CLICK TO ADD TEXT

ENT332

AWS re:INVENT

Getting Started with Serverless Computing Using AWS Lambda

Chris Munns – Serverless Senior Developer Advocate – AWS
Nicky Joshi – Director Software Engineering – Capital One

November 30, 2017

AWS re:Invent
aws

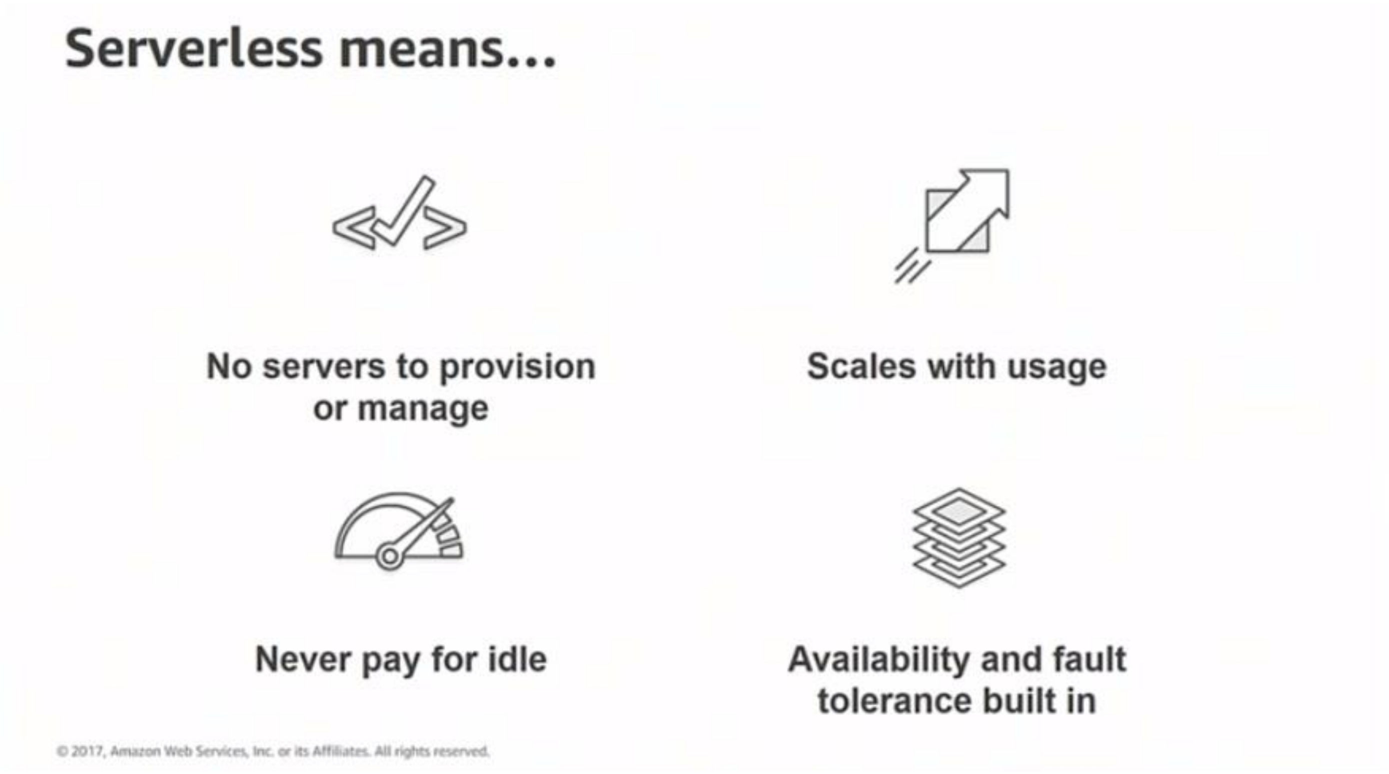
About me

Chris Munns – munns@amazon.com, @chrismunns

- Senior Developer Advocate — Serverless
- New Yorker
- Previously:
  - AWS Business Development Manager – DevOps, July '15–Feb. '17
  - AWS Solutions Architect Nov. '11–Dec. '14
  - Formerly on operations teams @Etsy and @Meetup
  - Little time at a hedge fund, Xerox, and a few other startups
- Rochester Institute of Technology: Applied Networking and Systems Administration '05
- Internet infrastructure geek

Why are we here today?

https://secure.flickr.com/photos/mgifford/4525333972



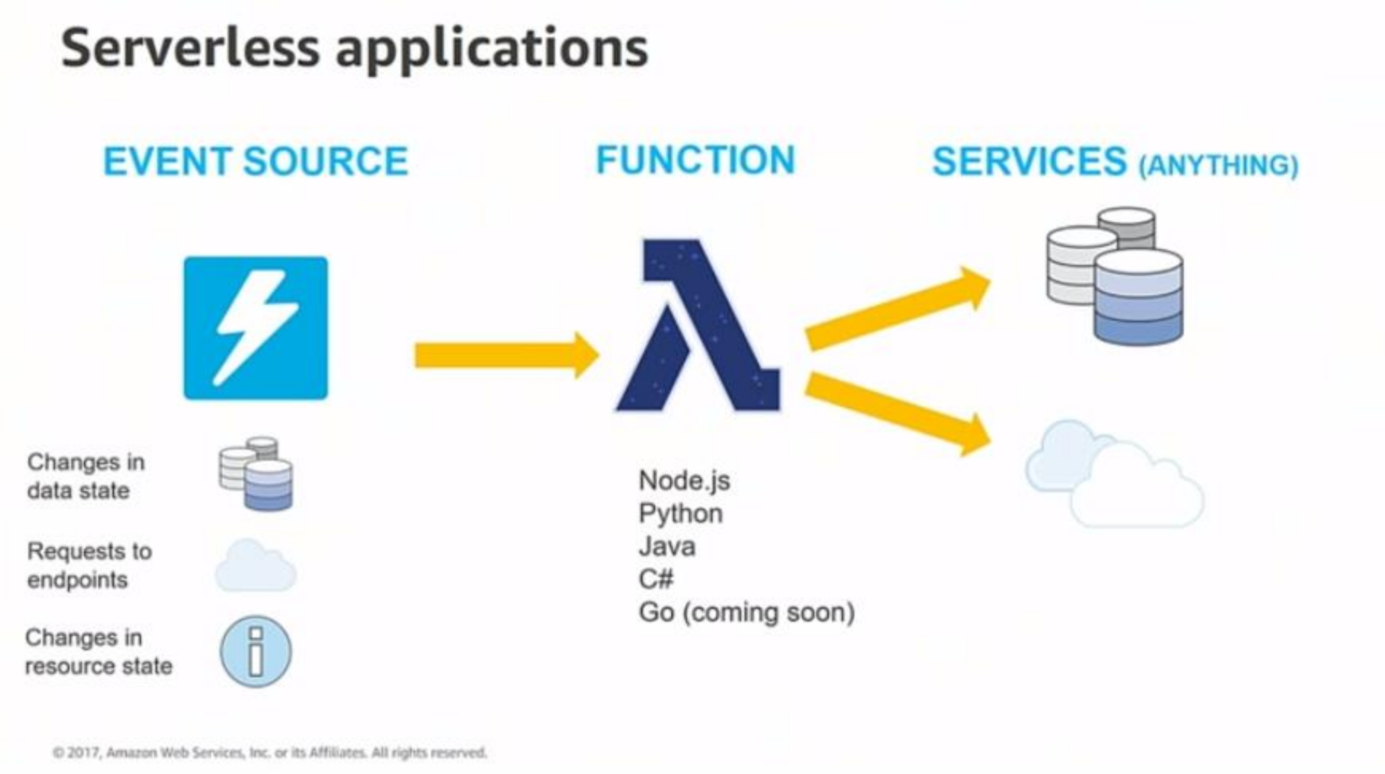
Serverless means…

No servers to provision or manage

Scales with usage

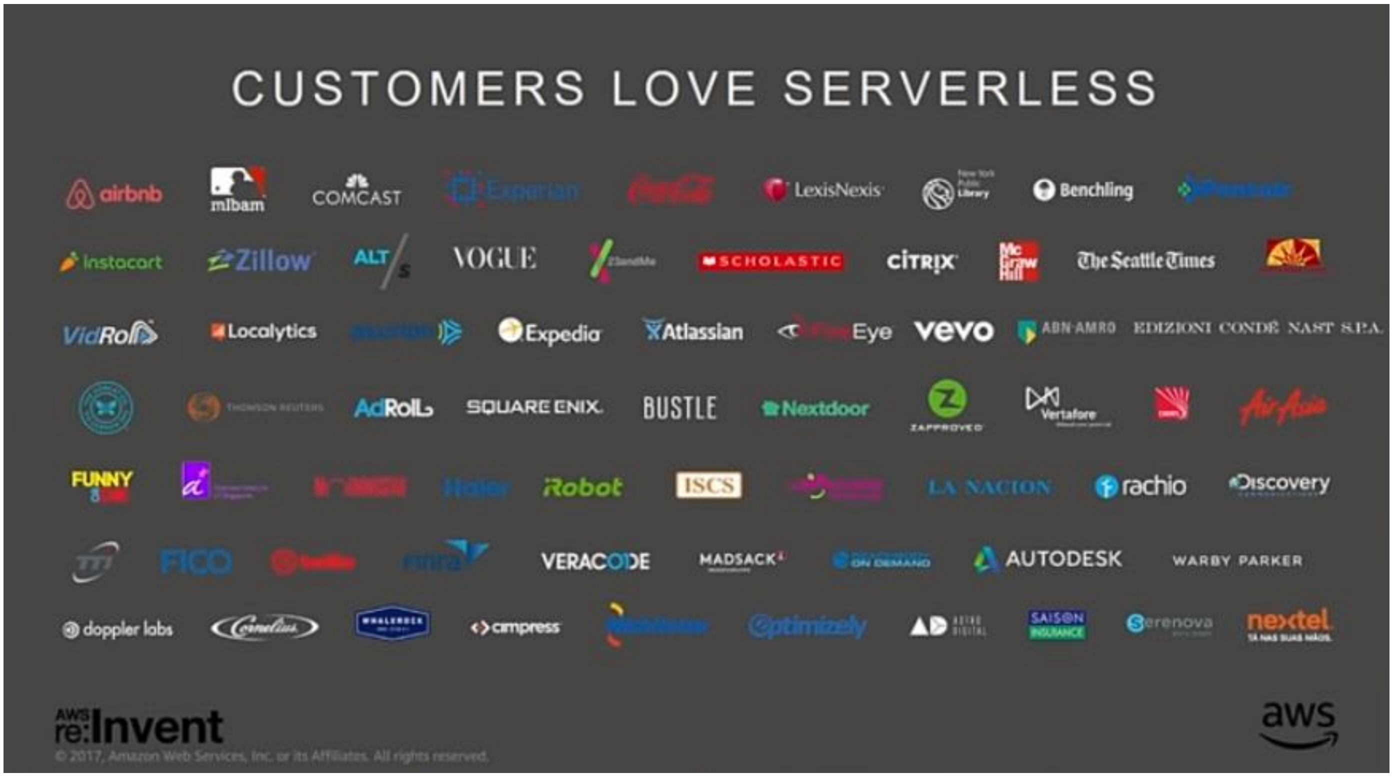
Never pay for idle

Availability and fault tolerance built in

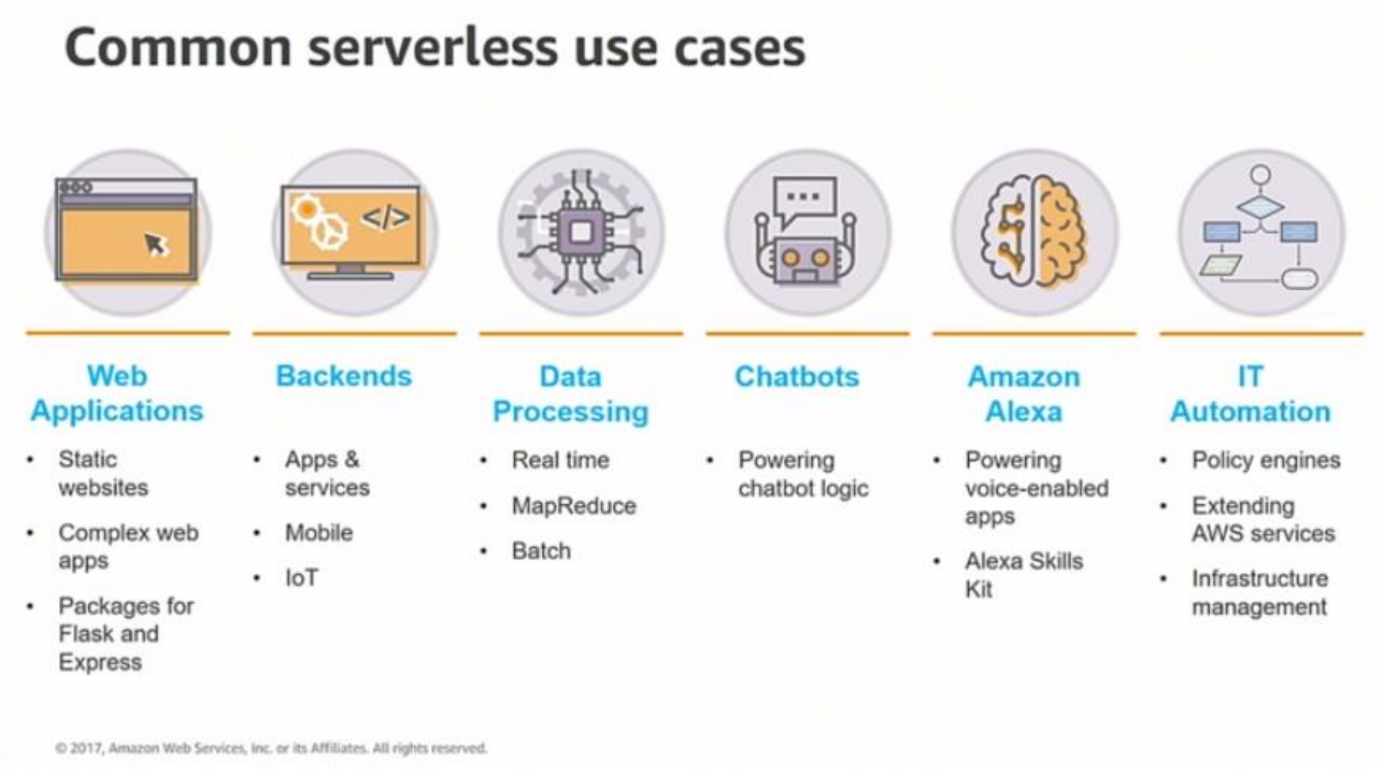


Serverless applications

Node.js
Python
Java
C#
Go (coming soon)



CUSTOMERS LOVE SERVERLESS



Common serverless use cases

Web Applications

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

Backends

- Apps & services
- Mobile
- IoT

Data Processing

- Real time
- MapReduce
- Batch

Chatbots

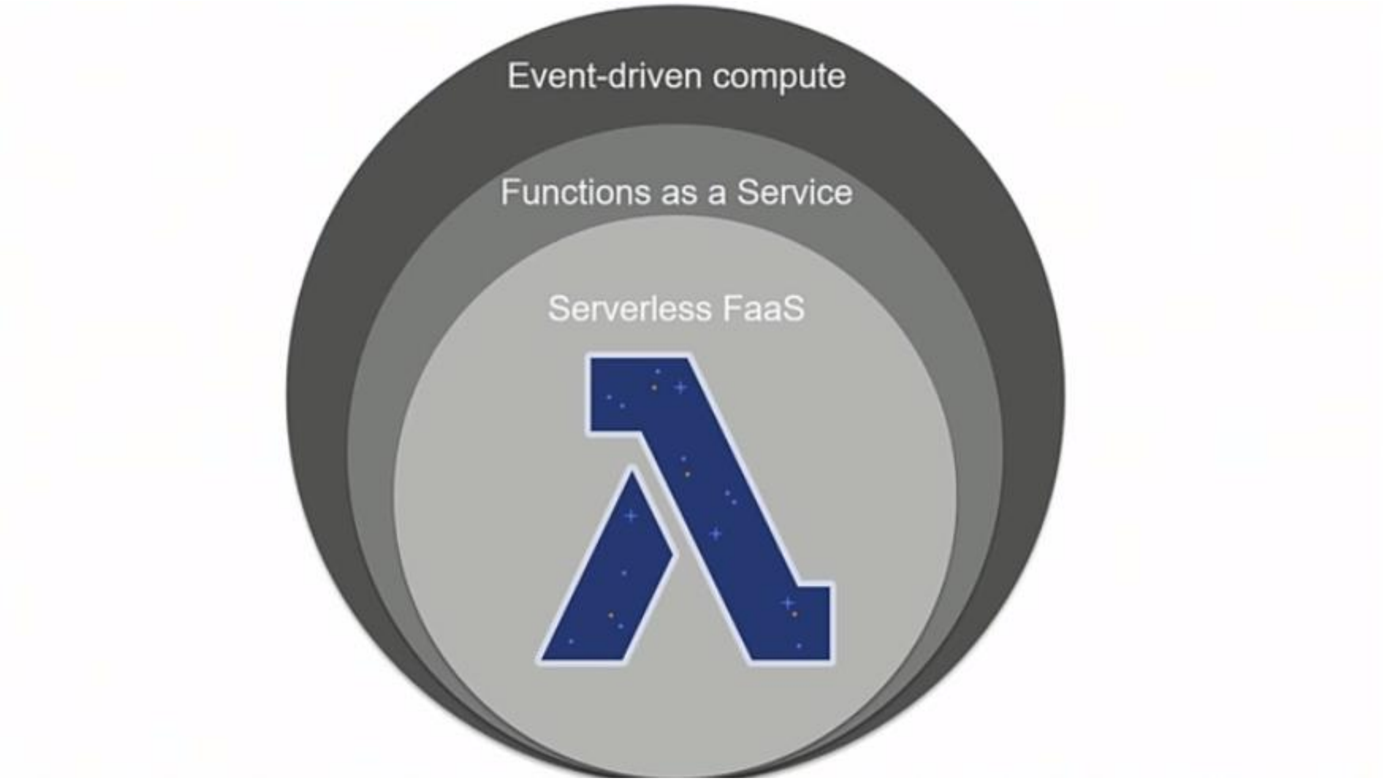
- Powering chatbot logic

Amazon Alexa

- Powering voice-enabled apps
- Alexa Skills Kit

IT Automation

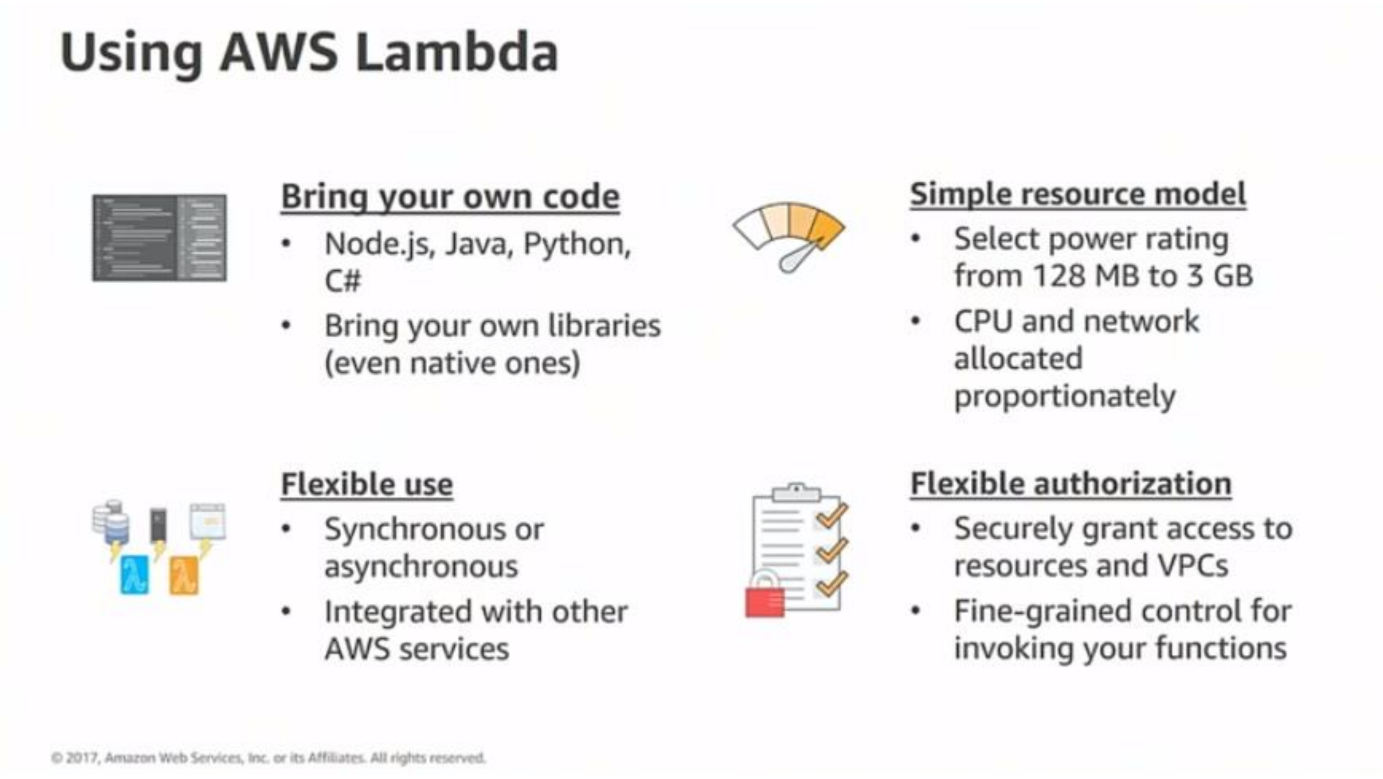
- Policy engines
- Extending AWS services
- Infrastructure management



Event-driven compute

Functions as a Service

Serverless FaaS



Using AWS Lambda

Bring your own code

- Node.js, Java, Python, C#
- Bring your own libraries (even native ones)

Simple resource model

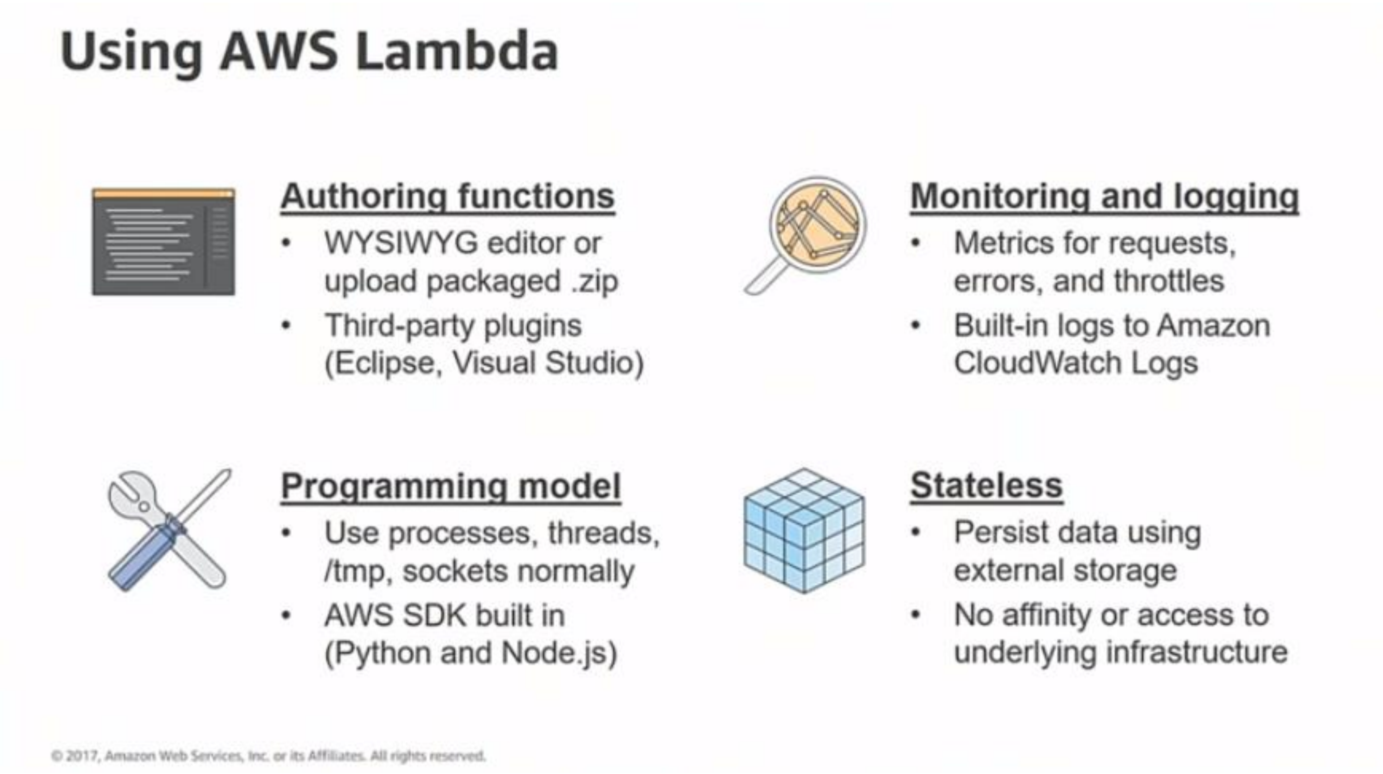
- Select power rating from 128 MB to 3 GB
- CPU and network allocated proportionately

Flexible use

- Synchronous or asynchronous
- Integrated with other AWS services

Flexible authorization

- Securely grant access to resources and VPCs
- Fine-grained control for invoking your functions



Using AWS Lambda

Authoring functions

- WYSIWYG editor or upload packaged .zip
- Third-party plugins (Eclipse, Visual Studio)

Monitoring and logging

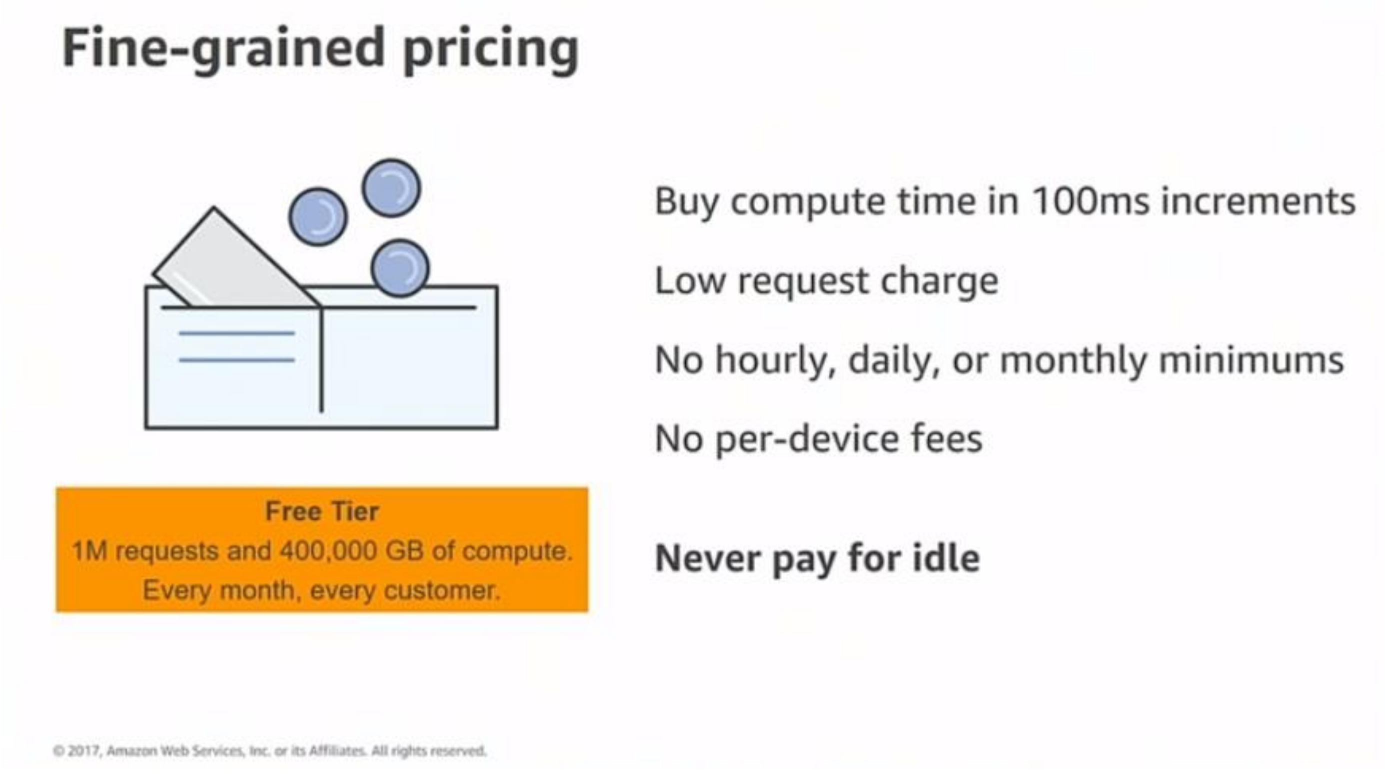
- Metrics for requests, errors, and throttles
- Built-in logs to Amazon CloudWatch Logs

Programming model

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

Stateless

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



Fine-grained pricing

Buy compute time in 100ms increments

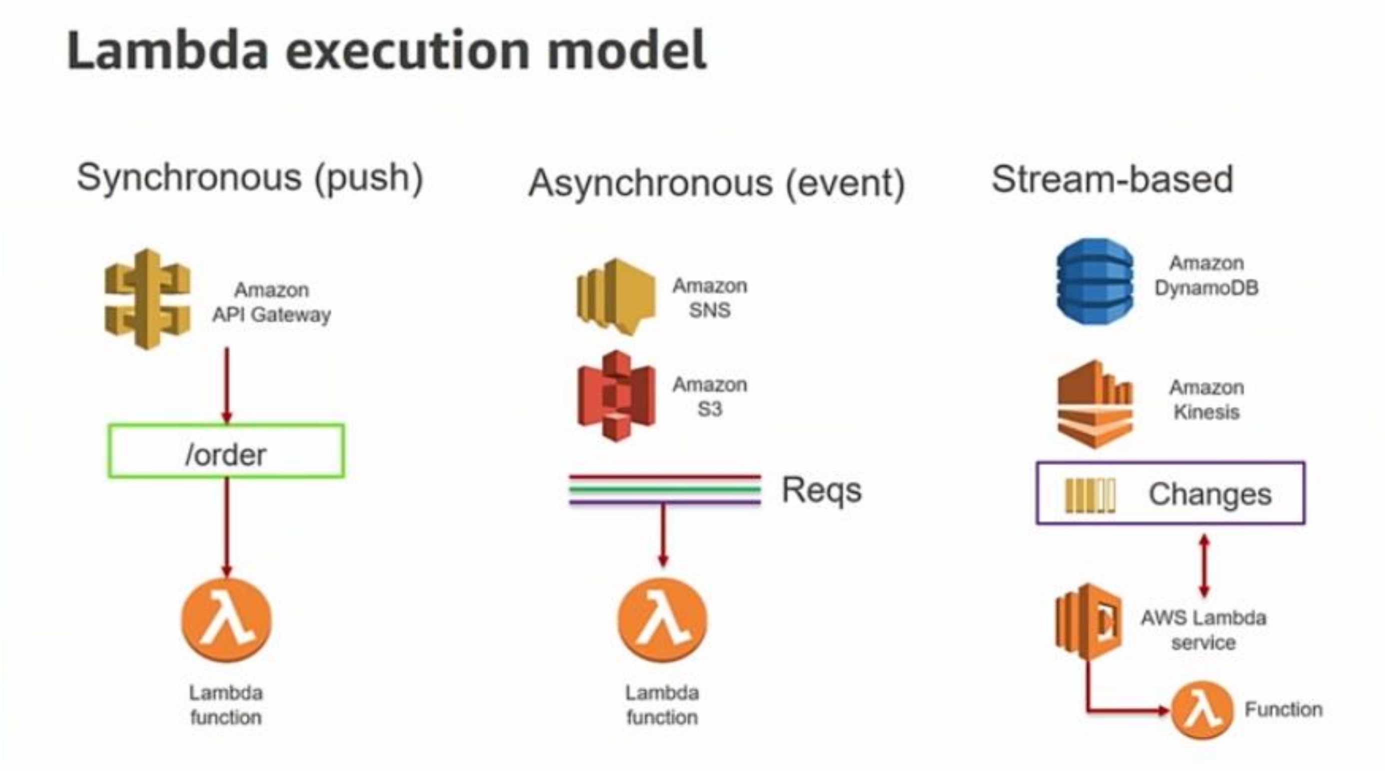
Low request charge

No hourly, daily, or monthly minimums

No per-device fees

Never pay for idle

Free Tier
1M requests and 400,000 GB of compute.
Every month, every customer.



Lambda execution model

Synchronous (push)

Amazon API Gateway

/order

Lambda function

Asynchronous (event)

Amazon SNS

Amazon S3

Reqs

Lambda function

Stream-based

Amazon DynamoDB

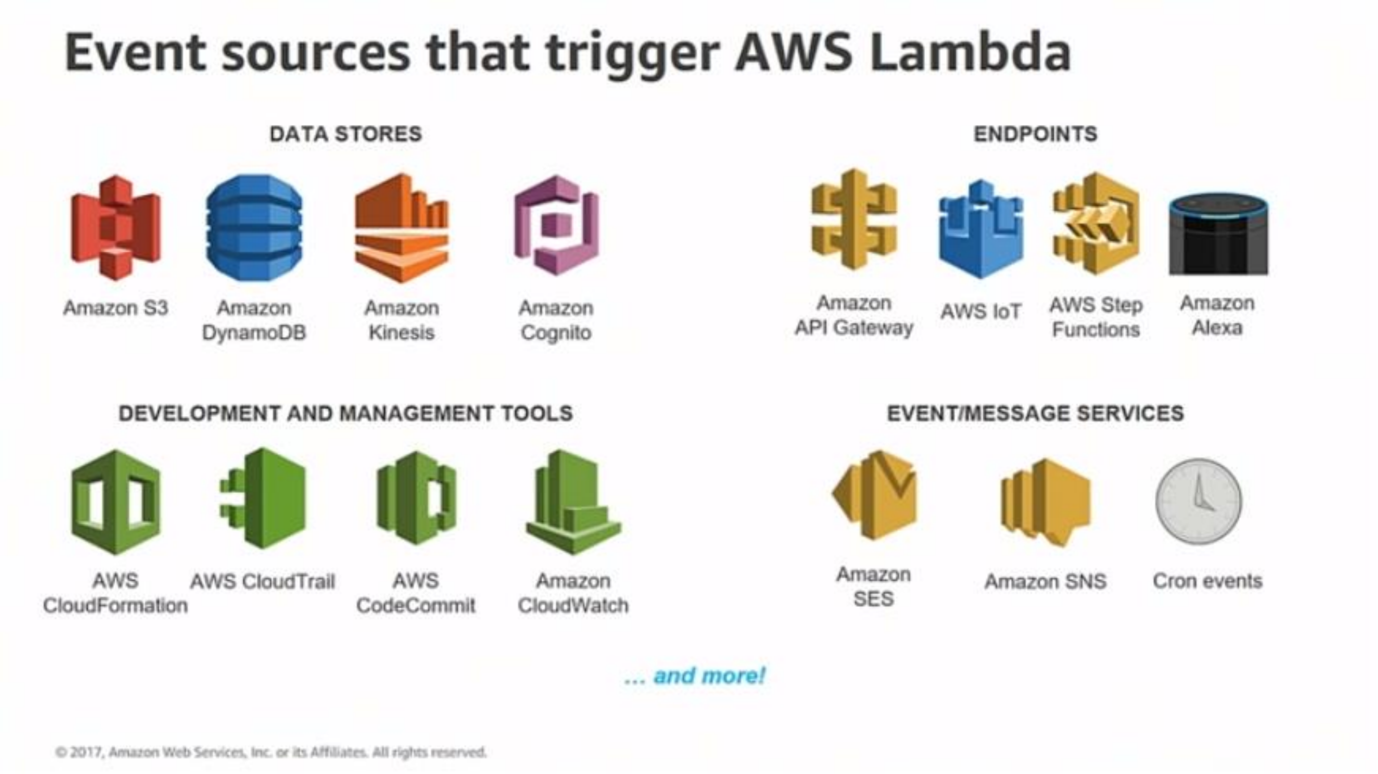
Amazon Kinesis

Changes

AWS Lambda service

Function

These are the 3 ways you can invoke your Lambda function.



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

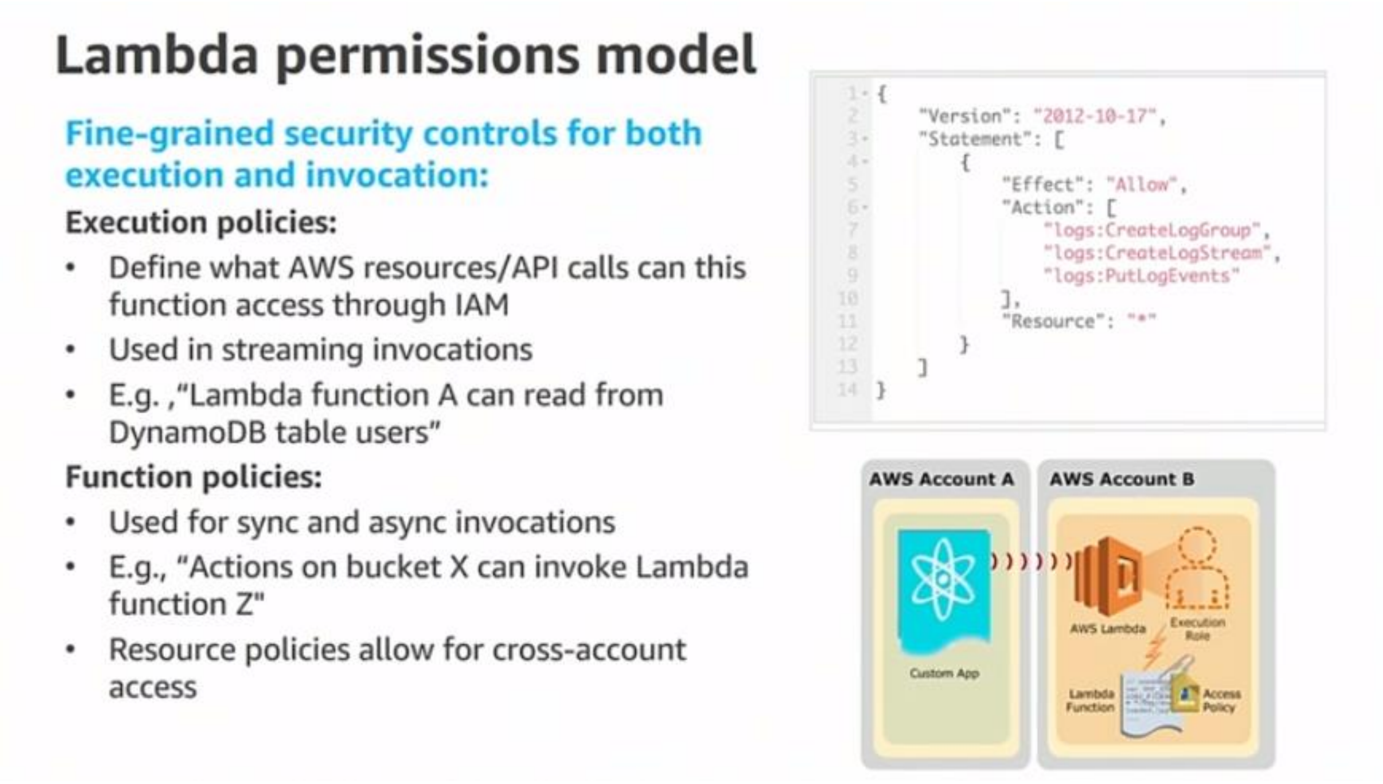
DEVELOPMENT AND MANAGEMENT TOOLS

AWS CloudFormation · AWS CloudTrail · AWS CodeCommit · Amazon CloudWatch

EVENT/MESSAGE SERVICES

Amazon SES · Amazon SNS · Cron events

... and more!



Lambda permissions model

Fine-grained security controls for both execution and invocation:

Execution policies:

- Define what AWS resources/API calls can this function access through IAM
- Used in streaming invocations
- E.g. ,"Lambda function A can read from DynamoDB table users"

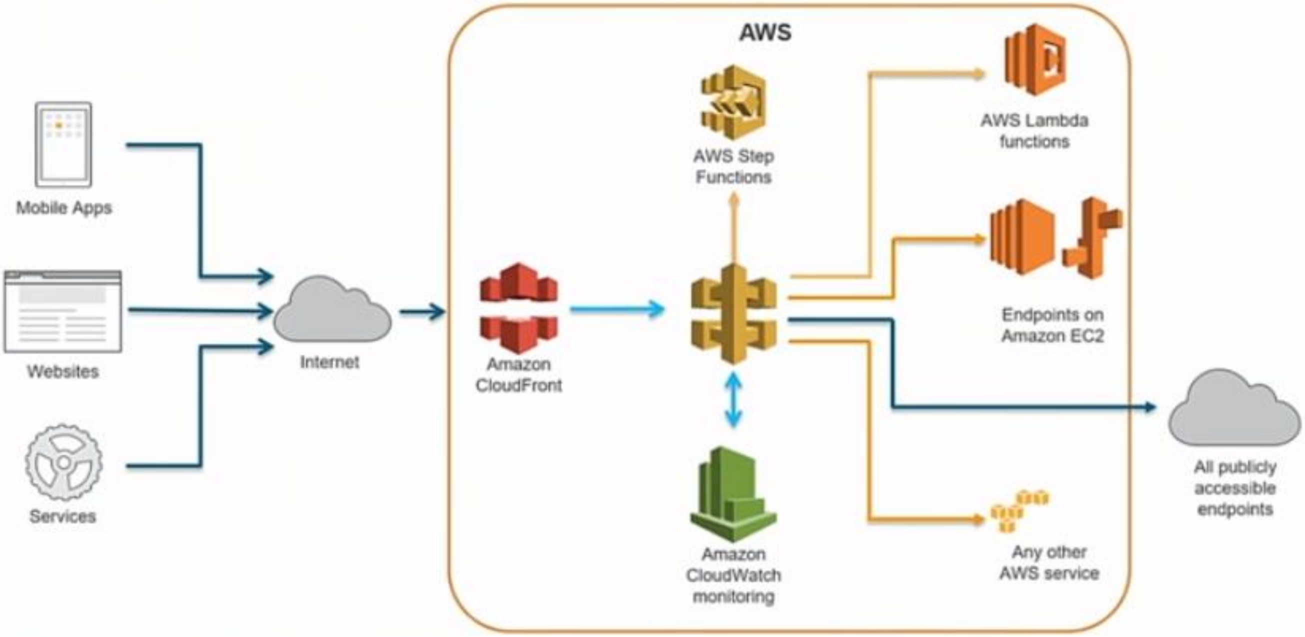
Function policies:

- Used for sync and async invocations
- E.g., "Actions on bucket X can invoke Lambda function Z"
- Resource policies allow for cross-account access

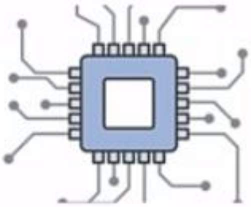
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

There are several ways to control use of your lambda function, the place where it sits like a VPC as well as its execution and operating policies.

Amazon API Gateway



Amazon API Gateway



Create a unified API front end for multiple micro-services

DDoS protection and throttling for your backend

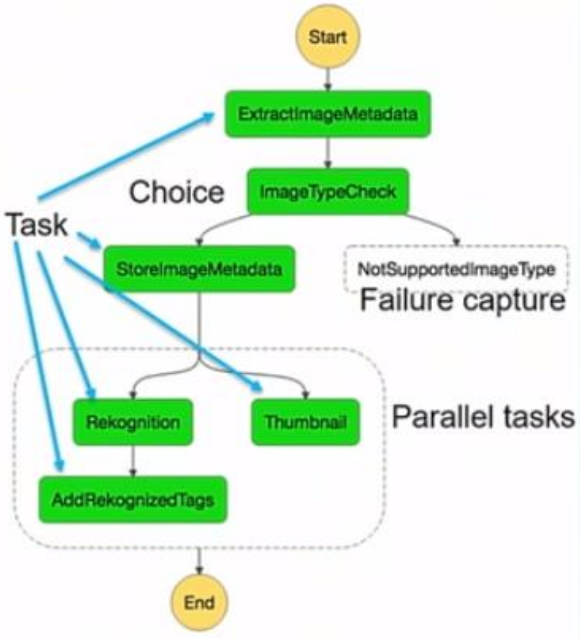
Authenticate and authorize requests to a backend

Throttle, meter, and monetize API usage by third-party developers

AWS Step Functions

"Serverless" workflow management with zero administration:

- Makes it easy to coordinate the components of distributed applications and microservices using visual workflows
- Automatically triggers and tracks each step, and retries when there are errors, so your application executes in order and as expected
- Logs the state of each step, so when things do go wrong, you can diagnose and debug problems quickly



Amazon Lex

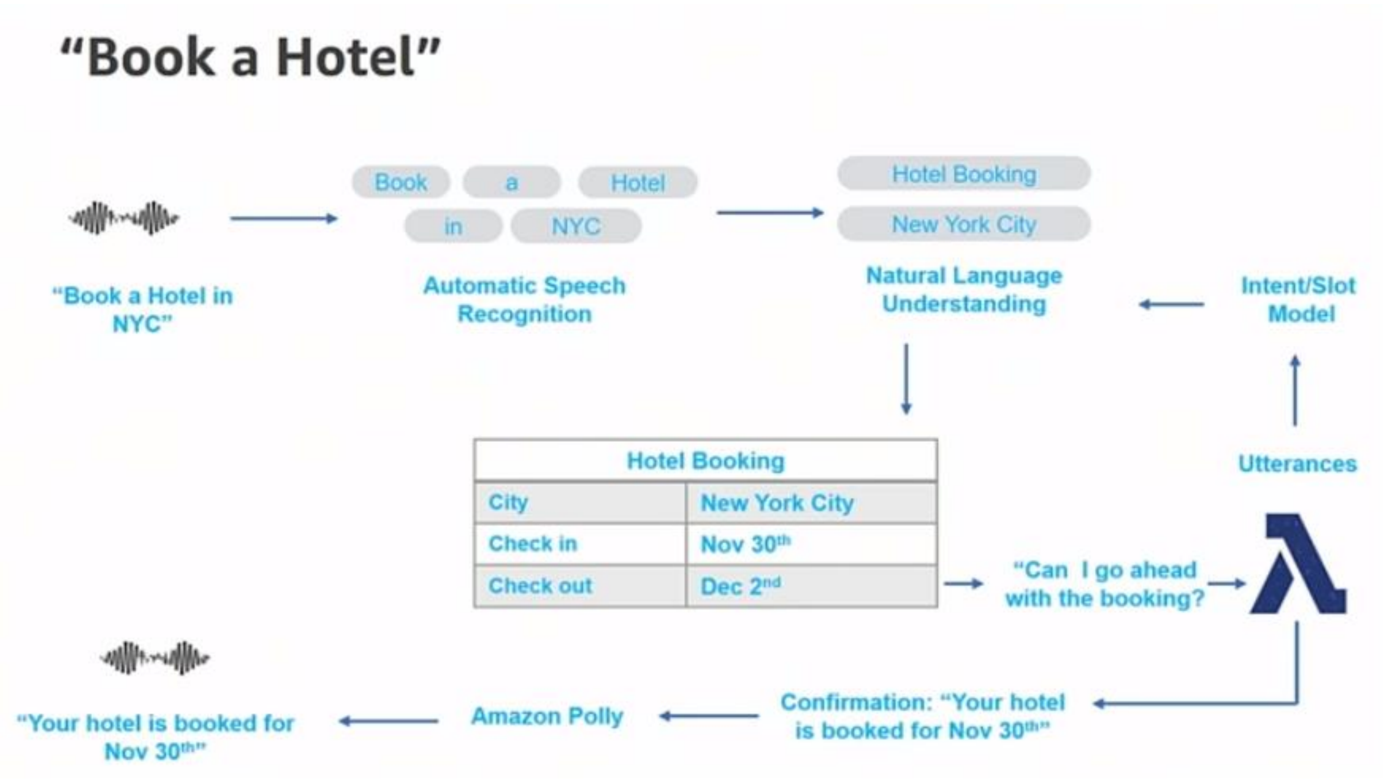
Service for building conversational interfaces into any application using voice and text

Automatic speech recognition (ASR) for converting speech to text

Natural language understanding (NLU) to recognize the intent of messages

Powered by the same deep learning technology as Alexa

Fully managed service



"Book a Hotel"

"Book a Hotel in NYC"

Book a Hotel in NYC

Automatic Speech Recognition

Hotel Booking

New York City

Natural Language Understanding

Intent/Slot Model

Utterances

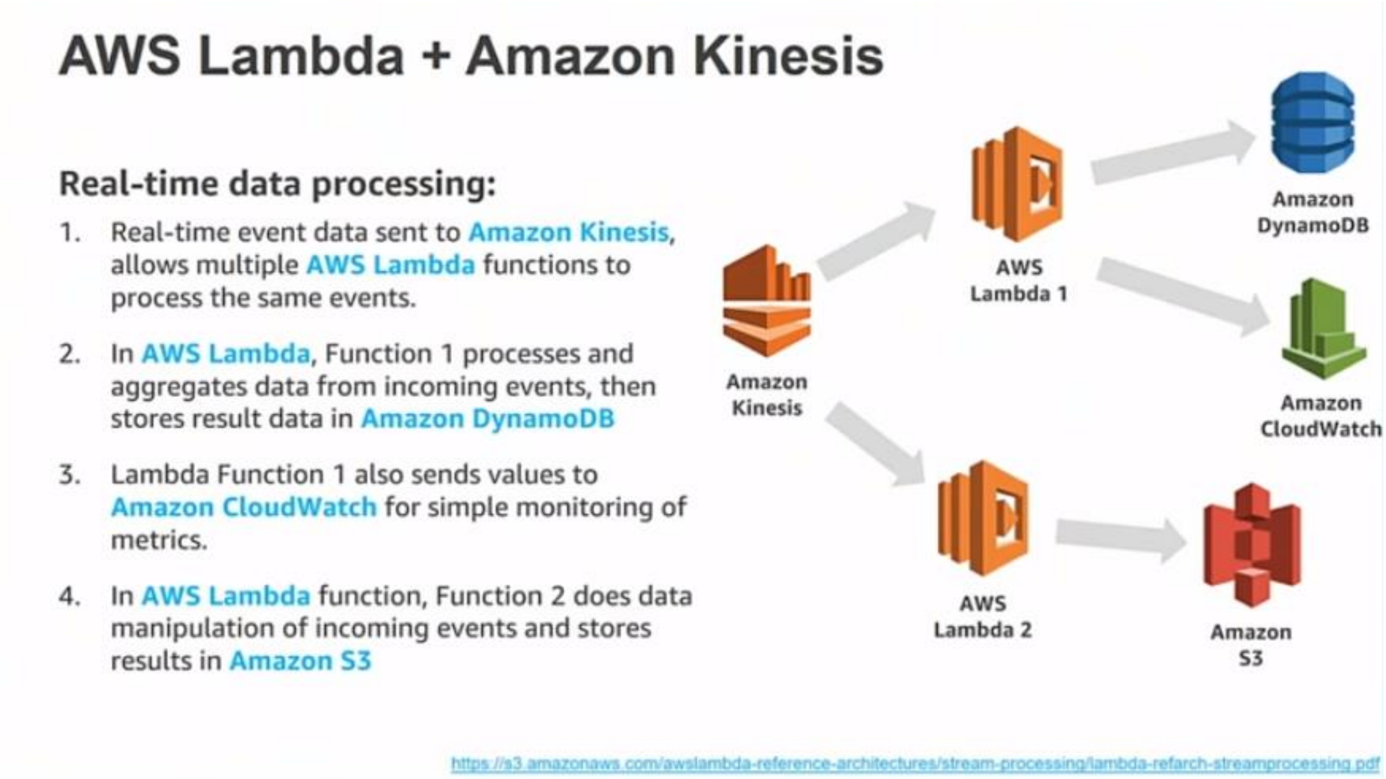
Hotel Booking	
City	New York City
Check in	Nov 30th
Check out	Dec 2nd

"Can I go ahead with the booking?

Confirmation: "Your hotel is booked for Nov 30th"

Amazon Polly

"Your hotel is booked for Nov 30th"



AWS Lambda + Amazon Kinesis

Real-time data processing:

1. Real-time event data sent to **Amazon Kinesis**, allows multiple **AWS Lambda** functions to process the same events.
2. In **AWS Lambda**, Function 1 processes and aggregates data from incoming events, then stores result data in **Amazon DynamoDB**
3. Lambda Function 1 also sends values to **Amazon CloudWatch** for simple monitoring of metrics.
4. In **AWS Lambda** function, Function 2 does data manipulation of incoming events and stores results in **Amazon S3**

https://s3.amazonaws.com/awslambda-reference-architectures/stream-processing/lambda-refarch-streamprocessing.pdf

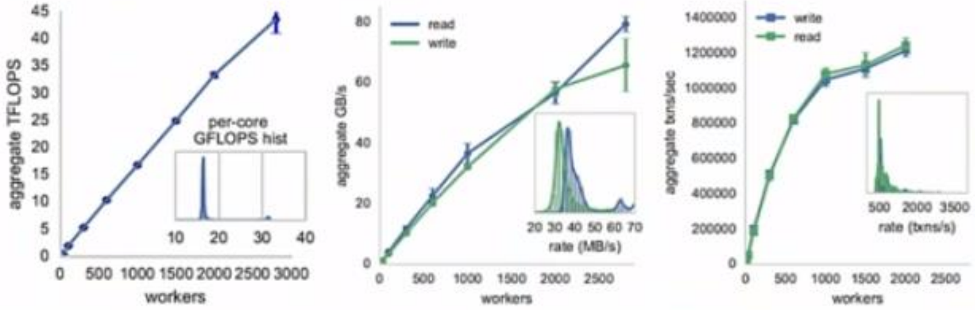
Serverless distributed computing: PyWren

PyWren Prototype developed at University of California, Berkeley

Uses Python with AWS Lambda stateless functions for large-scale data analytics

Achieved @ 30-40 MB write and read performance per-core to Amazon S3 object store

Scaled to 60-80 GB across 2,800 simultaneous functions



Build PCI and HIPAA-compliant serverless applications!

HIPAA

PCi

Serverless platform services that can be used in both:

AWS Lambda

Amazon S3

Amazon CloudFront

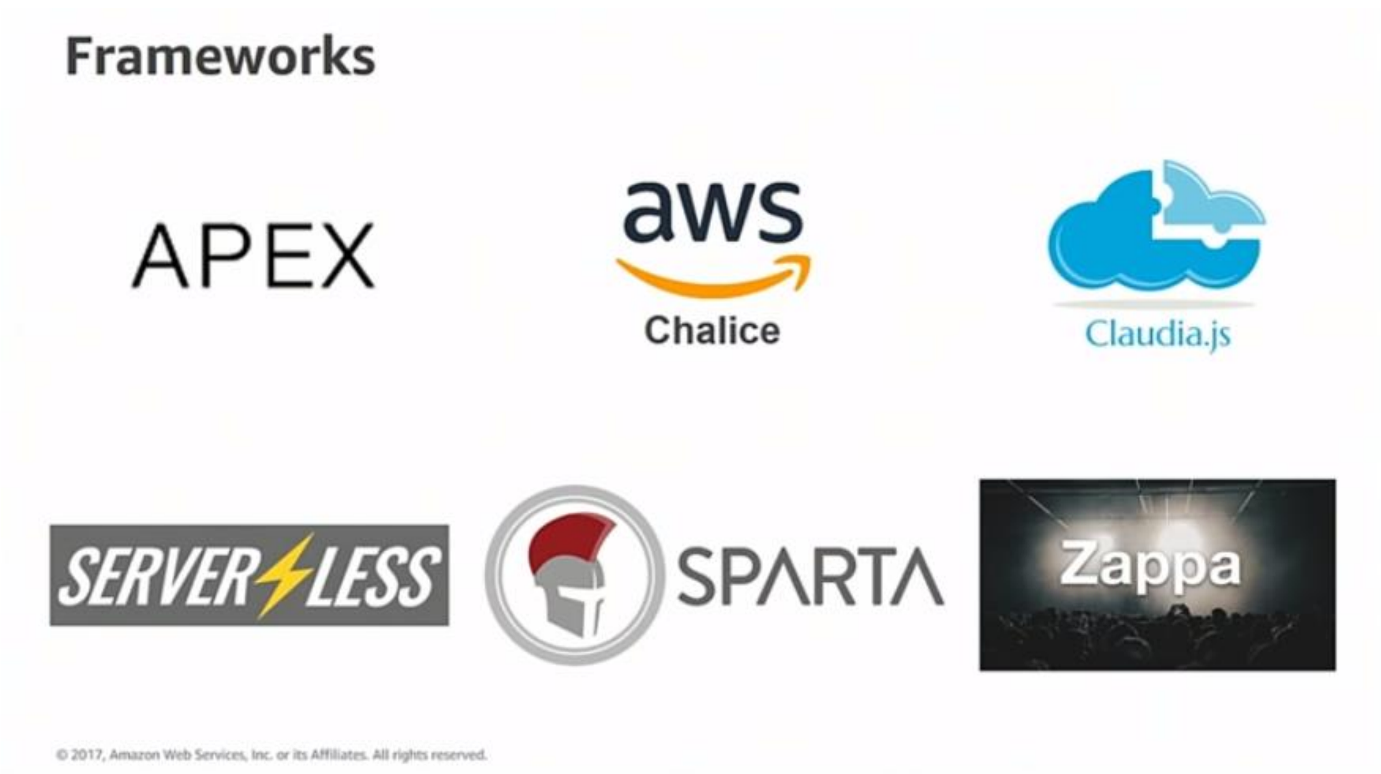
Amazon DynamoDB

Amazon Kinesis Streams

Amazon Cognito

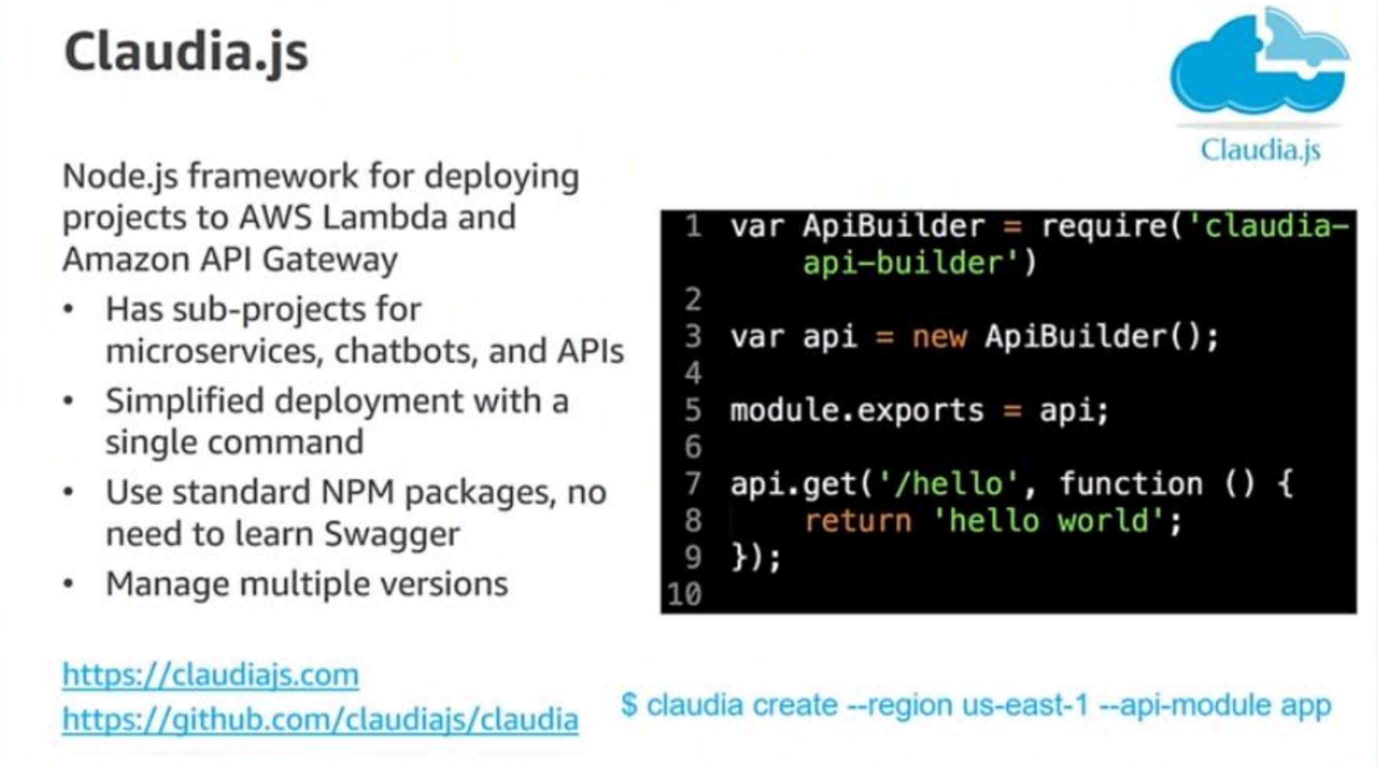
Amazon API Gateway

Amazon SNS



Frameworks

Find a framework that fits the need you are looking to feel.



Claudia.js

Claudia.js

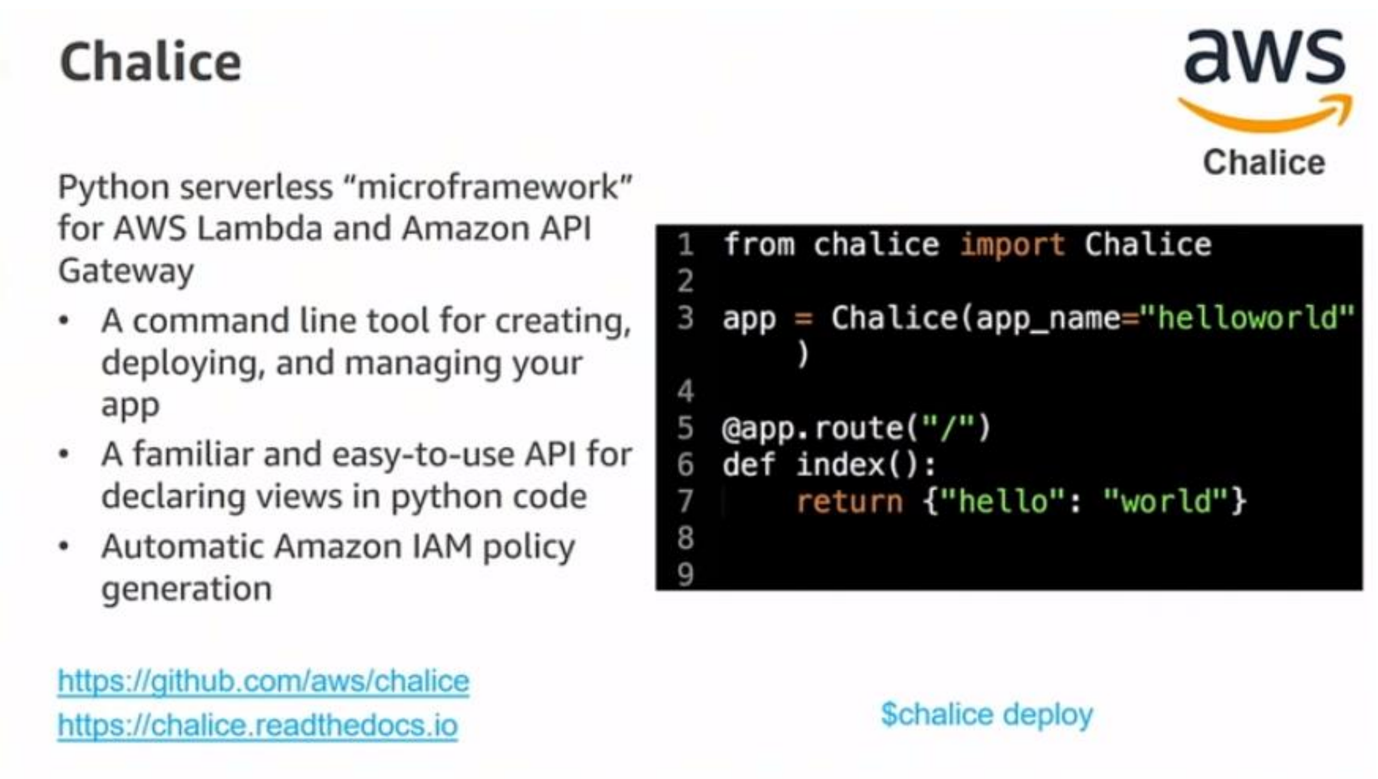
Node.js framework for deploying projects to AWS Lambda and Amazon API Gateway

- Has sub-projects for microservices, chatbots, and APIs
- Simplified deployment with a single command
- Use standard NPM packages, no need to learn Swagger
- Manage multiple versions

```javascript
var ApiBuilder = require('claudia-
    api-builder')

var api = new ApiBuilder();

module.exports = api;

api.get('/hello', function () {
    return 'hello world';
});

```

https://claudiajs.com
https://github.com/claudiajs/claudia

```
$ claudia create --region us-east-1 --api-module app
```



Chalice

aws
Chalice

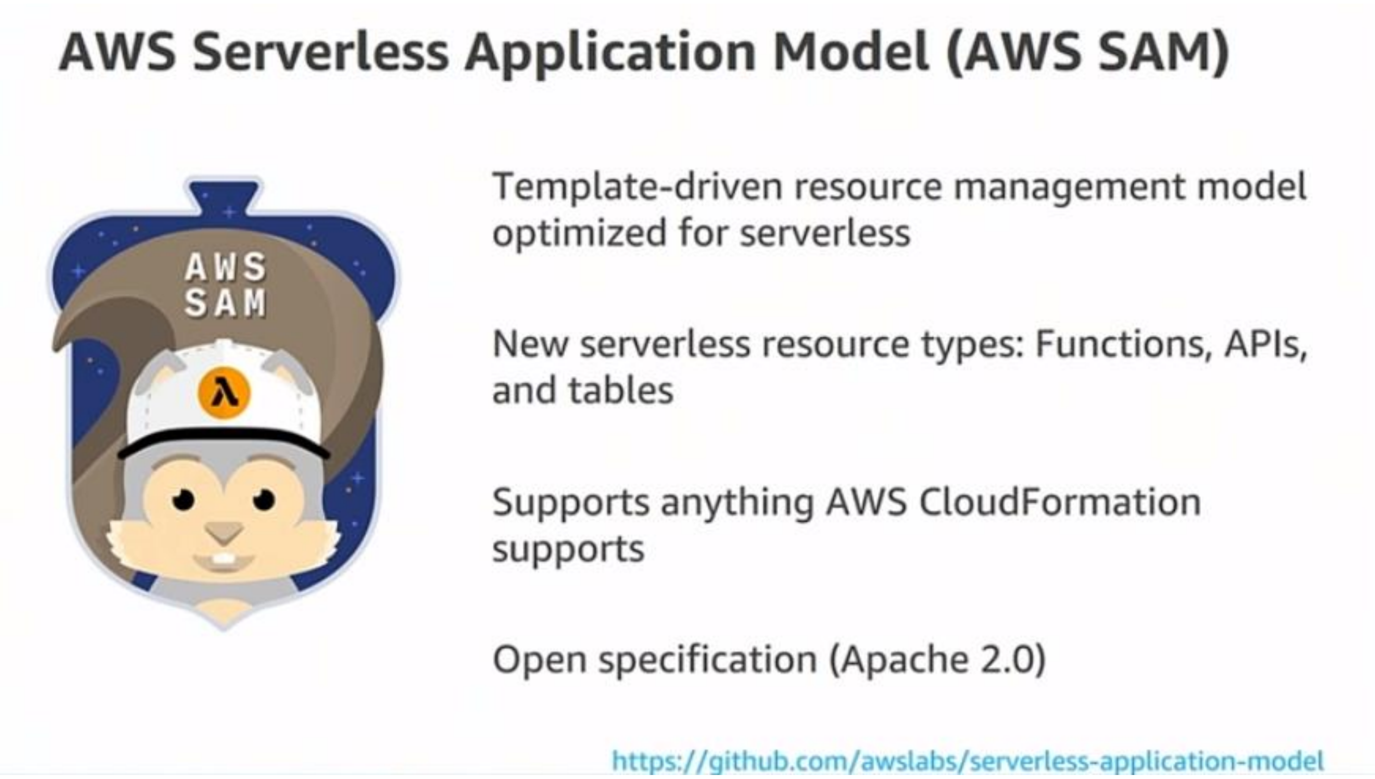
Python serverless "microframework" for AWS Lambda and Amazon API Gateway

- A command line tool for creating, deploying, and managing your app
- A familiar and easy-to-use API for declaring views in python code
- Automatic Amazon IAM policy generation

```python
from chalice import Chalice

app = Chalice(app_name="helloworld"
    )

@app.route("/")
def index():
    return {"hello": "world"}


```

https://github.com/aws/chalice
https://chalice.readthedocs.io

```
$chalice deploy
```



AWS Serverless Application Model (AWS SAM)

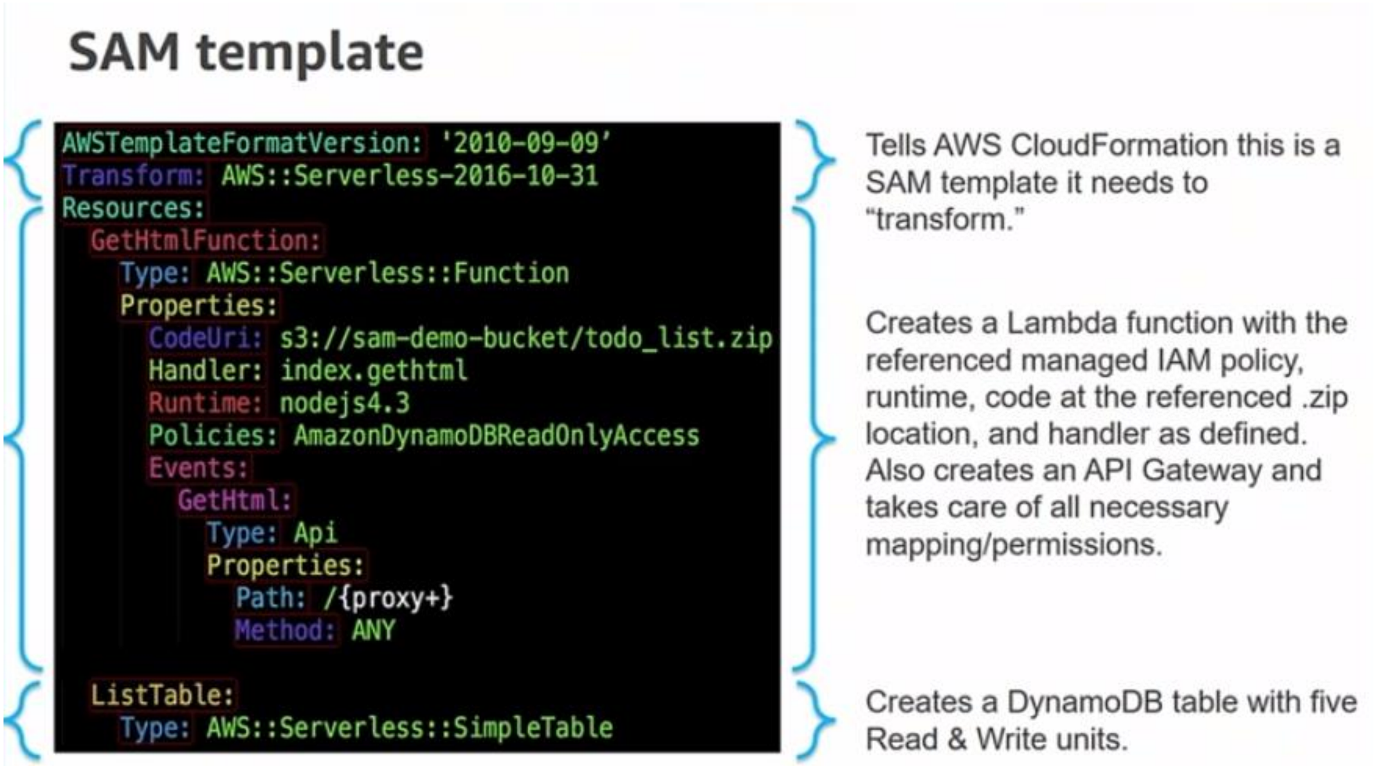
Template-driven resource management model optimized for serverless

New serverless resource types: Functions, APIs, and tables

Supports anything AWS CloudFormation supports

Open specification (Apache 2.0)

https://github.com/awslabs/serverless-application-model



SAM template

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  GetHtmlFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://sam-demo-bucket/todo_list.zip
      Handler: index.gethtml
      Runtime: nodejs4.3
      Policies: AmazonDynamoDBReadOnlyAccess
      Events:
        GetHtml:
          Type: Api
          Properties:
            Path: /{proxy+}
            Method: ANY

  ListTable:
    Type: AWS::Serverless::SimpleTable
```

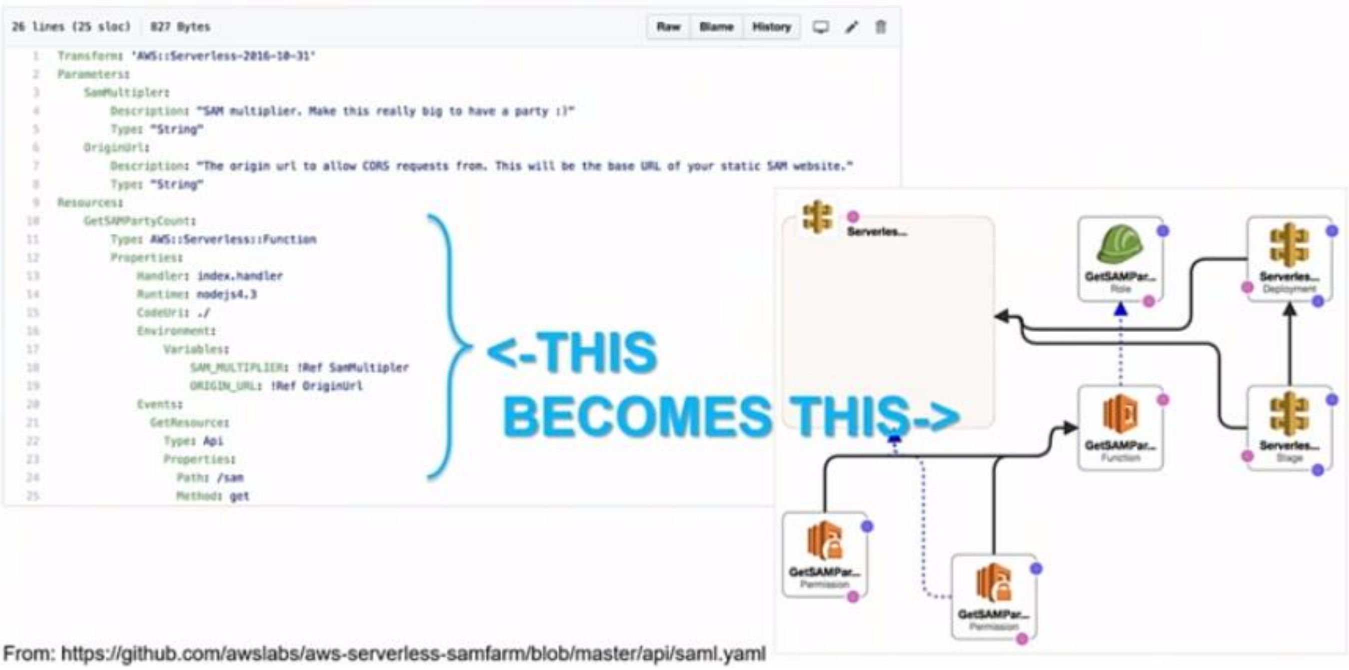
Tells AWS CloudFormation this is a SAM template it needs to "transform."

Creates a Lambda function with the referenced managed IAM policy, runtime, code at the referenced .zip location, and handler as defined. Also creates an API Gateway and takes care of all necessary mapping/permissions.

Creates a DynamoDB table with five Read & Write units.

SAM is all about managing the AWS resources,

SAM template



```
Transform: 'AWS::Serverless-2016-10-31'
Parameters:
    SamMultipler:
        Description: "SAM multiplier. Make this really big to have a party :)"
        Type: "String"
    OriginUrl:
        Description: "The origin url to allow CORS requests from. This will be the base URL of your static SAM website."
        Type: "String"
Resources:
    GetSAMPartyCount:
        Type: AWS::Serverless::Function
        Properties:
            Handler: index.handler
            Runtime: nodejs4.3
            CodeUri: ./
            Environment:
                Variables:
                    SAM_MULTIPLIER: !Ref SamMultipler
                    ORIGIN_URL: !Ref OriginUrl
            Events:
              GetResource:
                Type: Api
                Properties:
                  Path: /sam
                  Method: get
```

<-THIS
BECOMES THIS->

From: https://github.com/awslabs/aws-serverless-samfarm/blob/master/api/saml.yaml

This piece of code will generate the 6 different AWS resources on the right for you automatically. It is better than using the CLI to create these same resources.

Introducing SAM Local

CLI tool for local testing of serverless apps

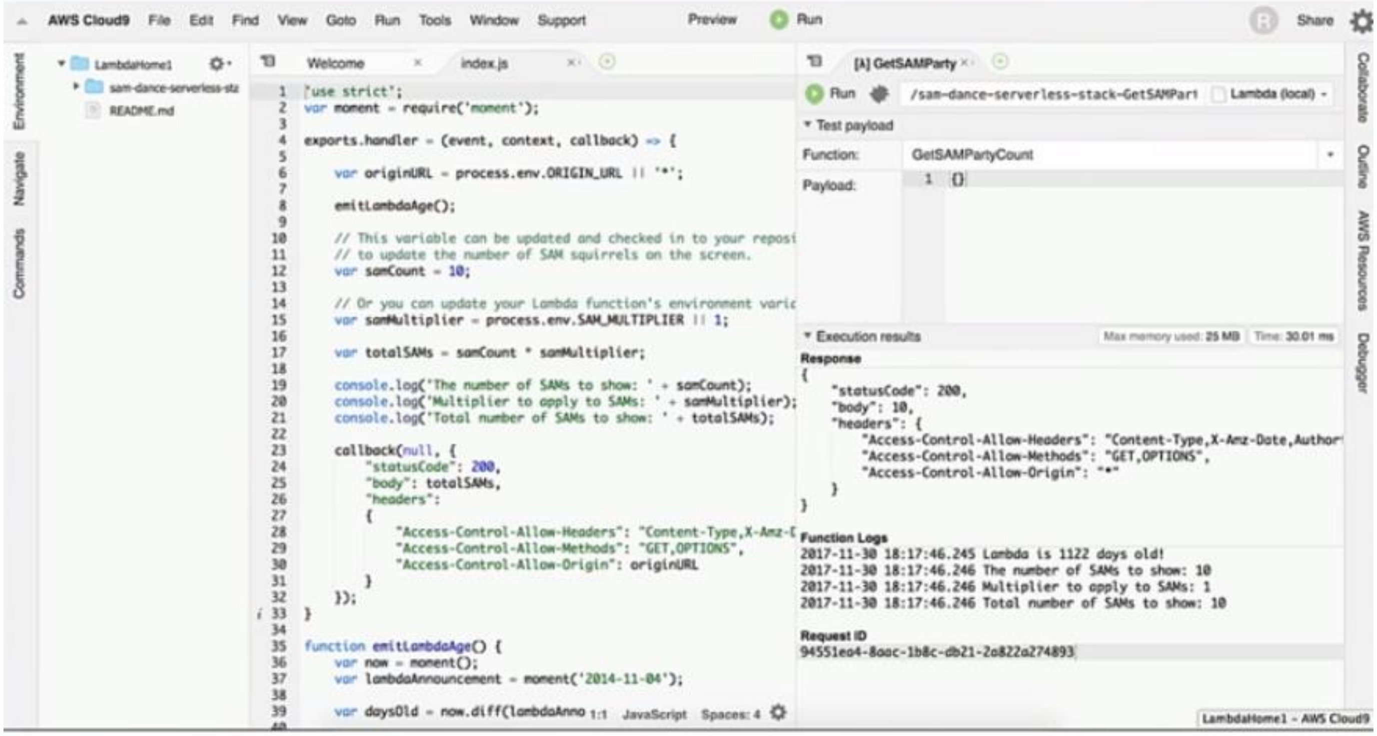
Works with Lambda functions and "proxy-style" APIs

Response object and function logs available on your local machine

Uses open-source docker-lambda images to mimic Lambda's execution environment:

- Emulates timeout, memory limits, runtimes

https://github.com/awslabs/aws-sam-local



```
'use strict';
var moment = require('moment');

exports.handler = (event, context, callback) => {

    var originURL = process.env.ORIGIN_URL || '*';

    emitLambdaAge();

    // This variable can be updated and checked in to your reposi
    // to update the number of SAM squirrels on the screen.
    var samCount = 10;

    // Or you can update your Lambda function's environment varic
    var samMultiplier = process.env.SAM_MULTIPLIER || 1;

    var totalSAMs = samCount * samMultiplier;

    console.log('The number of SAMs to show: ' + samCount);
    console.log('Multiplier to apply to SAMs: ' + samMultiplier);
    console.log('Total number of SAMs to show: ' + totalSAMs);

    callback(null, {
        "statusCode": 200,
        "body": totalSAMs,
        "headers":
        {
            "Access-Control-Allow-Headers": "Content-Type,X-Amz-[
            "Access-Control-Allow-Methods": "GET,OPTIONS",
            "Access-Control-Allow-Origin": originURL
        }
    });
}

function emitLambdaAge() {
    var now = moment();
    var lambdaAnnouncement = moment('2014-11-04');

    var daysOld = now.diff(lambdaAnno
```

/sam-dance-serverless-stack-GetSAMPart

▾ Test payload

Function: GetSAMPartyCount

Payload:

```
1 {}
```

▾ Execution results

Max memory used: 25 MB

Time: 30.01 ms

Response

```
{
    "statusCode": 200,
    "body": 10,
    "headers": {
        "Access-Control-Allow-Headers": "Content-Type,X-Amz-Date,Author
        "Access-Control-Allow-Methods": "GET,OPTIONS",
        "Access-Control-Allow-Origin": "*"
    }
}
```

Function Logs

```
2017-11-30 18:17:46.245 Lambda is 1122 days old!
2017-11-30 18:17:46.246 The number of SAMs to show: 10
2017-11-30 18:17:46.246 Multiplier to apply to SAMs: 1
2017-11-30 18:17:46.246 Total number of SAMs to show: 10
```

Request ID

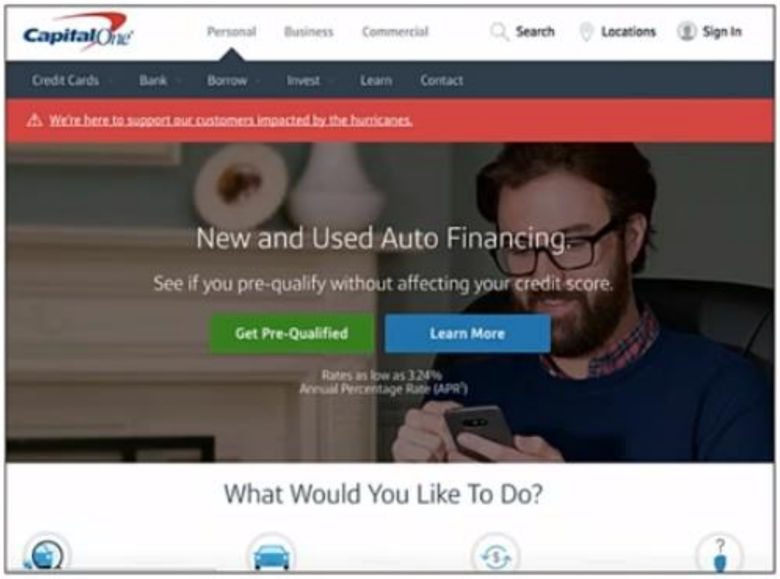
```
94551ea4-8aac-1b8c-db21-2a822a274893
```

Capital One Auto – Journey to Serverless

AWS re:Invent
aws

Let us see the way we migrated one of our marketing web applications to a Serverless architecture.

Capital One



We're here to support our customers impacted by the hurricanes.

New and Used Auto Financing.

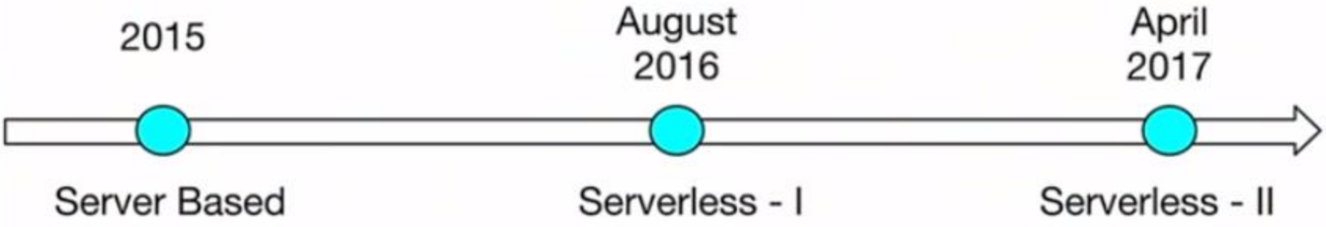
See if you pre-qualify without affecting your credit score.

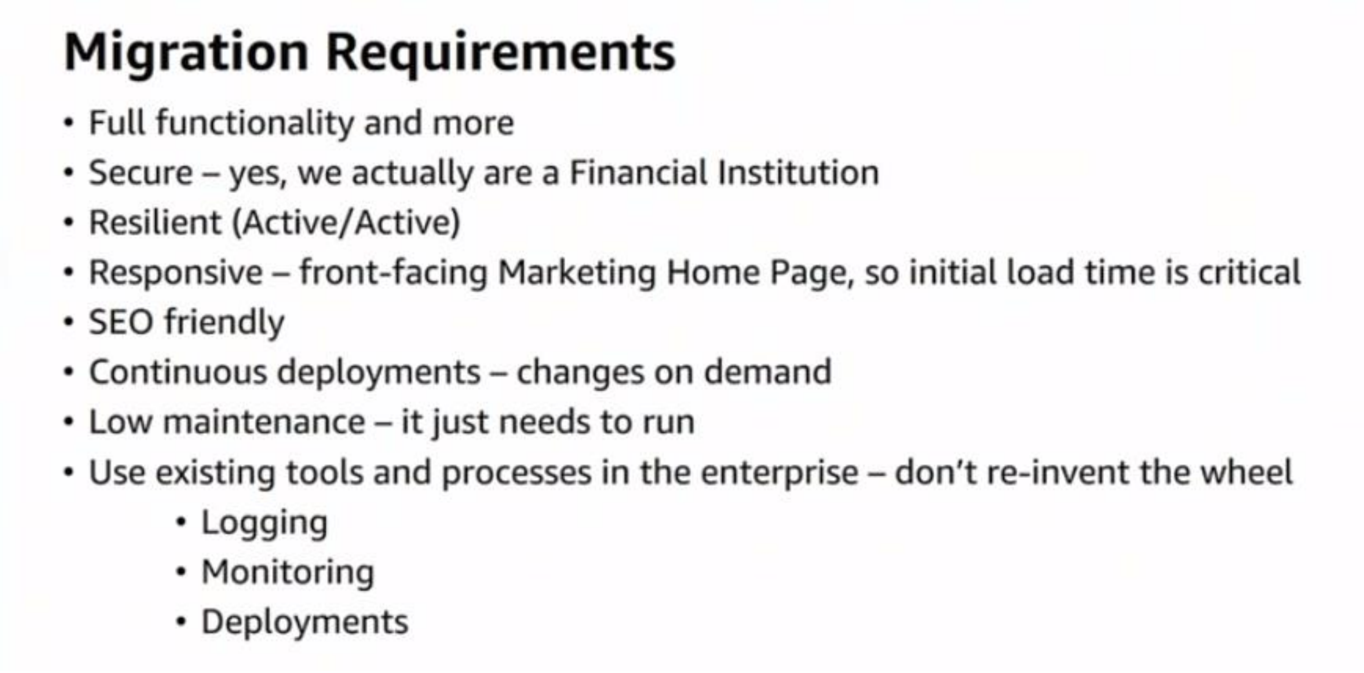
Rates as low as 3.24%
Annual Percentage Rate (APR*)

What Would You Like To Do?

- Top 10 U.S. Bank
- Alexa Website Ranking says capitalone.com traffic is "**81st** in U.S. websites"
- Auto Financing–1M hits/month

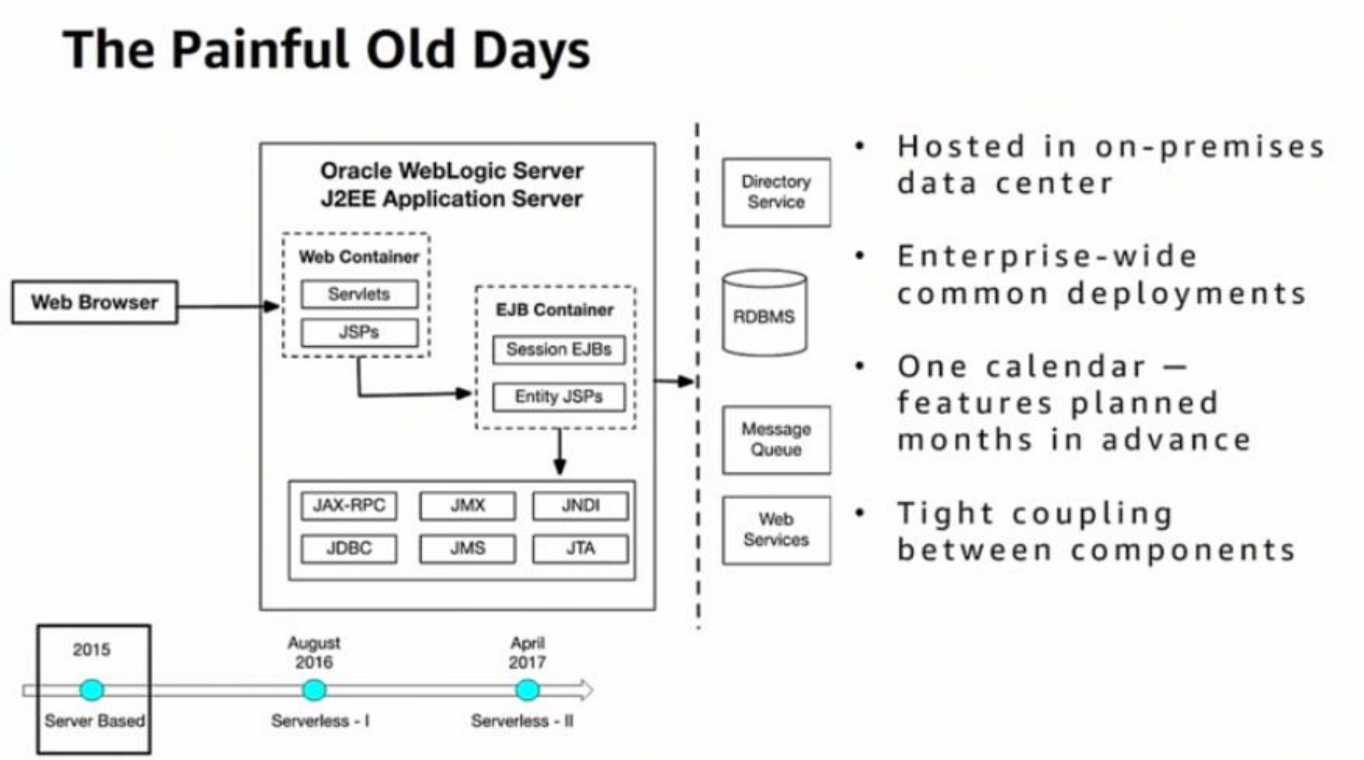
Migration Journey Timeline





Migration Requirements

- Full functionality and more
- Secure – yes, we actually are a Financial Institution
- Resilient (Active/Active)
- Responsive – front-facing Marketing Home Page, so initial load time is critical
- SEO friendly
- Continuous deployments – changes on demand
- Low maintenance – it just needs to run
- Use existing tools and processes in the enterprise – don't re-invent the wheel
    - Logging
    - Monitoring
    - Deployments



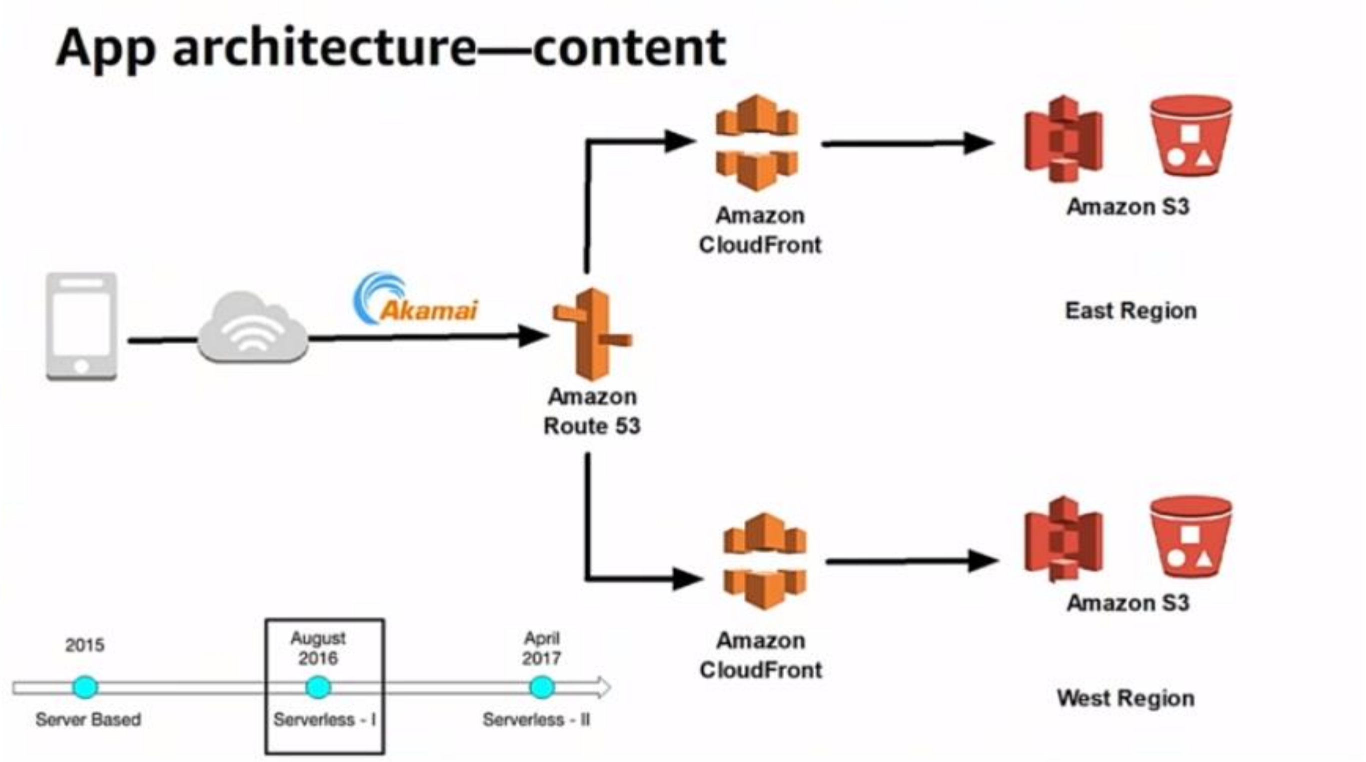
The Painful Old Days

- Hosted in on-premises data center
- Enterprise-wide common deployments
- One calendar – features planned months in advance
- Tight coupling between components

Serverless Migration – I

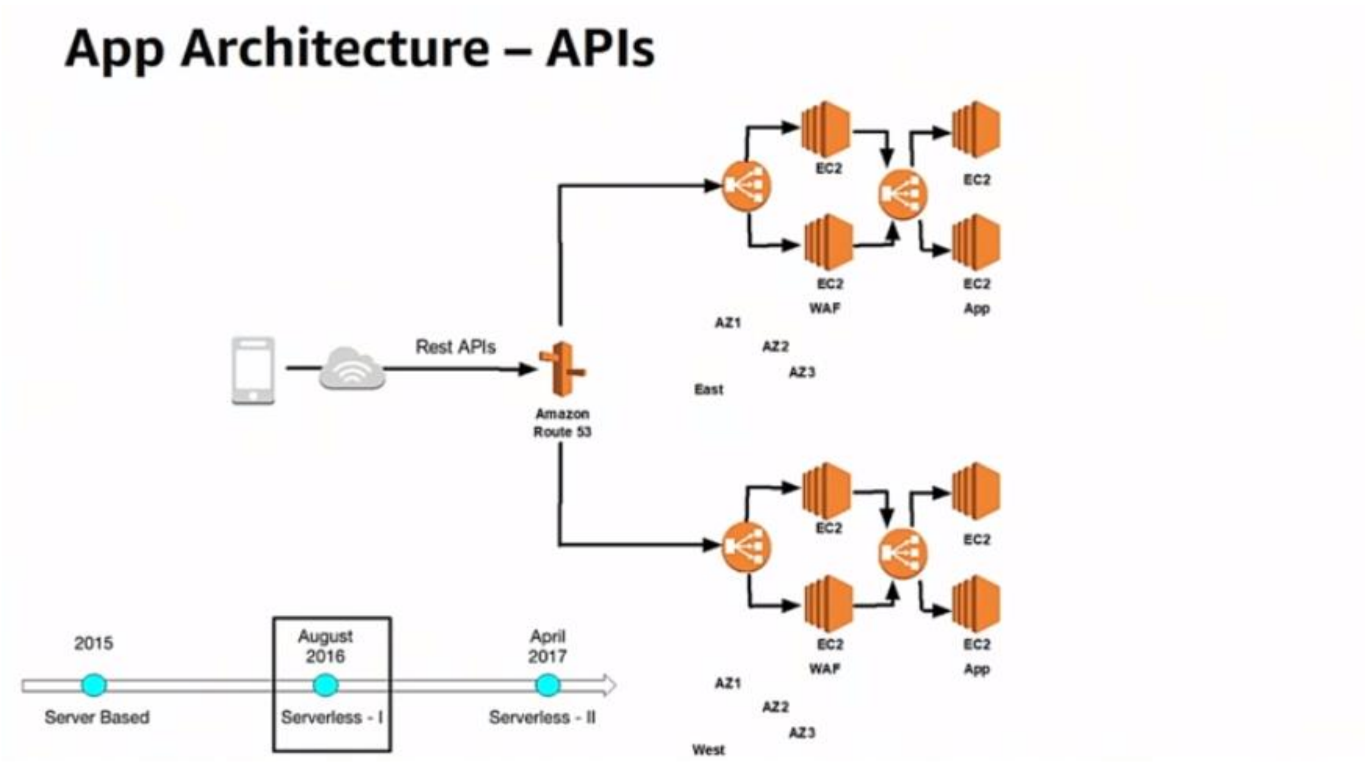
Move to AWS

Serverless Content Strategy



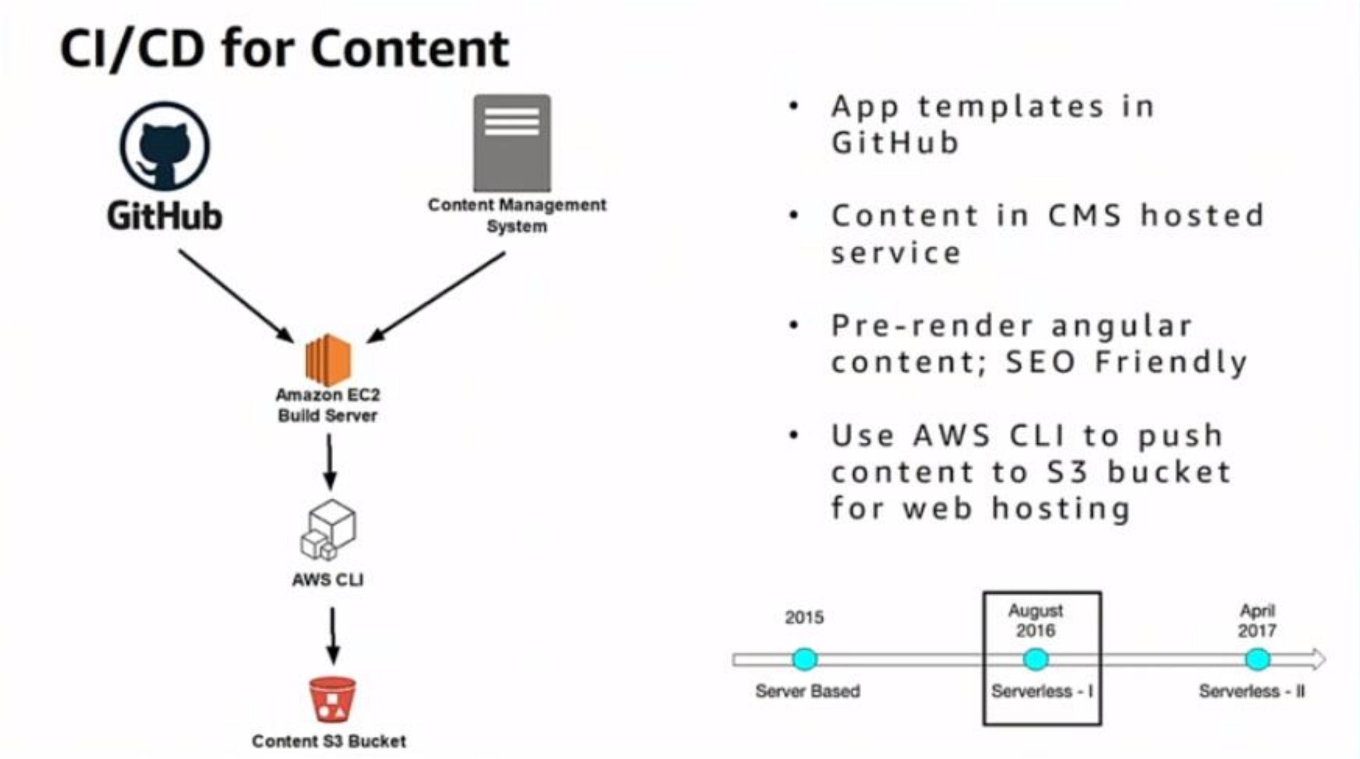
App architecture—content

We created an Angular SPA app that was bundled and stored in an S3 bucket with web hosting enabled. We then replicated the S3 bucket in both the East and West regions for an active-active deployment, then we fronted the web app using Cloudfront that has the certificates for SSL termination and also using Roue53. We were also able to leverage non-AWS services like Akamai for content caching.



App Architecture – APIs

This is the architecture that we used for our APIs. Due to internal company policies in 2016 we couldn’t go all serverless for our backends and had to come up with the alternate EC2-based architecture above. The first set of EC2 instances are running Apache with security configurations and serve as our web application firewall while the 2 EC2 instances behind the firewall are running Tomcat with our application logic running in them. Next, we then replicated this whole setup in another region for an active-active configuration.



CI/CD for Content

GitHub

Content Management System

Amazon EC2 Build Server

AWS CLI

Content S3 Bucket

- App templates in GitHub
- Content in CMS hosted service
- Pre-render angular content; SEO Friendly
- Use AWS CLI to push content to S3 bucket for web hosting

2015

Server Based

August 2016

Serverless - I

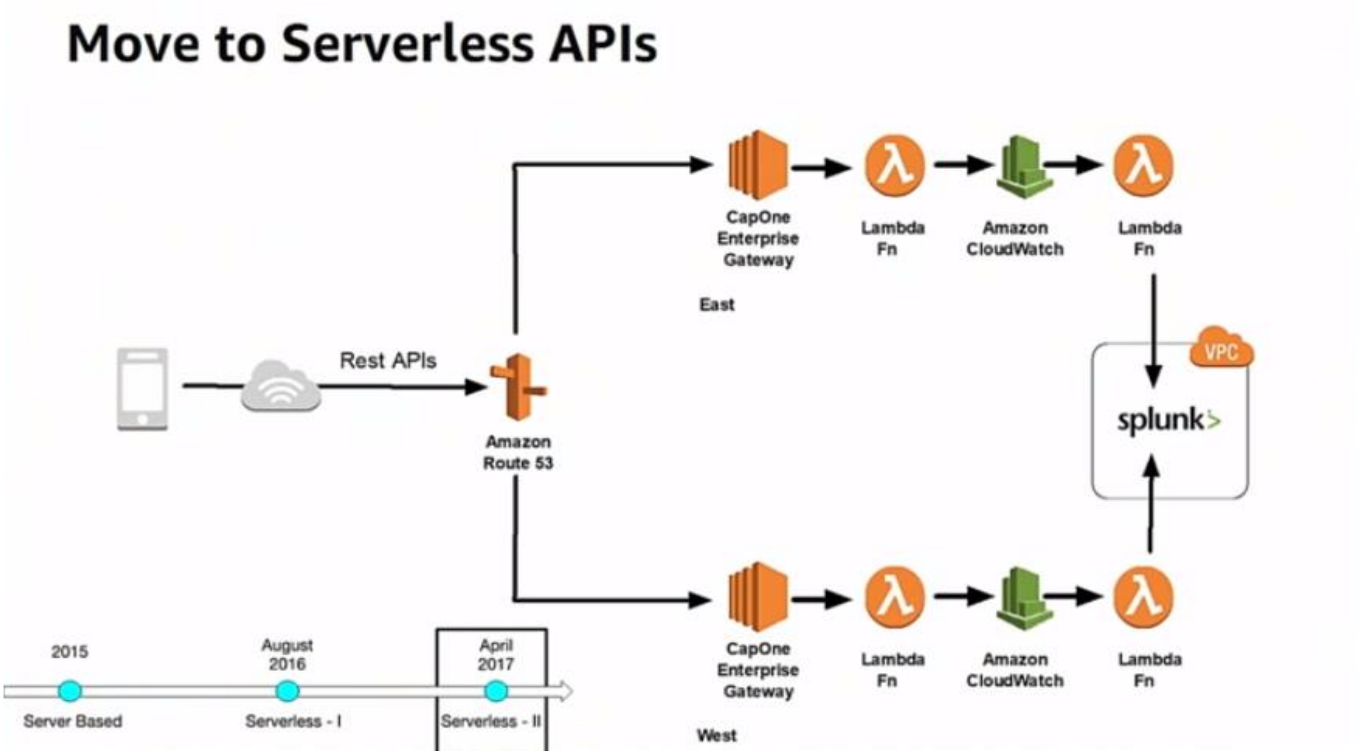
April 2017

Serverless - II

Serverless Migration - II

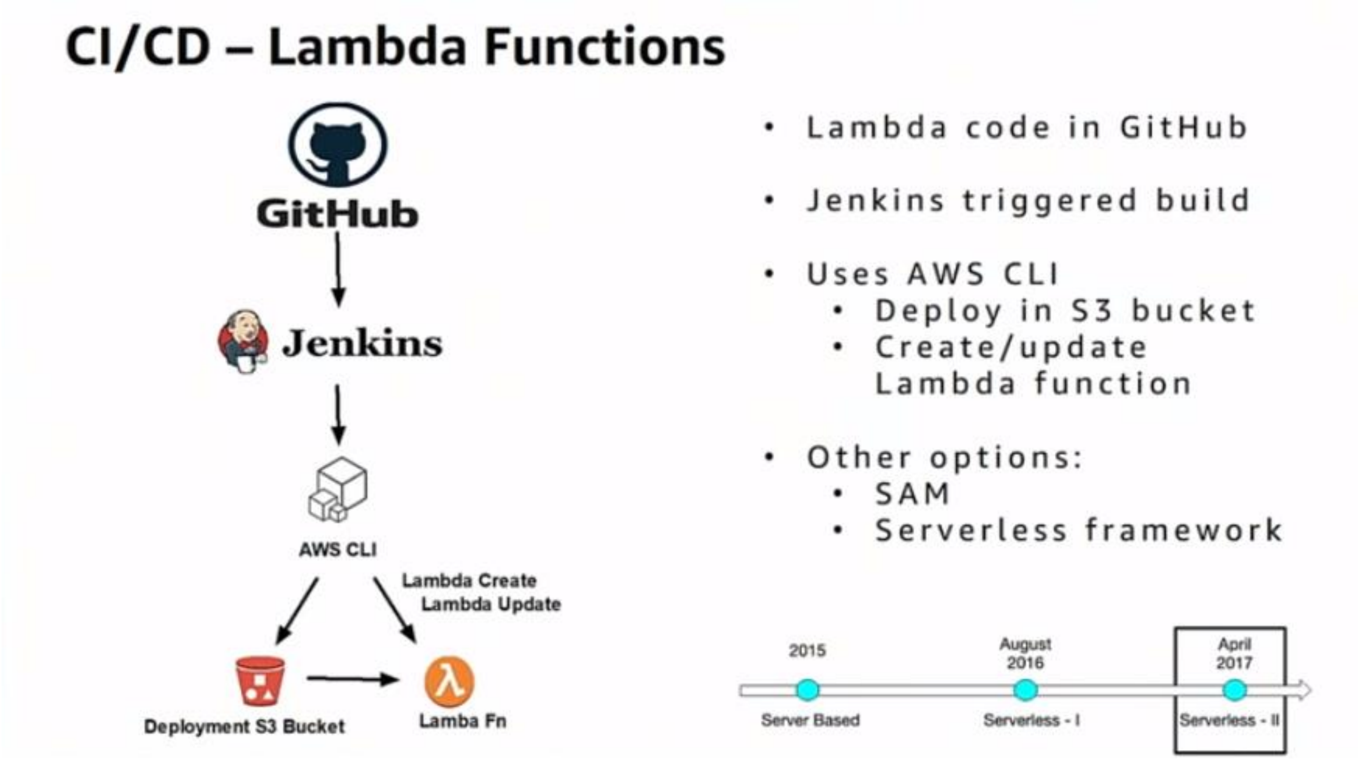
Lambda-based APIs

CI/CD Processes



Move to Serverless APIs

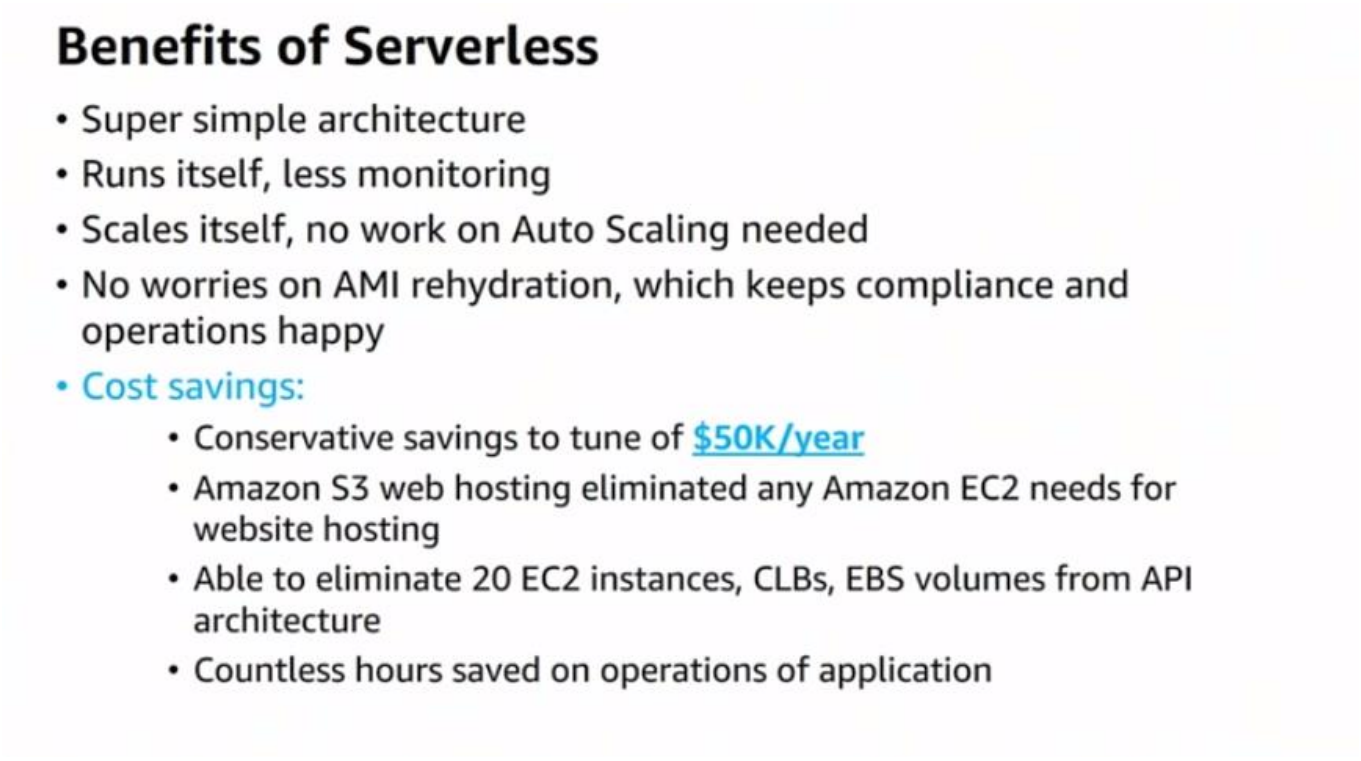
We have now eliminated the use of EC2 instances in our web app hosting architecture



CI/CD – Lambda Functions

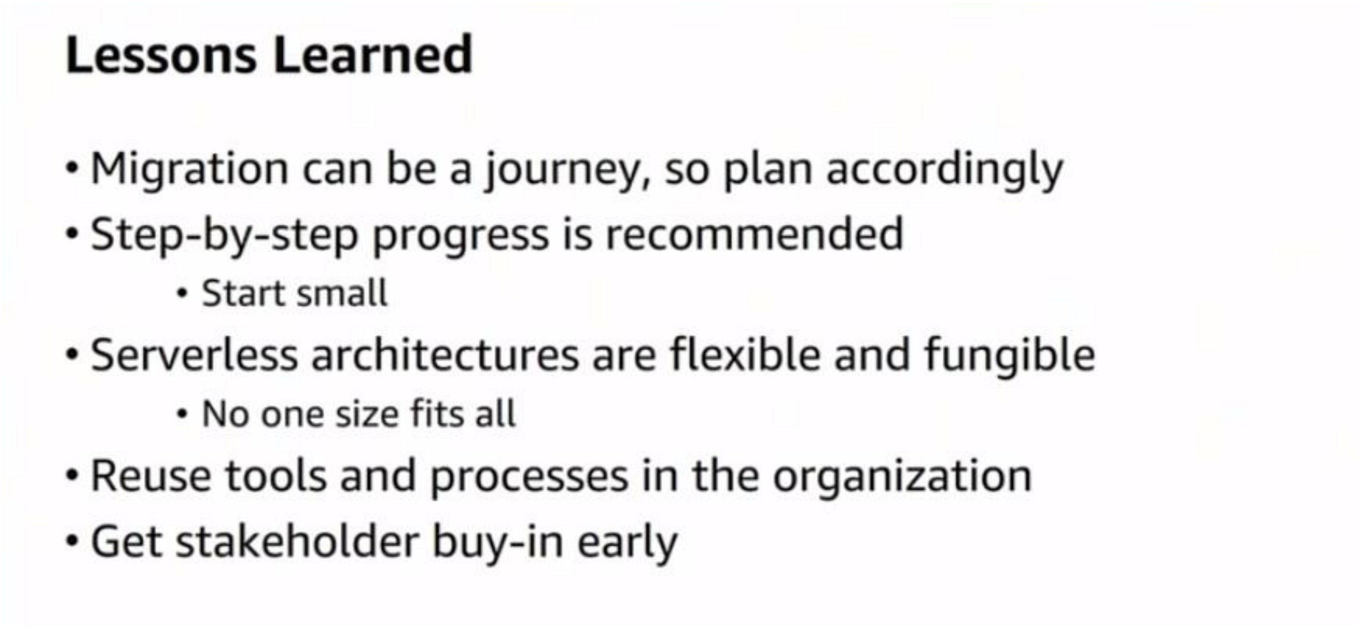
- Lambda code in GitHub
- Jenkins triggered build
- Uses AWS CLI
  - Deploy in S3 bucket
  - Create/update Lambda function
- Other options:
  - SAM
  - Serverless framework

For our CI/CD process for our lambda functions, Jenkins is our enterprise build tool of choice. We store our lambda function code in GitHub that triggers a build job in Jenkins, the Jenkins job then takes that code, uses the AWS CLI to create a deployment bundle that it stores in S3. By leveraging functions like lambda create and lambda update, we are able to create and update our lambda functions accordingly.



Benefits of Serverless

- Super simple architecture
- Runs itself, less monitoring
- Scales itself, no work on Auto Scaling needed
- No worries on AMI rehydration, which keeps compliance and operations happy
- Cost savings:
  - Conservative savings to tune of **$50K/year**
  - Amazon S3 web hosting eliminated any Amazon EC2 needs for website hosting
  - Able to eliminate 20 EC2 instances, CLBs, EBS volumes from API architecture
  - Countless hours saved on operations of application



Lessons Learned

- Migration can be a journey, so plan accordingly
- Step-by-step progress is recommended
  - Start small
- Serverless architectures are flexible and fungible
  - No one size fits all
- Reuse tools and processes in the organization
- Get stakeholder buy-in early

FIN, ACK

Serverless:

- No servers to manage
- No cost for idle
- Automatic scaling
- High availability

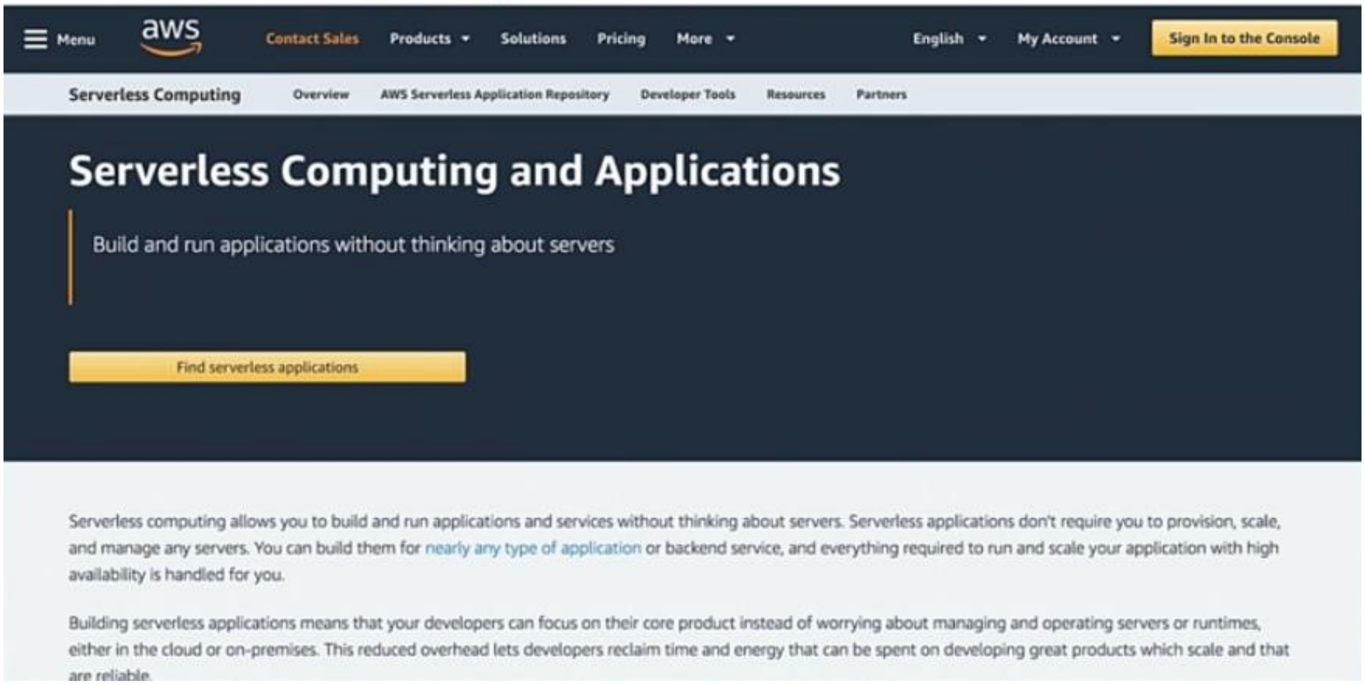
Use cases:

- Web applications
- Backends
- Data processing
- Chatbots
- Amazon Alexa
- IT Automation

Integrated across AWS:

- Amazon API Gateway
- AWS Step Functions
- Amazon S3
- Amazon Kinesis
- Amazon DynamoDB
- Amazon SNS
- Amazon Cognito
- AWS CloudFormation
- AWS CodePipeline
- Amazon CloudWatch
- AWS IoT
- many more!

aws.amazon.com/serverless



Serverless Computing and Applications

Build and run applications without thinking about servers

Serverless computing allows you to build and run applications and services without thinking about servers. Serverless applications don't require you to provision, scale, and manage any servers. You can build them for nearly any type of application or backend service, and everything required to run and scale your application with high availability is handled for you.

Building serverless applications means that your developers can focus on their core product instead of worrying about managing and operating servers or runtimes, either in the cloud or on-premises. This reduced overhead lets developers reclaim time and energy that can be spent on developing great products which scale and that are reliable.

ENT332

AWS re:Invent

Thank you!

Remember to review this session!

AWS re:Invent

aws