Deep Dive on AWS Lambda

Capabilities, benefits, and best practices

Presenter
Vyom Nagrani
Manager Product Management, AWS Lambda

Q&A Moderators
Stefano Buliani, Solution Architect, AWS Serverless
Vishal Singh, Sr. Product Manager, AWS Lambda
Orr Weinstein, Sr. Product Manager, AWS Lambda

January 18th, 2017

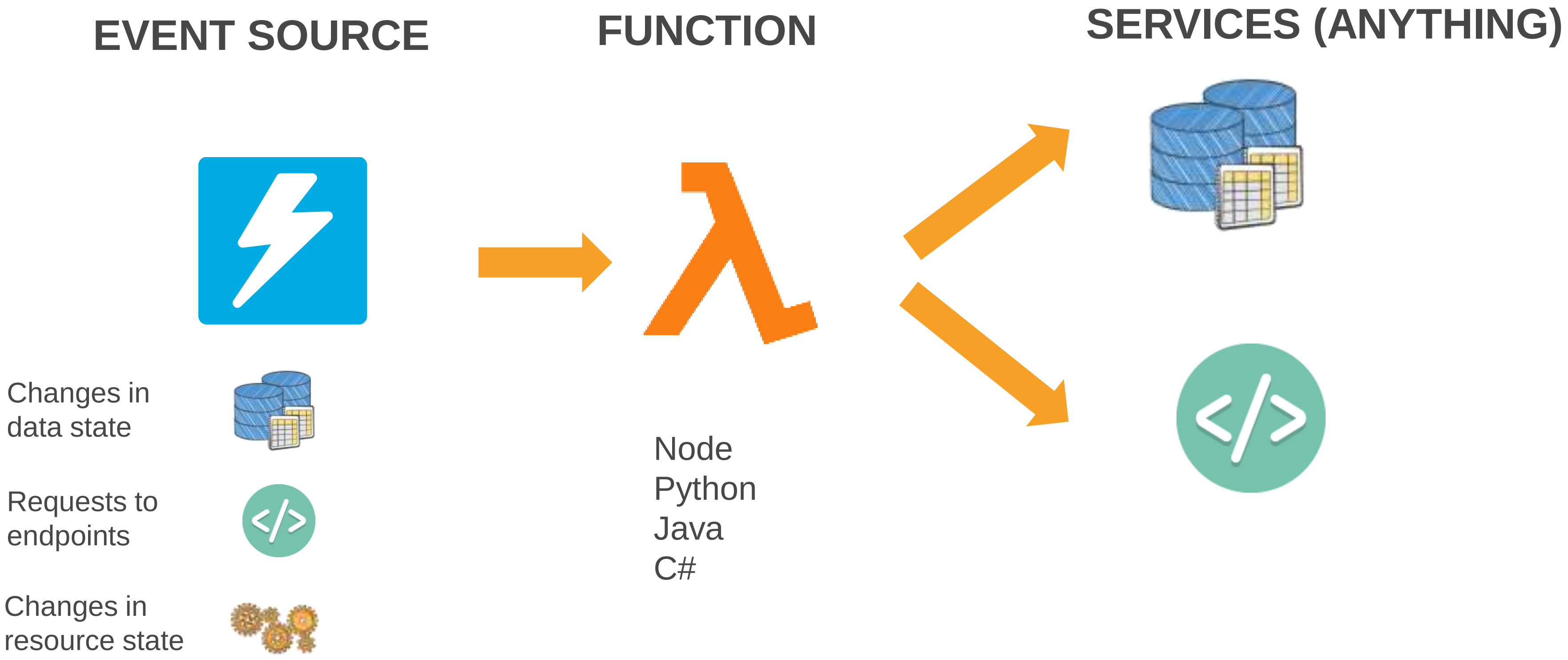
amazon web services | Webinars

Agenda for today’s Webinar

- ➢ Working with AWS Lambda
- ➢ Development and testing on AWS Lambda
- ➢ Deployment and ALM for AWS Lambda
- ➢ Security and scaling on AWS Lambda
- ➢ Debugging and operations for AWS Lambda
- ➢ Questions & answers

Working with AWS Lambda

Working with AWS Lambda



EVENT SOURCE

FUNCTION

SERVICES (ANYTHING)

Changes in data state

Requests to endpoints

Changes in resource state

Node
Python
Java
C#

Benefits of AWS Lambda

Productivity-focused compute platform to build powerful, dynamic, modular applications in the cloud

1

No infrastructure to manage

Focus on business logic

2

Cost-effective and efficient

Pay only for what you use

3

Bring your own code

Run code in standard languages

Event sources that trigger AWS Lambda

DATA STORES

Amazon S3

Amazon DynamoDB

Amazon Kinesis

Amazon Cognito

ENDPOINTS

Amazon Alexa

Amazon API Gateway

AWS IoT

CONFIGURATION REPOSITORIES

AWS CloudFormation

AWS CloudTrail

AWS CodeCommit

Amazon CloudWatch

EVENT/MESSAGE SERVICES

Amazon SES

Amazon SNS

Cron events

… and the list will continue to grow!

Key scenarios and use cases for AWS Lambda

Data processing

Stateless processing of discrete or streaming updates to your data-store or message bus

App backend development

Execute server side backend logic for web, mobile, device, or voice user interactions

Control systems

Customize responses and response workflows to state and data changes within AWS

Development and testing on AWS Lambda

Getting started on AWS Lambda

Bring your own code

- Node.js 4.3, Java 8, Python 2.7, C#

Simple resource model

- Select power rating from 128 MB to 1.5 GB
- CPU and network allocated proportionately

Flexible use

- Synchronous or asynchronous
- Integrated with other AWS services

Stateless

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

Anatomy of a Lambda function

Handler() function

- The method in your code where AWS Lambda begins execution

Event object

- Pre-defined object format for AWS integrations & events
- Java & C# support simple data types, POJOs/POCOs, and Stream input/output

Context object

- Use methods and properties like getRemainingTimeInMillis(), identity, awsRequestId, invokedFunctionArn, clientContext, logStreamName

FunctionConfiguration metadata

VpcConfig

- Enables private communication with other resources within your VPC
- Provide EC2 security group and subnets, auto-creates ENIs
- Internet access can be added though NAT Gateway

DeadLetterConfig

- Failed events sent to your SQS queue / SNS topic
- Redrive messages that Lambda could not process
- Currently available for asynchronous invocations only

Environment

- Add custom key/value pairs as part of configuration
- Reuse code across different setups or passwords
- Encrypted with specified KMS key on server, decrypted at container init

AWS Lambda limits

Resource Limits	Default Limit
Ephemeral disk capacity ("/tmp" space)	512 MB
Number of file descriptors	1024
Number of processes and threads (combined total)	1024
Maximum execution duration per request	300 seconds
Invoke request body payload size (RequestResponse)	6 MB
Invoke request body payload size (Event)	128 K
Invoke response body payload size (RequestResponse)	6 MB
Dead-letter payload size (Event)	128 K
Deployment Limits	**Default Limit**
Lambda function deployment package size (.zip/.jar file)	50 MB
Size of code/dependencies that you can zip into a deployment package (uncompressed zip/jar size)	250 MB
Total size of all the deployment packages that can be uploaded per region	75 GB
Total size of environment variables set	4 KB
Throttling Limits (can request service limit increase)	**Default Limit**
Concurrent executions	100

The container model

Container reuse

- Declarations in your Lambda function code outside handler()
- Disk content in /tmp
- Background processes or callbacks

- Make use of container reuse opportunistically, e.g.
  - Load additional libraries
  - Cache static data
  - Database connections

Cold starts

- Time to set up a new container and do necessary bootstrapping when a Lambda function is invoked for the first time or after it has been updated

- Ways to reduce cold start latency
  - More memory = faster performance, lower start up time
  - Smaller function ZIP loads faster
  - Node.js and Python start execution faster than Java and C#

The execution environment

Underlying OS

- Public Amazon Linux AMI version (*amzn-ami-hvm-2016.03.3.x86_64-gp2*)
- Linux kernel version (*4.4.23-31.54.amzn1.x86_64*)

- Compile native binaries against this environment – can be used to bring your own runtime!
- Changes over time, always check the latest versions supported here

Available libraries

- ImageMagick (nodejs wrapper and native binary)
- OpenJDK 1.8, .NET Core 1.0.1
- AWS SDK for JavaScript version 2.6.9
- AWS SDK for Python (Boto 3) version 1.4.1, Botocore version 1.4.61

- Embed your own SDK/libraries if you depend on a specific version

Deployment and ALM for AWS Lambda

Building a deployment package

Node.js & Python

- .zip file consisting of your code and any dependencies
- Use npm/pip to install libraries
- All dependencies must be at root level

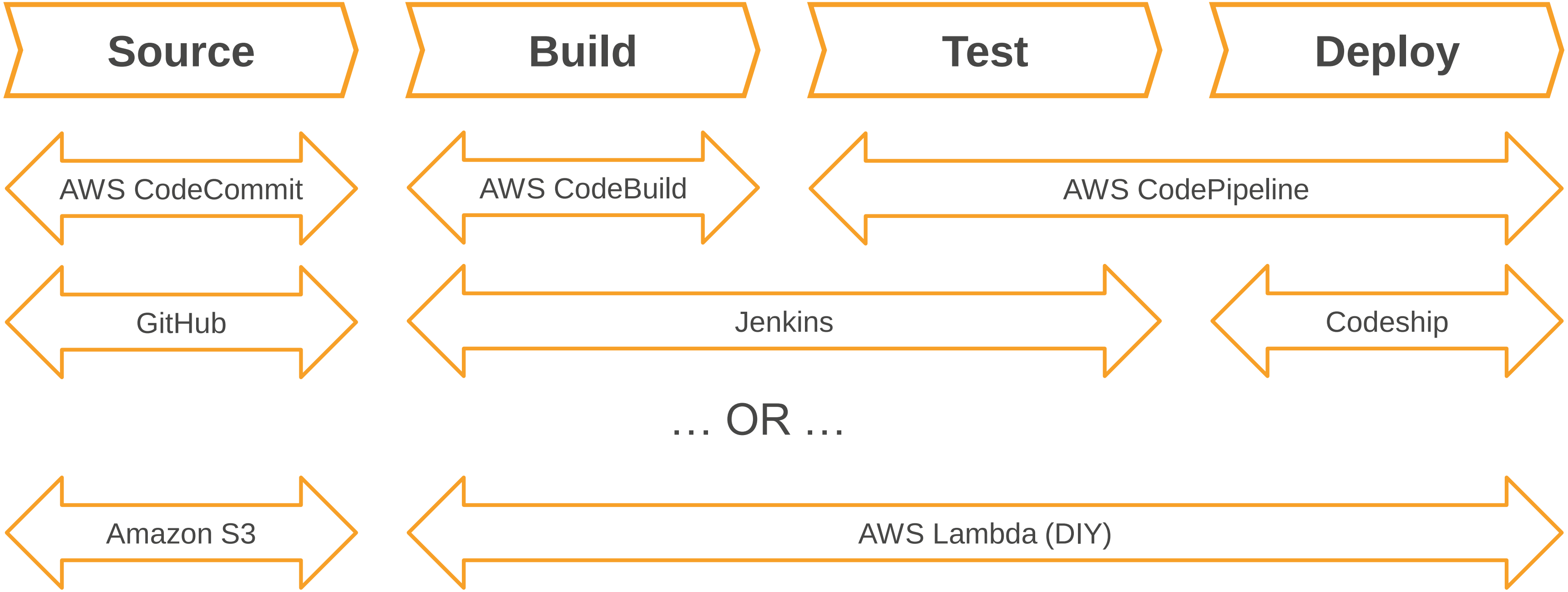
Java

- Either .zip file with all code/dependencies, or standalone .jar
- Use Maven / Eclipse IDE plugins
- Compiled class & resource files at root level, required jars in /lib directory

C# (.NET Core)

- Either .zip file with all code/dependencies, or a standalone .dll
- Use Nuget / VisualStudio plugins
- All assemblies (.dll) at root level

Managing continuous delivery



Source

Build

Test

Deploy

AWS CodeCommit

AWS CodeBuild

AWS CodePipeline

GitHub

Jenkins

Codeship

… OR …

Amazon S3

AWS Lambda (DIY)

Deployment tools and frameworks available

CloudFormation

- AWS Serverless Application Model - extension optimized for Serverless
- New Serverless resources – APIs, Functions, Tables
- Open specification (Apache 2.0)

Chalice

- Python serverless micro-framework
- Quickly create and deploy applications
- Set up AWS Lambda and Amazon API Gateway endpoint
- https://github.com/awslabs/chalice

Third-party tools

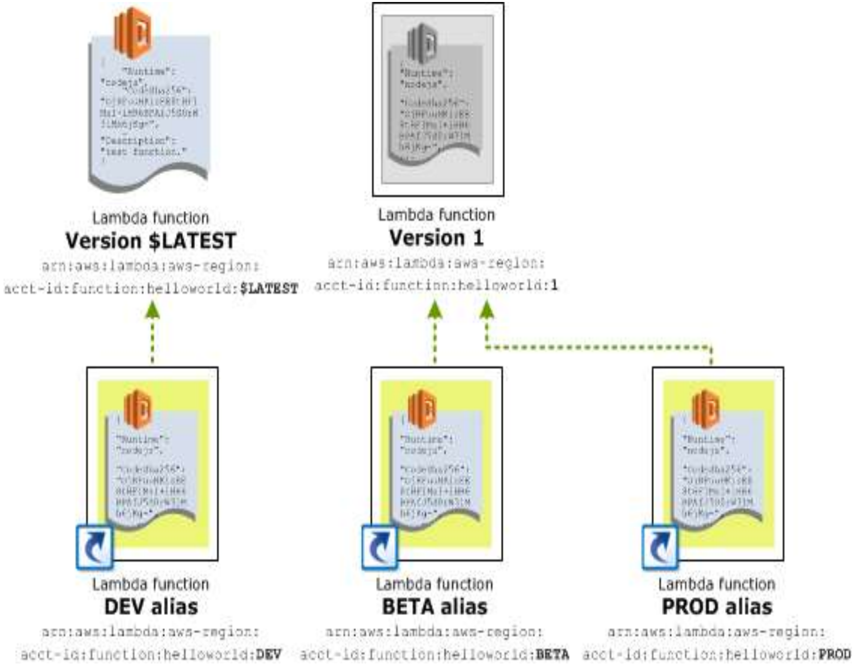
- Serverless Framework (https://serverless.com/)
- Apex Serverless Architecture (http://apex.run/)
- DEEP Framework by Mitoc Group (https://github.com/MitocGroup/deep-framework)

Function versioning and aliases

- Versions = immutable copies of code + configuration
- Aliases = mutable pointers to versions

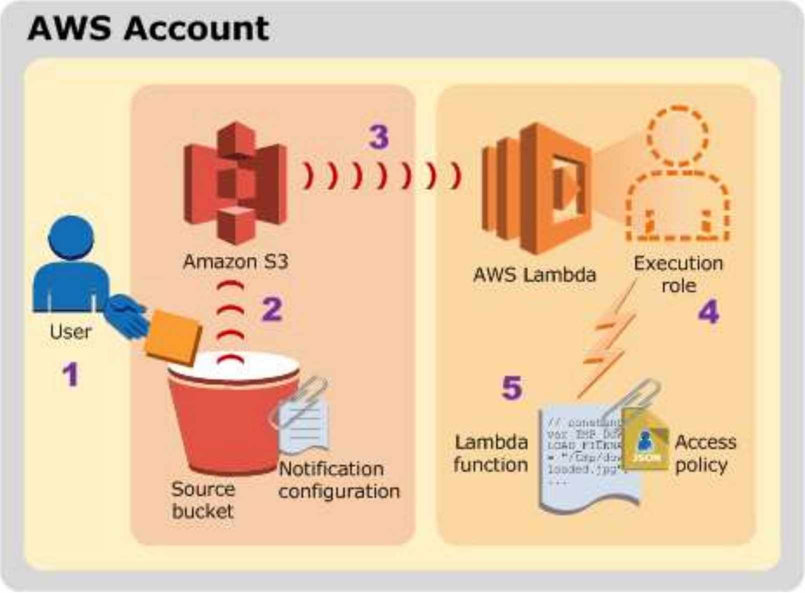
- Development against $LATEST version
- Each version/alias gets its own ARN

- Enables rollbacks, staged promotions, “locked” behavior for client


Lambda function
Version $LATEST
arn:aws:lambda:aws-region:
acct-id:function:helloworld:$LATEST
Lambda function
Version 1
arn:aws:lambda:aws-region:
acct-id:function:helloworld:1
Lambda function
DEV alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:DEV
Lambda function
BETA alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:BETA
Lambda function
PROD alias
arn:aws:lambda:aws-region:
acct-id:function:helloworld:PROD

Security and scaling on AWS Lambda

The push model and resource policies

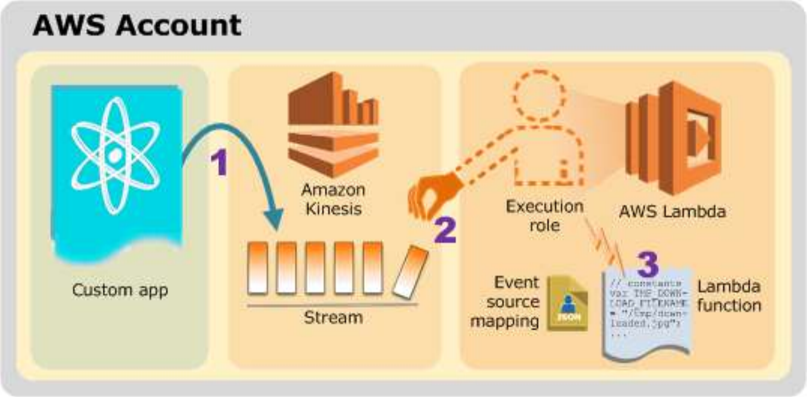


Function (resource) policy

- Permissions you grant to your Lambda function determine which service or event source can invoke your function

- Resource policies make it easy to grant cross-account permissions to invoke your Lambda function

The pull model and IAM roles



```
// constants
var IMG_DOWN-
LOAD_FILENAME
= "/tmp/down-
loaded.jpg";
...
```

IAM (execution) role

- Permissions you grant to this role determine what your AWS Lambda function can do
- If event source is Amazon DynamoDB or Amazon Kinesis, then add read permissions in IAM role

Concurrent executions and throttling

Determining concurrency

- For stream-based event sources: Number of shards per stream is the unit of concurrency
- For all other event sources: Request rate and duration drives concurrency (*concurrency = requests per second * function duration*)

Throttle behavior

- For stream-based event sources: Automatically retried until data expires
- For Asynchronous invocations: Automatically retried for up to six hours, with delays between retries
- For Synchronous invocations: Invoking application receives a 429 error and is responsible for retries

Other scaling considerations

For Lambda

- Remember, a throttle is NOT an error!
- If you expect sudden large spikes in demand, consider Asynchronous invocations to Lambda
- Proactively engage AWS Support to increase your throttling limits

For upstream/downstream services

- Build retries/backoff in client applications and upstream setup
- Make sure your downstream setup “keeps up” with Lambda scaling
- Limit concurrency when connecting to relational databases

Debugging and operations for AWS Lambda

Errors and retries

Types of errors

- 4xx Client Error: Can be fixed by developer, e.g. InvalidParameterValue (400), ResourceNotFound (404), RequestTooLarge (413), etc.
- 5xx Server Error: Most can be fixed by admin, e.g. EC2 ENI management errors (502)

Retry policy

- For stream-based event sources: Automatically retried until data expires
- For Asynchronous invocations: Automatically retried 2 extra times, then published to dead-letter queue
- For Synchronous invocations: Invoking application receives an error code and is responsible for retries

Tracing and tracking

Integration with AWS X-Ray

COMING SOON!

- Collects data about requests that your application serves
- Visibility into the AWS Lambda service (dwell time, number of retries, latency and errors)
- Detailed breakdown of your function’s performance, including calls made to downstream services and endpoints

Integration with AWS CloudTrail

- Captures calls made to AWS Lambda API; delivers log files to Amazon S3
- Tracks the request made to AWS Lambda, the source IP address from which the request was made, who made the request, when it was made
- All control plane APIs can be tracked (no versioning/aliasing and invoke API)

Troubleshooting and monitoring

Logs

- Every invocation generates START, END, and REPORT entries in CloudWatch Logs
- User logs included
  - Node.js – console.log(), console.error(), console.warn(), console.info()
  - Java – log4j.*, LambdaLogger.log(), system.out(), system.err()
  - Python – print, logging.*
  - C# – LambdaLogger.Log(), ILambdaContext.Logger.Log(), console.write(), console.writeline()

Metrics

- Default (Free) Metrics: Invocations, Duration, Throttles, Errors – available as CloudWatch Metrics
- Additional Metrics: Create custom metrics for tracking health/status
  - Function code vs log-filters
  - Ops-centric vs. business-centric

Conclusion and next steps

Key takeaway

AWS Lambda is one of the core components of the platform AWS provides to develop serverless applications

Next steps

1. Stay up to date with AWS Lambda on the Compute blog and check out our detail page for more scenarios.
2. Send us your questions, comments, and feedback on the AWS Lambda Forums.

Thank you!