Optimizing Your Serverless Applications

Chris Munns
Principal Developer Advocate
AWS Serverless

About me:

Chris Munns - munns@amazon.com, @chrismunns

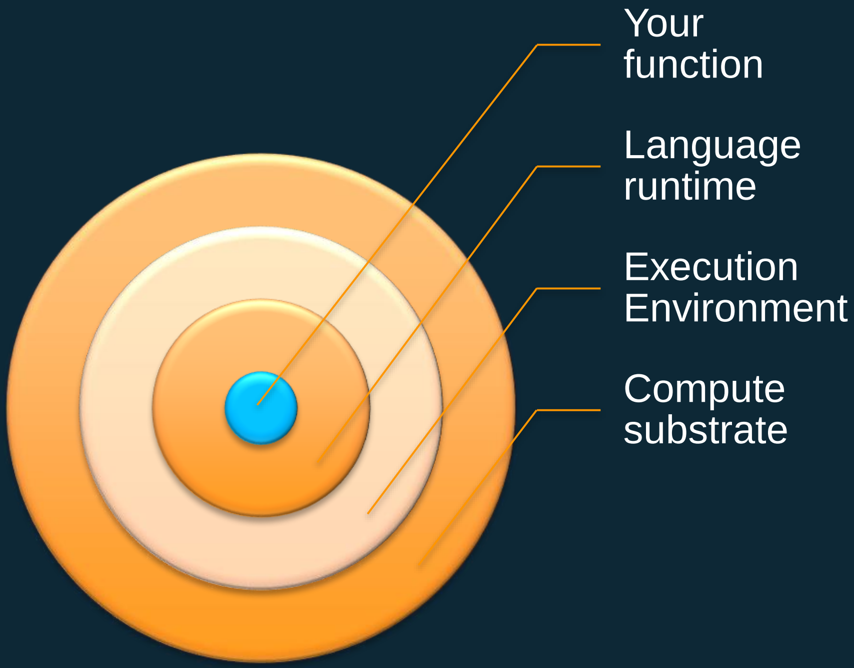
- Principal Developer Advocate - Serverless
- New Yorker
- Previously:
  - AWS Business Development Manager – DevOps, July ’15 - Feb ‘17
  - AWS Solutions Architect Nov, 2011- Dec 2014
  - Formerly on operations teams @Etsy and @Meetup
  - Little time at a hedge fund, Xerox and a few other startups
- 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

Today’s focus:

Anatomy of a Lambda function



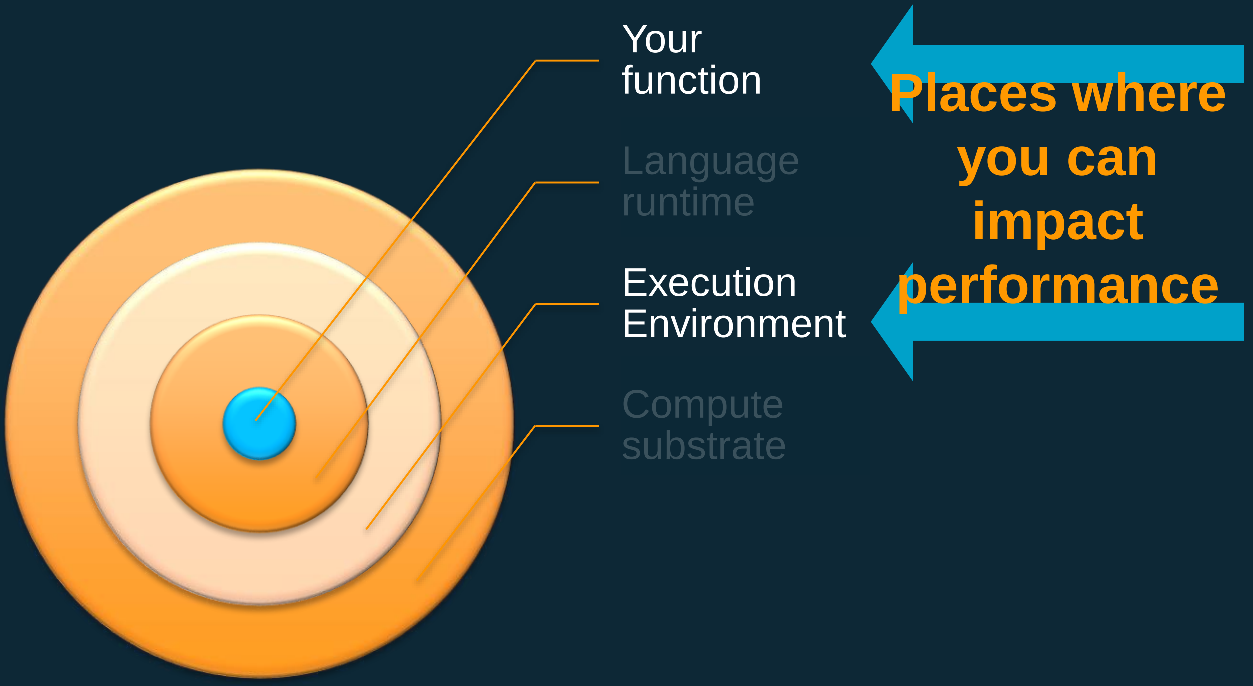
Your function

Language runtime

Execution Environment

Compute substrate

Anatomy of a Lambda function



Your function

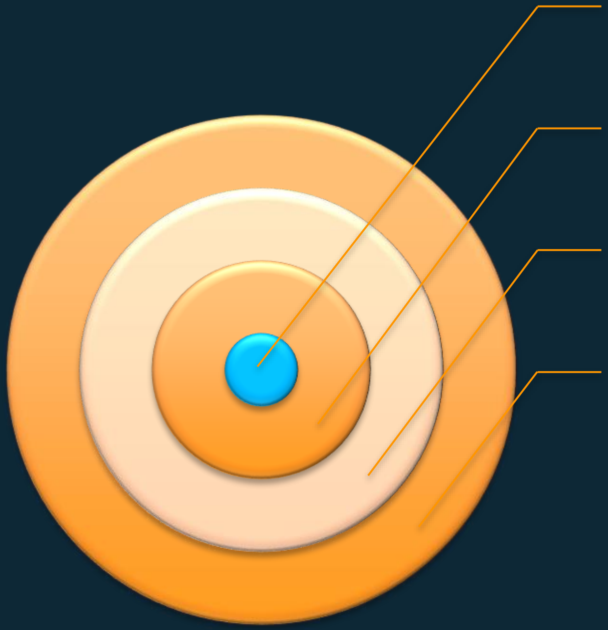
Language runtime

Execution Environment

Compute substrate

Places where you can impact performance

Anatomy of a Lambda function



Your function

Language runtime

Execution Environment

Compute substrate

Serverless applications

AWS Lambda

Serverless applications

Function

Node.js
Python
Java
C#
Go
Ruby
Runtime API

Anatomy of a Lambda function

Handler() function

Function to be executed upon invocation

Event object

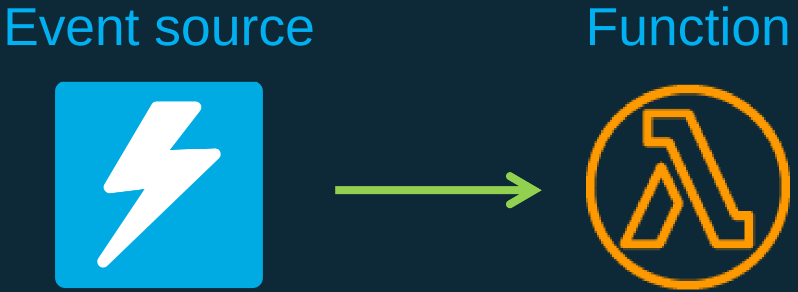
Data sent during Lambda function Invocation

Context object

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

```
import json

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

Serverless applications



Event source

Function

Changes in data state

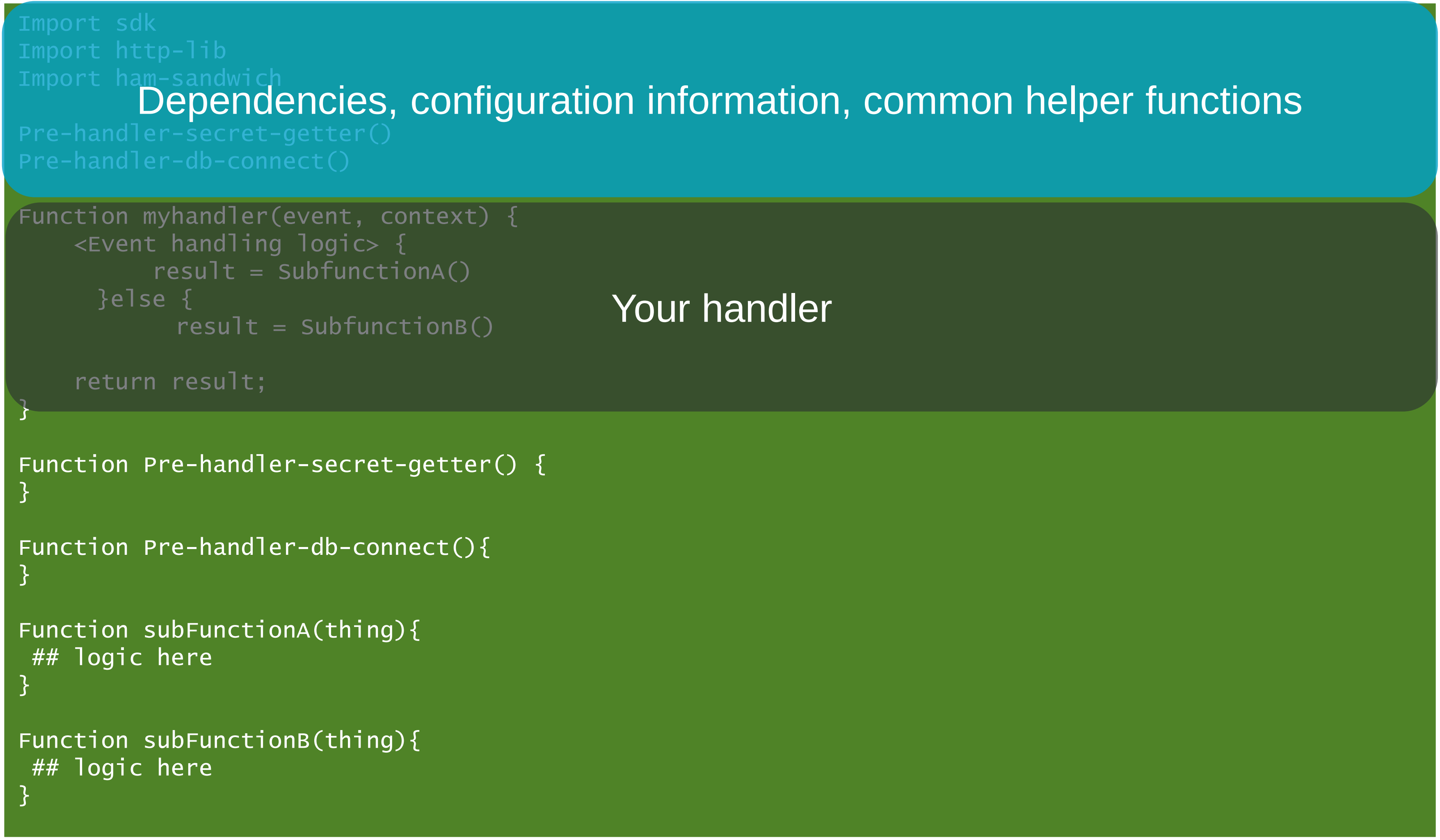
Requests to endpoints

Changes in Resource state

Node.js
Python
Java
C#
Go
Ruby
Runtime API

```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()

Function myhandler(event, context) {
    <Event handling logic> {
          result = SubfunctionA()
      }else {
           result = SubfunctionB()

    return result;
}

Function Pre-handler-secret-getter() {
}

Function Pre-handler-db-connect(){
}

Function subFunctionA(thing){
 ## logic here
}

Function subFunctionB(thing){
 ## logic here
}
```

Your handler



Dependencies, configuration information, common helper functions

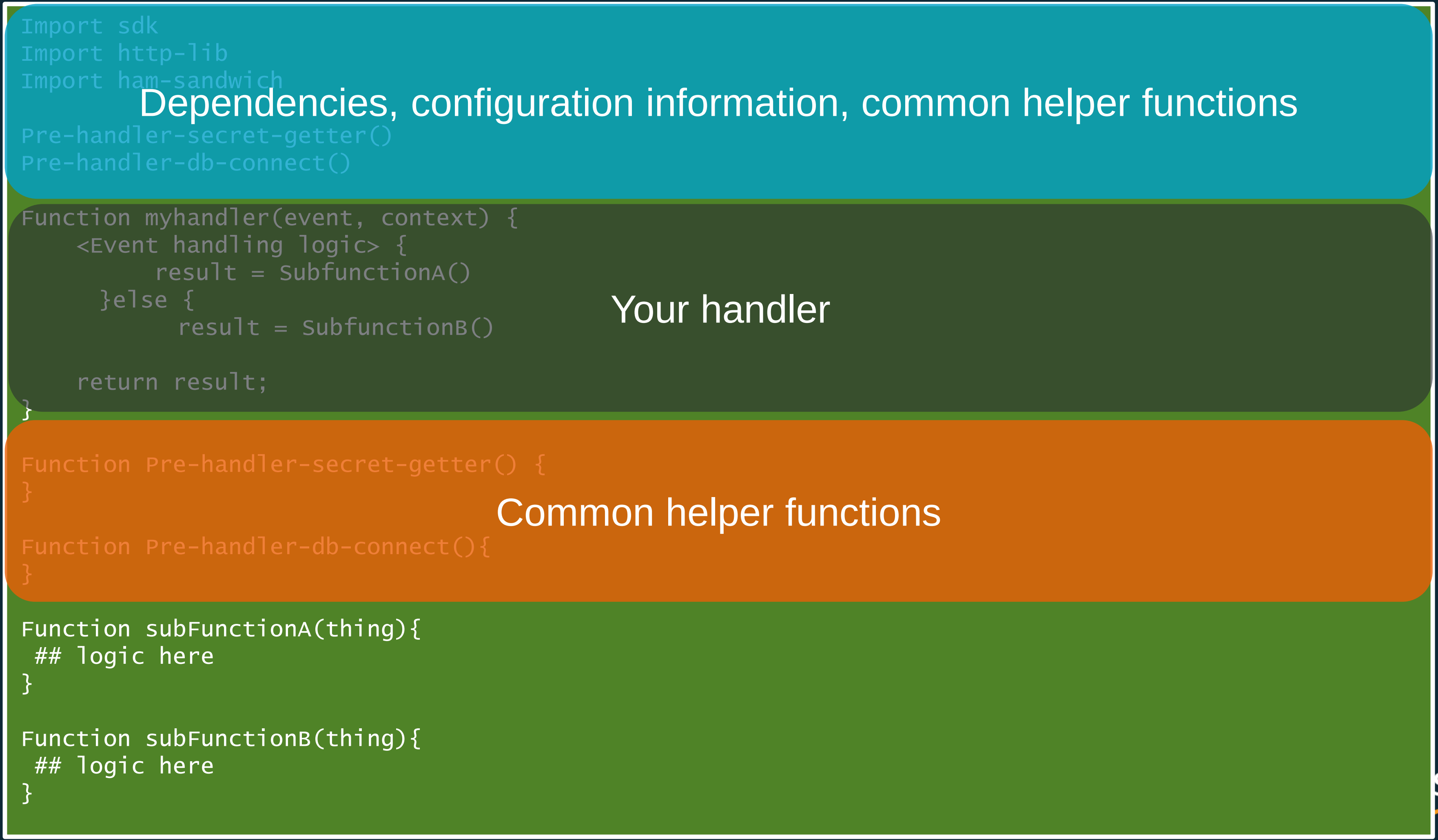
Your handler

```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()

Function myhandler(event, context) {
    <Event handling logic> {
          result = SubfunctionA()
      }else {
           result = SubfunctionB()

    return result;
}

Function Pre-handler-secret-getter() {
}

Function Pre-handler-db-connect(){
}

Function subFunctionA(thing){
 ## logic here
}

Function subFunctionB(thing){
 ## logic here
}
```

Pre-handler code, dependencies, variables

- Import only what you need
    - Where possible trim down SDKs and other libraries to the specific bits required
- Pre-handler code is great for establishing connections, but be prepared to then handle reconnections in further executions
- REMEMBER – execution environments are reused
    - Lazily load variables in the global scope
    - Don’t load it if you don’t need it – cold starts are affected
    - Clear out used variables so you don’t run into left-over state

```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()

Function myhandler(event,
context) {
....
```



Dependencies, configuration information, common helper functions

```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()
```

Your handler

```
Function myhandler(event, context) {
    <Event handling logic> {
          result = SubfunctionA()
      }else {
            result = SubfunctionB()

    return result;
}
```

Common helper functions

```
Function Pre-handler-secret-getter() {
}

Function Pre-handler-db-connect(){
}

Function subFunctionA(thing){
 ## logic here
}

Function subFunctionB(thing){
 ## logic here
}
```

AWS 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)

AWS Systems Manager – Parameter Store

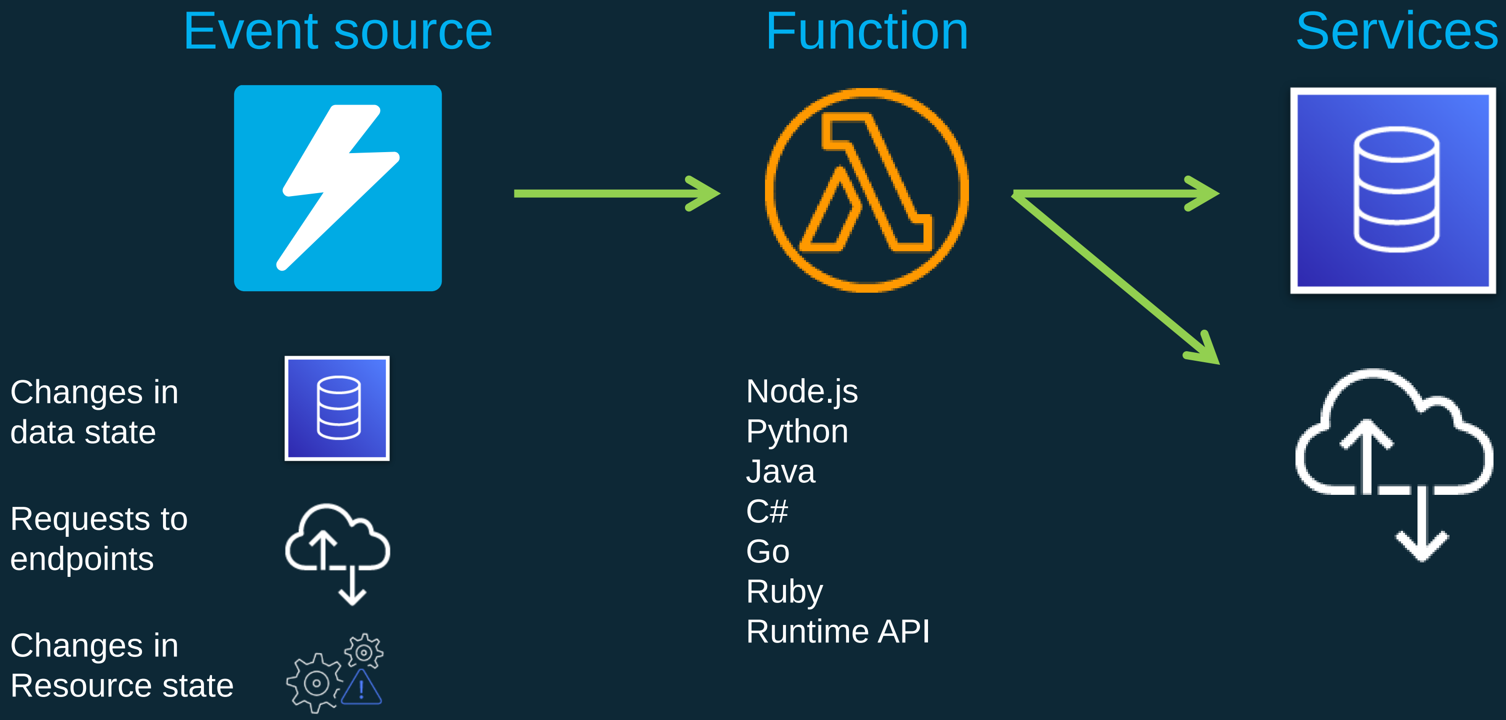
Centralized store to manage your configuration data

- 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
- Integrated with AWS Secrets Manager
- 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'],WithDec
ryption=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
```

Serverless applications



Event source

Function

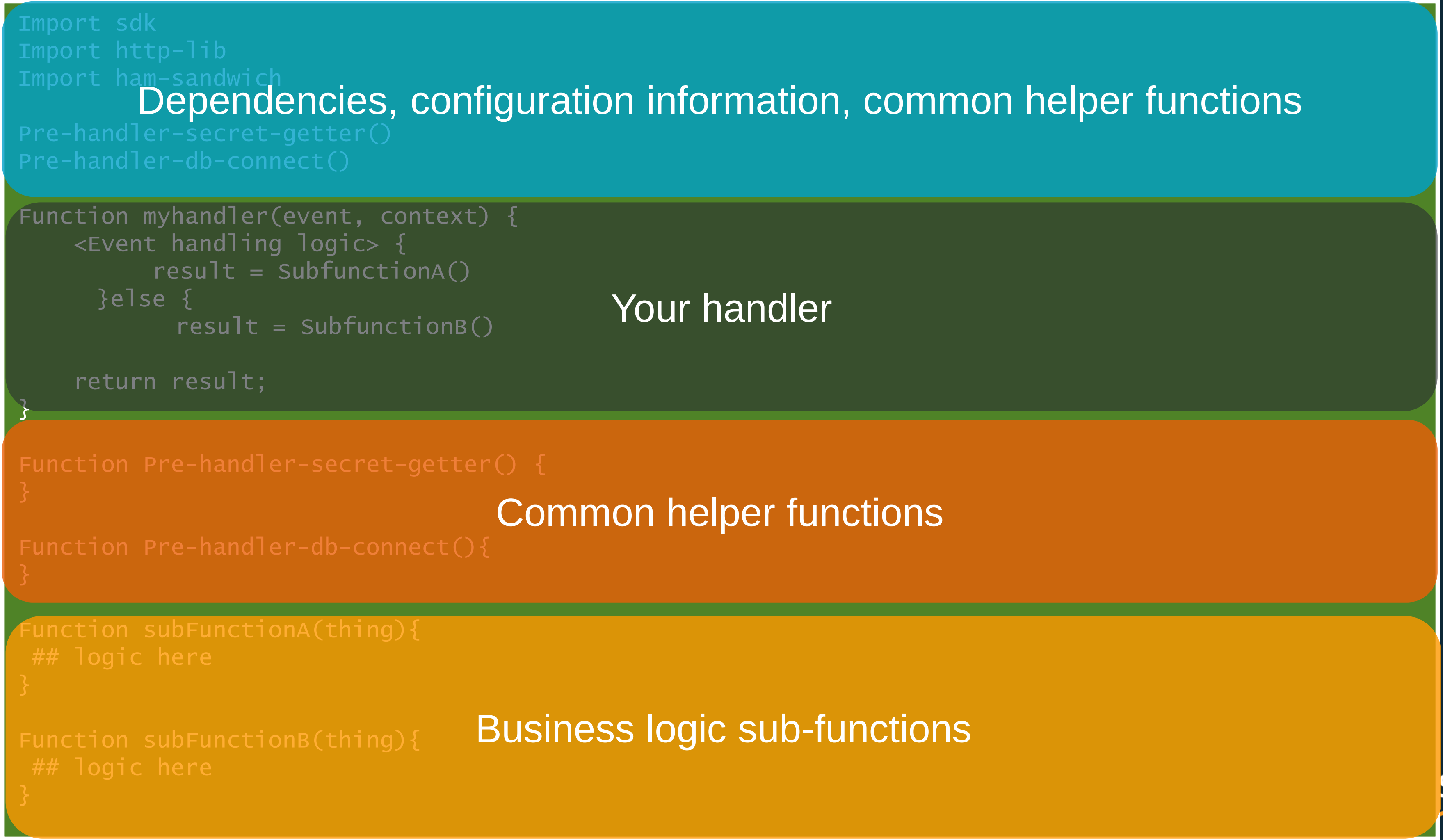
Services

Changes in data state

Requests to endpoints

Changes in Resource state

Node.js
Python
Java
C#
Go
Ruby
Runtime API



```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()
```

Dependencies, configuration information, common helper functions

```
Function myhandler(event, context) {
    <Event handling logic> {
          result = SubfunctionA()
      }else {
            result = SubfunctionB()

    return result;
}
```

Your handler

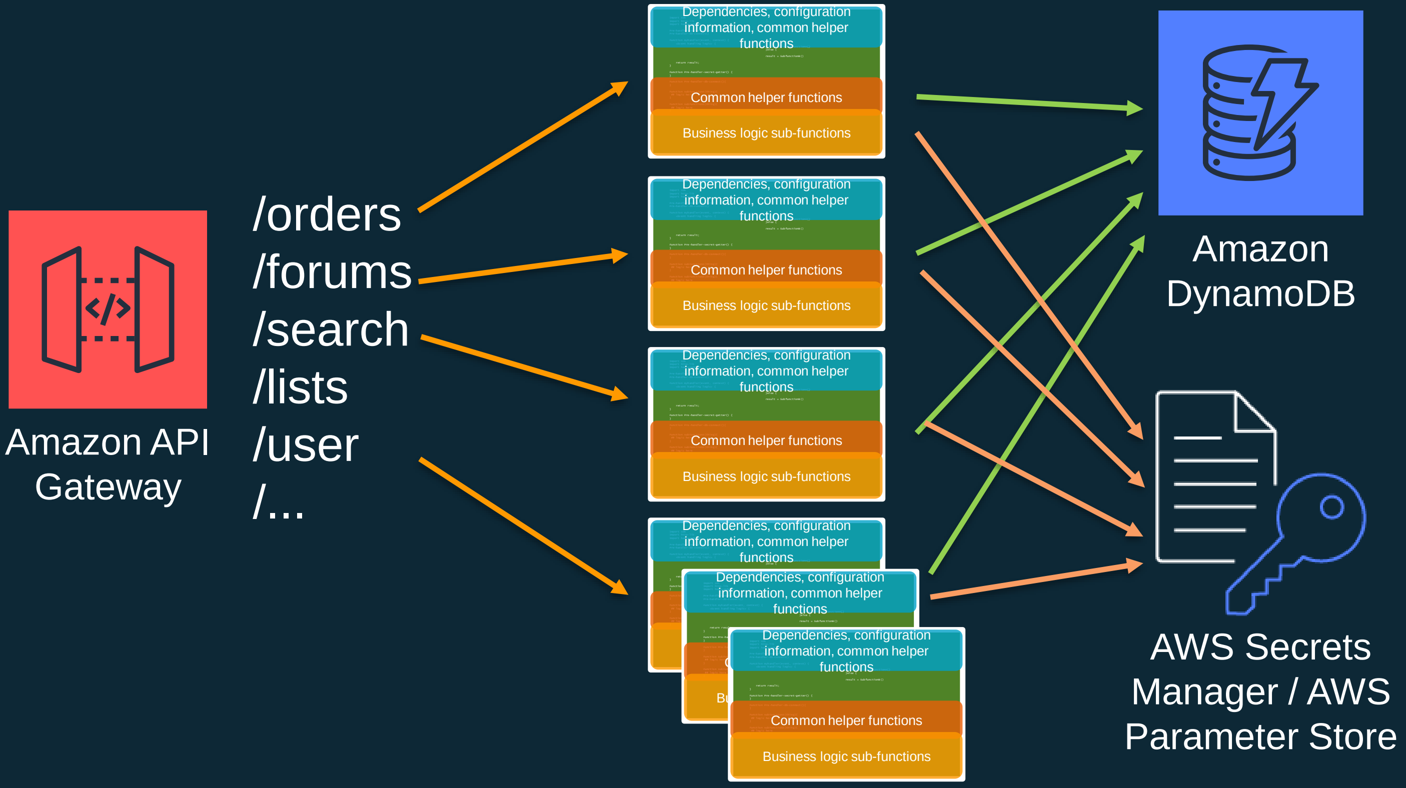
```
Function Pre-handler-secret-getter() {
}

Function Pre-handler-db-connect(){
}
```

Common helper functions

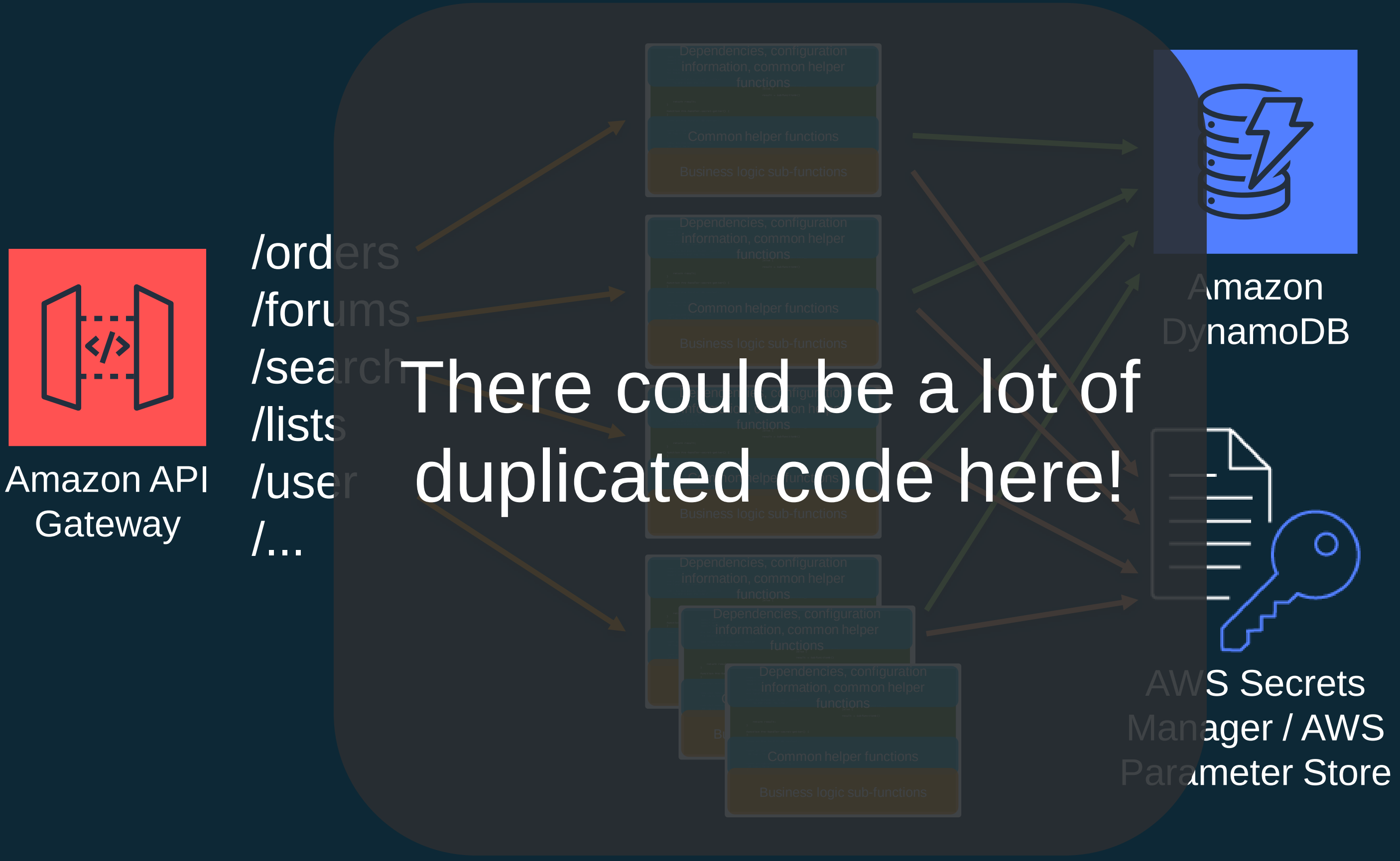
```
Function subFunctionA(thing){
 ## logic here
}

Function subFunctionB(thing){
 ## logic here
}
```

Business logic sub-functions

Anatomy of a serverless application


Amazon API Gateway
/orders
/forums
/search
/lists
/user
/...
Dependencies, configuration information, common helper functions
Common helper functions
Business logic sub-functions
Dependencies, configuration information, common helper functions
Common helper functions
Business logic sub-functions
Dependencies, configuration information, common helper functions
Common helper functions
Business logic sub-functions
Dependencies, configuration information, common helper functions
Dependencies, configuration information, common helper functions
Dependencies, configuration information, common helper functions
Common helper functions
Business logic sub-functions
Amazon DynamoDB
AWS Secrets Manager / AWS Parameter Store

Anatomy of a serverless application

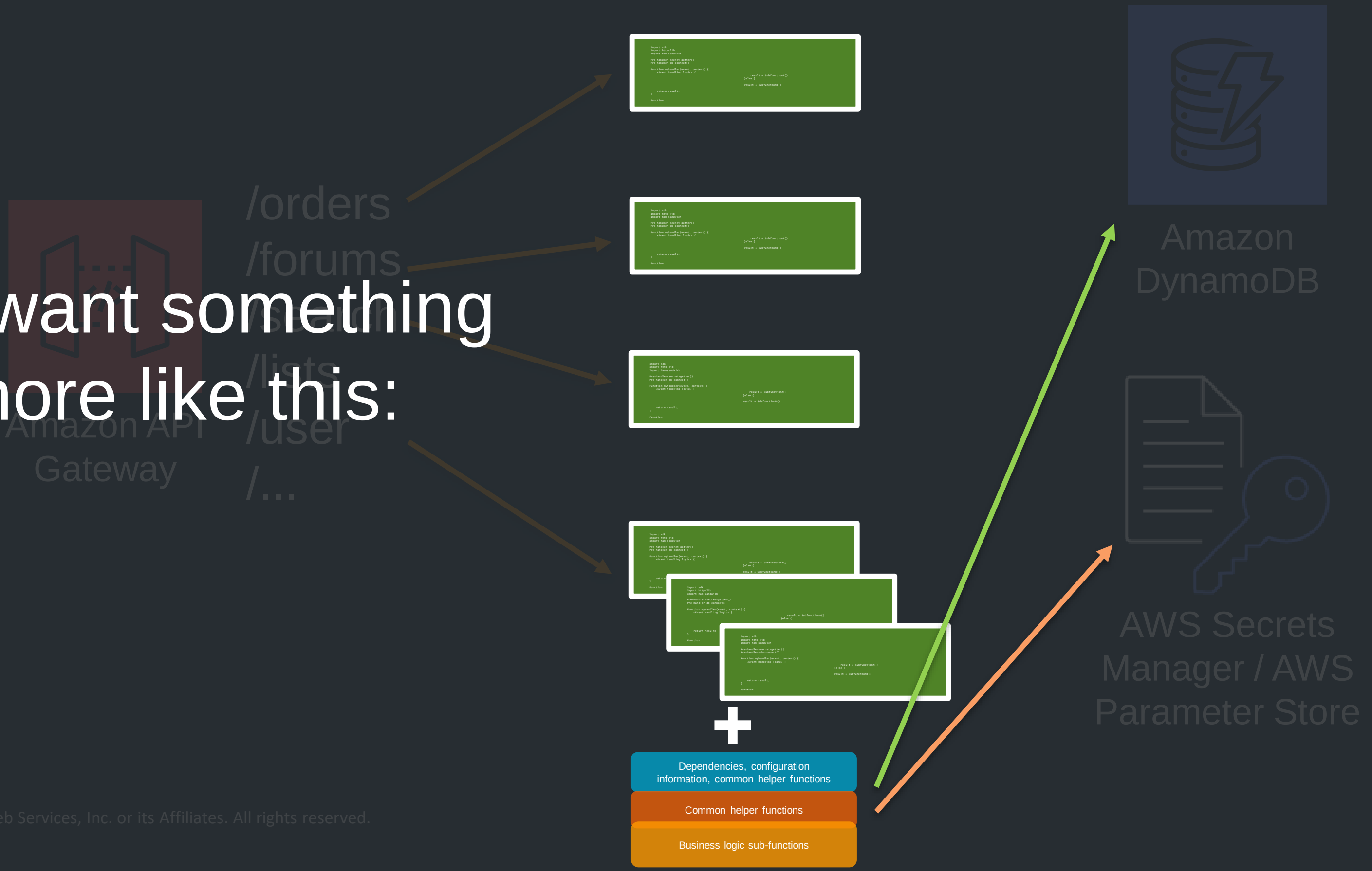


There could be a lot of duplicated code here!

Amazon API Gateway
/orders
/forums
/search
/lists
/user
/...
Dependencies, configuration information, common helper functions
Common helper functions
Business logic sub-functions
Amazon DynamoDB
AWS Secrets Manager / AWS Parameter Store

Anatomy of a serverless application

We want something more like this:



```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()

Function myhandler(event, context) {
    <Event handling logic> {
                                                       result = SubfunctionA()
                                                   }else {

                                                   result = SubfunctionB()


    return result;
}

Function
```

Dependencies, configuration information, common helper functions

Common helper functions

Business logic sub-functions

Lambda Layers

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

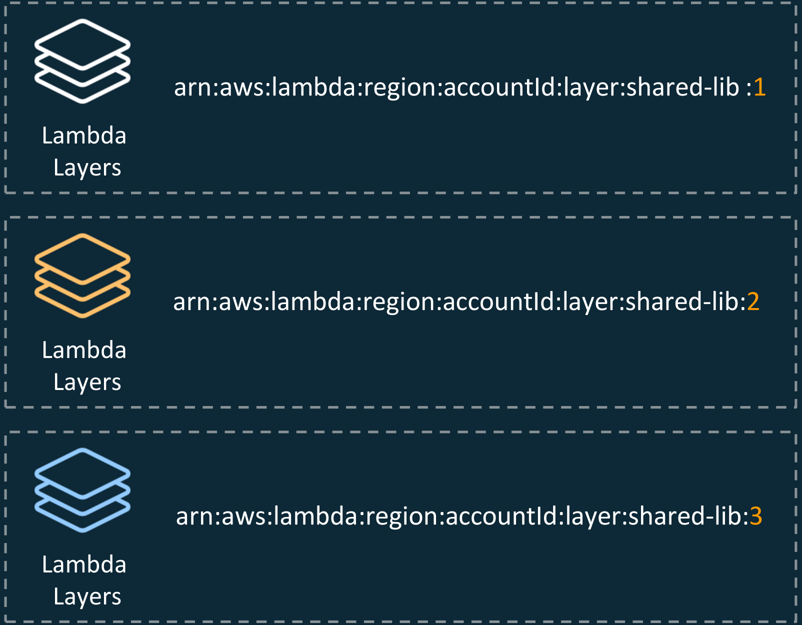
Layer can be anything: dependencies, training data, configuration files, etc

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

Built in support for secure sharing by ecosystem

Using Lambda Layers

- Put common components in a ZIP file and upload it as a Lambda Layer
- Layers are immutable and can be versioned to manage updates
- When a version is deleted or permissions to use it are revoked, functions that used it previously will continue to work, but you won't be able to create new ones
- You can reference up to five layers, one of which can optionally be a custom runtime



arn:aws:lambda:region:accountId:layer:shared-lib :1

arn:aws:lambda:region:accountId:layer:shared-lib:2

arn:aws:lambda:region:accountId:layer:shared-lib:3

How Lambda Layers Work

Order is important because each layer is a ZIP file, and they are all extracted in the same path

- /opt
- Each layer can potentially overwrite the previous one

This approach can be used to customize the environment

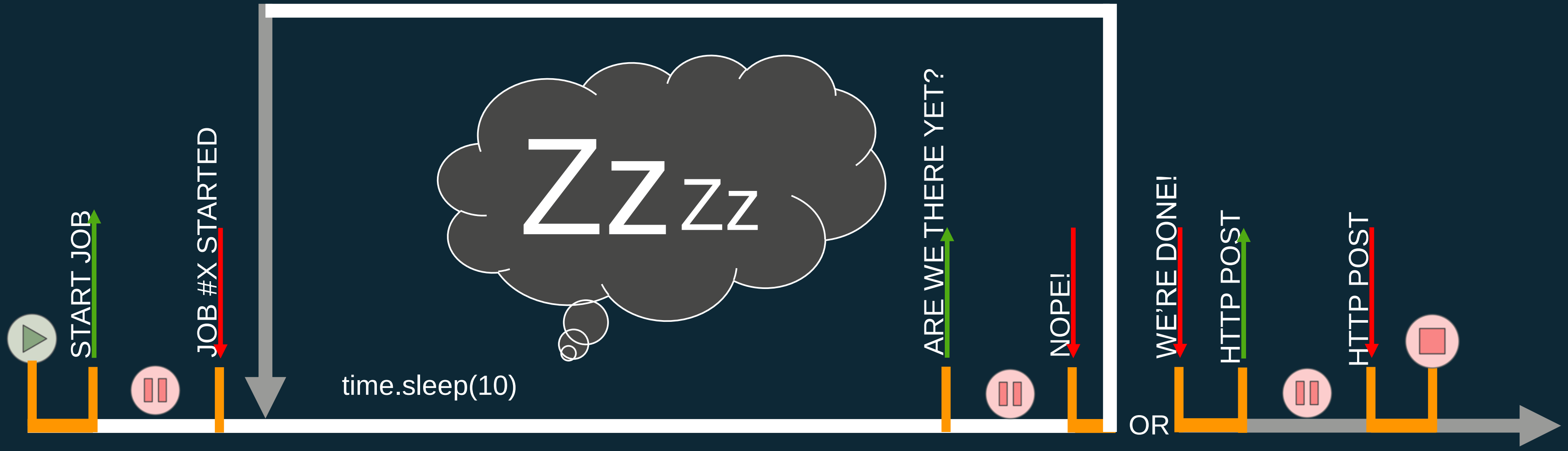
- For example, the first layer can be a custom runtime and the second layer adds specific versions of the libraries you need

The storage of your Lambda Layers takes part in the AWS Lambda Function storage per region limit (75GB)

Concise function logic

- Separate Lambda handler (entry point) from core logic
  - Providers cleaner starting point for re-use of code
- Use functions to **TRANSFORM**, not **TRANSPORT**
  - Use purposeful built services for communication fan-out, message handling, data replication, writing to data stores/databases
- Read only what you need. For example:
  - Message filters in Amazon SNS
  - Fine grained rules in Amazon EventBridge
  - Query filters in Amazon RDS Aurora
  - Use Amazon S3 Select
  - Properly indexed databases

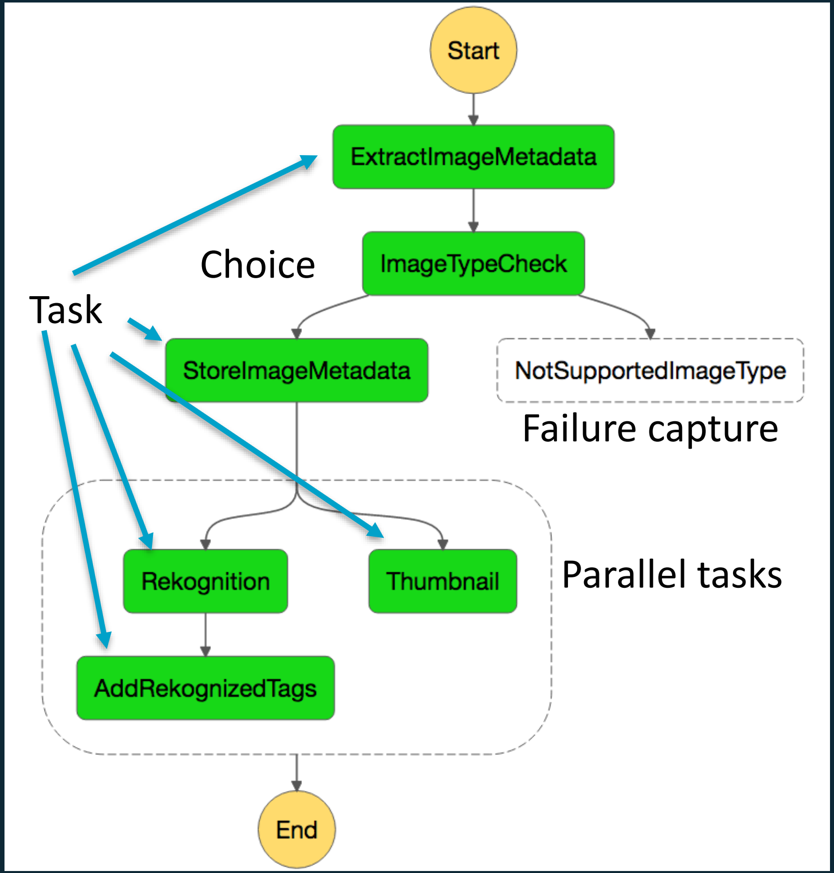
No orchestration in code



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



Step Functions: Integrations

AWS Step Functions

Simplify building workloads such as order processing, report generation, and data analysis

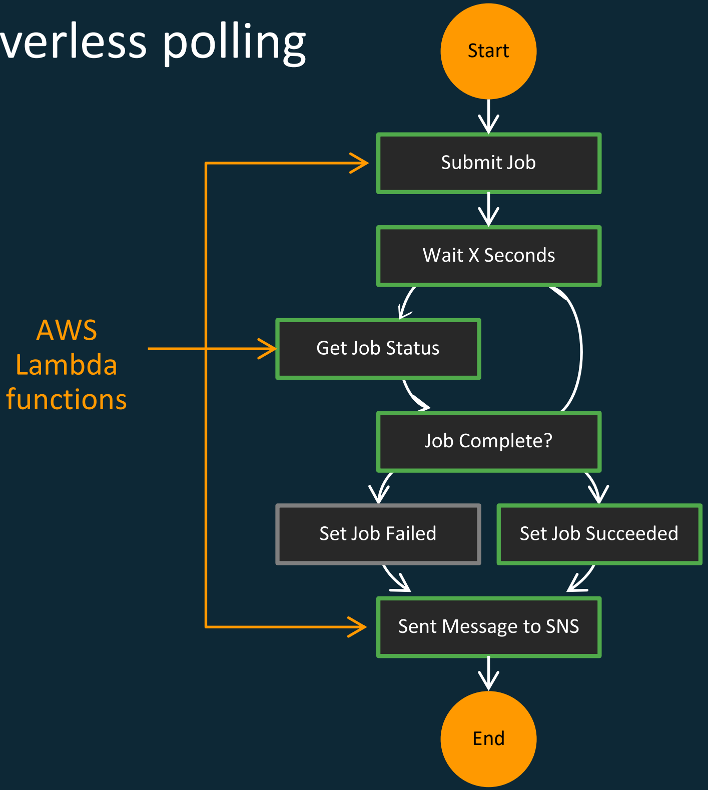
Write and maintain less code; add services in minutes

More service integrations:

- Amazon Simple Notification Service
- Amazon Simple Queue Service
- Amazon SageMaker
- AWS Glue
- AWS Batch
- Amazon Elastic Container Service
- AWS Fargate

Simpler integration, less code

With serverless polling


Start

Submit Job

Wait X Seconds

Get Job Status

Job Complete?

Set Job Failed

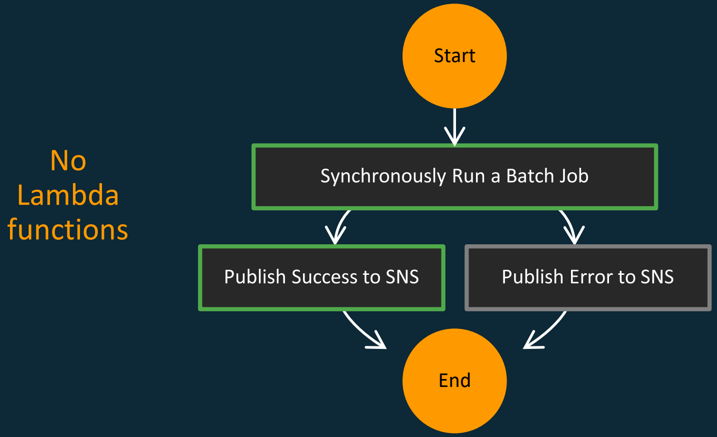
Set Job Succeeded

Sent Message to SNS

End

AWS Lambda functions

With direct service integration


Start

Synchronously Run a Batch Job

Publish Success to SNS

Publish Error to SNS

End

No Lambda functions

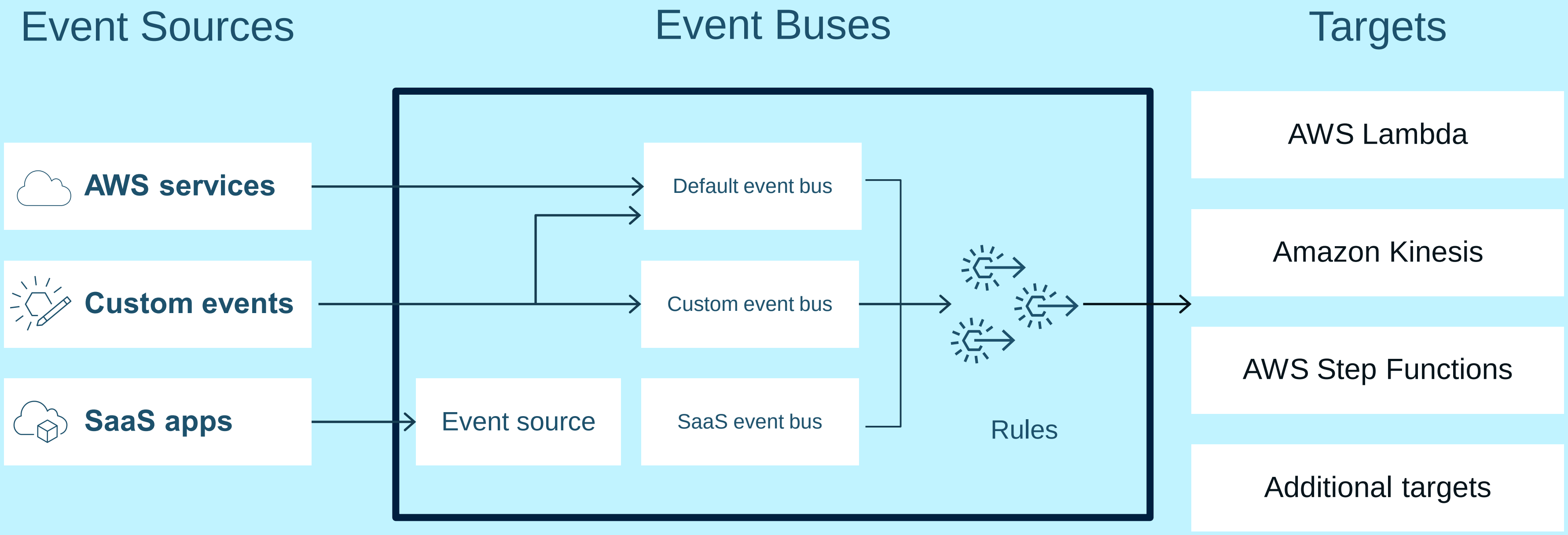
NEW!!!

Amazon EventBridge

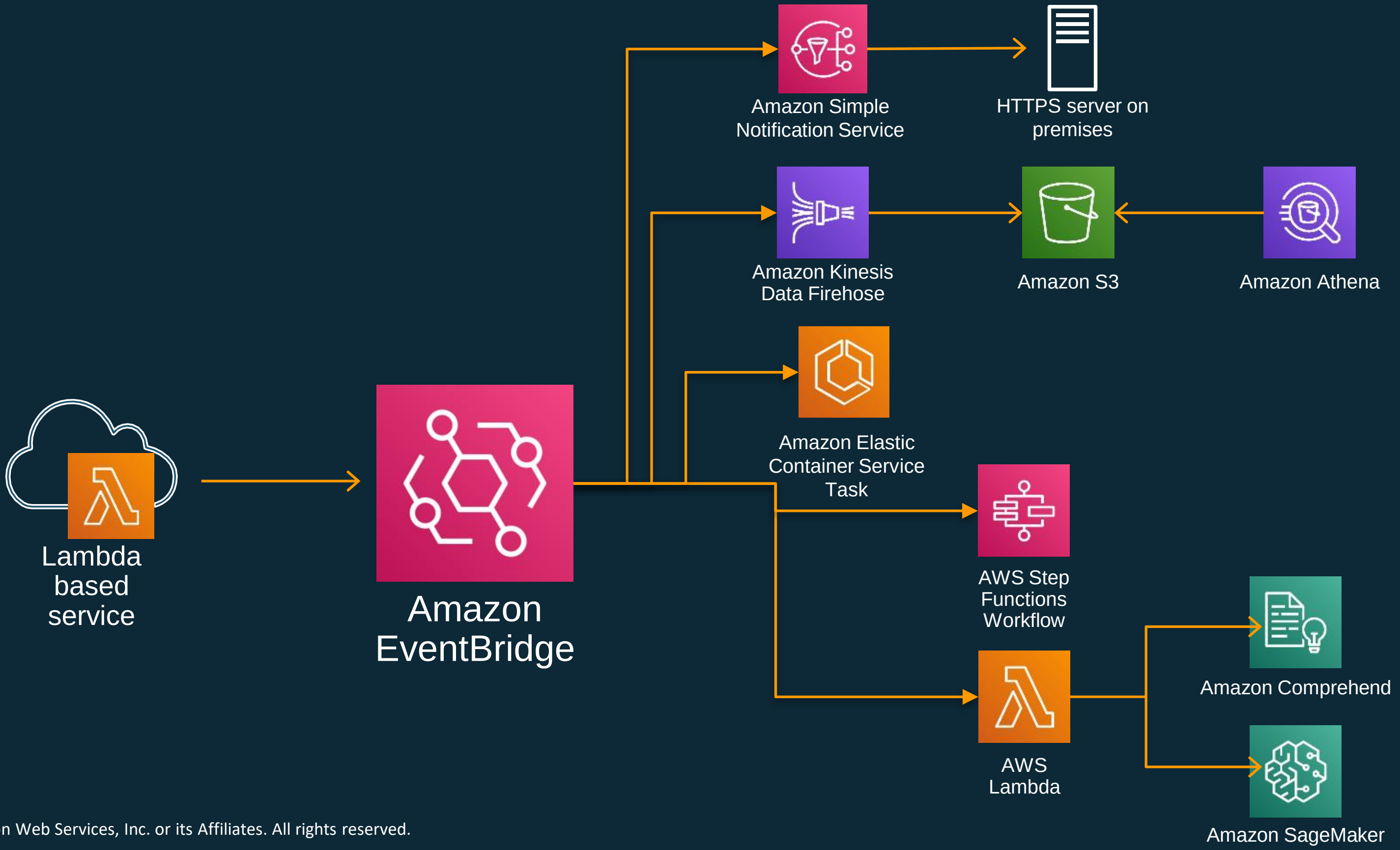
Serverless event bus for ingesting and processing data across AWS services and SaaS applications

- Removes friction of writing “point-to-point integrations”
- 90+ AWS Services as sources
- 17 AWS Services as targets
- Provides simple programming model

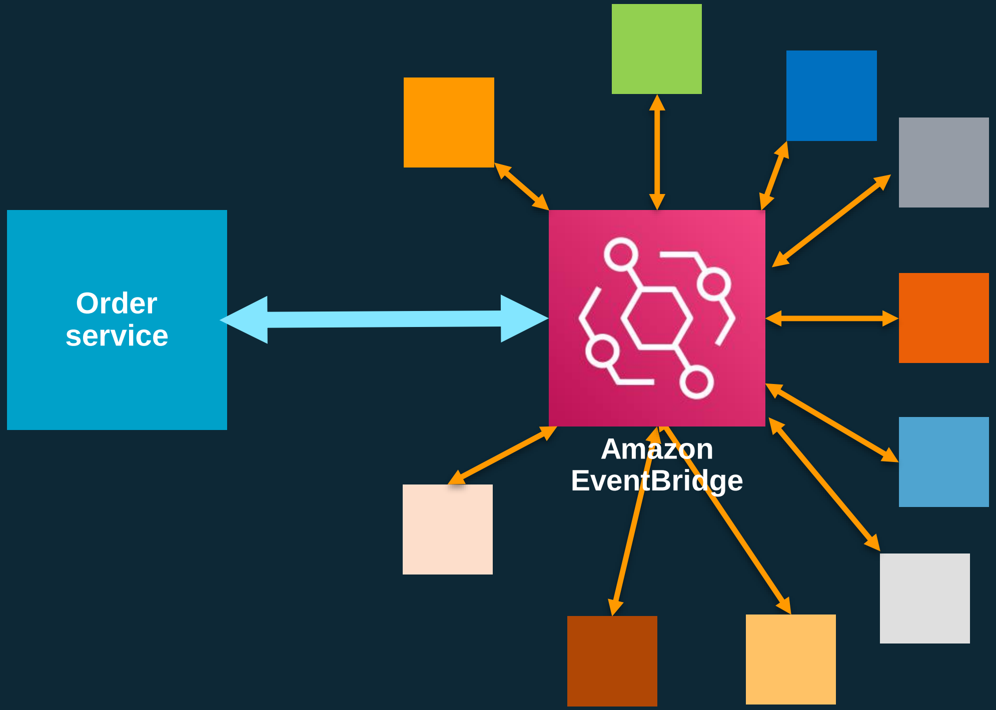
Amazon EventBridge



Event passing with Amazon EventBridge

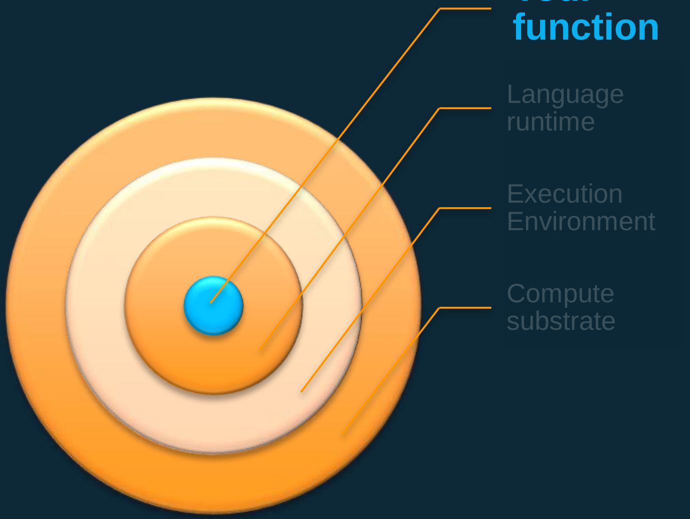


Events with Amazon EventBridge



- Your services can both produce messages onto the bus and consume just the messages they need from the bus
- Services don’t need to know about each other, just about the bus.

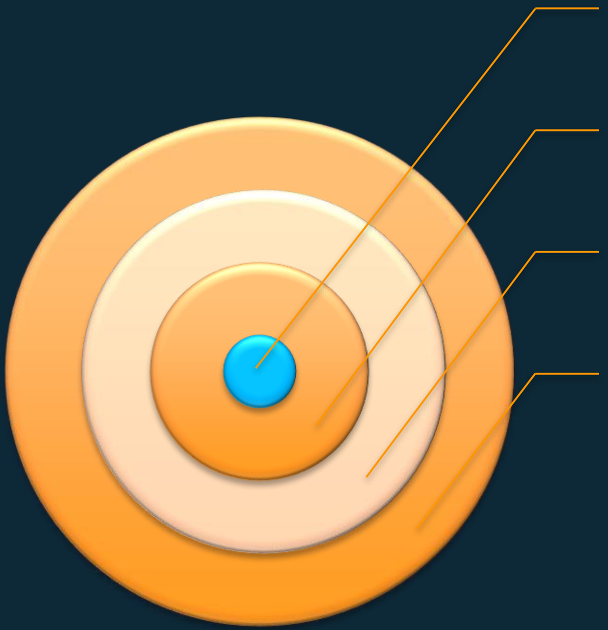
Anatomy of a Lambda function



Recap:

- Minimize dependencies
- Use pre-handler logic sparingly but strategically
- Share secrets based on application scope:
    - Single function: Env-Vars
    - Multi Function/shared environment: Parameter Store
- Think about how re-use impacts variables, connections, and dependency usage
- Layers save on code duplication and help enable standardization across functions
- Concise logic.
- Push orchestration up to Step Functions or messaging services like EventBridge, SNS, SQS, or Kinesis

Anatomy of a Lambda function



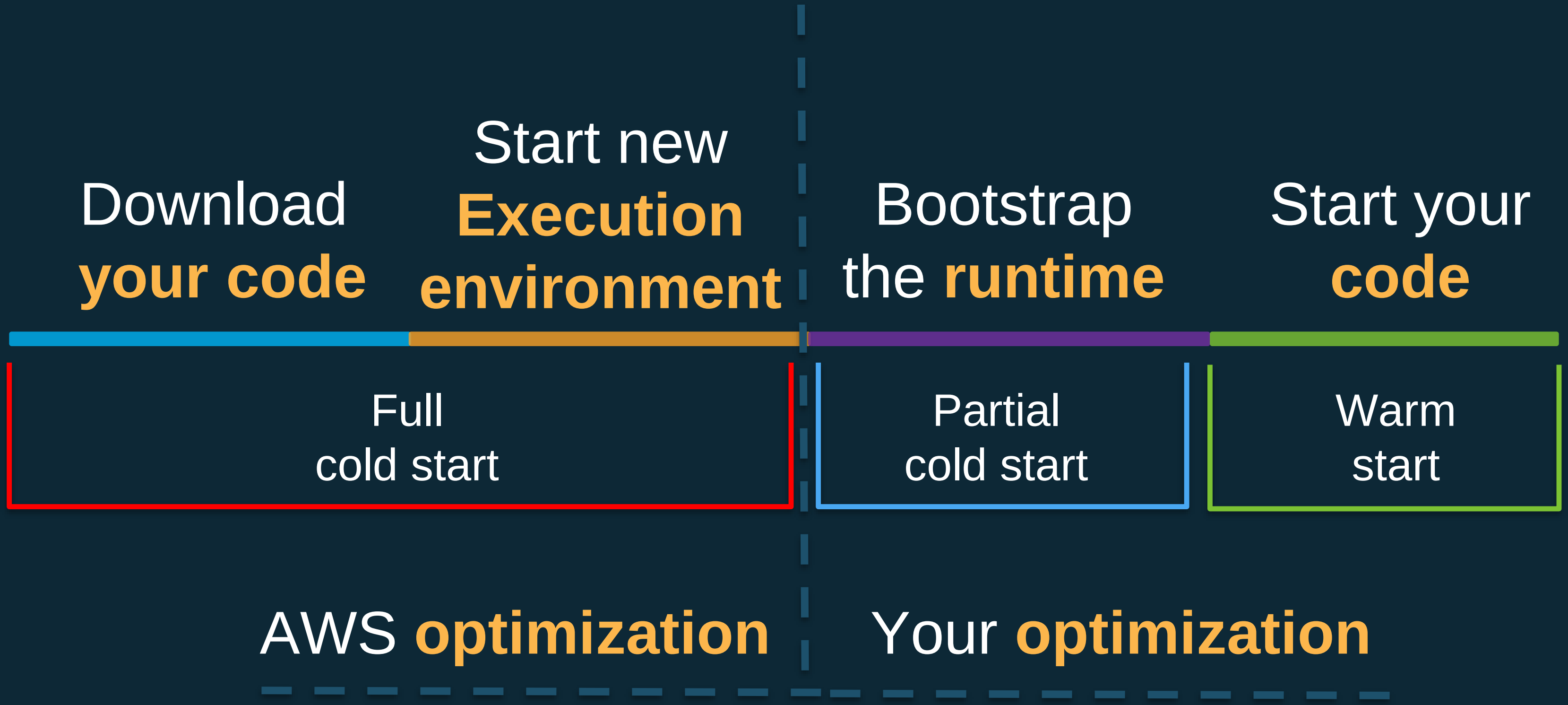
Your function

Language runtime

Execution Environment

Compute substrate

The function lifecycle



Download **your code**

Start new **Execution environment**

Bootstrap the **runtime**

Start your **code**

Full cold start

Partial cold start

Warm start

AWS **optimization**

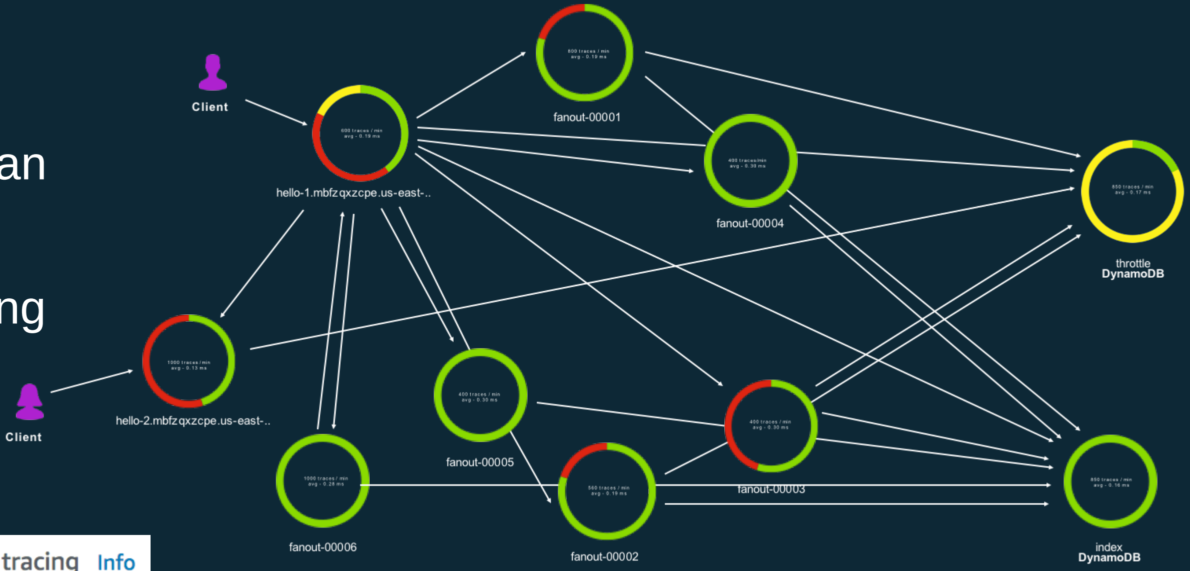
Your **optimization**

AWS X-Ray

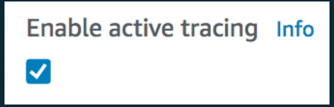
```
var AWSXRay = require(‘aws-xray-sdk-core‘);
var AWS = AWSXRay.captureAWS(require(‘aws-sdk’));
S3Client = AWS.S3();
```

Profile and troubleshoot serverless applications:

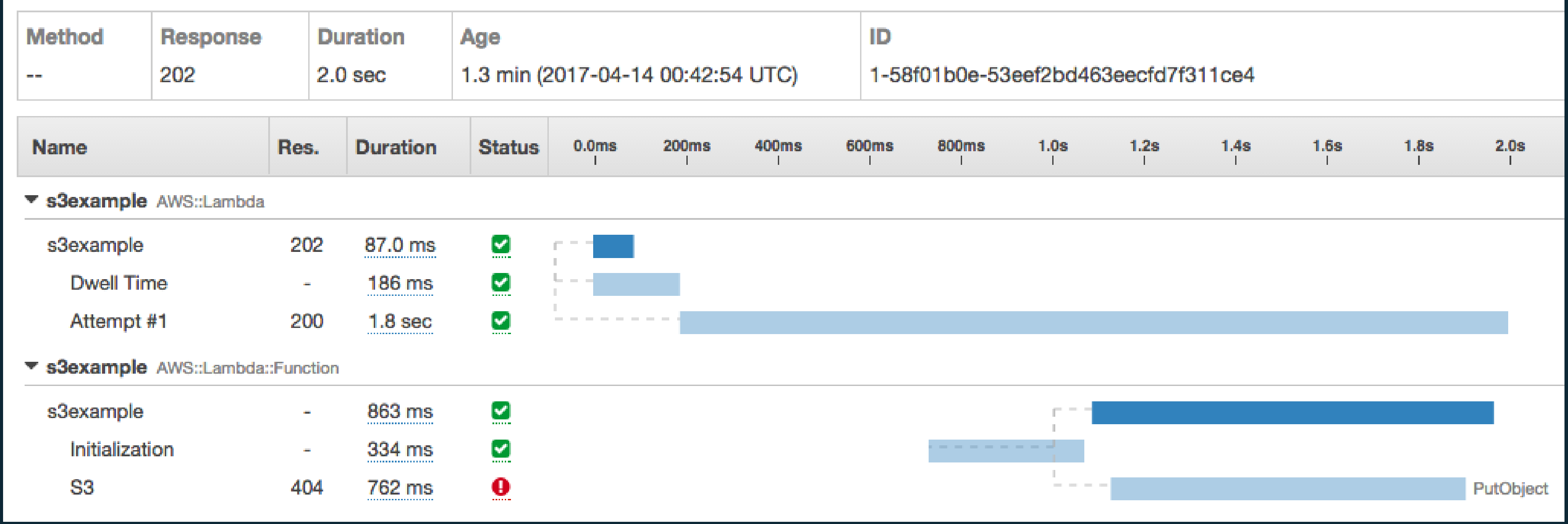
- Lambda instruments incoming requests for all supported languages and can capture calls made in code
- API Gateway inserts a tracing header into HTTP calls as well as reports data back to X-Ray itself



Enable X-Ray Tracing ☑


Enable active tracing Info
☑

X-Ray Trace Example



Method	Response	Duration	Age	ID
--	202	2.0 sec	1.3 min (2017-04-14 00:42:54 UTC)	1-58f01b0e-53eef2bd463eecfd7f311ce4

Name	Res.	Duration	Status
s3example AWS::Lambda			
s3example	202	87.0 ms	✓
Dwell Time	-	186 ms	✓
Attempt #1	200	1.8 sec	✓
s3example AWS::Lambda::Function			
s3example	-	863 ms	✓
Initialization	-	334 ms	✓
S3	404	762 ms	!

Tweak your function’s computer power

Lambda exposes only a memory control, with the **% of CPU core and network capacity** allocated to a function proportionally

Is your code CPU, Network or memory-bound? If so, it could be **cheaper** to choose more memory.

Smart resource allocation

Match resource allocation (up to 3 GB!) to logic

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

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

Green==Best **Red**==Worst

Smart resource allocation

Match resource allocation (up to 3 GB!) to logic

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

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

-10.256981sec

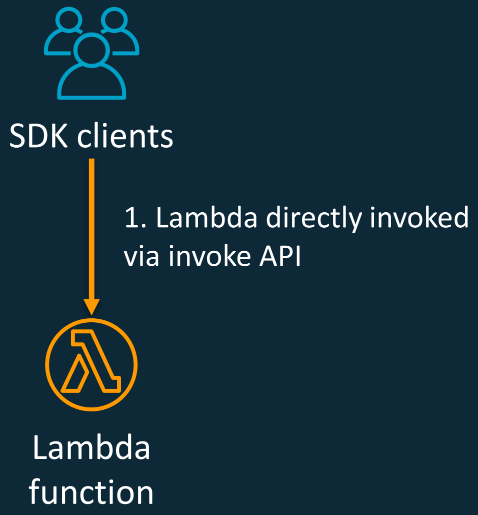
+$0.00001

Green==Best **Red**==Worst

Multithreading? Maybe!

- <1.8GB is still single core
  - CPU bound workloads won’t see gains – processes share same resources
- >1.8GB is multi core
  - CPU bound workloads will gains, but need to multi thread
- I/O bound workloads WILL likely see gains
  - e.g. parallel calculations to return

Lambda API



API provided by the Lambda service

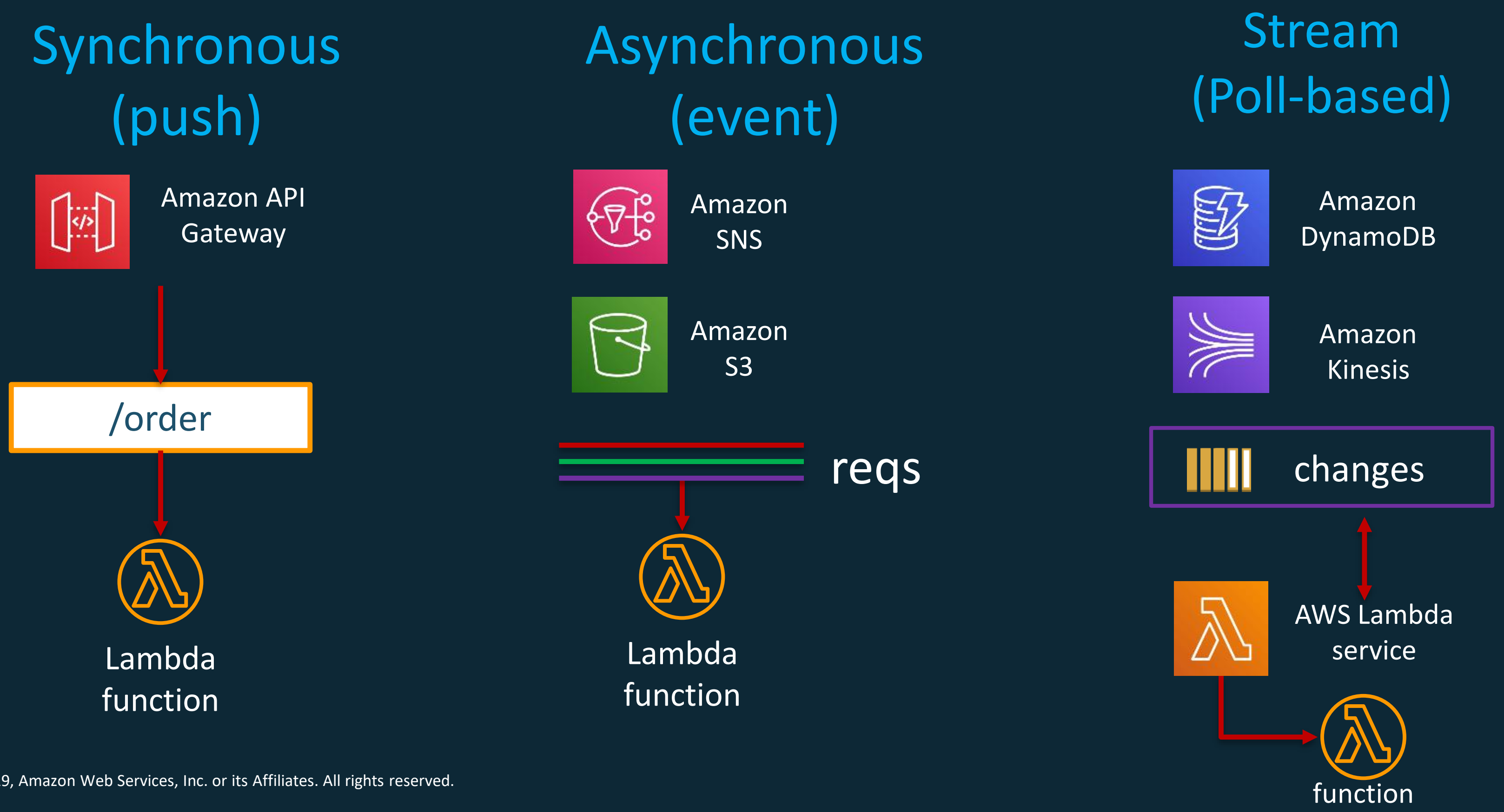
Used by all other services that invoke Lambda across all models

Supports sync and async

Can pass any event payload structure you want

Client included in every SDK

Lambda execution model



Synchronous (push)

Amazon API Gateway

/order

Lambda function

Asynchronous (event)

Amazon SNS

Amazon S3

reqs

Lambda function

Stream (Poll-based)

Amazon DynamoDB

Amazon Kinesis

changes

AWS Lambda service

function

If you don’t need a response, execute async

Use the Lambda APIs to start an asynchronous execution

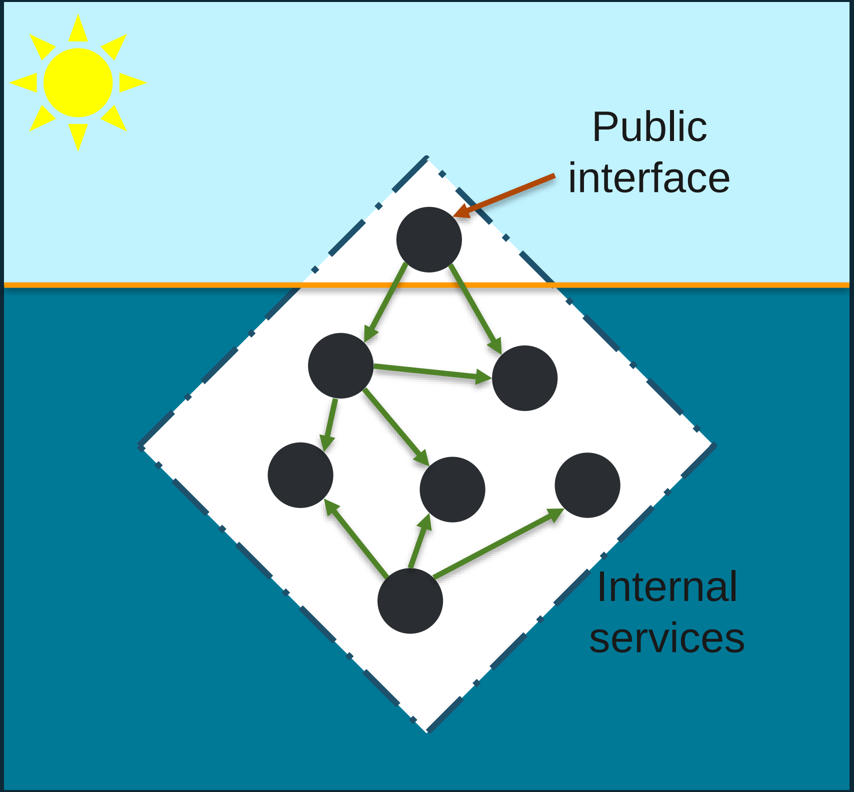
- Built-in queue (SQS behind the scenes)
- Automatic retries
- Dead letter queue for failed events

```
client = boto3.client("lambda")

client.invoke_async(
    FunctionName="test"
    InvokeArgs=json_payload
)
```

The microservices “iceberg”

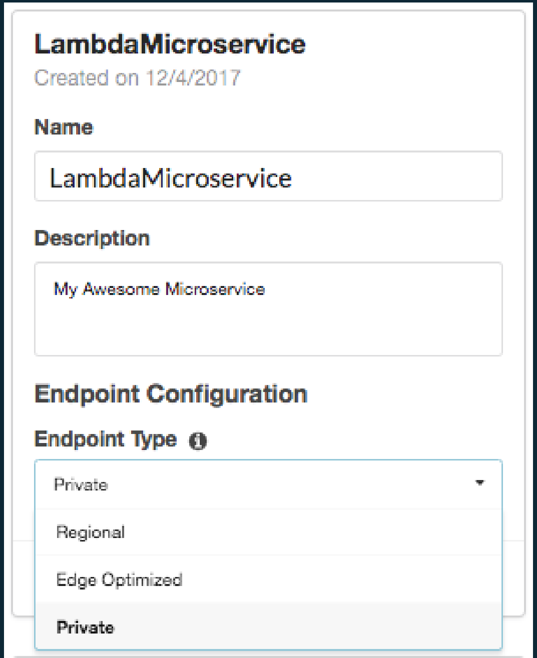
Common question: “Should every service of mine talk to another using an API?”

Maybe not!: Most microservices are internal only for a given product supporting their customer facing features. They may only need to pass messages to each other that are simple events and not need a full fledged interactive API.



Gateways and routers

- Choose suitable entry point for client applications
  - Single, custom client: Use the AWS SDK
  - In region only public API: Use regional endpoints on API Gateway
  - Calls from private microservices in a VPC: Use private endpoints on API Gateway
  - No need for a custom interface: look at a non API Gateway source
  - Fan-out: SNS or EventBridge
- Discard uninteresting events ASAP
  - S3 – Event prefix
  - SNS – Message filtering
  - EventBridge - Rules



LambdaMicroservice

Created on 12/4/2017

Name

LambdaMicroservice

Description

My Awesome Microservice

Endpoint Configuration

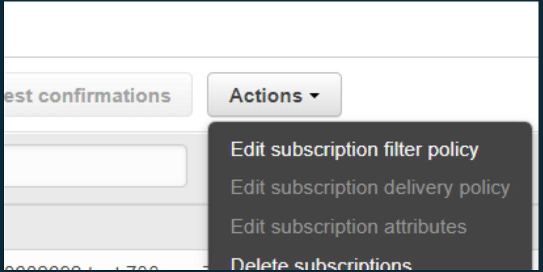
Endpoint Type

Private

Regional

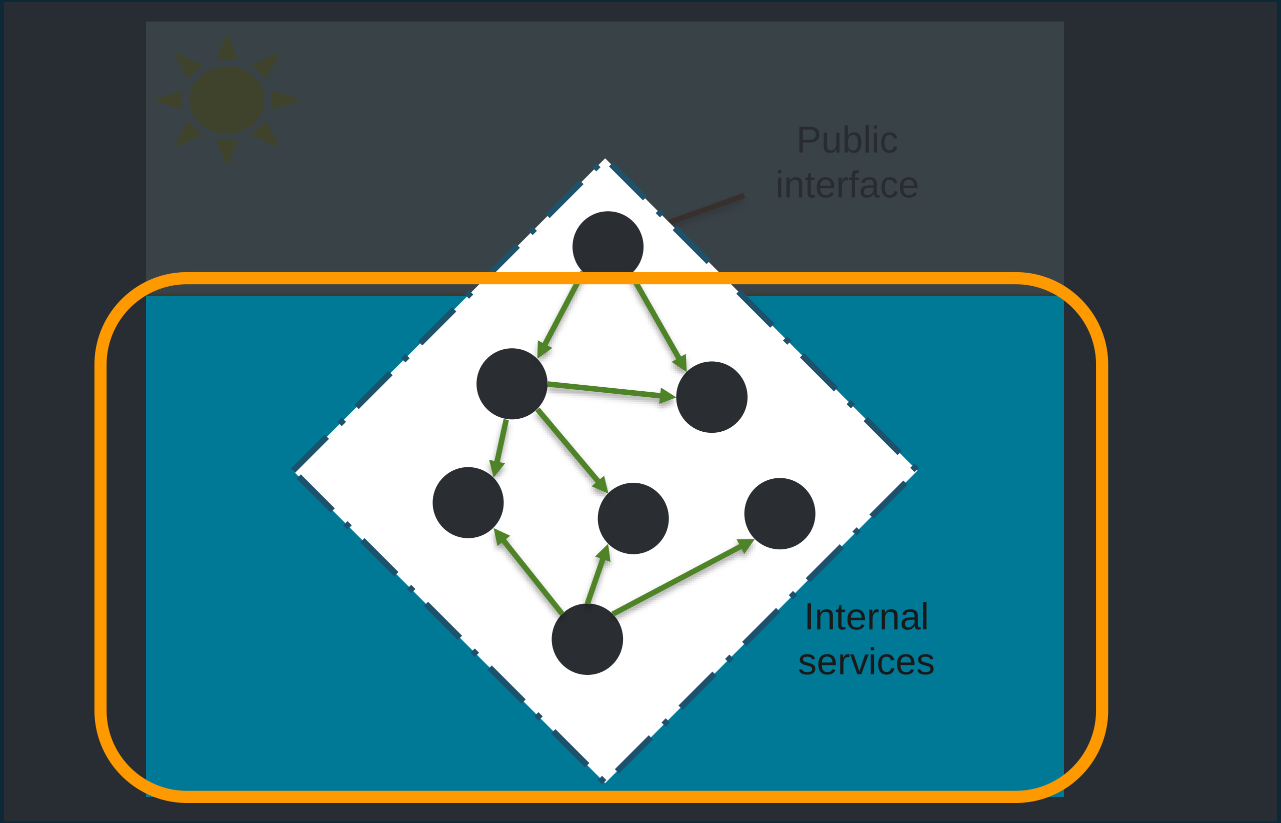
Edge Optimized

Private



est confirmations
Actions
Edit subscription filter policy
Edit subscription delivery policy
Edit subscription attributes
Delete subscriptions

Focusing below the water line



Ways to compare

Scale/Concurrency controls

Durability

Persistence

Consumption models

Retries

Pricing

Ways to compare

Scale/Concurrency controls

Durability

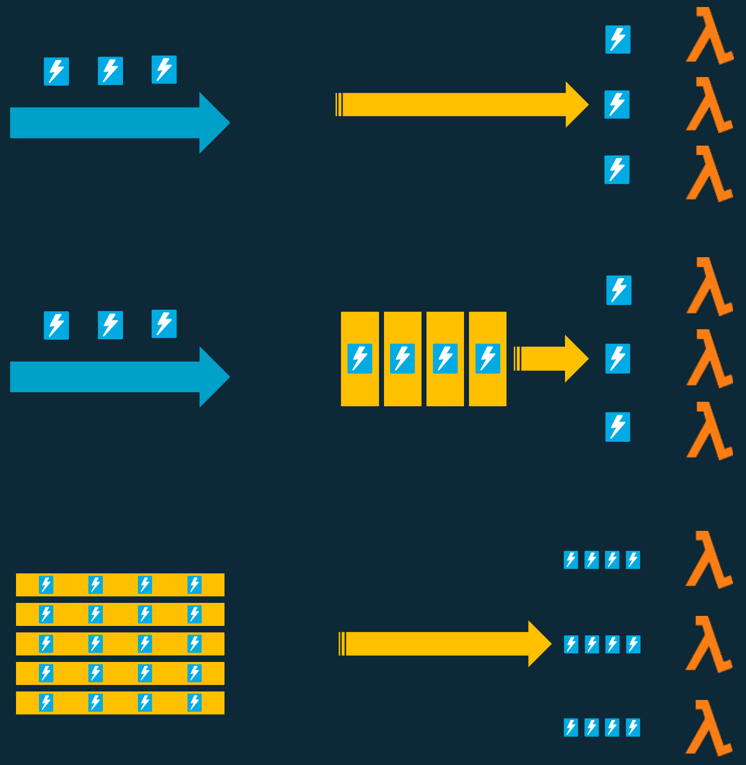
Persistence

Consumption models

Retries

Pricing

Concurrency across models



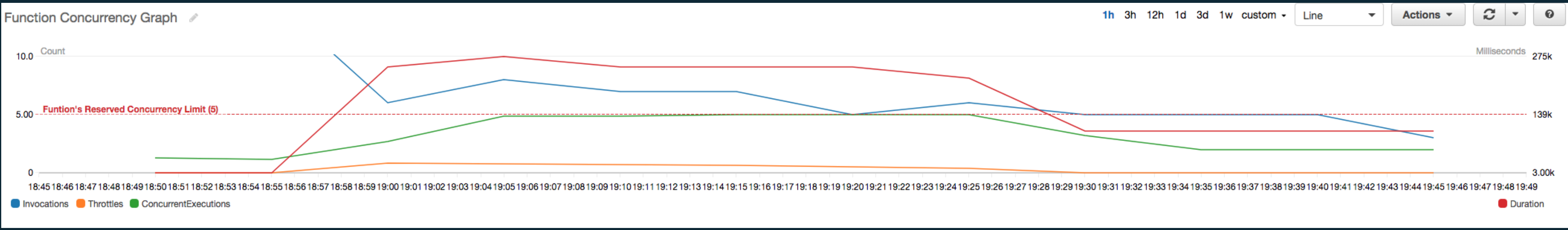
SNS/API
No event store

Queue based

Stream based

Lambda Per Function Concurrency controls

- Concurrency a shared pool by default
- Separate using per function concurrency settings
    - Acts as reservation
- Also acts as max concurrency per function
    - Especially critical for downstream resources like databases
- “Kill switch” – set per function concurrency to zero



Lambda Dead Letter Queues

“By default, a failed Lambda function invoked asynchronously is retried twice, and then the event is discarded.” – https://docs.aws.amazon.com/lambda/latest/dg/dlq.html

- Turn this on! (for async use-cases)
- Monitor it via an SQS Queue length metric/alarm
- If you use SNS, send the messages to something durable and/or a trusted endpoint for processing
  - Can send to Lambda functions in other regions
- If and when things go “**boom**” DLQ can save your invocation event information

"Action":"s3:*" makes puppies cry

Photo by Matthew Henry on Unsplash

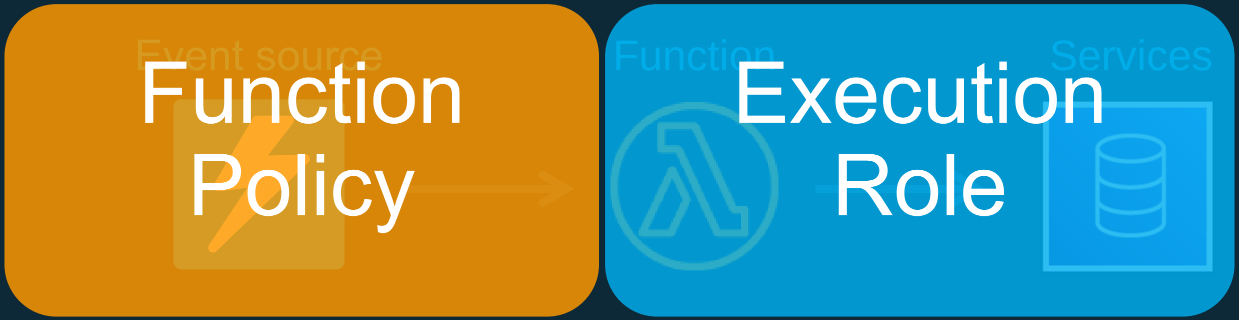
Lambda permissions model

Function policies:

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

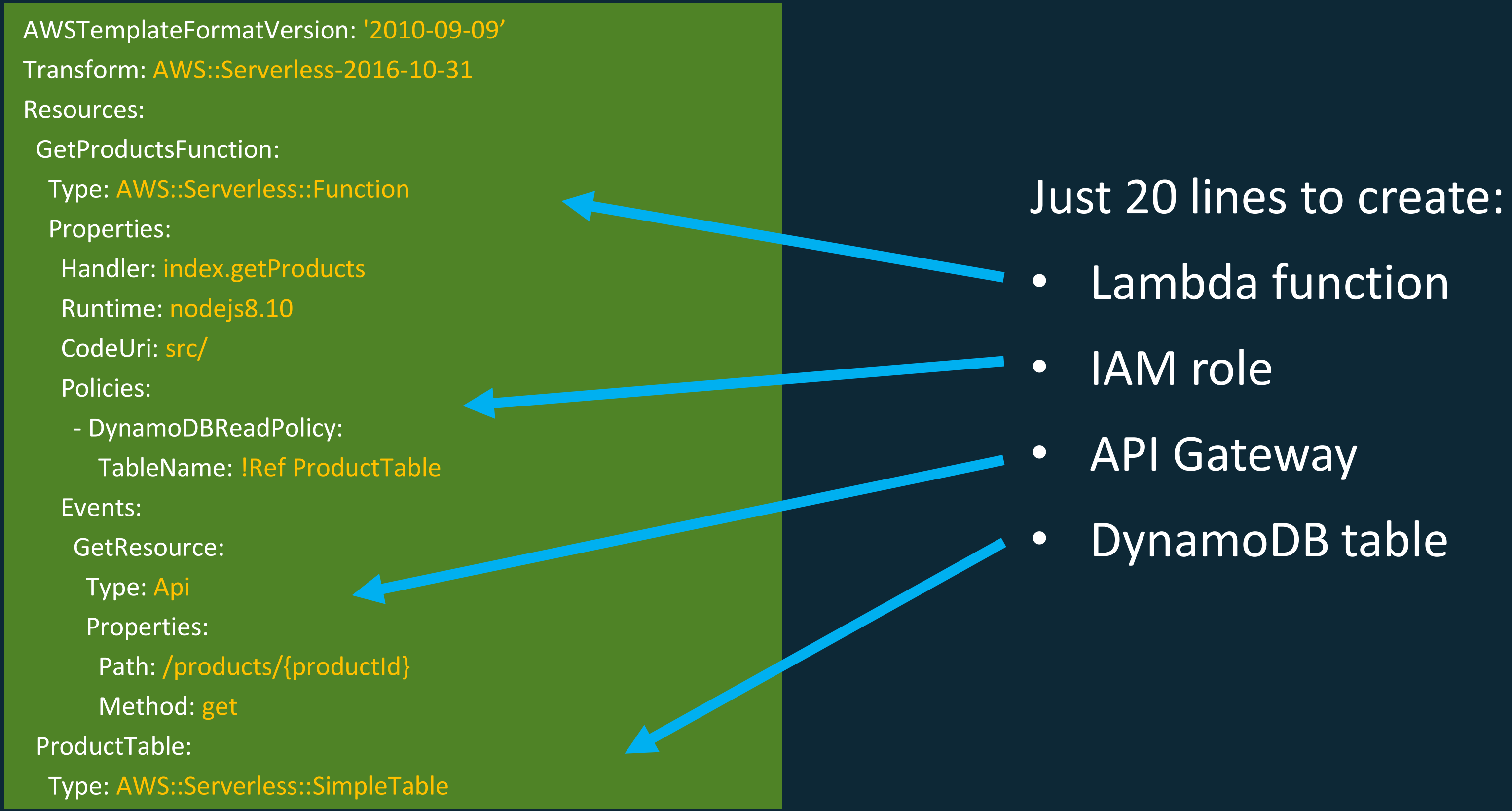
Execution role:

- “Lambda function A can read from DynamoDB table users”
- Define what AWS resources/API calls can this function access via IAM
- Used in streaming invocations



Meet AWS SAM!

AWS SAM Template



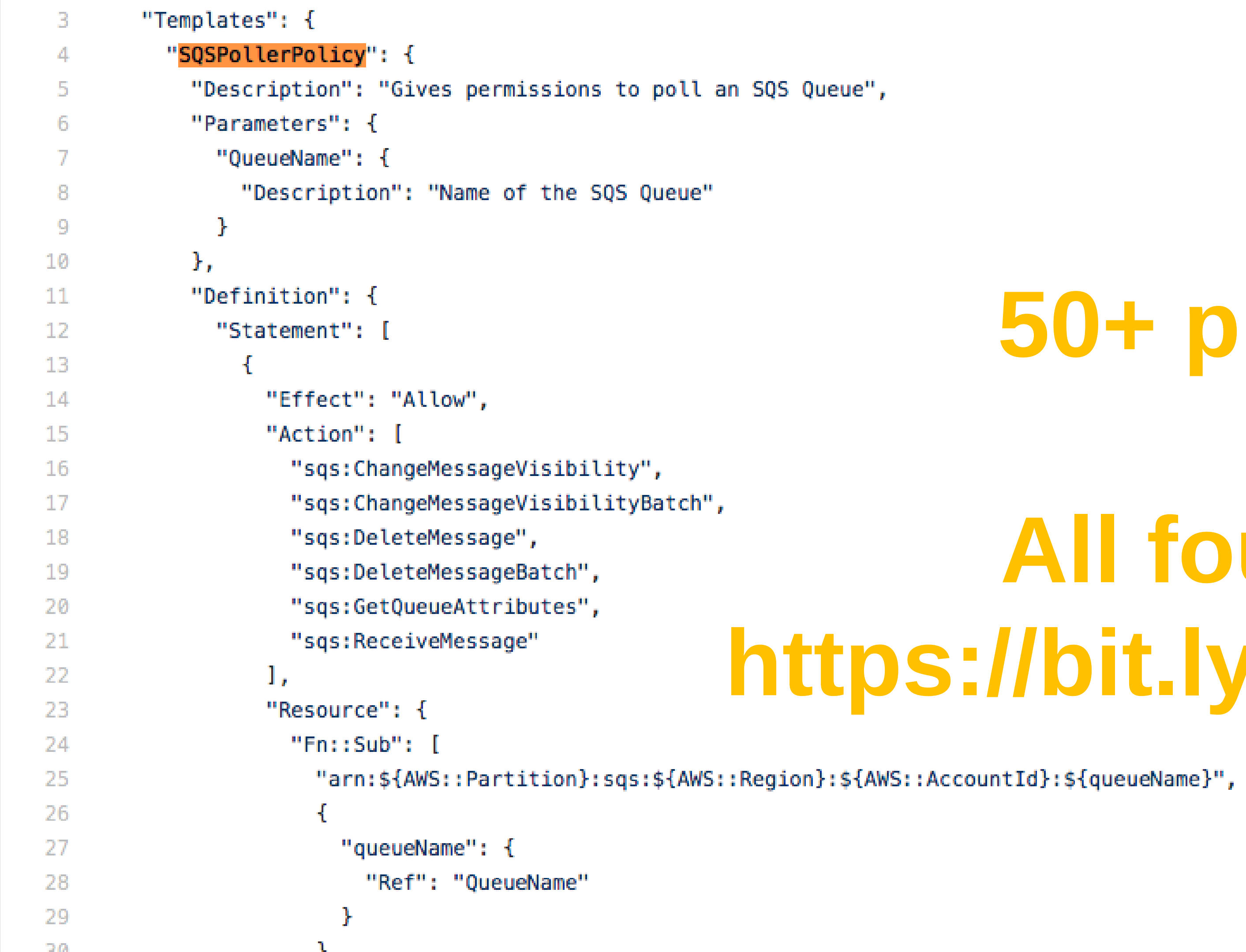
```
AWSTemplateFormatVersion: '2010-09-09’
Transform: AWS::Serverless-2016-10-31
Resources:
 GetProductsFunction:
  Type: AWS::Serverless::Function
  Properties:
   Handler: index.getProducts
   Runtime: nodejs8.10
   CodeUri: src/
   Policies:
    - DynamoDBReadPolicy:
      TableName: !Ref ProductTable
   Events:
    GetResource:
     Type: Api
     Properties:
      Path: /products/{productId}
      Method: get
 ProductTable:
  Type: AWS::Serverless::SimpleTable
```

Just 20 lines to create:

- Lambda function
- IAM role
- API Gateway
- DynamoDB table

AWS SAM Policy Templates

```
GetProductsFunction:
  Type: AWS::Serverless::Function
  Properties:
    ...
    Policies:
      - DynamoDBReadPolicy:
          TableName: !Ref ProductTable
    ...
ProductTable:
  Type: AWS::Serverless::SimpleTable
```

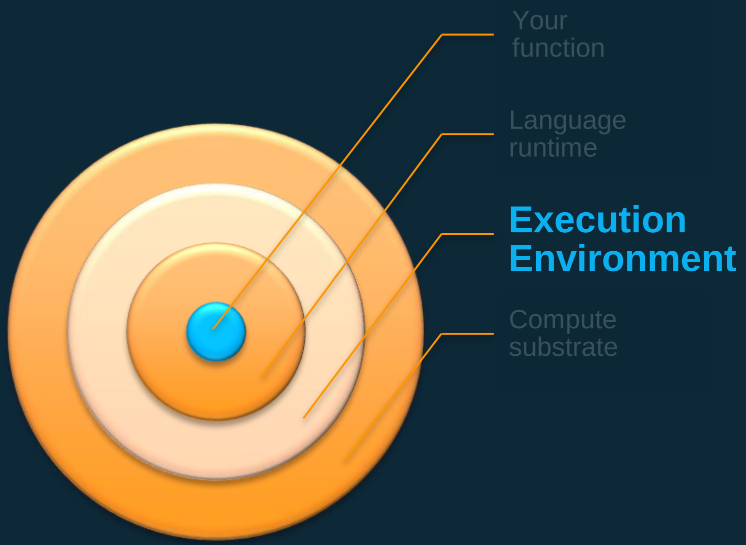


```
    "Templates": {
      "SQSPollerPolicy": {
        "Description": "Gives permissions to poll an SQS Queue",
        "Parameters": {
          "QueueName": {
            "Description": "Name of the SQS Queue"
          }
        },
        "Definition": {
          "Statement": [
            {
              "Effect": "Allow",
              "Action": [
                "sqs:ChangeMessageVisibility",
                "sqs:ChangeMessageVisibilityBatch",
                "sqs:DeleteMessage",
                "sqs:DeleteMessageBatch",
                "sqs:GetQueueAttributes",
                "sqs:ReceiveMessage"
              ],
              "Resource": {
                "Fn::Sub": [
                  "arn:${AWS::Partition}:sqs:${AWS::Region}:${AWS::AccountId}:${queueName}",
                  {
                    "queueName": {
                      "Ref": "QueueName"
                    }
                  }
```

50+ predefined policies

All found here: https://bit.ly/2xWycnj

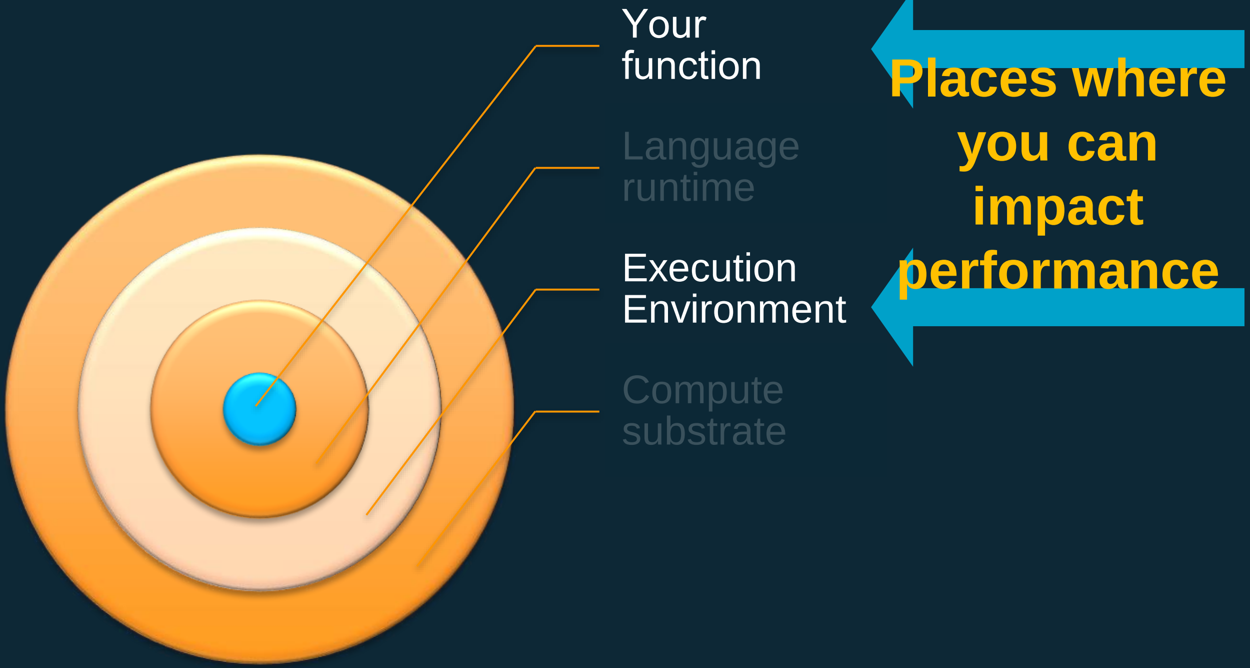
Anatomy of a Lambda function



Recap:

- More memory == More CPU and I/O (proportionally)
  - Can also be lower cost
- Use AWS X-Ray to profile your workload
- >1.8GB memory get’s you 2 cores, but you might not use/need it
- Think deeply about your execution model and invocation source needs
  - Not everything needs to be an API
- Thinking async will get you over some of the biggest scaling challenges
- Understand the various aspects to queues, topics, streams and event buses when using them
- Minimize the scope of IAM permissions
  - Leverage tooling like SAM

Anatomy of a Lambda function



FIN/ACK

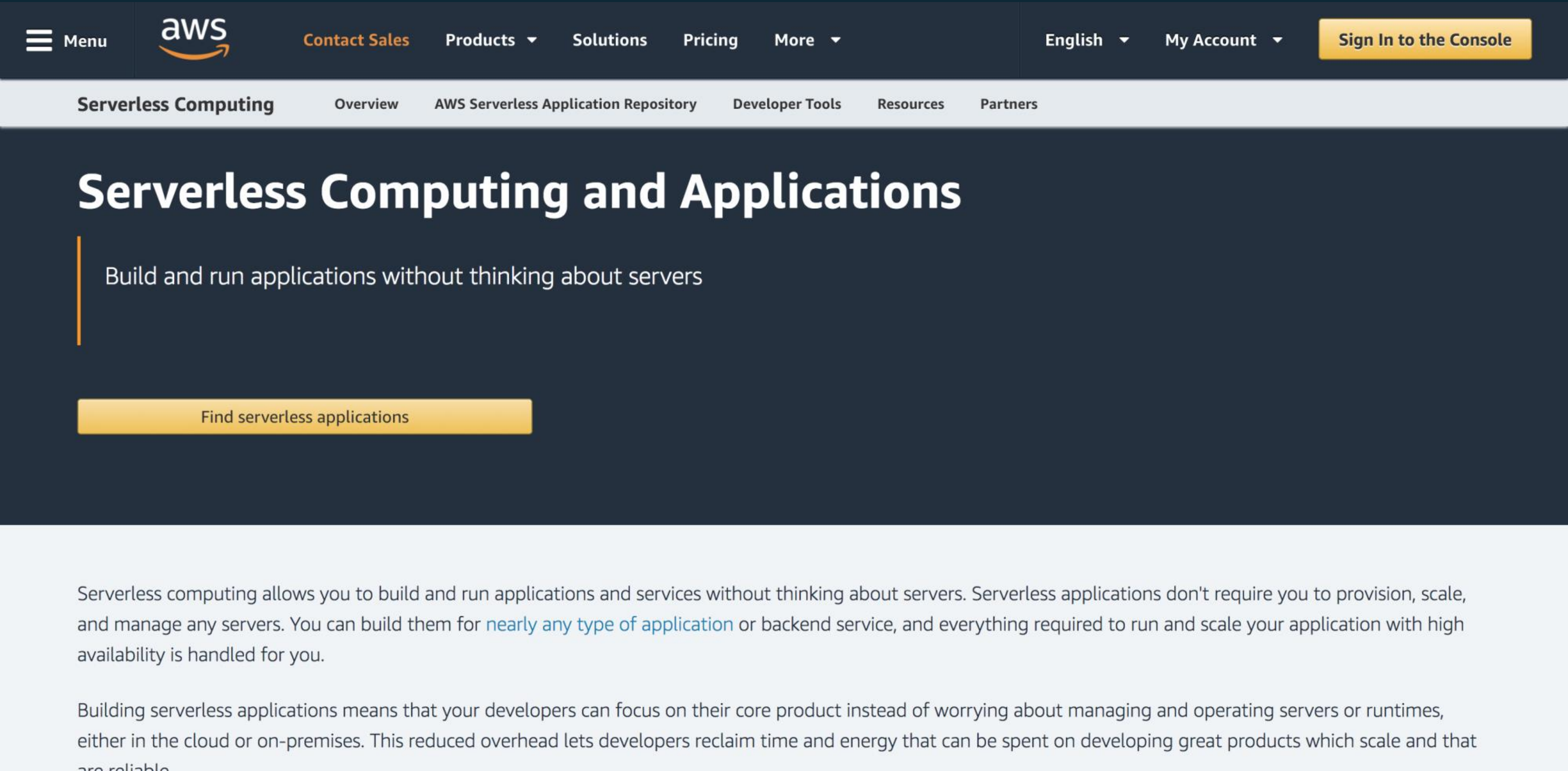
Your Function Recap:

- Minimize dependencies
- Use pre-handler logic sparingly but strategically
- Share secrets based on application scope:
  - Single function: Env-Vars
  - Multi Function/shared environment: Parameter Store
- Think about how re-use impacts variables, connections, and dependency usage
- Layers save on code duplication and help enable standardization across functions
- Concise logic.
- Push orchestration up to Step Functions or messaging services like EventBridge, SNS, SQS, or Kinesis

Execution Environment Recap:

- More memory == More CPU and I/O (proportionally)
  - Can also be lower cost
- Use AWS X-Ray to profile your workload
- >1.8GB memory get's you 2 cores, but you might not use/need it
- Think deeply about your execution model and invocation source needs
  - Not everything needs to be an API
- Thinking async will get you over some of the biggest scaling challenges
- Understand the various aspects to queues, topics, streams and event buses when using them
- Minimize the scope of IAM permissions
  - Leverage tooling like SAM

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

Chris Munns

munns@amazon.com

@chrismunns

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

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