A Discrete-Time Feedback Controller for Containerized Cloud Applications

Luciano Baresi, Sam Guinea, Alberto Leva, and Giovanni Quattrocchi
DEIB - Politecnico di Milano, Piazza Leonardo Da Vinci 32 - Milan, Italy
{luciano.baresi|sam.guinea|alberto.leva|giovanni.quattrocchi}@polimi.it

ABSTRACT

Modern Web applications exploit Cloud infrastructures to scale their resources and cope with sudden changes in the workload. While the state of practice is to focus on dynamically adding and removing virtual machines, we advocate that there are strong benefits in containerizing the applications and in scaling the containers. In this paper we present an autoscaling technique that allows containerized applications to scale their resources both at the virtual machine (VM) level and at the container level. Furthermore, applications can combine this infrastructural adaptation with platform-level adaptation. The autoscaling is made possible by our planner, which consists of a grey-box discrete-time feedback controller. The work has been validated using two application benchmarks deployed to Amazon EC2. Our experiments show that our planner outperforms Amazon's AutoScaling by 78% on average without containers; and that the introduction of containers allows us to improve by yet another 46% on average.

CCS Concepts

•Networks → Cloud computing; •Social and professional topics → Software selection and adaptation; Quality assurance; •Computing methodologies → Computational control theory;

Keywords

Adaptive systems; cloud computing; control theory; containers; software adaptation

1. INTRODUCTION

Nowadays many Web applications are deployed and executed in the cloud to scale more easily according to the current workload. Industry has developed various techniques for automating and improving the management of these kinds of applications. A concrete example is Amazon's AutoScaling [1], which allows system administrators to determine when and how an application's resources should dynamically increase or decrease. Academia has also provided a large body of work on cloud management [19, 24, 27, 47, 49], with a strong focus on self-adaptation [25, 32, 40] and on dynamic resource allocation at the infrastructure layer [37, 42, 44, 53].

The state of the art comprises many MAPE-based [38] approaches where planning is based on heuristics [22, 26, 50], artificial intelligence [34, 35, 45], queueing theory [33, 51] and control theory [30, 41, 46]. Moreover, literature distinguishes among three kinds of possible adaptations. *Infrastructure adaptation* changes the number of computing resources allocated to the application. *Platform adaptation* re-configures the platform stack on which the application runs its code (e.g., by changing the number of workers dedicated to an application server). *Feature adaptation* [39] temporarily removes optional features from the application when the system is saturated (e.g., by turning off a recommendation system of an e-commerce site).

In this context, the paper introduces a novel MAPE-based self-adaptation framework for cloud-based Web applications centered around three important novelties.

The first is that we allow system administrators to take advantage of both VM- and container-based resource management, effectively allowing them to mix two very different levels of granularity when conceiving infrastructure elasticity. Containerization is an emerging virtualization technique in which many processes —called containers— are run on the same physical machine without interference. Containers share a host operating system and are more lightweight and faster to boot than traditional VMs [29, 52]. Renting a cluster of VMs and using it to execute different containerized applications (or micro-services) is an emerging architectural pattern [16, 17, 31, 43, 48], but it still lacks a comprehensive approach for resource management.

The second is that we focus on coordinated infrastructure and platform adaptation, although the use of containers would also allow us to support feature adaptation. To the best of our knowledge, there are no approaches that support all the different kinds of adaptations. While infrastructure adaptation —at the VM level— is an industrial best practice, platform adaptation has yet to be thoroughly investigated. One of the main reasons for this is that the more aspects you want to adapt, the more your planner becomes complex, to the point in which it can lose generality and become over-fit for a specific application. Feature adaptation, on the other hand, has been implemented in the past.

Containers play an essential role in this coordinated approach. While the planning phase of our MAPE control loop

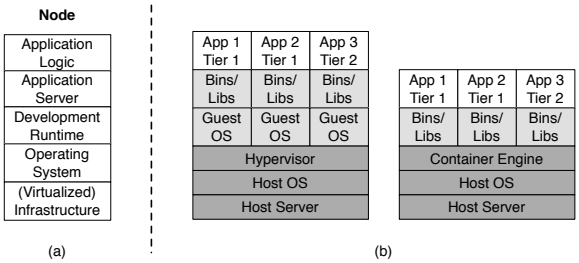

Figure 1: (a) Multi-level nature of a node; (b) Virtual machines (left) vs. containers (right).

treats containers as black boxes and focuses on their horizontal and vertical scaling, the containers themselves present well-defined technology- and application-specific callbacks through which their internals can be adapted. These callbacks are triggered every time something changes at the container infrastructure level.

The third is a novel MAPE planner that consists of a discrete-time feedback controller. Peculiar to our proposal is the structure of the said controller; it is tailored to the structure of the command-to-metrics dynamics to be governed, using a grey-box approach. The resulting controller is composed of a linear, time-invariant block plus a static linearization one. The planner is also endowed with an internal saturation management (anti-windup) mechanism.

The evaluation of our approach has been conducted using two cloud applications: RUBiS, an application for on-line auctions, and Pwitter, a simple Twitter-like social network. Our experiments show that our approach outperforms Amazon's AutoScaling (i.e., the industrial state of practice) by 78% on average when used solely with VMs and without containers. They also show that, if we introduce containers, and therefore adopt a finer granularity, we can improve the performance of our adaptation by another 46% on average.

The rest of the paper is organized as follows. Section 2 introduces our use of container-based technology. Section 3 presents a high-level overview of our approach. Section 4 discusses our control theory planner, while Section 5 explains how we combine infrastructure and platform adaptation. Section 6 presents the experiments done to validate our work. Section 7 discusses related approaches, while Section 8 concludes the paper.

2. ANATOMY OF A CLOUD APPLICATION

Our work focuses on Web applications that are deployed and executed on public, private, or hybrid clouds. The virtualized infrastructure is considered to be a black box: this means that we do not have access to the hypervisor or to the underlying physical machines.

Modern Web applications are typically developed using a multi-tier architecture. A *tier* logically and physically separates software components that deal with different functional aspects of an application. For example, a traditional 3-tier application comprises a presentation tier, which manages the user interface; a logic tier, which executes the application's business logic; and a data tier, which handles user data. Modern applications, however, can be made of more than three tiers. For example, one might decide to use two data tiers: one for a traditional relational database and one for a NoSQL database.

Each tier contains multiple nodes. A *node* is a computing instance that runs platform software (e.g., a JVM and an application server), as well as some actual application code (see Figure 1(a)). Given the definition of tier that we use within this paper, all the nodes in a tier execute the same technological stack. This is not an uncommon assumption; in fact, it directly mimics the way Amazon AWS defines application architectures in OpsWorks[1].

Managing multi-tiered cloud applications, while satisfying a set of functional and non-functional requirements, is a complex task. It requires understanding the dependencies that exist between nodes belonging to different tiers. For example, in a 3-tier application, a load balancer must know all the IP addresses of the application tier's nodes. Moreover, the nodes in a tier could change over time due to scale-in and scale-out policies.

2.1 Virtual Machines vs. Containers

A node is traditionally materialized as a VM. However, nowadays a node might also be a container. Containers provide a virtualization technique that operates at the Operating System (OS) level. They exploit several features of the Linux kernel, such as *namespaces* and *cgroups*, to create isolated views of the operating environment for different applications. A container has its own process space, virtualized network interface, and file system; and the operating system can allocate different amounts of resources (e.g., CPU, memory, and I/O) to each of them.

Figure 1(b) shows a layered comparison between VMs and containers. If we start our comparison by looking at the lower layers we can see that multiple VMs are managed by a single hypervisor that resides on a single host operating system. Each VM then contains its own guest operating system, its own platform stack composed of different libraries, middleware, and application servers, and its own application code.

On the other hand, containers are executed directly on top of the host operating system, optionally with the help of a container manager like Docker [5]. Each container has its own platform stack and its own application code. Containers have various advantages when compared to VMs: they are more lightweight and they are faster to boot and to terminate because they do not have to deal with a guest operating system [29, 52].

Industry is widely adopting containers as a means to favor portability and they are considered to be one of the main technological enablers of the DevOps movement [36]. Different development teams may use different operating systems and different platform stacks, making feature integration hard. However, thanks to containerization technology, features can be developed in isolation, with the guarantee that they will work the exact same way on any machine that supports containers.

Many containerization technologies exist. We chose Docker because it has the highest industry penetration and its performance can be considered best in class [29]. Docker is written in Go and it uses the *libcontainer* library to manage the Linux kernel. It distinguishes between images and container instances: the former are container snapshots that can then be used to generate new instances of these snapshots.

[1]OpsWorks is a configuration management tool provided by Amazon AWS for configuring and operating complex applications using DevOps technology.

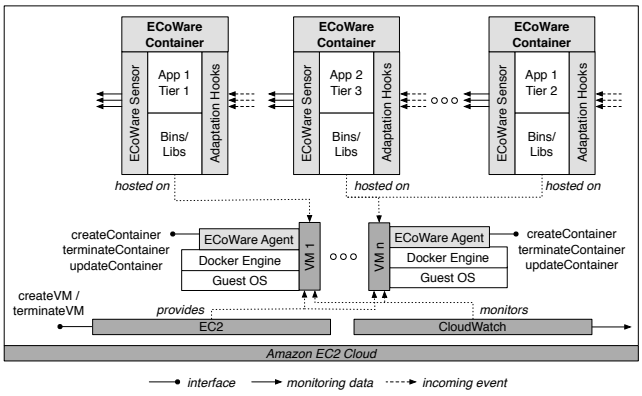

Figure 2: A deployment of ECoWare.

To conclude, there is no need to choose between VMs and containers. Containers can run inside VMs to increase security [6], and it is becoming quite common to find containers being used in conjunction with virtualized cloud infrastructures. For example, with Amazon EC2 Container Service [2] one can run Docker images directly across a cluster of EC2 VM instances.

3. ECOWARE

The solution presented in this paper extends ECoWare [20], our framework for the development of self-adaptive systems[2]. The main extensions in this paper are: (i) the support for containerized applications; (ii) an entirely new planner, designed from the ground up and based on control theory; and (iii) an execution processing model that allows for coordinated infrastructure and platform adaptation.

Figure 2 shows how containerized applications are deployed using ECoWare. EcoWare can also be used without containers, by deploying applications directly onto VMs, but in that case it looses the capability of performing a more fine-grained adaptation.

ECoWare is not limited to supporting a single application: it can manage multiple multi-tiered applications running on the same shared virtualized infrastructure. Each node in a tier is a container that is hosted on a VM. Each node is equipped with (i) an `ECoWare Sensor`, a component that has access to the node's internals and is responsible for generating application-specific monitoring data; and (ii) a set of `Adaptation Hooks` for platform adaptation.

Each employed VM also deploys an `ECoWare Agent`. This component is responsible for providing adaptation actions for creating and manipulating containers on that VM, and for collecting container-specific monitoring data, such as the containers' use of CPU and memory.

In the deployment shown in Figure 2 ECoWare uses Amazon EC2. It exploits EC2's VM management APIs to provide infrastructure adaptation actions for creating and manipulating VMs, and Amazon CloudWatch to obtain monitoring information about the VMs themselves.

Monitoring and Analysis. We have not extended ECoWare with any major novel monitoring and analysis capabilities. We did, however, produce new kinds of sensors to support both containers and VMs, and created a new Java template to facilitate the creation of application-specific sensors.

In EcoWare sensors generate data under the form of Service Data Objects (SDOs) and deliver them to a distributed messaging infrastructure implemented using RabbitMQ [13]. In turn, the distributed messaging infrastructure can deliver collected data to three different kinds of data manipulation tools: `Aggregators`, `KPI Processors`, and `Analyzers`.

ECoWare's data manipulation tools were built using Esper [28], an instrument for implementing complex event processing activities. Aggregators collect data from multiple sensors to create new composite SDOs. KPI Processors correlate and aggregate data coming from various sensors to generate business-level Key Performance Indicators, such as response times and throughputs. Analyzers predicate over the actual contents of an SDO, e.g., to see whether a response time goes beyond a given threshold.

Planning and Execution. Planning can be considered technology- and application-agnostic, since it only focuses on choosing how to adapt the infrastructural resources that are dedicated to each node in a tier. Now that ECoWare supports containers we can distinguish between two kinds of infrastructural adaptation: we can change either the VMs or the containers being used by an application, or both. Throughout the rest of this paper we will refer to the former in terms of *VM adaptation* and to the latter in terms of *CNT adaptation*; the term *infrastructure adaptation* will be used to refer to both jointly.

CNT adaptation operates directly on an application's containers. It allows for *instant* reaction to workload changes, both through horizontal and vertical container scaling. Horizontal scaling implies the addition or removal of containers; vertical scaling implies increasing or decreasing the amount of resources dedicated to an already existing container.

The delays imposed by VM management are removed. While booting a VM on a public cloud may take more than one minute, adding a container, or changing its resources, is practically instantaneous given the adopted time scale. Since multiple applications that share the same virtualized resources may see different workloads, the finer granularity and the higher adaptation speed of working with containers allow us to use the VMs that we have more efficiently, without necessarily having to add new ones.

Since containers are deployed inside VMs our planner can choose between five kinds of infrastructure adaptations. `create_vm()` allocates a new VM through the cloud provider. `terminate_vm(vm_id)` removes a VM identified by an id (`vm_id`). `create_container(vm_id, node_type, resources)` launches a container of type `node_type` on `vm_id` allocating to it `resources` (CPU cores and memory). `terminate_container(container_id)` removes the container with `container_id`. `scale_container(container_id, new_resources)` changes the resources allocated to `container_id` to `new_resources`.

ECoWare also supports platform adaptation through the use of adaptation callbacks. Whenever ECoWare operates a CNT adaptation a notification is sent to the interested `ECoWare Agents`. Their containers can then react by internally adapting their platform assets in technology- and application-specific ways. This is a novel contribution of this work and will be discussed in detail in Section 5.

[2]The implementation of all the components of ECoWare can be found at: https://github.com/deib-polimi/ecoware.

3.1 Applications

ECoWare needs a thorough description of the self-adapting systems that it is supposed to manage. This includes both a description of the cloud infrastructure to use and a description of the applications being deployed. The latter includes the definition of their tiers and of the dependencies that exist between them. This information is collected in an ECoWare `Applications Description` JSON file inspired by TOSCA [21].

Listing 1 provides a (partial) description of the two applications and of the infrastructure that we used for the evaluation in Section 6. We use it here to illustrate the information that is stored in the file.

Listing 1: Example of an Applications Description file.

```
{
  "infrastructure":{
    "cloud_driver":{
      "name":"aws-ec2",
      "credentials":"/usr/me/utils/aws.properties"
      "vm_flavor" : "t2.small"
      "vm_image_id" : "ami-eff8148f"
    },
    "max_vms":10
  },
  "apps":[
    {
      "name":"RUBiS",
      "tiers":{
        "loadbalancer":{
          "name":"Front Load Balancer",
          "max_node":"1",
          "docker_image":"haproxy",
          "depends_on":[
            "app_server"
          ],
          "on_dependency_scale":"/usr/me/
                utils/reload_server_pool",
          "max_rt":0.1
        },
        "app_server":{
          "name":"Application Logic Tier",
          "docker_image":"polimi/rubis-jboss",
          "depends_on":[
            "db"
          ],
          "on_node_scale":"/usr/me/utils/
                jboss_hook",
          "on_dependency_scale":"/usr/me/utils/
                reload_connections",
          "max_rt":0.6
        },
        "db":{
          "name":"Data tier",
          "max_node":"1",
          "docker_image":"mysql",
          "on_node_scale":"/usr/me/utils/
                mysql_hook",
          "max_rt":0.2
        }
      }
    },
    {
      "name":"Pwitter",
      "tiers" : { ... }
    }
  ]
}
```

The `infrastructure` JSON object describes the infrastructure layer on which the applications will be deployed. Its attribute `cloud_driver` contains the `name` of the provider, the `credentials` needed to access the provider, and the type (`vm_flavor`) and image id (`vm_image_id`) of the VM instances to use. Attribute `max_vm` sets a limit to the allocable VMs to avoid infinitely scaling the application and producing too high a bill.

In our example we used Amazon EC2 `t2.small` VM instances; each has 1 CPU core and 2GB of memory. For our image id we used an Amazon Machine Image identifier[3]. This VM image is configured to launch both Docker and an `ECoWare Agent` as soon as the VM has finished booting.

Each application is identified by a `name` and its tiers are described using attribute `tiers`. Each tier has a `name`, a maximum number of nodes permitted within that tier (`max_node`), a reference to the Docker image to be used for its nodes (`docker_image`), a list of the dependencies that the tier has with other tiers (`depends_on`), and any additional meta-data about its adaptation strategy. In our example, we do not want the load balancer to be replicated, so we set its attribute `max_node` to 1; the image for the load balancer tier is called `haproxy`; and the load balancer manages a pool of application servers, so it declares a dependency with that tier. Currently we support one-to-one, one-to-many, and many-to-one dependency relationships between tiers, where by *one* and *many* we mean the number of nodes in each tier. We do not support cyclic dependencies for the correctness of our processing model (see Section 5.1).

Each tier also specifies what adaptation callbacks it supports and when they should be executed. This is stated through two types of `Adaptation Hooks`. A container's `on_node_scale` hook is invoked every time the container is scaled at run time; a container's `on_dependency_scale` hook is invoked after something changes on a tier that the container's tier depends on. For example, a system administrator may write an `on_node_scale` hook to scale the number of workers of an application server when the resources allocated to the relevant container change. Similarly, s/he may write an `on_dependency_scale` hook to have the load balancer change its routing policy depending on the resources allocated to the servers.

Finally, the tier also declares the Service Level Agreement (SLA) that we want to guarantee using attribute `max_rt`, which refers to the maximum acceptable average response time passed to the planner.

4. PLANNING

This section describes the control-theoretical design path followed to obtain the planner algorithm. Our goal was to develop a decentralized solution, in which each application tier is endowed with a local controller devoted to maintaining a desired performance metric (i.e., response time) in the presence of exogenous disturbances, by computing the resources (i.e., CPU cores and memory) that must be made available to that tier. Once this has been done, we translate the computed resource allocations into concrete adaptation actions, taking into account the current state of the system.

4.1 Controlled System

We shall now start by formalizing the hypotheses used to derive the model for the controlled system, in control-theoretical terms.

Hypothesis 1. In any steady-state situation the metric is a function of the assigned resources and of some disturbance input that reflects the system's "load". This function does

[3]http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

not need to be linear but it must be regular enough to be inverted (at least) in well defined regions of the resource/load space. We assume that the steady-state response time depends on the ratio between the number of assigned cores and the request rate. Of course, the steady-state response time may also depend on the available memory but we focused on cores as they inherently provide a better granularity. Memory is either sufficient for the application (hence assigning more is useless) or insufficient. In the latter case the performance degradation will depend on many fine-grained facts (e.g., caches, swap system, disk, etc.) making the actuator-to-metric relation more difficult to model.

Hypothesis 2. The static function that we are describing acts on the metric through an asymptotically stable, linear, time-invariant dynamic system with unity gain and relative degree. This means that once the resource and the load stay constant, the metric will eventually reach its corresponding steady-state value, but not immediately. For example, this can be the case when acquiring a new core yields its response time improvement as some queue gets emptied. The hypothesis of unity relative degree means that the effects of an action on the assigned resource start showing up in the metrics from the control instant immediately after the action is applied.

Under these hypotheses, the evolution over the discrete time index k of the response time $\tau_r(k)$, as an effect of the assigned cores $c(k)$ and the request rate $r(k)$, is ruled by the following nonlinear, time-invariant dynamic system

$$\begin{cases} \widetilde{u}(k) &= f\big(c(k)/r(k)\big) \\ \tau_r(k) &= p\tau_r(k-1) + (1-p)\widetilde{u}(k-1) \end{cases} \tag{1}$$

that corresponds to the block diagram of Figure 3.

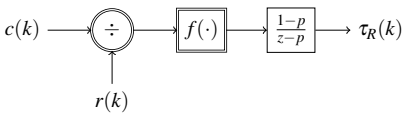

Figure 3: Block diagram of the dynamic model for the controlled system.

In this particular case, function $f(\cdot)$ is intuitively monotonically decreasing towards a possible lower horizontal asymptote, as it can be assumed that once the parallelism degree of an application is fulfilled by the available cores, adding new ones causes no further decrease in the response time. More specifically, we found a practically acceptable function to be

$$f\left(\frac{c(k)}{r(k)}\right) = c_1 + \frac{c_2}{1 + c_3 \frac{c(k)}{r(k)}} \tag{2}$$

where parameters c_1, c_2, and c_3 were obtained through profiling.

4.2 Control Synthesis

The model structure of Figure 3 suggests a controller that is the compound of a lineariser plus a linear, time-invariant feedback regulator. Such a scheme is shown in Figure 4. The key point is to select function $f_C(\cdot,\cdot)$ in such a way that $\widetilde{u}(k) = u(k)$, thereby leading the controller to see just a linear block. This means setting

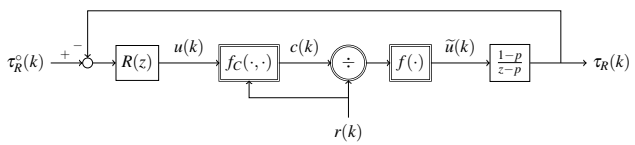

Figure 4: Block diagram of the dynamic model for the closed-loop control system.

$$f\left(\frac{1}{c(k)} f_C\big(u(k), r(k)\big)\right) = u(k). \tag{3}$$

which, assuming $f(\cdot)$ invertible at least in the signal range of interest, gives

$$f_C\big(u(k), r(k)\big) = r(k) f^{-1}\big(u(k)\big). \tag{4}$$

With the selected $f_C(\cdot,\cdot)$, the directed relationship from $u(k)$ to $\tau_r(k)$ reduces to the $\mathbb{Z}$-domain transfer function $(1-p)/(z-p)$, thus the relationship between $\tau_r^\circ(k)$ and $\tau_r(k)$ also reduces to the $\mathbb{Z}$-domain transfer function

$$\frac{T_r(z)}{T_r^\circ(z)} = \frac{R(z)\frac{1-p}{z-p}}{1 + R(z)\frac{1-p}{z-p}}. \tag{5}$$

As frequently done in other control domains, we conduct the selection of $R(z)$ by prescribing the transfer function from set point to controlled variable.

The target for this transfer function must have a relative degree of at least one for realizability reasons. We additionally select a unity gain to ensure asymptotic set point tracking and disturbance rejection and use the single pole α —chosen in the range $(0,1)$— to require a faster ($\alpha \to 0$) or slower ($\alpha \to 1$) error convergence. In summary, we set

$$\tau_r^\circ(k) \frac{T_r(z)}{T_r^\circ(z)} = \frac{1-\alpha}{z-\alpha} \tag{6}$$

and, solving for $R(z)$, we obtain

$$R(z) = \frac{\alpha - 1}{p - 1} \frac{z - p}{z - 1}. \tag{7}$$

The eigenvalues of the obtained control system are the prescribed one $\alpha \in (0,1)$ and the cancelled one p, in the range $(-1,1)$ in force of the stability properties we assumed for the controlled system. Hence, the closed-loop system is guaranteed to be asymptotically stable.

To turn (7) into the control algorithm required by the planner, we first rewrite (7) to highlight its direct feedthrough and strictly proper dynamics terms, that is

$$R(z) = \frac{\alpha - 1}{p - 1}\left(1 + \frac{1-p}{z-1}\right) \tag{8}$$

and we obtain the controller in state-space form as

$$\begin{cases} x_R(k) &= x_R(k-1) + (1-p)\big(\tau_r^\circ(k-1) - \tau_r(k-1)\big) \\ c(k) &= r(k) f^{-1}\left(\frac{\alpha-1}{p-1}\left(x_R(k) + \tau_r^\circ(k) - \tau_r(k)\right)\right) \end{cases} \tag{9}$$

This provides the control algorithm but it does not manage saturations. When confronted with a large change in the set point, or a sudden and relevant disturbance, the controller may compute an action $c(k)$ that is not feasible,

typically either because it is negative or because it exceeds the maximum number of available cores. If c_{min} and c_{max} denote the minimum and maximum number of cores respectively, $c(k)$ must be clamped within the range, and the state of (9) must be recomputed to maintain consistency with the input and output, that is

$$x_R(k) = \frac{p-1}{\alpha-1} f\left(\frac{c(k)}{r(k)}\right) - \tau_r^\circ(k) + \tau_r(k) \qquad (10)$$

Algorithmically, if we omit initializations and highlight the error $e(k) = \tau_r^\circ(k) - \tau_r(k)$, we have

$$\begin{aligned} e &:= \tau_r^\circ - \tau_r; \\ x_R &:= x_{R_p} + (1-p) * e_p; \\ c &:= r * f_{inv}((\alpha-1)/(p-1) * (x_R + e)); \\ c &:= max(c_{min}, min(c_{max}, c)); \\ x_{R_p} &:= (p-1)/(\alpha-1) * f(c/r) - e; \\ e_p &:= e; \end{aligned}$$

where the "p" subscript denotes "previous" values, i.e., those corresponding to the previous step, while "f" and "f_{inv}" correspond to function $f(\cdot)$ and its inverse, respectively.

Finally, we present a simulation example that shows the controller in action. Figure 5 reports the behavior of a system subject to the load in the top-left plot, while the required response time is the dashed line in the top-right plot. The dark line in the same plot is the actual response time obtained with the model matching the real process, while the light line represents the same results with a quite relevant parametric mismatch (up to 20% in the coefficients of $f(\cdot)$ and up to 10% in p). The bottom plot presents the allocated cores in the nominal (dark line) and parametrically perturbed (light line) cases. Although a formal robustness proof is out of the scope of this paper, and is therefore deferred to future ones, the controller behaves satisfactorily even in the presence of unavoidable model mismatches.

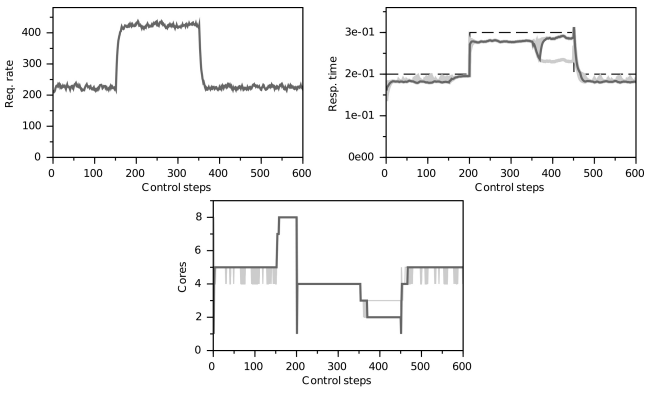

Figure 5: Simulation example to test the synthesised controller.

4.3 From Resource Allocation to Actions

As previously stated, each tier of each application has its own controller. The controllers are all deployed onto a centralized ECoWare Planner node and are synchronized. After each control step the outputs of these controllers are aggregated to create a new resource allocation file. This file contains the number of CPU cores and memory units needed to sustain the current workload for each tier of each application. (One memory unit is equal to 512MB.)

The `Planner Translator` takes as input the new desired resource allocations for each tier of each application and translates them into a mix of infrastructure adaptation actions that must be activated. It uses the actions discussed in Section 3: `create_vm`, `terminate_vm`, `create_container`, `scale_container`, `terminate_container`.

For example, let us imagine that the controllers dedicated to managing a RUBiS application request that the load balancer tier receive 3 cores and 2 memory units, and that the business logic tier receive 5 cores and 3 memory units. In this case, the list of adaptation actions could be to (i) create a new VM; (ii) create a new container on that VM for the business logic tier, with 3 cores and 2 memory units; and (iii) update the container dedicated to load balancing by increasing its resources by 2 cores and 1 memory unit. These actions will depend on the resources already associated with the various tiers at that point in time.

The `Planner Translator` uses the information it receives to search for a new VM and container allocation through the formulation of an Integer Programming problem, i.e., a variation of the two-dimensional *bin packing problem*. The bins are VMs and we must pack the containers inside them. The two dimensions that we take into account are CPU cores and memory.

To guarantee that the problem will terminate we calculate the upper-bound for the number of VMs as the total number of VMs needed to satisfy the desired resource allocation for each tier in each application. This number will depend on the `vm_flavor` being used, which is described in the ECoWare `Applications Description` file. For example, if tier t_1 needs 3 cores, tier t_2 needs 1 core, and we are using a VM flavor that provides 2 cores, then we would need 2 VMs for t_1 and 1 VM for t_2. The upper-bound for VMs would be the sum, that is 3 VMs.

We additionally consider the following constraint: a VM can only host one container per tier of an application. The reason is that we are able to vertically scale containers, thus there is no need to create more than one instance per VM.

Each of the five infrastructure adaptation actions is associated with a weight w, such that $w(terminate\_container) < w(scale\_container) < w(create\_container) << w(terminate\_vm) < w(create\_vm)$. These weights always favor container manipulation over VM manipulation; moreover, termination has a lower weight than creation to prefer the removal of unused resources.

We also add the additional constraint that the sum of the resources that are dedicated to the containers that are deployed on a single VM must be less than, or equal to, the total resources provided by the VM itself (as in the bin packing problem).

By minimizing the weighted usage of containers and VMs, we can find an optimal setup of containers and VMs. We solve the problem using *or-tools*, an ILP solver by Google[4]. Its output is the input for our `Executor`, i.e., the list of infrastructure adaptation actions that need to be performed.

[4]https://developers.google.com/optimization/.

5. EXECUTION

The `Executor` takes the list of actions created by the `Planner Translator` and enacts them using various actuators. These actuators are used to ask the cloud provider to create/terminate VMs, to ask Docker to create/update/terminate containers, and to access the container `Adaptation Hooks` defned in the `Applications Description` file.

When we started our project Docker could not dynamically update the resources allocated to a running container. This led us to extend Docker[5]; however, Docker 1.10.0 (released in February of 2016) now supports an `update` command that fulfills our needs. This command dynamically changes the CPU cores and the memory that are allocated to a container using options `--cpuset-cpus` and `--memory`. The former specifies what CPU cores the container can use, e.g., `--cpuset-cpus="0,2"` states that the container can only use `core 0` and `core 2`. The latter limits the amount of memory that the container can use, e.g., `--memory="512M"` means that the container can only use 512MB of RAM. The same options are also available to the `run` command, which is used to create a new container.

ECoWare's `Executor` node uses three sub-modules: `Topology Graph`, `Topology Manager`, and `Cloud Driver`. `Topology Graph` uses the ECoWare `Applications Description` file to generate and maintain tier-dependency graphs for the applications being managed. `Topology Manager` keeps track of how the applications are deployed on containers and VMs. It also maintains the metadata required to interact with the containers: VM and container ids, IP addresses, ports, etc. Finally, `Cloud Driver` is used to interact with the VM management APIs of the selected cloud provider. Our current implementation of ECoWare supports AWS EC2 as our main public cloud provider and uses Vagrant [15] for simple private cloud setups. However, the system is modular and can support new cloud providers.

`ECoWare Agent`s are pre-installed into the VM image that is specified in the ECoWare `Applications Description` file. The agents ensure that the sets of CPU cores allocated to different containers do not intersect (i.e., each CPU core must be used by only one container), and invoke the `Adaptation Hooks` when specific events occur in the system.

5.1 Adaptation Hooks and Processing Model

`Adaptation Hooks` are the mechanism through which ECoWare keeps the planner technology- and application-agnostic, yet still offering the capability of performing platform adaptations.

`Adaptation Hooks` are bash or python scripts that are mounted directly onto their related Docker containers. These scripts are launched by the `ECoWare Agent`s using the Docker `exec` command. The `on_node_scale` hook takes as input the previous and the new resource allocations: `old_cpu_cores`, `new_cpu_cores`, `old_mem_units`, `new_mem_units`. The `on_dependency_scale` hook takes as input the `old_tier_state` and the `new_tier_state`. These parameters contain metadata about the old and the new states of the nodes in a tier that the container depends on (e.g., the IP address of the nodes, the resources allocated to the nodes, etc.).

The `Executor` identifies the order of action execution. It starts by executing all the VM adaptation actions (i.e., VM creates and terminates) through `Cloud Driver`. These actions are not application specific and they take a relatively long time to execute. As soon as the new VMs (if any) are up and running, the CNT adaptation actions are executed through the `ECoWare Agent`s installed on each machine.

The order in which the CNT adaptation actions and `Adaptation Hooks` are executed depends on the dependencies stored in the Executor's `Topology Graph` sub-module.

If a tier t_1 depends on tier t_2, we start by executing the CNT actions of t_2, while t_1 waits. The `ECoWare Agent`s in t_2 proceed to execute their assigned CNT adaptation actions. If the CNT adaptations they must execute are a `create_container` or a `scale_container`, once that action has been completed the agent proceeds to execute that container's `on_node_scale` hook.

When all the CNT adaptation actions and `on_node_scale` hooks for all the containers in tier t_2 have completed, adaptation on t_1 can be re-activated, but only if t_1 does not depend on any other tiers that are still performing adaptation.

Adaptation for t_1 proceeds in the following order. Its `ECoWare Agents` start by executing the CNT adaptation actions for the containers of t_1, proceed to execute the `on_node_scale` hooks for those containers, and conclude by executing their `on_dependency_scale` hooks. This is possible since all the tier's dependencies have been previously resolved. Once completed, if there are any tiers that, in turn, depend on t_1 they will activate their adaptation, and so on.

6. EVALUATION

We evaluated our work by using two Web applications: RUBiS and Pwitter. RUBiS [14] is a well-known Internet application benchmark that simulates an on-line auction site. Our deployment of RUBiS uses three tiers: a load balancer, a scalable pool of JBoss application servers that run Java Servlets, and a MySQL database. Pwitter was developed in-house[6]. It is a simple Twitter-like social network that stores pweets, i.e., texts that are not limited in length and that are indexed by the polarity of their sentiment. Pwitter is written in Python and is also a 3-tier application. It has a load balancer, a scalable pool of Gunicorn application servers, and a Redis database.

For our experiments we implemented two application-specific sensors; they capture the data required to calculate the average response time of a JBoss Application Server and of a Gunicorn Application server, respectively. We also implemented two application-specific `Adaptation Hooks` to enable platform adaptation; these hooks dynamically reconfigure JBoss and Gunicorn to better exploit the available resources following industry best practices [8–10]. For example, best practices for Gunicorn suggest that the number of workers be $(2 \times num\_cores) + 1$.

The goal of our experiments was to maintain the average response time of the application servers below a certain threshold. After a profiling phase, we set the threshold (SLA) to 0.6 seconds for both applications. Indeed, both are able to sustain this value —under various kinds of workloads— using from 1 to 10 cores (i.e., the maximum amount of resources that we can afford for the experiments). The experiments answer the following questions:

[5]The forked version of Docker is available at: https://github.com/deib-polimi/ecoware-docker

[6]Pwitter is available at https://github.com/deib-polimi/pwitter.

- *Question 1*: if we only use VMs, and no containers, will ECoWare perform better or worse than the current state of practice (i.e., AWS AutoScaling)?
- *Question 2*: if we take into account containers, will ECoWare perform better or worse than with VMs only?

Performance is evaluated using the *core* × *second* metric, i.e., we calculate how many cores are used during the experiments. The lower the value, the better the adaptation works. To simulate a varying workload we used JMeter [11].

Question 1. We compared ECoWare against Amazon EC2's AutoScaling capabilities. We made this choice because it is, in practice, the most widely adopted scaling solution. For both applications we focused on adapting the business logic tiers. We created two AutoScaling groups on EC2, one per application. These groups were configured to scale from 1 to 10 `t2.small` VM instances (each instance had 1 CPU core and 2GB of memory). Both AutoScaling groups were configured to use Amazon Elastic Load Balancers. The databases were deployed on `m4.xlarge` instances (each instance had 4 CPU cores and 16GB of memory). With this setup and workload the databases were over-allocated and never became the bottleneck during the experiments.

The EC2 AutoScaling groups were each given two policies. The first added 1 VM instance when the average CPU utilization, for the entire group, was over 90% for 1 interval of 1 minute. The second removed 1 VM instance when the average CPU utilization, for the entire group, was below 40% for 1 interval of 1 minute. These thresholds were chosen following real-world best practices [12]. We also decided to set the duration of the adaptation control to be as fast as possible, to allow the AWS AutoScaling system to be as quick as possible to react to workload changes. Note, however, that Amazon's AutoScaling does not activate a scaling action if another action is executing. This means that, for example, when a new VM is added we must wait for it to turn on, boot-up, and be linked to the load balancer.

We measured that, on average, it took a VM 150 seconds to complete its boot-up process; this was evaluated from the launch of the VM creation command to when it had successfully booted and linked to the Elastic Load Balancer. This has two intertwined consequences. First, to preserve the unity relative degree hypothesis (see Hypothesis 2 in Section 4.1), the control period had to be longer than this delay. Second, unless an unacceptably long period was selected, a conservative control action (i.e., an alpha close to one) was required to prevent oscillations or even instability. This is not equally relevant if containers are used, allowing for a lower alpha and for a more aggressive control action and stronger disturbance rejection. For these reasons we parameterized our planner as follows: we set the control interval to 180 seconds, we measured the response time and averaged it over 30 seconds, we set the planner to always read the most up-to-date measured response time before computing the next plan, and we set the single pole alpha to 0.95. We also defined the set point of the planner to be 0.5 seconds, i.e., about 10% less than the SLA. This way the planner would have been able to range near the set point without violating the SLA. Finally, we stimulated the two applications with three different workloads: *low*, *medium*, and *high*. All the experiments lasted 75 minutes; the number of users for each interval (in minutes) is shown in Table 1. We repeated the experiments five times.

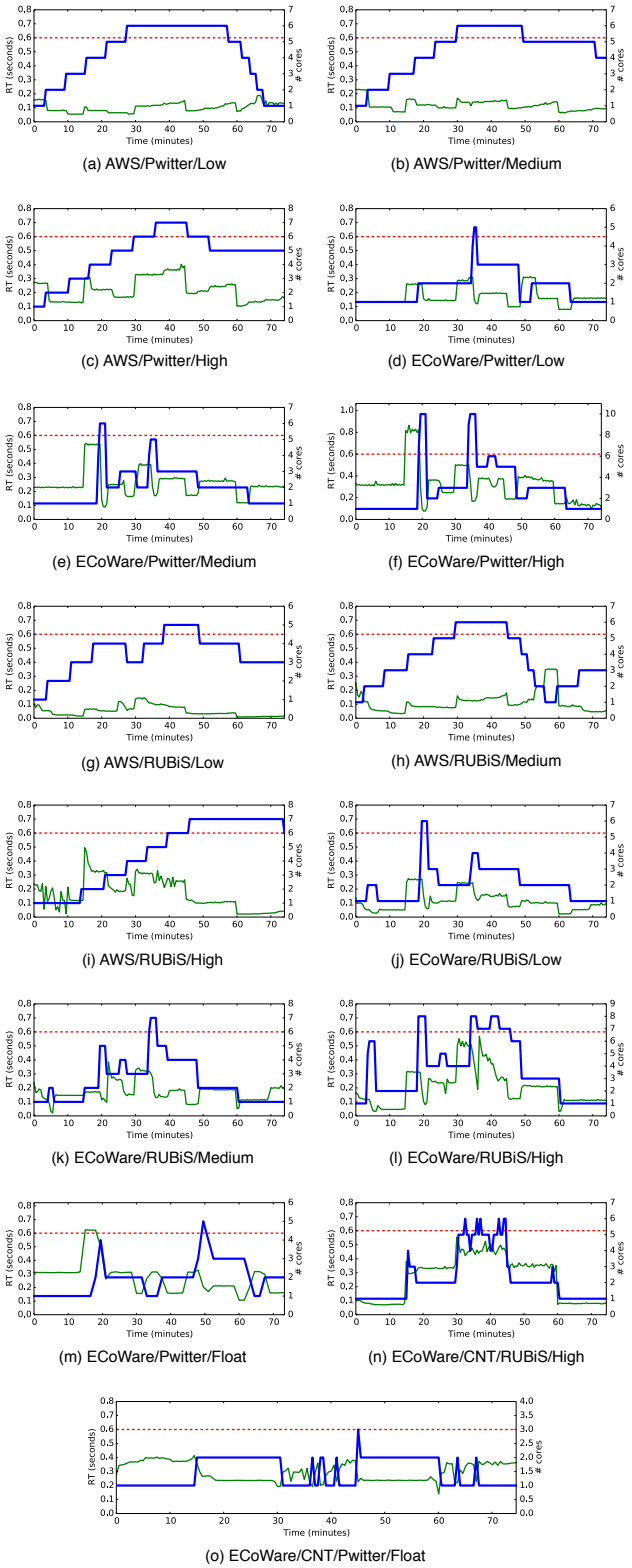

Figure 6: Obtained results. (CNT stands for Containers).

Table 1: Workloads: number of users per time interval (in minutes).

Experiment	0-15	15-30	30-45	45-60	60-75
Pwitter/Low	10	20	40	20	10
Pwitter/Med	15	35	60	35	15
Pwitter/High	20	50	100	50	20
RUBiS/Low	50	200	400	200	400
RUBiS/Med	100	300	600	300	100
RUBiS/High	100	500	1000	500	100
Pwitter/Float	20	40	20	40	20

Figures 6(a)-(c) show AWS AutoScaling's behavior with Pwitter stimulated with the three different workloads. This behavior is compared against the experiments illustrated in Figures 6(d)-(f), which used ECoWare's adaptation capabilities. The horizontal dotted line is the SLA and the thin line is the response time; they both refer to the left y-axis. The thicker line represents the allocated cores, and refers to the right y-axis. The metric *core* × *second* corresponds to the area under the thicker line (cores).

AWS AutoScaling clearly over-allocates resources. This occurs even for *low* workloads and even if we used a very high threshold for CPU utilization. This is because both JBoss and Gunicorn tend to use nearly 100% of their allocated CPUs, even with moderate workloads. Furthermore, this value is also affected by how quickly a spike in the number of users is reached.

ECoWare, on the other hand, allocates less cores on average. The SLA is only violated with the *high* workload for around 5 minutes, which is understandable since the approach is reactive. After a large spike ECoWare over-allocates resources for one control interval and then slowly deallocates them and converges to a stable value, as can be seen in Figure 6(d) at minute 35 and in Figures 6(e)-(f) at around minutes 20 and 35. Though the two example applications are completely different, we see similar results in Figures 6(g)-(l), which focus on RUBiS. ECoWare never incurs in violations, while AWS over-allocates resources in all the scenarios. AWS can only add a static number of VMs when reacting to an increment in CPU utilization, so it is easy to find a workload (with high spikes) in which AWS is too slow to allocate resources, causing various SLA violations to occur. We did not find it useful to show such a case, given the lack of space. Table 2 shows the results of these experiments using the *core* × *second* metric. ECoWare outperforms AWS by 107% on average when managing Pwitter and by 49% on average when managing RUBiS.

Question 2. With the second set of experiments we wanted to assess the benefit of using containers, instead of focusing solely on VMs. To do this we compared ECoWare, as used previously, against a new deployment that used an Amazon `m4.2xlarge` VM instance (8 cores and 32GB of memory). On this machine we installed the dockerized versions of the business logic tiers of the two applications[7]. Our hypothesis was that the two applications had different workloads, which means that there had to be moments in which one application would request *a lot* of resources, while the other would not. This allows us to exploit the advantages of using containers. We only used one VM in our experiments to make the benefits of using containers emerge clearer.

[7]Dockerized RUBiS can be found at: https://hub.docker.com/u/polimi/.

Table 2: AWS vs ECoWare without containers. Values are in CPU *core* × *second*.

Experiment	AWS	ECoWare	Gain
Pwitter/Low	18810	7920	138%
Pwitter/Med	19965	9090	120%
Pwitter/High	20970	12930	62%
RUBiS/Low	15390	8970	72%
RUBiS/Med	16320	10830	50%
RUBiS/High	20265	16290	24%

Table 3: ECoWare with VMs vs ECoWare with containers. Values are in CPU *core* × *second*.

Experiment	VMs	Containers	Gain
Pwitter/Float	8685	6580	32%
RUBiS/High	16290	10210	60%

The use of containers allows ECoWare to have a faster control rate since creating, updating, and terminating containers can be done in milliseconds (around 4 to 6 ms). We parametrized the planner with a control interval of 20 seconds, we measured the response time and averaged it over 10 seconds, we set the planner to always read the most up-to-date measured response time before computing the next plan, and we set the single pole alpha to 0.45. We then simulated two different workloads for Pwitter and RUBiS simultaneously. For RUBiS we used the *high* workload, for Pwitter we used the *float* workload (see Table 1).

Figure 6(m) shows that ECoWare without containers violated the SLA on Pwitter and allocated a peak of 5 cores at minute 50. Figure 6(o) shows that ECoWare with containers never violated the SLA and remained quite close to the optimal resource allocation (1 core for 20 users, 2 cores for 40 users). Similar results emerged with RUBiS. Working with VMs only (Figure 6(l)) required allocating up to 8 cores, while the use of containers (Figure 6(n)) allowed us to allocate no more than 6 cores (the optimal allocation for 1000 users is 5 cores). This is due to the different parametrization of the planner, and to the different control intervals. Table 3 shows the results in detail. If we aggregate the two experiments, ECoWare with containers outperforms ECoWare with VMs by 46%.

To conclude, the experiments show that a high level of CPU utilization does not always equate to a saturated system. Indeed, both JBoss and Gunicorn continue to operate acceptably (i.e., they continue to satisfy their SLAs), even when their CPU utilization is close to 100% (see Figure 6(a,g)). Furthermore, the experiments also show that, in these two applications, memory saturation is not an issue.

6.1 Threats to Validity

Even though we performed our experiments on two applications built on different technologies, the two are similar both in terms of domain and architectural design. They are both CPU-bound, and in both cases the bottleneck lies in the application tier. Nevertheless, these two applications cover a wide range of real-world cases. Further research will evaluate ECoWare with applications that follow a completely different architectural design (e.g., Map/Reduce, media streaming applications, etc.).

In our experiments we used the cloud infrastructure in two different ways, depending on whether we were dealing with VMs or with containers. When dealing with VMs we used up to 10 EC2 instances per application; each VM had 1 core.

When dealing with containers we used a single EC2 instance with 8 cores and shared it between both applications. The reason for using a VM with 8 cores is that they were enough to satisfy the SLAs, given the used workloads.

The reason for this difference is that to truly take advantage of container technology one must have multiple applications running (with different workloads) on multiple shared VMs. This kind of setup is becoming increasingly common in companies, due to the success of micro-service architectures. Furthermore, to efficiently manage containers, one should have a number of cores (on each shared VM) that is sufficiently large so that CNT adaptations can be effective. Ideally, one should have (at least) a number of cores that is equal to the number of application tiers being managed on that VM plus one.

Memory was not an issue in these experiments. Our controller only computes the amount of cores required per tier and requests an appropriate amount of memory units based on this computation, i.e., the controller requests 1GB of RAM per core. This is always possible because EC2's VMs always provide more GBs of RAM than cores. The importance of memory must be re-evaluated with applications that are memory-bound; this will be part of our future work.

7. RELATED WORK

The solution presented in this paper finds "competitors" in both industry and academia.

Amazon recently presented the EC2 Container Service [2], which allows one to deploy Docker containers inside an EC2 VM cluster. ECoWare is more open and flexible: it provides full access to Docker to build the `Adaptation Hooks` mechanism and allows for the vertical scaling of containers.

Apache Mesos [3] is a cluster manager that supports Docker containers. It is a middleware platform that hides the complexity of a distributed infrastructure and provides application level APIs for many services (e.g., Hadoop, Spark, and Elastic Search) for resource management and for scheduling. Google Kubernetes [7] is another container cluster manager. It offers a comprehensive solution for deploying applications using containers and offers an API for scaling and replicating groups of containers. Both Mesos and Kubernetes focus on the infrastructure layer and lack comprehensive multi-level adaptation. ECoWare may embed these solutions as alternative ways to manage containers and it could complement them with advanced adaptation capabilities. Cloudify [4] can only drive infrastructure auto-scaling based on a simple rule-based solution very similar to Amazon Autoscaling.

As for research initiatives, Brun et al. [23] witness the importance of control loops and autonomic computing when dealing with complex software systems, while Dustdar et al. [27] provide an interesting definition of elastic computing. They advocate that elastic processes should be modeled on three metrics: resources, cost, and quality. Our work focuses on resources and cost, and partially on quality. Our main goal is to keep response time under a certain threshold, while *quality* provides more facets and ECoWare could be further extended to support them in the future.

Different approaches deal with the problem of resource allocation. For example, Hu et al. [33] propose a resource allocation system for cloud computing. They render application environments (AEs) by means of queuing models and use a global arbiter to allocate resources among AEs. Ardagna et al. [18] present a game theoretical approach for the runtime management of the infrastructure of a IaaS provider that rents resources (VMs) to multiple SaaS providers to fulfill their SLAs. They formulate the Generalized Nash Equilibrium problem and prove the existence of at least one equilibrium. Padala et al. [46] present a hierarchical controller that allocates resources to multiple multi-tier applications in data centers. They show how the controller can improve the utilization of servers and help applications meet their SLAs. Our approach is different with respect to these works, and to other similar ones (e.g., [25, 37, 42, 44, 53]), because of the use of containers and the ability to exploit platform adaptation. Furthermore, our solution adopts the perspective of cloud users, with no access to the policies that regulate the behavior of the hypervisor.

Finally, as for feature adaptation, Klein et al. [39] introduce the Brownout adaptation paradigm based on optional code that can be dynamically (de-)activated through a controller. As already said, we are confident that our `Adaptation Hooks` mechanism can also support feature adaptation and provide a similar solution, but this is not the focus of this paper.

8. CONCLUSIONS AND FUTURE WORK

The paper proposed a novel self-adaptation framework based on: (i) VM- and container-based resource management, (ii) coordinated infrastructure and platform adaptation, and (iii) a planner that implements a discrete-time feedback controller. Experiments demonstrate that fine-grained adaptation capabilities can greatly improve performance when autoscaling Cloud-based Web applications.

Our future work comprises the integration of feature adaptation by extending the adaptation hook mechanism, an extension of the planner to make it work hierarchically with respect to the controlled resources, an even finer-grained solution to control the CPU cores allocated to a container, and further evaluation on more case studies of different kinds.

Acknowledgments

We would like to thank Lorenzo Affetti and Dmitrii Stebliuk for their contributions, and Danilo Ardagna for his most valuable feedback. The work presented in this paper has been partially supported by project EEB - Edifici A Zero Consumo Energetico In Distretti Urbani Intelligenti (Italian Technology Cluster For Smart Communities) - CTN01_00034_594053.

9. REFERENCES

[1] Amazon EC2 Autoscaling. https://aws.amazon.com/autoscaling/.
[2] Amazon EC2 Container Service (ECS). https://aws.amazon.com/ecs/.
[3] Apache Mesos. http://mesos.apache.org.
[4] Cloudify. Cloud Orchestration and Automation Made Easy. http://getcloudify.org.
[5] Docker. https://docker.com.
[6] Docker: Container Security White Paper. https://www.docker.com/sites/default/files/WP_Intro%20to%20container%20security_03.20.2015%20(1).pdf.
[7] Google Kubernetes. http://kubernetes.io.
[8] Gunicorn Documentation: How Can I Change the Number of Workers Dynamically?

http://docs.gunicorn.org/en/stable/faq.html#how-can-i-change-the-number-of-workers-dynamically.

[9] Gunicorn Documentation: How Many Workers? http://docs.gunicorn.org/en/stable/design.html#how-many-workers.

[10] JBoss Perfomance Tuning Guide. https://www.redhat.com/f/pdf/JB_JEAP5_PerformanceTuning_wp_web.pdf.

[11] JMeter. https://jmeter.apache.org.

[12] Netflix: AutoScaling in the Amazon Cloud. http://techblog.netflix.com/2012/01/auto-scaling-in-amazon-cloud.html.

[13] RabbitMQ. Robust Messaging for Applications. http://rabbitmq.com.

[14] RUBiS. The Rice University Bidding System. http://rubis.ow2.org.

[15] Vagrant. https://www.vagrantup.com.

[16] Announcing Amazon EC2 Container Service (ECS) - Container Management for the AWS Cloud. https://aws.amazon.com/blogs/aws/cloud-container-management, 2014.

[17] Microsoft - Windows Containers. https://msdn.microsoft.com/en-us/virtualization/windowscontainers/about/about_overview, 2016.

[18] D. Ardagna, B. Panicucci, and M. Passacantando. A Game Theoretic Formulation of the Service Provisioning Problem in Cloud Systems. In *Proceedings of the International Conference on World Wide Web*, pages 177–186. ACM, 2011.

[19] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[20] L. Baresi and S. Guinea. Event-Based Multi-level Service Monitoring. In *Proceedings of the 20th International Conference on Web Services*, pages 83–90, 2013.

[21] T. Binz, G. Breiter, F. Leyman, and T. Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(3):80–85, 2012.

[22] P. Bodık, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson. Statistical Machine Learning makes Automatic Control Practical for Internet Datacenters. In *Proceedings of the 2009 conference on Hot topics in Cloud Computing*, pages 12–12, 2009.

[23] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering Self-adaptive Systems through Feedback Loops. In *Software Engineering for Self-adaptive Systems*, pages 48–70. Springer, 2009.

[24] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. CloudSim: a toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[25] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive Software needs Quantitative Verification at Runtime. *Communications of the ACM*, 55(9):69–77, 2012.

[26] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 103–116. ACM, 2001.

[27] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong. Principles of Elastic Processes. *IEEE Internet Computing*, (5):66–71, 2011.

[28] EsperTech. Complex event processing. http://esper.codehaus.org, 2010.

[29] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An Updated Performance Comparison of Virtual Machines and linux Containers. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 171–172. IEEE, 2015.

[30] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, Model-driven Autoscaling for Cloud Applications. In *Proceedings of the 11th International Conference on Autonomic Computing*, pages 57–64. USENIX Association, 2014.

[31] Google Cloud Platform. An Introduction to Containers, Kubernetes, and the Trajectory of Modern Cloud Computing. https://cloudplatform.googleblog.com/2015/01/in-coming-weeks-we-will-be-publishing.html, 2015.

[32] P. Hoenisch, S. Schulte, S. Dustdar, and S. Venugopal. Self-adaptive Resource Allocation for Elastic Process Execution. In *Proceedings of the Sixth IEEE International Conference on Cloud Computing*, pages 220–227. IEEE, 2013.

[33] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu. Resource Provisioning for Cloud Computing. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 101–111. IBM Corp., 2009.

[34] P. Jamshidi, A. Ahmad, and C. Pahl. Autonomic Resource Provisioning for Cloud-based Software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 95–104. ACM, 2014.

[35] P. Jamshidi, A. M. Sharifloo, C. Pahl, A. Metzger, and G. Estrada. Self-Learning Cloud Controllers: Fuzzy Q-Learning for Knowledge Evolution. In *Proceedings of the 2015 International Conference on Cloud and Autonomic Computing*, pages 208–211. IEEE, 2015.

[36] JavaWorld. How Containers Change Everything. http://www.javaworld.com/article/2940858/cloud-computing/how-containers-change-everything.html, 2015.

[37] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang. Joint VM Placement and Routing for Data Center Traffic Engineering. In *Proceedings of the 31st IEEE International Conference on Computer Communications*, pages 2876–2880. IEEE, 2012.

[38] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[39] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez. Brownout: Building more Robust Cloud Applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711. ACM, 2014.

[40] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai. Performance Model Driven QoS Guarantees and Optimization in Clouds. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 15–22. IEEE Computer Society, 2009.

[41] X. Liu, X. Zhu, S. Singhal, and M. Arlitt. Adaptive Entitlement Control of Resource Containers on Shared Servers. In *Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management*, pages 163–176. IEEE, 2005.

[42] X. Meng, V. Pappas, and L. Zhang. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In *Proceedings of the 2010 IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2010.

[43] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, 2014.

[44] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proceedings of the 2013 USENIX International Conference on Automated Computing*, 2013.

[45] A. Y. Nikravesh, S. A. Ajila, and C.-H. Lung. Towards an Autonomic Auto-scaling Prediction System for Cloud Resource Provisioning. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 35–45. IEEE Press, 2015.

[46] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 289–302. ACM, 2007.

[47] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski. Introducing STRATOS: A Cloud Broker Service. In *Proceedings of the Fifth IEEE International Conference on Cloud Computing*, pages 891–898. IEEE, 2012.

[48] Rackspace. Carina by Rackspace Simplifies Containers with Easy-To-Use, Instant-On Native Container Environment. http://goo.gl/Khf84h, 2015.

[49] W. Rankothge, J. Ma, F. Le, A. Russo, and J. Lobo. Towards Making Network Function Virtualization a Cloud Computing Service. In *Proceedings of the 13rd IFIP/IEEE International Symposium on Integrated Network Management*, pages 89–97. IEEE, 2015.

[50] S. Schulte, D. Schuller, P. Hoenisch, U. Lampe, S. Dustdar, and R. Steinmetz. Cost-driven Optimization of Cloud Resource Allocation for Elastic Processes. *International Journal of Cloud Computing*, 1(2):1–14, 2013.

[51] F. Seracini, M. Menarini, I. Krueger, L. Baresi, S. Guinea, and G. Quattrocchi. A Comprehensive Resource Management Solution for Web-based Systems. In *Proceedings of the 11th International Conference on Autonomic Computing*. USENIX, 2014.

[52] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, volume 41, pages 275–287. ACM, 2007.

[53] Z. Xiao, W. Song, and Q. Chen. Dynamic Resource Allocation using Virtual Machines for Cloud Computing Environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1107–1117, 2013.