2018 IEEE 16th Int. Conf. on Dependable, Autonomic & Secure Comp., 16th Int. Conf. on Pervasive Intelligence & Comp., 4th Int. Conf. on Big Data Intelligence & Comp., and 3rd Cyber Sci. & Tech. Cong.

Multilevel Observability in Cloud Orchestration

Rodolfo Picoreti*, Alexandre P. do Carmo†, Felippe M. de Queiroz*, Anilton S. Garcia*,
Raquel F. Vassallo* and Dimitra Simeonidou‡
*Electrical Engineering Department
Federal University of Espírito Santo, Vitória, Brasil
Email: rodolfo.picoreti@gmail.com, mendonca.felippe@gmail.com, anilton.garcia@ufes.br, raquel@ele.ufes.br
†Electrical Engineering Department
Federal Institute of Espírito Santo, Guarapari, Brasil
Email: alexandre.carmo@ifes.edu.br
‡High Performance Networks Group
University of Bristol, Bristol, United Kingdom
Email: dimitra.simeonidou@bristol.ac.uk

***Abstract*—Cloud infrastructures can provide resource sharing between many applications and usually can meet the requirements of most of them. However, in order to enable an efficient usage of these resources, automatic orchestration is required. Commonly, automatic orchestration tools are based on the observability of the infrastructure itself, but that is not enough in some cases. Certain classes of applications have specific requirements that are difficult to meet, such as low latency, high bandwidth and high computational power. To properly meet these requirements, orchestration must be based on multilevel observability, which means collecting data from both the application and the infrastructure levels. Thus in this work we developed a platform aiming to show how multilevel observability can be implemented and how it can be used to improve automatic orchestration in cloud environments. As a case study, an application of computer vision and robotics, with very demanding requirements, was used to perform two experiments and illustrate the issues addressed in this paper. The results confirm that cloud orchestration can largely benefit from multilevel observability by allowing specific application requirements to be met, as well as improving the allocation of infrastructure resources.**

***Keywords*-cloud computing; observability; intelligent spaces;**

I. Introduction

Modern cloud applications are usually composed of many smaller components forming a distributed system architecture known as microservice oriented architecture. This architecture has many advantages over a monolithic architecture. Services are now small, have a single responsibility and thus are easier to understand. Additionally, services are no longer tightly coupled with the monolith and can be easily changed, scaled, deployed independently from the rest of the application. Therefore orchestrators are able to scale an application more efficiently by scaling the components that are under heavy load, instead of having to scale the whole monolith even when one of its services does not require scaling. However, when decomposing a complex application in many smaller distributed services, the ability to easily troubleshoot it is normally lost [1]–[3].

For instance a request made to a frontend service can trigger requests to many other backend services. In that scenario the response latency of the frontend service will be tightly coupled with the latency of the backend services. Therefore to be able to really understand the system behavior (to have visibility) and for example detect bottlenecks, the interaction between those services must be monitored and analyzed in many levels. Without proper tools that task can be very cumbersome [4].

Consequently for a microservice architecture to be realistically used in a production environment, it is paramount that developers and operators become able to understand how such complex distributed systems behave, thus regaining the ability to troubleshoot it [5]. The ability to visualize the system behavior is usually called observability. Observability is normally obtained in three forms: logs, metrics and traces, which will be addressed in the following sections.

Another aspect that can affect the behavior of an application in a cloud environment is the infrastructure where it is hosted. One of the main advantages of cloud computing is the ability to provision infrastructure on demand and be more cost-efficiently than managing your own infrastructure. Some cloud providers have even more cost-efficient options than on demand provisioning. For instance, with AWS spot instances cloud, users can save up to 90% when compared to the on demand prices [6]. In this category, users choose how much they are willing to pay for a particular compute instance. If the instance price is lower than the one specified by the user, the resource is provisioned. Although, the instance can latter be interrupted if it's price exceeds the user's limit.

In this context, it is possible to notice the interdependence between infrastructure and applications. On one hand, infrastructure requires application metrics in order to provision resources in a cost-efficiently manner. On the order

978-1-5386-7518-2/18/$31.00 ©2018 IEEE
DOI 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00134

hand, applications are heavily dependent on the underlying infrastructure to meet their specific requirements. Therefore, application observability can be mutually beneficial to infrastructure and applications by not only improving the ability of developers to understand their application, but also to be used to better provision infrastructure resources.

Unfortunately, observability is usually implemented just in the infrastructure layer, and in general the observation of the infrastructure layer can not capture important aspects of the application layer. As a consequence, the orchestrator is unable to efficiently allocate resources, often resulting in an under/over-provisioning scenario.

Furthermore, observability should be achieved in different levels: infrastructure (network, compute, storage) and application; so orchestration decisions can be improved in a cloud environment.

An example of a similar work can be observed in [7]. In this paper, the authors describe a system that uses multilevel metrics and machine learning techniques for cloud orchestration. However, the work tries to meet only the applications requirements without looking for the best use of the infrastructure resources. In addition, only the conceptual idea is shown and no tests are performed.

With this is mind, in this article we are going to explore how we achieved observability in the infrastructure and application levels of our platform, and how the collection of such metrics can be used to improve resource orchestration, that is, meet application requirements while improving resource utilization. In addition, we present some experimental tests with real applications to validate the proposed approach.

The next sections are organized as follows. Section II describes the concepts related to observability, observability in cloud platforms and different tools that can be used to implement observability. Section III describes the platform where we deployed observability mechanisms. In Section IV a case study is presented and used as proof of concept to show how multilevel observability can improve resource provisioning by the infrastructure and also meet the application specific requirements. Finally, in Section V, the conclusions and future work are discussed.

II. Observability

While there might be many definitions to the term observability, it usually refers to the ability to visualize and comprehend the behavior of a system by gathering, processing system data and presenting it in a suitable manner for the different type of users, like application developers, system administrator and orchestration systems. This process allows not only the analysis of the present behavior, but also the inference of the future behavior of a system.

The observability of a distributed system can be improved by employing what is commonly referred as the three pillars of observability: logs, metrics and tracing.

A log is an immutable record of a discrete event that happened at some point in time. For instance, an event describing that a server started processing a request. Logs can be classified into unstructured (free form text without a particular schema) and structured (following a particular schema, binary or text). They can also be produced by different sources like applications, operational systems, etc. Logs are usually collected by a unified log aggregation layer to be preprocessed then persisted into a data store for further processing and analysis.

Logs can carry a large volume of data from all the system entities, and therefore can be used to identify almost anything that is considered useful. Because of that, they are frequently used for audit or troubleshooting. However, this has a cost in the performance of the service itself. The higher the amount of logs and the more detailed they are, the more they will impact the service performance under heavy load. In order words, logging overhead increases linearly with number of events.

Metrics are numbers collected periodically forming a time series. Metrics are produced by aggregating events of an entity. Therefore, they can show trends about the behavior of a system. For instance, with metrics we can see the histogram of RPC (Remote Procedure Call) latencies to a specific service or the amount of requests that the service have processed so far. Metrics are normally used to generate alerts.

Alerts are actions triggered by events when a predetermined condition is met. For instance if a system is producing too many error responses on the last minute we can generate an alert on a dashboard or even take some predefined action in order to fix it. To that end, alerts allow situations that require immediate intervention to be monitored, either automatically or through a warning for manual intervention.

Like logs, metrics are collected periodically to a central entity to be processed and persisted. Although, the overhead of collecting metrics is constant with system activity, depending only on the amount of metrics. Therefore, the overhead only increases if the number of monitored elements is increased, for example, with the inclusion of new services or new metrics in an existing services. Furthermore, metrics generate a much smaller volume of data compared to logs.

Metrics and logs provide information about services individually, but it is not enough for applications with multiple services distributed across an infrastructure. To identify the information flow of an application with distributed services and to understand its behavior completely, it is necessary to use tracing.

Traces can show the causality of events in a system and therefore see how one event interacts with another. The cause-effect relation of a trace is very important to help understand how one event propagates through the system.

In order to infer the causality of events in a trace, each service must contribute by propagating a tracing context

along with the usual payload. This context is also sent directly to a tool that will later use it to correlate the chain of events. This last context can also contain useful troubleshooting data like stack traces and other application specific annotations [5].

The overhead of tracing is similar to that of logs, that is, the cost of tracing increases linearly with the number of events being produced.

The three pillars of observability can be employed in two distinct monitoring models: whitebox and blackbox. In whitebox monitoring we have access to the internal state of the system through information provided by itself. In order words, the whitebox model allows the use of metrics, logs and tracing provided by the application.

On the order hand, in blackbox monitoring the system does not provide any information regarding its internal state. The state of the application is defined by observing the relation between the inputs and outputs of the observed element via an external entity, similar to the user experience. So in order to monitor this type of systems one would need to probe or sniff them, for instance. To give an illustration, the average response time of a server can be inferred by probing it regularly. Another solution would be to intercept and analyze all the traffic going to the server.

One recurring approach being used in modern cloud applications is the use of the sidecar pattern. In this pattern a sidecar element is deployed together with one application sharing the same lifecycle. This element provides extra features to the application [8]. For instance, in the observability context, a sidecar proxy can be used to offload instrumentation and common network functionalities from the service [9].

A. *Observability in Cloud Platforms*

Applications developed for a cloud environment can be quite complex due to the interaction between its various components like applications and the infrastructure where they are hosted.

Since a cloud infrastructure can host a large number of applications composed of multiple services, deploying observability in such environment requires careful analysis of its various elements so it does not affect the overall system performance.

A monitoring system must be able to receive data from all elements participating in the cloud environment, both at the application and the infrastructure level. These data should be as varied and complete as possible, in order to allow a full understanding of the environment and possible behaviors depending on their interaction. Thus, no failure or problem will be overlooked due to the lack of data.

However, collecting a large volume of data can be unfeasible due to network or storage limits. Additionally, gathering a huge amount of data without limitations can lead to problems in system performance, as well as having data collected needlessly. Thus collecting a large volume of data can have the opposite effect, that is, the observability of the system can be reduced instead of increased. This may occur due to information loss or data collection being delayed. Moreover, relevant information can be ignored given the complexity of analyzing a larger volume of data.

In addition to the reduction of observability, the mentioned performance problems may also lead to an increase in false positives/negatives. A large number of false negatives, as well as false positives, can affect the reliability of the monitoring system and can reach a level on which it becomes useless.

To reduce the overhead of observability in these scenarios adaptive sampling techniques can be used. High throughput services can be aggressively sampled without big loss of information. On the other hand, low throughput services do not require sampling and every request can be monitored without impacting the system performance [5].

That said, a monitoring system should be as simple and accurate as possible. Both its development and operation should be designed to facilitate its maintenance and usage. Therefore, during its project, the monitoring system objectives should be very clear in order to answer questions like: What will be collected? How will it be collected? And How often will it be collected? Once that's defined, each user should have access to the information that is most relevant to them. Where users could range from developers and operators to orchestrators.

Orchestrators are responsible for provisioning infrastructure resources in order to meet different application requirements. To verify if the requirements are being met, the orchestrator uses metrics to define whether or not the application needs more resources. If the metrics satisfy some predetermined condition like reaching certain thresholds, the orchestrator can act providing more or less resources to the application.

Among the most common actions taken by the orchestrator are vertical and horizontal scaling. Which consist of increasing the amount of resources and the number of replicas of an entity respectively. The act of scaling resources automatically is called auto-scaling.

The metrics used by the orchestrator for auto-scaling are usually metrics obtained from a view of the own infrastructure. For example, to find out if an application needs more of a given resource, the infrastructure simply monitors the CPU used by each of the services that make up the application. If any of them reaches a certain threshold, more CPU is provided to the service through vertical or horizontal auto-scaling.

This is an example where the infrastructure tries to infer the application requirements from the own infrastructure metrics. Although this approach may be enough for many applications, it may be insufficient for applications that have specific requirements. These requirements may not be

directly tied to infrastructure metrics and utilizing them for orchestration may lead to wrong decisions. Either from the application point of view, that may not have its requirements met, or from the infrastructure point of view, that will not have the best use of its resources.

Section IV will show a case study where integrating metrics from the infrastructure and application levels, in a cloud environment, ensures more suitable observability, resulting in better resource orchestration.

B. Open Source Tools

There are different tools that can be used to implement observability in distributed systems. Each of these tools have different functions and maturity levels.

In this section we present a set of open source tools used to increase the observability level of an environment. These open source tools were chosen because they have a high level of maturity, being used in systems of different sizes and complexity, both in development and production environments.

They also allow replication and improvement on what has already been done by other research groups.

- Fluentd [10] and Logstash [11] are log collection tools that enable the unification of data collection and consumption. Both tools utilize a pluggable architecture and are able to collect, process and persist logs from multiple sources and destinations.
- Prometheus [12] is a widely adopted systems monitoring and alerting tool originally developed at SoundCloud.
- Zipkin [13] and Jaeger [14] are distributed tracing systems based on the Google Dapper model [5]. Aside from collecting tracing data, these tools also allow them to be visualized through a web UI. Both have their own API but also have support for the Opentracing API.
- Opentracing [15] is a attempt to create a vendor-neutral open standard for distributed tracing.
- Grafana [16] and Kibana [17] are tools that enable the analysis and visualization of logs and metrics by creating charts, tables, etc.

III. Platform Implementation

To help demonstrating the observability mechanisms previously discussed in cloud systems, the platform described in [18] was used. It consists of an intelligent space developed using a network of IP cameras as the main sensor of the environment. This network is capable of capturing images and streaming videos in real time to a software infrastructure, which processes and analyses the images and, as a result, generates commands to control actuators such as a robot.

The software infrastructure of this intelligent space was designed as a platform for developing computer vision applications. Also it was considered that the platform users are developers of applications that can use different types of services distributed in the cloud. In order to provide the necessary programmability for the platform, its design was based on a SOA (Service-Oriented Architecture) model and implemented over a cloud infrastructure known as IaaS (Infrastructure as a Service).

In that work, the platform was designed to meet the most demanding requirements of computer and robotic vision applications, such as low latency, high bandwidth and high computational power. However, it did not have the necessary mechanisms to allow an adequate observability of the infrastructure and of the applications that are executed in it. Therefore, in certain situations, it was not able to meet some of the applications requirements.

To include multilevel observability to this platform, in this work we chanced the design of its architecture to add new functional blocks related to observability, as shown in Fig. 1.

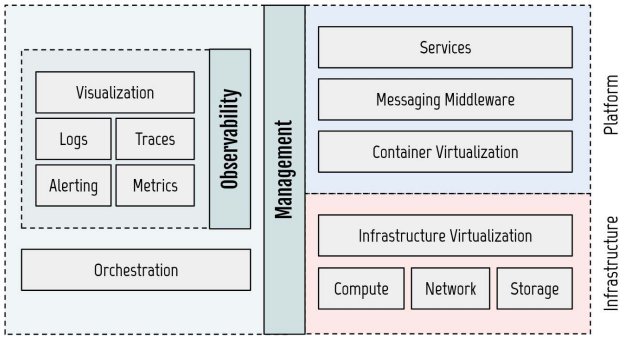

Figure 1: Platform Architecture

In this architecture, the messaging middleware is responsible for providing communication between the various entities of the system. The communication is done using a publish-subscribe pattern. That is, messages are not sent directly to a receiver but instead published to a topic where one or more receivers can subscribe to. The container virtualization layer performs the encapsulation and isolation of services. Thus the services can be migrated, scaled or updated in a simple and automatic way.

The management layer operates at different levels: from services that make up the application to physical resources in the cloud infrastructure. Therefore data from all these levels can be obtained to measure observability of the whole environment. The infrastructure is not part of the platform, but can be partially or totally managed by the platform.

To implement observability in this architecture, the tools described in Table I were used.

Table I: Observability tools

Logs	Metrics and Alerting	Tracing	Visualization
Fluentd	Prometheus	Zipkin & Opentracing	Grafana

One of the goals of the platform is to perform automatic orchestration of resources based on multilevel observability. To do that, we decided to integrate Prometheus, which was the metrics tool chosen for the platform, with Kubernetes, which was the tool chosen for container orchestration.

Kubernetes is an open source system for automating the deployment, scaling, and management of applications composed of containers [19]. It was based on two container management systems used internally at Google: Borg and Omega, and years of experience running containers at scale at Google [20].

Kubernetes has native support for automatic orchestration based on infrastructure metrics like CPU and memory usage. Kubernetes has a API aggregation layer that enables it's core functionality to be extended. That can be used to make Kubernetes aware of other types of metrics.

Fig. 2 shows how Kubernetes can be extended to be able to perform orchestration using metrics at both the application and the infrastructure level.

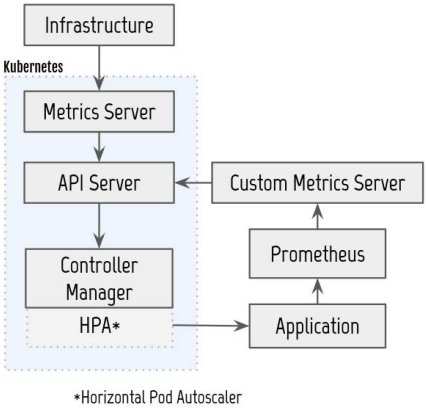

Figure 2: Metric Collection Integration

The custom metric server is an API adapter responsible for collecting Prometheus application metrics and converting them to a format so HPA (Horizontal Pod Autoscaler) is able to consume. With metrics from both layers, Kubernetes can perform the orchestration of the services as intended in this work.

In the next section we present a case study with two experiments. These experiments were defined in order to help observe resource orchestration in the cloud through multilevel observation, using the implemented platform.

IV. Case Study

As mentioned before, there are applications that possess specific requirements to function correctly. To properly provide the needed resources, the infrastructure must have access to metrics that can show if the requirements are being met or if more resources should be provided. Nevertheless, these metrics can't be generated by the infrastructure, they can only be provided by the own application. Besides that, these metrics must also show when resources allocated to the application are no longer needed and can be freed by the infrastructure.

In this section we are going to discuss two experiments that involves an application with specific requirements. Our aim is to highlight the need of metrics that cover the application and the infrastructure level as well. In this way, the orchestration of resources can be done in order to meet both the application requirements and to enable a better allocation of resources by the infrastructure.

The selected scenario includes a computer vision application, which belongs to a class of applications that demands a lot of resources and strict requirements. High computer capacity, wide bandwidth and low latency are examples of what is usually required by this type of application.

The application used as a case study is the detection of human skeletons based on RGB images, as illustrated in Fig. 3. The skeletons are further used to recognize dynamic gestures or actions, such as hand pointing or suspicious activities.

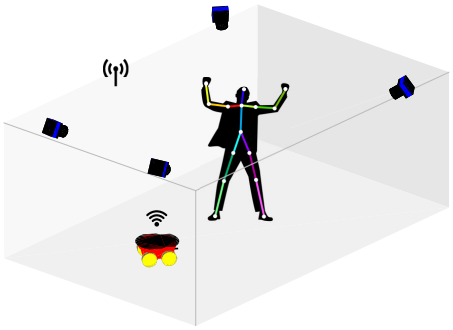

Figure 3: Skeleton Identification

The application is composed by the following services: Camera Gateway (GW), Skeleton identification (SkID) and Action recognition (AcR). The diagram showing the information flow of this application is illustrated in Fig. 4.

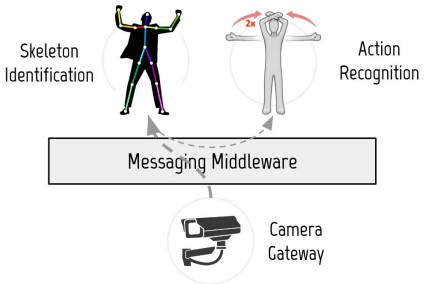

Figure 4: Application's Data Flow

The GW service is responsible for standardizing the

access to cameras resources. On other words, this service exposes the camera functionalities with a well defined and uniform interface, converting requests to the camera's proprietary protocol. Images are periodically published on a topic and can be consumed by any other service/application. There is one instance of this service per camera.

The SkID subscribes to images from the cameras and publishes, for each one, a vector containing the skeletons of the people detected in the environment. If no person is detected in an image, the output associate to that one is an empty vector. This service is implemented as a stateless, single threaded and CPU bound service

Finally, the resulting skeletons are processed by the AcR service, that performs action recognition based on temporal analyses.

A. Experiments

To validate the ideas presented in this work, and without loss of generality, we emulated the previously discussed application, instead of deploying it in a real environment. Although it does not represent the real case in terms of size, physical structure, time and processing demand, the emulation was able to illustrate the essential characteristics of the class of applications addressed. Information on how to reproduce the experiments can be found on our github repository [1].

In the experiments, the horizontal auto-scaling of one of the application services will be performed. The aim is to meet the specific application requirements on each of the experiments and yet use the infrastructure resources as best as possible.

Since it is the one with the highest computational requirements, the SkID service was chosen as the one to be auto-scaled. By running this service in a test environment, we are able to measure the service mean latency by using the platform observability. The mean latency for SkID service was measure as 30 milliseconds. Consequently, a single instance of this service support approximately 33 requests per second. This information will be used to select scheduling targets on the following subsections.

A Kubernetes HPA was used to orchestrate the SkID service. It calculates the number of instances that should be running at any given time according to Equation 1. That is, the current metric value for each instance is summed then divided by the target/desired metric value, the $ceil$ of this result will then produce the number of instances that should be running.

$$N_{TargetInstances} = ceil(\frac{\sum_0^N MetricValue_{current}}{MetricValue_{target}}) \quad (1)$$

The behavior of the HPA can be configured by adjusting the up/downscaling delays, evaluation period and scaling tolerance. The delays correspond to the time the scheduler must wait, since the last scaling, to perform another one. They help avoiding noises that may affect the monitored metric due to an instance start/stop. The evaluation period determines the rate at which the controller will query metrics and evaluate the desired number of instances for a service. The scaling tolerance is the minimum change (starting from 1.0) in the desired-to-actual metrics ratio for the autoscaler to consider scaling. For instance if a tolerance of 10% is used, the autoscaler will only upscale instances if the ratio goes above 1.1, similarly it will only downscale instances if the ratio goes below 0.9. For the experiments both scaling delays where set to 2 minutes, the controller evaluation period to 30 seconds and the scaling tolerance to 0.1.

1) Experiment 1: In this experiment, four cameras are used to grab images from the intelligent space, all of them working at a frame rate of 20 fps (frames per second). This image rate is necessary in order for the AcR service to have enough resolution to properly perform action recognition.

From previous measurements, in order to process images at this frame rate, 3 instances of the SkID service are required to run in parallel. If there are fewer instances, the service will not be able to process all the images and some of them will be discarded, directly affecting the AcR service.

However, processing images at this rate is only needed if there is someone in the environment. Thus, the application can only process a subset of the collected images in order to check if there is any person in the environment. If that is the case, the application can then request the service scaling to the infrastructure.

That said, the objective of this experiment is to make the infrastructure initially start the application with only a single instance of the SkID service. With only one instance, the application will not be able to recognize actions, but can detect the presence of people in the environment. If a person is detected, the application generates a metric that will advise the orchestrator about the necessity of scaling the SkID service. This same metric should also indicate to the orchestrator that, when no more people are detected in the room, the number of instances should return to one.

2) Experiment 2: In this experiment, again four cameras are used. Although, in contrast with the experiment 1, the frame rate of these cameras will now vary dynamically (5, 10, 15 and 20 fps). The variation in the camera frame rate will directly influence the deadline for processing the images, which corresponds to the sampling period of the camera.

A single instance of the SkID service can only process a certain amount of requests at a time. If the request rate is higher than this limit, the deadline for detecting skeletons in an image will not be met. If the number of missed deadlines exceeds a certain threshold, the service must be scaled so that new instances can fulfill the increased demand.

Although, the number of missed deadlines cannot be used

[1] https://github.com/rodolfo-picoreti/picom18

to orchestrate the service since this metric does not indicate when the orchestrator can free resources. For example, when the number of missed deadlines increase, the orchestrator would increase the number of instances. By doing that the problem would be solved and the metric value would drop to a value close to zero, which would incorrectly indicate to the orchestrator that the resources could be freed. So using the number of missed deadlines to orchestrate this service would create an oscillatory behavior.

On the other hand, if the mean latency of this service is known, the amount of requests that a single instance can process at a time can be estimated. Now this application metric could then be used to scale the service appropriately.

Therefore, the objective of this experiment is to show how an application metric (total request rate) can be used to properly scale a service while meeting its requirements. This orchestration will then be contrasted with one done using only an infrastructure metric (CPU usage).

B. Result Analysis

We start by analyzing the results of Experiment 1 shown in Fig. 5a, 5c and 5b. Fig. 5a shows the average and the sum of CPU usage across all the instances of the SkID service. Fig. 5c shows the value of the specific application metric and the number of replicas of the SkID service over time. Fig. 5b shows the rate of status code produced by the requests to the SkID service over time. In this figure, OK/s illustrates the rate of requests successfully processed and DE/s (deadlines exceeded per second) represents the rate of requests that were unable to be processed in time.

For this experiment, a metric value of 1 will cause the orchestrator to scale the service to the maximum number of instances (3). By looking at Fig. 5c, we can note that there is a single instance when no person is in the environment, and 3 instances when someone enters the environment. When that person leaves the environment, represented by the metric value returning to zero, the number of instances returns to 1 as expected.

Fig. 5b illustrates that a single instance is not able to keep up with the amount of requests, which is not a problem since no one is in the environment. However after the upscaling is performed, the 3 instances are able to properly process the requests in time.

Usually, the orchestrator would scale the number of service instances according to metrics provided by the infrastructure, such as the average usage of CPU.

From Fig. 5a we can observe that by using the average CPU usage in this case, the orchestration would keep the number of instances always high. But that is not necessary when there is no one in the environment.

This is a simple experiment where an on/off control is used to define the number of instances of a service. From this example, we can see how the orchestrator, using a metric generated by the application, can scale the number

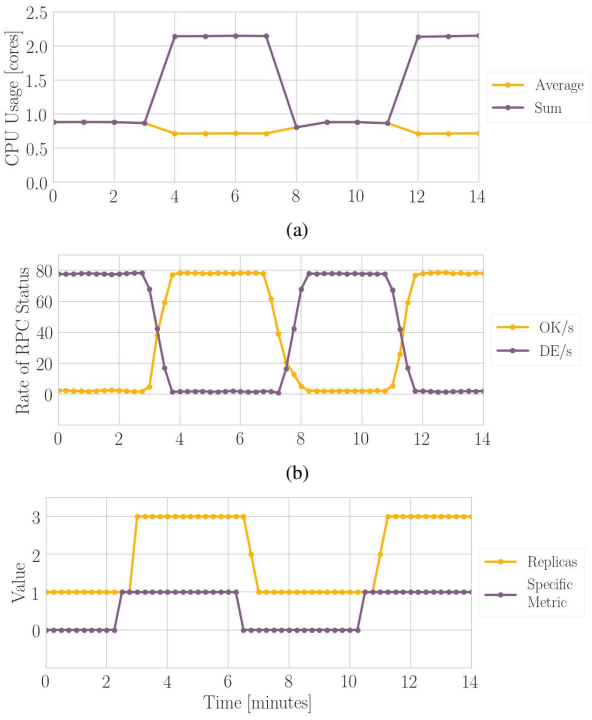

Figure 5: Results of Experiment 1

of instances of a service more properly than using a metric produced by the infrastructure.

For the second experiment three different orchestration strategies were tested. On the first two tests, the scaling was performed based on the average CPU usage. The first with a target usage of 0.75 cores, whereas the second with a target of 0.55 cores. On the third test, the scheduling was performed based on an application specific metric: rate of requests per second (rps). As mentioned before, based on the service average latency, the maximum rps that the SkID service can handle is approximately 33 rps. Although, to allow for some variation in the processing time of the service the value of 27 rps was used as the target for this last test. For simplicity the tests will be referred as CPU75, CPU55, and RPS27 respectively.

The results of all three tests are shown in Fig. 6. Each column shows respectively the results of CPU75, CPU55 and RPS27. On the first row, Fig. 6a, 6b, and 6c show the CPU usage across SkID instances. On the second row, Fig. 6d, 6e, and 6f show the status code rate over all SkID instances. On the third and last row, Fig. 6g, 6h, and 6i show the number of SkID instances over time.

CPU characteristics can vary a lot between services. Therefore, the choice of scaling target can be sometimes arbitrary. Furthermore, a high threshold is typically used to avoid over-provisioning of resources. Thus, initially the

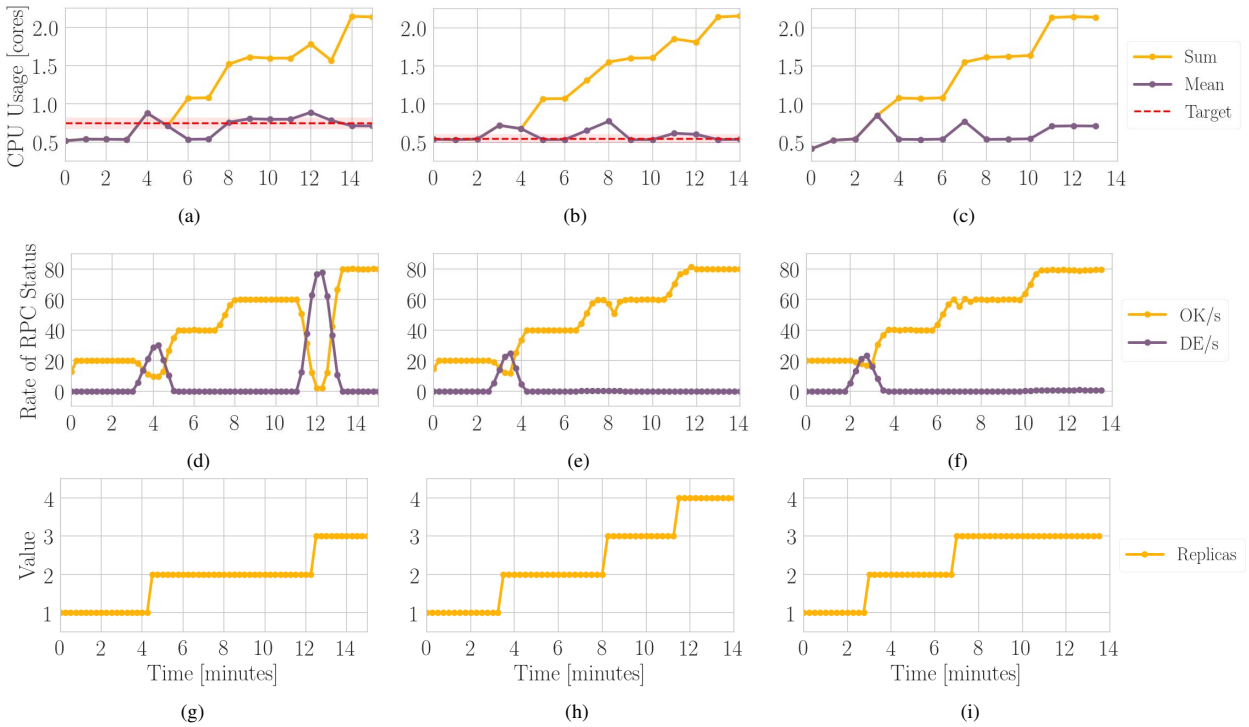

Figure 6: Results of Experiment 2

service was scaled using a CPU target of 0.75 cores. By looking at Fig. 6d, we can see that the service was not able to keep up with requests at two points. Both of them happen when there is an abrupt change on the request rate. The CPU usage of the overloaded services slowly increases until they exceed the scaling target. At which point the scheduler increases the number of replicas and the deadline rate decreases.

The scaling delay can be reduced by decreasing the CPU target. Which is done in the second test where a CPU target of 0.55 cores was used. In this test, we can observe by looking at Fig. 6e, that the overall DE/s decrease having a single peak at the first change in request rate. The decrease in the CPU target allowed the scheduler to quickly respond to the increase in demand. Although, it also caused more instances to be provisioned overall.

By comparing the results, we can see the choice of target creates a trade-off between the amount of resources allocated and the application requirements being met. Since the relation between applications requirements and the CPU usage are not always clear, choosing an appropriately CPU target can be difficult.

In the third test, as shown in Fig. 6f, we can observe a single peak in DE/s similar to CPU55. Although, without allocating a fourth instance. Therefore, by using requests per second as the scaling target we where able to reduce the number of provisioned resources while fulfilling the application requirements.

V. Conclusion

At their core, cloud infrastructures allow shared resources to be provisioned between many applications. Due to the elasticity of such environments, automatic orchestration tools are used to improve resource utilization even further. These tools try to meet application requirements by scaling them appropriately. Although they usually only take into account instrumentation performed by the infrastructure itself.

However, certain classes of applications have specific requirements that are difficult to meet, such as low latency, high bandwidth and high computational power. Applications involving computer vision and real-time robot control are examples of such class.

To properly meet these demanding requirements, orchestration must be based on multilevel observability. This includes collecting data from both the application and the infrastructure level, so they can be analyzed and an appropriate orchestration policy defined. This orchestration must meet the application requirements, as well as provide the best possible allocation of infrastructure resources.

With this in mind, in this work, we developed a platform that collects both application and infrastructure metrics to

demonstrate how multilevel observation can be implemented to perform automatic orchestration in cloud environments.

As a case study, two experiments were carried out in a scenario, where an application with very specific requirements was used to illustrate the issues addressed in this paper.

For the first experiment, a simple orchestration based on an on/off control was performed. The results show it was possible to better provision resources by using a metric generated by an information only available at the application level, and therefore, impossible to be inferred by the infrastructure.

In the second experiment, we compared the results of scaling the application with infrastructure and application metrics. By comparing the results of CPU75 and CPU55, we may conclude that a higher CPU target reduces the amount of resource allocation, but increases the amount of DE/s overall. Whereas with a lower CPU target the opposite occurs. We also showed that by using an application metric we where able to met the application requirements while keeping resource usage low.

This happens because the infrastructure tries to infer the application behavior by using its own metrics, although it is possible to obtain better results by using directly application metrics, as shown in RPS27.

In addition, an important future contribution would be the integration of a tracing tool to the orchestration. That would allow the whole context of a RPC to be taken into account when performing scheduling decisions, enabling the scheduler to automatically discover bottlenecks and scale the element that is most relevant to the system as a whole.

Therefore, as future work we intend to improve the orchestration tool in order to enable different ways of applying multilevel observability, and thus perform a better resource orchestration while meeting application requirements.

ACKNOWLEDGMENT

The research leading to these results received funding from the European Commission H2020 program under grant agreement no. 761508 (5GCity, a distributed cloud & radio platform for 5G neutral hosts). This work was also partially supported by CNPq.

REFERENCES

[1] C. Esposito, A. Castiglione, and K.-K. R. Choo, "Challenges in delivering software in the cloud as microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 10–14, 2016.

[2] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[3] F. Oliveira, T. Eilam, P. Nagpurkar, C. Isci, M. Kalantar, W. Segmuller, and E. Snible, "Delivering software with agility and quality in a cloud environment," *IBM Journal of Research and Development*, vol. 60, no. 2-3, pp. 10–1, 2016.

[4] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 105–118.

[5] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Technical report, Google, Inc, Tech. Rep., 2010.

[6] AWS amazon web services spot instances. [Online]. Available: https://aws.amazon.com/pt/ec2/spot/

[7] S. Taherizadeh and V. Stankovski, "Incremental learning from multi-level monitoring data and its application to component based software engineering," in *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, vol. 2. IEEE, 2017, pp. 378–383.

[8] B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems." in *HotCloud*, 2016.

[9] M. Klein, "Lyft's envoy: Experiences operating a large service mesh," 2017.

[10] Fluentd open source data collector for unified logging layer. [Online]. Available: https://www.fluentd.org/

[11] Logstash collect, parse, transform logs. [Online]. Available: https://www.elastic.co/products/logstash

[12] Prometheus monitoring system and time series database. [Online]. Available: https://prometheus.io/

[13] Zipkin a distributed tracing system. [Online]. Available: https://zipkin.io/

[14] Jaeger a distributed tracing system. [Online]. Available: http://jaegertracing.io/

[15] Opentracing a vendor-neutral open standard for distributed tracing. [Online]. Available: http://opentracing.io/

[16] Grafana the open platform for analytics and monitoring. [Online]. Available: https://grafana.com/

[17] Kibana explore, visualize, discover data. [Online]. Available: https://www.elastic.co/products/kibana

[18] D. Almonfrey, A. P. do Carmo, F. M. de Queiroz, R. Picoreti, R. F. Vassallo, and E. O. T. Salles, "A flexible human detection service suitable for intelligent spaces based on a multi-camera network," *International Journal of Distributed Sensor Networks*, vol. 14, no. 3, p. 1550147718763550, 2018.

[19] Kubernetes production-grade container orchestration. [Online]. Available: https://kubernetes.io/

[20] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, p. 10, 2016.