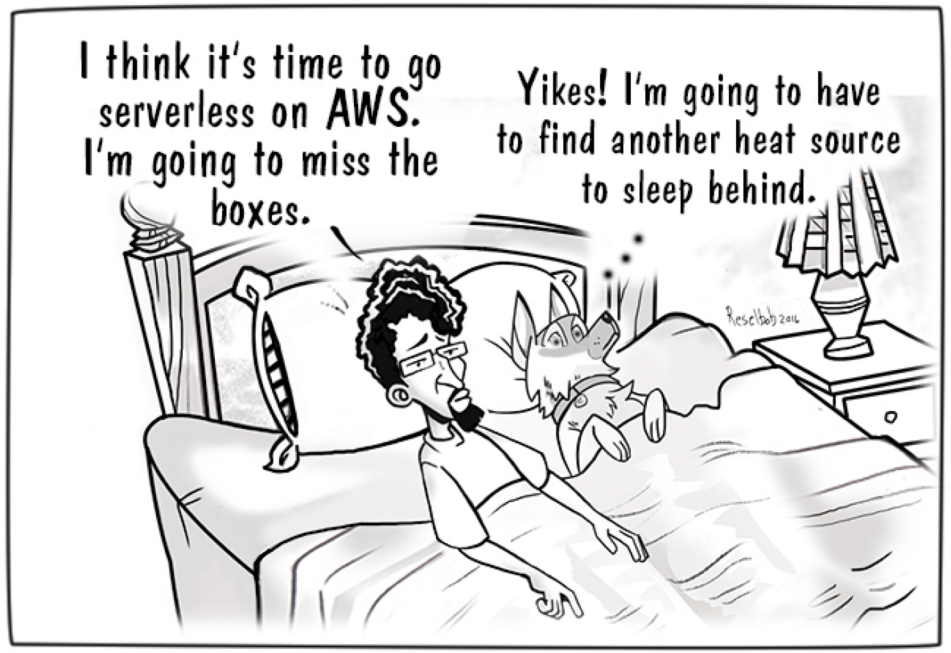
Serverless Computing
aka
Function as a Service (FaaS)

Tejas Parikh (t.parikh@northeastern.edu)
Spring 2018

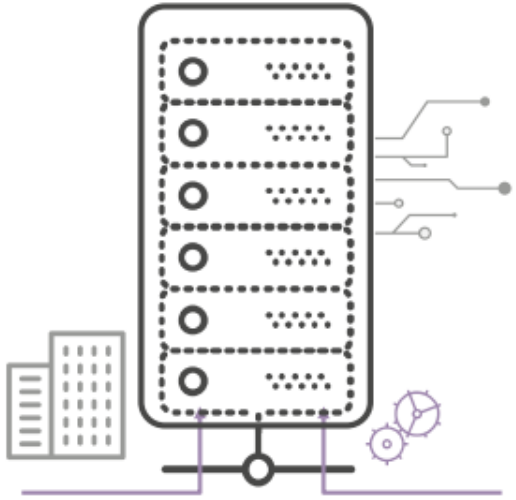
CSYE 6225
Northeastern University
https://spring2018.csye6225.com



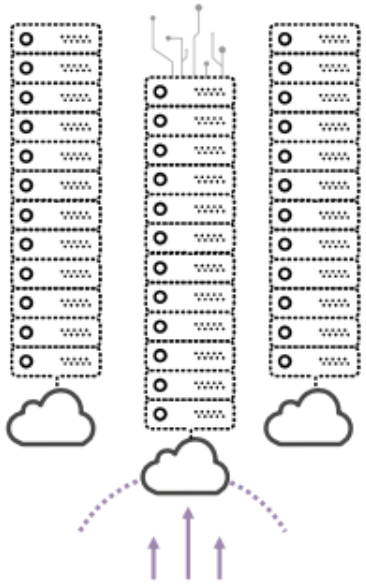
Evolution of Computing

Physical servers in datacenters

Virtual servers in datacenters



Virtual servers in the cloud

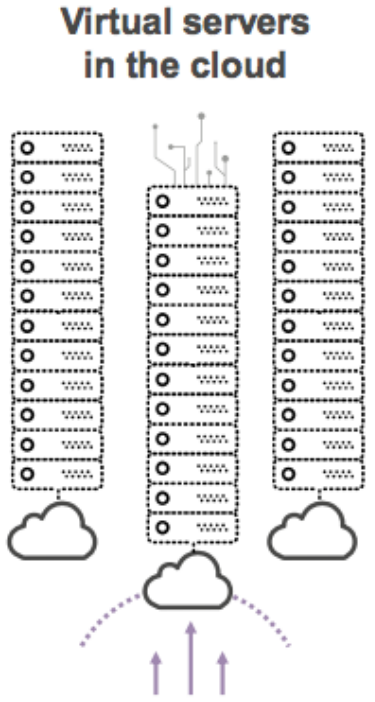


amazon web services

Each progressive step was better

- Higher utilization
- Faster provisioning speed
- Improved uptime
- Disaster recovery
- Hardware independence

- Trade CAPEX for OPEX
- More scale
- Elastic resources
- Faster speed and agility
- Reduced maintenance
- Better availability and fault tolerance

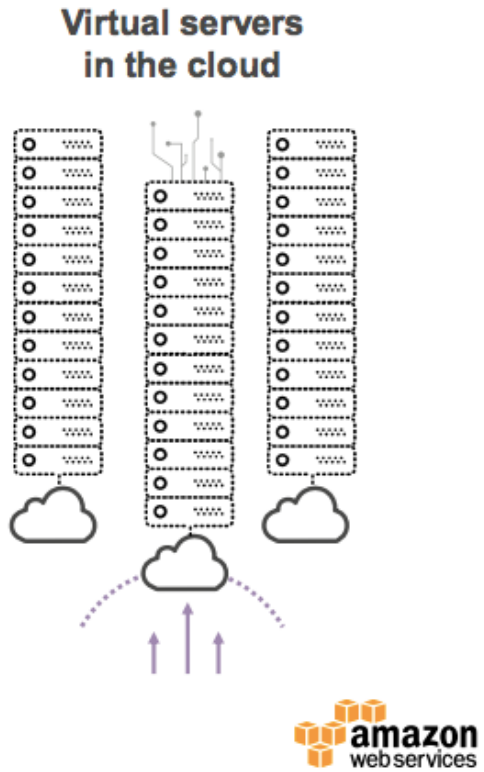


amazon web services

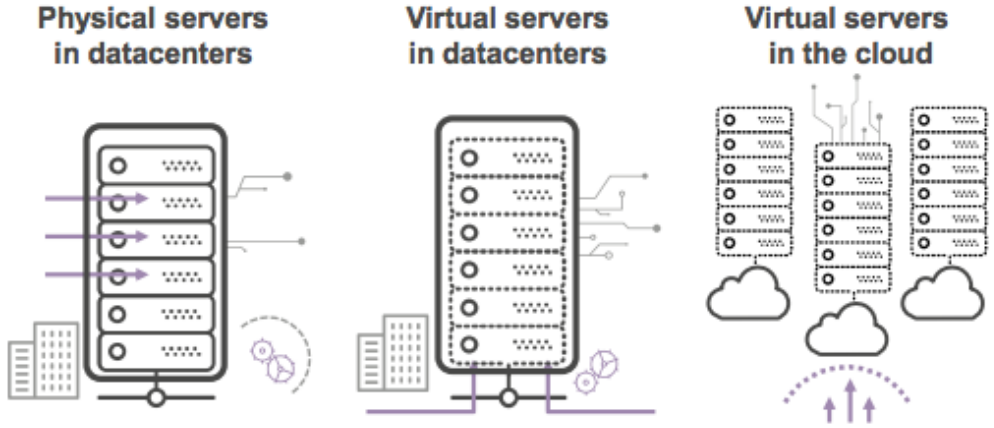
But there are still limitations

- Still need to administer virtual servers
- Still need to manage capacity and utilization
- Still need to size workloads
- Still need to manage availability, fault tolerance
- Still expensive to run intermittent jobs

- Trade CAPEX for OPEX
- More scale
- Elastic resources
- Faster speed and agility
- Reduced maintenance
- Better availability and fault tolerance



Evolving to Serverless



amazon web services

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.

AWS Lambda Pricing

- First 1 million requests per month are free.
- $0.20 per 1 million requests thereafter ($0.0000002 per request).
- Duration is calculated from the time your code begins executing until it returns or otherwise terminates, rounded up to the nearest 100ms.
- The price depends on the amount of memory you allocate to your function. You are charged $0.00001667 for every GB-second used.

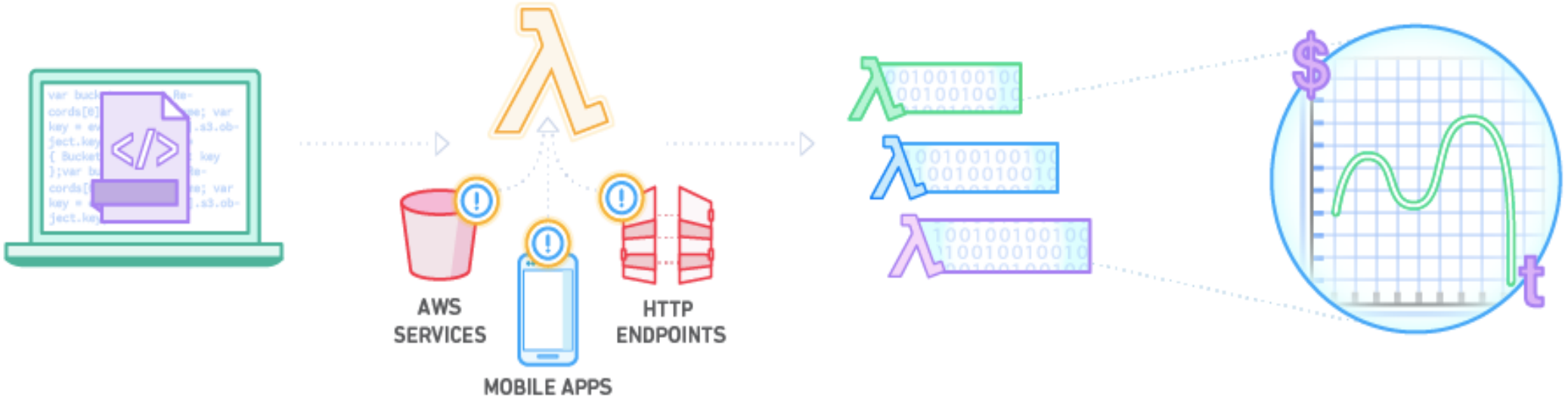
Using AWS Lambda

- Bring your own code (node.js, java or python) 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.

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 amount 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.

AWS Lambda Resource Limits per Invocation

Resource	Limits
Memory allocation range	Minimum = 128 MB / Maximum = 1536 MB (with 64 MB increments). If the maximum memory use is exceeded, function invocation will be terminated.
Ephemeral disk capacity ("/tmp" space)	512 MB
Number of file descriptors	1,024
Number of processes and threads (combined total)	1,024
Maximum execution duration per request	300 seconds
Invoke request body payload size (RequestResponse/synrchronous invocation)	6 MB
Invoke request body payload size (Event/asynchronous invocation)	128 K

AWS Lambda Account Limits Per Region

Resource	Default Limit
Concurrent executions (see Lambda Function Concurrent Executions)	1000

AWS Lambda Deployment Limits

Item	Default Limit
Lambda function deployment package size (compressed .zip/.jar file)	50 MB
Total size of all the deployment packages that can be uploaded per region	75 GB
Size of code/dependencies that you can zip into a deployment package (uncompressed .zip/.jar size). **Note** Each Lambda function receives an additional 500MB of non-persistent disk space in its own `/tmp` directory. The `/tmp` directory can be used for loading additional resources like dependency libraries or data sets during function initialization.	250 MB
Total size of environment variables set	4 KB

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

https://spring2018.csye6225.com/