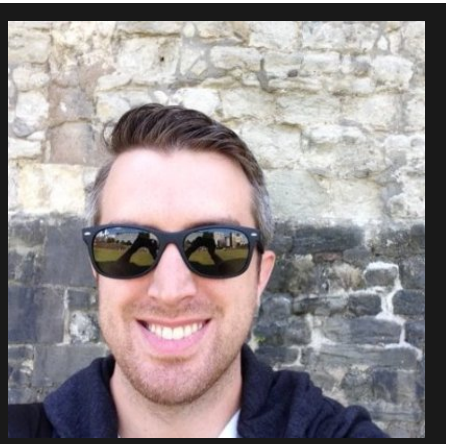
From Your Keyboard to Your Customers without a Server to Manage In-between

Chris Munns – Senior Developer Advocate - Serverless

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, 2011- Dec 2014
  - 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


BUT SERVERLESS STILL HAS SERV..
THAT'S NOT THE POINT

Serverless means…

No servers to provision or manage

Scales with usage

Never pay for idle

Availability and fault tolerance built in

Serverless application

EVENT SOURCE

Changes in data state

Requests to endpoints

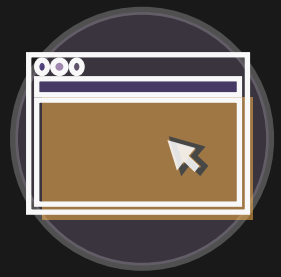
Changes in resource state

FUNCTION

Node.js
Python
Java
C#

SERVICES (ANYTHING)

Common Lambda use cases

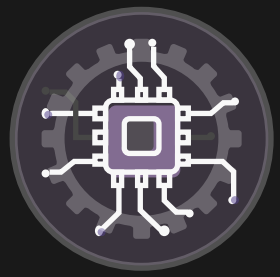


Web Applications

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

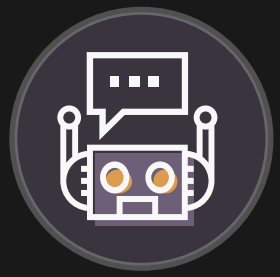
Backends

- Apps & services
- Mobile
- IoT



Data Processing

- Real time
- MapReduce
- Batch

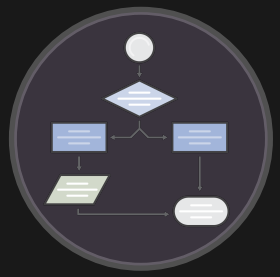


Chatbots

- Powering chatbot logic

Amazon Alexa

- Powering voice-enabled apps
- Alexa Skills Kit



IT Automation

- Policy engines
- Extending AWS services
- Infrastructure management


Where do you ..
START
?
https://secure.flickr.com/photos/stevendepolo/5749192025/

Frameworks

APEX

aws
Chalice

Claudia.js

SERVERLESS

SPARTA

ClaudiaJS

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

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

https://claudiajs.com

https://github.com/claudiajs/claudia

app.js:

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

module.exports = api;

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

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

Chalice

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

https://github.com/aws/chalice

https://chalice.readthedocs.io

app.py:

```
from chalice import Chalice
app = Chalice(app_name="helloworld")

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

$chalice deploy

Chalice – a bit deeper


```
from chalice import Chalice
from chalice import BadRequestError

app = Chalice(app_name='apiworld-hot')

FOOD_STOCK = {
        'hamburger': 'yes',
        'hotdog': 'no'
}

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

@app.route('/list_foods')
        def list_foods():
                return FOOD_STOCK.keys()

@app.route('/check_stock/{food}')
        def check_stock(food):
                try:
                        return {'in_stock': FOOD_STOCK[food]}
        except KeyError:
                raise BadRequestError("Unknown food '%s', valid choices are: %s" % (food, ', '.join(FOOD_STOCK.keys())))

@app.route('/add_food/{food}', methods=['PUT'])
        def add_food(food):
                return {"value": food}
```

Chalice – a bit deeper

Chalice

```
from chalice import Chalice
from chalice import BadRequestError

app = Chalice(app_name='apiworld-hot')

FOOD_STOCK = {
        'hamburger': 'yes',
        'hotdog': 'no'
}

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

@app.route('/list_foods')
        def list_foods():
                return FOOD_STOCK.keys()

@app.route('/check_stock/{food}')
        def check_stock(food):
                try:
                        return {'in_stock': FOOD_STOCK[food]}
        except KeyError:
                raise BadRequestError("Unknown food '%s', valid choices are: %s" % (food, ', '.join(FOOD_STOCK.keys())))

@app.route('/add_food/{food}', methods=['PUT'])
        def add_food(food):
                return {"value": food}
```

application routes

error handling

http method support

Meet SAM!

AWS Serverless Application Model (SAM)

CloudFormation extension optimized for serverless

New serverless resource types: functions, APIs, and tables

Supports anything CloudFormation supports

Open specification (Apache 2.0)

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

SAM template

```yaml
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
```

SAM template

```yaml
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 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 mapping/permissions necessary

Creates a DynamoDB table with 5 Read & Write units

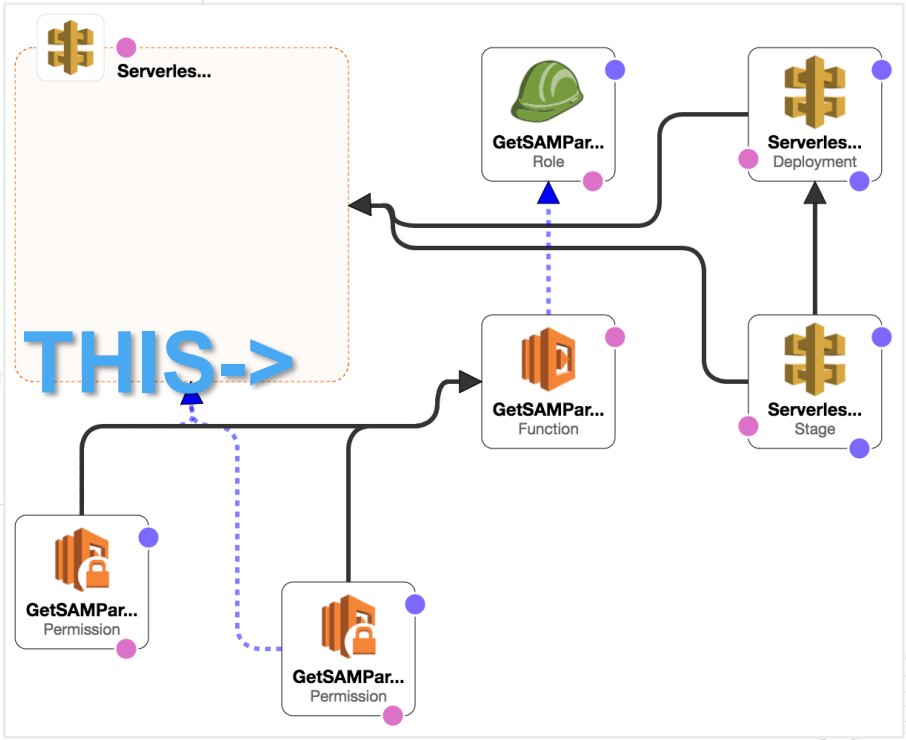
SAM template

26 lines (25 sloc) | 827 Bytes

Raw | Blame | History

```yaml
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

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

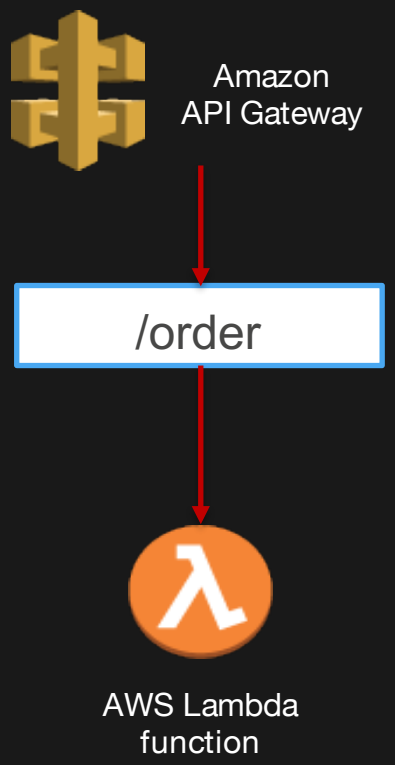
Currently supports Java, Node.js and Python

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

DEMO!

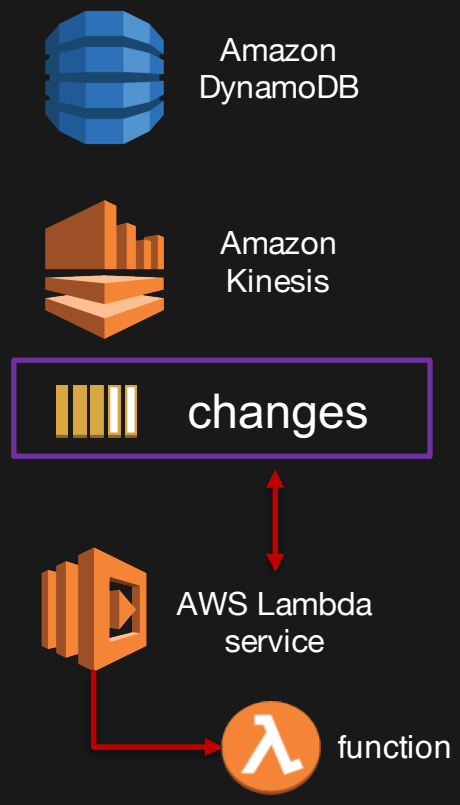
Lambda execution model

Synchronous (push)



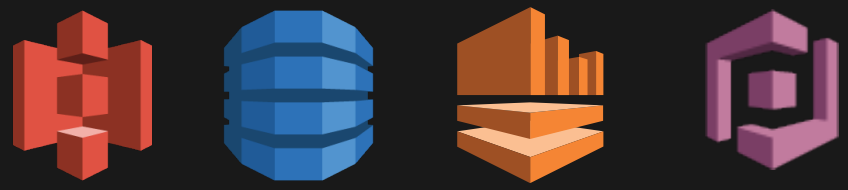
Asynchronous (event)

Stream-based



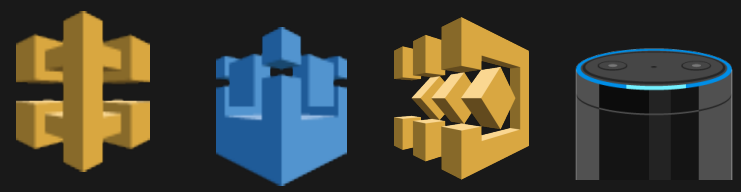
Event sources that trigger AWS Lambda

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

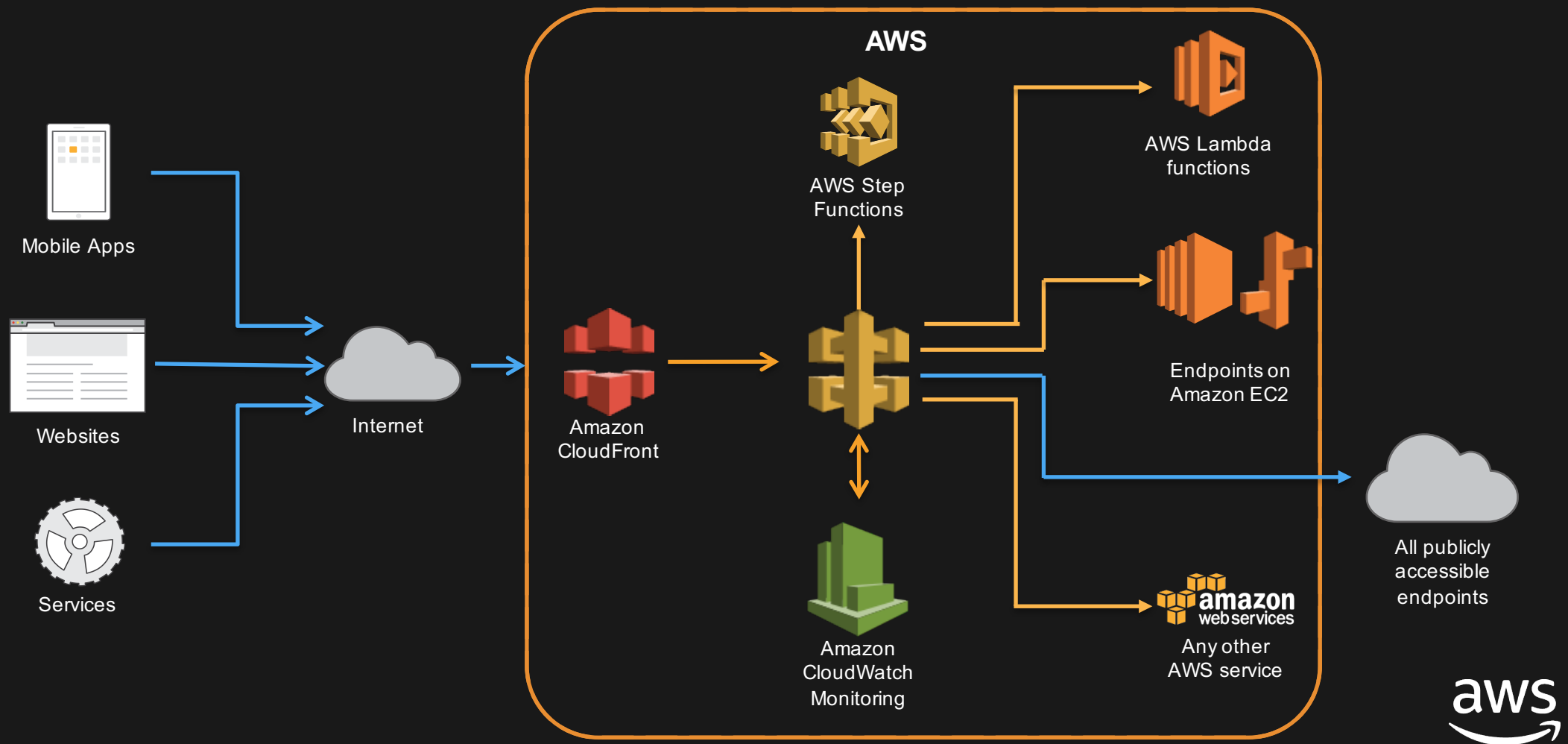
AWS CloudFormation | AWS CloudTrail | AWS CodeCommit | Amazon CloudWatch

EVENT/MESSAGE SERVICES

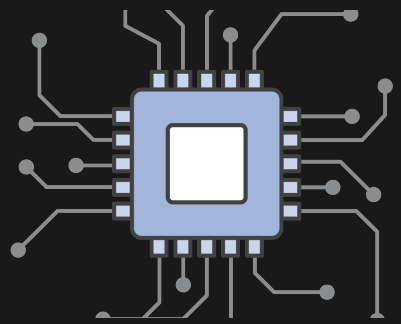
Amazon SES | Amazon SNS | Cron events

... and more!

Amazon API Gateway



Amazon API Gateway



Create a unified API frontend for multiple micro-services

DDoS protection and throttling for your backend

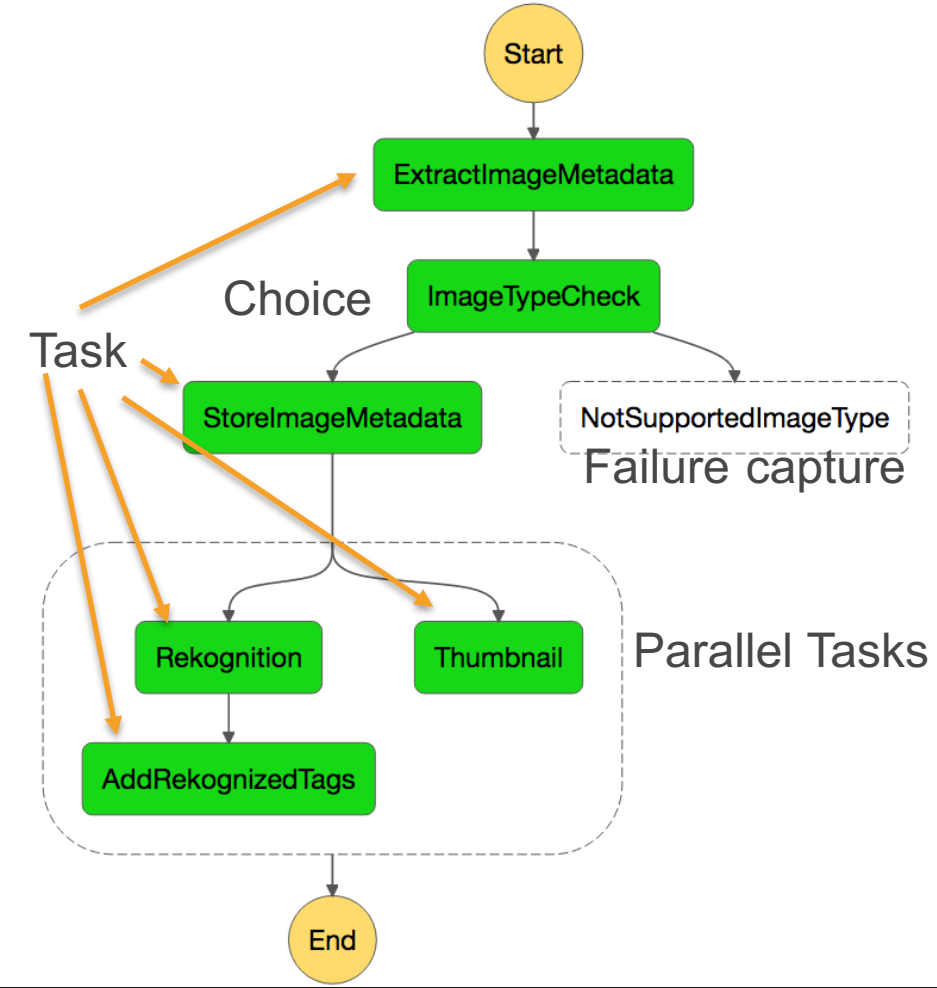
Authenticate and authorize requests to a backend

Throttle, meter, and monetize API usage by 3rd party developers

AWS Step Functions

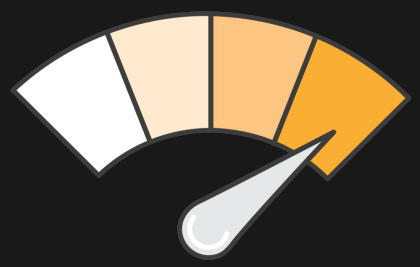
"Serverless" workflow management with zero administration

- 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



Computer power

> Memory > Cores



Lambda exposes only a memory control, this also affects the **% of CPU core** allocated to a function

Is your code CPU, Network or memory-bound, it could be **cheaper** to give more memory

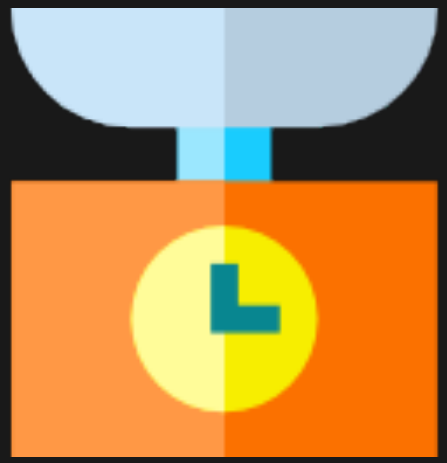
Less memory is not always cheaper

Stats for Lambda function that calculates **1000 times** all prime numbers **<= 1000000**

128mb	11.722965sec	$0.024628
256mb	6.678945sec	$0.028035
512mb	3.194954sec	$0.026830
1024mb	1.465984sec	$0.024638

aws

Lambda function runtimes



Separate business logic from function signature

Choose dependencies/frameworks carefully

Interpret languages initialize much quicker but not necessarily faster overall

Separate business logic from function signature

```
app = Todo()

def lambda_handler(event, context):
    ret = app.dispatch(event)

    return {
        'statusCode':  ret["status_code"],
        'headers':     ret["headers"],
        'body':        json.dumps(ret["body"])
    }
```

Leverage container reuse

- Lazily load variables in the **global scope** – functions stay warm for several minutes
- Don't load it if you don't need it – **cold starts** are affected

```
s3 = boto3.resource('s3')
db = db.connect()

def lambda_handler(event, context):
    global db
    # verify if still connected
    # otherwise carry on
    if not db:
        db = db.connect()
...
```

Cold start: Understand the function lifecycle

Download
your code

aws

Cold start: Understand the function lifecycle

Download
your code

Start new
container

aws

Cold start: Understand the function lifecycle

Download **your code**

Start new **container**

Bootstrap the **runtime**

aws

Cold start: Understand the function lifecycle

Download **your code**

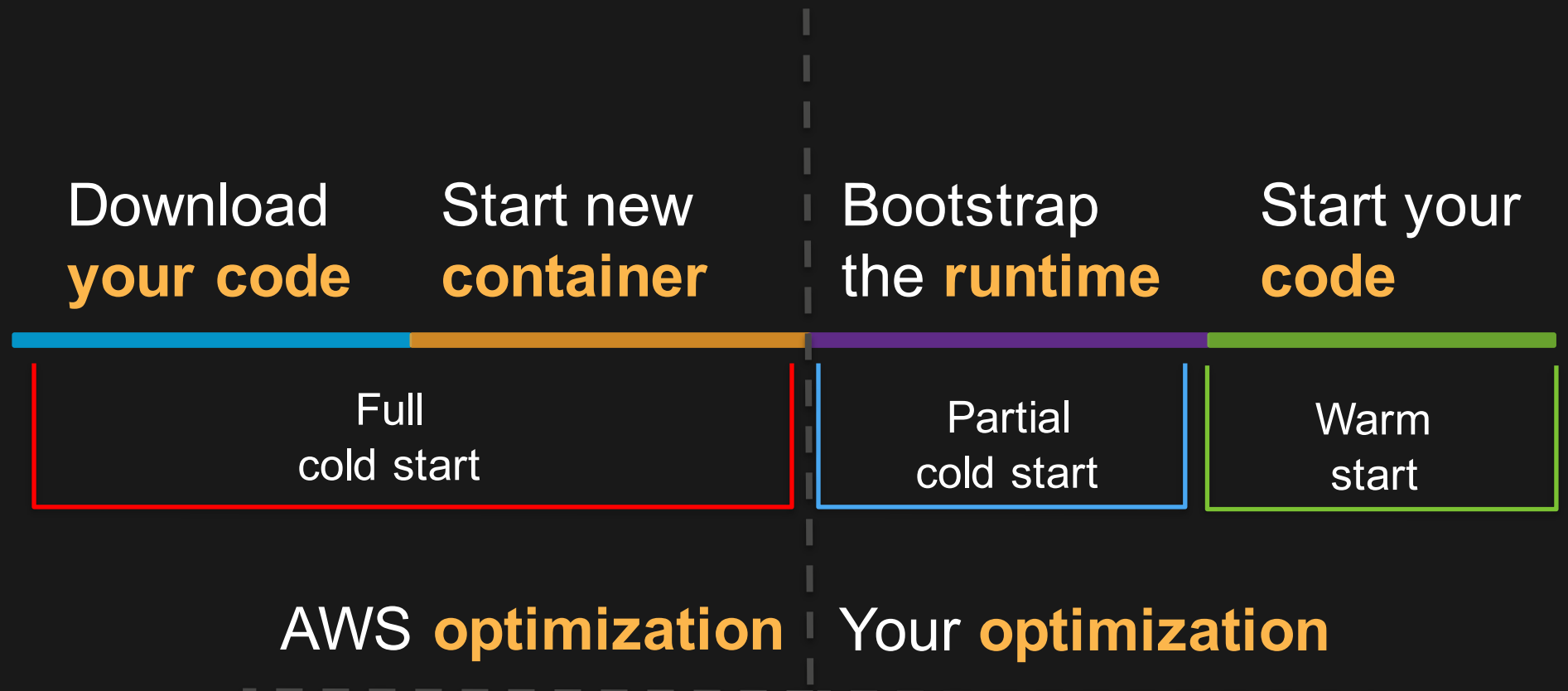
Start new **container**

Bootstrap the **runtime**

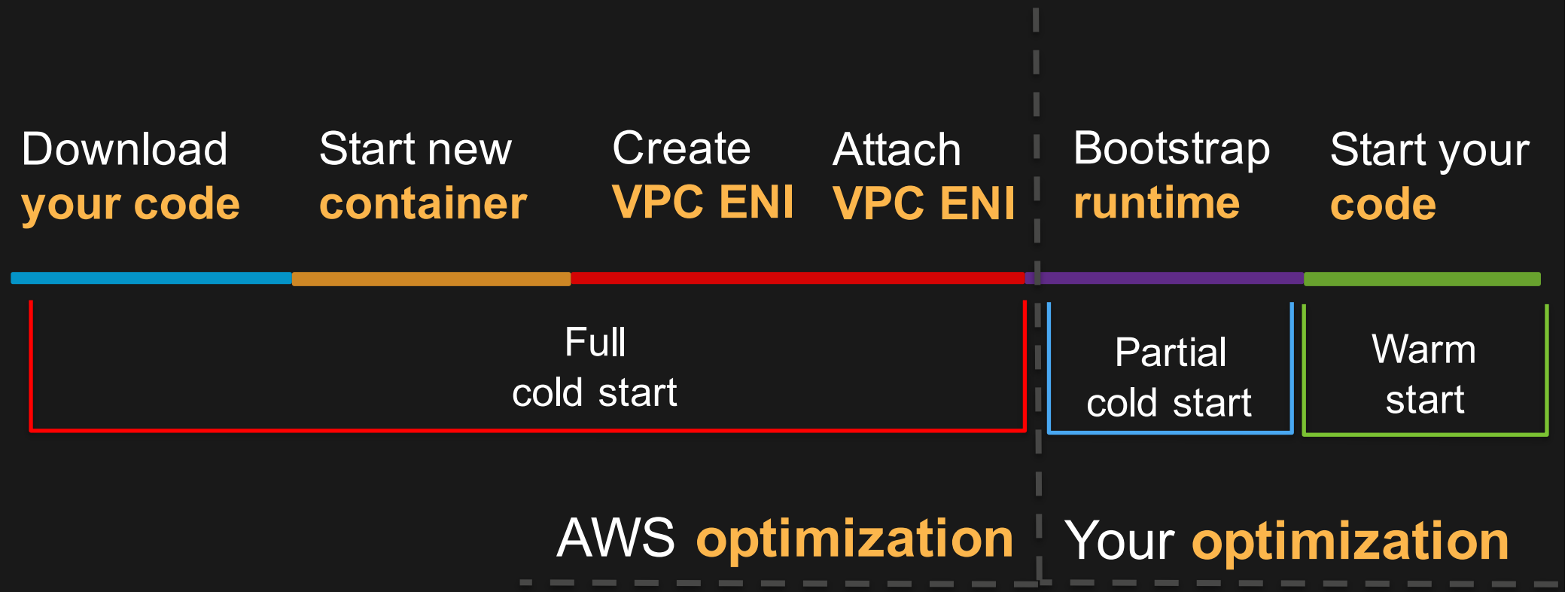
Start your **code**

aws

Cold start: Understand the function lifecycle



Cold start: Understand the function lifecycle (VPC)



DEMO!

Takeaways

Cold starts look bad during development, **not frequent in prod**

More memory is not always **more expensive**

Don’t **over-optimize** your code, just use the global scope wisely

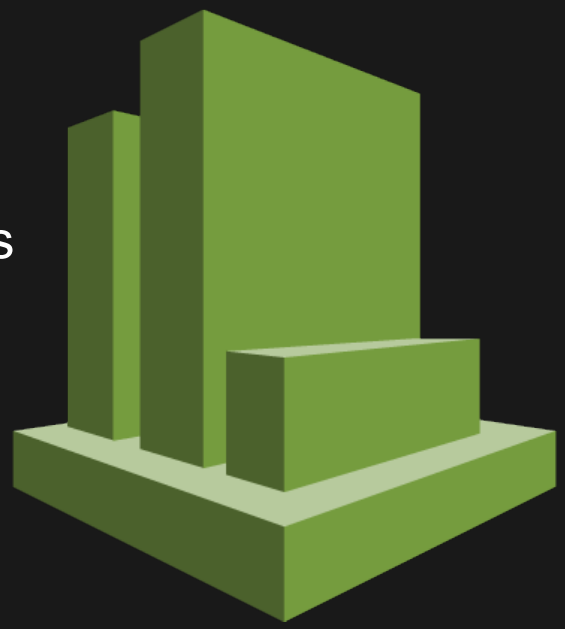
aws


Can’t move fast if you can’t measure what's going on.
https://secure.flickr.com/photos/jasoneppink/499531891

Metrics and logging are a universal right!

CloudWatch Metrics:

- 6 Built in metrics for Lambda
  - Invocation Count, Invocation duration, Invocation errors, Throttled Invocation, Iterator Age, DLQ Errors
  - Can call "put-metric-data" from your function code for custom metrics
- 7 Built in metrics for API-Gateway
  - API Calls Count, Latency, 4XXs, 5XXs, Integration Latency, Cache Hit Count, Cache Miss Count
  - Error and Cache metrics now support *averages and percentiles*

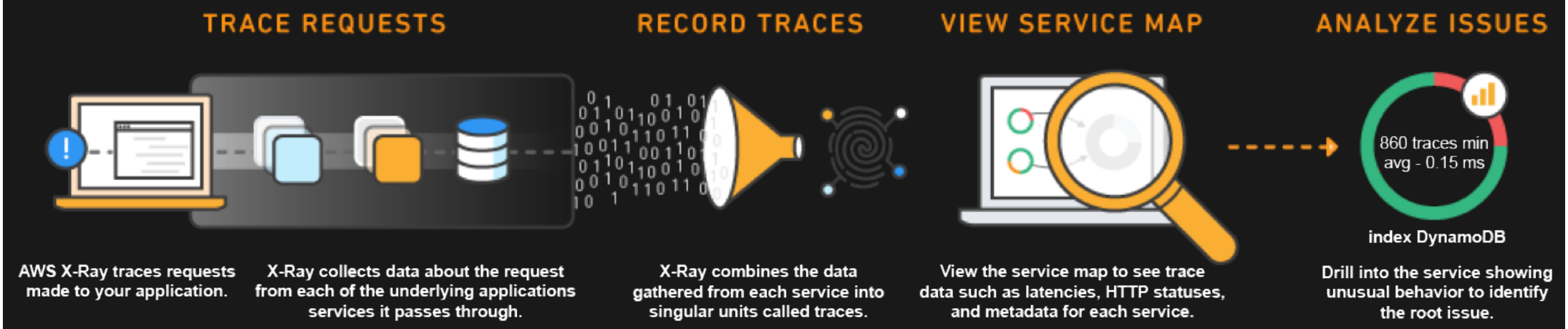


Metrics and logging are a universal right!

CloudWatch Logs:

- API Gateway Logging
  - 2 Levels of logging, ERROR and INFO
  - Optionally log method request/body content
  - Set globally in stage, or override per method
- Lambda Logging
  - Logging directly from your code with your language's equivalent of console.log()
  - Basic request information included
- Log Pivots
  - Build metrics based on log filters
  - Jump to logs that generated metrics
- Export logs to AWS ElastiCache or S3
  - Explore with Kibana or Athena/QuickSight

AWS X-Ray



AWS X-Ray traces requests made to your application.

X-Ray collects data about the request from each of the underlying applications services it passes through.

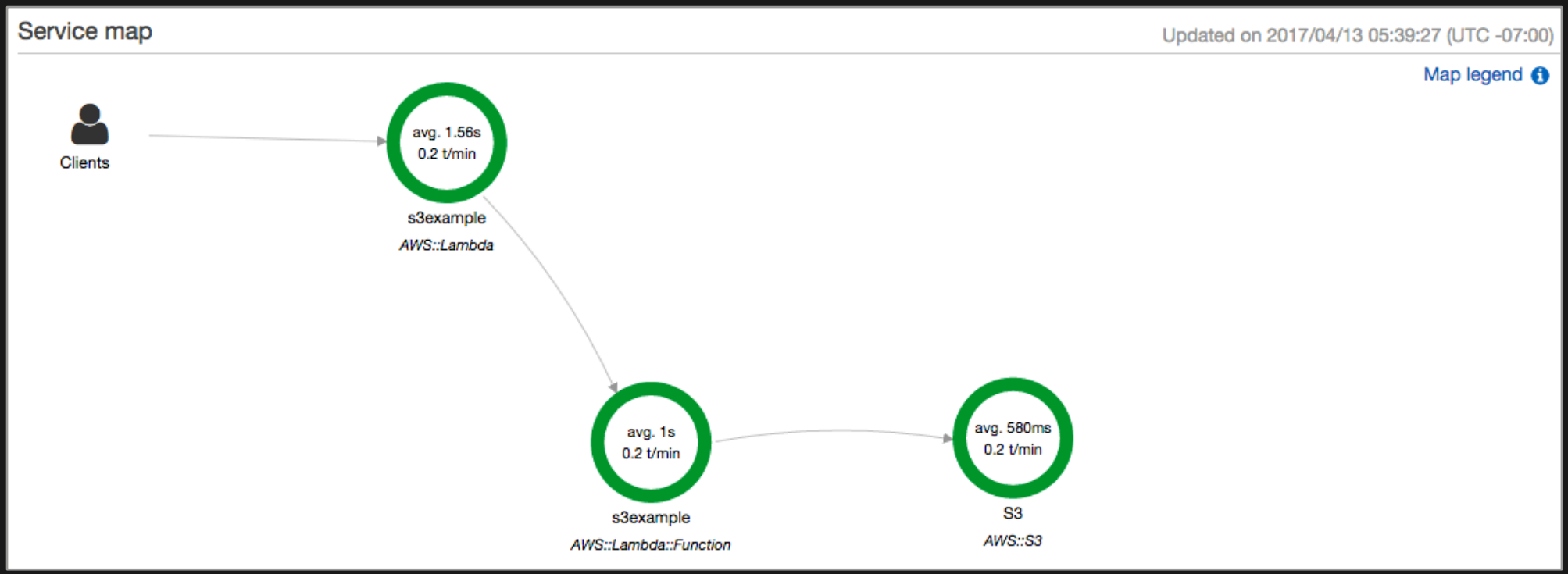
X-Ray combines the data gathered from each service into singular units called traces.

View the service map to see trace data such as latencies, HTTP statuses, and metadata for each service.

Drill into the service showing unusual behavior to identify the root issue.

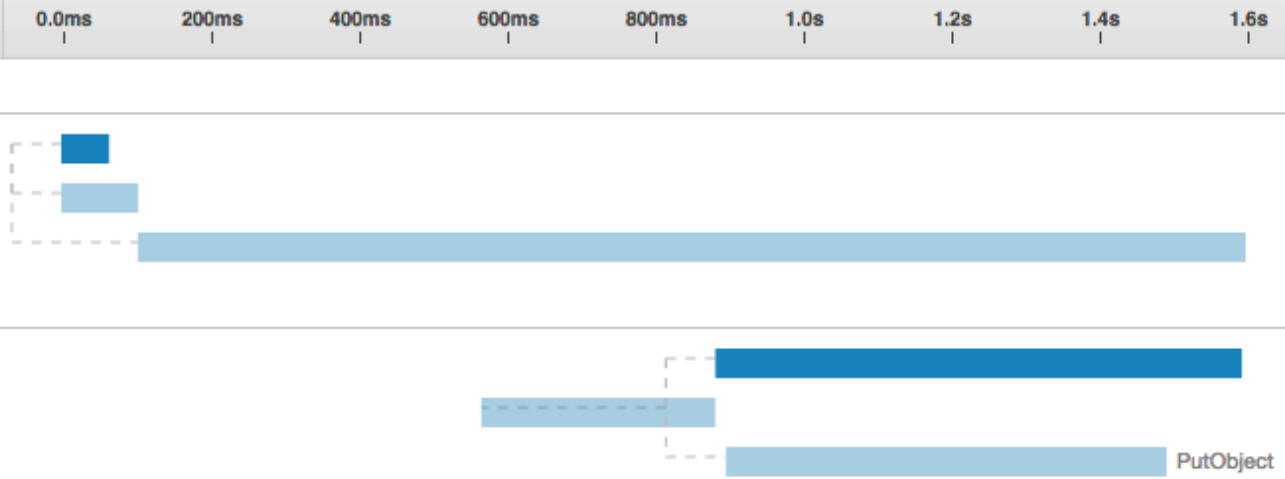
Application instrumentation (Node.js)

```
1  var AWSXRay = require('aws-xray-sdk-core');
2  var AWS = AWSXRay.captureAWS(require('aws-sdk'));
3  s3 = new AWS.S3({signatureVersion: 'v4'});
4
5  exports.handler = (event, context, callback) => {
6
7      var params = {Bucket: 'tim-example-bucket', Key: 'MyKey', Body: 'Hello!'};
8
9      s3.putObject(params, function(err, data) {});
10 };
```


Service map
Updated on 2017/04/13 05:39:27 (UTC -07:00)
Map legend
Clients
avg. 1.56s
0.2 t/min
s3example
AWS::Lambda
avg. 1s
0.2 t/min
s3example
AWS::Lambda::Function
avg. 580ms
0.2 t/min
S3
AWS::S3

Method	Response	Duration	Age	ID
--	202	1.6 sec	18.1 sec (2017-04-14 00:39:13 UTC)	1-58f01a31-24551f535d0ed5f5a70bdbf2

Name	Res.	Duration	Status
s3example AWS::Lambda			
s3example	202	63.0 ms	✓
Dwell Time	-	101 ms	✓
Attempt #1	200	1.5 sec	✓
s3example AWS::Lambda::Function			
s3example	-	693 ms	✓
Initialization	-	308 ms	✓
S3	200	580 ms	✓




Build & test your application
https://secure.flickr.com/photos/spenceyc/7481166880

AWS CodeBuild

Fully managed build service that compiles source code, runs tests, and produces software packages

Scales continuously and processes multiple builds concurrently

You can provide custom build environments suited to your needs via Docker images

Only pay by the minute for the compute resources you use

Can be used as a “Test” action in CodePipeline

Launched with CodePipeline and Jenkins integration

buildspec.yml Example

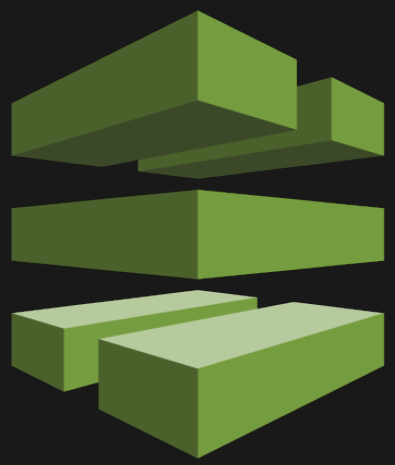
```
version: 0.1

environment_variables:
  plaintext:
      "INPUT_FILE": "saml.yaml”
      "S3_BUCKET": ""

phases:
  install:
    commands:
      - npm install
pre_build:
    commands:
      - eslint *.js
build:
    commands:
      - npm test
post_build:
    commands:
      - aws cloudformation package --template $INPUT_FILE --s3-
bucket $S3_BUCKET --output-template post-saml.yaml
artifacts:
  type: zip
  files:
      - post-saml.yaml
      - beta.json
```

buildspec.yml Example

```
version: 0.1

environment_variables:
  plaintext:
      "INPUT_FILE": "saml.yaml”
      "S3_BUCKET": ""

phases:
  install:
    commands:
      - npm install
pre_build:
    commands:
      - eslint *.js
build:
    commands:
      - npm test
post_build:
    commands:
      - aws cloudformation package --template $INPUT_FILE --s3-
bucket $S3_BUCKET --output-template post-saml.yaml
artifacts:
  type: zip
  files:
      - post-saml.yaml
      - beta.json
```

- Variables to be used by phases of build
- Examples for what you can do in the phases of a build:
  - You can install packages or run commands to prepare your environment in ”install”.
  - Run syntax checking, commands in “pre_build”.
  - Execute your build tool/command in “build”
  - Test your app further or ship a container image to a repository in post_build
- Create and store an artifact in S3

AWS CodePipeline



Continuous delivery service for fast and reliable application updates

Model and visualize your software release process

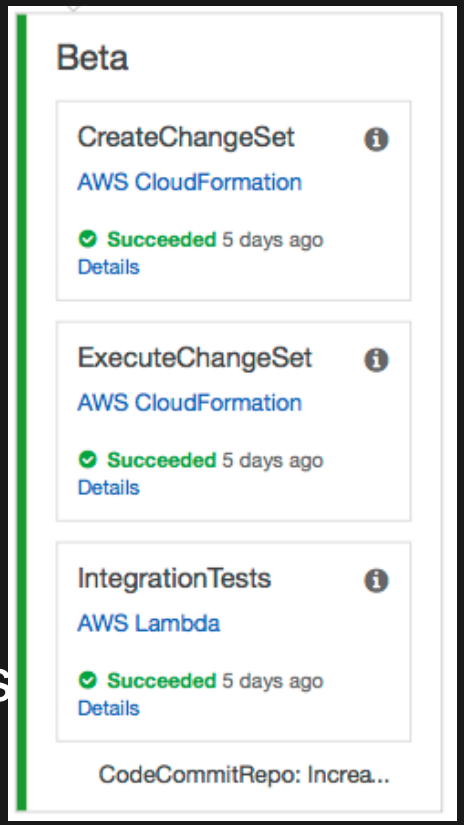
Builds, tests, and deploys your code every time there is a code change

Integrates with third-party tools and AWS

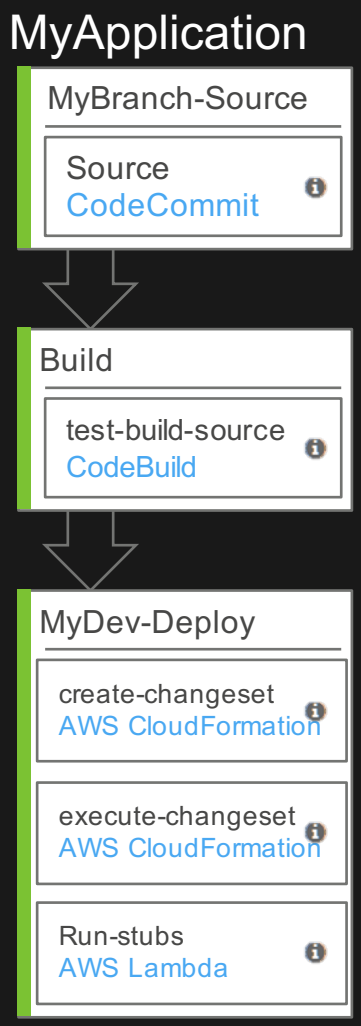
Delivery via CodePipeline

Pipeline flow:

1. Commit your code to a source code repository
2. Package/Test in CodeBuild
3. Use CloudFormation actions in CodePipeline to create or update stacks via SAM templates
   - Optional: Make use of ChangeSets
4. Make use of specific stage/environment parameter files to pass in Lambda variables
5. Test our application between stages/environments
   - Optional: Make use of Manual Approvals



An example minimal Developer's pipeline:



This pipeline:

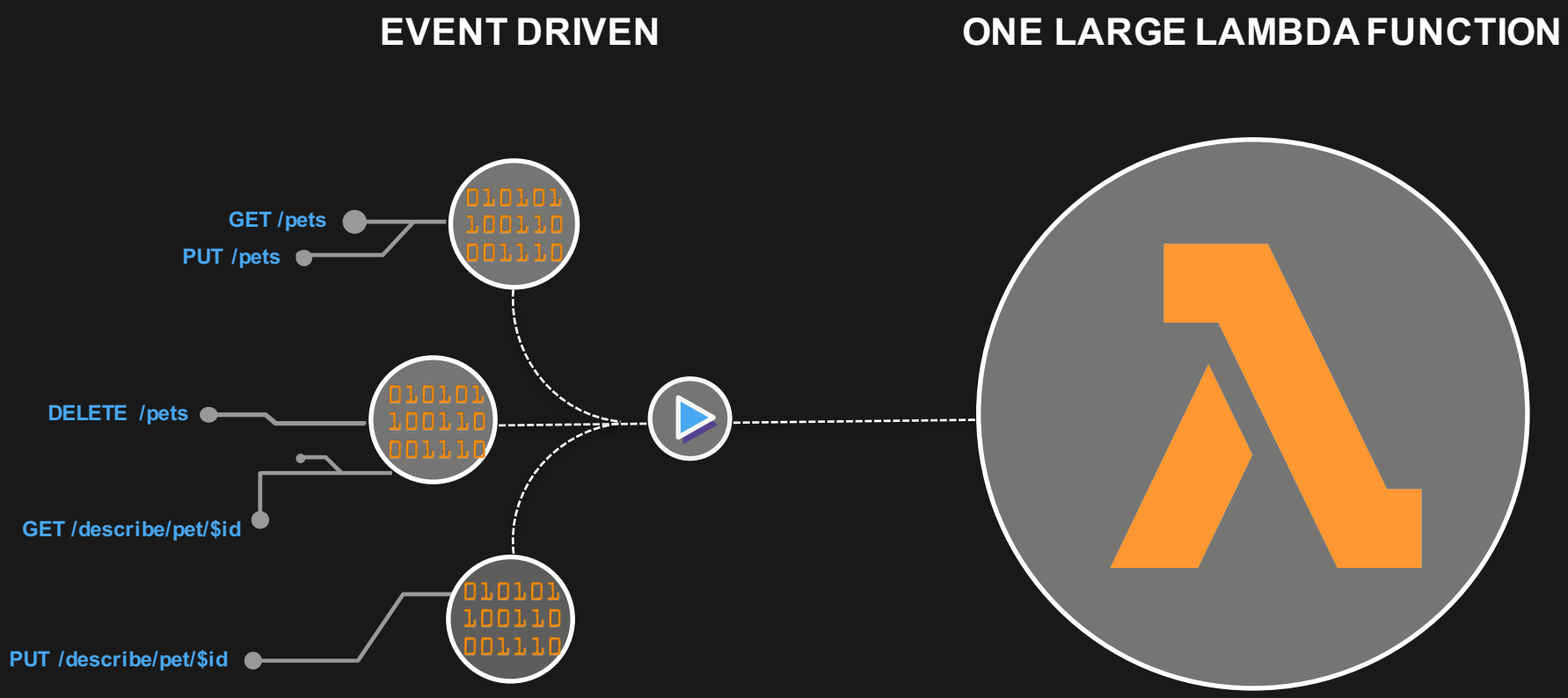
- Three Stages
- Builds code artifact
- One Development environment
- Uses SAM/CloudFormation to deploy artifact and other AWS resources
- Has Lambda custom actions for running my own testing functions

Start with AWS CodeStar

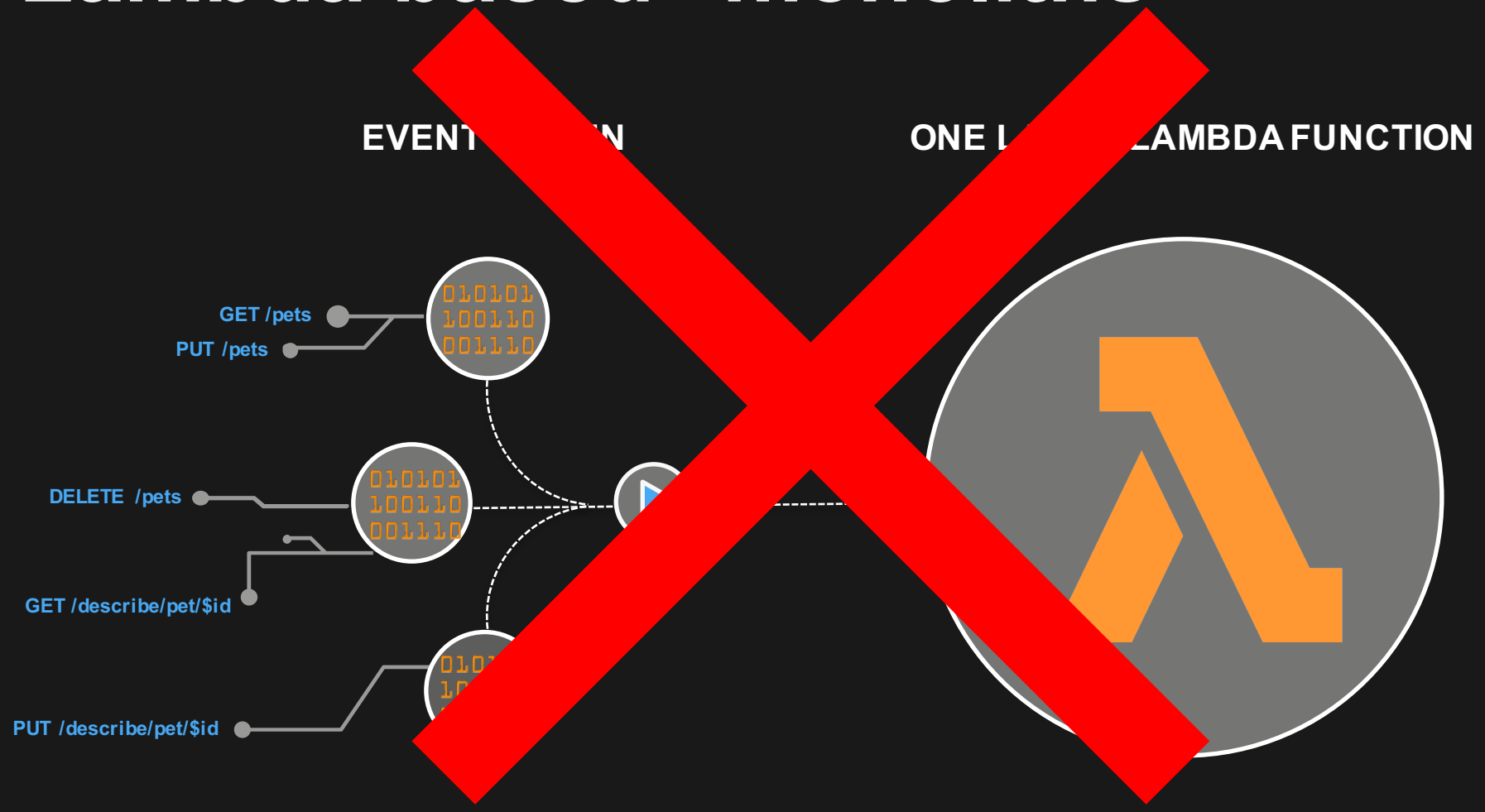
Best Practices

aws

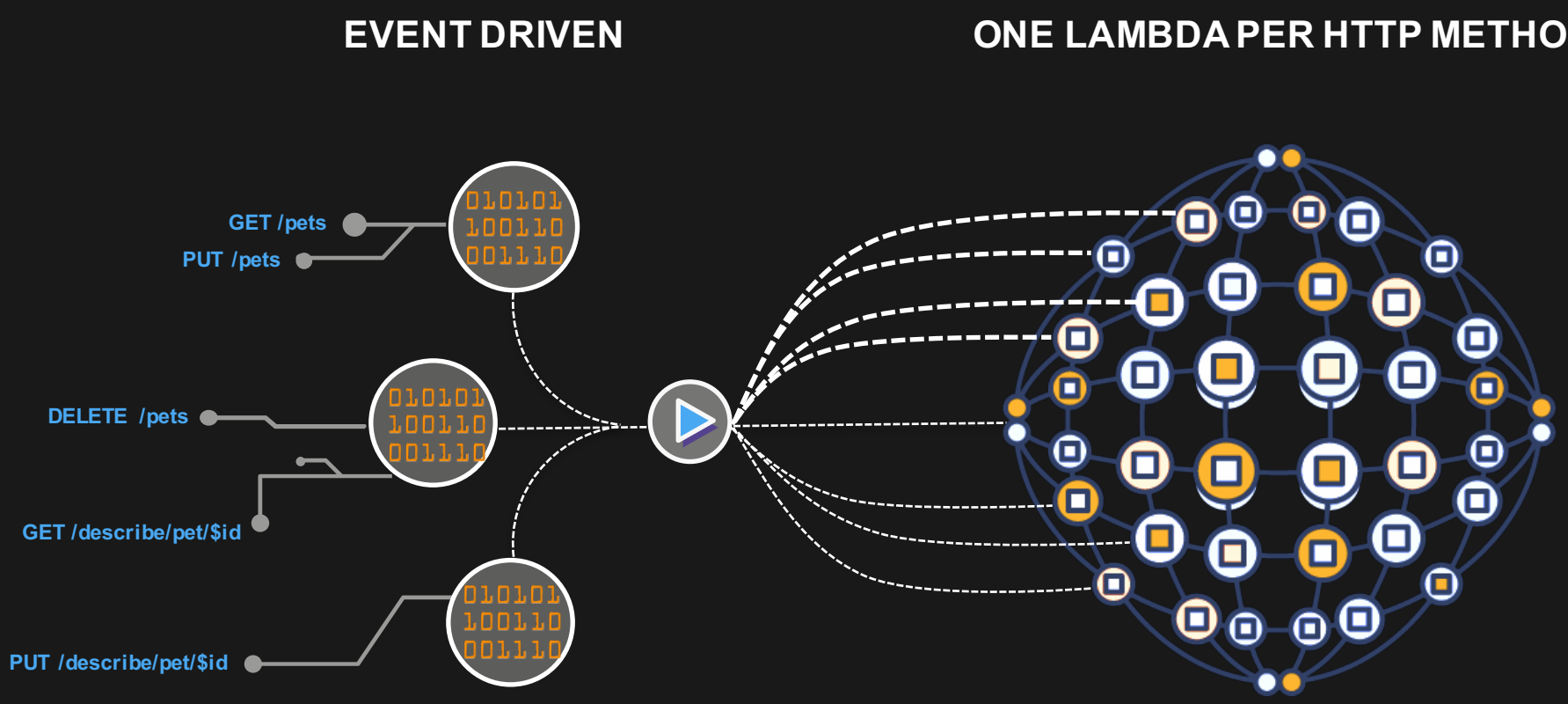
Lambda based “monoliths”



Lambda based “monoliths”



Lambda based “nano-services”



SAM Best Practices

- Unless function handlers share code, split them into their own independent Lambda functions files or binaries
  - Another option is to use language specific packages to share common code between functions
- Unless independent Lambda functions share event sources, split them into their own code repositories with their own SAM templates
- Locally validate your YAML or JSON SAM files before committing them. Then do it again in your CI/CD process

Lambda Environment Variables

- Key-value pairs that you can dynamically pass to your function
- Available via standard environment variable APIs such as process.env for Node.js or os.environ for Python
- Can optionally be encrypted via AWS Key Management Service (KMS)
  - Allows you to specify in IAM what roles have access to the keys to decrypt the information
- Useful for creating environments per stage (i.e. dev, testing, production)

API Gateway Stage Variables

- Stage variables act like environment variables
- Use stage variables to store configuration values
- Stage variables are available in the $context object
- Values are accessible from most fields in API Gateway
  - Lambda function ARN
  - HTTP endpoint
  - Custom authorizer function name
  - Parameter mappings

Lambda and API Gateway Variables + SAM

```
Parameters:
  MyEnvironment:
    Type: String
    Default: testing
    AllowedValues:
      - testing
      - staging
      - prod
    Description: Environment of this stack of
resources
  SpecialFeature1:
    Type: String
    Default: false
    AllowedValues:
      - true
      - false
    Description: Enable new SpecialFeature1
…
```

```
…
#Lambda
MyFunction:
  Type: 'AWS::Serverless::Function'
  Properties:
  …
  Environment:
    Variables:
      ENVIRONMENT: !Ref: MyEnvironment
      Spec_Feature1: !Ref: SpecialFeature1
…
#API Gateway
MyApiGatewayApi:
  Type: AWS::Serverless::Api
   Properties:
…
   Variables:
    ENVIRONMENT: !Ref: MyEnvironment
    SPEC_Feature1: !Ref: SpecialFeature1
…
```

AWS Systems Manager – Parameter Store

Centralized store to manage your configuration data

- supports hierarchies
- plain-text or encrypted with KMS
- Can send notifications of changes to Amazon SNS/ AWS Lambda
- Can be secured with IAM
- Calls recorded in CloudTrail
- Can be tagged
- Available via API/SDK

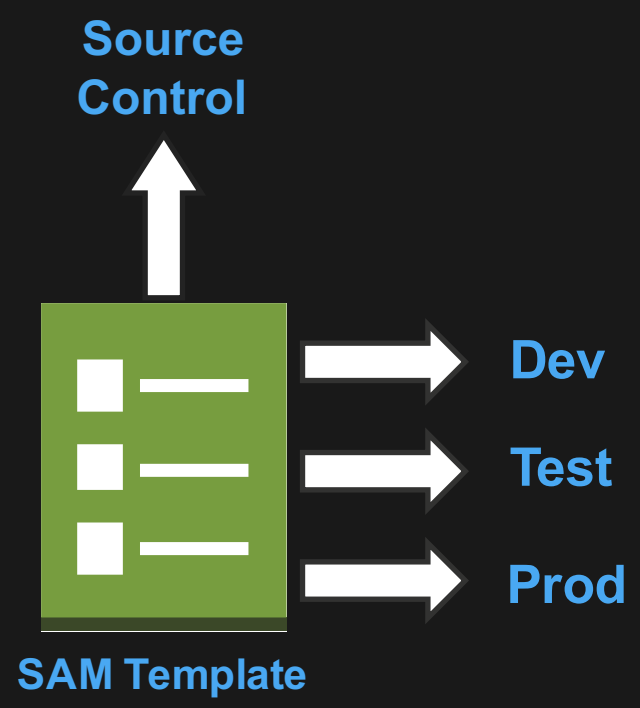
Useful for: centralized environment variables, secrets control, feature flags

```
from __future__ import print_function
import json
import boto3
ssm = boto3.client('ssm', 'us-east-1')

def get_parameters():
    response = ssm.get_parameters(
        Names=['LambdaSecureString'],WithDe
cryption=True
    )
    for parameter in
response['Parameters']:
        return parameter['Value']

def lambda_handler(event, context):
    value = get_parameters()
    print("value1 = " + value)
    return value  # Echo back the first key
value
```

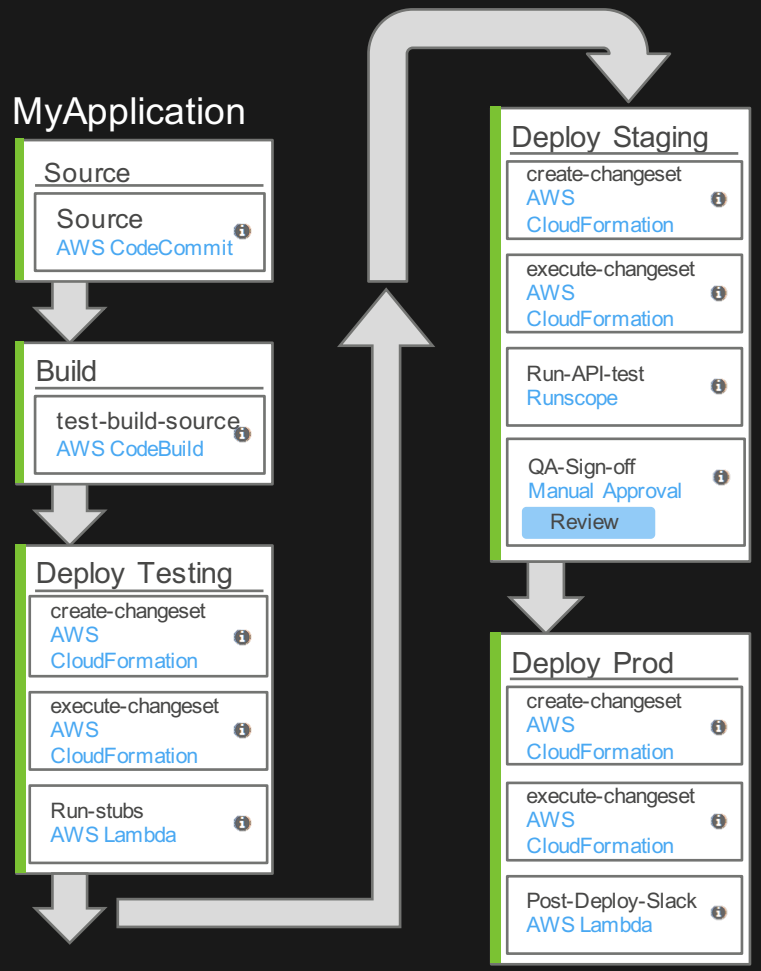
SAM Best Practices

Create multiple environments from one template:

- Use Parameters and Mappings when possible to build dynamic templates based on user inputs and pseudo parameters such as AWS::Region
- Use ExportValue & ImportValue to share resource information across stacks
- Build out multiple environments, such as for Development, Test, Production and even DR using the same template, even across accounts



Local development should lead to a formal pipeline!



This CodePipeline pipeline:

- Five Stages
- Builds code artifact w/ CodeBuild
- Three deployed to “Environments”
- Uses SAM/CloudFormation to deploy artifact and other AWS resources
- Has Lambda custom actions for running my own testing functions
- Integrates with a 3rd party tool/service
- Has a manual approval before deploying to production

Sounds easy!

https://secure.flickr.com/photos/macwagen/94975613

Serverless application

EVENT SOURCE

Changes in data state

Requests to endpoints

Changes in resource state

FUNCTION

Node.js
Python
Java
C#

SERVICES (ANYTHING)

aws.amazon.com/serverless

Menu | amazon web services | Products | Solutions | Pricing | Software | Support | Customers | More | English | My Account | Sign In to the Console

Serverless Computing and Applications

Build and run applications without thinking about servers

Get Started

AWS Lambda | Getting Started Resources | Use Cases | Developer Tools | Partner Solutions | Compute Blog

Build Serverless Applications for Production

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 virtually 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

aws.amazon.com/serverless/developer-tools

Menu | amazon web services | Products | Solutions | Pricing | Resources | More | English | My Account | Sign In to the Console

Serverless Application Developer Tooling

Tools for serverless application and AWS Lambda developers

Get Started

Frameworks | CI/CD | Monitoring & Performance | Local Testing and IDEs | Partner Solutions | Serverless Computing

AWS and its partner ecosystem provide tools and services which help you develop serverless applications on AWS Lambda and other AWS services. These frameworks, deployment tools, SDKs, IDE plugins, and monitoring solutions help you rapidly build, test, deploy, and monitor serverless applications. Below is a selection of tools that you can use for your serverless application development cycle.

Chris Munns
munns@amazon.com
@chrismunns

https://www.flickr.com/photos/theredproject/3302110152/

https://secure.flickr.com/photos/dullhunk/202872717/