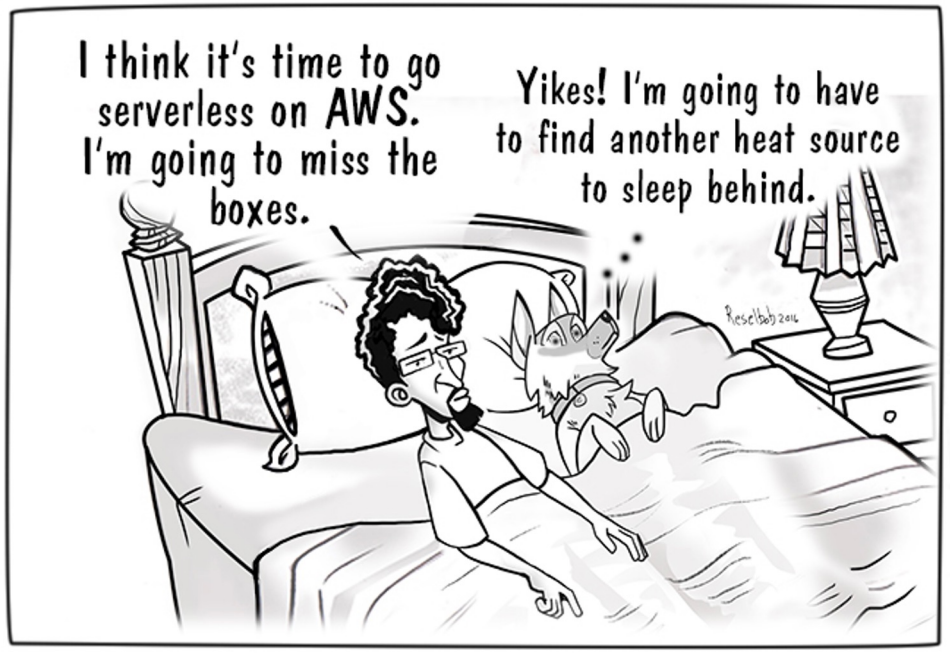
Serverless Computing
aka
Function as a Service (FaaS)

Tejas Parikh
(t.parikh@northeastern.edu)

CSYE 6225
Northeastern University



What is Serverless Computing?

- Serverless computing also known as function as a service (FaaS) refers to a model where the existence of servers is simply hidden from developers. I.e. that even though servers still exist developers are relieved from the need to care about their operation.
- Developers are relieved from the need to worry about low-level infrastructural and operational details such as scalability, high-availability, infrastructure-security, and so forth.
- Serverless computing is essentially about reducing maintenance efforts to allow developers to quickly focus on developing value-adding code.
- Serverless computing encourages and simplifies developing microservices oriented solutions in order to decompose complex applications into small and independent modules that can be easily exchanged.

Units of Scale

- Virtual Machines
  - Machine as the unit of scale
  - Abstracts the hardware
- Containers
  - Application as the unit of scale
  - Abstracts the OS
- Serverless
  - Functions as the unit of scale
  - Abstracts the language runtime

Reactive Computing Design

Code is triggered by events or called by APIs.

Example triggers:

- PUT in Amazon S3 bucket
- Updates to DynamoDB tables
- Message on Kinesis queue
- etc.

Using AWS Lambda

- Bring your own code including libraries
- Code can be connected to other AWS services.
- Call or send eventsAccess controlled via IAM
- Resources allocated via “power” setting. Select from 128mb to 1.5Gb memory and CPU, disk i/o and bandwidth are allocated automatically.

AWS Lambda Concepts

- **Function** – A script or program that runs in AWS Lambda. Lambda passes invocation events to your function. The function processes an event and returns a response.
- **Runtimes** – Lambda runtimes allow functions in different languages to run in the same base execution environment. You configure your function to use a runtime that matches your programming language.
- **Event source** – An AWS service, such as Amazon SNS, or a custom service, that triggers your function and executes its logic.
- **Downstream resources** – An AWS service, such as DynamoDB tables or Amazon S3 buckets, that your Lambda function calls once it is triggered.
- **Log streams** – While Lambda automatically monitors your function invocations and reports metrics to CloudWatch, you can annotate your function code with custom logging statements that allow you to analyze the execution flow and performance of your Lambda function to ensure it's working properly.

Programming Model

- You write code for your Lambda function in one of the languages AWS Lambda supports.
- Regardless of the language you choose, there is a common pattern to writing code for a Lambda function that includes the same core concepts.

Programming Model - Handler

- Handler is the function AWS Lambda calls to start execution of your Lambda function.
- You identify the handler when you create your Lambda function. When a Lambda function is invoked, AWS Lambda starts executing your code by calling the handler function.
- AWS Lambda passes any event data to this handler as the first parameter.
- Your handler should process the incoming event data and may invoke any other functions/methods in your code.

Programming Model - Context

- AWS Lambda also passes a context object to the handler function, as the second parameter.
- Via this context object your code can interact with AWS Lambda.
- For example, your code can find the execution time remaining before AWS Lambda terminates your Lambda function.
- In addition, for languages such as Node.js, there is an asynchronous platform that uses callbacks. AWS Lambda provides additional methods on this context object. You use these context object methods to tell AWS Lambda to terminate your Lambda function and optionally return values to the caller.

Programming Model - Logging

- Your Lambda function can contain logging statements.
- AWS Lambda writes these logs to CloudWatch Logs.
- Specific language statements generate log entries, depending on the language you use to author your Lambda function code.
- Logging is subject to CloudWatch Logs limits.
- Log data can be lost due to throttling or, in some cases, when the execution context is terminated.

Programming Model - Exceptions

- Your Lambda function needs to communicate the result of the function execution to AWS Lambda.
- Depending on the language you author your Lambda function code, there are different ways to end a request successfully or to notify AWS Lambda an error occurred during execution.
- If you invoke the function synchronously, then AWS Lambda forwards the result back to the client.

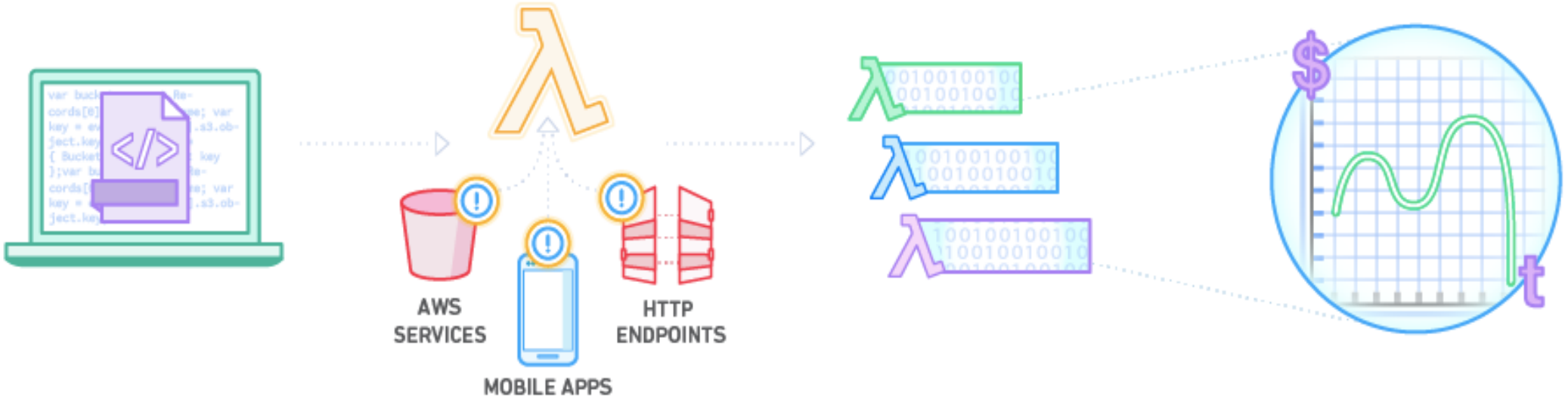
Programming Model - Concurrency

- When your function is invoked more quickly than a single instance of your function can process events, Lambda scales up by running additional instances.
- Each instance of your function handles only one request at a time, so you don't need to worry about synchronizing threads or processes.
- You can, however, use asynchronous language features to process batches of events in parallel, and save data to the /tmp directory for use in future invocations on the same instance.

Programming Model

- “Lambda” is THE web server.
- OS abstractions such as processes, threads, /tmp and sockets are available.
- Stateless - Application state must be stored elsewhere.

How It Works



Upload your code to AWS Lambda

Set up your code to trigger from other AWS services, HTTP endpoints, or in-app activity

Lambda runs your code only when triggered, using only the compute resources needed

Pay just for the compute time you use

How It Works

- AWS Lambda takes care of provisioning and managing resources needed to run your Lambda function.
- When a Lambda function is invoked, AWS Lambda launches a container (that is, an execution environment) based on the configuration information, such as the amount of memory and maximum execution time that you want to allow for your Lambda function.
- After a Lambda function is executed, AWS Lambda maintains the container for some time in anticipation of another Lambda function invocation.
- When you write your Lambda function code, do not assume that AWS Lambda always reuses the container because AWS Lambda may choose not to reuse the container. Depending on various other factors, AWS Lambda may simply create a new container instead of reusing an existing container.

Startup Latency

- It takes time to set up a container and do the necessary bootstrapping, which adds some latency each time the Lambda function is invoked.
- You typically see this latency when a Lambda function is invoked for the first time or after it has been updated because AWS Lambda tries to reuse the container for subsequent invocations of the Lambda function.

Best Practices (Container Re-use)

- Take advantage of container re-use to improve the performance of your function.
- Make sure any externalized configuration or dependencies that your code retrieves are stored and referenced locally after initial execution.
- Limit the re-initialization of variables/objects on every invocation. Instead use static initialization/constructor, global/static variables and singletons.
- Keep alive and reuse connections (HTTP, database, etc.) that were established during a previous invocation.

Best Practices (Use Environment Variables)

- Use Environment Variables to pass operational parameters to your function.
- For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

Best Practices (Control Dependencies)

- Control the dependencies in your function's deployment package.
- The AWS Lambda execution environment contains a number of libraries such the AWS SDK for the Node.js and Python runtimes.
- To enable the latest set of features and security updates, Lambda will periodically update these libraries.
- These updates may introduce subtle changes to the behavior of your Lambda function.
- To have full control of the dependencies your function uses, we recommend packaging all your dependencies with your deployment package.

Best Practices (Manage Deployment Package Size)

- Minimize your deployment package size to its runtime necessities.
- This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation.
- For functions authored in Java or .NET Core, avoid uploading the entire AWS SDK library as part of your deployment package.
- Instead, selectively depend on the modules which pick up components of the SDK you need (e.g. DynamoDB, Amazon S3 SDK modules and Lambda core libraries).

Best Practices (Optimize Deployment Packages)

- Reduce the time it takes Lambda to unpack deployment packages authored in Java by putting your dependency .jar files in a separate /lib directory.
- This is faster than putting all your function’s code in a single jar with a large number of .class files.

Best Practices (Reduce Complexity)

- Minimize the complexity of your dependencies. Prefer simpler frameworks that load quickly on container startup. For example, prefer simpler Java dependency injection (IoC) frameworks like Dagger or Guice, over more complex ones like Spring Framework.

Best Practices (Misc.)

- Write your Lambda function code in a stateless style, and ensure there is no affinity between your code and the underlying compute infrastructure.
- Instantiate AWS clients outside the scope of the handler to take advantage of connection re-use.
- **Make sure you have set +rx permissions on your files in the uploaded ZIP** to ensure Lambda can execute code on your behalf.
- Lower costs and improve performance by minimizing the use of startup code not directly related to processing the current event.
- Use the built-in CloudWatch monitoring of your Lambda functions to view and optimize request latencies.
- Delete old Lambda functions that you are no longer using.

AWS Lambda Limits

Every Lambda function is allocated with a fixed number of specific resources regardless of the memory allocation, and each function is allocated with a fixed amount of code storage per function and per account.

Resource	Quota
Function memory allocation	128 MB to 10,240 MB, in 1-MB increments.
Function timeout	900 seconds (15 minutes)
Function environment variables	4 KB, for all environment variables associated with the function, in aggregate
Function resource-based policy	20 KB
Function layers	five layers
Function burst concurrency	500 - 3000 (varies per Region)
Invocation payload (request and response)	6 MB (synchronous) 256 KB (asynchronous)
Deployment package (.zip file archive) size	50 MB (zipped, for direct upload) 250 MB (unzipped) This quota applies to all the files you upload, including layers and custom runtimes. 3 MB (console editor)
Container image code package size	10 GB
Test events (console editor)	10
`/tmp` directory storage	512 MB to 10,240 MB, in 1-MB increments.
File descriptors	1,024
Execution processes/threads	1,024

Benefits of Serverless Computing

- No servers to manage
- Continuous Scaling
- Never pay for idle servers
- Reduced Operational Costs

Drawbacks

- Loss of Server optimizations
- No in-server state for Serverless FaaS
- Startup Latency
- Vendor Lockin

Use Cases

- Event driven programming
- On-demand Lambda function invocation over HTTPS
- On-demand Lambda function invocation
- Scheduled events

Additional Resources

See Lecture Page