
slides: bit.ly/sls-devops
SERVERLESS
DevOps
with @DavidWells

# DevOps

1. **Code**
2. **Build**
3. **Test**
4. **Package**
5. **Release**
6. **Configure**
7. **Monitor**
8. **Security**
9. **Controlling Costs**
10. **...**



# Serverless


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

# SERVERLESS

Code that runs on-demand
inside the cloud of your choice

- **Pay per execution pricing**
- **Never pay for idle servers**
- **Auto scales for you**
- **Event driven workflows**
- **Leverage third party services**

# SERVERLESS

Code that runs on-demand
inside the cloud of your choice





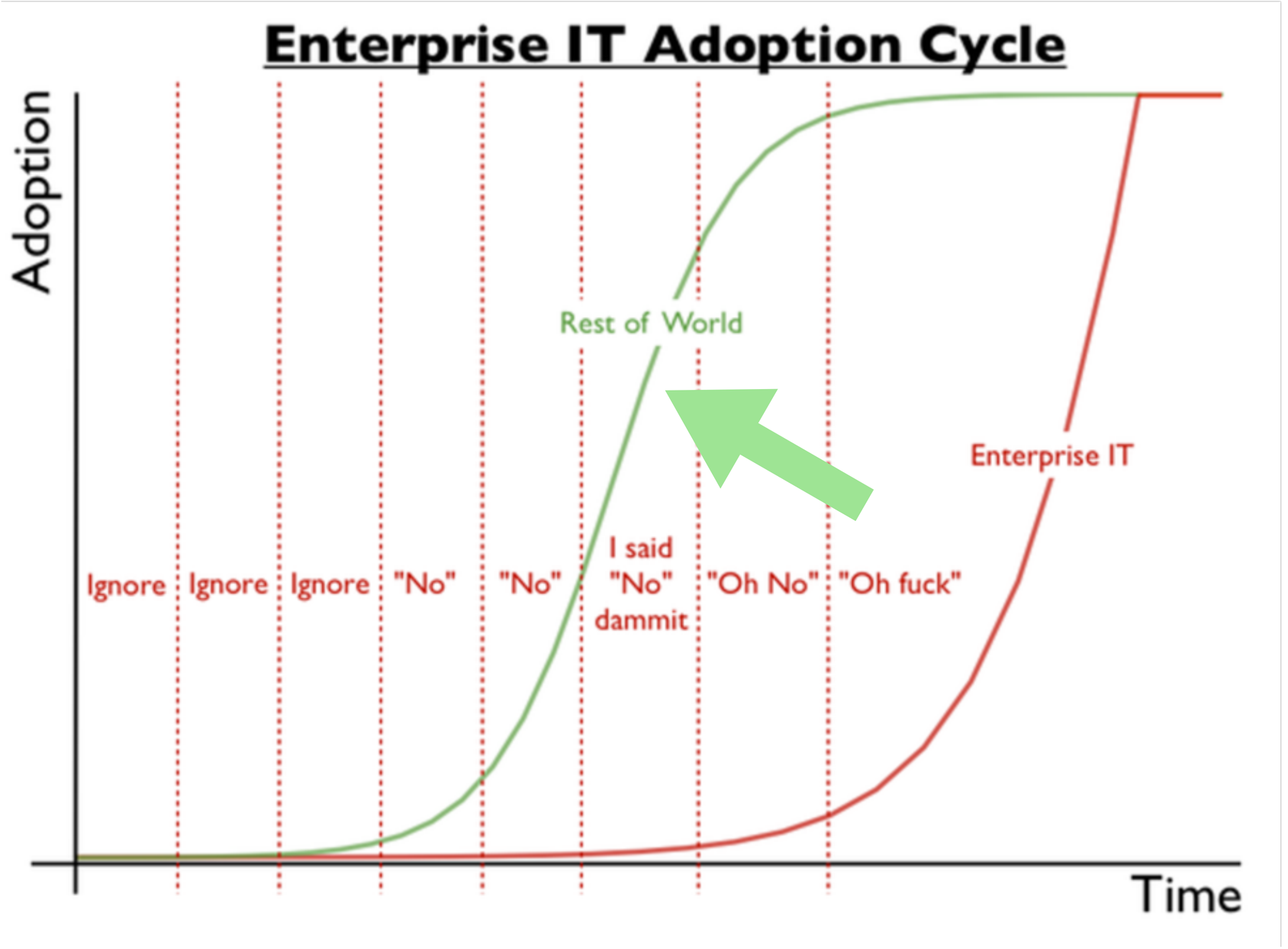








and auth0 webtasks, oracle functions, kubernetes, & spotInst....


Enterprise IT Adoption Cycle
Adoption
Rest of World
Enterprise IT
Ignore
Ignore
Ignore
"No"
"No"
I said "No" dammit
"Oh No"
"Oh fuck"
Time


use cases

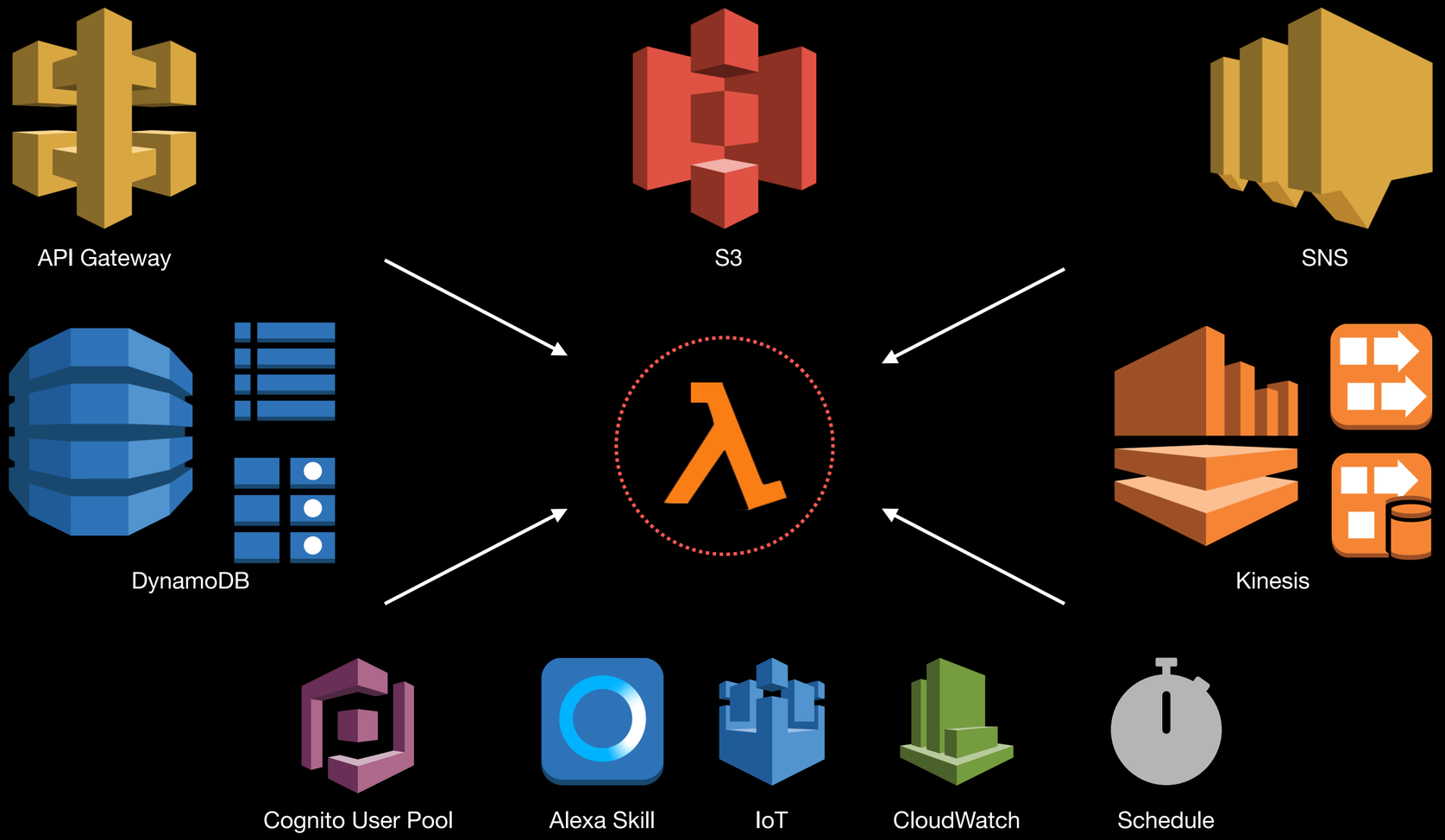
# Use Cases

- **Web + Mobile backend APIs**
- **Form processing**
- **Image processing**
- **Web scraping / testing**
- **Running binaries**
- **Serving HTML / dynamic binaries**
- **Webhook listeners**
- **Cron jobs**
- **DevOps/Infrastructure Automation**
- **Auditing resources for security**
- **Infrastructure events**
  - **S3** - new files, updated files etc
  - **DynamoDB -** new items in DB
- **Alexa skills**
- **IoT events**
- **Bots (Chat/Voice)**

**github.com/serverless/examples**

# event sources

that trigger your functions



# Serverless DevOps

1. **Code**
2. **Configure**
3. **Build**
4. **Test**
5. **Package**
6. **Release**
7. **Monitor**
8. **Security**
9. **Controlling Costs**
10. **...**

# 1. **Code**

Write your code as per usual.

```
module.exports.myFunction = (event, context, callback) => {

  // Do stuff

  const response = {
    statusCode: 200,
    body: JSON.stringify({
      data: {}
    }),
  }

  return callback(null, response)
}
```

Top level function must match lambda signature

# 2. **Configure**

Infrastructure as code

Terraform

CloudFormation




serverless

# serverless.yml config file

serverless

```
service: user-service

provider:
  name: aws
  runtime: nodejs4.3

functions:
  saveUser:
    handler: handler.saveUser
    events:
      - http:
          path: save-user
          method: post
          cors: true
  makeProfileImageThumbnail:
    handler: handler.makeThumbnail
    events:
      - s3:
          bucket: user-thumbnail-bucket
          event: s3:ObjectCreated:*
```

**Where** to run

**What** to run

**When** to run

# serverless.yml config file

serverless

```
service: user-service

provider:
  name: aws
  runtime: nodejs4.3

functions:
  saveUser:
    handler: handler.saveUser
    events:
      - http:
          path: save-user
          method: post
          cors: true
  makeProfileImageThumbnail:
    handler: handler.makeThumbnail
    events:
      - s3:
          bucket: user-thumbnail-bucket
          event: s3:ObjectCreated:*
```

**Where** to run

**What** to run

**When** to run

```yaml
# 'service' is the name of the service
service: service-name-xyz

# 'provider' specifies where to deploy the service. Aka deploy to 'aws'
provider:
  name: aws
  runtime: nodejs6.10
  region: ${opt:region, 'us-east-1'}
  stage: ${opt:stage, 'dev'}
  # 'environment' is where you define any process.env environment variables
  environment:
    GITHUB_WEBHOOK_SECRET: ${env:GITHUB_WEBHOOK_SECRET}

# 'functions' references all the functions and events in the service
functions:
  getOpenIssues:
    handler: handler.getOpenIssues
    events:
      - http:
          path: open
          method: get
          cors: true

# 'resources' defines any infrastructure needed for the service aka a database
resources:
  Resources:
    MyDynamoDBCloudformation:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: my-table-name
        AttributeDefinitions:
          - AttributeName: number
```

serverless

**BYOC**loudformation

```yaml
# 'resources' defines any infrastructure needed
resources:
  Resources:
    MyDynamoDBCloudformation:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: my-table-name
        AttributeDefinitions:
          - AttributeName: number
            AttributeType: N
        KeySchema:
          - AttributeName: number
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
        TimeToLiveSpecification:
          AttributeName: ttl
          Enabled: true
```

serverless.yml is transpiled into cloudformation


```
"ServerlessDeploymentBucket": {
  "Type": "AWS::S3::Bucket"
},
"GithubWebhookListenerLogGroup": {
  "Type": "AWS::Logs::LogGroup",
  "Properties": {
    "LogGroupName": "/aws/lambda/serverless-status-board-prod-githubWebhookListener"
  }
},
"GetOpenIssuesLogGroup": {
  "Type": "AWS::Logs::LogGroup",
  "Properties": {
    "LogGroupName": "/aws/lambda/serverless-status-board-prod-getOpenIssues"
  }
},
"GetClosedIssuesLogGroup": {
  "Type": "AWS::Logs::LogGroup",
  "Properties": {
    "LogGroupName": "/aws/lambda/serverless-status-board-prod-getClosedIssues"
  }
},
"SetupLogGroup": {
  "Type": "AWS::Logs::LogGroup",
  "Properties": {
    "LogGroupName": "/aws/lambda/serverless-status-board-prod-setup"
  }
},
"TeardownLogGroup": {
  "Type": "AWS::Logs::LogGroup",
  "Properties": {
    "LogGroupName": "/aws/lambda/serverless-status-board-prod-teardown"
  }
},
"IamRoleLambdaExecution": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": [
              "lambda.amazonaws.com"
            ]
          },
          "Action": [
            "sts:AssumeRole"
          ]
        }
      ]
    },
    "Policies": [
      {
        "PolicyName": {
          "Fn::Join": [
            "-",
            [
              "prod",
              "serverless-status-board",
              "lambda"
            ]
          ]
        },
        "PolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
            {
              "Effect": "Allow",
              "Action": [
                "logs:CreateLogStream"
              ],
              "Resource": [
                {
                  "Fn::Sub": "arn:aws:logs:${AWS::Region}:${AWS::AccountId}:log-group:/aws/lambda/serverless-status-board-prod-githubWebhookListener:*"
                },
                {
                  "Fn::Sub": "arn:aws:logs:${AWS::Region}:${AWS::AccountId}:log-group:/aws/lambda/serverless-status-board-prod-getOpenIssues:*"
                },
                {
```

# 3. **Build**

Some things to note:

- Dependancies bundled with deploy artifact
- **50mb** limit on compressed .zip/.jar file size
- **250mb** limit on uncompressed .zip/.jar size

Tips:

- Keep lambdas as small as possible. This helps mitigate cold starts as well.

# 4. **Test**

No changes here, use your favorite testing tools

**Tips:**

- Use multiple stages in CI/CD flow to run unit tests AND integration tests
- Mocking is hard. Integration tests are your friend.
- If you really want local experience, Checkout localstack

# 5. **Package**

Functions must be zipped with all deps.

**Pro Tip:**

- For building, packaging, and releasing use a tool

Apex SAM

Claudia.js Sparta

Chalice Zappa



# 6. **Release**

Use your favorite CI/CD tool of choice.



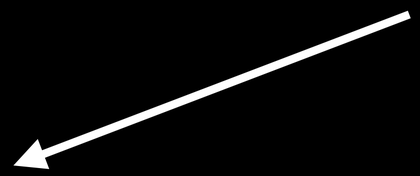
```
➜ user-service git:(master) ✗ sls deploy
Serverless: Packaging service...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Uploading service .zip file to S3 (2.31 MB)...
Serverless: Validating template...
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
.....................
Serverless: Stack update finished...
Service Information
service: growth-user-service
stage: prod
region: us-west-2
api keys:
  None
endpoints:
  POST - https://f5b6b6c9pa.execute-api.us-west-2.amazonaws.com/prod/users-by-email
functions:
  authenticate: growth-user-service-prod-authenticate
  getUserByEmail: growth-user-service-prod-getUserByEmail
Serverless: Removing old service versions...
```



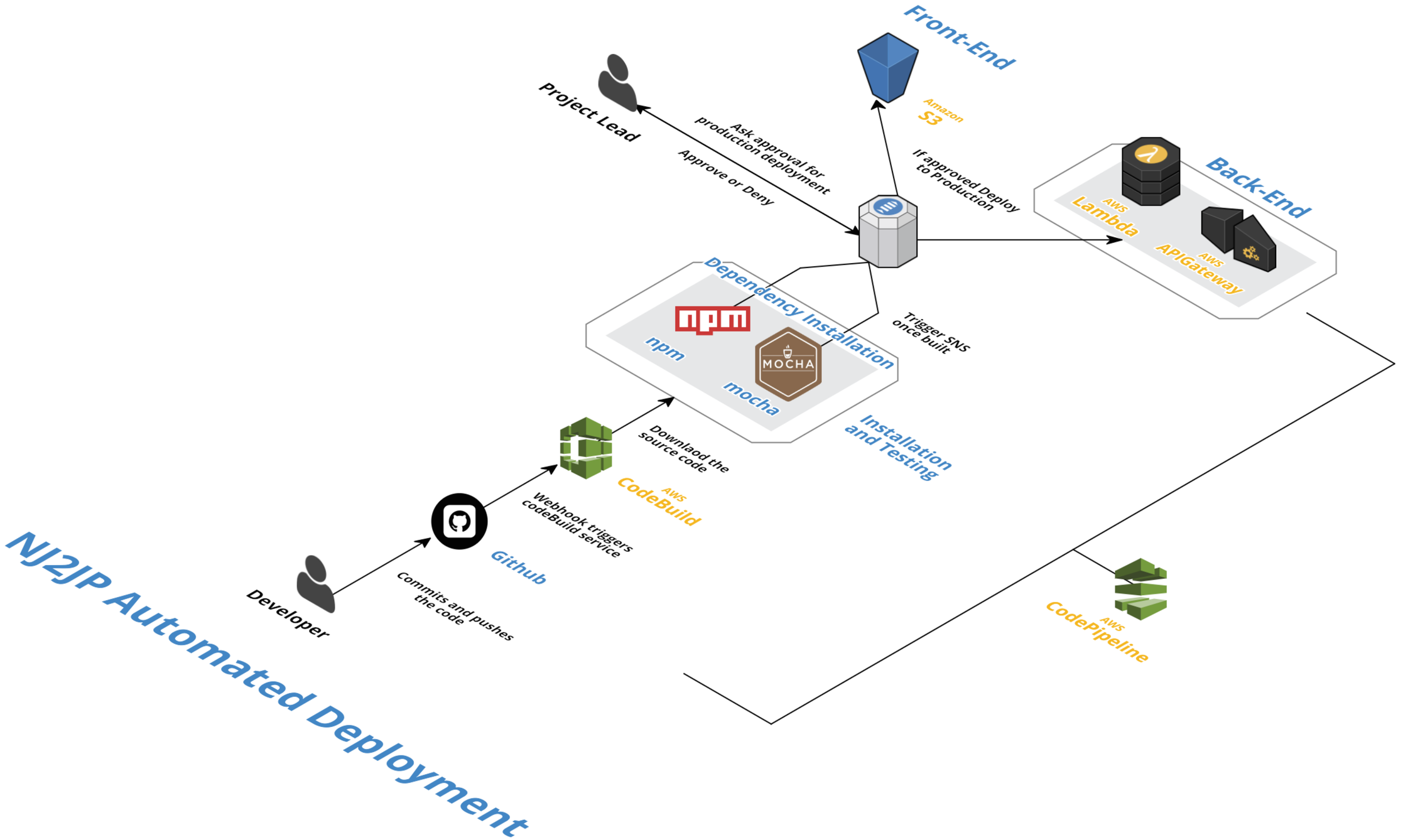
serverless deploy in cwd



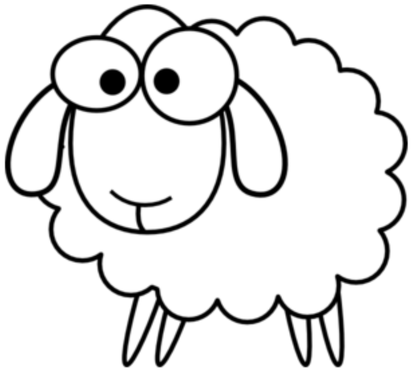
framework packages & deploys code

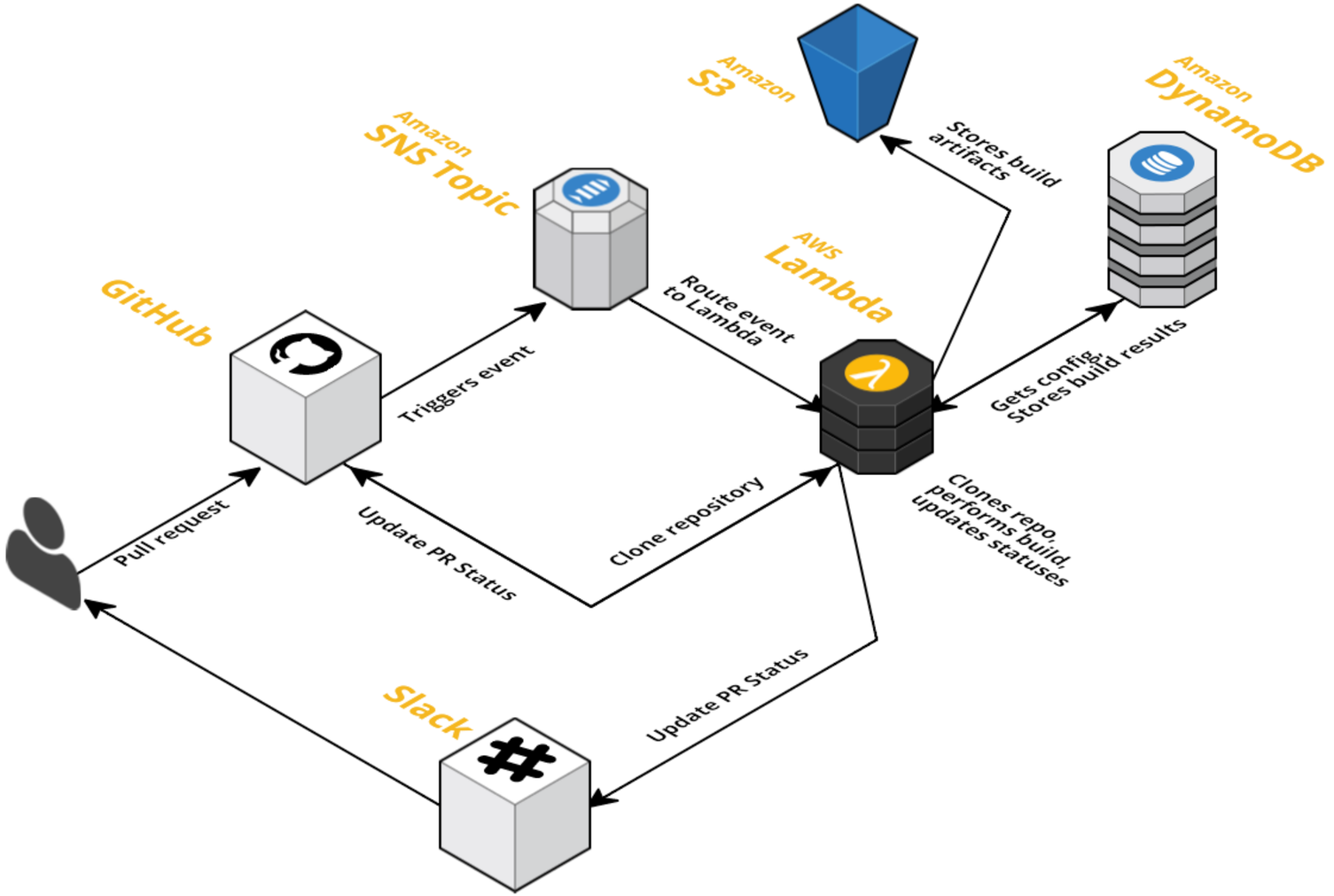


returns live service data


NJ2JP Automated Deployment
Front-End
Amazon S3
Back-End
AWS Lambda
AWS APIGateway
Project Lead
Ask approval for production deployment
Approve or Deny
If approved Deploy to Production
Dependency Installation
npm
mocha
MOCHA
Trigger SNS once built
Installation and Testing
Downlaod the source code
AWS CodeBuild
Webhook triggers codeBuild service
Github
Developer
Commits and pushes the code
AWS CodePipeline

# LambCI



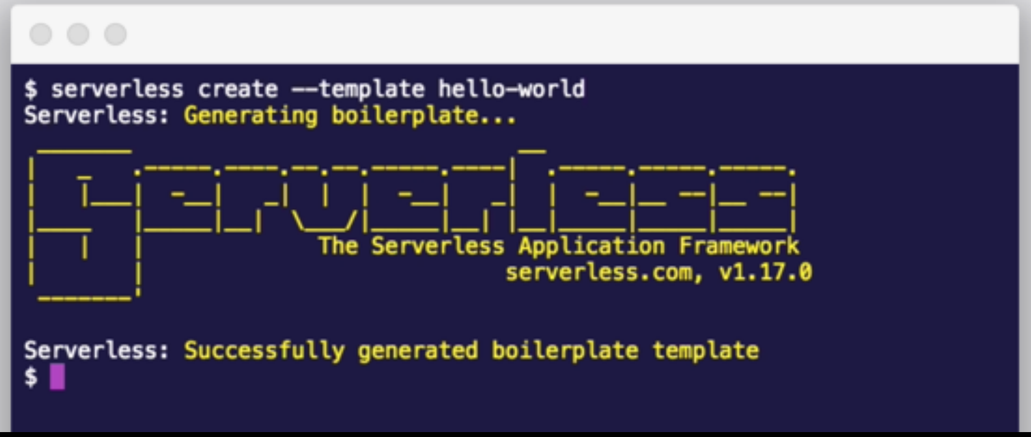



github.com/lambci/lambci

# SEED

A fully-configured CI/CD
pipeline for Serverless Framework

Create a Free Account




https://seed.run

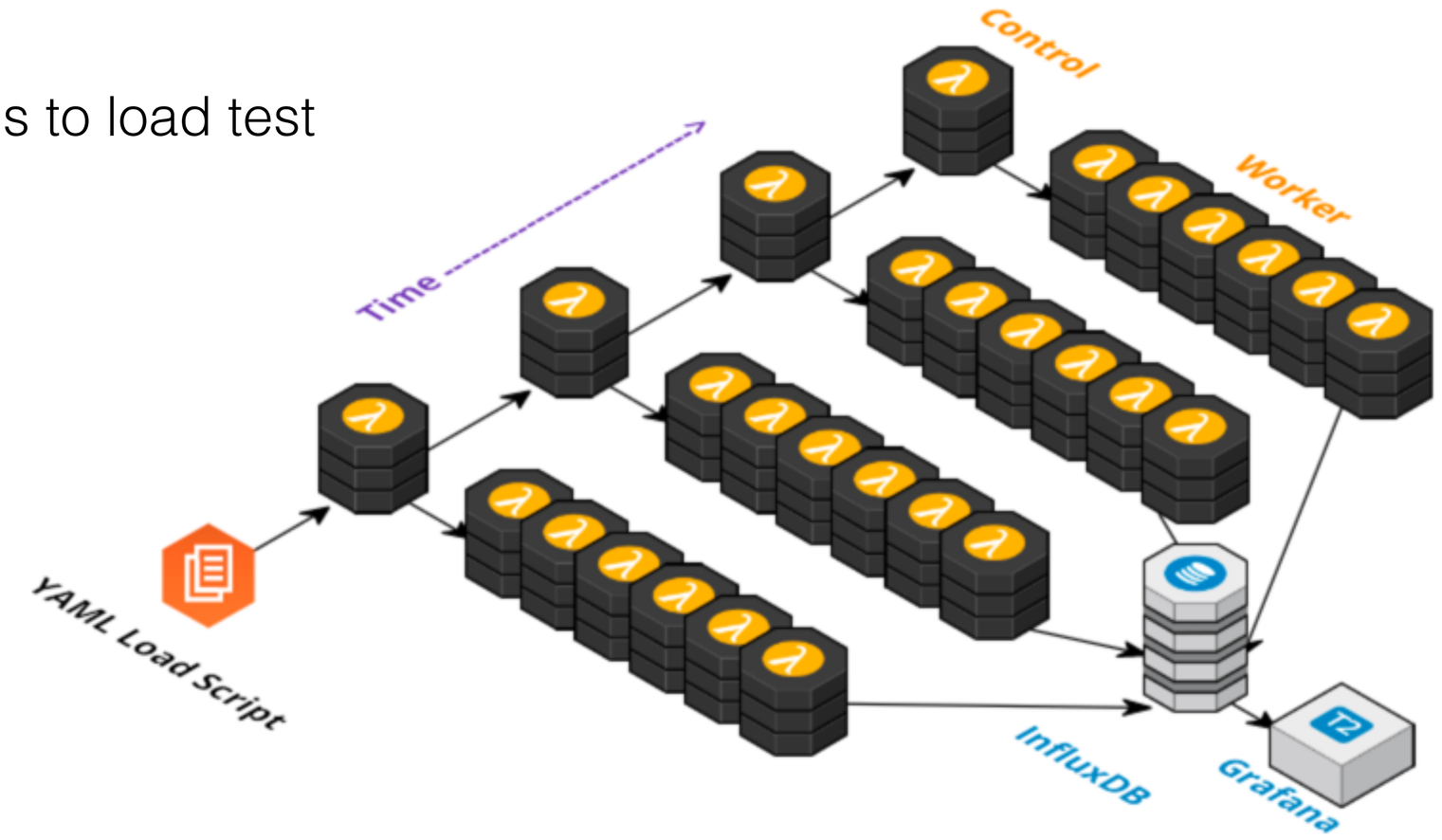
# Load Testing

On a single EC2 instance, you can easily get ~300 RPS (Requests Per Second) of load from artillery.io, in fact you may be able to coax >1,000 RPS with a larger instance & some tweaking...

but **what about when you need 50,000+ RPS to test your system**?

## Serverless Artillery.io

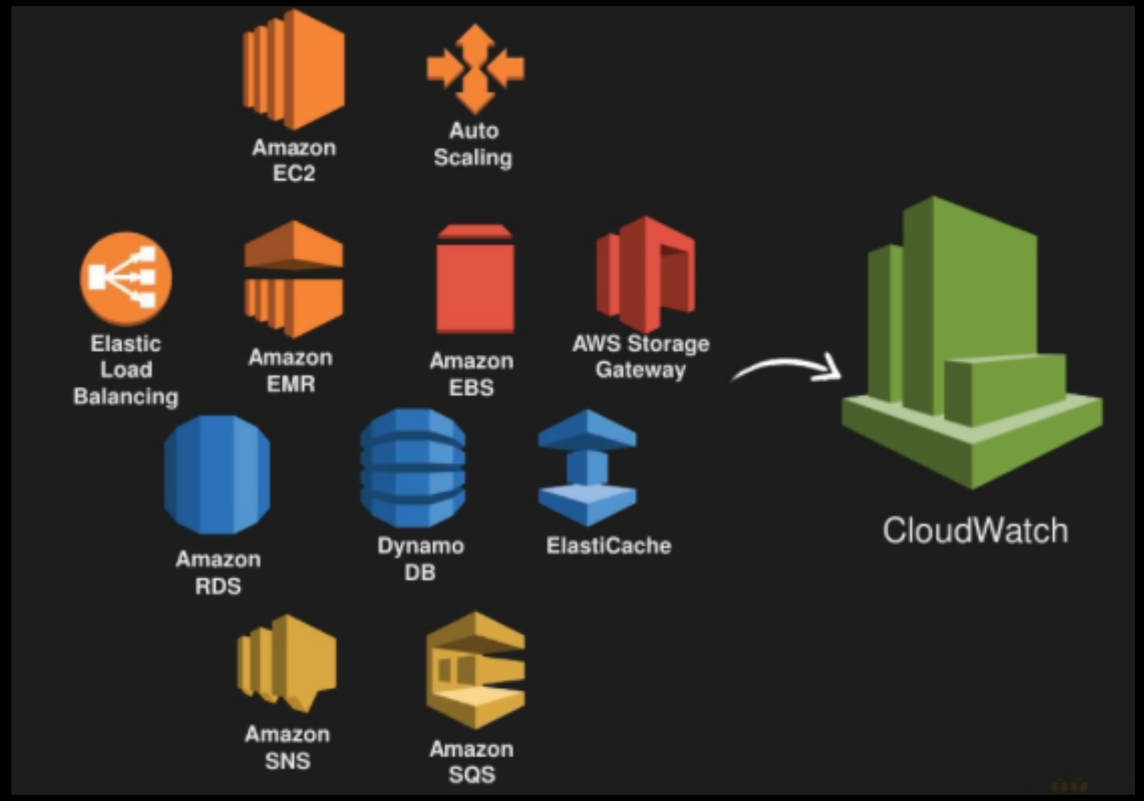
Spin up "n" number of lambda's to load test



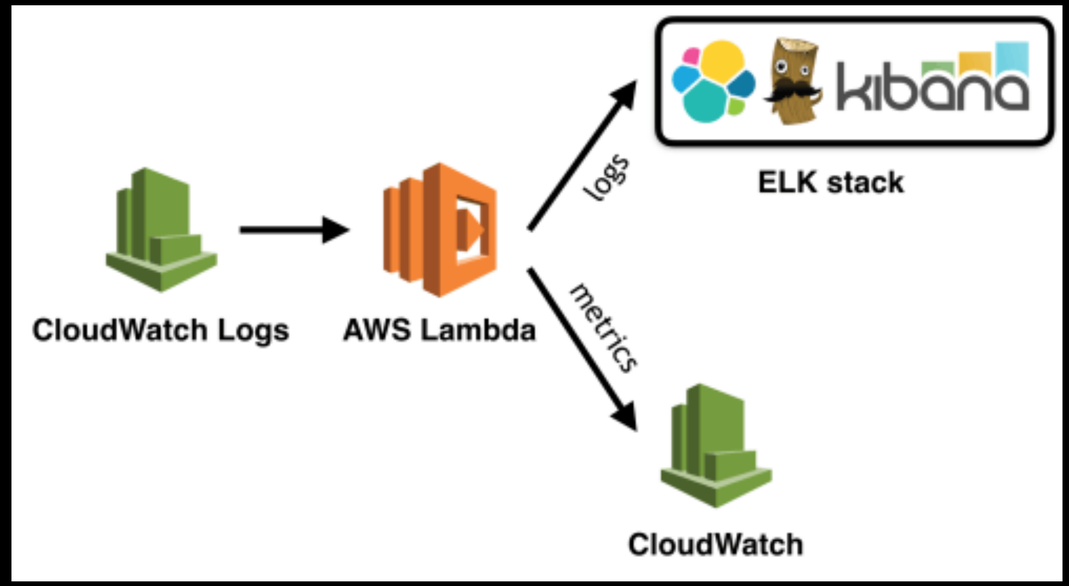

**github.com/Nordstrom/serverless-artillery**

**github.com/Nordstrom/serverless-artillery-workshop**

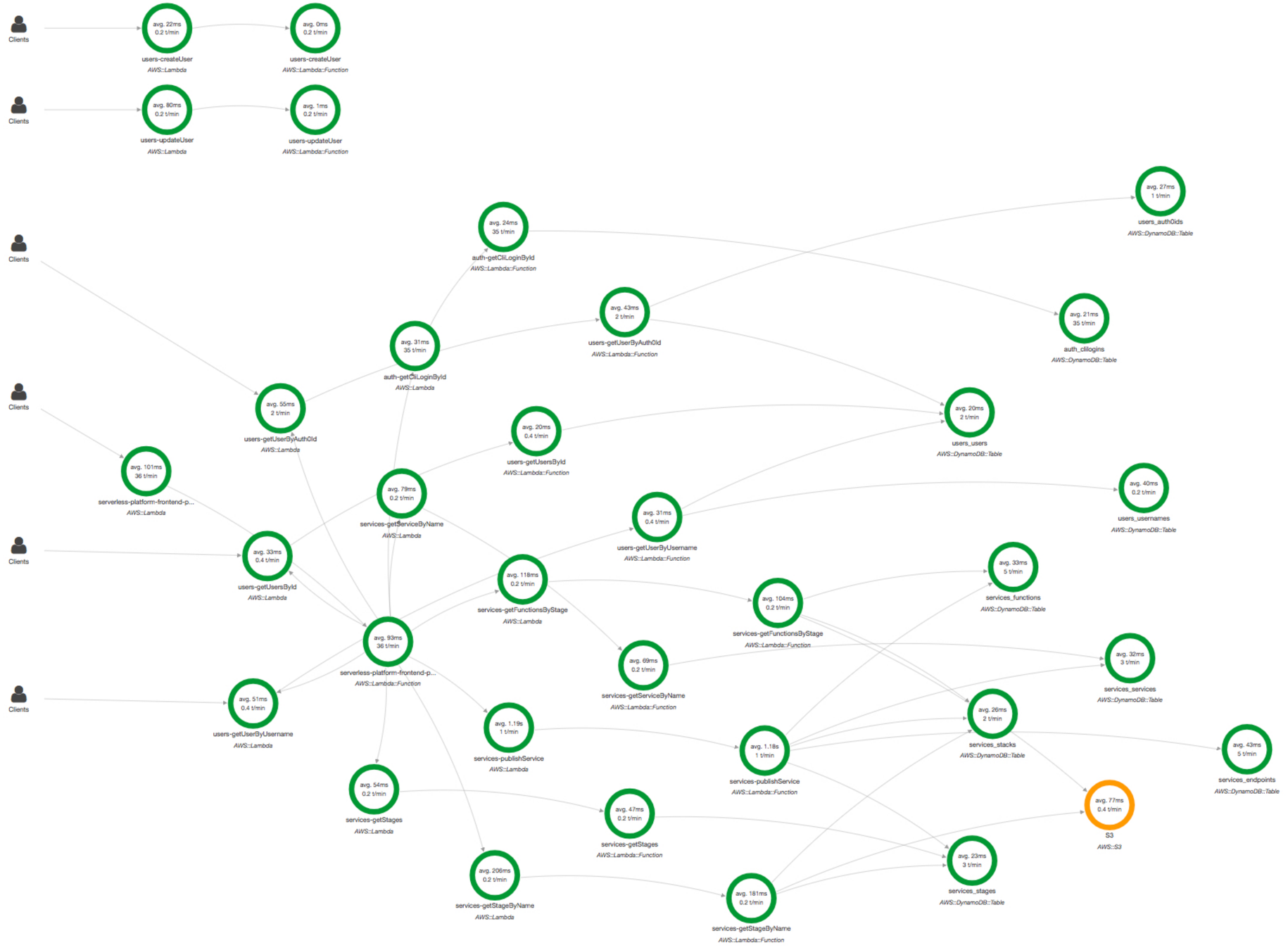
# 7. **Monitor**




IO|pipe

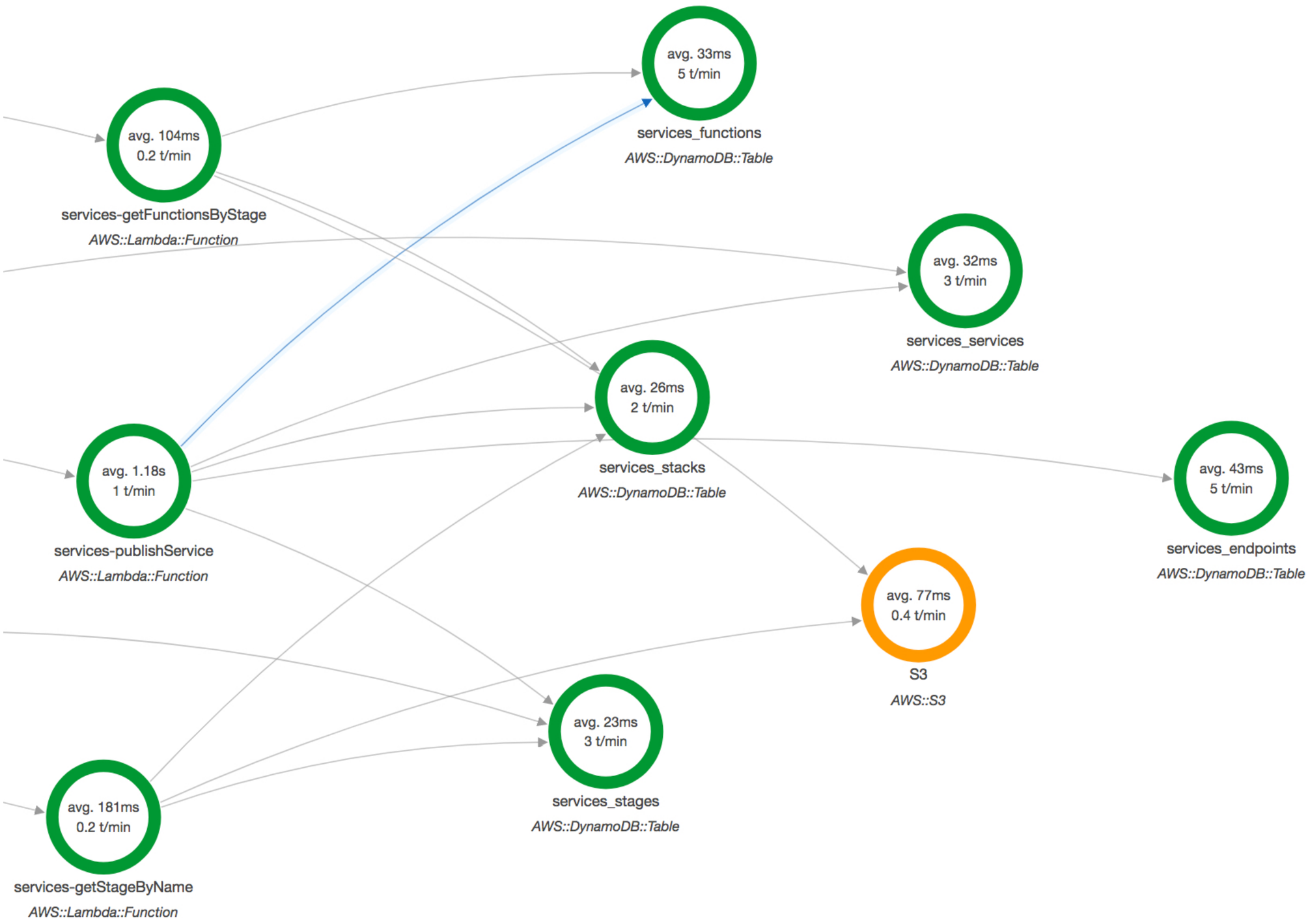


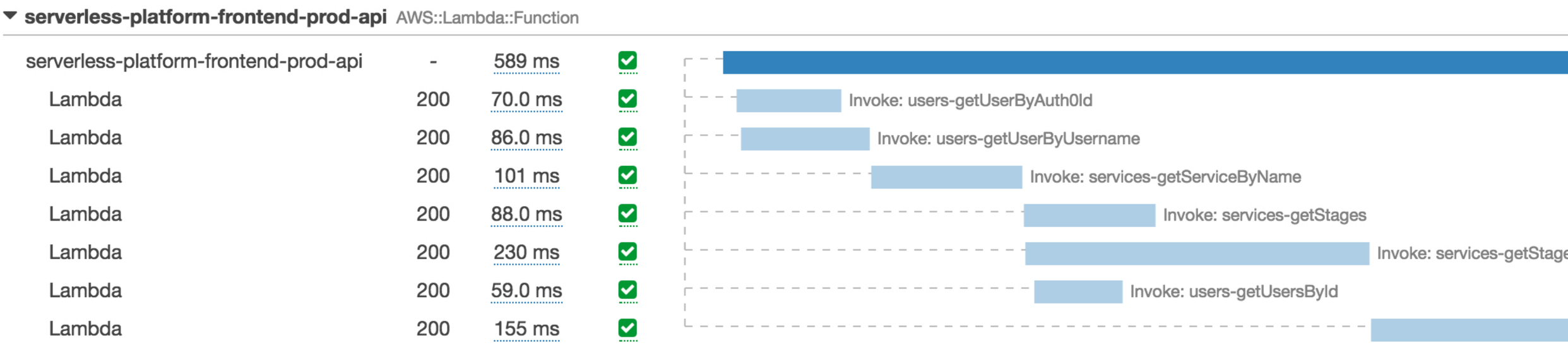
# Service map




x-ray

# x-ray


avg. 33ms
5 t/min
services_functions
AWS::DynamoDB::Table
avg. 104ms
0.2 t/min
services-getFunctionsByStage
AWS::Lambda::Function
avg. 32ms
3 t/min
services_services
AWS::DynamoDB::Table
avg. 26ms
2 t/min
services_stacks
AWS::DynamoDB::Table
avg. 1.18s
1 t/min
services-publishService
AWS::Lambda::Function
avg. 43ms
5 t/min
services_endpoints
AWS::DynamoDB::Table
avg. 77ms
0.4 t/min
S3
AWS::S3
avg. 23ms
3 t/min
services_stages
AWS::DynamoDB::Table
avg. 181ms
0.2 t/min
services-getStageByName
AWS::Lambda::Function

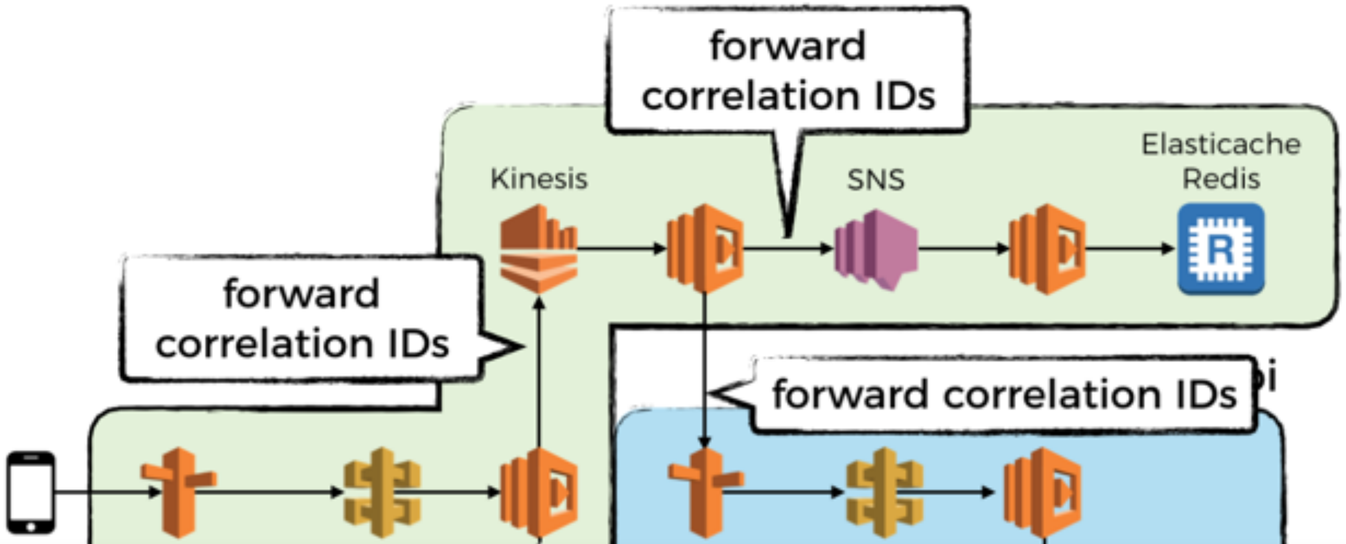

serverless-platform-frontend-prod-api AWS::Lambda::Function
serverless-platform-frontend-prod-api - 589 ms
Lambda 200 70.0 ms
Lambda 200 86.0 ms
Lambda 200 101 ms
Lambda 200 88.0 ms
Lambda 200 230 ms
Lambda 200 59.0 ms
Lambda 200 155 ms
Invoke: users-getUserByAuth0Id
Invoke: users-getUserByUsername
Invoke: services-getServiceByName
Invoke: services-getStages
Invoke: users-getUsersById



Yan Cui Follow
Senior Developer @ Space Ape Games. Author of AWS Lambda in Motion
Sep 1, 2017 · 10 min read

# Capture and forward correlation IDs through different Lambda event sources

Serverless architectures are microservices by default, you need correlation IDs to help debug issues that spans across multiple functions, and possibly different event source types—asynchronous, synchronous and streams.




bit.ly/sls-tracing

# 8. **Security**



- follow the principle of least privilege
- use AWS IAM users & roles
- no passwords, use separate sets of credentials
- rotate your AWS security credentials
- encrypt data at rest: AWS CloudHSM, AWS KMS, AWS SSM
- have “security drills”, measure speed of detection & execution
- audit logs and analyze them

# Serverless IAM Roles Per Function Plugin

bit.ly/iam-per-func

## Installation

```
npm install --save-dev serverless-iam-roles-per-function
```

Add the plugin to serverless.yml:

```
plugins:
  - serverless-iam-roles-per-function
```

## Usage

Define `iamRoleStatements` definitions at the function level:

```
functions:
  func1:
    handler: handler.get
    iamRoleStatementsName: my-custom-role-name #optional custom role name setting instead of the default generated or
    iamRoleStatements:
      - Effect: "Allow"
        Action:
          - dynamodb:GetItem
        Resource: "arn:aws:dynamodb:${self:provider.region}:*:table/mytable"
    ...
  func2:
    handler: handler.put
    iamRoleStatements:
```

The **cfn-nag** tool looks for patterns in CloudFormation templates that may indicate insecure infrastructure.

It will look for:

- IAM rules that are too permissive (wildcards)
- Security group rules that are too permissive (wildcards)
- Access logs that aren't enabled
- Encryption that isn't enabled




**github.com/stelligent/cfn_nag**

# Handling secrets

- AWS SSM (Systems Manager Parameter Store)
- AWS Key Management Service (KMS)

# Using SSM in serverless.yml

```yaml
# 'service' is the name of the service
service: service-name-xyz

# 'provider' specifies where to deploy the service. Aka deploy to 'aws'
provider:
  name: aws
  runtime: nodejs6.10
  region: ${opt:region, 'us-east-1'}
  stage: ${opt:stage, 'dev'}

# 'functions' references all the functions and events in the service
functions:
  getOpenIssues:
    handler: handler.save
    events:
      - http: POST save-user
    environment:
      SUPER_SECRET_STUFF: ${ssm:/path/to/service/myParam}
```

# 9. **Controlling Costs**

- follow the principle of least privilege
- use AWS IAM users & roles
- no passwords, use separate sets of credentials
- rotate your AWS security credentials
- encrypt data at rest: AWS CloudHSM, AWS KMS, AWS SSM
- have “security drills”, measure speed of detection & execution
- audit logs and analyze them

# CapitalOne Cloud Custodian




Ensuring real time compliance for:

- **security policies** - like encryption & access requirements
- **tag policies**
- **cost management** - via garbage collection of unused resources & off-hours resource management.

**github.com/capitalone/cloud-custodian**

# …Other Use Cases

# Cloudwatch

- Check if too many resources are being provisioned too quickly
- Using cloudwatch + lambda to automate security requirements on EC2 instances, bucket policies etc.
- Using cloudwatch + lambda to automate fine tuned autoscaling in DynamoDB tables
- Alerting for console logins or for SSM parameter changes
- check the tags newly-created resources (DDB tables) to make sure they comply with your company's policies.

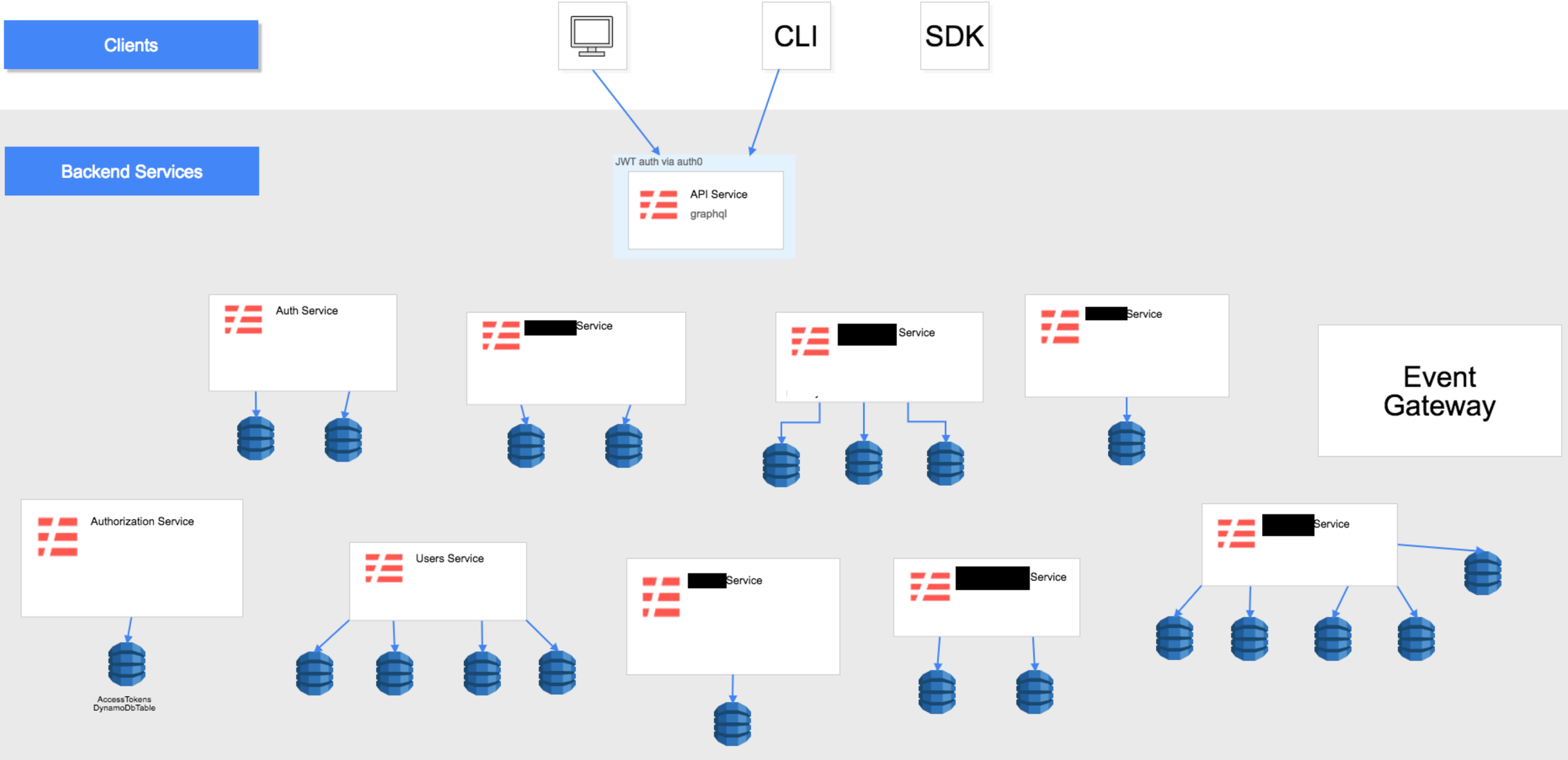
# Cron

- Using lambda cron to port over maintenance tasks
- Use scheduled functions to perform penetration tests!
- Scan for under-utilized services and spin them down


USE YOUR IMAGINATION!

# Serverless Inc CI/CD Flow

# serverless Platform Architecture



# How do we do it?

- New branches trigger travis builds
- First unit tests, linting etc run
- Then every service is `sls deployed` to a unique stage id `stage-xyz123`
- After services and resources are live integration tests run against them
- If everything is good, we can `sls deploy --stage prod` through travis

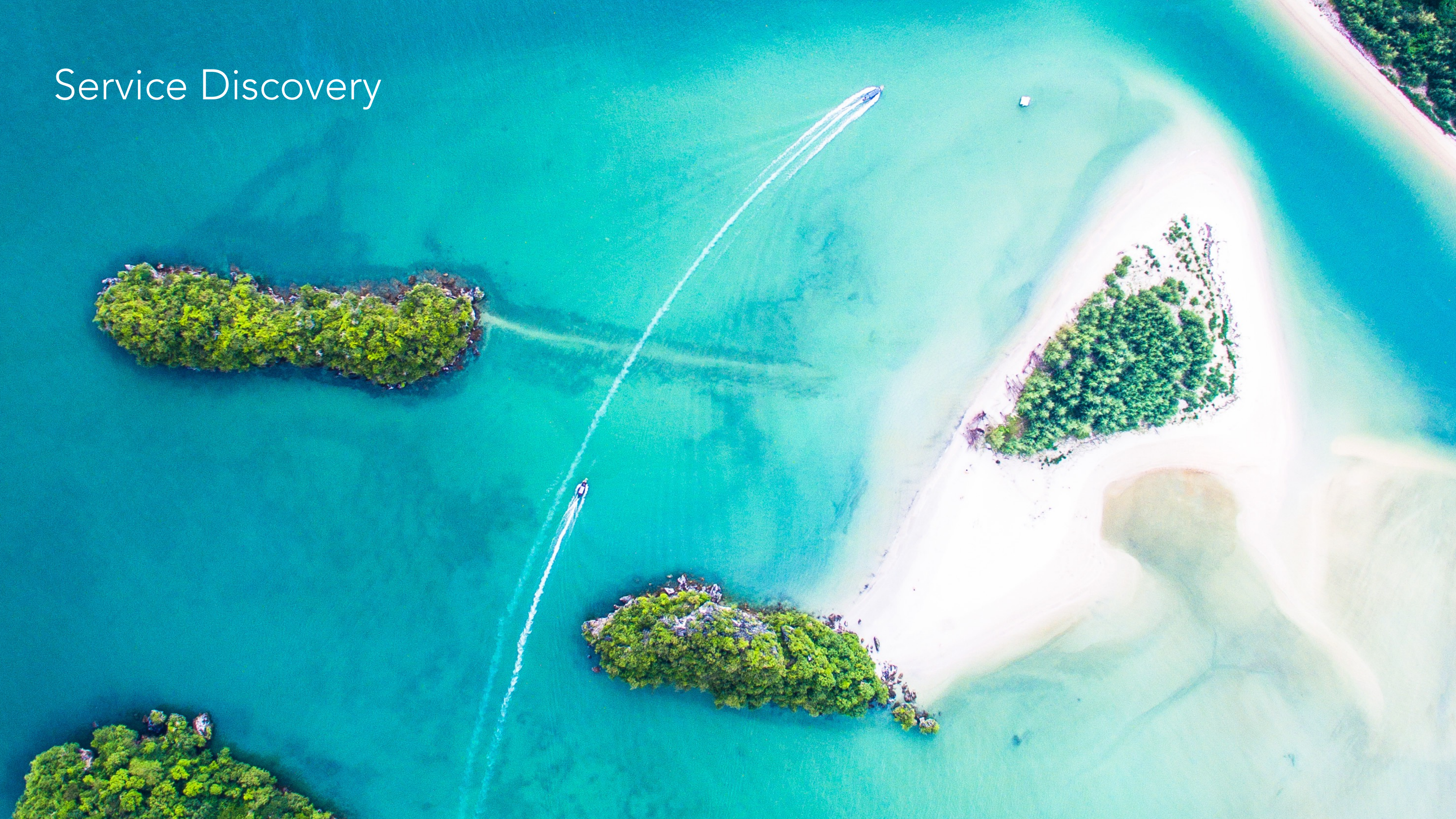
Tip: For this to work, hardcoded resource names are a no-no



# Challenges



# Service Discovery




Secret Management


Logging

# Tracing




Thanks for listening!
**Questions?**
Tweet **@DavidWells**
or contact me at
**davidwells.io**
slides: **bit.ly/sls-devops**