When designing microservices there are a number of things to think about. Just for starters, the bounds of their functionality, how they communicate with their dependencies, and how they provide an interface for their own consumers. Serverless technologies such as AWS Lambda change paradigms around code structure, usage of libraries, and how you deploy and manage your applications. In this session, we show you how by combining microservices and serverless technologies, you can achieve the ultimate flexibility and agility that microservices aim for, while providing business value in how serverless greatly reduces operational overhead and cost.

In addition, National Geographic will share how it built its NG1 platform using a serverless, microservices architecture. The NG1 platform provides National Geographic consumers with content personalized to their preferences and behaviors in an intuitive, easy-to-use way on smartphones.

WHY MICROSERVICES?

Business Goals

Time-to-market
The size and risk of change reduces the rate of change increases, which enables greater agility and faster time-to-market

Ownership
Teams act within a small and well-understood bounded context and are empowered to work independently

Design Pillars

Resilience
Isolate failure and to improve the overall availability

Scalability
Scale horizontally and independently

Continuous Delivery
Changes are frequent and failures are cheap

HOW SERVERLESS RELATES TO MICROSERVICES?

- The serverless approach aligns with the microservices design principles and best practices
- Some of them come out of the box
- Others are much easier to achieve than with traditional approaches
- Serverless is not a silver bullet
- Let's review each principle through a serverless lens…

RESILIENCE

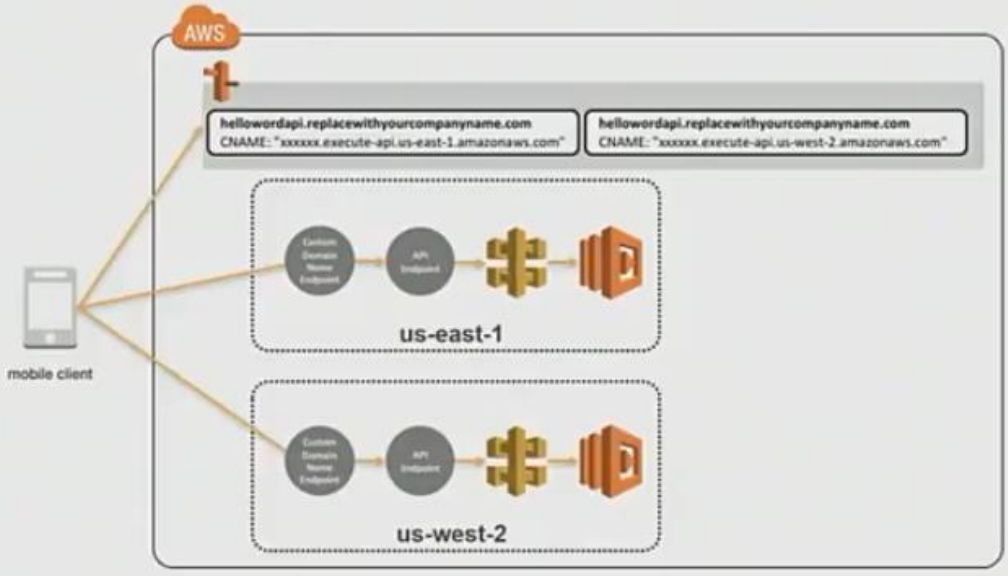
THINGS WILL FAIL, PLAN FOR IT

- Architect for high-availability
- Apply throttles
- Employ retries and set the right timeouts

HIGH-AVAILABILITY

- AWS Lambda is designed to use replication and redundancy to provide high availability
- Keep your microservices regionalized for additional resiliency

MULTI-REGION SERVERLESS APPLICATION



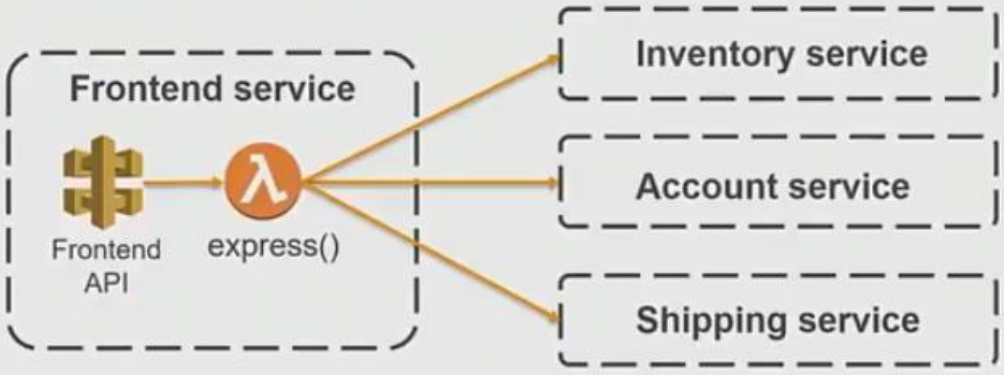
LATENCY-BASED HEALTH CHECK

If the health-check endpoint is not healthy, Amazon Route 53 selects the latency record with the next best latency

```
HealthcheckRegion1:
  Type: "AWS::Route53::HealthCheck"
  Properties:
    HealthCheckConfig:
      Port: "443"
      Type: "HTTPS_STR_MATCH"
      SearchString: "ok"
      ResourcePath: "/prod/healthcheck"
      FullyQualifiedDomainName: !Ref Region1HealthEndpoint
```


THROTTLING

Set throttles to prevent your downstream APIs from being overwhelmed by too many requests



HARD PROBLEM TO SOLVE

Backend

- Manage API keys
- Measure steady state rates
- Measure burst rates
- For each method

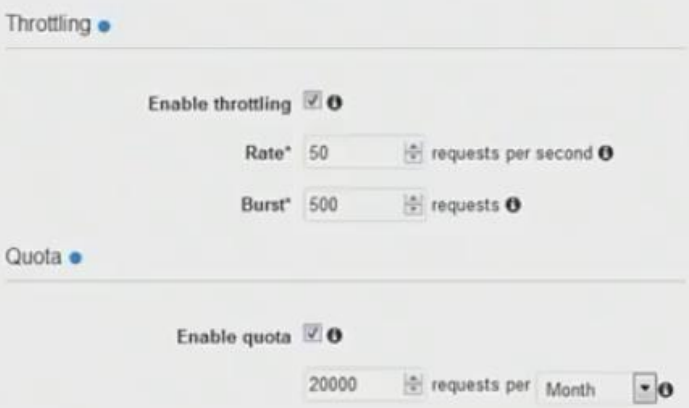
Client

- Identify
- Adjust

USAGE PLANS IN API GATEWAY

Create usage plans to control:

- Throttling—overall request rate (average requests per second) and a burst capacity
- Quota—number of requests that can be made per day, week, or month
- API/stages—the API and API stages that can be accessed



USE AWS X-RAY TO DEBUG THROTTLE ERRORS

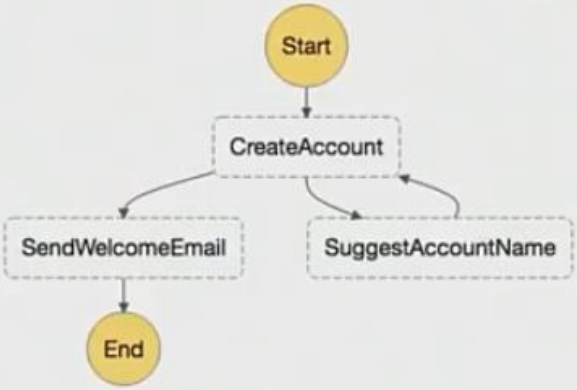
SET TIMEOUTS AND RETRIES

- In event-based architecture with many downstream calls, timeouts and retries are important to get right
- Have no timeouts at all, and a downstream API being down could hang your whole microservice
- Log when timeouts occur, look at what happens, and change them accordingly
- Limit number of retries and employ exponential backoff to avoid resource exhaustion and backlog
- Duplicates may happen; code should be idempotent

EASIER IN A SERVERLESS WORLD

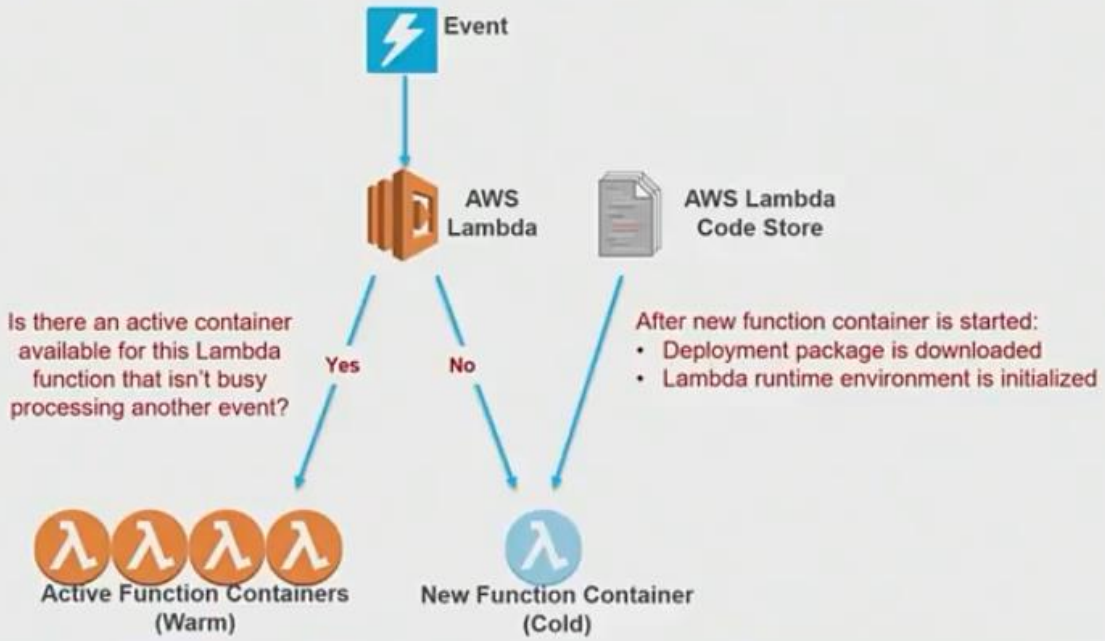
- Lambda lets you set a timeout for your function
- Use synchronous execution when response is needed
  - The invoking application receives a 429 error and is responsible for retries
- Use asynchronous execution when no response is needed
  - Messages are queued until executed
  - Automatic retries with exponential backoff
  - Dead Letter Queue (DLQ) for failed events
- Create and enable one DLQ per function—SQS or SNS
  - When unable to send error to configured DLQ, DLQ Errors metric will be updated

USE STEP FUNCTIONS FOR ADVANCED ORCHESTRATION

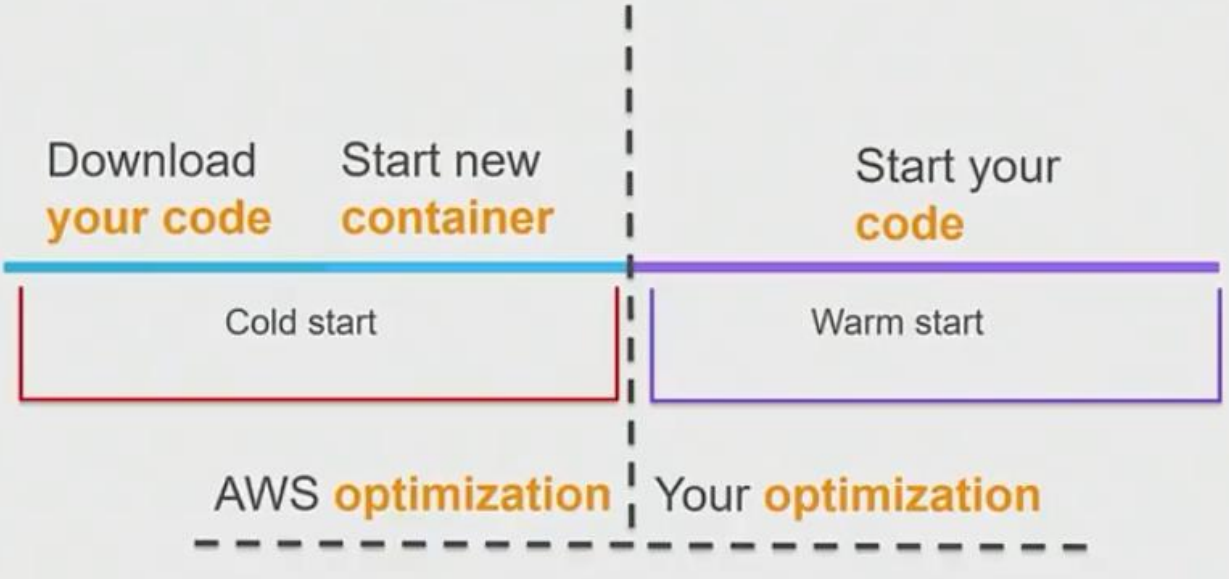


Scalability

HOW DO YOUR FUNCTIONS SCALE?



COLD START: UNDERSTAND THE FUNCTION LIFECYCLE



COLD START: SIMPLIFY AND MINIMIZE DEPENDENCIES

- **Minimize** your deployment package size
  - Leverage modularity of the AWS SDK for Java and .NET
- **Reduce the complexity** of your dependencies

WARM START: 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 every time

```
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()
...
```

Continuous Delivery

WHAT IS A GOOD PROCESS?

- ☐ Easy to interpret
- ☐ Frequent and small changes
- ☐ Changes have scoped impact
- ☐ Automated deployments

AWS SERVERLESS APPLICATION MODEL (SAM)

AWS 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 lets you have a template where you can define all the resources that are involved in building your microservice. This framework is supported by CloudFormation, so once you templatize your application you can also then version it and share it with others. You can also use CloudFormation as the deployment engine for your applications.

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 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 five Read and Write units

You can version this file as you change your microservice.

WHAT'S A GOOD PROCESS?

- [x] Easy to interpret
- [ ] Frequent and small changes
- [ ] Changes have scoped impact
- [ ] Automated deployments

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 the Lambda execution environment:

- Emulates timeout, memory limits, runtimes

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

WHAT'S A GOOD PROCESS?

- [x] Easy to interpret
- [x] Frequent and small changes
- [ ] Changes have scoped impact
- [ ] Automated deployments

TRAFFIC SHIFTING IN LAMBDA ALIASES

Apply a "weight" to Lambda function aliases to shift traffic between two versions of your function!

- Create "canary" deploys
- Do blue/green deployments between versions
- Easily shift traffic between versions and see different metrics and logs unique to each
- Test new functionality and roll back quickly if necessary

AWS LAMBDA—TRAFFIC SHIFTING IN THE CLI

```
# Update $LATEST version of function
aws lambda update-function-code --function-name myfunction …

# Publish new version of function
aws lambda publish-version --function-name myfunction

# Point alias to new version, weighted at 5% (original version at 95%
of traffic)
aws lambda update-alias --function-name myfunction --name myalias --
routing-config '{"Additionalversionweights" : {"2" : 0.05} }'
```

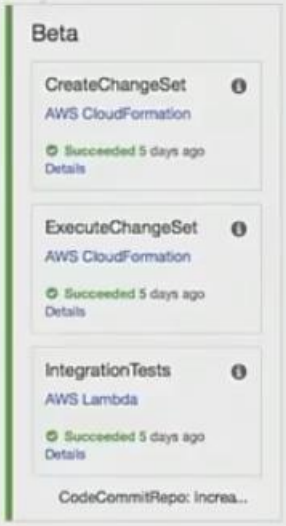
WHAT'S A GOOD PROCESS?

- [x] Easy to interpret
- [x] Frequent and small changes
- [x] Changes have scoped impact
- [ ] Automated deployments

DELIVERY VIA CODEPIPELINE

Pipeline flow:

1. Commit your code to a source code repository
2. Package/test in AWS CodeBuild
3. Use CloudFormation actions in AWS 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



WHAT'S A GOOD PROCESS?

- ☑ Easy to interpret
- ☑ Frequent and small changes
- ☑ Changes have scoped impact
- ☑ Automated deployments

BRINGING IT ALL TOGETHER

Resilience	• Built-in resiliency and high availability • Orchestration available inside and outside of your code
Scalability	• Scales automatically • Pay only for work done
Continuous Delivery	• Easy to model and reason about deployments • Think pure business logic • Can now test locally • Automated process helps deliver frequently and safely

NATIONAL GEOGRAPHIC | FURTHER

NIK KHILNANI
SR. DIRECTOR, PLATFORM DEVELOPMENT
NATIONAL GEOGRAPHIC PARTNERS
nik.khilnani@natgeo.com

NATIONAL GEOGRAPHIC | FURTHER

Products

- TV channels
- Magazines and articles
- Travel and eCommerce
- Kids and live events
- And more

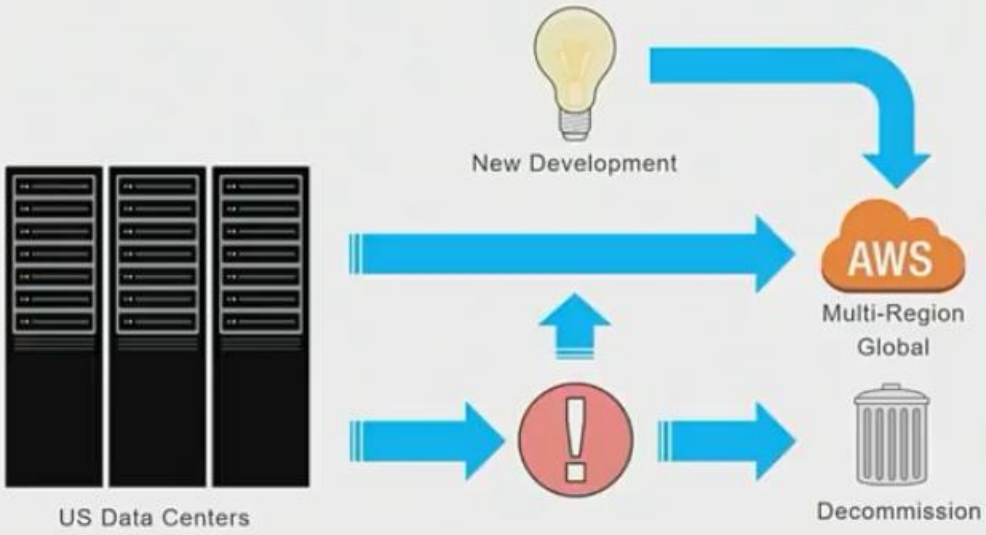
Reach

- Content engagement: 6.4 billion/month
- Households: 485 million
- Social followers: 396 million
- Countries: 172, languages: 43
- Proceeds returned to the non-profit National Geographic Society: 27%

Technology

- Strong emphasis on security
- Independent regional systems
- Heterogeneous system architectures
- Re-platform in progress
- Expanding internationally

RE-PLATFORM | EXPANSION



THE "NATGEO1" MOBILE APP

Localized

- Mobile native application launched in Australia
- Exclusive to Optus, Australia's second largest telecom
- App specific premium content

OPTUS

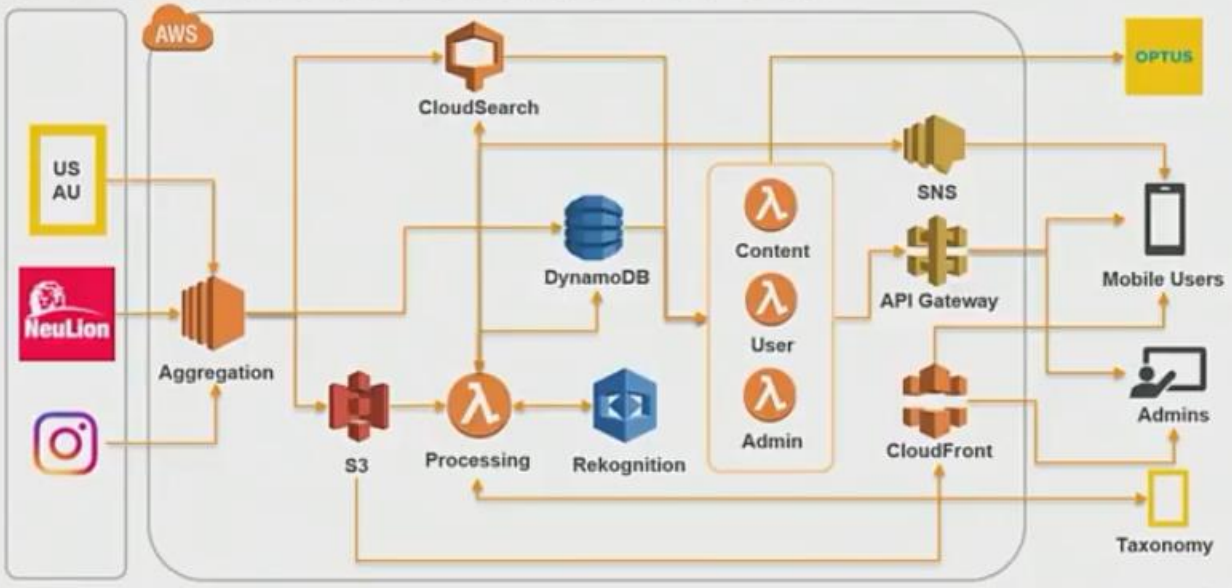
Content

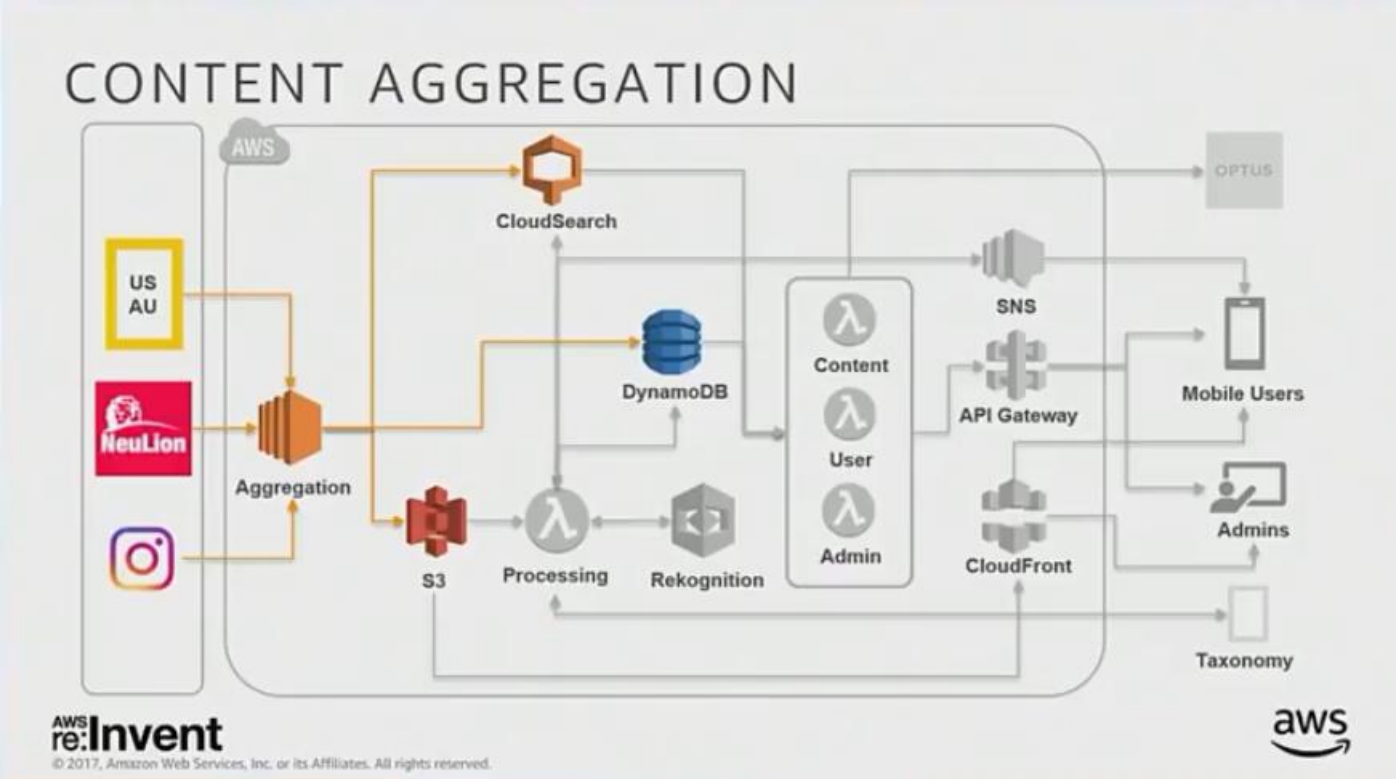
- Content spans full history of Nat Geo—1888 onwards
- Short-form (clips) and long-form (VOD, Live) video
- International magazine and article content
- Social content from Instagram

Personalized experience

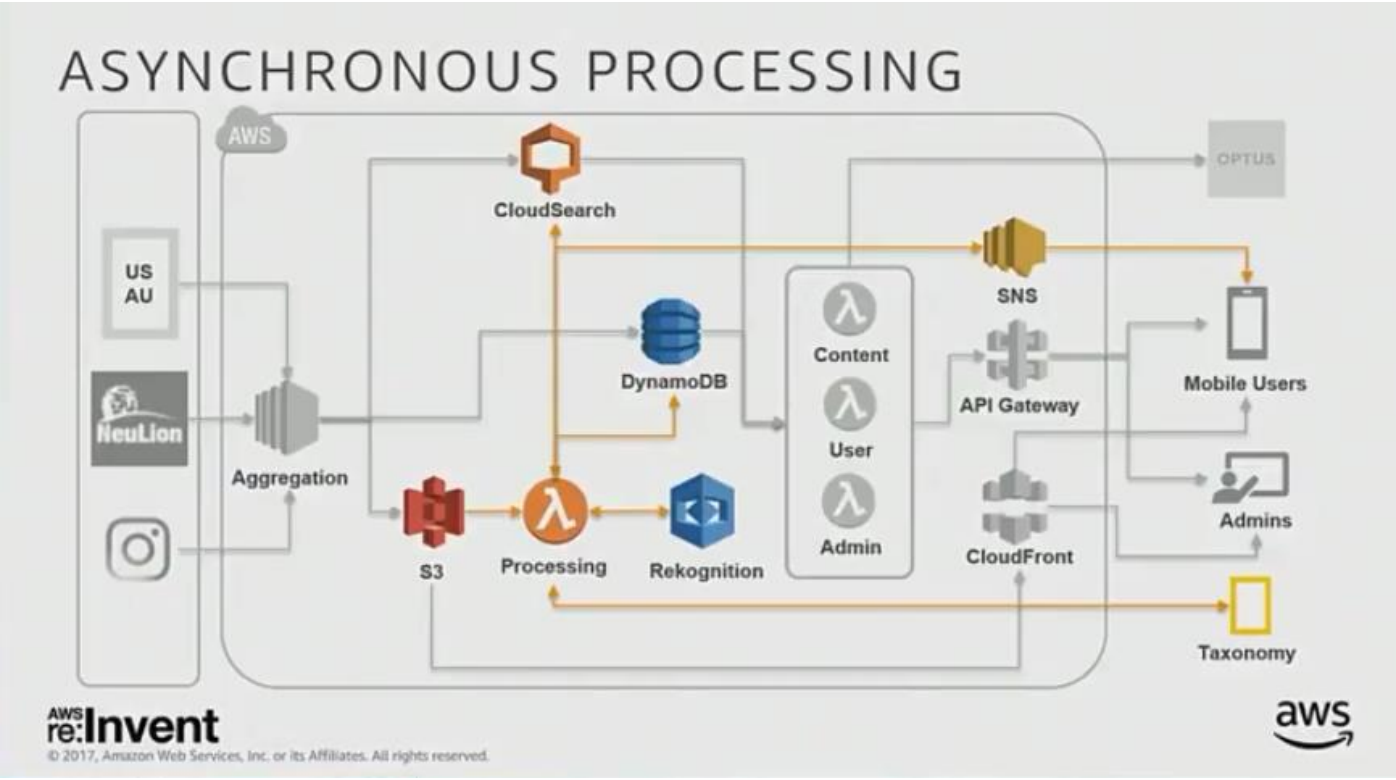
- Content adapts to user activity
- Push notifications
- iOS and Android, with video casting

APPLICATION ARCHITECTURE

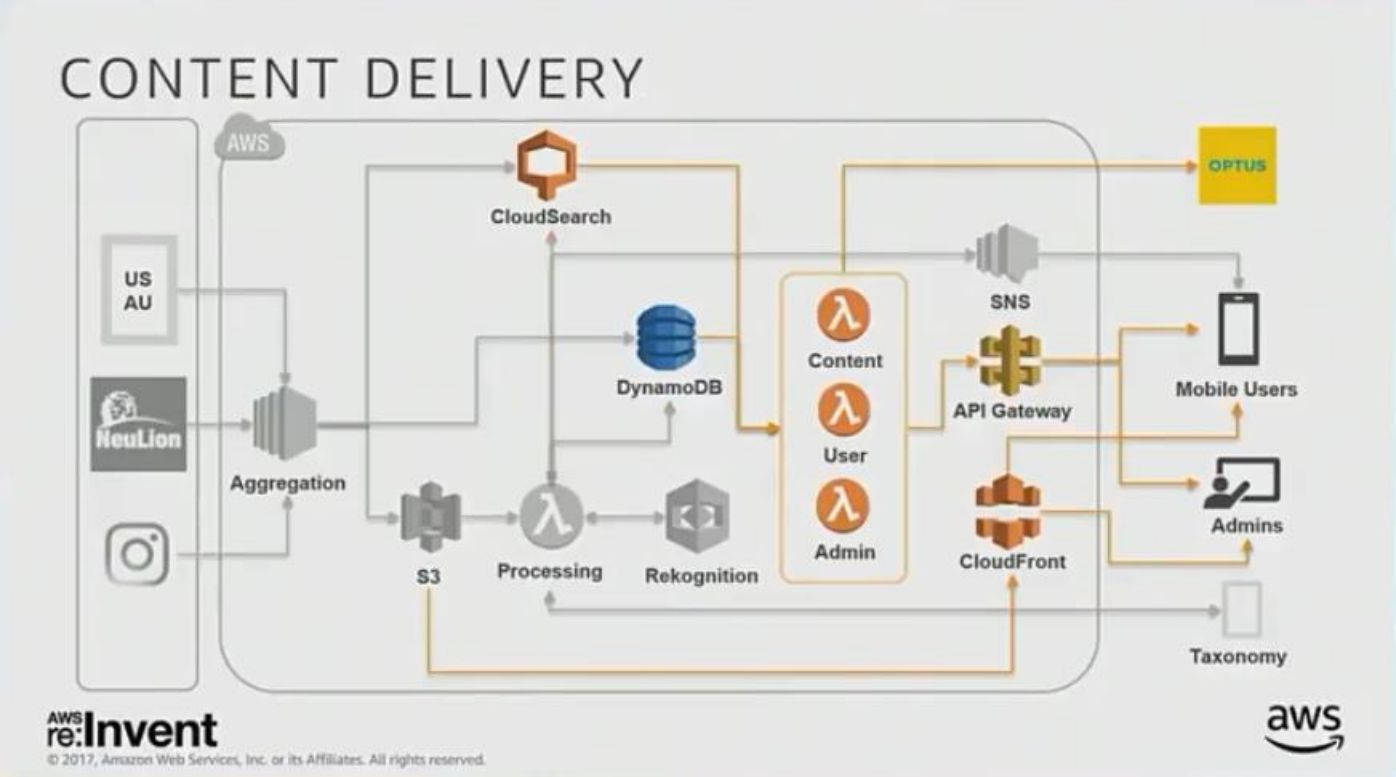




The upstream system helps to aggregate all the needed content and normalized it. We also have different systems for video that has to be aggregated, normalized, formatted for mobile app use and prepared for delivery at the last mile.



We also have lambda functions for doing things like image resizing and cropping for certain mobile formats. We also have lambdas that work with the image recognition service, also lambdas that keep information for the local market



The ***API Gateway*** is broken down into 3 parts, Content, User, and Admin APIs that can work independent of each other. The ***Content APIs*** work with data that is in ***CloudSearch***, which is the primary source of most of the content that goes into the applications. The ***User APIs*** are associated with managing user state and session, what the user is doing inside the application, it is mostly pushing data back into DynamoDB. The ***Admin APIs*** do mostly positioning functions, like what the user can see. Like turning off some articles or content that is not appropriate for a specific market using tagging.

IMAGE CLASSIFICATION

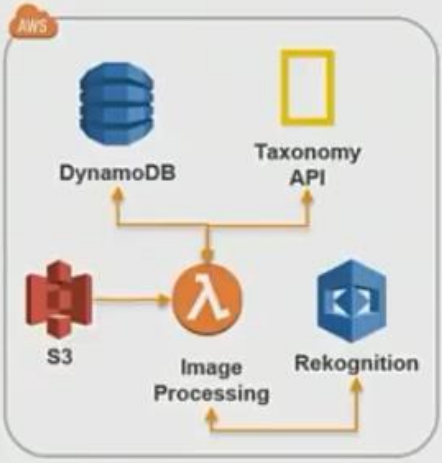
Goal

- Display images within sections of the application without human involvement

Lessons learned

- Plan for service limits early
- Review synchronous versus asynchronous behavior of Amazon Rekognition
- Review classification in relation to larger context of image use
- Lambda managing Lambda is tricky

Labels	Confidence
Arctic	98.9
Glacier	98.9
Ice	98.9
Mountain	98.9
Outdoors	98.9
Snow	98.9

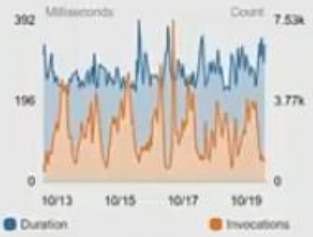


We found that we already have a lot of metadata on the text content but not enough metadata on the images. This affects how we surface content like magazine articles and images to the user when reading our online and print assets. We could do manual image retagging or use the AWS Rekognition service to tag our images for metadata generation to use along with the extra information the service gives in addition to the images tags keywords.

PERFORMANCE

Goal

- Performant microservices
- Low operational overhead



Lambda Invocation Duration

API Gateway Overhead

Lessons learned

- Really understand service nuances
- Understand cold starts/container lifecycle
- Lambda execution consistent across spikes
- Amazon API Gateway adds minimal overhead
- Design with read/write capacity in mind
- Amazon DynamoDB performance is very good

DynamoDB Performance

WHAT'S NEXT

Additional AWS services

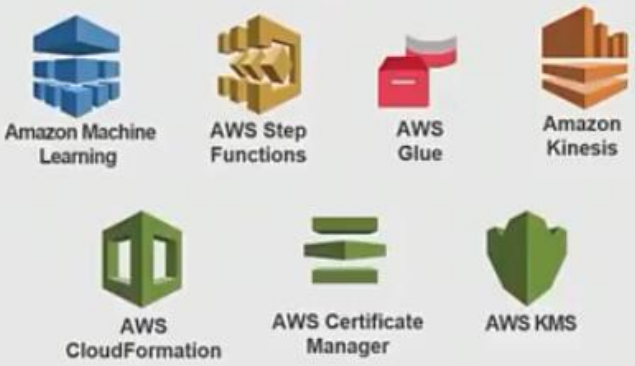
- Step Functions—launched after development started
- Machine learning—take personalization further

Technology improvements

- International expansion
- Continued emphasis on security
- Alignment to 21st Century Fox strategy
- Multi-tenancy and vendor abstractions

Operations and monitoring

- Additional custom metrics
- Improve log analytics
- Infrastructure and application automation



SERVERLESS BENEFITS

High availability

- Continuous scaling
- Concurrency
- Resilience
- Predictability

Increased productivity

- Self-service
- Empowered developers
- Increased reuse
- Focus on feature development

Reduced time-to-market

- Aggressive fixed launch date
- Coordinated launch between Nat Geo and Optus telecom

Batteries included

- Security
- Managed services
- Continuous delivery

Easier innovation

- From on-premises to serverless
- Image classification

Cost effective

- Pay for what you use
- Reduced financial risk