Four-fold Auto-scaling for Docker Containers

Philipp Hoenisch[1,2] Ingo Weber[2,3], Stefan Schulte[1],
Liming Zhu[2,3] and Alan Fekete[2,4]
{p.hoenisch, s.schulte}@infosys.tuwien.ac.at,
{firstname.lastname}@nicta.com.au
[1]TU Wien, Austria,
[2]Software Systems Research Group, NICTA, Sydney
[3]School of Computer Science & Engineering,
University of New South Wales
[4]School of Information Technologies, University of Sydney, Australia

Technical Report
UNSW-CSE-TR-201513
July 2015

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Virtual machines (VMs) are quickly becoming the default method of hosting Web applications (apps), whether operating in public, private, or hybrid clouds. Hence, in many cloud-based systems, auto-scaling of VMs has become a standard practice. However, VMs suffer from several disadvantages, e.g., the overhead of needed resources as a full operating system (OS) needs to be started, a degree of vendor lock-in and the relatively coarse-grained nature. This can be overcome by using lightweight container technologies like Docker as the OS is not included in a container, instead, the one from the host machine is used. Like VMs, containers offer resource elasticity, isolation, flexibility and dependability. On the one hand, containers need to run on a compatible OS and share resources through the outside OS, on the other hand, exactly this fact leads to benefits such as a faster start-up time and less overhead in terms of used resources.

A common approach is to run containers on top of VMs, e.g., in a public cloud. Doing so, the flexibility for auto-scaling increases, since VMs can then be sub-divided. However, the additional freedom also means that scaling decisions become more complex: considering horizontal and vertical scaling on both, the container and the VM level, auto-scaling is now four-fold.

We address four-fold auto-scaling by (i) capturing the decision space as a multi-objective optimization model, (ii) solving instances of that model dynamically to find an optimal solution, and (iii) executing the dynamic scaling decision through a scaling platform for managing Docker containers on top of VMs. We evaluated our approach with realistic apps, and found that using our approach the average cost per request is about 20-28% lower.

This technical report provides details omitted from a short conference paper [3], as follows. Section 1 presents the optimization model in detail, after first introducing the preliminaries needed to understand the model. The control architecture is described in Section 2, and the evaluation of our approach is described in Section 3. A broader introduction, a motivating example, and a discussion of related work in turn can be found in [3].

1 Optimization Approach

The following describes an optimization model to solve the four-fold auto-scaling, i.e., this multi-objective optimization model computes a VM leasing plan and places Docker containers among them while ensuring given SLAs and a minimization of cost.

1.1 Preliminaries

We assume a PaaS scenario: A service provider hosts several different *apps* of a specific *type*. These apps are provided by developers or other content providers who want to have a fixed hosted solution. Because of this, each app may come with a different *Service Level Agreement (SLA)* defining different *Service Level Objectives (SLO)*, e.g., the *response time* should not exceed a certain value or a specific *throughput* should be possible.

The service provider leases *Virtual Machines (VMs)* from a Cloud provider and deploys apps onto them, thus they are available to potential clients. For that, the app is bundled in a Docker *container* which is then deployed on a VM resulting in a certain *container instance*. Depending on the app's SLA, the container may have a specific *container configuration* defining the requirements to the underlying VM, e.g., specifying the least amount of of available *CPU* or *RAM*.

The requirements in terms of CPU are defined in *CPU shares*. By default, each VM has 1024 CPU shares available which are then shared among the hosted container instances. For example, considering a VM hosting three containers where as one of the three containers get a CPU share of 512 and the other two 256 CPU shares each. If all three app will attempt to use 100% of their available shares, this will result in 50% of CPU for the first container, and 25% for each of the other two containers. In contrast to that, the requirements in terms of RAM are fixed values defined in *Mega Bytes (MB)*.

The provider can deploy one or many different container instances (of different types) on a single VM. In addition, to achieve a highly available system, the provider may deploy a specific container type on several VMs resulting in various container instances, where each container instance may have a different configuration.

1.2 System Model

In order to account for several different apps we consider multiple container types. Each container represents a single app. The set of container types is defined by $D = \{1, ..., d^{\#}\}$ and d is a specific container type. Each container has a specific configuration defining the requirements of resources. For the sake of simplicity, we generalize all types of resources here, however, within our implementation we differentiate the resources and account for CPU and RAM. A specifically configured container is defined as c_d. Complimentary to that, $C_d = \{1_d, ..., c_d^{\#}\}$ defines the set of different configurations of a specific container type d. The exact requirements of resources of a specifically configured container of type c_d are defined by $r_{(R,c_d)}$.

Analogous to containers, we also account for different VM types. The set of VM types is defined by $V = \{1, ..., v^{\#}\}$ where v is a specific VM type, e.g., a

single-, dual-, or quad-core. Each VM type has a different supply of resources (e.g., CPU cores, RAM, disc size, etc.). Notably, as for the container types, we do not differentiate between resource types within the paper. However, the actual differentiation is done within our implementation. VMs need to be instantiated in order to host containers. Theoretically, unlimited instances of a specific VM type can be leased, however, we assume a maximum of leasable VM instances at a certain time, i.e., $k_v^{\#}$. Therefore, the set of leasable VM instances of type v is given by $K_v = \{1_v, ..., k_v^{\#}\}$, and a specific VM instance by k_v.

The cost which apply for leasing a VM instance is defined by c_v and apply once for a billing cycle called *Billing Time Unit (BTU)*. A BTU is also the minimum leasing duration. This means, releasing a VM instance k_v before the end of the BTU corresponds to wasting already paid resources. Hence, the actual leasing duration of a specific VM instance k_v is a multiple of the BTU. The remaining leasing duration of a VM instance k_v at time t is defined by $d_{(k_v,t)}$ and is given in milliseconds. At the end of this time, the VM instance will be terminated automatically or has to be leased for an additional BTU.

As previous stated, a container has to be deployed on a VM instance. The corresponding *deployment time* Δ_d depends on the container type d. The time unit is given in milliseconds and only applies for instantiating a container for the first time on a specific VM instance k_v. The container files are cached within the VM making future deployment times negligibly small, i.e., a few milliseconds to seconds. The decision if a container d should be deployed on a VM instance k_v at time t is defined in the decision variable $x_{(c_d,k_v,t)}$. A value of 1 means that the container of type d and configuration c should be deployed on the VM instance k_v at time t. If a specifically configured container c_d has already been deployed on a VM instance k_v, we use $z_{(d,k_v,t)} = 1$. Analogously to the container deployment time Δ_d, the VM *start-up time* of a specific instance type v is labeled with Δ_v. For specifying that a specific VM instance k_v should be leased at time t the variable $y_{(k_v,t)}$ is used. A value of 1 means that the VM instance k_v should be leased for one BTU. If a specific VM instance k_v is already running at time t, we use the variable $\beta_{(k_v,t)}$. A value of 1 means the VM is running and 0 means it is terminated. The total amount of leased VMs of type v at time t are defined by the variable $\gamma_{(v,t)}$.

Having the system model explained in this section, we next present the optimization model. This model takes as input a set of containers (D) including their configurations and optional SLAs, a set of VM types (V) including the information of how many instances are possible for each type including their configuration and the information of how many requests are to be expected at time t for each container type d, which is expressed as $i_{(d,t)}$.

1.3 Optimization Model

The overall objective function is defined in (1.1): it is subject for minimization the overall cost. This function comprises four terms. The first term, i.e., $\sum_{v \in V} c_v \cdot \gamma_{(v,t)}$ computes the overall leasing cost which accrue if $\gamma_{(v,t)}$ VM instances of type v with cost c_v each at time t are leased. The second term, i.e., $\sum_{d \in D} \sum_{c_d \in C_d} \sum_{v \in V} \sum_{k_v \in K_v} \left((1 - z_{(d,k_v,t)}) \cdot (x_{(c_d,k_v,t)} \cdot \Delta_d) \right)$ sums up the time which is needed to deploy a container (Δ_d) on a specific VM instance. If a container gets deployed the first time on a VM instance, some data needs to be downloaded from the container registry. Hence, this procedure may take

some time. However, this data is cached on the VM instance as long as it is running, thus, future deployment of the same container type will be much faster as no data has to be downloaded. This information is used to prioritize placements of containers on VM instance where such a cache already exists. The third term, i.e., $\sum_{v \in V} \sum_{k_v \in K_v} (\omega_f^R \cdot f_{(R,k_v,t)})$ computes the amount of *free* resources. In order to control the value of this term within the objective function, we weight the free resources using the weight ω_f^R. This term ensures that containers are deployed on already leased VM instances instead of leasing additional once, provided enough resources are available. The fourth term, i.e., $\sum_{d \in D} \sum_{c_d \in C_d} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_s \cdot s_{(i,c_d,t)} \cdot x_{(c_d,k_v,t)}\right)$ sums up the amount of deployed Docker containers for each container type at time t. It aims at at reducing the risk of over-provisioning in a way, that it demands to lease the smallest amount of containers while still fulfilling the demand. As in the third term, the value of available resources per Docker container ($s_{(i,c_d,t)}$) is weighted with a constant value ω_s in order to reduce the weight of this term within the overall objective function.

$$\min \Bigg[\sum_{v \in V} c_v \cdot \gamma_{(v,t)} + \sum_{d \in D} \sum_{c_d \in C_d} \sum_{v \in V} \sum_{k_v \in K_v} \left((1 - z_{(d,k_v,t)}) \cdot (x_{(c_d,k_v,t)} \cdot \Delta_d) \right) + \sum_{v \in V} \sum_{k_v \in K_v} \omega_f^R \cdot f_{(R,k_v,t)} + \sum_{d \in D} \sum_{c_d \in C_d} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_s \cdot s_{(i,c_d,t)} \cdot x_{(c_d,k_v,t)}\right) \Bigg] \tag{1.1}$$

The constraint (1.2) ensures that enough resources in any dimension are available on each VM, this involve CPU and RAM equally. For that, we check if the resource supply of a specific VM instance k_v is greater or equal than the sum of required resources of each container (and for each configuration, i.e., for all $d \in D$ and all $c_d \in C_d$). Notably, the resource demand of a specific container is only considered if the corresponding container will be deployed on that VM instance, i.e., if $x_{(c_d,k_v,t)} = 1$. On the right side of this equation, one can find the variable $g_{(k_v,t)}$ which will be introduced in constraint (1.12). A value of 1 indicates that the corresponding VM instance k_v will be leased or is running at time t.

$$\sum_{d \in D} \sum_{c_d \in C_d} \left(r_{(R,c_d)} \cdot x_{(c_d,k_v,t)}\right) \leq s_{(R,v)} \cdot g_{(k_v,t)} \tag{1.2}$$

As mentioned before, the objective function in (1.1) aims at minimizing the amount of unused resources for each leased VM instance. The corresponding value $f_{(R,k_v,t)}$ is computed in constraint (1.3). For that, the amount of required resources $r_{(R,c_d)}$ are add up for each container $d \in D$ and each configuration $c_d \in C_d$ which are meant to be deployed on a specific VM instance k_v, i.e., if $x_{(c_d,k_v,t)} = 1$. This value is than subtracted by the resource supply $s_{(R,v)}$.

$$g_{(k_v,t)} \cdot s_{(R,v)} - \sum_{d \in D} \sum_{c_d \in C_d} \left(r_{(R,c_d)} \cdot x_{(c_d,k_v,t)}\right) \leq f_{(R,k_v,t)} \tag{1.3}$$

The constraint (1.4) ensures that enough containers of a specific type d are deployed at time t. For that, we sum up the amount of possible invocations on

each container of type d (all $d \in D$) with configuration c which are meant to be deployed at time t, i.e., $x_{(c_d,k_v,t)} = 1$. The sum has to be greater or equal $i_{(c_d,t)}$. The corresponding variable $s_{(i,c_d,t)}$ represents the maximum amount of requests which are possible on a certain container configuration c_d while ensuring given QoS. $s_{(i,c_d,t)}$ is defined in (1.13).

$$\sum_{c_d \in C_d} \sum_{v \in V} \sum_{k_v \in K_v} s_{(i,c_d,t)} \cdot x_{(c_d,k_v,t)} \geq i_{(d,t)} \tag{1.4}$$

The constraint (1.5) ensures that a specific VM instance k_v hosts maximum one container of a specific type d (for all possible configurations, i.e., $c_d \in C_d$).

$$\sum_{c_d \in C_d} x_{(c_d,k_v,t)} \leq 1 \tag{1.5}$$

The sum of $y_{(k_v,t)} \geq 1$ (for all $v \in V$) values in (1.6) indicates the total number of VM instances of type v which have to be leased in time t. The sum is represented in $\gamma(v,t)$.

$$\sum_{k_v \in K_v} y_{(k_v,t)} \leq \gamma_{(v,t)} \tag{1.6}$$

In constraint (1.3) we compute the amount of *unused* resources. This is only required if a VM instance k_v is leased at time t. For this, we use the helper variable $g_{(k_v,t)}$ which takes the value 1 if the VM instance k_v is already leased in time t ($\beta_{(k_v,t)} = 1$) or will be leased ($y_{(k_v,t)} = 1$) in time t (constraints (1.7)-(1.9)).

$$g_{(k_v,t)} \geq y_{(k_v,t)} \tag{1.7}$$

$$g_{(k_v,t)} \geq \beta_{(k_v,t)} \tag{1.8}$$

$$g_{(k_v,t)} \leq y_{(k_v,t)} + \beta_{(k_v,t)} \tag{1.9}$$

The decision variable $x_{(c_d,k_v,t)}$ indicates if a certainly configured container c_d should be deployed on VM instance k_v at time t. This requires the VM instance to be running which is ensured in (1.10). Since it might be the case that several different container types are assigned to one particular VM instance k_v, the left-hand side of this equation may exceed the value 1. Thus, in order to satisfy this constraint, the right-hand side of this equation is multiplied with an *arbitrary* large number M, i.e., $g_{(k_v,t)} \cdot M$. We chose 1,000 for M as it is unlikely that a single VM instance will host more than 1,000 different containers a time.

$$\sum_{d \in D} \sum_{c_d \in C_d} x_{(c_d,k_v,t)} \leq g_{(k_v,t)} \cdot M \tag{1.10}$$

The optimization model ensures that enough resources are leased to handle the upcoming demand. Hence, the outcome of the optimization model is two-fold: 1) it indicates whether additional VM instances need to be leased or if already leased VM instances can be terminated and 2) it indicated on which VM instance what container types should be deployed. Consequently, the system landscape is under a continuous change, i.e., VM instances may come up or

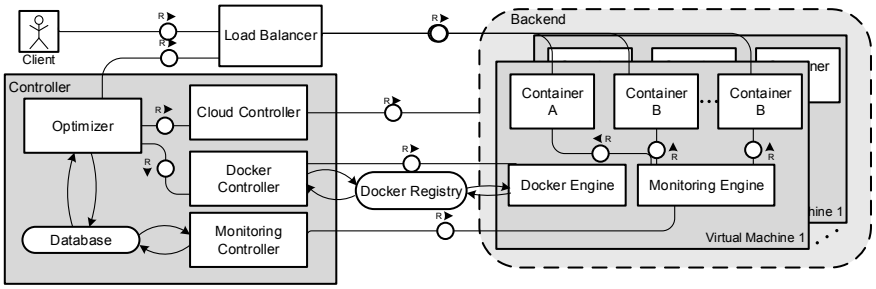


Figure 2.1: Architecture

disappear from time to time. The decision whether a particular VM should be accounted as available in time t, i.e., the resources are available for deploying containers is ensured in constraint (1.11) for all $d \in D$, $c_d \in C_d$. This constraint demands, that either a specific VM instance k_v is leased at least another $60,000$, or if the instance has to be leased for another full leasing cycle, i.e., a full BTU. If the VM instance's remaining leasing duration is less than 60,000, i.e., $d_{(k_v,t)} \cdot \beta_{(k_v,t)} \leq 60,000$ and the VM instance will not be leased ($y_{(k_v,t)} = 0$), then no containers will be deployed on that particular VM instance. We haven chosen a value of 60,000 milliseconds as this is the maximum interval of running the optimization.

$$d_{(k_v,t)} \cdot \beta_{(k_v,t)} + BTU \cdot y_{(k_v,t)} \geq x_{(c_d,k_v,t)} \cdot 60,000 \tag{1.11}$$

The remaining constraints in (1.12) demand the co-domain of the used variables. The optimization model is assembled by the presented constraints (1.1)-(1.13) and solved in a regular interval of 60 seconds.

$$x_{(c_d,k_v,t)}, y_{(k_v,t)}, g_{(k_v,t)}, \beta_{(k_v,t)} \in \{0,1\} \tag{1.12}$$

$$i_{(c_d,t)}, s_{(i,c_d,t)} \in \mathbb{N}_0 \tag{1.13}$$

2 Architecture

We here show how the optimization model from Section 1 can be used to control scaling decisions in a deployment of Docker containers. For concreteness, we describe the prototype implementation of our architecture, done in Java 1.7 and using IBM CPLEX [1] as a solver, accessed through the Java ILP interface. We plan to make the source code of our prototype available in the coming months.[2] The architecture is presented in Fig. 2.1 and depicts two top level entities: the *Controller*, hosting the core decision-making functionality, and the *Backend*, consisting out of an arbitrary number of VMs, each hosting one to many Docker containers and a *Monitoring Engine* to collect CPU and RAM load statistics.

[1] `http://www.ibm.com/software/commerce/optimization/cplex-optimizer`, accessed 29/07/15

[2] `https://reliableops.com`

The Controller includes the *Optimizer* which implements the optimization model and is responsible for creating a detailed VM leasing and Docker container placement plan. To set up the optimization task, the Optimizer collects required data such as information about the actual system state and VM leasing and Docker placement plan. This information can be gained from the *Database*. Solving the optimization model happens at regular intervals or it can be triggered by relevant events.

The outcome of the optimization model needs to be actioned. The VM provisioning is done by the *Cloud Controller*. It is connected with the IaaS provider, in our prototype an OpenStack-based cloud (OpenStack Folsom)[3]. The Cloud Controller arranges the leasing of additional VM instances or terminates unneeded ones. It makes use of the open-source JClouds library [4]. The Cloud Controller can perform three different actions. First, *leasing* a new VM instance for a full BTU cycle. Second, *terminating* an existing VM instance, i.e., releasing it. And third, *renewed leasing* an existing VM instance for another full BTU cycle.

The *Docker Controller* is responsible for placing the Docker containers as indicated by the solution of the optimization task. It is connected to the *Docker Engine* on each Backend VM using the standard Docker remote API [5]. The Docker Controller can arrange to *start* a new container of a certain type, *stop* an existing container, or *resize* an existing container, i.e., change the amount of available resources for that particular container. A *Database* is deployed which stores monitoring information from the Backend VMs as well as the outcome of each optimization run.

The *Backend* consists out of an arbitrary number of VM instances, called *Backend VMs*. Each hosts one or many Docker containers of different types. In addition, each Backend VM instance provides a *Docker Engine* for deployment, and a *Monitoring Engine* which collects information from the Docker Engine using the standard Docker API. This data will be requested by the Monitoring Controller.

The system also provides a *Load Balancer* and a *Docker Registry*. The Load Balancer is the entry point for Clients who want to access the apps. It serves as a single entry point and transparently forwards the requests to the hosted containers based on a round-robin adjusted for the available resources. We use the open-source HAProxy [6] for our Load Balancer. The Docker Registry contains the Docker images for the various apps.

3 Evaluation

Our proposed optimization-based control of scaling has been evaluated through measuring behavior of the prototype implementation described in Section 2. We compare this against two state-of-the art baselines for making scaling decisions. The details of the evaluation scenario including the arrival patterns are explained in Section 3.1-3.3 and the outcome is discussed in Section 3.4.

[3] `http://www.openstack.org/software/folsom/`, accessed 27/7/15
[4] `https://jclouds.apache.org/`, accessed 27/7/15
[5] `https://docs.docker.com/reference/api/docker_remote_api/`, accessed 27/7/15
[6] `http://www.haproxy.org`, accessed 27/7/15

3.1 Arrival Patterns

In our evaluation we take up the scenario from Section 1.1 and use three different Docker container types which differ in their technologies as well as in the demand of computing resources. We apply two different arrival patterns which can be seen in Fig. 3.1a and Fig. 3.2a. Each figure shows how a different amount of parallel requests were sent to each app, round-by-round. A round lasts for one minute, i.e., the whole scenario ran for about 40 minutes. A value of 50 on the vertical axis means that 50 requests for the particular app were send over a time span of 10 seconds. The available VM types in our evaluation were 1) a *single-core* VM with 1 CPU and 1,820 MB RAM, 2) a *dual-core* VM with 2 CPUs and 3,750 MB RAM, and 3) a *quad-core* VM with 4 CPUs and 7,680 MB RAM, with costs that made a quad-core cheaper than two dual-cores, and a dual-core cheaper than two single-cores.

3.2 Baseline

We compare our approach against two different baseline scaling techniques, which are threshold-based: additional resources are leased if a particular upper threshold has been exceeded, or released if the load dropped below a lower threshold [4, 2, 1]. The thresholds differ for each container and type and their values have been chosen through stress testing each configuration prior to our evaluation. These same thresholds are used for our approach as for the baselines. The first baseline scales with a *One-for-All* placement strategy: each VM has all the container types. Initially a quad-core VM is leased hosting a container of each type. If the load increases past the threshold, an additional quad-core VM will be leased and given one container of each app type. If the load decreases, one VM will be released.

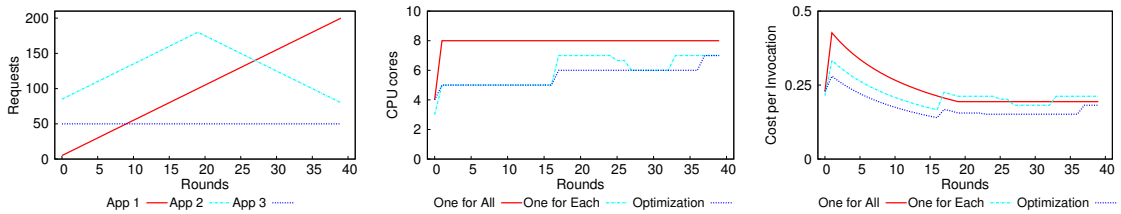
The second baseline scenario follows a *One-for-Each* strategy with a separate VM for each container. At the beginning three single-core VMs are leased having exactly one container deployed each. An additional single-core VM (with another one instance of the same container type) will be leased if the load passes the threshold on a current VM with that container type; and a VM-container pair will be released if the load gets too low.

3.3 Metrics

To assess the quality of our approach, we use different metrics and compare them against the results of the baselines. The main objective of our approach is to minimize cost. Hence, comparing the VM *leasing cost* is an obvious option. Cost for leasing a VM instance apply only once per BTU. Leasing not enough resources will definitely have an impact on the QoS. Hence, we measured the response times and compare the overall *SLA violations*. Last but not least, we compute the *cost per invocation*. We run each scenario and setting three times to get reliable numbers and also provide the standard deviation σ.

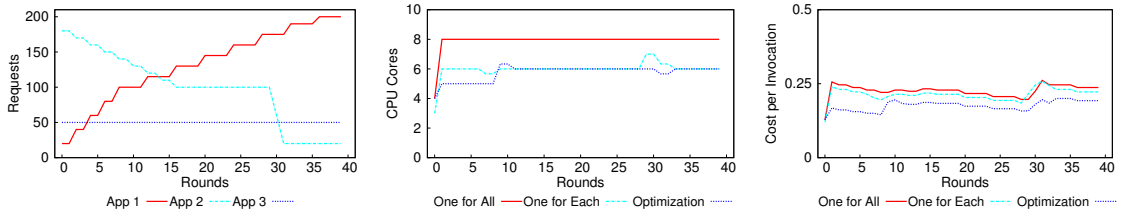
3.4 Discussion

The result of our evaluation can be found in Table 3.1 and Table 3.2 and are shown as charts in Fig. 3.1c and Fig. 3.1b for arrival pattern 1, and for arrival



(a) Arrival Pattern (b) Total Leased Cores (c) Cost per Invocation

Figure 3.1: Arrival Pattern 1



(a) Arrival Pattern (b) Total Leased Cores (c) Cost per Invocation

Figure 3.2: Arrival Pattern 2

Table 3.1: Evaluation Results – Arrival Pattern 1

	Arrival Pattern 1		
	Optimized	One-for-All	One-for-Each
Leased Cores (σ)	28.13 (0.38)	39.5 (0)	29.68 (0.14)
Leasing Cost (σ)	378.4 (0.92)	505.6 (0)	474.67 (2.31)
Cost/Invocations (σ)	0.17 (0.012)	0.23 (0.22)	0.21 (0.02)
SLA Adherence	97.47%	98.02%	98.22%

pattern 2 in Fig. 3.2c and Fig. 3.2b. For each chart, on the horizontal axis, the rounds are presented. On the vertical axis either the amount of leased CPUs or the cost per invocation are shown. As can be seen in Table 3.1 and Table 3.2, the SLA adherence for both scenarios and for each scaling strategy are very similar. They vary only for about a few percentage points. This can be reduced to the fact that we used the same thresholds for scaling up or down. Hence, in the following we will focus more about the leased CPUs and produced cost. It is not surprising that the One-for-All scenario produced the most cost. In this scaling strategy, only quad-core VMs were leased, and each app was allocated the same number of containers (equal to the number of VMs), leading to a highly overprovisioned system as the resources were not needed for whichever app was getting low load in that round. Hence, leasing smaller VM types makes more sense (∼5% cheaper). However, even leasing smaller VMs may not be perfect. In the One-for-Each scenario, only single-core VMs were leased, each hosting exactly one container type. As single-core VMs can not hold as much

Table 3.2: Evaluation Results – Arrival Pattern 2

	Arrival Pattern 2		
	Optimized	One-for-All	One-for-Each
Leased Cores	28.75	39.5	29.88
(σ)	(0.43)	(0)	(0.25)
Leasing Cost	393.2	505.6	478.00
(σ)	(1.38)	(0)	(4)
Cost/Invocations	0.17	0.24	0.22
(σ)	(0.03)	(0.06)	(0.04)
SLA Adherence	96.95%	96.92%	97.1%

load as higher-cored VMs, more instances are needed. In addition, based on the cost model, leasing two single-core VMs is more expensive than leasing one dual-core VM (∼20% more expensive). Thus vertical scaling would have been helpful: leasing the right VM size depending on the need will lead eventually to less leasing cost. Our four-fold optimization approach can benefit from this. This can be seen in Fig. 3.1c and Fig. 3.2c which show the cost per invocation. The high values at the beginning are related to the low number of requests, as more requests come in, the lower the cost per invocation gets. Eventually, using arrival pattern 1, we achieved savings of ∼28% (with ∼33% less cores) over the One-for-All scenario and a monetary saving of ∼23% (∼4% less cores) over the One-for-Each. Using arrival pattern 2, we achieved with our optimization a cost saving of ∼25% (∼32% less cores) over the One-for-All and a saving of ∼20% (∼4% less cores) over the One-for-Each scaling strategy.

4 Conclusion

Traditionally, apps are hosted directly on VMs. However, this approach suffers from different disadvantages such as the overhead of needing a full OS leading to a slow start-up time, a degree of vendor lock-in and their relatively coarse-grained nature. Those problems can be overcome by using lightweight linux container technologies such as Docker containers. These add an additional abstraction layer on top of VMs which allows more efficient resource usage. However, when containers are deployed on top of VMs, auto-scaling decisions are getting more complex.

We addressed this now four-fold auto scaling as a *multi-objective optimization problem*, and we proposed a control architecture which is able to dynamically and elastically adjusted the VM and container provisioning. Based on a prototype implementation in Java 1.7, we evaluated our approach and compared it against naive scaling strategies: One-for-All and One-for-Each. The numbers reveal, that following our approach, a cost reduction of 20-28% can be achieved.

Bibliography

[1] Marc E. Frincu, Stéphane Genaud, and Julien Gossa. On the Efficiency of Several VM Provisioning Strategies for Workflows with Multi-threaded Tasks on Clouds. *Computing*, 96:1059–1086, 2014.

[2] Stéphanse Genaud and Julien Gossa. Cost-wait Trade-offs in Client-side Resource Provisioning with Elastic Clouds. In *4th Intern. Conf. on Cloud Computing (CLOUD 2011)*, pages 1–8. IEEE, 2011.

[3] Philipp Hoenisch, Ingo Weber, Stefan Schulte, Liming Zhu, and Alan Fekete. Four-fold auto-scaling on a contemporary deployment platform using Docker containers. In *International Conference on Service Oriented Computing (ICSOC)*, Goa, India, November 2015.

[4] Philipp Leitner, Waldemar Hummer, Benjamin Satzger, Christian Inzinger, and Schahram Dustdar. Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In *5th Intern. Conf. on Cloud Computing (CLOUD 2012)*, pages 213–220. IEEE, 2012.