

A Framework for Automatic Toggle between Serverless and VM Computing Mode Based on Throughput

Anne Martine Augustin
George Mason University

Abstract

Typically, applications that roll with a serverless computing paradigm are those that do not issue high-volume traffic, considering the high cost of per-function trigger cost. However, many cloud data-intensive applications exhibit fluctuate diurnal patterns, where the daily traffic shows spikes and troughs. At peak hours, applications serve millions of requests per second, and therefore, utilizing serverless computing would incur significant monetary cost. During idle times, VM-based application deployment suffers from cost inefficiency since tenants pay for a running VM even if the applications running inside the VM is mostly idle. This paper presents a framework that effectively combines the best of both world – a framework that automatically switches to serverless computing mode when traffic is low, and switches to the VM mode when traffic is high.

1 Introduction

Cost-efficiency is a turning point in cloud adoption decision. Many aim to cut costs while achieving desired performance. Amazon Web Services, a leader in cloud computing, offers solutions in a wide range of areas including machine learning, Internet of Things, storage, containers and serverless computing. AWS Lambda, a serverless compute service, allows you to run code without provisioning or managing servers. You only pay when you code is running [2]. While this can be ideal for short-lived bursty applications, the cost for Lambda can quickly ramp up when the number of invocation increases. On the other hand, EC2 instances designed to make web-scale cloud computing friction free, provides secure, resizable compute capacity. There are four ways to pay for Amazon EC2 instances: On-Demand, Reserved Instances, Spot Instances and Dedicated Hosts [1]. With on-demand instances, you pay for compute capacity per hour or per second depending on which instances you run. Thus, you can't scale down to 0 like with Lambda. You need to keep a running instance at any time. This paper proposes a framework that can effectively

combine those two services. The framework uses a threshold to toggle between AWS Lambda and EC2 instances.

The rest of the document is organized as follows. Section 2 outlines related work on cost efficiency in the cloud. Section 3 presents the design of the framework. Section 4 discusses the results obtained for a case study of a simple web application.

2 Related Work

With increasing interest in serverless computing, more researches are being published on different aspects of the cloud. OpenLambda [6] is an open-source platform for building web services and applications using the serverless computing model. Many authors have presented works on improving cost-efficiency in the cloud with the right combination of available services. Conley [5] proposed a systematic methodology for measuring the I/O capabilities of high performance VMs with the objective to determine the optimal price points for data-intensive applications. Pérez [7] designed a programming model that allows the creation of highly-parallel event-driven file-processing serverless applications. Roloff [8] used multiple instances to create heterogeneous cluster systems in the cloud. He came to the conclusion that heterogeneous clouds can have a better cost efficiency than homogeneous systems.

Our contribution is the design of a framework that embeds the advantages of AWS Lambda and EC2 instances by automatically switch to one mode based on the incoming traffics. This proposed solution is different from previous works as it adapts dynamically to the incoming traffic as the number of request per second varies.

3 Design

The main objective of the framework is to successfully switch to the VM-based mode when the traffic is high then toggle back to the serverless mode when the traffic decreases. Figure 1 shows the proposed architecture. Two main targets are

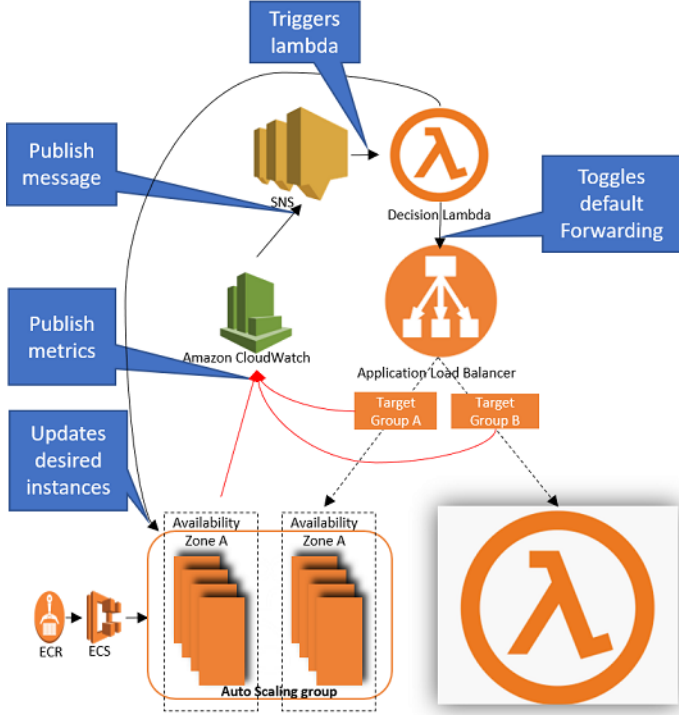


Figure 1: Proposed architecture in Amazon Web Services

configured for the deployment. For the VM-based mode, we use Amazon Elastic Container Service (Amazon ECS) to deploy a container image on EC2 instances. For the serverless mode, we use an AWS Lambda function. An application load balancer distributes incoming traffic to one of those two targets. When the traffic goes above or below a certain threshold, we trigger a lambda function that updates the default target of the load balancer. We discuss more in details each part of the architecture in this section.

3.1 Heterogenous computing mode

The biggest advantage of the proposed computing model is the combination of serverless and VM-based deployment. AWS Lambda is an event-driven compute service that automatically manages the compute resources. It has the capacity to scale down to zero, costing only when the code is running. Amazon EC2 Container Service (Amazon ECS) is a highly scalable, high performance container management service that supports Docker containers and allows applications to be run easily on a managed cluster of Amazon EC2 instances [4]. Our proposed architecture leverage the capabilities of those two services. We manually define our lambda function, our cluster, its services, scaling policies and tasks.

3.2 Load balancer with toggling target group

Another key part of our architecture is the load balancer. We assume that a load balancer would have been used and therefore do not add up to the cost. We use Elastic Load Balancing featuring scalability, availability and robust security. There are three types of balancers: application, classic and network. The one that interests us is the application type best suited for load balancing of HTTP and HTTPS traffics. The load balancer uses the listener rules to route requests to targets. Therefore, our load balancer is configured with two listener rules. One for HTTP protocol on port 80 and another one for HTTPS protocol on port 443 rules. We have also define two target groups. A target group pointing to our lambda function and the second one attached to our cluster.

3.3 CloudWatch alarms

Many services used in our achitecture publish metrics to CloudWatch. We configure three alarms using metrics from the two target groups. Whenever the alarms go off, Clouwatch publish a message to a SNS topic:

1. The first alarm uses *RequestCount* from the Lambda target group of the load Balancer. When the number of requests for a datapoint is above the threshold, the alarm goes off. In order for the alarm to stay in its Ok status even when Cloudwatch is not receiving any data from the lambda target group, we treat missing data as good (not breaching threshold).
2. The second alarm is the opposite of the first one. It uses *RequestCount* from the EC2 target group of the load Balancer. When the number of requests is below the threshold, the alarm goes off. In order for the alarm to still go off even Cloudwatch is not receiving any data from the EC2 target group. We treat missing data as bad (breaching threshold).
3. The last one goes off when there is at least one healthy host in the EC2 target. The *Healthy Hosts* metric of the EC2 target group is used.

3.4 Decision Lambda triggered through SNS

We finally use boto3 to define a lambda function that subscribes to the SNS topic of our alarms. It updates the default target of our load balancer based on the alarm that triggers it. When the number of request goes below the threshold. The function updates the two listeners of the load balancer to forward their request to lambda and then shut down the EC2 instances by setting the desired capacity for the auto scaling group of the cluster to 0.

```
if (message == "switchToLambda") :
    client.modify_listener(
```

```

ListenerArn= 'arn:aws:mainloadbalancer
  /httplistener ',
DefaultActions=[
  {
    'Type': 'forward ',
    'TargetGroupArn': 'arn:aws:
      targetgroup/lambda ',
  },
]
)
client.modify_listener(
ListenerArn='arn:aws:mainloadbalancer/
  httpslistener ',
DefaultActions=[
  {
    'Type': 'forward ',
    'TargetGroupArn': 'arn:aws:
      targetgroup/lambda ',
  },
]
)
asg.set_desired_capacity(
  AutoScalingGroupName='amazon-ecs-
    asg ',
  DesiredCapacity=0
)
return "Successfully switched to
  Lambda"

```

When the number of request for Lambda goes above the threshold, the function relaunch the EC2 instances by setting back the desired capacity for the auto scaling.

```

if (message == "switchToEC2"):
  asg.set_desired_capacity(
    AutoScalingGroupName='amazon-ecs-
      asg ',
    DesiredCapacity=capacity
  )
  return "Launching instances - will
    switch when instances start"

```

However this time, it doesn't immediately update the load balancer default target to EC2. It will trigger again when the load balancer is able to successfully forward request to the EC2 instances.

```

if (message == "EC2available"):
  client.modify_listener(
    ListenerArn= 'arn:aws:mainloadbalancer
      /httplistener ',
    DefaultActions=[
      {
        'Type': 'forward ',
        'TargetGroupArn': 'arn:aws:
          targetgroup/EC2 ',

```

```

      },
    ]
  )
  client.modify_listener(
    ListenerArn='arn:aws:mainloadbalancer/
      httpslistener ',
    DefaultActions=[
      {
        'Type': 'forward ', Section ~\ref{
          sec:design} presents the
          design of the framework.
        'TargetGroupArn': 'arn:aws:
          targetgroup/EC2,
      },
    ]
  )
  return "Successfully switched to EC2"

```

4 Case study

In this section, we present the cost analysis to define our threshold as well as the results we obtained for the deployment of a simple web page. Our lambda function simply returns an HTTP message. We also containerized a simple python application that also prints a simple HTTP message for the image of our cluster.

4.1 Cost analysis

For our lambda function, we have allocated 128 MB of memory and the average execution of our Lambda is below 100ms. We choose t3.nano as the instance characteristic of our cluster and set the number of instance to 1. We set the default target of our load balancer to the Lambda function. Table

Resource	Description	Cost per hour
Lambda	128MB memory	\$0.0073743
t3.nano	2 vCPUs 0.5 GiB memory	\$0.0052

Table 1: Description of test environment

¹ summarizes the chosen resources and their cost per hour. We used the equations below to estimate the cost per hour of the Lambda function for 30M invocations per month with an average execution duration of 100 ms.

$$TotalCostLambda = computeCost + invocationCost$$

with

$$computeCost = (totalCompute^1 - 400000^2) * 0.00001667^3$$

¹Expressed in Gb-s

²Free Tier compute

³Monthly compute price per GB-s

$$invocationCost = (nbInvocation - 10000000^4) * 0.0000002^5$$

Once we decided on the configuration of our test environment, the next step was to define the threshold for our alarm. Our objective is to keep the cost per hour of our lambda function below the cost per hour of our instance. We estimated that 26M invocations to our Lambda function will cost us approximately \$0.0051 per hour. Thus, we set our threshold to

$$26000000 / (730 * 60) = 605$$

. That is whenever, the number of request to lambda for the past minute is equal or greater to 605, we switch to EC2. Whenever, it goes below 605 we switch to Lambda.

4.2 Results

Finally we used wrk [3] to test our framework. We conducted three different test phases for about 20 minutes:

1. The first phase consisted of two benchmarks using 1 thread and keeping 1 connection open. One benchmark produced 119 requests in 5.08s with an average latency of 50.08ms. The other one generated 399 requests in 15.08s with an average latency of 42.27ms.
2. The second phase started with a benchmark using 1 thread and keeping 2 connections open. It generated 767 requests in 15.08s with an average latency of 42.42ms. We then increased the time to 30s for the next two benchmarks. One producing 2799 request with an average latency of 23.04ms and the second one produced 2869 requests with an average latency of 21.09ms.
3. We completed our test with two benchmarks: one benchmarks of 19083 requests in 180s with an average latency of 19.53 ms and the last one of 12273 requests in 240s with an average latency of 40.59ms. Table 2 summarizes the phases for the benchmark test.

To estimate the cost of our case study, we do not take into account the free tier. We evaluated the cost based on the number of requests received by each target as reported in Cloudwatch metrics. The cost for Lambda was estimated for an allocation of 128 MB and the minimum execution duration of 100ms. The cost for the EC2 instance is the cost for a minute regardless the number of requests it received. We plot the cost (in blue), if the EC2 instance was not used. The cost of our solution is in orange.

As shown in Figure 2, between 10 and 15 minutes, the requests are only handled by the EC2 instance and would have cost more if we forwarded them to our lambda function.

⁴Free Tier invocation

⁵Monthly invocation cost

No. requests	Average Latency
Phase 1	
119	50.8ms
399	42.27ms
Phase 2	
767	42.42ms
2799	23.04ms
2869	21.09ms
Phase 3	
19083	19.53ms
12273	40.59ms

Table 2: Benchmark Test

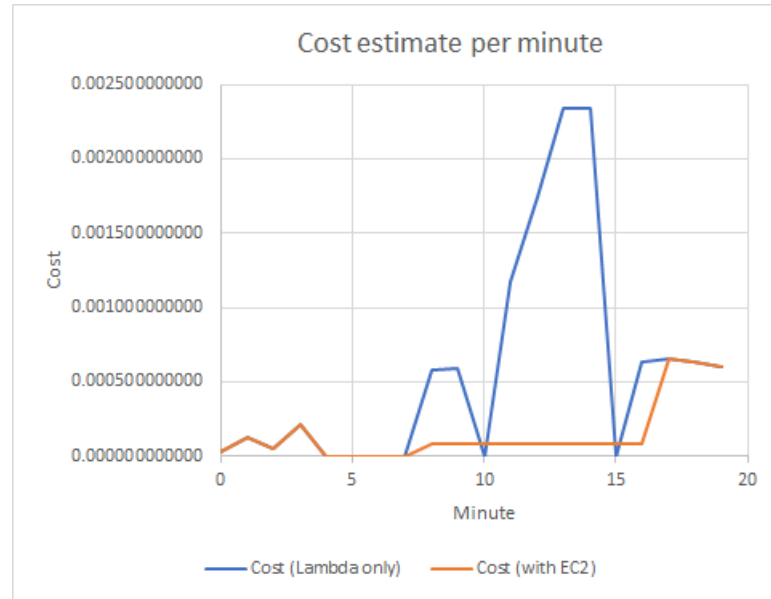


Figure 2: Cost estimate

Conclusion

This framework combines AWS Lambda and EC2 instances for applications with fluctuate diurnal patterns. The configuration we chose for our case study show better performance using Lambda. However, more performant EC2 instances can be implemented and the threshold can be accordingly updated. One possible improvement is taking into account the patterns of the incoming requests to define the threshold. This will have two main benefits:

1. Prevent unnecessary trigger of the decision lambda
2. Allow time for EC2 instances to start and terminate accordingly before another scaling action.

Acknowledgments

This study was proposed and conducted in collaboration with Professor Yue Cheng at George Mason University.

References

- [1] Amazon ec2. <https://aws.amazon.com/ec2/>. Accessed: 2019-06-10.
- [2] Aws lambda. <https://aws.amazon.com/lambda/>. Accessed: 2019-06-10.
- [3] wrk. <https://github.com/wg/wrk>. Accessed: 2019-06-17.
- [4] Chris Barclay. Better together: Amazon ecs and aws lambda. <https://aws.amazon.com/blogs/compute/better-together-amazon-ecs-and-aws-lambda/>. Accessed: 2019-05-20.
- [5] Michael Conley, Amin Vahdat, and George Porter. Achieving cost-efficient, data-intensive computing in the cloud. In *SoCC '15*, pages 302–314, 2015. <https://doi.org/10.1145/3297280.3297292>.
- [6] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.
- [7] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. A programming model and middleware for high throughput serverless computing applications. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC)*, pages 106–113, 2019. <https://dx.doi.org/10.1145/2806777.2806781>.
- [8] Eduardo Roloff, Matthias Diener, Emmanuell D. Carreno, Francis B. Moreira, Luciano P. Gasparry, and Philippe O. A. Navaux. Exploiting price and performance trade-offs in heterogeneous clouds. In *UCC '17 Companion*, pages 71–76, 2019. <https://dx.doi.org/10.1145/3147234.3148103>.