



# ***Serverless Applications in Python***

October 29, 2021

**Dmitry Pavlov**, Sr. Solutions Architect, Academic Medical Centers

# Introduction



Dmitry Pavlov

Dmitry Pavlov has a twenty-year background in research and development in medical imaging, coming to AWS from a career in healthcare software development industry. He holds an MS in Electrical Engineering, and has particular interest in research, medical imaging, AI, and open source software.

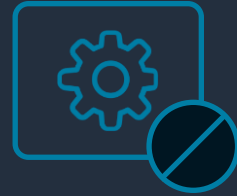
# Why Serverless Python?

## Why Python?

- Easy to use
- Many libraries available
- Powerful
- Code readability

## Why Serverless?

# What is serverless?



No infrastructure provisioning,  
no management



Automatic scaling

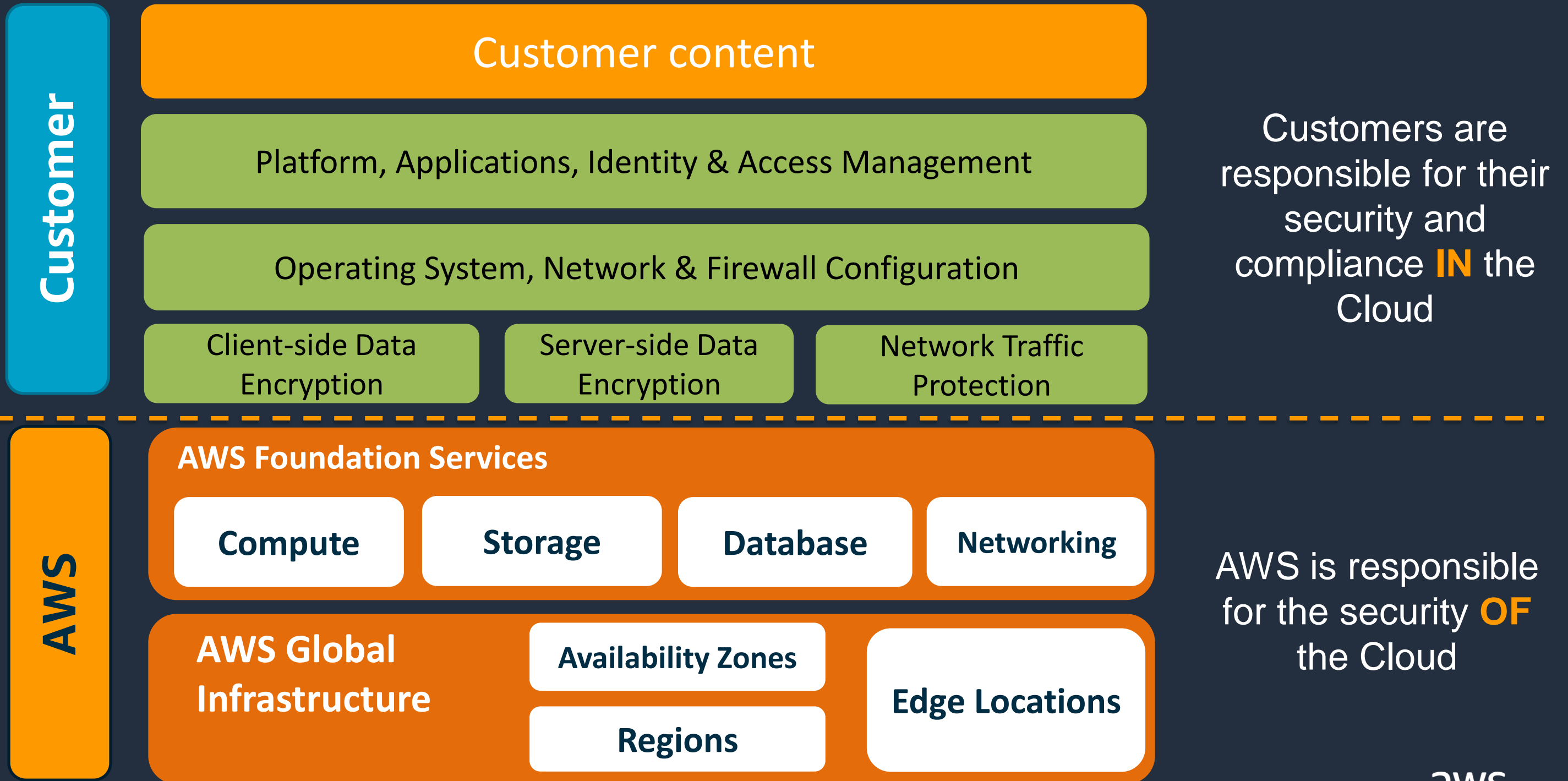
Pay for value



Highly available and secure



# Shared responsibility model – for Security



# Shifted responsibilities – for Serverless

Customer

Customer applications

Customers are responsible for their applications

AWS

Lambda

SageMaker

.....

DynamoDB

Aurora

Virtualization and container services

Compute

Storage

Security

Networking

AWS Global Infrastructure

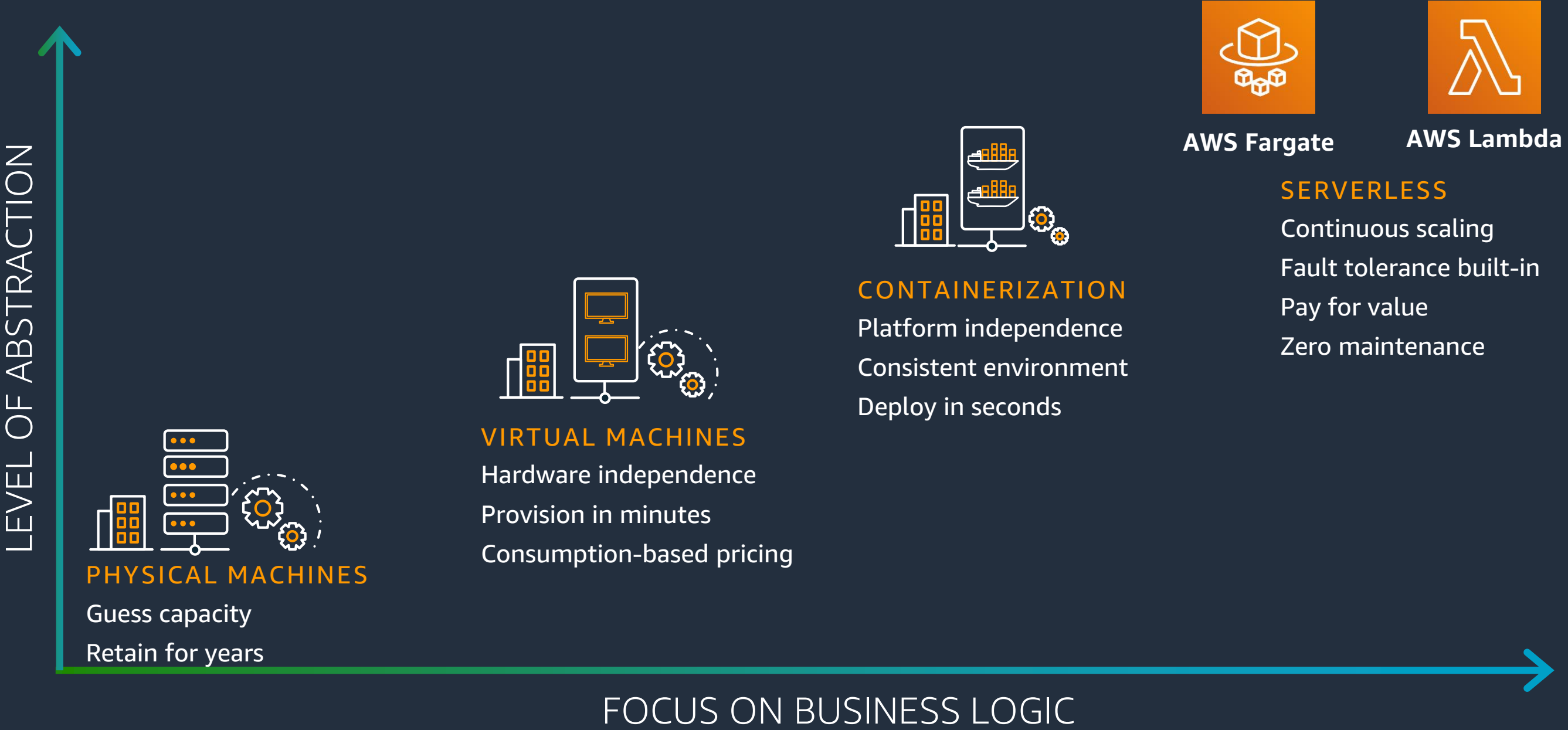
Availability Zones

Regions

Edge Locations

AWS is responsible for the management **OF** the servers , framework and infrastructure

# Computing evolution



# Choice of compute



AWS Fargate

## Serverless Containers

- Long-running
- Abstracts the platform
- Fully-managed orchestration
- Fully-managed cluster scaling



AWS Lambda

## Serverless Functions

- Event-driven
- Many language runtimes
- Data source integrations
- Fully-managed infrastructure

# Serverless spans many different categories of services

## COMPUTE



AWS  
Fargate



AWS  
Lambda

## DATA STORES



Amazon  
S3

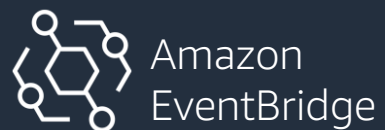


Amazon Aurora  
Serverless



Amazon  
DynamoDB

## INTEGRATION



Amazon  
EventBridge



Amazon  
API Gateway



Amazon  
SQS



Amazon  
SNS



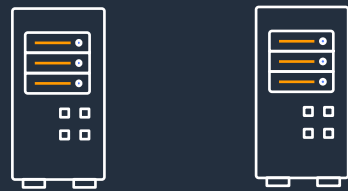
AWS  
Step Functions



AWS  
AppSync

# Three-tier web application architecture

## Traditional



Web servers



Application servers



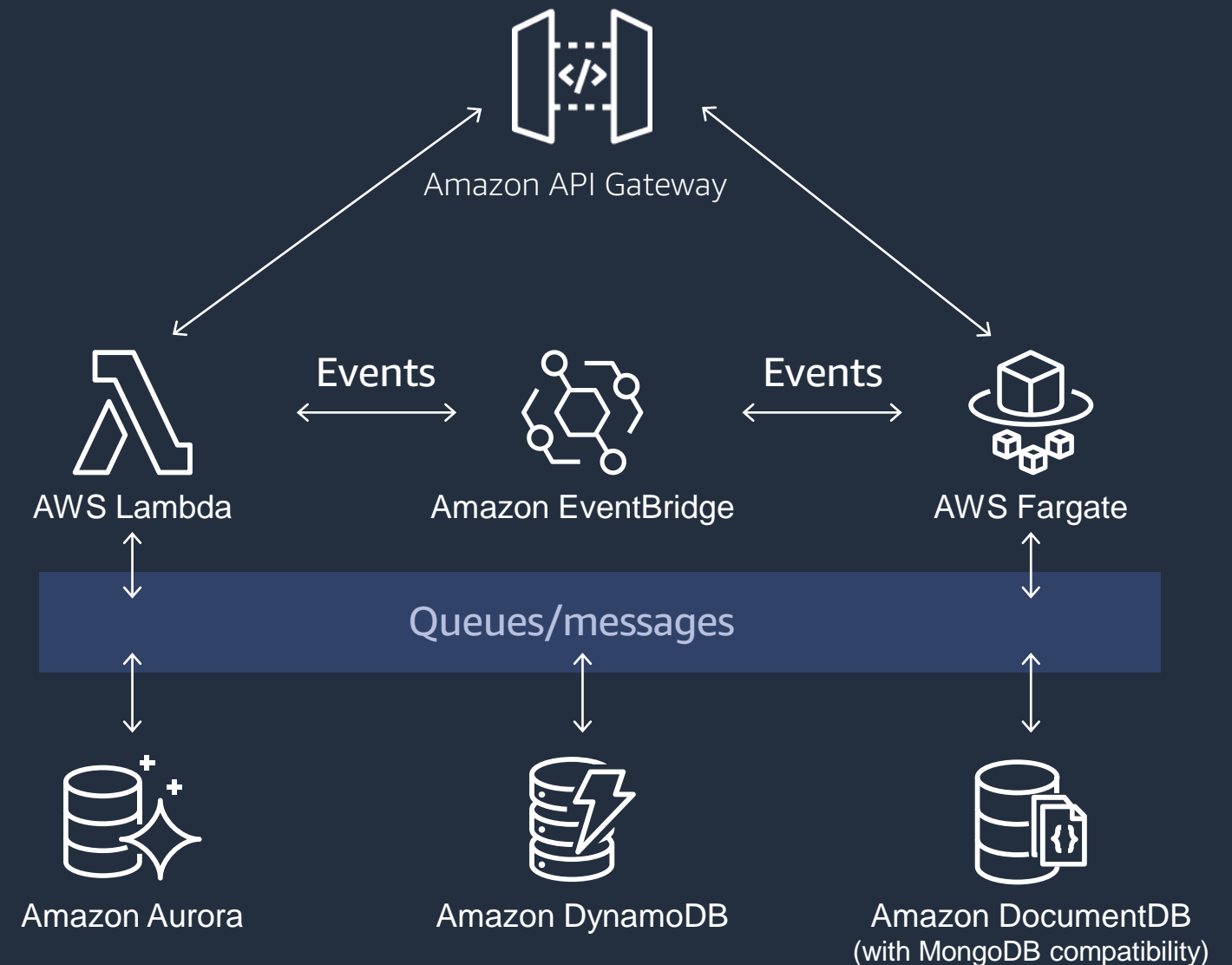
Database servers

## Presentation

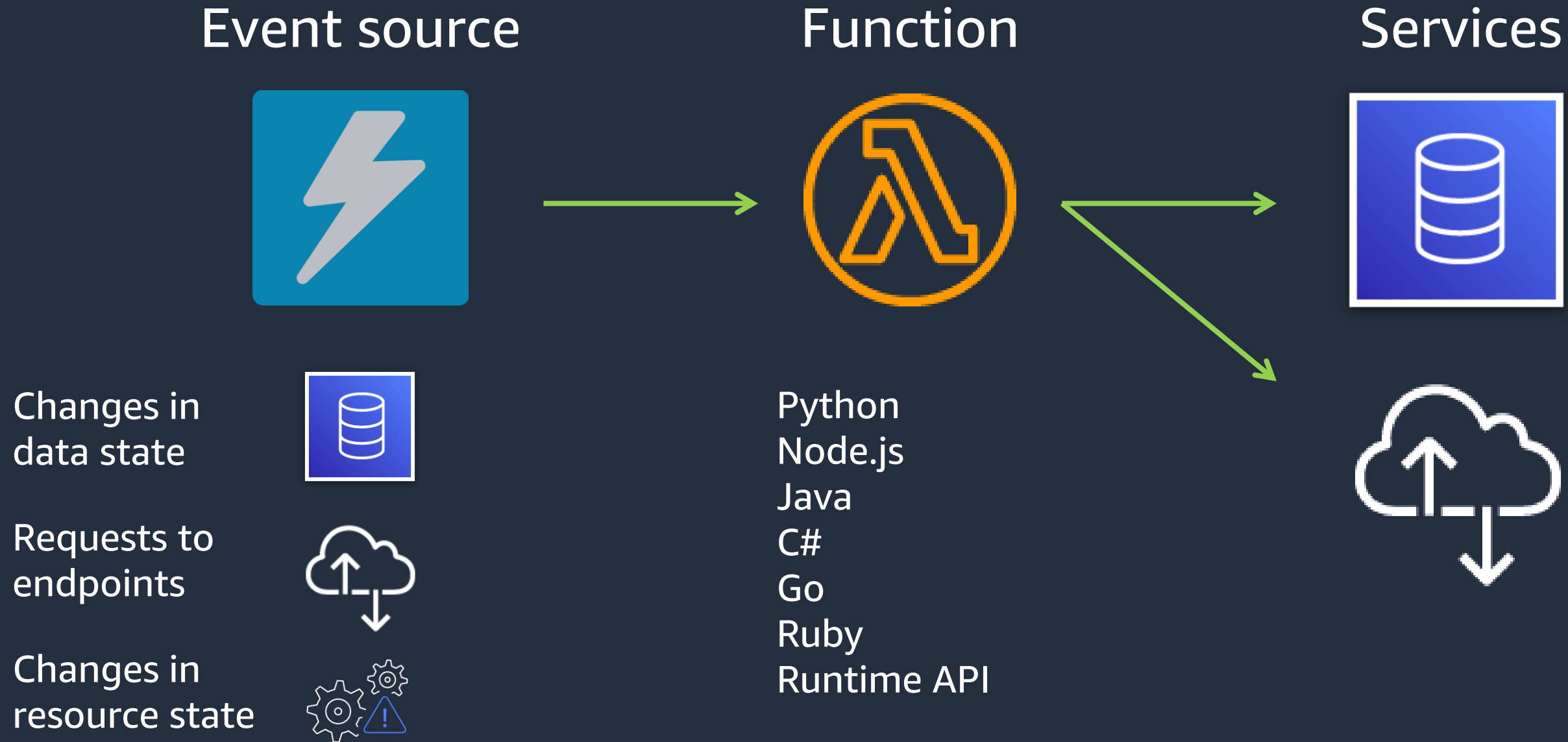
## Business logic

## Data layer

## Modern



# Serverless applications with AWS Lambda



# Example event sources that trigger AWS Lambda

## DATA STORES



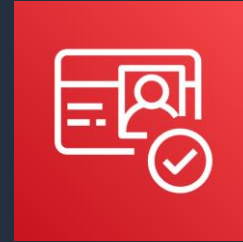
Amazon S3



Amazon  
DynamoDB



Amazon  
Kinesis

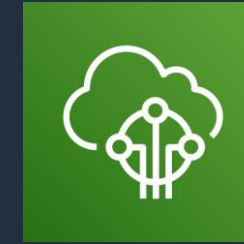


Amazon  
Cognito

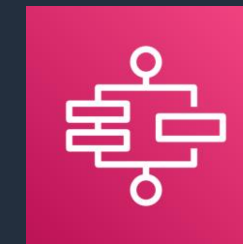
## ENDPOINTS



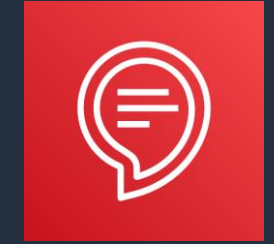
Amazon  
API Gateway



AWS IoT

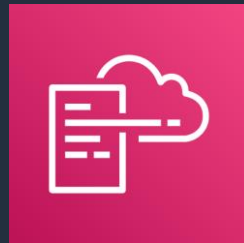


AWS Step  
Functions

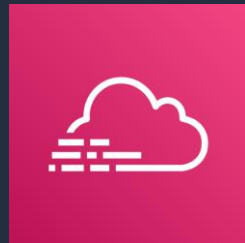


Amazon  
Alexa

## CONFIGURATION REPOSITORIES



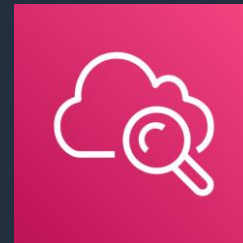
AWS  
CloudFormation



AWS CloudTrail

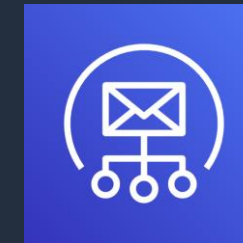


AWS  
CodeCommit



Amazon  
CloudWatch

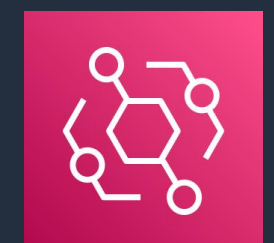
## EVENT/MESSAGE SERVICES



Amazon SES



Amazon SNS



Amazon EventBridge

# Lambda Function Invocation

## Lambda function handler

Function to be executed upon invocation

## Event object

Information from the user or invoking AWS service

## Context object

Information about the invocation, function, and execution environment

```
import json

def lambda_handler(event, context):
    myname = event['name']
    print(f"Saying hello to {myname}")
    print(f"Logging to CloudWatch group: {context.log_group_name}")
    return {
        'statusCode': 200,
        'body': json.dumps(f"Hello, {myname}!")
    }
```

```
2021-02-14T15:20:30.097-05:00    START RequestId: ff01de3c Version: $LATEST
2021-02-14T15:20:30.097-05:00    Saying hello to Andrew
2021-02-14T15:20:30.097-05:00    Logging to CloudWatch group: /aws/lambda/hello-00
2021-02-14T15:20:30.098-05:00    END RequestId: ff01de3c
2021-02-14T15:20:30.098-05:00    REPORT RequestId: ff01de3c Billed Duration: 1 ms Memory Size: 128 MB
```

# Using AWS Lambda



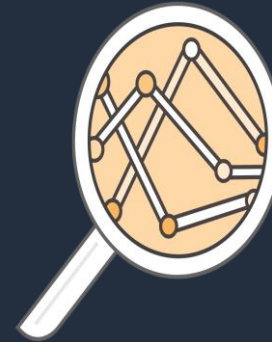
## Authoring functions

- AWS Console
- Cloud9
- WYSIWYG editor or upload packaged .zip
- Third-party plugins (Eclipse, VS Code)



## Programming model

- Use processes, threads, /tmp, sockets normally
- AWS SDK built in (Python and Node.js)



## Monitoring and logging

- Metrics for requests, errors, and throttles
- Built-in logs to Amazon CloudWatch Logs
- AWS X-Ray integration



## Stateless

- Persist data using external storage
- No affinity or access to underlying infrastructure

# AWS Console Lambda Editor

The screenshot shows the AWS Lambda console interface for editing a function named "Process\_S3\_Objects". The breadcrumb navigation at the top reads "Lambda > Functions > Process\_S3\_Objects".

**Function overview**

- Function name: Process\_S3\_Objects
- Layers: (1)
- Triggers: S3
- Buttons: "+ Add trigger" and "+ Add destination"

**Code source**

The code editor shows the following Python code:

```
1 import json
2 import os
3 import pydicom
4 import boto3
5 import botocore
6
7 debug = True
8 bucket_out = "dmpavlov-study-metadata2"
9 tmpDir = "/tmp/" # base folder for temp files
10 base_prefix = 'dicomweb/'
11 s3 = boto3.client('s3', region_name='us-east-2')
12
13 def lambda_handler(event, context):
14     for record in event['Records']:
15         bucket = record['s3']['bucket']['name']
16         key = record['s3']['object']['key']
17         size = record['s3']['object']['size']
18         # directory to download DICOM data
19         dicom_file = tmpDir + key
20
21         studyData = {}
22
23         # Download S3 object into a file
24         with open(dicom_file, 'wb') as data:
25             try:
26                 s3.download_fileobj(bucket, key, data)
27                 data.close()
28             except botocore.exceptions.ClientError as error:
```

# Lambda layers



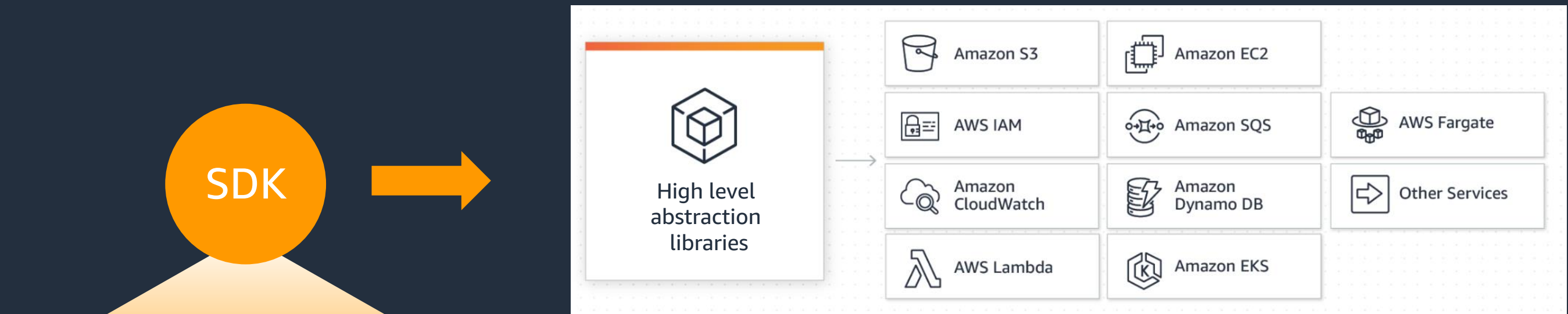
Lets functions easily share code; upload layer once, reference within any function

Layer can be anything: libraries, training data, configuration files, etc.

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

Built-in support for secure sharing by ecosystem

# AWS SDKs simplify the use of AWS services



JavaScript, Python, Java, .NET,  
TypeScript, Ruby, PHP, Go,  
Node.js, C++.

- SDK Libraries: Higher level abstractions saving developers time to concentrate on logic rather than low level API calls.
- Best practices by default (e.g., retries, credential handling)

# Boto3: The AWS SDK for Python

Boto3 allows you to integrate your Python application, library, or script with AWS services, including Amazon S3, Amazon DynamoDB, and more

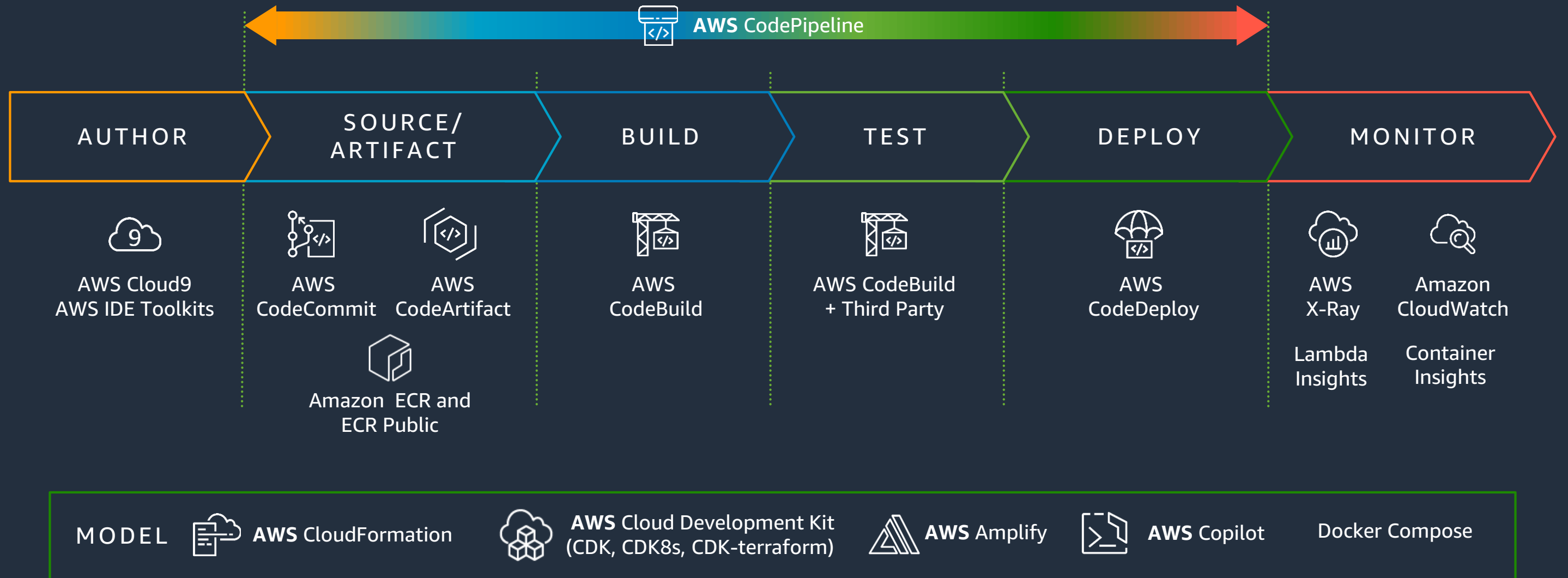
Boto3 provides native support to Python versions 2.7+ and 3.3+

```
import boto3

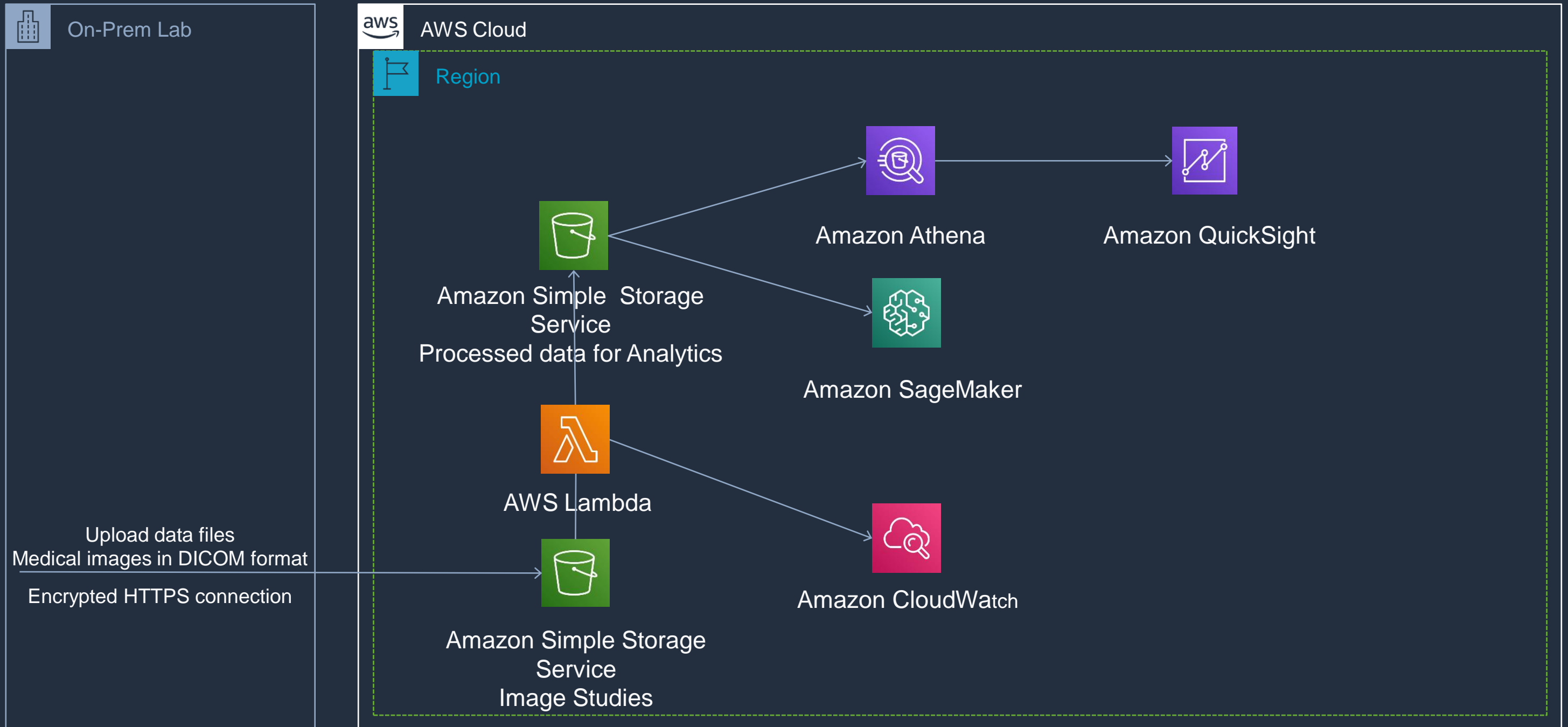
# List all DynamoDB tables
dynamodb = boto3.resource('dynamodb')
for table in dynamodb.tables.all():
    print(table.table_name)

# Create an S3 bucket and upload an object
import boto3
s3 = boto3.resource('s3')
s3.create_bucket(Bucket='mybucket')
s3.Object('mybucket', 'hello.txt') \
    .put(Body=open('/tmp/hello.txt', 'rb'))
```

# Automate deployment with AWS developer tools



# Processing Medical Images with Lambda in Python



# Getting Started

```
def lambda_handler(event, context):  
    for record in event['Records']:  
        bucket = record['s3']['bucket']['name']  
        key = record['s3']['object']['key']  
        with open("/tmp/"+key, 'wb') as data:  
            s3.download_fileobj(bucket, key, data)
```

- AWS Lambda function handler
- Event Object contains bucket name and key
- Lambda provides file system with /tmp directory
  - 500MB limit

# Roles and permissions

- By default Lambda function cannot access any resources
- We must create a role and attach permissions to the role:
  - Permission to read from the source S3 bucket
  - Permission to write to the output bucket
  - Permission to write logs to Amazon CloudWatch

```
1  {
2      "Version": "2012-10-17",
3      "Statement": [
4          {
5              "Sid": "VisualEditor0",
6              "Effect": "Allow",
7              "Action": "s3:GetObject",
8              "Resource": "arn:aws:s3:::my-bucket/*"
9          }
10     ]
11 }
```

# Process data

Use Python libraries to process files:

```
# Data Object with attributes from study
studyData = {}
dcm_data = pydicom.filereader.dcmread("/tmp/"+filename, stop_before_pixels=True)
studyData['Modality'] = dcm_data["Modality"].value
.....
```

Write results to the output S3 bucket:

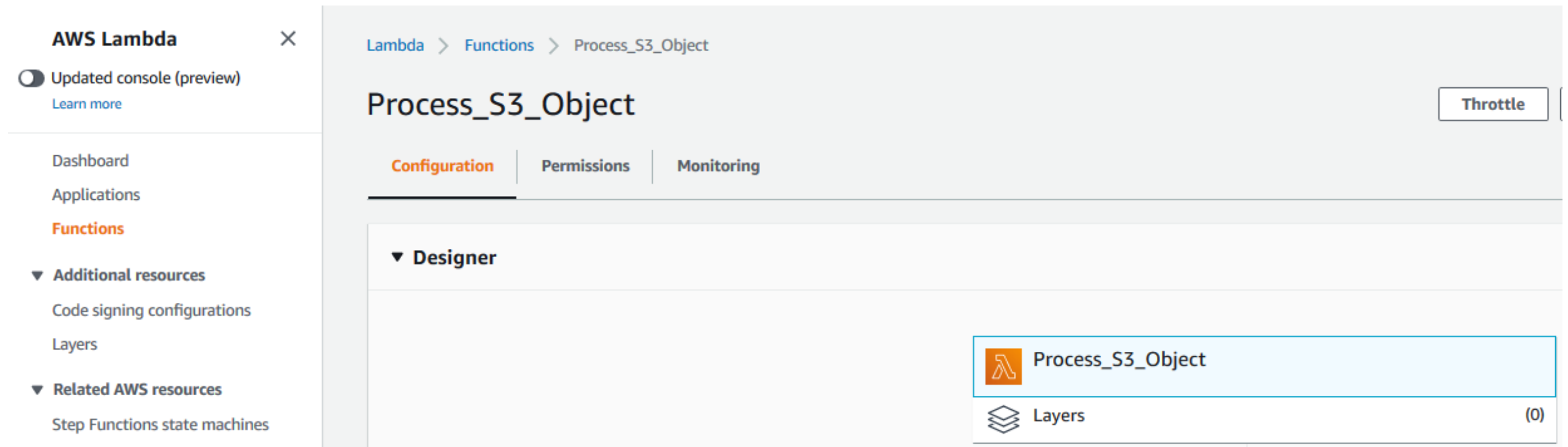
```
# Convert dictionary to JSON
json_data = json.dumps(studyData)

# Put JSON object in the output bucket
s3.put_object(
    Body=str(json_data),
    Bucket=metadata_bucket,
    Key = jsonKey
)
```

# Upload Python package in Layers

```
mkdir -p lambda-layer/python/lib/python3.8/site-packages
pip3 install pydicom --target lambda-layer/python/lib/python3.8/site-packages
cd lambda-layer
zip -r9 lambda-layer.zip .
```

## From the AWS Console:



The screenshot shows the AWS Lambda console interface. On the left is a navigation sidebar with 'AWS Lambda' at the top, followed by 'Updated console (preview)' and 'Learn more'. Below are 'Dashboard', 'Applications', and 'Functions' (highlighted). Under 'Additional resources' are 'Code signing configurations' and 'Layers'. Under 'Related AWS resources' is 'Step Functions state machines'. The main content area shows the breadcrumb 'Lambda > Functions > Process\_S3\_Object' and the function name 'Process\_S3\_Object' with a 'Throttle' button. Below the name are tabs for 'Configuration', 'Permissions', and 'Monitoring'. The 'Designer' section is expanded, showing a list with 'Process\_S3\_Object' and 'Layers (0)'.



# QuickSight interactive analytics

