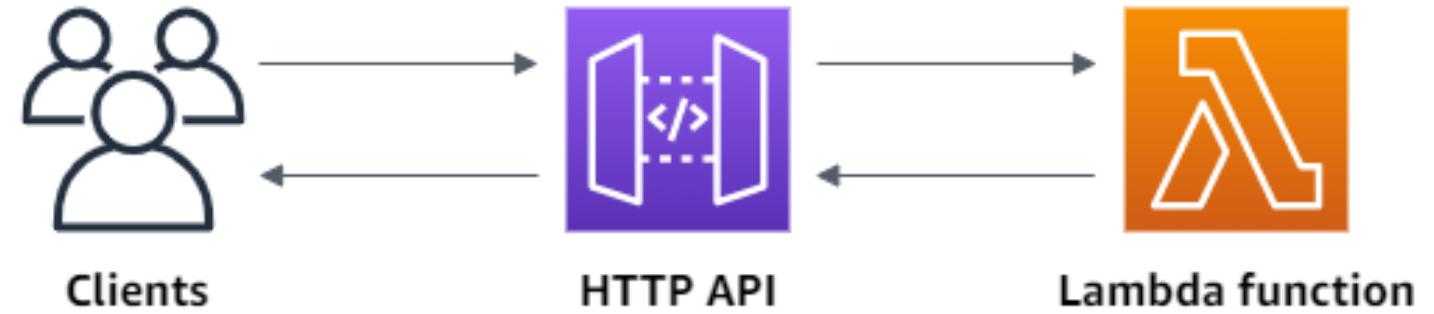
API Gateway sends query string parameters to your backend integration if they are included in a request to an HTTP API.

- <https://api-id.execute-api.us-east-2.amazonaws.com/product?id=4&type=phone>,  
the query string parameters **?id=4&type=phone** are sent to your integration.



# API Gateway - HTTP API - Routing API requests

- API Gateway selects a route in below order;
- **GET /product/phone/1**
- **GET /product/phone/{id}**
- **GET /product/{proxy+}**
- **ANY /{proxy+}**
- **\$default**

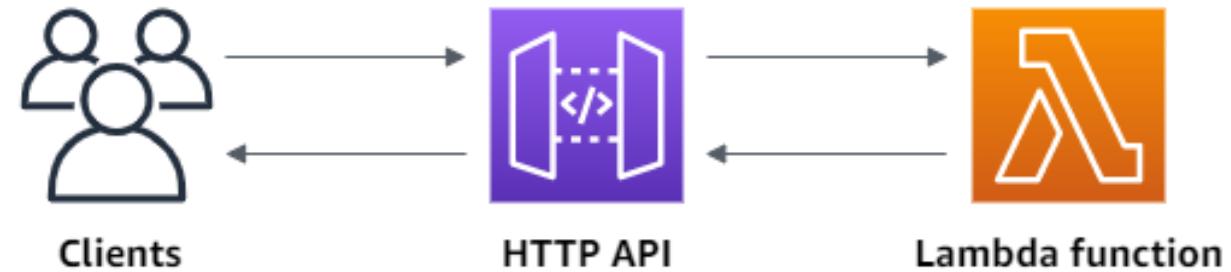


# API Gateway - HTTP API - Core Concepts; Integrations

- **Integrations**

Integrations connect a route to backend resources.

- Specify the backend services that your API will communicate with. These are called integrations.
- Create integrations with Lambda functions or public HTTP endpoints while you create your API.
- For a Lambda integration, API Gateway invokes the Lambda function and responds with the response from the function.
- For HTTP integration, API Gateway sends the request to the URL that you specify and returns the response from the URL.



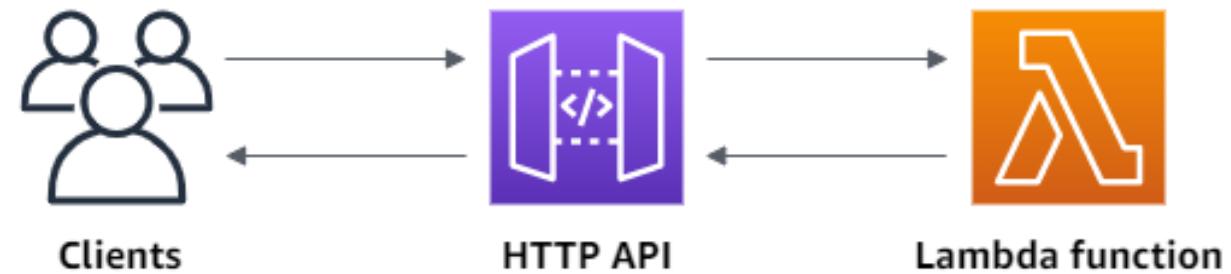
# API Gateway - HTTP API - Core Concepts; Stages

- **Stages**

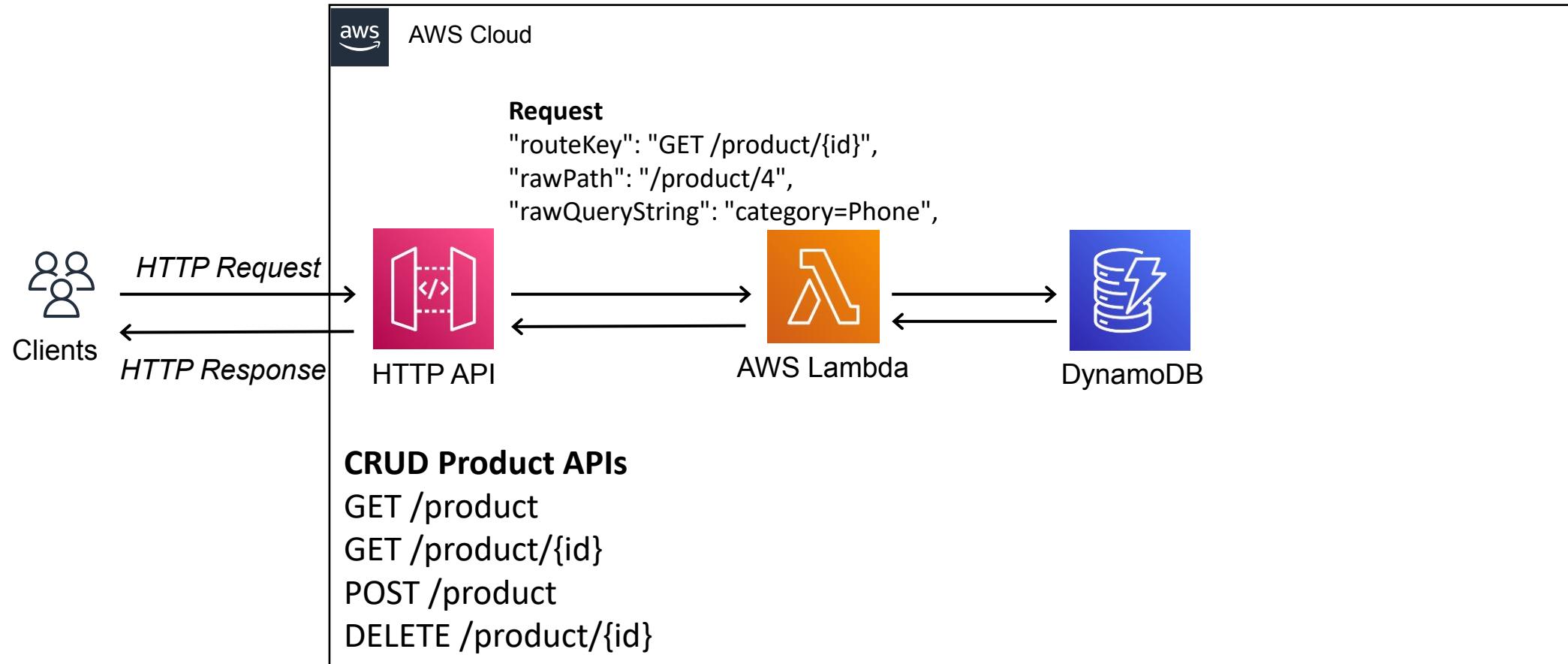
Stages are independently configurable environments.

Deploy to a stage for API configuration changes.

- Default stage named \$default. Add stages that represent environments such as development or production.
- Logical reference to a lifecycle state of our API like dev, prod and so on.
- API stages are identified by their API ID and stage name, and they're included in the URL of invoke the API.
- Create a \$default stage; [https://{{api\\_id}}.execute-api.{{region}}.amazonaws.com/](https://{{api_id}}.execute-api.{{region}}.amazonaws.com/).
- A deployment is a snapshot of your API configuration. Enable automatic deployments.

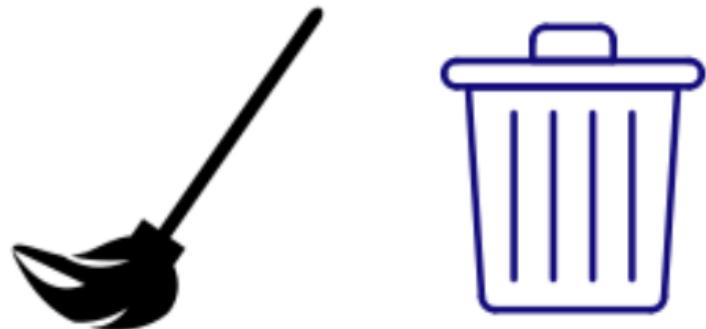


# Hands-on Lab: Build CRUD Microservice with HTTP API and Lambda



# Clean up Resources

- Delete AWS Resources that we create during the section.



# API Gateway - REST API - Walkthrough with AWS Management Console

→ DEMO - Amazon API Gateway – REST API – Walkthrough with AWS Management Console

# API Gateway - REST API

- **API Gateway**

Creating, deploying, and managing a RESTful application programming interface (API) to expose backend HTTP endpoints. API Gateway has 3 main API types;

- **API Gateway REST API**
- API Gateway HTTP API
- API Gateway WebSocket API

- **API Gateway REST API**

A collection of HTTP resources and methods integrated with backend HTTP endpoints, Lambda functions, or other AWS services. We can distribute this collection in one or more "stages".

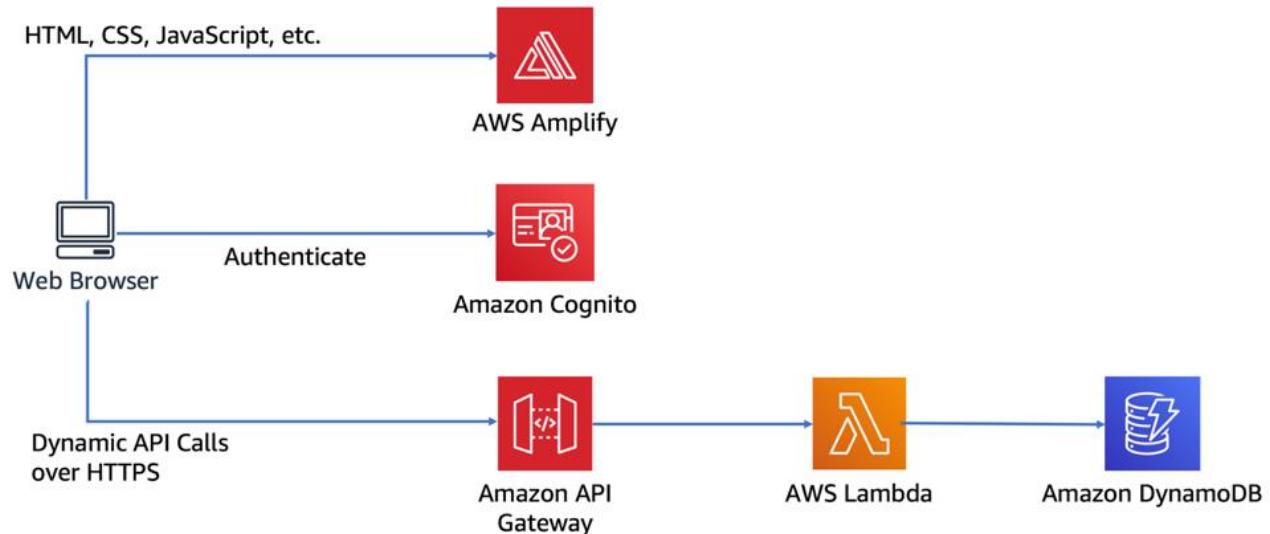
- Each API resource can expose one or more API methods with unique HTTP verbs powered by API Gateway.

- Core Concepts for API Gateway - REST API like

- Resources
- Methods
- Stages
- Authorizers

# API Gateway REST API Core Concepts; Resources, Methods, Stages, Authorizers

- API Gateway **REST API** is made up of resources and methods.
  - A resource is a logical entity
  - A method corresponds to a **REST API** request
  - For example, "**/products**" could be a path to a resource
  - HTTP verbs such as **GET**, **POST**, **PUT**, **PATCH**, and **DELETE**
  - **POST /product** method can create a new product, and a **GET /basket** method can query for basket data.
- API Gateway REST API is encapsulated with method requests and method responses.



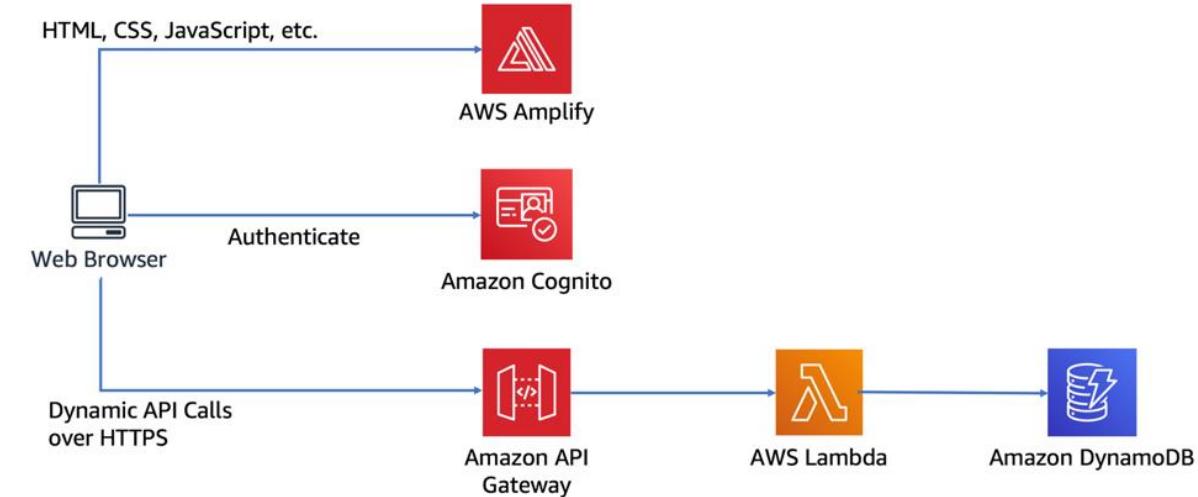
# API Gateway REST API Core Concepts; Stages, Authorizers

- **Stage**

A stage is a named reference to a deployment, which is a snapshot of the API. You use a Stage to manage and optimize a particular deployment. Configure stage settings to enable caching, customize request throttling, configure logging, define stage variables, or attach a canary release for testing.

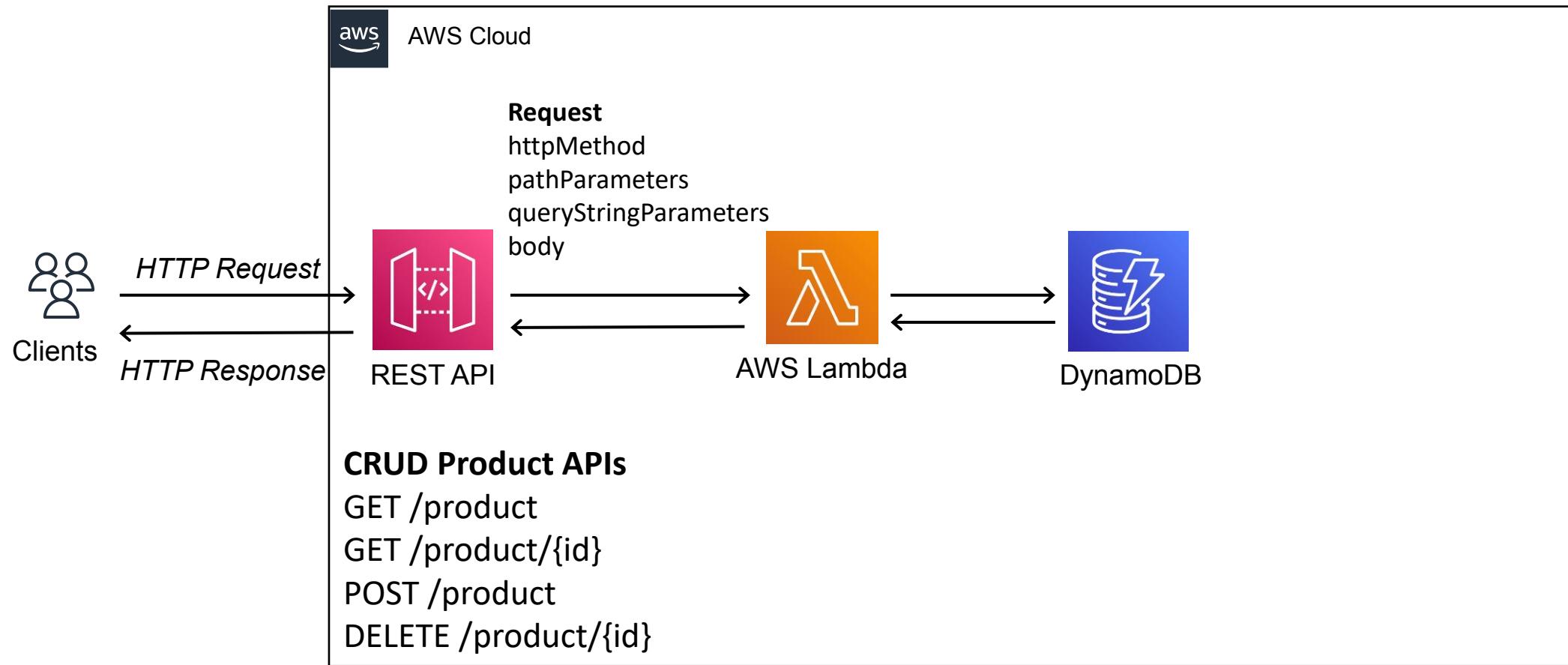
- **Authorizers**

Authorizers enable you to control access to your APIs using Amazon Cognito User Pools or a Lambda function.



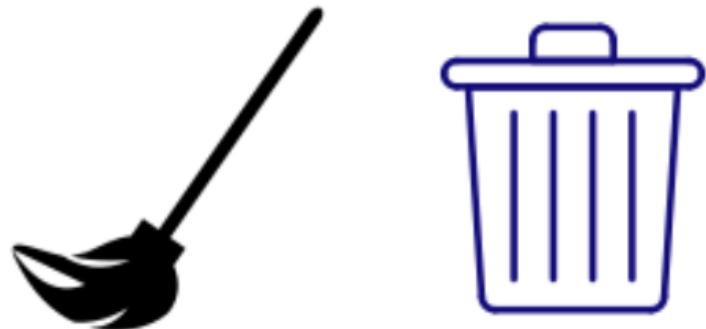
<https://aws.amazon.com/tr/getting-started/hands-on/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito>

# Hands-on Lab: Build CRUD Microservice with REST API and Lambda



# Clean up Resources

- Delete AWS Resources that we create during the section.



# API Gateway - WebSocket API - Walkthrough with Console

→ DEMO - Amazon API Gateway – WebSocket API – Walkthrough with AWS Management Console

# API Gateway - WebSocket API

- **API Gateway**

Creating, deploying, and managing a RESTful application programming interface (API) to expose backend HTTP endpoints. API Gateway has 3 main API types;

- API Gateway REST API
- API Gateway HTTP API
- **API Gateway WebSocket API**

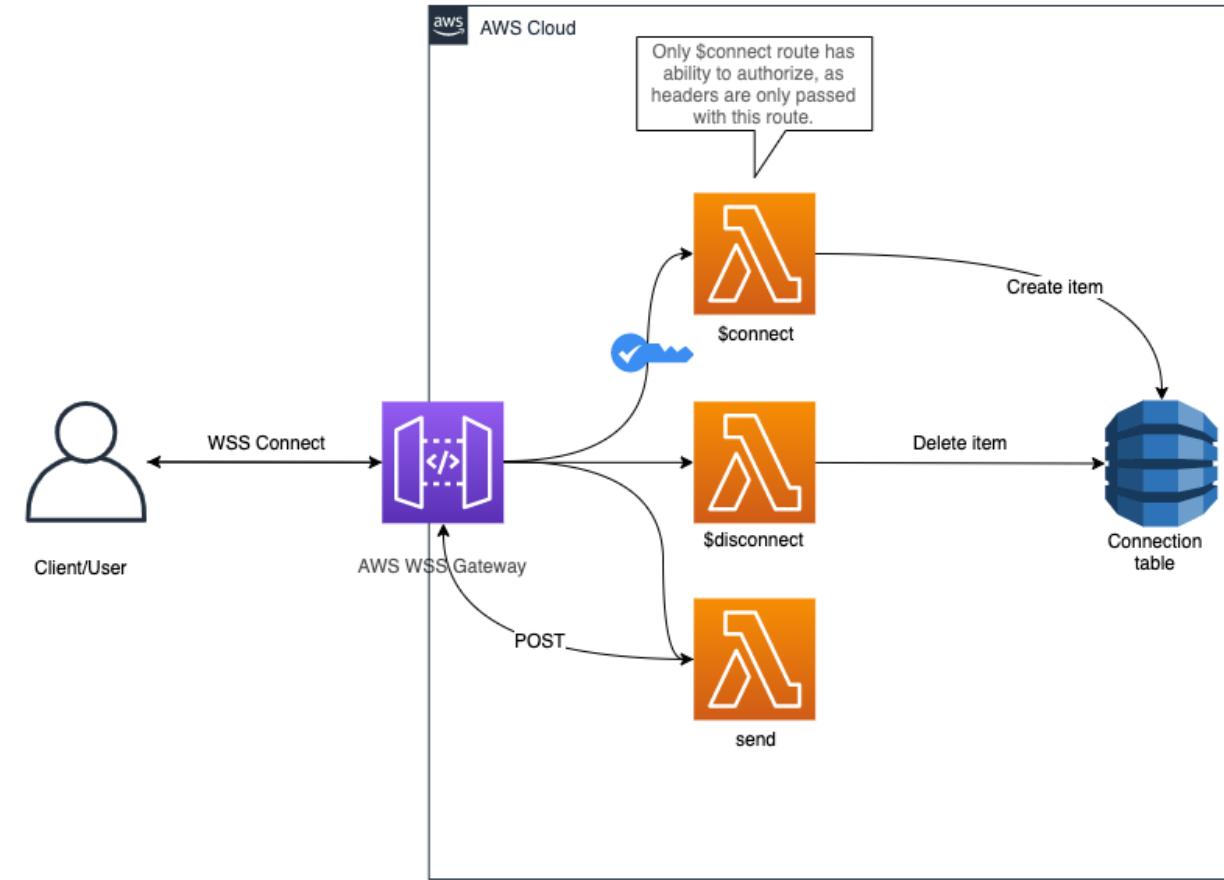
- **API Gateway WebSocket API**

A collection of WebSocket routes and route keys that are integrated with backend HTTP endpoints, Lambda functions, or other AWS services. We can Build real-time two-way communication applications, such as chat apps and streaming dashboards, with WebSocket APIs.

- API Gateway maintains a persistent connection to handle message transfer between your backend service and your clients.

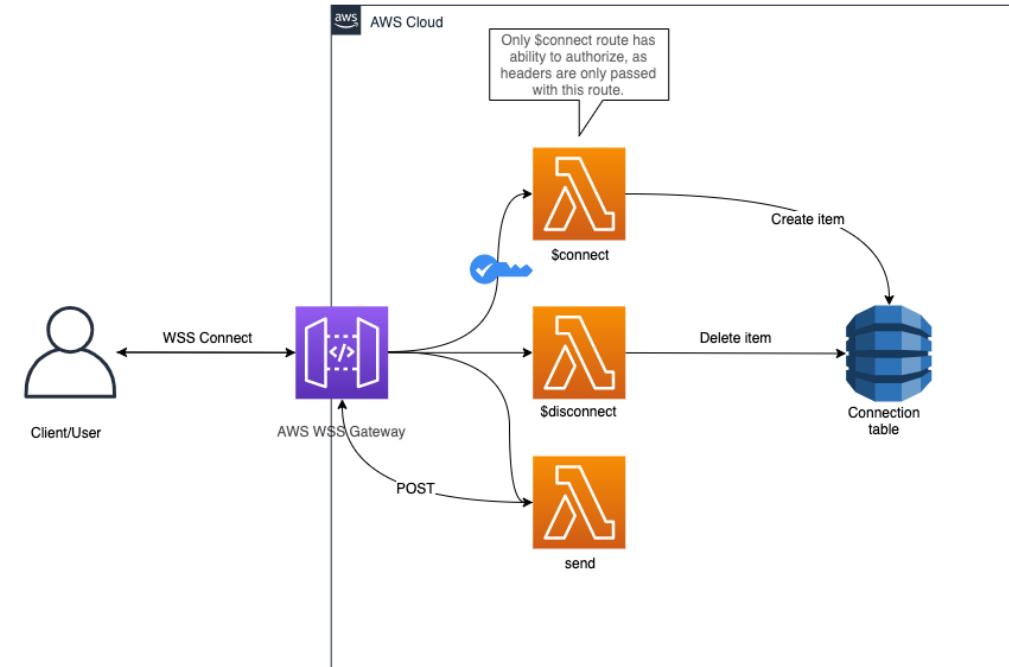
# Use API Gateway to create WebSocket APIs

- In a **WebSocket API**, the client and the server can both send messages to each other at any time. Backend servers can easily **push data** to connected users.
- API Gateway WebSocket API and AWS Lambda to **send and receive messages** to and from individual users in a chat room.
- Use API Gateway WebSocket APIs to build **secure, real-time communication** applications
  - Chat applications
  - Real-time dashboards such as stock tickers
  - Real-time alerts and notifications
- We will see **Core Concepts** for API Gateway - WebSocket API like **Routes, Integrations and Stages**.



# API Gateway WebSocket API Core Concepts; Routes, Integrations, Stages

- WebSocket API, the client and the server can both send messages to each other at any time. It is A collection of WebSocket routes and route keys that are integrated with backend HTTP endpoints, Lambda functions, or other AWS services.
- **Route selection expression**  
Tells API Gateway which route to call when a client sends a message. Uses the route selection expression to determine which route to invoke when a client sends a message.
- **Routes**  
API Gateway uses routes to expose integrations to clients.  
API Gateway evaluates the route selection expression of your API at runtime to determine which route to invoke.
  - Predefined routes
  - Custom routes



# API Gateway WebSocket API Core Concepts; Routes, Integrations, Stages – Part 2

- **Predefined routes**

\$connect, \$disconnect and \$default routes. **\$connect route** is triggered when a client connects to your API. **\$disconnect route** is triggered when either the server or the client closes the connection. **\$default route** is triggered if no matching route is found.

- **Custom Routes**

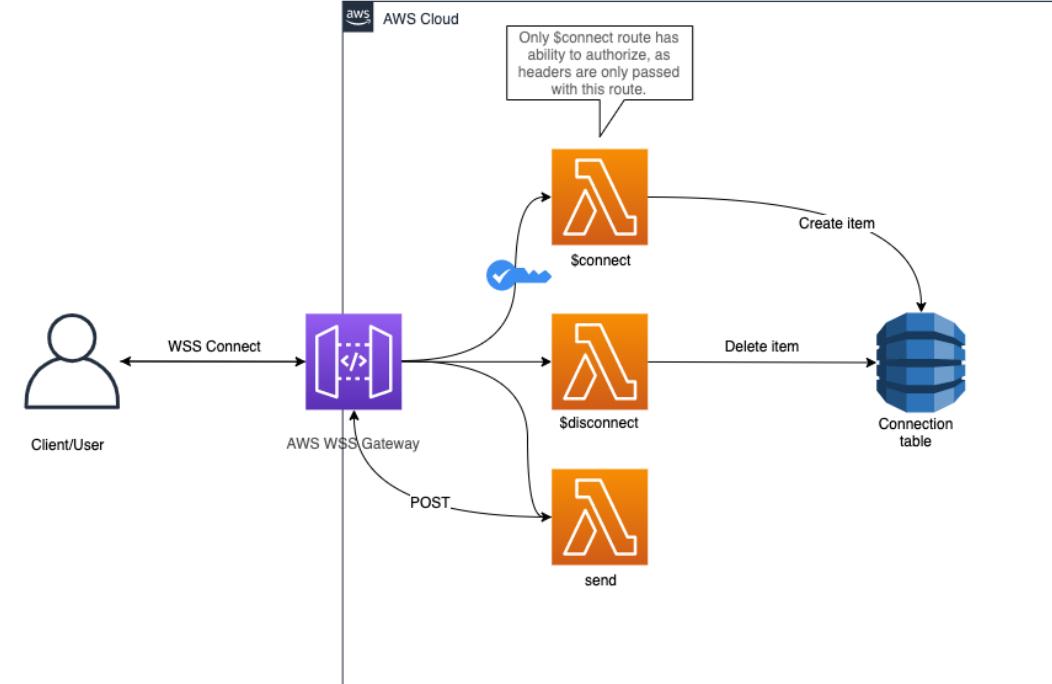
Invoke a specific integration based on message content. When the evaluated route selection expression matches a custom route, API Gateway invokes the integration.

- **Integrations**

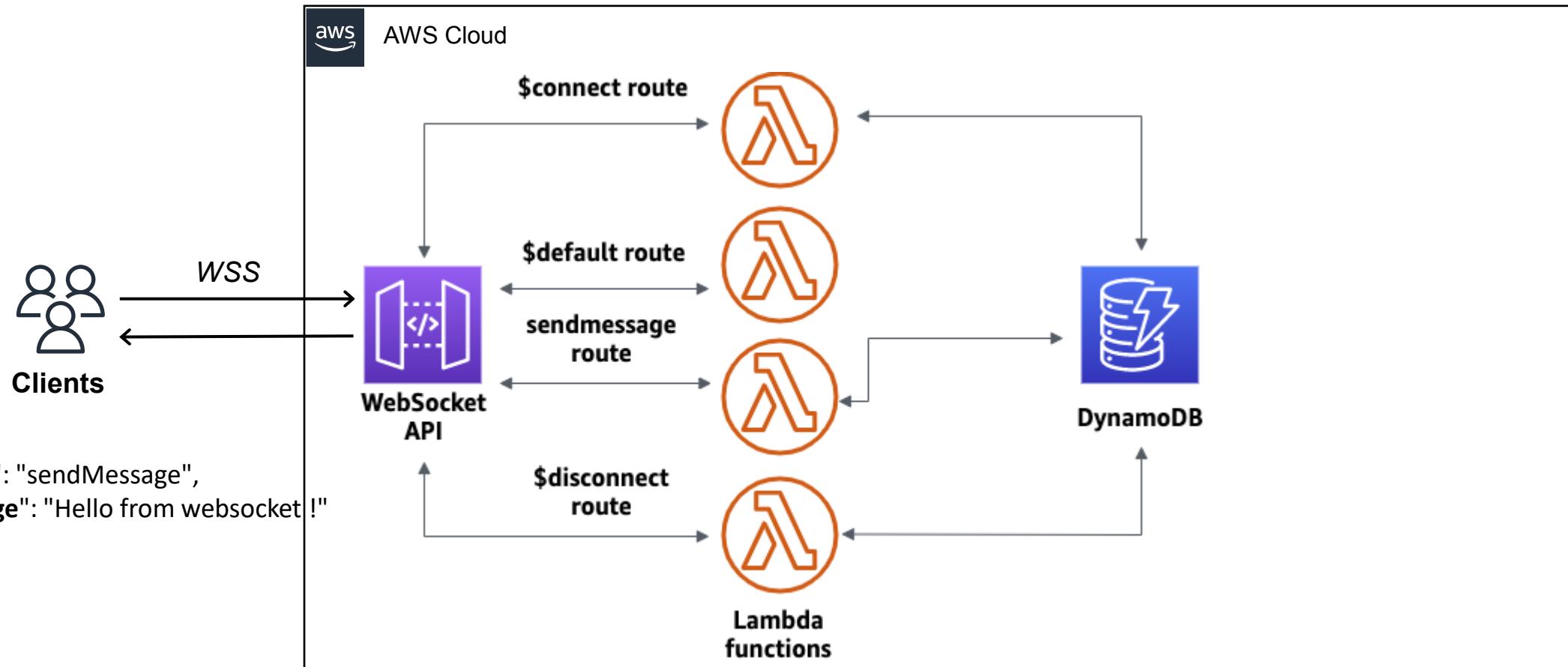
To deploy API, we must set up at least one route. All routes must have an integration attached.

- **Stages**

Independently configurable environments. Must deploy to a stage for API configuration changes to take effect.

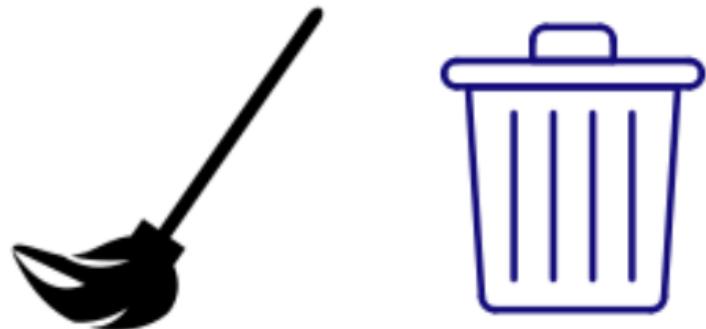


# Hands-on Lab: Build Serverless Chat App with a WebSocket API and Lambda



# Clean up Resources

- Delete AWS Resources that we create during the section.

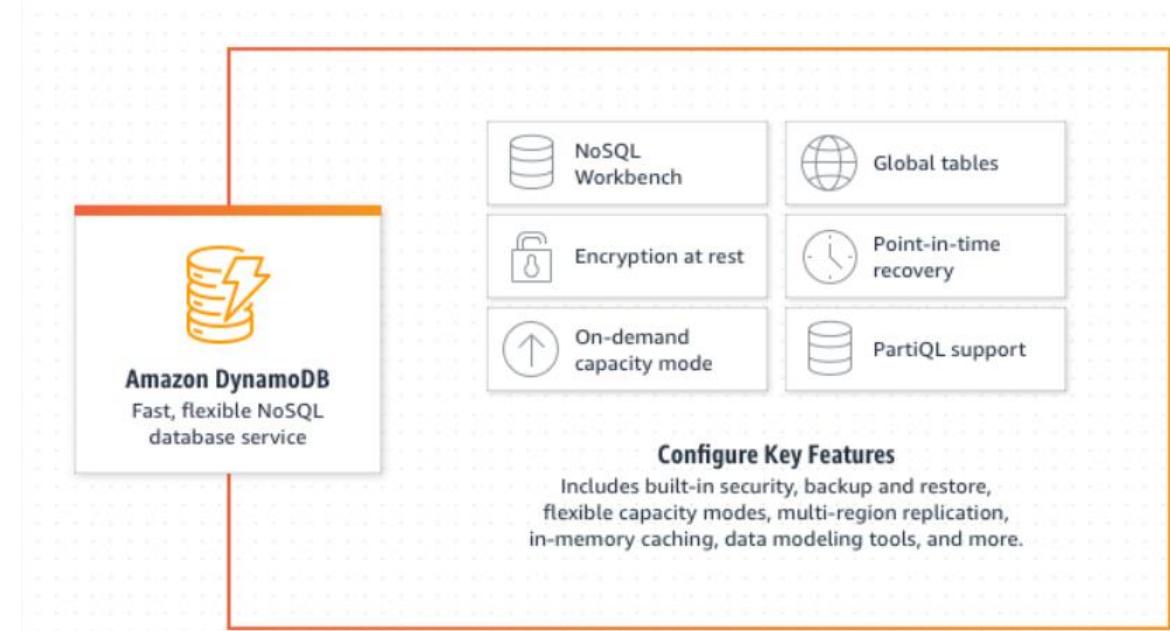


# Amazon DynamoDB - Serverless NoSQL Data Persistence

→ Building AWS DynamoDB Serverless NoSQL Data Persistence

# What Is Amazon DynamoDB?

- **Amazon DynamoDB** is a fully managed NoSQL database service that provides **fast** and **predictable** performance with seamless scalability.
- **Serverless, key-value NoSQL database** designed to run high-performance applications at any scale.
- Create **database tables** that can store and retrieve any amount of data and serve any level of request traffic.
- **Scale up or down** the throughput of your tables without downtime or performance degradation. DynamoDB provides on-demand backup capability.
- **High Availability and Durability**  
DynamoDB automatically spreads data and traffic for your tables across enough servers to meet your throughput.



<https://aws.amazon.com/dynamodb/>

# AWS DynamoDB Core Concepts - Tables, Items, Attributes, Indexes

- Tables, Items, and Attributes are the core components.
- Uses primary keys and secondary indexes to uniquely identify each item in a table for greater query flexibility.

- **Tables**

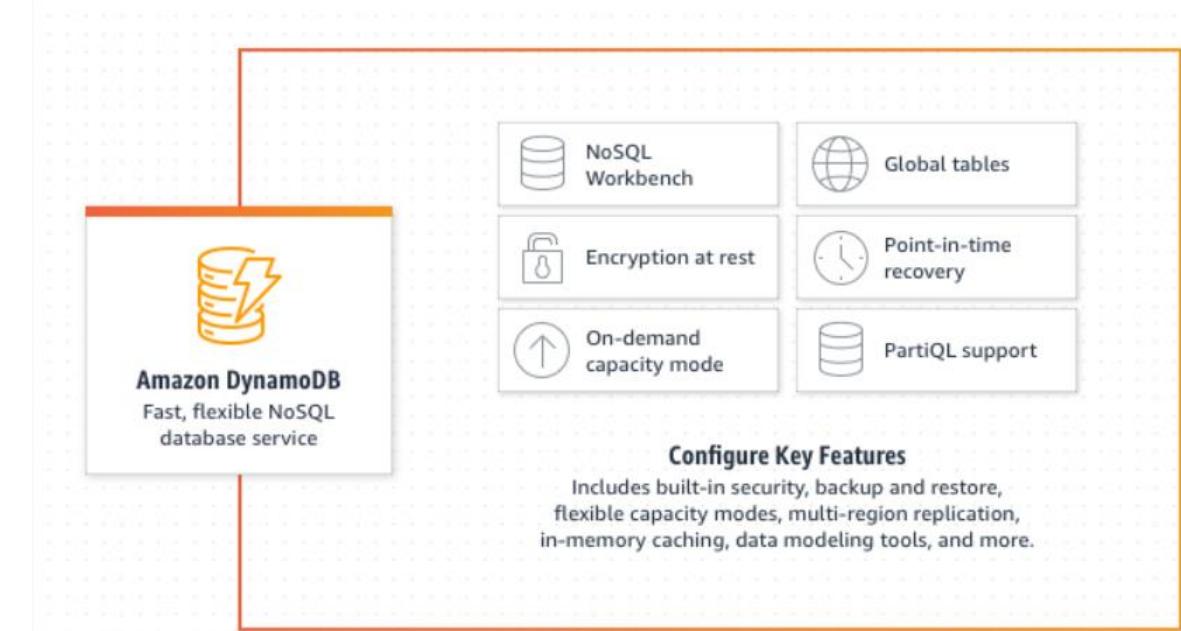
DynamoDB stores data in tables. A table is a collection of data items. For example, see the sample table People.

- **Items**

Each table contains zero or more items. An item is a set of attributes that can be uniquely identified among all of the other items.

- **Attributes**

Each item is composed of one or more attributes. An attribute is a fundamental data element.



<https://aws.amazon.com/dynamodb/>

# AWS DynamoDB Core Concepts - Tables, Items, Attributes, Indexes

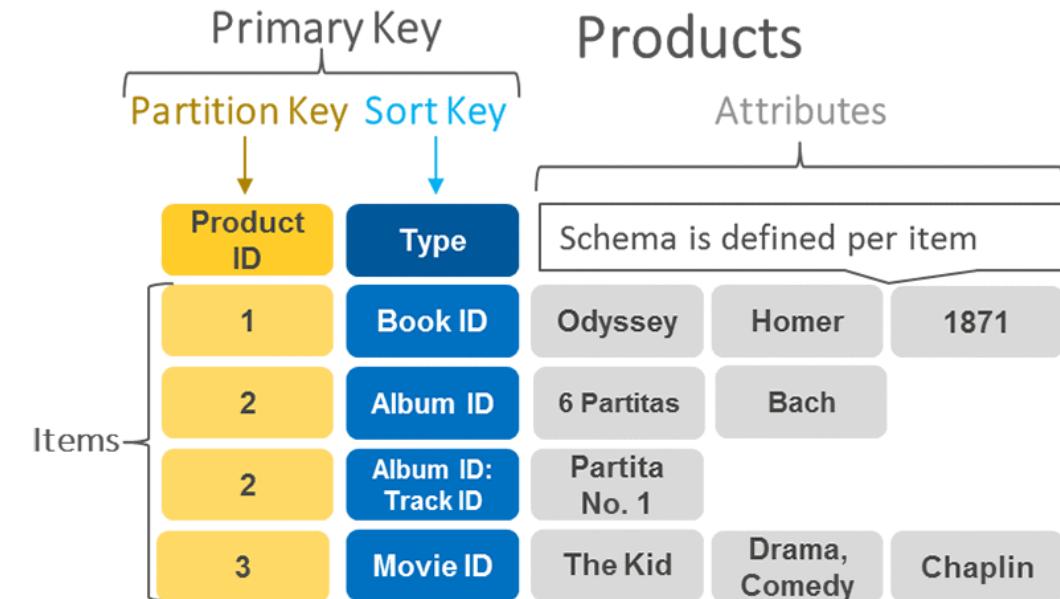


<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html>



# DynamoDB Primary Key, Partition Key and Sort Key

- A primary key **uniquely identifies** each item in the table, so no two items can have the same key. DynamoDB supports two different kinds of primary keys:
  - Partition key
  - Partition key and sort key
- **Partition key**  
A simple primary key, composed of one attribute known as the partition key.
- **Partition key and Sort Key**  
It is Referred to as a composite primary key, this type of key is composed of two attributes. The first attribute is the partition key, and the second attribute is the sort key.
- DynamoDB uses the partition key value as input to an internal hash function. A composite primary key gives you additional flexibility when **querying data**.



<https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/>

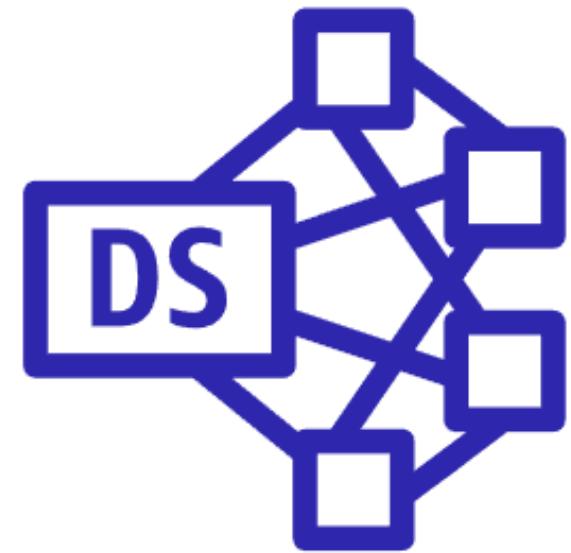
# DynamoDB Partitions and Data Distribution with Partition Key and Sort Key

- **Partition** is an allocation of storage for a table, backed by SSD drivers and automatically replicated across multiple Availability Zones.
- DynamoDB allocates sufficient partitions for the table so handle provisioned throughput requirements. Partition management performs automatically.
- **Data Distribution: Partition Key**  
If table has a simple primary key; partition key only, DynamoDB stores and gets each item based on its partition key value.
- **Data Distribution: Partition Key and Sort Key**  
If the table has a composite primary key; partition key and sort key, DynamoDB calculates the hash value of the partition key. Stores all the items with the same partition key values are stores physically close together, and ordered by sort key value.



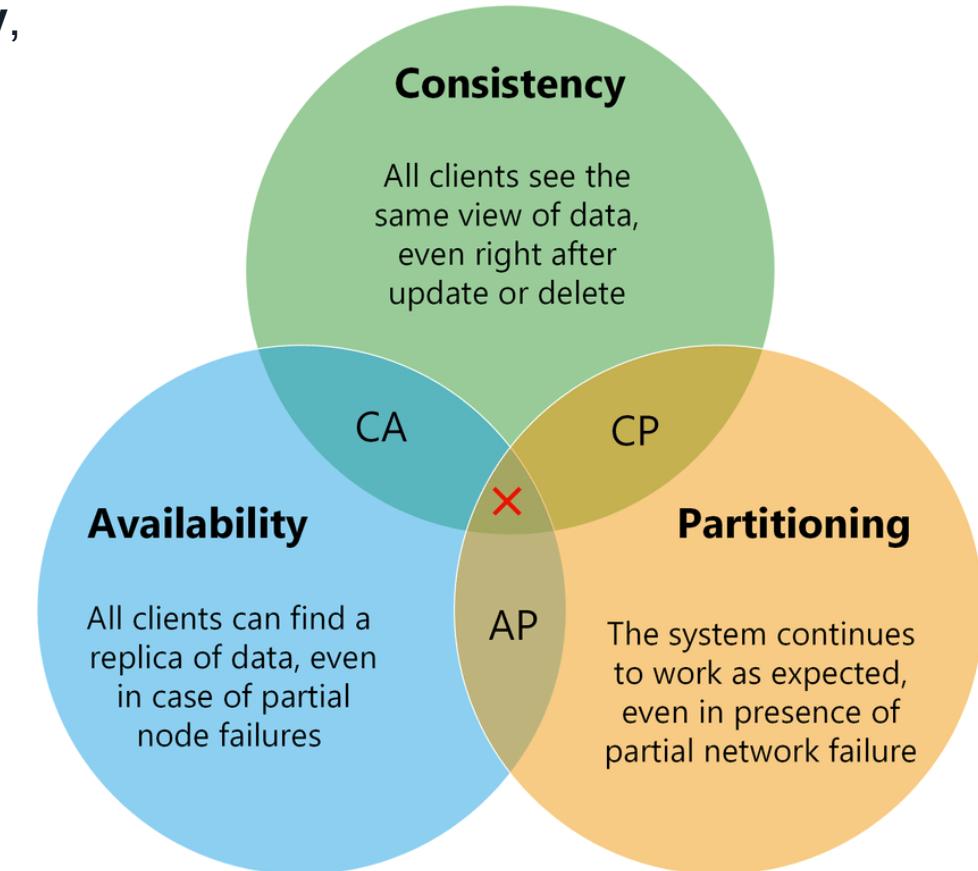
# Amazon DynamoDB Availability

- Amazon DynamoDB available in multiple AWS Regions around the world. Regions are independent and isolated.
- Every AWS Region consists of multiple distinct locations called Availability Zones. Each Availability Zone is isolated from failures in other Availability Zones.
- When our application writes data to a DynamoDB table, the data is eventually consistent across all storage locations.
- DynamoDB supports eventually consistent and strongly consistent reads.
  - **Eventually Consistent Reads**
  - **Strongly Consistent Reads**



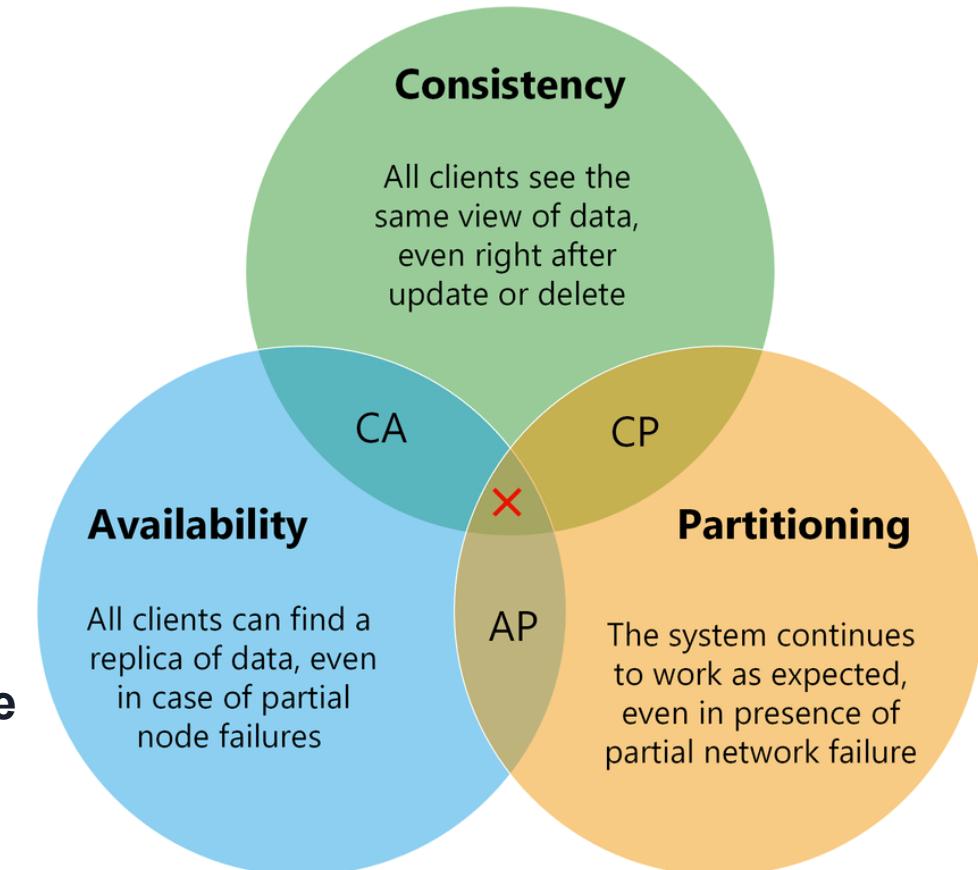
# CAP Theorem and Eventual Consistency

- The **CAP Theorem** prove that in a distributed system, **Consistency**, **Availability**, and **Partition Tolerance** cannot all be achieved at the same time. Distributed systems should sacrifice between consistency, availability, and partition tolerance.
- **Consistency**  
If the system get any read request, the data should return last updated value from database. When consistent not provide, the system must block the request until all replicas update.
- **Availability**  
If distributed system can respond all request any time, the system has high availability. Availability in a distributed system ensures the system remains operational 100% of the time.
- **Partition Tolerance**  
Network partitioning. Parts of your system are located in different networks. The ability of the system to continue its life in case of any communication problem.



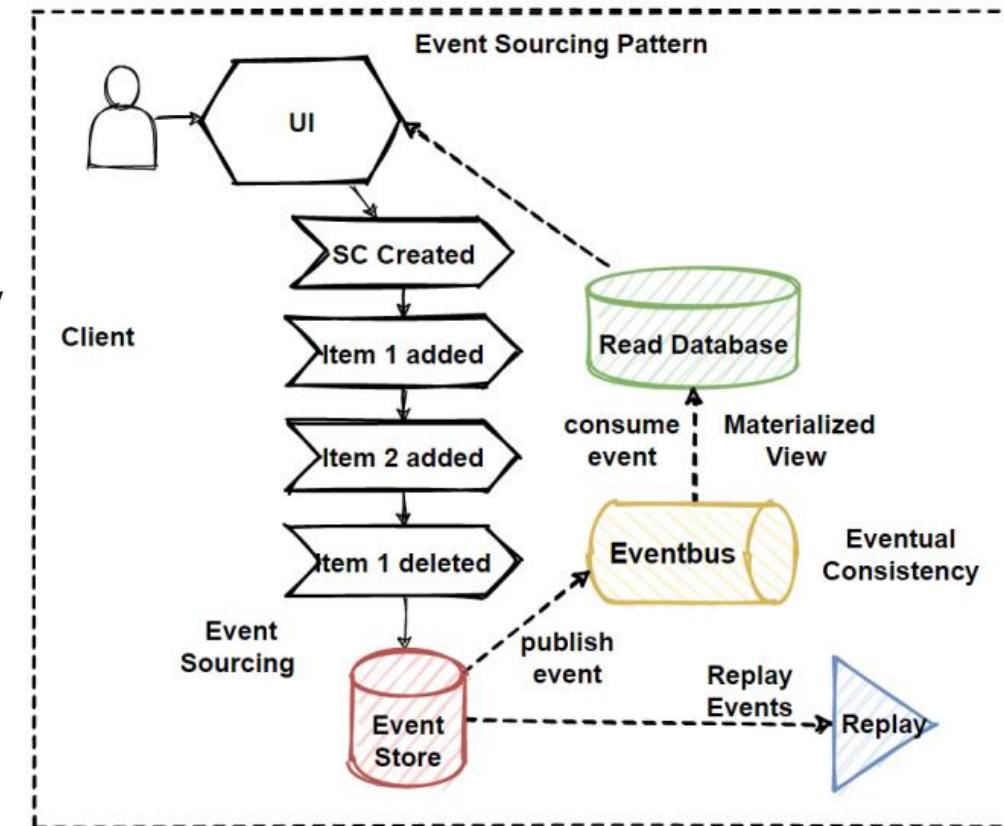
# Consistency and Availability at the same time ?

- We should **sacrifice Availability or Consistency** in distributed systems.
- **Partition Tolerance** is a must for distributed architectures.  
Emergence of **NoSQL databases** is to easily overcome the Single Point of Failure problem.
- Distributed architecture, **Partition Tolerance** seems to be a **must-have feature**.
- No-sql database systems like **MongoDB, Cassandra, DynamoDB** you can see that none of them gave up on Partition Tolerance and made a choice between Availability and Consistency.
- Mostly in **microservices architectures** choose **Partition Tolerance** with **High Availability** and follow **Eventual Consistency** for data consistency.



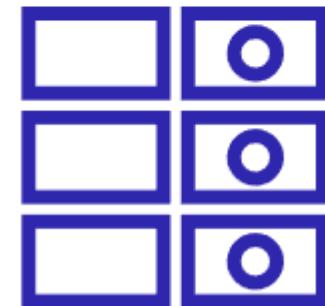
# Eventual Consistency Principle

- Used for systems that prefer high availability to instant consistency. The system will become consistent after a certain time, but it does not guarantee instant consistency.
- Offers to be consistent after a certain time.
- According to CAP Theorem, we need to consider the "consistency level" that we need. There are 2 type of "consistency level":;
- **Strict Consistency**  
When we save data, the data should affect and seen immediately for every client. Debit or withdraw on bank account.
- **Eventual Consistency**  
when we write any data, it will take some time for clients reading the data. Youtube video counters, we seen different video seen numbers in different sessions.



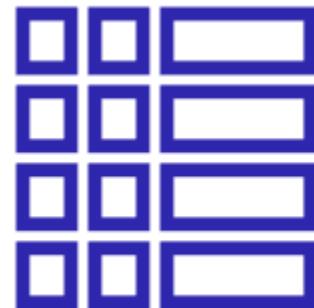
# Amazon DynamoDB Read Consistency

- DynamoDB supports eventually consistent and strongly consistent reads.
  - Eventually Consistent Reads
  - Strongly Consistent Reads
- Eventually Consistent Reads**  
When you read data from a DynamoDB table, the response might not reflect the results of a recently completed write operation.
- Strongly Consistent Reads**  
When you request a strongly consistent read, DynamoDB returns a response with the most up-to-date data.

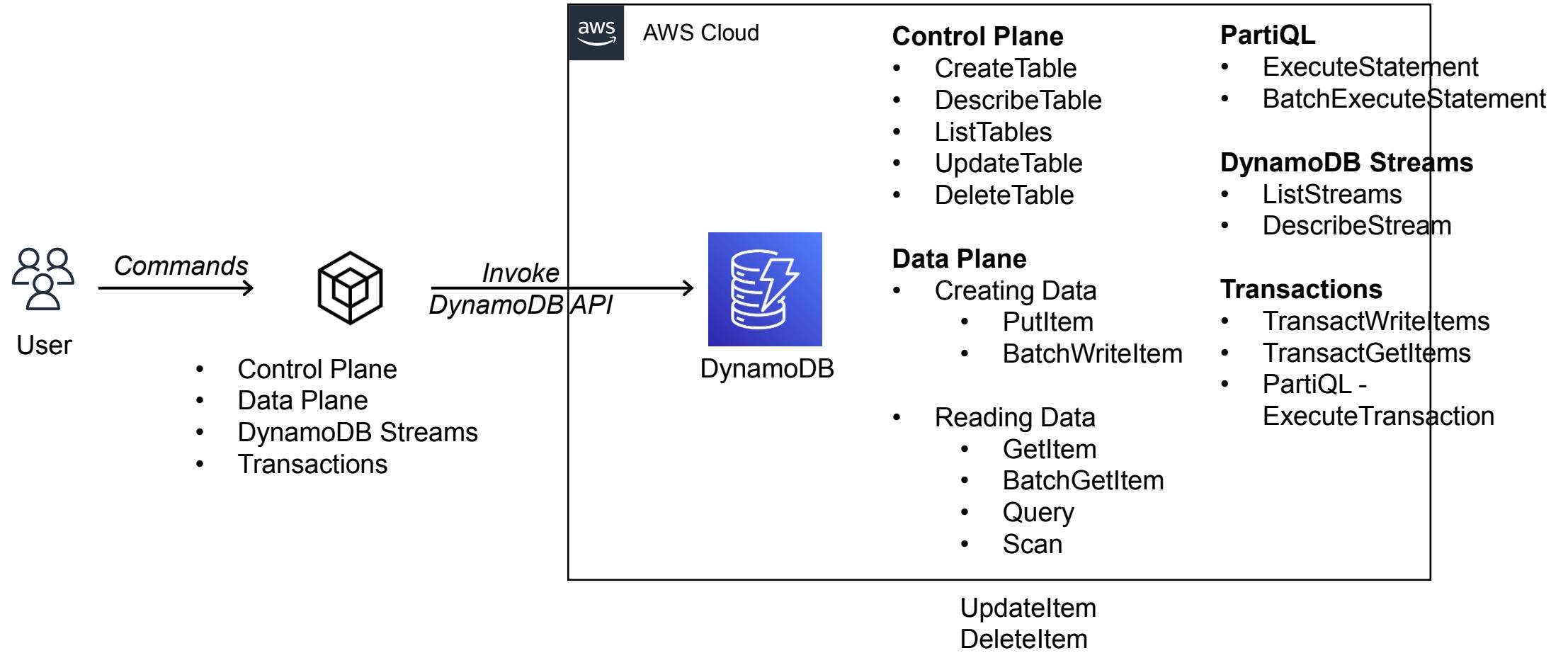


# Amazon DynamoDB Read and Write Capacity Mode

- Amazon DynamoDB has two read/write capacity modes
  - On-demand
  - Provisioned (default)
- The read/write capacity mode controls how you are charged for read and write throughput and how you manage capacity.
- **On-Demand Mode**  
DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you pay only for what you use.
- **Provisioned Mode**  
Specify the number of reads and writes per second that you require for your application. You can use auto scaling to adjust your table's provisioned capacity automatically in response to traffic changes.



# Amazon DynamoDB API References

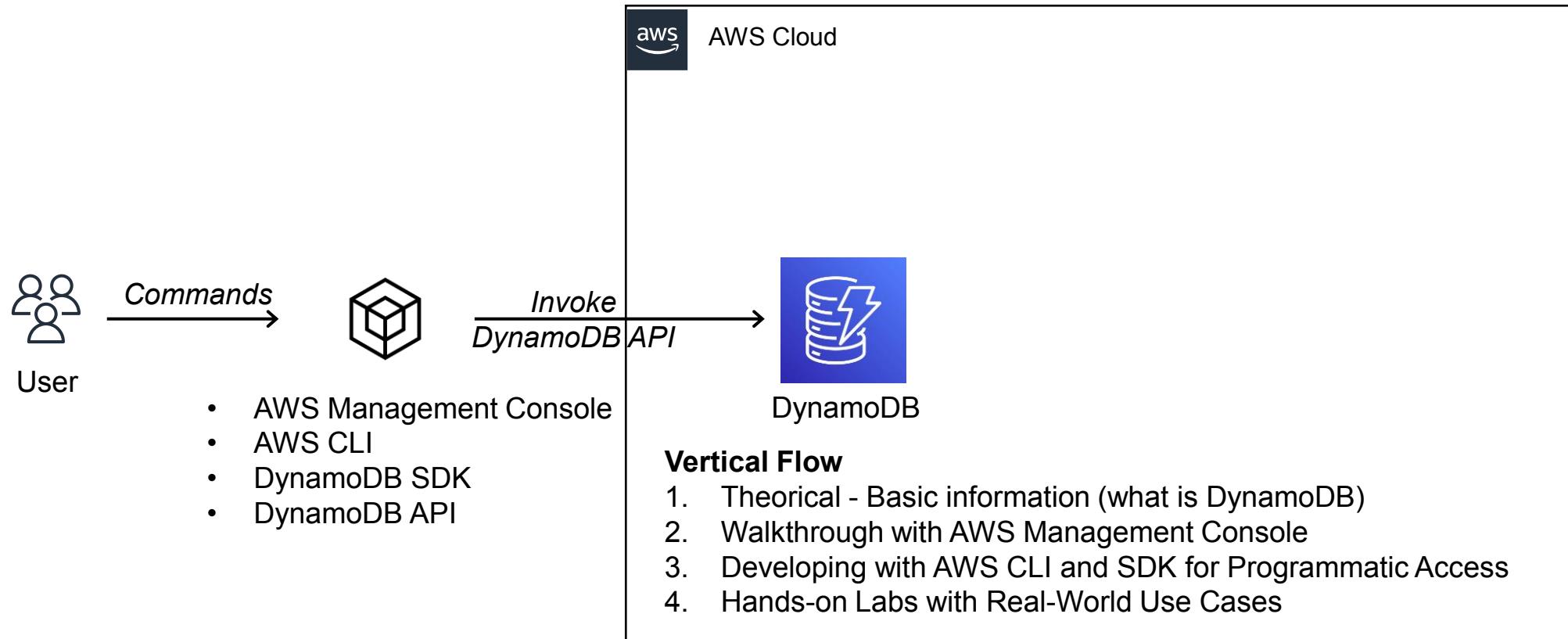


# Amazon DynamoDB PartiQL - A SQL-Compatible Query Language for DynamoDB

- **PartiQL** provides **SQL-compatible query access** across multiple data stores containing structured data, semi structured data, and nested data.
- Amazon **DynamoDB** supports **PartiQL**, a SQL-compatible query language, to **select**, **insert**, **update**, and **delete** data in Amazon DynamoDB. Using PartiQL, you can easily interact with DynamoDB tables and run ad hoc queries.
- Example Commands:
  - aws dynamodb execute-statement --statement "INSERT INTO Music \\\n VALUE \\\n {'Artist':'Acme Band','SongTitle':'PartiQL Rocks'}
- Where we can use PartiQL in Amazon DynamoDB ?
  - The DynamoDB console
  - The NoSQL Workbench
  - The AWS Command Line Interface (AWS CLI)
  - The DynamoDB APIs



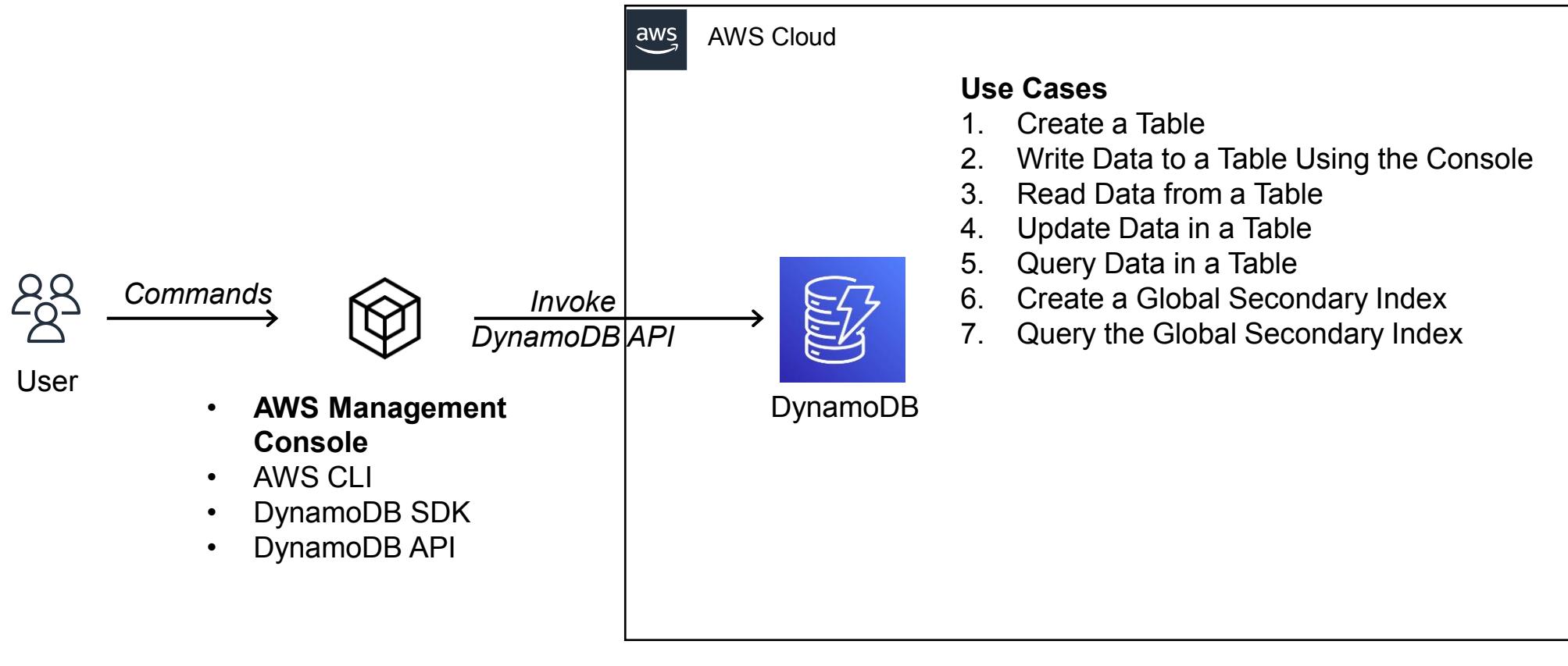
# Accessing DynamoDB with Different Ways



# Amazon DynamoDB Walkthrough with AWS Management Console

→ DEMO - Amazon DynamoDB Walkthrough with AWS Management Console

# Amazon DynamoDB Walkthrough with AWS Management Console



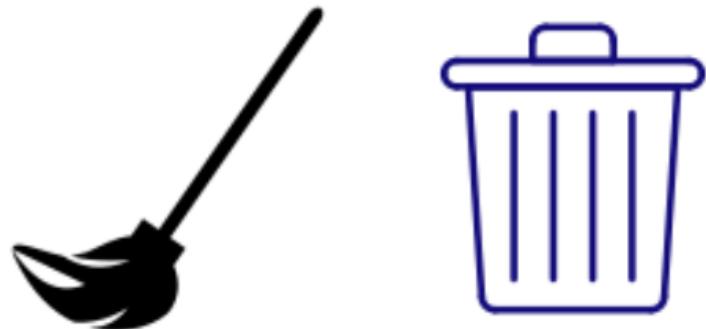
Npm  
packages



Mehmet Ozkaya

# Clean up Resources

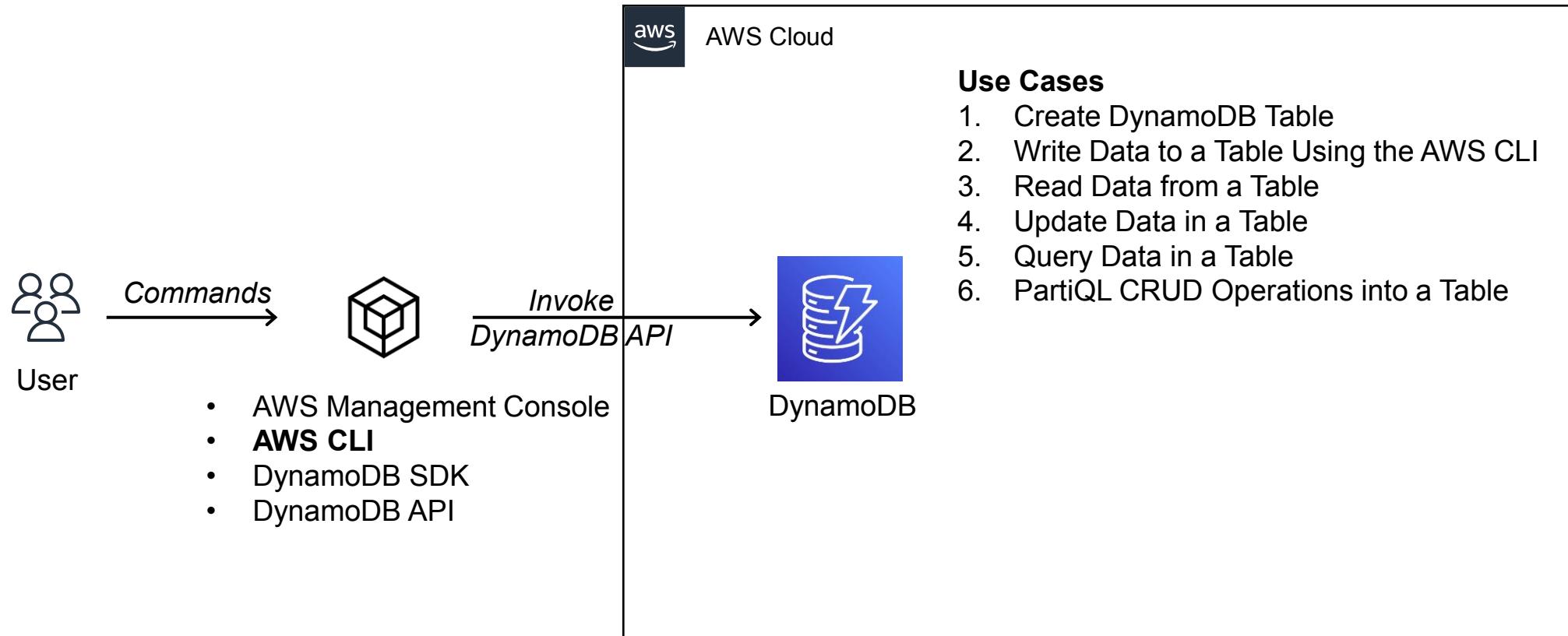
- Delete AWS Resources that we create during the section.



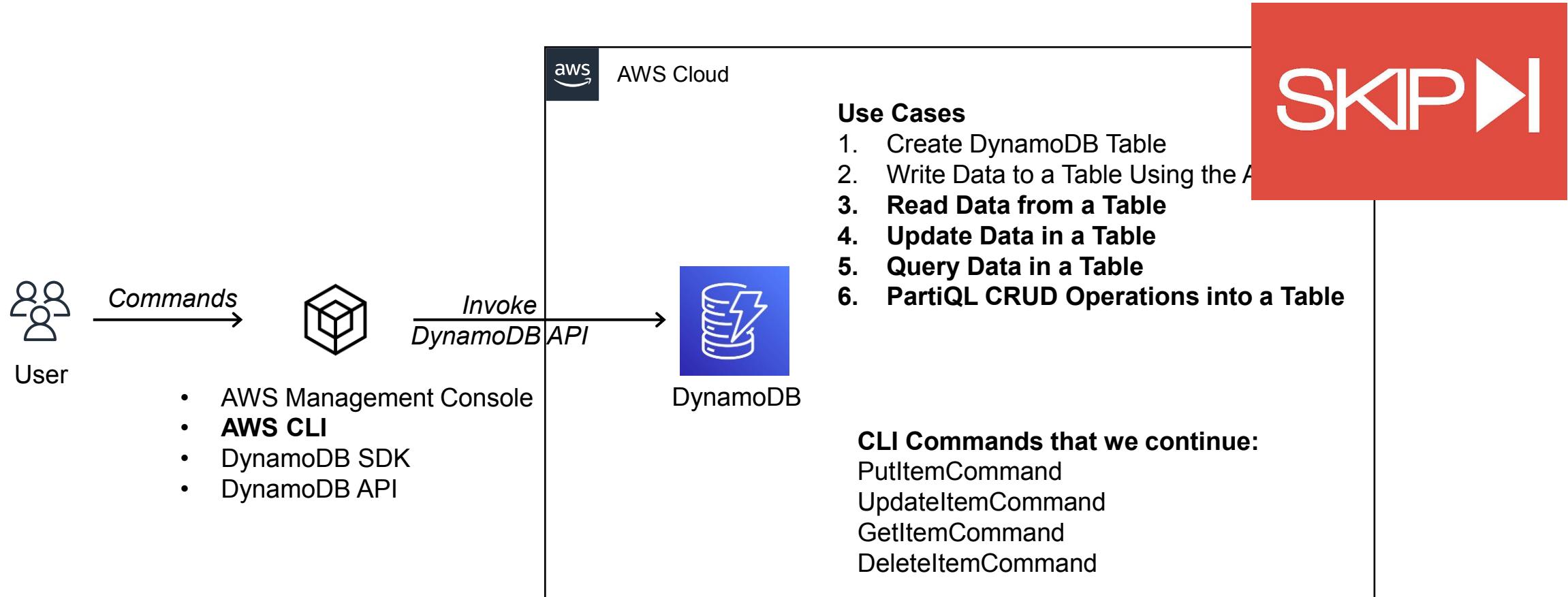
# Amazon DynamoDB Walkthrough with AWS CLI

→ DEMO - Amazon DynamoDB Walkthrough with AWS CLI

# Amazon DynamoDB Walkthrough with AWS CLI

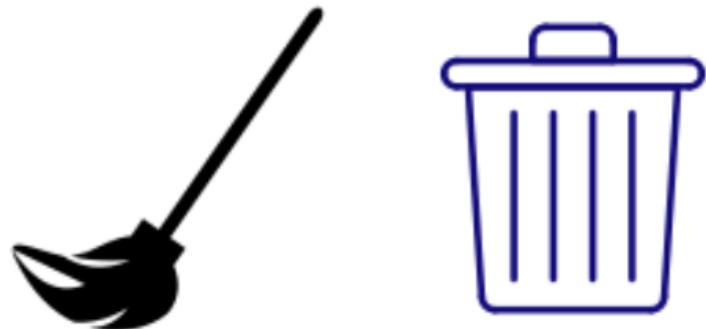


# Amazon DynamoDB Walkthrough with AWS CLI



# Clean up Resources

- Delete AWS Resources that we create during the section.

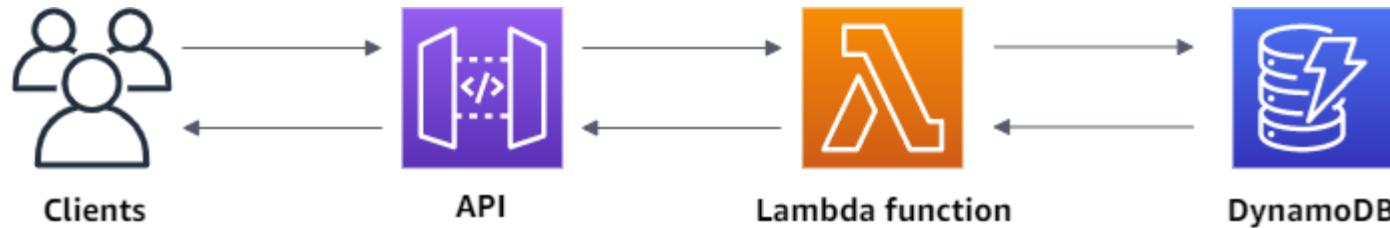


# Building RESTful Microservices with AWS Lambda, API Gateway and DynamoDB

→ Building RESTful Microservices with AWS Lambda, API Gateway and  
DynamoDB

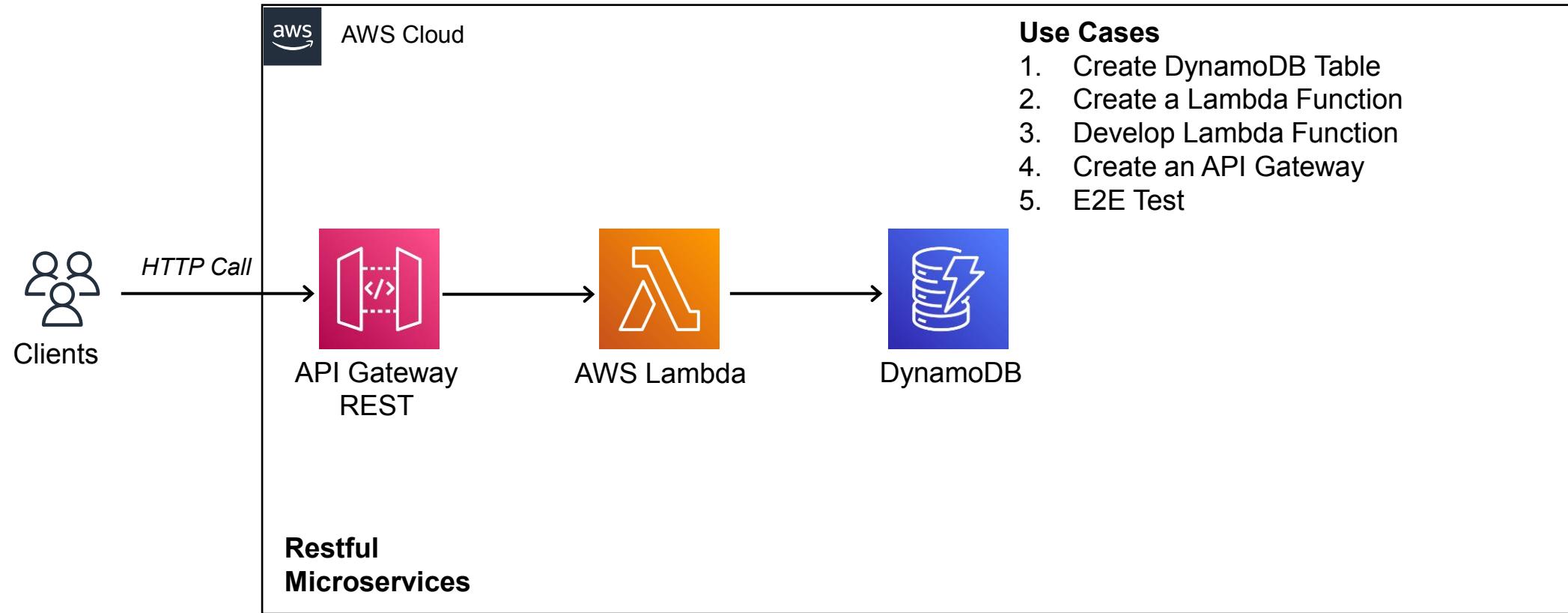
# RESTful Microservices with AWS Lambda, Api Gateway and DynamoDb

- Create a Serverless API that creates, reads, updates, and deletes items from a **DynamoDB table**.
- Create a **DynamoDB table** using the DynamoDB console.
- Create a **Lambda function** using the AWS Lambda console.
- Create an **REST API** using the API Gateway console. Lastly, we test your API.



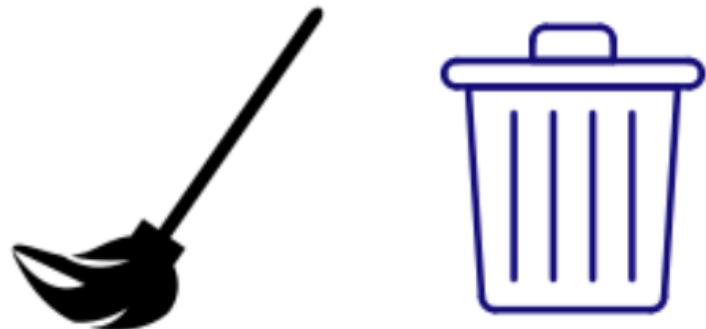
- Clients **send request** to our **microservices** by making HTTP API calls.
- Amazon **API Gateway** hosts RESTful HTTP requests and responses to customers.
- AWS **Lambda** contains the business logic to process **incoming API calls** and leverage DynamoDB as a persistent storage.
- Amazon DynamoDB **persistently stores** **microservices** data and scales based on demand.

# Hands-on Lab: Building RESTful Microservices with AWS Lambda, API Gateway and DynamoDB



# Clean up Resources

- Delete AWS Resources that we create during the section.

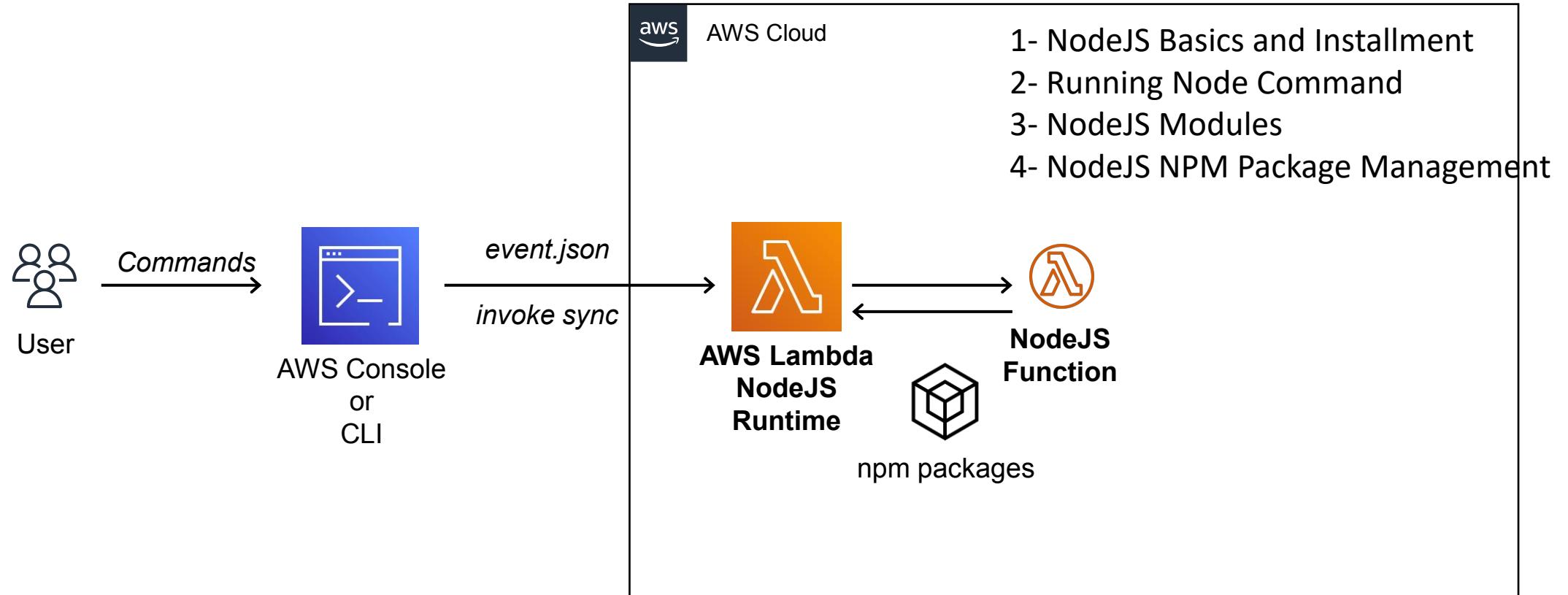


# NodeJS Basics - Recap

## NodeJS

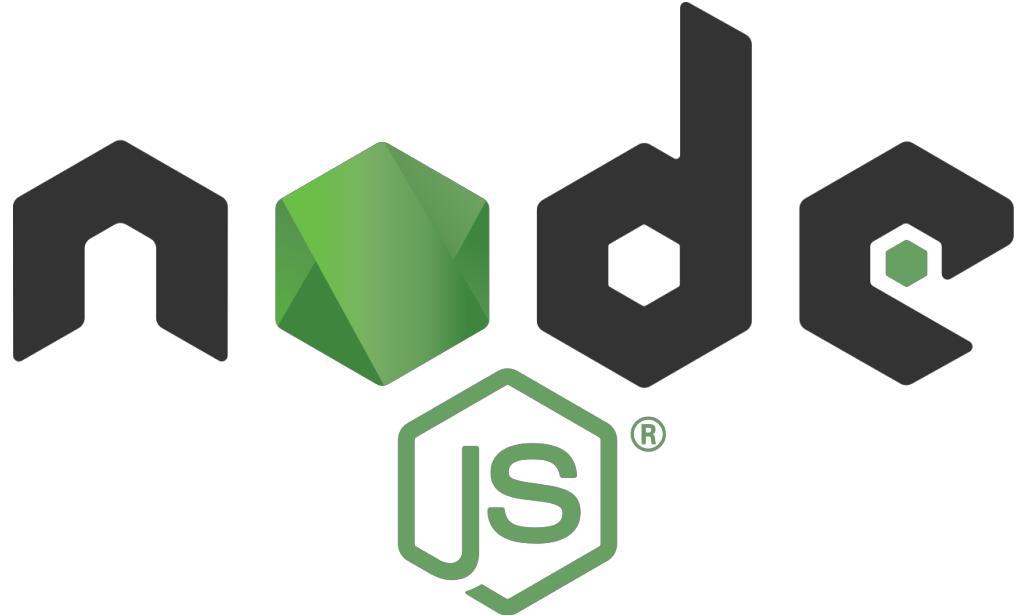
→ Understanding NodeJS fundamentals that we use during the course.

# Why we need to know NodeJS Basics ? Skip or Continue ?



# NodeJS Basics and Installment

- Node.js is an **open-source, cross-platform, back-end JavaScript runtime environment** that runs on the V8 engine and executes JavaScript code outside a web browser.
- Develop our **lambda microservices** with NodeJS runtime.
- [Installing or updating the latest version of the Node JS](#)
- AWS Lambda uses Node.js (>= 14.13.0).
- Download **LTS**



# AWS SDK for JavaScript - Developer Guide for SDK Version 3

→ Develop Lambda Functions with AWS SDK for JavaScript - Developer Guide for SDK Version 3

# What is AWS SDK ?

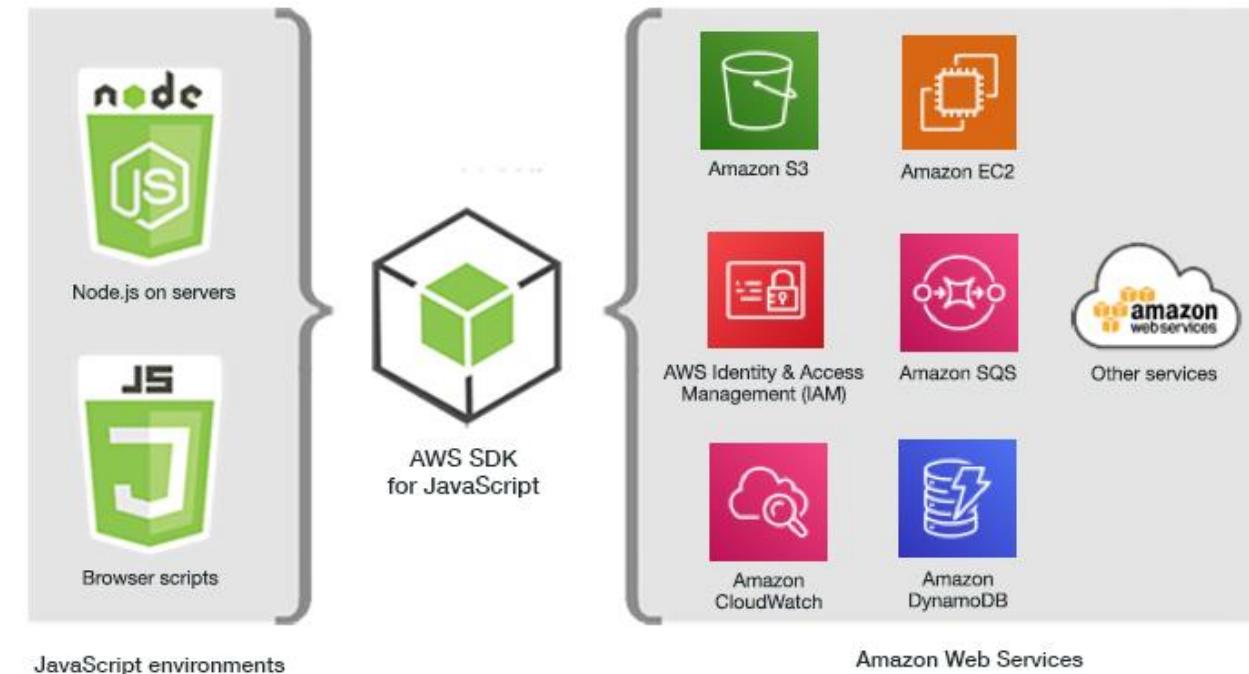
- **AWS SDK - Software Development Kit**  
Simplifies use of AWS Services by providing a set of libraries that are consistent and familiar for developers.
- Tools for developing and managing applications on AWS
- Use **AWS-SDK** in our microservices codes when interacting with AWS DynamoDB, EventBridge and SQS.



[https://docs.amazonaws.cn/en\\_us/sdk-for-javascript/v2/developer-guide/s3-examples.html](https://docs.amazonaws.cn/en_us/sdk-for-javascript/v2/developer-guide/s3-examples.html)

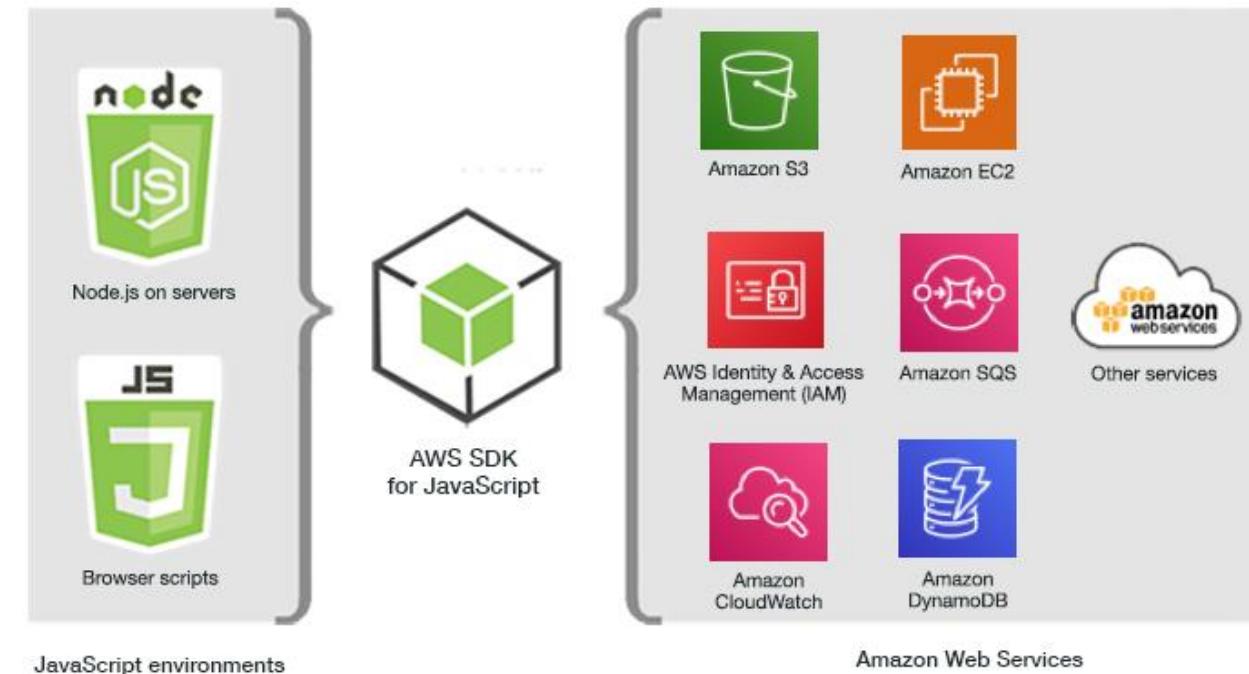
# AWS SDK for JavaScript Version 3 and Lambda Interactions

- Product Lambda Microservice Function is chosen the runtime as a **NodeJS** function.
- Use **AWS SDK for JavaScript Version 3**.
- First-class **TypeScript** support and a new middleware stack
- We have 2 main resources.
- [AWS SDK for JavaScript - Developer Guide for SDK Version 3](#)
- [AWS SDK for JavaScript v3 API Reference Guide](#)



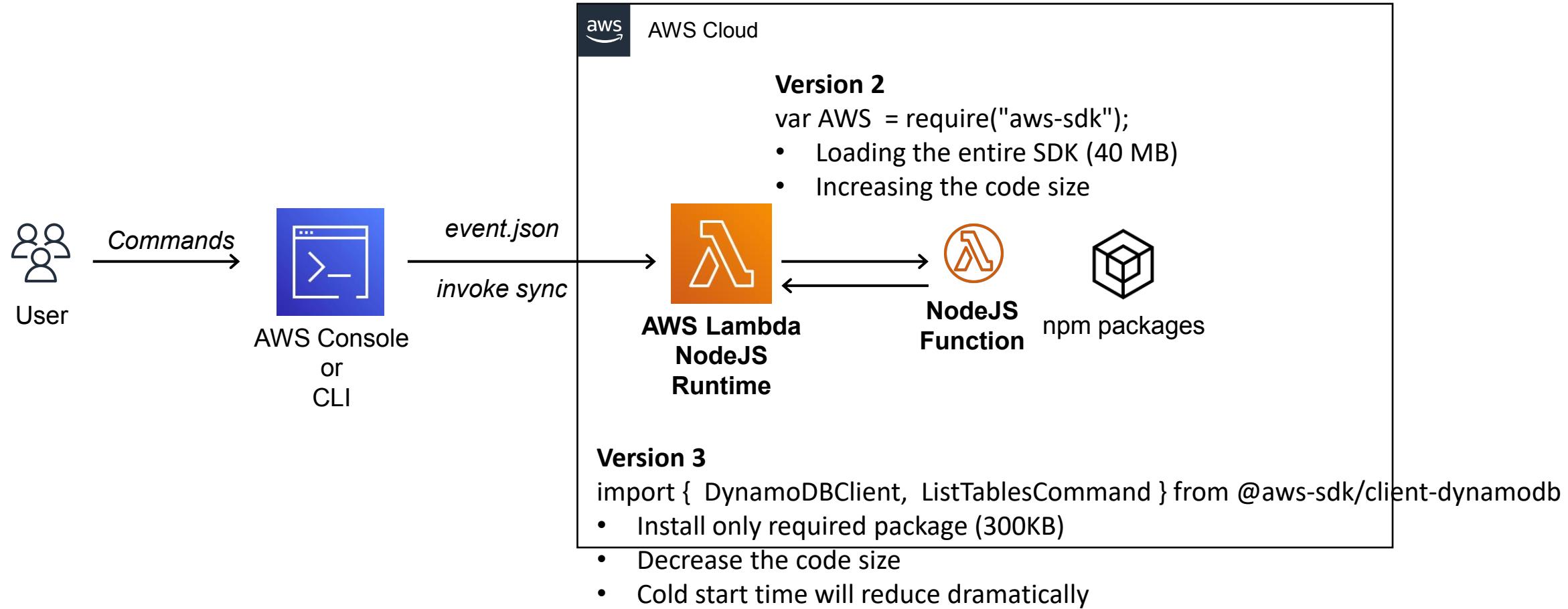
# What's new in AWS SDK for JavaScript V3 and Cold Start Benefit

- The AWS SDK for JavaScript v3 is a rewrite of v2 with some great new features. Includes many frequently requested features, such as a first-class TypeScript support and a new middleware stack.
- Modularized packages;** Users can now use a separate package for each service.
- New middleware stack;** Users can now use a middleware stack to control the lifecycle of an operation call.
- AWS SDK is written in TypeScript, which has many advantages, such as static typing.



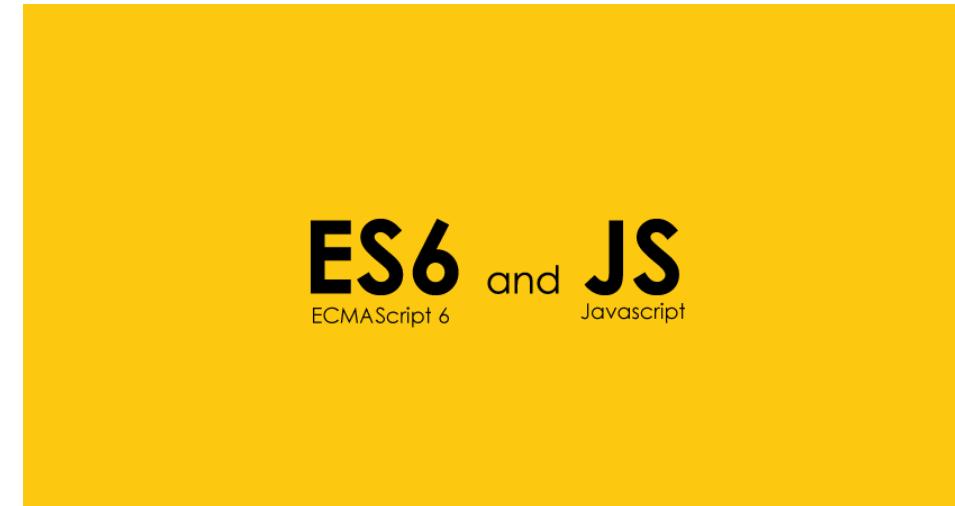
<https://aws.amazon.com/blogs/developer/modular-aws-sdk-for-javascript-is-now-generally-available/>

# AWS SDK JavaScript Version 3 - Modularized packages



# Important - ECMAScript 6 (ES6) Requirements

- We will use **AWS SDK for JavaScript v3** when interacting AWS resources from our NodeJS Lambda functions. Understand ECMAScript 6 (ES6) and its requirements.
- In the course we use **AWS SDK JS V3** and develop our codes **ECMAScript 6 (ES6)** standards.
- ES6 brings new syntax and new features to make your code more modern and readable, and do more.
- 2 type of module approaches in NodeJS
  - CommonJS
  - ECMAScript 6 (ES6)
- ES5 example
  - ```
const { DynamoDBClient, BatchExecuteStatementCommand } = require("@aws-sdk/client-dynamodb");
```
- ES6+ example
 - ```
import { DynamoDBClient, BatchExecuteStatementCommand } from "@aws-sdk/client-dynamodb";
```



## ES5 vs ES6 Differences :

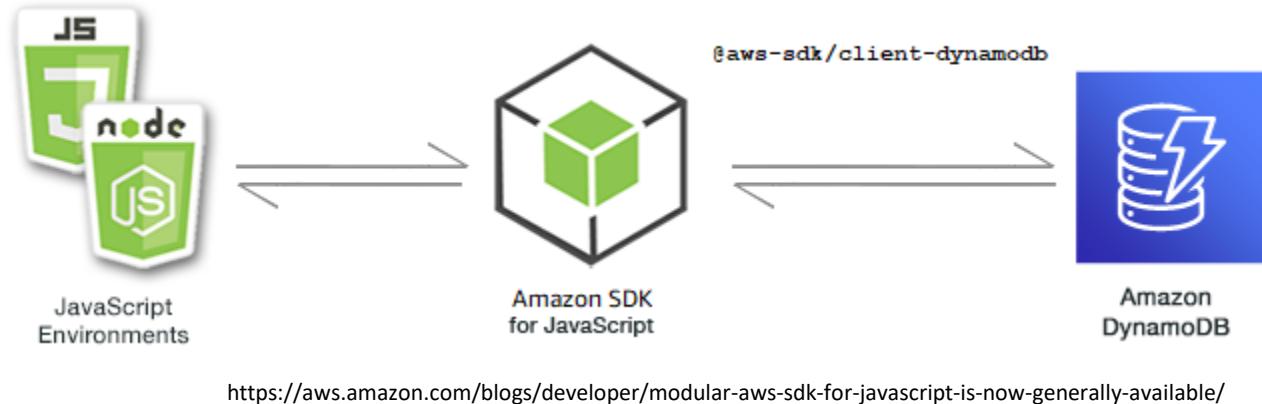
- import vs const
- export vs module.exports
- add "type" : "module" into the package.json

# Amazon DynamoDB - Developing with AWS SDK

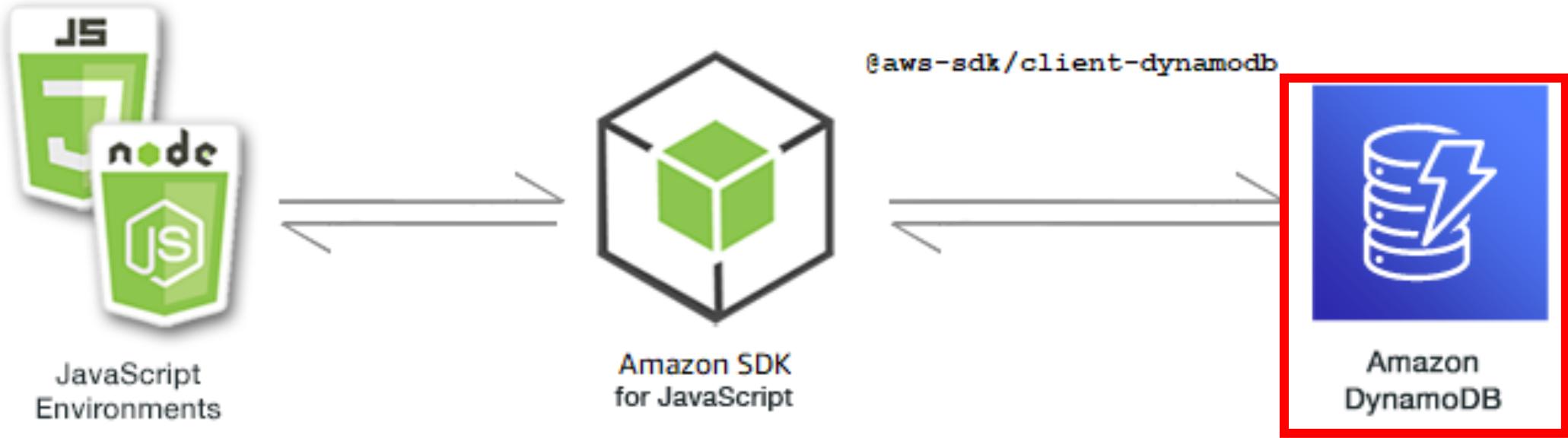
→ Amazon DynamoDB - Developing with AWS SDK - Programmatic Access w/ Serverless APIs

# Amazon DynamoDB - Developing with AWS SDK

- Amazon DynamoDB is a fully managed NoSQL cloud database that supports both document and key-value store models.
- The JavaScript API for DynamoDB is exposed through the DynamoDB, DynamoDBStreams, and DynamoDB.DocumentClient client classes.
- Main Topics
  - Creating and using tables in DynamoDB
  - Reading and writing an item in DynamoDB
  - Reading and writing items in batch in DynamoDB
  - Querying and scanning a DynamoDB table
- We have 2 main resources.
  - [AWS SDK for JavaScript - Developer Guide for SDK Version 3](#)
  - [AWS SDK for JavaScript v3 API Reference Guide](#)

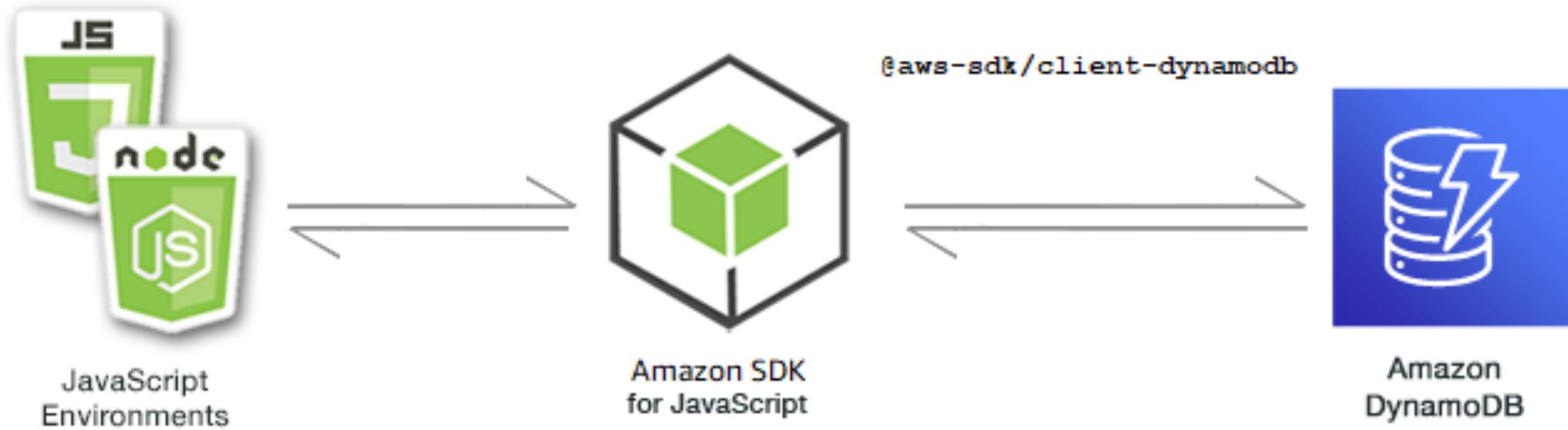


# Understanding DynamoDB Interactions - Working with Items and Attributes



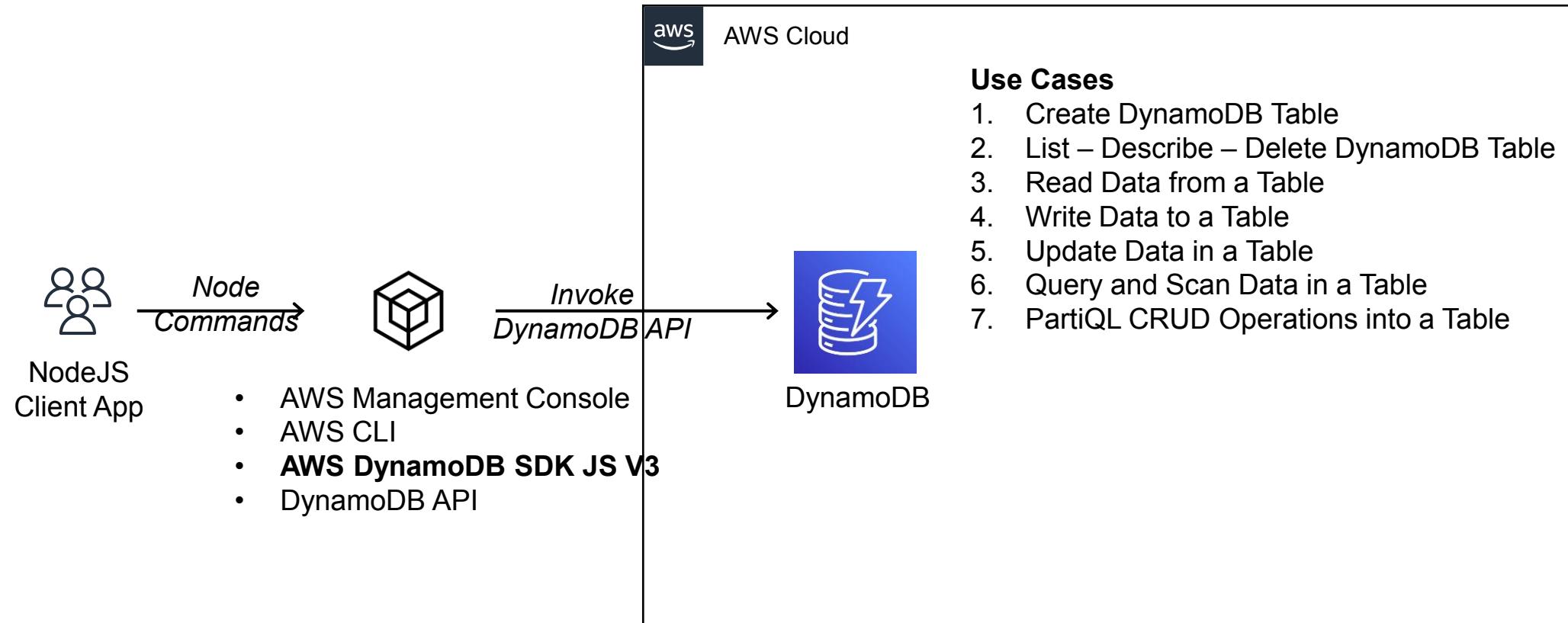
<https://aws.amazon.com/blogs/developer/modular-aws-sdk-for-javascript-is-now-generally-available/>

# Create NodeJS Project with DynamoDB SDK Packages

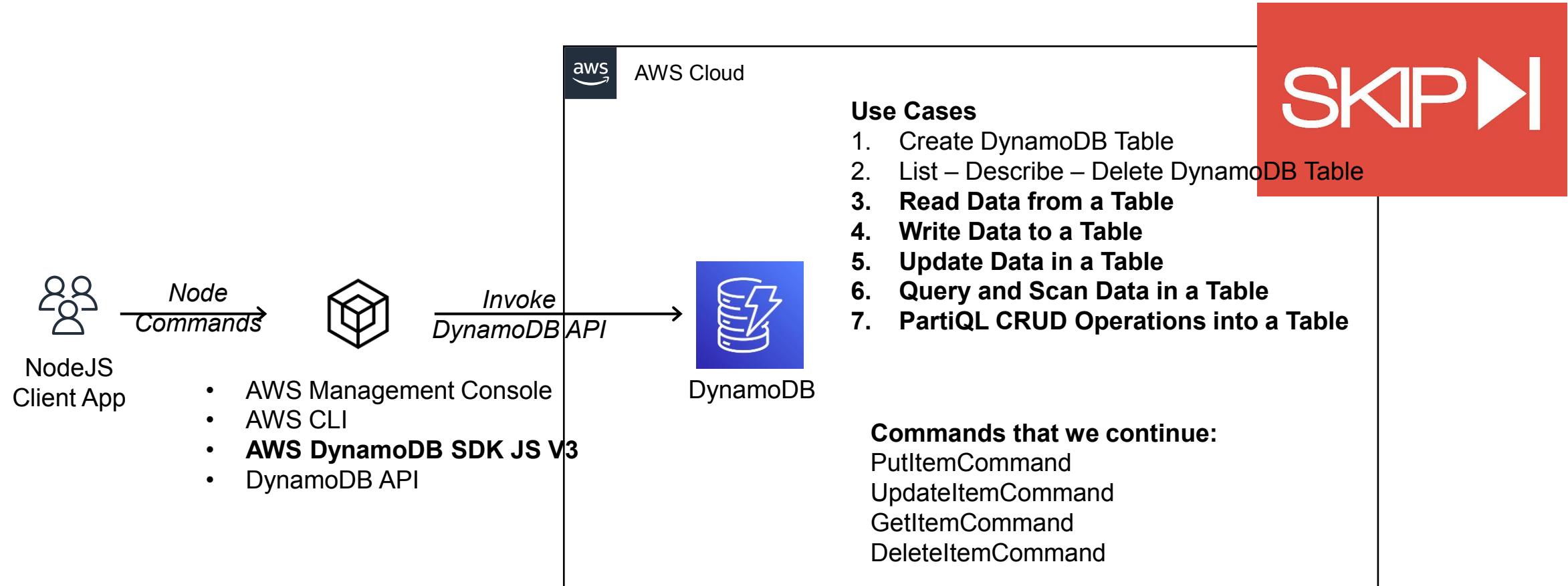


<https://aws.amazon.com/blogs/developer/modular-aws-sdk-for-javascript-is-now-generally-available/>

# Amazon DynamoDB Walkthrough with AWS SDK JS V3

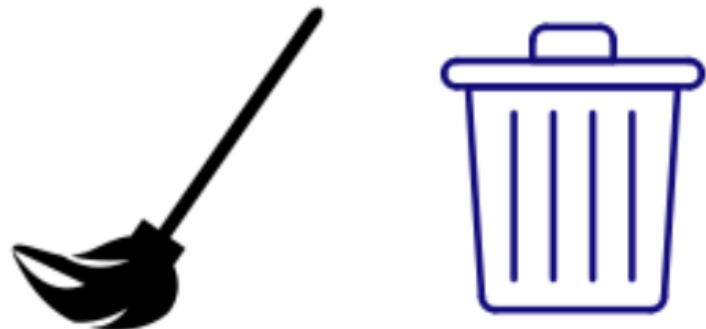


# Amazon DynamoDB Walkthrough with AWS SDK JS V3



# Clean up Resources

- Delete AWS Resources that we create during the section.

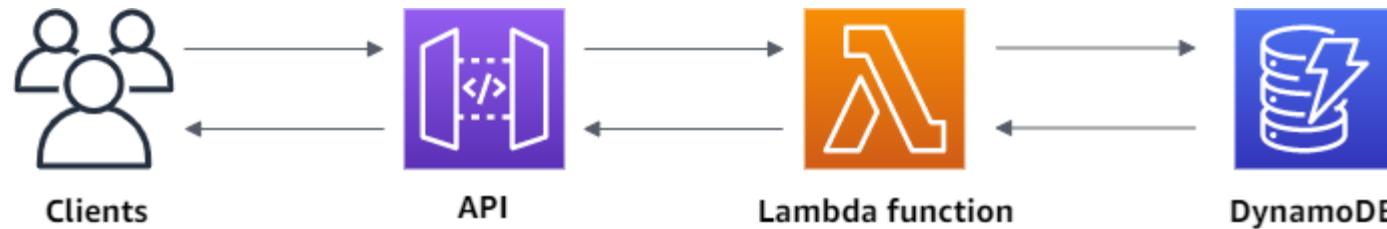


# Build CRUD RESTful Microservices with AWS Lambda, API Gateway, DynamoDB

- Hands-on Lab : Build CRUD RESTful Microservices with AWS Lambda, API Gateway, DynamoDB using Node.js AWS-SDK V3

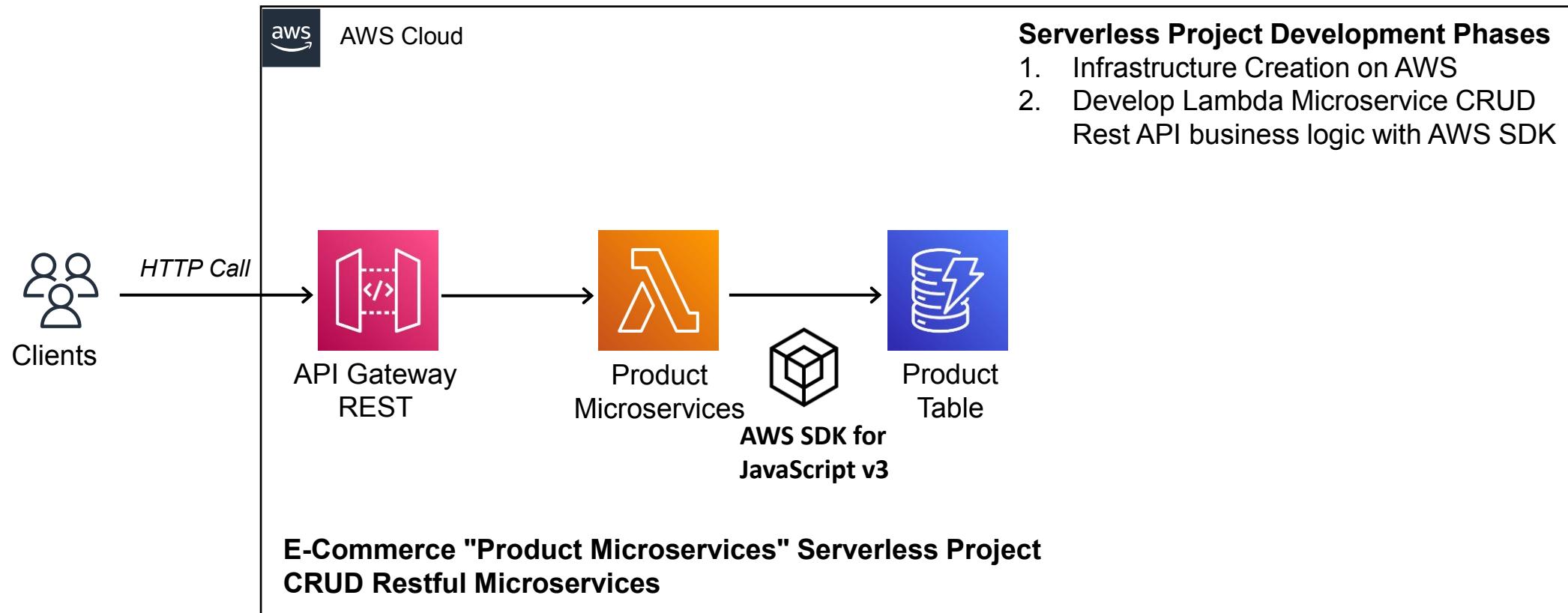
# Developing Lambda Microservices CRUD functions with AWS SDK

- Create a Serverless API that creates, reads, updates, and deletes items from a **DynamoDB table**.
- Create a **DynamoDB table** using the DynamoDB console.
- Create a **Lambda function** using the AWS Lambda console.
- Create an **REST API** using the API Gateway console. Lastly, we test your API.



- Clients **send request** to our **microservices** by making HTTP API calls.
- Amazon **API Gateway** hosts RESTful HTTP requests and responses to customers.
- AWS **Lambda** contains the business logic to process **incoming API calls** and leverage DynamoDB as a persistent storage.
- Amazon DynamoDB **persistently stores** **microservices** data and scales based on demand.

# Hands-on Lab: Building RESTful Microservices with AWS Lambda, API Gateway and DynamoDB



# Serverless Project Development Phases

1

## Infrastructure Creation on AWS

Create API Gateway, Lambda Function and DynamoDB table on AWS Cloud - Also we can automate this part with IaC using CDK in the last sections but now we will create infrastructure with console or cli

2

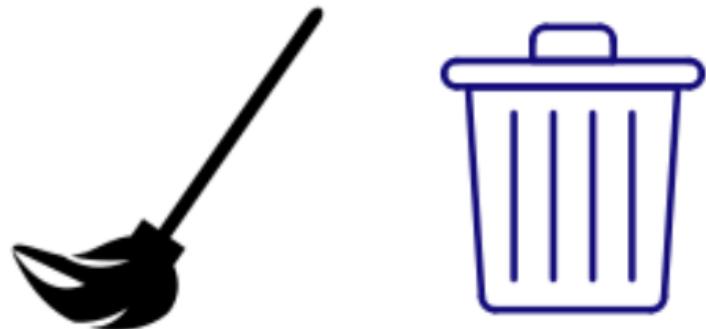
## Develop Lambda Microservice CRUD Rest Api business logic with AWS SDK

Use AWS SDK JS v3 with ES6 standards to implement crud functions into lambda function.



# Clean up Resources

- Delete AWS Resources that we create during the section.

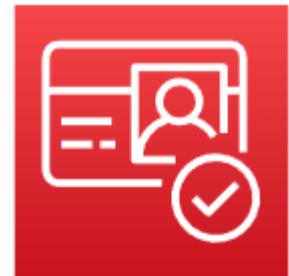


# Amazon Cognito - Authentication and Authorization

→ Learn Amazon Cognito for Authentication and Authorization in Serverless Applications

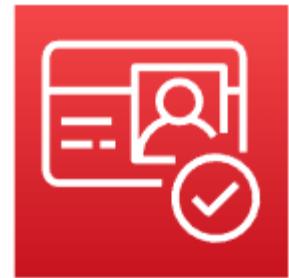
# What is Amazon Cognito ?

- **Amazon Cognito** provides authentication, authorization, and user management for apps. Users can sign in directly with a user name and password, or third party such as Facebook, Amazon, Google or Apple.
- **Amazon Cognito** provides to add user sign-up, sign-in, and access control to web and mobile applications quickly and easily.
- Supports sign-in with **social identity providers**, like Apple, Facebook, Google, and Amazon, and enterprise identity providers via **SAML 2.0** and **OpenID Connect**.
- Provide controls user **authentication** and access for mobile applications on internet-connected devices. **Accelerate the mobile application development** process.



# What is Amazon Cognito used for ?

- **Amazon Cognito** enables simple, secure user authentication, authorization and user management for web and mobile apps.
- **Amazon Cognito** provides to add user sign-up, sign-in, and access control to web and mobile applications quickly and easily.
- Easily add user sign-up, sign-in and access control to their apps with its built-in user interface (UI) and easy configuration
- Federate identities from social identity providers
- Synchronize data across multiple devices and applications
- Provide secure access to other AWS services
- Developers can focus on creating application development



# Amazon Cognito Main Features

- **Secure and Scalable Identity Server**

Amazon Cognito User Pools provide a secure identity server that scales to millions. User Pools can easily set up without provisioning any infrastructure.

- **Social Identity Federation**

Users can sign in with social identity providers such as Apple, Google, Facebook, and Amazon.

- **Standards-based authentication**

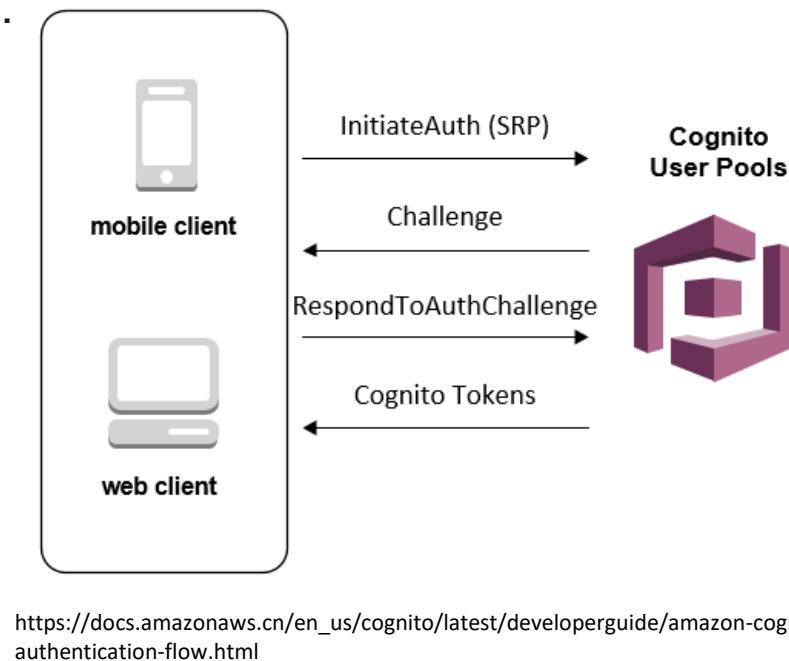
Supports identity and access management standards, for example; OAuth 2.0, SAML 2.0, and OpenID Connect.

- **Access for AWS resources**

Control access to AWS resources from our app. Define roles and map users that can access only the resources that are authorized for each user.

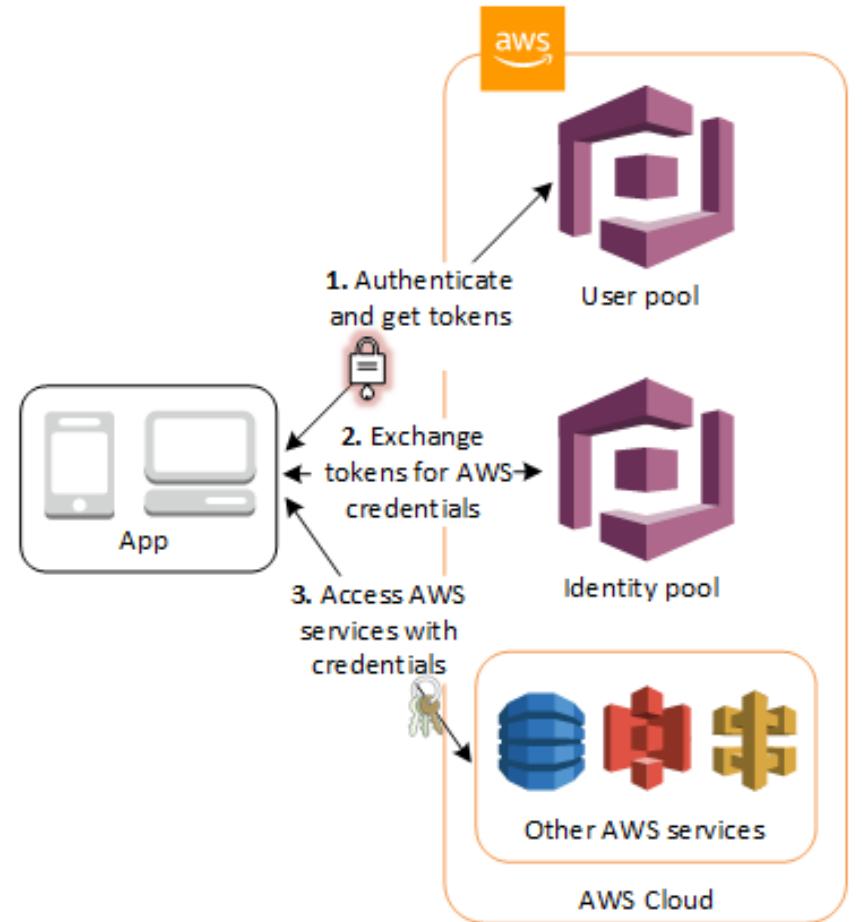
- **Easy integration**

Built-in UI and easy configuration for federating identity providers supports integrate Amazon Cognito.



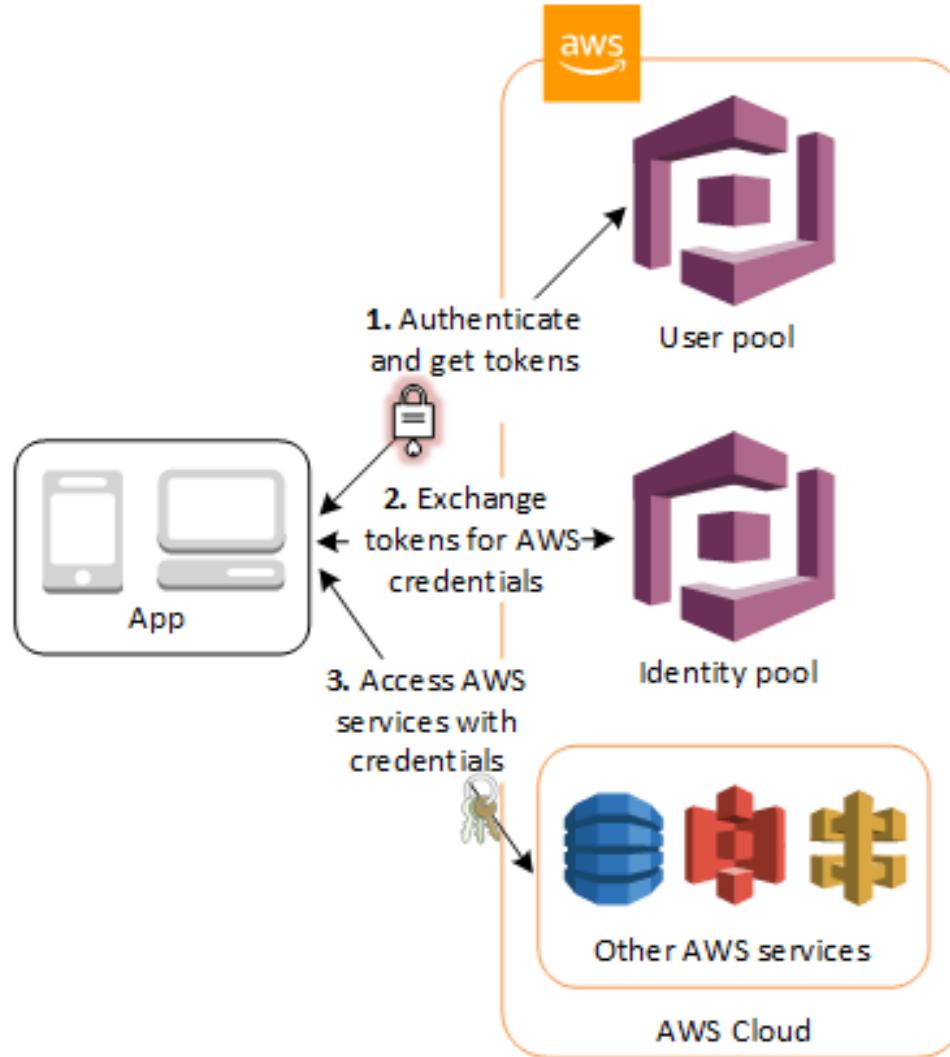
# Amazon Cognito Types - User Pools and Identity Pools

- There are **two main components** of Amazon Cognito;
  - User pools
  - Identity pools.
- **User pools** are user directories that provide sign-up and sign-in options for app users.
- Users can sign in to your web or mobile app through Amazon Cognito, or federate through a third-party identity providers.
- The user pool have a directory profile that you can access through an SDK.
- **Identity pools** enable you to grant users access to other AWS services. We can use identity pools and user pools separately or together.
- Users can obtain temporary AWS credentials to access AWS services, such as Amazon S3 and DynamoDB. Identity pools support anonymous guest users.



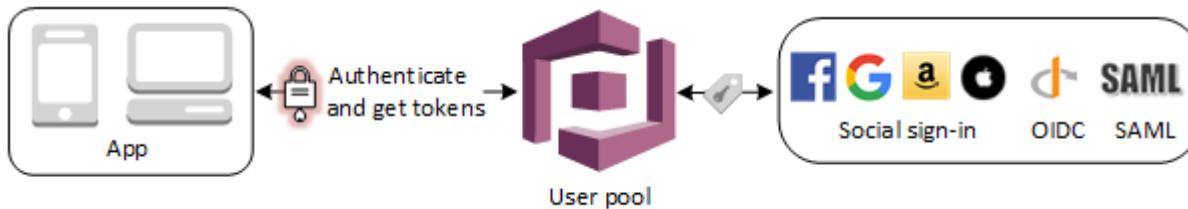
<https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html>

# Amazon Cognito - How it Works ?

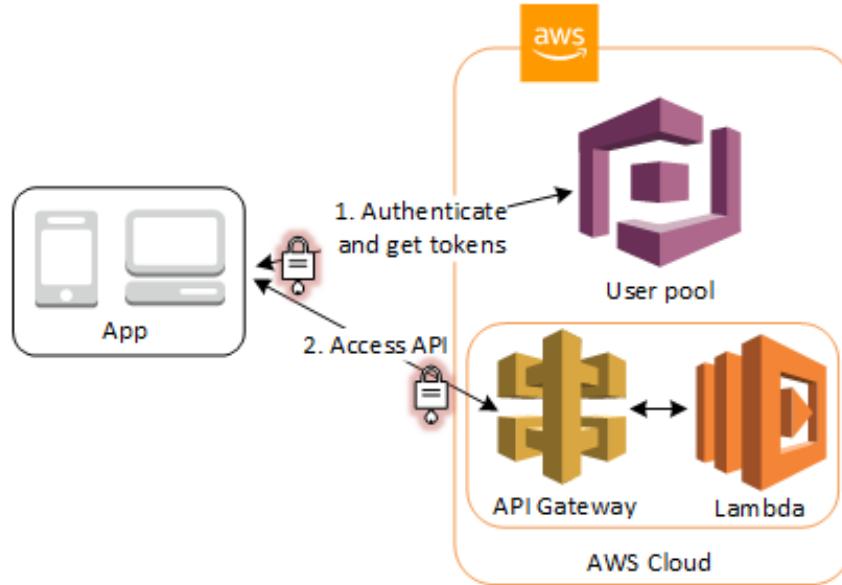


# Amazon Cognito Use Cases

## 1. Authenticate with User Pool



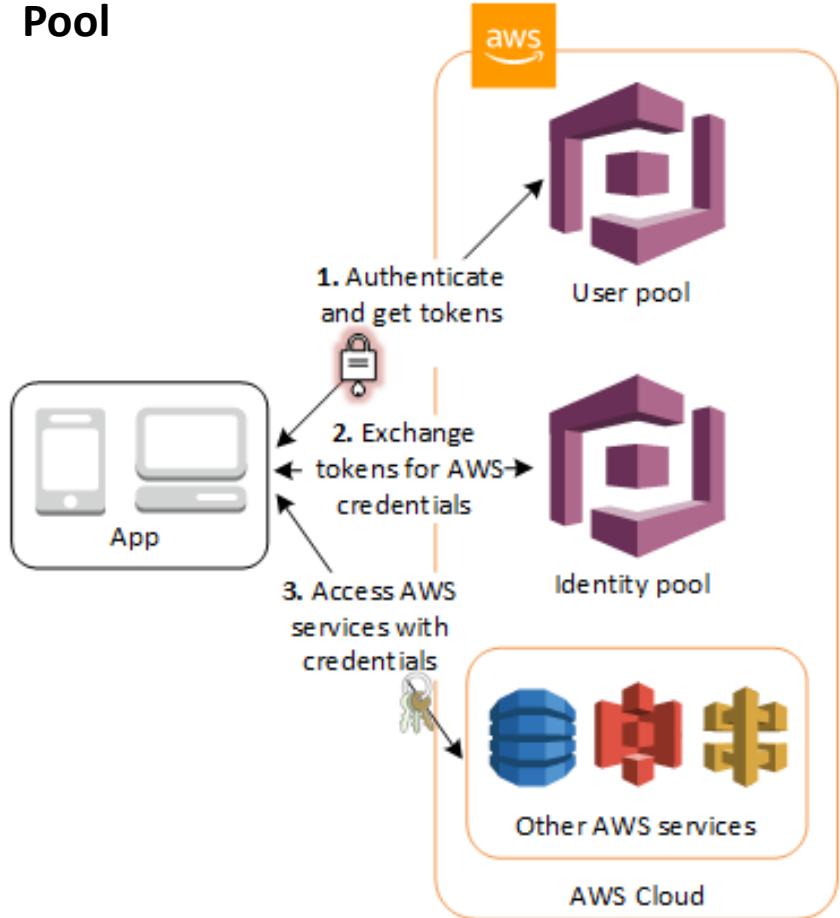
## 2. Cognito User Pool to Access Resources with API Gateway and Lambda



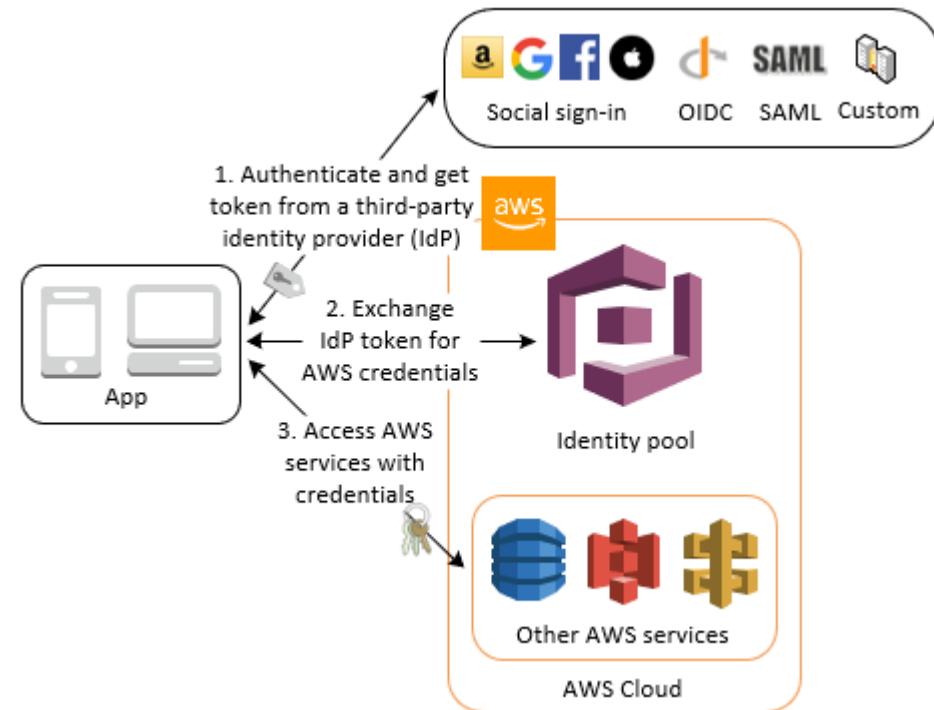
<https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html>

# Amazon Cognito Use Cases Part 2

## 3. Access AWS services with User Pool and Identity Pool



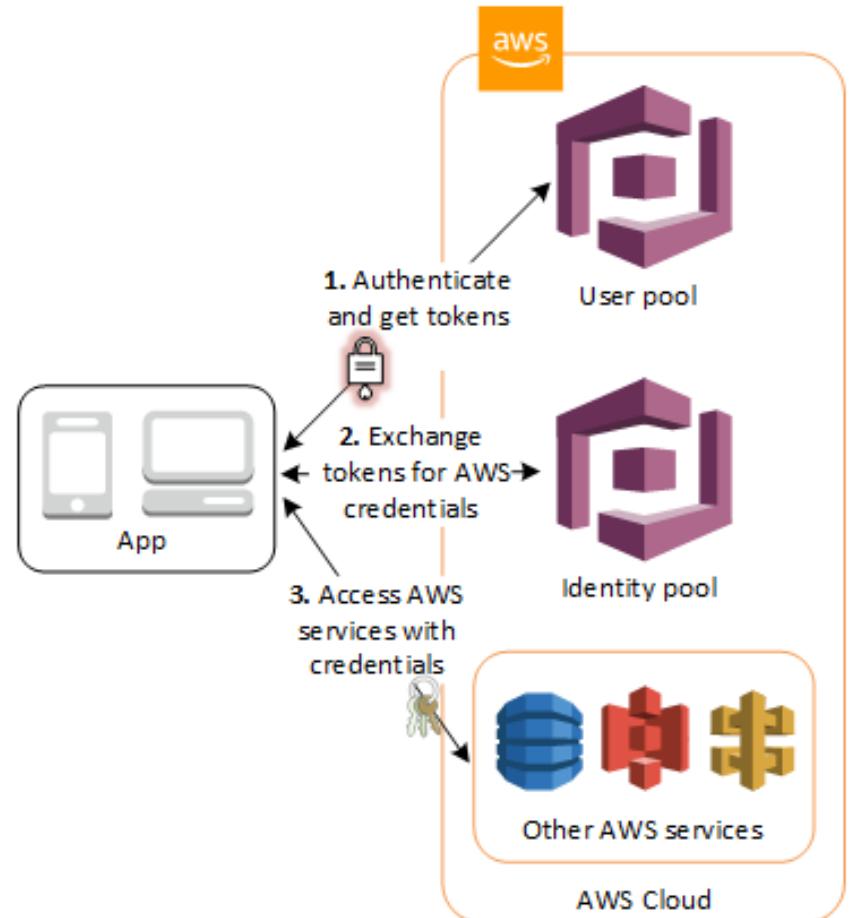
## 4. Authenticate with Third party and Access AWS Services with Identity Pool



<https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html>

# Amazon User Pools

- **User pool** is a **user directory** in Amazon Cognito. **User pools** are user directories that provide sign-up and sign-in options for app users.
- Application users can sign in either directly through a user pool, or federate through a third-party identity provider (IdP).
- User pool have a directory profile that you can access through a Software Development Kit (SDK).
- **User Pools provides;**
  - Sign-up and sign-in services.
  - A built-in, customizable web UI to sign in users.
  - Social sign-in with Facebook, Google, Amazon as well as sign-in with SAML identity providers from your user pool.
  - User directory management and user profiles.
  - Security features such as multi-factor authentication (MFA)
  - Customized workflows and user migration through AWS Lambda triggers.

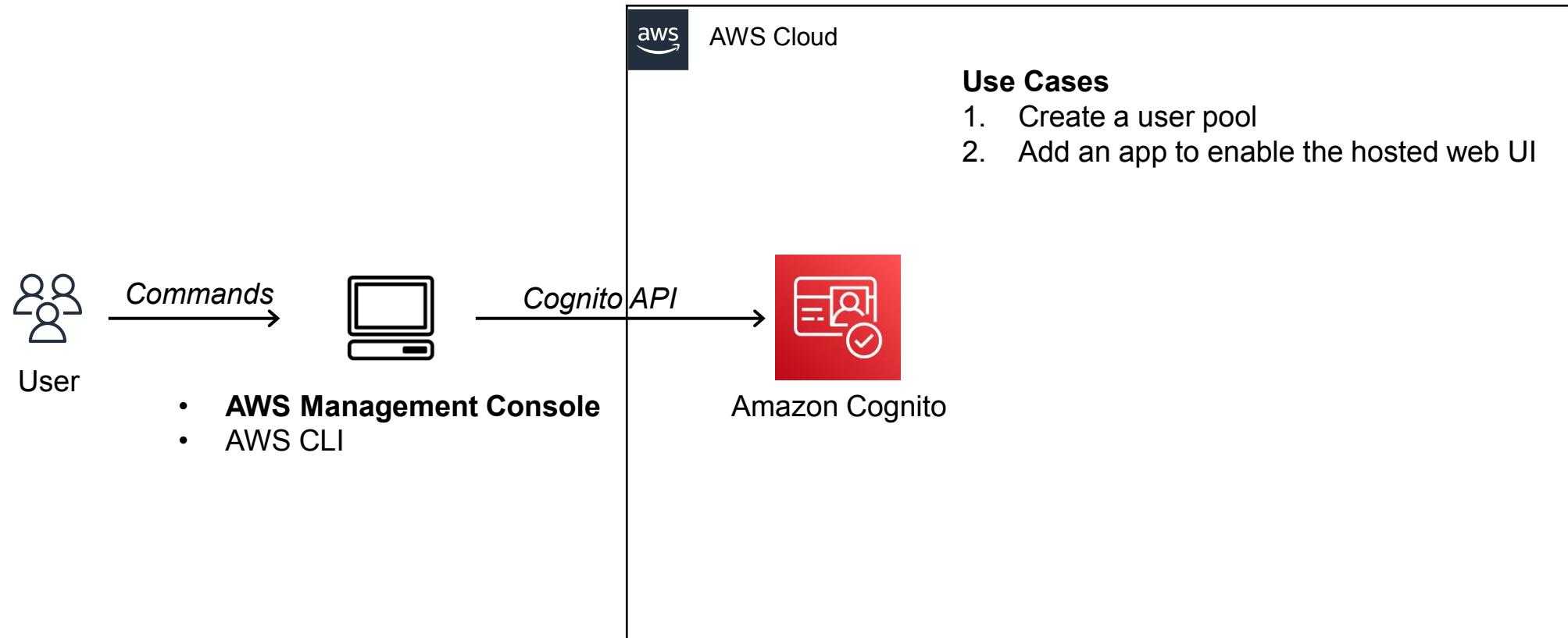


<https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html>

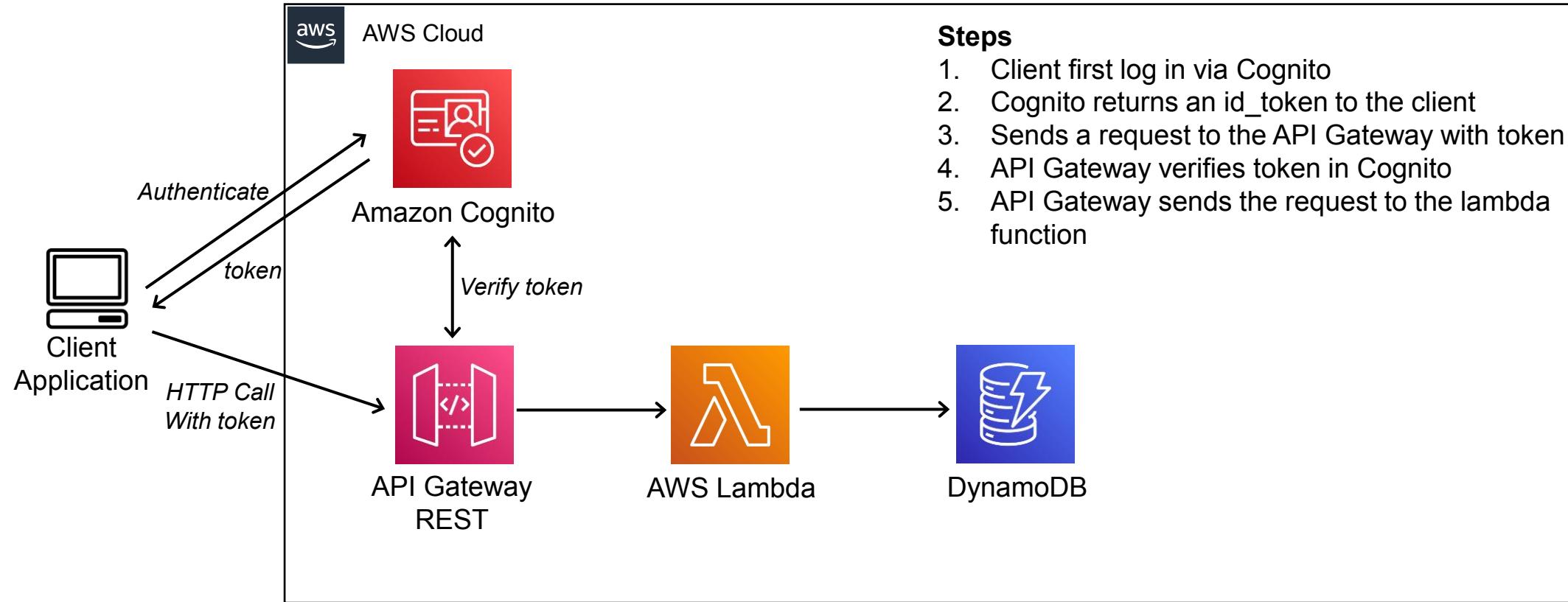
# Amazon Cognito - Walkthrough with AWS Management Console

→ DEMO - Amazon Cognito - Walkthrough with AWS Management  
Console

# Amazon Cognito - Walkthrough with AWS Management Console

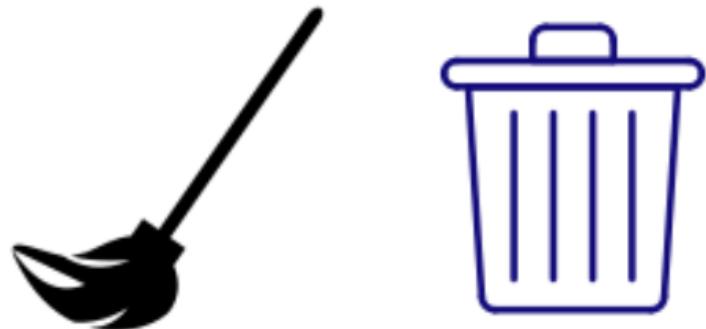


# Hands-on Lab: Secure API Gateway with Amazon Cognito User Pools



# Clean up Resources

- Delete AWS Resources that we create during the section.



# Amazon S3 - Cloud Object Storage

→ Learn Amazon S3 - Cloud Object Storage

# What is Amazon S3 ?

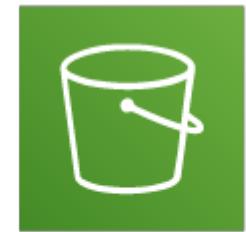
- **Cloud storage** is a web service where data can be stored, accessed, and quickly backed up by users on the internet. It is more reliable, scalable, and secure than traditional on-premises storage systems.
- **Amazon S3** stands for Amazon Simple Storage Service is a Object storage built to retrieve any amount of data from anywhere. Designed for durability.
- Provides object storage, which is built for **storing** and **recovering** any amount of data from anywhere over the internet.
- **Enables users** to **store** and **retrieve** any amount of data at any time or place, giving developers access to highly scalable, reliable, fast and inexpensive data storage.
- **Use Amazon S3** to store and protect any amount of data for a range of use cases, like data lakes, websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics.



Amazon Simple  
Storage Service (S3)

# Amazon S3 Benefits and Features

- **Storage Classes;** S3 offers a range of storage classes designed for different use cases. S3 Standard, S3 Standard-IA, S3 Galcier.
- **Storage Management;** S3 has storage management features that we can use to manage costs, meet regulatory requirements, reduce latency.
- **Access Management;** S3 provides features for auditing and managing access to buckets and objects.
- **Data Processing;** To transform data and trigger workflows to automate a variety of other processing activities at scale.
- **Storage logging and monitoring;** S3 provides logging and monitoring tools that you can use to monitor and control how your Amazon S3 resources are being used. S3 also offers features to gain visibility into your storage usage.
- **Strong Consistency;** S3 provides strong read-after-write consistency for PUT and DELETE requests of objects in your Amazon S3 bucket in all AWS Regions.



Amazon  
S3

# Amazon S3 Benefits

- **Durability;** S3 provides near to 100 percent durability.
- **Low cost;** store data in a range of “storage classes.” S3 Galcier provide lowest cost.
- **Scalability;** S3 provides features for auditing and managing access to buckets and objects.
- **Availability;** S3 offers 99.99 percent availability of objects
- **Security;** S3 offers a range of access management tools and encryption features that provide top-notch security.
- **Flexibility;** S3 is ideal for a wide range of uses like data storage, data backup, software delivery, data archiving, disaster recovery, website hosting, mobile applications, IoT devices.
- **Data Transfer;** S3 offers simple data transfer and easy to use for transfer.



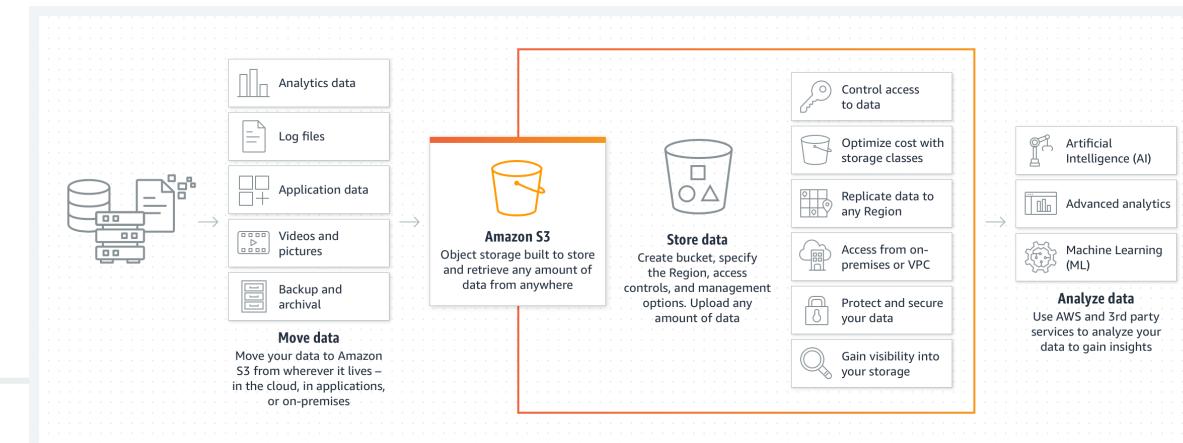
# How Amazon S3 Work ?

- **Amazon S3** provide to optimize costs, organize data, and configure access controls to meet specific business requirements With cost-effective storage classes and easy-to-use management feature.
- S3 focuses on **two key** components: **Buckets and Objects**
- **Amazon S3 Objects**; S3 is an object storage service that stores data as objects within buckets. Creates a bucket; the bucket stores objects in the cloud.
- Each object has a key, which is the unique identifier for the object within the bucket.
- **Amazon S3 Buckets**; bucket is a container for objects. objects are saved in the buckets. Create bucket and specify bucket name and AWS Region. Then, upload data to the bucket as objects in Amazon S3.
- Amazon S3 buckets are **globally unique**. No other AWS account in the same region can have the same bucket names.



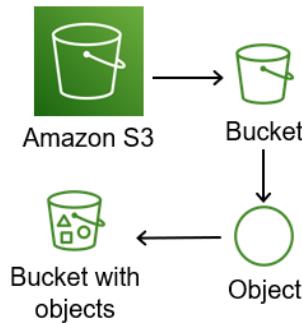
# How Amazon S3 Work ? - Part 2

- **Buckets** and the **objects** are **private** and can be accessed only if you explicitly grant access permissions. Use bucket policies, AWS Identity and Access Management (IAM) policies, access control lists (ACLs), and S3 Access Points.
- We can create up to **100 buckets** in each of your AWS cloud accounts, with no limit on the number of objects you can store in a bucket.
- When create a bucket, we have the ability to **choose the AWS region** to store it in. it's best practice to select a region that's **geographically closest** to you.
- **Configure to support** your specific use case. Use **S3 Versioning** to keep multiple versions of an object in the same bucket in order to to restore objects that are accidentally deleted or overwritten.



# Amazon S3 Core Concepts - Buckets and Objects

- **Buckets;** container for objects stored in Amazon S3. Store any number of objects in a bucket and can have up to 100 buckets in your account. Every object is contained in a bucket.
- **Objects;** consist of object data and metadata with name-value pairs. These pairs includes the date last modified, and standard HTTP metadata, such as Content-Type.
- **Keys;** object key is the unique identifier for an object within a bucket. Every object in a bucket has exactly one key. The combination of a bucket, object key, and version ID uniquely identify each object.
- **S3 Versioning;** use S3 Versioning to keep multiple variants of an object in the same bucket.
- **Bucket policy;** is a resource-based AWS IAM policy that use to grant access permissions to bucket and the objects in it.
- **S3 Access Points;** named network endpoints with dedicated access policies that describe how data can be accessed using that endpoint.
- **Access control lists (ACLs);** use ACLs to grant read and write permissions to authorized users for individual buckets and objects.



# Amazon S3 Use Cases



data storage  
data archiving  
application hosting for deployment  
software delivery  
data backup  
disaster recovery (DR)  
running big data analytics tools  
data lakes  
mobile applications  
internet of things (IoT) devices  
media hosting for images, videos and music files  
website hosting

<https://aws.amazon.com/s3/>

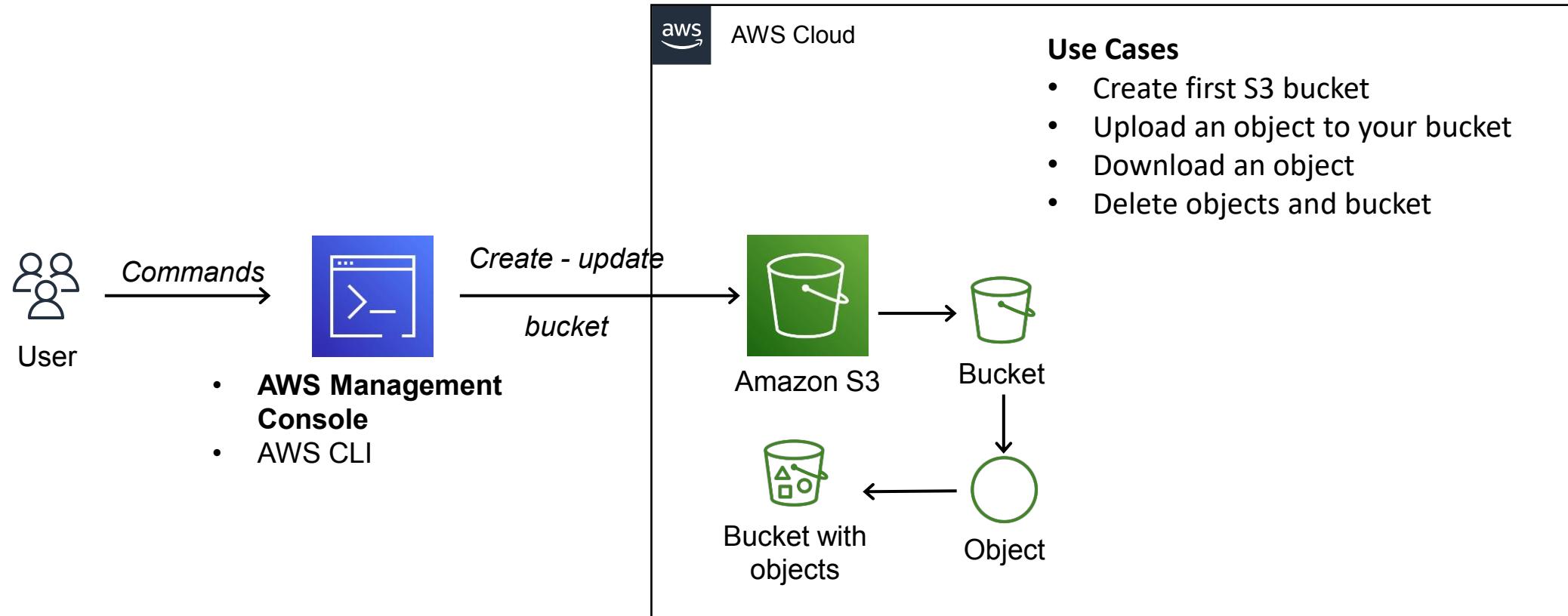


Mehmet Ozkaya

# Amazon S3 - Walkthrough with AWS Management Console

→ Getting started with Amazon S3 with AWS Management Console

# Getting started with Amazon S3 with AWS Management Console



# AWS Lambda & Serverless Course Learning Path

**1**

## Theoretical Information

AWS Service overview, core concepts, features, uses cases and general information

**2**

## Walkthrough with AWS Console

AWS Service Walkthrough with AWS Management Console performs main use cases

**3**

## Developing with AWS SDK

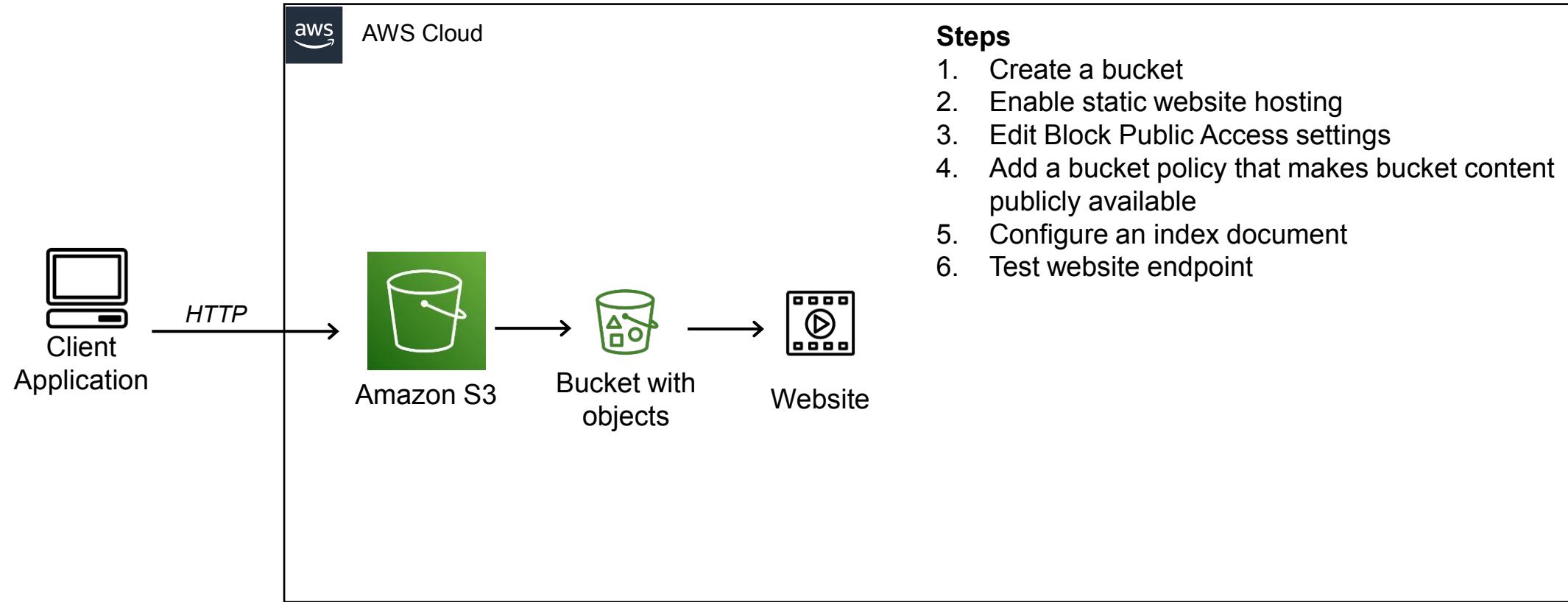
AWS Service Programmatic Access interaction with Serverless APIs using AWS SDK or CLI

**4**

## Hands-on Labs Real-World Apps

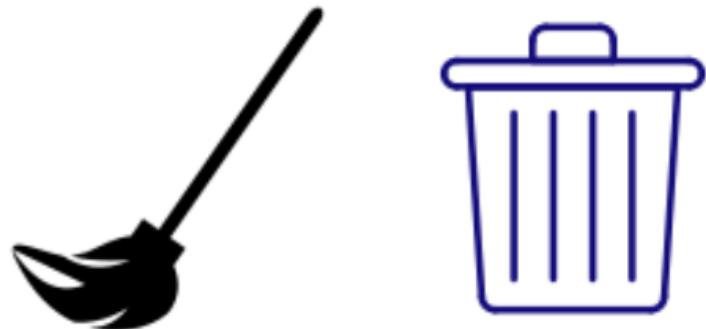
AWS Service Hands-on Labs implementation with Real-World Use Cases

# Hands-on Lab: Static Website Hosting on Amazon S3 using AWS Management Console



# Clean up Resources

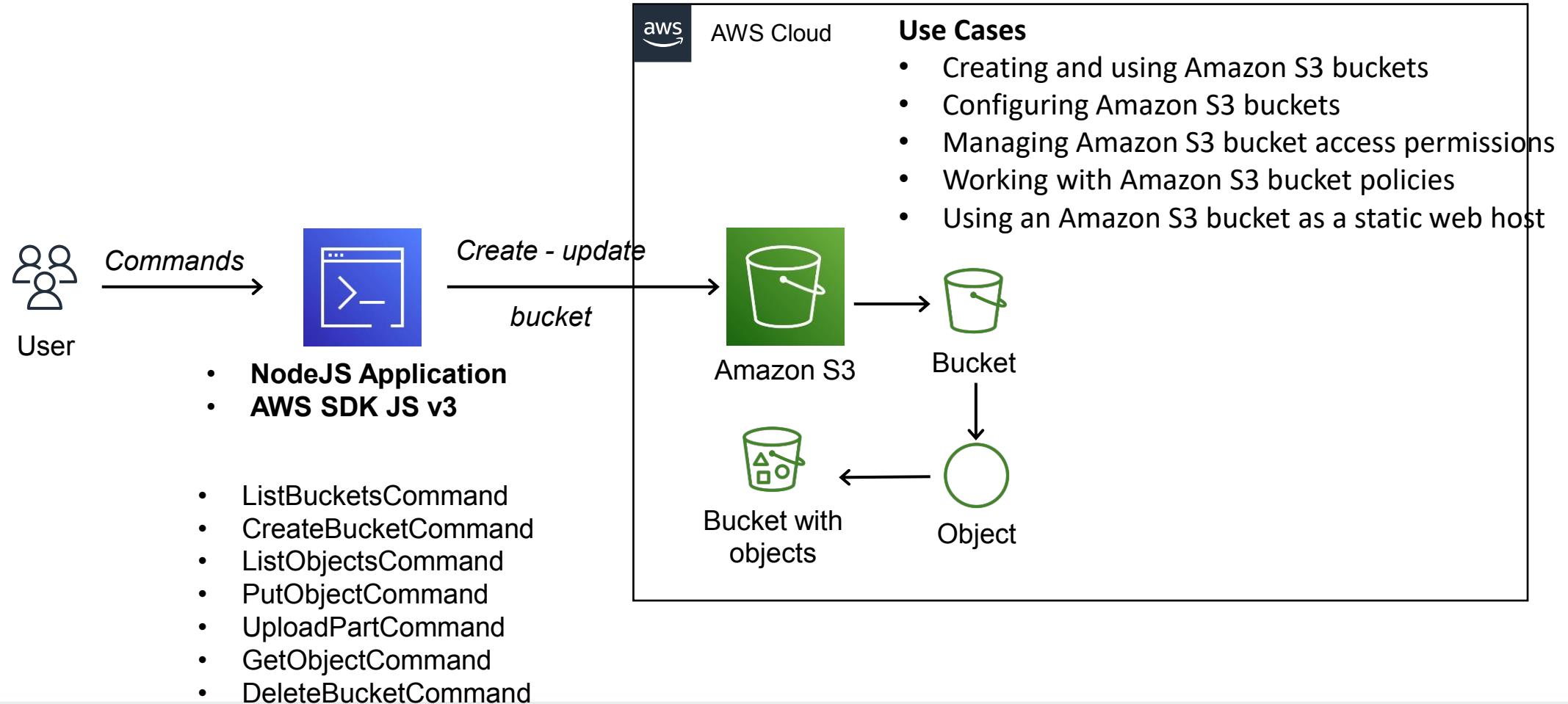
- Delete AWS Resources that we create during the section.



# Amazon S3 - Developing with AWS SDK

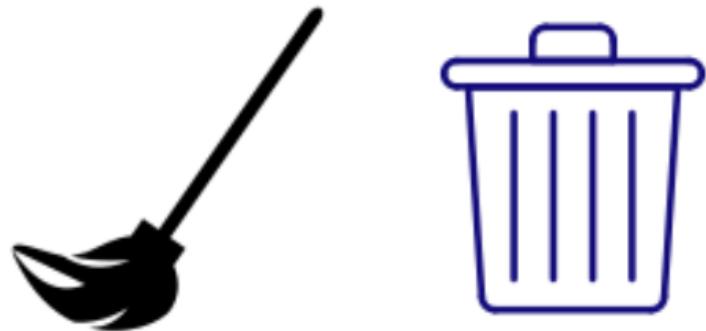
→ Amazon S3 - Developing with AWS SDK interaction to Serverless APIs Programmatic Access

# Amazon S3 SDK Examples using AWS SDK Javascript v3



# Clean up Resources

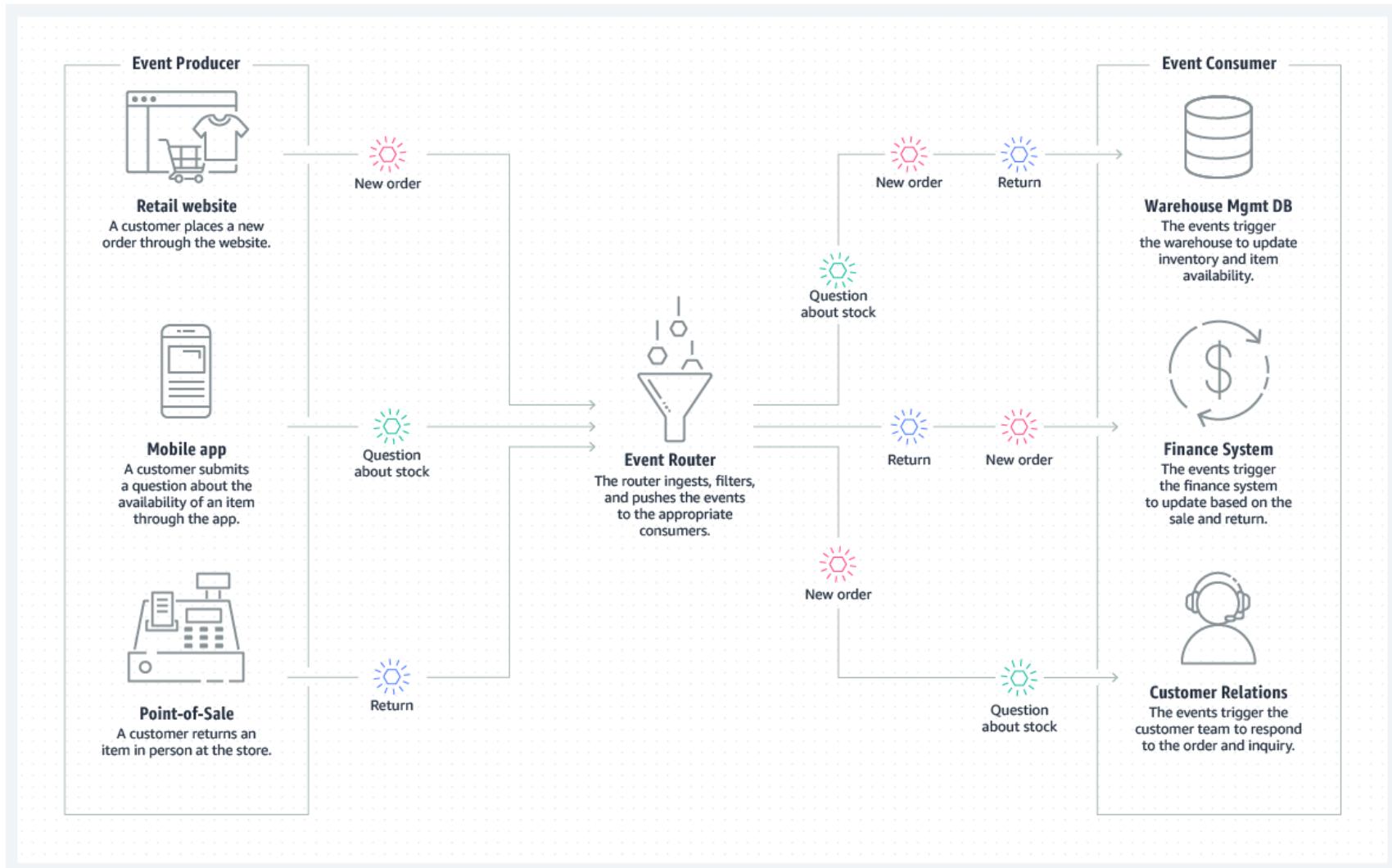
- Delete AWS Resources that we create during the section.



# AWS Lambda Event-Driven Architectures and Invocation Types

→ Learn How Event-Driven Architectures fits with AWS Lambda and Invocation Types

# Event-driven Architectures



<https://aws.amazon.com/event-driven-architecture/>



# Benefits of an Event-driven Architectures

- **Scale and fail independently**

With decoupling your services, they are only aware of the event router, not each other.

- **Develop with agility**

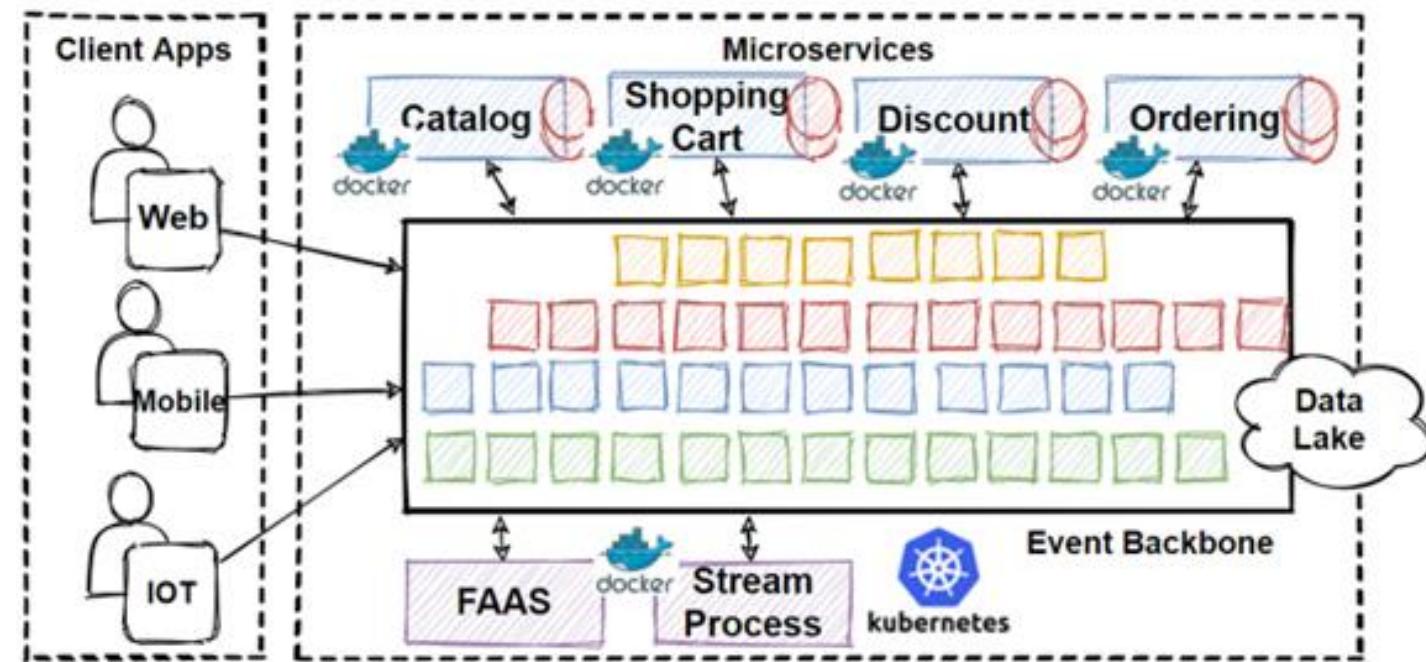
the event router will automatically filter and push events to consumers.

- **Audit with ease**

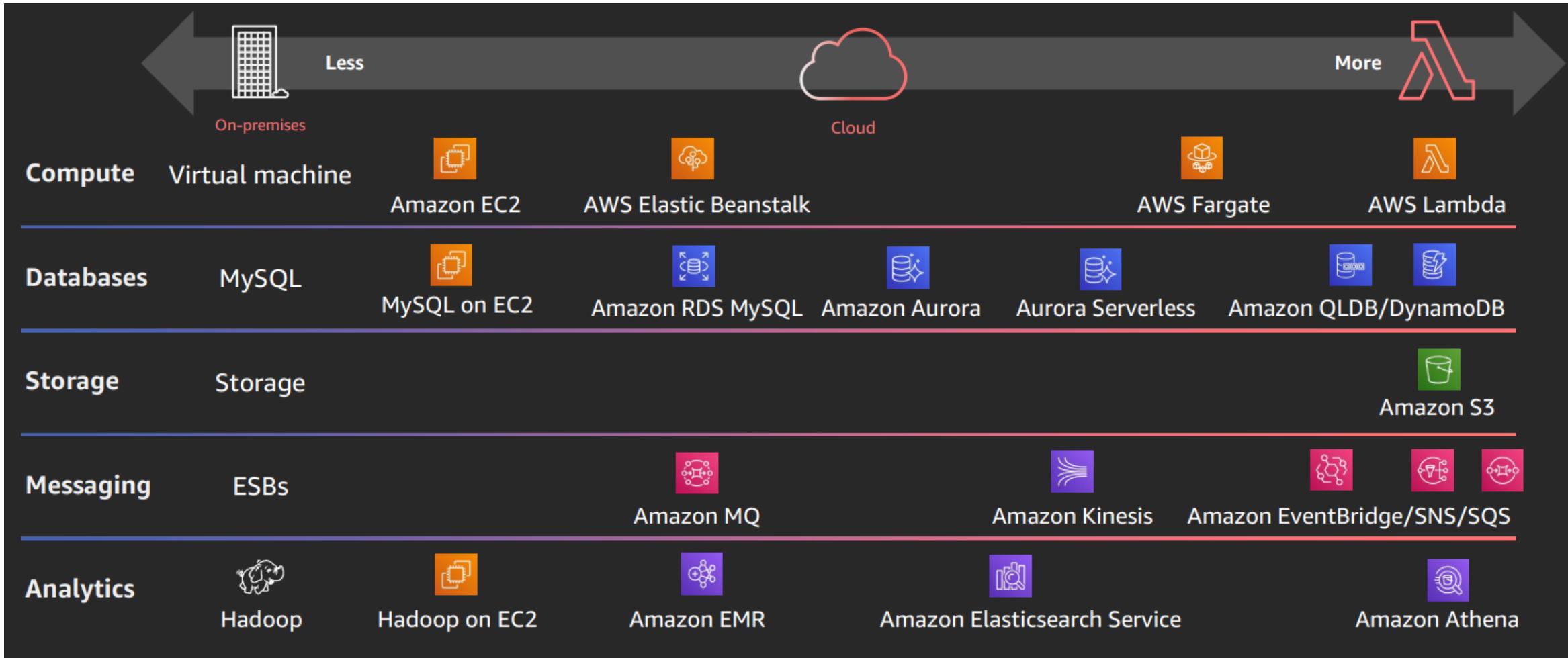
Restrict who can publish and subscribe to a router and control which users and resources have permission to access your data.

- **Cut costs**

Event-driven architectures are push-based, so everything happens on-demand as the event presents itself in the router.



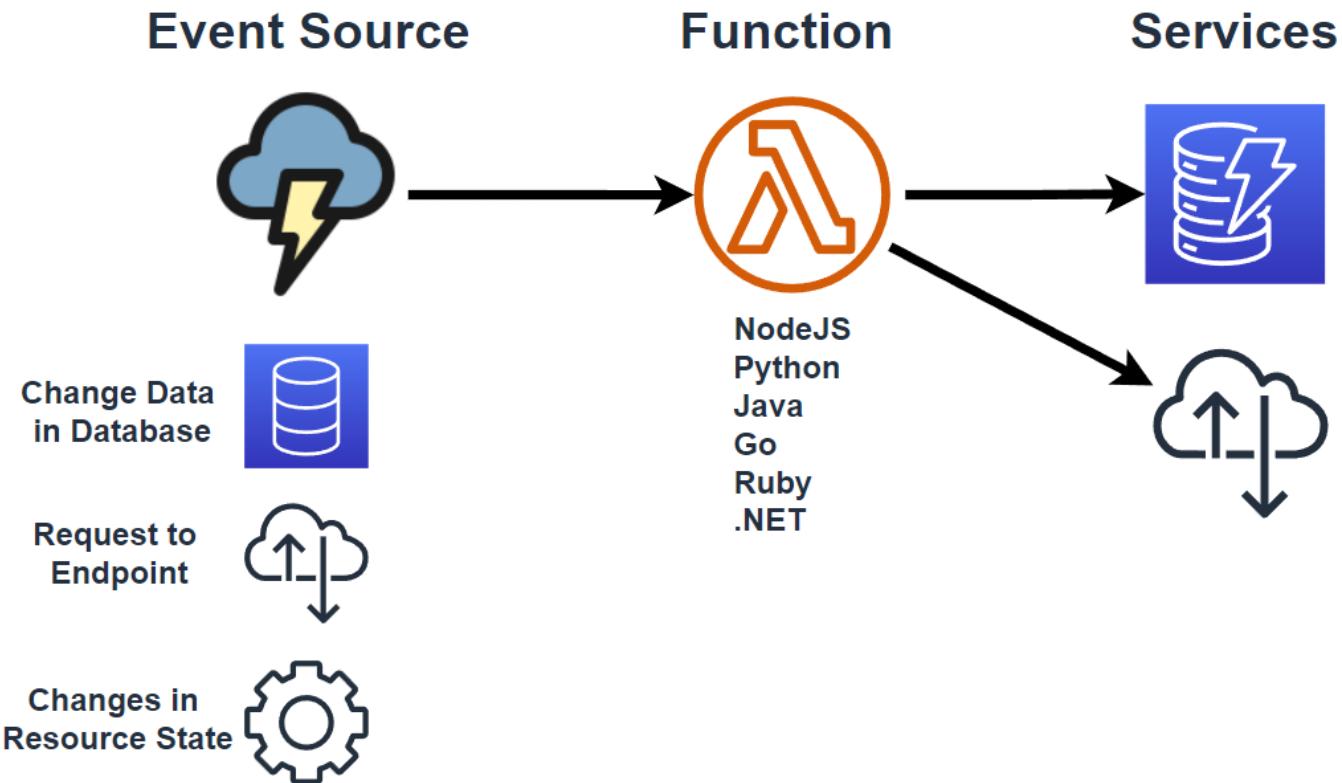
# AWS Event-Driven Services



[https://d1.awsstatic.com/events/reinvent/2019/REPEAT\\_3\\_Serverless\\_architectural\\_patterns\\_and\\_best\\_practices\\_ARC307-R3.pdf](https://d1.awsstatic.com/events/reinvent/2019/REPEAT_3_Serverless_architectural_patterns_and_best_practices_ARC307-R3.pdf)

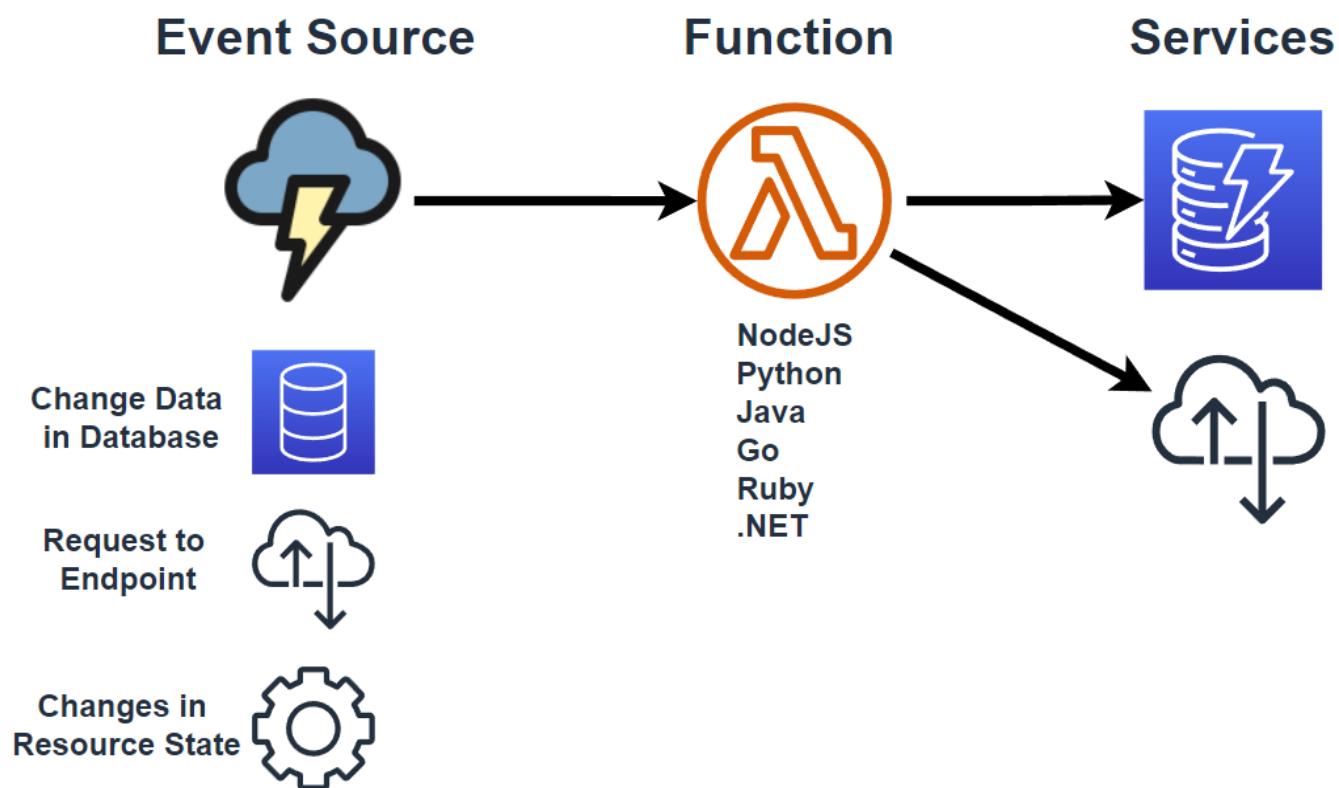
# Lambda Event Sources and Destination Trigger Services

- AWS Lambda integrates with other AWS services to invoke functions or take other actions.
- There is an event source or trigger, and actual Function code and then there is the destination.
- Event source has a number of services; they can be a http call, cron job, uploading an object into S3 bucket, third party call like payment done through stripe
- Triggering event to lambda function, lambda launch the execution environment with different language and runtimes
- Lambda has destinations that can be interaction with your function code



# Use Cases Lambda Event Sources and Destination Trigger Services

- Invoke a function in response to resource lifecycle events, such as with Amazon Simple Storage Service (Amazon S3)
- Respond to incoming HTTP requests. Using Lambda with API Gateway.
- Consume events from a queue. Using Lambda with Amazon SQS. Lambda poll queue records from Amazon SQS.
- Run a function on a schedule. Using AWS Lambda with Amazon EventBridge (CloudWatch Events).



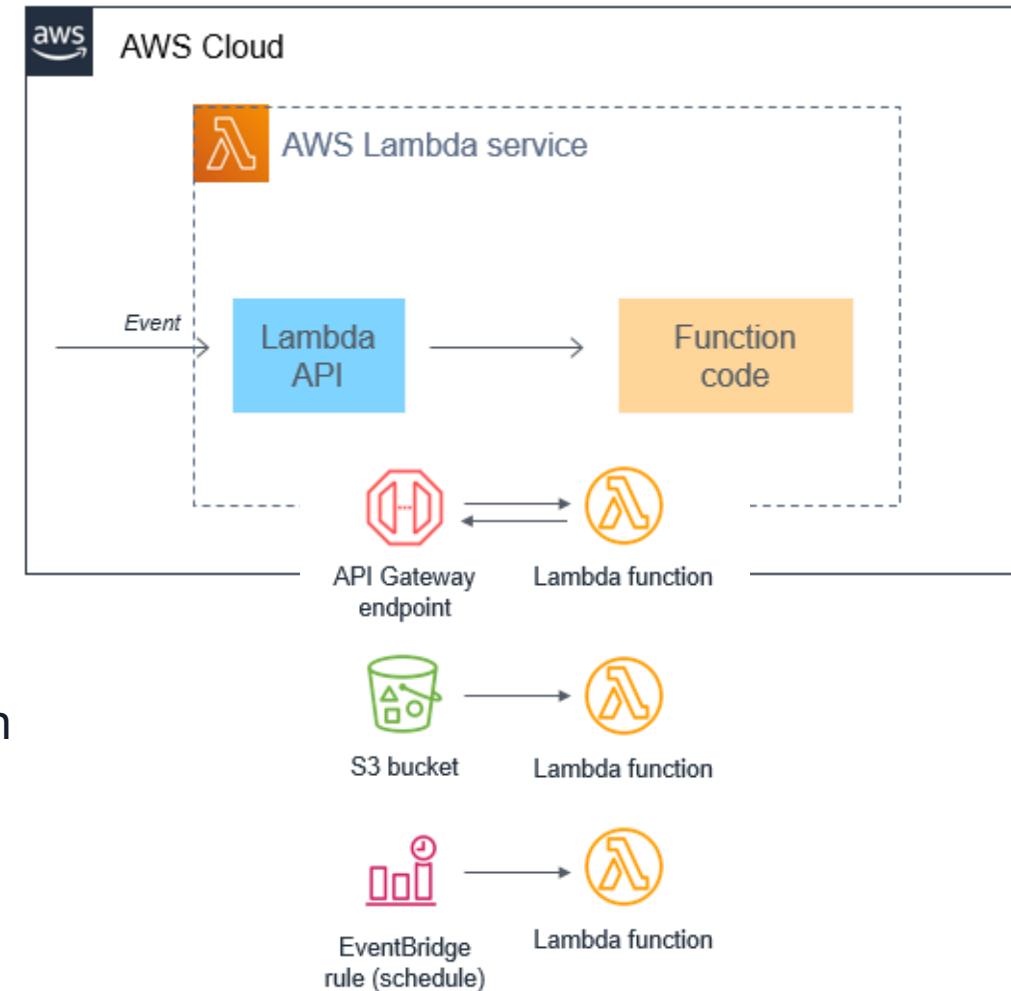
# List of Services Lambda Event Sources

| Service                                | Method of invocation                  |
|----------------------------------------|---------------------------------------|
| Amazon API Gateway                     | Event-driven; synchronous invocation  |
| AWS CloudFormation                     | Event-driven; asynchronous invocation |
| Amazon CloudFront (Lambda@Edge)        | Event-driven; synchronous invocation  |
| Amazon EventBridge (CloudWatch Events) | Event-driven; asynchronous invocation |
| Amazon CloudWatch Logs                 | Event-driven; asynchronous invocation |
| AWS CodeCommit                         | Event-driven; asynchronous invocation |
| AWS CodePipeline                       | Event-driven; asynchronous invocation |
| Amazon Cognito                         | Event-driven; synchronous invocation  |
| AWS Config                             | Event-driven; asynchronous invocation |
| Amazon Connect                         | Event-driven; synchronous invocation  |
| Amazon DynamoDB                        | Lambda polling                        |
| Amazon Elastic File System             | Special integration                   |
| Amazon Lambda (Application and Data)   | Event-driven; asynchronous invocation |

<https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html#eventsources-sqs>

# AWS Lambda Best Practices and Event-driven Architecture

- AWS Lambda **design principles** and the **best practices** when developing our Lambda-based Event-driven Serverless e-commerce applications.
- Lambda is very **good fit** with **Event-driven Architectures**.
- AWS services generate events for **communicating** each other, most of AWS services are **event sources** for Lambda.
- Lambda always **handle all interactions** with the **Lambda API** and there is no direct invocation of functions from outside the service.
- The main purpose of lambda functions is to **handle events**. Even the simplest Lambda-based application uses at least **one event**.
- Lambda functions are limited to **15 minutes** in duration.
- An event triggering a Lambda function could be **almost anything**.



# AWS Lambda Events

- The event is a **JSON object** that contains all information about what happened.
  - Represents a change in the system state.
- The first parameter of every **Lambda handler function** contains the event json object.
  - With using this event json object, we can access the event parameters into lambda function.
- An event could be custom-generated from another microservice,
  - New order generated in an ecommerce application.
- The event also can be generated from **existing AWS service**
  - Amazon SQS when a new queue message is available in a queue
- **Event-driven architectures** rely on creating events into all application state changes that are observable by other services
  - Loosely coupled services.

The screenshot shows the AWS Lambda function configuration and the code editor for a Lambda function named "NewOrderEvent".

**Event name:** NewOrderEvent

**Event body (JSON content):**

```
1 [{ "source": "myApplication",
2 "detail": "submitOrder",
3 "customerId": "customer123",
4 "orderId": "order-A1234B56",
5 "paymentStatus": "open",
6 "cart": [
7 {
8 "product12": {
9 "qty": 2,
10 "itemPx": 35.22,
11 "currency": "USD"
12 },
13 "product44": {
14 "qty": 5,
15 "itemPx": 71.57,
16 "currency": "USD"
17 }
18],
19 "timestamp": 1607774286
20 }
21 }
22]
```

**index.js (Lambda Function Code):**

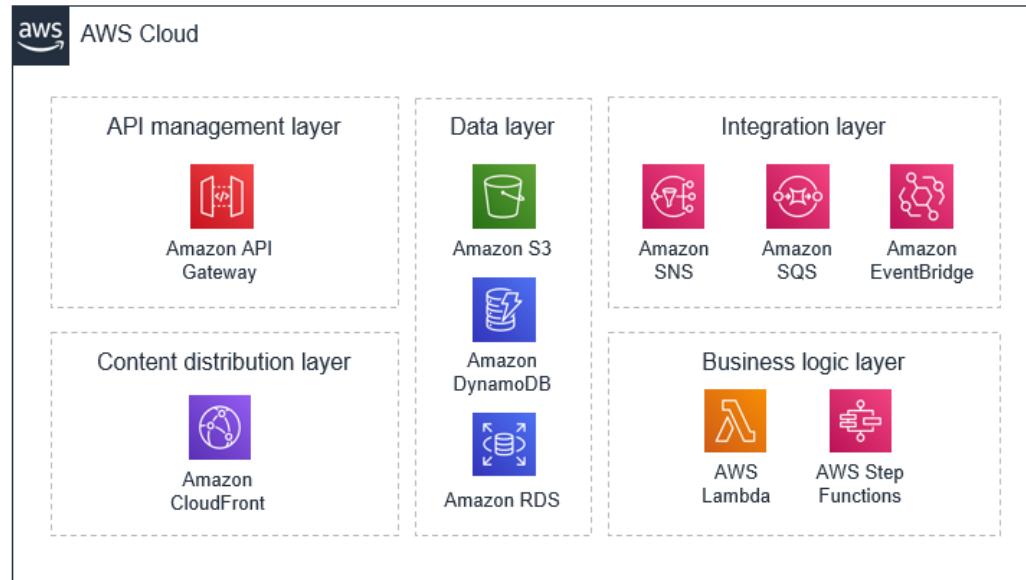
```
i 1 const AWS = require('aws-sdk')
i 2 AWS.config.region = process.env.AWS_REGION
i 3
i 4 const s3 = new AWS.S3()
i 5
i 6 exports.handler = async (event) => {
i 7
i 8 console.log(JSON.stringify(event,0, null))
i 9
i 10 const Bucket = event.Records[0].s3.bucket.name
i 11 const Key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, ' '))
i 12
i 13 const result = await s3.getObject({
i 14 Bucket,
i 15 Key
i 16 }).promise()
i 17
i 18 const data = result.Body
i 19 console.log('PDF text length: ', data.length)
i 20
i 21 }
```

**Annotations:**

- 1**: Points to the `event.source` and `event.detail` variables in the Lambda function code.
- 2**: Points to the `Bucket` and `Key` variables used to retrieve the PDF file from S3.

# AWS Lambda Best Practices and Event-driven Architecture

- Most Lambda-based applications use a **combination of AWS services** for different requirements about **Storage, API Management** and **integrating** with other system and services.
- Lambda is **connecting between services**, providing business logic to transform data that moves between services.
- You can find mostly **integrated AWS Services** which using Lambda functions.
- **Design patterns in Distributed architectures** with AWS Lambda
- When your application needs one of these patterns, we can use the corresponding AWS service.
- These services and patterns are **designed to integrate** with AWS Lambda functions



| Pattern                     | AWS service        |
|-----------------------------|--------------------|
| Queue                       | Amazon SQS         |
| Event bus                   | Amazon EventBridge |
| Publish/subscribe (fan-out) | Amazon SNS         |
| Orchestration               | AWS Step Functions |
| API                         | Amazon API Gateway |
| Event streams               | Amazon Kinesis     |

<https://aws.amazon.com/blogs/compute/operating-lambda-design-principles-in-event-driven-architecture/>



# AWS Lambda Invocation Types

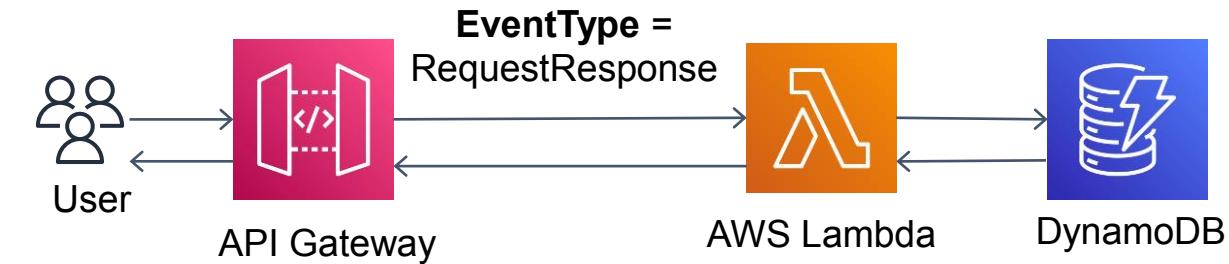
- Triggered lambda functions with different AWS Lambda Invocation Types
- AWS Lambda has 3 Invocation Types
- **Lambda Synchronous invocation**
- **Lambda Asynchronous invocation**
- **Lambda Event Source Mapping with polling invocation**



<https://aws.amazon.com/blogs/architecture/understanding-the-different-ways-to-invoke-lambda-functions/>

# AWS Lambda Synchronous Invocation

- Execute immediately when you perform the Lambda Invoke API call.
- Wait for the function to process the function and return back to response.
- API Gateway + Lambda + DynamoDB
- Invocation-type flag should be “RequestResponse”
- Responsible for inspecting the response and determining if there was an error and decide to retry the invocation



- Example of synchronous invocation using the AWS CLI:  
aws lambda invoke —function-name MyLambdaFunction —invocation-type RequestResponse —payload '{ "key": "value" }'
- Triggered AWS services of synchronous invocation; ELB (Application Load Balancer), Cognito, Lex, Alexa, API Gateway, CloudFront, Kinesis Data Firehose

# AWS Lambda Asynchronous Invocation

- Lambda **sends the event** to a **internal queue** and returns a **success response** without any additional information
- Separate process **reads events** from the **queue** and **runs** our lambda function
- **S3 / SNS + Lambda + DynamoDB**
- **Invocation-type** flag should be “**Event**”
- AWS Lambda sets a **retry policy**

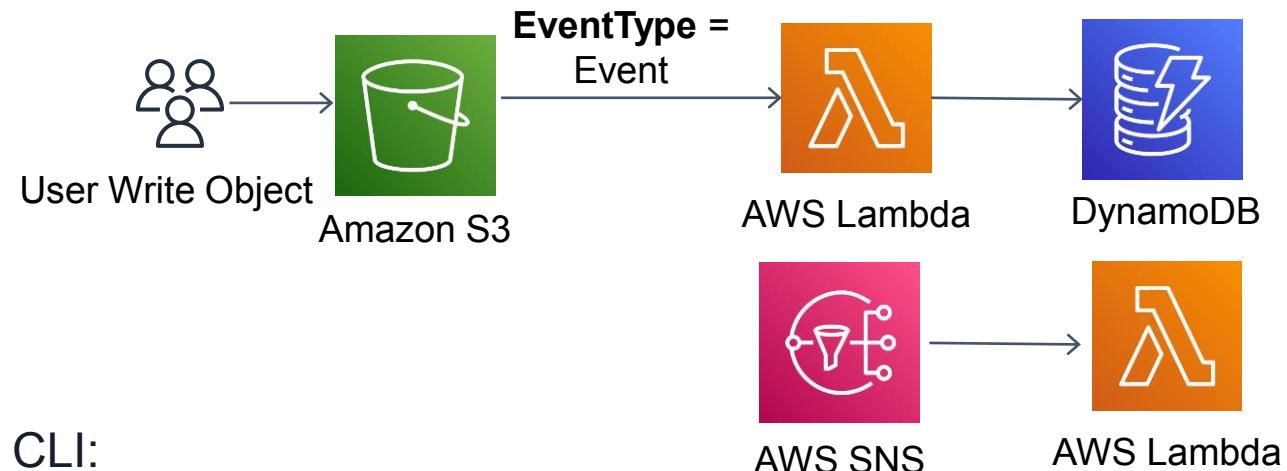
Retry Count = 2

Attach a Dead-Letter Queue (DLQ)

- Example of asynchronous invocation using the AWS CLI:

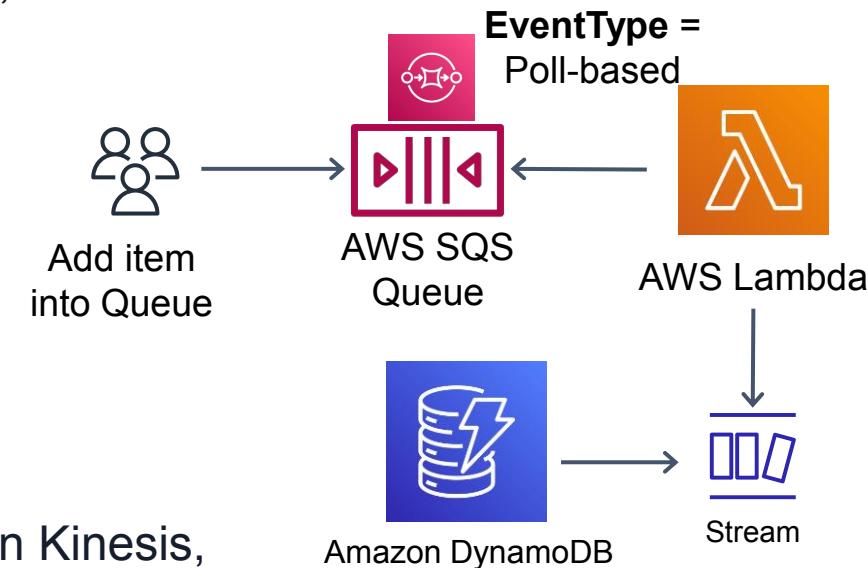
```
aws lambda invoke --function-name MyLambdaFunction --invocation-type Event --payload '{ "key": "value" }'
```

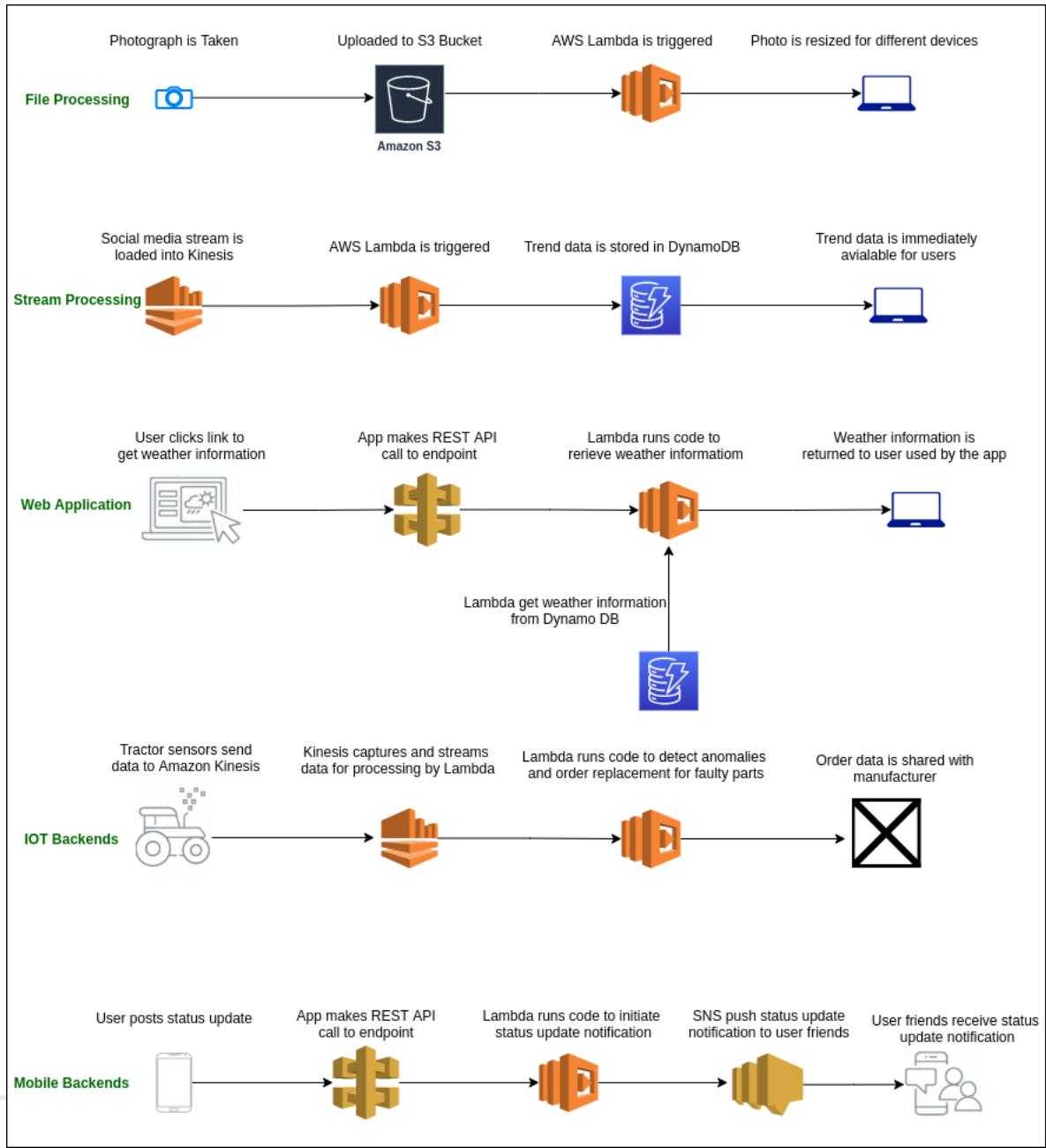
- Triggered **AWS services of asynchronous invocation**; S3, EventBridge, SNS, SES, CloudFormation, CloudWatch Logs, CloudWatch Events, CodeCommit



# AWS Lambda Event Source Mapping with Polling Invocation

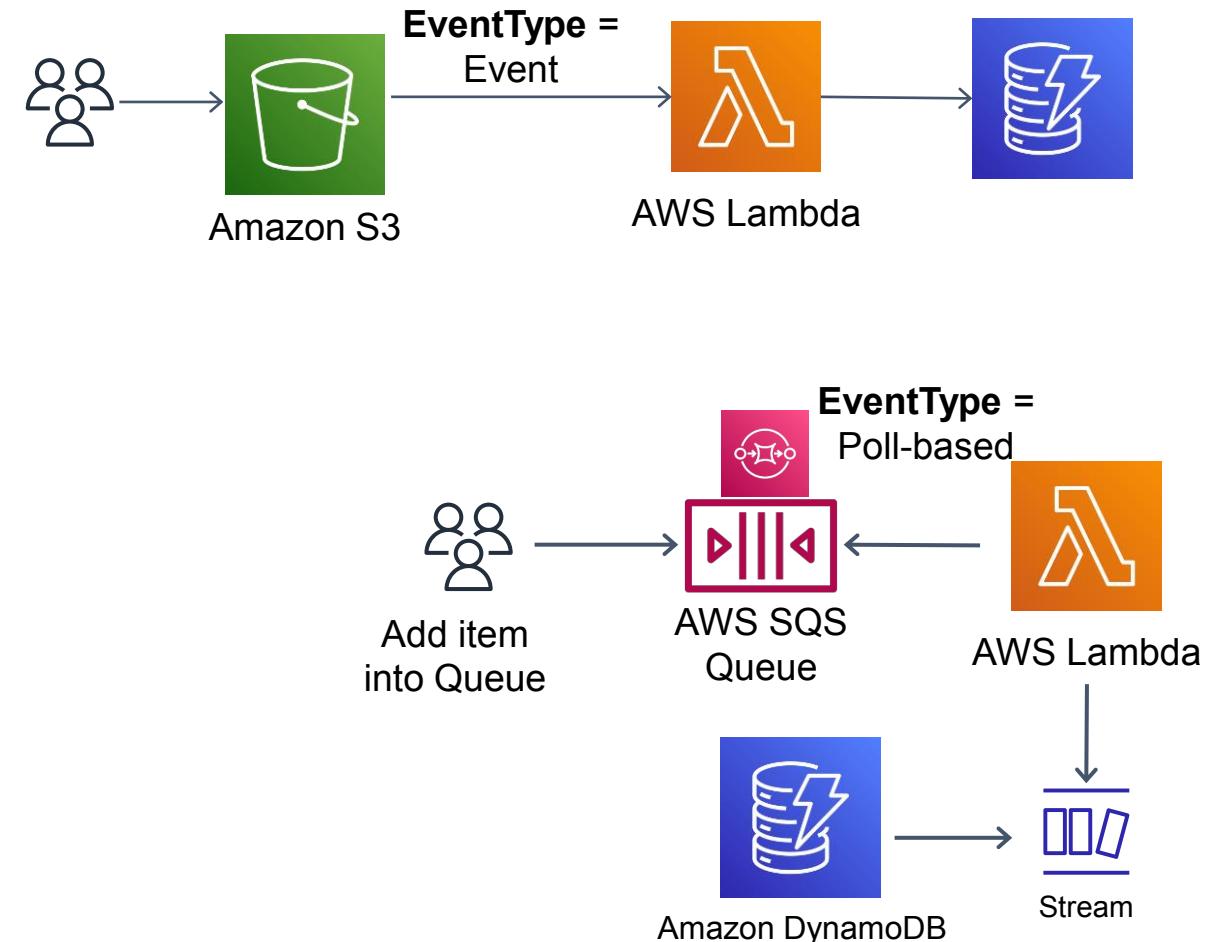
- **Pool-Based invocation** model allows us to **integrate** with **AWS Stream** and **Queue based services**.
- Lambda will **poll** from the AWS SQS or Kinesis streams, retrieve records, and invoke functions.
- Data stream or queue are **read in batches**,
- The function receives multiple items when execute function.
- **Batch sizes** can configure according to service types
- **SQS + Lambda**
- **Stream based processing** with **DynamoDB Streams + Lambda**
- Triggered **AWS services** of **Event Source Mapping invocation**; Amazon Kinesis, DynamoDB, Simple Queue Service (SQS)





# Hands-on Labs for Event-Driven Architectures

- **Hands-on Lab 1 :** Amazon S3 pushes events and invokes a Lambda function
- S3 can publish events of different types, such as PUT, POST, COPY, and DELETE object events on a bucket.
- **Hands-on Lab 2 :** AWS Lambda pulls events from a Kinesis or DynamoDB stream and invokes a Lambda function
- For poll-based event sources, AWS Lambda polls the source and then invokes the Lambda function when records are detected on that source.



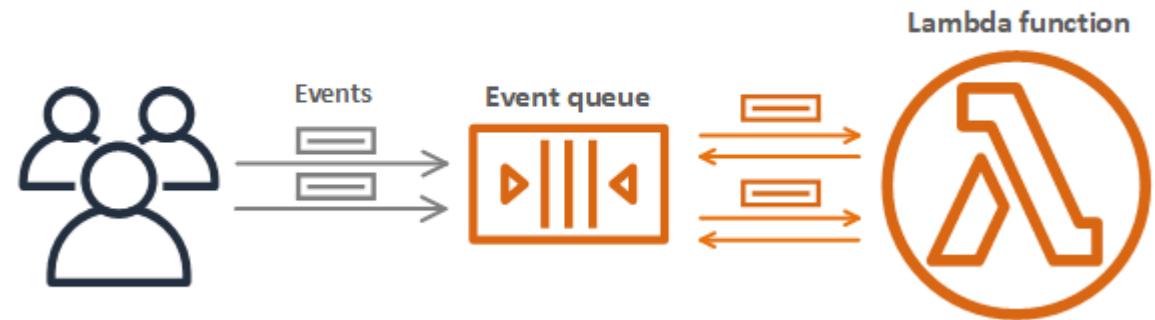
# Hands-on Lab: Invoke Lambda Asynchronously Using Amazon S3

→ Developing Hands-on Lab: Invoke Lambda Asynchronously Using Amazon S3 to Trigger Lambda

# AWS Lambda Asynchronous Invocation

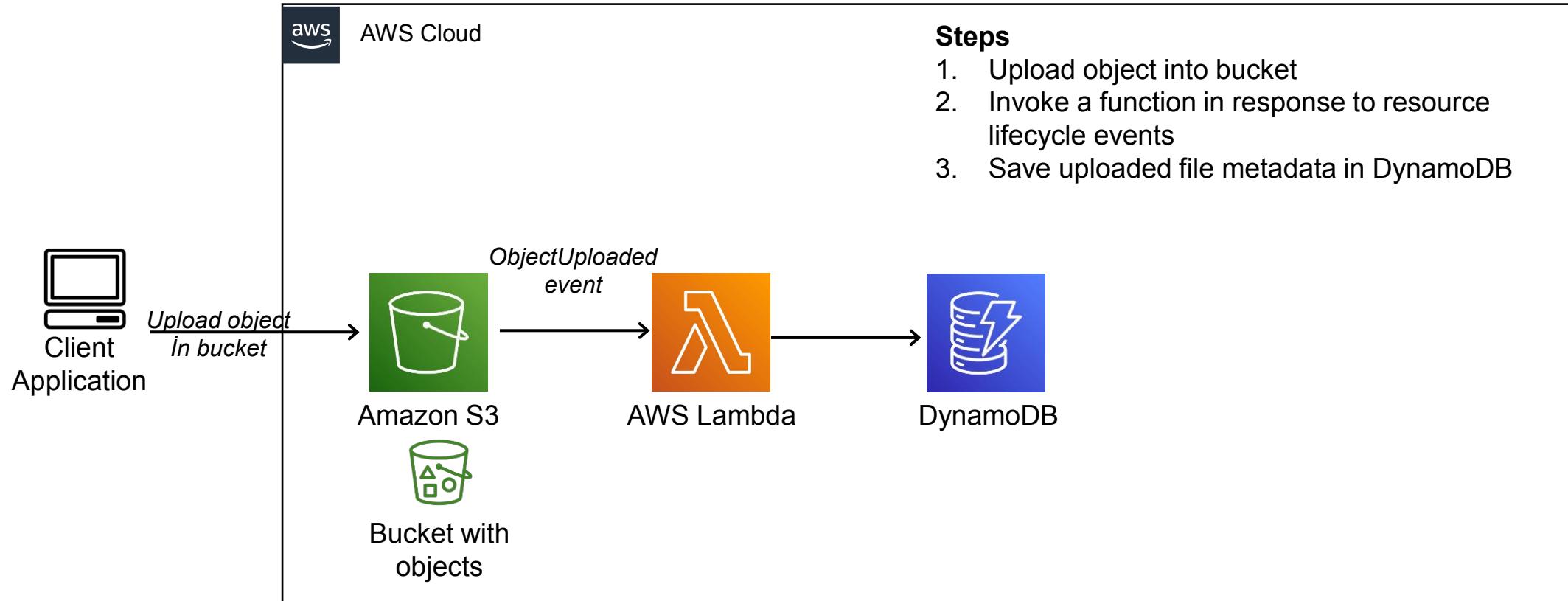
- AWS Lambda queues the events before sending them to the function.
- Lambda places the event in a queue and returns a success response without additional information.
- Separate process reads events from the queue and sends them to your function.
- ```
aws lambda invoke \
--function-name calculator \
--invocation-type Event \
--cli-binary-format raw-in-base64-out \
--payload '{ "key": "value" }' response.json
```

Asynchronous Invocation



<https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html>

Hands-on Lab: Using an Amazon S3 trigger to invoke a Lambda function and persist on DynamoDB



Serverless Project Development Phases

1

Infrastructure Creation on AWS

Create API Gateway, Lambda Function and DynamoDB table on AWS Cloud - Also we can automate this part with IaC using CDK in the last sections but now we will create infrastructure with console or cli

2

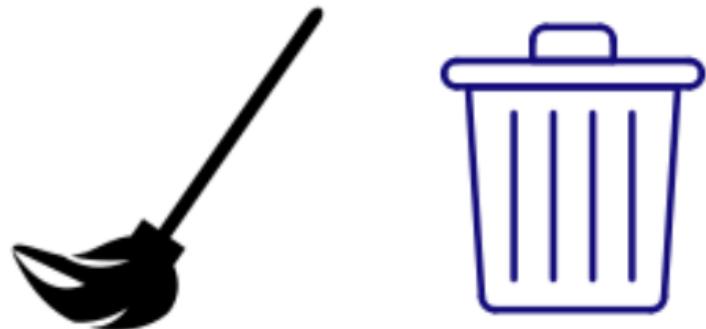
Develop Lambda Microservice CRUD Rest Api business logic with AWS SDK

Use AWS SDK JS v3 with ES6 standards to implement crud functions into lambda function.



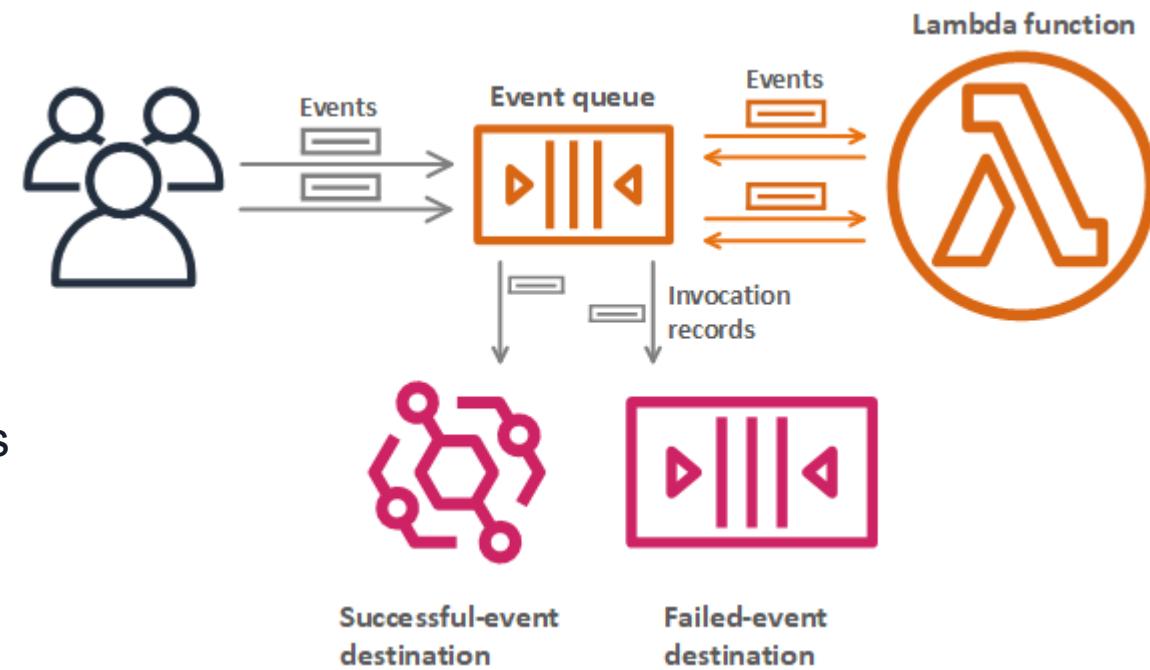
Clean up Resources

- Delete AWS Resources that we create during the section.



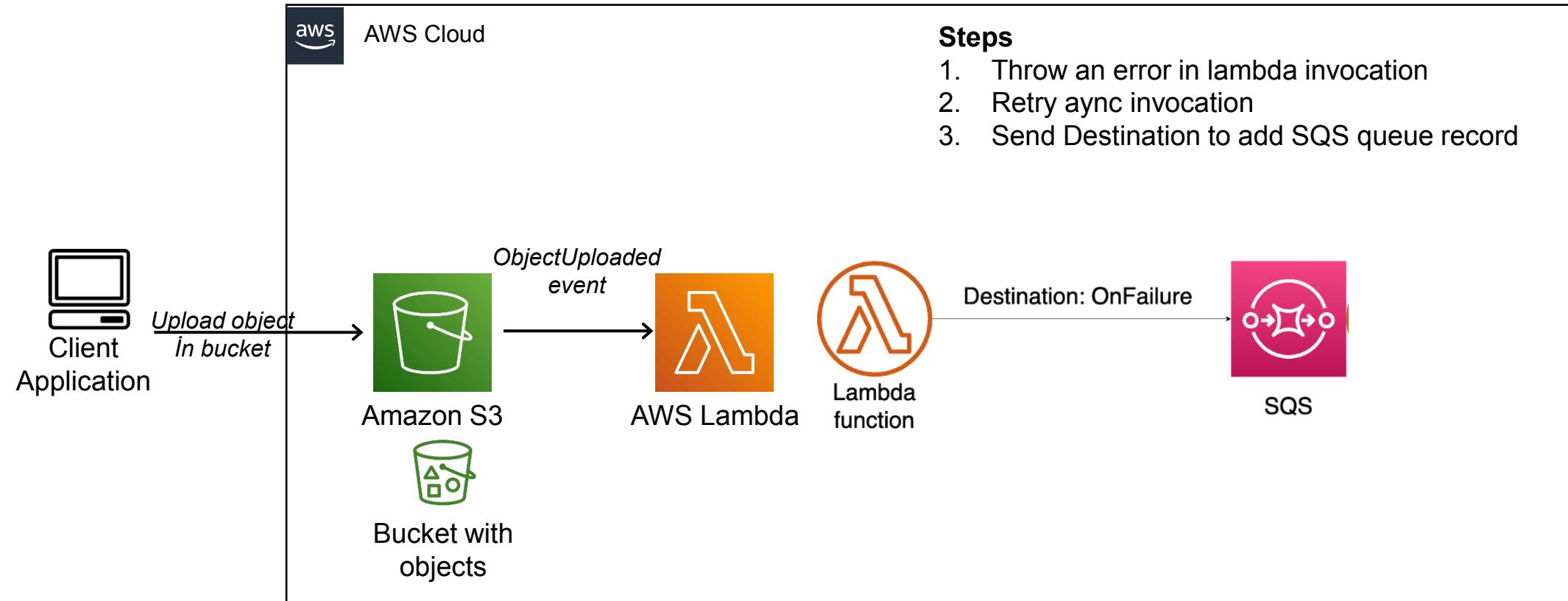
AWS Lambda Lambda Destinations

- Lambda can define Destination when invoke async way.
Configure Lambda to send an invocation record to another service.
 - Amazon SQS – A standard SQS queue.
 - Amazon SNS – An SNS topic.
 - AWS Lambda – A Lambda function.
 - Amazon EventBridge – An EventBridge event bus.
- When the function success response, Lambda sends to EventBridge event bus. When an event fails, Lambda sends to Amazon SQS queue.
- To send events to a destination, function needs additional permissions.
 - Amazon SQS – sqs:SendMessage
 - Amazon SNS – sns:Publish
 - Lambda – InvokeFunction
 - EventBridge – events:PutEvents



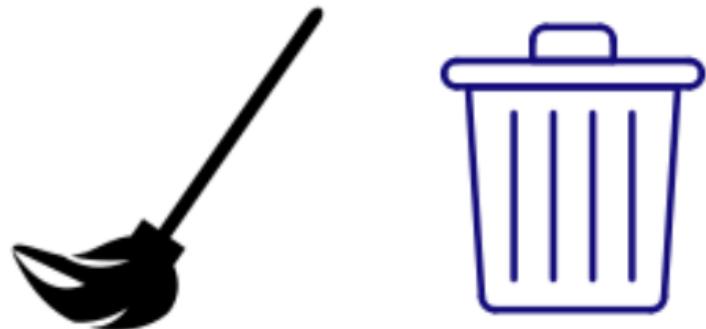
<https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#invocation-async-destinations>

Hands-on Lab: AWS Lambda Destination to SQS - DLQ Case



Clean up Resources

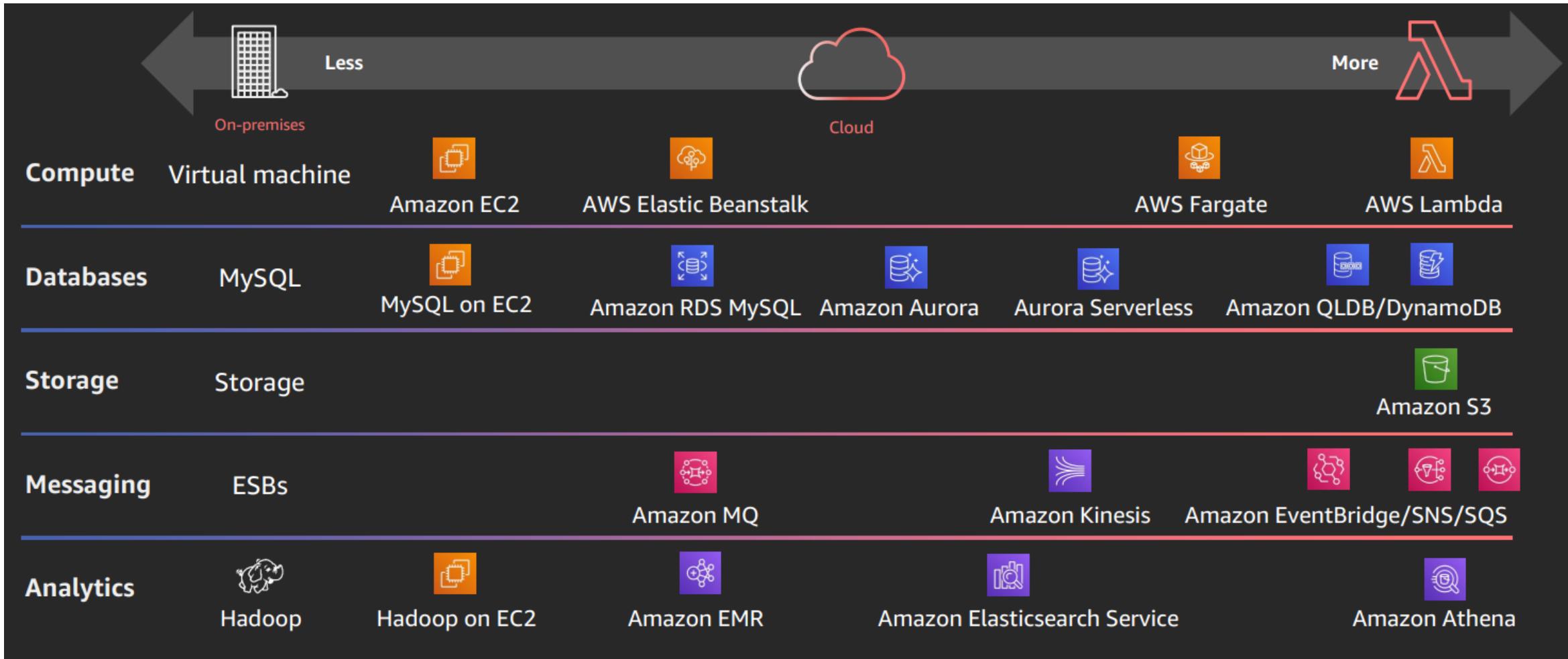
- Delete AWS Resources that we create during the section.



Amazon SNS - Fully Managed Pub/Sub Messaging

→ Learning Amazon SNS - Fully Managed Pub/Sub Messaging

AWS Application Integration Services

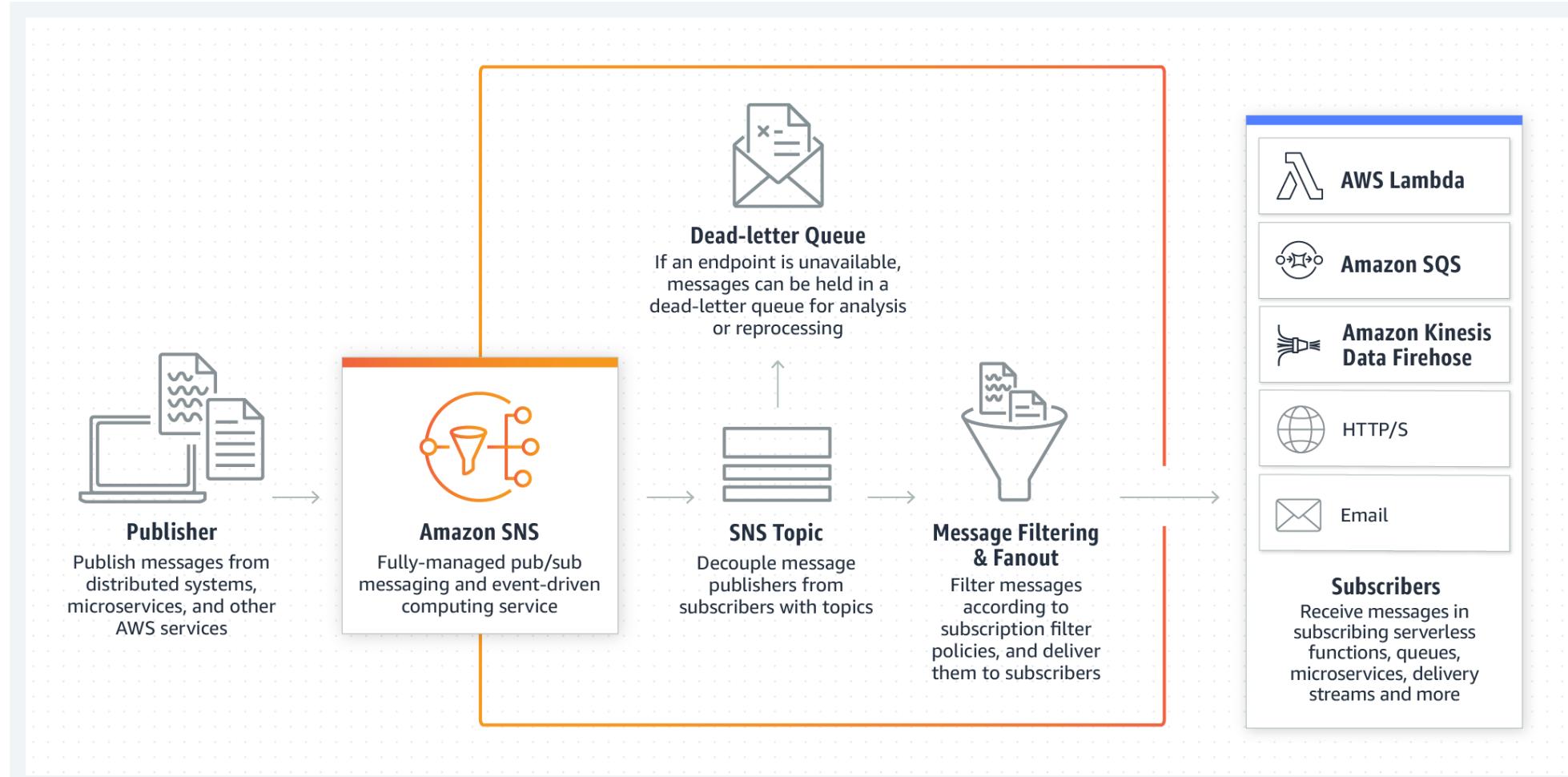


https://d1.awsstatic.com/events/reinvent/2019/REPEAT_3_Serverless_architectural_patterns_and_best_practices_ARC307-R3.pdf

Amazon SNS: Fully Managed Pub/Sub Messaging



Application
Integration



<https://aws.amazon.com/sns/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=deschttps%3A%2F%2Fdocs.aws.amazon.com%2Fsns%2Flatest%2Fdg%2Fwelcome.html>

Amazon SNS: Fully Managed Pub/Sub Messaging



Application
Integration

- **Application integration**

The Fanout scenario is when a message published to an SNS topic is replicated and pushed to multiple endpoints.

- **Application alerts**

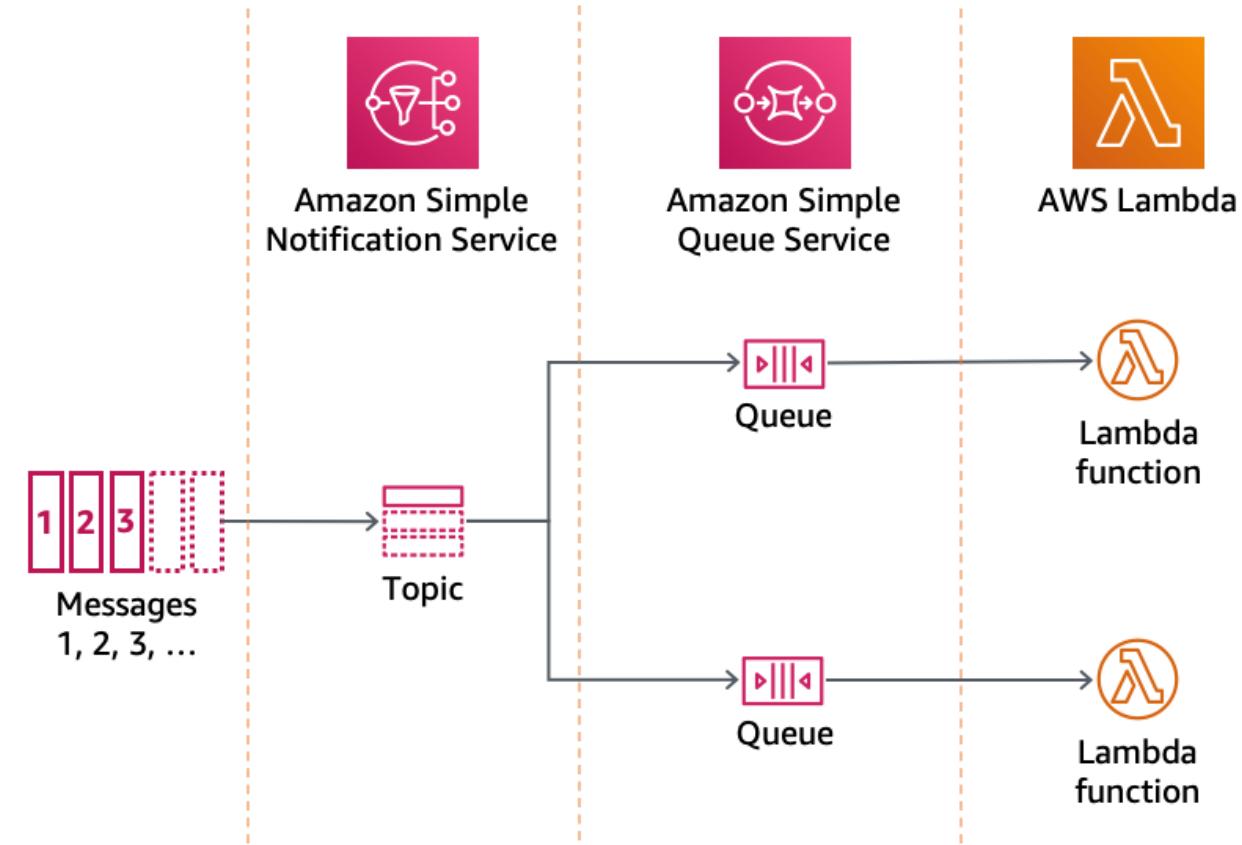
Amazon SNS can send notifications to specified users via SMS and email.

- **User notifications**

Amazon SNS can send push email messages and text messages to individuals or groups.

- **Mobile push notifications**

Mobile push notifications enable you to send messages directly to mobile apps.



Amazon SNS Event Sources and Destinations



Application
Integration

Amazon SNS Event Sources

Application integration services

- EventBridge
- Step Functions

Compute services

- Lambda
- EC2

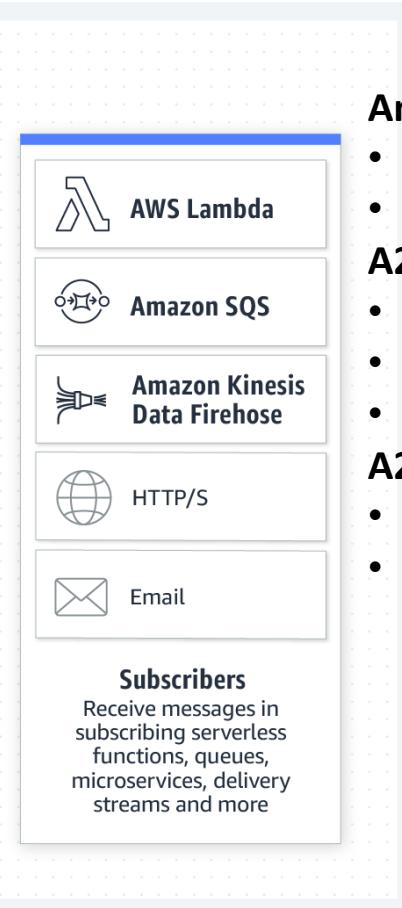
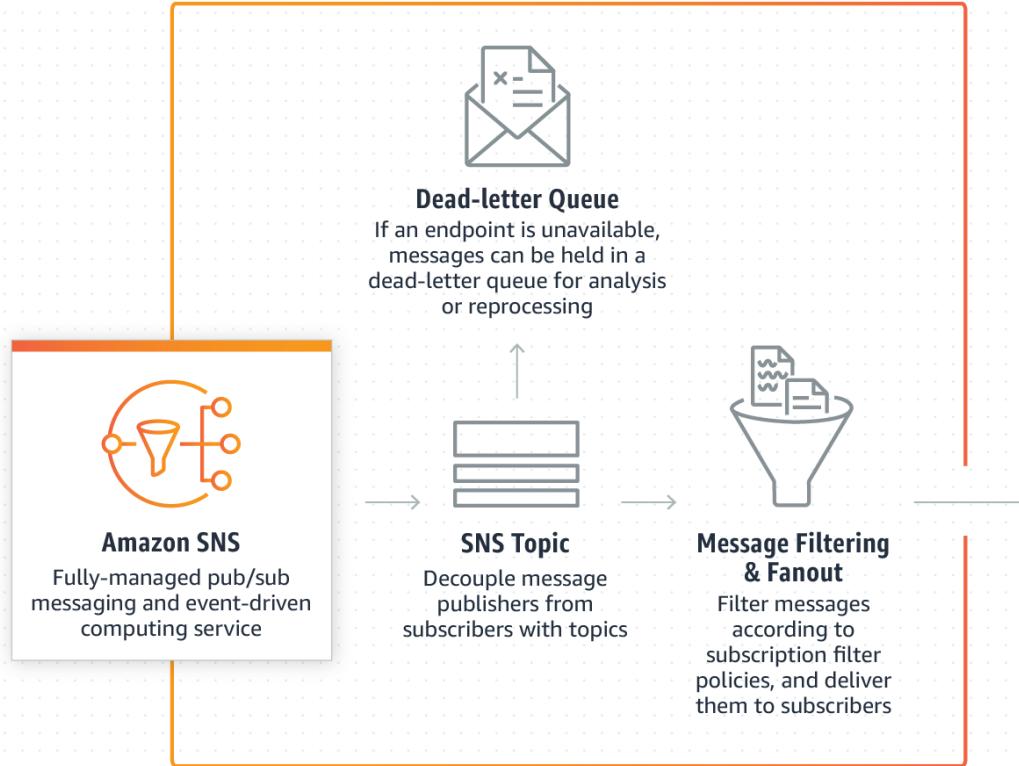
Database services

- DynamoDB
- Elatic Cache



Publisher

Publish messages from distributed systems, microservices, and other AWS services



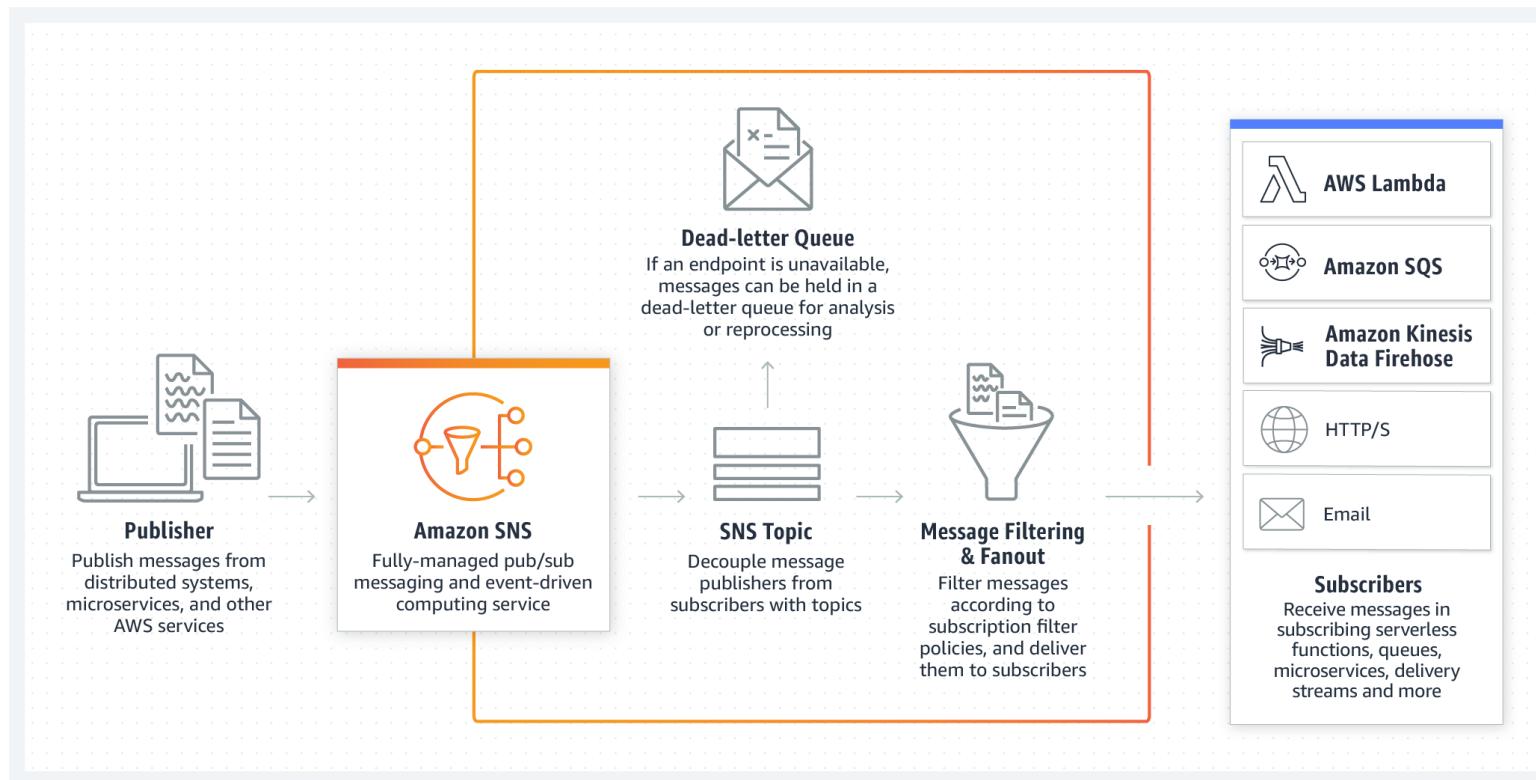
<https://aws.amazon.com/sns/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=deschttps%3A%2F%2Fdocs.aws.amazon.com%2Fsns%2Flatest%2Fdg%2Fwelcome.html>

Amazon SNS Actions



Amazon SNS Actions

- Create a topic
- Delete a subscription
- Delete a topic
- List topics
- Publish an SMS text message
- Publish to a topic
- Set a dead-letter queue for a subscription
- Set a filter policy
- Set the default settings for sending SMS messages
- Set topic attributes
- Subscribe a Lambda function to a topic
- Subscribe a mobile application to a topic

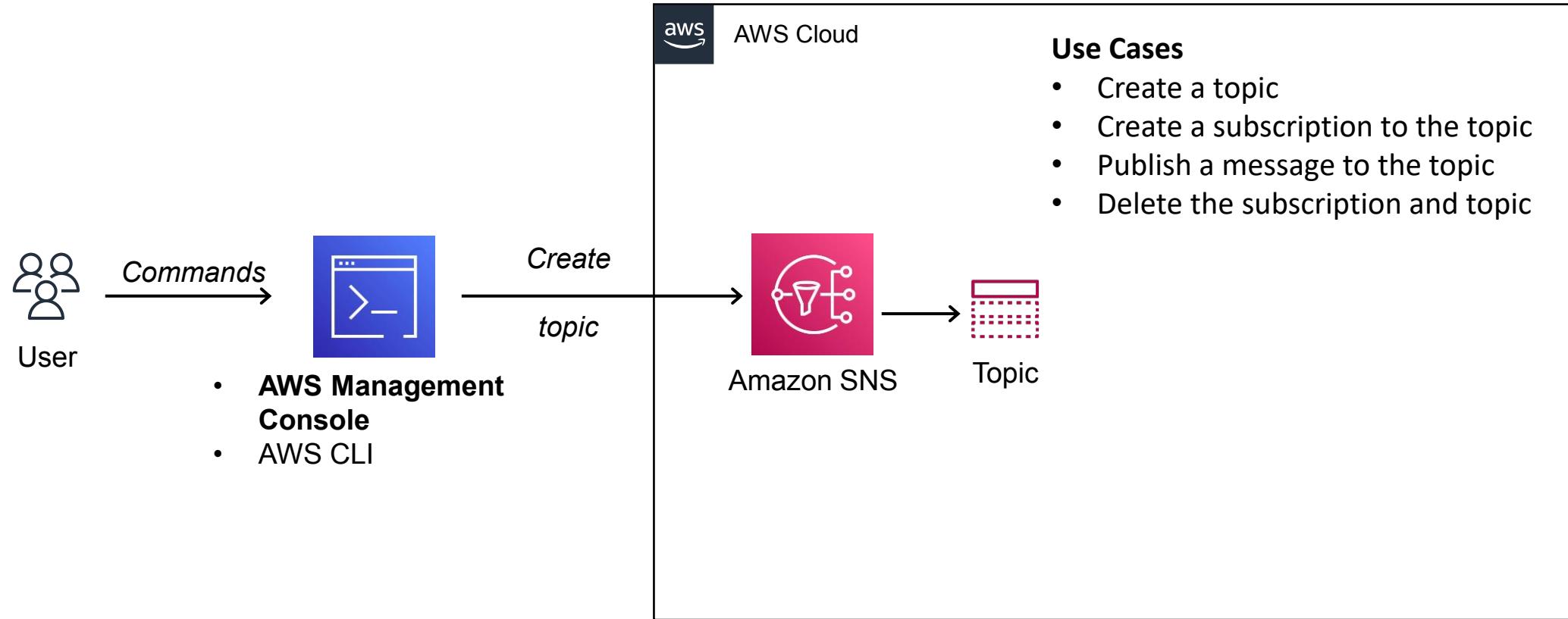


<https://aws.amazon.com/sns/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=deschttps%3A%2F%2Fdocs.aws.amazon.com%2Fsns%2Flatest%2Fdg%2Fwelcome.html>

Amazon SNS - Walkthrough with AWS Management Console

→ DEMO - Amazon SNS - Walkthrough with AWS Management Console

Getting started with Amazon SNS with AWS Management Console



AWS Lambda & Serverless Course Learning Path

1

Theoretical Information

AWS Service overview, core concepts, features, uses cases and general information

2

Walkthrough with AWS Console

AWS Service Walkthrough with AWS Management Console performs main use cases

3

Developing with AWS SDK

AWS Service Programmatic Access interaction with Serverless APIs using AWS SDK or CLI

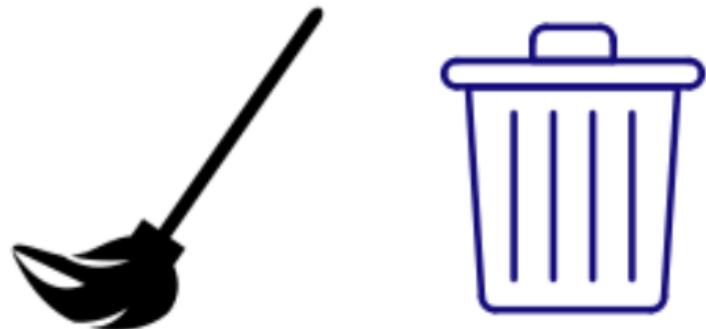
4

Hands-on Labs Real-World Apps

AWS Service Hands-on Labs implementation with Real-World Use Cases

Clean up Resources

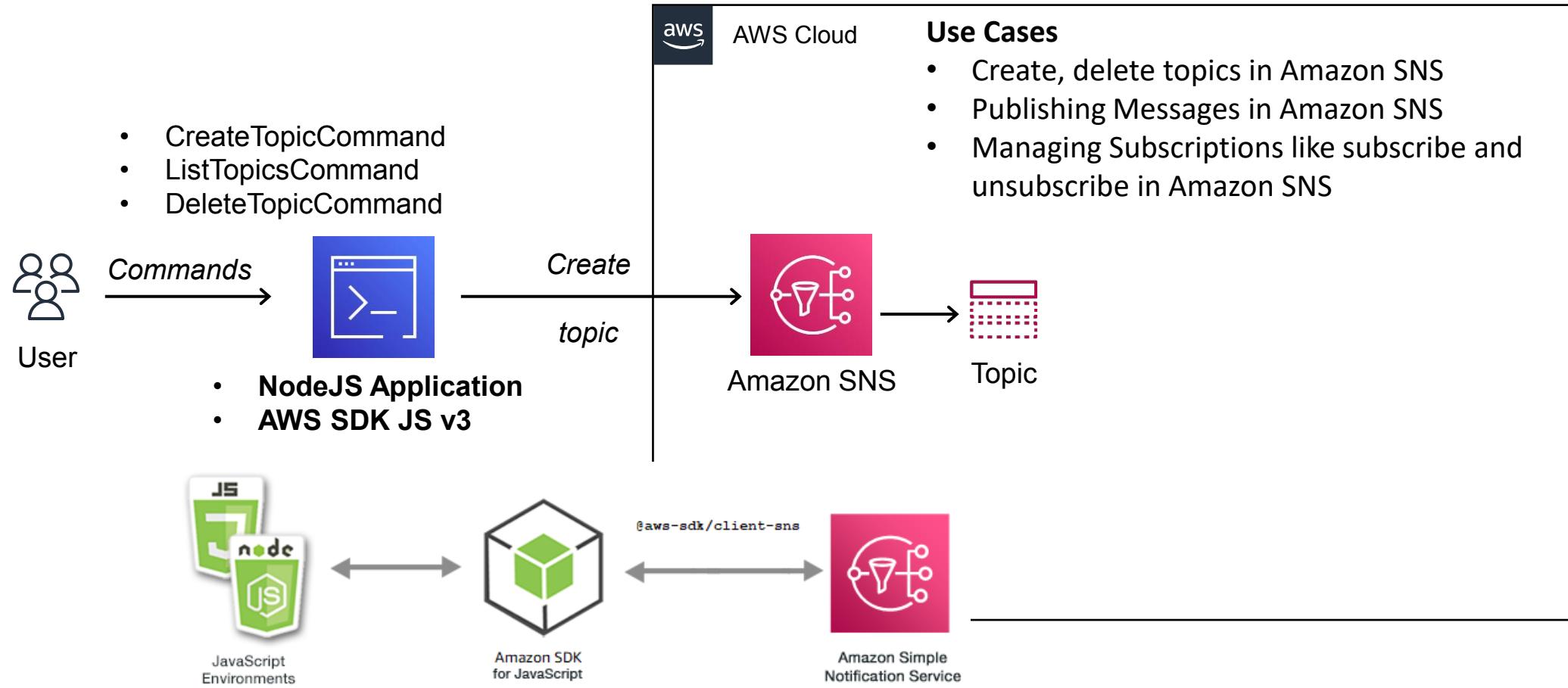
- Delete AWS Resources that we create during the section.



Amazon SNS - Developing with AWS SDK

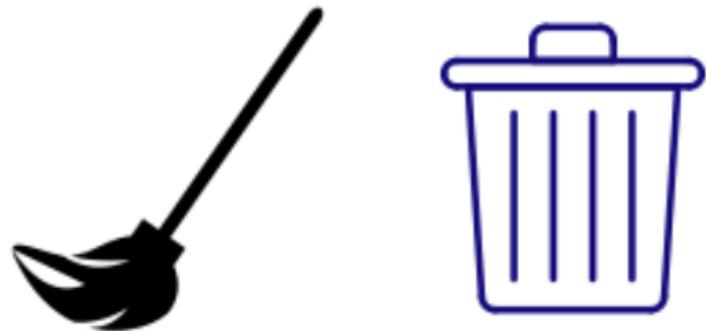
→ Amazon SNS - Developing with AWS SDK interaction to Serverless APIs Programmatic Access

Amazon SNS SDK Examples using AWS SDK Javascript v3



Clean up Resources

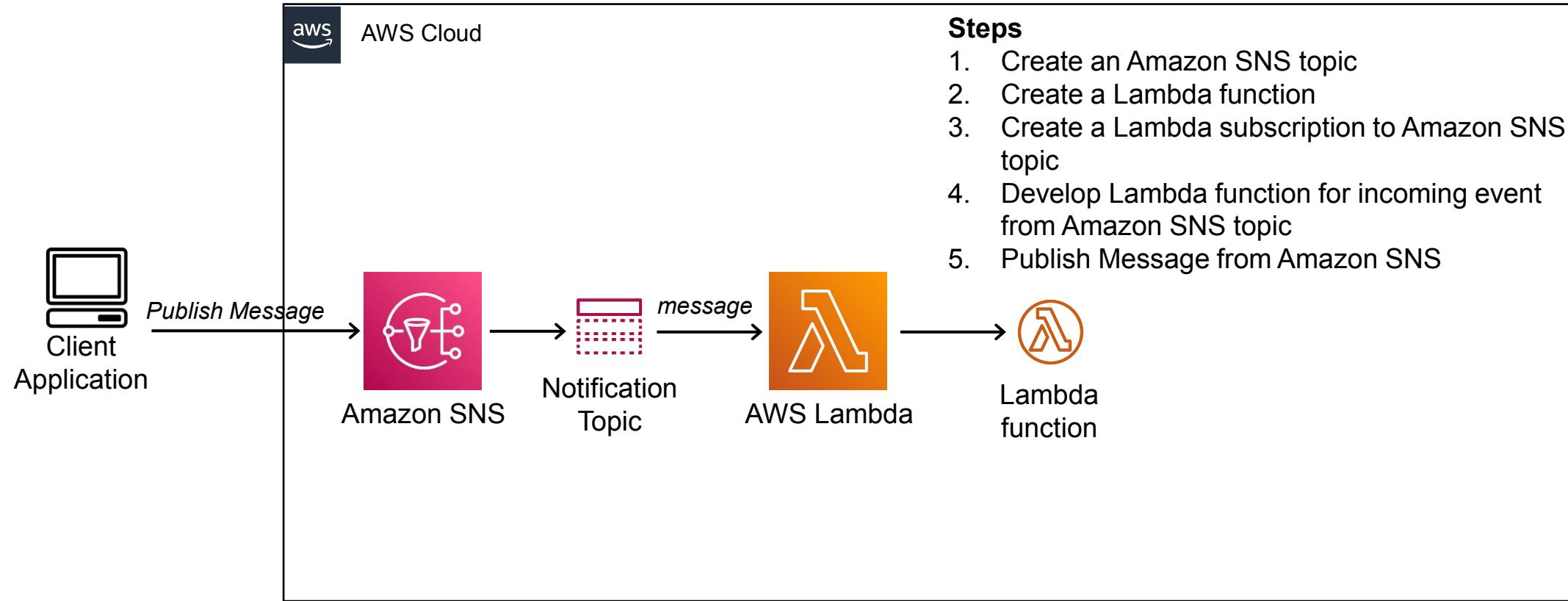
- Delete AWS Resources that we create during the section.



Hands-on Lab: Amazon SNS Notifications Subscribe From Lambda

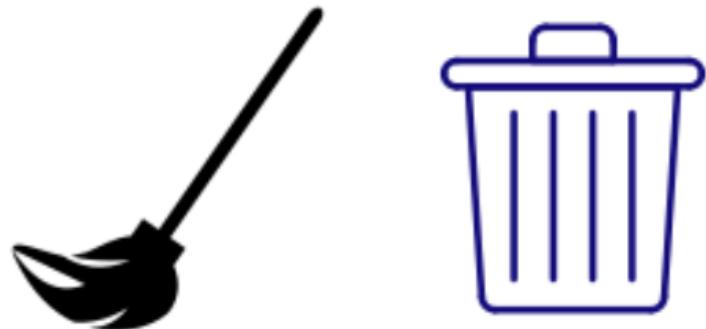
→ Developing Hands-on Lab: Amazon SNS Notifications Subscribe From AWS Lambda

Hands-on Lab: Amazon SNS Notifications Subscribe From AWS Lambda



Clean up Resources

- Delete AWS Resources that we create during the section.



Amazon SQS - Fully Managed Message Queues

- Learning Amazon SQS - Fully Managed Message Queues
 - How we can use SQS as a Queue for implementing queue patterns in ordering microservices with using Event-Driven Architecture best practices.

What is Amazon SQS ?



Amazon SQS

- Amazon SQS stands for **Simple Queue Service** is **fully managed message queues** for microservices, distributed systems, and serverless applications.
- Enables you to **decouple** and **scale microservices**, distributed systems, and serverless applications.
- **Eliminates the complexity** and overhead associated with managing and operating **message-oriented middleware**.
- **Send, store, and receive messages** between software components at any volume.
- Two types of message queues.
 - **Standard queues** offer maximum throughput, best-effort ordering, and at-least-once delivery.
 - **FIFO queues** are designed to guarantee that messages are processed exactly once, in the exact order that they are sent.
- **Integrate and decouple** distributed software systems and components.
- Provides a **generic web services API** that you can access using any programming language that the **AWS SDK** supports.

Benefits of Amazon SQS



Amazon SQS

- **Eliminate administrative overhead**

AWS manages all ongoing operations and underlying infrastructure needed to provide a highly available and scalable message queuing service. SQS queues are dynamically created and scale automatically.

- **Durability and Reliability deliver messages**

Amazon SQS stores them on multiple servers. Standard queues support at-least-once message delivery, and FIFO queues support exactly-once message processing. SQS locks your messages during processing, so that multiple producers can send and multiple consumers can receive messages at the same time.

- **Scalability and Availability and cost-effectively**

SQS scales elastically with your application so you don't have to worry about capacity planning and pre-provisioning. There is no limit to the number of messages per queue, and standard queues provide nearly unlimited throughput.

- **Security - Keep sensitive data secure**

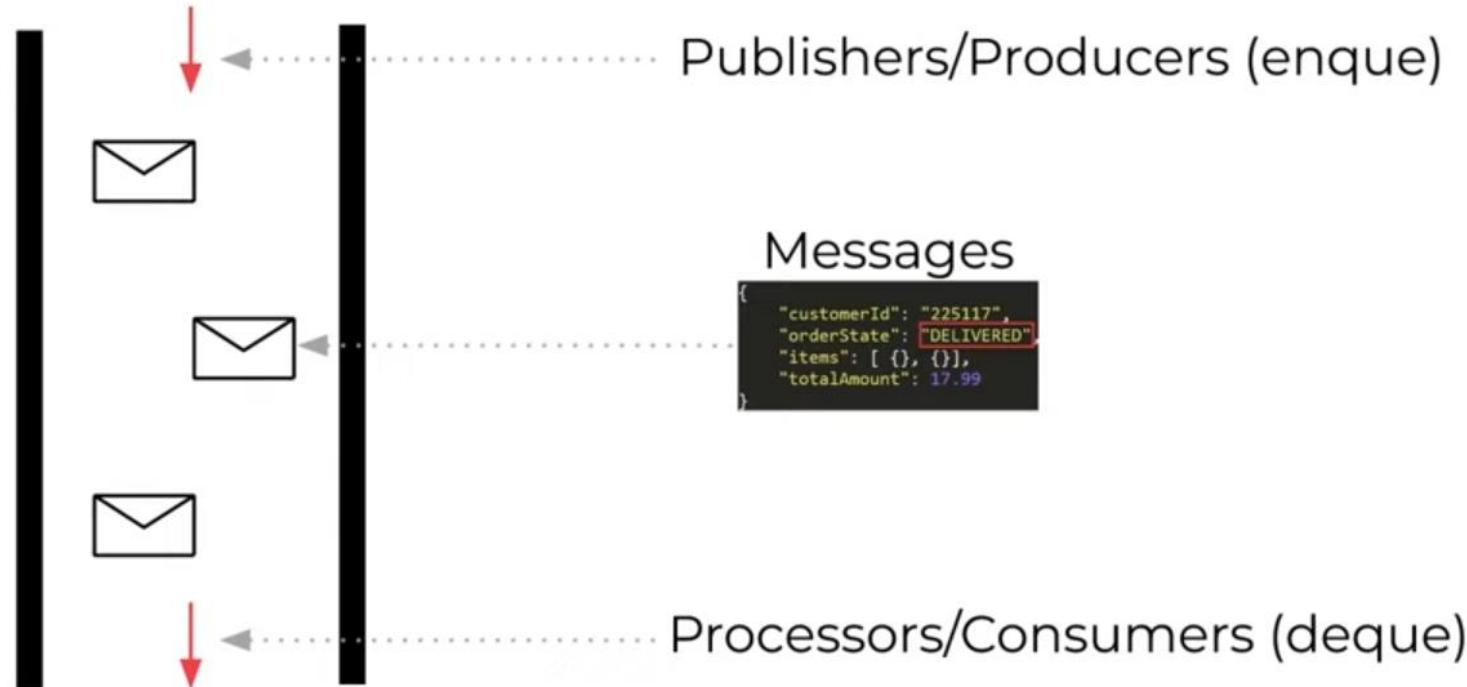
Amazon SQS to exchange sensitive data between applications using server-side encryption (SSE) to encrypt each message body.

Amazon SQS architecture and How SQS works



Amazon SQS

Queue



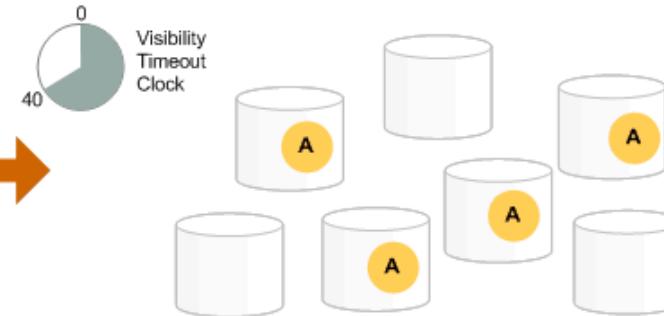
<https://www.youtube.com/watch?v=CyYZ3adwboc&t=152s>

The lifecycle of an Amazon SQS message



Amazon SQS

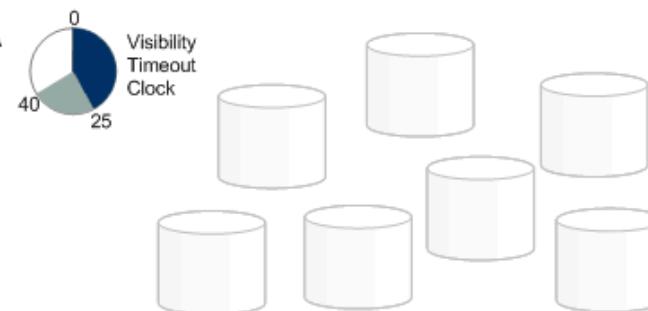
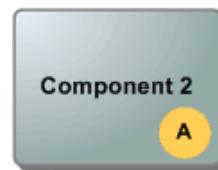
- 1 Component 1 sends Message A to the queue



- 2 Component 2 retrieves Message A from the queue and the visibility timeout period starts

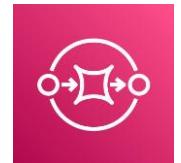


- 3 Component 2 processes Message A and then deletes it from the queue during the visibility timeout period



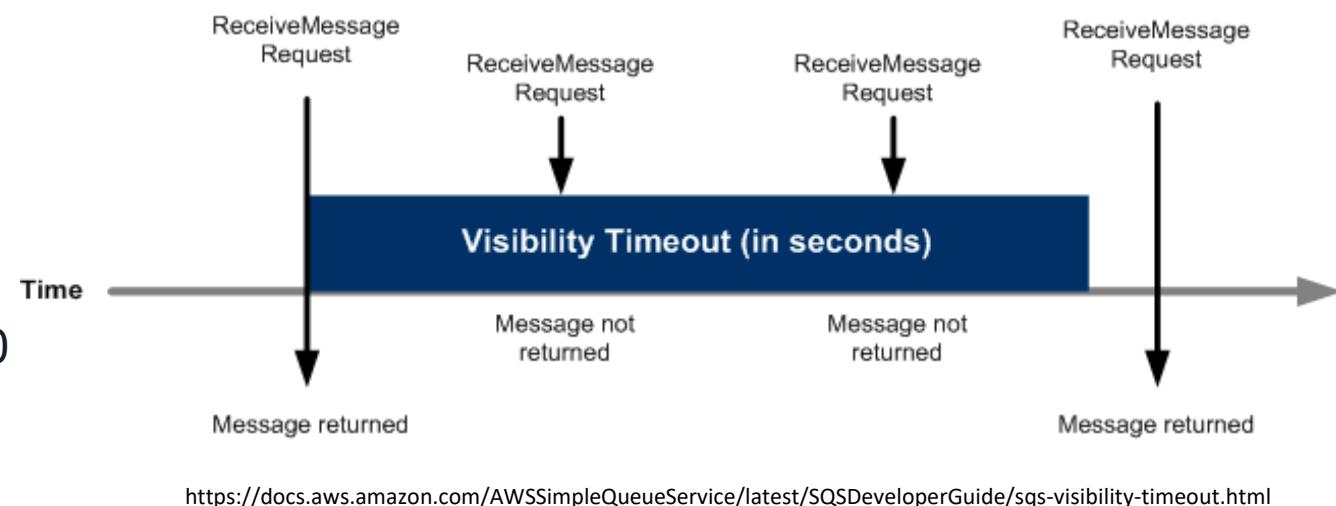
<https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-basic-architecture.html>

Amazon SQS Visibility Timeout

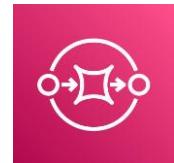


Amazon SQS

- When a **consumer receives** and **processes** a message from a **queue**, the message remains in the queue.
- the consumer **must delete the message** from the queue after receiving and processing it.
- **Visibility timeout**; a period of time during which Amazon SQS prevents other consumers from receiving and processing the message.
- The **default visibility timeout** for a message is 30 seconds.
- Configuring **visibility timeout** for a **queue** using the console.

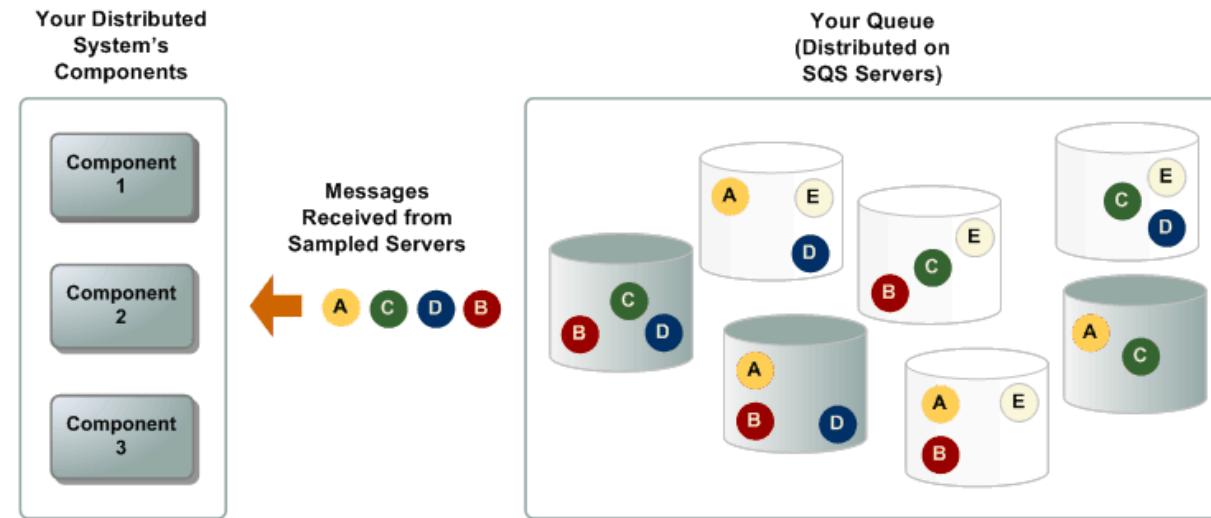


Amazon SQS Short and Long polling



Amazon SQS

- By default, queues use **short polling**.
- **Short polling**, the **ReceiveMessage request** queries only a subset of the servers to find messages that are available to include in the response.
- **Long polling**, the **ReceiveMessage request** queries all of the servers for messages. Amazon SQS sends a response after it collects at least one available message.
- Consume messages using **short polling**, Amazon SQS samples a **subset** of its servers and returns messages from only those servers.
- When the **wait time** for the **ReceiveMessage API** action is greater than 0, long polling is in effect. Long polling helps **reduce the cost of using Amazon SQS** by eliminating the number of empty responses.



<https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-short-and-long-polling.html>

Amazon SQS Dead-letter Queues



Amazon SQS

- Amazon SQS supports **dead-letter queues (DLQ)**
- Messages **can't be processed** because of a variety of possible issues.
- **Example:** User places a web order with a particular product ID, but the product ID is deleted. Code fails and displays an error, and the message with the order request is sent to a **dead-letter queue**.
- The main task of a dead-letter queue is to **handle the lifecycle of unconsumed messages**.
- A **dead-letter queue** lets you set **aside** and **isolate** messages that can't be processed correctly to determine why their processing didn't succeed.
- Setting up a dead-letter queue
 - Configure an alarm for any messages moved to a dead-letter queue.
 - Examine logs for exceptions that might have caused messages to be moved to a dead-letter queue.
 - Analyze the contents of messages moved to a dead-letter queue.
 - Determine whether you have given your consumer sufficient time to process messages.

SQS Queue types - Standard Queues and FIFO Queues



Amazon SQS

- SQS Provides two types of Message Queue
 - Standard Queue
 - FIFO Queue
- **Standard Queue** is the default queue type offered by AWS SQS. This provides unlimited throughput, best effort ordering and at least once message delivery.
- **FIFO Queue** is simply means that messages will be ordered in the queue and first message to arrive in the queue will be first to leave the queue.



Best Effort Ordering
At Least Once Delivery
Unlimited Throughput



First In First Out Ordering
Exactly Once Processing
300 TPS Max (3000 With Batching)
Slightly (~25%) more expensive

<https://aws.amazon.com/sqs/features/>

SQS Queue types - Standard Queues and FIFO Queues



Amazon SQS

- **Message Ordering**

SQS Standard queues provide best-effort ordering. Occasionally more than one copy of a message might be delivered out of order. FIFO queues offers first-in-first-out delivery.

- **Delivery**

Standard Queue guarantees at least once delivery but sometimes duplicates or more than one copy of a message can be delivered. FIFO queues ensure a message is delivered exactly once.

- **Throughput**

Standard queues offers unlimited throughput. FIFO queues on the other hand are limited to 300 transactions per second per API action.

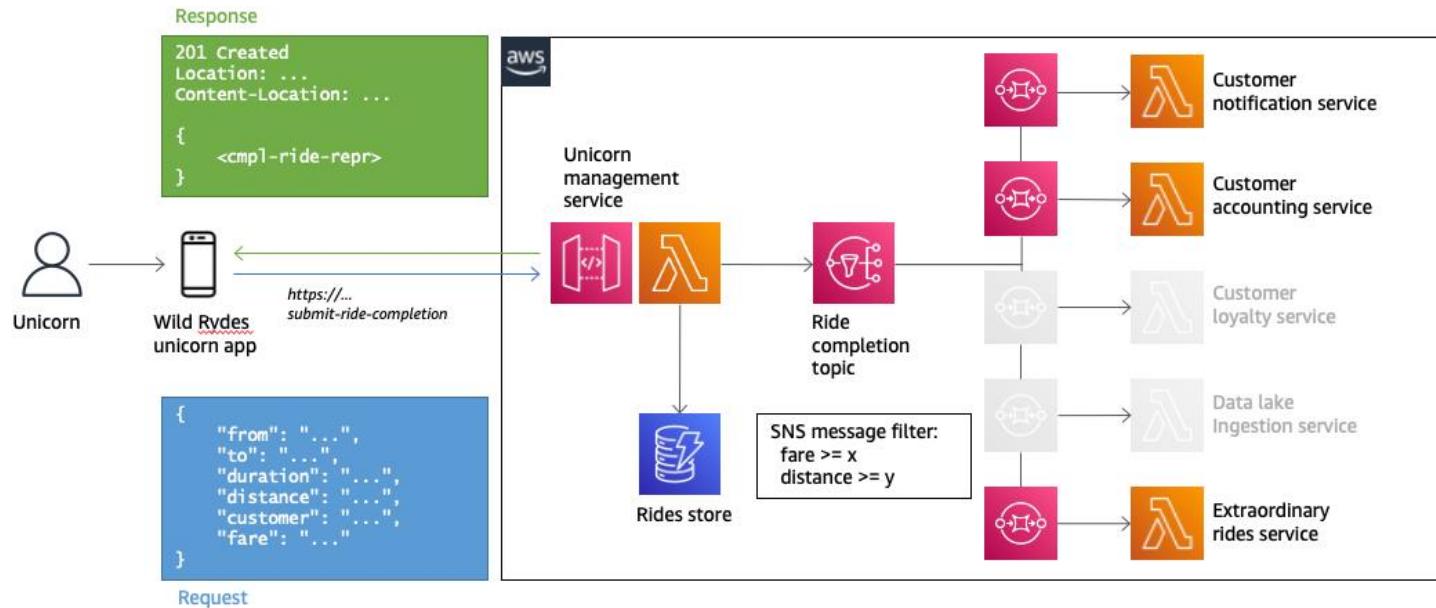
- **Use cases**

Use Standard queue as long as you are able to process duplicates and out of order messages. FIFO queues can be used when ordering of message is must and duplicates are not accepted at any cost.

- Standard queues provide at-least-once delivery, which means that each message is delivered at least once.
- FIFO queues provide exactly-once processing.

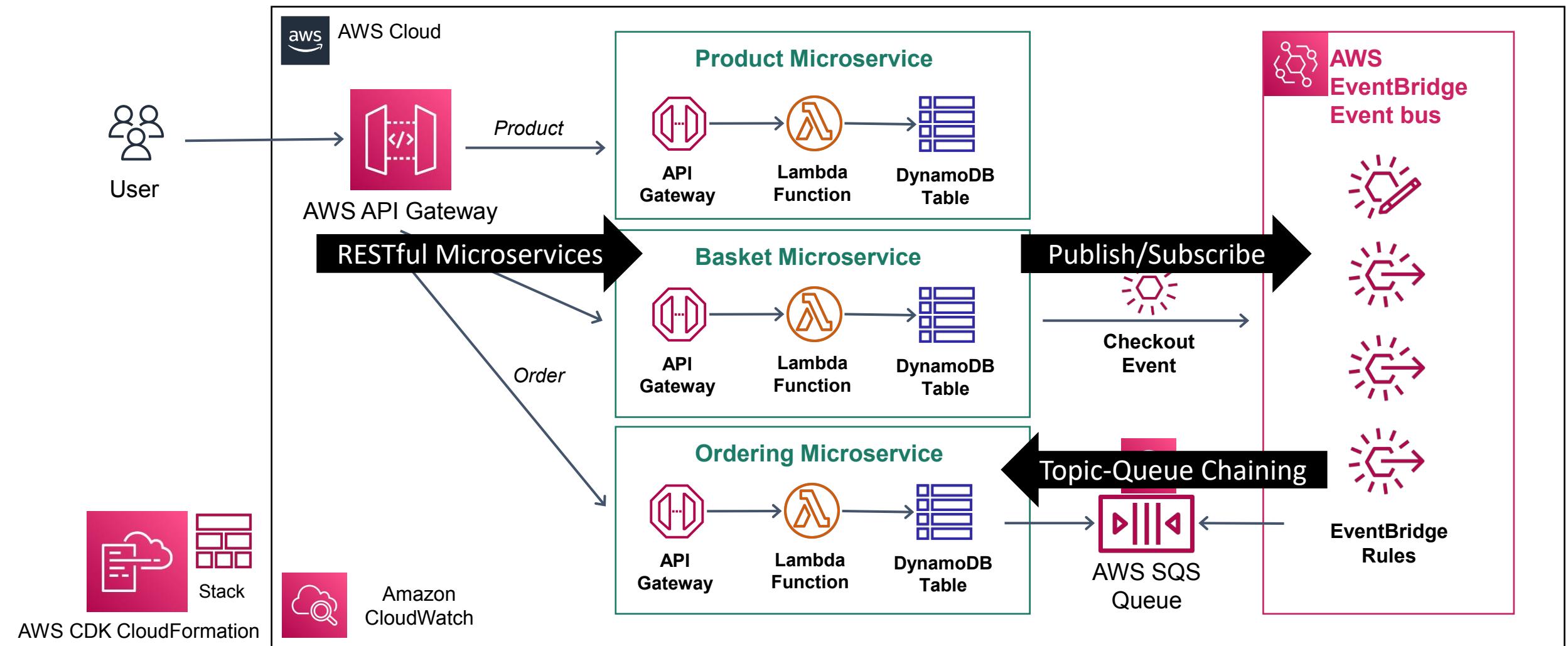
Topic-Queue Chaining & Load Balancing Pattern

- Use a **queue** that acts as a **buffer** between the service to **avoid loss data** if the service to fail.
- **Services** can be **down** or **getting exception** or taken offline for maintenance, then **events** will be **loses**, **disappeared** and can't process after the subscriber service is up and running.
- Put **Amazon SQS** between **EventBridge** and **Ordering** microservices.
- Store this event messages into **SQS queue** with **durable** and **persistent** manner, no message will get lost
- **Queue** can act as a buffering **load balancer**



<https://async-messaging.workshop.aws/fan-out-and-message-filtering.html>

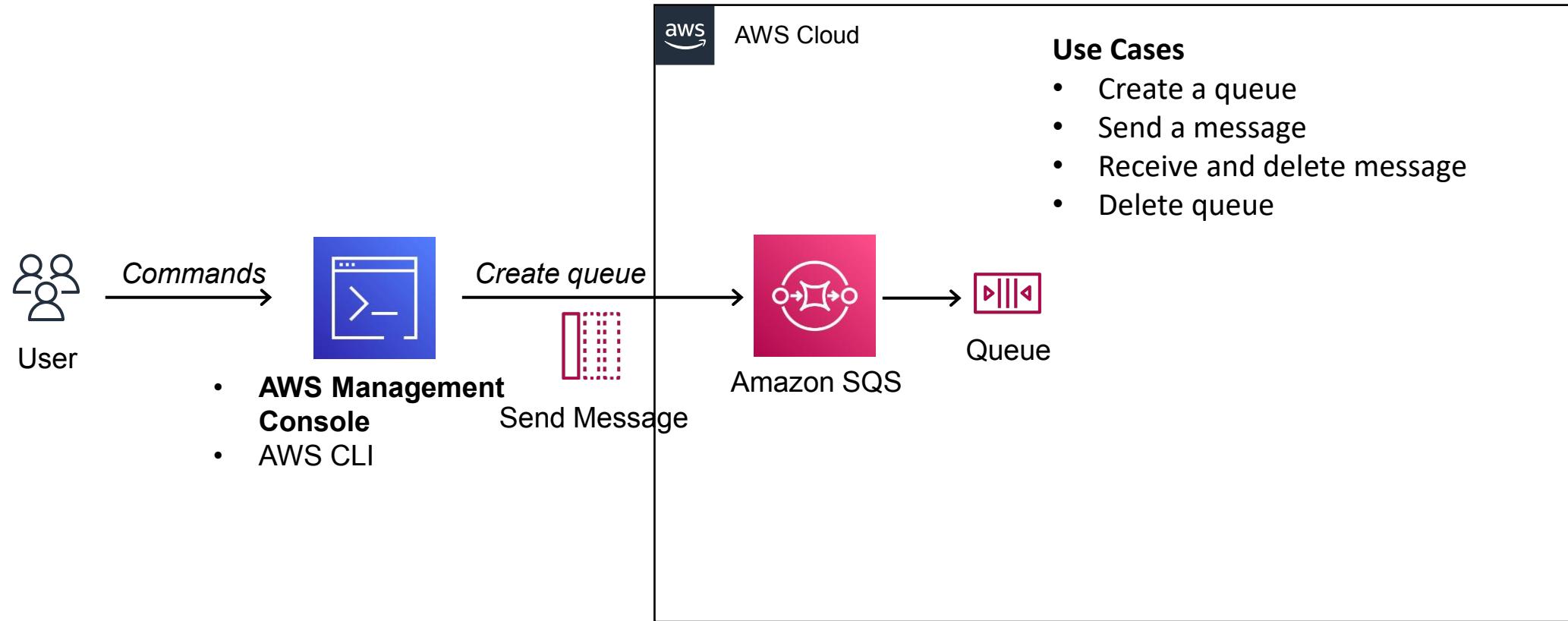
Serverless Patterns for Microservices



Amazon SQS - Walkthrough with AWS Management Console

→ DEMO - Amazon SQS - Walkthrough with AWS Management
Console

Getting started with Amazon SQS with AWS Management Console



AWS Lambda & Serverless Course Learning Path

1

Theoretical Information

AWS Service overview, core concepts, features, uses cases and general information

2

Walkthrough with AWS Console

AWS Service Walkthrough with AWS Management Console performs main use cases

3

Developing with AWS SDK

AWS Service Programmatic Access interaction with Serverless APIs using AWS SDK or CLI

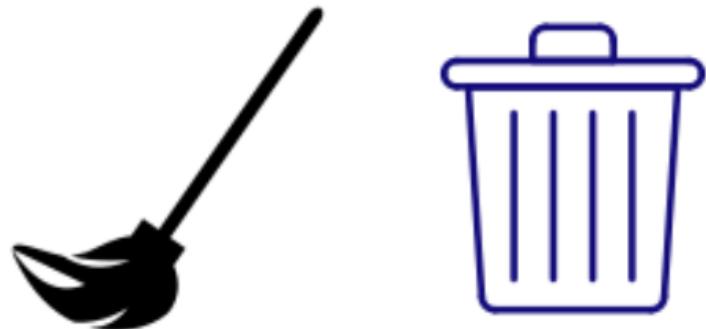
4

Hands-on Labs Real-World Apps

AWS Service Hands-on Labs implementation with Real-World Use Cases

Clean up Resources

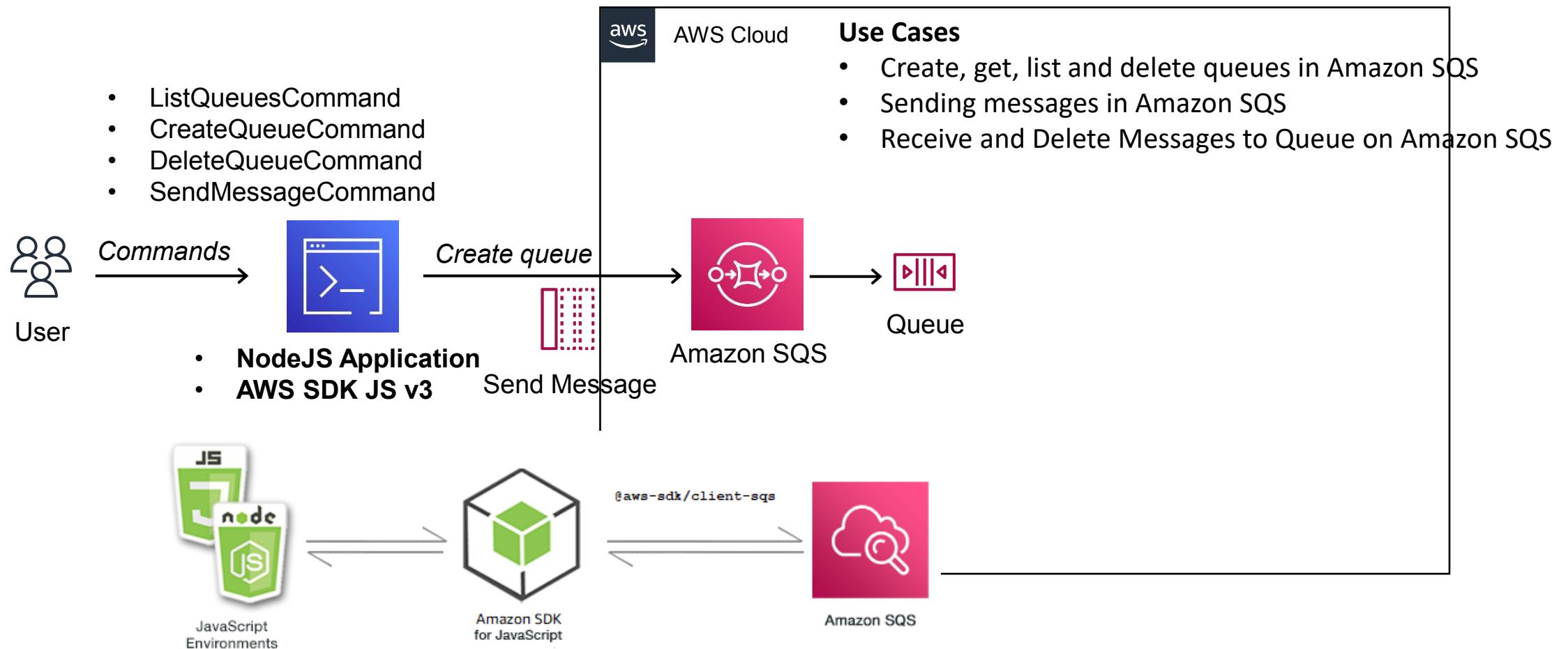
- Delete AWS Resources that we create during the section.



Amazon SQS - Developing with AWS SDK

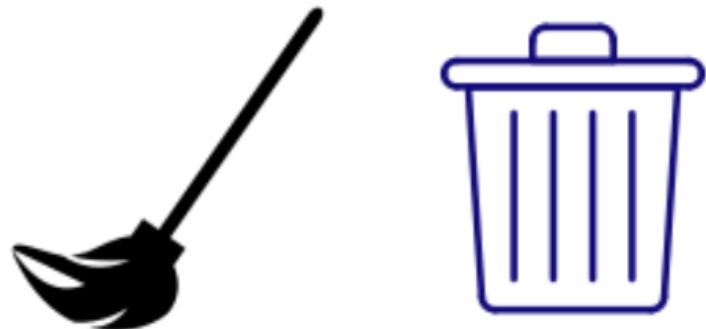
→ Amazon SQS - Developing with AWS SDK interaction to Serverless APIs Programmatic Access

Amazon SQS SDK Examples using AWS SDK Javascript v3



Clean up Resources

- Delete AWS Resources that we create during the section.

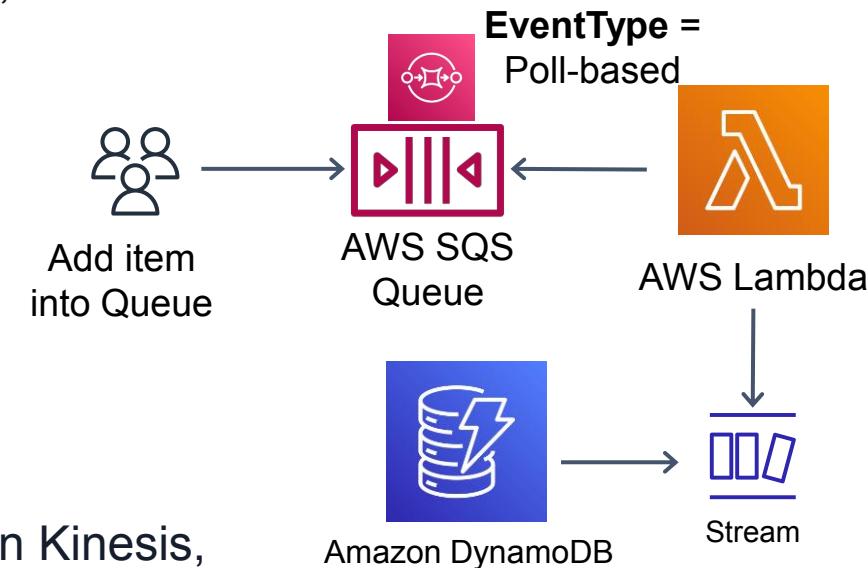


Hands-on Lab: Amazon SQS Queue Polling From AWS Lambda

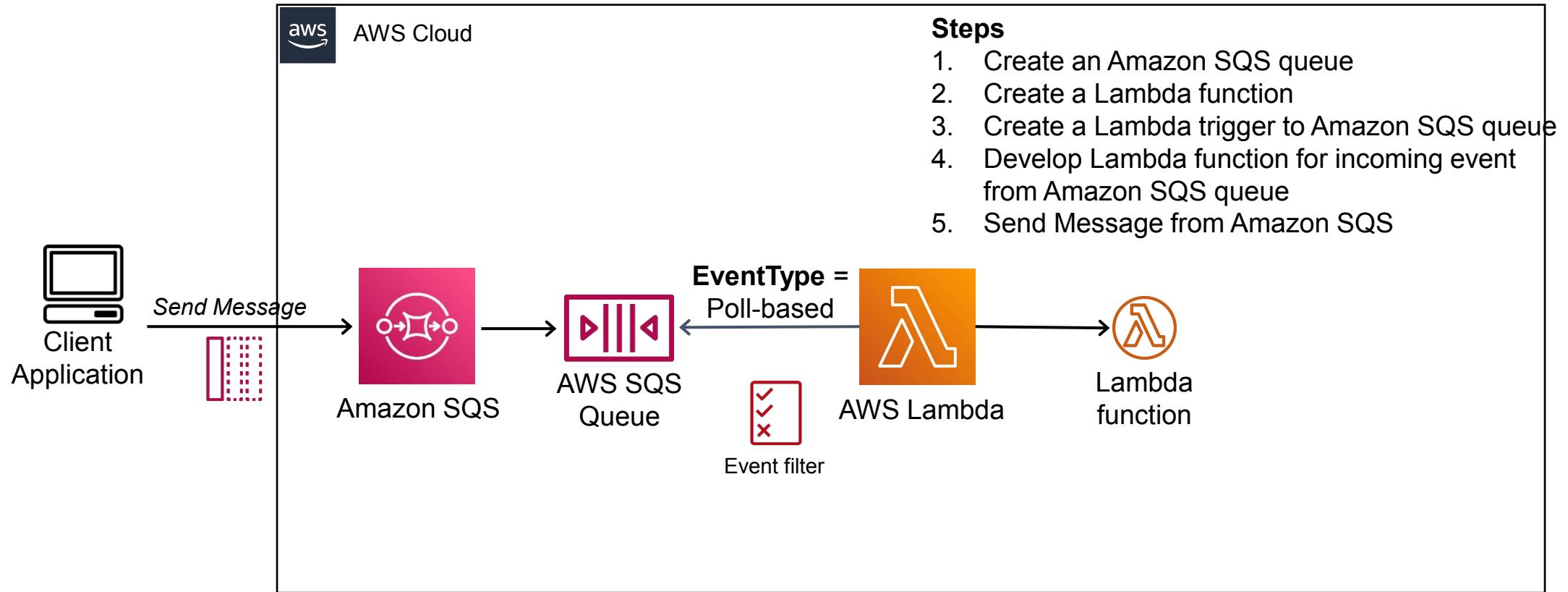
→ Developing Hands-on Lab: Amazon SQS Queue Polling From AWS Lambda

AWS Lambda Event Source Mapping with Polling Invocation

- **Pool-Based invocation** model allows us to **integrate** with **AWS Stream** and **Queue based services**.
- Lambda will **poll** from the AWS SQS or Kinesis streams, retrieve records, and invoke functions.
- Data stream or queue are **read in batches**,
- The function receives multiple items when execute function.
- **Batch sizes** can configure according to service types
- **SQS + Lambda**
- **Stream based processing** with **DynamoDB Streams + Lambda**
- Triggered **AWS services** of **Event Source Mapping invocation**; Amazon Kinesis, DynamoDB, Simple Queue Service (SQS)



Hands-on Lab: Amazon SQS Queue Polling From AWS Lambda



Serverless Project Development Phases

1

Infrastructure Creation on AWS

Create API Gateway, Lambda Function and DynamoDB table on AWS Cloud - Also we can automate this part with IaC using CDK in the last sections but now we will create infrastructure with console or cli

2

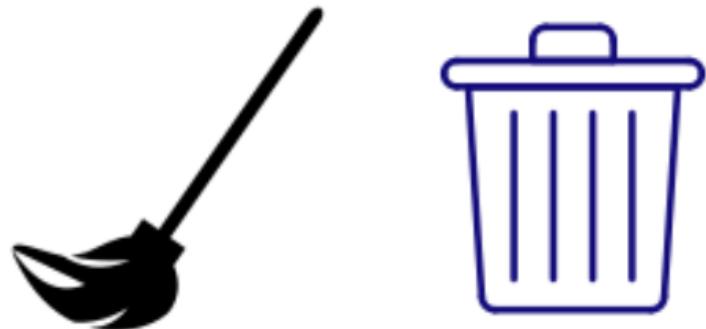
Develop Lambda + SQS business logic with AWS SDK

Use AWS SDK JS v3 with ES6 standards to implement crud functions into lambda function.



Clean up Resources

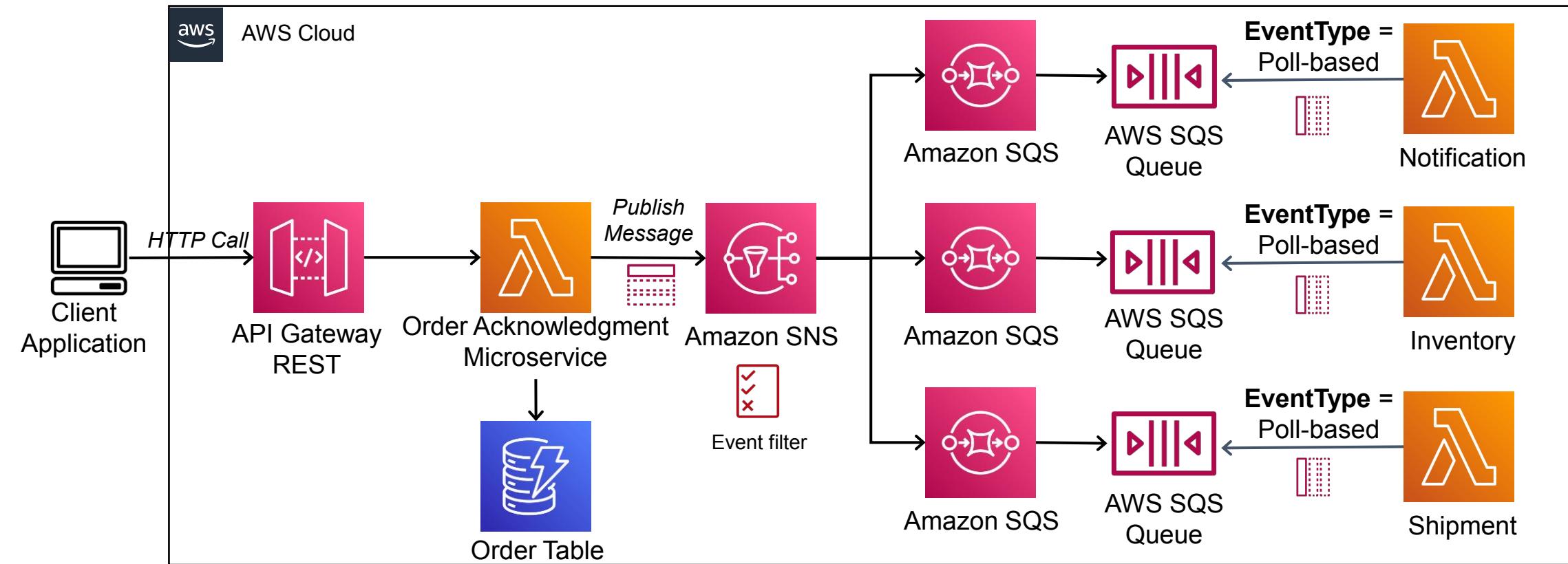
- Delete AWS Resources that we create during the section.



Hands-on Labs : Fan-Out Serverless Architectures Using SNS, SQS and Lambda

→ Developing Hands-on Labs : Fan-Out Serverless Architectures Using SNS, SQS and Lambda

Hands-on Lab: Fan-Out Serverless Architectures Using SNS, SQS and Lambda



Serverless Project Development Phases

1

Infrastructure Creation on AWS

Create API Gateway, Lambda Function and DynamoDB table on AWS Cloud - Also we can automate this part with IaC using CDK in the last sections but now we will create infrastructure with console or cli

2

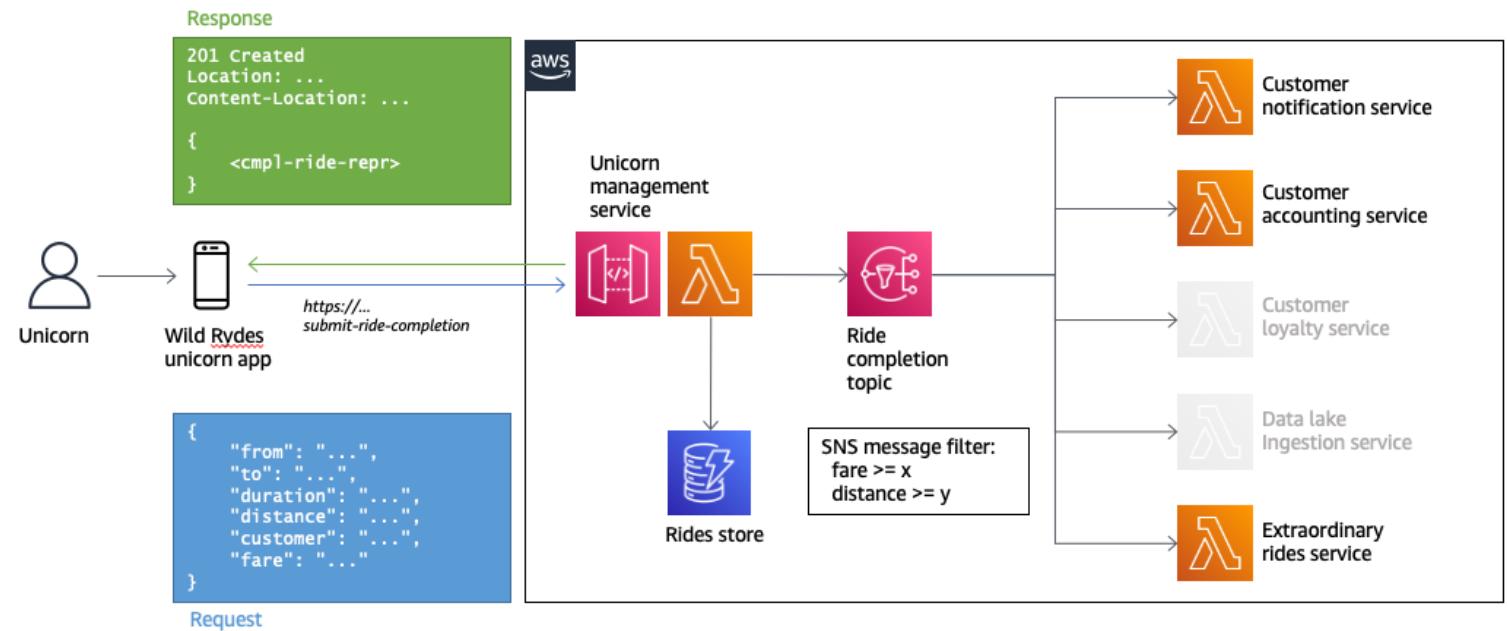
Develop Lambda + SQS business logic with AWS SDK

Use AWS SDK JS v3 with ES6 standards to implement crud functions into lambda function.



Fan-Out & Message Filtering with Publish/Subscribe Pattern

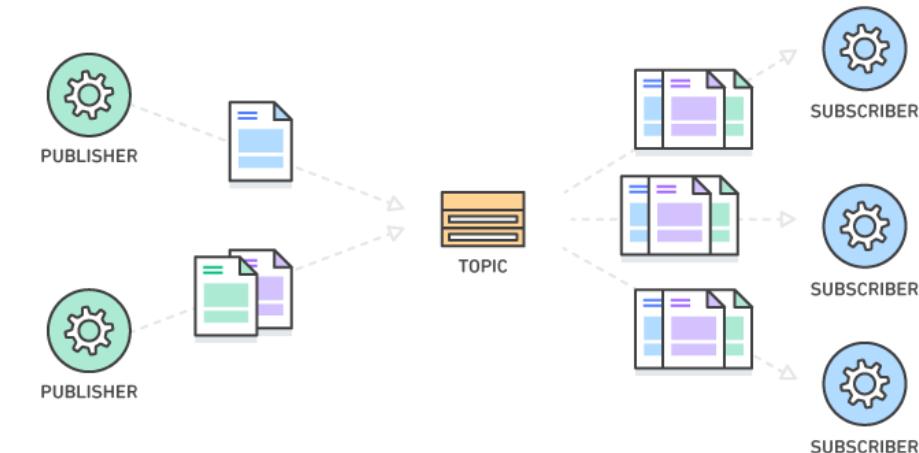
- **Async Communication** for performing one-to-many and publish/subscribe mechanisms.
- Client service **publish a message** and it **consumes from several microservices** which's are subscribing this message on the message broker system.
- **Decouples Messaging** between services, building loosely-coupled architectures.
- Using in **event-driven** architectures
- **Publishes** an event something happen:
- Price change in a product microservice
- **Subscribed** from SC microservice to update basket price



<https://async-messaging.workshop.aws/fan-out-and-message-filtering.html>

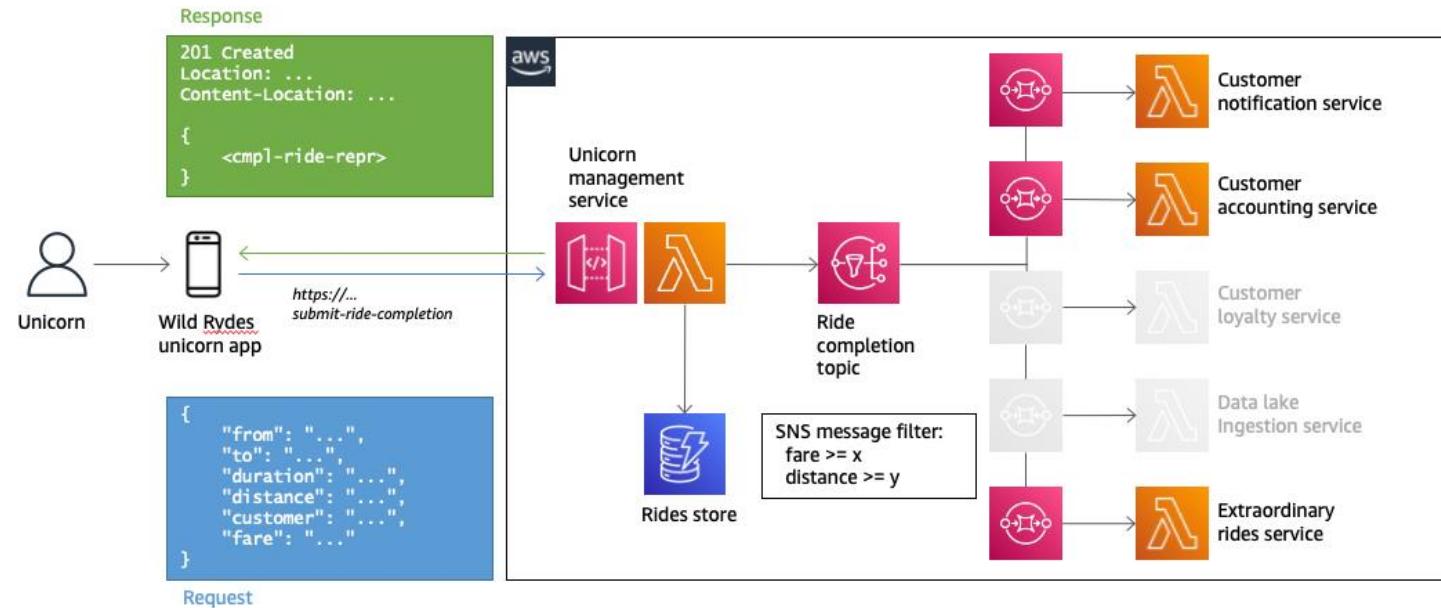
Publish/Subscribe Messaging Pattern

- Publish/subscribe messaging, or pub/sub messaging, is a form of asynchronous service-to-service communication used in serverless and microservices architectures.
- Any message published to a topic is immediately received by all of the subscribers to the topic.
- Enable event-driven architectures, or to decouple applications in order to increase performance, reliability and scalability.
- Applications are decoupled into smaller, independent building blocks that are easier to develop, deploy and maintain.
- Publish/Subscribe (Pub/Sub) messaging provides instant event notifications for these distributed applications
- All components that subscribe to the topic receive every message that is broadcast, unless a message filtering policy is set by the subscriber.



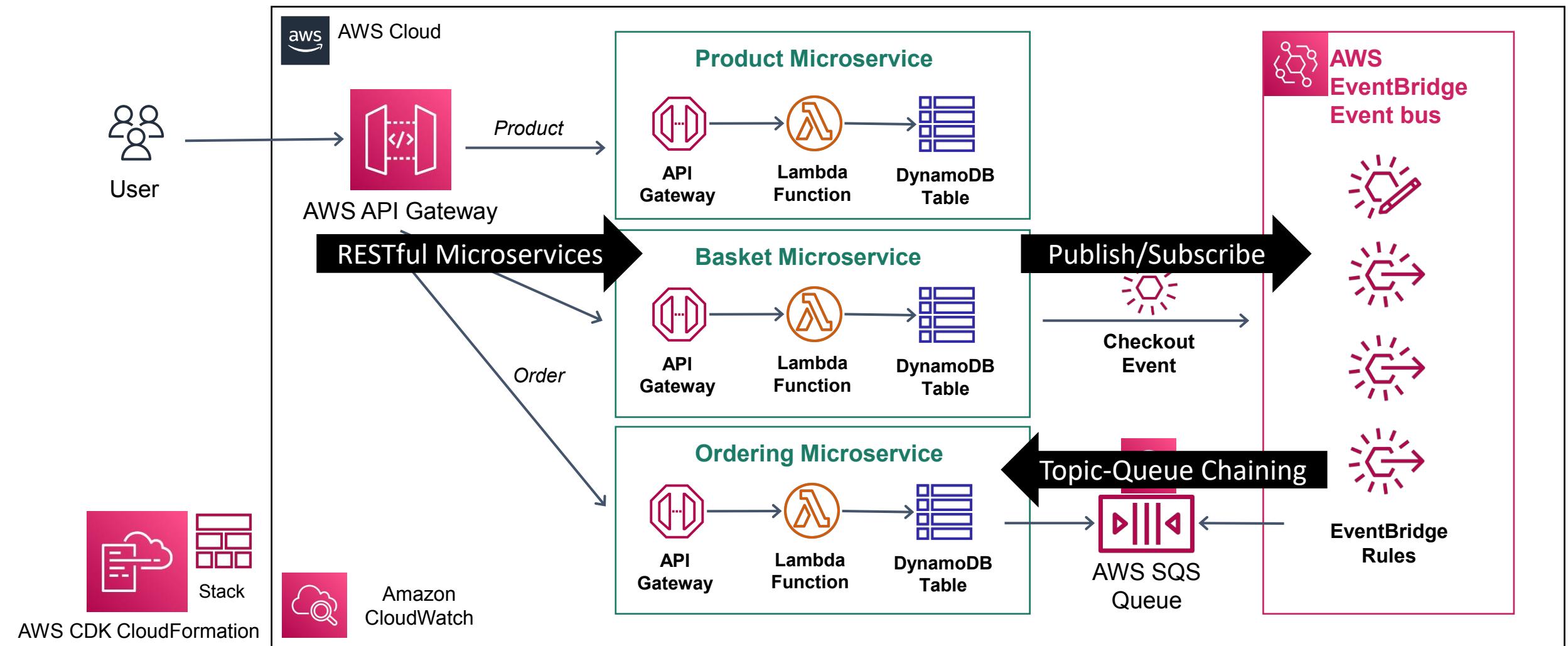
Topic-Queue Chaining & Load Balancing Pattern

- Use a **queue** that acts as a **buffer** between the service to **avoid loss data** if the service to fail.
- **Services** can be **down** or **getting exception** or taken offline for maintenance, then **events** will be **loses**, **disappeared** and can't process after the subscriber service is up and running.
- Put **Amazon SQS** between **EventBridge** and **Ordering** microservices.
- Store this event messages into **SQS queue** with **durable** and **persistent** manner, no message will get lost
- **Queue** can act as a buffering **load balancer**



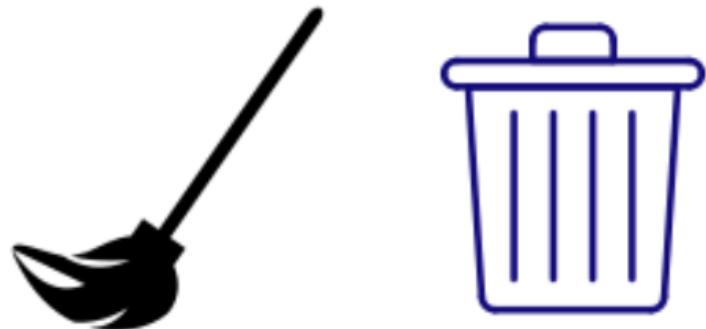
<https://async-messaging.workshop.aws/fan-out-and-message-filtering.html>

Serverless Patterns for Microservices



Clean up Resources

- Delete AWS Resources that we create during the section.

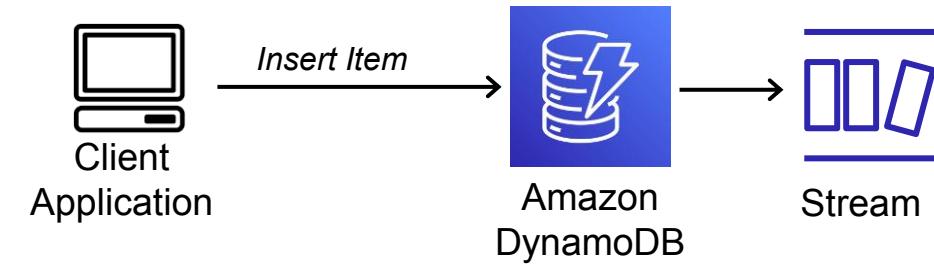


DynamoDB Streams — Using AWS Lambda to Process DynamoDB Streams

- DynamoDB Streams — Using AWS Lambda to Process DynamoDB Streams for Change Data Capture of DynamoDB Tables

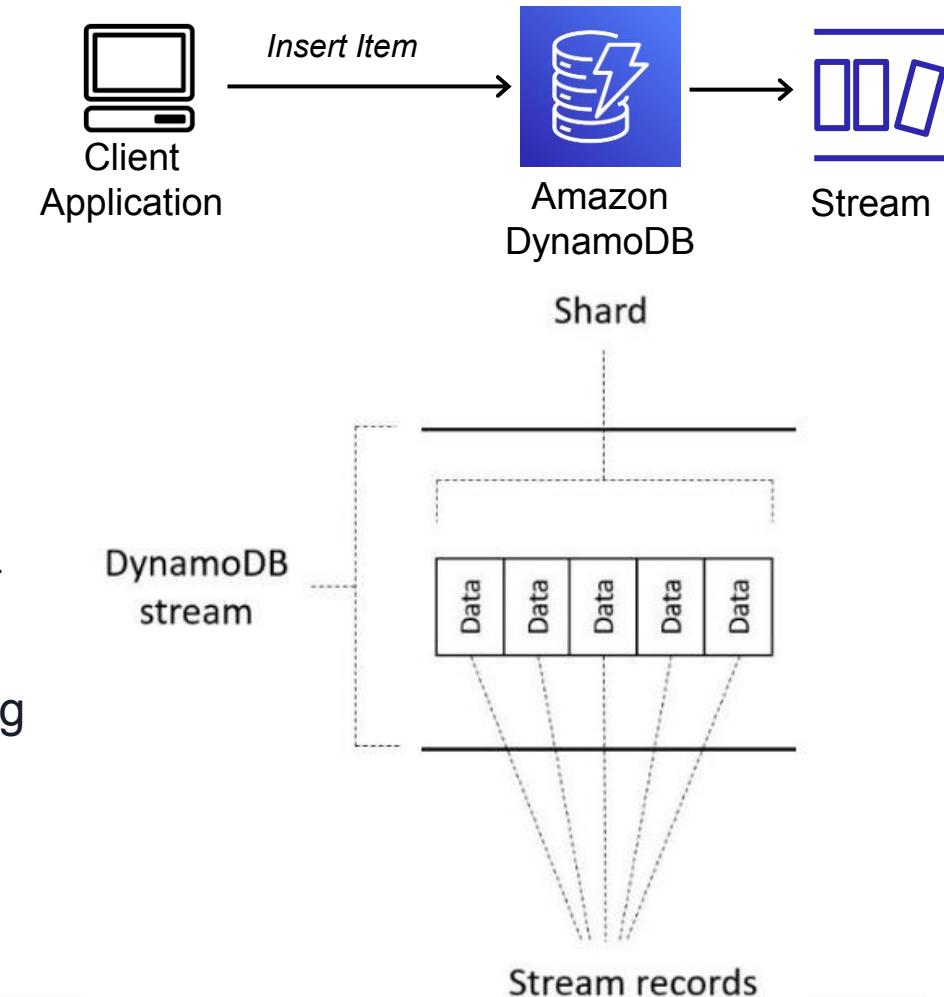
Working with Streams on Amazon DynamoDB

- DynamoDB supports streaming of change data capture records in the near-real time.
- Most of the applications can benefit from data capturing changes into DynamoDB table.
- New Customer Added into DynamoDB Table. Invokes send welcome email to customer.
- Application sends notifications to the mobile devices for user interactions.
- Mobile application modifies data in a DynamoDB table, it could be view count of YouTube video or like pictures on Instagram.
- Financial application modifies stock market data in a DynamoDB table.



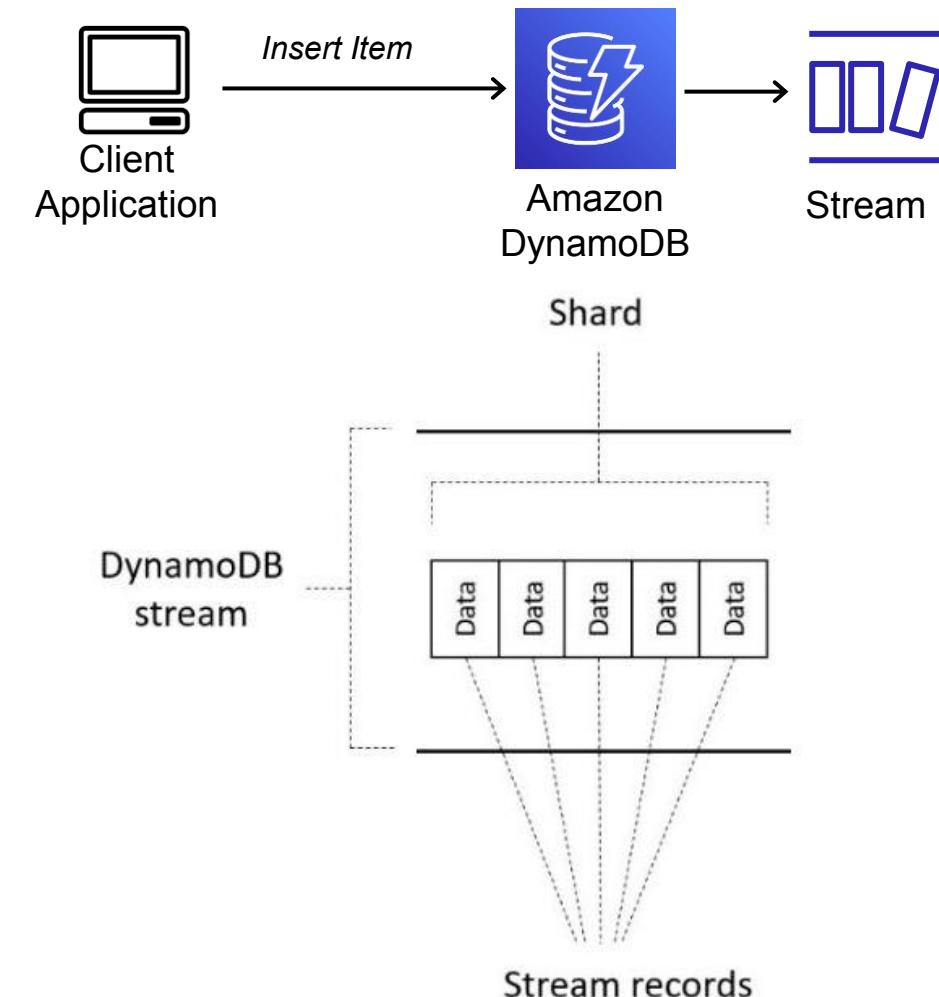
Working with Streams on Amazon DynamoDB – Part2

- When we enable a stream on a table, DynamoDB captures information about every modification to data items in the table.
- Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attributes of the items that were modified.
- **Endpoints for DynamoDB Streams;** AWS separate endpoints for DynamoDB and DynamoDB Streams. The AWS SDKs provide separate clients for DynamoDB and DynamoDB Streams.
- Connect to both endpoints, Application must instantiate two clients—one for DynamoDB and one for DynamoDB Streams.
- **Enabling a Stream;** enable a stream on a new or existing table using the AWS CLI or the AWS SDKs.
- On the DynamoDB console dashboard, choose Tables and select an existing table. On the Exports and streams tab, in the DynamoDB stream details section, set Enable.

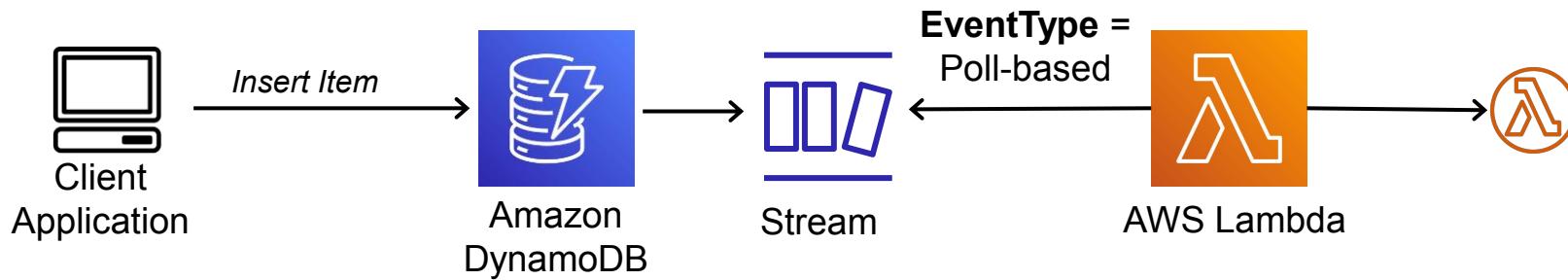


Reading and Processing Amazon DynamoDB Stream

- Read and process a stream, our application must connect to a DynamoDB Streams endpoint and send API request
- Stream records are organized into groups, or shards.
- Each shard acts as a container for multiple stream records, and contains information.
- Each stream record represents a single data modification in the DynamoDB table.
- Shards are ephemeral: They are created and deleted automatically. Any shard can also split into multiple new shards.
- The DynamoDB Streams API provides the actions;
 - ListStreams
 - DescribeStream
 - GetShardIterator
 - GetRecords



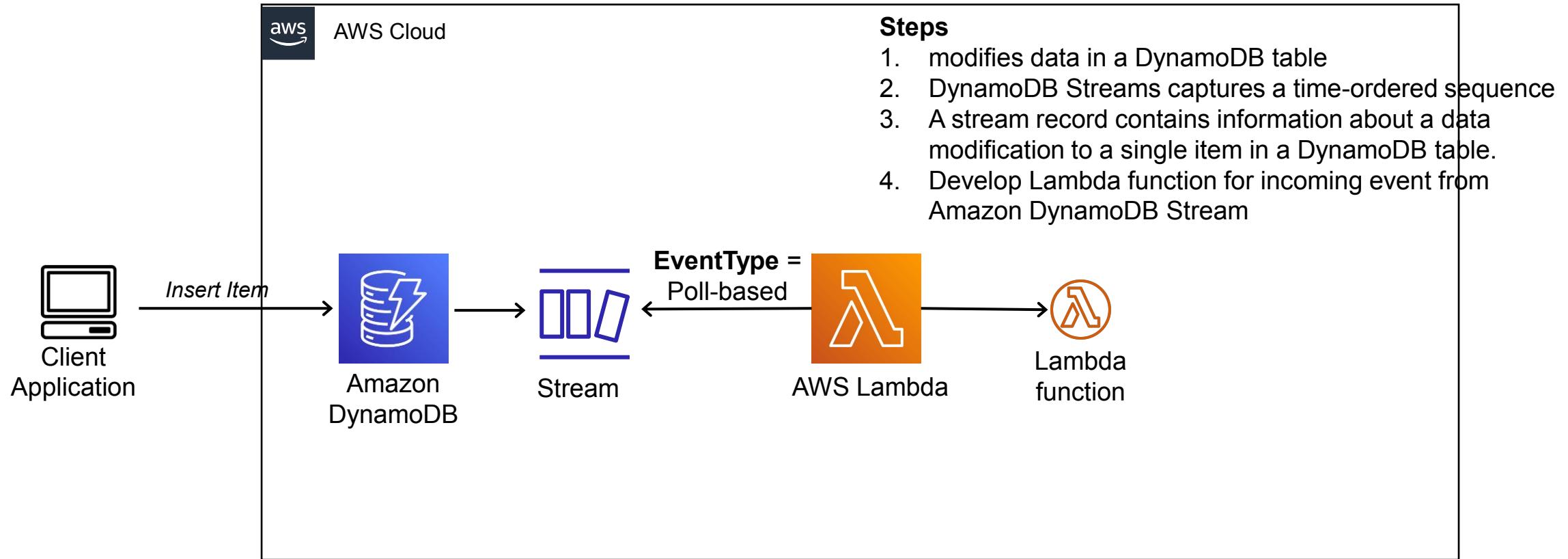
Using AWS Lambda with Amazon DynamoDB Streams



Execution role permissions
dynamodb:DescribeStream
dynamodb:GetRecords
dynamodb:GetShardIterator
dynamodb>ListStreams

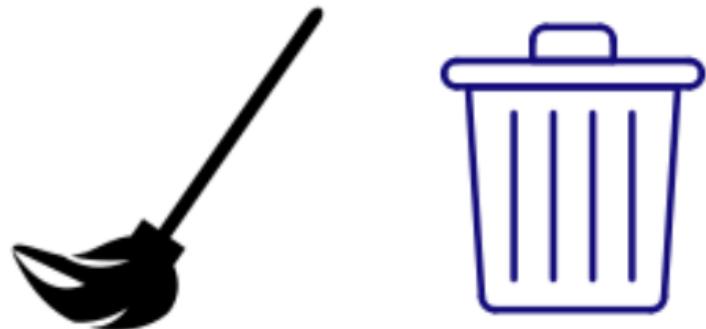
```
{  
  "Records": [  
    {  
      "eventID": "1",  
      "eventVersion": "1.0",  
      "dynamodb": {  
        "Keys": {  
          "Id": {  
            "N": "101"  
          }  
        },  
        "NewImage": {  
          "Message": {  
            "S": "New item!"  
          }  
        },  
        "Id": {  
          "N": "101"  
        }  
      },  
      "StreamViewType": "NEW_AND_OLD_IMAGES",  
      "SequenceNumber": "111",  
      "SizeBytes": 26  
    }  
  ]  
}
```

Hands-on Lab: Process DynamoDB Streams using AWS Lambda for Change Data Capture of DynamoDB Tables



Clean up Resources

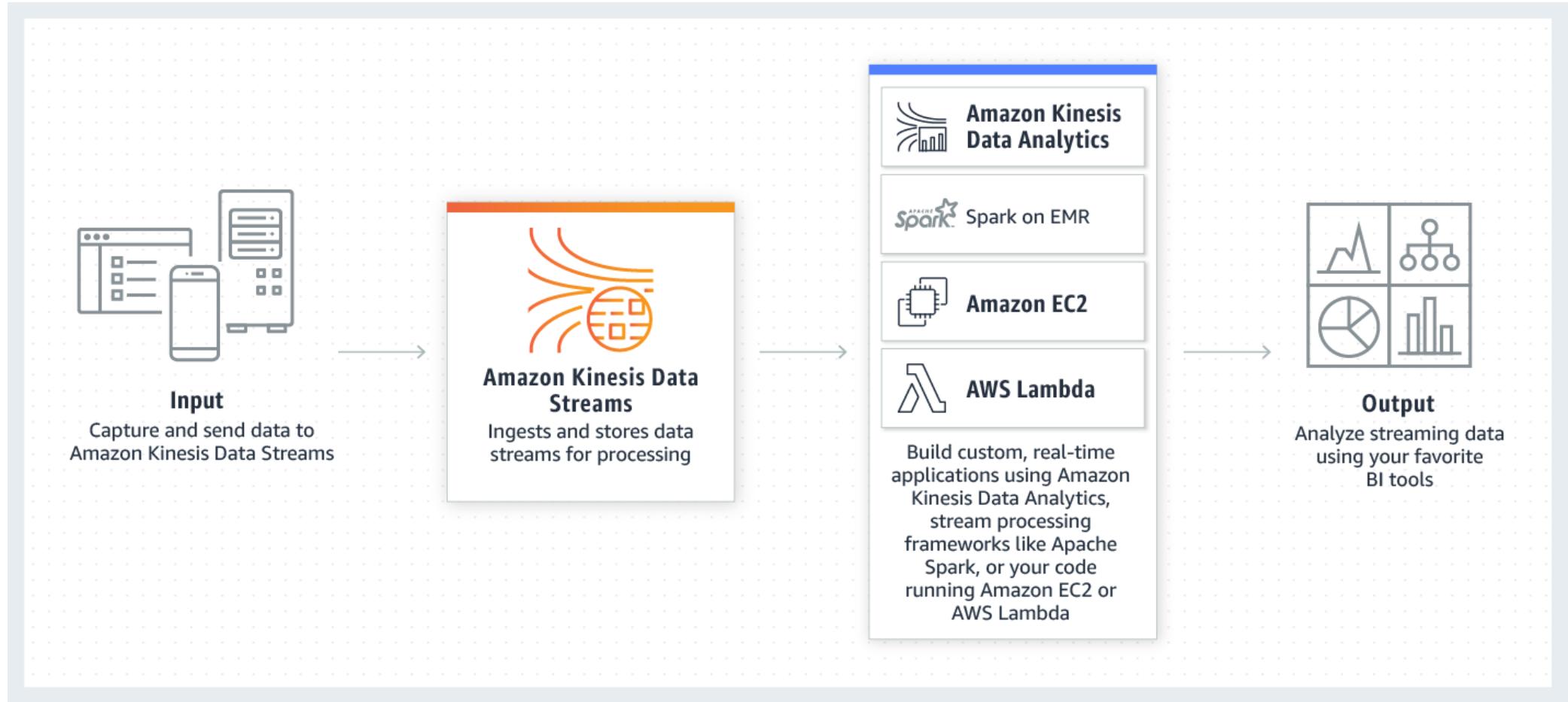
- Delete AWS Resources that we create during the section.



Kinesis Streams — Using AWS Lambda to Process Kinesis Streams

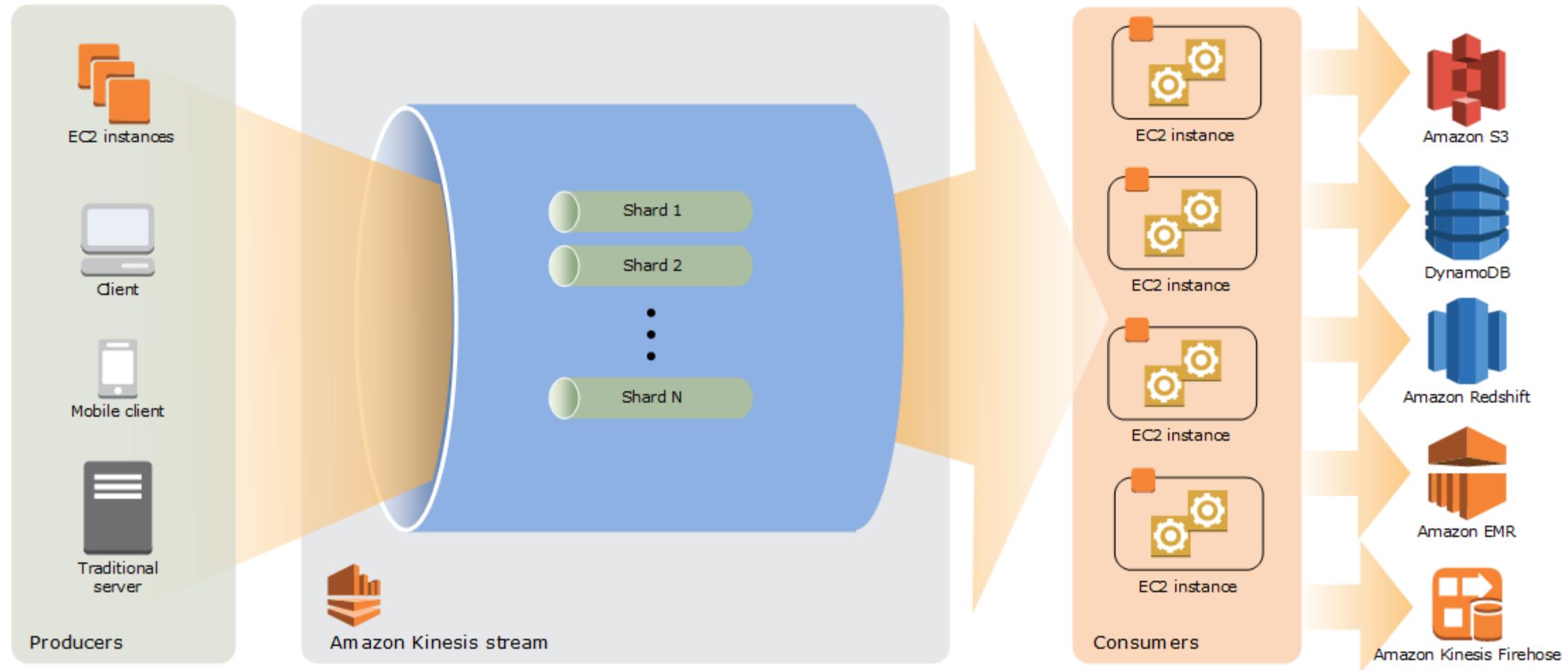
→ Developing Kinesis Streams — Using AWS Lambda to Process Kinesis Streams

What is Kinesis and Kinesis Streams ?

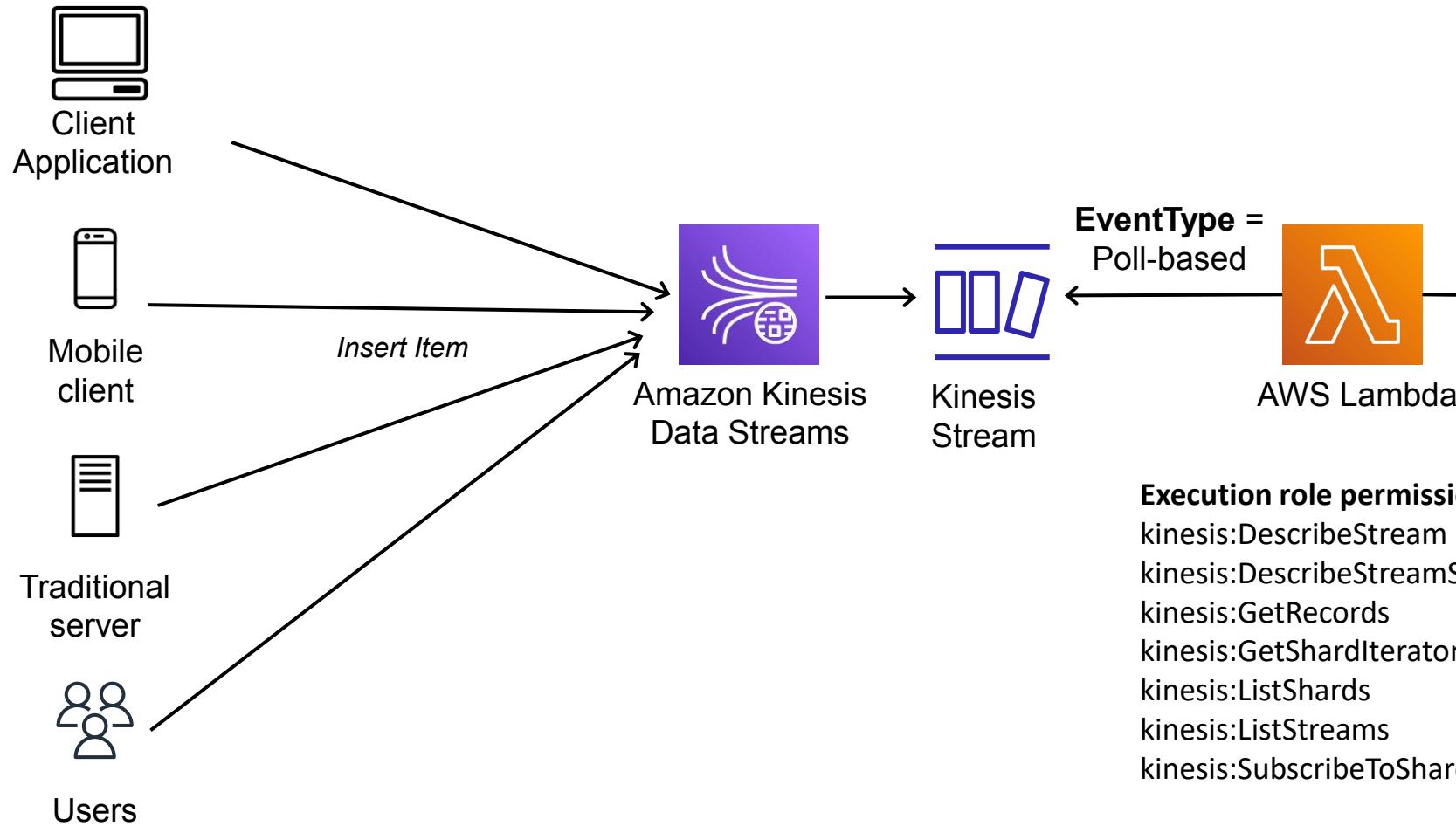


<https://aws.amazon.com/kinesis/>

Kinesis Data Streams High-Level Architecture

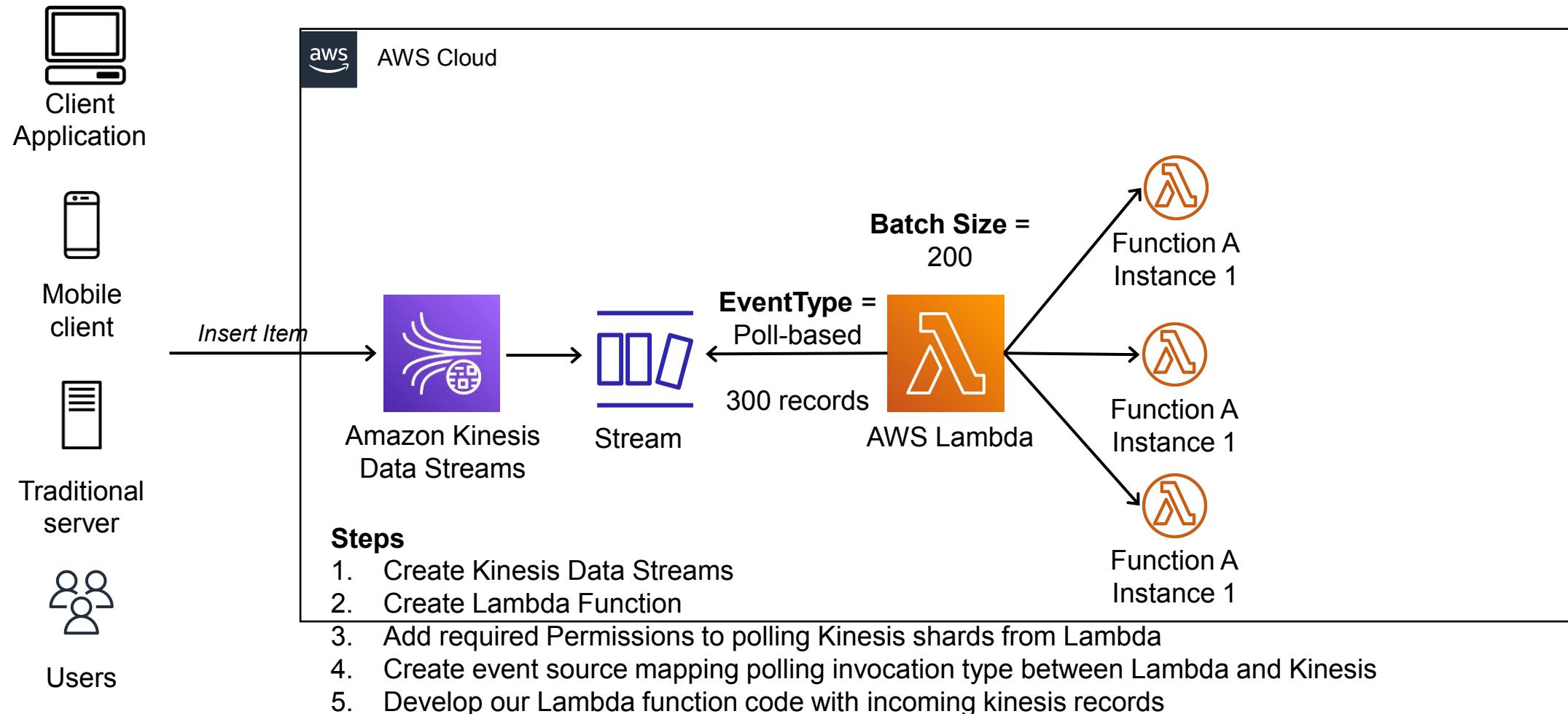


Using AWS Lambda to Process Kinesis Streams



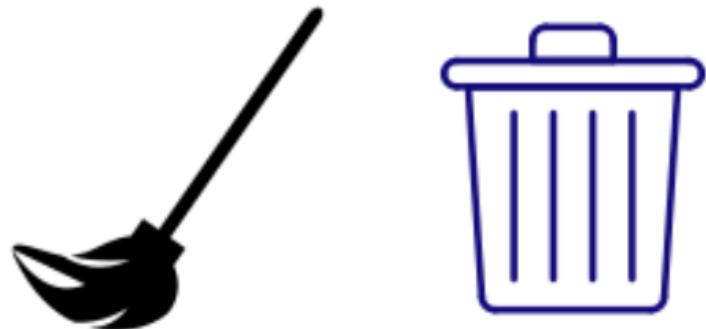
```
{  
  "Records": [  
    {  
      "kinesis": {  
        "kinesisSchemaVersion": "1.0",  
        "partitionKey": "1",  
        "sequenceNumber":  
          "495903382714902566085596925383615710959215  
        "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0  
        "approximateArrivalTimestamp": 1545084650  
      },  
      "eventSource": "aws:kinesis",  
      "eventVersion": "1.0",  
      "eventID": "",  
      "eventName": "aws:kinesis:record",  
      "invokedIdentityArn": "arn:aws:iam::123456789012:  
      "awsRegion": "us-east-2",  
      "eventSourceARN": "arn:aws:kinesis:us-east-2:  
      stream"  
    },  
    {  
      "kinesis": {  
        "kinesisSchemaVersion": "1.0",  
        "partitionKey": "1",  
        "sequenceNumber":  
          "495903382714902566085596925409257027593242  
        "data": "VGhpccyBpcyBvbmxAIGEgdGVzdC4=",  
        "approximateArrivalTimestamp": 1545084710  
      },  
      "eventSource": "aws:kinesis",  
      "eventVersion": "1.0",  
      "eventID": "",  
      "eventName": "aws:kinesis:record",  
      "invokedIdentityArn": "arn:aws:iam::123456789012:  
      "awsRegion": "us-east-2",  
      "eventSourceARN": "arn:aws:kinesis:us-east-2:  
      stream"  
    }  
  ]  
}
```

Hands-on Lab: Process Kinesis Streams using AWS Lambda



Clean up Resources

- Delete AWS Resources that we create during the section.

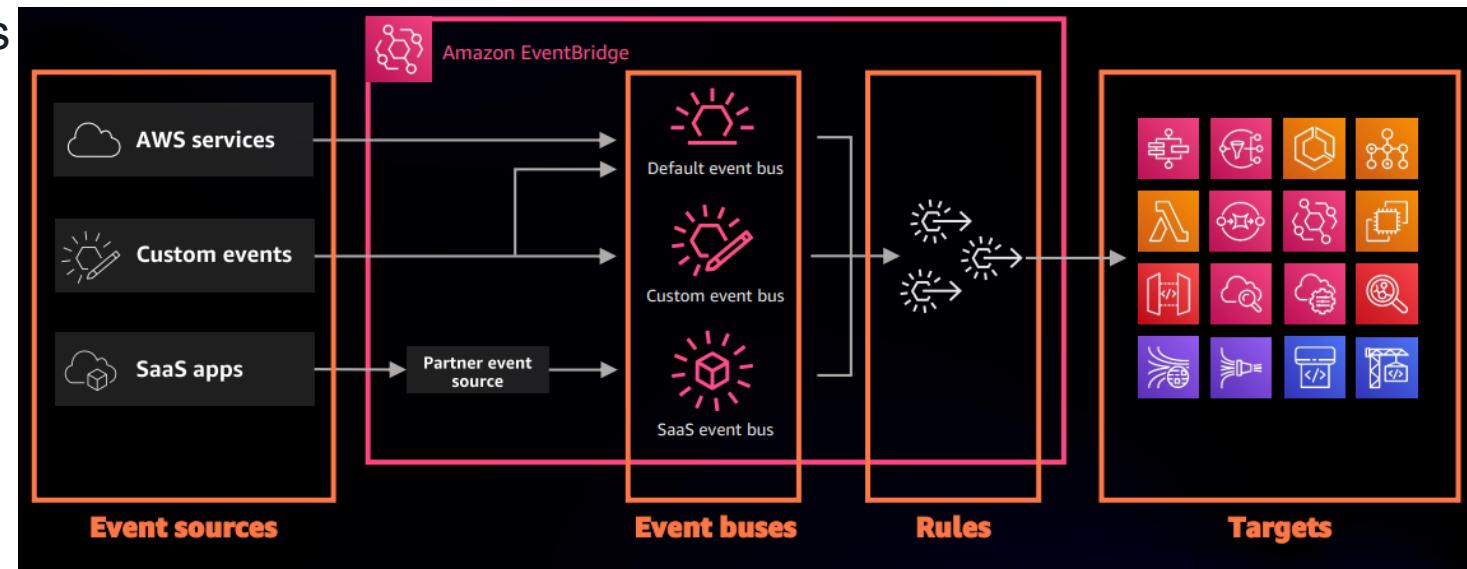


Amazon EventBridge - Decouple Services with Event-Driven Architecture

→ Learning Amazon EventBridge - Decouple Services with Event-Driven Architecture

What is Amazon EventBridge ?

- **Serverless event bus service for AWS services**
 - Build **event-driven applications** at scale using events generated from your apps
 - Use to **connect** your applications with data from a variety of sources, **integrated SaaS applications**
 - AWS services to **targets** such as **AWS Lambda functions**
 - Formerly called Amazon CloudWatch Events



<https://da-public-assets.s3.amazonaws.com/serverlessland/pdf/2021+-+Serverlesspresso+exhibit+-+PDF.pdf>

Benefits of Amazon EventBridge



Amazon EventBridge

- **Build event-driven architectures**

With EventBridge, your event targets don't need to be aware of event sources because you can filter and publish directly to EventBridge. Improves developer agility as well as application resiliency with loosely coupled event-driven architectures.

- **Connect SaaS apps**

EventBridge ingests data from supported SaaS applications and routes it to AWS services and SaaS targets. SaaS apps to trigger workflows for customer support, business operations.

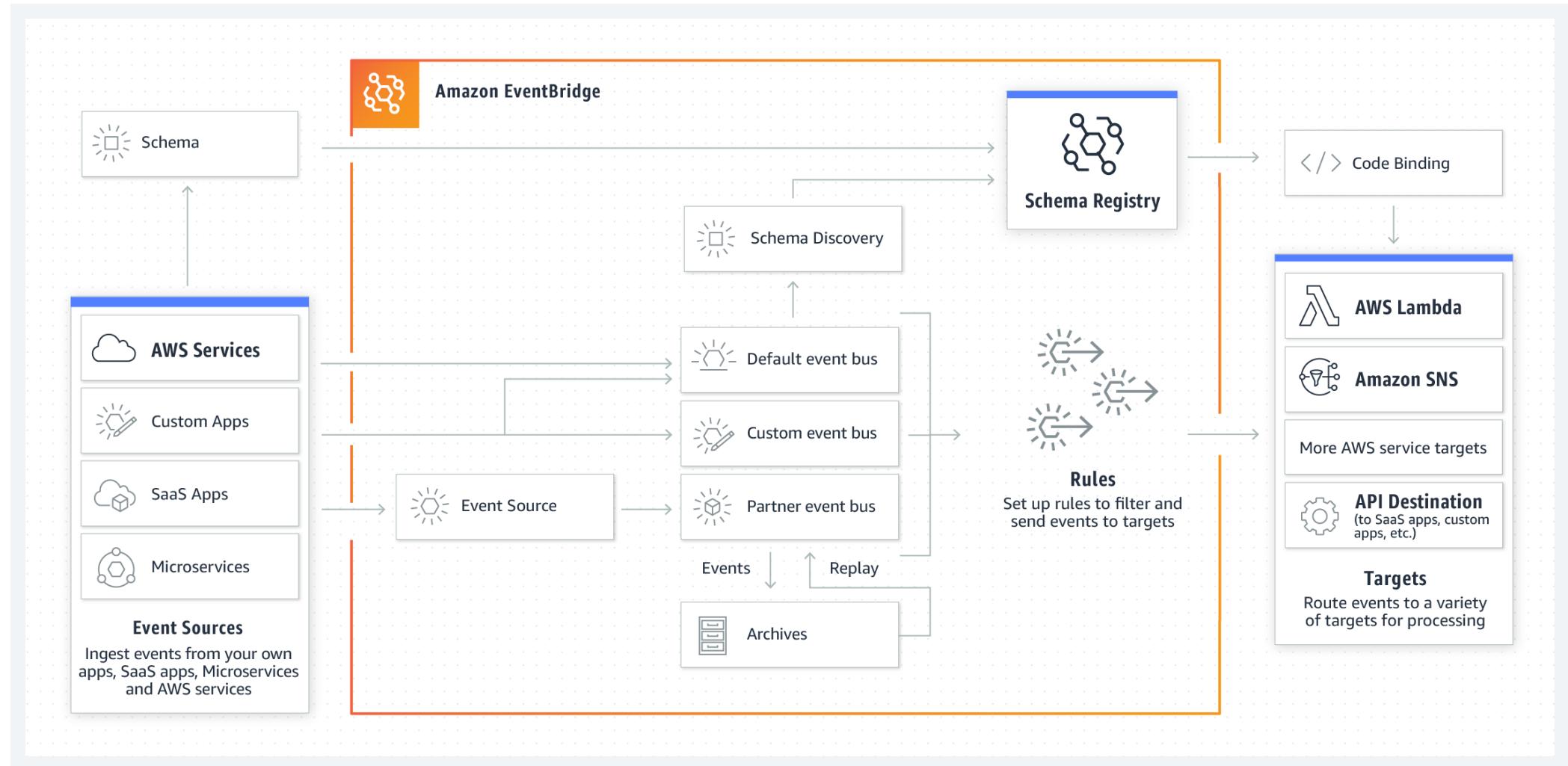
- **Write less custom code**

You can ingest, filter, transform and deliver events without writing custom code. The EventBridge schema registry stores a collection of easy-to-find event schemas.

- **Reduce operational overhead**

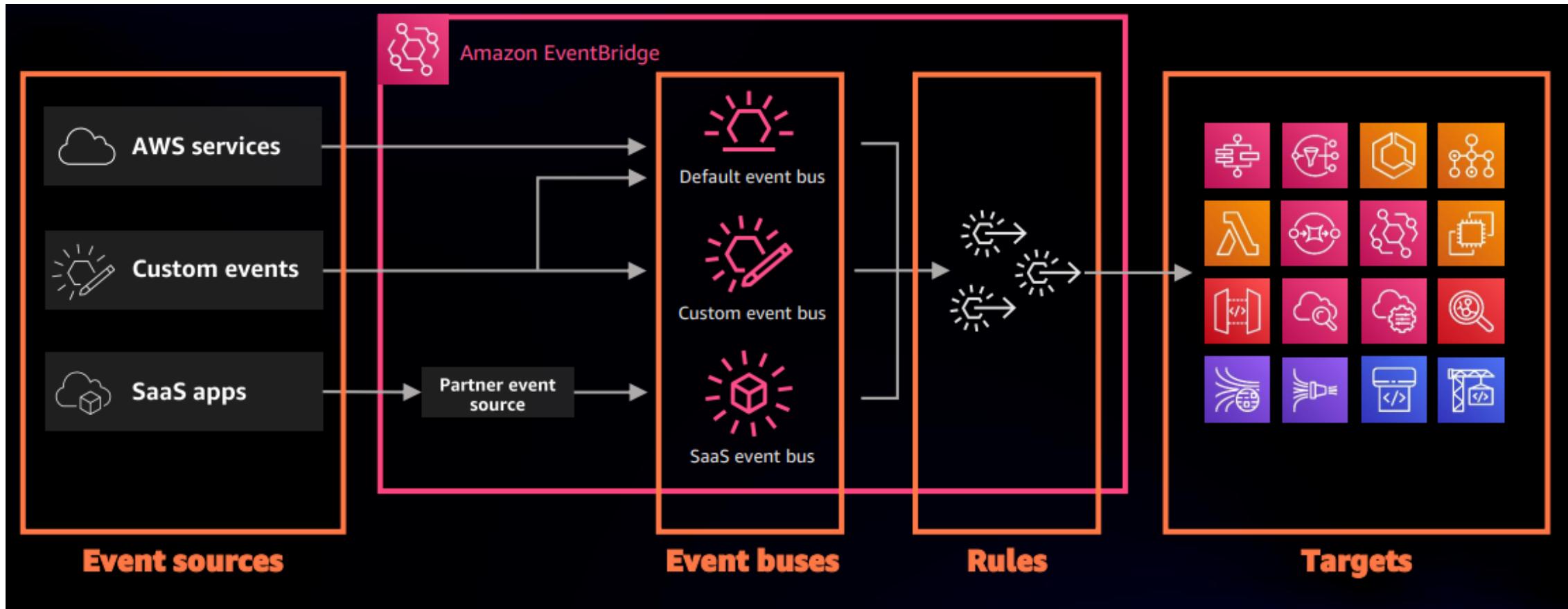
There are no servers to provision, patch, and manage. automatically scales based on the number of events ingested. Built-in distributed availability and fault-tolerance. Native event archive and replay capability.

How Amazon EventBridge works ?



<https://aws.amazon.com/eventbridge/>

How Amazon EventBridge works ?



<https://da-public-assets.s3.amazonaws.com/serverlessland/pdf/2021+-+Serverlesspresso+exhibit+-+PDF.pdf>

EventBridge Concepts - Events - Event Buses - Rules - Targets

- **Amazon EventBridge Events**

An event indicates a change in an environment such as an AWS environment or a SaaS partner service. Events are represented as JSON objects and they all have a similar structure, and the same top-level fields.

- **Amazon EventBridge Rules**

A rule matches incoming events and sends them to targets for processing. A single rule can send an event to multiple targets, which then run in parallel. An event pattern defines the event structure and the fields that a rule matches.

- **Amazon EventBridge Targets**

A target is a resource or endpoint that EventBridge sends an event to when the event matches the event pattern defined for a rule. The rule processes the event data and sends the relevant information to the target.

- **Amazon EventBridge Event Buses**

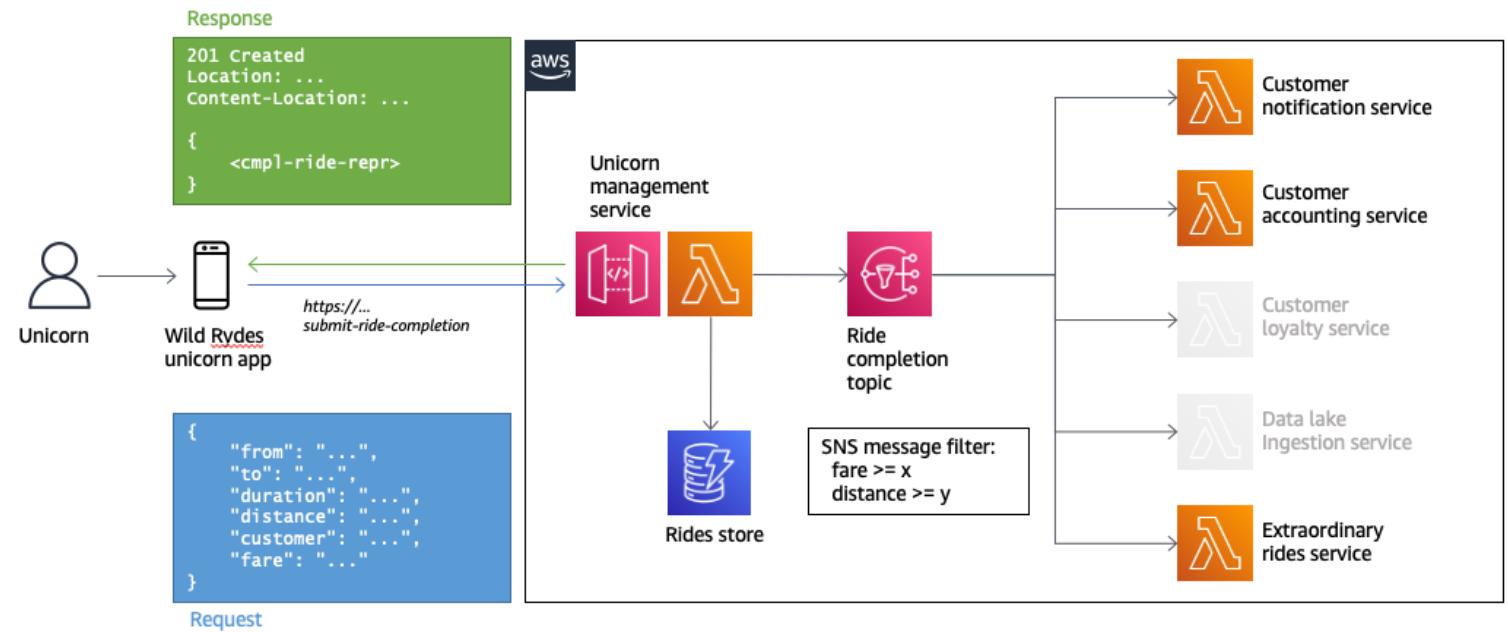
An event bus is a pipeline that receives events. Rules associated with the event bus evaluate events as they arrive. A resource-based policy specifies which events to allow, and which entities have permission to create or modify rules or targets for an event.

EventBridge Event JSON Object

```
{  
    "version": "0",  
    "id": "6a7e8feb-b491-4cf7-a9f1-bf3703467718",  
    "detail-type": "EC2 Instance State-change Notification",  
    "source": "aws.ec2",  
    "account": "111122223333",  
    "time": "2017-12-22T18:43:48Z",  
    "region": "us-west-1",  
    "resources": [  
        "arn:aws:ec2:us-west-1:123456789012:instance/i-  
        1234567890abcdef0"  
    ],  
    "detail": {  
        "instance-id": "i-1234567890abcdef0",  
        "state": "terminated"  
    }  
}
```

Fan-Out & Message Filtering with Publish/Subscribe Pattern

- **Async Communication** for performing one-to-many and publish/subscribe mechanisms.
- Client service **publish a message** and it **consumes from several microservices** which's are subscribing this message on the message broker system.
- **Decouples Messaging** between services, building loosely-coupled architectures.
- Using in **event-driven** architectures
- **Publishes** an event something happen:
- Price change in a product microservice
- **Subscribed** from SC microservice to update basket price

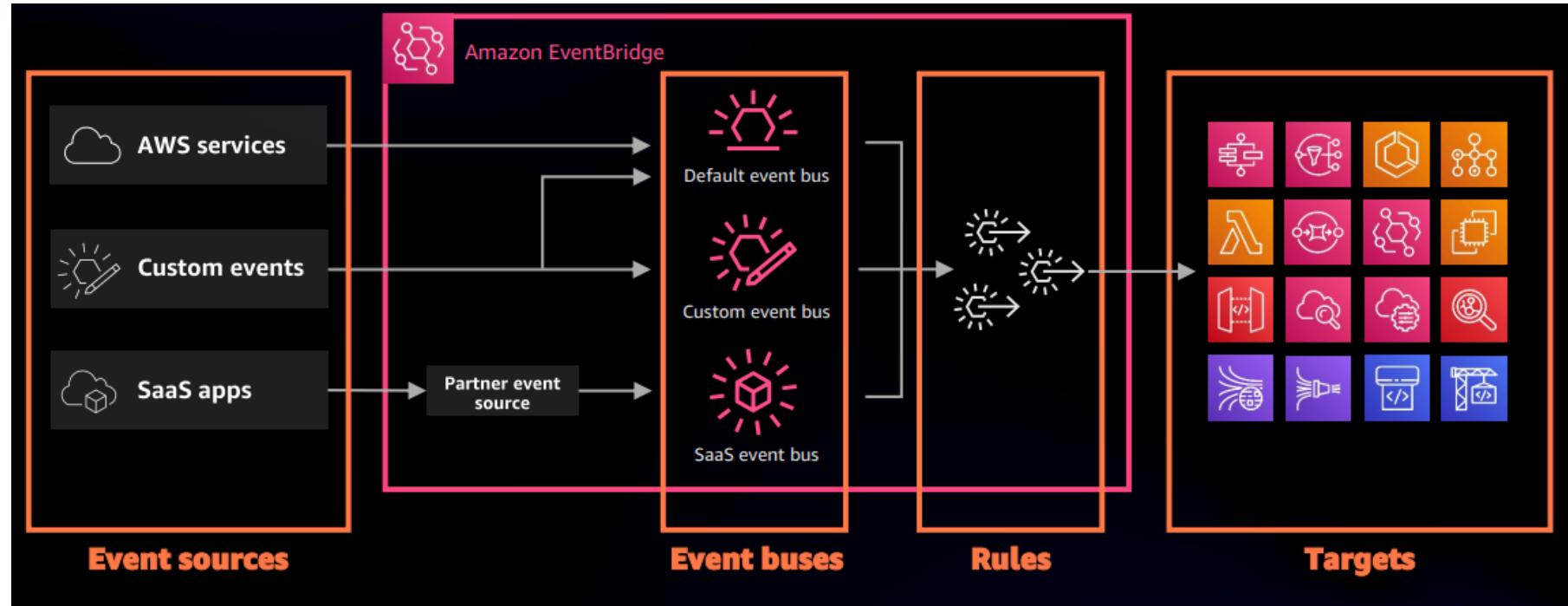


<https://async-messaging.workshop.aws/fan-out-and-message-filtering.html>

Publish/Subscribe Pattern with Amazon EventBridge

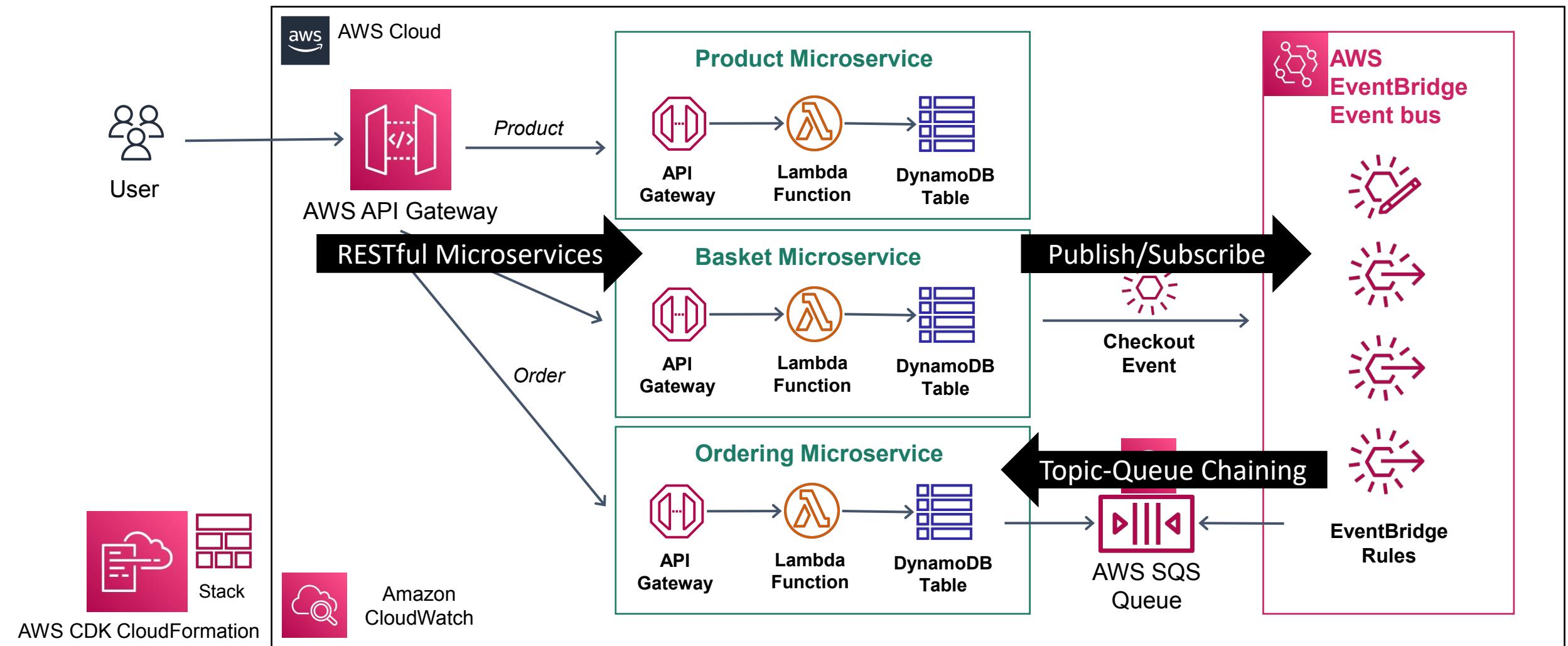
- **Amazon EventBridge** for Event-Driven asynchronous Communication between Microservices
 - Event Sources
 - Event Buses
 - Rules
 - Targets

The diagram illustrates the architecture of Amazon EventBridge. On the left, a box labeled "AWS services" contains a cloud icon. An arrow points from this box to a central vertical bar. The top section of this bar is purple and features the "Amazon EventBridge" logo (a gear-like icon) and the text "Amazon EventBridge". Below this, the bar continues in a pink color. From the bottom of the pink section, two arrows point to two separate boxes. The first box is orange and labeled "Default event bus", containing a sunburst icon. The second box is also orange and contains two smaller icons: one representing a Lambda function and another representing a database.

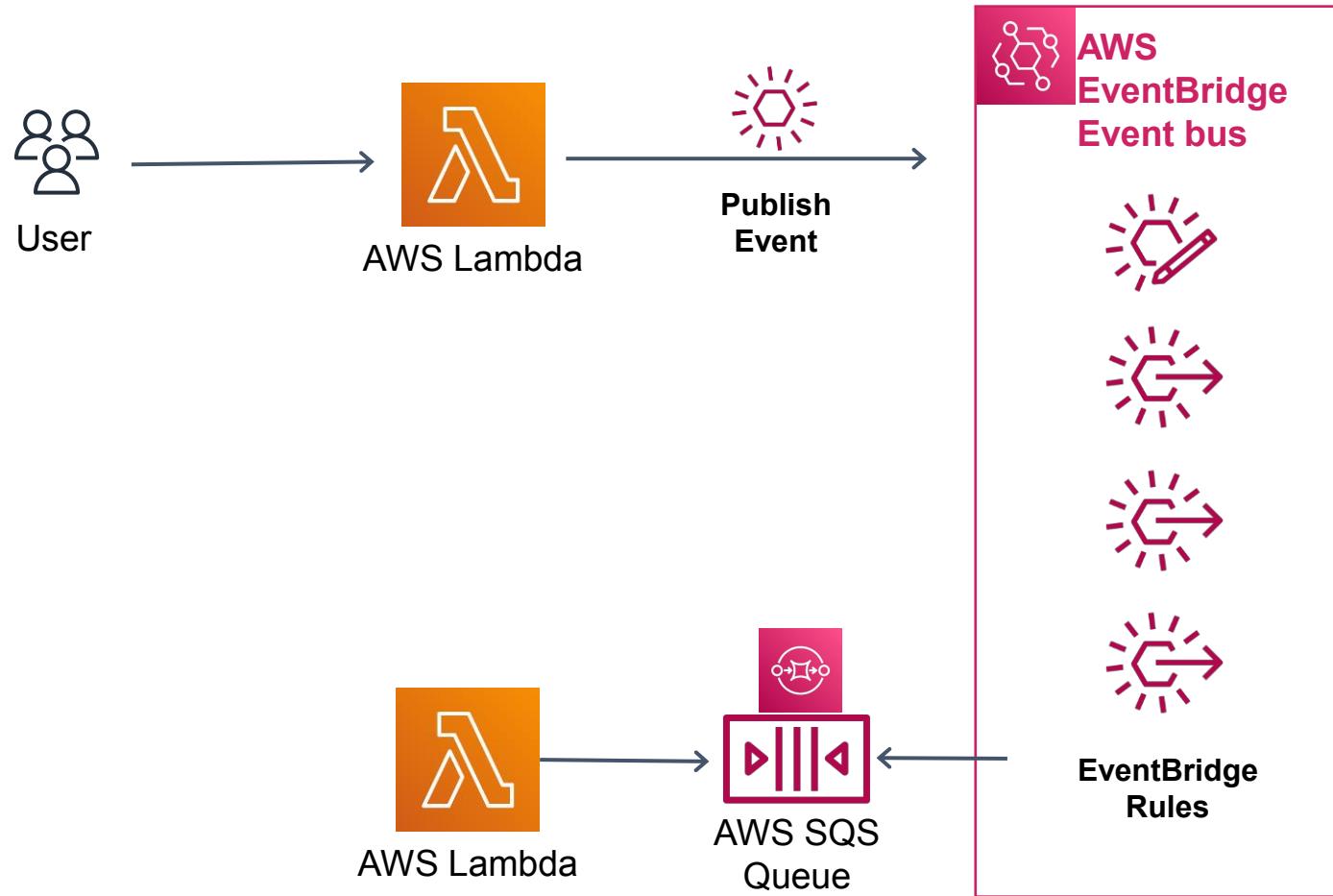


<https://da-public-assets.s3.amazonaws.com/serverlessland/pdf/2021---Serverlesspresso+exhibit---PDF.pdf>

Serverless Patterns for Microservices



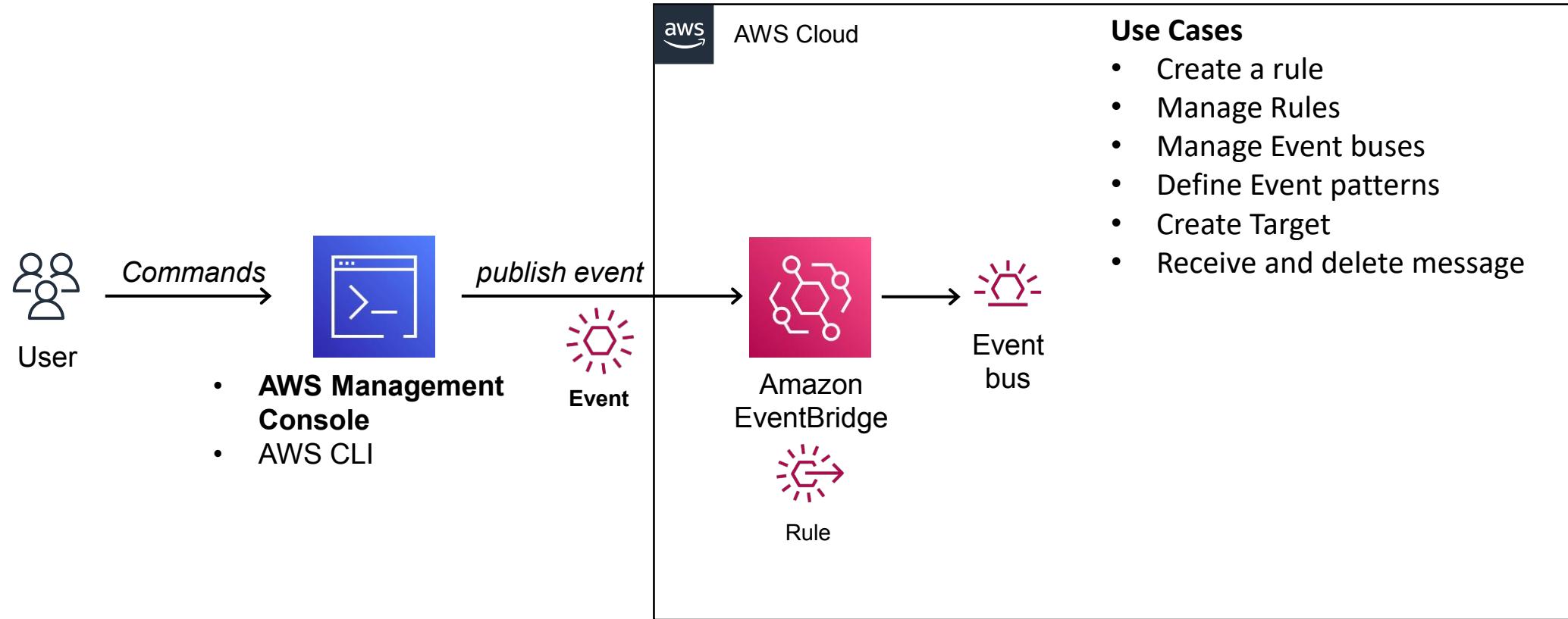
Using AWS Lambda with Other Services - Amazon EventBridge



Amazon EventBridge - Walkthrough with AWS Management Console

→ DEMO - Amazon EventBridge - Walkthrough with AWS Management Console

Getting started with Amazon EventBridge with AWS Console



AWS Lambda & Serverless Course Learning Path

1

Theoretical Information

AWS Service overview, core concepts, features, uses cases and general information

2

Walkthrough with AWS Console

AWS Service Walkthrough with AWS Management Console performs main use cases

3

Developing with AWS SDK

AWS Service Programmatic Access interaction with Serverless APIs using AWS SDK or CLI

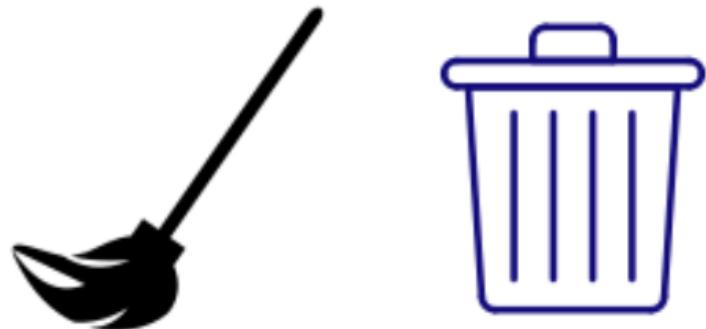
4

Hands-on Labs Real-World Apps

AWS Service Hands-on Labs implementation with Real-World Use Cases

Clean up Resources

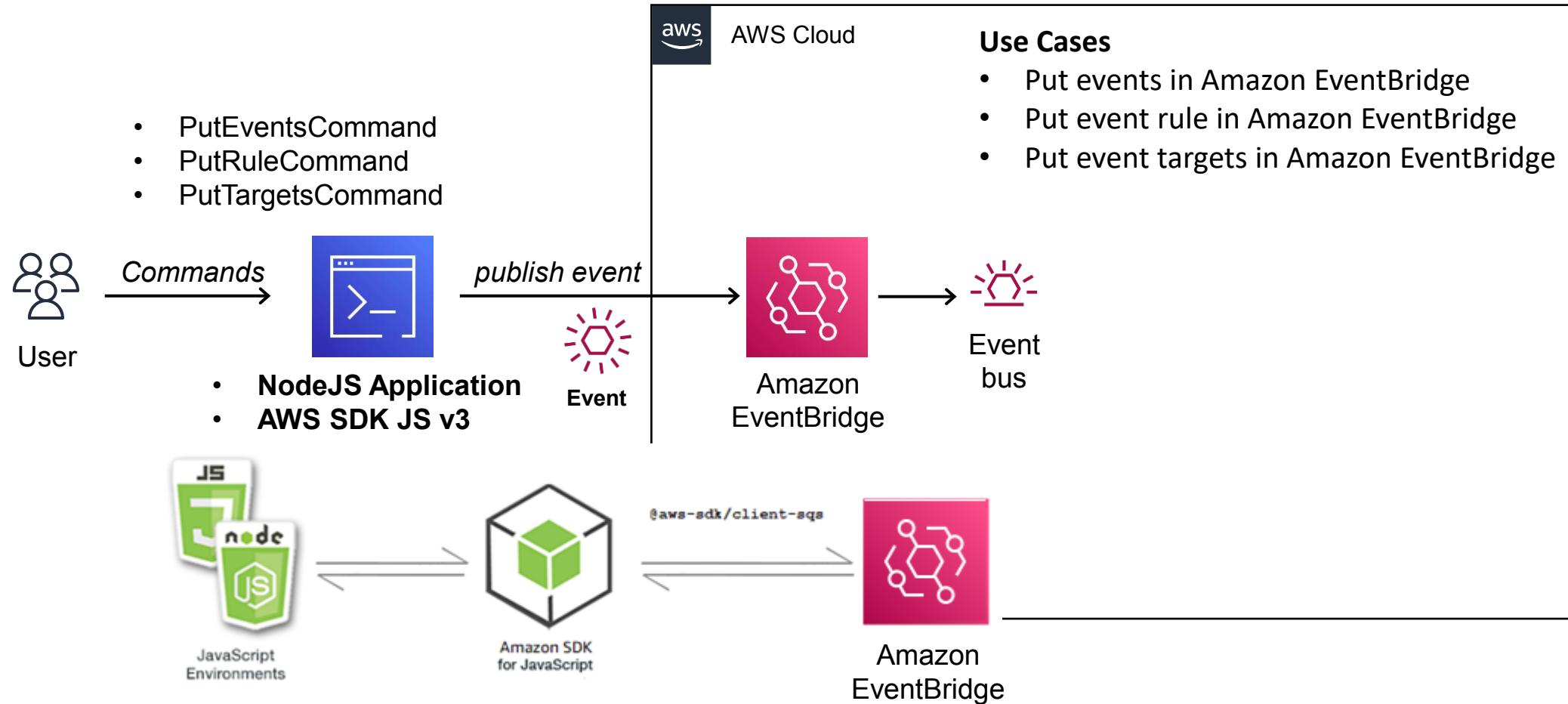
- Delete AWS Resources that we create during the section.



Amazon EventBridge - Developing with AWS SDK

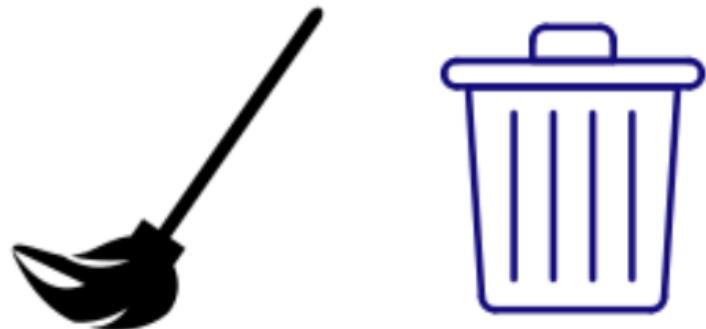
→ Amazon EventBridge - Developing with AWS SDK interaction to
Serverless APIs Programmatic Access

Amazon EventBridge SDK Examples using AWS SDK Javascript v3



Clean up Resources

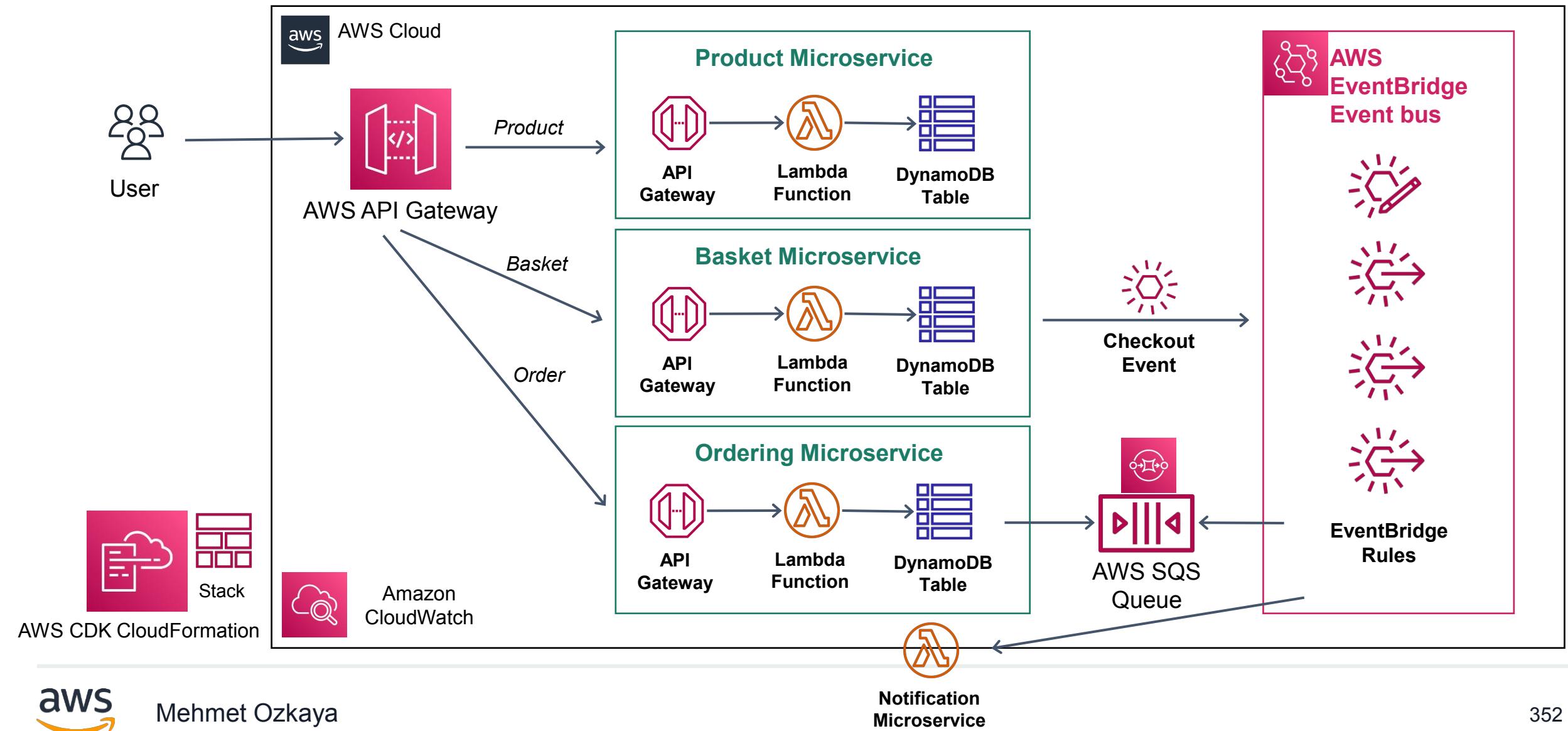
- Delete AWS Resources that we create during the section.



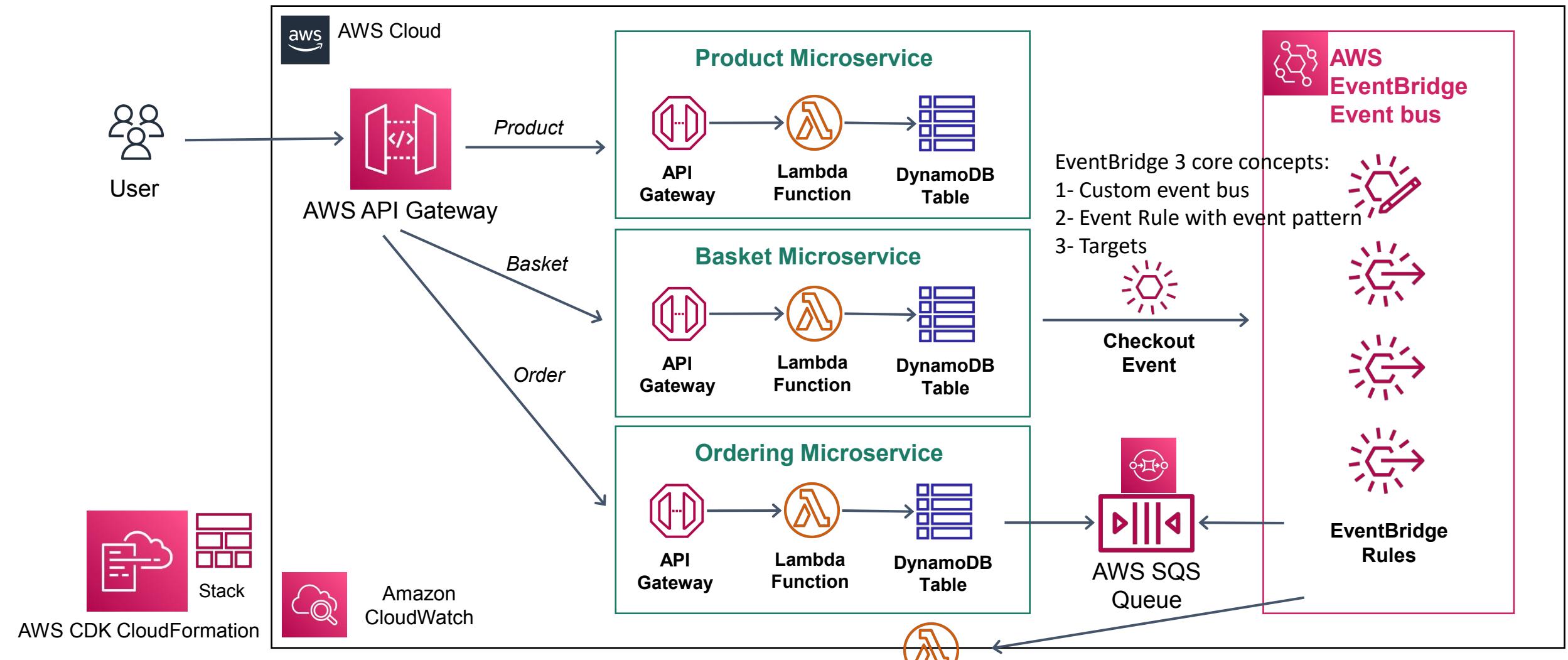
Hands-on Labs : Event-Driven Microservice Architecture Using Amazon EventBridge

→ Developing Hands-on Labs : Event-Driven Microservice Architecture Using Amazon EventBridge, SQS and Lambda

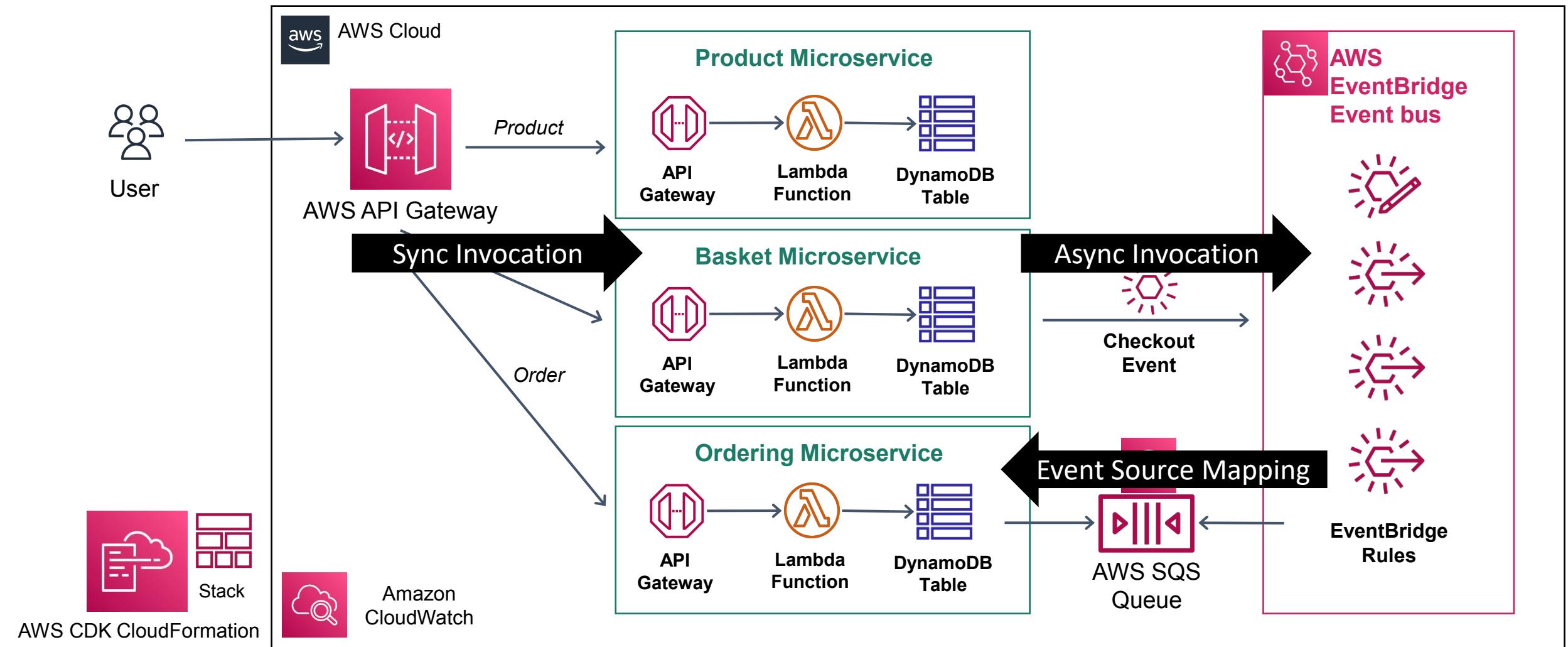
Hands-on Lab: AWS Serverless Microservices for Ecommerce using Amazon EventBridge



Hands-on Lab: AWS Serverless Microservices for Ecommerce using Amazon EventBridge



AWS Serverless Microservices for Ecommerce Application



Serverless Project Development Phases

1

Infrastructure Creation on AWS

Create API Gateway, Lambda Function and DynamoDB table on AWS Cloud - Also we can automate this part with IaC using CDK in the last sections but now we will create infrastructure with console or cli

2

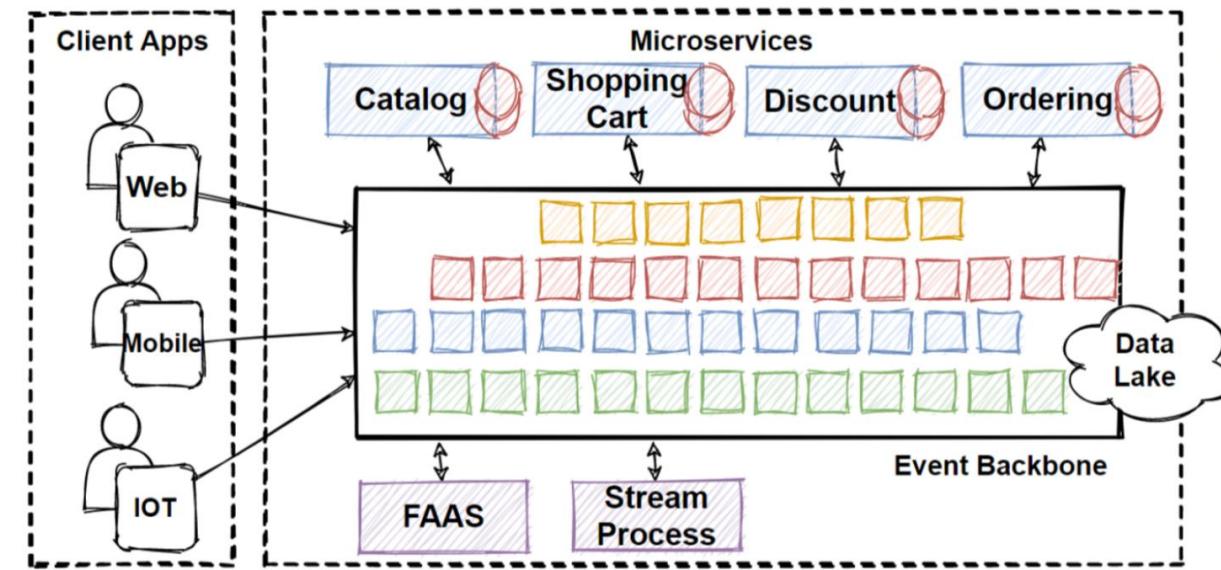
Develop Lambda + SQS business logic with AWS SDK

Use AWS SDK JS v3 with ES6 standards to implement crud functions into lambda function.



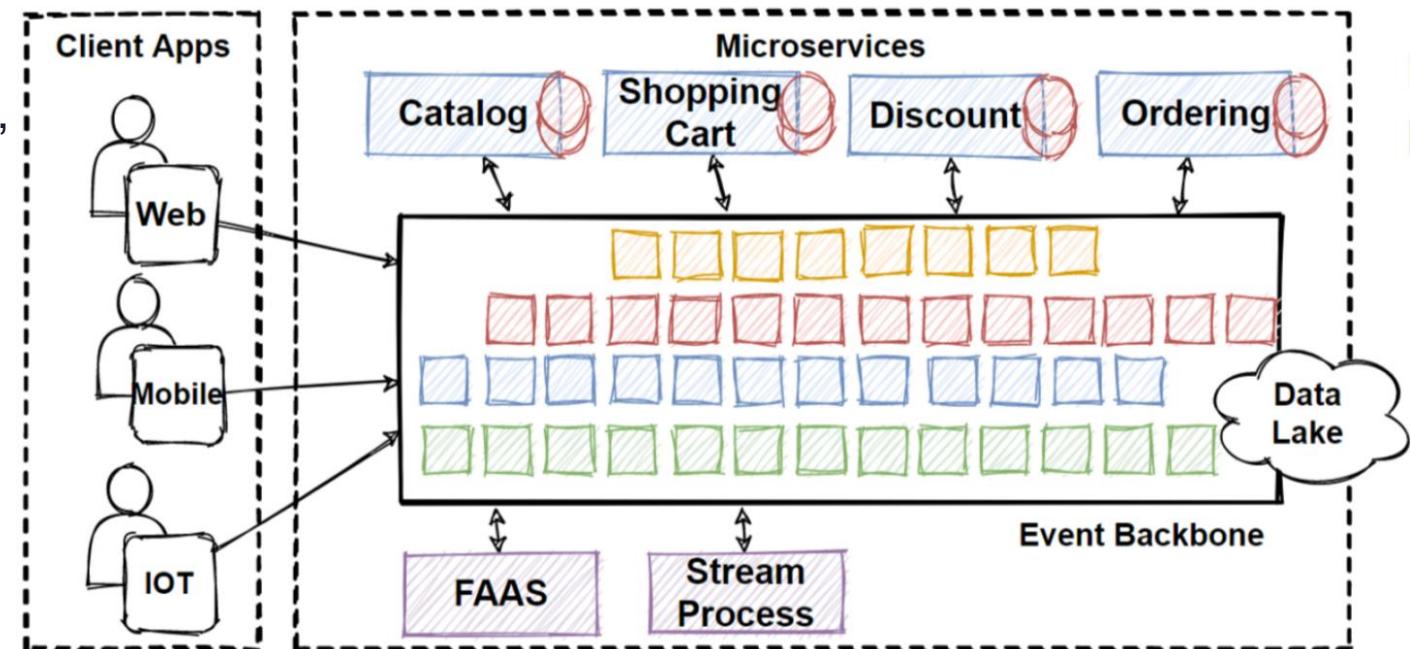
Event-Driven Microservices Architecture

- Communicating with microservices via **event messages**.
Do **asynchronous behavior** and **loosely coupled** structures.
- **E-commerce** application **use cases** whichs are a customer create orders with some products and if the payment is successful, the products should be delivered to the customer.
- **Flow of events** like;
 - a customer creates an order
 - the customer receives a payment request
 - if the payment is successful the stock is updated and the order is delivered
 - if the payment in not successful, rollback the order and set order status is not completed.
- **Human readable** and if a new business requirement appears, it is easier to change the flow.



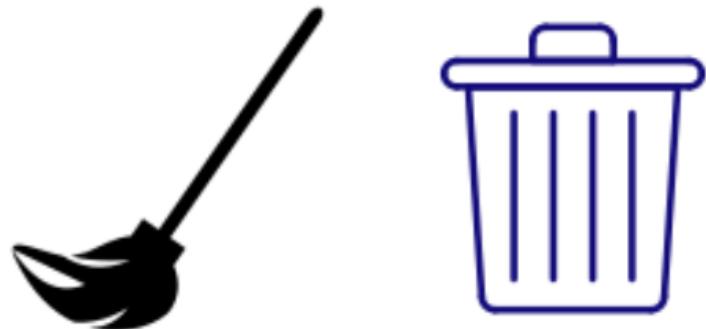
Event-Driven Microservices Architecture

- **Microservices** will only care about the **events**, not about the other microservices. they **process** only **events** and **publish** new **event** to **trigger** other services.
- Event-Driven Microservices Architectures like using **real-time messaging platforms**, **stream-processing**, **event hubs**, real-time processing, batch processing, data intelligence and so on.
- Communication via **Event-Hubs**. Think Event-Hubs is huge event store database that can make real-time processing.
- Every microservices, application, IOT devices, even **FAAS serverless services** can interact with each other with subscribing events in **Event Hub**.



Clean up Resources

- Delete AWS Resources that we create during the section.

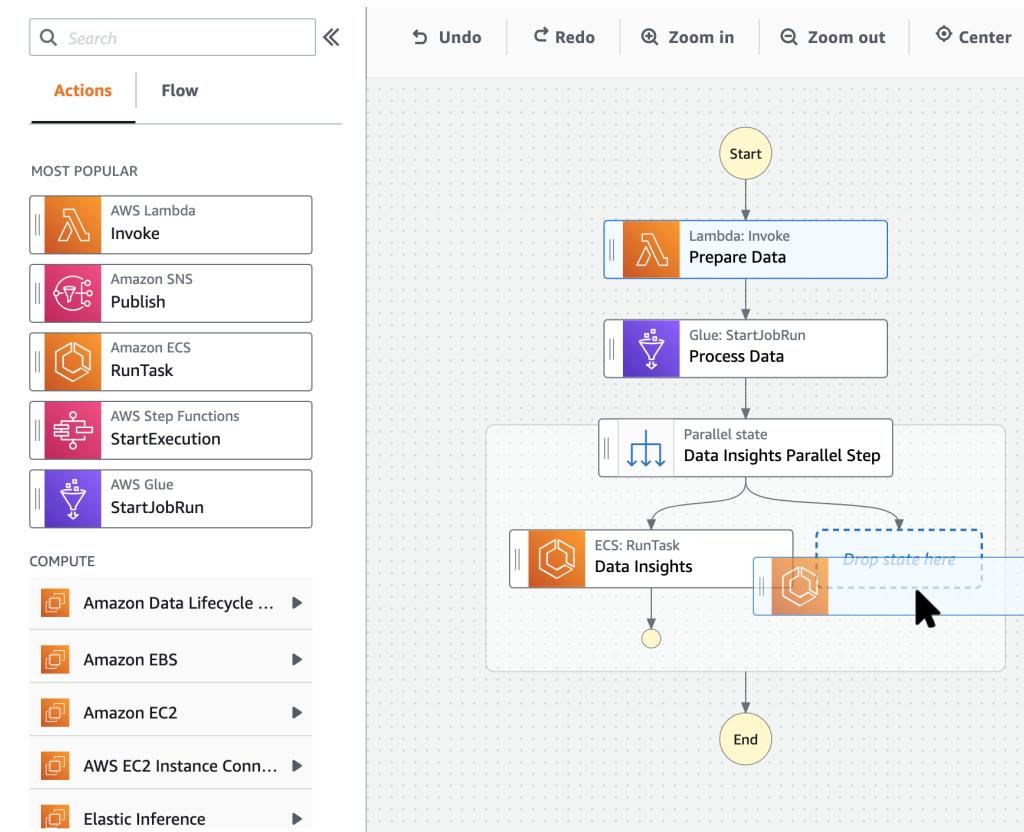


AWS Step Functions - Orchestrate Distributed Transactions

→ Learning AWS Step Functions - Orchestrate Distributed Transactions
for Microservices Architectures

What is AWS Step Functions ?

- **AWS Step Functions** is a low-code, **visual workflow** service that developers use to build **distributed applications**, automate IT and business processes.
- **Step Functions** is a **serverless orchestration** service that combine AWS Lambda functions and other AWS services to build business critical applications.
- Step Functions is based on **state machines** and **tasks**. A **state machine** is a **workflow**. A **task** is a **state** in a **workflow** that represents a single unit of work that another AWS service performs.
- AWS Step Functions makes it **easier to orchestrate** multiple AWS services to accomplish tasks.
- Step Functions allows you to **create steps** in a **process** where the output of one step becomes the input for another step, all using a visual workflow editor.



<https://aws.amazon.com/step-functions>

Benefits of using AWS Step Functions



AWS Step Functions

- **Build and deploy rapidly**

Get started quickly with Workflow Studio, a simple drag-and-drop interface. With Step Functions, you can express complex business logic as low-code, event-driven workflows.

- **Write less integration code**

Compose AWS resources from over 200 services including Lambda, ECS, Fargate, Batch, DynamoDB, SNS, SQS, SageMaker, EventBridge.

- **Build fault-tolerant and stateful workflows**

Built-in try/catch, retry, and rollback capabilities deal with errors and exceptions automatically based on your defined business logic.

- **Designed for reliability and scale**

Step Functions offers two workflow types - Standard or Express - that can be used depending on your specific use case.

- Decouple application workflow logic from business logic that is another best practice of Serverless development.

Drawbacks of using AWS Step Functions



AWS Step Functions

- **Complex Configuration**

Configuration with the Amazon States Language Amazon States Language is quite complex. Its syntax hard to read and modify.

- **Over Engineering when using unnecessary cases**

Decoupling business logic from task sequencing can make your code harder to understand While decoupling services from the orchestration layer can make things more scalable and easier to operate.

- **Vendor Lock-in**

Vendor lock-in The Amazon States Language is proprietary and can only be used on AWS. Therefore, if you decide to migrate to a different cloud provider, you'll need to re-implement the orchestration layer.

Use Cases of AWS Step Functions



AWS Step Functions

- **Automate Extract, Transform, and Load (ETL) process**

Long-running, multiple ETL jobs run in order and complete successfully, instead of manually orchestrating those jobs or maintaining a separate application.

- **Prepare Data for Machine Learning**

Enable machine learning, source data to be collected, processed, and normalized ML modelling systems like Amazon SageMaker can train on that data.

- **Orchestrate microservices**

We can use multiple AWS Lambda functions into responsive serverless applications and microservices and also also orchestrate data and services.

- **IT and security automation**

IT automation can help manage increasingly complex and time-consuming operations, such as upgrading and patching software, deploying security updates to address vulnerabilities, selecting infrastructure.

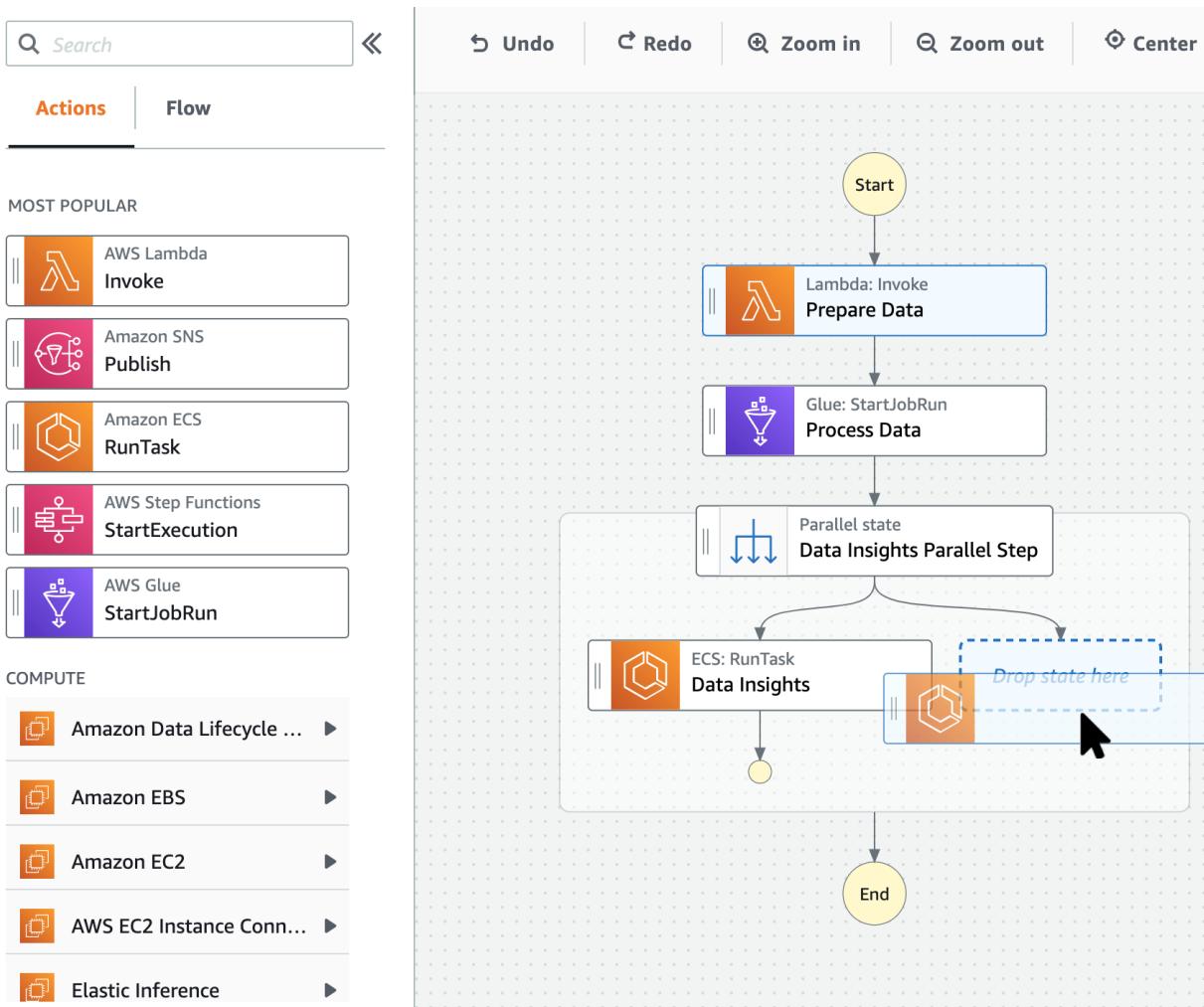
- **Event-driven workflow**

Use Step Functions to microservices, without having to write code for workflow logic, parallel processes, error handling, timeouts, or retries.

AWS Step Functions Core Concepts - State Machine, States, Tasks

State Machine States

- Task state
- Choise state
- Fail/Succeed state
- Pass state
- Wait state
- Parallel state
- Map state



Example state – HelloWorld:

```
"HelloWorld": {  
    "Type": "Task",  
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:  
    "Next": "AfterHelloWorldState",  
    "Comment": "Run the HelloWorld Lambda function"  
}
```

AWS Step Functions Workflow types - Standard vs Express



AWS Step Functions

- **Standard Workflows**

Choose a standard workflow for processes that are long-running or that require human intervention. Standard Workflows are ideal for long-running, durable, and auditable workflows. They can run for up to a year and you can retrieve the full execution history using the Step Functions API

- **Express workflows**

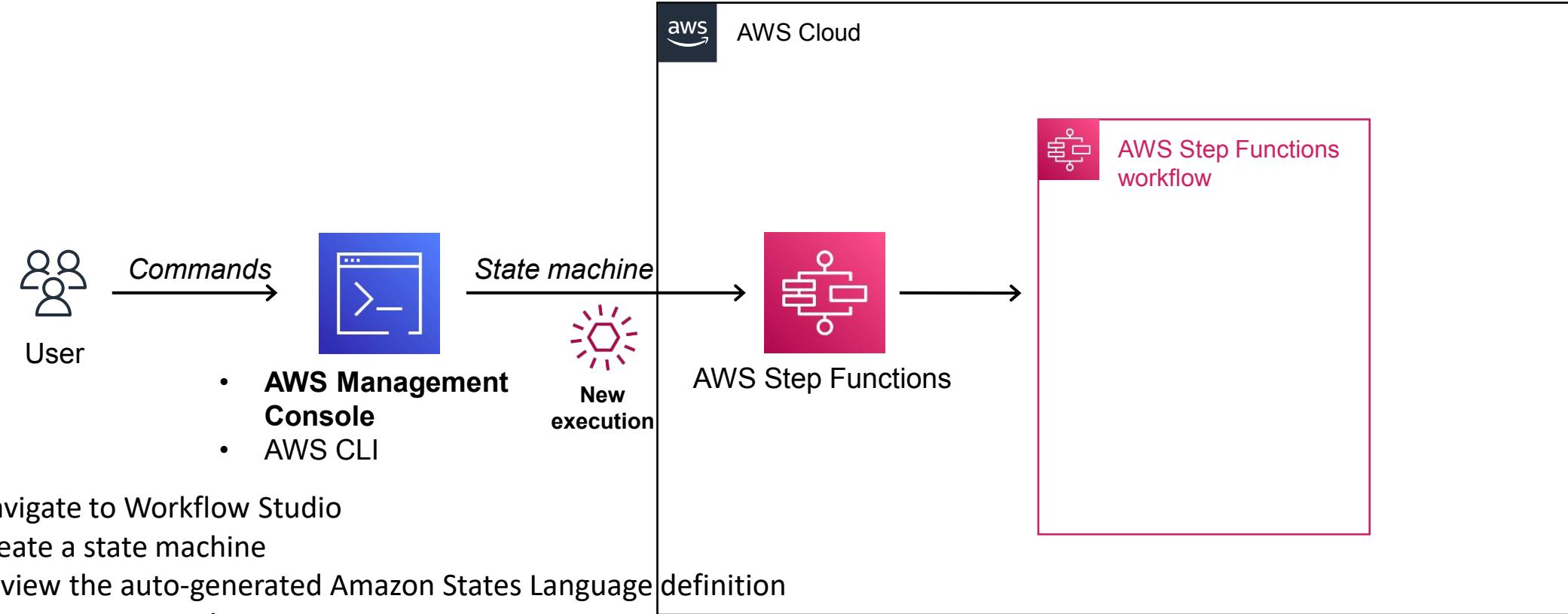
Express workflows are well-suited for short-running about fewer than five minutes, high-volume processes. Express Workflows are ideal for high-volume, event-processing workloads such as IoT data ingestion, streaming data processing and transformation, and mobile application backends.

- There are **two types of Express Workflows**; Asynchronous Express Workflows and Synchronous Express Workflows.
- **Asynchronous Express Workflows** return confirmation that the workflow was started, but do not wait for the workflow to complete. Asynchronous Express Workflows can be used when you don't require immediate response output, such as messaging services, or data processing.
- **Synchronous Express Workflows** start a workflow, wait until it completes, then return the result. Synchronous Express Workflows can be used to orchestrate microservices.

AWS Step Functions - Walkthrough with AWS Management Console

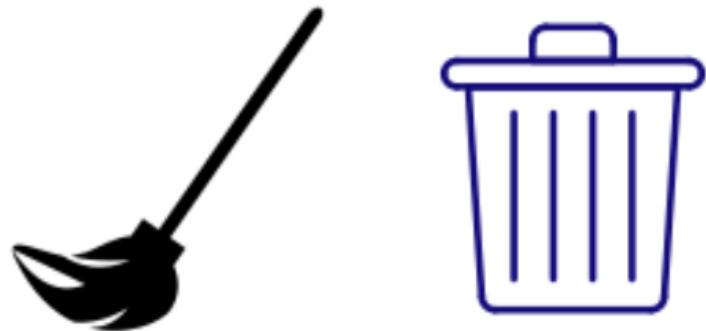
→ DEMO - AWS Step Functions - Walkthrough with AWS Management
Console

Getting started with AWS Step Functions with AWS Console



Clean up Resources

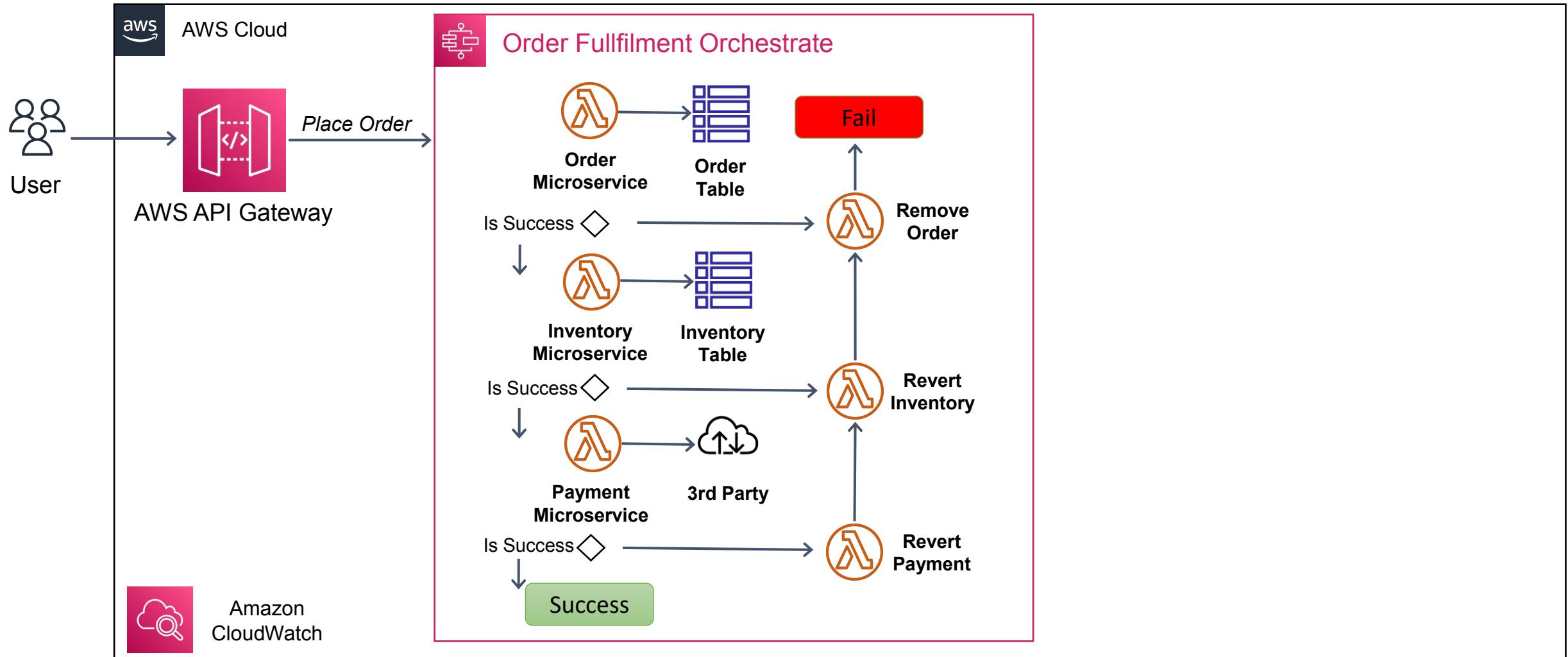
- Delete AWS Resources that we create during the section.



Hands-on Labs : Saga Pattern for Orchestrate Distributed Transactions

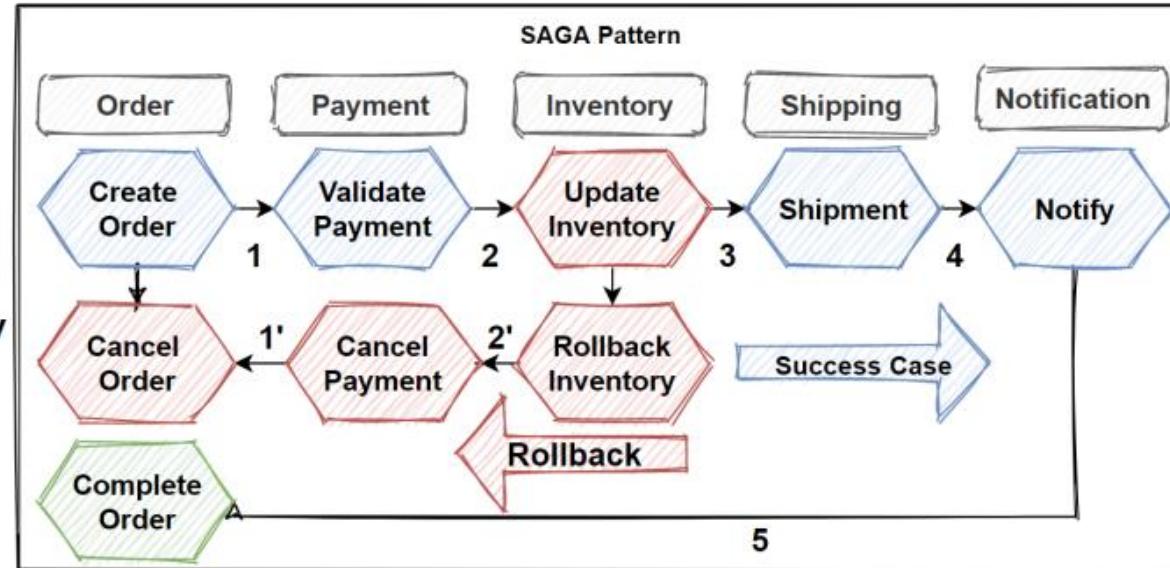
- Developing Hands-on Labs: Saga Pattern for Orchestrate Distributed Transactions using AWS Step Functions

Hands-on Lab: Saga Pattern for Orchestrate Distributed Transactions using AWS Step Functions



Saga Pattern for Distributed Transactions

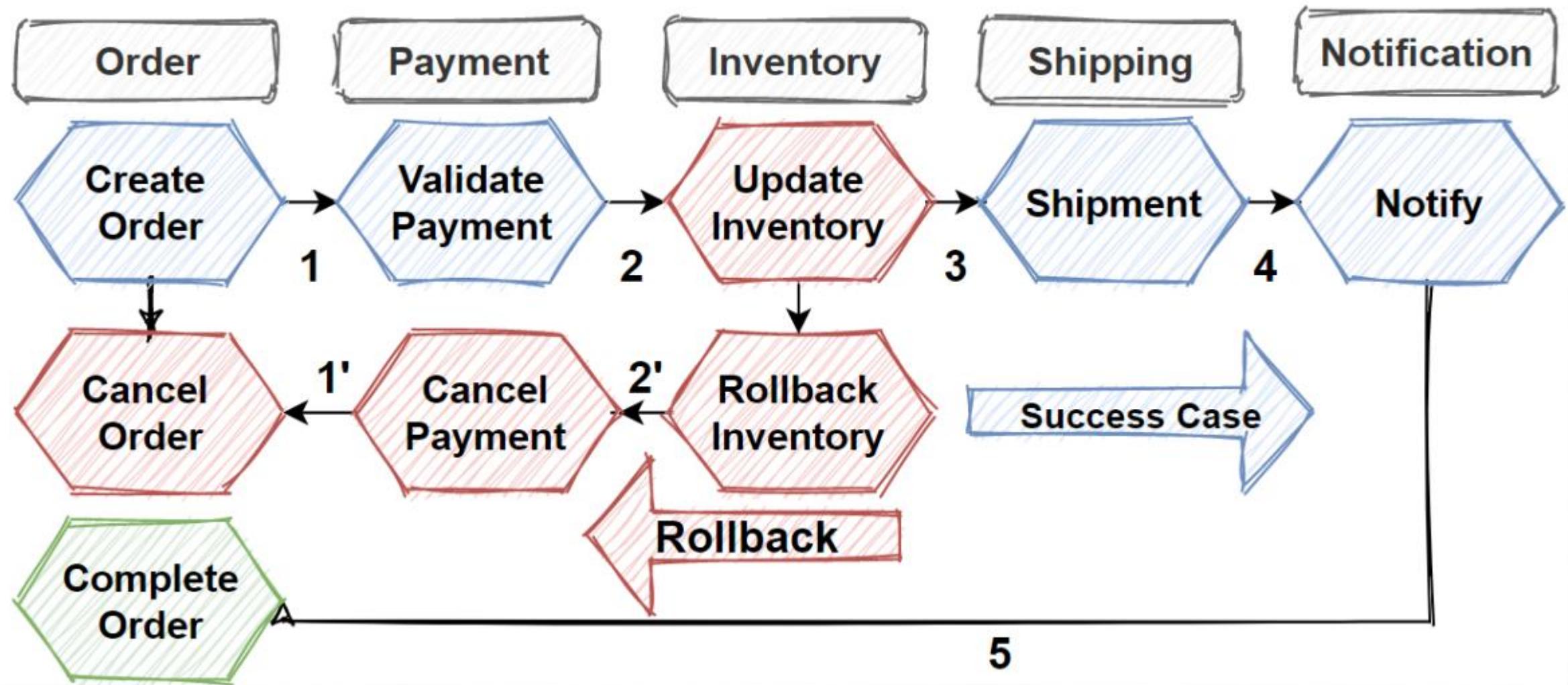
- Manage data consistency across microservices in distributed transaction cases
- Create a set of transactions that update microservices sequentially
- Publish events to trigger the next transaction
- If failed, trigger to rollback transactions
- Grouping these local transactions and sequentially invoking one by one



Saga implementation ways

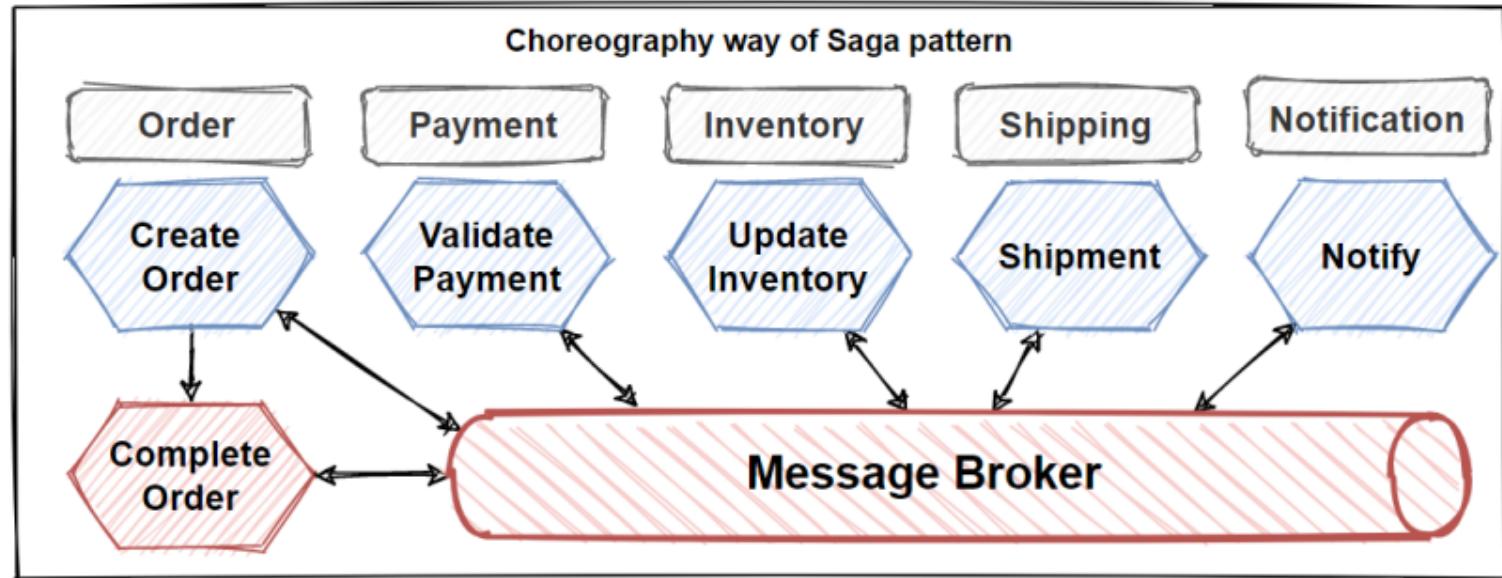
- **Choreography or orchestration**

SAGA Pattern



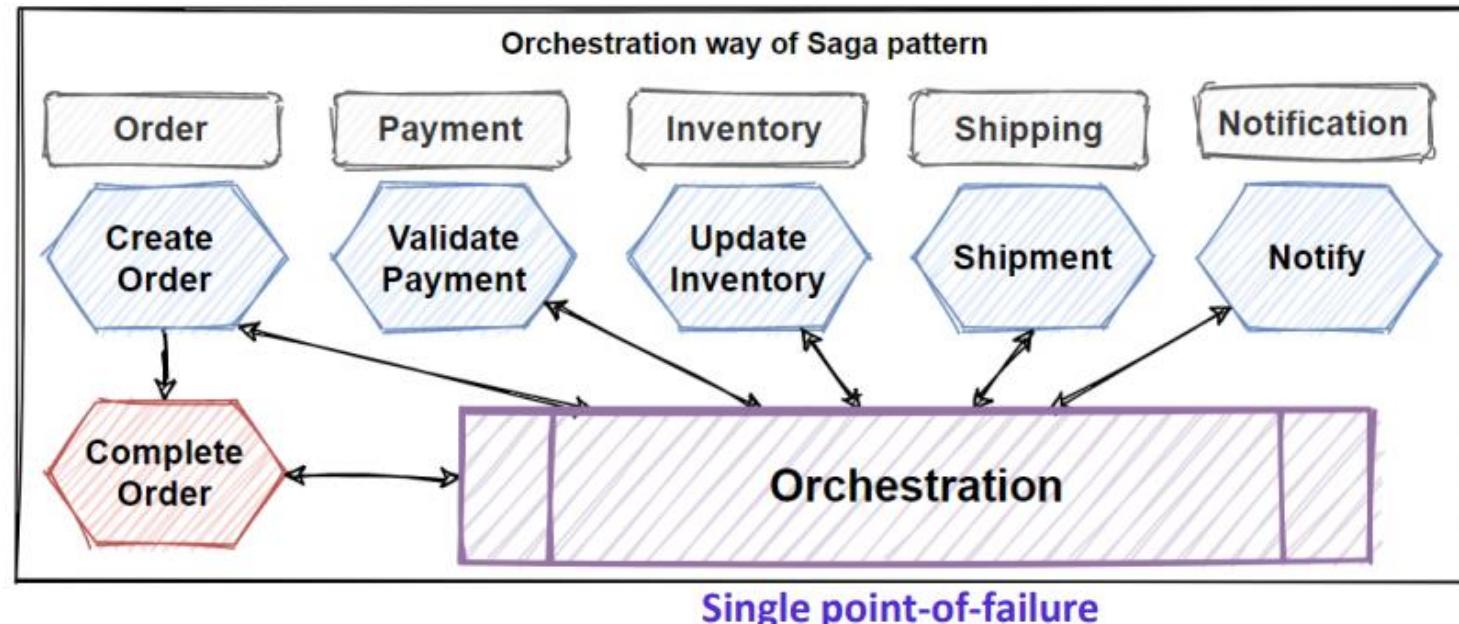
Choreography Saga Pattern

- Coordinate sagas with applying publish-subscribe principles
- Each microservices run its own local transaction
- Publish events to trigger the next transaction
- Workflow steps increase, then it can become confusing and hard to manage transaction
- Decouple direct dependency



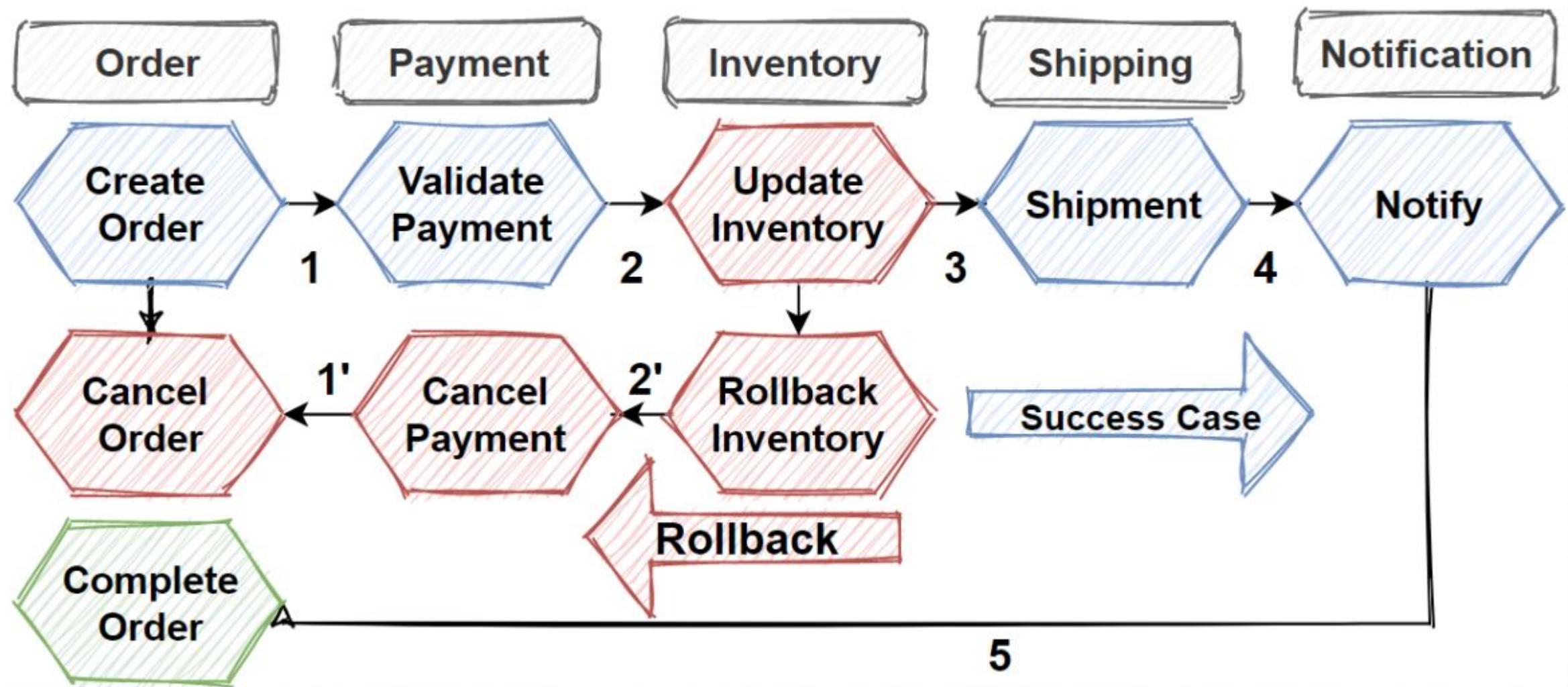
Orchestration Saga Pattern

- Coordinate sagas with a centralized controller microservice
- Invoke to execute local microservices transactions sequentially

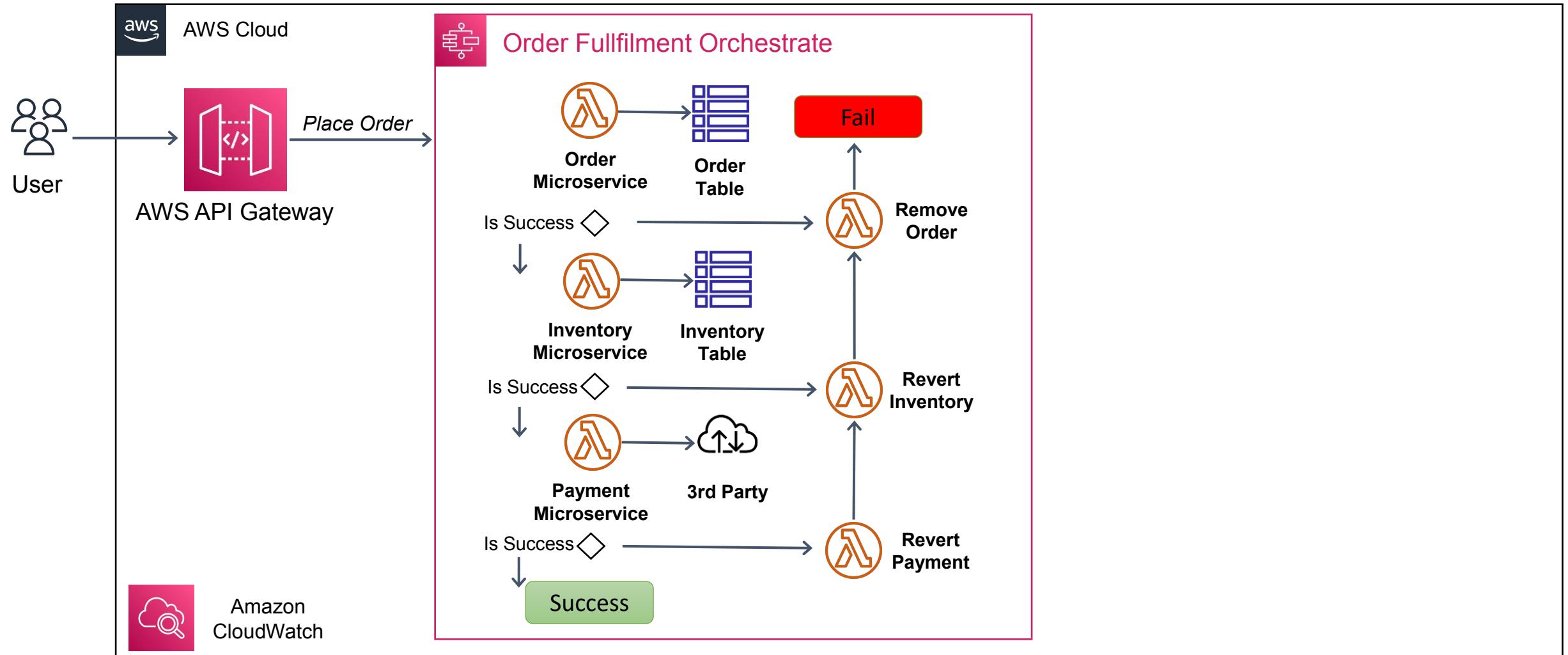


- Execute saga transaction and manage them in centralized way and if one of the step is failed, then executes rollback steps with compensating transactions
- Good for complex workflows which includes lots of steps

SAGA Pattern



Hands-on Lab: Saga Pattern for Orchestrate Distributed Transactions using AWS Step Functions



Serverless Project Development Phases

1

Infrastructure Creation on AWS

Create API Gateway, Lambda Function and DynamoDB table on AWS Cloud - Also we can automate this part with IaC using CDK in the last sections but now we will create infrastructure with console or cli

2

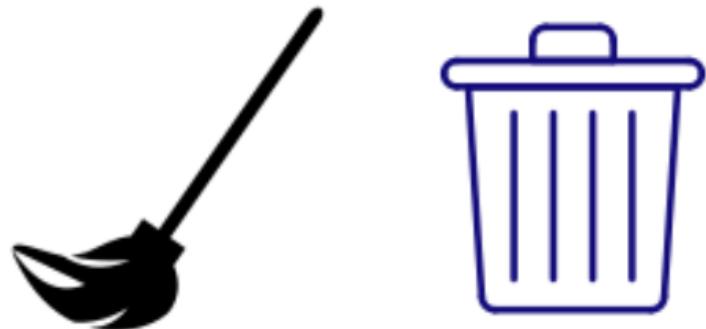
Develop Lambda + SQS business logic with AWS SDK

Use AWS SDK JS v3 with ES6 standards to implement crud functions into lambda function.



Clean up Resources

- Delete AWS Resources that we create during the section.



Serverless Deployment Frameworks and AWS CDK

→ Learning Serverless Deployment Frameworks and AWS CDK

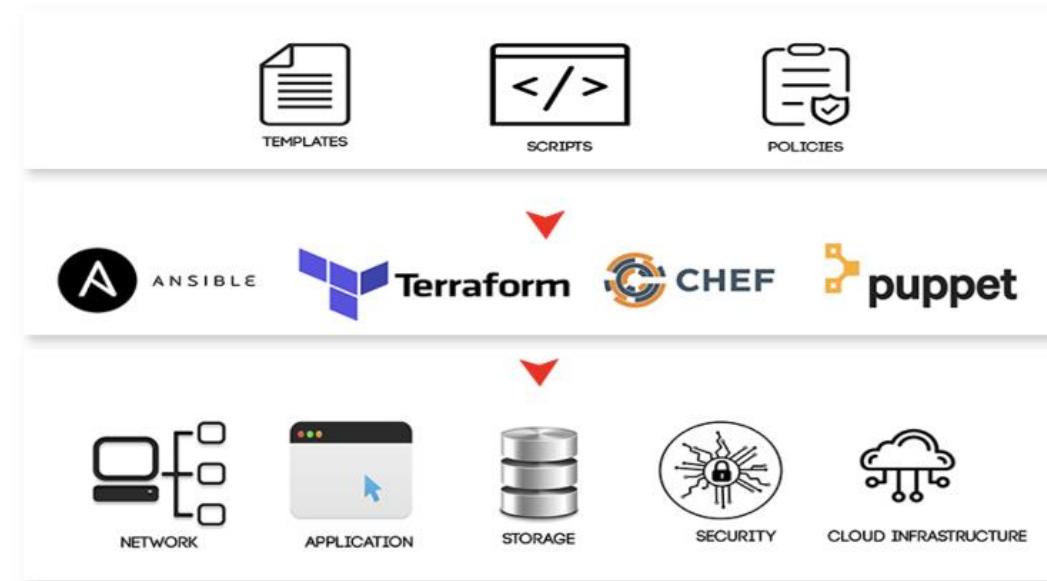
Introduction - Serverless Deployment Frameworks

- AWS has different Serverless Deployment Frameworks. But Before that we should learn;
- What is IaC - Infrastructure as Code
- AWS Cloud Formation
- AWS Serverless Deployment Frameworks
- We can Choose from a variety of AWS, open source, and third-party web frameworks that simplify serverless application development and deployment.
- AWS Serverless Application Model (AWS SAM) is an open-source framework for building serverless applications.
- AWS Cloud Development Kit (AWS CDK) is an open source software development framework to define your cloud application resources using familiar programming languages.



What is IaC - Infrastructure as Code

- **Infrastructure as Code** or IaC is the process of provisioning and managing infrastructure defined through code
- It allows users to easily **edit** and **distribute configurations**, you can create **reproducible** infrastructure configurations.
- IaC is a process that **automates** the **provisioning** and management of cloud resources.
- IaC software takes input **scripts describing the desired state** and then communicates with the cloud vendors.
- **Programmable Infrastructure**; IaC configures infrastructure exactly like programming software.
- IaC is one way of **raising the standard** of infrastructure management and time to deployment.
- IaC can safely create and configure infrastructure elements **in seconds**.



<https://dzone.com/articles/5-principles-of-infrastructure-as-code-iac>

Benefits of Infrastructure as Code

- **Speed**

By avoiding manual intervention, infrastructure deployments are quick and safe.

- **Consistency**

Deploy identical infrastructure across the board, avoiding edge-cases and one-off configurations.

- **Reusability**

IaC makes it easy to create reusable modules.

- **Reduced cost**

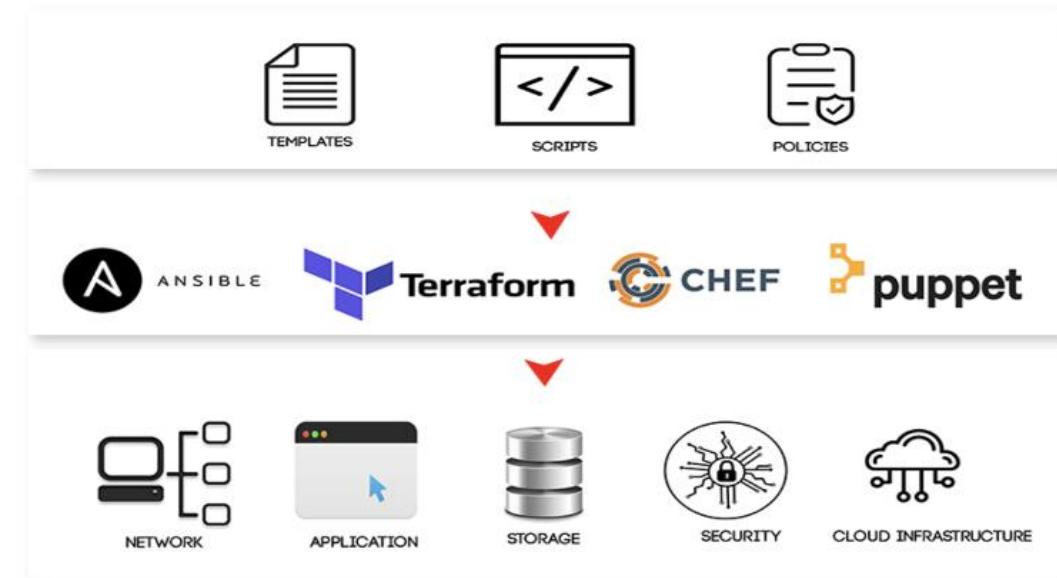
IaC allows virtual machines to be managed programmatically

- **Source control**

Code can be checked in source control for increased transparency and accountability.

- **Agility**

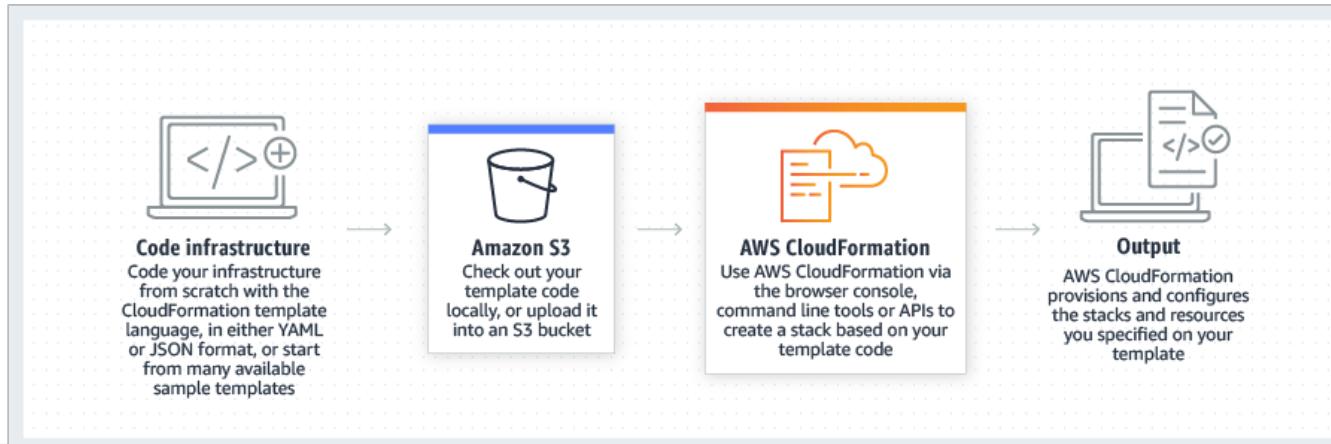
DevOps has made software delivery more efficient.



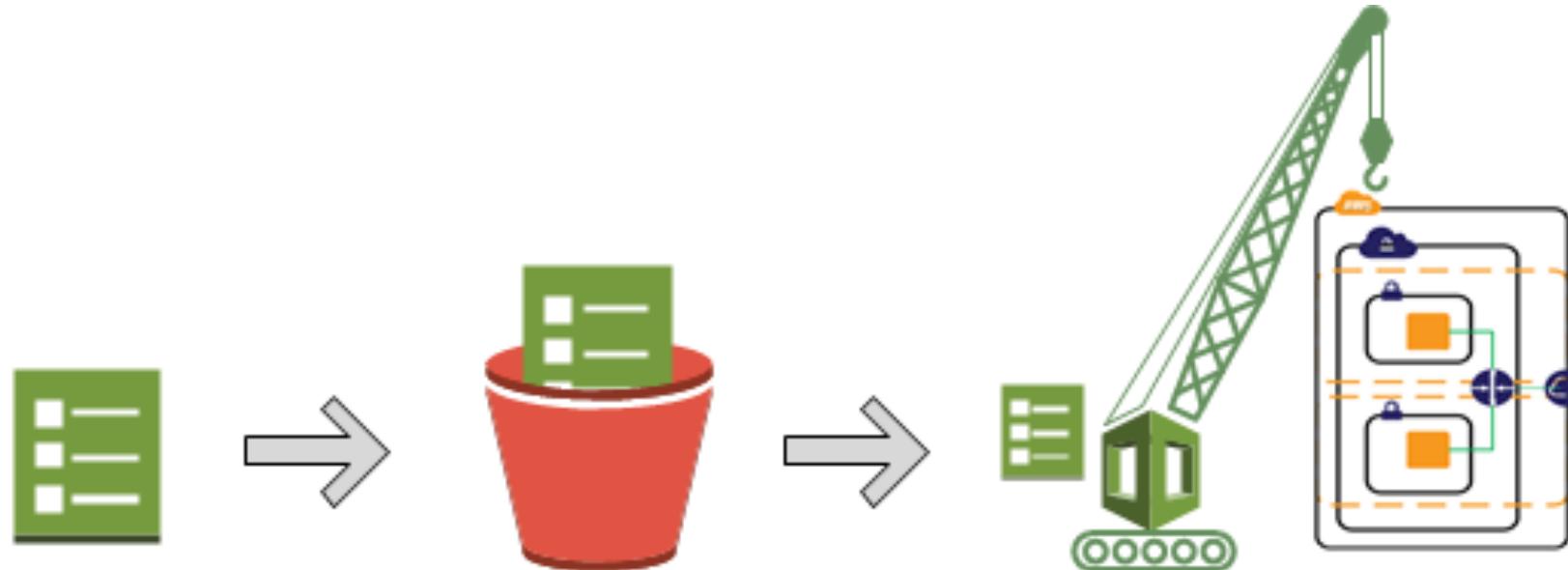
<https://dzone.com/articles/5-principles-of-infrastructure-as-code-iac>

AWS Cloud Formation

- **Model, provision, and manage** AWS resources by treating **infrastructure as code**.
- Infrastructure **automation platform** for AWS that deploys AWS resources in a **repeatable, testable and auditable** manner.
- Uses **template files** to automate the setup of AWS resources
- Enables you to **create and provision** AWS infrastructure deployments **predictably and repeatedly**.
- Described as **infrastructure automation** or Infrastructure-as-Code (IaC) tool.
- Create a **template** that describes all the AWS resources and **CloudFormation** takes care of **provisioning**.



How does AWS CloudFormation work?



1 Create or use an existing template

```
{  
  "AWSTemplateFormatVersion" : "2010-09-09",  
  "Description" : "A simple EC2 instance",  
  "Resources" : {  
    "MyEC2Instance" : {  
      "Type" : "AWS::EC2::Instance",  
      "Properties" : {  
        "ImageId" : "ami-0ff8a91507f77f867",  
        "InstanceType" : "t2.micro"  
      }  
    }  
  }  
}
```

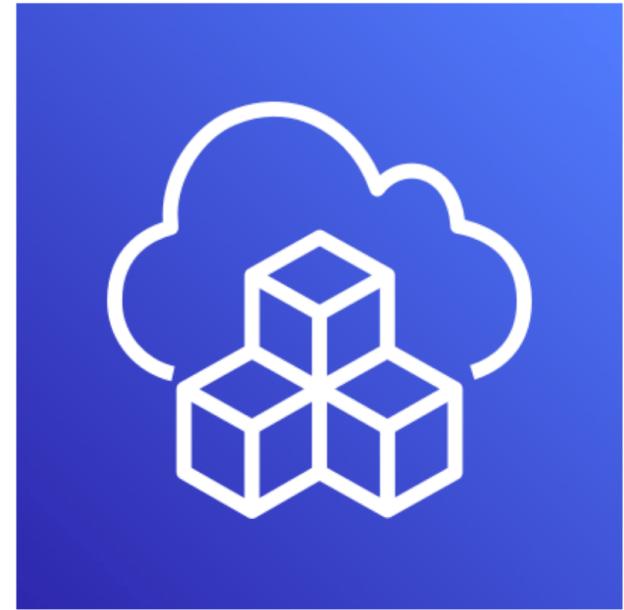
2 Save locally or in S3 bucket

3 Use AWS CloudFormation to create a stack based on your template. It constructs and configures your stack resources.

<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-whatis-howdoesitwork.html>

What is AWS Cloud Development Kit (AWS CDK) ?

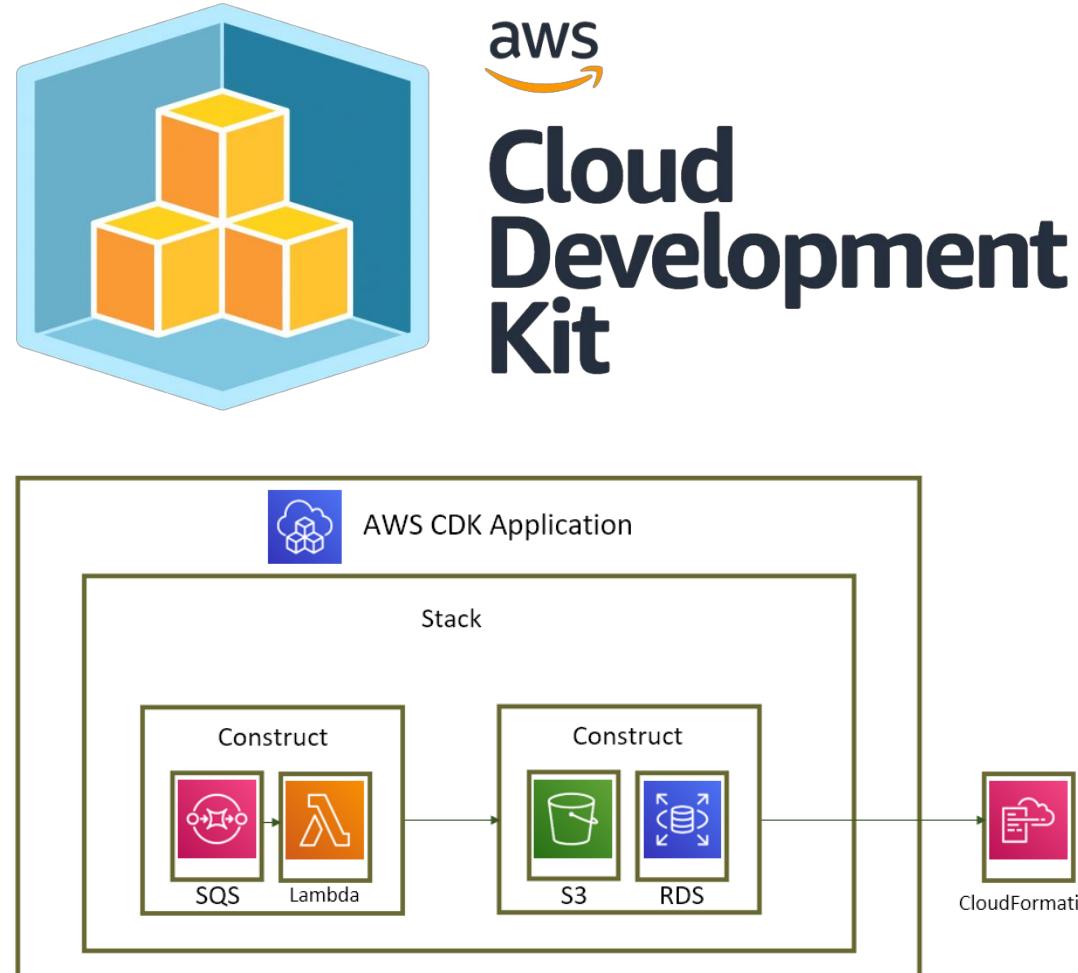
- Open-source software development framework to **define** your cloud application **resources** using **familiar programming languages**.
- Uses the **familiarity** and **expressive** power of **programming languages** for modeling your applications.
- Provides **high-level components** called constructs that **preconfigure cloud resources** with proven defaults.
- Provides a **library of constructs** in many programming languages to easily automate **AWS infrastructure**.
- **Provisions** your **resources** in a safe, repeatable manner through **AWS CloudFormation**.
- First-class support for **TypeScript**, **JavaScript**, **Python**, **Java**, and **C#**.
- **Why** we use AWS CDK ?



AWS CDK:
What is it?

Benefits of AWS CDK

- **Choose a programming language of your choice**
AWS CDK currently supports JavaScript, TypeScript, Python, Java, C#, and Go programming languages..
- **Powered by AWS CloudFormation**
Enables you to define your infrastructure with code and provision it through AWS CloudFormation.
- **Auto-complete and inline documentation**
Autocomplete and inline documentation while coding the infrastructure.
- **Deploy infrastructure and runtime code together**
Reference your runtime code assets in the same project with the same programming language.
- **Developer-friendly command-line interface (CLI)**
Enables you to interact with your CDK applications.



AWS CDK Core Concepts - Apps - Stacks - Constructs - Environments

- **Apps**

Include everything needed to deploy your app to a cloud environment.

- **Stack**

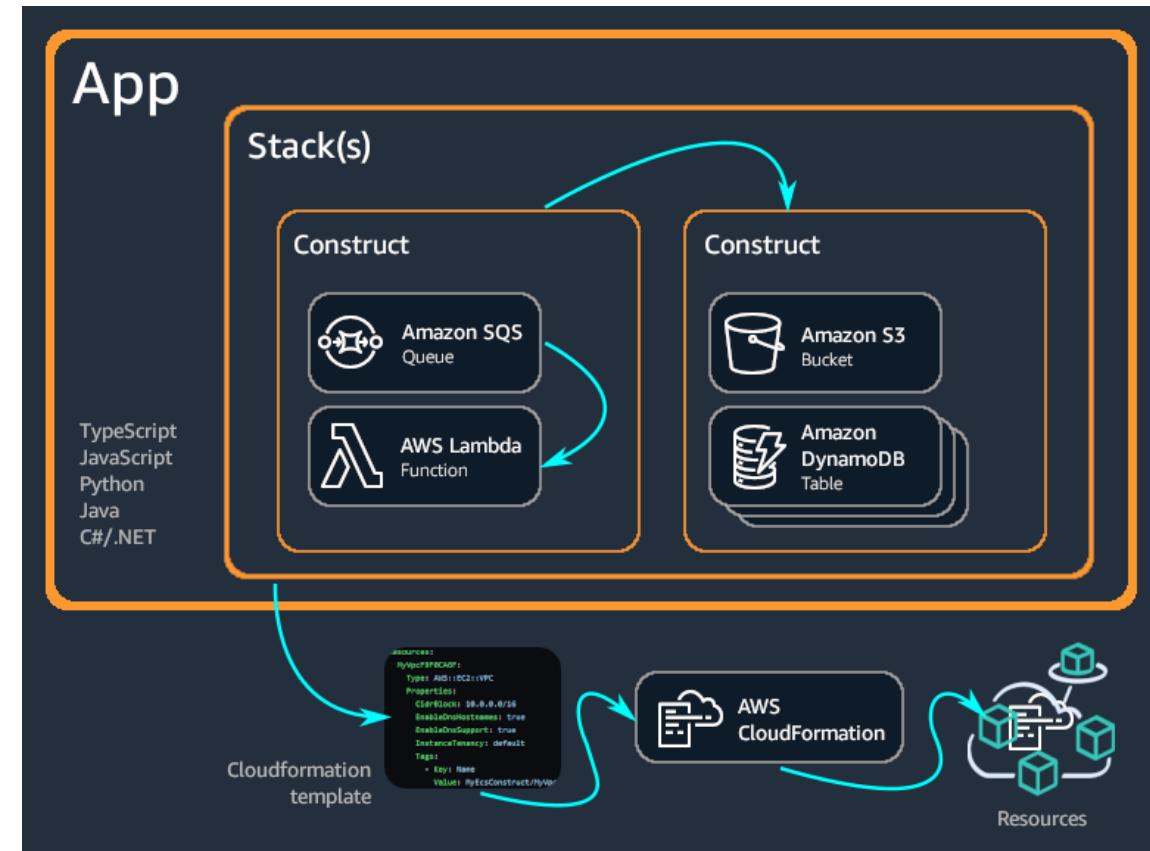
The unit of deployment in the AWS CDK is called a stack.

- **Constructs**

The basic building blocks of AWS CDK apps. A construct represents a “cloud component”.

- **Environments**

Each Stack instance in your AWS CDK app is explicitly or implicitly associated with an environment.



<https://docs.aws.amazon.com/cdk/v2/guide/home.html>

Example CDK application includes Apps, Stacks, Constructs and Environments

```
import { App, Stack, StackProps } from 'aws-cdk-lib';
import * as s3 from 'aws-cdk-lib/aws-s3';

class HelloCdkStack extends Stack {
    constructor(scope: App, id: string, props?: StackProps) {
        super(scope, id, props);

        new s3.Bucket(this, 'MyFirstBucket', {
            versioned: true
        });
    }
}

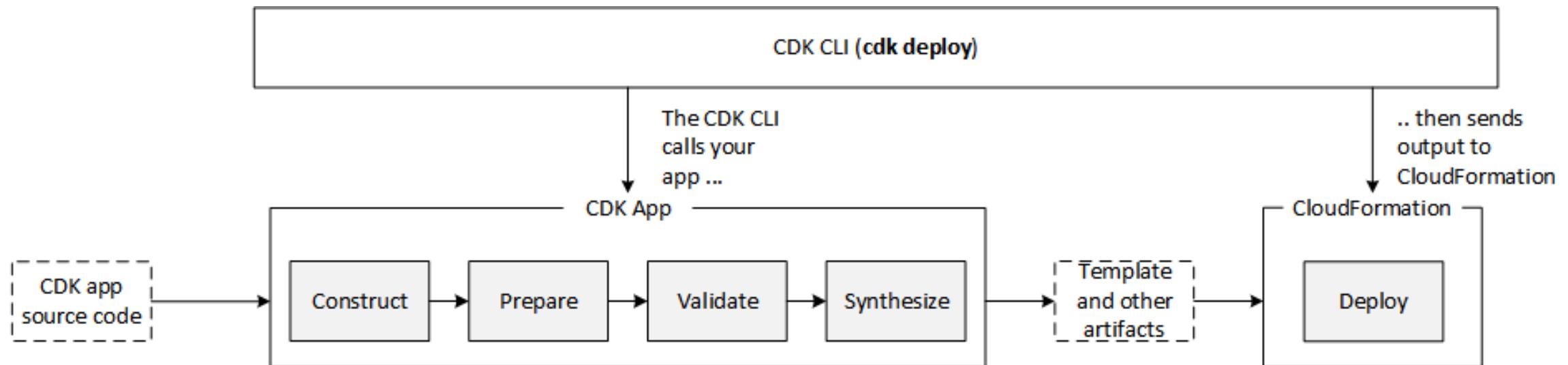
const app = new App();
new HelloCdkStack(app, "HelloCdkStack");
```

<https://docs.aws.amazon.com/cdk/v2/guide/constructs.html>



AWS CDK Lifecycle

- Construction – Preparation – Validation – Synthesis - Deployment



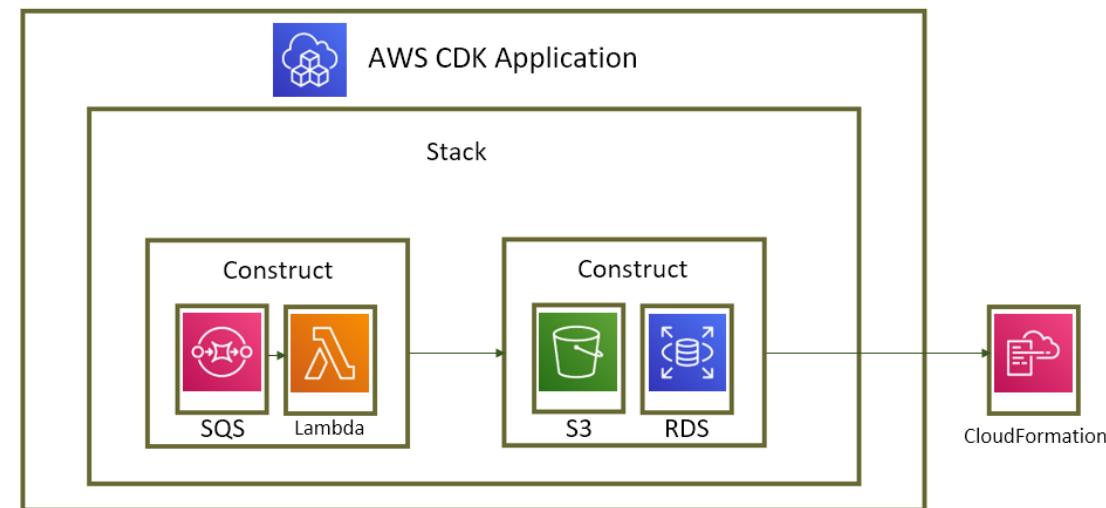
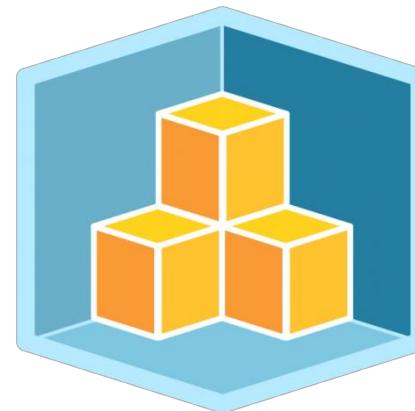
<https://docs.aws.amazon.com/cdk/v2/guide/apps.html>

Getting Started with AWS CDK with Developing our first CDK Application

→ Getting Started with AWS CDK with Developing our first CDK Application.

Getting Started with AWS CDK

- Learn **structure** of a **AWS CDK project**.
- Learn how to use the **AWS Construct Library** to define AWS resources using code.
- Learn how to **synthesize**, **diff**, and **deploy** collections of resources using the AWS CDK Toolkit command-line tool.
- Breakdown what are going to do;
 - Create the app from a **template** provided by the AWS CDK
 - Add code to the app to create resources within stacks
 - **Build** the app its optional; the **AWS CDK Toolkit** will do it for you
 - **Synthesize** stacks in the app to create an AWS CloudFormation template
 - **Deploy** one or more stacks to your AWS account that we configured.
- Follow best practices when developing cdk applications;
 - The **build** step **catches syntax** and type errors.
 - The **synthesis** step catches logical errors in defining your AWS resources
 - The **deployment** may find permission issues.



Prerequisites - AWS CDK Toolkit

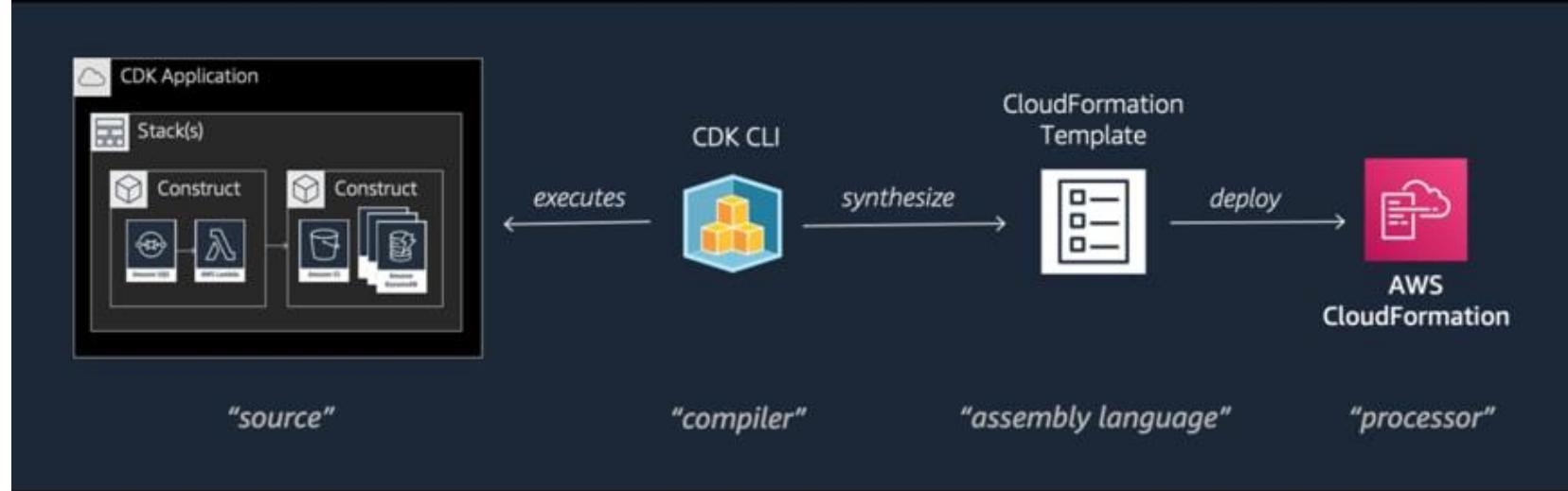
- The AWS CDK Toolkit, the CLI command `cdk`, is the primary tool for **interacting with your AWS CDK app**.
- Executes your app, manages the application model you defined, and produces and deploys the **AWS CloudFormation** templates generated by the AWS CDK.
- **AWS CDK Toolkit** is installed with the npm - Node Package Manager.
- **npm install -g aws-cdk**
- Install latest version



<https://aws.amazon.com/blogs/developer/tag/aws-cdk/>

Bootstrapping CDK Stack

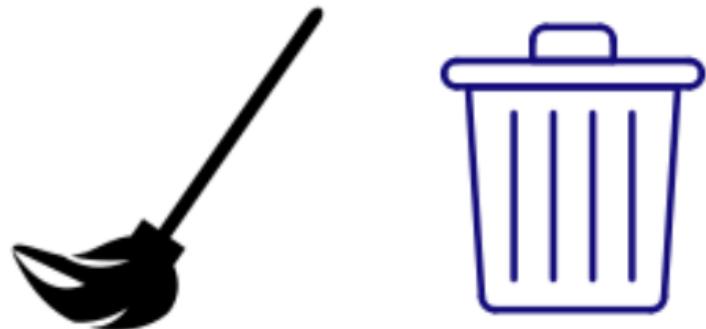
- **Lifecycle** of CDK applications. It required to bootstrap CDK at the beginning of the project.



- In official explanation, **Deploying AWS CDK** apps into an AWS environment (a combination of an AWS account and region) may require that you provision resources the AWS CDK needs to perform the deployment.
- These resources include an **Amazon S3 bucket** for storing files and **IAM roles** that grant permissions needed to perform deployments.

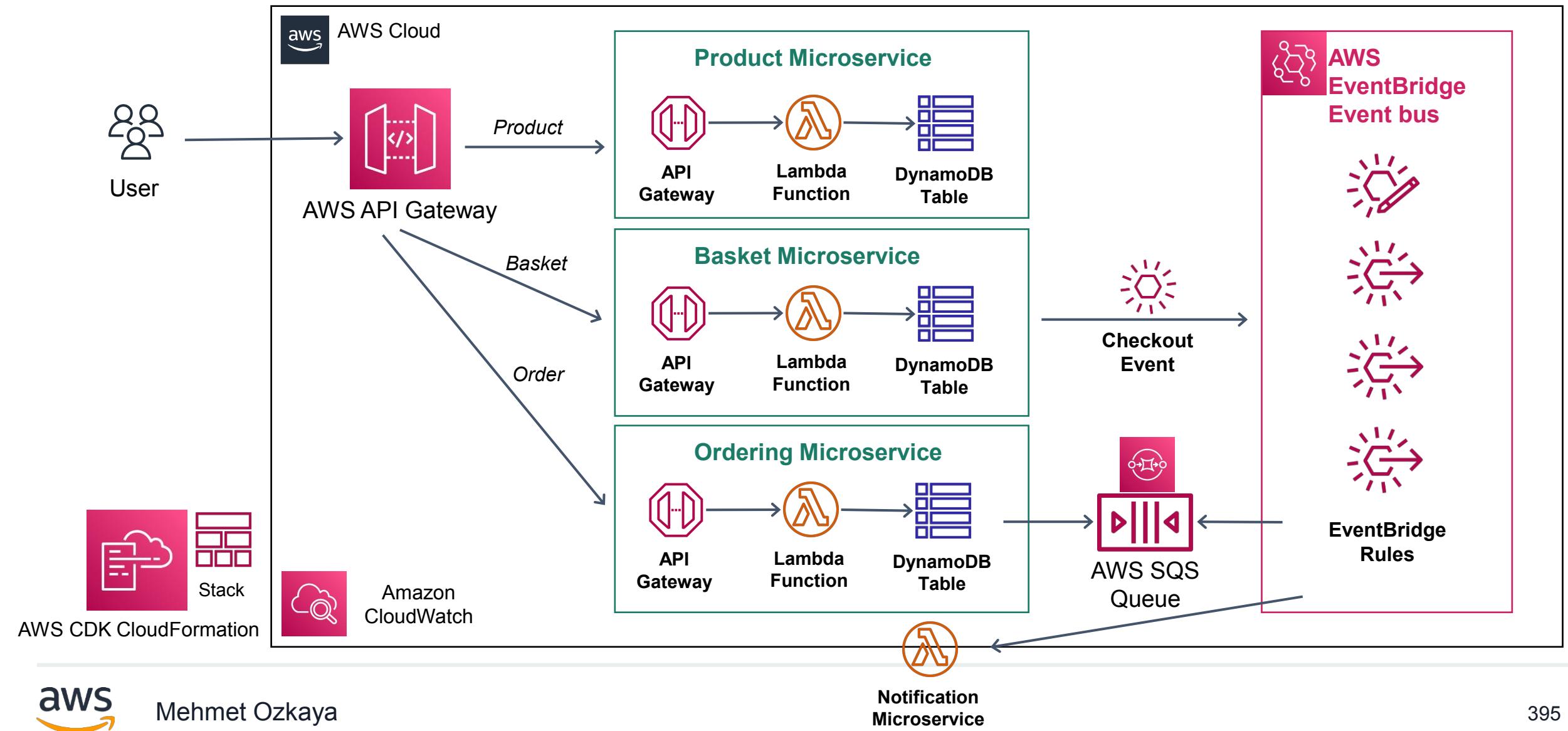
Clean up Resources

- Delete AWS Resources that we create during the section.



Assignment : AWS CDK - Hands-on IaC Development

Serverless Microservices for Ecommerce



Thanks

→ Thank you so much for being with me on this journey.
Reviews and feedback is really encourage to me for pushing forward
to create new courses like this.
Mehmet Ozkaya



Course Logo

