

# Robust Performance Control for Web Applications in the Cloud

Corina Stratan, Guillaume Pierre, Hector Fernandez, Eliana Tirsia and Valentin Cristea

Vrije Universiteit Amsterdam and University Politehnica of Bucharest

## 1 Introduction

Over the last few years, we have witnessed numerous debates on blogs, conferences and media on whether to use *grid computing* or *cloud computing*. Even though some of the early promises of grid computing, sound the same as the promises of the cloud, cloud efforts were focused on applications which will be long-running on the infrastructure. As a result, today’s most well-known enterprises use cloud infrastructures as a tool on which deploy their applications. Similarly, one key properties of the clouds is *elasticity*, *i.e.*, the possibility for cloud users, to dynamically adapt the quantity of resources at his disposal at run-time. Unfortunately, this notion of infinite resource availability has a financial cost, as cloud users have to pay for the resources being used from moment to moment.

**CORINA:** *Not sure if we should bring grid computing into discussion, the readers might not be familiar with it (and I think we should aim to shorten the paper).*

Therefore it seems crucial to choose the best cloud provisioning offers in order to minimize the resource consumption, and thereby the running costs while assuring several performance requirements for our applications. Nowadays, there are lots of research papers dedicated to sophisticated techniques on dynamic resource provisioning. However, if we look at real deployments we see that cloud providers rely on extremely simple techniques, and completely ignore the results from academic research on the topic. Two reasons can be the explanations for this discrepancy: *(i)* the gains of using sophisticated provisioning strategies are too low for anyone to bother; *(ii)* imple-

menting these techniques is a difficult exercise, which is why real cloud systems rely on simpler techniques.

y match allocated capacity to such observed workload changes. While long-term variations can be handled by manually allocating server capacity to an application, short-term fluctuations require an ability to dynamically provision capacity to react to workload changes in an agile and timely manner. Recently a number of dynamic capacity provisioning techniques have been proposed in the literature [24, 25, 26, 6]. Most of these techniques track or predict changes in the workload [24, 25] and then provision sufficient capacity to handle this expected volume of requests.

**CORINA:** *I think the statement “completely ignore the results from academic research” is a bit too bold. I would say something like the real deployments tend to use very simple strategies rather than the sophisticated techniques proposed by researchers.*

This paper tries to identify the real cause. We do this by implementing a sophisticated provisioning system in realistic conditions, and reporting on *(i)* how hard implementation was; and *(ii)* potential gains from using the better technique as compared to a simple strawman. To achieve this, we designed and implemented these resource provisioning strategies on ConPaaS, an open source platform-as-a-service environment for hosting cloud applications [13]. For the sake of comparison and discussion, we deployed the Mediawiki application and used real access traces to validate our technique, opening doors to real implementation of slightly sophisticated auto-scaling systems.

, we argue that provisioning server capacity based on the expected volume of requests may not be suf-

cient to accurately capture the service demands of a web application. In particular it has been observed that in real production applications ranging from enterprise applications to large e-commerce sites, workload is highly variable and the request mix exhibits nonstationarity [22, 26]. C

Workload fluctuations seen by Internet applications can be caused by changes in the volume of incoming requests or by changes in their mix; frequently, workload fluctuations are accompanied by both types of changes. It is clear that a significant change in the request volume requires changes in the provisioned capacity; for example, if the request rate doubles, the application needs twice as much server capacity. A change in the workload mix (i.e., the relative frequencies of different request types) is a more subtle phenomenon that has an equally important impact on capacity provisioning; however, the impact of such non-stationary workloads has not been adequately addressed in the research literature. We argue why the workload mix matters using two illustrative examples

CORINA: *Maybe not; we could just make a summary phrase out of that (e.g. we used more sophisticated methods based on trend detection, dynamic weight adjustment for load balancing etc.).*

Section 2 introduces the ConPaaS runtime environment. Section 3 describes the application and its realistic benchmark to validate our system. Section 4 focus on the different resource provisioning strategies implemented in ConPaaS. Section 5 details the experimental campaign and its results. Section 6 discusses related works. Section 7 draws a conclusion.

## 2 ConPaaS overview

ConPaaS is an open-source runtime environment for hosting applications in Cloud infrastructures [13]. Within the Cloud computing paradigm, ConPaaS belongs to the platform-as-a-service family, in which a variety of systems aim to simplify the deployment of applications in the Cloud. Using ConPaaS, developers can now focus their attention on application-

specific concerns rather than making their applications suitable for the cloud.

### 2.1 Architecture

In ConPaaS, an application is designed as a composition of one or more elastic and distributed components, called *services*. Each service is dedicated to host a particular type of functionality of an application. At the moment, ConPaaS supports six different types of services: two web application hosting services respectively specialized for hosting PHP and JSP applications; a MySQL database service; a NoSQL database service built around the Scalarix key-value store; a MapReduce service; and a Task-Farming service for high-performance batch processing.

These services are built based on an architecture composed of two main building blocks: agents and managers.

CORINA: *Maybe include here one of the diagrams that we have, that show the architecture of ConPaaS?*

- **Agent:** A service is composed of one or several agents VMs which host the needed components to provide the service-specific functionality. As an example, the PHP web hosting service needs at least three agents: one http proxy, one web server and one PHP server.
- **Manager:** For each service, there is only one manager VM. The manager is in charge of executing all management requests, centralizing governance and performance monitoring data, and controlling the allocation of resources assigned to one service. The actual application traffic is not addressed to the manager. Requests from end users willing to access the application must be directed directly to the agents hosting the application.

CORINA: *We should point out that the agents can be on a single VM (for small applications) or on multiple VMs (for larger applications).*

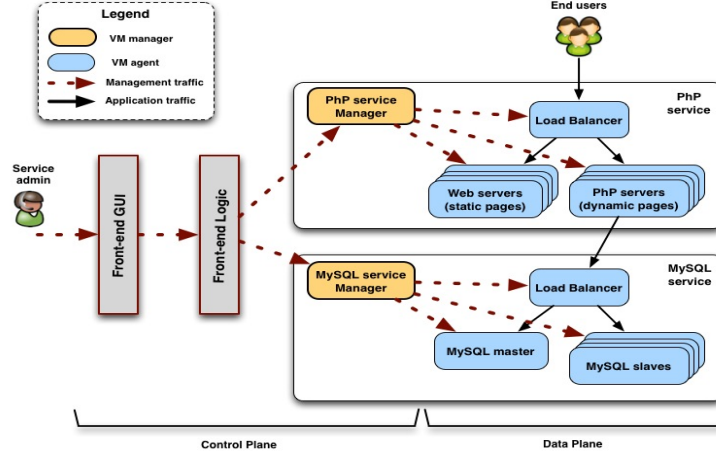


Figure 1: ConPaaS system architecture

## 2.2 Hosting Elastic Applications

The main features that distinguish ConPaaS from other PaaS systems are its approach for autonomous application scaling and its interoperability with a wide variety of private and public IaaS clouds. In particular to provide such autonomous scaling capabilities, ConPaaS includes a monitoring data analysis mechanism and a resource provisioning system.

ConPaaS incorporates a scalable distributed monitoring engine which is based on the Ganglia [5] monitoring system. Ganglia consists of a server component (gmetad) that aggregates monitoring statistics from various VMs, and a reporting agent (gmond) which runs inside each VM. By default, Ganglia monitors only system-level metrics such as disk, CPU, memory and network usage. Unfortunately, these metrics often do not provide enough information about system performance due to the heterogeneity of the applications. As a consequence, in ConPaaS, we enhanced ganglia to also monitor service workloads by enhancing the reporting agent to track service-specific logs at runtime, and report statistics over a reporting period of 5 minutes. For instance, the PHP web hosting service includes new ganglia metrics that report statistics about the response time and request rate for static and non/static user requests, respec-

tively. Once the monitoring data is collected from the agents VMs, the resource provisioning algorithm decides whether to trigger scaling operations based on this data.

Unlike of implementing traditional trigger-based provisioning systems that scale services independently of whether they are part of an application. In ConPaaS, we designed a performance control model for multi-service applications. Our model takes into consideration the fact that services may have different roles in an application, so that services collaborate to guarantee several performance requirements. It improves the effectiveness and accuracy of the scaling decisions. Indeed, this allows to rapidly detect performance bottlenecks in applications, and thereby to minimize the resource consumption.

## 3 The Wikipedia workload

To evaluate the behavior and accuracy of ConPaaS when hosting web applications, we prepared a realistic and complex enough scenario to assess any PaaS. In particular, we deployed the Wikipedia web application called MediaWiki [18], and used a web hosting benchmark called WikiBench [4].

The architecture of the Wikipedia website uses a http-proxy, a http-web server, a database and one or more PhP servers. To deploy the Wikipedia services on ConPaaS, we composed two different services: the PHP web hosting and MySQL service. In the MySQL service, we installed a complete copy of English Wikipedia database which contains about 30GB in Wikipedia articles. In the PhP service, an initial configuration was composed of one Nginx http-proxy, one Apache server, and one or more PhP (FastCGI Process Manager) servers. Each PhP server hosts the MediaWiki application which is the main component of this system.

In order to benchmark ConPaaS when hosting the Wikipedia services, we used the WikiBench research tool which generates realistic benchmarks with adaptable traffic properties. WikiBench has a number of advantages compared to the existing benchmark tools for web applications. First of all, WikiBench traces add a high degree of realism, since it is entirely based on the Wikipedia software and data. Indeed, the benchmark workloads are generated based on real access traces from the Wikimedia Foundation. These traces contain detailed traffic logs of requests made to Wikipedia by its users. Since the original Wikipedia traces can reach peaks of 50000 or 60000 reqs./secs, WikiBench uses the original 10% sample of these traces which can generate a workload up to about 5000 reqs./secs. As an example, in Figure 2, we show the workload of one trace, named "test", as the number of PhP requests per minute during approximately 24h. In this paper, we focus on the behavior of PhP requests which makes particularly difficult to predict their execution times using auto-scaling systems.

Even though we use a 10% of the real traces, they are very heterogeneous in terms of workload-mix, and thus explaining the irregular performance pattern followed by web applications. To illustrate this heterogeneity, in Figure 3, we present the distribution of the response time values for the PhP requests during the execution of the trace "test". Note that, static provisioning was utilized to execute this trace. A first observation shows an irregular dispersion of the response time values in two levels: (i) a long-term level on which the values vary along the execution

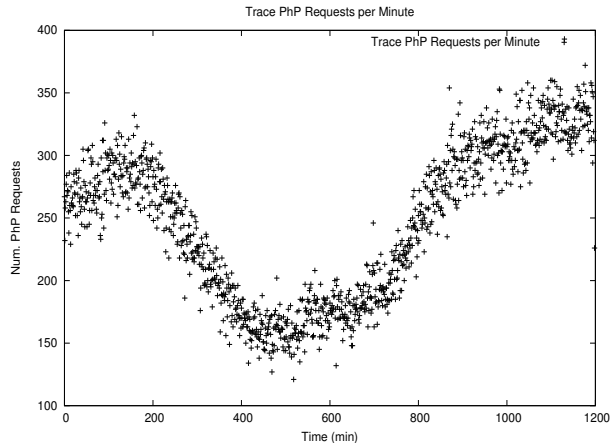


Figure 2: Wikipedia trace workload

without following any pattern; and (ii) a short-term level on which the response times widely diverge under short period of time such as a minute. There are two reasons for this dispersion: (i) PhP pages often require database queries; (ii) PhP pages need third-party static files. These issues avoids the utilization of provisioning techniques which scale applications only based on recent response values. Similarly, as shown in Figure 2 and Figure 3, there is not any correlation between the PhP request volumes and their response times. More precisely in Figure 3, the highest response time values obtained in the interval of time between 200 and 300min. match up with a drop in the request rate during the same interval in Figure 2. Therefore, any provisioning technique that makes decisions based on the request rate can incur errors by under- or over-provisioning an application, which drastically reduces the efficiency of the scaling actions.

Other properties may be considered important of the Wikipedia workload are:

- The intervarrival time between requests follows a Poisson distribution.
- The distribution of page popularity varies from very popular pages to those being accessed very infrequently.

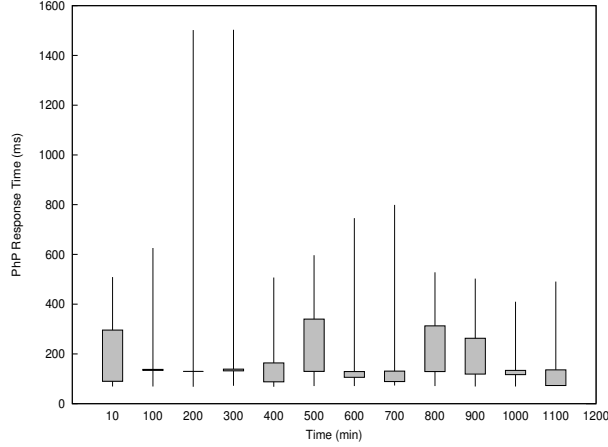


Figure 3: Complexity of PhP requests

- The mix of static/dynamic requests presents a strong variation.
- The ratio of read/write operations vary having more reads than editions or creations of wiki pages.
- A considerable amount of requests for non-existing pages and files add realism to the traffic.

Since Wikipedia has a variable amount of data and visitors, it represents a valid example of elastic web applications. In this paper, we focus in the scalability of the PhP web hosting service, and thereby as the number of PhP servers hosting MediaWiki scale out or back based on the demanding workload.

## 4 Resource provisioning algorithms

In this section, we describe the different provisioning techniques implemented in ConPaaS.

### 4.1 Load-based provisioning

The existing infrastructures offer provisioning mechanisms that adjust the amount of resources based on

the load of the currently allocated resources. Within this group of provisioning mechanisms, we include the simple trigger-based systems that define threshold rules to increase and decrease the computational power of an application in order to guarantee several performance requirements. As an example, the Auto Scaling system offered by Amazon EC2 [2] scales out an application whenever its CPU usage in the last 10min exceeds a specific threshold (Amazon recommends to establish an upper bound of 70%). This rules-based technique is currently used in mature cloud platforms such as RightScale [14] or OpenShift [12].

For the sake of comparison, we decided to design and implement a trigger-based provisioning mechanism in ConPaaS, called "load-based provisioning". This algorithm monitors CPU usage and response time metrics, and dynamically adjusts the computational power of an application by analyzing whether the monitoring data exceed their thresholds, as illustrated in Algorithm 1. Obviously, the lower and upper bound of each threshold are pre-defined by the user before execution.

---

#### Algorithm 1: Load-based

---

**Data:**  
System-level metrics (CPU, Resp. time)  
- Pre-defined metric threshold ranges <sup>1</sup>, *thr*

**Result:** Scaling decisions

```

while auto-scaling is ON do
  Collect monitoring data of each metric, data;
  while no recent scaling operation do
    if avg(datai) >= thri then
      | ADD resources;
    else if avg(datai) < thri then
      | REMOVE resources;
    end
  end
end

```

---

Even though these type of mechanisms are simple and widely used in cloud platforms, they are excessively reactive and not so precise when provisioning web applications due to several factors:

- **Workload mix and web traffic:** In web applications, the system performance behavior fluctuates following an irregular pattern caused by the

traffic and its workload mix, which increase the complexity to predict future fluctuations.

- **Reactiveness:** An excessively reactive system can affect the system performance when handling flash crowds or slashdot effects. A high frequency of scaling operations can provokes sharp and sudden fluctuations that affect the performance instead of improving it. So, it is particularly difficult to decide when to scale out or back an application.
- **Services as black boxes:** Services are handled as black boxes, the definition of threshold rules often only covers system-level metrics such as response time and CPU usage. Therefore, when provisioning web applications, metrics such as request rate of static/dynamic files may be also taken into consideration to improve the accuracy of the decisions.
- **Heterogeneity:** The performance of virtual instances provided by current clouds is largely heterogeneous, even when requesting the exact same type of instance each time, as stated in [3].

Based on these factors, we believe trigger-based provisioning mechanisms can be optimized without drastically increasing its complexity. A solution seems to be the utilization of techniques that handle web traffic and workload mix without being excessively reactive. Moreover, the implementation of these techniques have to remain a simple exercise to facilitate its integration in existing auto-scaling systems. In the following we present two techniques that aims at solving the aforementioned drawbacks by relying on predictive and more accurate methods.

## 4.2 Weighted-metric feedback provisioning

Based on our previous knowledge from load-based provisioning, we designed and implemented an algorithm which improves the accuracy of our scaling actions when hosting web applications. To achieve that, our algorithm relies on three simple mechanisms: the definition of weights to each metric included in the

performance requirements, the use of flexible thresholds and the estimation of the workload trend.

**Weighted metrics:** Traditional algorithms would scale out and back whenever a system-level metric exceeds its beforehand defined threshold range. Nevertheless, through the definition of weight values to application and system metrics, our algorithm takes into consideration its weight when making scaling decisions. More precisely, when hosting web applications, our algorithm associates weights in an ascending order to the following metrics: request rate, CPU usage and response time. Accordingly the response time has a higher weight than the request rate, since higher values in the response time rapidly indicate the existence of a performance degradation in a web application. Indeed, scaling decisions only make based on the request rate can incur errors by under- or over-provisioning a web application, due to the large diversity in the complexity of the requests [?].

---

### Algorithm 2: Weighted-metric feedback

---

**Data:**

System and App-level metrics  
 - Pre-defined metric threshold ranges, *threshold*  
 Define the weight of each metric, *w*

**Result:** Scaling decisions

Create a queue to store historical workload, *q*;  
 Establish flexible thresholds from *threshold* ranges;  
 - Scalings operations could be triggered, *pred\_thr*;  
 - Scalings operations must be triggered, *reac\_thr*;

**while** *auto-scaling is ON* **do**

Collect monitoring data of each metric, *data*;

**if** *avg(data<sub>i</sub>) ≥ pred\_thr<sub>i</sub>* **then**

  | Increment chances of *scaling\_out* using *w<sub>i</sub>*, *s\_out*;

**else if** *avg(data<sub>i</sub>) < pred\_thr<sub>i</sub>* **then**

  | Increment chances of *scaling\_in* using *w<sub>i</sub>*, *s\_in*;

**end**

Add to *q* the most recent workload value;

Estimate historical workload trends (last ~20min), *td*;

  - If trend is increasing then *td* = 1;

  - If trend is decreasing then *td* = 0;

  - Undetermined *td* = -1;

**while** *no recent scaling operation* **do**

**if** *avg(data<sub>i</sub>) ≥ reac\_thr<sub>i</sub>* **and** *td* = 1 **and** *s\_out* >

*s\_in* **then**

      | ADD resources;

**else if** *avg(data<sub>i</sub>) < reac\_thr<sub>i</sub>* **and** *td* = 0 **and** *s\_out* <

*s\_in* **then**

      | REMOVE resources;

**end**

**end**

**end**

---

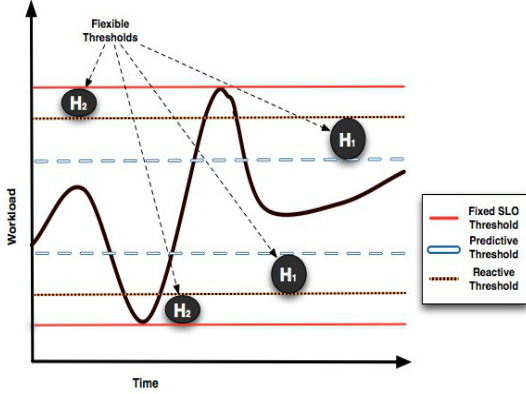


Figure 4: Flexible thresholds

**Flexible thresholds:** This algorithm uses two levels of threshold ranges for each metric: *predictive* and *reactive*. As shown on Figure 4.2, there are two "head rooms" between the SLA threshold (pre-defined by the user) and the flexible thresholds. First, the head-room  $H_1$  is between the predictive threshold and the reactive thresholds, and is intended to alert of possible workload alterations in advance. Thereby, when the system performance exceeds the predictive ranges, there is an increment in the chances of scaling actions will be triggered to tackle future SLA violations. The second head-room  $H_2$  comprises between the SLA and reactive thresholds is used to trigger scaling actions. Performance values exceeding the reactive threshold launch scaling operations if other conditions are also satisfied. As an example of flexible thresholds for the CPU-usage, we established a predictive range comprised between 30% and 70% , and a reactive comprised between 20% and 80%.

**Workload's trend estimation:** Nowadays, there is a wide literature on mathematical models that try to predict future alterations in web application's workload. However, the workload mix and network traffic of web applications make more difficult to provide accurate predictions using these models. Besides, the complexity of these models sometimes prevent its integration in real auto scaling systems. To design a robust and simple provisioning system, we

decided to use a feedback mechanism that analyzes the behavior of the system performance during an interval of time. An exhaustive analysis of the monitoring data during a small interval of time (approx. during the last 20min) provides enough information to detect the workload's trend, and thereby to classify the type of workload alteration as *constant* or *temporal*. Obviously, only *constant* variations may trigger scaling actions to avoid frequent fluctuations in the system performance caused by short and sudden *temporal* variations (flash crowds), as we will detail in Section 5.

Nevertheless, the use of these three mechanisms must follow an order when making scaling decisions, as illustrated in Algorithm 2. Initially, the user has to specify the thresholds ranges and determine a weight to each metric. Next the flexible thresholds are defined based on Amazon recommendations and statistically-chose performance measures<sup>2</sup>. Once the monitoring data is collected from the agents, the decision making process can start. Firstly, these data is analyzed to verify if it exceeds the predictive threshold ranges (denoted by  $p\_thr$ ), if so the probability of triggering scaling actions increases proportionally in function of the metric's weight. In order to keep track of workload variations, this algorithm stores in a queue (denoted by  $q$ ) the most recent system performance values, and analyzed them to estimate the workload trend (denoted by  $td$ ). Finally, to trigger any scaling actions, several conditions have to be satisfied: (i) no previous scaling actions have been taken over the last 15min; (ii) the monitoring data have to exceed the reactive threshold ranges; (iii) the workload trend has to follow a constant pattern (increasing/decreasing); and (iv) the result of predictive thresholds evaluation has to match up with that obtained in (ii).

Even though the combination of these techniques improves the accuracy of our measurements, and avoids an excessive reactive behavior. The heterogeneous nature of the VM instances requires more dy-

<sup>2</sup>These performance values are obtained from previous executions of the same application using similar hardware configurations

namic provisioning algorithms, as we pointed out in [8].

fixmeworkload mix makes more difficult to predict the workload

### 4.3 Workload mix-aware provisioning

The heterogeneity of cloud platforms, and therefore, of their VMs affect to the accuracy of the provisioning decisions. VMs with better hardware configuration can sustain higher workload intensities. In addition, the mixture of static/dynamic requests included into the workload of web applications makes more difficult to distribute these requests across multiple backend servers. Most of existing web load-balancer systems provide simple methods which do not consider the workload mix. Hence, methods such *asround-robin* distributes the requests according to the servers with respect of its server weight; and the *least connections* method which distributes requests to the server with the least connections. Unfortunately, these load-distribution methods are not so accurate when having a large diversity in the complexity of the requests.

As a solution, the workload mix-aware algorithm proposes to use a dynamic-weight load balancing method in conjunction with the weighted-metric feedback algorithm. By using this dynamic load-balancing mechanism, each backend server has a weight value which is dynamically adjusted based on its monitoring data, and thereby based on its own performance behavior. This mechanism allows to distribute requests across the servers taken into consideration the current workload intensity and the server throughput, which vary depending of its hardware configuration. As illustrated in Algorithm 3, this algorithm assigns the same weight to each backend servers at the beginning of the process, and progressively adjusts their weights (every  $\sim 15\text{min}$ ) depending on the monitoring data collected from each backend. By doing so, the load-balancing takes into account the complexity of the served requests and inherently the server throughput improving the distribution.

---

#### Algorithm 3: Workload mix-aware

---

**Data:**  
System and App-level metrics  
- Pre-defined metric threshold ranges, *threshold*  
Define the weight of each metric, *w*

**Result:** Scaling decisions

Create a queue to store historical workload, *q*;  
Establish flexible thresholds from *threshold* ranges;  
- Scalings operations could be triggered, *pred\_thr*;  
- Scalings operations must be triggered, *reac\_thr*;  
Initialize load-balancing weights for the backend servers;

**while** *auto-scaling is ON* **do**  
Collect monitoring data of each metric, *data*;  
**if** *data<sub>i</sub> >= pred\_thr<sub>i</sub>* **then**  
| Increment chances of *scaling\_out* using *w<sub>i</sub>, s\_out*;  
**else if** *data<sub>i</sub> < pred\_thr<sub>i</sub>* **then**  
| Increment chances of *scaling\_in* using *w<sub>i</sub>, s\_in*;  
**end**  
Add to *q* the most recent workload value;  
Estimate historical workload trends (last  $\sim 20\text{min}$ ), *td*;  
- If trend is increasing then *td* = 1;  
- If trend is decreasing then *td* = 0;  
**while** *no recent scaling operation* **do**  
**if** *data<sub>i</sub> >= reac\_thr<sub>i</sub> and td = 1 and s\_out >= s\_in*  
**then**  
| ADD resources;  
**else if** *data<sub>i</sub> < reac\_thr<sub>i</sub> and td = 0 and s\_out < s\_in*  
**then**  
| REMOVE resources;  
**end**  
**end**  
Adjust load-balancing weights based on the workload  
( $\sim 15\text{min}$ );  
**end**

---

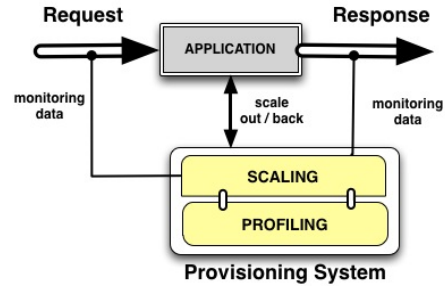


Figure 5: Profiling Resource Provisioning



## 5 Experimental evaluation

In this section we conducted our experiments on a heterogeneous infrastructure like Amazon EC2 [2] and on a homogeneous infrastructure like DAS-4 (the Distributed ASCI Supercomputer 4) [1]. DAS-4 is the Dutch Computational Infrastructure, a six-cluster wide-area distributed system designed with research purposes. We compare the degree of SLA fulfillment and resource consumption for each one of the provisioning algorithms included in ConPaaS.

Note that, since DAS-4 is a homogeneous infrastructure, the profiling techniques were only evaluated on a heterogeneous platform like Amazon EC2.

**FIXME: Include all the assumptions ??**

CORINA: *I guess only the important ones (but what kind of assumptions were you thinking about?*

**FIXME: I did not mention the size of our checking window: 5min, 5min also**

CORINA: *I think we should mention that.*

### 5.1 Testbed configuration

As a realistic and representative scenario, we deployed Mediawiki application using ConPaaS on both clouds, and we ran the Wikibench tools utilizing Wikipedia workload traces.

To provide the Wikipedia services, an initial configuration was composed of 4 VMs, and 1 VM to host the Wikibench tools. The 4 VMs include a PHP service manager, a FPM-PHP agent, a web server and a http-proxy agent (in the same VM), and a MySQL service to store the English Wikipedia data, as explained in Section 3.

Thus, the provisioning system will scale out and back the number of VMs hosting FPM-PHP agents to guarantee the SLO (Service Level Objective). Initially, we fixed two SLOs one of 700ms (milliseconds) at the service's side and another of 1500ms at the client's side. Our measurements shows the behavior of the Wikipedia services during 24h under a workload generated from real access traces. Note that, our experiments only focus on the average of PHP response time and the resource consumption of our

algorithms. The response time of static file requests is not evaluated due to the lightweight nature of the static files employed by Wikipedia articles.

### 5.2 Homogeneous Infrastructure

Our experiments on DAS4 relies on OpenNebula as Infrastructure-as-a-service (IaaS). To deploy the Wikipedia services, we used small instances for the PHP service (manager and agents) and a medium instance for the MySQL service (agent). OpenNebula's small instances provision VMs equipped with 1 CPU of 2Ghz, and 1GiB of memory, while medium instances are equipped with 4 CPU's of 2Ghz, and 4GB of memory.

Figure 6 and Figure 7 depict the degree of SLA fulfillment for the naive and history-aware algorithms, indicating the average of response times obtained during the execution of the Wikipedia workload trace. The results show how the naive provisioning algorithm tends to generate more SLA violations due to its excessive reactive behavior. These violations are comprised between *700ms* (see the yellow Line) and *1500ms* (see the red Line) response time values. As we mentioned, this algorithm is an easy target to flash crowds effects, as new VMs can be added or removed to handle sharp and sudden variations in the workload. In contrast on Figure 7, the system performance (*i.e.*, response time) do not fluctuate greatly showing a more stable behavior during the whole experiment. As well we also appreciate a reduction in the number of SLA violations regarding the total amount of PHP-served pages.

Nevertheless to better understand the behavior of both algorithms, we may focus on the resource consumption, as depicted on Figure 8. Firstly, the excessive reactive behavior of the naive algorithm is again illustrated at the interval  $t=350min$  and  $t=820min$ , where two scaling operations under-provision the system during a short period of time. These provisioning decisions provoked fluctuations in the system performance that incremented the financial cost, and therefore throughput alterations. When using the history-aware algorithm, the system makes provisioning decisions by analyzing workload's trend during a considerable interval of time. Scaling actions are

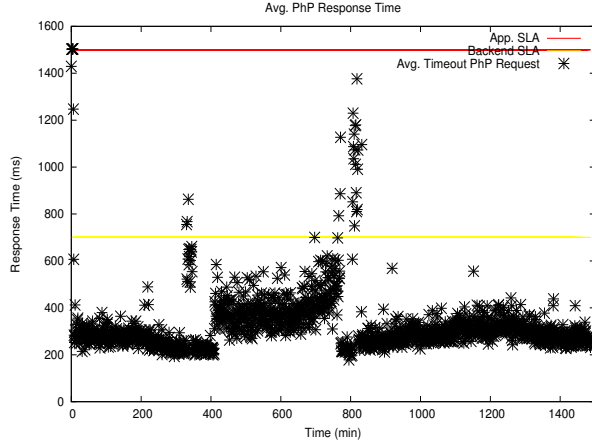


Figure 6: PhP resp. time on DAS4 – Naive.

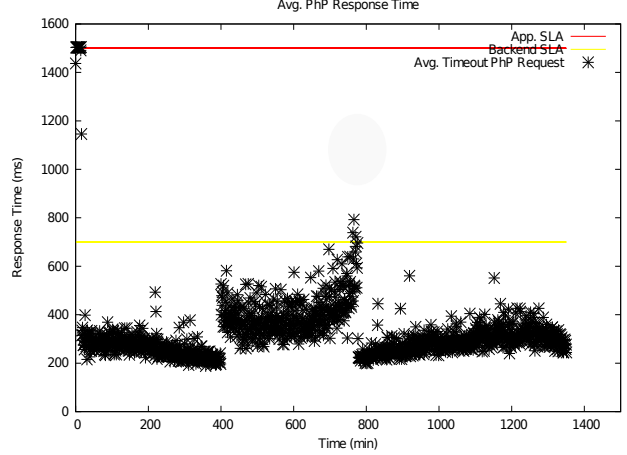


Figure 7: PhP resp. time on DAS4 – History-aware.

only triggered when having constant alterations in the Wikipedia workload, providing a more efficient resource usage. Indeed, the workload alterations depicted on Figure 2, match with the provisioning decisions made by the history-aware algorithm on Figure 8.

**FIXME: I will run more time the History-aware exp. on DAS4.**

### 5.2.1 Discussion

Using the naive provisioning algorithm, the system performance fluctuates greatly following a pattern similar to the web traffic that increases the number of SLA violations. The reactive behavior of this algorithm triggers scaling actions that affect to the system performance instead of improving it, and as a consequence they are also wasteful in terms of resource consumption. Unlike history-aware algorithm offers an efficient resource usage and a constant performance behavior while meeting the application's SLA. Therefore this algorithm finds the tradeoff between accuracy and cost savings.

Both algorithms are best-effort regarding the SLA fulfillment, and thereby temporal alterations of the workload (with a duration of 5min approx.) cannot be guaranteed. The heterogeneity of the PhP-served

pages, containing images and requiring multiple Db queries, and the startup time of VMs are in part responsible of these SLA violations.

## 5.3 Heterogeneous Infrastructure

Our experiments on EC2 used small instances for the PhP service (manager and agents) and a medium instance for the MySQL service (agent). EC2 small instances provision VMs equipped with 1 EC2 CPU, and 1.7GiB of memory, while medium instances are equipped with 2 EC2 CPU's, and 3.75GiB of memory.

In the following, we analysis the behavior of our algorithms when making provisioning decisions on a heterogeneous infrastructure. Figure 9, Figure 10 and Figure 11 show the system performance of the naive, history-aware and profiling-based algorithms, respectively. As depicted on Figure 9, the performance fluctuates greatly following an irregular pattern when using the naive algorithm. More precisely, two of the three workload peaks caused at  $t=300min$  and  $t=820min$ , are explained from the variations on the Wikipedia workload described on Figure 2. However, there is a third peak that corresponds to the interval of time on which the workload trace significantly decreases the number of user's requests. This

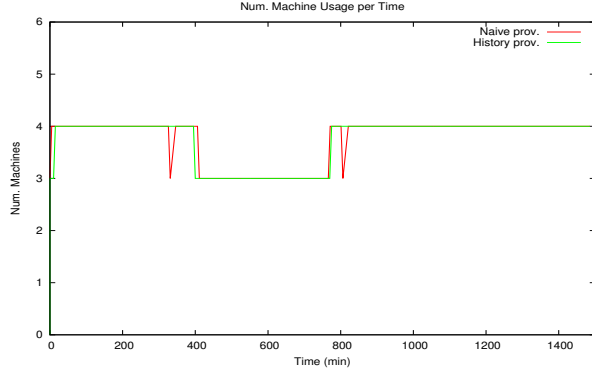


Figure 8: Resource consumption on DAS4.

explains the degradation of the SLA fulfillment as an effect associated to very frequent scaling actions.

On the other hand, Figure 10 and Figure 11 show as the history-aware and profiling-based algorithm behave similarly. Even though both algorithms are best-effort, there is an important reduction in the number of SLA violations during the whole experiment. Like on DAS-4, the history-aware algorithm follows a constant performance pattern without having sharp and sudden alterations. Besides, as shown on Figure 11, the profiling-based algorithm has a similar behavior to the history-aware algorithm in terms of system performance, however. The profiling-based algorithm improves the SLA fulfillment in a 11% in comparison with the history-aware algorithm. Therefore we demonstrate how the use of online profiling techniques although intrusive do not cause time delays or throughput alterations.

**FIXME: performance or responsiveness**

The resource usage on EC2 presents important alterations as shown on Figure 12. When using the naive provisioning, the fluctuations in the system performance are explained as a result of a high frequency of scaling operations. In concrete, these fluctuations caused at the interval of time comprised between  $t=400min$  and  $t=500min$  (see on Figure 9) match with the provisioning decisions made during the same interval of time on Figure 12. If we now pay attention

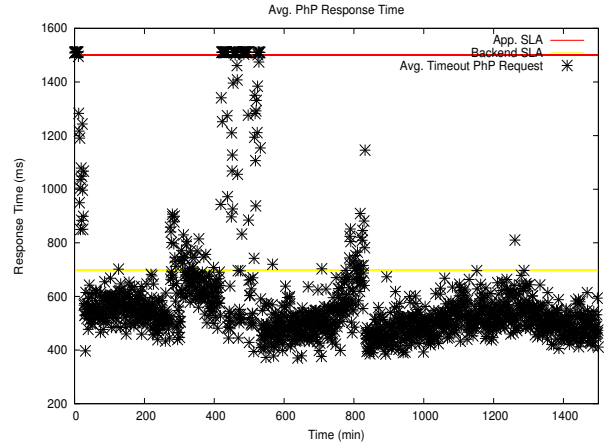


Figure 9: PhP response time on EC2 – Naive.

to the history-aware, and profiling-based algorithms, their resource consumptions are identical during the whole experiment. Indeed, both algorithms decided to scale out the system during the interval of time comprised between  $t=1050min$  and  $t=1400min$ , to prevent future SLA violations. It demonstrates the benefits of using flexible threshold ranges to provide a predictive provisioning mechanism, improving the user experience.

### 5.3.1 Discussion

**FIXME: Do you prefer this organization for the discussions ? One inside of each type of cloud, and one to summarize.**

## 5.4 Discussion

Generally, the result of our measurements show how the behavioral performance pattern and the resource consumption vary depending on the infrastructures on which we ran our experiments. Different hardware configurations such as those provided by DAS-4 and EC2, offer two distinct scenarios to validate our provisioning algorithms. In these experiments, we demonstrate how trigger-based provisioning mechanisms can affect the system performance instead of

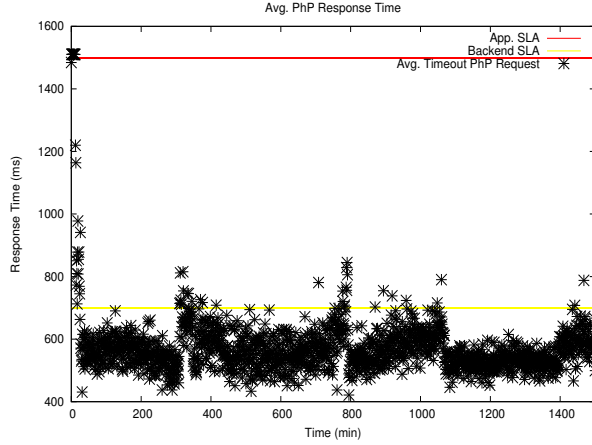


Figure 10: PhP resp. time on EC2– History-aware.

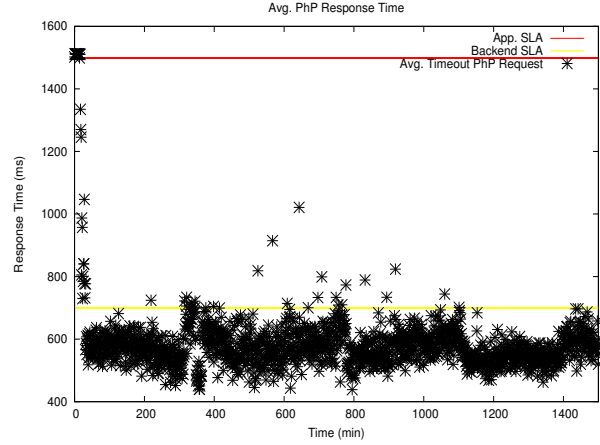


Figure 11: PhP resp. time on EC2– Profiling-based.

improving it, as well as are wasteful in terms of resource usage. Online profiling techniques, although intrusive, were used without producing performance alterations, in fact they slightly reduced the number of SLA violations in comparison with the history-aware algorithm. We also show the benefits by using history-aware and profiling-based provisioning algorithms which find the tradeoff between the accuracy and cost savings.

However, in these experiments, the flexible threshold ranges were pre-defined before execution for all VMs. These threshold values have to change depending the type of instance to be provisioned. Therefore, we believe that offline profiling techniques may be used to define these values depending of the type of instance, thus improving the effectiveness of our predictions.

**FIXME:** We used the same statistically-chose performance threshold for both infrastructures.

**FIXME:** Aggressive provisioning also increases the chances of degraded application performance due to the accompanying increase in violation of performance thresholds.

## 6 Related studies

There is a wide literature on issues related to dynamic resource provisioning for cloud web applications. Different approaches present solutions based on queuing models [16], feedback loops techniques [6], mathematical models [11] or even approaches using neural networks techniques [7]. However, most of these models require a deep understanding in mathematics or machine learning techniques which are not easily interpreted by non specialists. Besides the traffic in web applications is shaped by a combination of different factors such as diurnal and seasonal cycles, sociological and psychological, that follows an irregular pattern. It makes extremely challenging the design and development of realistic and accurate dynamic provisioning mechanisms.

These well-known drawbacks force to IaaS like Amazon EC2 and Windows Azure [10], or PaaS like RightScale [14] and OpenShift [12], to design simple threshold rule-based auto-scaling systems, instead of relying on approaches from academic research. Unfortunately, these scaling systems are naive, wasteful in terms of resource consumption and cost savings, and an easy target for flash crowds.

In the following, we present some of the most relevant and realistic academic approaches that proposed

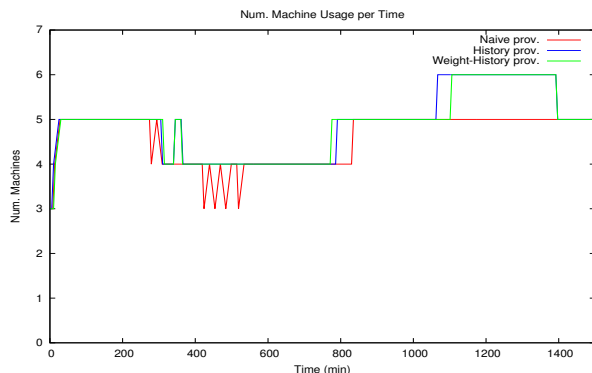


Figure 12: Resource consumption on EC2.

dynamic resource provisioning mechanisms for multi-tier applications.

In [15, 16], the authors designed and implemented a predictive and reactive provisioning mechanism. They used a queuing model G/G/1 to decide the server pool size to be provisioned, and an admission control mechanism to face extreme workload variations. Offline profiling techniques were employed to gather information about the resource requirements of the incoming requests for each tier, and thereby to selectively admit/reject requests for the lightweight files. An evaluation using real-traces on a homogeneous infrastructures shows the benefits of this approach when handling flash crowds. Unfortunately, its admission control mechanism incurs into sporadic SLA violations (if the server utilization exceed a pre-defined threshold) reducing the QoS of the service, and therefore affecting user experience. Similarly to the previous work, [17] extended queuing models and transaction mix models to design a predictive and reactive provisioning system. To model the application performance, they integrated proactive control and feedback control methods that dynamically adjusted the CPU capacity allocated to servers. This work only considered SLA constraints at the system-level, while others constraints at application-specific level such as response time and request rate were not taken into consideration. Besides, an evaluation of CPU variations on a homogeneous infrastructure,

when processing traces from a non real-world application, lack arguments to valid its approach.

Regarding the management of flash crowds [19], a proactive application workload manager was designed to separate the user requests between two groups of servers: one named 'base workload' referred to the smaller and smoother variations in the workload; and the other 'trespassing' referred to the temporal burstly workloads caused by flash crowds. To do this, the authors attempt to divide the data items into popular and less popular, and place them in the right group of servers. Even tough a realistic evaluation was conducted on Amazon EC2 utilizing real traces (Yahoo video streaming), authors do not explain in details how the dynamic resource provisioning is done. Recently, in [9], online profiling techniques have been utilized for managing the tradeoff between performance overload, and cost savings for dynamic resource provisioning. The authors replicate at runtime a regular server hosting an application, with a new server with profiling instrumentation. Their experimental results show how profiling techniques can be included in a resource provisioning system, without causing important response time delays or throughput alterations in comparison with non-profiling provisioning. As we mentioned in Section 5, profiling techniques can report more benefits than performance degradations or expenses.

## 7 Conclusion

Excessive reactive algorithms trend to temporally overprovision applications affected by flashcrowds or slashdot effects. This type of algorithms increase the resource consumption and infrastructure costs than other ...

Experiments based on real-traces "Wikipedia" and conducted on heterogeneous and homogeneous cloud infrastructures.

The use of offline-profiling techniques allow to identify the threshold of the resources of a cloud infrastructure.

The provisioning system remains independent of the infrastructure on which the apps run.

## 8 Acknowledgments

This work is partially funded by the FP7 Programme of the European Commission in the context of the Contrail project under Grant Agreement FP7-ICT-257438.

<b>FIXME: Probably also mention ERRIC here. ??</b>
--

## References

- [1] Advanced School for Computing and Imaging (ASCI). The Distributed ASCI SuperComputer 4. <http://www.cs.vu.nl/das4/>.
- [2] Amazon Elastic Compute Cloud (EC2). <http://aws.amazon.com/ec2>.
- [3] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. EC2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*, November 2009.
- [4] Guido Urdaneta Erik-Jan van Baaren and Guillaume Pierre. Wikibench: A distributed, wikipedia based web application benchmark. 2009.
- [5] Ganglia monitoring system. <http://ganglia.sourceforge.net/>.
- [6] Jiayu Gong and Cheng-Zhong Xu. A gray-box feedback control approach for system-level peak power management. In *2010 39th International Conference on Parallel Processing (ICPP)*, pages 555–564, September 2010.
- [7] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.*, 28(1):155–162, January 2012.
- [8] Dejun Jiang. *Performance Guarantees For Web Applications*. Phd thesis, VU University Amsterdam, March 2012.
- [9] Nima Kaviani, Eric Wohlstadter, and Rodger Lea. Profiling-as-a-service: adaptive scalable resource profiling for the cloud in the cloud. In *Proceedings of the 9th international conference on Service-Oriented Computing, IC-SOC'11*, pages 157–171, Berlin, Heidelberg, 2011. Springer-Verlag.
- [10] Microsoft Windows- Windows Azure. <http://www.windowsazure.com/en-us>.
- [11] Sireesha Muppala, Xiaobo Zhou, Liqiang Zhang, and Guihai Chen. Regression-based resource provisioning for session slowdown guarantee in multi-tier internet servers. *Journal of Parallel and Distributed Computing*, 72(3):362–375, March 2012.
- [12] OpenShift. <https://openshift.redhat.com/app/flex>.
- [13] Guillaume Pierre and Corina Stratan. Con-PaaS: a platform for hosting elastic cloud applications. *IEEE Internet Computing*, 16(5):88–92, September-October 2012.
- [14] RightScale. <http://www.rightscale.com/>.
- [15] Bhuvan Urgaonkar and Prashant Shenoy. Cat-clysm: Scalable overload policing for internet applications. *Journal of Network and Computer Applications*, 31(4):891–920, November 2008.
- [16] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1–39, March 2008.
- [17] Zhikui Wang, Yuan Chen, D. Gmach, S. Singhal, B.J. Watson, W. Rivera, Xiaoyun Zhu, and C.D. Hyser. AppRAISE: application-level performance management in virtualized server environments. *IEEE Transactions on Network and Service Management*, 6(4):240–254, December 2009.
- [18] Wikipedia Foundation. MediaWiki a free software wiki package. <http://www.mediawiki.org>.

- [19] Hui Zhang, Guofei Jiang, Kenji Yoshihira, Haifeng Chen, and Akhilesh Saxena. Resilient workload manager: taming bursty workload of scaling internet applications. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 45–46, New York, NY, USA, 2009. ACM.