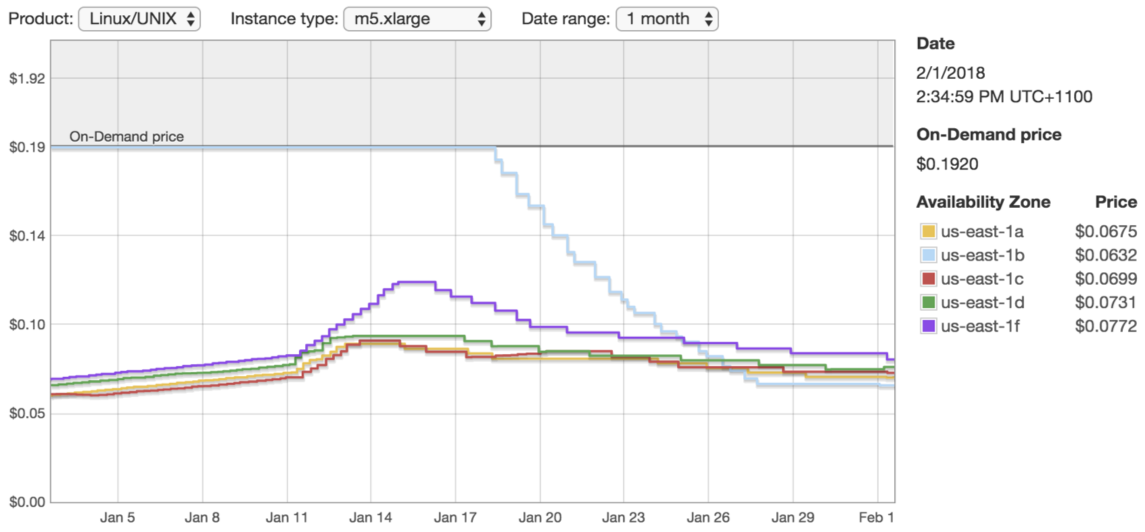
CHAPTER 1: INTRODUCTION

Cloud Infrastructure providers allow the following six types of virtual instance purchasing options to optimize operating cost based on specific computational needs which are On-demand, Reserved, Scheduled, Spot, Dedicated Host, and Dedicated Instances. ("Instance Purchasing Options, 2018", n.d.)

The 'Spot Instance Pricing System' was introduced by Amazon Web Services to trade their spare computational capacity on the open market through a bidding mechanism. The cloud provider sets the price of the 'Spot Instance' (SP) according to real-time demand and supply. To use SP, a user needs to create a 'Spot Fleet Request' by including the maximum price they want to pay per hour per instance and additional information such as type and availability zone for the instance. If the bid amount exceeds the current spot instance price (**Fig.** 1) and capacity is available, the request is fulfilled automatically.

Data from January 1, 2018 to February 1, 2018



Product: Linux/UNIX Instance type: m5.xlarge Date range: 1 month

Date
2/1/2018
2:34:59 PM UTC+1100

On-Demand price
$0.1920

Availability Zone	Price
🟨 us-east-1a	$0.0675
🟦 us-east-1b	$0.0632
🟥 us-east-1c	$0.0699
🟩 us-east-1d	$0.0731
🟪 us-east-1f	$0.0772

Figure 1: *One month spot price history of AWS m5.xlarge in different availability zones.*

These of SP can reduce cloud infrastructure bills by 80% to 90% compared to On-Demand instances. However, this process is dynamic and can be challenging as SP can be terminated by the cloud provider when the demand for SP rises and a higher bid is received for the same instance. In such cases, the cloud provider issues a warning notice of 120 seconds prior to terminating the SP ("How Spot Instances Work, 2018", n.d.). Although SP is comparatively very cheap, web-application cannot be deployed utilizing only SP because of this nature.

Our proposed auto-scaling mechanism (**Fig.** 2) uses a combination of SP and On-Demand Instance which not only reduces financial cost but also ensures a fault-tolerant application. This mechanism prompt n-Demand Instances to scale up only when an SP termination notice is received from the cloud provider and automatically scales down when a new Spot fleet request is fulfilled. This simple strategy can reduce cloud infrastructure costs significantly. SP is ideal for availability and non-time-critical applications which requires massive computational capacity. However, it is generally believed that the SP model is not suitable for persisting web-application or general web application. In this paper, we demonstrate that by using an intelligent auto-scaling mechanism and carefully designing a fault-tolerant mechanism, it t is possible to host web applications or any type of persistent web application efficiently in terms of cost and availability.

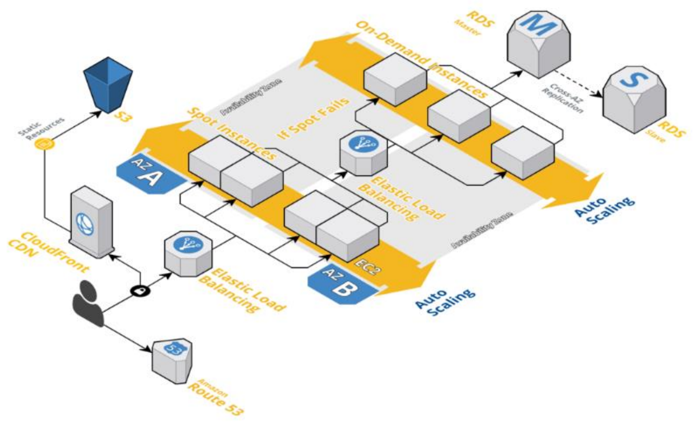


Figure 2: Proposed Auto-Scaling mechanism using Spot and On-Demand Instances

CHAPTER 2: LITERATURE REVIEW

2.1 Horizontal Approach of Auto-Scaling for Web Application

Most of the research that has been carried out in cloud auto-scaling is relevant to ‘Horizontal Auto-Scaling’ strategies (“Auto-scaling - Wikipedia”, n.d.). Fundamentally, auto-scaling strategies can be classified into three categories: reactive, proactive and hybrid or mixed. The reactive auto-scaling approach scales applications according to changes in traffic-load. The proactive approach predicts future traffic-loads and performs scaling of applications in advance. Hybrid scaling is a dynamic scaling strategy which scales applications reactively and proactively based on workloads.

Most of the Cloud Service Providers follow reactive-based auto-scaling mechanism. Among them, the Amazon Web Services (AWS) Auto-Scaling service (**Fig.** 3) is the most frequently used (“AWS Auto Scaling”, n.d.; Hu, Deng, & Peng 2016). To use the AWS auto-scaling service, the user needs to create an auto-scaling group with cloud-formation instructions to identify which types of virtual machines and system images to use when auto-scaling new instances. A scaling policy is defined in auto-scaling group using event-based metrics from Cloudwatch like “Scale-up 3 new instances when CPU usage is greater than 80%” (“AWS Auto Scaling”, n.d.).

RightScale, a popular cloud service provider, scale applications based on a voting mechanism that allows all currently running virtual instances to vote whether to grow or shrink cluster sizes of the application depending on their own condition (Toosi, Khodadadi, & Buyya 2015).

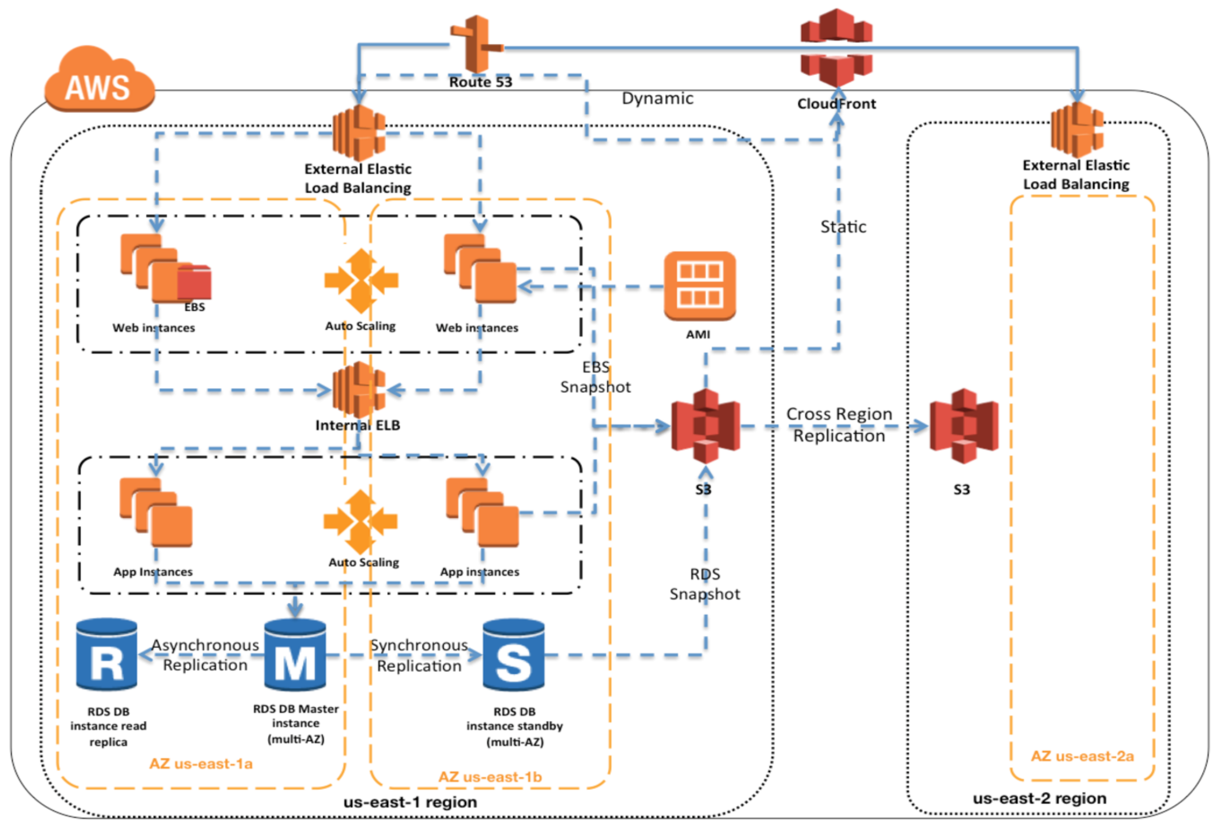


Figure 3: *The most popular horizontal Auto-Scaling system used by cloud users (PATIL 2017; Jeff Barr & Varia 2011).*

A significant number of dynamic Auto-Scaling mechanisms have been proposed by researchers which go beyond simple Cloudwatch metrics rules to make scaling decisions. These auto-scaling models aim to answer fundamental questions regarding the amount of computational resources required to serve a specific amount of traffic pressure considering constraints of fault-tolerance and quality of service. Most auto-scaling models were developed using simple profiling methods, including queuing strategies (Urgaonkar et al., 2008;Jiang et al. 2013; Han et al. 2014; Fernandez, Pierre, & Kielmann, 2014; Chen & Bahsoon 2015; Vondra & Šedivý 2017; Hu, Deng, & Peng 2016) which obscure the application model as a set of computational parallel queues or virtual network of queues. Other auto-scaling mechanisms involve reinforcement strategies (Tighe & Bauer, 2014; Barrett, Howley, & Duggan, 2012; Bu, Rao, & Xu, 2013).

The proactive auto-scaling approach is also widely used because there is some latency associated with the launch and configuration of virtual instances which creates a computational resource gap when workload or traffic pressure suddenly rises beyond capacity (Hirashima, Yamasaki, & Nagura 2016). Cloud providers use this approach to comply with strict service level agreements (SLA) as it is important to provide computational resources before the workload is actually increased. As traffic load of web applications usually follow temporal patterns, state-of-art time series analysis and pattern recognition techniques can be used to predict the future traffic load of the application. This technique has been used in auto-scaling of applications in a significant number of studies (Jiang et al., 2013; Roy, Dubey, & Gokhale, 2011; Yang et al.,2013; Caron, Desprez, & Muresan, 2011; Islam et al., 2012; Fang et al., 2012; Herbst et al., 2014; Dutta et al., 2012).

Most studies in auto-scaling have been conducted using homogenous cloud resources while few have used heterogeneous cloud resource to provision web applications. Sharma et al. (2011) and Srirama and Ostovar (2014) have adopted integer linear programming (ILP) to represent optimal heterogeneous cloud resources configuration issue supporting strict service level agreement limitations. Fernandez, Pierre, and Kielmann (2014) have applied tree paths to determine different sequences of heterogeneous cloud resources and examined the tree-path to discover most suitable auto-scaling plan according to cloud users' SLA conditions.

Most of the above research was aimed at maximizing provisioning of web applications using the least amount of cloud resources and homogeneous On-demand Instance (Aslanpour, Ghobaei-Arani, & Toosi, 2017), but no emphasis was placed on increasing fault-tolerance in web applications applying auto-scaling semantics and reducing the cost of heterogeneous SP.

2.2 Auto-Scaling Utilizing Spot Instance

SP are generally used in a non-time-critical application context. There has been some research to utilize SP in non-availability critical application (Costache et al., 2012; Binnig et al.,2015; Poola, Ramamohanarao, & Buyya, 2014; Lu et al., 2013; Voorsluys & Buyya, 2012; Chen, Lee, & Tang, 2014; Knauth & Fetzer, 2012; Zafer, Song, & Lee, 2012; Chu & Simmhan, 2014; Gong, Li, & Fan, 2008) mostly in high performance computational processes, cloud data analytics (big data), MapReduce and scientific experimental workflow.

For these purposes, the fault-tolerance strategy is often established on check-pointing, migration and replication. SP has been utilized in multiple studies using a check-pointing approach (Jangjaimon & Tzeng, 2015; Yi, Andrzejak, & Kondo, 2012; Jung et al., 2011). Researchers combined multiple fault-tolerant mechanisms to increase the cost-efficiency and performance of batch processing using SP.

Zhao et al. (2012) and Qu, Calheiros, and Buyya (2016) have proposed strategies to predict future computational resources and scale systems using On-Demand and SP. However, they only utilized homogeneous cloud resources in line with their aim to utilize knowledge of future developments into the design of cloud resource usage. There was no emphasis on building a fault-tolerant application by Auto-Scaling cloud resources dynamically.

Sharma et al. (2015) designed an IaaS cloud platform - SpotChcek), utilizing SP. The objective of this study was to provide end users with high availability of SP, utilizing nested virtualization, live Virtual Machine migration, and time-constraint virtual machine migration with a memory check-pointing mechanism. It dynamically moves users virtual machines when SP are terminated by the cloud providers. This approach is beneficial for cloud brokers and large corporations but does not empower smaller groups to explore the power of SP.

CHAPTER 3:

PROPOSED AUTO-SCALING SYSTEM MODEL

3.1 Auto-Scaling Architecture using Spot Instance

As demonstrated in **Fig. 4**, the proposed Auto-Scaling mechanism utilizes On-Demand and SP under an Elastic Load Balancer. Both, homogeneous instances (On-Demand VM) and heterogeneous instances (SP VM) have been incorporated into the Auto-Scaling group configurations. SP are 80 to 90% cheaper compared to On-Demand Instances (Tang, Yuan, & Li, 2012). AWS Cloud-watch has been enabled to take the role of monitoring module in this web-application server configuration which triggers virtual machine usage metrics through events and we can subscribe to those events using AWS Cloud SDK (Anand 2017). In our Auto-Scaling architecture, we try to reduce infrastructure costs by utilizing Spot Instance as much as possible. However, Spot Instance can be terminated anytime by the Cloud Provider by transmitting 2 minutes signal notice into event metrics logs. So, we get 120 seconds to scale-up new On-Demand instance from a pre-configured Elastic Block Storage loaded with AMI (Amazon Machine Image) configured with our codebase snapshots.

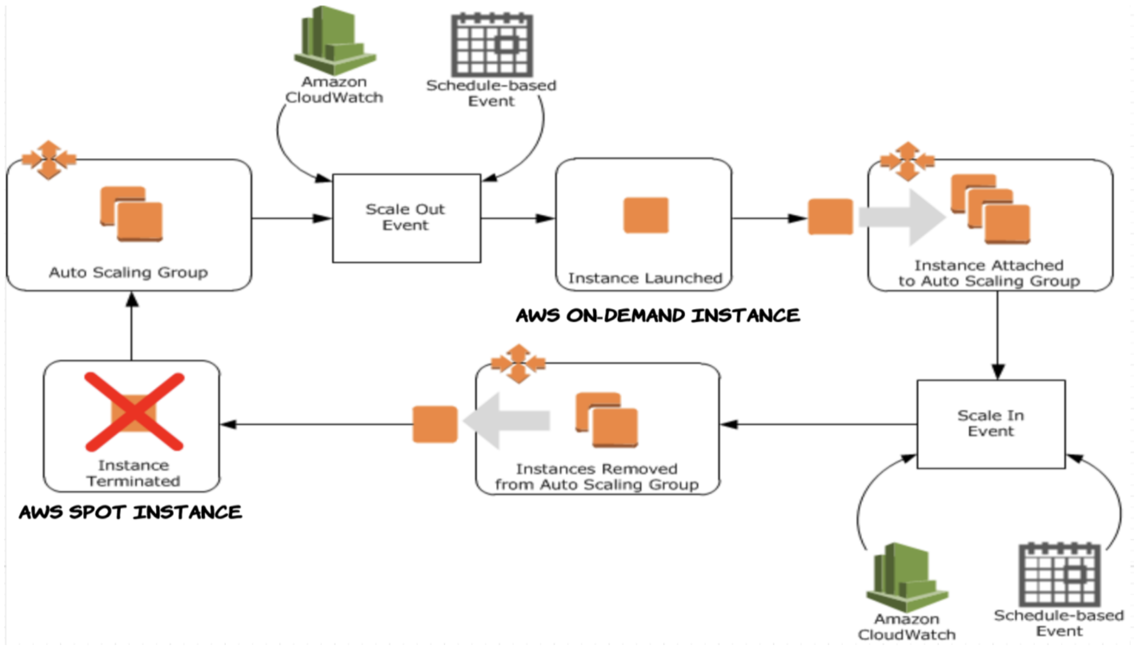


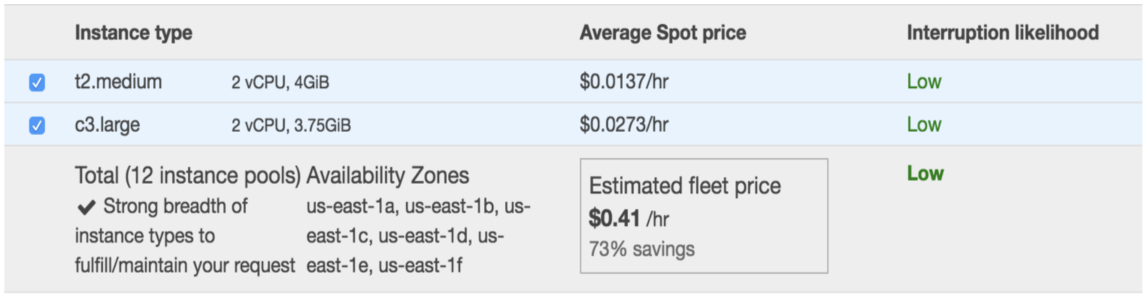
Figure 4: *Dynamic Auto-Scaling between Spot Instance and On-Demand Instance utilizing Cloud-watch event metrics.*

3.2. Fault-tolerance Strategy

If the demand for the certain type of the SP rises, the price of that particular SP's VM also goes up (Jung et al., 2011). If the entire application system uses just one SP or one type of SP, we need to scale up 100% computational capacity to retain availability of the system. A solution is to configure the system so that multiple types of SP can be used. Thus, if the demand for particular types of SP goes up, it does not impact the fault-tolerance of the system.

Using an AWS Software Development Kit (SDK) based SP search mechanism and utilizing Spot Fleet API, it becomes apparent that for a consistent CPU/memory computational job cost can be reduced by 73% compared to applying On-Demand VM (**Fig.** 5). Selecting a multiple instances pool (2 t2 medium and 2 c3 large VCPU) and availability zone reduces the interruption likelihood significantly. Cost can be lowered by 85% for a sample large-scale web service application (simulating average traffic load) by selecting multiple instance pools (2 t2 medium, 32 cc2 large, 32 cr1 large) in multiple availability zones (**Fig. 6**). Furthermore, cost can be reduced by 74% for a sample large-scale Map Reduce job (simulating average workload) by selecting multiple instance pools (various Spot Instance types) in multiple availability zones (**Fig.** 7).

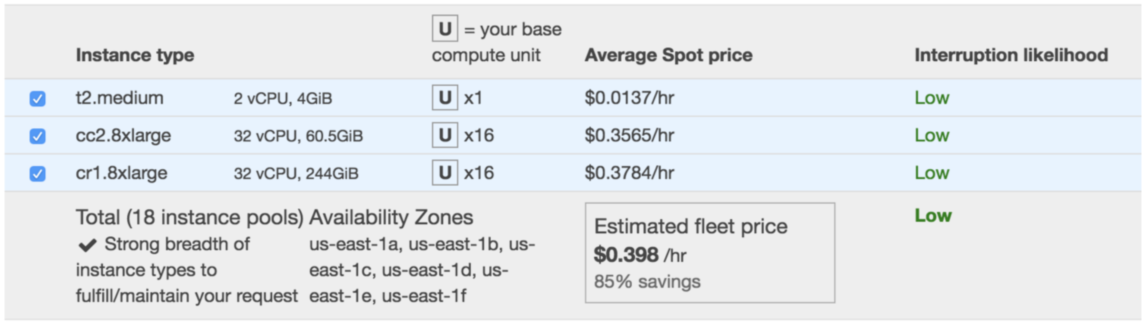
(AWS Console, Feb 02, 2018)



	Instance type		Average Spot price	Interruption likelihood
✓	t2.medium	2 vCPU, 4GiB	$0.0137/hr	Low
✓	c3.large	2 vCPU, 3.75GiB	$0.0273/hr	Low
	Total (12 instance pools) ✓ Strong breadth of instance types to fulfill/maintain your request	Availability Zones us-east-1a, us-east-1b, us-east-1c, us-east-1d, us-east-1e, us-east-1f	Estimated fleet price **$0.41** /hr 73% savings	Low

Figure 5: *Spot Instance selection using Spot Fleet API for consistent CPU/memory ratio at the best price*

(AWS Console, Feb 02, 2018)



	Instance type		U = your base compute unit	Average Spot price	Interruption likelihood
✓	t2.medium	2 vCPU, 4GiB	U x1	$0.0137/hr	Low
✓	cc2.8xlarge	32 vCPU, 60.5GiB	U x16	$0.3565/hr	Low
✓	cr1.8xlarge	32 vCPU, 244GiB	U x16	$0.3784/hr	Low

Total (18 instance pools) ✓ Strong breadth of instance types to fulfill/maintain your request

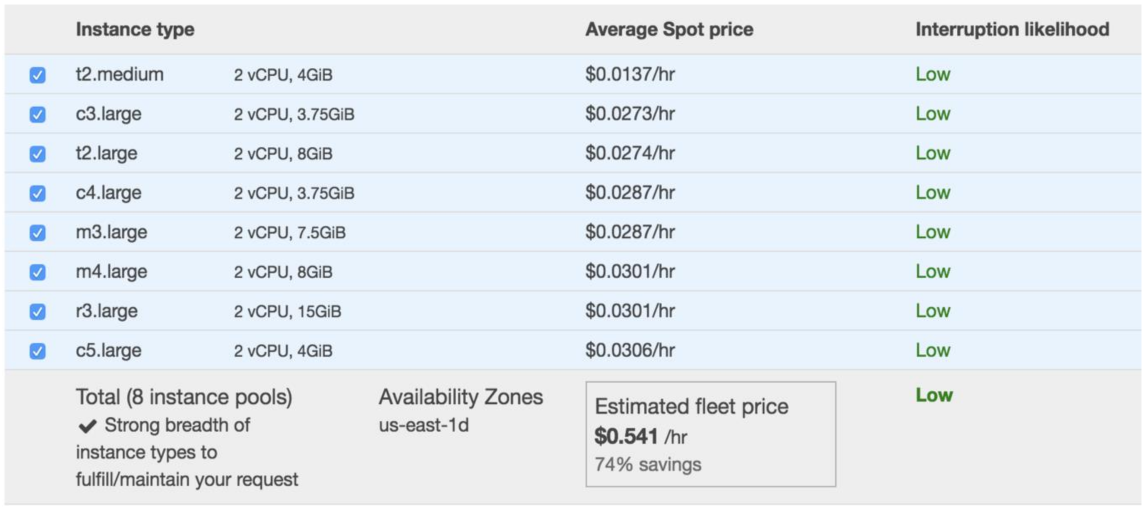
Availability Zones us-east-1a, us-east-1b, us-east-1c, us-east-1d, us-east-1e, us-east-1f

Estimated fleet price **$0.398** /hr 85% savings

Low

Figure 6: *Spot Instance selection using Spot Fleet API for sample Web service configuration at the best price*

(AWS Console, Feb 02, 2018)



	Instance type		Average Spot price	Interruption likelihood
✓	t2.medium	2 vCPU, 4GiB	$0.0137/hr	Low
✓	c3.large	2 vCPU, 3.75GiB	$0.0273/hr	Low
✓	t2.large	2 vCPU, 8GiB	$0.0274/hr	Low
✓	c4.large	2 vCPU, 3.75GiB	$0.0287/hr	Low
✓	m3.large	2 vCPU, 7.5GiB	$0.0287/hr	Low
✓	m4.large	2 vCPU, 8GiB	$0.0301/hr	Low
✓	r3.large	2 vCPU, 15GiB	$0.0301/hr	Low
✓	c5.large	2 vCPU, 4GiB	$0.0306/hr	Low

Total (8 instance pools) ✓ Strong breadth of instance types to fulfill/maintain your request

Availability Zones us-east-1d

Estimated fleet price **$0.541** /hr 74% savings

Low

Figure 7: *Spot Instance selection using Spot Fleet API for sample MapReduce job at the best price (Feb 02, 2018)*

Upon receiving a termination notice from cloud-watch event metrics, a new on-Demand instance is launched and Elastic Block Storage with the proposed codebase snapshot is attached to the new Instance. The Elastic Load Balancer resolves traffic-load or workload to the new virtual instance.

Simultaneously, a new suitable SP is sourced by using Spot Fleet API and utilizing the SP price history to predict the best bidding price for the SP. If the bidding price meets the standard Spot Instance price (demand and supply mechanism) at that particular moment (Jangjaimon & Tzeng, 2015), the request is awarded a new SP. The On-Demand Instance can be terminated to reduce cloud operational cost as Elastic Load Balancer will now rotate to the newly awarded SP.

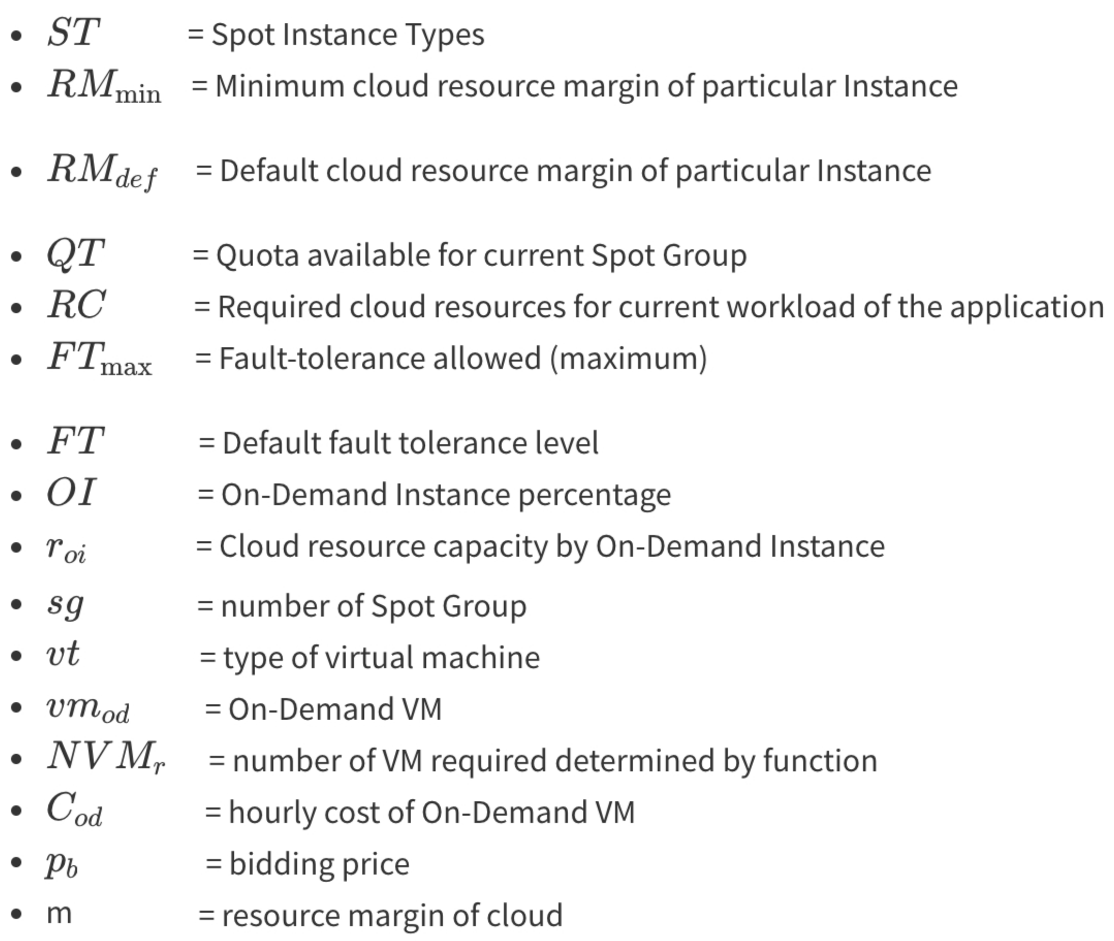
Elastic Load Balancer simply redirects the traffic and workload to the newly launched On-Demand instances bypassing the SP marked for termination. Thus, the termination of the first SP does not affect the availability of our application as the Auto-Scaling group and the Load balancer migrate the workload into the newly configured On-Demand Instance before the release of the SP.

This research utilizes intelligent script AWS Software Development Kit (SDK) which starts to search for a new SP using AWS Spot Fleet API on receiving termination request of any Spot Instance.

3.3. Reliability and Cost Efficiency of proposed model

Through the Auto-Scaling provisioning shown in **Fig.** 4, cloud resource costs are reduced significantly (by more than 80% on average) compared to On-Demand Instance cost. Quick Auto-scaling mechanism utilizing On-Demand Instance also secures fault-tolerance in the web application.

In the proposed model, fault-tolerance depends on the number of Spot Instances and On-Demand Instances and the computational resource capacity they provide, the workload of the application and the resource margin of the cloud.



• ST	= Spot Instance Types
• $RM_{\min}$	= Minimum cloud resource margin of particular Instance
• RM_{def}	= Default cloud resource margin of particular Instance
• QT	= Quota available for current Spot Group
• RC	= Required cloud resources for current workload of the application
• $FT_{\max}$	= Fault-tolerance allowed (maximum)
• FT	= Default fault tolerance level
• OI	= On-Demand Instance percentage
• r_{oi}	= Cloud resource capacity by On-Demand Instance
• sg	= number of Spot Group
• vt	= type of virtual machine
• vm_{od}	= On-Demand VM
• NVM_r	= number of VM required determined by function
• C_{od}	= hourly cost of On-Demand VM
• p_b	= bidding price
• m	= resource margin of cloud

Figure 7: Symbol representations

$$FT = \Sigma\left(\frac{ST \times RM_{def} + OI \times r_{oi} \times vm_{od}}{RC \times m}\right) \text{ —— (8a)}$$

According to **Formula. 8a**, the number of On-Demand Instances will depend on the number of SP that are interrupted by the cloud provider. Thus, the percentage of the On-Demand Instances equals the number of On-Demand instances being used divided by the total number of VM required by the application system. Thus, if a request is received to terminate three SPs, three On-Demand Instances of the same computing capacity or combination of On-Demand Instance capacity which can handle the same workload that terminated Spot Instance could do.

$$OI = \Sigma\left(\frac{vm_{od}}{NVM_r}\right) \text{ ——- (8b)}$$

According to **Formulae 8b and 8c**, the number of Spot and On-Demand Instances depend on the application workload, cloud margin and minimum cloud resource required to support the web application.

$$ST \ \alpha \ RC \times m \times RM_{\min} \text{ ———- (8c)}$$

According to **Formula 8d,** cloud resource costs depends on the hourly bidding price of SP, the number of SPs used, the number of On-Demand Instances and the hourly cost of On-Demand Instances and the workload of the application.

$$Total\ Cost,\ TC = \frac{P_b \times ST \times QT + C_{od} \times vm_{od}}{\frac{1}{RC}} \cdot m \text{ ———— (8d)}$$

CHAPTER 4: AUTO SCALING POLICY

Auto-Scaling Policy

Based on the preceding Auto-Scaling model, a highly cost-efficient Auto-Scaling policy is recommended that complies with suggested fault-tolerance semantics.

The Dynamic Scaling up policy is invoked when SP is interrupted by the cloud provider or the current VM provision cannot cope with the workload for unspecified reasons. Similarly, the scaling down policy is invoked to terminate an On-Demand Instance when the request for a new SP using Spot Fleet API has been fulfilled.

4.1. Scaling up Algorithm

According to **formula 8e**, the new virtual machine is only scaled up when the computation usage goes above 80% in the application system.

$If\ RC > 80\%\ of\ (\ ST \times vm_{od} \times m)$ —— (8e)

$vm_{od} \mathrel{+}= 1\ AND\ \ ST \mathrel{+}= 1$

$If\ ST = -1$ (gets termination notice) [120 seconds]

$vm_{od} \mathrel{+}= 1\ AND\ \ ST \mathrel{+}= 1$ — (8f)

An On-Demand instance is requested as soon as the Spot Instance termination notice arrives and a request for an SP is made concurrently (**Formula 8f**).

4.2. Spot Instance Bidding Price Algorithm

The minimum price of the SP is determined by using a Spot Fleet Request API.

$P_b = $ min Of current standard Spot Instance Price of specific type

4.3. Scaling down of On-Demand Instance Algorithm

As soon the Spot Instance has been deployed and is running with the codebase snapshot virtual machine, the On-Demand Instance is scaled down (**Formula 8g**).

$if\ sg\ =\ NVM_r\ +\ new\ ST$ —— (8g)

$vm_{od}\ -=\ 1$

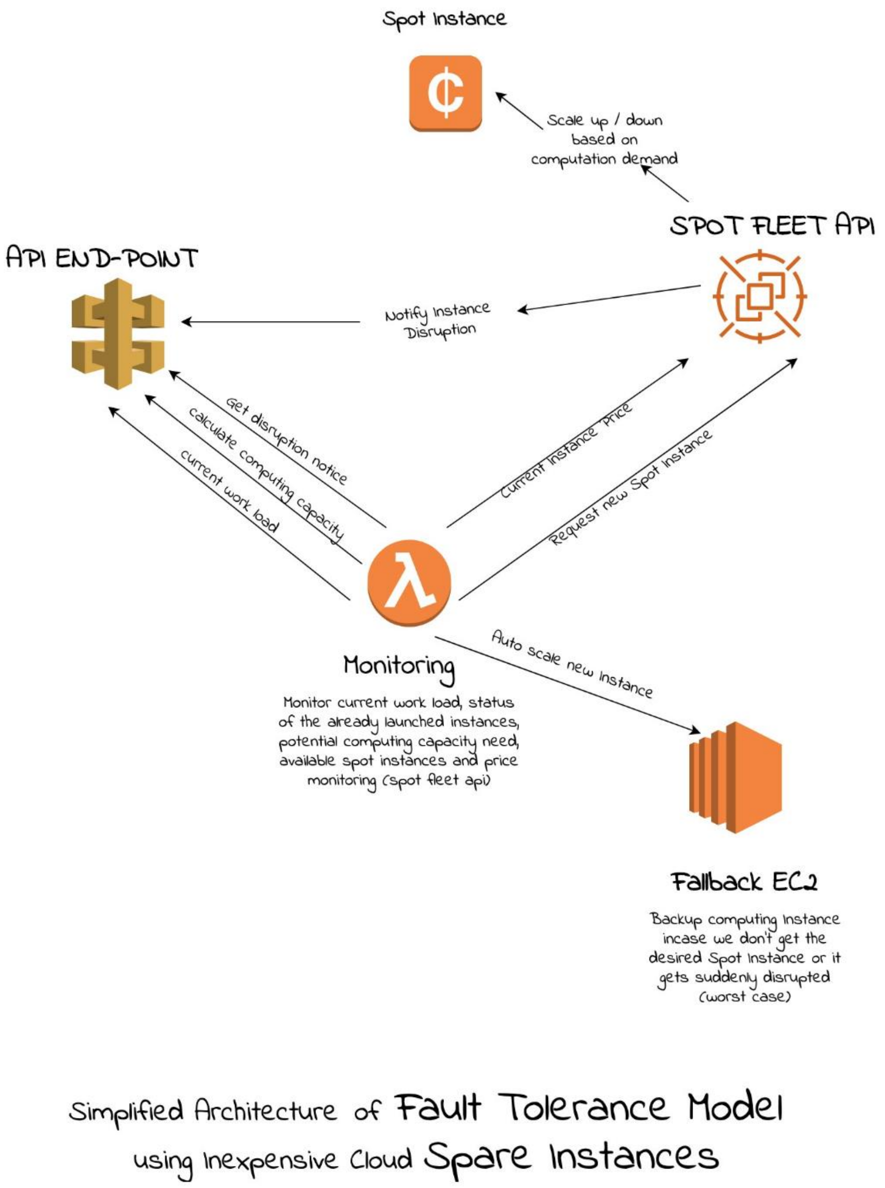

Spot Instance
Scale up / down based on computation demand
SPOT FLEET API
API END-POINT
Notify Instance Disruption
Get disruption notice
calculate computing capacity
current work load
Current Instance Price
Request new Spot Instance
Auto scale new instance
Monitoring
Monitor current work load, status of the already launched instances, potential computing capacity need, available spot instances and price monitoring (spot fleet api)
Fallback EC2
Backup computing instance incase we don't get the desired Spot Instance or it gets suddenly disrupted (worst case)
Simplified Architecture of Fault Tolerance Model using Inexpensive Cloud Spare Instances

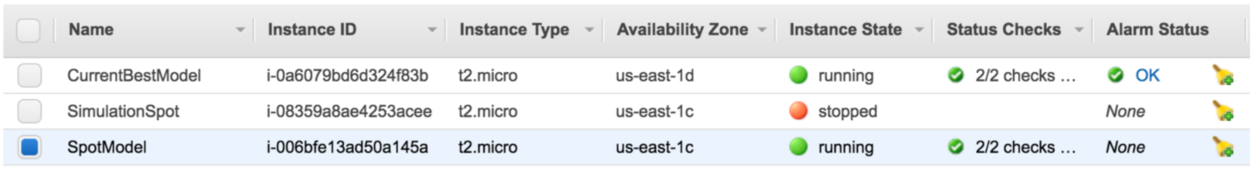
Figure 9: Simplified Architure of Fault Tolerance Model

CHAPTER 5:

PREFERENCE EVALUATION OF THE PROPOSED MODEL

5.1. Simulation Environment on AWS

A simulation testbed of the web application in AWS was configured which permitted a comparison of the performance of different cloud configurations and policies utilizing traces from real applications and the SP markets. We have used dynamic SP and On-Demand virtual machine Instance selection through AWS SDK, combining various computational capacities to meet the application workload on different traffic-load groupings **(Table A).**



	Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status
	CurrentBestModel	i-0a6079bd6d324f83b	t2.micro	us-east-1d	● running	✓ 2/2 checks ...	✓ OK
	SimulationSpot	i-08359a8ae4253acee	t2.micro	us-east-1c	● stopped		*None*
	SpotModel	i-006bfe13ad50a145a	t2.micro	us-east-1c	● running	✓ 2/2 checks ...	*None*

Table A: *Costs comparison between different combination of On-Demand and Spot Instance usage*

B. Costs experiments with different configurations

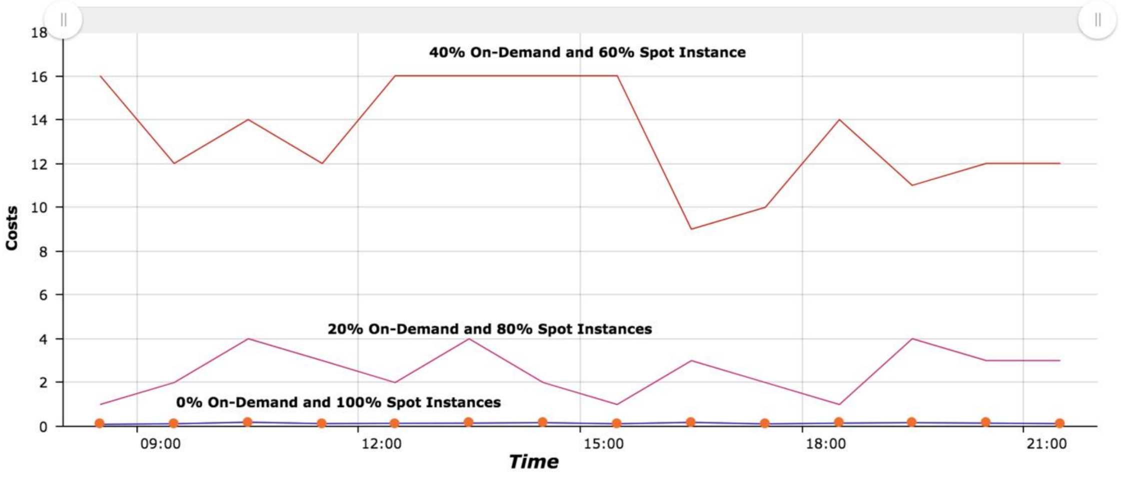


Figure 10: *Costs comparison between different combination of On-Demand and Spot Instance usage*

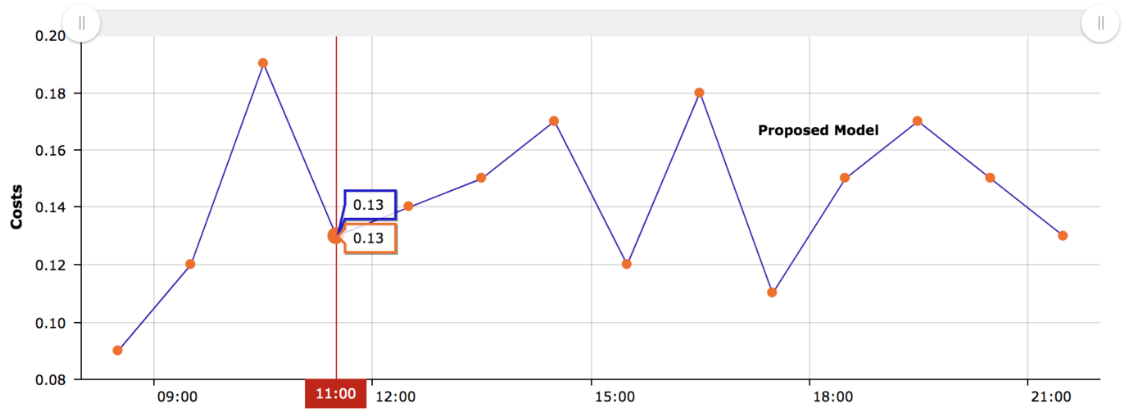


Figure 11: *Hourly costs in USD of a sample fault-tolerant web application using our proposed model (Tested in AWS, Feb 02 2018)*

According to **Fig. 10 and 11**, the proposed Auto-Scaling model significantly reduces cloud infrastructure costs by maximizing the utilization of SP. However, it also shows that higher fault-tolerance levels incurs additional cost. During high market demand for SP, the cloud resource costs are increased significantly. In the simulation test, there was no case where more than one whole Spot Group failed or the system crashed due to the higher workload. Every time SP was interrupted by the cloud provider, a new On-Demand Instance was scaled up within 2 minutes of the termination notice. Tus, there was no system level application crash due to traffic load or workload. However, although the simulation test application operated with fault-tolerances and higher availability, higher workload caused significant cloud resource costs.

From the performance evaluation charts **(Figure 10 and 11), the** Dynamic On-Demand and SP generation and the maintenance mechanisms described in this model, clearly reduce fault-tolerance costs by 10 to 15% compared to the current heterogeneous Spot Instance auto-scaling approach and 80 to 90% when compared to homogeneous only On-Demand Instance auto-scaling strategies.

CHAPTER 6: CONCLUSION

Conclusion

In this research, we examined how reliably and cost-efficiently Auto-Scale web applications utilize a combination of on-demand and heterogeneous SP. Firstly, we suggested a fault-tolerance mechanism which can handle unexpected interruptions of SP by the cloud provider. We then proposed an innovative cost-efficient Auto-Scaling strategy that fulfills various fault-tolerant interpretation for hourly billed cloud market. We deployed a prototype simulation of the proposed Auto-Scaling mechanism on Amazon Web Service EC2 for testing purposes. We carried out both real experiments and workload simulation to illustrate the efficiency of the Auto-Scaling approach by analyzing the results against benchmarks.

APPENDIX

NOTE JS Code:

```
package autospotting

import (
	"errors"
	"fmt"
	"strings"
	"time"

	"github.com/aws/aws-sdk-go/aws"
	"github.com/aws/aws-sdk-go/service/autoscaling"
	"github.com/aws/aws-sdk-go/service/ec2"
)

type autoScalingGroup struct {
	*autoscaling.Group

	name                string
	region              *region
	launchConfiguration *launchConfiguration
	instances           instances
	minOnDemand         int64
	config              AutoScalingConfig
}

func (a *autoScalingGroup) loadLaunchConfiguration() error {
	//already done
	if a.launchConfiguration != nil {
		return nil
	}

	lcName := a.LaunchConfigurationName

	if lcName == nil {
		return errors.New("missing launch configuration")
	}

	svc := a.region.services.autoScaling

	params := &autoscaling.DescribeLaunchConfigurationsInput{
		LaunchConfigurationNames: []*string{lcName},
	}
	resp, err := svc.DescribeLaunchConfigurations(params)

	if err != nil {
		logger.Println(err.Error())
		return err
	}

	a.launchConfiguration = &launchConfiguration{
		LaunchConfiguration: resp.LaunchConfigurations[0],
	}
```

```
      return nil
}

func (a *autoScalingGroup) needReplaceOnDemandInstances() bool {
      onDemandRunning, totalRunning :=
a.alreadyRunningInstanceCount(false, "")
      if onDemandRunning > a.minOnDemand {
            logger.Println("Currently more than enough OnDemand instances
running")
            return true
      }
      if onDemandRunning == a.minOnDemand {
            logger.Println("Currently OnDemand running equals to the
required number, skipping run")
            return false
      }
      logger.Println("Currently fewer OnDemand instances than required !")
      if a.allInstancesRunning() && a.instances.count64() >=
*a.DesiredCapacity {
            logger.Println("All instances are running and desired capacity
is satisfied")
            if randomSpot := a.getAnySpotInstance(); randomSpot != nil {
                  if totalRunning == 1 {
                        logger.Println("Warning: blocking replacement of
very last instance - consider raising ASG to >= 2")
                  } else {
                        logger.Println("Terminating a random spot
instance",
                              *randomSpot.Instance.InstanceId)
                        switch a.config.TerminationMethod {
                        case DetachTerminationMethod:
                              randomSpot.terminate()
                        default:

      a.terminateInstanceInAutoScalingGroup(randomSpot.Instance.InstanceId
)
                        }
                  }
            }
      }
      return false
}

func (a *autoScalingGroup) allInstancesRunning() bool {
      _, totalRunning := a.alreadyRunningInstanceCount(false, "")
      return totalRunning == a.instances.count64()
}

func (a *autoScalingGroup) calculateHourlySavings() float64 {
      var savings float64
      for i := range a.instances.instances() {
            savings += i.typeInfo.pricing.onDemand - i.price
      }
      return savings
```

```
}

func (a *autoScalingGroup) licensedToRun() (bool, error) {
      defer savingsMutex.Unlock()
      savingsMutex.Lock()

      savings := a.calculateHourlySavings()
      hourlySavings += savings

      monthlySavings := hourlySavings * 24 * 30
      if (monthlySavings > 1000) &&
            strings.Contains(a.region.conf.Version, "nightly") &&
            a.region.conf.LicenseType == "evaluation" {
            return false, fmt.Errorf(
                  "would reach estimated monthly savings of $%.2f when
processing this group, above the $1000 evaluation limit",
                  monthlySavings)
      }
      return true, nil
}

func (a *autoScalingGroup) process() {
      var spotInstanceID string
      a.scanInstances()
      a.loadDefaultConfig()
      a.loadConfigFromTags()

      logger.Println("Finding spot instances created for", a.name)

      spotInstance := a.findUnattachedInstanceLaunchedForThisASG()
      debug.Println("Candidate Spot instance", spotInstance)

      shouldRun := cronRunAction(time.Now(), a.config.CronSchedule,
a.config.CronScheduleState)
      debug.Println(a.region.name, a.name, "Should take replacement
actions:", shouldRun)

      if ok, err := a.licensedToRun(); !ok {
            logger.Println(a.region.name, a.name, "Skipping group, license
limit reached:", err.Error())
            return
      }

      if spotInstance == nil {
            logger.Println("No spot instances were found for ", a.name)

            onDemandInstance := a.getAnyUnprotectedOnDemandInstance()

            if onDemandInstance == nil {
                  logger.Println(a.region.name, a.name,
                        "No running unprotected on-demand instances were
found, nothing to do here...")
                  return
            }
```

```
		if !a.needReplaceOnDemandInstances() {
			logger.Println("Not allowed to replace any of the running
OD instances in ", a.name)
			return
		}

		if !shouldRun {
			logger.Println(a.region.name, a.name,
				"Skipping run, outside the enabled cron run
schedule")
			return
		}

		a.loadLaunchConfiguration()
		err := onDemandInstance.launchSpotReplacement()
		if err != nil {
			logger.Printf("Could not launch cheapest spot instance:
%s", err)
		}
		return
	}

	spotInstanceID = *spotInstance.InstanceId

	if !a.needReplaceOnDemandInstances() || !shouldRun {
		logger.Println("Spot instance", spotInstanceID, "is not need
anymore by ASG",
			a.name, "terminating the spot instance.")
		spotInstance.terminate()
		return
	}
	if !spotInstance.isReadyToAttach(a) {
		logger.Println("Waiting for next run while processing",
a.name)
		return
	}

	logger.Println(a.region.name, "Found spot instance:",
spotInstanceID,
		"Attaching it to", a.name)

	a.replaceOnDemandInstanceWithSpot(spotInstanceID)

}

func (a *autoScalingGroup) scanInstances() instances {

	logger.Println("Adding instances to", a.name)
	a.instances = makeInstances()

	for _, inst := range a.Instances {
		i := a.region.instances.get(*inst.InstanceId)
```

```
            debug.Println(i)

            if i == nil {
                  continue
            }

            i.asg, i.region = a, a.region
            if inst.ProtectedFromScaleIn != nil {
                  i.protected = i.protected || *inst.ProtectedFromScaleIn
            }

            if i.isSpot() {
                  i.price =
i.typeInfo.pricing.spot[*i.Placement.AvailabilityZone]
            } else {
                  i.price = i.typeInfo.pricing.onDemand
            }

            a.instances.add(i)
      }
      return a.instances
}

func (a *autoScalingGroup) replaceOnDemandInstanceWithSpot(
      spotInstanceID string) error {

      minSize, maxSize := *a.MinSize, *a.MaxSize
      desiredCapacity := *a.DesiredCapacity

      // temporarily increase AutoScaling group in case it's of static
size
      if minSize == maxSize {
            logger.Println(a.name, "Temporarily increasing MaxSize")
            a.setAutoScalingMaxSize(maxSize + 1)
            defer a.setAutoScalingMaxSize(maxSize)
      }

      // get the details of our spot instance so we can see its AZ
      logger.Println(a.name, "Retrieving instance details for ",
spotInstanceID)
      spotInst := a.region.instances.get(spotInstanceID)
      if spotInst == nil {
            return errors.New("couldn't find spot instance to use")
      }
      az := spotInst.Placement.AvailabilityZone

      logger.Println(a.name, spotInstanceID, "is in the availability
zone",
            *az, "looking for an on-demand instance there")

      odInst := a.getUnprotectedOnDemandInstanceInAZ(az)

      if odInst == nil {
            logger.Println(a.name, "found no on-demand instances that 
could be",
                  "replaced with the new spot instance", 
*spotInst.InstanceId,
                  "terminating the spot instance.")
            spotInst.terminate()
            return errors.New("couldn't find ondemand instance to 
replace")
      }
      logger.Println(a.name, "found on-demand instance", 
*odInst.InstanceId,
            "replacing with new spot instance", *spotInst.InstanceId)
      // revert attach/detach order when running on minimum capacity
      if desiredCapacity == minSize {
            attachErr := a.attachSpotInstance(spotInstanceID)
            if attachErr != nil {
                  logger.Println(a.name, "skipping detaching on-demand due 
to failure to",
                        "attach the new spot instance", 
*spotInst.InstanceId)
                  return nil
            }
      } else {
            defer a.attachSpotInstance(spotInstanceID)
      }

      switch a.config.TerminationMethod {
      case DetachTerminationMethod:
            return a.detachAndTerminateOnDemandInstance(odInst.InstanceId)
      default:
            return 
a.terminateInstanceInAutoScalingGroup(odInst.InstanceId)
      }
}

// Returns the information about the first running instance found in
// the group, while iterating over all instances from the
// group. It can also filter by AZ and Lifecycle.
func (a *autoScalingGroup) getInstance(
      availabilityZone *string,
      onDemand bool,
      considerInstanceProtection bool,
) *instance {

      for i := range a.instances.instances() {

            // instance is running
            if *i.State.Name == ec2.InstanceStateNameRunning {

                  // the InstanceLifecycle attribute is non-nil only for 
spot instances,
                  // where it contains the value "spot", if we're looking 
for on-demand
```

```
			// instances only, then we have to skip the current
instance.
			if (onDemand && i.isSpot()) || (!onDemand && !i.isSpot())
{
				debug.Println(a.name, "skipping instance",
*i.InstanceId,
					"having different lifecycle than what we're
looking for")
				continue
			}

			if considerInstanceProtection &&
(i.isProtectedFromScaleIn() || i.isProtectedFromTermination()) {
				debug.Println(a.name, "skipping protected
instance", *i.InstanceId)
				continue
			}

			if (availabilityZone != nil) && (*availabilityZone !=
*i.Placement.AvailabilityZone) {
				debug.Println(a.name, "skipping instance",
*i.InstanceId,
					"placed in a different AZ than what we're
looking for")
				continue
			}
			return i
		}
	}
	return nil
}

func (a *autoScalingGroup) getUnprotectedOnDemandInstanceInAZ(az *string)
*instance {
	return a.getInstance(az, true, true)
}
func (a *autoScalingGroup) getAnyUnprotectedOnDemandInstance() *instance {
	return a.getInstance(nil, true, true)
}

func (a *autoScalingGroup) getAnyOnDemandInstance() *instance {
	return a.getInstance(nil, true, false)
}

func (a *autoScalingGroup) getAnySpotInstance() *instance {
	return a.getInstance(nil, false, false)
}

func (a *autoScalingGroup) hasMemberInstance(inst *instance) bool {
	for _, member := range a.Instances {
		if *member.InstanceId == *inst.InstanceId {
			return true
		}
	}
	return false
}

func (a *autoScalingGroup) findUnattachedInstanceLaunchedForThisASG()
*instance {
	for inst := range a.region.instances.instances() {
		for _, tag := range inst.Tags {
			if *tag.Key == "launched-for-asg" && *tag.Value == a.name
{
				if !a.hasMemberInstance(inst) {
					return inst
				}
			}
		}
	}
	return nil
}

func (a *autoScalingGroup) getAllowedInstanceTypes(baseInstance *instance)
[]string {
	var allowedInstanceTypesTag string

	// By default take the command line parameter
	allowed := strings.Replace(a.region.conf.AllowedInstanceTypes, " ",
",", -1)

	// Check option of allowed instance types
	// If we have that option we don't need to calculate the compatible
instance type.
	if tagValue := a.getTagValue(AllowedInstanceTypesTag); tagValue !=
nil {
		allowedInstanceTypesTag = strings.Replace(*tagValue, " ", ",",
-1)
	}

	// ASG Tag config has a priority to override
	if allowedInstanceTypesTag != "" {
		allowed = allowedInstanceTypesTag
	}

	if allowed == "current" {
		return []string{baseInstance.typeInfo.instanceType}
	}

	// Simple trick to avoid returning list with empty elements
	return strings.FieldsFunc(allowed, func(c rune) bool {
		return c == ','
	})
}

func (a *autoScalingGroup) getDisallowedInstanceTypes(baseInstance
*instance) []string {
	var disallowedInstanceTypesTag string
	// By default take the command line parameter
	disallowed := strings.Replace(a.region.conf.DisallowedInstanceTypes,
" ", ",", -1)

	// Check option of disallowed instance types
	// If we have that option we don't need to calculate the compatible
instance type.
	if tagValue := a.getTagValue(DisallowedInstanceTypesTag); tagValue
!= nil {
		disallowedInstanceTypesTag = strings.Replace(*tagValue, " ",
",", -1)
	}

	// ASG Tag config has a priority to override
	if disallowedInstanceTypesTag != "" {
		disallowed = disallowedInstanceTypesTag
	}

	// Simple trick to avoid returning list with empty elements
	return strings.FieldsFunc(disallowed, func(c rune) bool {
		return c == ','
	})
}

func (a *autoScalingGroup) setAutoScalingMaxSize(maxSize int64) error {
	svc := a.region.services.autoScaling

	_, err := svc.UpdateAutoScalingGroup(
		&autoscaling.UpdateAutoScalingGroupInput{
			AutoScalingGroupName: aws.String(a.name),
			MaxSize:              aws.Int64(maxSize),
		})

	if err != nil {
		// Print the error, cast err to awserr.Error to get the Code
and
		// Message from an error.
		logger.Println(err.Error())
		return err
	}
	return nil
}

func (a *autoScalingGroup) attachSpotInstance(spotInstanceID string) error
{

	svc := a.region.services.autoScaling

	params := autoscaling.AttachInstancesInput{
		AutoScalingGroupName: aws.String(a.name),
		InstanceIds: []*string{
			&spotInstanceID,
		},
	}
	resp, err := svc.AttachInstances(&params)

	if err != nil {
		logger.Println(err.Error())
		// Pretty-print the response data.
		logger.Println(resp)
		return err
	}
	return nil
}

// Terminates an on-demand instance from the group,
// but only after it was detached from the autoscaling group
func (a *autoScalingGroup) detachAndTerminateOnDemandInstance(
	instanceID *string) error {
	logger.Println(a.region.name,
		a.name,
		"Detaching and terminating instance:",
		*instanceID)
	// detach the on-demand instance
	detachParams := autoscaling.DetachInstancesInput{
		AutoScalingGroupName: aws.String(a.name),
		InstanceIds: []*string{
			instanceID,
		},
		ShouldDecrementDesiredCapacity: aws.Bool(true),
	}

	asSvc := a.region.services.autoScaling

	if _, err := asSvc.DetachInstances(&detachParams); err != nil {
		logger.Println(err.Error())
		return err
	}

	// Wait till detachment initialize is complete before terminate
instance
	time.Sleep(20 * time.Second * a.region.conf.SleepMultiplier)

	return a.instances.get(*instanceID).terminate()
}

// Terminates an on-demand instance from the group using the
// TerminateInstanceInAutoScalingGroup api call.
func (a *autoScalingGroup) terminateInstanceInAutoScalingGroup(
	instanceID *string) error {
	logger.Println(a.region.name,
		a.name,
		"Terminating instance:",
		*instanceID)
	// terminate the on-demand instance
	terminateParams :=
autoscaling.TerminateInstanceInAutoScalingGroupInput{
		InstanceId:                      instanceID,
		ShouldDecrementDesiredCapacity: aws.Bool(true),
	}

	asSvc := a.region.services.autoScaling
	if _, err :=
asSvc.TerminateInstanceInAutoScalingGroup(&terminateParams); err != nil {
		logger.Println(err.Error())
		return err
	}

	return nil
}

// Counts the number of already running instances on-demand or spot, in
any or a specific AZ.
func (a *autoScalingGroup) alreadyRunningInstanceCount(
	spot bool, availabilityZone string) (int64, int64) {

	var total, count int64
	instanceCategory := "spot"

	if !spot {
		instanceCategory = "on-demand"
	}
	logger.Println(a.name, "Counting already running on demand instances
")
	for inst := range a.instances.instances() {
		if *inst.Instance.State.Name == "running" {
			// Count running Spot instances
			if spot && inst.isSpot() &&
				(*inst.Placement.AvailabilityZone ==
availabilityZone || availabilityZone == "") {
				count++
				// Count running OnDemand instances
			} else if !spot && !inst.isSpot() &&
				(*inst.Placement.AvailabilityZone ==
availabilityZone || availabilityZone == "") {
				count++
			}
			// Count total running instances
			total++
		}
	}
	logger.Println(a.name, "Found", count, instanceCategory, "instances
running on a total of", total)
	return count, total
}
```

```
package autospotting

import (
	"fmt"
	"math"
	"path/filepath"
	"sort"
	"strconv"
	"strings"
	"sync"
	"time"

	"github.com/aws/aws-sdk-go/aws"
	"github.com/aws/aws-sdk-go/service/ec2"
	"github.com/davecgh/go-spew/spew"
)

// The key in this map is the instance ID, useful for quick retrieval of
// instance attributes.
type instanceMap map[string]*instance

type instanceManager struct {
	sync.RWMutex
	catalog instanceMap
}

type instances interface {
	add(inst *instance)
	get(string) *instance
	count() int
	count64() int64
	make()
	instances() <-chan *instance
	dump() string
}

func makeInstances() instances {
	return &instanceManager{catalog: instanceMap{}}
}

func makeInstancesWithCatalog(catalog instanceMap) instances {
	return &instanceManager{catalog: catalog}
}

func (is *instanceManager) dump() string {
	is.RLock()
	defer is.RUnlock()
	return spew.Sdump(is.catalog)
}
func (is *instanceManager) make() {
	is.Lock()
	is.catalog = make(instanceMap)
	is.Unlock()
}
```

```
func (is *instanceManager) add(inst *instance) {
	if inst == nil {
		return
	}
	debug.Println(inst)
	is.Lock()
	defer is.Unlock()
	is.catalog[*inst.InstanceId] = inst
}

func (is *instanceManager) get(id string) (inst *instance) {
	is.RLock()
	defer is.RUnlock()
	return is.catalog[id]
}

func (is *instanceManager) count() int {
	is.RLock()
	defer is.RUnlock()

	return len(is.catalog)
}

func (is *instanceManager) count64() int64 {
	return int64(is.count())
}

func (is *instanceManager) instances() <-chan *instance {
	retC := make(chan *instance)
	go func() {
		is.RLock()
		defer is.RUnlock()
		defer close(retC)
		for _, i := range is.catalog {
			retC <- i
		}
	}()

	return retC
}

type instance struct {
	*ec2.Instance
	typeInfo  instanceTypeInformation
	price     float64
	region    *region
	protected bool
	asg       *autoScalingGroup
}

type acceptableInstance struct {
	instanceTI instanceTypeInformation
	price      float64
```

```
}

type instanceTypeInformation struct {
      instanceType             string
      vCPU                     int
      PhysicalProcessor        string
      GPU                      int
      pricing                  prices
      memory                   float32
      virtualizationTypes      []string
      hasInstanceStore         bool
      instanceStoreDeviceSize  float32
      instanceStoreDeviceCount int
      instanceStoreIsSSD       bool
      hasEBSOptimization       bool
      EBSThroughput            float32
}

func (i *instance) calculatePrice(spotCandidate instanceTypeInformation)
float64 {
      spotPrice :=
spotCandidate.pricing.spot[*i.Placement.AvailabilityZone]
      debug.Println("Comparing price spot/instance:")

      if i.EbsOptimized != nil && *i.EbsOptimized {
            spotPrice += spotCandidate.pricing.ebsSurcharge
            debug.Println("\tEBS Surcharge : ",
spotCandidate.pricing.ebsSurcharge)
      }

      debug.Println("\tSpot price: ", spotPrice)
      debug.Println("\tInstance price: ", i.price)
      return spotPrice
}

func (i *instance) isSpot() bool {
      return i.InstanceLifecycle != nil &&
            *i.InstanceLifecycle == "spot"
}

func (i *instance) isProtectedFromTermination() bool {

      // determine and set the API termination protection field
      diaRes, err := i.region.services.ec2.DescribeInstanceAttribute(
            &ec2.DescribeInstanceAttributeInput{
                  Attribute:  aws.String("disableApiTermination"),
                  InstanceId: i.InstanceId,
            })

      if err == nil &&
            diaRes.DisableApiTermination != nil &&
            diaRes.DisableApiTermination.Value != nil &&
            *diaRes.DisableApiTermination.Value {
```

```
            logger.Printf("\t: %v Instance, %v is protected from
termination\n",
                  *i.Placement.AvailabilityZone, *i.InstanceId)
            return true
      }
      return false
}

func (i *instance) isProtectedFromScaleIn() bool {
      if i.asg == nil {
            return false
      }

      for _, inst := range i.asg.Instances {
            if *inst.InstanceId == *i.InstanceId &&
                  *inst.ProtectedFromScaleIn {
                  logger.Printf("\t: %v Instance, %v is protected from
scale-in\n",
                        *inst.AvailabilityZone,
                        *inst.InstanceId)
                  return true
            }
      }
      return false
}

func (i *instance) canTerminate() bool {
      return *i.State.Name != ec2.InstanceStateNameTerminated &&
            *i.State.Name != ec2.InstanceStateNameShuttingDown
}

func (i *instance) terminate() error {
      svc := i.region.services.ec2
      if i.canTerminate() {
            _, err := svc.TerminateInstances(&ec2.TerminateInstancesInput{
                  InstanceIds: []*string{i.InstanceId},
            })
            if err != nil {
                  logger.Printf("Issue while terminating %v: %v",
*i.InstanceId, err.Error())
                  return err
            }
      }
      return nil
}

func (i *instance) isPriceCompatible(spotPrice float64) bool {
      if spotPrice == 0 {
            logger.Printf("\tUnavailable in this Availability Zone")
            return false
      }

      if spotPrice <= i.price {
            return true
```

```
	}

	logger.Printf("\tNot price compatible")
	return false
}

func (i *instance) isClassCompatible(spotCandidate
instanceTypeInformation) bool {
	current := i.typeInfo

	debug.Println("Comparing class spot/instance:")
	debug.Println("\tSpot CPU/memory/GPU: ", spotCandidate.vCPU,
		" / ", spotCandidate.memory, " / ", spotCandidate.GPU)
	debug.Println("\tInstance CPU/memory/GPU: ", current.vCPU,
		" / ", current.memory, " / ", current.GPU)

	if i.isSameArch(spotCandidate) &&
		spotCandidate.vCPU >= current.vCPU &&
		spotCandidate.memory >= current.memory &&
		spotCandidate.GPU >= current.GPU {
		return true
	}
	logger.Println("\tNot class compatible (CPU/memory/GPU)")
	return false
}

func (i *instance) isSameArch(other instanceTypeInformation) bool {
	thisCPU := i.typeInfo.PhysicalProcessor
	otherCPU := other.PhysicalProcessor

	ret := (isIntelCompatible(thisCPU) && isIntelCompatible(otherCPU))
||
		(isARM(thisCPU) && isARM(otherCPU))

	if !ret {
		logger.Println("\tInstance CPU architecture mismatch, current
CPU architecture",
			thisCPU, "is incompatible with candidate CPU
architecture", otherCPU)
	}
	return ret
}

func isIntelCompatible(cpuName string) bool {
	return isIntel(cpuName) || isAMD(cpuName)
}

func isIntel(cpuName string) bool {
	// t1.micro seems to be the only one to have this set to 'Variable'
	return strings.Contains(cpuName, "Intel") ||
strings.Contains(cpuName, "Variable")
}

func isAMD(cpuName string) bool {
	return strings.Contains(cpuName, "AMD")
}

func isARM(cpuName string) bool {
	// The ARM chips are so far all called "AWS Graviton Processor"
	return strings.Contains(cpuName, "AWS")
}

func (i *instance) isEBSCompatible(spotCandidate instanceTypeInformation)
bool {
	if spotCandidate.EBSThroughput < i.typeInfo.EBSThroughput {
		logger.Println("\tEBS throughput insufficient:",
spotCandidate.EBSThroughput, "<", i.typeInfo.EBSThroughput)
		return false
	}
	return true
}

// Here we check the storage compatibility, with the following evaluation
// criteria:
// - speed: don't accept spinning disks when we used to have SSDs
// - number of volumes: the new instance should have enough volumes to be
//   able to attach all the instance store device mappings defined on the
//   original instance
// - volume size: each of the volumes should be at least as big as the
//   original instance's volumes
func (i *instance) isStorageCompatible(spotCandidate
instanceTypeInformation, attachedVolumes int) bool {
	existing := i.typeInfo

	debug.Println("Comparing storage spot/instance:")
	debug.Println("\tSpot volumes/size/ssd: ",
		spotCandidate.instanceStoreDeviceCount,
		spotCandidate.instanceStoreDeviceSize,
		spotCandidate.instanceStoreIsSSD)
	debug.Println("\tInstance volumes/size/ssd: ",
		attachedVolumes,
		existing.instanceStoreDeviceSize,
		existing.instanceStoreIsSSD)

	if attachedVolumes == 0 ||
		(spotCandidate.instanceStoreDeviceCount >= attachedVolumes &&
			spotCandidate.instanceStoreDeviceSize >=
existing.instanceStoreDeviceSize &&
			(spotCandidate.instanceStoreIsSSD ||
				spotCandidate.instanceStoreIsSSD ==
existing.instanceStoreIsSSD)) {
		return true
	}
	logger.Println("\tNot storage compatible")
	return false
}
```

```
func (i *instance) isVirtualizationCompatible(spotVirtualizationTypes
[]string) bool {
	current := *i.VirtualizationType
	if len(spotVirtualizationTypes) == 0 {
		spotVirtualizationTypes = []string{"HVM"}
	}
	debug.Println("Comparing virtualization spot/instance:")
	debug.Println("\tSpot virtualization: ", spotVirtualizationTypes)
	debug.Println("\tInstance virtualization: ", current)

	for _, avt := range spotVirtualizationTypes {
		if (avt == "PV") && (current == "paravirtual") ||
			(avt == "HVM") && (current == "hvm") {
			return true
		}
	}
	logger.Println("\tNot virtualization compatible")
	return false
}

func (i *instance) isAllowed(instanceType string, allowedList []string,
disallowedList []string) bool {
	debug.Println("Checking allowed/disallowed list")

	if len(allowedList) > 0 {
		for _, a := range allowedList {
			if match, _ := filepath.Match(a, instanceType); match {
				return true
			}
		}
		logger.Println("\tNot in the list of allowed instance types")
		return false
	} else if len(disallowedList) > 0 {
		for _, a := range disallowedList {
			// glob matching
			if match, _ := filepath.Match(a, instanceType); match {
				logger.Println("\tIn the list of disallowed
instance types")
				return false
			}
		}
	}
	return true
}

func (i *instance)
getCompatibleSpotInstanceTypesListSortedAscendingByPrice(allowedList
[]string,
	disallowedList []string) ([]instanceTypeInformation, error) {
	current := i.typeInfo
	var acceptableInstanceTypes []acceptableInstance

	// Count the ephemeral volumes attached to the original instance's
block
```

```
	// device mappings, this number is used later when comparing with
each
	// instance type.

	usedMappings :=
i.asg.launchConfiguration.countLaunchConfigEphemeralVolumes()
	attachedVolumesNumber := min(usedMappings,
current.instanceStoreDeviceCount)

	// Iterate alphabetically by instance type
	keys := make([]string, 0)
	for k := range i.region.instanceTypeInformation {
		keys = append(keys, k)
	}
	sort.Strings(keys)

	// Find all compatible and not blocked instance types
	for _, k := range keys {
		candidate := i.region.instanceTypeInformation[k]

		candidatePrice := i.calculatePrice(candidate)
		logger.Println("Comparing current type", current.instanceType,
"with price", i.price,
			"with candidate", candidate.instanceType, "with price",
candidatePrice)

		if i.isAllowed(candidate.instanceType, allowedList,
disallowedList) &&
			i.isPriceCompatible(candidatePrice) &&
			i.isEBSCompatible(candidate) &&
			i.isClassCompatible(candidate) &&
			i.isStorageCompatible(candidate, attachedVolumesNumber)
&&

	i.isVirtualizationCompatible(candidate.virtualizationTypes) {
			acceptableInstanceTypes = append(acceptableInstanceTypes,
acceptableInstance{candidate, candidatePrice})
			logger.Println("\tMATCH FOUND, added",
candidate.instanceType, "to launch candiates list")
		} else if candidate.instanceType != "" {
			debug.Println("Non compatible option found:",
candidate.instanceType, "at", candidatePrice, " - discarding")
		}
	}

	if acceptableInstanceTypes != nil {
		sort.Slice(acceptableInstanceTypes, func(i, j int) bool {
			return acceptableInstanceTypes[i].price <
acceptableInstanceTypes[j].price
		})
		debug.Println("List of cheapest compatible spot instances
found, sorted ascending by price: ",
			acceptableInstanceTypes)
		var result []instanceTypeInformation
```

```
            for _, ai := range acceptableInstanceTypes {
                  result = append(result, ai.instanceTI)
            }
            return result, nil
      }

      return nil, fmt.Errorf("No cheaper spot instance types could be
found")
}

func (i *instance) launchSpotReplacement() error {
      instanceTypes, err :=
i.getCompatibleSpotInstanceTypesListSortedAscendingByPrice(
            i.asg.getAllowedInstanceTypes(i),
            i.asg.getDisallowedInstanceTypes(i))

      if err != nil {
            logger.Println("Couldn't determine the cheapest compatible
spot instance type")
            return err
      }

      //Go through all compatible instances until one type launches or we
are out of options.
      for _, instanceType := range instanceTypes {
            az := *i.Placement.AvailabilityZone
            bidPrice := i.getPricetoBid(i.price,
                  instanceType.pricing.spot[az])

            runInstancesInput :=
i.createRunInstancesInput(instanceType.instanceType, bidPrice)
            logger.Println(az, i.asg.name, "Launching spot instance of
type", instanceType.instanceType, "with bid price", bidPrice)
            logger.Println(az, i.asg.name)
            resp, err :=
i.region.services.ec2.RunInstances(runInstancesInput)

            if err != nil {
                  if strings.Contains(err.Error(),
"InsufficientInstanceCapacity") {
                        logger.Println("Couldn't launch spot instance due
to lack of capcity, trying next instance type:", err.Error())
                  } else {
                        logger.Println("Couldn't launch spot instance:",
err.Error(), "trying next instance type")
                        debug.Println(runInstancesInput)
                  }
            } else {
                  spotInst := resp.Instances[0]
                  logger.Println(i.asg.name, "Successfully launched spot
instance", *spotInst.InstanceId,
                        "of type", *spotInst.InstanceType,
                        "with bid price", bidPrice,
                    "current spot price",
instanceType.pricing.spot[az])

                debug.Println("RunInstances response:", spew.Sdump(resp))
                return nil
            }
        }

        logger.Println(i.asg.name, "Exhausted all compatible instance types
without launch success. Aborting.")
        return err
}

func (i *instance) getPricetoBid(
        baseOnDemandPrice float64, currentSpotPrice float64) float64 {

        logger.Println("BiddingPolicy: ", i.region.conf.BiddingPolicy)

        if i.region.conf.BiddingPolicy == DefaultBiddingPolicy {
            logger.Println("Bidding base on demand price",
baseOnDemandPrice)
            return baseOnDemandPrice
        }

        bufferPrice := math.Min(baseOnDemandPrice,
currentSpotPrice*(1.0+i.region.conf.SpotPriceBufferPercentage/100.0))
        logger.Println("Bidding buffer-based price", bufferPrice)
        return bufferPrice
}

func (i *instance) convertBlockDeviceMappings(lc *launchConfiguration)
[]*ec2.BlockDeviceMapping {
        bds := []*ec2.BlockDeviceMapping{}
        if lc == nil || len(lc.BlockDeviceMappings) == 0 {
            debug.Println("Missing block device mappings")
            return bds
        }

        for _, lcBDM := range lc.BlockDeviceMappings {

            ec2BDM := &ec2.BlockDeviceMapping{
                DeviceName:  lcBDM.DeviceName,
                VirtualName: lcBDM.VirtualName,
            }

            if lcBDM.Ebs != nil {
                ec2BDM.Ebs = &ec2.EbsBlockDevice{
                    DeleteOnTermination: lcBDM.Ebs.DeleteOnTermination,
                    Encrypted:           lcBDM.Ebs.Encrypted,
                    Iops:                lcBDM.Ebs.Iops,
                    SnapshotId:          lcBDM.Ebs.SnapshotId,
                    VolumeSize:          lcBDM.Ebs.VolumeSize,
                    VolumeType:          lcBDM.Ebs.VolumeType,
                }
```

```
		}

		// handle the noDevice field directly by skipping the device
if set to true
		if lcBDM.NoDevice != nil && *lcBDM.NoDevice {
			continue
		}
		bds = append(bds, ec2BDM)

	}
	return bds
}

func (i *instance) convertSecurityGroups() []*string {
	groupIDs := []*string{}
	for _, sg := range i.SecurityGroups {
		groupIDs = append(groupIDs, sg.GroupId)
	}
	return groupIDs
}

func (i *instance) launchTemplateHasNetworkInterfaces(id, ver *string)
(bool, []*ec2.LaunchTemplateInstanceNetworkInterfaceSpecification) {
	res, err := i.region.services.ec2.DescribeLaunchTemplateVersions(
		&ec2.DescribeLaunchTemplateVersionsInput{
			Versions:         []*string{ver},
			LaunchTemplateId: id,
		},
	)

	if err != nil {
		logger.Println("Failed to describe launch template", *id,
"version", *ver,
			"encountered error:", err.Error())
	}

	if err == nil && len(res.LaunchTemplateVersions) == 1 {
		lt := res.LaunchTemplateVersions[0]
		nis := lt.LaunchTemplateData.NetworkInterfaces
		if len(nis) > 0 {
			return true, nis
		}
	}
	return false, nil
}

func (i *instance) createRunInstancesInput(instanceType string, price
float64) *ec2.RunInstancesInput {
	var retval ec2.RunInstancesInput

	// information we must (or can safely) copy/convert from the
currently running
	// on-demand instance or we had to compute in order to place the
spot bid
```

```
      retval = ec2.RunInstancesInput{

            EbsOptimized: i.EbsOptimized,

            InstanceMarketOptions: &ec2.InstanceMarketOptionsRequest{
                  MarketType: aws.String("spot"),
                  SpotOptions: &ec2.SpotMarketOptions{
                        MaxPrice: aws.String(strconv.FormatFloat(price,
'g', 10, 64)),
                  },
            },

            InstanceType: aws.String(instanceType),
            MaxCount:     aws.Int64(1),
            MinCount:     aws.Int64(1),

            Placement: i.Placement,

            SecurityGroupIds: i.convertSecurityGroups(),

            SubnetId:          i.SubnetId,
            TagSpecifications: i.generateTagsList(),
      }

      if i.asg.LaunchTemplate != nil {
            ver := i.asg.LaunchTemplate.Version
            id := i.asg.LaunchTemplate.LaunchTemplateId

            retval.LaunchTemplate = &ec2.LaunchTemplateSpecification{
                  LaunchTemplateId: id,
                  Version:          ver,
            }

            if having, nis := i.launchTemplateHasNetworkInterfaces(id,
ver); having {
                  for _, ni := range nis {
                        retval.NetworkInterfaces =
append(retval.NetworkInterfaces,
                              &ec2.InstanceNetworkInterfaceSpecification{
                                    AssociatePublicIpAddress:
ni.AssociatePublicIpAddress,
                                    SubnetId:                 i.SubnetId,
                                    DeviceIndex:
ni.DeviceIndex,
                                    Groups:
i.convertSecurityGroups(),
                              },
                        )
                  }
                  retval.SubnetId, retval.SecurityGroupIds = nil, nil
            }
      }

      if i.asg.launchConfiguration != nil {
            lc := i.asg.launchConfiguration

            if lc.KeyName != nil && *lc.KeyName != "" {
                  retval.KeyName = lc.KeyName
            }

            if lc.IamInstanceProfile != nil {
                  if strings.HasPrefix(*lc.IamInstanceProfile,
"arn:aws:iam:") {
                        retval.IamInstanceProfile =
&ec2.IamInstanceProfileSpecification{
                              Arn: lc.IamInstanceProfile,
                        }
                  } else {
                        retval.IamInstanceProfile =
&ec2.IamInstanceProfileSpecification{
                              Name: lc.IamInstanceProfile,
                        }
                  }
            }
            retval.ImageId = lc.ImageId

            retval.UserData = lc.UserData

            BDMs := i.convertBlockDeviceMappings(lc)

            if len(BDMs) > 0 {
                  retval.BlockDeviceMappings = BDMs
            }

            if lc.InstanceMonitoring != nil {
                  retval.Monitoring = &ec2.RunInstancesMonitoringEnabled{
                        Enabled: lc.InstanceMonitoring.Enabled}
            }

            if lc.AssociatePublicIpAddress != nil || i.SubnetId != nil {
                  // Instances are running in a VPC.
                  retval.NetworkInterfaces =
[]*ec2.InstanceNetworkInterfaceSpecification{
                        {
                              AssociatePublicIpAddress:
lc.AssociatePublicIpAddress,
                              DeviceIndex:              aws.Int64(0),
                              SubnetId:                 i.SubnetId,
                              Groups:
i.convertSecurityGroups(),
                        },
                  }
                  retval.SubnetId, retval.SecurityGroupIds = nil, nil
            }
      }

      return &retval
}
```

```
func (i *instance) generateTagsList() []*ec2.TagSpecification {
	tags := ec2.TagSpecification{
		ResourceType: aws.String("instance"),
		Tags: []*ec2.Tag{
			{
				Key:   aws.String("launched-by-autospotting"),
				Value: aws.String("true"),
			},
			{
				Key:   aws.String("launched-for-asg"),
				Value: aws.String(i.asg.name),
			},
		},
	}

	if i.asg.LaunchTemplate != nil {
		tags.Tags = append(tags.Tags, &ec2.Tag{
			Key:   aws.String("LaunchTemplateID"),
			Value: i.asg.LaunchTemplate.LaunchTemplateId,
		})
		tags.Tags = append(tags.Tags, &ec2.Tag{
			Key:   aws.String("LaunchTemplateVersion"),
			Value: i.asg.LaunchTemplate.Version,
		})
	} else if i.asg.LaunchConfigurationName != nil {
		tags.Tags = append(tags.Tags, &ec2.Tag{
			Key:   aws.String("LaunchConfigurationName"),
			Value: i.asg.LaunchConfigurationName,
		})
	}

	for _, tag := range i.Tags {
		if !strings.HasPrefix(*tag.Key, "aws:") &&
			*tag.Key != "launched-by-autospotting" &&
			*tag.Key != "launched-for-asg" &&
			*tag.Key != "LaunchTemplateID" &&
			*tag.Key != "LaunchTemplateVersion" &&
			*tag.Key != "LaunchConfiguationName" {
			tags.Tags = append(tags.Tags, tag)
		}
	}
	return []*ec2.TagSpecification{&tags}
}

// returns an instance ID as *string, set to nil if we need to wait for
the next
// run in case there are no spot instances
func (i *instance) isReadyToAttach(asg *autoScalingGroup) bool {

	logger.Println("Considering ", *i.InstanceId, "for attaching to",
asg.name)

	gracePeriod := *asg.HealthCheckGracePeriod
```

```
      instanceUpTime := time.Now().Unix() - i.LaunchTime.Unix()

      logger.Println("Instance uptime:",
time.Duration(instanceUpTime)*time.Second)

      // Check if the spot instance is out of the grace period, so in that
case we
      // can replace an on-demand instance with it
      if *i.State.Name == ec2.InstanceStateNameRunning &&
            instanceUpTime > gracePeriod {
            logger.Println("The spot instance", *i.InstanceId,
                  " has passed grace period and is ready to attach to the
group.")
            return true
      } else if *i.State.Name == ec2.InstanceStateNameRunning &&
            instanceUpTime < gracePeriod {
            logger.Println("The spot instance", *i.InstanceId,
                  "is still in the grace period,",
                  "waiting for it to be ready before we can attach it to
the group...")
            return false
      } else if *i.State.Name == ec2.InstanceStateNamePending {
            logger.Println("The spot instance", *i.InstanceId,
                  "is still pending,",
                  "waiting for it to be running before we can attach it to
the group...")
            return false
      }
      return false
}

// Why the heck isn't this in the Go standard library?
func min(x, y int) int {
      if x < y {
            return x
      }
      return y
}
```

```
package autospotting
```

```
import (
	"io/ioutil"
	"log"
	"os"
	"strings"
	"sync"

	"github.com/aws/aws-sdk-go/aws"
	"github.com/aws/aws-sdk-go/aws/session"
	"github.com/aws/aws-sdk-go/service/ec2"
	"github.com/aws/aws-sdk-go/service/ec2/ec2iface"
)

var logger, debug *log.Logger

var hourlySavings float64
var savingsMutex = &sync.RWMutex{}

// Run starts processing all AWS regions looking for AutoScaling groups
// enabled and taking action by replacing more pricy on-demand instances
with
// compatible and cheaper spot instances.
func Run(cfg *Config) {

	setupLogging(cfg)

	debug.Println(*cfg)

	// use this only to list all the other regions
	ec2Conn := connectEC2(cfg.MainRegion)

	addDefaultFilteringMode(cfg)
	addDefaultFilter(cfg)

	allRegions, err := getRegions(ec2Conn)

	if err != nil {
		logger.Println(err.Error())
		return
	}

	processRegions(allRegions, cfg)

}

func addDefaultFilteringMode(cfg *Config) {
	if cfg.TagFilteringMode != "opt-out" {
		debug.Printf("Configured filtering mode: '%s', considering it
as 'opt-in'(default)\n",
			cfg.TagFilteringMode)
		cfg.TagFilteringMode = "opt-in"
	} else {
		debug.Println("Configured filtering mode: 'opt-out'")
	}
```

```
}

func addDefaultFilter(cfg *Config) {
	if len(strings.TrimSpace(cfg.FilterByTags)) == 0 {
		switch cfg.TagFilteringMode {
		case "opt-out":
			cfg.FilterByTags = "spot-enabled=false"
		default:
			cfg.FilterByTags = "spot-enabled=true"
		}
	}
}

func disableLogging() {
	setupLogging(&Config{LogFile: ioutil.Discard})
}

func setupLogging(cfg *Config) {
	logger = log.New(cfg.LogFile, "", cfg.LogFlag)

	if os.Getenv("AUTOSPOTTING_DEBUG") == "true" {
		debug = log.New(cfg.LogFile, "", cfg.LogFlag)
	} else {
		debug = log.New(ioutil.Discard, "", 0)
	}

}

// processAllRegions iterates all regions in parallel, and replaces
instances
// for each of the ASGs tagged with tags as specified by slice represented
by cfg.FilterByTags
// by default this is all asg with the tag 'spot-enabled=true'.
func processRegions(regions []string, cfg *Config) {

	var wg sync.WaitGroup

	for _, r := range regions {

		wg.Add(1)
		r := region{name: r, conf: cfg}

		go func() {

			if r.enabled() {
				logger.Printf("Enabled to run in %s, processing
region.\n", r.name)
				r.processRegion()
			} else {
				debug.Println("Not enabled to run in", r.name)
				debug.Println("List of enabled regions:",
cfg.Regions)
			}
```

```
                        wg.Done()
                }()
        }
        wg.Wait()
}

func connectEC2(region string) *ec2.EC2 {

        sess, err := session.NewSession()
        if err != nil {
                panic(err)
        }

        return ec2.New(sess,
                aws.NewConfig().WithRegion(region))
}

// getRegions generates a list of AWS regions.
func getRegions(ec2conn ec2iface.EC2API) ([]string, error) {
        var output []string

        logger.Println("Scanning for available AWS regions")

        resp, err := ec2conn.DescribeRegions(&ec2.DescribeRegionsInput{})

        if err != nil {
                logger.Println(err.Error())
                return nil, err
        }

        debug.Println(resp)

        for _, r := range resp.Regions {

                if r != nil && r.RegionName != nil {
                        debug.Println("Found region", *r.RegionName)
                        output = append(output, *r.RegionName)
                }
        }
        return output, nil
}
```

```
package autospotting
```

```
import (
	"time"

	"github.com/aws/aws-sdk-go/aws"
	"github.com/aws/aws-sdk-go/service/ec2"
)

type spotPrices struct {
	data []*ec2.SpotPrice
	conn connections
}

// fetch queries all spot prices in the current region
func (s *spotPrices) fetch(product string,
	duration time.Duration,
	availabilityZone *string,
	instanceTypes []*string) error {

	logger.Println(s.conn.region, "Requesting spot prices")

	ec2Conn := s.conn.ec2
	params := &ec2.DescribeSpotPriceHistoryInput{
		ProductDescriptions: []*string{
			aws.String(product),
		},
		StartTime:        aws.Time(time.Now().Add(-1 * duration)),
		EndTime:          aws.Time(time.Now()),
		AvailabilityZone: availabilityZone,
		InstanceTypes:    instanceTypes,
	}

	resp, err := ec2Conn.DescribeSpotPriceHistory(params)

	if err != nil {
		logger.Println(s.conn.region, "Failed requesting spot
prices:", err.Error())
		return err
	}

	s.data = resp.SpotPriceHistory

	return nil
}
```

```
package autospotting
```

```
import (
	"encoding/json"
	"errors"

	"github.com/aws/aws-lambda-go/events"
	"github.com/aws/aws-sdk-go/aws"
	"github.com/aws/aws-sdk-go/aws/session"
	"github.com/aws/aws-sdk-go/service/autoscaling"
	"github.com/aws/aws-sdk-go/service/autoscaling/autoscalingiface"
	"github.com/aws/aws-sdk-go/service/ec2"
	"github.com/aws/aws-sdk-go/service/ec2/ec2iface"
)

const (
	// DefaultTerminationNotificationAction is the default value for the
termination notification
	// action configuration option
	DefaultTerminationNotificationAction =
AutoTerminationNotificationAction
)

//SpotTermination is used to detach an instance, used when a spot instance
is due for termination
type SpotTermination struct {
	asSvc  autoscalingiface.AutoScalingAPI
	ec2Svc ec2iface.EC2API
}

//InstanceData represents JSON structure of the Detail property of
CloudWatch event when a spot instance is terminated
//Reference = https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-
interruptions.html#spot-instance-termination-notices
type instanceData struct {
	InstanceID     string `json:"instance-id"`
	InstanceAction string `json:"instance-action"`
}

//NewSpotTermination is a constructor for creating an instance of
spotTermination to call DetachInstance
func NewSpotTermination(region string) SpotTermination {

	logger.Println("Connection to region ", region)

	session := session.Must(
		session.NewSession(&aws.Config{Region: aws.String(region)}))

	return SpotTermination{

		asSvc:  autoscaling.New(session),
		ec2Svc: ec2.New(session),
	}
}
```

```
//GetInstanceIDDueForTermination checks if the given CloudWatch event data
is triggered from a spot termination
//If it is a termination event for a spot instance, it returns the
instance id present in the event data
func GetInstanceIDDueForTermination(event events.CloudWatchEvent)
(*string, error) {

	var detailData instanceData
	if err := json.Unmarshal(event.Detail, &detailData); err != nil {
		logger.Println(err.Error())
		return nil, err
	}

	if detailData.InstanceAction != "" {
		return &detailData.InstanceID, nil
	}

	return nil, nil
}

//DetachInstance detaches the instance from autoscaling group without
decrementing the desired capacity
//This makes sure that the autoscaling group spawns a new instance as soon
as this instance is detached
func (s *SpotTermination) detachInstance(instanceID *string, asgName
string) error {

	logger.Println(asgName,
		"Detaching instance:",
		*instanceID)

	detachParams := autoscaling.DetachInstancesInput{
		AutoScalingGroupName: aws.String(asgName),
		InstanceIds: []*string{
			instanceID,
		},
		ShouldDecrementDesiredCapacity: aws.Bool(false),
	}
	if _, detachErr := s.asSvc.DetachInstances(&detachParams); detachErr
!= nil {
		logger.Println(detachErr.Error())
		return detachErr
	}

	logger.Printf("Detached instance %s successfully", *instanceID)

	s.deleteTagInstanceLaunchedForAsg(instanceID)

	return nil
}

//TerminateInstance terminate the instance from autoscaling group without
decrementing the desired capacity
```

```
//This makes sure that any LifeCycle Hook configured is triggered and the
autoscaling group spawns a new instance
// as soon as this instance begin terminating.
func (s *SpotTermination) terminateInstance(instanceID *string, asgName
string) error {

      logger.Println(asgName,
            "Terminating instance:",
            *instanceID)
      // terminate the spot instance
      terminateParams :=
autoscaling.TerminateInstanceInAutoScalingGroupInput{
            InstanceId:                     instanceID,
            ShouldDecrementDesiredCapacity: aws.Bool(false),
      }

      if _, err :=
s.asSvc.TerminateInstanceInAutoScalingGroup(&terminateParams); err != nil
{
            logger.Println(err.Error())
            return err
      }
      return nil
}

func (s *SpotTermination) getAsgName(instanceID *string) (string, error) {
      asParams := autoscaling.DescribeAutoScalingInstancesInput{
            InstanceIds: []*string{instanceID},
      }

      result, err := s.asSvc.DescribeAutoScalingInstances(&asParams)

      var asgName = ""
      if err == nil {
            asgName = *result.AutoScalingInstances[0].AutoScalingGroupName
      }

      return asgName, err
}

// ExecuteAction execute the proper termination action (terminate|detach)
based on the value of
// terminationNotificationAction and the presence of a LifecycleHook on
ASG.
func (s *SpotTermination) ExecuteAction(instanceID *string,
terminationNotificationAction string) error {
      if s.asSvc == nil {
            return errors.New("AutoScaling service not defined. Please use
NewSpotTermination()")
      }

      asgName, err := s.getAsgName(instanceID)

      if err != nil {
            logger.Printf("Failed get ASG name for %s with err: %s\n",
*instanceID, err.Error())
            return err
      }

      switch terminationNotificationAction {
      case "detach":
            s.detachInstance(instanceID, asgName)
      case "terminate":
            s.terminateInstance(instanceID, asgName)
      default:
            if s.asgHasTerminationLifecycleHook(&asgName) {
                  s.terminateInstance(instanceID, asgName)
            } else {
                  s.detachInstance(instanceID, asgName)
            }
      }

      return nil
}

func (s *SpotTermination) deleteTagInstanceLaunchedForAsg(instanceID
*string) error {
      ec2Params := ec2.DeleteTagsInput{
            Resources: []*string{
                  aws.String(*instanceID),
            },
            Tags: []*ec2.Tag{
                  {
                        Key: aws.String("launched-for-asg"),
                  },
            },
      }
      _, err := s.ec2Svc.DeleteTags(&ec2Params)

      if err != nil {
            logger.Printf("Failed to delete Tag 'launched-for-asg' from
spot instance %s with err: %s\n", *instanceID, err.Error())
            return err
      }

      logger.Printf("Tag 'launched-for-asg' deleted from spot instance
%s", *instanceID)

      return nil
}

func (s *SpotTermination)
asgHasTerminationLifecycleHook(autoScalingGroupName *string) bool {
      asParams := autoscaling.DescribeLifecycleHooksInput{
            AutoScalingGroupName: autoScalingGroupName,
      }

      result, err := s.asSvc.DescribeLifecycleHooks(&asParams)
      if err != nil {
            logger.Println(err.Error())
            return false
      }

      var hasHook = false
      for _, lfh := range result.LifecycleHooks {
            if *lfh.LifecycleTransition ==
"autoscaling:EC2_INSTANCE_TERMINATING" {
                  hasHook = true
                  logger.Println("Found Hook", *lfh.LifecycleHookName)
                  break
            }
      }

      return hasHook
}
```

REFERENCE

References

1. Hu, Yazhou, Bo Deng, and Fuyang Peng. 2016. “Autoscaling Prediction Models for Cloud Resource Provisioning”. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*. IEEE. doi:10.1109/compcomm.2016.7924927.

2. PATIL, JAYENDRA. 2017. “AWS High Availability & Fault Tolerance Architecture – Certification”. http://jayendrapatil.com/aws-high-availability-fault-tolerance-architecture-certification/. http://jayendrapatil.com/aws-high-availability-fault-tolerance-architecture-certification/.

3. Jeff Barr, Attila Narin, and Jinesh Varia. 2011. “Building Fault-Tolerant Applications on AWS”. *Amazon Web Services Whitepapers*. https://media.amazonwebservices.com/AWS_Building_Fault_Tolerant_Applications.pdf.

4. Toosi, Adel Nadjaran, Farzad Khodadadi, and Rajkumar Buyya. 2015. “SipaaS: Spot Instance Pricing as a Service Framework and Its Implementation in OpenStack”. *Concurrency and Computation: Practice and Experience* 28 (13). Wiley-Blackwell: 3672–90. doi:10.1002/cpe.3749.

5. Urgaonkar, Bhuvan, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. 2008. “Agile Dynamic Provisioning of Multi-Tier Internet Applications”. *ACM Transactions on Autonomous and Adaptive Systems* 3 (1). Association for Computing Machinery (ACM): 1–39. doi:10.1145/1342171.1342172.

6. Jiang, Jing, Jie Lu, Guangquan Zhang, and Guodong Long. 2013. “Optimal Cloud Resource Auto-Scaling for Web Applications”. In *2013 13th IEEE/ACM International Symposium on Cluster Cloud, and Grid Computing*. IEEE. doi:10.1109/ccgrid.2013.73.

7. Han, Rui, Moustafa M. Ghanem, Li Guo, Yike Guo, and Michelle Osmond. 2014. “Enabling Cost-Aware and Adaptive Elasticity of Multi-Tier Cloud Applications”. *Future Generation Computer Systems* 32 (March). Elsevier BV: 82–98. doi:10.1016/j.future.2012.05.018.

8. Fernandez, Hector, Guillaume Pierre, and Thilo Kielmann. 2014. “Autoscaling Web Applications in Heterogeneous Cloud Infrastructures”. In *2014 IEEE International Conference on Cloud Engineering*. IEEE. doi:10.1109/ic2e.2014.25.

9. Chen, Tao, and Rami Bahsoon. 2015. “Toward a Smarter Cloud: Self-Aware Autoscaling of Cloud Configurations and Resources”. *Computer* 48 (9). Institute of Electrical and Electronics Engineers (IEEE): 93–96. doi:10.1109/mc.2015.278.

10. Vondra, T., and J. Šedivý. 2017. “Cloud Autoscaling Simulation Based on Queueing Network Model”. *Simulation Modelling Practice and Theory* 70 (January). Elsevier BV: 83–100. doi:10.1016/j.simpat.2016.10.005.

11. Tighe, Michael, and Michael Bauer. 2014. “Integrating Cloud Application Autoscaling with Dynamic VM Allocation”. In *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE. doi:10.1109/noms.2014.6838239.

12. Barrett, Enda, Enda Howley, and Jim Duggan. 2012. “Applying Reinforcement Learning towards Automating Resource Allocation and Application Scalability in the Cloud”. *Concurrency and Computation: Practice and Experience* 25 (12). Wiley-Blackwell: 1656–74. doi:10.1002/cpe.2864.

13. Bu, Xiangping, Jia Rao, and Cheng-Zhong Xu. 2013. “Coordinated Self-Configuration of Virtual Machines and Appliances Using a Model-Free Learning Approach”. *IEEE*

Transactions on Parallel and Distributed Systems 24 (4). Institute of Electrical and Electronics Engineers (IEEE): 681–90. doi:10.1109/tpds.2012.174.

14. Hirashima, Yoko, Kenta Yamasaki, and Masataka Nagura. 2016. "Proactive-Reactive Auto-Scaling Mechanism for Unpredictable Load Change". In *2016 5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*. IEEE. doi:10.1109/iiai-aai.2016.180.

15. Roy, Nilabja, Abhishek Dubey, and Aniruddha Gokhale. 2011. "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting". In *2011 IEEE 4th International Conference on Cloud Computing*. IEEE. doi:10.1109/cloud.2011.42.

16. Yang, Jingqi, Chuanchang Liu, Yanlei Shang, Zexiang Mao, and Junliang Chen. 2013. "Workload Predicting-Based Automatic Scaling in Service Clouds". In *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE. doi:10.1109/cloud.2013.146.

17. Caron, Eddy, Frédéric Desprez, and Adrian Muresan. 2011. "Pattern Matching Based Forecast of Non-Periodic Repetitive Behavior for Cloud Clients". *Journal of Grid Computing* 9 (1). Springer Nature: 49–64. doi:10.1007/s10723-010-9178-4.

18. Islam, Sadeka, Jacky Keung, Kevin Lee, and Anna Liu. 2012. "Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud". *Future Generation Computer Systems* 28 (1). Elsevier BV: 155–62. doi:10.1016/j.future.2011.05.027.

19. Fang, Wei, ZhiHui Lu, Jie Wu, and ZhenYin Cao. 2012. "RPPS: A Novel Resource Prediction and Provisioning Scheme in Cloud Data Center". In *2012 IEEE Ninth International Conference on Services Computing*. IEEE. doi:10.1109/scc.2012.47.

20. Herbst, Nikolas Roman, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. 2014. “Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning”. *Concurrency and Computation: Practice and Experience* 26 (12). Wiley-Blackwell: 2053–78. doi:10.1002/cpe.3224.

21. Dutta, Sourav, Sankalp Gera, Akshat Verma, and Balaji Viswanathan. 2012. “SmartScale: Automatic Application Scaling in Enterprise Clouds”. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE. doi:10.1109/cloud.2012.12.

22. Sharma, Upendra, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. 2011. “A Cost-Aware Elasticity Provisioning System for the Cloud”. In *2011 31st International Conference on Distributed Computing Systems*. IEEE. doi:10.1109/icdcs.2011.59.

23. Srirama, Satish Narayana, and Alireza Ostovar. 2014. “Optimal Resource Provisioning for Scaling Enterprise Applications on the Cloud”. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. IEEE. doi:10.1109/cloudcom.2014.24.

24. Aslanpour, Mohammad Sadegh, Mostafa Ghobaei-Arani, and Adel Nadjaran Toosi. 2017. “Auto-Scaling Web Applications in Clouds: A Cost-Aware Approach”. *Journal of Network and Computer Applications* 95 (October). Elsevier BV: 26–41. doi:10.1016/j.jnca.2017.07.012.

25. Costache, Stefania, Nikos Parlavantzas, Christine Morin, and Samuel Kortas. 2012. “Themis: Economy-Based Automatic Resource Scaling for Cloud Systems”. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE. doi:10.1109/hpcc.2012.56.

26. Binnig, Carsten, Abdallah Salama, Erfan Zamanian, Muhammad El-Hindi, Sebastian Feil, and Tobias Ziegler. 2015. “Spotgres - Parallel Data Analytics on Spot Instances”. In *2015 31st IEEE International Conference on Data Engineering Workshops*. IEEE. doi:10.1109/icdew.2015.7129538.

27. Poola, Deepak, Kotagiri Ramamohanarao, and Rajkumar Buyya. 2014. “Fault-Tolerant Workflow Scheduling Using Spot Instances on Clouds”. *Procedia Computer Science* 29. Elsevier BV: 523–33. doi:10.1016/j.procs.2014.05.047.

28. Lu, Sifei, Xiaorong Li, Long Wang, Henry Kasim, Henry Palit, Terence Hung, Erika Fille Tupas Legara, and Gary Lee. 2013. “A Dynamic Hybrid Resource Provisioning Approach for Running Large-Scale Computational Applications on Cloud Spot and On-Demand Instances”. In *2013 International Conference on Parallel and Distributed Systems*. IEEE. doi:10.1109/icpads.2013.117.

29. Voorsluys, William, and Rajkumar Buyya. 2012. “Reliable Provisioning of Spot Instances for Compute-Intensive Applications”. In *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*. IEEE. doi:10.1109/aina.2012.106.

30. Chen, Changbing, Bu Sung Lee, and Xueyan Tang. 2014. “Improving Hadoop Monetary Efficiency in the Cloud Using Spot Instances”. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. IEEE. doi:10.1109/cloudcom.2014.35.

31. Knauth, Thomas, and Christof Fetzer. 2012. “Spot-on for Timed Instances: Striking a Balance between Spot and On-Demand Instances”. In *2012 Second International Conference on Cloud and Green Computing*. IEEE. doi:10.1109/cgc.2012.61.

32. Zafer, Murtaza, Yang Song, and Kang-Won Lee. 2012. "Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs". In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE. doi:10.1109/cloud.2012.59.

33. Chu, Hsuan-Yi, and Yogesh Simmhan. 2014. "Cost-Efficient and Resilient Job Life-Cycle Management on Hybrid Clouds". In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. doi:10.1109/ipdps.2014.43.

34. Gong, Zhiyong, Dingan Li, and Qi Fan. 2008. "Research on Relation between Stock Index Futures and Spot Market". In *2008 4th International Conference on Wireless Communications Networking and Mobile Computing*. IEEE. doi:10.1109/wicom.2008.2239.

35. Jangjaimon, Itthichok, and Nian-Feng Tzeng. 2015. "Effective Cost Reduction for Elastic Clouds under Spot Instance Pricing Through Adaptive Checkpointing". *IEEE Transactions on Computers* 64 (2). Institute of Electrical and Electronics Engineers (IEEE): 396–409. doi:10.1109/tc.2013.225.

36. Yi, Sangho, Artur Andrzejak, and Derrick Kondo. 2012. "Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances". *IEEE Transactions on Services Computing* 5 (4). Institute of Electrical and Electronics Engineers (IEEE): 512–24. doi:10.1109/tsc.2011.44.

37. Jung, Daeyong, SungHo Chin, KwangSik Chung, HeonChang Yu, and JoonMin Gil. 2011. "An Efficient Checkpointing Scheme Using Price History of Spot Instances in Cloud Computing Environment". In *Lecture Notes in Computer Science*, 185–200. Springer Berlin Heidelberg. doi:10.1007/978-3-642-24403-2_16.

38. Zhao, Han, Miao Pan, Xinxin Liu, Xiaolin Li, and Yuguang Fang. 2012. “Optimal Resource Rental Planning for Elastic Applications in Cloud Market”. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE. doi:10.1109/ipdps.2012.77.

39. Qu, Chenhao, Rodrigo N. Calheiros, and Rajkumar Buyya. 2016. “A Reliable and Cost-Efficient Auto-Scaling System for Web Applications Using Heterogeneous Spot Instances”. *Journal of Network and Computer Applications* 65 (April). Elsevier BV: 167–80. doi:10.1016/j.jnca.2016.03.001.

40. Sharma, Prateek, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. 2015. “SpotCheck”. In *Proceedings of the Tenth European Conference on Computer Systems - EuroSys 15*. ACM Press. doi:10.1145/2741948.2741953.

41. Singh, Rahul, Prateek Sharma, David Irwin, Prashant Shenoy, and K.K. Ramakrishnan. 2014. “Here Today Gone Tomorrow: Exploiting Transient Servers in Datacenters”. *IEEE Internet Computing* 18 (4). Institute of Electrical and Electronics Engineers (IEEE): 22–29. doi:10.1109/mic.2014.40.

42. Tang, ShaoJie, Jing Yuan, and Xiang-Yang Li. 2012. “Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance”. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE. doi:10.1109/cloud.2012.134.

43. Anand, Aishwarya. 2017. “Managing Infrastructure in Amazon Using EC2 CloudWatch EBS IAM and CloudFront”. *International Journal of Engineering Research And* V6 (03). ESRSA Publications Pvt. Ltd. doi:10.17577/ijertv6is030335.

44. Jung, Daeyong, SungHo Chin, KwangSik Chung, HeonChang Yu, and JoonMin Gil. 2011. “An Efficient Checkpointing Scheme Using Price History of Spot Instances in Cloud Computing Environment”. In *Lecture Notes in Computer Science*, 185–200. Springer Berlin Heidelberg. doi:10.1007/978-3-642-24403-2_16.

45. Amazon Docs, 2018. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-purchasing-options.html.