

# Intro to AWS

## MODULE 2 – LAMBDA FUNCTIONS



# Lambda Functions

## Expectations:

### Create Functions

- How Lambda works
- Best practices for writing Lambda functions
- Configuration considerations

### Deploy and Monitor Functions

- Deploying and testing serverless applications
- Monitoring and troubleshooting Lambda functions

# Lambda Functions

## What is AWS Lambda?

### Serverless computing

Serverless computing extends the abstraction of infrastructure in the cloud. You focus on the code for your applications. There are a few things to think about differently when you write and test your code for serverless computing. But the reduced overhead lets you experiment and innovate faster once you adapt to a serverless approach.

# Lambda Functions

## Serverless computing

### Deployment and Operations

- Configure an Instance
- Update OS
- Install App Platform
- Build and deploy apps
- Configure automatic scaling and load balancing
- Continuously secure and monitor instances
- Monitor and maintain apps

### Serverless Deployment and Operation

- Build and deploy apps
- Monitor and maintain apps

# Lambda Functions

## Services in the AWS Serverless Platform

### Compute



AWS Lambda  
Lambda@Edge

### API Proxy



Amazon API Gateway

### Storage



Amazon S3

### Data Stores



Amazon DynamoDB  
AWS AppSync

### Interprocess Messaging



Amazon SNS  
Amazon SQS

### Orchestration



AWS Step Functions

### Analytics



Amazon Kinesis  
Amazon Athena

### Developer Tools



Serverless Application  
Framework (SAM)

# Lambda Functions

## AWS Lambda is the compute service for serverless

The AWS Serverless Platform includes a number of fully managed services that have tight integration with Lambda and are well suited for serverless applications. There are also developer tools including the Serverless Application Model (SAM) which help simplify deployment of your Lambda functions and serverless applications.



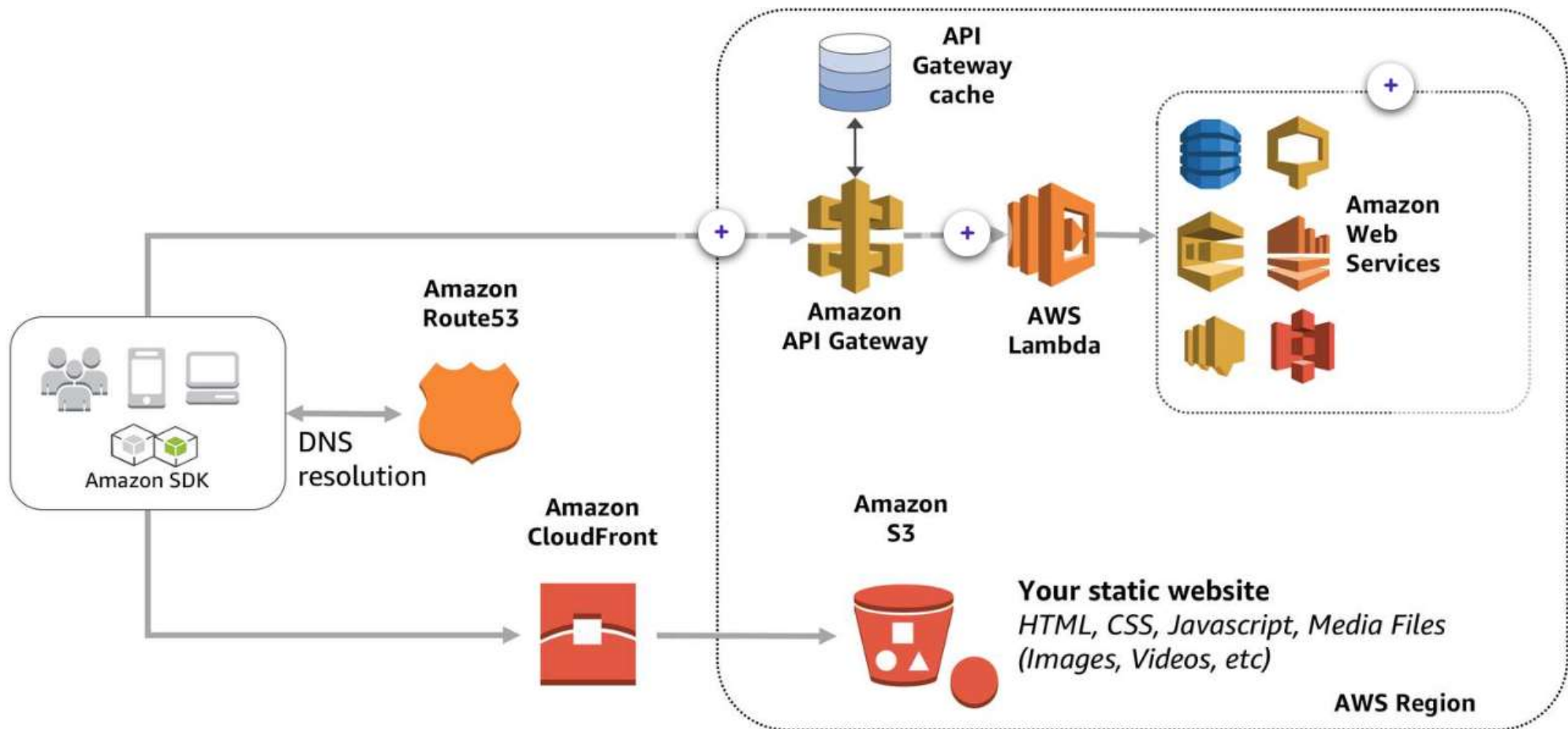
AWS Lambda

### AWS Lambda is at the Heart of Serverless

- Let's you **run code** without provisioning or managing servers.
- **Triggers** on your behalf in response to events.
- **Scales** automatically.
- Provides built in code **monitoring and logging** via Amazon CloudWatch

# Lambda Functions

## Example serverless architecture for a web application





# Lambda Functions

## AWS Lambda features

**Bring your own code:** The code that you write for Lambda isn't written in a new language you have to learn. Development in Lambda is not tightly coupled to AWS so you can easily port code in and out of AWS.

**Integrates with and extends other AWS services:** Within your Lambda function, you can do anything traditional applications can do, including calling an AWS SDK or invoking a third-party API, whether on AWS, in your datacenter, or on the internet.

**Flexible resource and concurrency model:** Instead of scaling by adding servers, Lambda scales in response to events. You configure memory settings and AWS handles details like CPU, Network, and IO throughput.

**Flexible permissions model:** Lambda's permissions model uses AWS Identity & Access Management (IAM) to securely grant access to the desired resources and give you fine-grained control for invoking your functions.



# Lambda Functions

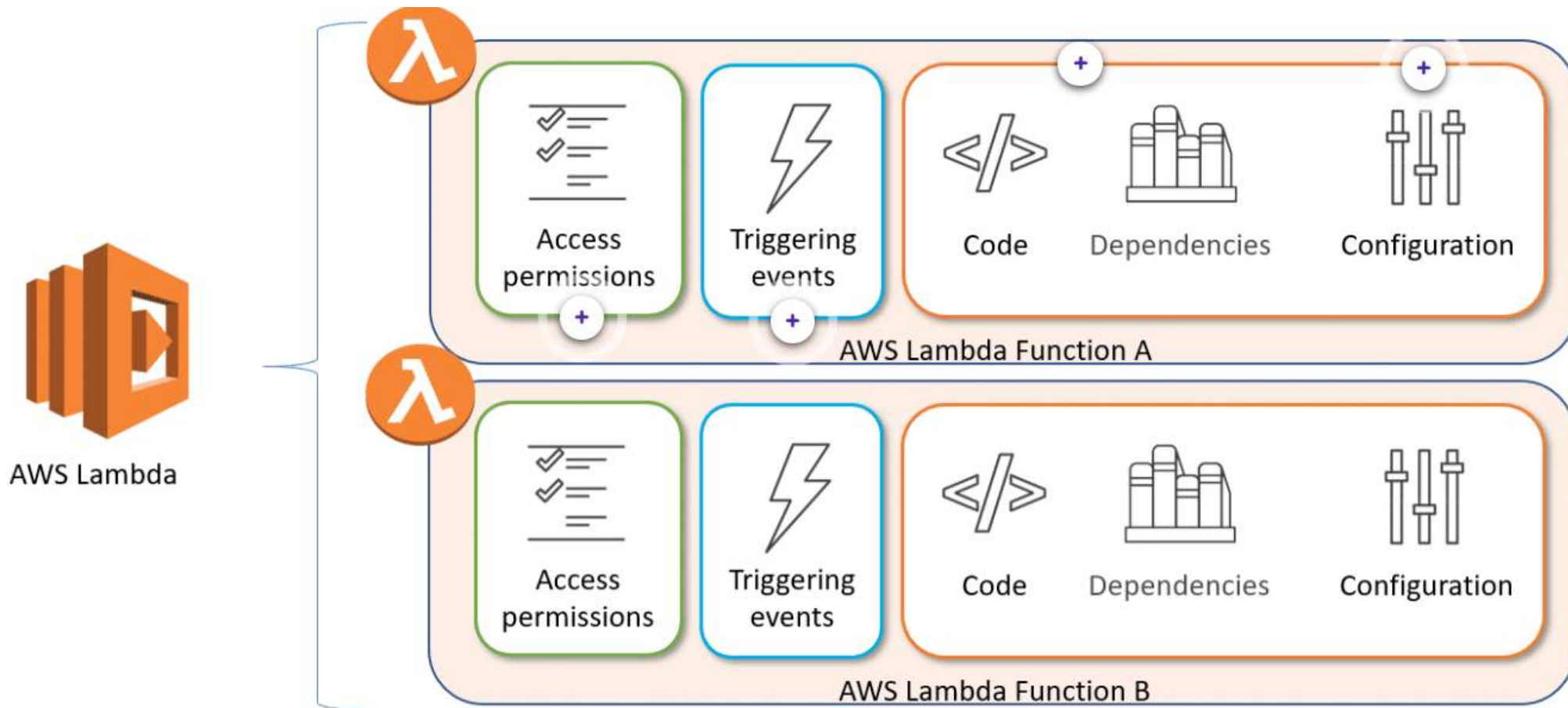
## AWS Lambda features

**Availability and fault tolerance are built in:** Because Lambda is a fully managed service, high availability and fault tolerance are baked into the service without any additional configuration on your part.

**Don't pay for idle:** Lambda functions only run when you trigger them, you never pay for idle capacity.

# Lambda Functions

You build AWS Lambda functions



# Lambda Functions Permissions

## Two sides to access permission

IAM Resource Policy/Function Policy

Permissions to invoke the function

IAM Execution Role

What the function is permitted to do

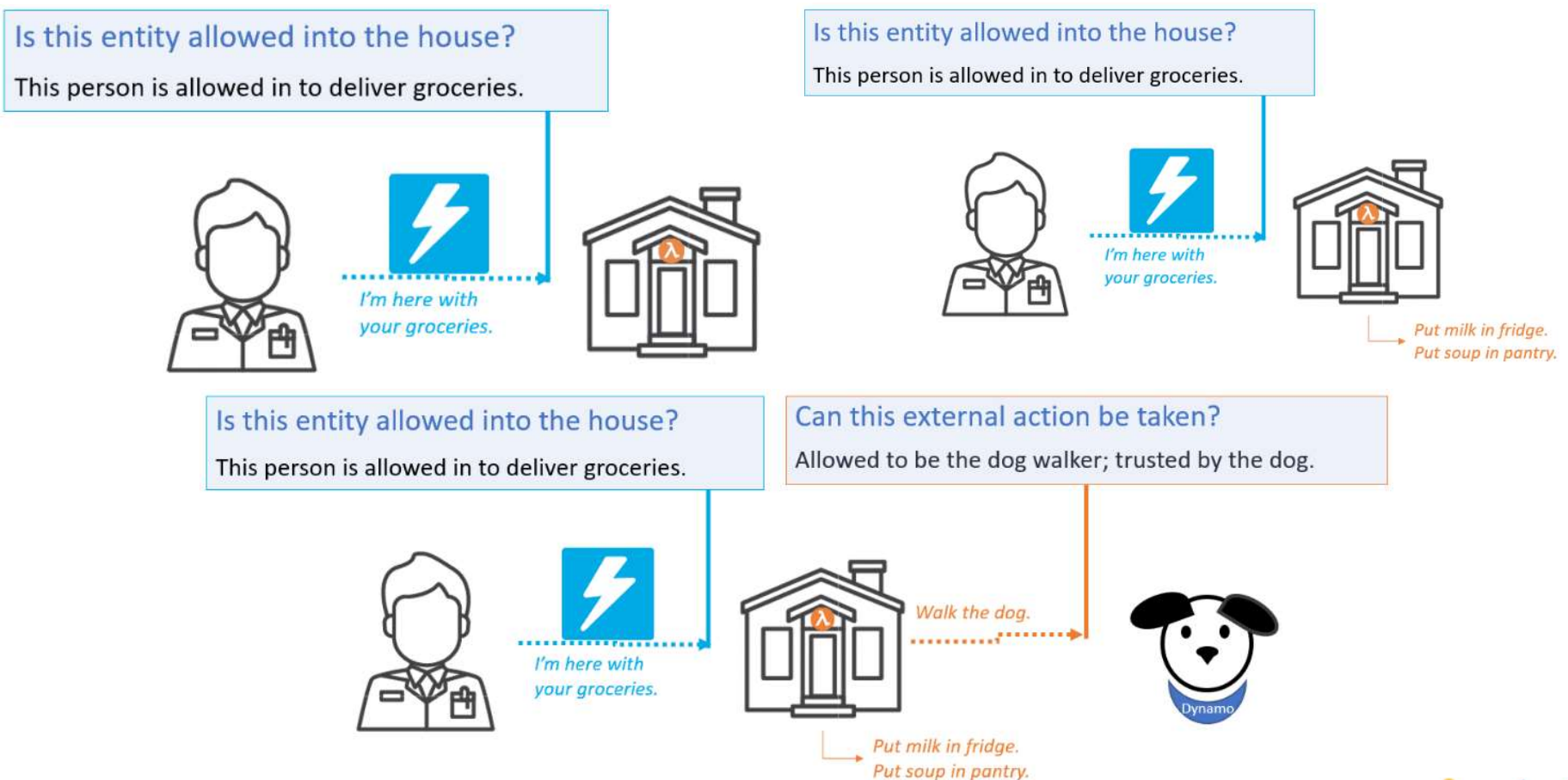


A principal may be a user, role, service, or account. However, in this course, we'll focus on the use case of another AWS service triggering a Lambda function

# Lambda Functions Permissions

## Distinct permissions for distinct purposes

Let's Look at a simple analogy to highlight how these two different components handle Lambda permissions.



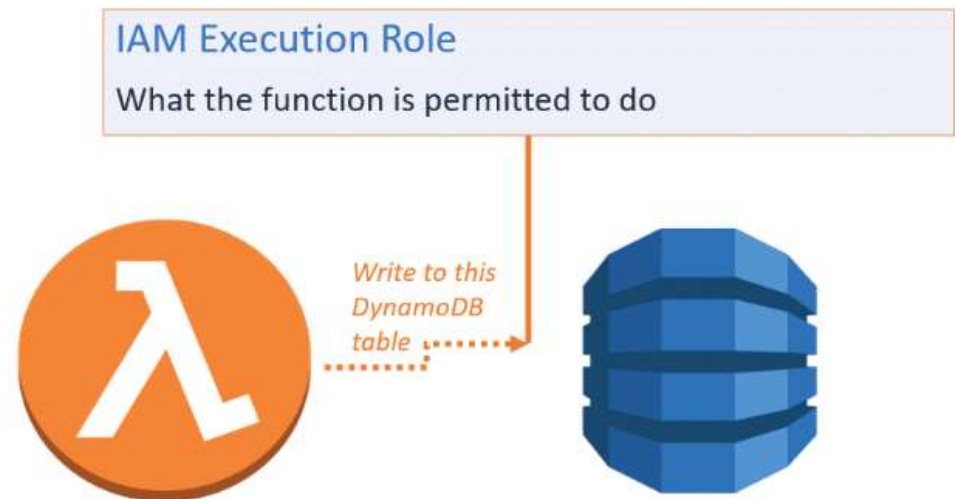
# Lambda Functions Permissions

## Execution role

### Execution role gives permission to interact with services

#### IAM ROLE

- **Selected or created** when you create a Lambda function
- IAM policy includes **actions that can be taken** with the resource
- **Trust policy** that allows Lambda to **AssumeRole**
- **Creator** must have permission for **iam:PassRole**



# Lambda Functions Permissions

## Example execution role policy definitions

### IAM Policy

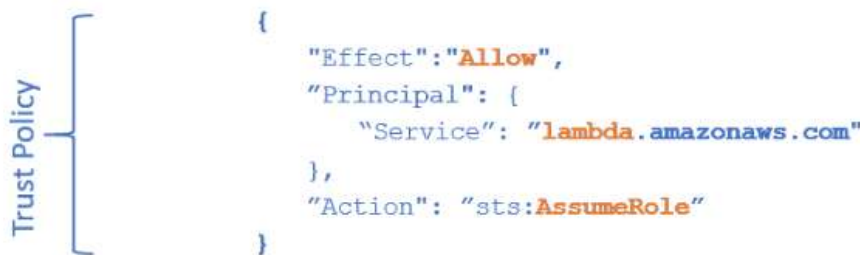
This IAM policy allows the function to perform the DynamoDB PutItem action against a table named Test in the US-West-2 region.



```
{
  "Effect": "Allow",
  "Action": [
    "dynamodb:PutItem"
  ],
  "Resource": "arn:aws:dynamodb:us-west-2:###:table/test"
}
```

### Trust Policy

And the trust policy shows that this role gives Lambda the permission to assume the role and invoke the function on your behalf.



```
{
  "Effect": "Allow",
  "Principal": {
    "Service": "lambda.amazonaws.com"
  },
  "Action": "sts:AssumeRole"
}
```

# Lambda Functions Permissions

## Accessing resources in a VPC

To enable your Lambda function to access resources inside your private virtual private cloud (VPC), you must provide additional VPC-specific configuration information, which includes **VPC subnet IDs and security group IDs**. You also need to include an execution role with permissions to create, describe, and delete elastic network interfaces. Lambda provides a permissions policy named **AWSLambdaVPCAccessExecutionRole** for this purpose.



# Lambda Functions Permissions

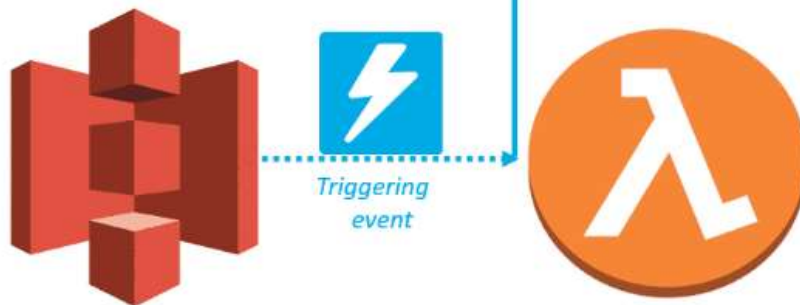
## Resource policy

### Resource policy gives permission to invoke

A resource policy is used to tell the Lambda service which events have permission to invoke the Lambda function. Resource policies also make it easy to grant access to the Lambda function across AWS accounts.

IAM Resource Policy/Function Policy

Permissions to invoke the function



### Lambda function policy

- Policy associated with a "push" event source
- Created when you add a trigger to a Lambda function
- Allows the event source to take the `lambda:InvokeFunction` action

# Lambda Functions Permissions

## Example resource policy (also called Lambda function policy)

This resource policy gives Amazon S3 permission to invoke a Lambda function called myFirstFunction.

Lambda Function Policy

```
{
  "Version": "2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "lambda-fd269e28-988b-4d2b-96ae-eabcd7dc399c",
      "Effect": "Allow",
      "Principal": {
        "Service": "s3.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:function:myFirstFunction",
      "Condition": {
        "ArnLike": {
          "AWS:SourceArn": "arn:aws:s3:::myBucket1"
        }
      }
    }
  ]
}
```

# Lambda Functions Permissions

## Resource policies and execution roles on the AWS Lambda console

The screenshot shows the AWS Lambda console interface for a function's permissions. The top navigation bar includes 'Lambda > Functions > FAQ'. The main header has tabs for 'Configuration', 'Permissions' (selected), and 'Monitoring'. On the right, there are buttons for 'Throttle', 'Qualifiers', 'Actions', 'BasicTest', 'Test', and 'Save'. A 'View role JSON' button is highlighted with a red circle and a cursor.

**Resource summary**

AWS X-Ray  
2 actions, 1 resource

Below is a summary of the AWS X-Ray resources and API actions that your function's code has access to, as configured by its execution role. [Manage these permissions](#)

By action | **By resource**

Resource	Actions
All resources	Allow: xray:PutTraceSegments Allow: xray:PutTelemetryRecords

ⓘ Lambda obtained this information from the following policy statements:

- Managed policy AWSLambdaTracerAccessExecutionRole-fac04ded-77fe-4dc8-8694-66feb7db75cc, statement 0

**Function policy** Info

```
1 {
2   "Version": "2012-10-17",
3   "Id": "default",
4   "Statement": [
5     {
6       "Sid": "lambda-863af989-1722-45a1-af45-9b0f0c16e7bf",
7       "Effect": "Allow",
8       "Action": [
9         "xray:PutTraceSegments",
10        "xray:PutTelemetryRecords"
11      ],
12       "Resource": "*"
13     }
14   ]
15 }
```

**Role JSON**

```
1 {
2   "permissionsBoundary": {},
3   "roleName": "lambda_basic_execution",
4   "policies": [
5     {
6       "document": {
7         "Version": "2012-10-17",
8         "Statement": [
9           {
10            "Effect": "Allow",
11            "Action": [
12              "sqs:SendMessage"
13            ],
14            "Resource": "*"
15          }
16        ]
17      },
18      "name": "SQSSendMessage",
19      "type": "inline"
20    },
21    {
22      "document": {
23        "Version": "2012-10-17",
24        "Statement": {
25          "Effect": "Allow",
26          "Action": [
27            "xray:PutTraceSegments",
28            "xray:PutTelemetryRecords"
29          ],
30          "Resource": "*"
31        }
32      }
33    }
34  ]
35 }
```

## Resource policies and execution roles on the AWS Lambda console



# Lambda Functions Permissions

## Resource policies and execution roles on the AWS Lambda console



# How Lambda Works

## Event sources

### Lots of services can be event sources

An event source is the entity that publishes events, and a Lambda function is the custom code that processes the events. Configuration of services as event triggers is referred to as Event Source Mapping.

Event source **triggers** your  
Lambda function



#### Event Sources

- Data stores
- Endpoints
- Repositories
- Message services



Amazon S3



Amazon  
DynamoDB



Amazon  
Kinesis



Amazon  
Cognito



Amazon RDS  
Aurora

# How Lambda Works

## Event sources

Event source **triggers** your  
Lambda function



### Event Sources

- Data stores
- Endpoints
- Repositories
- Message services



Amazon  
Alexa



Amazon API  
Gateway



Amazon Step  
Functions



Amazon  
IoT



# How Lambda Works

## Event sources

Event source **triggers** your  
Lambda function



### Event Sources

- Data stores
- Endpoints
- **Repositories**
- Message services



Amazon  
CloudWatch



Amazon  
CloudFormation



Amazon  
CloudTrail



Amazon  
CodeCommit

# How Lambda Works

## Event sources

Event source **triggers** your  
Lambda function



### Event Sources

- Data stores
- Endpoints
- Repositories
- Message services



Amazon SES



Amazon SNS



Amazon SQS



Cron Events

# How Lambda Works

## Execution models for invoking Lambda functions

Each of the event sources just mentioned will invoke Lambda in one of these execution models:

### Push Events

- Synchronous
- Asynchronous

Service delivers events directly to function



### Polling Events

- Stream-based
- Not stream-based

Lambda polls for events and delivers to function



A *synchronous* push is like UPS delivering a package, and waiting for a signature in order to complete delivery. An *asynchronous* push is the mailman leaving it in your mailbox where you pick it up on your schedule.

Think of *polling* events as having the Lambda service watch the suitcases on the airport luggage carousel and deliver them to your house in batches.

# How Lambda Works

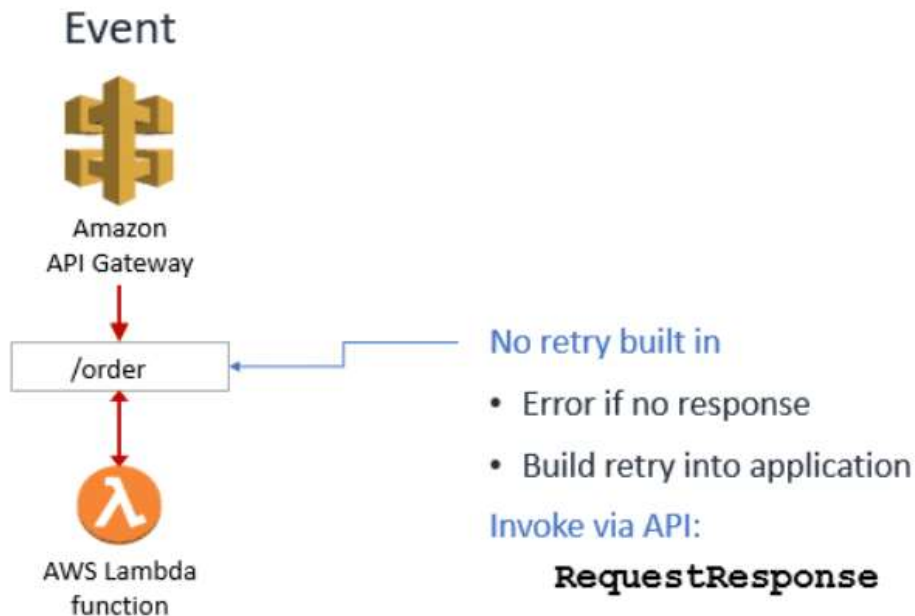
## Synchronous vs. asynchronous event sources

Synchronous events expect an immediate response from the function invocation.

With this execution model, there is no built-in retry in Lambda. You must manage your retry strategy within your application code.

To invoke Lambda synchronously via API, use **RequestResponse**.

### Synchronous Push



# How Lambda Works

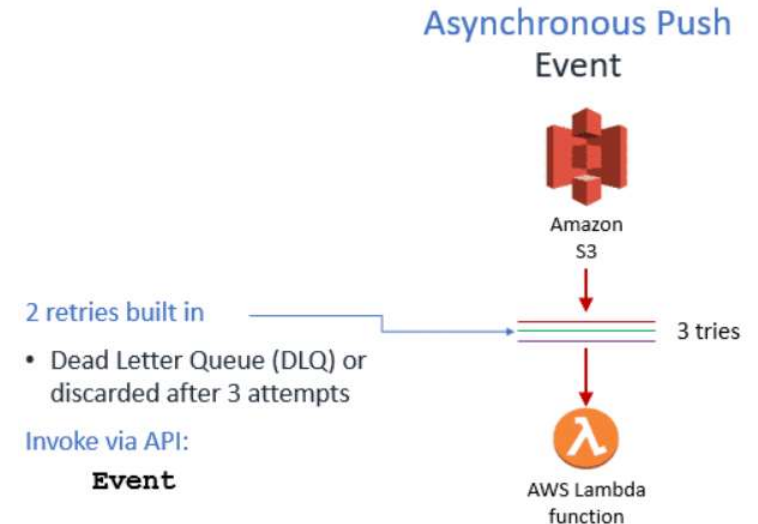
## Synchronous vs. asynchronous event sources

Asynchronous events are queued, and the requestor doesn't wait for the function to complete.

This model makes sense for batch processes. With an async event, Lambda automatically retries the invoke twice more on your behalf. You also have the option to enable a dead-letter queue on your Lambda function.

In November, 2019, two new error handling options were added to give you more control over failed records from asynchronous event sources: Maximum Event Age and Maximum Retry Attempts.

To invoke functions asynchronously via API, use `Event`.



# How Lambda Works

## Notes:

- When invoking a Lambda function programmatically, you must specify the invocation type.
- When AWS services are sources, the invocation type is predetermined.
- For your convenience, the Lambda console shows all event sources for a given function.
- When you select Test from the Lambda console, it always invokes your Lambda function synchronously.

# How Lambda Works

## Examples of AWS services that invoke Lambda synchronously

- Amazon API Gateway
- Amazon Cognito
- AWS CloudFormation
- Amazon Alexa
- Amazon Lex
- Amazon CloudFront



# How Lambda Works

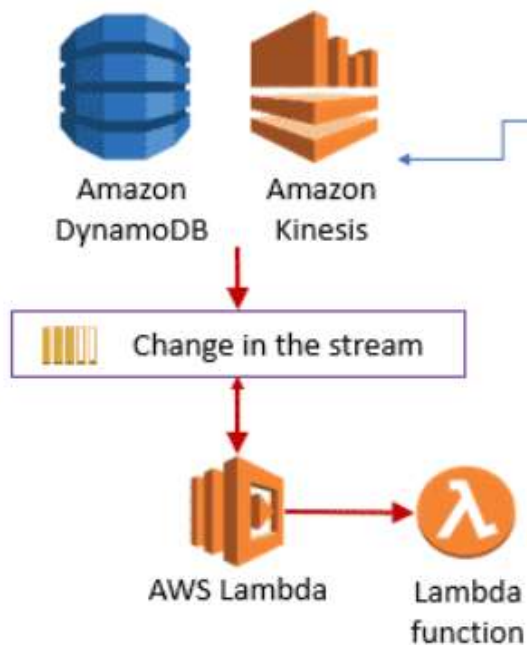
## Examples of AWS services that invoke Lambda synchronously

- Amazon API Gateway
- Amazon Cognito
- AWS CloudFormation
- Amazon Alexa
- Amazon Lex
- Amazon CloudFront

# How Lambda Works

## Polling events

### Stream-Based Polling Event



Services put items into the stream or queue

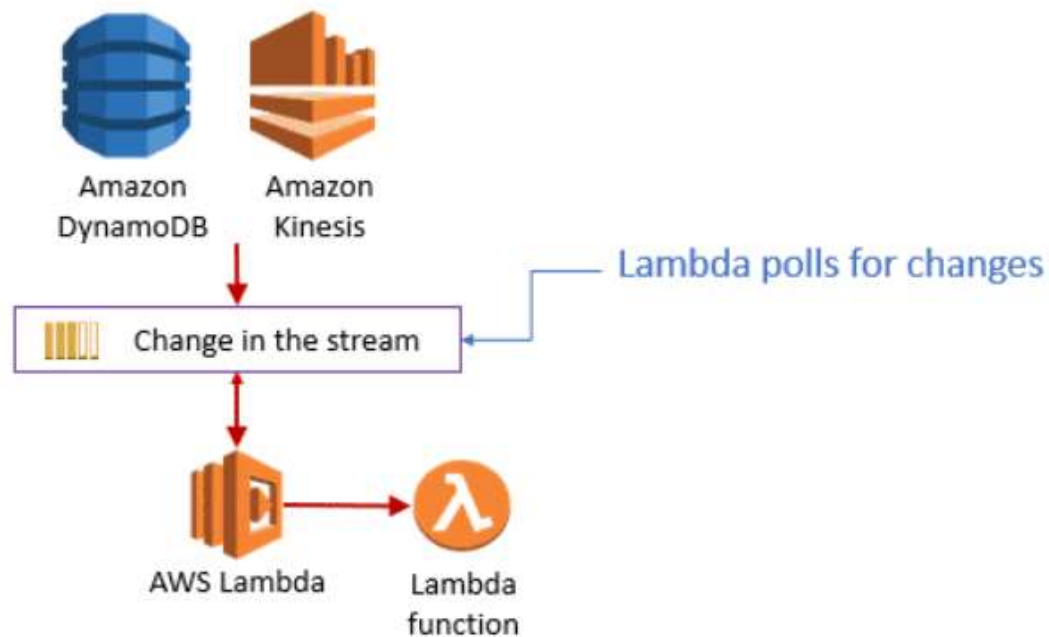
### Non-Streaming Polling Event



# How Lambda Works

## Polling events

### Stream-Based Polling Event



### Non-Streaming Polling Event



# How Lambda Works

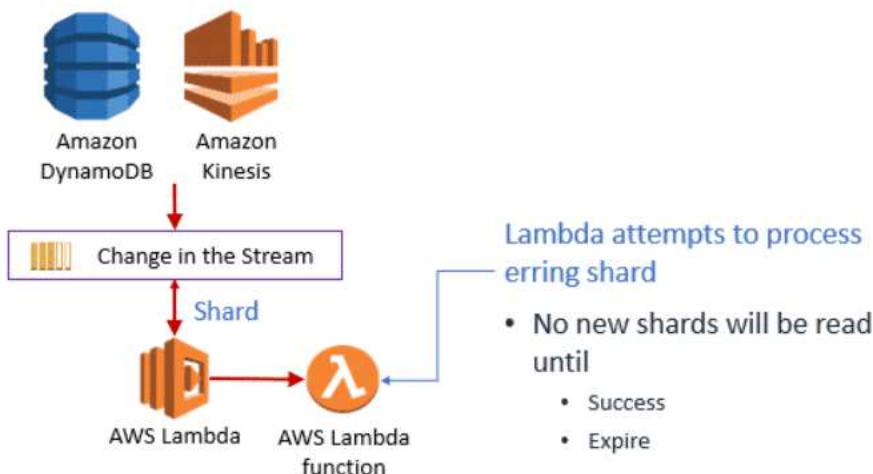
## Stream vs. queue

One consideration is that with stream-based polling there is no cost to make the polling calls, but with SQS polling, standard SQS rates apply for each request. Additionally, there is a key distinction between how errors are handled:

With streams, errors in a shard block further processing

A failure in this model blocks Lambda from reading any new records from the stream until the failed batch of records either expires or is processed successfully. This is important because the **events in each shard from the stream need to be processed in order.**

### Stream-based Polling Event



# How Lambda Works

## Stream vs. queue

With queues, errors in a batch are returned to queue

Lambda will keep retrying a failed message until it is processed successfully, or the retries or retention period are exceeded. If the message fails all retries, it will either go to the DLQ if configured, or it will be discarded.

An error doesn't stop processing of the batch, but there **may be a change to the order in which messages are processed.**



# How Lambda Works

## Stream vs. queue

With queues, errors in a batch are returned to queue

Lambda will keep retrying a failed message until it is processed successfully, or the retries or retention period are exceeded. If the message fails all retries, it will either go to the DLQ if configured, or it will be discarded.

An error doesn't stop processing of the batch, but there **may be a change to the order in which messages are processed.**



# How Lambda Works

## Lifecycle of a Lambda function

1. When a function is first invoked, an execution environment is launched and bootstrapped. Once the environment is bootstrapped, your function code executes. Then, Lambda freezes the execution environment, expecting additional invocations.
2. If another invocation request for the function is made while the environment is in this state, that request goes through a warm start. With a warm start, the available frozen container is thawed and immediately begins code execution without going through the bootstrap process.
3. This thaw and freeze cycle continues as long as requests continue to come in consistently. But if the environment becomes idle for too long, the execution environment is recycled.
4. A subsequent request starts the lifecycle over, requiring the environment to be launched and bootstrapped. This is a cold start.



# How Lambda Works

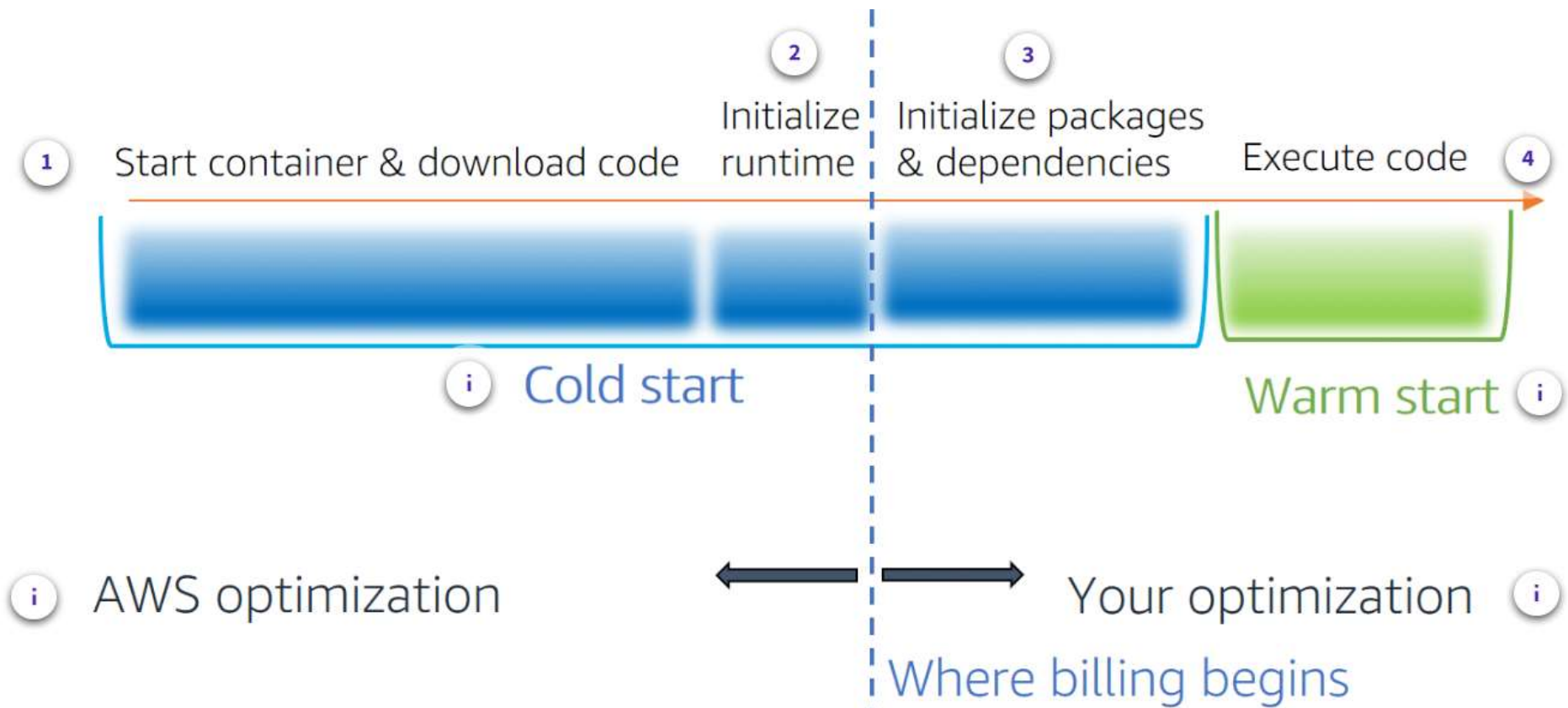
**Best practice: Write functions to take advantage of a warm start**

1. Store and reference dependencies locally.
2. Limit re-initialization of variables.
3. Add code to check for and reuse existing connections.
4. Use tmp space as transient cache.
5. Check that background processes have completed.

# How Lambda Works

## Best practice: Minimize cold start times

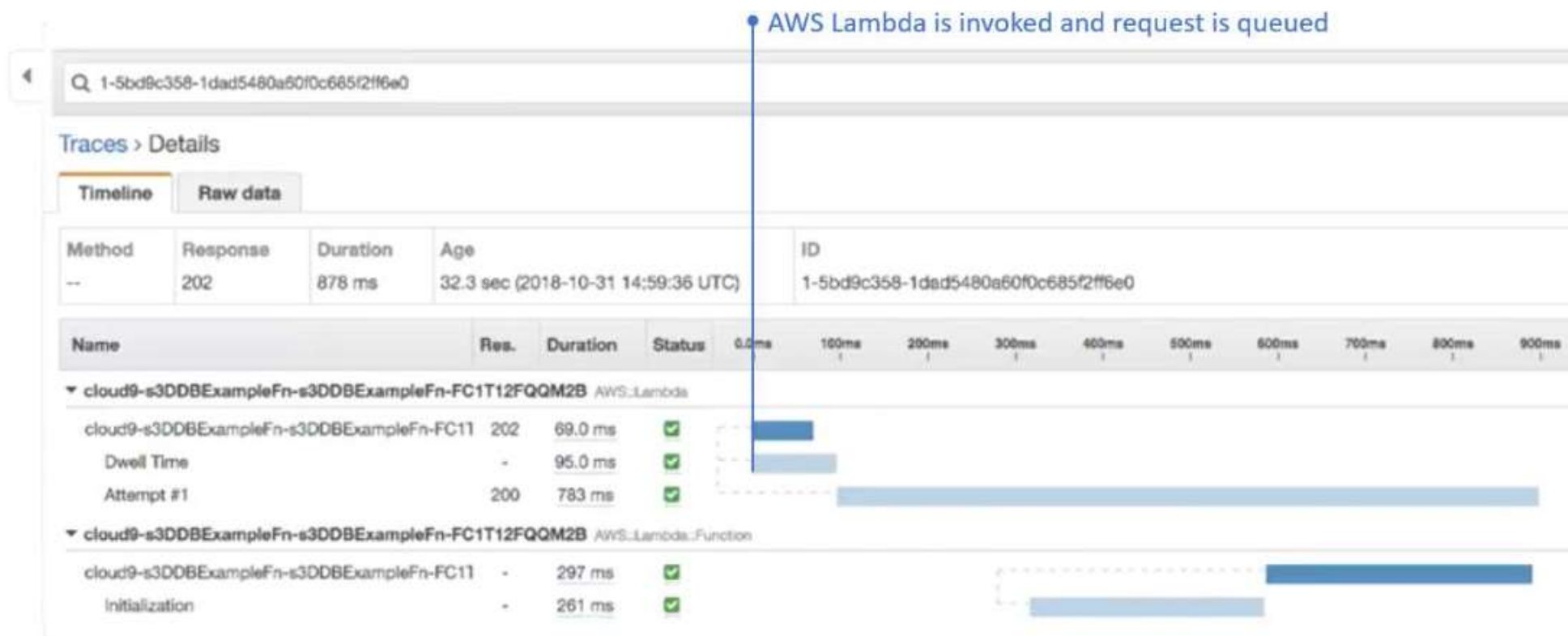
Plan for real-world conditions and traffic patterns—cold starts may not have much impact on your application and you could use provisioned concurrency to keep environments warm. But it's always a good idea to minimize the impact of a cold start.



# How Lambda Works

## Analyzing a cold start using AWS X-Ray

In this example, an object added to a designated S3 bucket triggers the Lambda function invocation.



# Authoring Lambda Functions

## AWS Lambda programming model

### Bring your own code

With Lambda, you can use the language and IDE that you are most familiar with and bring code you've already written. The code may need some adjustments to make it serverless.

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

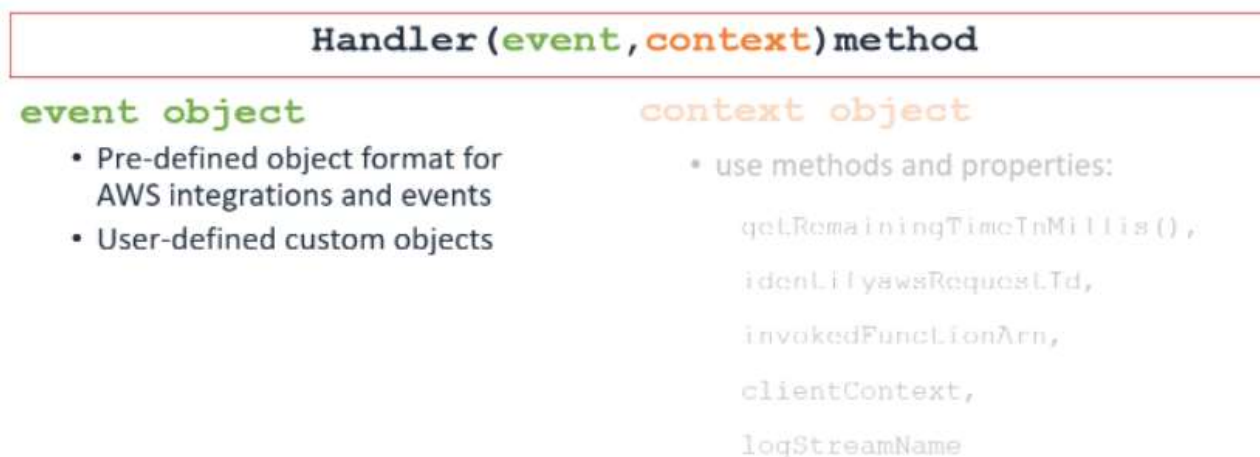
# Authoring Lambda Functions

## Start with the handler method

The handler method is the entry point that AWS Lambda calls to start executing your Lambda function. The handler method always takes two objects: the event object and the context object.

## Event Object

The event object provides information about the event that triggered the Lambda function. This could be a pre-defined object that an AWS service generates, or it could be a custom user-defined object in the form of a serializable string. For example, it could be a pojo or a json stream.



# Authoring Lambda Functions

## Start with the handler method

The handler method is the entry point that AWS Lambda calls to start executing your Lambda function. The handler method always takes two objects: the event object and the context object.

## Context Object

The context object is generated by AWS, and provides metadata about the execution. You can use it to interact with Lambda. For example, it includes `awsRequestId`, `logStreamName`, and the `getRemainingTimeInMillis()` function.

**Handler (event, context) method**

### event object

- Pre-defined object format for AWS integrations and events
- User-defined custom objects

### context object

- Use methods and properties:

```
getRemainingTimeInMillis(),  
awsRequestId,  
invokedFunctionArn,  
clientContext,  
logStreamName
```

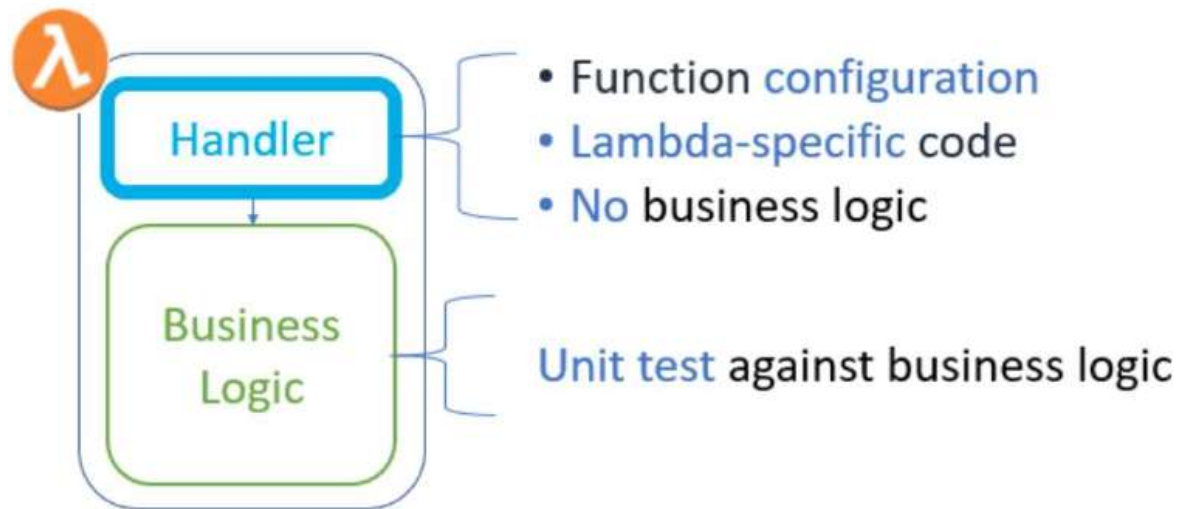
# Authoring Lambda Functions

## Design best practices

### Separate business logic

Separate your core business logic from the handler method.

This not only makes your code more portable, it allows you to target unit tests at your code without worrying about the configuration of the function.



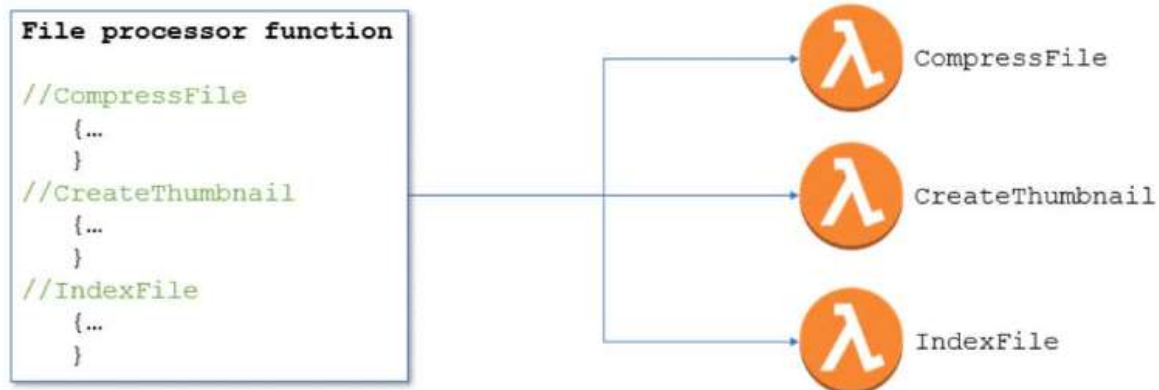
# Authoring Lambda Functions

## Design best practices

### Write modular functions

Create single purpose functions.

Follow the same principles you would apply to developing microservices.





# Authoring Lambda Functions

## Design best practices

### Treat functions as stateless

No information about state should be saved within the context of the function itself.

Because your functions only exist when there is work to be done, it's particularly important for serverless applications to treat each function as stateless.

- DynamoDB is serverless and scales horizontally to handle your Lambda invocations. It also has single-millisecond latency, which makes it a great choice for storing state information.
- Consider Amazon ElastiCache if you have to put your Lambda function in a VPC. It may be less expensive than DynamoDB.
- Amazon S3 can be used as an inexpensive way to store state data, if throughput isn't critical and the type of state data you are saving won't change rapidly..

# Authoring Lambda Functions

## Design best practices

### Only include what you need

Minimize both your deployment package's dependencies and its size. This can have a significant impact on the startup time for your function. For example, only choose the modules that you need—don't include an entire AWS SDK.

Reduce the time it takes Lambda to unpack deployment packages authored in Java. Put your dependency .jar files in a separate /lib directory.

Opt for simpler Java dependency injection (IoC) frameworks. For example choose Dagger or Guice, over more complex ones like Spring Framework.

# Authoring Lambda Functions

## Best practices for writing code

### Include logging statements

Lambda functions can and should include logging statements which are written to CloudWatch. Here's an example of logging using the logger function in Python.

Python example:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
def my_logging_handler(event, context):
    logger.info('got
event{}'.format(event))
    logger.error('something went wrong')
    return 'Hello from Lambda!'
```

# Authoring Lambda Functions

## Best practices for writing code

### Include results info

Functions must give Lambda information about the results of their execution.

Use the return coding that is appropriate for your selected language to exit your code.

For languages such as Node.js, Lambda provides additional methods on the context object for callbacks. You use these context-object methods to tell Lambda to terminate your function and optionally return values to the caller.

Node.js example

```
exports.myHandler = function(event,
context, callback) {
    ... function code
    callback(null, "some success message");
    //or
    //callback("some error type");
}
```

# Authoring Lambda Functions

## Best practices for writing code

### Use environment variables

Take advantage of environment variables for operational parameters.

They allow you to pass updated configuration settings without changes to the code itself.

You can also use environment variables to store sensitive information required by the function.

# Authoring Lambda Functions

## Best practices for writing code

### Avoid recursive code

Avoid a situation where a function calls itself.

This could continue to spawn new invocations that would make you lose control of your concurrency.

You can quickly set the concurrent execution limit to zero by using the console or command line to immediately throttle requests if you accidentally deploy recursive code.

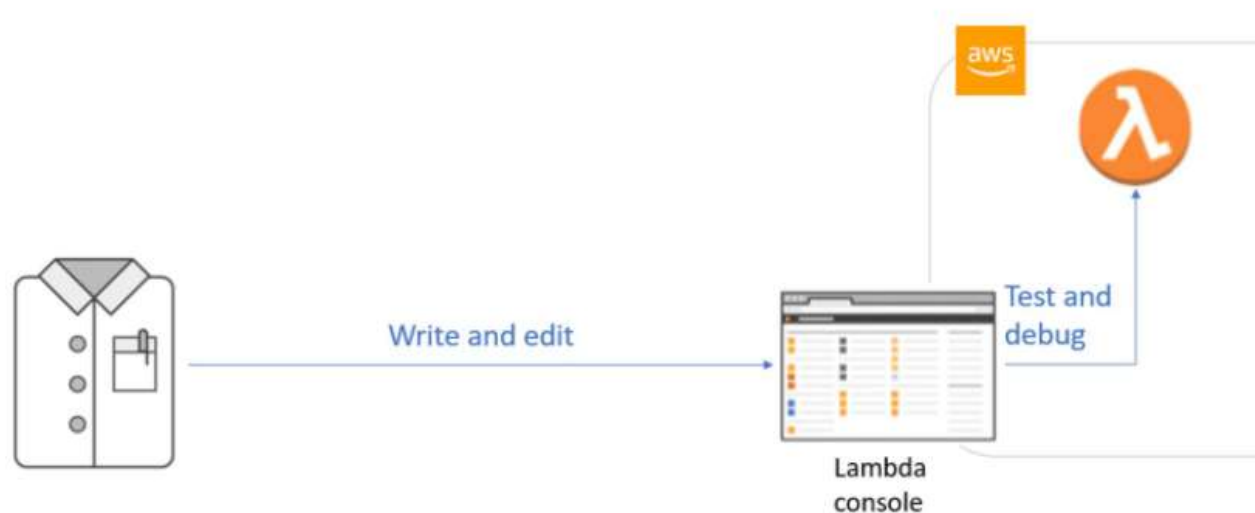
# Authoring Lambda Functions

## Authoring Lambda code

You can use any one of three methods to author Lambda Functions:

### Lambda Console Editor

Edit your code inline through the Lambda Management Console.



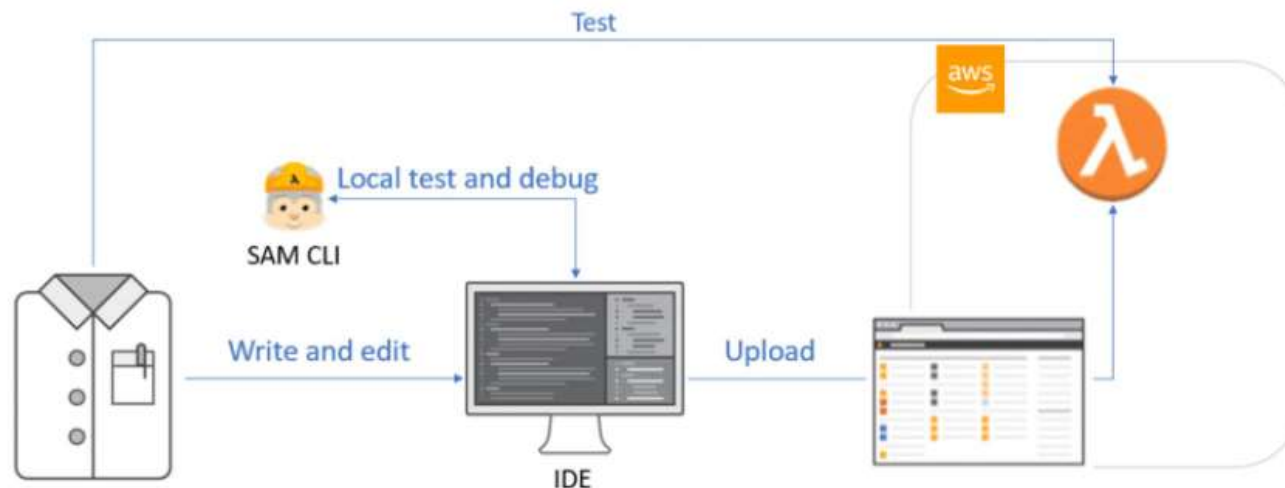
# Authoring Lambda Functions

## Authoring Lambda code

You can use any one of three methods to author Lambda Functions:

### Upload to Lambda Console

Create your deployment package with your IDE and upload the package to Lambda using the console's Upload Package option.





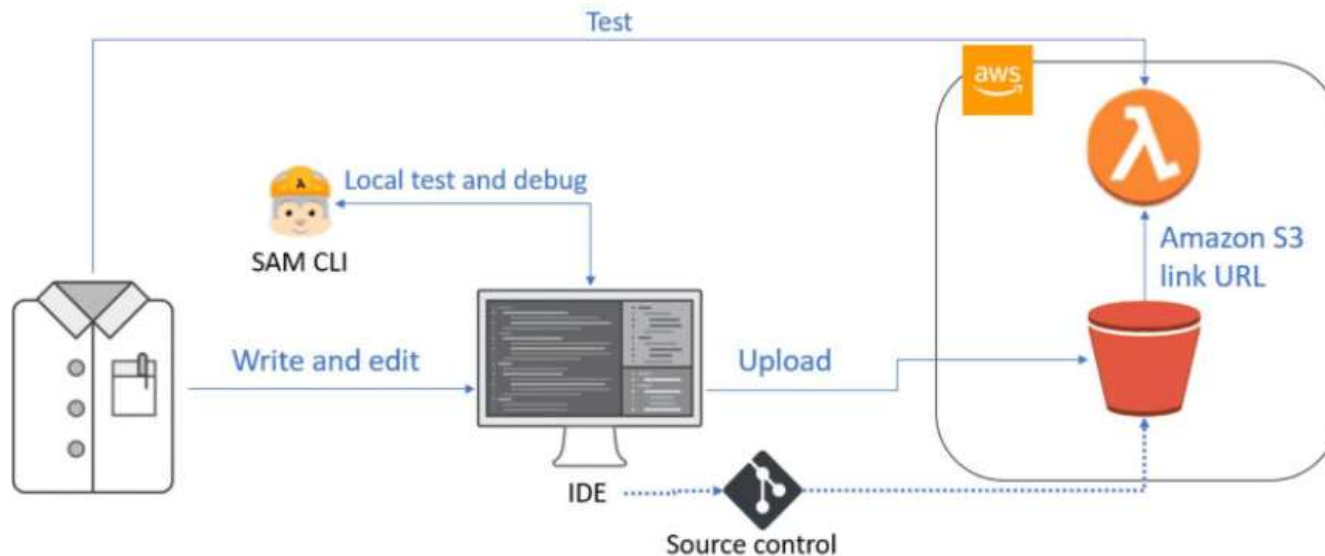
# Authoring Lambda Functions

## Authoring Lambda code

You can use any one of three methods to author Lambda Functions:

### Link to Amazon S3

Upload your deployment package to Amazon S3, and then specify the Amazon S3 URL for the object in the Lambda console.

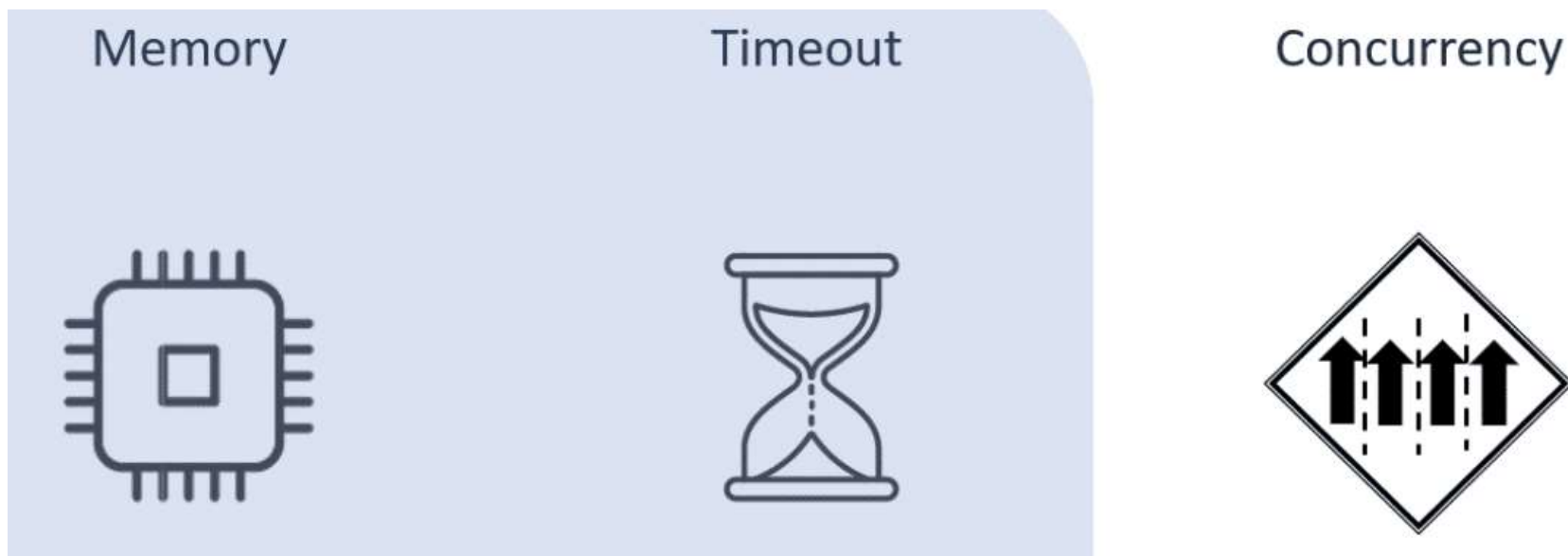


# Configuring Your Lambda Functions

## Memory and timeout

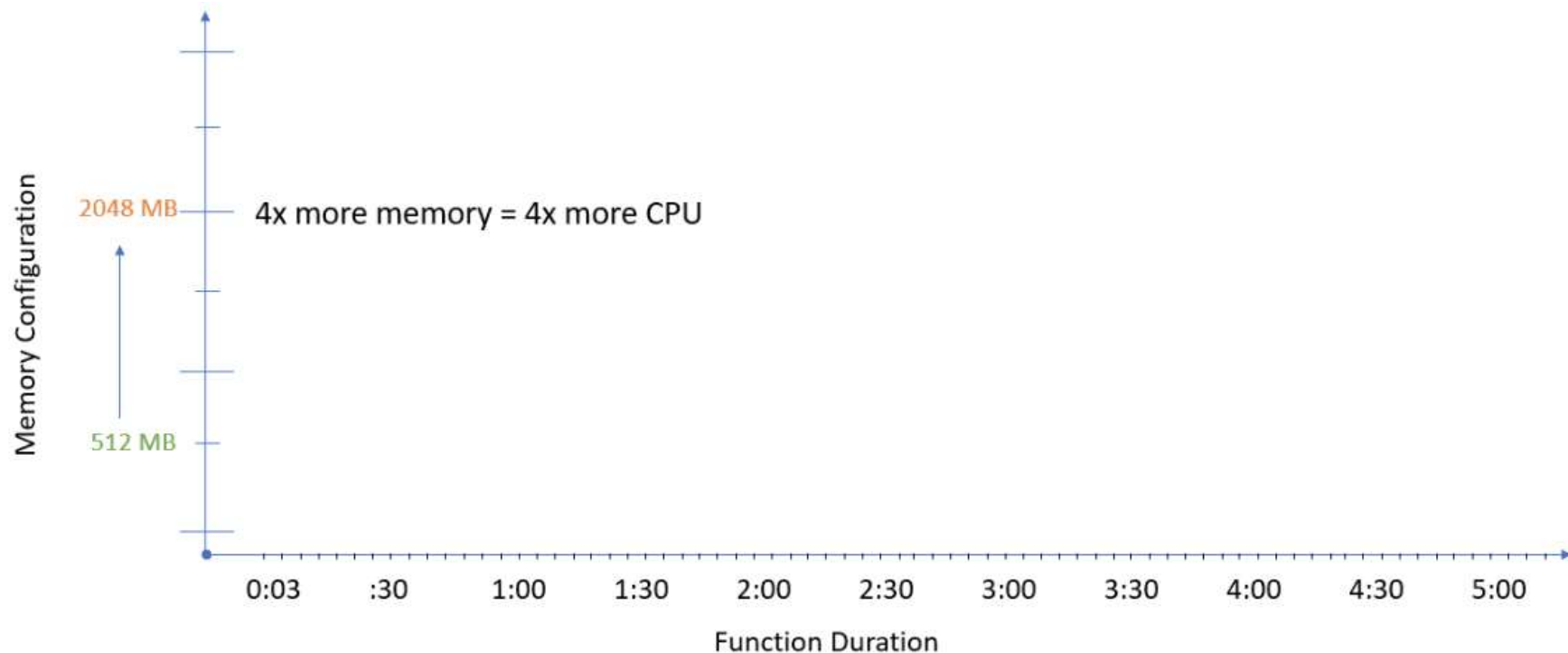
### Three core components to configuration

Memory, timeout, and concurrency are at the heart of how each function performs. To get the best combination for your desired customer experience, test in real-world conditions against peak volume and use monitoring options like Amazon CloudWatch and AWS X-Ray to ensure the desired results for your application.



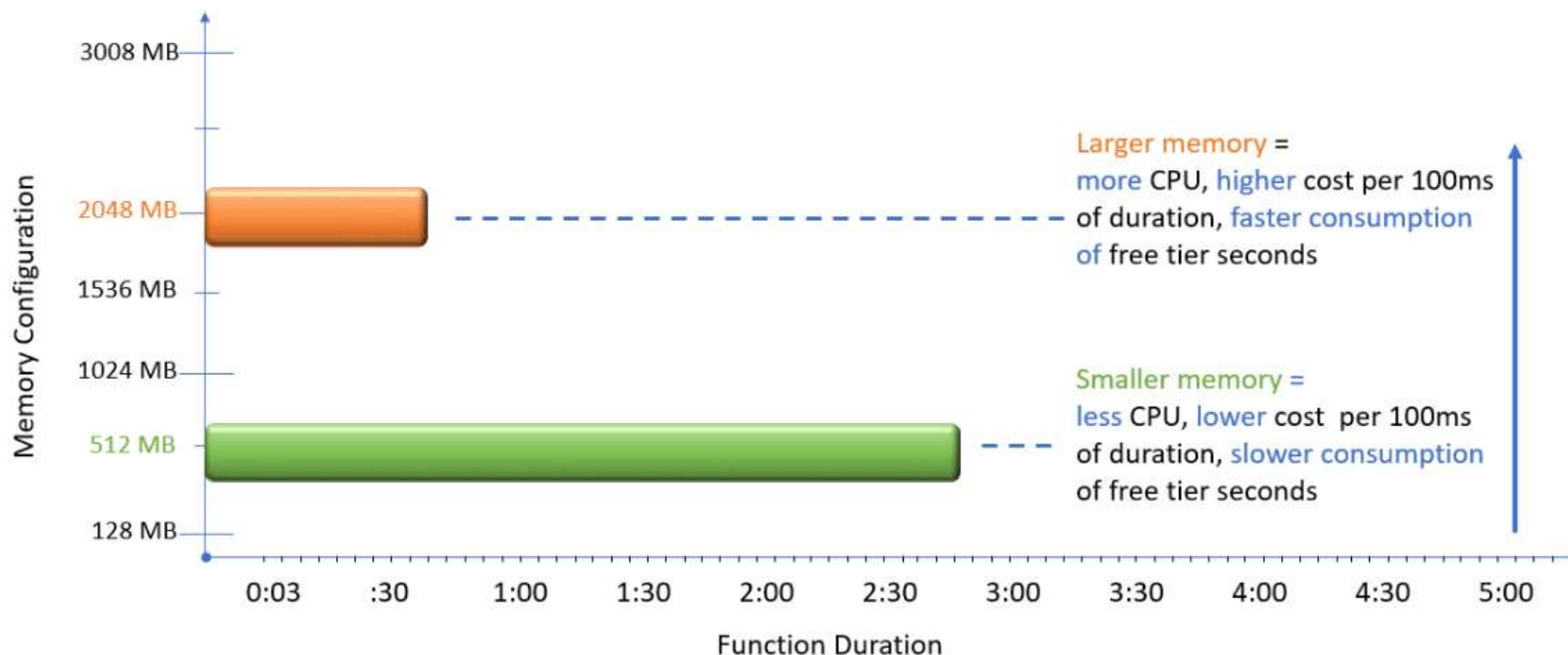
# Configuring Your Lambda Functions

You are billed based on memory + duration



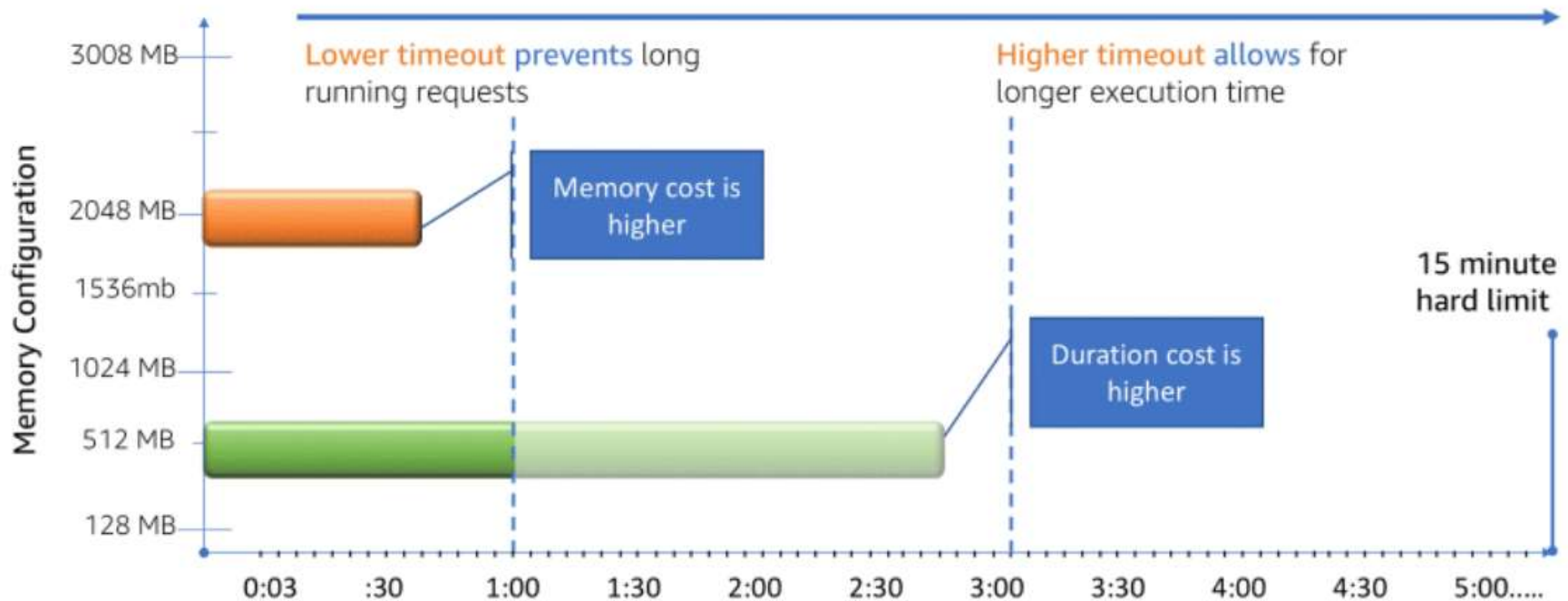
# Configuring Your Lambda Functions

You are billed based on memory + duration



# Configuring Your Lambda Functions

You are billed based on memory + duration



The timeout setting allows you to control the maximum duration for a function. This can prevent higher costs from long running functions. The trade off is finding the right balance between not running too long, but being able to finish under normal circumstances.

# Configuring Your Lambda Functions

## Find the sweet spot for power vs. duration

Depending on the function, you may find that the higher memory level might actually cost less because the function can complete much more quickly than at lower memory configuration.

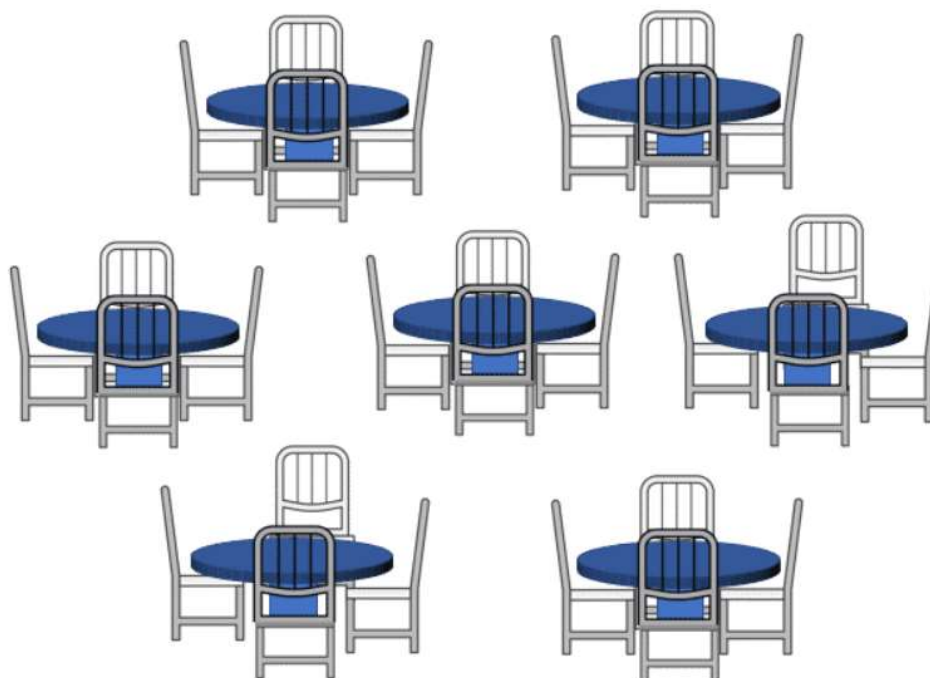
You can use an open-source tool called Lambda Power Tuning to find the best configuration for a function. The tool can help you visualize and fine-tune the memory/power configuration of Lambda functions. The tool runs in your own AWS account—powered by AWS Step Functions—and supports three optimization strategies: cost, speed, and balanced.

# Configuring Your Lambda Functions

## Concurrency and scaling

### Concurrent executions

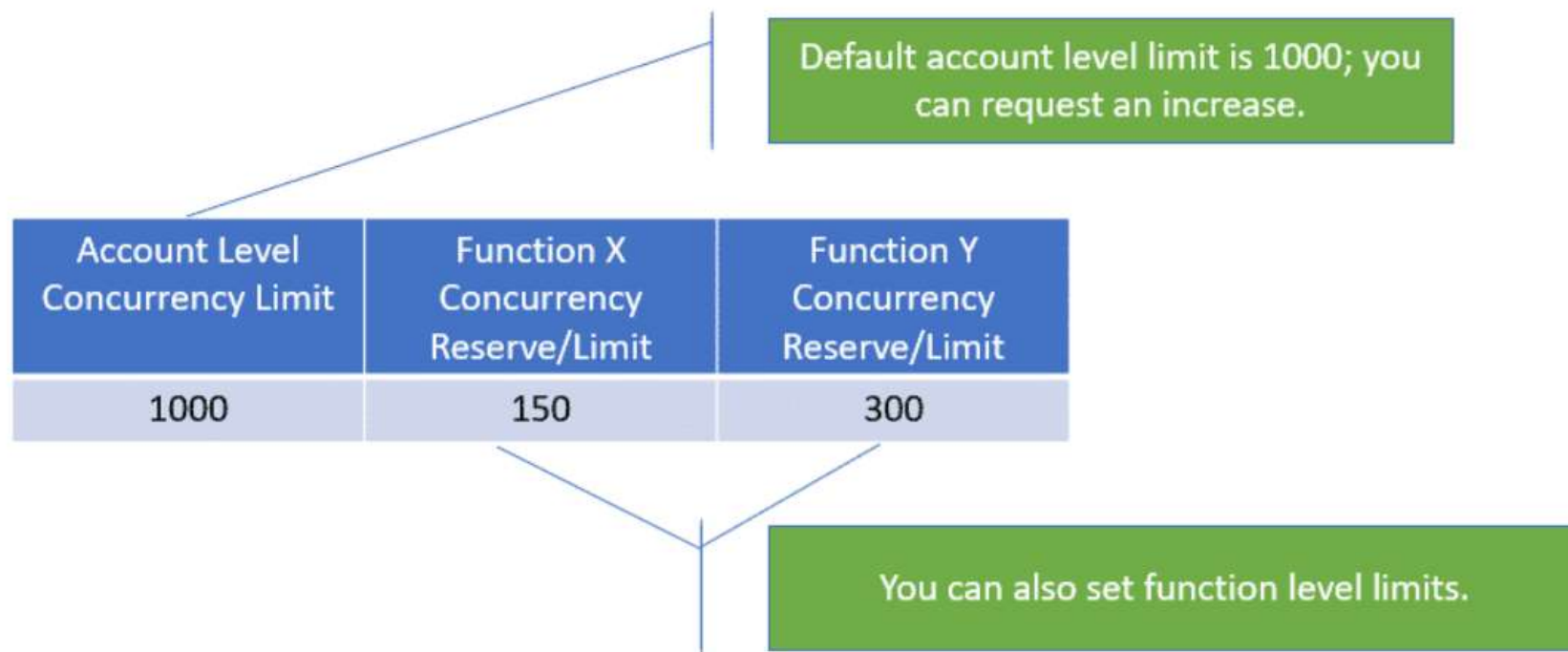
Concurrency is the unit at which scaling is measured. Think of concurrent executions as the capacity of a restaurant to serve a certain number of diners at a specific time.



# Configuring Your Lambda Functions

## Concurrency and scaling

### Concurrent executions



Account limits are set at 1000 by default, but you can request an increase via support ticket. You may also set specific concurrent execution limits (also called reserves) per function.




# Configuring Your Lambda Functions

## Concurrency and scaling

### Concurrent executions

Account Limit - Reserved Concurrency = Unreserved Concurrency



Account Level Concurrency Limit	Function X Concurrency Reserve/Limit	Function Y Concurrency Reserve/Limit	Unreserved Concurrency Limit
1000	150	300	550

The difference between the account limit and the sum of reserved currency is the Unreserved Concurrency.

# Configuring Your Lambda Functions

## Concurrency and scaling

### Concurrent executions

Account Limit - Reserved Concurrency = Unreserved Concurrency

Account Level Concurrency Limit	Function X Concurrency Reserve/Limit	Function Y Concurrency Reserve/Limit	Unreserved Concurrency Limit
1000	150	300	550

Functions with no reserved  
concurrency share the unreserved  
concurrency pool.

Functions without a reserved concurrency specified will all use the remaining available concurrency (the unreserved pool). In absence of any reserved concurrency, the entire pool is unreserved.

# Configuring Your Lambda Functions

## Concurrency and scaling

### Concurrent executions

Sum of all function reserves  $\leq$  (Account Currency Limit — 100)

Account Level Concurrency Limit	Function X Concurrency Reserve/Limit	Function Y Concurrency Reserve/Limit	Unreserved Concurrency Limit
1000			$\geq 100$

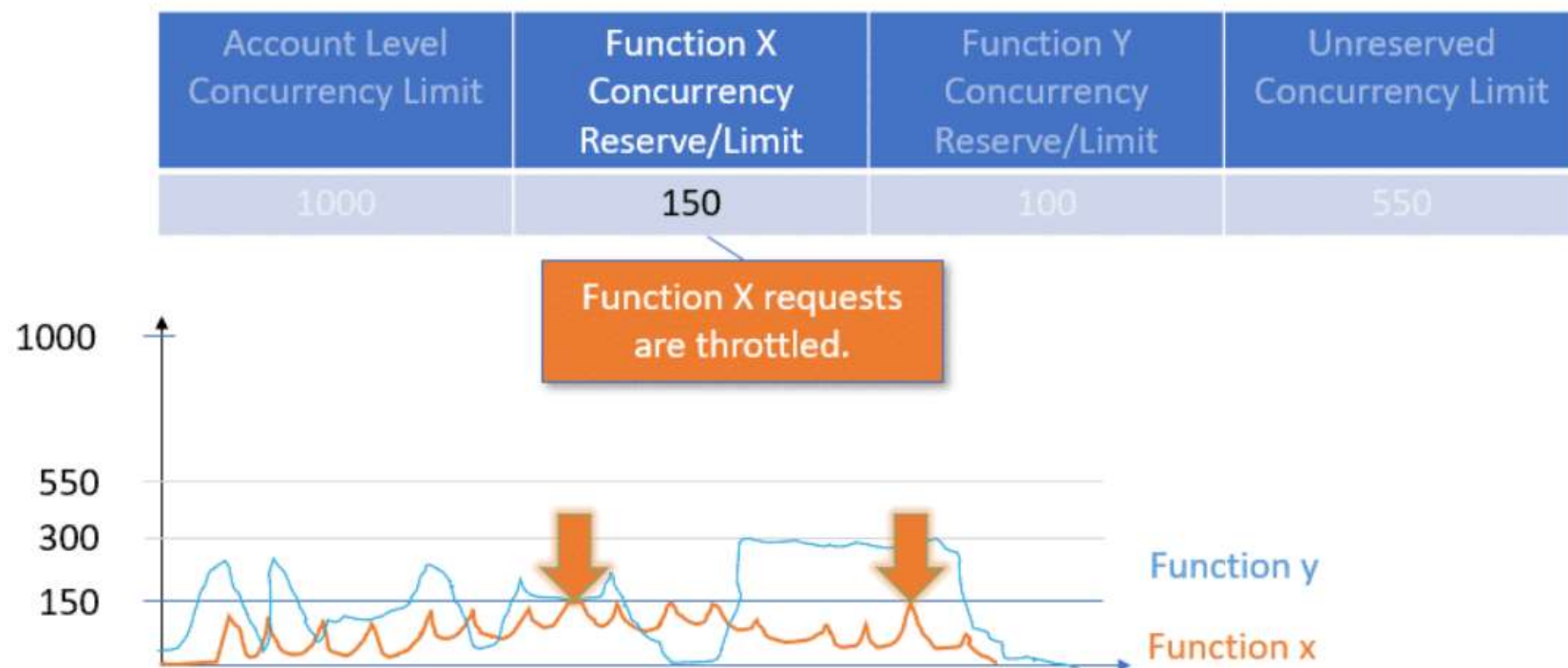
AWS Lambda will keep the Unreserved pool at a minimum of 100 to ensure that functions without reserve can be invoked.

AWS Lambda will keep the unreserved concurrency pool at a minimum of 100 concurrent executions, so that functions that do not have specific limits set can still process requests.

# Configuring Your Lambda Functions

## Concurrency and scaling

### Concurrent executions



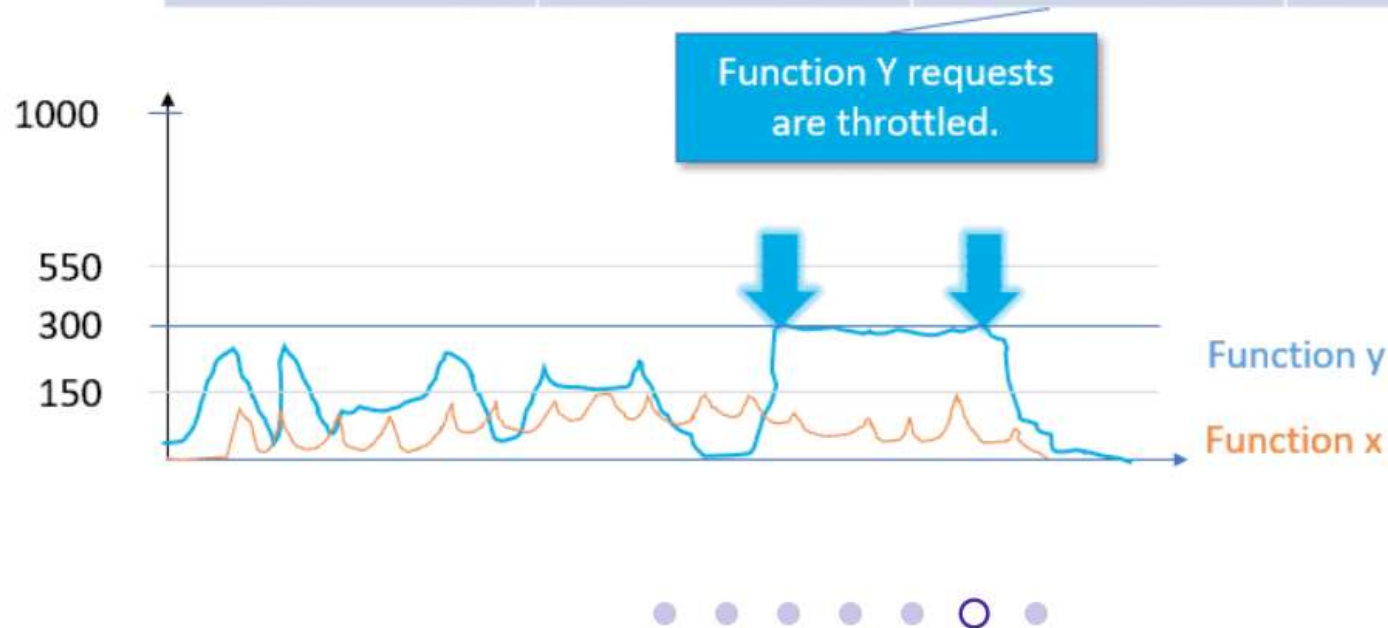
If function X has a reserve limit set at 150, requests for that function will get throttled whenever the concurrent requests **for that** function hit 150.

# Configuring Your Lambda Functions

## Concurrency and scaling

### Concurrent executions

Account Level Concurrency Limit	Function X Concurrency Reserve/Limit	Function Y Concurrency Reserve/Limit	Unreserved Concurrency Limit
1000	150	300	550



Function Y requests will be throttled only when there are 300 concurrent executions of that function. Neither function impacts the other's available concurrency.

# Configuring Your Lambda Functions

## Concurrency and scaling

### Concurrent executions

Account Level Concurrency Limit	Function X Concurrency Reserve/Limit	Function Y Concurrency Reserve/Limit	Unreserved Concurrency Limit
1000	150	300	550



In the unreserved pool, everyone else is competing for the remaining concurrency.

# Configuring Your Lambda Functions

**Why would you set concurrency limits?**

## **Limit Concurrency**

**You may want to limit a function's concurrency:**

- For cost reasons
- To regulate how long it takes you to process a batch of events
- To match it with a downstream resource



# Configuring Your Lambda Functions

**Why would you set concurrency limits?**

## **Reserve Concurrency**

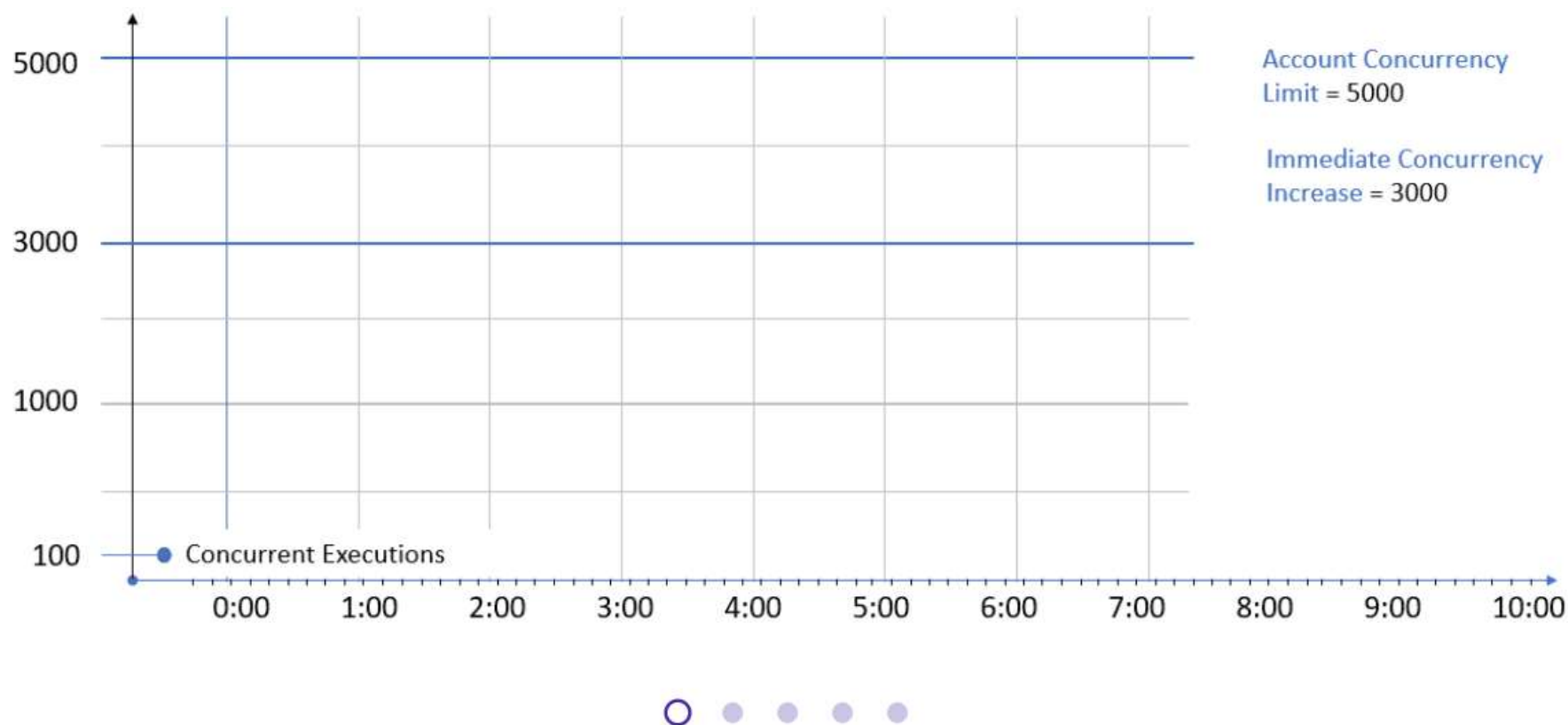
**You may want to reserve function concurrency:**

- To handle anticipated peaks
- To address invocation errors



# Configuring Your Lambda Functions

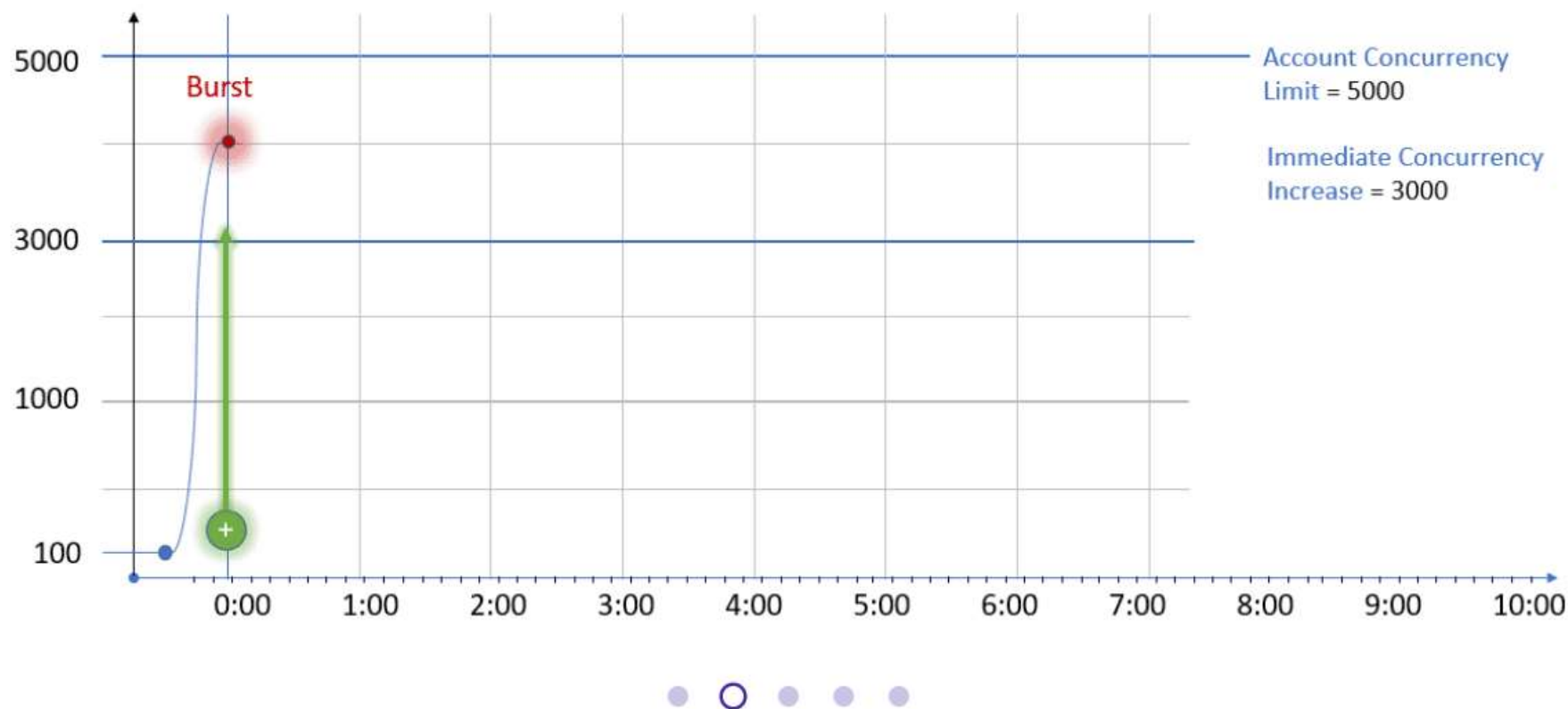
## How concurrency bursts are managed



Bursts are handled based on a combination of your limits, and a predetermined **Immediate Concurrency Increase** amount that is dependent on the region where your Lambda function is running. In this example the account limit is set for 5,000 concurrent executions in a region with an immediate concurrency increase of 3,000.

# Configuring Your Lambda Functions

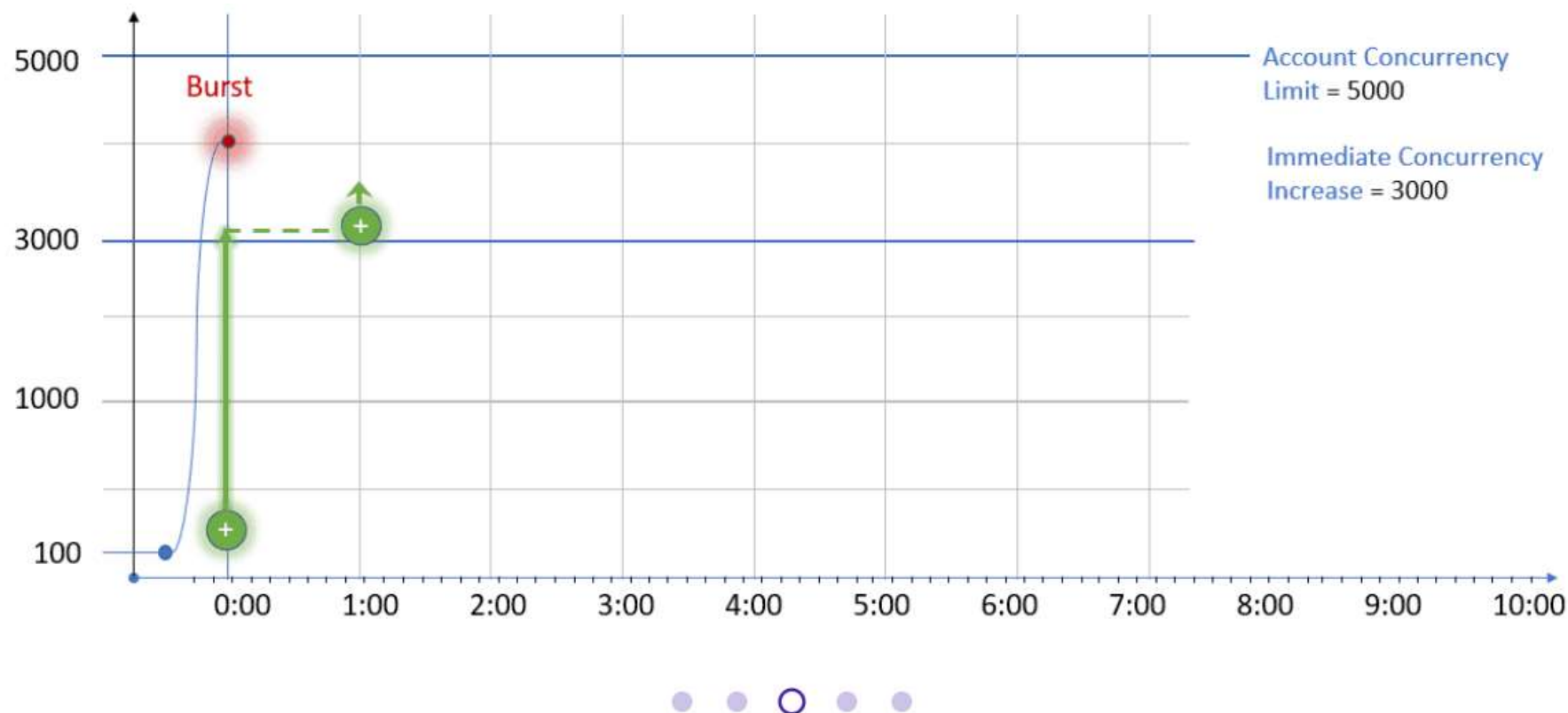
## How concurrency bursts are managed



In this example, if the concurrent executions burst from 100 to 4,000, Lambda will respond by adding the Immediate Concurrency Increase of 3,000 to the 100 concurrent executions that were running prior to the burst.

# Configuring Your Lambda Functions

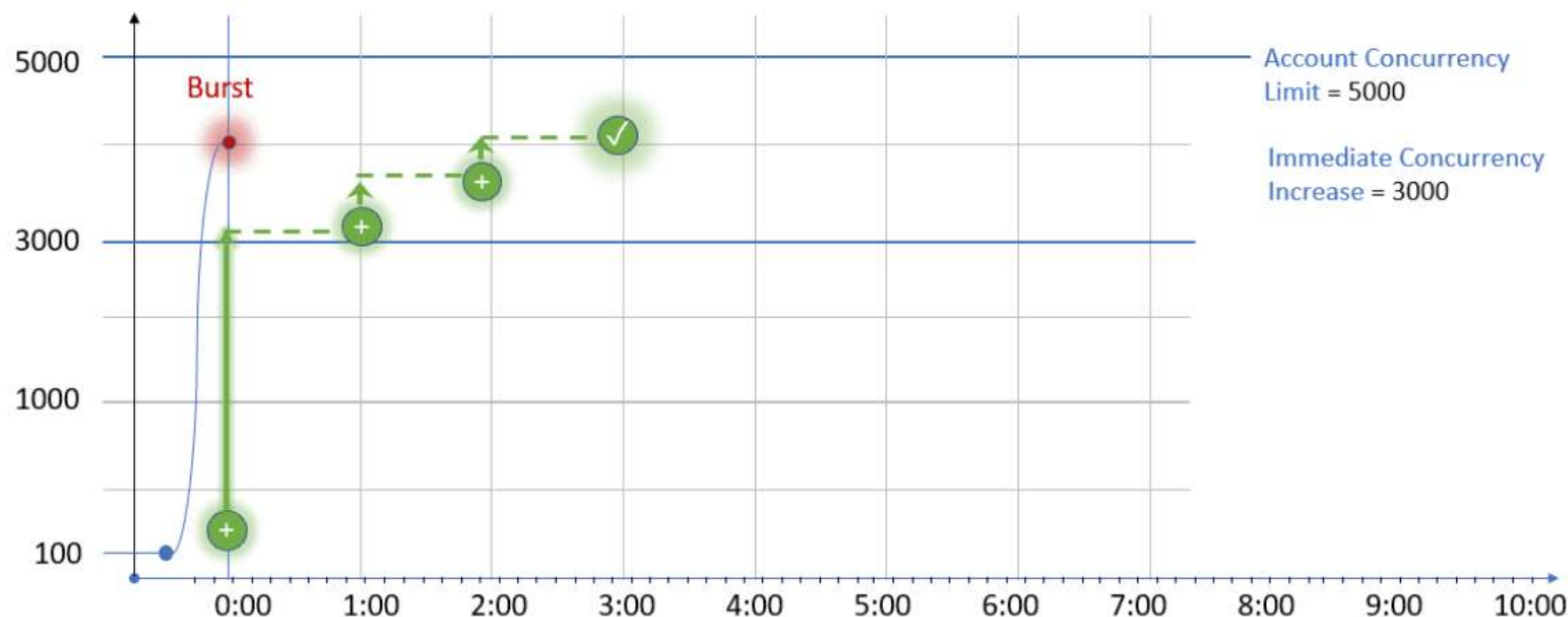
## How concurrency bursts are managed



After one minute, Lambda would evaluate whether more were needed and add 500 additional concurrent executions if neither the account limit nor the burst level has been reached. In this example 500 more concurrent executions would be added.

# Configuring Your Lambda Functions

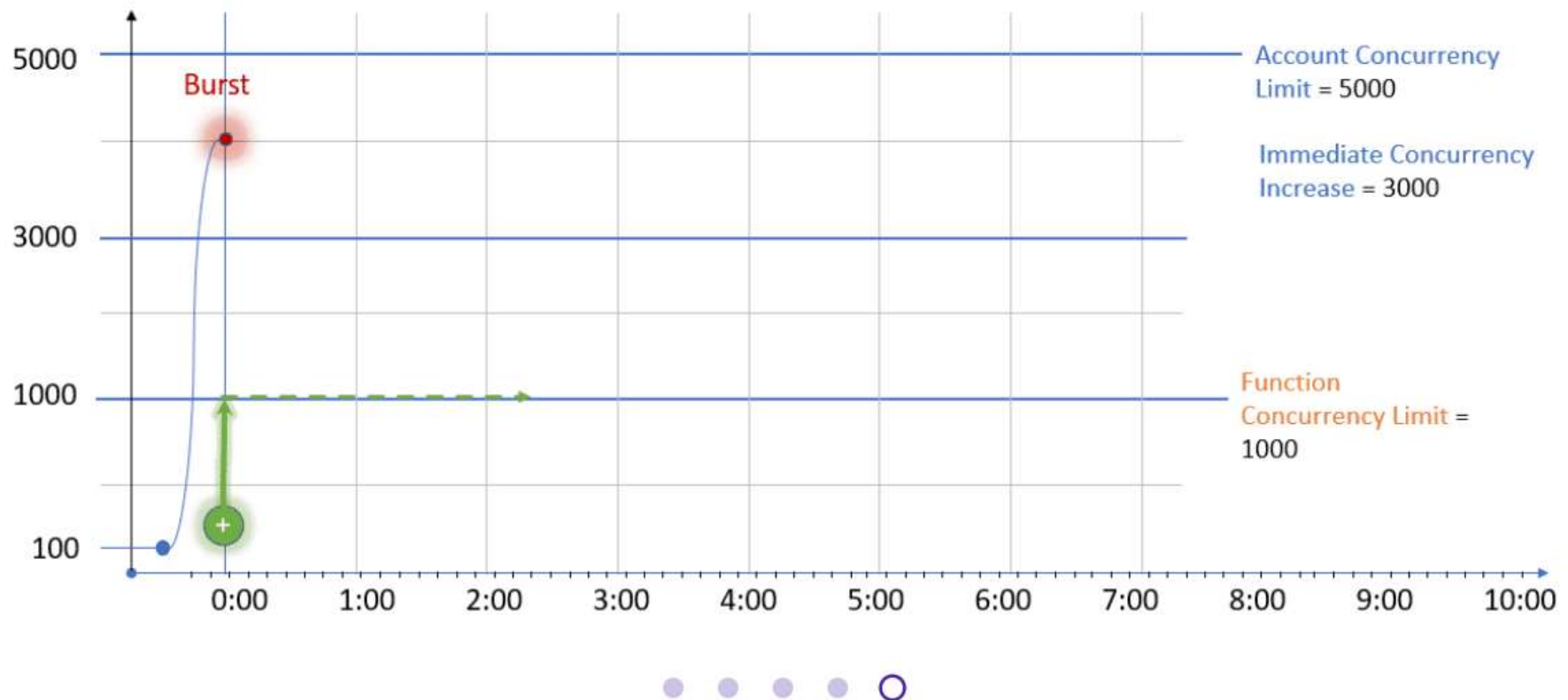
## How concurrency bursts are managed



After a second minute another 500 concurrent executions would be added. After another minute Lambda would evaluate that there are enough concurrent executions available to handle the burst and the evaluate/add cycle ends.

# Configuring Your Lambda Functions

## How concurrency bursts are managed



If we look at the same burst for a function that has a concurrency limit set for 1,000, Lambda would immediately increase the concurrent executions to 1,000, and having hit the function limit would start throttling invocations to keep the concurrent executions no higher than 1,000.

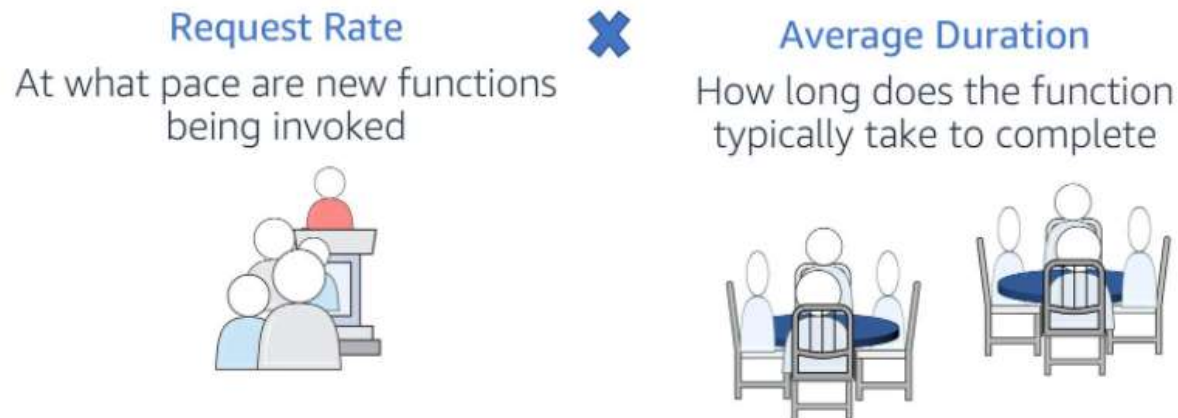
# Configuring Your Lambda Functions

## Estimating concurrency

To properly plan for where to set your concurrency limits, you need to understand how the concurrent executions are counted, and be able to measure the concurrent executions in a realistic test.

## Non-polling events

For non-stream based events you can estimate the concurrent executions by using the request rate - which in this case is simply the rate at which events are published multiplied by the average duration of the function.



10 invocations per second x 3 second duration = 30 Concurrent Executions

# Configuring Your Lambda Functions

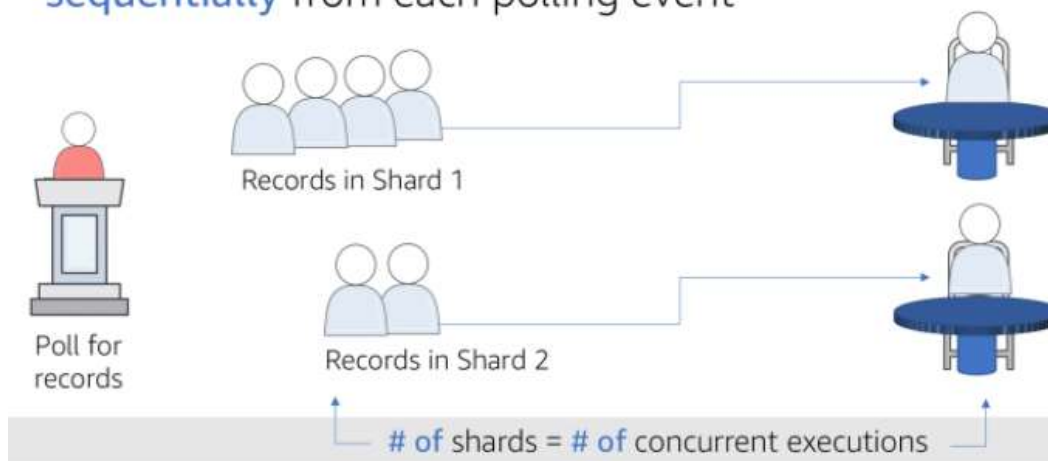
## Estimating concurrency

### Stream polling events

Tune polling frequency to request rate to “right-size” processing. When Lambda polls the stream it pulls back the records currently in each shard and invokes one concurrent Lambda function per shard to process records from that shard in sequential order until all of the records from that shard from that polling event are processed successfully (or expire).

Request rate = concurrent executions (# of shards)/average function duration

There is **one concurrent execution per shard** processing records **sequentially** from each polling event





# Configuring Your Lambda Functions

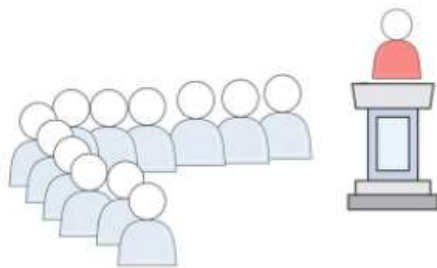
## Estimating concurrency

### Non-stream polling events (SQS)

With SQS, there is a cost every time you poll, so the Lambda integration with SQS is designed to find the balance between cost and performance.

Lambda will poll at an initial rate of 5 per second, and if we see the queue depth continue to increase, automatically scales up polling activity until the number of concurrent function executions reaches 1000, the account concurrency limit, or the (optional) function concurrency limit, whichever is lower.

Polling **scales** based on **queue**



- 5 polls per second to start
- **Automatically scale** polling based on queue depth
- Scale until **concurrent execution limit** is reached
- Initial **burst of 5 invocations** and then **60 added per minute** to reach the limit



# Configuring Your Lambda Functions

## CloudWatch Metrics for concurrency

### **ConcurrentExecutions**

Sum of concurrent executions for a given function at a given point in time

All functions in the account

Functions that have a custom concurrency limit specified

### **UnreservedConcurrentExecutions:**

Sum of the concurrency of the functions that do not have a custom concurrency limit specified.

# Configuring Your Lambda Functions

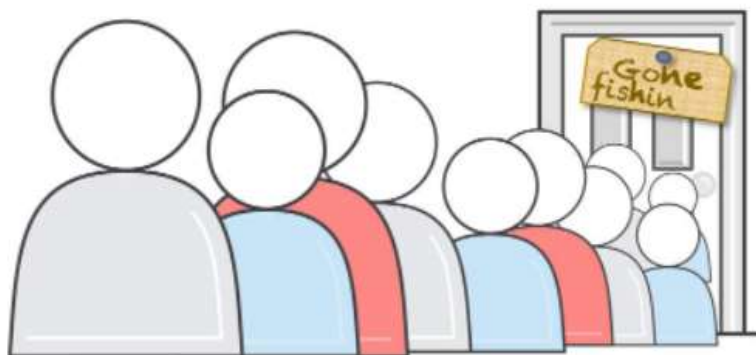
## Provisioned concurrency

You can configure part of your available concurrency limit as provisioned concurrency. This lets you ensure that function invocations are executed in initialized ("warm") environments. For more information, visit the [What's New blog for December 2019](#) and [Configuring Provisioned Concurrency in the AWS Lambda Developer Guide](#).

# Configuring Your Lambda Functions

## Testing concurrency

Probably the most important thing for deciding how to set your concurrency, and your memory and timeout for that matter, is to make sure you are testing in real world conditions.

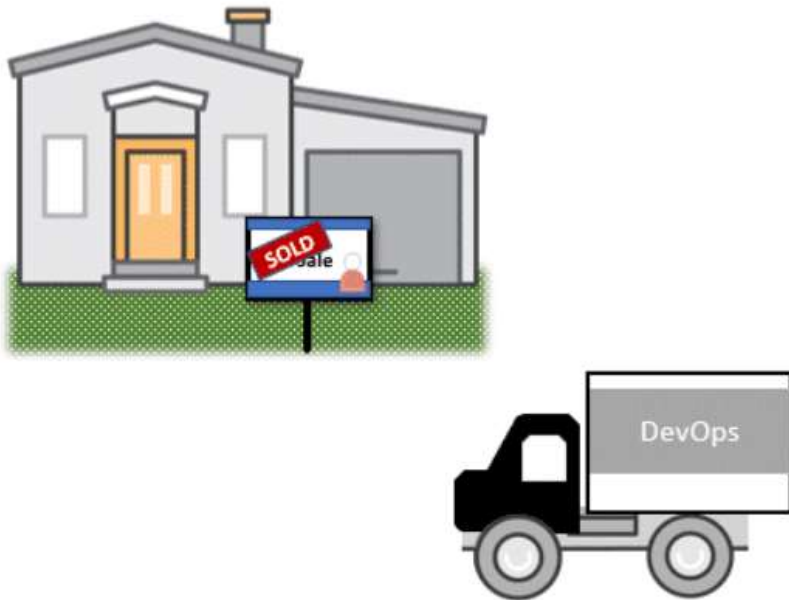


- Do your **performance tests** account for **peak** + ?
- Can your existing **backend** handle the **speed of requests** you will send it?
- Does your **error handling** work as expected?

# Deploying and Testing Serverless Applications

How is serverless deployment different?

Server-based vs. serverless deployment



**Server-based deployment is like moving into a house.**

You need to know what the infrastructure supports and how you and your family will fit into it, but you're working with what's there. Pack your stuff and hand off to the movers. Kind of like checking in your code, after which DevOps puts your code into a build and deploys it.

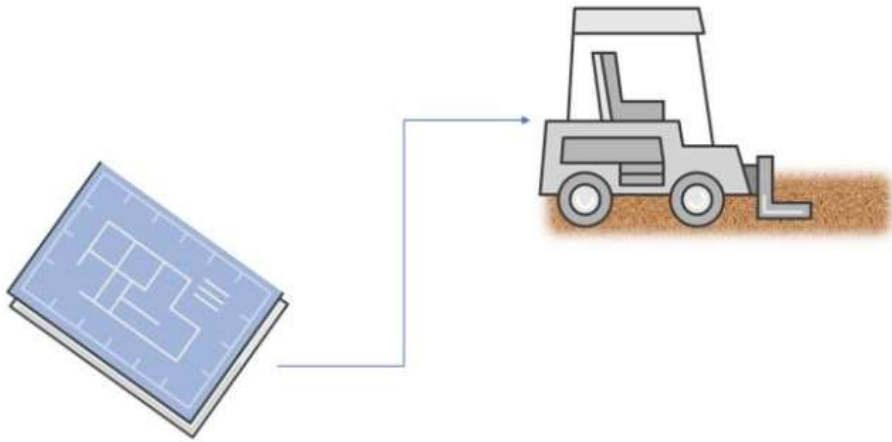
# Deploying and Testing Serverless Applications

How is serverless deployment different?

Server-based vs. serverless deployment

**Serverless is more like building a house.**

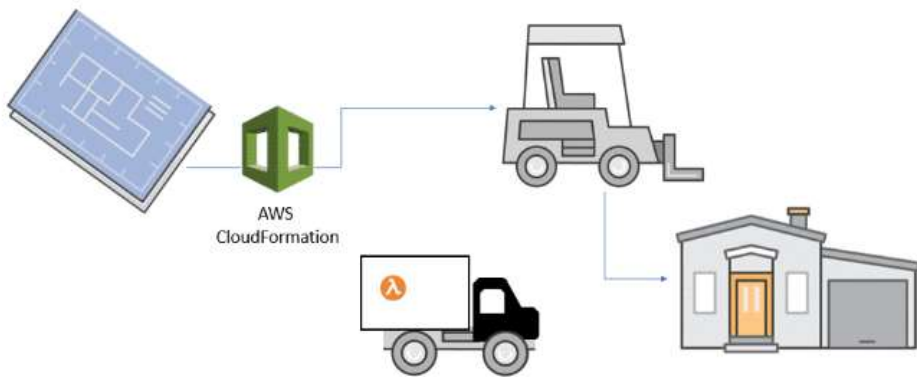
You are designing the entire structure to very detailed specifications. You convey this via a blueprint. The blue print allows you to know exactly what the home is going to look like before its been built.



# Deploying and Testing Serverless Applications

How is serverless deployment different?

## Serverless deployment



When you deploy Lambda functions, this blueprint is kind of like an **AWS CloudFormation** template.

You model your infrastructure in a CloudFormation template and use that to deploy your desired stack without writing custom scripts.

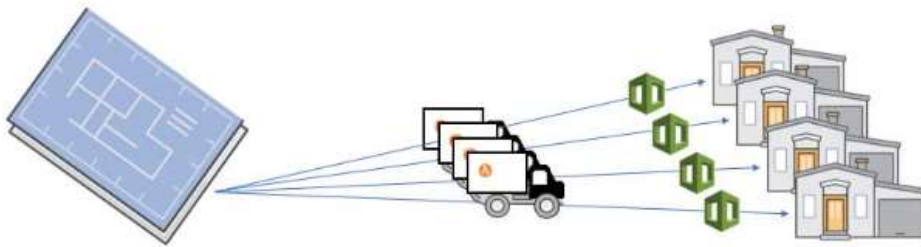
# Deploying and Testing Serverless Applications

How is serverless deployment different?

## Serverless deployment

The template is then the single source of truth for deploying that stack into any account.

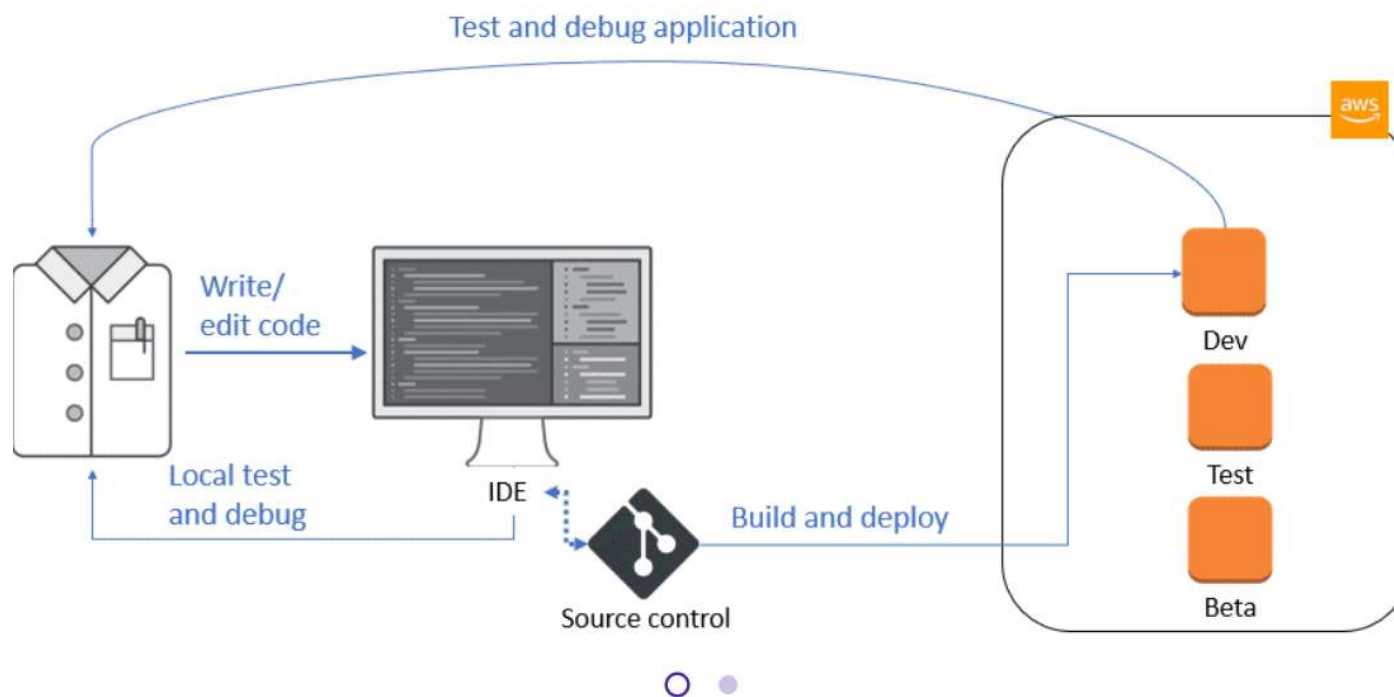
And each time the function is invoked, it's going to execute using information from the CloudFormation template.



# Deploying and Testing Serverless Applications

## Server-based vs. serverless development environments

### Server-based development environment



#### Development environment for a server-based application

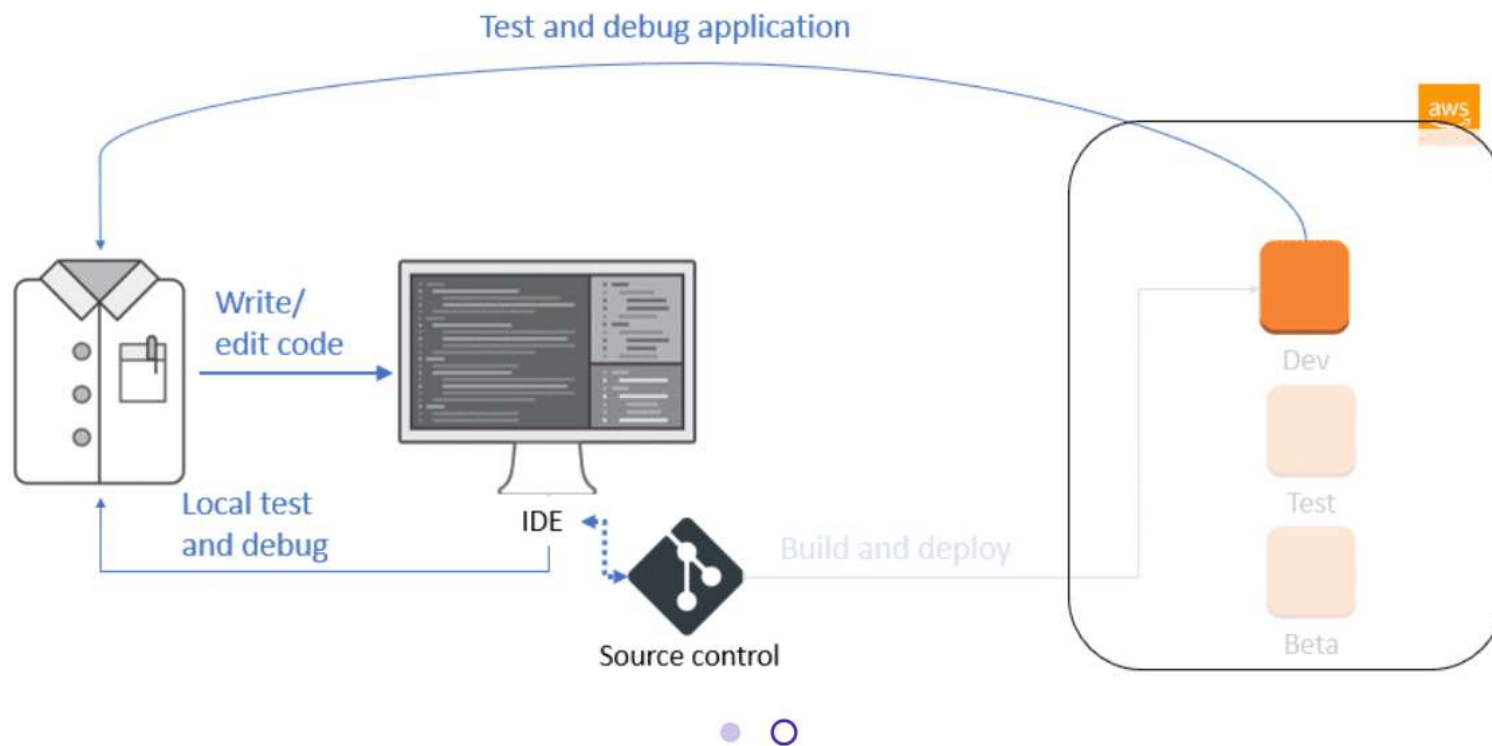
This workflow typically means pulling down a local copy of the application, working locally via your IDE to code, test, then debug your code, and then checking those changes into source control.



# Deploying and Testing Serverless Applications

## Server-based vs. serverless development environments

### Server-based development environment



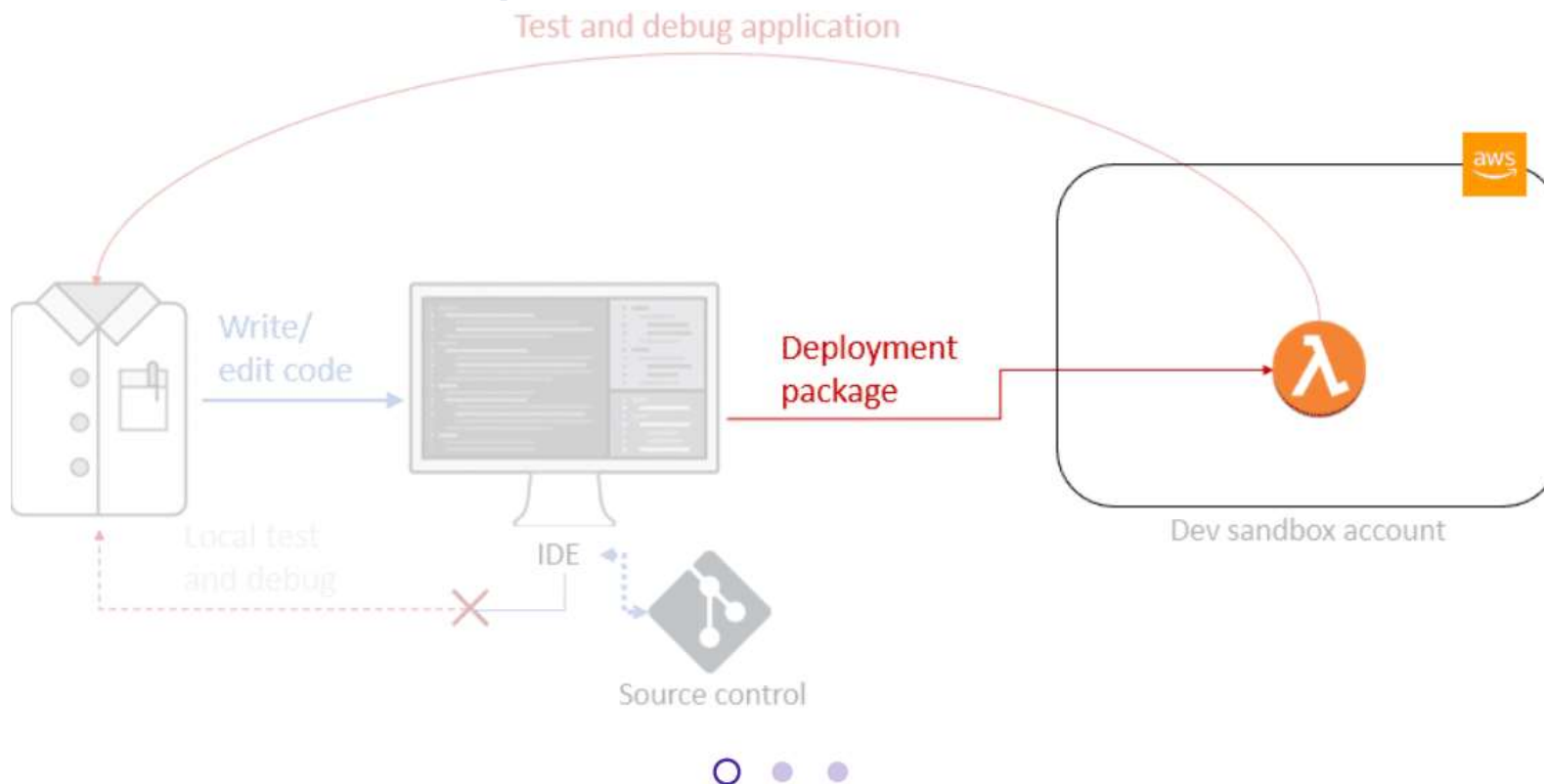
#### Development environment for a server-based application

A DevOps team member validates the build and deploys the updated application to designated instances. Developers have access to defined test instances to perform application/ integration testing & debugging.

# Deploying and Testing Serverless Applications

## Server-based vs. serverless development environments

### Serverless development environment



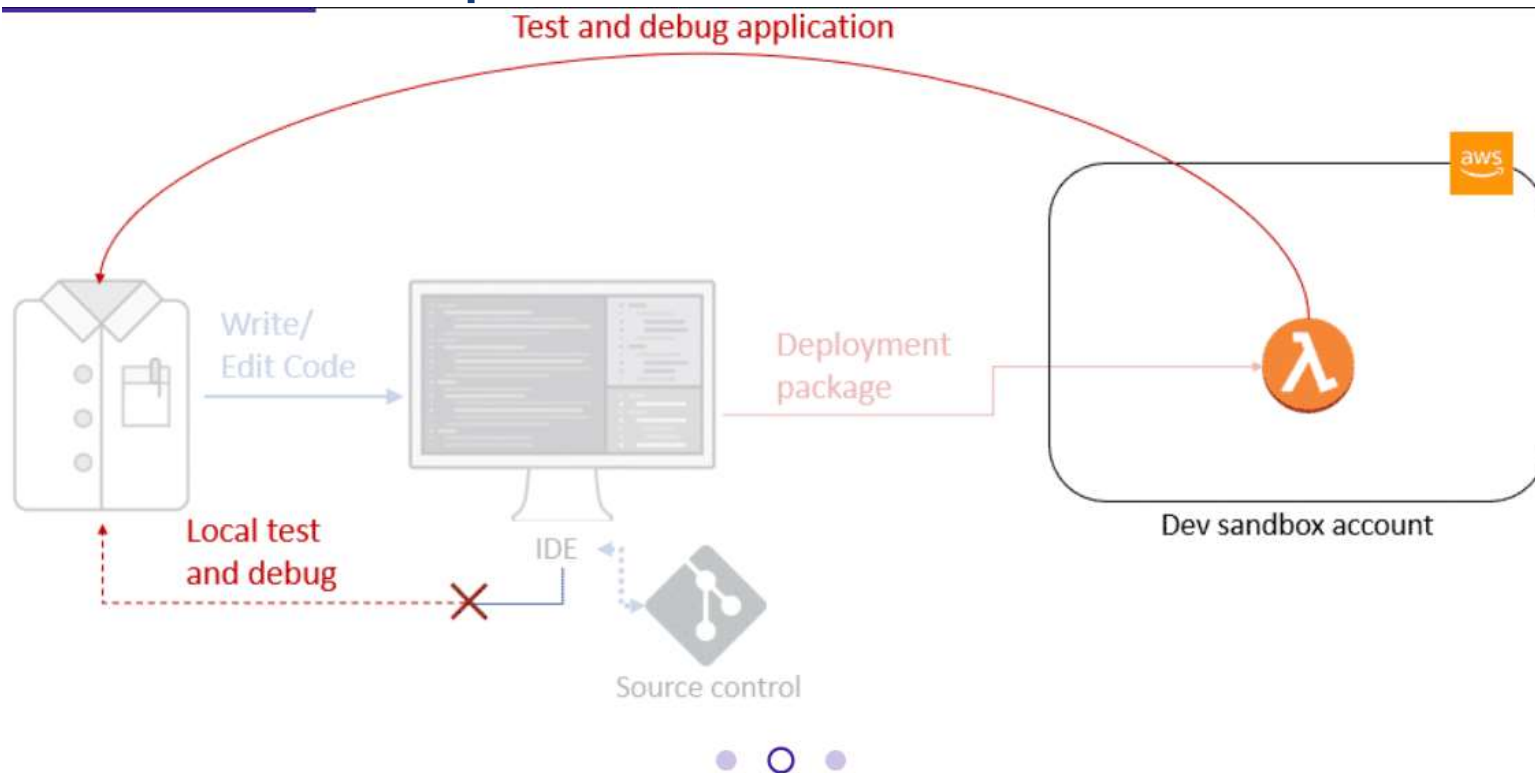
#### Development in a serverless environment

With serverless applications the developer must provide everything needed to deploy a function - the code, bundled with any necessary dependencies AND that blue print for setting up the infrastructure to be deployed to an AWS account.

# Deploying and Testing Serverless Applications

## Server-based vs. serverless development environments

### Serverless development environment



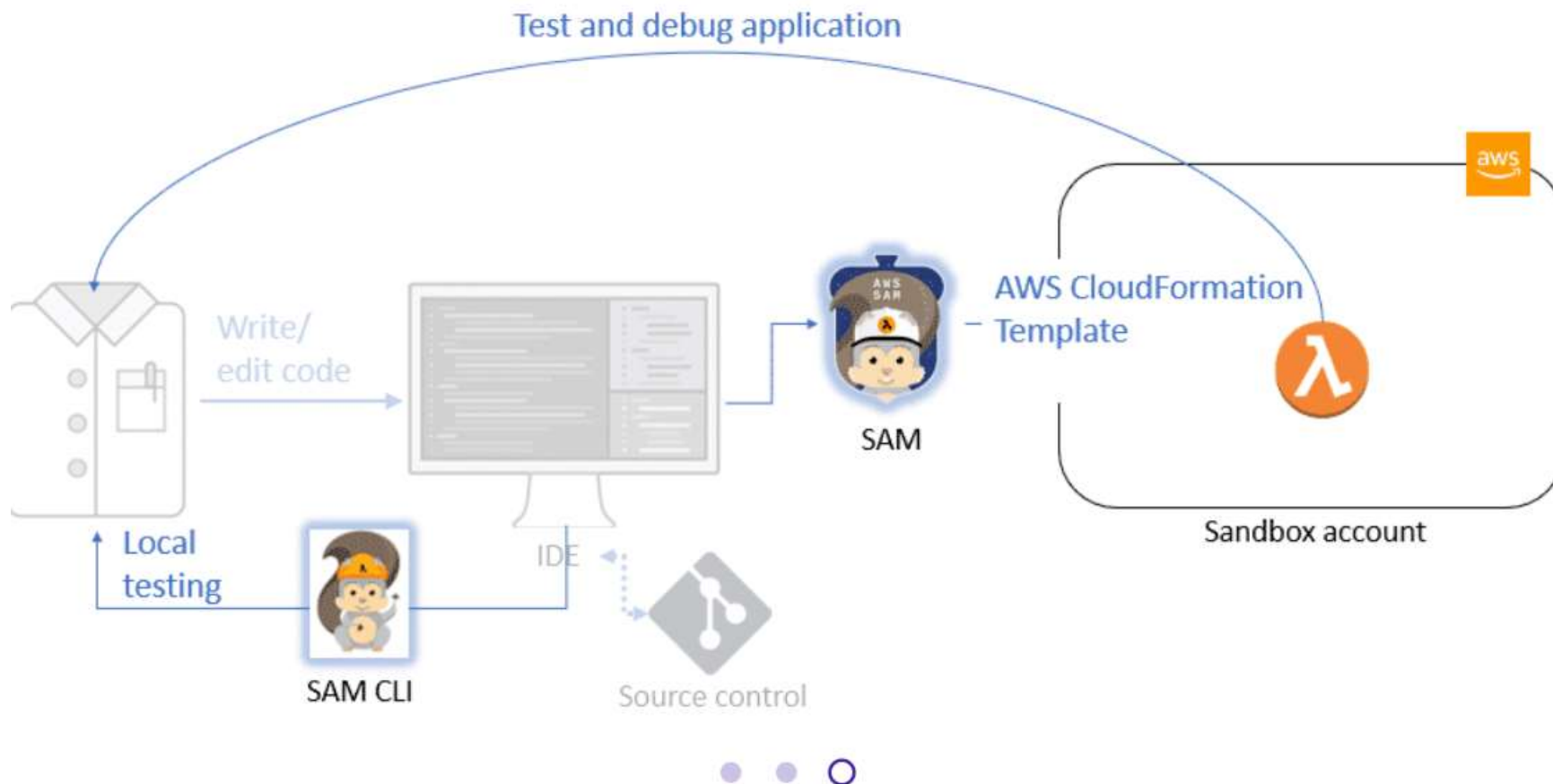
#### Development in a serverless environment

You can't create the environment specified in your 'blueprint' locally. You aren't connecting to a specific server from which you debug code. You need the ability to deploy your stack to each AWS account.

# Deploying and Testing Serverless Applications

## Server-based vs. serverless development environments

### Serverless development environment

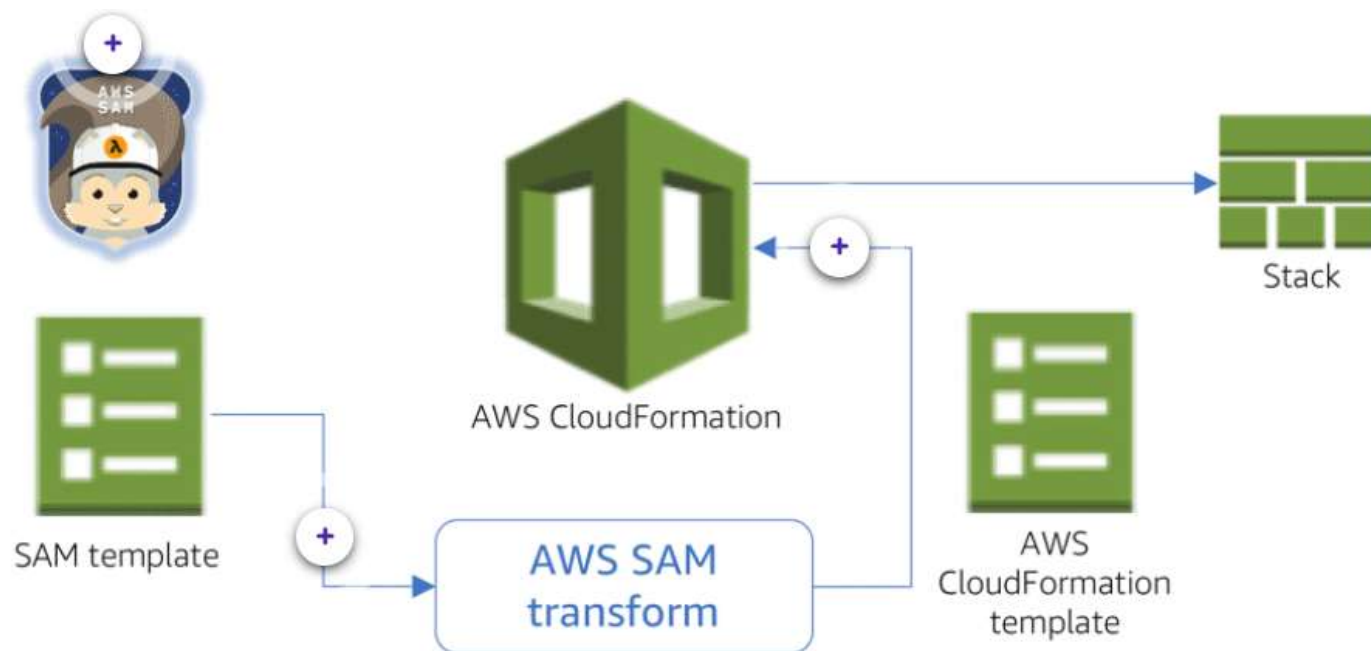


This is where an application framework like the Serverless Application Model (SAM) comes in to play.

# Deploying and Testing Serverless Applications

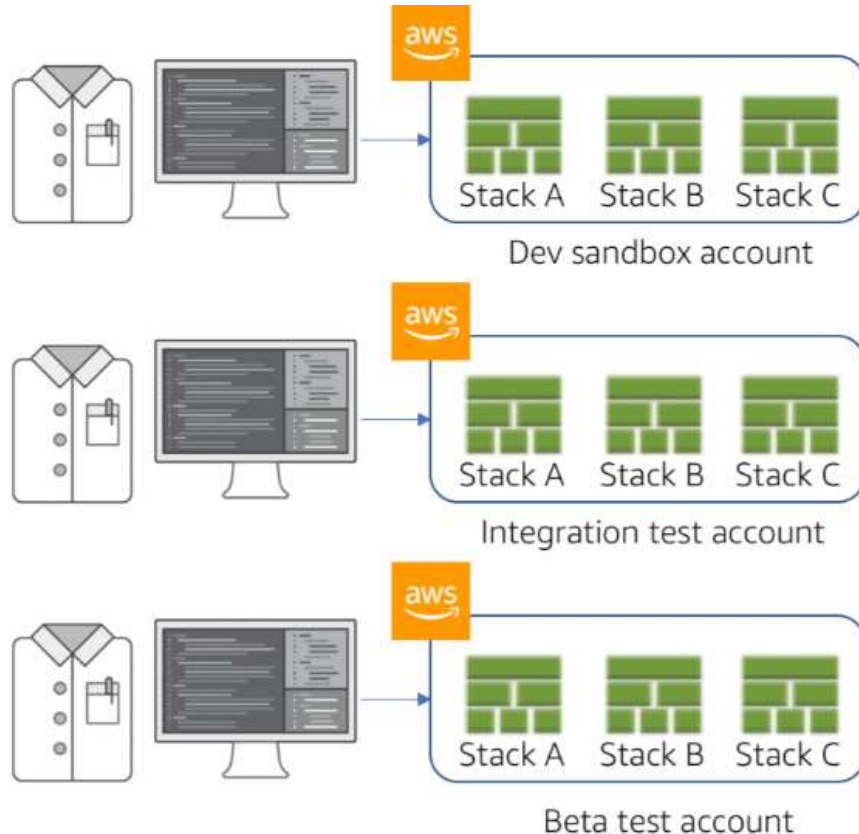
## Using SAM as your serverless application framework

There are a number of serverless frameworks available. Find a framework that works with your developer tool chain and minimizes the work to specify the details of your execution environment. For our examples we'll look at SAM.



# Deploying and Testing Serverless Applications

## Use SAM to deploy a stack to multiple accounts



## Ensure environmental parity

SAM makes it simple to create a stack, and deploy the exact same stack to each account.

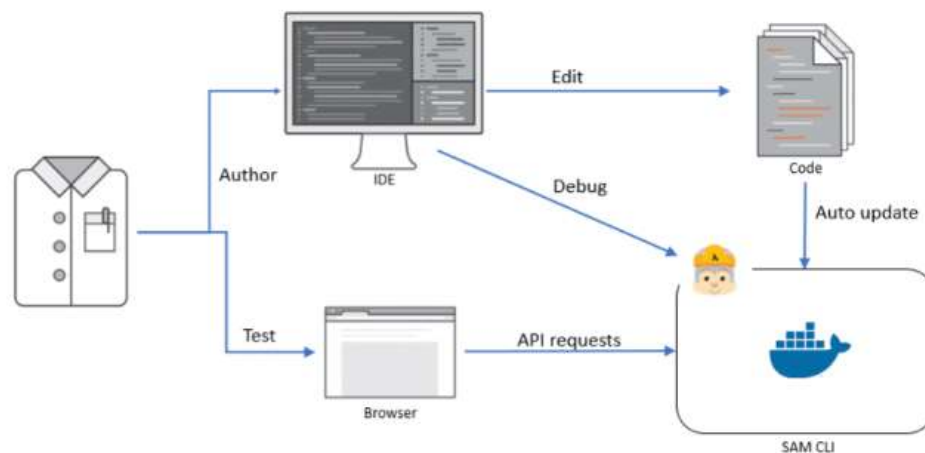
## Simplify experimentation

Without the overhead of maintaining instances, SAM makes it possible to quickly spin up stacks for different feature branches and allow for experimentation without incurring costs outside of the actual invocations executed on that environment.

# Deploying and Testing Serverless Applications

## SAM CLI helps you test and deploy

SAM CLI launches a Docker container that you can interact with to test and debug your Lambda Functions. It is important to note that even with a tool like SAM CLI, local testing will only cover a subset of what needs to be tested before code should go into your production application.



## SAM CLI for testing

- Invoke functions and run automated tests locally.
- Generate sample event source payloads.
- Run API Gateway locally.
- Debug code.
- Review Lambda function logs.
- Validate SAM templates.



# Deploying and Testing Serverless Applications

## SAM CLI

You can install the SAM CLI locally to assist in testing your serverless applications, validating your SAM templates, and streamlining your deployments.

**Init** - Initialize a serverless application.

**Local** - Run your application locally.

**Validate** - Validate an AWS SAM template.

**package (implicitly performed with deploy)** - Package an AWS application. Alias for aws cloudformation package command.

As of November, 2019, the SAM deploy command now implicitly performs the functionality of the package command. You can use the SAM deploy command directly to package and deploy your application.



# Deploying and Testing Serverless Applications

## SAM CLI

You can install the SAM CLI locally to assist in testing your serverless applications, validating your SAM templates, and streamlining your deployments.

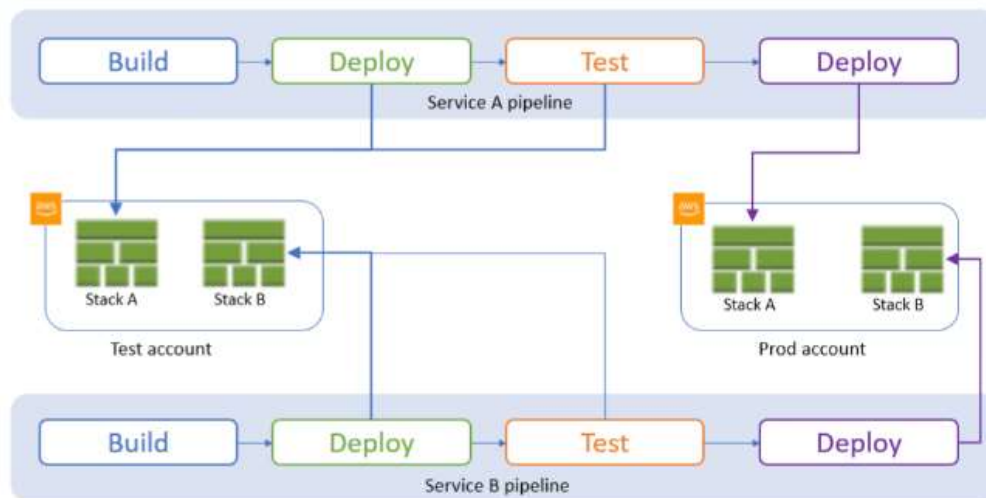
**Deploy** - Deploy an AWS SAM application.

This command comes with a guided interactive mode, which you can enable by specifying the `--guided` parameter. The interactive mode walks you through the parameters required for deployment, provides default options, and saves these options in a configuration file in your project folder. You can execute subsequent deployments of your application by executing the SAM deploy command. The needed parameters will be retrieved from the AWS SAM CLI configuration file.

Deploying Lambda functions through AWS CloudFormation requires an Amazon S3 bucket for the Lambda deployment package. The SAM CLI now creates and manages this Amazon S3 bucket for you.

# Deploying and Testing Serverless Applications

## Serverless CI/CD pipeline



You can incorporate additional tools to create an automated CI/CD pipeline for your serverless applications that is integrated with SAM. for example:

- **CodeBuild:** automate the process of packaging code and executing tests before the code is deployed
- **CodeDeploy:** use version management options to ensure safe deployments to production.

# Deploying and Testing Serverless Applications

## Serverless Application Repository

Applications packaged with SAM templates can be shared on the Serverless Application Repository.

- Publish your own applications to share across your AWS accounts or with the public.
- Publicly shared applications include a link to the application's source code.
- Verified author badges let consumers easily find authors they are familiar with.

# Deploying and Testing Serverless Applications

## Versioning and aliases for safe deployments

When you create a Lambda function, there is only one version, identified by \$LATEST in the ARN. Reference the \$Latest version of the function using its Amazon Resource Name (ARN). Add additional versions using the Publish action.

When you create a Lambda function, there is only one version, identified by \$LATEST in the ARN.

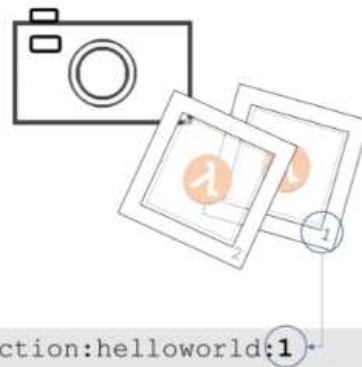
- Reference the \$Latest version of the function using its Amazon Resource Name (ARN).
- Add additional versions using the Publish action.

# Deploying and Testing Serverless Applications

## Publish makes a snapshot copy of \$Latest

Enable versioning to create immutable snapshots of your function every time you publish it.

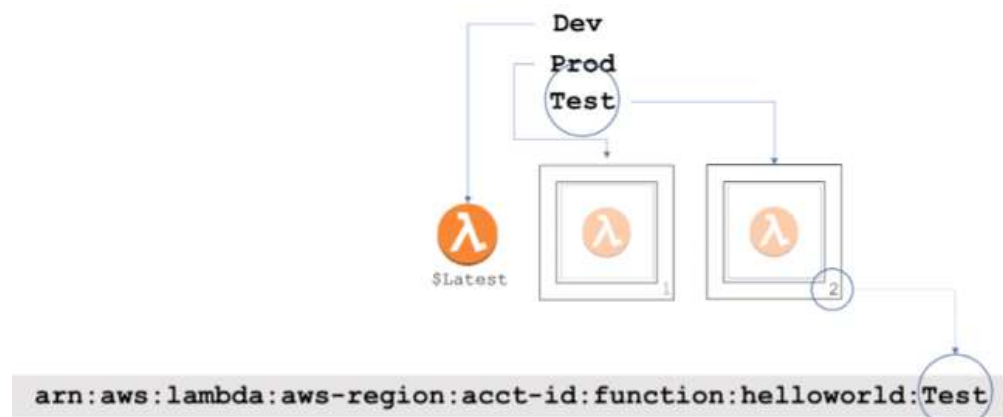
- Publish as many versions as you need.
- Each version results in a new sequential version number.
- Add the version number to the function ARN to reference it.



# Deploying and Testing Serverless Applications

## Aliases point to specific versions

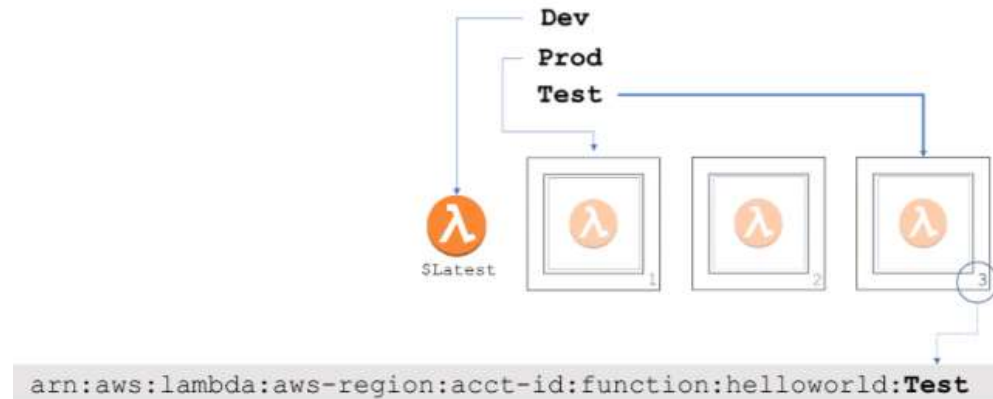
You can use aliases to point to specific versions and then use the alias in the ARN to reference the Lambda function version currently associated with that alias.



# Deploying and Testing Serverless Applications

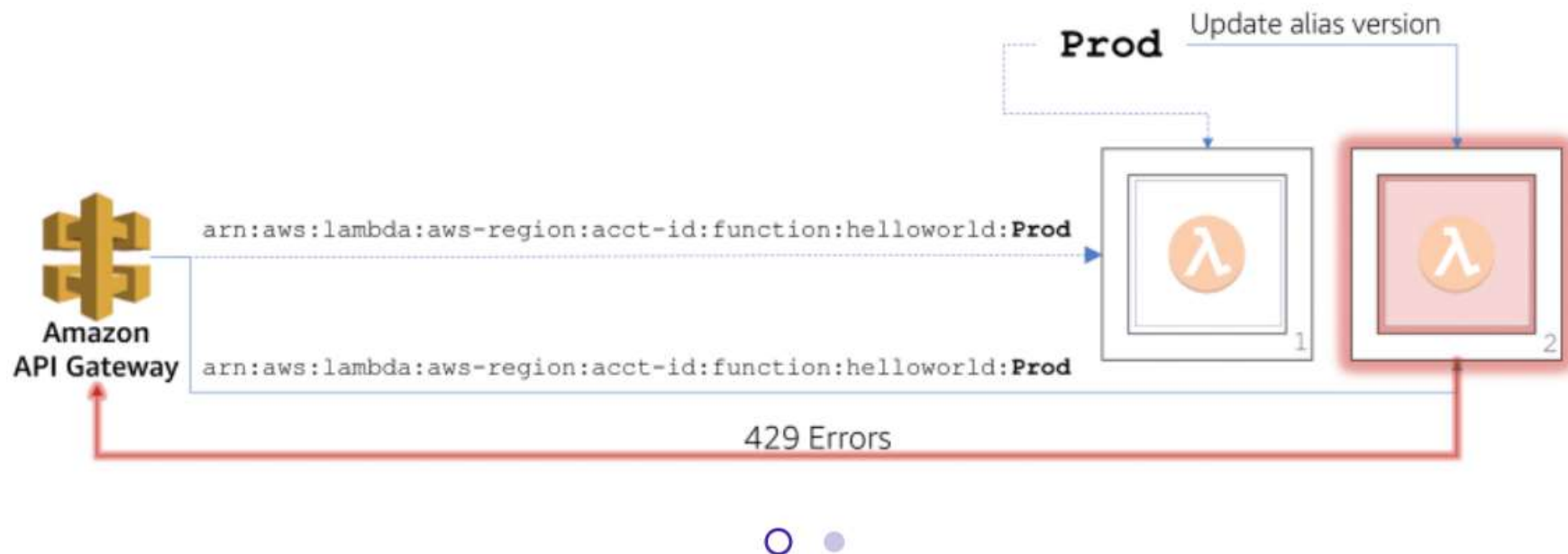
## Aliases point to specific versions

This makes it easy to promote or roll back versions without changing any code. The alias abstracts the need to know which version is being referenced.



# Deploying and Testing Serverless Applications

## Using aliases for traffic shifting

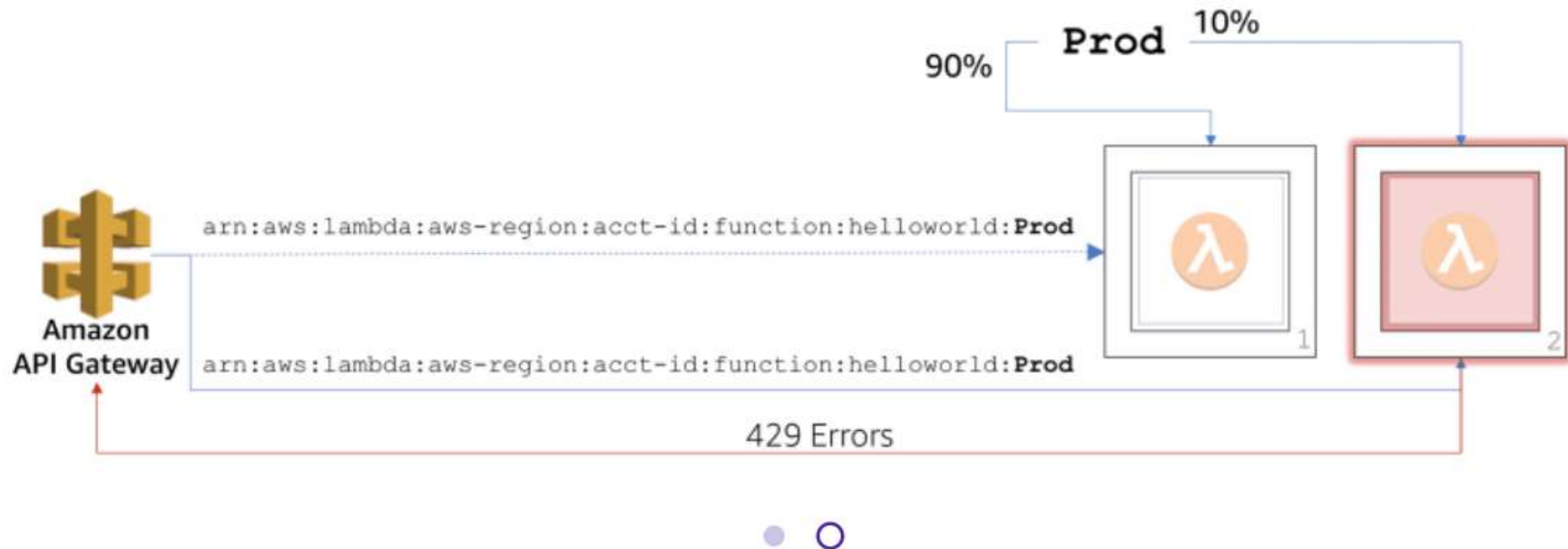


When an alias is updated to point to a new version, incoming requests immediately point to the new version. If there were any problems with the new version, it could impact all of your users immediately.



# Deploying and Testing Serverless Applications

## Using aliases for traffic shifting



One way to mitigate that risk is to use weighted aliases on your Lambda function. You can do this via the console in the 'Create Alias' dialog, or use the routing-config parameter via the CLI.

# Deploying and Testing Serverless Applications

## Automate safe deployments with AWS CodeDeploy

Lambda is integrated with **CodeDeploy** for automated rollout with traffic shifting.

- Canary: Traffic is shifted in two increments: X % first, remaining after Y minutes.
- Linear: Traffic is shifted in equal increments of X with Y minutes between each increment.

# Monitoring and Troubleshooting

## Monitoring with CloudWatch

**CloudWatch provides seven metrics out of the box for Lambda:**

### Invocations

Number of times a function is invoked in response to an event or invocation API call.

This metric is available directly from the Lambda Dashboard.

### Errors

Number of invocations that failed due to errors in the function (response code 4XX).

This metric is available directly from the Lambda Dashboard.

# Monitoring and Troubleshooting

## Monitoring with CloudWatch

**CloudWatch provides seven metrics out of the box for Lambda:**

### Duration

Elapsed time from when the function code starts executing to when it stops executing.

This metric is available directly from the Lambda Dashboard.

### Throttles

Number of Lambda function invocation attempts that were throttled due to invocation rates exceeding the customer's concurrent limits (error code 429).

This metric is available directly from the Lambda Dashboard.

# Monitoring and Troubleshooting

## Monitoring with CloudWatch

**CloudWatch provides seven metrics out of the box for Lambda:**

### **IteratorAge**

Emitted for stream-based invocations only. Measures the age of the last record for each batch of records processed. Age is the difference between the time Lambda received the batch and the time the last record in the batch was written to the stream.

### **ConcurrentExecutions**

Measures the sum of concurrent executions for a given function at a given point in time. Must be viewed as an average metric if aggregated across a time period.

### **UnreservedConcurrentExecutions**

Represents the sum of the concurrency of the functions that do not have a custom concurrency limit specified. Must be viewed as an average metric if aggregated across a time period.

# Monitoring and Troubleshooting

## Additional monitoring and troubleshooting tools

### AWS X-Ray

Traces path and timing of an invocation to locate bottlenecks and failures

- Use for performance tuning
- Identify call flow of Lambda functions and API calls

You can run X-Ray out of the box, and it will give you a high level view of your functions, but you can also instrument your code to trace individual API calls. (There is an example of X-Ray traces in the Lambda Lifecycle section).

### AWS Cloudtrail

Logs API calls made by or on behalf of a function

- Audit actions made against your application
- Integrate with a CloudWatch rules to respond to audit findings
- Export for additional analysis

CloudTrail can be an important tool for auditing serverless deployments and rolling back unplanned deployments.

# Monitoring and Troubleshooting

## Additional monitoring and troubleshooting tools

### Dead Letter Queues

Dead Letter Queues (DLQs) help you capture application errors that you cannot just discard but must respond to.

- Use DLQs to analyze failures for follow-up or code corrections.
- Available for asynchronous and non-stream polling events
- Can be an Amazon SNS topic or Amazon SQS queue

You can configure a DLQ from the Lambda Console.