Channel Islands

CALIFORNIA STATE UNIVERSITY

AWS NoAuto Scaling Group

A Thesis Presented to

The Faculty of the Computer Science Department

In (Partial) Fulfillment

of the Requirements for the Degree

Masters of Science in Computer Science

by

Student Name:
Kaveh Arashvand

Advisor:
Dr. Michael Soltys

May 2021

APPROVED FOR MS IN COMPUTER SCIENCE

05/27/2021

Advisor: Dr. Michael Soltys **Date**

May 27, 2021

Dr. Brian Thoms **Date**

05/26/2021

Dr. Bahareh Abbasi **Date**

APPROVED FOR THE UNIVERSITY

Jill Leafstedt **Date**

Non-Exclusive Distribution License

In order for California State University Channel Islands (CSUCI) to reproduce, translate and distribute your submission worldwide through the CSUCI Institutional Repository, your agreement to the following terms is necessary. The author(s) retain any copyright currently on the item as well as the ability to submit the item to publishers or other repositories.

By signing and submitting this license, you (the author(s) or copyright owner) grants to CSUCI the nonexclusive right to reproduce, translate (as defined below), and/or distribute your submission (including the abstract) worldwide in print and electronic format and in any medium, including but not limited to audio or video.

You agree that CSUCI may, without changing the content, translate the submission to any medium or format for the purpose of preservation.

You also agree that CSUCI may keep more than one copy of this submission for purposes of security, backup and preservation.

You represent that the submission is your original work, and that you have the right to grant the rights contained in this license. You also represent that your submission does not, to the best of your knowledge, infringe upon anyone's copyright. You also represent and warrant that the submission contains no libelous or other unlawful matter and makes no improper invasion of the privacy of any other person.

If the submission contains material for which you do not hold copyright, you represent that you have obtained the unrestricted permission of the copyright owner to grant CSUCI the rights required by this license, and that such third party owned material is clearly identified and acknowledged within the text or content of the submission. You take full responsibility to obtain permission to use any material that is not your own. This permission must be granted to you before you sign this form.

IF THE SUBMISSION IS BASED UPON WORK THAT HAS BEEN SPONSORED OR SUPPORTED BY AN AGENCY OR ORGANIZATION OTHER THAN CSUCI, YOU REPRESENT THAT YOU HAVE FULFILLED ANY RIGHT OF REVIEW OR OTHER OBLIGATIONS REQUIRED BY SUCH CONTRACT OR AGREEMENT.

The CSUCI Institutional Repository will clearly identify your name(s) as the author(s) or owner(s) of the submission, and will not make any alteration, other than as allowed by this license, to your submission.

Title of Item

3 to 5 keywords or phrases to describe the item

Author(s) Name (Print)

Author(s) Signature Date

AWS NoAuto Scaling Group

Kaveh Arashvand

May 25, 2021

Abstract

Amazon Web Services (AWS) is one the most popular on demand cloud computing services providers, currently offering over 175 fully featured services globally. One of these services, which is highly on demand by various clients all around the world, is Amazon Elastic Compute Cloud (EC2): a commercial web service which allows customers to rent their computing resources. Amazon Elastic Compute Cloud (EC2) provides storage, processing, and other services to its customers. In this thesis, I propose an approach to minimize the costs of performing automatic scaling over Auto Scaling groups: a collection of EC2 instances, by introduction of NoAuto Scaling groups.

Contents

List of Figures

List of Tables

1 Introduction

Amazon Web Services (AWS) is one of the pioneers of cloud computing services providers in the world, which has been offering IT infrastructure services to companies and individuals in the form of web services since 2006 and in over 190 countries [**1**]. There are many benefits using cloud computing, some of them are [**2**]:

- Agility: Quick and easy access to a vast variety of technologies and solutions.
- Reliability: The ability of performing tasks and functions correctly and constantly when they are expected to.
- Elasticity: Clients can utilize resources in regards to their real demands with no need of resources' over-provisioning to handle unforeseen peak levels of business needs in the future.
- Cost Efficiency: Maintenance of data centers at any size (from small ones consisting of a few number of servers to enterprise level data centers) would cost a lot of money and time. However, the Amazon Web Services allows customers to only pay for their services as they consume it.

There are also different types of cloud computing policies provided by AWS which can be directly compared. The mutual aim of these services is "giving

customers the ability to focus on what matters most and avoid undifferentiated work like procurement, maintenance, and capacity planning" [**3**]. The main three policy types are as follows:

- Infrastructure as a Service (IaaS): It contains the building blocks of cloud IT with the provision of sufficient amount of computing and networking features.

- Platform as a Service (PaaS): Eliminates the need of being concerned about the infrastructures, instead directing attention to a focus on customers and developers that would be mostly operating via their applications frameworks.

- Software as a Service (SaaS): A completed product that clients can take advantages of while it is being run and managed by AWS [**3**].

As it is mentioned earlier, Amazon Web Services provides more than 175 various services to over 190 countries globally, one of them which is on high demanded by clients at the current time is the Amazon Elastic Compute Cloud(EC2). Amazon Elastic Compute Cloud provides scalable computing capacity in the Amazon Web Services Cloud and it is categorized as IaaS. Scalability means the ability of adding or removing EC2 instances anytime and from anywhere based on the needs of the clients. This is a huge advantage which not only eliminates some costs (cost efficiency), but also it provides agility and reliability.

For a better understanding of what scalability means, let's imagine we require two EC2 instances (servers) for keeping our online market up and running. During most days of the year, those two instances have sufficient resources to keep our online market responsive enough for incoming shopping requests like searching among items, add and remove items to/from shopping carts, getting invoices, managing payments and so forth; however, either randomly or at very specific days or weeks (scheduled), the number of shopping requests increases nationally and consequently more shoppers would reach to our online market to check and buy from. If those two EC2 instances (servers) would not be able to process all incoming requests in an appropriate response time due to a lack of resources, we might lose our customers, reputation and also revenue; however, because of the EC2 scalability feature, adding new EC2 instances to the servers' farm quickly in direct response to load changes is possible. After that, when the load on the systems have been degraded to its normal value, it is also possible to terminate extra instances and continue with our original two desired EC2 instances for avoiding extra costs of over provisioning resources. During the aforementioned example, not only the significance of having EC2 scalability feature for handling unforeseen load changes on systems is shown, but it also provides a recognizable example of cost efficiency.

Adding or removing Amazon EC2 instances can be done manually where customers are able to add and remove EC2 instances directly from portal, CLI, APIs or even available SDKs or automatically with no live contribution

of customers. Manual scaling is a very direct way to manage the number of in service EC2 instances and there is no need for other AWS service's contributions for accomplishing scaling task. On the other hand with automatic scaling, the process of adding or removing EC2 instances would be done through taking advantage of other AWS services like Amazon Auto Scaling service with correlation of Amazon CloudWatch service which makes is more advanced approach.

1.1 Amazon Auto Scaling Group

Amazon Auto Scaling group(ASG) is a collection of EC2 instances and it is the core component of Amazon EC2 Auto Scaling service. In addition to automatic scaling, Auto Scaling group also provides some other features such as unhealthy EC2 instances replacement or load balancing based upon availability zones [**4**].

Any Auto Scaling group has some attributes which need to be defined during the process of the creation. One of these attributes is the size or the capacity boundaries of the group. This attribute defines the upper and lower boundaries of the Auto Scaling group. In other words, this represents the maximum and the minimum number of allowed in service EC2 instances within a group. Amazon EC2 Auto Scaling ensures the clients that the number of in service EC2 instances within any Auto Scaling group never goes under the defined minimum and also would never go above the predefined maximum boundaries. It is worth noting that this agreement has some excep-

tions as when weighed instances are being used within Auto Scaling groups [**4**]. Another attribute for any Auto Scaling group which should be defined during the creation is the desired capacity of in service EC2 instances within the Auto Scaling group, this number should sit between the defined minimum and the maximum capacity boundaries.

Scaling the size of any Auto Scaling groups is the process of adjusting the desired capacity of the group in accordance with variable changes on the loads. This process can be either adding more EC2 instances to the Auto Scaling group which is referred as Scale out process or the process of removing and terminating extra EC2 instances from the Auto Scaling group which is referred to as Scale in process.

1.2 Scaling Auto Scaling Groups

Scaling the size of any Auto Scaling group can be done via three general ways as follows: manual scaling, scheduled scaling and dynamic scaling.

1.2.1 Manual Scaling:

Any changes on the amount of the minimum, maximum or the desired capacity of any Auto Scaling group can lead to launch or termination of EC2 instances. In addition to the aforementioned generic way, the other approach for manually scaling the size of any Auto Scaling group is to attach or detach in service EC2 instances to or from the Auto Scaling group if they pass all the required prerequisites.

The mutual characteristic shared among both aforementioned approaches for scaling the size of the Auto Scaling group manually is that they can be done independently, with no contribution of other AWS services, also both will be performed in an active manner, upon direct commands of the customers.

1.2.2 Scheduled Scaling:

Scheduled scaling is a time based approach for scaling the number of in service EC2 instances within the Auto Scaling group based on some predefined conditions. These conditions are generally a tuple consisits of exact time and date of the changes with their size of desired changes. The main difference between scheduled scaling and manual scaling is where scheduled scaling will be done passively, there is no need for presence of the client during the scaling process.

1.2.3 Dynamic Scaling:

In contrast with both aforementioned approaches, the most advanced way to scaling the number of in service EC2 instances within any Auto Scaling group is to adjust them adaptive to live performance changes. In other words, if there are any needs for more EC2 instances at a given moment, the Auto Scaling service has the ability to initiate the Scale out process to add extra EC2 instances to the group, and if there be any number of idle EC2 instances within the Auto Scaling group, also it has the ability to

trigger Scale in process in order to terminate idle EC2 instances to stop extra charges. The aim of having dynamic scaling is to provide scaling based on the live changes on systems' performance vectors in the absence of administrators (customers). This is the key feature that makes dynamic scaling very popular among AWS Auto Scaling service users.

The deployment of dynamic scaling on any Auto Scaling groups would be done by defining some scaling policies which are some predefined rules use to instruct the Auto Scaling service on how to manage and adjust the number of in service EC2 instances within the Auto Scaling group. In other words, for having Amazon Auto Scaling service being able to perform dynamic scaling or scaling based on the demand changes automatically, Scaling policies are required to be defined and configured; otherwise, a fix number of in service EC2 instances would be kept as the configured desired capacity with no live responses to real time performance vectors changes.

As it is mentioned earlier, dynamic scaling is able to adjust the number of in service EC2 instances based on the performance changes; hence, a live performance monitoring service is required in order to track changes on one or more desired systems' performance metrics, this is where Amazon CloudWatch service comes to the scene.

1.3 Amazon CloudWatch Service

Amazon CloudWatch service is AWS' monitoring service responsible for collecting logs and events from other Amazon resources and services, one of

them which sends live (near to real time) logs and events to Amazon CloudWatch service is amazon EC2. In fact, Amazon CloudWatch service acts like an application which not only stores logs and any metric changes of EC2 instances within its repository, but also would be able to call and trigger other Amazon Services as required in response to those metrics' changes. Metric is the desired computing resource which projects the performance vector of any EC2 instances. Some of the available performance metrics for EC2 instances to choose between are as follows [**5**]:

- CPUUtilization
- DiskReadOps
- DiskWriteOps NetworkIn
- NetworkOut
- NetworkPacketsIn
- NetworkPacketsOut

Through utilization of Amazon CloudWatch service and by defining the desired metrics to be monitored, Amazon CloudWatch service is able to track live changes over the desired performance vectors for any number of EC2 instances. If those changes were above or below any of the predefined thresholds, Amazon CloudWatch service is also able to call for other AWS services for giving the best responses in accordance to the changes. It is worth noting

that by default, Amazon CloudWatch service gathers information from each EC2 instance every five minutes with no extra cost; however, by enabling detailed monitoring this interval can be set to less amounts such as every minutes. Detailed monitoring has extra costs[**6**].

1.4 Dynamic Scaling Policies

In order to scale any Auto Scaling group based on live changes on their CloudWatch metrics (dynamic scaling), defining Auto Scaling policies is required. In fact, Auto Scaling policies bound desired responses to each defined CloudWatch alarms. Auto Scaling policies are as follows: target tracking policies, step scaling policies and simple scaling policies [**7**].

1.4.1 Target Tracking Scaling Policy

With this kind of scaling policy, the desired metric and its value would be defined by customers. Then, Amazon EC2 Auto Scaling service creates and manages the corresponding CloudWatch alarms based on those selections. The Auto-Created CloudWatch alarms are responsible for triggering the Auto Scaling processes when any CloudWatch alarm breaches (when a CloudWatch alarm breaches, the CloudWatch service moves into in-alarm state, notifying the Auto Scaling service to perform the desired actions).

Amazon Auto Scaling service with target tracking policy in place tries to keep the value of the selected CloudWatch metric as close as possible to its predefined desired number with launch or termination of EC2 instances

accordingly very similar the way that thermostat maintain the temperature of a home. Please note that Auto Scaling service still follows the predefined Auto Scaling group's capacity boundaries to not passing predefined maximum capacity of the Auto Scaling group by launching extra EC2 instances or also to not pass the predefined minimum capacity by terminating extra EC2 instances from the Auto Scaling group.

1.4.2 Step Scaling Policy

With step scaling, client has more granular control over the behavior of the Auto Scaling service by being able to define various steps with the desired actions based on the selected CloudWatch metrics. In fact, each step is a tuple consists of a threshold for the selected CloudWatch metric with its exact corresponding action.

1.4.3 Simple Scaling Policy

Simple scaling policy is very similar to step scaling policy but instead of being able to define multiple Steps and actions; here, customers can select just one step. Please note simple scaling policy has some built-in limitations which need to be considered, one of them is as follows: while new EC2 instances are being launched as a response to any Scale out alarm and they are still in their pending time (time taken by EC2 instances before getting ready to enter into InService state, where they start servicing like another Auto

Scaling group members), Auto Scaling service would not respond back to any new CloudWatch alarms, like its service is being suspended temporary, which means slower convergence in comparison with the two other approaches [**7**].

Although, dynamic scaling provides reliability and availability for any Auto Scaling groups, it is also intended to provide cost efficiency to its customers with preventing excess resource allocation, dynamic scaling might lead to some hidden costs for its customers in unpredictable load changing environments which made for the basis of my thesis framework, but first, let's have a look at various charging models available for EC2 instances.

1.5 AWS Billing Methods

Amazon Web Services (AWS) charges EC2 users either on per hour or per second, dependent upon type, size, operating system, and the AWS region where the instances are launched. In an hourly basis, instances are billed for following 60 minutes even a portion of that paid 60 minutes get used by them. Let's look at some examples coming directly from the Amazon documents:

- "Instance for 30 minutes and then terminate the instance, IT is billed for one instance-hour".

- "One instance for 10 minutes, stop the instance, and then start the instance again, it is billed for two instance-hours".

- "Two EC2 instances of the same type for 30 minutes each, it is billed for two instance-hours" [**8**].

The last example, elaborates a very specific behavior of Auto Scaling service which can lead to extra costs for customers in some scenarios, for instance where an already paid EC2 instance get removed from the Auto Scaling group in response to a Scale in CloudWatch alarm, then within the same billing period one or more new EC2 instances get launched based on new scaling out alarms.

1.6 My Contributions

After a full explanation of how Auto Scaling groups can be scaled in or out automatically via dynamic scaling policies, the default behavior of dynamic scaling over a simulated case study will be examined and it will be shown how it can lead to extra costs pushed onto customers in similar ways.

After acknowledging the possibility of dynamic scaling policies hidden costs due to unpredictable load changes on Auto Scaling groups when dynamic scaling is setup to manage its scaling processes, a couple of workarounds done by Amazon Web Services (AWS) or independent scholars for making dynamic scaling more cost efficient will be reviewed. Then some of my examined approaches within another case study alongside with full explanation of their restrictions will be studied and among them the best one which made the foundation of my final approach will be discussed. Finally, a practical

approach for minimizing hidden costs problem through the introduction of NoAuto Scaling group will be introduced, implemented and its results versus the outcomes of default Auto Scaling service behavior over a simulated scenario will be examined.

2 Background

Amazon Web Services(AWS)introduced Auto Scaling service in 2009 (three years after first introduction of Elastic Compute Cloud (EC2) instances in 2006) in order to make EC2 instances possible to respond quickly to demands changes [**9**]. One of the main reasons for having Auto Scaling service over Auto Scaling groups (ASG) is to manage the number of in service EC2 instances dynamically based on the real needs for allocated resources. In this regard, terminating unwanted EC2 instances to prevent resources over provisioning which causes extra costs for clients is also included into Auto Scaling service. For being able to enhance Auto Scaling service behavior over any Auto Scaing groups, having a solid understanding of EC2 instances life-cycle (from the moment a new EC2 instance is launched to the moment of its termination) is inevitable.

2.1 EC2 Instances Life-cycle Components

The Life-cycle of any EC2 instances within any Auto Scaling groups (ASG) would be like Figure 1 [**10**] which differs from EC2 instances outside of the Auto Scaling groups. An EC2 instance life-cycle starts when the EC2 instance is launched, either by manual scaling, scheduled scaling or dynamic scaling in response to any Scale out requests, and it finishes with instance termination: when the instance would be taken out of the Auto Scaling group

either by terminating process or EC2 detachment process.

EC2 instances life-cycle consists of various states which any EC2 instance can experience during its existence, some of them are optional like life-cycle hooks or Standby states and some of them are mandatory like pending or InService states.

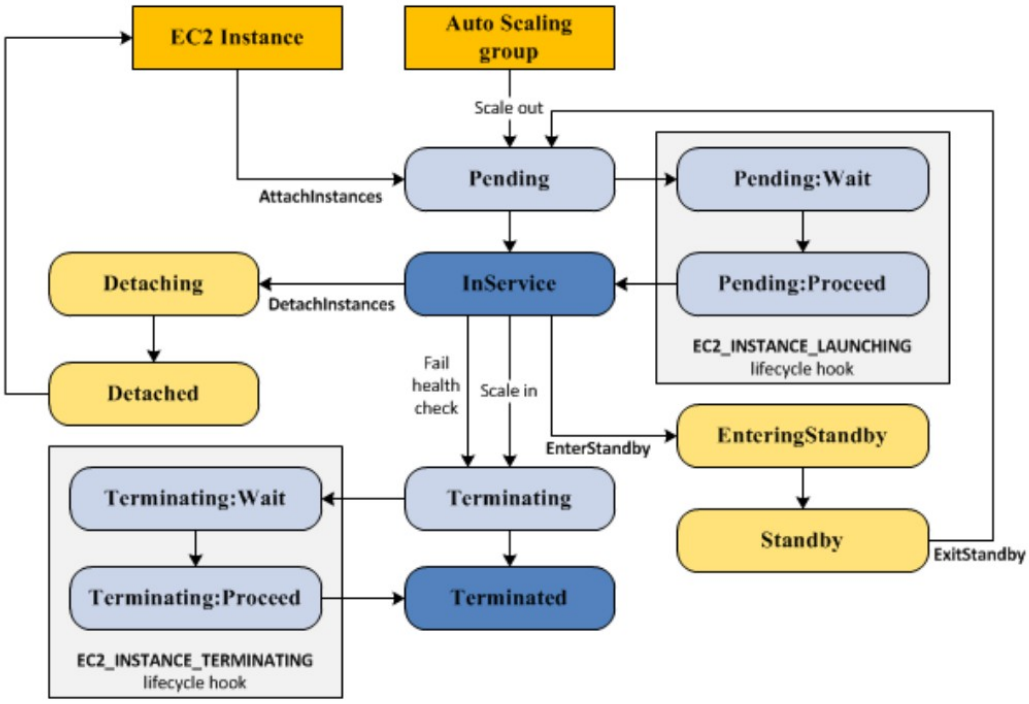

Figure 1: EC2 Auto Scaling Life-cycle

For a better understanding of what happens to any EC2 instances within any Auto Scaling groups from the beginning to the moment of termination, let's take a closer look at the EC2 life-cycle's components.

2.1.1 Life-cycle Entrance

Increasing the size of any Auto Scaling group (Scale out) can be done in various ways including manual scaling, scheduled scaling and dynamic scaling.

2.1.1.1 Manual scaling: This is a very direct way to increase (Scale out) or decrease (Scale in) the number of in service EC2 instances within any Auto Scaling groups where changes on Auto Scaling group's boundaries or its number of desired capacity of in service EC2 instances can lead to the launch or termination of EC2 instances consequently. These changes can be done with direct command of the customers (administrators) via AWS management Portal, CLI, APIs or even SDKs. In addition to the aforementioned way, another approach which can consider in the same category as manual scaling is to attach already running EC2 instances to the Auto Scaling group since this approach would also require a direct live contribution of customers to be done. Prior to be able to attach any running EC2 instances to any Auto Scaling groups, some criteria need to be checked as follows [**11**]:

- EC2 instances should be in running state.
- Amazon Machine Image (AMI) of the desired instances should be still available.
- Desired EC2 instances should not be a member of other Auto Scaling groups.

- Desired EC2 instances must be launched in one of the supported availability zones by the Auto Scaling group.

By attaching new EC2 instances to an Auto Scaling group, the desired capacity of the group increases accordingly, If this number exceeds the predefined maximum capacity size, then the request will be dropped [**11**].

2.1.1.2 Scheduled scaling: The process of launching new EC2 instances can be done based on a set of predefined dates and time. Schedule scaling is a tuple consists of an exact time and date as its first variable in addition to the amount of changes over the number of in service EC2 instances as its second variable.

2.1.1.3 Dynamic scaling: With taking advantages of dynamic scaling, customers are not required to launch or terminate EC2 instances directly. Instead, Auto Scaling Service is responsible for performing both aforementioned tasks. For dynamic scaling being able to perform its tasks, having dynamic scaling policies are required; otherwise, Auto Scaling service will not respond back to any CloudWatch alarms requesting Scale out or Scale in processes. In other words, having an Auto Scaling group with dynamic scaling while there are no scaling policies defined, is like to seize the ability of automatic scaling from Auto scaling Service over that particular Auto Scaling group.

2.1.2 Pending State

Regardless of the way uses to scale an Auto Scaling group out, prior to put any EC2 instances into the InService state, where they start servicing like other InSerivce EC2 instances, some preparation processes are required. Some processes including installing operating systems, patching or updating them, installation and configuration of required applications and so forth. These processes are time consuming ones; hence, these kind of tasks would be completed in the Pending state. It is worth noting that although customers will be charged for their EC2 instances in Pending state, it assures EC2 instances will not start servicing while they are not completely ready for it yet. In addition to Pending time, clients also might setup Warm-Up value for their EC2 instances, which is extra time needed for newly launched instances prior to become an Auto Scaling group's member and start servicing [**12**]. After the Pending state, there are two paths available for EC2 instances to proceed as follows: moving to either InService state or life-cycle hook.

2.1.3 InService State

InService state is where EC2 instances start servicing as a member of an Auto Scaling group and their performance's parameters start being monitored by Amazon CloudWatch Service.

2.1.4 Life-cycle Hook

Life-cycle hook is an optional step, providing more control over EC2 instances before having them added into the Auto Scaling groups or prior to terminating EC2 instances and removing them from the groups. Amazon Auto Scaling life-cycle hook introduces two new states during Scale out process as Pending:Wait and Pending:Proceed states, also two new states for Scale in process which are the Terminating:Wait and Terminating:Proceed states as it is shown in Figure 2 [**13**]. In general, life-cycle hook is known as an extra option for performing desired changes or taking required logs and backup before adding or removing instances. This is worth noting again EC2 instance can wait in life-cycle hooks for 48 hours without customers get billed for them. EC2 instances can stay in this step for 48 hours at maximum and after that time they automatically continue with moving into Inservice state or be terminated. Also instances within the life-cycle hook are not billed; however, please note that some AWS resources, such as Amazon EBS volumes and Elastic IP Addresses incur charges regardless of the instance state [**13**].

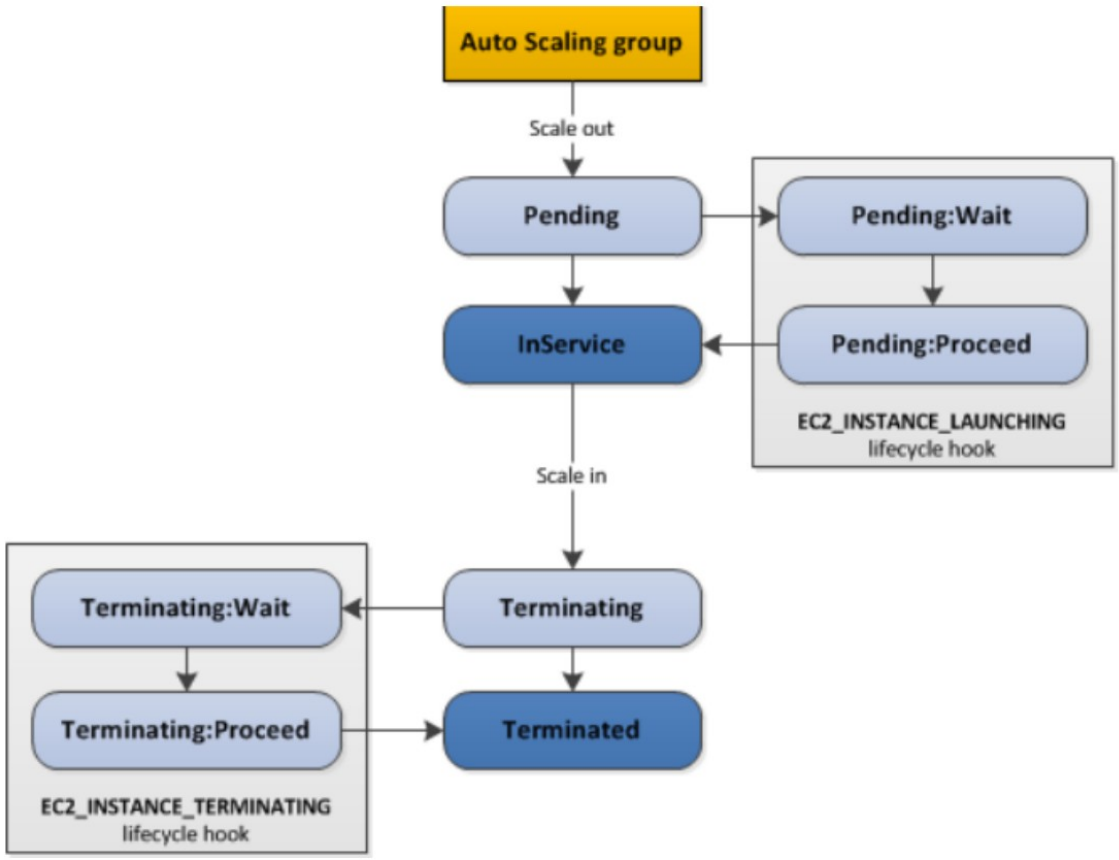

Figure 2: EC2 Life-cycle Hook Diagram

2.1.5 Exiting from Inservice State

Four events can change the state of any EC2 instances from InService to others as follows: detaching instances, terminating them, failing periodic health checks and going to Standby state.

2.1.5.1 Detaching Instances: Detaching process on any EC2 instances from the Auto Scaling group moves them outside of the group while they are still up and running to be managed independently like other stand alone EC2 instances. When we choose to detach any EC2 instances, we can decrease the number of desired capacity size of the group to prevent launching any new EC2 instances as the replacement by Auto Scaling service[**14**].

2.1.5.2 Termination phase: Another way to exit from InService state is to move EC2 instances to the Termination phase which would be done either via Scale in process or with direct commands of the clients. As it is mentioned earlier, designated EC2 instances can also get moved into the Termination life-cycle hook optionally in order to perform more custom actions on, like gathering logs and data from, or simply stopping and removing them from the Auto Scaling group.

2.1.5.3 Failing Periodic Health Check: Auto Scaling service monitors and evaluates EC2 instances health status within the Auto Scaling groups periodically and if it detects any defected EC2 instances, it will replace those instances with healthy new EC2 instances.

2.1.5.4 Moving into the Standby State: The last way to exit from InService state is to move EC2 instances to the Standby state. Since moving into Standby state and exiting from that would be a part of my final ap-

proach, I would discuss Standby state with more details is a separate section as follows.

2.1.6 Standby State

Standby state is designed for providing some maintenance over in service EC2 instances if required. Some maintenance like system updates, software changes or doing some troubleshooting while those EC2 instances are still considered as an Auto Scaling group's member but not in the InService state. By default, putting any allowed number of in service EC2 instances into Standby state would not lead to launch any new EC2 instances since the number of desired capacity of the Auto Scaling group would be decremented by the number of Standby EC2 instances. This is designed to prevent the Launch of any new EC2 instances as the replacement of Standby EC2 instances within the Auto Scaling group. Please note, customers are billed for their Standby EC2 instances, hence this default behavior of Auto Scaling service can save clients from extra charges. After moving any Standby instances back into InService state, the desired capacity would be incremented by the number of returned instances. The aforementioned behavior can be modified in a way to not decrease or increase the amount of desired capacity size while moving EC2 instance to or from Standby mode [**15**].

Now, let's have a closer look at how Standby state works, as it is shown in Figure 3 [**15**], entering into Standby state from Inservice state is a very straightforward process with absence of any intermediary steps; however,

moving any Standby EC2 instances to the Inservice state, first entering into Pending state is required.

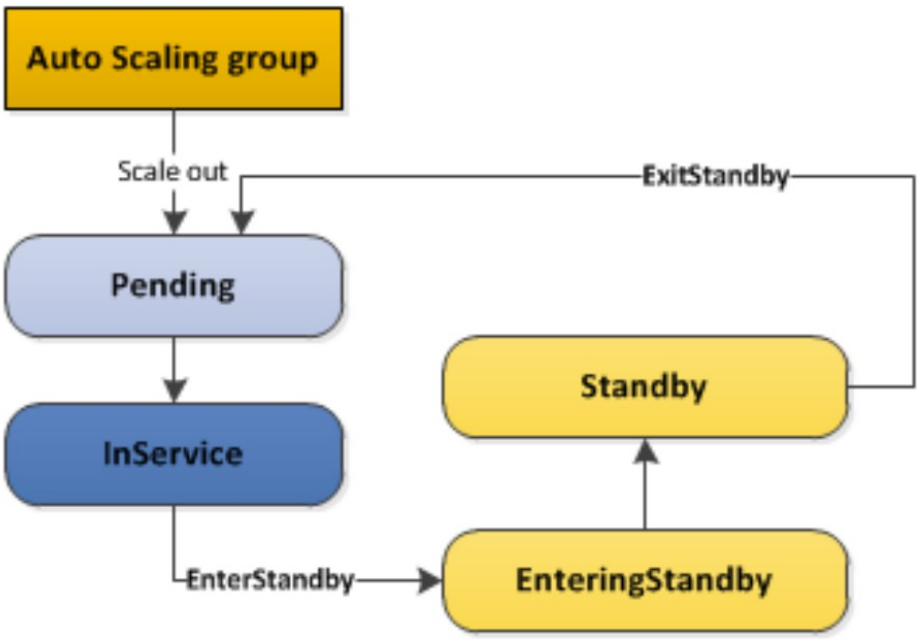

Figure 3: Life-cycle of Standby EC2 Instances

.

2.2 Auto Scaling Policies:

In order to be able to scale the size of any Auto Scaling group dynamically based on near to real time changes on the load of the desired performance metrics, some predefined rules are required, these rules are referred to as scal-

ing policies which dictate the desired responses to the CloudWatch alarms. A scaling policy instructs Amazon EC2 Auto Scaling service to track a specified CloudWatch metric as its monitoring parameter, also it defines what action should be taken as the desired response when that CloudWatch metric exceeds its predefined thresholds or technically goes into in-alarm state. Auto Scaling policies are as follows [**16**]:

2.2.1 Target Tracking Policy:

This is the recommended way by Amazon Web Services for most of clients since it is very straightforward and easy to maintenance, customer defines a desired scaling metric with its target value, then Auto Scaling service takes care of the rest of works with trying to keep the value of selected performance metric as close as possible to the desired target value. This process consists of creation of corresponding CloudWatch alarms and management of required Scale in or out processes.

2.2.2 Step Scaling Policy:

This approach, which provides more granular control over the Auto Scaling groups, requires customers to define their CloudWatch alarms themselves, also defining the corresponding actions in the form of various steps. Hence, customers can dictate their willing over the behavior of their Auto Scaling groups in a more detailed manner; however, it requires more works to have the scaling policies setup and tuned.

2.2.3 Simple Scaling Policy:

This one is very similar to step scaling policy; however, with simple scaling policy customers can define just one step or action instead of having multiple steps like what step scaling is made for.

2.3 Billing Plans

Amazon EC2 instances are billed either in a hourly basis or for some Linux instances based on used seconds. In my thesis, I considered the hourly based charging plan which covers more range of the scenarios. Based on hourly billing plan, each EC2 instance is billed for the next 60 minutes repetitively once it is launched until it gets terminated completely or enters into the life-cycle hooks as discussed earlier [**17**].

2.4 Demonstration of the Problem

In the following section, within a simulated scenario, one situation that dynamic scaling would cause extra charges will be demonstrated. In this thesis, these kind of costs are referred to as hidden costs. Let's assume we have an Auto Scaling group with the following capacity size settings:

- Minimum Capacity: 1
- Desired Capacity: 4
- Maximum Capacity: 10

Also, let's assume we have the following step scaling policy as desired dynamic scaling policy. This policy tracks the average CPU usage of all in service EC2 instances within this Auto Scaling group and has the following reactions in response to any breached CloudWatch alarm once the average CPU usage exceeds above or below the defined thresholds.

- CPU usage 50%: Number of desired in service instances: 4
- CPU usage 35% : Scale in 2 EC2 instances.
- CPU usage 75% : Scale out 2 EC2 instances.

For simplicity, also let's assume all four initial EC2 instances are launched at the exact same time; thereby, their billing time also would be at the exact same time since in this thesis, hourly billing basis would be studied.
Let's imagine after a short time during the first billing hour, the average CPU usage of in service EC2 instances within this Auto Scaling group drops to 35%, as expected: a Scale in alarm breaches and based on defined scaling policies, two EC2 instances would be selected for getting terminated. In this regard, two EC2 instances from the Auto Scaling group enter into the termination phase and after a short time the number of in service instances will be degraded to two instances.
So far, we paid for 4 instances; however, we have two in service EC2 instances. let's continue with assuming the average CPU usage boosts up again and reaches to 50%; consequently, two new EC2 instances as a response to the breached Scale out alarm will be launched and here is where hidden costs

are shown up as demonstrated in Table 1 (Auto Scaling Default Behavior), although the number of in service EC2 instances in this Auto Scaling group is still four, it is billed for six EC2 instances so far within the first billing hour.

We can even go further and imagine based on some loads on the group, the average CPU usage increases to 75% and in respond to this change, two more new EC2 instances launch (Scale out).

Again, if the average CPU usage drops to 50%, two EC2 instances would be terminated in response to an emerged Scale in alarm aligning the size of desired capacity to four instances one more time.

- Number of in service EC2 instances at the end of very first billing hour: 4
- Number of billed EC2 instances during that period: 8

Table 1: Auto Scaling Default Behavior

CPU usage	In service instances	Paid instances	Description
50%	4	4	Desired numbers as 4 when the average CPU usage is at 50%
35%	2	4	Scale in alarm: Terminating two instances
50%	4	6	The average CPU usage went back to Normal (50%); therefore, a new scaling out alarm breaches: adding two more instances to the Auto Scaling group.
75%	6	8	The average CPU usage boosted, scaling out process adds two more instances to the Auto Scaling group, the total number of paid instances would be 8
50%	4	8	The average CPU usage went back to Normal (50%); Therefore, another Scale in alarm breaches: Terminating two EC2 instances from the Auto Scaling group.

As it is demonstrated above, fluctuations on the selected CloudWatch metrics (in our example: The average CPU usage) can cause extra costs depending on the number of launched or terminated EC2 instances. The aim of my work in this thesis is to find a way to minimize the costs of having dynamic scaling in place for managing the size of Auto Scaling groups based on the demands. In this regard, we need to find a way to maximize the amount of time when already paid EC2 instances can be accessible to any time closer to their renewal time based on the hourly paid basis.

2.5 Predictive Scaling and Hybrid Auto Scaling

lets start with a short review on some works have already been done to make Auto Scaling groups more cost efficient. In 2018, Amazon Web Services introduced a new feature named predictive scaling available for its clients. AWS predictive scaling uses trained Machine Learning models to predict future load changes in daily or weekly patterns. As AWS says once the initial set of predictions were ready, it can forecast the loads on EC2 instances for next following two days. Predictive scaling supports clients' dynamic scaling target tracking policies and in this case predictive scaling sets the minimum capacity and target tracking adjusts the desired capacity dynamically [**9**]. Although, in case of unpredictable load vector changes, still the chance of facing hidden costs is high since predictive scaling is designed to provide capacity before the load changes and not after that; However, since it can predict future load decreases as well, it can reduce the chance of resource over provisioning with a better adjustment of dynamic scaling plans.

Another approach which has common ideas with my work in this thesis is a hybrid auto-scaling technique, a novel approach proposed by Biswas et al, 2017 [**18**]. In their work they proposed how an intermediary firm can act like a broker between AWS and clients by providing idle already paid resources to another client while still following all SLA (Service Level Agreements) policies. In other words, with their approach, scaling in process for a particular client would not terminate EC2 instances in fact, it just stop

that client from being billed for that specific EC2 instance, on the other hand, that specific EC2 instance can now be assigned to another client who needs more resources. With this idea, they are also trying to make most profit from the already paid EC2 instances very similar to my goal in this thesis.

2.6 Some Possible Approaches

After the review which is done at last section over both predictive scaling and aslo Hyberid auto-scaling technique (two workaround already have been done by AWS and also some scholars to minimizing the costs of Auto Scaling services), lets continue with my first approach for resolving hidden cost's problem and also acknowledging its restriction within another simulated scenario. After that, my second approach will be proposed and why it is not practical at the moment will be discussed. The aim of bringing both unsuccessful approaches here is because all lesson learned from both approaches helped me to draw up the fundamentals of my final proposed approach.

2.6.1 Suspending Scale in Service

With Auto Scaling policies in place, both scaling out (the process of adding new EC2 instances to the Auto Scaling group) and scaling in (Terminating EC2 instances and removing them from the Auto Scaling group) processes are triggered as soon as their corresponding CloudWatch alarms breaches. In order to force an Auto Scaling group to keep its EC2 instances for the

entire current billing hour, while dynamic scaling policies are in place, we need to find a way to change the default behavior of Scale in process; instead of terminating EC2 instances once any Scale in alarm breaches, postponing EC2 instances termination process to any time closer to their renewal time might be a solution. To do so, Terminate micro service we be suspended over any Auto Scaling groups.

Amazon Auto Scaling service itself consists of some micro services as follows:

- Launch
- Terminate
- Health Check
- Replace Unhealthy
- AZ Rebalance (AZ stands for Availability Zones)
- Alarm Notification
- Scheduled Actions
- Add To Load Balancer
- Instance Refresh

Each one of the aforementioned services is responsible for handling a specific task over any Auto Scaling groups. For instance, Launch micro service handles Scale out processes while Terminate micro service handles Scale in

processes [**19**]. This is possible to suspend any number of these micro services as needed and still take advantage of other micro services on any Auto Scaling groups. In addition, these changes are domestic for selected Auto Scaling group and will not impact other Auto Scaling groups.

2.6.1.1 Evaluation of the Approach During the First Billing Hour

As it is mentioned earlier, one potential solution for changing the default behavior of Auto Scaling service to not terminate EC2 instances in response to Scale in alarms is to suspend the Terminate micro service over an Auto Scaling group, therefore, if any EC2 instances are launched and are in InServive state, Auto Scaling service would not terminate them automatically, guarantees the maximum usage of any paid EC2 instances . With this approach, Auto Scaling service handles only Scaling out processes. Table 2 (Auto Scaling Behavior with Terminate Micros Service Suspended) evaluates the results of this approach over our original scenario:

Table 2: Auto Scaling Behavior with Terminate Micros Service Suspended

CPU usage	in service instances	paid instances	Description
50%	4	4	Desired numbers as 4 when the average CPU usage is at 50%
35%	4	4	No Scale in Progress will be triggered since Terminate micro service is suspended!
50%	4	4	The average CPU usage went back to Normal (50%), no scaling out process needed. The number of paid instance and in service instances are still the same which means we could save money here
75%	6	6	The average CPU usage boosted, scale out policy adds two more instances to our Auto Scaling group, the total number of paid instances would be 6
50%	6	6	The average CPU usage went back to Normal (50%), however no Scale in process would be triggered since its micro service is suspended.

As it is shown above, with suspending Terminate micro service over exampled Auto Scaling group, we could save money: instead of finishing with 8 paid instances and 4 in service instances at the end of the very first billing hour, we ended with 6 paid instances and 6 in service instances.

2.6.1.2 Evaluation of the Approach During Next Billing Hours

In both aforementioned scenarios, and seeing changes with the very first billing period, it is shown that during that time, having the Scale in process suspended could lead to costs efficiency. However, we need to evaluate this approach and its outcomes over a longer period of time. This is done by

the following detailed case study which is another simulated scenario where both aforementioned approaches (with or without Terminate micro service suspension) are modeled for more than one hour billing period.Let's start with some definitions:

- Time-Stamp: a list consists of the pairs as follows: $[x_1, x2]$ where x_1 represents Time and x_2 represents the number of required EC2 instances. In fact, each pair within Time-Stamp list represents a CloudWatch alarm, it has Time when the CloudWatch alarm breaches and also its requested tuning number as x_2 value.

  In order to differentiate between Scale in and Scale out alarms , positive numbers indicate Scale out alarms while negative numbers indicate Scale in requests. For instance, [3:30, +2] indicates one scaling out alarm asking for the launch of two new EC2 instances at 3:30 AM. On the other hand, [16:10, -3] means there was a Scale in alarm at 4:10 PM. asking for termination of three EC2 instances within our Auto Scaling group.

- Event-List: a list consists of the pairs as follows: $[x_1, x_2]$ where x_1 represents Time and x_2 represents the number of in service EC2 instances. Event-List represents the actual outcomes of our approaches.

- Renewal-List: a list consists of the pairs as follows: $[x_1, x_2]$ where x_1 represents Time for next charge and x_2 represents the number of EC2 instances. Due to the fact that in my thesis I am assuming all EC2

instances are billed based on hourly basis, keeping the amount of minutes for x_1 will be sufficient. For instance, instead of keeping 2:30 AM. as X_1, we are going to keep 30 since it does not matter if time is 3:30 in the morning or 6:30 afternoon; as far as this instance is up and running it would be charged exactly 30 minutes after each hour repetitively.

- waiting-List: an Integer! for a better understanding why it is not required to have a list of pairs like other variables and keeping an Integer as the type of this variable is sufficient, let's assume waiting-List was a pair as $[x_1, x_2]$; x_1 same as all other variables was representing Time: when a Scale in alarm breaches and x_2 was representing the number of requested EC2 instances to scale in.

  Here is a very important default rule while applying multiple Amazon Auto Scaling policies over any Auto Scaling groups: Auto Scaling service tends to keep the largest possible number of InService EC2 instances (largest capacity) if there are multiple policies in place[**17**]. For example, if there are two Scale in alarms at the same time, one of which asks for three EC2 instances termination while the other one asks for terminating two EC2 instances, the Auto Scaling Service would select the later one as the winner and consequently discards the former request asking for termination of three instances.

  Same behavior is also valid for Scale out process, in competition between two concurrent scaling out alarms, the winner would be the one which leads to a larger Auto Scaling group capacity. For instance, if

one Scaling out alarm asks for launching two new EC2 instances while the other one is asking for launching three new EC2 instances, the Auto Scaling service would pick the later request and will add three new EC2 instances to the Auto Scaling group and the former request would be discarded [**17**]. Thereby; regardless of the number of Scale in alarms within a billing period, the selection of the winner request is a competitive task, the winner is the one that reduces the number of in service instances the least, or in other words, keeps the capacity of the group at the max.

In addition, keeping the time that Scaling in alarm breaches as the value of this variable is not critical since our goal is to postpone scaling in processes to the first arrived EC2 instance renewal time; hence, keeping an Integer as the number of nominated EC2 instances for being terminated is sufficient.

Now, let's create our ordered Time-Stamp for our example with assuming we have already created an Auto Scaling group as follows:

- Desired Capacity: 4
- Minimum Capacity Size: 1
- Maximum Capacity Size: 10

And average CPU usage is selected as the desired ClouldWatch metric for implemented dynamic scaling policy.

Amazon Auto Scaling service supports multiple Scale in policies in order to provide more control over designated EC2 instances to be terminated by Amazon Auto Scaling service. In other words, by customizing Scale in policies, customers are able to apply their will through selection of one of the following available policies[**20**].

- OldestLaunchTemplate
- OldestLaunchConfiguration
- ClosestToNextInstanceHour
- NewestInstance
- OldestInstance

In this example, let's assume the Scale in policy is set as OldestInstance policy. Hence, the oldest EC2 instances will be nominate for being removed from the Auto Scaling group by the Scale in process.Also for simplicity, let's assume all four EC2 instances have been launched at the exact same time: 8:30 AM.

In addition to all of the aforementioned assumptions, we also can assume Pending time for all Auto Scaling's processes is equal to zero. Although in real world this amount can be anything more than zero depending on various factors, assuming The Pending time equal to zero does not break the logic

of my works in this thesis. Let's list our current variables as follows:

- Time: 8:30 AM
- Time-Stamp: [[8:30,4]]
- Event-List: [[8:30 ,4]]

If everything remains the same in regards to our selected CloudWatch metric (The average CPU usage), the next billing time for these four instances would be at 9:30 AM. and after that 10:30 AM. and so on. Thereby, the first entry for our Renewal-List would be as the follows:

- Renewal-List: [[30,4]]

Now, let's assume at 10:45 AM., a Scale out alarm breaches, requesting for the addition of one new EC2 instances in response to a CloudWatch alarm.

- Time: 10:45 AM
- Time-Stamp: [[8:30,4], [10:45,1]]

Here, based on Auto Scaling service default behavior and in respond to the recently breached Scale out alarm, two new EC2 instances would get launched immediately; consequently, Event-List and Renewal-List will be changed as the follows:

- Event-List: [[8:30,4], [10:45,1]]
- Renewal-List: [[30,4], [45,1]]
- Number of in service instances: 5
- Number of paid instances: 5
- The most close renewal time: 11:30 AM. for four instances
- Time remained to the most close renewal time: 45 minutes

Now, let's assume this is 10:55 AM. and a Scale in alarm for terminating two EC2 instances breaches. We have two different approaches to compare:

1. With the default Auto Scaling service behavior.
2. While Terminate micro service is suspended.

Let's list our variables in these two cases:
With the Amazon Auto Scaling service's default approach, we would have the following variables:

- Time: 10:55 AM.
- Time-Stamp: [[8:30,4], [10:45,1], [10:55,-2]]
- Event-List: [[8:30,4], [10:45,1], [10:55,-2]]

The Auto Scaling service immediately selects two EC2 instances from our Auto Scaling group based on its termination policy to get terminated. Based on our assumption, oldest instances would be selected by Auto Scaling service in response to the Scale in alarm, in our case: EC2 instances which had been launched at 8:30 AM and renewing at 11:30 AM.
Please note that here in this example and at the current step, even having ClosestToNextInstanceHour as our selected termination policy would lead to the same result and same EC2 instances would be terminated in response to the current scale in alarm.

- Renewal-List: [[30,2], [45,1]]
- Number of in service instances: 3
- Number of paid instances: 5
- The most close renewal time: 11:30 AM for 2 EC2 instance (two are already terminated)
- Time remained to the most close renewal time: 35 minutes

Now, let's list what would be in action if we have Terminate micro service suspended over the Auto Scaling group.

- Time: 10:55 AM.
- Time-Stamp: [[8:30,4], [10:45,1], [10:55,-2]]

- Event-List: [[8:30,4], [10:45,1]]

Auto Scaling service would not respond back to this request. Hence, the last entry inside of Time-Stamp would not be moved into the event-List. Also for the fist time, our Waiting-List variable gets a value equal to two.

- waiting-List: 2
- Renewal-List: [[30,4], [45,1]]
- Number of in service instances: 5
- Number of paid instances: 5
- The most close renewal time: 11:30 AM. for 4 instances
- Time remained to the most close renewal time: 35 minutes

Let's continue our scenario with assuming this is 11:10 AM, another Scale out alarm breaches, this time for the addition of three new EC2 instances to our Auto Scaling group while the Terminate micro service is enabled, so the desired capacity of the group would be 6 instances.
Again we have two scenarios to compare as follows: when Amazon Auto Scaling service uses its default approaches:

- Time: 11:10 AM.
- Time-Stamp: [[8:30,4], [10:45,1], [10:55,-2], [11:10,3]]

- Event-List: [[8:30,3], [10:45,2], [10:55,-2], [11:10,3]]

Auto Scaling service immediately launches three EC2 instances.

- Renewal-List: [[10,3], [30,2], [45,1]] (Sorted)
- Number of in service instances: 6
- Number of paid instances: 8
- The most close renewal time: 11:30 AM for two instance
- Time remained to the most close renewal time: 20 minutes

Now, let's list outcomes of having Terminate micro service suspended.

- Time: 11:10 AM.
- Time-Stamp: [[8:30,4], [10:45,1], [10:55,-2], [11:10,1]]

Please note that prior to the time when the most recent Scale out alarm is breached, the number of in service instances within the Auto Scaling group was five, and the desired capacity number of in service instances based on the current scaling out alarm would be six. Hence, adding just one EC2 instance to the Auto Scaling group is sufficient.

- Event-List: [[8:30,4], [10:45,1],[11:10,1]]

Auto Scaling service immediately launches just one new EC2 instance.

- Renewal-List: [[10,1], [30,4], [45,1]] (Sorted)
- Number of in service instances: 6
- Number of paid instances: 6
- The most close renewal time: 11:30 AM for 4 instances
- Time remained to the most close renewal time: 20 minutes

So far and for one more time, Auto Scaling hidden costs are being demonstrated. Also, it is shown suspending Auto Scaling process can be a potential solution for preventing extra charges. However, in the following, the main weakness of this approach will be discussed.

Let's assume this is 11:20 AM and another Scale in alarm breaches, this time for termination of three EC2 instances. The followings are the outcomes of two approaches:

With Amazon Auto Scaling service's default approach, we would have the following results:

- Time: 11:20 AM.
- Time-Stamp: [[8:30,4], [10:45,1], [10:55,-2], [11:10,3], [11:20,-3]]
- Event-List: [[8:30,4], [10:45,1], [10:55,-2], [11:10,3], [11:20,-3]]

Auto Scaling service immediately selects two EC2 instances from Auto Scaling group based on its termination policy to get terminated. In our case, two of them would be the only remaining instance which had been launched at 8:30 AM, the other one though would be selected from the next elder EC2 instances, the ones which have been triggered at 10:45 AM.

- Renewal-List: [[10,3], [45,1] (Sorted)

- waiting-List: 0

- Number of in service instances: 4
- Number of paid instances: 8
- The most close renewal time :11:45 for 1 instance

- Time remained to the most close renewal time: 25 minutes

Now, let's list what would be in action if we suspend Terminate micro service over our Auto Scaling group.

- Time: 11:20 AM.

- Time-Stamp: [[8:30,4], [10:45,1], [10:55,-2], [11:10,1], [11:20,-3]] Auto Scaling service would not respond to this request.
- Event-List: [[8:30,4], [10:45,1], [11:10,3]]
- waiting-List: 2

Please note that the waiting-List's size, still would remain as two and will not alter to three! This is because we also prefer to have most in service instances in our Auto Scaling group, in fact here we are following Auto Scaling service behavior in the regards of Service Level Agreements.

- Renewal-List: [[10,1], [30,34], [45,1]] (Sorted)
- Number of in service instances: 6
- Number of paid instances: 6
- The most close renewal time: 11:30 AM for 4 instances
- Time remained to the most close renewal time: 10 minutes

Now, let's compare two Renewal-Lists together as follows:

- where Terminate micro service was enabled: [[45,1], [10,3]
- where Terminate micro service was suspended: [[10,1], [30,3], [45,2]]

Here is where the weakness of having Terminate micro service suspended pops up. If the load over the Auto Scaling group stays stable in regards to its CloudWatch metric, starting from next billing hour, we are billed just for four in service EC2 instances with default Auto Scaling approach while with suspending Terminate micro service this number would be six for all upcoming billing periods which is not good. The main issue with the proposed approach is where Amazon Auto Scaling service does not deal with Scale in processes as we wished for with performing two following steps:

1. To postpone EC2 instances termination time to anytime closer to their renewal time arrival, guarantees the maximum usage of EC2 instances.

2. To terminate existing Standby EC2 instances (the ones in waiting-List) at their very first Renewal time arrival.

Not being able to perform the second step is the actual weakness of this approach where with suspending Terminate micro service, Auto Scaling service would not be able to remove extra EC2 instances from the Auto Scaling group which leads to extra costs over customers due to the fact that Amazon Web Service will charge customers for their standby instances. During the remaining parts, first, a promising solution for enhancing Scale in process will be proposed, then its restriction will be discussed. At the end, the final solution with its implementations will be provided.

2.6.2 Escape-Gate

By having a closer look at Figure 4, it is obvious that the red line section is a Dead-End for any EC2 instances. By default, if any EC2 instances enter into the Terminating phase, there is no way to exit and returning back into the InService state.

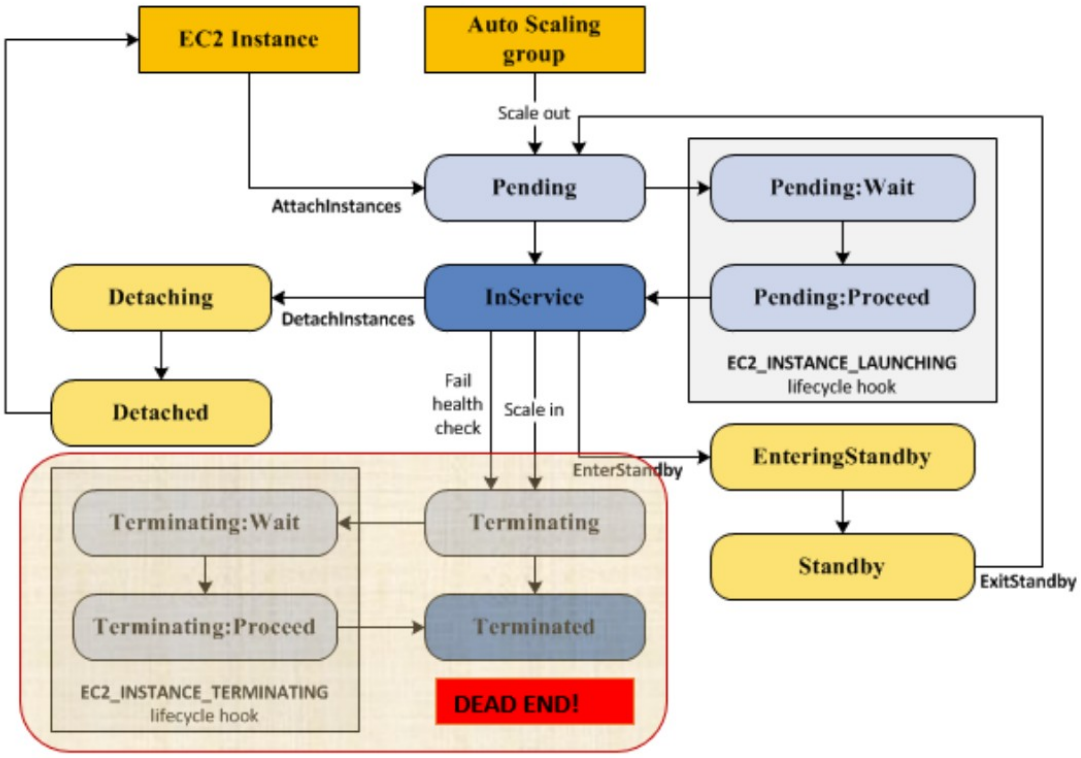

Figure 4: EC2 Life-cycle Dead-End

Perhaps, the Escape-Gate as it is shown in Figure 5 would be another potential solution for hidden cost problem where in services instances could

get moved into Terminating:Wait state in response to any Scale in alarms and waiting there even for 48 hours at maximum without even get billed for them [**13**]. Then, during that available 48 hours and in response to any brached Scale out alarm, instead of launching new EC2 instances, moving Terminantin:Wait instances to Panding state like what Escape-Gate does. Otherwise, at the arrival of the last renewal time (48th one), move designated EC2 instances from Terminating:Wait state into Terminating:Proceed state to complete termination process for avoiding extra costs. This would be the best solution of maximizing customers' benefit while using Auto Scaling groups with dynamic scaling in place. However, there are no available means to implement the Escape-Gate solution.

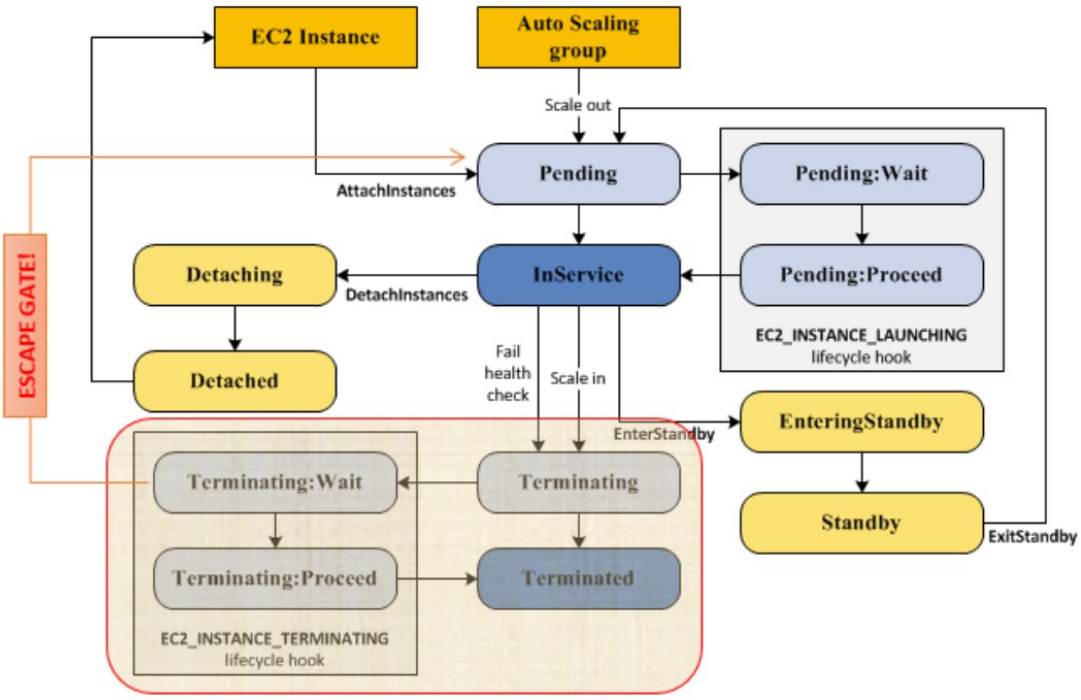

Figure 5: EC2 Life-cycle with Escape-Gate

2.6.3 Using Standby State instead of Terminating Wait State

As discussed in previous section, perhaps being able to implement any similar logic as Escape-Gate solution would be promising; hence, the following approach is proposed: instead of putting EC2 instances into Termination:Wait state when any Scale in alarm breaches, moving them into Standby state, then until their renewal time arrive, if any Scale out alarm breached, first move standby instances into InService state, otherwise terminating them at their renewal time arrival. However, prior to move forward with evaluating

the cons and pros of the proposed approach, lets check whether the suspension of Terminate and launch micro services over the Auto Scaling group is required or not and also what are the consequences of suspending Launch and Termination micro services.

The behavior of scaling in process is already educated, and it is shown that for having dynamic scaling configured over the Auto Scaling group, Terminate micro service needs to be suspended; otherwise, it will directly put any EC2 instances into the Terminating phase in response to any Scale in alarms. Please note that the above statement even can be altered to not disabling Terminate micro service while we are not going to use any built in Auto Scaling policies in the near future! This will be discussed at the end of my works while the last comprehensive solution will be proposed. By then, let's say: with having Auto Scaling policies in place, we need to suspend Terminate micro service over our Auto Scaling group.

The very exact same reason is also valid for Launch micro service while either one of the target tracking policy, step scaling policy or simple scaling policy is selected as the Auto Scaling group's scaling policies. By default, once any Scale out alarm breaches, Launch micro service, instead of working with already Standby instances first (the ones in the Waiting-list), trying to turn them back into the InService state, by default it launches new EC2 instances directly from its predefined Launch templates. Hence, already Standby EC2 instances would never get the chance to exit from Standby

state by Launch micro service default behavior. Although, this behavior of Launch micro service might encourage us to suspend it as well; however, by doing that, the function of making changes on the size of the Auto Scaling group would be disabled and two main issues emerge consequently:

1. would not be able to attach new EC2 instances to the Auto Scaling group anymore.
2. would not be able to move Standby instances to InService state anymore.

According to both aforementioned points, we need to have Launch micro service always running on our scaling groups; however, if we want to have scaling policies in place for performing dynamic scaling processes over the Auto Scaling groups, finding a way to change the default behavior of launch micro service is also inevitable. In this regard, keeping the size of desired capacity of running EC2 instances always equal to the size of Auto Scaling group's maximum boundary would be sufficient since as it is mentioned earlier, by design the number of running EC2 instances within any Auto Scaling groups will not go beyond of defined boundaries. Hence, if the current maximum boundary size be equal to the desired capacity of the group, any scale out alarm would be discarded. Finally, for performing scaling out processes, changing the maximum boundary of the Auto Scaling group adaptive to the desired changes would be necessary.

To put all in a shell, with having Auto Scaling policies in place, Terminate Micro service needs to be suspended while having Launch micro service running. For both Scale in and out processes, we can-not rely on default Auto Scaling group behaviors and we need to implementing and performing both manually. Manually here means by utilization of other Amazon Web Services like Amazon Lambda functions and Amazon Simple Notification Services (SNS)as it will be explained during remained parts. Although the aforementioned approach is practical, if we are going to perform both Scale in and Scale out processes manually, there is no need for having any Auto Scaling policies in place for scaling the size of the Auto Scaling groups dynamically.

2.7 Introduction of NoAuto Scaling Group

Let's define a NoAuto Scaling group attributes as follows:

- An Auto Scaling group with no need of dynamic scaling policies.
- An Auto Scaling group whose dynamic scaling processes would be handled by the help of Lambda functions, Amazon SNS service.
- An Auto Scaling group where the number of its desired capacity always will be kept equal to its defined maximum capacity size, and vice versa.

In fact, for minimizing Auto Scaling service hidden costs, I propose utilization of NoAuto Scaling groups with aforementioned characteristics, where

dynamic scaling can reduce hidden costs. The proposed approach is dynamic since it will track the real time performance changes by use of CloudWatch alarms, it is also Automatic since there is no need for any active contributions of customers, once it gets configured, it handles scaling processes over the NoAuto Scaling group automatically and finally, it can be more cost efficient since it maximized the use of already paid EC2 instances within the group.

2.8 Two Obstacles

In the following section, two obstacles for implementing AWS NoAuto Scaling groups will be discussed and solutions for them will be provided. The first issue describes how to track renewal time arrival of any Standby EC2 instances within the NoAuto Scaling groups and the second one explains how to terminate already Standby instances at their renewal time arrival.

2.8.1 Renewal Time Tracking Issue

Due to the fact that Scale in processes would not terminate EC2 instances any more and instead, we are going to put those termination designated EC2 instances into the Standby state on our NoAuto Scaling groups once a Scale in alarm breaches, tracking EC2 instances' Up-Time calculation should also be done manually in order to terminating them prior to their renewal time arrival. To do so, CloudWatch scheduled tasks in the forms of CloudWatch Cron expressions or CloudWatch Rate expressions can be used. In fact, with Cron expression or Rate expression, the creation of Self-Triggered Cloud-

Watch alarms is doable [**21**]. The goal is to define an acceptable missing time (the number of minutes that we can ignore from each 60 minutes paid period). For instance having missing time equal to five means that we want to use our EC2 instances at least for 55 minutes and being terminated at any time less than five minutes is acceptable for us.

With having the definition of acceptable missing time, we use this amount as the desired scheduled interval time for running a Lambda function which calculates each EC2 instances UP-Time. Then, if the remaining time to the instance renewal time (which simply can be calculated by having the instance UP-Time since the renewal interval is each 60 minutes) was less than the accepted missing time and the instance was in Standby mode, we can terminate the instance from our Auto Scaling group in order to avoiding extra costs with having idle EC2 instances transferred to the next billing hour.

2.8.2 Standby EC2 Instances Termination Issue

The process of Terminating Standby EC2 instances directly within any NoAuto Scaling groups, or even Auto Scaling groups is not a direct approach which can be done directly via commands. Generally because EC2 Auto Scaling Service does not perform health checks over Standby EC2 instances, Auto Scaling Service would not provide any executable command over Standby EC2 instances and for being able to terminate Standby EC2 instances at their renewal time arrival, performing the following steps is required:

1. First, move Standby EC2 instances into the InService state.

2. Then, detaching them from our NoAuto Scaling group.

3. Finally, terminating them out of the Auto Scaling group to avoid any extra costs.

2.9 Lambda Functions

In order to deploy a NoAuto Scaling group, AWS Lambda functions are the main components where all the logic will be implemented by them. In fact, with having Lambda functions joined with our CloudWatch alarms, the desired responses to any breached CloudWatch alarms would be given via the corresponding Lambda Functions. In other words, Amazon CloudWatch events reflect changes on their status, once they entered into the in Alarm state, this state change invokes AWS Lambda functions to take the desired actions over. AWS Lambda lets clients run code without provisioning or managing servers. they pay only for the compute time they consume [**22**].

2.10 SNS Topics

In order to invoke Lambda function by CloudWatch alarms, implementation of Amazon Simple Notification Service or in short SNS is required. In fact, whenever a CloudWatch alarm breaches, this status changes can be delivered to AWS Lambda service by SNS topics [**23**].

3 Algorithm of the Solution with its Implementation Source Code

The following section consists of:

1. Providing the flowchart of a NoAuto Scaling group life-cycle.
2. Providing detailed algorithm of a NoAuto Scaling group.
3. Comparing Auto scaling group with NoAuto Scaling group behavior on a simulated scenario
4. Pseudo-code of required Lambda functions
5. Implemented code in Python with using Boto3: Amazon Web Service's SDK in Python)

3.1 NoAuto Scaling Group Life-cycle

Figure 6 is the flowchart of how EC2 instances life-cycle inside of a NoSuto Scaling group looks like.

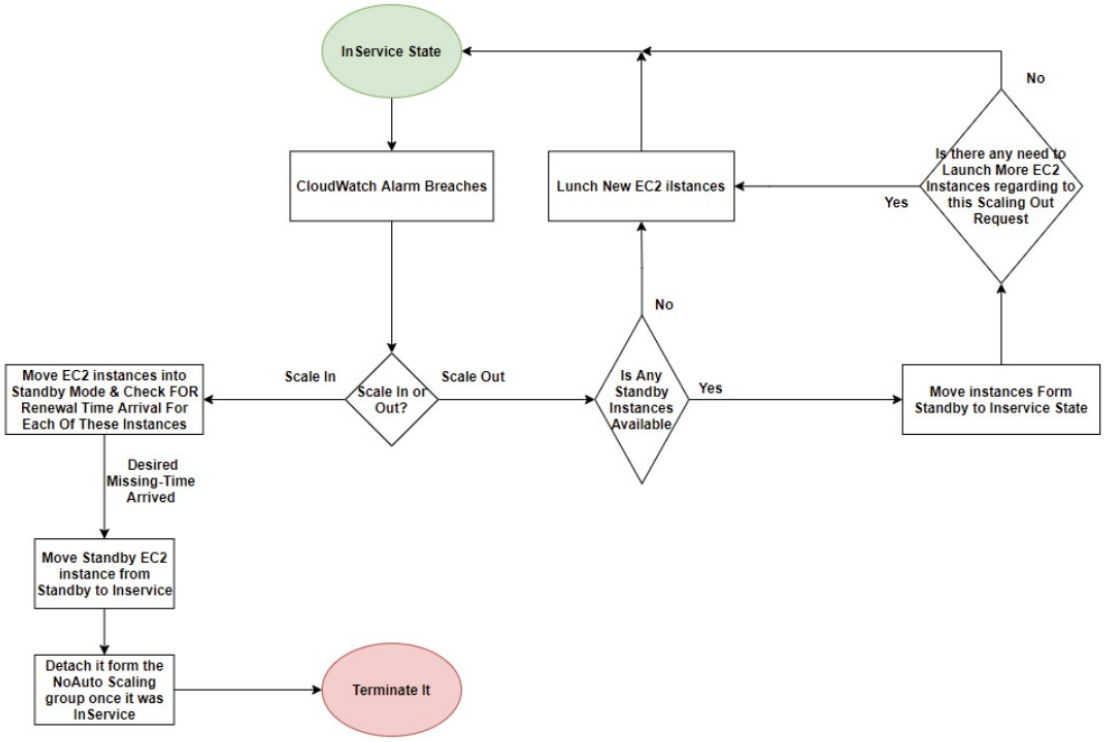

Figure 6: NoAuto Scaling Group Flowchart

3.2 Algorithm:

Pseudocode for how to scale a NoAuto scaling group in or out would be as follows:

Algorithm 1 NoAuto Scaling Group Pseudocode

```
MT ← Desired Missing Time
RT ← Time remainded to Renewal Time
if (RT ≤ MT) and (State = Standby) then
    move Standby instance to Inservice state
    Wait until it goes to Inservice state
    Detach it from NoAuto Scaling group
    Terminate it
end if
if Scale in alarm breached then
    x ← number of requested instances
    move x number of Inservice instances into Standby state
end if
if Scale out alarm breached then
    x ← number of requested instances
    y ← number of standby instances
    if x ≤ y then
        move x number of standby EC2 instances to Inservice state
    else
        z ← x − y
        move x number of standby EC2 instances to Inservice state
        launch z number new EC2 instances
    end if
end if
```

3.3 NoAuto Scaling Group vs Auto Scaling Group

In the following section, over a simulated case study with randomly generated scaling events (Figure 7), the outcomes of NoAuto Scaling group behavior versus the ones driven from Auto Scaling group are compared. This simulated case study modeled an environment with maximum ten EC2 instances with desired capacity of four as its start point. Various randomly created load changes, causing random scaling requests during 24 hours (1 day) has been recorded and finally the number of newly launched EC2 based on both NoAuto Scaling group and Auto Scaling group approaches are compared together in Table 3.

Table 3: NoAuto Scaling Group vs Auto Scaling Group Simulation Data

Time	In Service Instance	New Instances Launched by Auto Scaling Group	New Instances Launched by NoAuto Scaling Group
12:29 AM	4	4	4
1:09 AM	7	3	3
1:34 AM	9	2	2
2:01 AM	10	1	1
2:48 AM	5	0	0
3:15 AM	7	2	1
3:49 AM	6	0	0
4:06 AM	8	2	1
6:16 AM	9	1	1
7:10 AM	5	0	0
7:18 AM	7	2	0
8:57 AM	10	3	3
11:22 AM	3	0	0
11:40 AM	4	1	0
12:09 PM	7	3	3
12:56 PM	8	1	1
1:20 PM	5	0	0
1:45 PM	8	3	0
3:34 PM	9	1	1
4:28 PM	6	0	0
4:37 PM	9	3	1
6:30 PM	8	0	0
8:28 PM	7	0	0
10:38 PM	6	0	0

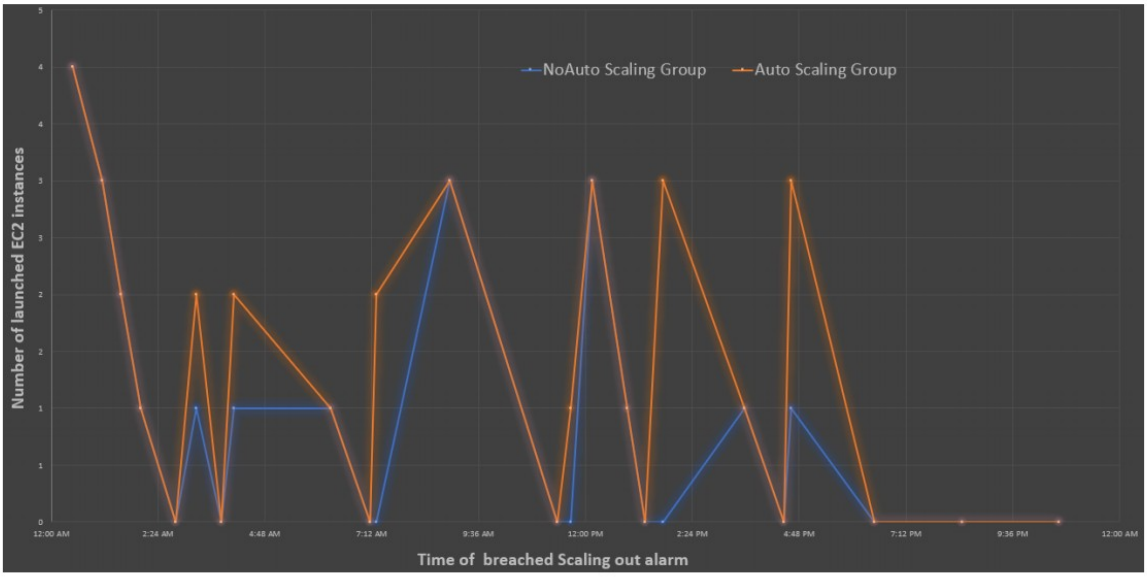

Figure 7: NoAuto Scaling Group vs Auto Scaling Group Outcomes

As it is demonstrated in Figure 7, the number of newly lunched EC2 instances based on NoAuto Scaling group approach (blue line) will never go above the orange line which presents the outcomes driven from the Auto Scaling group in responding to exact same scaling requests. This study shows how using NoAuto Scaling groups can maximize the usage of already paid EC2 instances in comparison to default Auto scaling group. Please note, even in the worst case scenario, the number of newly launched EC2 instances within a NoAuto Scaling group would be equal to the results driven from its counterpart (Auto Scaling group) and no more. This is the significance of using NoAuto Scaling groups in huge unpredictable environments where maximizing the utilization time of each paid EC2 instances would be important. This difference in number of launched EC2 instances between two

approaches would be even more recognizable during a longer period of time as per month when AWS charges its customers for their used services.

3.4 Source Code in Boto3

Boto3 is AWS SDK for Python and it can be used for creating, configuring and managing AWS services. Main lambda functions for scaling a NoAuto Scaling group in and out are as follows:

- scale-out function
- find-standby function
- terminate-instances function
- detach-ins function
- time-remained function
- scale-in function

For being able to maximize the usage of in service EC2 instances, as it is mentioned earlier, different termination policies can be used with Auto Scaling service to select the termination nominated EC2 instance. Among them, using ClosestToNextInstanceHour termination policy is the one that maximize the usage of already paid EC2 instances since it selects instances which are closest to the next billing hour [**20**]. To implement same idea over a NoAuto Scaling group when Scale-in function is looking for moving InService EC2 instances into Stnadby state, time-remained function helps scale-in function with looking after EC2 instances with less than 5 minutes remained to their renewal time first. If more instances were required, it checks instances with

less than 10 minutes and then less than 15 minutes remained to their renewal time so it assures moving InService EC2 instances into Standby state is similar enough to built-in AWS ClosestToNextInstanceHour termination policy. At the end, if there were any needs for scaling in more EC2 instances, Scale-in function selects among remained InService instances based on its internal algorithm.

```
import boto3
import datetime

ec2 = boto3.resource('ec2')
asg = boto3.client('autoscaling')
asgname = 'EC2_T2Micro_Test'
max_boundary = 10

def scale_out (num_of_inc):
  current_size=response['AutoScalingGroups'][0]['MaxSize']
  new_size = current_size + num_of_inc
  If new_size <= max_boundary:
    start_num = 0
    for j in range (0 , num_of_inc):
      instance_id = find_standby()
      if instance_id is not None:
        asg.exit_standby(
          InstanceIds=[instance_id],
          AutoScalingGroupName= 'EC2_T2Micro_Test'
          )
        instance = ec2.Instance(id=instance_id)
        instance.wait_until_running()
        current_size+=1
        asg.update_auto_scaling_group(
          AutoScalingGroupName=asgname,
          MaxSize=current_size
          )
```

```
      continue
      else:
        ins_id =(launch_new())
        instance = ec2.Instance(id=ins_id)
        instance.wait_until_running()
        add_to_asg(ins_id)
        current_size+=1
        asg.update_auto_scaling_group(
          AutoScalingGroupName=asgname,
          MaxSize=current_size
          )
  asg.update_auto_scaling_group(AutoScalingGroupName=asgname,
                                DesiredCapacity =,
                               current_size)


def find_standby():
    for i in instances:
    stat= i.get('LifecycleState')
    instance_id = i.get('InstanceId')
    if (stat =='Standby'):
      return(instance_id)

def terminate_instances(id):
  ins_id = [id]
  ec2_client = boto3.resource('ec2','us-east-1')
  ec2_client.instances.filter(InstanceIds = ins_id).terminate
```

```
                                ()


def detach_ins (num_of_dec):
    current_size=response['AutoScalingGroups'][0]['MaxSize']
    new_size = current_size - num_of_dec
    for i in range (num_of_dec):
      instance_id = find_standby()
      asg.exit_standby(InstanceIds=[instance_id],
                                AutoScalingGroupName='
                                EC2_T2Micro_Test')
      instance.wait_until_running()
      asg.detach_instances(InstanceIds=[str(instance_id)],
      AutoScalingGroupName='EC2_T2Micro_Test',
                                ShouldDecrementDesiredCapacity
                                =TRUE)
      terminate_instances (str(instance_id))
    asg.update_auto_scaling_group(AutoScalingGroupName=
                                    asgname,MaxSize=new_size)


def time_remained(desired_min):
  for i in instances:
    launchtime=ec2.Instance(i.get('InstanceId')).launch_time
    current_time = datetime.datetime.now(launchtime.tzinfo)
    running_time = current_time - launchtime
    running_time_minutes = running_time.seconds//60
    meanutes_inthisPerioed = running_time_minutes%60
    remaining_time2Bill = 60 - meanutes_inthisPerioed
```

```
    instance_id = i.get('InstanceId')
    if remaining_time2Bill <= desired_min:
      return (instance_id)

def scale_in (num_of_dec):
  while num_of_dec != 0:
    ins_id_less5= time_remained(5)
    if ins_id_less5 is not None:
      asg.enter_standby(
        InstanceIds=[ins_id_less5],
        AutoScalingGroupName= 'EC2_T2Micro_Test',
        ShouldDecrementDesiredCapacity= True)
      num_of_dec -=1
      continue
    ins_id_less10= time_remained(10)
    if ins_id_less10 is not None:
      asg.enter_standby(
        InstanceIds=[ins_id_less10],
        AutoScalingGroupName= 'EC2_T2Micro_Test',
        ShouldDecrementDesiredCapacity= TRUE)
      num_of_dec -=1
      continue
    ins_id_less15= time_remained(15)
    if ins_id_less15 is not None:
      asg.enter_standby(
        InstanceIds=[ins_id_less15],
        AutoScalingGroupName= 'EC2_T2Micro_Test',
```

```
        ShouldDecrementDesiredCapacity= TRUE)
      num_of_dec -=1
      continue
    for i in instances:
      stat= i.get('LifecycleState')
      instance_id = i.get('InstanceId')
      if(stat =='InService'):
        asg.enter_standby(
          InstanceIds=[instance_id],
          AutoScalingGroupName= 'EC2_T2Micro_Test',
          ShouldDecrementDesiredCapacity= TRUE)
        num_of_dec -=1
new_size = response['AutoScalingGroups'][0]['
                              DesiredCapacity'])
asg.update_auto_scaling_group(AutoScalingGroupName=asgname,
                              MaxSize=new_size)
```

4 Conclusion and Future Works

In this thesis, an approach to enhance built-in AWS dynamic Scaling over any Auto Scaling group is proposed by introduction of NoAuto Scaling group as my final solution to minimize customers' costs for having and maintaining their Auto Scaling groups. The amount of benefit would be increased as the number of EC2 instances and their Metrics fluctuation do, so the proposed approach can be useful for Enterprise level firms and customers with large numbers of EC2 instances with unpredictable significant load changes, like Stock Markets, Trade Markets and so on.

Although I tried to demonstrate each obstacle, examine and explain various approaches to resolve them, there are still some fields left for future works as follows:

Find a solution for renewal time tracking issue. In best scenario, each instances in Standby mode should be able to track its own renewal time, so any time near to its renewal time triggers a CloudWatch alarm to initiate the Termination phase. This can be done through deployment of some service agents for EC2 instances.

Another work left for future is to aind a solution to implement the Escape-Gate. As it is discussed earlier this can be the most optimum approach for minimizing dynamic scaling hidden costs. This approach might be doable with a Third Party Cloud Services Provider which acts like an intermediary

step between customers and Amazon Web Services.

5 References

[1] https://aws.amazon.com/about-aws/

[2] https://aws.amazon.com/what-is-cloud-computing/

[3] https://aws.amazon.com/types-of-cloud-computing/

[4] https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html

[5] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/viewing_metrics_with_cloudwatch.html

[6] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-cloudwatch-new.html

[7] https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scale-based-on-demand.html

[8] https://aws.amazon.com/premiumsupport/knowledge-center/ec2-instance-hour-billin

[9] https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-mach

[10] https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroupLifecycl
html

[11] https://docs.aws.amazon.com/autoscaling/ec2/userguide/attach-instance-asg.
html

[12] https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-simple-step.
html#as-step-scaling-warmup

[13] https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroupLifecycl
html#as-lifecycle-hooks

[14] https://docs.aws.amazon.com/autoscaling/ec2/userguide/detach-instance-asg.
html

[15] https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-enter-exit-standby.
html

[16] https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scale-based-on-demand
html

[17] https://aws.amazon.com/ec2/pricing/on-demand/

[18] Biswas.A, Majumdar.S, and Haraki.A.E, 2017. A hybrid auto-scaling technique for clouds processing applications with service level agreements. Journal of Cloud Computing: Advances, Systems and Applications. DOI 10.1186/s13677-017-0100-5

[19] https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-suspend-resume-proces
html?icmpid=docs_ec2as_help_panel

[20] https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-instance-termination.
html

[21] https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/ScheduledEvents.
html

[22] https://aws.amazon.com/lambda/

[23] https://docs.aws.amazon.com/sns/latest/dg/sns-lambda-as-subscriber.
html