# Customisable Fault and Performance Monitoring Across Multiple Clouds

Giuseppe Bisicchia[1][a], Stefano Forti[1][b], Alberto Colla[2] and Antonio Brogi[1][c]
[1]*Department of Computer Science, University of Pisa, Pisa, Italy*
[2]*Consortium GARR, Roma, Italy*

Keywords: Cloud Computing, Fault Monitoring, Performance Monitoring, Declarative Management, OpenStack.

Abstract: Monitoring the proper functioning and performance of an infrastructure spanning multiple Cloud datacentres is challenging. It requires continuously aggregating monitored data across multiple source machines and processing them so to obtain useful alerts and insights. In this article, we propose a simple open-source prototype tool to perform highly customisable fault and performance monitoring across multiple Clouds. Differently from commercial tools, our prototype is simpler to deploy and it can be configured through a declarative approach, by simply specifying data monitoring tasks and aggregation policies. We illustrate such peculiarities over a use case relying on three datacentres under the Italian Research and Education Network Consortium.

## 1 INTRODUCTION

Cloud providers rely on multiple datacentres (DCs) distributed across various geographical regions. To guarantee proper functioning and availability of their resources, Cloud providers continuously monitor the health of their infrastructure against faults and performance degradation as dictated by Service Level Objectives (SLOs) (Ding et al., 2019). For instance, application response times, failure rates and network throughput and latency are among fundamental SLOs (Nastic et al., 2020). Through monitoring, Cloud administrators can avoid service downtime and provide their users with suitable Quality of Service (QoS) (Odun-Ayo et al., 2018).

Cloud monitoring tools should therefore be able to promptly detect, collect and report failing services or infrastructure assets, which might indicate one or more problems (Aceto et al., 2013). However, monitoring data generated by Cloud DCs can be very large and heterogeneous. Thus, monitoring tools also need to present such collected data in a way that delivers a simple, yet insightful, vision of the system's global state. They should offer a mechanism to aggregate monitored data across multiple geographically distributed Virtual Machines (VMs) so as to assemble a single picture of the system's health (Zareian et al., 2016). Also, SLO violations must be reported in quasi real-time so to ensure that Cloud providers can promptly act to solve them.

Last, Cloud SLOs evolve over time along with Cloud assets. For instance, the availability of GPUs or an update on the physical network can cause an improvement in the SLOs. Consequently, monitoring systems should be easily extensible and customisable, offering a simple way for Cloud administrators to add, remove and update monitored metrics and SLOs.

In this article, we propose CloudWatcher, a simple open-source[1] prototype to perform highly customisable fault and performance monitoring across multiple Clouds through a declarative and task-oriented approach, enabling monitoring custom metrics. Customisation, indeed, allows Cloud administrators to easily adapt the monitoring metrics to their needs and requirements. Our prototype is equipped with a Telegram-based alert system and a Web GUI based on Node-RED that permits visualising the encountered faults, the current values of the SLO metrics and their evolution over time. CloudWatcher is also assessed over a real use case within a significant portion (3 geographic areas) of the Cloud powered by the *Italian Research and Education Network Consortium* (GARR).

The rest of this article is organised as follows. First, Section 2 illustrates the design and implementation of CloudWatcher. Then, Section 3 showcases a real use case exploiting CloudWatcher to monitor a portion of the GARR infrastructure. Finally, Sec-

[a] https://orcid.org/0000-0002-1187-8391
[b] https://orcid.org/0000-0002-4159-8761
[c] https://orcid.org/0000-0003-2048-2468

[1]Freely available at: https://github.com/di-unipi-socc/cloudWatcher

Bisicchia, G., Forti, S., Colla, A. and Brogi, A.
Customisable Fault and Performance Monitoring Across Multiple Clouds.
DOI: 10.5220/0011849500003488
In *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, pages 212-219
ISBN: 978-989-758-650-7; ISSN: 2184-5042

tions 4 and 5 discuss some related work and draw some conclusions, respectively.

## 2 DESIGN & IMPLEMENTATION

**CloudWatcher's Design.** CloudWatcher aims at monitoring customised health and performance metrics (e.g., service availability and response time, status of the network, throughput of a VM's disk) of multi-Cloud systems, by providing configurable alerts associated with such measurements, and it is extensible to accommodate further metrics. Such an objective is pursued through a simple declarative and modular methodology for monitoring multiple DCs.

CloudWatcher relies on a set of distributed agents, called *Managers*, deployed in each monitored DC[2]. *Managers* exploit some dedicated Virtual Machines, called *Probes*, that are deployed across all DCs and periodically queried by their *Manager* to collect data on failures and performance. Last, *Probes* run in different types, according to the VM types available in each Cloud datacentre (e.g. `tiny`, `medium`, `large`) the OS they run (e.g. Ubuntu, CentOS), and their purpose (e.g. backup, database).

Fig. 1 shows an example deployment of CloudWatcher across two DCs (viz., `cloud1` and `cloud2`). *Managers* self-organise into independent overlay networks with their *Probes* across the different DCs. Each *Manager* requires at least one *Probe* of each considered type in each DC. For instance, *Manager* $M_1$ connects with two *Probes* of types $T_1$ and $T_2$ in `cloud1` and two *Probes* of types $T_1$ and $T_2$ in `cloud2`. Similarly $M_2$ exploits two *Probes* of types $T_1$ and $T_2$ in `cloud1` and two *Probes* of types $T_1$ and $T_2$ in `cloud2`. Indeed, for each deployment, all the *Managers* must deploy the same set of *Probe* types, in order to collect homogeneous data in the different DCs.[3]

The main duty of a *Manager* is to build a picture of the health of the monitored system from the point of view of the DC in which the *Manager* is deployed. A *Manager* constructs its view of the system by creating, interacting with, and deleting *Probes*. Administrators can both design each *Probe* as a passive or active entity.

[2]At least one *Manager* must be deployed in each DC of the monitored infrastructure for CloudWatcher to work properly. It is also possible to deploy multiple *Managers* per DC to provide greater robustness and fault tolerance and enable parallel data collection.

[3]To better support the possible heterogeneity among the different clouds (e.g. different VM flavours). It is possible to exploit another CloudWatcher deployment where a different type $T_3$ is considered.

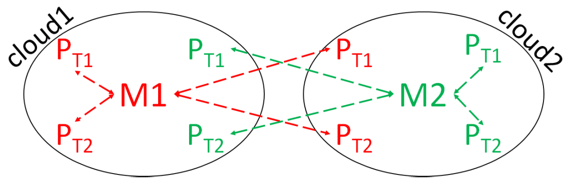


Figure 1: Example of a CloudWatcher deployment.

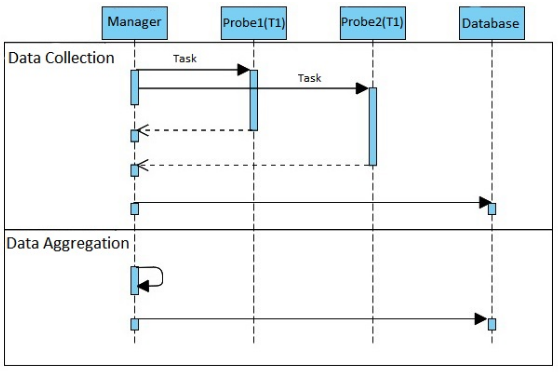


Figure 2: *Task* execution diagram.

A passive *Probe* exists only as a function of its manager to collect fault and performance data. As an example, passive *Probes* can be used to monitor the time required to execute a remote script or the network's bandwidth and latency. On the contrary, an active *Probe*, in addition to interacting with its *Manager*, can also carry out its own activities, e.g., execute one or more services, manage a database. This mode allows the measurement of the performance of the possible services and/or activities carried out by the *Probe*. In this case, the type of *Probe*, as well as the characteristics of the VM, can also indicate the type of services offered. For instance, active *Probes* can be used to monitor the performance and space exploited by a database or the time required.

Interactions between a *Manager* and a *Probe* occur through *Tasks*. A *Task* is a particular activity initiated by a *Manager* (e.g., remote script execution, deletion of a VM). Through the execution of a *Task* (and the possible interactions with the *Probe*), the *Manager* collects qualitative and quantitative information as well as errors and encountered faults.

Each *Manager* works in parallel and independently from the others to guarantee better scalability and robustness. The failure of a *Manager* or a particular *Task* does not affect the operations of other *Managers* nor the execution of other *Tasks*. The independence between different *Managers* and *Probes* also contributes to minimising the footprint of CloudWatcher, avoiding interactions that do not strictly concern the execution of a *Task* and, hence, reducing generated network traffic.

Each *Manager* periodically starts a monitoring step to perform the *Tasks* related to each controlled *Probe*. The sequence of *Tasks* is determined by the *Probe* type, i.e., the same *Task* sequence is performed for all the *Probes* of a certain type. For each *Task*, the execution is divided into two phases (Fig. 2):

1. *Data Collection*, the *Manager* executes the *Task* activity, according to a *data collection function*, collecting all the relevant information (e.g., performance, faults). This phase is performed in parallel and independently for all the controlled *Probes*.
2. *Data Aggregation*, the collected data are aggregated based on a *Task*'s *aggregation policy*. Both the *Data Collection function* and the *Aggregation policy* are defined by the Cloud administrators. Finally, the aggregated data as well as the data collected individually from every single *Probe*, are stored in a *NoSQL* database, leaving them available for further analyses and to be appropriately displayed through an alert system or a dashboard.

The possibility of using different types of *Probes* allows administrators to evaluate the same *Tasks* under different VM configurations or even develop different *Tasks* for different *Probe* types. Moreover, the aggregated vision by Cloud enables a simple and effective assessment of the health of each DC from the point of view of a particular *Manager* and DC.

Note that CloudWatcher is designed considering that each *Manager* deploys *Probes* on every single DC, hence, ensuring redundancy in both the *Managers* and the *Probes*. Although an architecture where each *Manager* deploys *Probes* only on its DC is possible, such a choice reduces the flexibility and the fault tolerance of the system. In our design, a *Manager* that interacts directly with its *Probes*, scattered through all the DCs, can perform a complete assessment of the connectivity between DC pairs, making it possible for each *Manager* to execute *Tasks* in any DC, without the need of relying on the other *Managers*. Such a design ensures independence between *Managers* and consequently improves the fault tolerance of CloudWatcher. At the same time, an architecture in which *Managers* have *Probes* only in their DC requires frequent exchange of data between them to keep updating each *Manager* about the health of other DCs, without reducing the overhead of CloudWatcher, in which *Managers* directly interact with *Probes* in all DCs.

**CloudWatcher's Implementation.** CloudWatcher is implemented in Python3 leveraging the *Fabric* and *OpenStack SDK* libraries that enable the interactions through SSH between machines and with Cloud services. The prototype was designed to interact with OpenStack Clouds, i.e. the most popular open-source Cloud platform.

CloudWatcher features a declarative and fault-tolerant interface to the functionalities offered by Fabric and OpenStack SDK. It has been designed to make available high-level, declarative operations for managing Virtual Machines (e.g., running a script, creating and initialising a VM). At the end of the execution of each function of the interface, a `Report` object is returned. It contains information on the possible errors encountered during its execution, the execution time and the number of iterations to complete.

Thanks to its interface, CloudWatcher also provides a way for Cloud administrators to specify customised *Tasks* as Python functions. A *Task* is composed of a data collection function and an aggregation policy both in the form of two Python functions. The collection function exploits the CloudWatcher's interface to perform its activity, which may require, for instance, the execution of one or more commands in a *Probe* or, in general, interaction with it and/or with the Cloud services. In this case, when commands are executed in the *Probes*, the output is printed in the `stdout`, which is caught by *Fabric* and then further processed by the *Manager*'s collection function.

CloudWatcher can be configured by providing it with the following files:

**`clouds.yaml`,** required by OpenStack to work with Cloud services, is *"a configuration file that contains everything needed to connect to one or more clouds. It may contain private information and is generally considered private to a user"*.[4]

**`config.ini`,** contains all the configuration parameters of CloudWatcher including the clouds to monitor, the type of *Probes* to use, how many machines per type to deploy per *Manager* and the number of *Managers* per Cloud to use. Through this file, it is possible to configure different deployments of CloudWatcher.

**`machines.json`,** used by the Cloud administrators to manage declaratively the available *Probe*'s types and the related *Tasks*. For each *Probe* type, the VM flavour and OS, as well as the local files to export to the remote machine, the access key pairs and the security groups are defined. Furthermore, for each type, a list of scenarios is declared, where each scenario is a list of *Tasks*.[5] Finally, for each

[4]From the OpenStack Documentation: https://docs.openstack.org/python-openstackclient/pike/configuration/index.html

[5]*Tasks* within a scenario are executed in parallel, while the scenarios, instead, are executed sequentially. Before

*Task*, it is possible to define some *Service Level Objectives* both for the data collected by the execution of a task and for the aggregated ones. SLOs can be declared by specifying the maximum or minimum value allowed (e.g., the maximum latency, the minimum success ratio for a Cloud service operation, the maximum number of tolerated faults) beyond which there is a violation. It is also possible to specify a series of threshold values as SLO, associating each of them with a label indicating the severity of the violation.

Finally, we developed a Web GUI with an alert system via Telegram notifications (Fig. 3). The Web GUI is implemented with *Node-RED* and shows the information on the last report received, the SLO violations and the evolution over time of the task measurements, divided by Cloud and type of *Probe* pairs and by *Managers*. The data can be consulted by specifying the search period and the *Task* name and optionally filtering by the cloud-type pair and/or a particular *Manager*. Furthermore, as soon as a report containing SLO violations is received, it is shown in the form of timed notifications in the dashboard. Moreover, we developed also an alert system through a *Telegram* channel in which the detected SLO violations are automatically published in real-time. We decided to develop our own Web GUI and alert system only to have interfaces able to show all the features of CloudWatcher. However, it is possible to connect CloudWatcher to different types of GUIs (e.g., Graphana) elaborating and rendering the published reports.

## 3 USE CASE

**Use Case Objective.** In this section, we discuss a use case of CloudWatcher within a portion of the Italian Research and Education Network Consortium (GARR) Cloud, consisting of three DCs in Naples, Palermo and Catania. The objective of this use case is to illustrate how CloudWatcher can be used to monitor the health of the network between DCs as well as that of the deployed VMs and Cloud services.

In particular with our deployment we use CloudWatcher to monitor:

- the VM disks I/O performance through simple stress tests;
- the featured network latency and available upload and download bandwidth;
- the required average time and success rate of SSH requests to the VMs, and
- the required average time and success rate featured to create, configure and delete VMs.

Besides, we aim at assessing those metrics while considering different versions of the Ubuntu OS, so to spot possible differences.

**Use Case Setup.** We deployed a *Manager* for each DC viz., Naples, Palermo and Catania. As for the *Probes*, two types were developed[6], each of them based on a VM with 1 vCPU, 6GB of RAM and 20GB of disk. In one of the two types, the OS used was Ubuntu 20.04, while Ubuntu 22.04 was used in the other. For each type of *Probe*, each *Manager* deployed two VMs per DC, for a total of 3 overlay networks each of 13 nodes (12 *Probes* and 1 *Manager*).

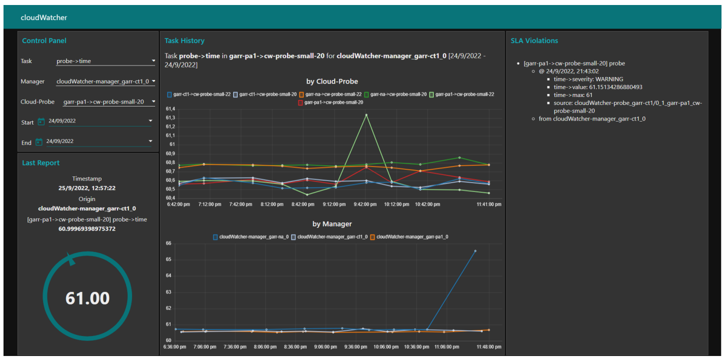
Fig. 4 illustrates an excerpt of the actual `machines.json` file developed for the use case. The *key pairs* (line 1) and the *security groups* (lines 2–11) are both defined declaratively and their reference used inside the *Probe*'s type `cw-probe-small-20` (lines 17, 19). The `tokens` (line 12), instead, are used as variables inside the `machines.json` file. The *Probe* type features a set of named scripts (e.g., `setup`, line 20). In this way, it is possible to request the execution of a script in the VM only using the script name. Note that for the `network` task (lines 23–33), we declared the data collection function (i.e., `probe_network`, line 24) and the aggregation policy (i.e., `aggregate_network`, line 26). Inside the aggregation policy, we defined some SLOs for the aggregated latency (line 28) and the bandwidth (lines 29–31). For each *Task*, it is also possible to provide some optional arguments to pass in input to the data collection function (line 35). Last, we can define a script (line 36), to be executed during the VM setup phase, containing the commands necessary for allowing the *Task* execution[7].

**Experiment Execution.** To accomplish our first objective we decided to measure the disk I/O performance by exploiting the *Fio* open-source tool. As shown in Fig. 5 the data collection function is `disk_performance` (lines 27–40) through which we execute in the *Probe* machine the `fio` command (a script in the type declaration inside `machines.json`), substituting on the command the token `<SIZE>` with

moving from one scenario to the next, all the *Tasks* of the previous scenario must be completed.

[6]The `machines.json` used is available at: https://github.com/di-unipi-socc/cloudWatcher

[7]The possibility of defining a setup subsequence within the *Tasks* and not only within the main setup script (line 20) has been made available to have greater separation of concerns, allowing to clearly identify why certain commands are part of the setup and, therefore, improving the readability and maintainability of the `machines.json` file.


cloudWatcher
by Cloud-Probe
garr-ct1->cw-probe-small-22
garr-ct1->cw-probe-small-20
garr-na->cw-probe-small-22
garr-na->cw-probe-small-20
garr-pa1->cw-probe-small-22
garr-pa1->cw-probe-small-20
by Manager
cloudWatcher-manager_garr-na_0
cloudWatcher-manager_garr-ct1_0
cloudWatcher-manager_garr-pa1_0
61.00

Control Panel

Task: probe->time
Manager: cloudWatcher-manager_garr-ct1_0
Cloud-Probe: garr-pa1->cw-probe-small-20
Start: 24/09/2022
End: 24/09/2022

Last Report

Timestamp
**25/9/2022, 12:57:22**
Origin
**cloudWatcher-manager_garr-ct1_0**
[garr-pa1->cw-probe-small-20] probe->time
**60.99969398975372**

Task History

Task **probe->time** in **garr-pa1->cw-probe-small-20** for **cloudWatcher-manager_garr-ct1_0** [24/9/2022 - 24/9/2022]

SLA Violations

- [garr-pa1->cw-probe-small-20] probe
  - @ 24/9/2022, 21:43:02
    - time->severity: WARNING
    - time->value: 61.15134286880493
    - time->max: 61
    - source: cloudWatcher-probe_garr-ct1/0_1_garr-pa1_cw-probe-small-20
  - from cloudWatcher-manager_garr-ct1_0

Figure 3: The CloudWatcher's dashboard.

```
{"keypairs": {"socc": {"public_key": KEY, "key_filename": PATH_TO_PEM_FILE, "user": "ubuntu"}},
"security_groups": {
  "CloudWatcher-sec-group": {
    "description": "CloudWatcher security group",
    "rules": [{
      "protocol": "TCP",
      "port_range_min": 22,
      "port_range_max": 22,
      "remote_ip_prefix": "0.0.0.0/0",
      "direction": "ingress",
      "ethertype": "IPv4"}]}},
"tokens":{"<BASEPATH>": "/home/ubuntu/cloudWatcher"},
"machines": {
  "cw-probe-small-20": {
    "image": "Ubuntu 20.04 - GARR",
    "specs": {"vcpus": 1, "ram": 6000, "disk": 20},
    "key_name": "socc",
    "network": "default",
    "security_groups": ["CloudWatcher-sec-group"],
    "scripts": {"setup": ["sudo apt update","sudo apt -f install -y"]},
    "files": [{"source": "./key.pem", "destination": "<BASEPATH>/key.pem"}],
    "tasks": [[{
      "name": "network",
      "function": {"name": "probe_network"},
      "aggregate": {
        "name": "aggregate_network",
        "slo":{
            "latency":{"avg": {"max": 5}},
            "bandwidth":{
              "upload":{"avg": {"min": [[500000000, "WARNING"], [2000000000, "CRITICAL"]]}},
              "download":{"avg": {"max": [[500000000, "WARNING"],  [2000000000, "CRITICAL"]]}}}}},
        args": [],
        "setup": ["sudo apt install -y iperf3","sudo iperf3 -s -D"]}],]},}
```

Figure 4: An example of `machines.json`.

the actual size required for the file contained in `args` (line 38). The output of `fio` is then parsed to extract a *JSON* object (lines 39–40). Generally speaking, a data collection function takes as input the data of the target *Probe* machine (i.e., an OpenStack *Server object*[8]) and a series of possible optional arguments and outputs a dictionary representing the collected data.

Aggregation is done by the `aggregate_disk_performance` function (lines 41–45) which averages the numerical values and stores a single copy of the other (line 43). The success rate is also calculated by computing how many times the tool completes its execution correctly (line 44). Generally speaking, the aggregation policy is defined by a function that takes as input a list of the dictionaries obtained in output from the execution

[8]From the OpenStack Documentation: https://docs.openstack.org/openstacksdk

```
def disk_performance(machine, args):
    report = machines.exec_script(machine.name, "fio_cmd", {"<SIZE>": args}).to_dict()
    dict = "\n".join((report["data"].split("\n"))[2:-2]) #parse the stdout
    return json.loads(dict)

def aggregate_disk_performance(ls):
    new_ls = [l for l in ls if l != {} and l is not None]
    res = average_dicts(new_ls) #compute the average of the numeric values
    res["success"] = float(len(new_ls) / len(ls)) * 100
    return res
```

Figure 5: The python functions of the `disk_performance` *Task*.

of the associated data collection function, on all the *Probes* of the same type and in the same DC and outputs a single dictionary that aggregates the obtained data, possibly adding, removing or extending the existing fields.

In such a way we are able to develop a simple *Task* to assess the disk performance with only a few lines of code, most of them dedicated to parsing the output of the fio command.

To accomplish our second objective, the performance of the network is measured in terms of featured latency and available upload and download bandwidth. Latency is calculated by measuring the ping from the *Manager* to the *Probe*. As regards the bandwidth, the *iperf3* tool is used.[9] Also in this case, the aggregation is done by averaging each parameter and computing the success rate of the measurements.

To accomplish the third objective, as illustrated in Fig. 6, the *Manager* requests via SSH the execution of a simple script on the *Probe* (line 47) in which the script name is defined in `args`. Then the performance is calculated in terms of execution time, the number of iterations required and how many and which errors were possibly encountered, divided by the type of error. In the aggregate (lines 55–56), the success rate and the average of the measurements are calculated, and the errors are added together.

Even in this case, we fulfil our objective with only a few lines of code, most of them dedicated to parsing the reports.

Finally, we accomplish the fourth objective, deleting each *Probe*'s VM with a probability of 50%, and then creating a new equivalent *Probe*, so as to assess the deployment process of the VMs. After being created, the setup script and related task setups are run in the new *Probe*. The performance of both deletion and creation is measured in terms of times, iterations required and errors. In addition, the same parameters are also measured with regard to the execution of the setup script. Finally, the time elapsing between the creation of the VM and the first successful access via SSH to the *Probe* is considered.

[9]In the setup phase of the probe machine, an *iperf3* server is opened to carry out the measurements.

**Lessons Learnt.** Our CloudWatcher deployment in the GARR Cloud allows us to easily develop custom declarative *Tasks* to monitor the required network and Cloud performance, with only a few tens of lines of code per *Task*. Furthermore, we are able to notice in near real-time changes in the network performance as well as to spot in good time, thanks to our alert system, failures on the Cloud services, in the Cloud stack or connectivity issues among DCs, and to circumscribe the faults both in time and space (i.e., in which DC). At the same time, through the dashboard, we were able to highlight the differences in performance between the three DCs as regards the network and the performance of the offered services and VMs.

CloudWatcher proved to be enough extensible to easily support the development of different *Tasks* simply and straightforwardly. At the same time, also the declarative management of the *Probes* thanks to the `machines.json` file proved to be pretty easy.

Although the customisability of CloudWatcher, through the concepts of *Probes*' types and *Tasks*, allows Cloud administrators to easily develop, change and adapt the monitoring activities to their needs, a factor that may hinder the applicability of our proposal could be found in the requirements for Cloud administrators, to develop, as discussed in this section, their own monitoring activities. However, it could be useful to build a shared repository of common activities to be used in a plug&play fashion or to easily adapt to specific needs.

## 4 RELATED WORK

Cloud monitoring is a widely studied topic, with various proposals for prototypes and commercial tools (Aceto et al., 2013; De Chaves et al., 2011; Moses et al., 2011). In (Fatema et al., 2014; Alhamazani et al., 2015) and (Ward and Barker, 2014) some of the

```
def exec_script(machine, args):
    report = machines.exec_script(machine.name, args).to_dict()
    return {"success": 100 if report["status"] is True else 0,
            "time": report["time"],
            "iterations": report["iterations"],
            "errors": report["errors"],
            "#errors": {
                "per_type": len(report["errors"]),
                "total": sum([int(report["errors"][e]) for e in r["errors"]])}}

def aggregate_script(ls):
    return aggregate_reports(ls)
```

Figure 6: The python functions of the `exec_script` *Task*.

most popular are discussed and compared.

Among the commercial tools, Prometheus [10] and Graphite[11] are two of the most used. However, both are oriented towards storing numeric samples for named time series, with at most a set of labelled dimensions.

Various tools rely on *plugins* to extend their monitoring capabilities, each system with different complexity and degrees of freedom. In OpenNebula (Milojičić et al., 2011), for example, it is possible to collect information relevant to the hosts and the VMs, by executing on them a set of probe programs. The management of the probe's programs, however, is done through the file system with limited flexibility. With a similar approach, Logic Monitor [12], Nagios [13] and Zabbix [14], can be extended through user-defined scripts to execute on the monitored VMs. However, the integration and configuration of the scripts into the ecosystems are not straightforward. Similar problems can be encountered in PCMONS (De Chaves et al., 2011) which leverages Nagios.

With a simpler integration mechanism, both Rally [15] and Munin [16] offer developers the possibility of extending their monitoring activity through plugins. However, Rally is more of a testing tool that allows administrators to execute stress test scenarios and collect information for benchmarking and profiling OpenStack-based Clouds (Pflanzner et al., 2016). Munin, instead, was designed to monitor cloud infrastructures and services but it lacks flexibility when accessing monitored hosts and its text file configuration model can be quite complex (Bicaku et al., 2016).

In the literature, it is also possible to find proposals relating to new monitoring architectures. For example, in (Uriarte and Westphall, 2014), the authors discuss a possible multi-agent architecture for monitoring private Clouds that considers the characteristics of the autonomic system. In (Andreolini et al., 2012), instead, the authors focus on the scalability and high availability of their proposal discussing an architecture which combines a hierarchical approach with decentralised monitors.

Finally, as regards possible performance metrics, in (Mancaş, 2019) the authors proposed some possible performance evaluation tests for Cloud environments, to compare different virtualisation technologies. With a similar goal in (Bystrov et al., 2021), a performance analysis of the communication- and computation-intensive discrete element method SaaS on the OpenStack Cloud is discussed.

In comparison with the other approaches illustrated, we focused our proposal on designing a monitoring system that is both flexible and easy to use and customise. Indeed, even if almost all the monitoring tools discussed support some form of customisability, the proposed mechanisms are often complex and require users to follow several implementation steps among different components of the monitoring tools. Instead, through our declarative approach, based on the *Probe*'s types and *Tasks*, Cloud administrators can easily adapt and extend the monitoring activities and metrics of CloudWatcher, as well as the SLOs, to their needs, even in complex infrastructures such as the multi-Cloud and multi-datacentres environments.

## 5 CONCLUDING REMARKS

In this article, we proposed CloudWatcher, an open-source customisable fault and performance monitoring tool for multi-cloud systems, employing declarative and task-oriented management through which it is possible to simply deploy and expand with custom metrics the monitoring activities.

[10]https://prometheus.io/
[11]https://graphiteapp.org/
[12]https://www.logicmonitor.com/
[13]https://www.nagios.com
[14]https://www.zabbix.com/
[15]https://rally.readthedocs.io/
[16]http://guide.munin-monitoring.org/

CloudWatcher is based on a two-tier architecture, in which a set of *Managers* controls overlay networks composed of *Probe* VMs spread among all the monitored DCs. Periodically, each *Manager* collects data on the status of the infrastructure interacting with its *Probes* while performing *Tasks*, i.e., activities to collect fault and performance data.

Both the types of *Probe* and the *Tasks* are managed declaratively through a `machines.json` file and designed to be easily customised and extended so as to integrate personalised monitoring activities and metrics. Each *Task* employs a data collection function for gathering data on the health of the monitored DCs, an aggregation policy to generate a single global report from the single *Tasks*' outputs and possibly a set of SLO thresholds, also defined declaratively.

As a use case, we deployed CloudWatcher on the cloud of the Italian Research and Education Network Consortium (GARR), consisting of three datacentres scattered throughout the Italian territory. We employed a total of 3 overlay networks and 12 probes per *Manager*. During the monitoring, we measured the disk I/O performance, the latency and bandwidth of the network and the behaviour of the Clouds while executing a remote script and during the random deletion and re-creation of the VMs. Additionally, a Web Dashboard and alarm system using *Telegram* were developed. In our future work, we intend to:

- *Data Analysis Pipeline.* Design and implement a pipeline for the automatic production of human-readable, insightful reports for the Cloud administrators, based on the data available on the database, highlighting the evolution over time and the critical aspects of the monitored parameters.
- *Large Scale Assessment.* Deploy CloudWatcher in a large-scale infrastructure for a long period to assess its behaviour, also comparing it with other tools. For example, in terms of the overhead produced by CloudWatcher, the reactivity of the monitoring activities on spotting possible failures or unusual measurements and how CloudWatcher reacts on cloud errors
- *Cloud-Edge Applicability.* Study the feasibility of design and develop an extension of CloudWatcher suitable for a dynamic and very heterogeneous environment, e.g. Cloud-Edge computing. Such an extension should be also able to both manage the scale of such infrastructures and the mobility of Edge and IoT resources.

## REFERENCES

Aceto, G., Botta, A., De Donato, W., and Pescapè, A. (2013). Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115.

Alhamazani, K. et al. (2015). An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art. *Computing*, 97(4).

Andreolini, M., Colajanni, M., and Pietri, M. (2012). A scalable architecture for real-time monitoring of large information systems. In *2012 2nd Symp. on Network Cloud Computing and Applications*, pages 143–150.

Bicaku, A., Balaban, S., Tauber, M. G., Hudic, A., Mauthe, A., and Hutchison, D. (2016). Harmonized monitoring for high assurance clouds. In *2016 IEEE IC2EW*, pages 118–123.

Bystrov, O., Pacevič, R., and Kačeniauskas, A. (2021). Performance of communication- and computation-intensive saas on the openstack cloud. *Applied Sciences*, 11(16).

De Chaves, S. A., Uriarte, R. B., and Westphall, C. B. (2011). Toward an architecture for monitoring private clouds. *IEEE Comm.Mag.*, 49(12):130–137.

Ding, J., Cao, R., Saravanan, I., Morris, N., and Stewart, C. (2019). Characterizing service level objectives for cloud services: Realities and myths. In *2019 IEEE ICAC*, pages 200–206.

Fatema, K., Emeakaroha, V. C., Healy, P. D., Morrison, J. P., and Lynn, T. (2014). A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *JPDC*, 74(10):2918–2933.

Mancaş, C. (2019). Performance analysis in private and public cloud infrastructures. In *RoEduNet*, pages 1–6.

Milojičić, D., Llorente, I. M., and Montero, R. S. (2011). Opennebula: A cloud management tool. *IEEE Internet Computing*, 15(2):11–14.

Moses, J., Iyer, R., Illikkal, R., Srinivasan, S., and Aisopos, K. (2011). Shared resource monitoring and throughput optimization in cloud-computing datacenters. In *2011 IEEE IPDPS*, pages 1024–1033.

Nastic, S., Morichetta, A., Pusztai, T., Dustdar, S., Ding, X., Vij, D., and Xiong, Y. (2020). Sloc: Service level objectives for next generation cloud computing. *IEEE Internet Computing*, 24(3):39–50.

Odun-Ayo, I., Ajayi, O., and Falade, A. (2018). Cloud computing and quality of service: Issues and developments. In *IMECS*.

Pflanzner, T., Tornyai, R., Gibizer, B., Schmidt, A., and Kertesz, A. (2016). Performance analysis of an openstack private cloud. SciTePress.

Uriarte, R. B. and Westphall, C. B. (2014). Panoptes: A monitoring architecture and framework for supporting autonomic clouds. In *2014 IEEE NOMS*, pages 1–5.

Ward, J. S. and Barker, A. (2014). Observing the clouds: a survey and taxonomy of cloud monitoring. *Journal of Cloud Computing*, 3(1):1–30.

Zareian, S., Fokaefs, M., Khazaei, H., Litoiu, M., and Zhang, X. (2016). A big data framework for cloud monitoring. In *Proc. BIGDSE '16*, page 58–64. ACM.