AWS Lambda

Run code without provisioning or managing servers

Scott Bateman

Nov 17, 2020

Introduction to AWS Lambda

- Function-as-a-Service
- Run code without provisioning or managing servers
- Pay only for the compute time you consume
- Automatically runs your code with high availability
- Scale with usage

Lambda handles

- Load balancing
- Auto scaling
- Handling failures
- Security isolation
- OS management
- Managing utilization

(and many other things) for you

What is Serverless?

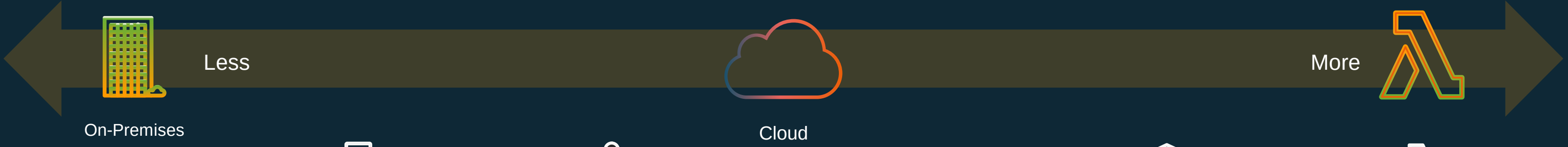
No infrastructure provisioning, no management

Automatic scaling

Pay for value

Highly available and secure

AWS operational responsibility models



Compute	Virtual Machine	EC2	Elastic Beanstalk		Fargate	AWS Lambda
Databases	MySQL	MySQL on EC2	RDS MySQL	RDS Aurora	Aurora Serverless	DynamoDB
Storage	Storage					S3
Messaging	ESBs		Amazon MQ	Kinesis		SQS / SNS
Analytics	Hadoop	Hadoop on EC2	EMR	Elasticsearch Service		Athena

Comparison of operational responsibility

More opinionated

	AWS manages	Customer manages
AWS Lambda Serverless functions	• Data source integrations • Physical hardware, software, networking, and facilities • Provisioning	• Application code
AWS Fargate Serverless containers	• Container orchestration, provisioning • Cluster scaling • Physical hardware, host OS/kernel, networking, and facilities	• Application code • Data source integrations • Security config and updates, network config, management tasks
ECS/EKS Container-management as a service	• Container orchestration control plane • Physical hardware software, networking, and facilities	• Application code • Data source integrations • Work clusters • Security config and updates, network config, firewall, management tasks
EC2 Infrastructure-as-a-Service	• Physical hardware software, networking, and facilities	• Application code • Data source integrations • Scaling • Security config and updates, network config, management tasks • Provisioning, managing scaling and patching of servers

Less opinionated

Anatomy of a Lambda Function

Serverless applications

AWS
Lambda

Serverless applications

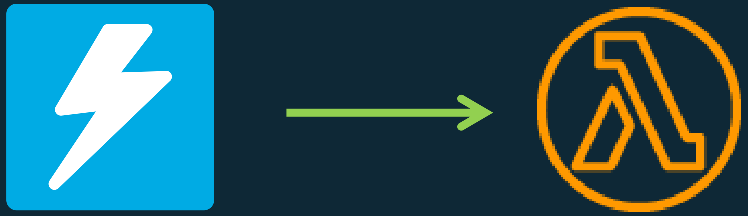
Function

Node.js
Python
Java
C#
Go
Ruby
Runtime API

Serverless Applications

Event source

Function



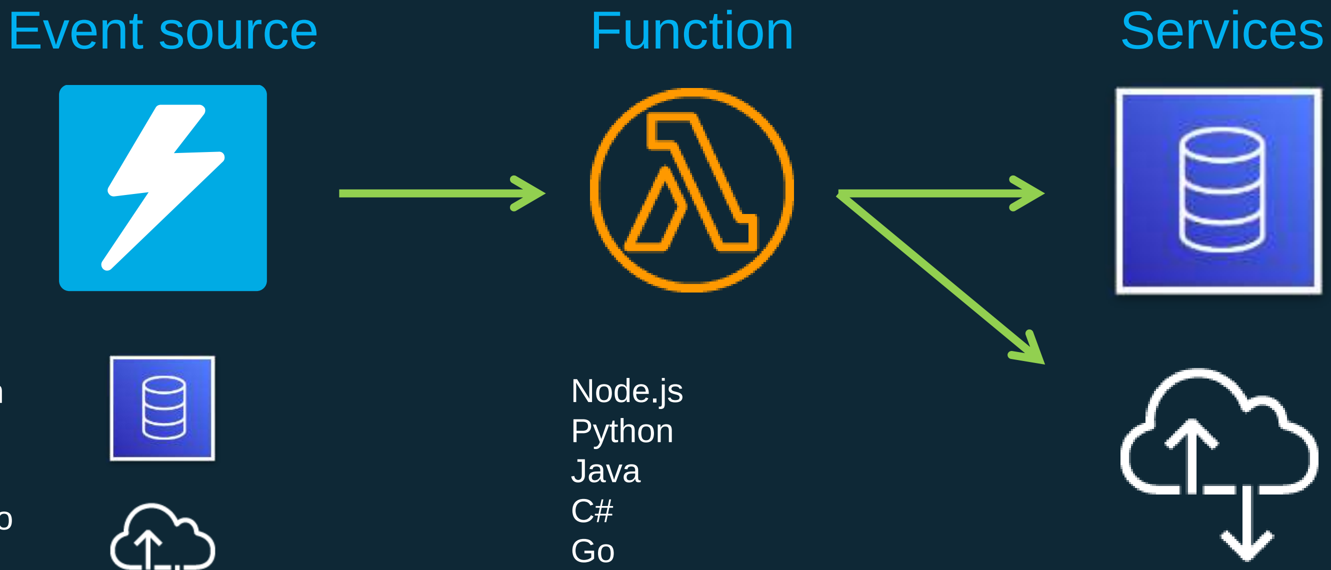
Changes in data state

Requests to endpoints

Changes in Resource state

Node.js
Python
Java
C#
Go
Ruby
Runtime API

Serverless Applications



Changes in data state
Requests to endpoints
Changes in Resource state

Node.js
Python
Java
C#
Go
Ruby
Runtime API

Anatomy of a Lambda Function

Handler() function

Function to be executed upon invocation

Event object

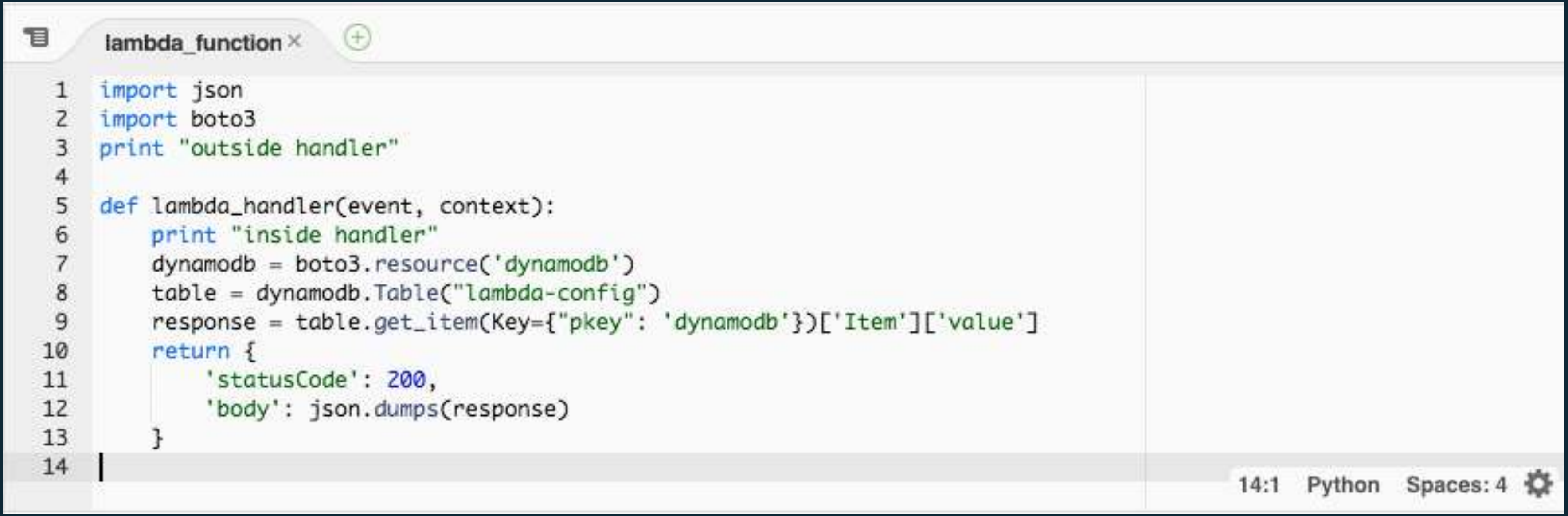
Data sent during Lambda function Invocation

Context object

Methods available to interact with runtime information (request ID, log group, more)

```python
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello World!')
    }
```

Instantiating inside the handler



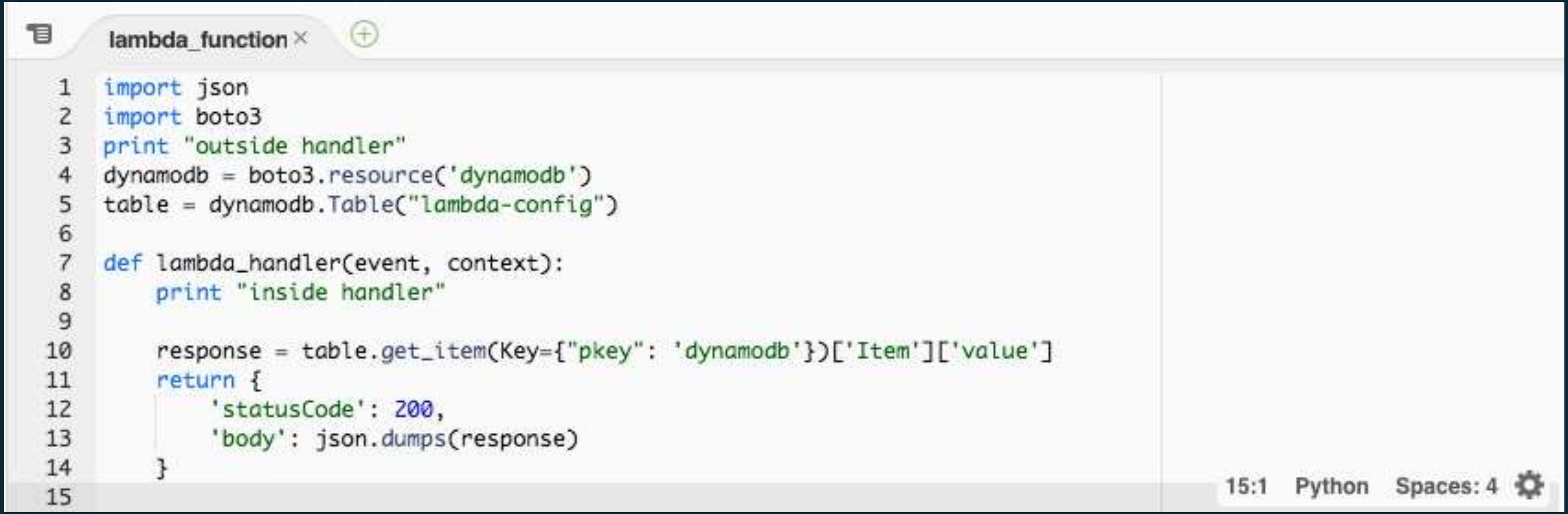
lambda_function

```python
import json
import boto3
print "outside handler"

def lambda_handler(event, context):
    print "inside handler"
    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table("lambda-config")
    response = table.get_item(Key={"pkey": 'dynamodb'})['Item']['value']
    return {
        'statusCode': 200,
        'body': json.dumps(response)
    }

```

14:1 Python Spaces: 4

200-300ms execution times

Instantiating outside the handler

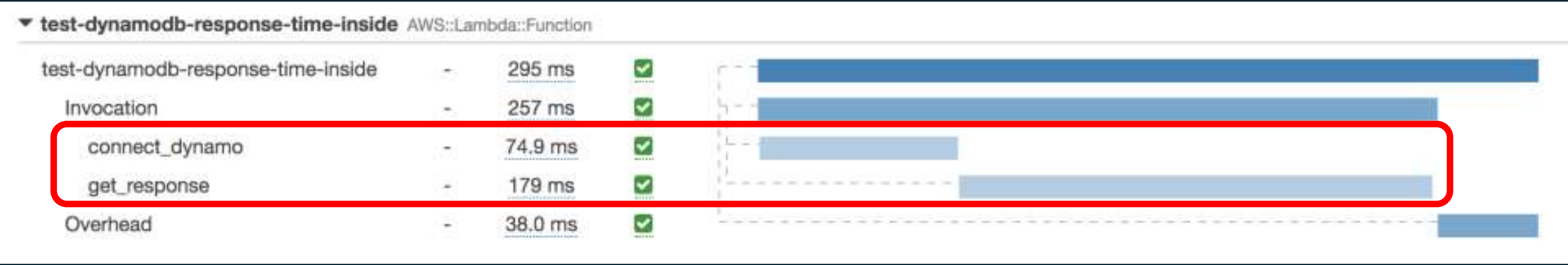


lambda_function ×

```python
import json
import boto3
print "outside handler"
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table("lambda-config")

def lambda_handler(event, context):
    print "inside handler"

    response = table.get_item(Key={"pkey": 'dynamodb'})['Item']['value']
    return {
        'statusCode': 200,
        'body': json.dumps(response)
    }

```

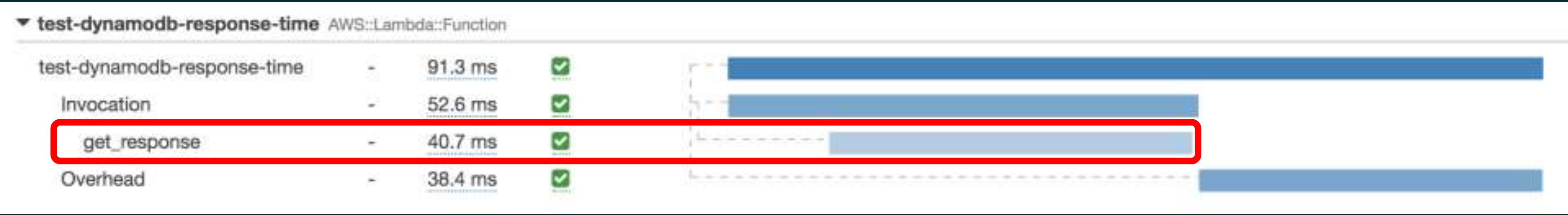
double digit ms execution times

DynamoDB connection instantiated **inside** the handler ~300ms



test-dynamodb-response-time-inside AWS::Lambda::Function			
test-dynamodb-response-time-inside	-	295 ms	✓
Invocation	-	257 ms	✓
connect_dynamo	-	74.9 ms	✓
get_response	-	179 ms	✓
Overhead	-	38.0 ms	✓

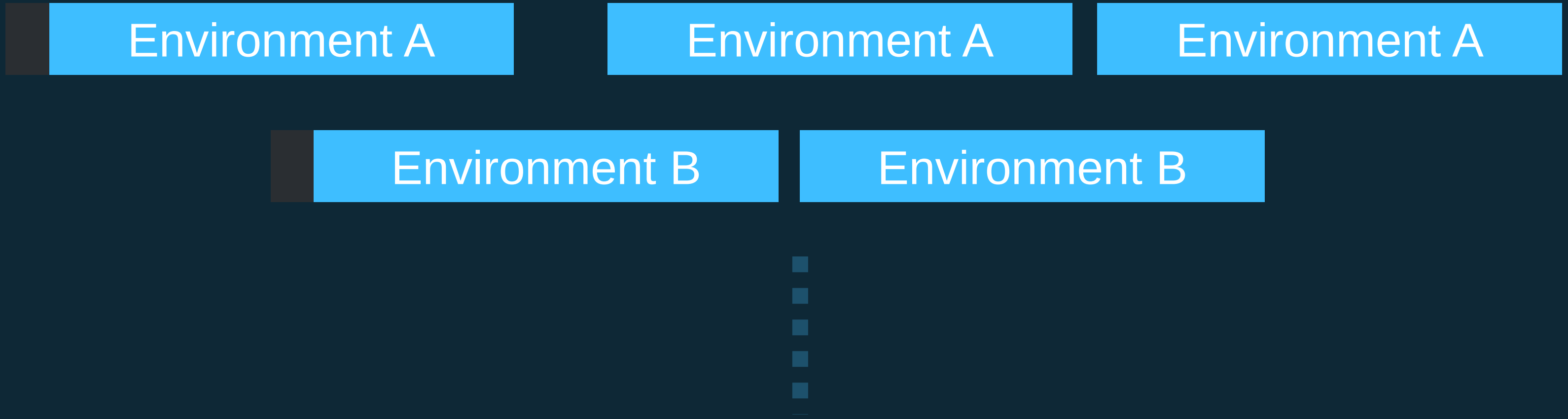
DynamoDB connection instantiated **outside** the handler ~90ms



test-dynamodb-response-time AWS::Lambda::Function			
test-dynamodb-response-time	-	91.3 ms	✓
Invocation	-	52.6 ms	✓
get_response	-	40.7 ms	✓
Overhead	-	38.4 ms	✓

Reusing database connections

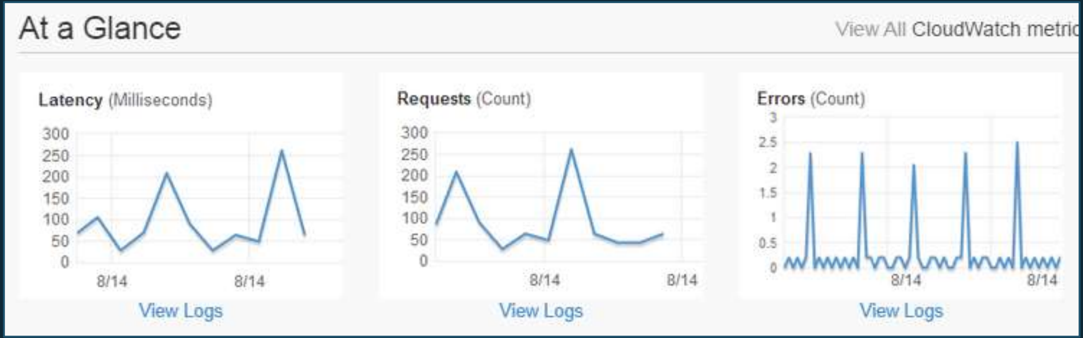
- Connections made outside the handler will be reused



Max number of database connections = Max number of concurrent executions

Monitoring and debugging Lambda functions

- AWS Lambda console includes a dashboard for functions
  - Lists all Lambda functions
  - Easy editing of resources, event sources and other settings
  - At-a-glance metrics
- Metrics automatically reported to Amazon CloudWatch for each Lambda function
  - Requests
  - Errors
  - Latency
  - Throttles



Lambda Layers

Lets functions easily share code: Upload layer once, reference within any function

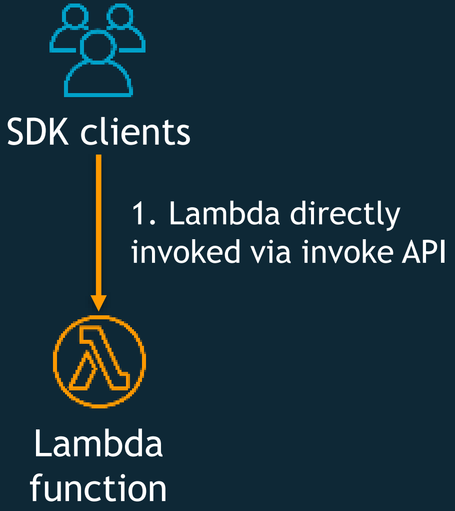
Promote separation of responsibilities, lets developers iterate faster on writing business logic

Built in support for secure sharing by ecosystem

Lambda Layers: Uses cases

- Custom code, that is used by more than one function
- Libraries, modules, frameworks to simplify the implementation of your business logic
    - Security/monitoring service
- Shared code that does not change frequently
- Bring your own Runtime
    - C++
    - Rust
    - PHP

Lambda API



API provided by the Lambda service

Used by all other services that invoke Lambda across all models

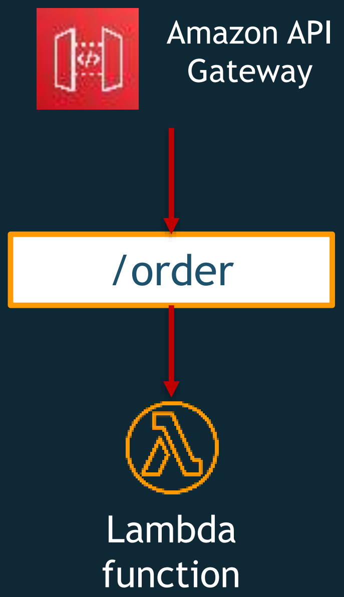
Supports sync and async

Can pass any event payload structure you want

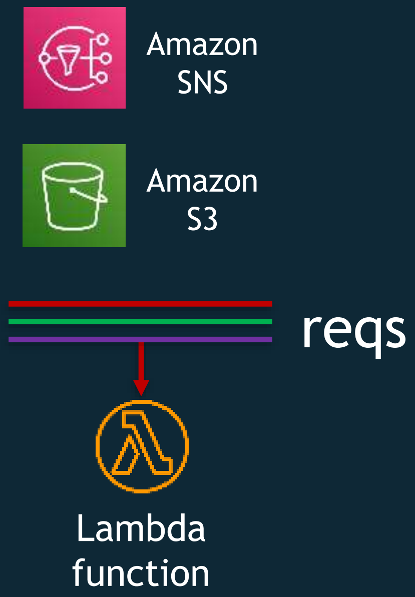
Client included in every SDK

Lambda Execution Models

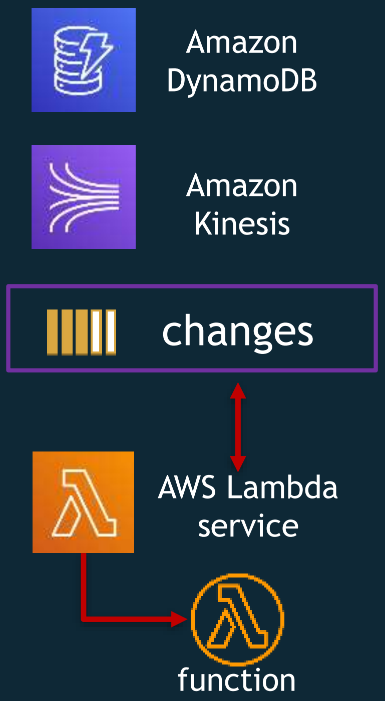
Synchronous (push)



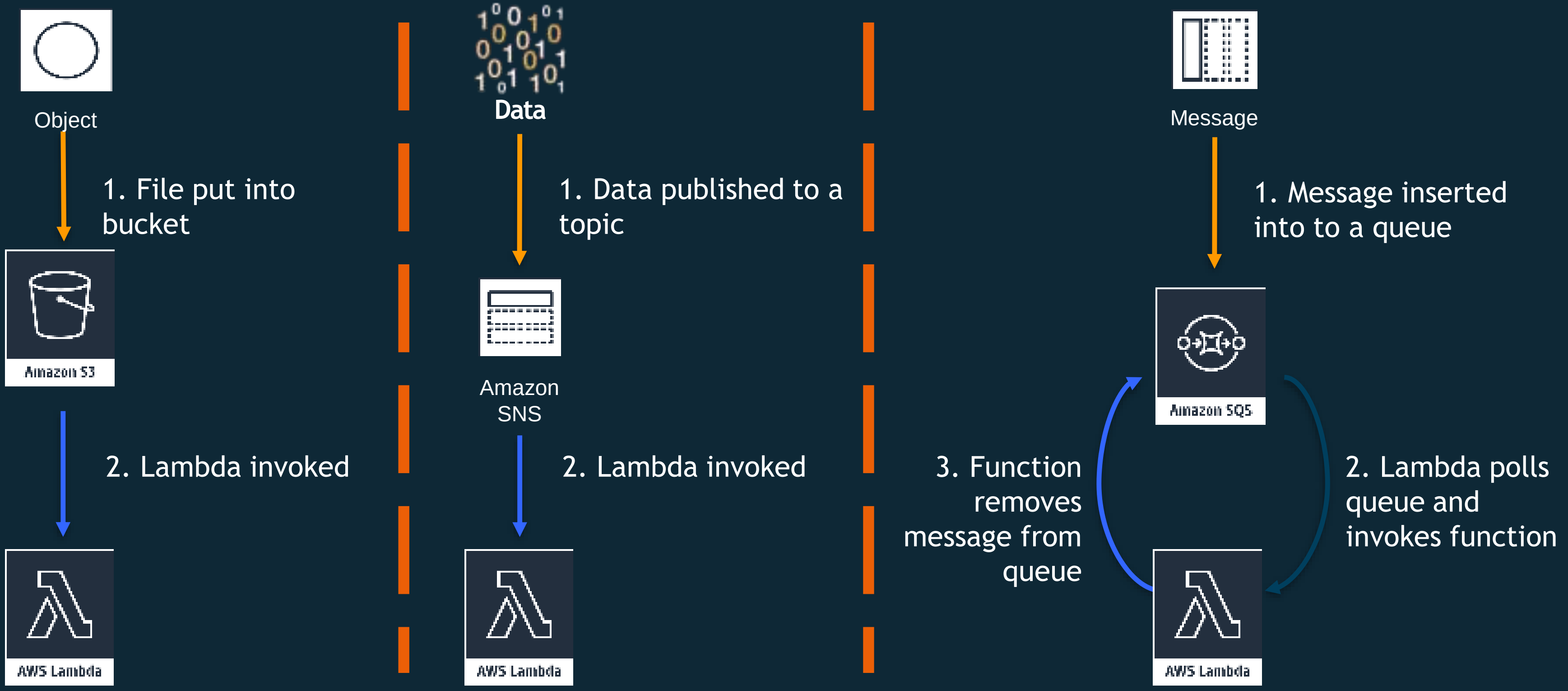
Asynchronous (event)



Stream (Poll-based)

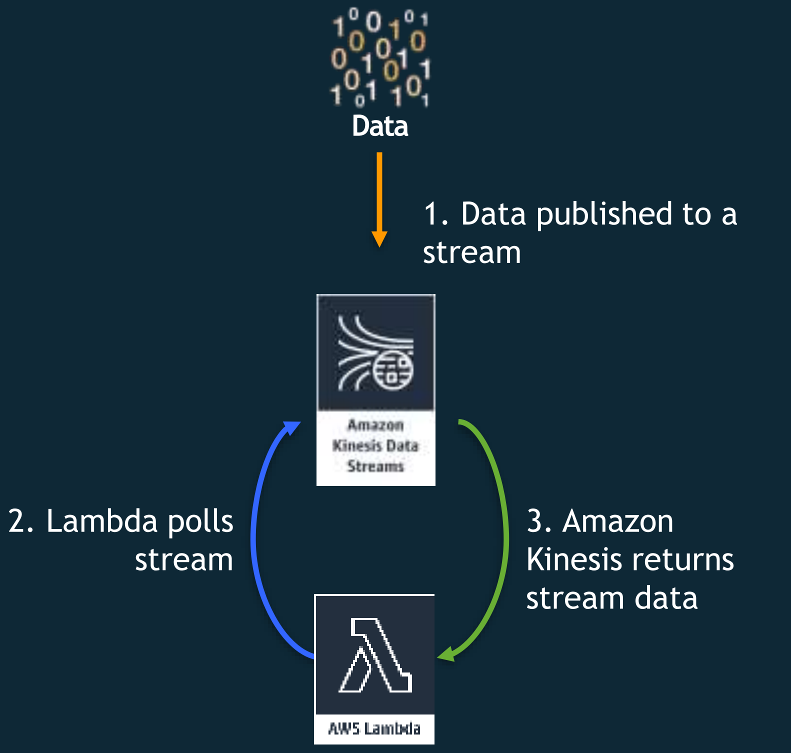


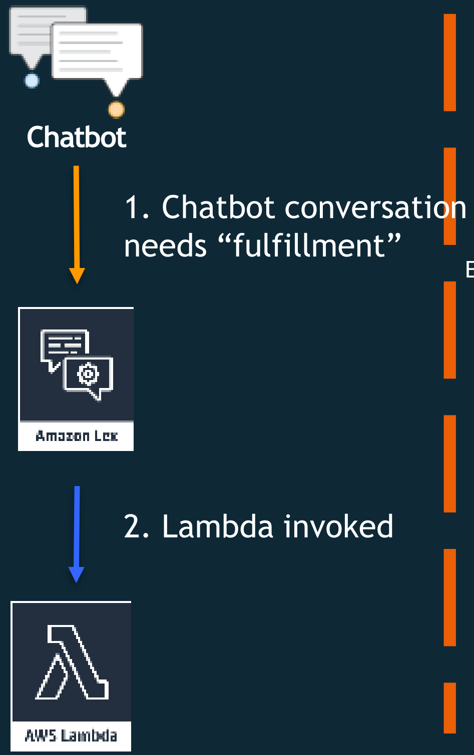
Serverless architectures

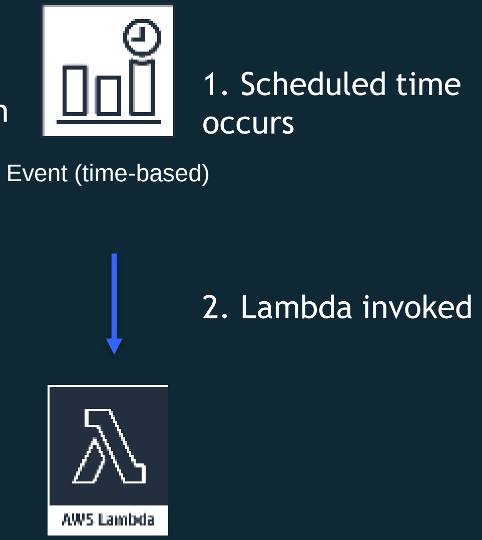


Object
1. File put into bucket
Amazon S3
2. Lambda invoked
AWS Lambda
Data
1. Data published to a topic
Amazon SNS
2. Lambda invoked
AWS Lambda
Message
1. Message inserted into to a queue
Amazon SQS
2. Lambda polls queue and invokes function
3. Function removes message from queue
AWS Lambda

Serverless Architectures


Data
1. Data published to a stream
Amazon Kinesis Data Streams
2. Lambda polls stream
3. Amazon Kinesis returns stream data
AWS Lambda





Example event sources that trigger AWS Lambda

DATA STORES

- Amazon S3
- Amazon DynamoDB
- Amazon Kinesis
- Amazon Cognito

ENDPOINTS

- Amazon API Gateway
- AWS IoT
- AWS Step Functions
- Amazon Alexa

CONFIGURATION REPOSITORIES

- AWS CloudFormation
- AWS CloudTrail
- AWS CodeCommit
- Amazon CloudWatch

EVENT/MESSAGE SERVICES

- Amazon SES
- Amazon SNS
- Cron events

… and more on the way!

Dead-Letter Queue

- Asynchronous Lambda invocations are retried two more times (3 times total)
- Lambda can forward payloads that were not processed to a dead-letter queue (IF configured!)
- A mechanism to handle exceptions and failures gracefully

Common AWS Lambda use cases

Web Apps

- Static websites
- Complex web apps
- Packages for Flask and Express

Backends

- Apps & services
- Mobile
- IoT

Data Processing

- Real time
- MapReduce
- Batch

Chatbots

- Powering chatbot logic

Amazon Alexa

- Powering voice-enabled apps
- Alexa Skills Kit

IT Automation

- Policy engines
- Extending AWS services
- Infrastructure management

When do we choose Lambda over other compute offerings?

AWS Compute Offerings

Service	Amazon EC2	Amazon ECS	AWS Fargate	AWS Lambda
Unit of scale	VM	Task	Task	Function
Level of abstraction	H/W	OS	OS	Runtime

AWS Compute Offerings

Service	**Amazon EC2**	**Amazon ECS**	**AWS Fargate**	**AWS Lambda**
How do I choose?	I want to configure servers, storage, networking, and my OS	I want to run servers, configure applications, and control scaling	I want to run my containers	Run my code when it's needed

The two Serverless compute options

AWS Lambda

Serverless event-driven code execution

Short-lived

All language runtimes

Data source integrations

AWS Fargate

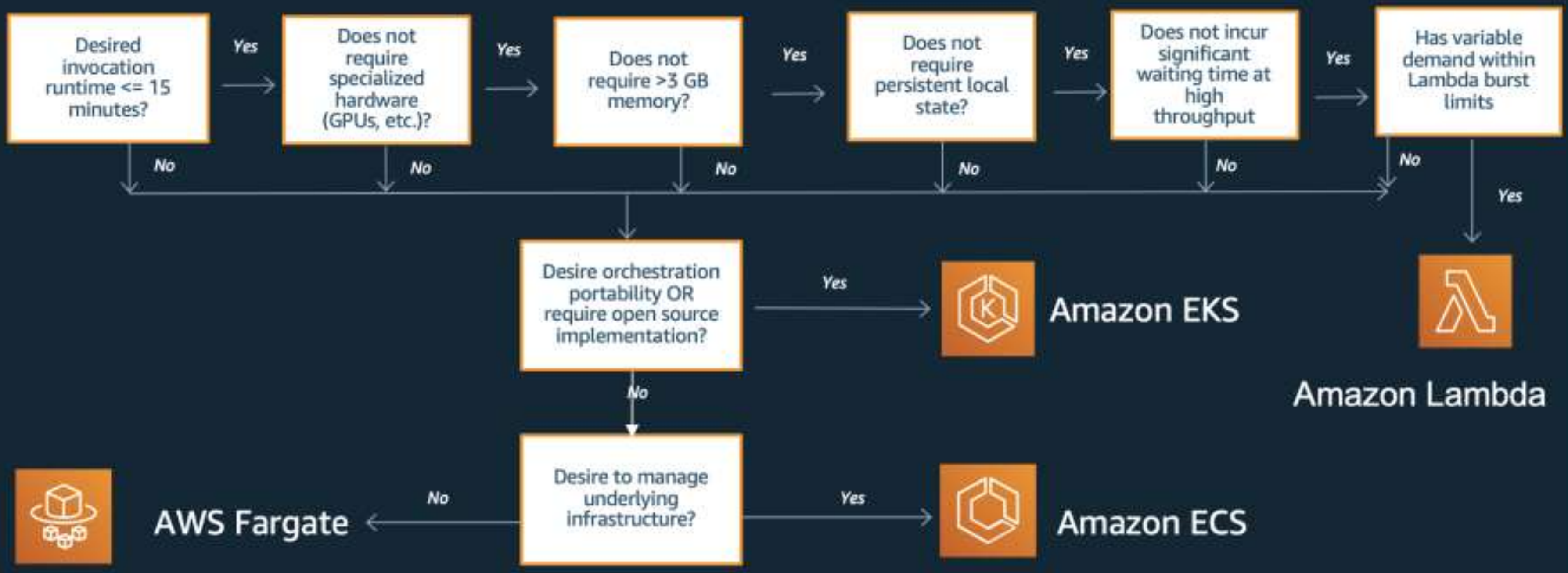
Serverless compute engine for containers

Long-running

Bring existing code

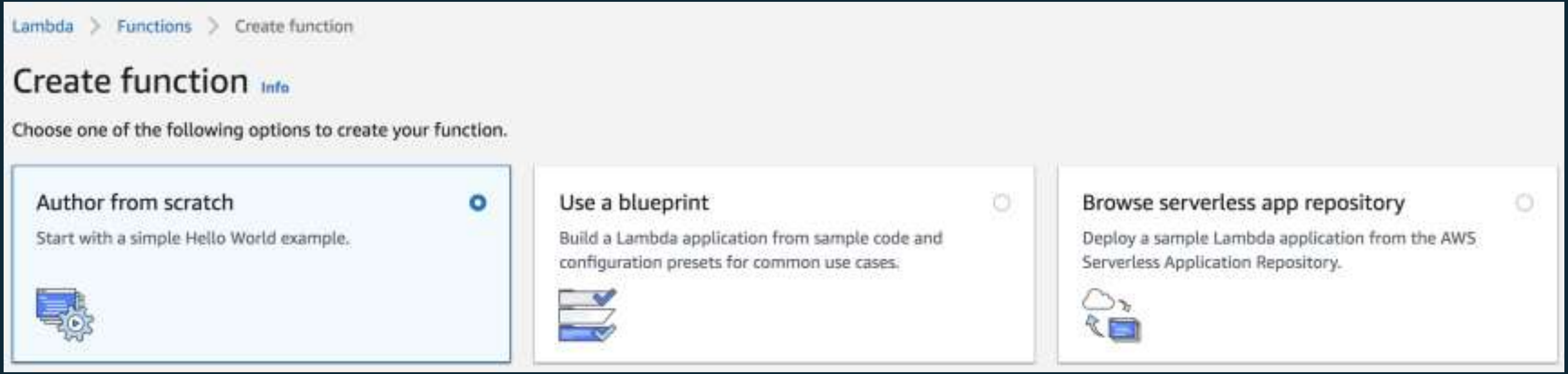
Fully-managed orchestration

Picking a Compute Platform: Containers vs. Lambda


Desired invocation runtime <= 15 minutes?
Yes
Does not require specialized hardware (GPUs, etc.)?
Yes
Does not require >3 GB memory?
Yes
Does not require persistent local state?
Yes
Does not incur significant waiting time at high throughput
Yes
Has variable demand within Lambda burst limits
No
No
No
No
No
No
Yes
Desire orchestration portability OR require open source implementation?
Yes
Amazon EKS
Amazon Lambda
No
Desire to manage underlying infrastructure?
No
AWS Fargate
Yes
Amazon ECS

How to get started?

Getting started with Lambda



Lambda > Functions > Create function

Create function Info

Choose one of the following options to create your function.

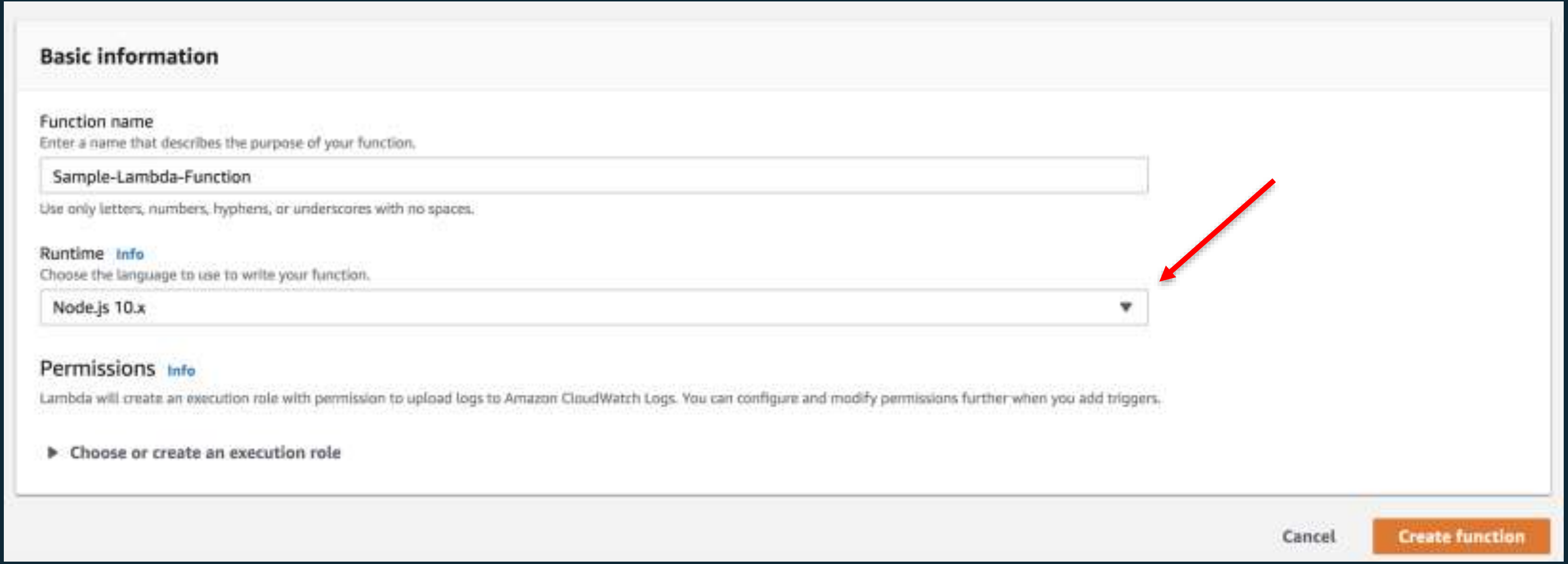
[✓ Author from scratch]
Start with a simple Hello World example.

Use a blueprint
Build a Lambda application from sample code and configuration presets for common use cases.

Browse serverless app repository
Deploy a sample Lambda application from the AWS Serverless Application Repository.

Getting started with Lambda

1. Select your runtime



Basic information

Function name
Enter a name that describes the purpose of your function.
Sample-Lambda-Function
Use only letters, numbers, hyphens, or underscores with no spaces.

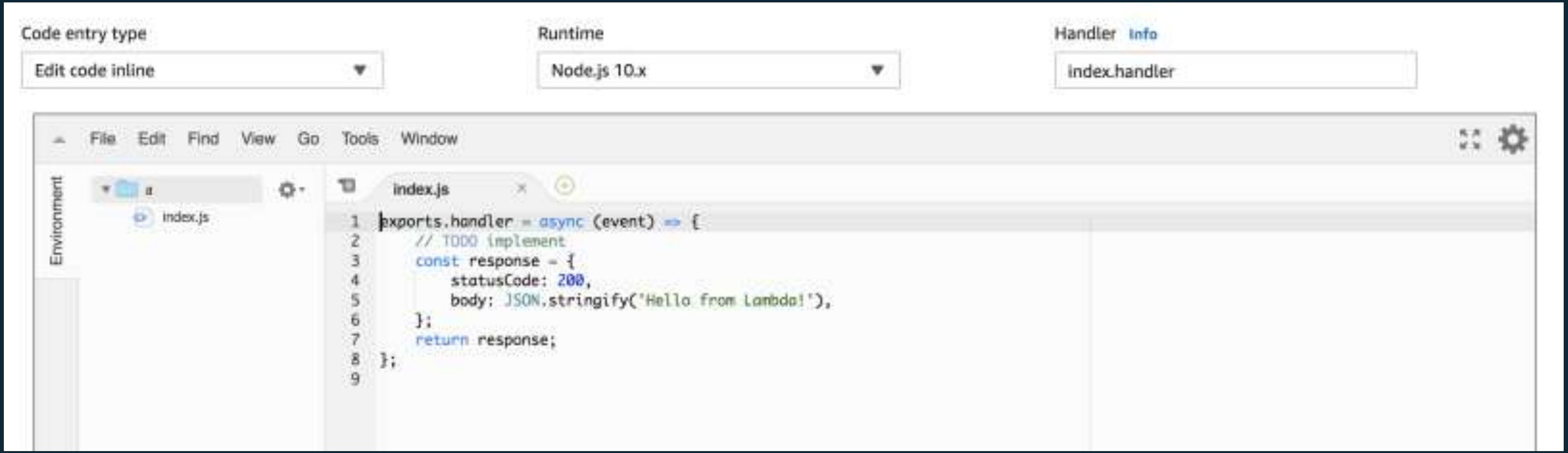
Runtime Info
Choose the language to use to write your function.
Node.js 10.x

Permissions Info
Lambda will create an execution role with permission to upload logs to Amazon CloudWatch Logs. You can configure and modify permissions further when you add triggers.

▶ Choose or create an execution role

Getting started with Lambda

2. Write your code (your handler function)



Code entry type: Edit code inline

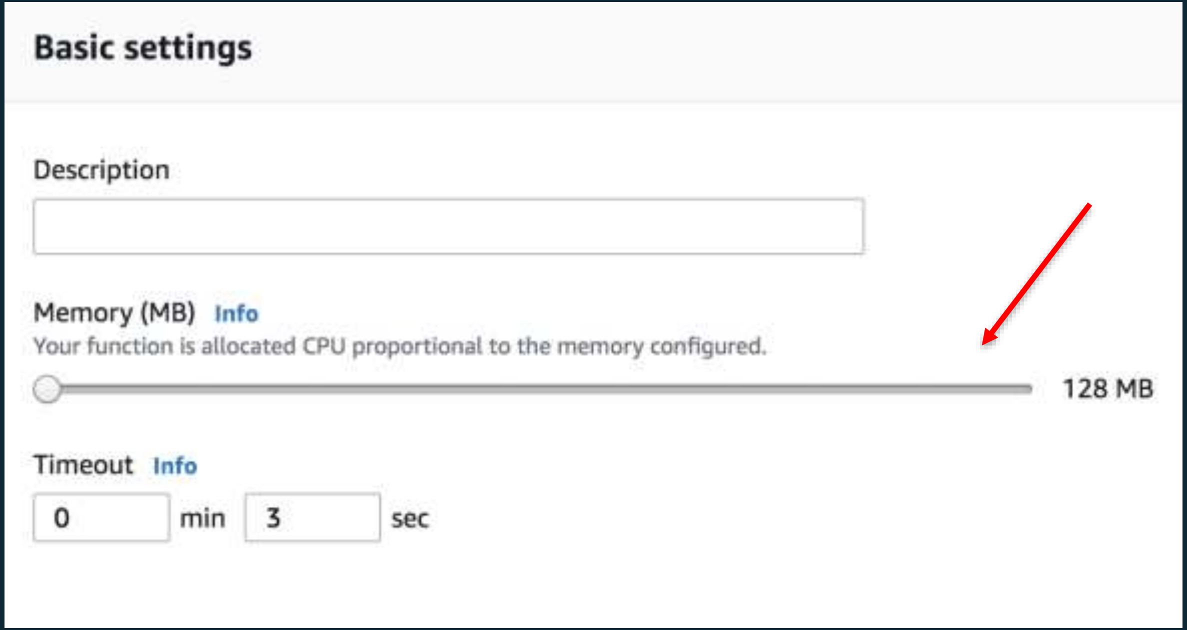
Runtime: Node.js 10.x

Handler Info: index.handler

```
exports.handler = async (event) => {
    // TODO implement
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

Getting started with Lambda

3. Select memory size



Basic settings

Description

Memory (MB) Info

Your function is allocated CPU proportional to the memory configured.

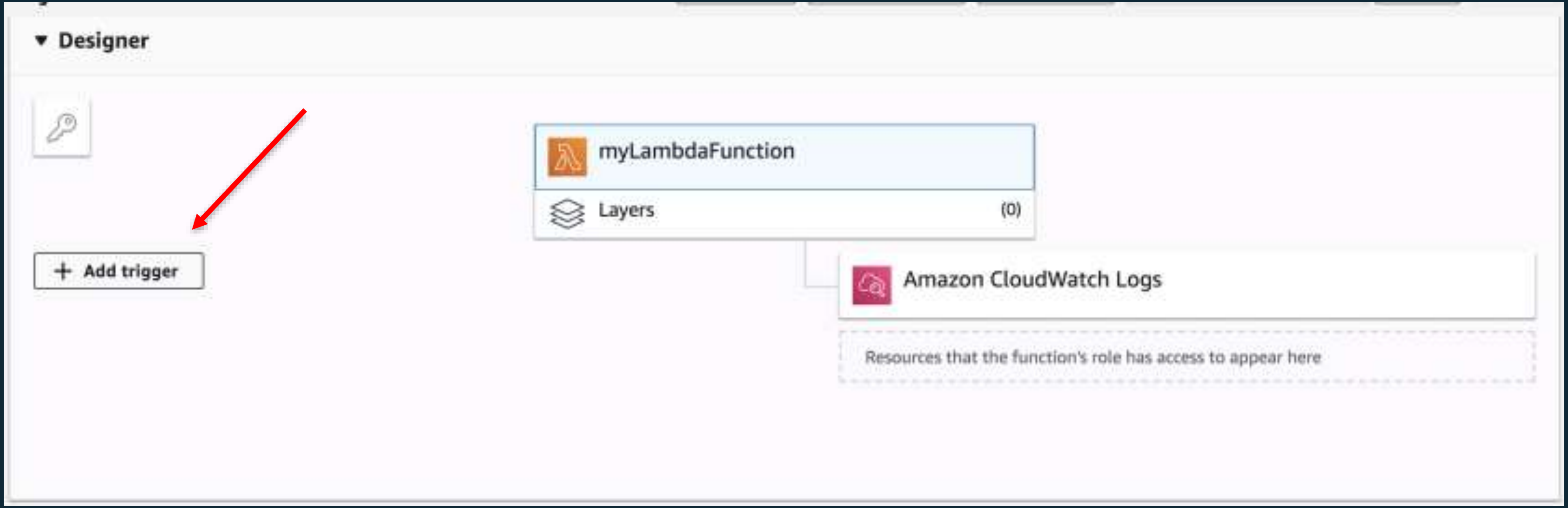
128 MB

Timeout Info

0 min 3 sec

Getting started with Lambda

4. Add a trigger to start invoking your Lambda function!



Resources that the function's role has access to appear here

Step Functions

Orchestration for serverless apps

"I want to sequence functions"

"I want to select functions based on data"

"I want to run functions in parallel"

"I want to retry functions"

"I want to try/catch/finally"

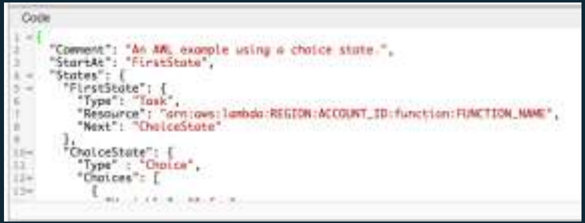
"I want to run code for hours"

AWS Step Functions

AWS Step Functions

Easily coordinate multiple Lambda functions using visual workflows

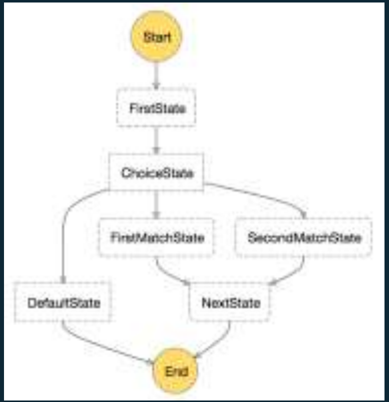
Define in JSON



Code

```
{
  "Comment": "An AWL example using a choice state.",
  "StartAt": "FirstState",
  "States": {
    "FirstState": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
      "Next": "ChoiceState"
    },
    "ChoiceState": {
      "Type" : "Choice",
      "Choices": [
        {
```

Visualize in the Console



Monitor Executions

Benefits of Step Functions orchestration

Productivity

Coordinate and visualize Lambda functions as a series of steps to quickly create serverless apps

Resilience

Automatically trigger and track each step at scale and handle errors with built-in retry and fallback

Agility

Change and add steps without writing code to evolve applications and innovate faster

AWS Service integrations with Step Functions

AWS Lambda

AWS Batch

Amazon DynamoDB

AWS Fargate

Amazon Elastic Container Service

Amazon Simple Notification Service

AWS Step Functions

Amazon Simple Queue Service

Amazon SageMaker

AWS Glue

AWS Lambda best practices

- Limit your function/code size
- Node – remember execution is asynchronous
- 500 MB /tmp directory provided to each function
- Don’t assume function will reuse underlying infrastructure
    - But take advantage of it when it does occur
- You own the logs
    - Include details from service-provided context
- Create custom metrics
    - Operations-centric vs. business-centric

Additional best practices

- Use environment variables
  - Parameterize code and change parameters independent of code updates
  - Use for securing credentials and keeping them out of code
- Externalize authorization to IAM roles whenever possible
  - Least privilege <u>and</u> separate IAM roles
- Externalize configuration
  - DynamoDB is great for this
- Take advantage of dead letter queues
  - Use to handle failed invocations

Additional best practices

- Make sure your downstream setup “keeps up” with Lambda scaling
    - Limit concurrency when talking to relational databases
- Be aware of service throttling
    - Engage AWS Support to increase your limits
- Contact AWS Support before known large scaling event
    - Infrastructure Event Management (IEM) offers real-time support for large scaling events
    - IEM is available for Enterprise and Business support customers

Thank you!