

Automated Probe Life-Cycle Management for Monitoring-as-a-Service

Alessandro Tundo, Marco Mobilio, Oliviero Riganelli, and Leonardo Mariani, *Senior Member, IEEE*



arXiv:2309.11870v1 [cs.DC] 21 Sep 2023

Abstract—Cloud services must be continuously monitored to guarantee that misbehaviors can be timely revealed, compensated, and fixed. While simple applications can be easily monitored and controlled, monitoring non-trivial cloud systems with dynamic behavior requires the operators to be able to rapidly adapt the set of collected indicators. Although the currently available monitoring frameworks are equipped with a rich set of probes to virtually collect any indicator, they do not provide the automation capabilities required to quickly and easily change (i.e., deploy and undeploy) the probes used to monitor a target system. Indeed, changing the collected indicators beyond standard platform-level indicators can be an error-prone and expensive process, which often requires manual intervention.

This paper presents a Monitoring-as-a-Service framework that provides the capability to *automatically* deploy and undeploy arbitrary probes based on a user-provided set of indicators to be collected. The life-cycle of the probes is fully governed by the framework, including the detection and resolution of the *erroneous states* at deployment time. The framework can be used jointly with *existing monitoring technologies*, without requiring the adoption of a specific probing technology.

We experimented our framework with cloud systems based on containers and virtual machines, obtaining evidence of the efficiency and effectiveness of the proposed solution.

Index Terms—Cloud monitoring, Monitoring framework, Monitoring-as-a-Service, Probes deployment

1 INTRODUCTION

Cloud-based solutions emerged as the key paradigm to support the operation of large-scale distributed systems composed of many interconnected services [1], [2]. Indeed, these systems are characterized by highly *dynamic* and *complex* behaviors that include the capability to adapt to changes in the available computational resources, to dynamically update and scale services without interrupting operation, and to be resilient to network problems and software failures.

Due to their size and complexity, every element of a cloud system must be *continuously observed*, to timely react to anomalous behaviors, generating alerts, and activating countermeasures [3]–[7]. In fact, cloud-based solutions are systematically enriched with monitoring capabilities, either natively offered by cloud platforms (e.g., Kubernetes [8]), or provided by external tools (e.g., Elastic Stack [9] and Prometheus [10]).

These monitoring solutions are mainly designed to collect a stable set of indicators over time, being *challenged by scenarios that require rapidly modifying the set of collected indicators*. In contrast, there are many well-known causes of sudden changes to the set of collected indicators. The *goals of the operators* change with the technical and business objectives of the organization, consequently causing changes in the set of the indicators that must be collected. The *software usage patterns* that emerge from the field continuously evolve, often determining the need of adjusting the monitored indicators accordingly. The collected indicators must be adapted to changes in the *workload*, which must be carefully observed to timely reveal any symptom of stress on the services. Moreover, *service updates* normally require putting in place ad-hoc monitoring capabilities that target the updated services to measure their reliability and timely detect misbehaviors. Sometimes, the observation of *failures* generates the need of continuously observing the services that fail often, to prevent new failures and localize the causes of problems; and *dynamically deployed scenarios* (e.g., to timely react to disasters and emergencies) require quickly deploying new functional services and the corresponding monitoring components. Relevantly, all these factors are *dynamic and cannot be entirely anticipated*.

Changing the set of collected indicators often requires changing the set of probes running in the field. However, configuring and deploying new probes, as well as undeploying the existing probes, are *non-trivial and time-consuming activities*. For instance, a tech company running many cloud services needs to collect KPIs at different granularity levels, taking into account both business and technical needs [5]. The needs of managers shall follow business goals and market evolution, while the needs of technicians shall follow QoS goals and software evolution. These needs evolve independently, and simultaneous changes in both business and technology may generate a rapidly increasing number of requests for the operators responsible of configuring the monitoring system. Operators may struggle adapting their monitoring systems at some point, especially when a large number of services has to be monitored. For this reason, research focused on *increasing the level of automation of probe management*. Figure 1 shows the increasing levels of automation that have been introduced in monitoring systems.

Simple *manually configurable* monitoring systems (Figure 1 (a)), such as Elastic Stack [9] and Prometheus [10], require configuring and deploying probes manually, that is, the life-

• The authors are with the Department of Informatics, Systems and Communication (DISCo), University of Milano-Bicocca, Milan, Italy.
E-mail: {alessandro.tundo, marco.mobilio, oliviero.riganelli, leonardo.mariani}@unimib.it

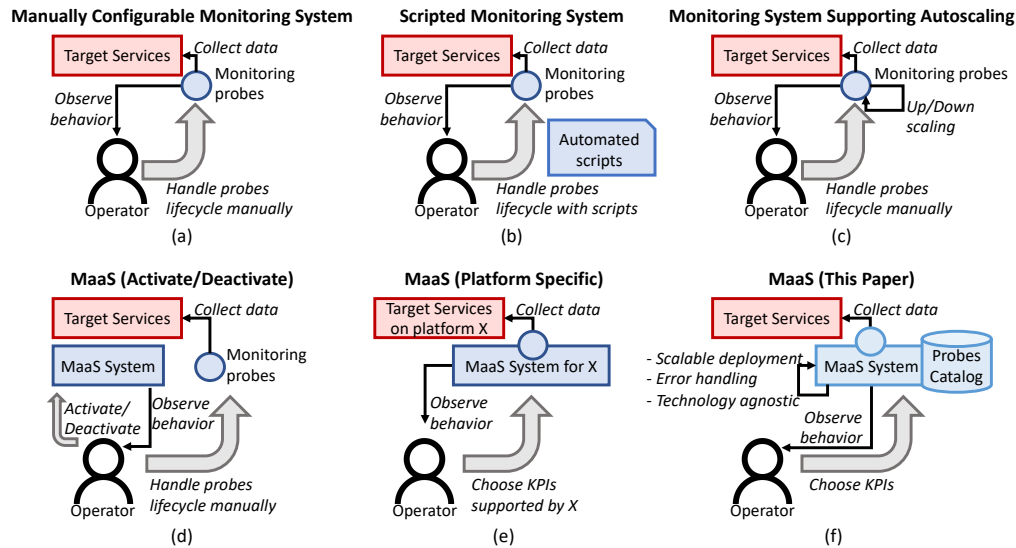


Fig. 1: Monitoring automation.

cycle of every component of the monitoring system must be handled manually by developers. Although useful, these monitoring systems are expensive to use in presence of frequent changes to the set of collected indicators, and badly adapt to dynamic scenarios.

Some probe deployment tasks could be implemented using *general purpose deployment systems* (Figure 1 (b)), such as Ansible [11] and Puppet [12]. However, these systems are not designed to specifically serve monitoring frameworks, and defining and controlling the deployment strategies would still be entirely on the shoulder of the operator. As discussed next in this paper, general purpose deployment systems can be indeed used as basic building blocks of more sophisticated deployment solutions tailored to address cloud monitoring.

A simple form of automation present in some systems consists of the *support to autoscaling* (Figure 1 (c)), that is, probes automatically adapt to a changing number of replica of a monitored service [13]. This is a useful feature, although limited to a specific scenario, missing to cope with the many changes that must be actuated as a consequence of changes on the set of collected indicators and monitored services.

To obtain a sufficient level of flexibility to address the aforementioned characteristics, *Monitoring-as-a-Service* (MaaS) solutions have been recently studied [13]–[16] (Figure 1 (d)–(f)). In fact, MaaS frameworks provide operators with the capability to flexibly decide the set of indicators to be collected, alleviating them from the burden of configuring and handling the life-cycle of the probes. In principle, an operator using a MaaS framework can simply specify the set of indicators that must be collected, while the operational aspects are automated by the framework.

Unfortunately, in many cases, automation is *limited to the activation of manually pre-deployed probes* [13] (Figure 1 (d)), that is, probes that have been already installed and configured *manually*. Adding probes to collect new indicators and removing existing probes must still be done manually by operators.

A higher degree of automation is provided by some *specific platforms* (Figure 1 (e)) that natively offer monitoring capabilities (e.g., Kubernetes). These solutions are effective

but significantly limit both the range of platforms and indicators that can be used. *So far, there is no general MaaS solution that can be used to collect virtually any KPI on any platform.* Note that a MaaS system that fully handles the life-cycle of probes is *the only solution that can entirely free operators from the burden of handling probe deployment*; in fact they would be able to control the monitoring system by simply specifying the set of indicators to be collected.

In this paper we address this challenge by presenting a MaaS monitoring framework (Figure 1 (f)) that exploits both a catalog of probes annotated with metadata and access to the API of the platform running the monitored services, to deliver *full MaaS capabilities including error-handling*.

This work extends our former tool demo paper [17] by (i) proposing a consolidated and scalable architecture, (ii) introducing error handling capabilities in the monitoring system, (iii) providing a rigorous presentation of the monitoring framework, and (iv) reporting results from an extensive empirical evaluation of the effectiveness of the approach. In a nutshell, this paper makes the following contributions:

- **Automated life-cycle management of probes.** We present a MaaS framework that fully automates the deployment and undeployment of arbitrary probes starting from declarative inputs (i.e., the list of indicators to be collected) entered by the operators.
- **Scalable and independent control processes.** We *rigorously* describe the probe deployment and undeployment procedures that guarantee the correctness of the resulting behavior. The involved processes are defined as stateless services to guarantee the scalability of the resulting system.
- **Deployment error handling routines.** We present how errors in probe deployment can be autonomously detected and fixed by the MaaS framework. So far, *error handling* capabilities received little attention, with approaches mostly focusing on error-free deployment scenarios or relying on the direct intervention of operators, despite the importance of error handling for long-running systems, such as a monitoring system [15], [16].
- **Technology-agnostic control processes.** We detail how the presented framework can be integrated with existing

technologies (e.g., probe technology, backend database, and target environment) without the need of using ad-hoc solutions.

- **Empirical evidence.** We empirically study the *effectiveness* of the framework with both *containers and virtual machines*, the *efficiency of error-handling*, and the *scalability* for an increasing number of requests.

The paper is organized as follows. Section 2 introduces a running example that is used to illustrate the approach throughout the paper. Section 3 presents the MaaS framework that automates life-cycle management of probes, rigorously describing the control processes that govern probe deployment and undeployment. Section 4 presents error handling. Section 5 discusses support to frameworks and probe technologies. Section 6 presents empirical evidence. Section 7 discusses related work. Finally, Section 8 provides concluding remarks.

2 RUNNING EXAMPLE

In this section, we introduce a running example that we use to illustrate and exemplify how the proposed MaaS framework works. The example consists of a PostgreSQL instance TARGET-PSQL running as part of a larger cloud system. Such an instance is of interest for two operators: operator OP-A and operator OP-B. Operator OP-A is mostly interested in infrastructure KPIs and is collecting network consumption data related to TARGET-PSQL. Operator OP-B is interested in both infrastructure and application KPIs, and is collecting 3 KPIs: network consumption data, CPU consumption data, and database metrics. We refer to this initial configuration as INIT-CONF.

In this context, operator OP-A may notice anomalous data in the network traffic and decide to collect information about two additional KPIs: CPU consumption and user session data. We refer to the configuration where operator OP-A is also collecting these two additional KPIs as the 2-MORE-KPIS-CONF.

Finally, operator OP-B may lose interest for the PostgreSQL service, for instance because the services maintained by operator OP-B may stop using PostgreSQL. In such a case, operator OP-B stops collecting any indicator from TARGET-PSQL. We refer to this final configuration as OP-B-LEFT-CONF.

We will refer to these sample scenarios and configurations in the rest of the paper to explain how the set of probes necessary to collect the indicators required by operators OP-A and OP-B can be adjusted automatically and transparently to the operators.

3 MAAS FRAMEWORK

The proposed MaaS framework exploits a few relevant domain concepts to organize the responsibilities of the components. In the following, we first introduce these key concepts, both informally and rigorously, and then discuss the framework architecture.

A **Target** represents an application, a system, or a resource that can be monitored by a monitoring framework. It is characterized by the hosting platform (e.g., Microsoft Azure, Kubernetes), a unique identifier within the hosting platform

(e.g., the Kubernetes Deployment name or VM name in Microsoft Azure), and its execution environment (e.g., virtual machine, or container). In our running example, the target is a PostgreSQL instance that we assume can be identified with the label TARGET-PSQL in both Kubernetes (as deployment name) and Microsoft Azure (as VM name).

A **Probe** represents a *deployable artifact* that can be used to collect indicators from targets in different environments. Probes are annotated with metadata that describe how they can be deployed and configured.

More rigorously, a probe p is a tuple $p = (I, meta, artifact)$, where $I = \{i_1, \dots, i_n\}$ is a set of indicators that can be collected with the probe, $meta$ is a set of key-value pairs that represent the metadata associated with the probe, and $artifact$ is a reference to the artifacts that implement the actual software probe. We use the notation p^I , p^{meta} and $p^{artifact}$ to refer to the individual components of a probe p .

A **Monitoring Claim** specifies the indicators that an operator may want to collect for a specific target. More rigorously, a monitoring claim mc is a tuple $mc = (I, op, t)$ where $I = \{i_1, \dots, i_k\}$ is the set of indicators to be collected from the target t for the operator op . The claim is intended as a complete specification for the specified target, thus if the operator is already monitoring an indicator i for a given target t and the newly submitted claim does not include the indicator, the monitoring system will stop collecting i from t . For example, operator OP-A shall submit a monitoring claim $(\{NETWORK_CONSUMPTION, CPU_CONSUMPTION, USER_SESSION_DATA\}, op-A, target-PSQL)$ to start collecting CPU consumption and user session data, in addition to network consumption. Similarly, operator OP-B shall submit a monitoring claim $(\{\}, op-B, target-PSQL)$ to stop collecting data.

A **Monitoring Request** is a collection of Monitoring Claims submitted with a single request by an operator. More rigorously, a monitoring request mr submitted by operator op is a set $mr = \{mc_1, \dots, mc_m\}$ where $mc_i = (I_i, op, t_i)$.

A **Monitoring Unit** is an execution unit (e.g., a virtual machine or a container) that runs one or more probes. When needed, our monitoring framework dynamically creates and destroys monitoring units to collect the indicators specified by the operators in their monitoring claims. A monitoring unit is also characterized by a hosting platform, which represents the environment where the unit is executed, and a configuration, which captures how the probes in the monitoring unit are configured.

More rigorously, a monitoring unit mu is a tuple $mu = (host, mus, C)$, where $host$ identifies the platform that provides the unit, mus indicates the strategy used to configure the unit (i.e., single probe or multi-probe), and C is the configuration of the unit, which consists of zero or more *probe configurations*, depending on the number of probes installed. Specifically a probe configuration $c \in C$ is a tuple $c = (p, I, op)$, where p is a probe, $I \subseteq p^I$ represents the set of indicators that p is configured to collect, and op is the operator who asked for the probe configuration c .

We use the notation mu_P to refer to the set of probes in the current configuration of mu , that is, $mu_P = \{p | \exists (p, \cdot, \cdot) \in$

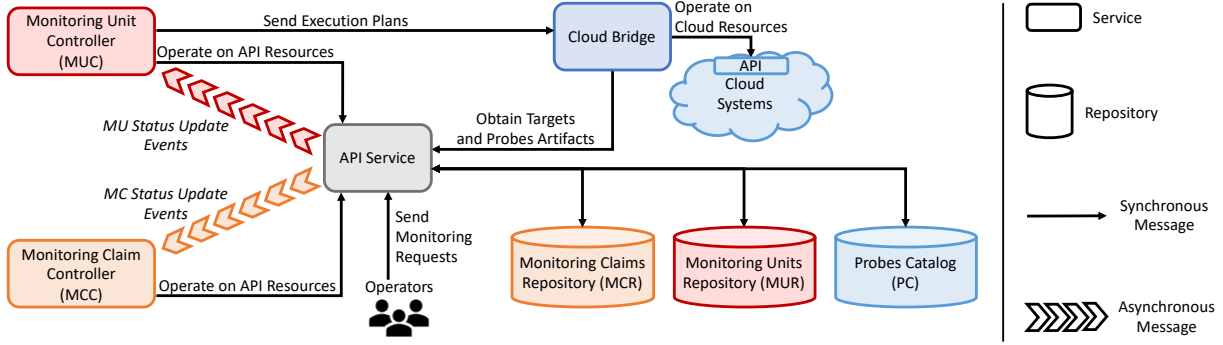


Fig. 2: Monitoring Framework Architecture.

$C\}^1$. Finally, given a probe configuration (p, I, op) , we use the notation $I(p)$ to refer to the indicators that p is configured to monitor, that is, $I(p) = I$.

Our framework implements two **monitoring unit configuration strategies**: the multi-probe monitoring unit and the single-probe monitoring unit. The *multi-probe monitoring unit strategy* uses one monitoring unit (e.g., a virtual machine) per monitored target (e.g., an instance of PostgreSQL), hosting in the unit all the probes that share a same target (e.g., every probe that collects indicators about PostgreSQL). This strategy is well suited for virtual machines, which are heavyweight units that typically run multiple processes. The *single-probe monitoring unit strategy* uses one monitoring unit (e.g., a container) per deployed probe (e.g., a Metricbeat probe for CPU consumption). This strategy is well suited for containers, which are lightweight units that preferably run a single process.

For instance, the initial configuration of the running example, where virtual machines running on Microsoft Azure are used, implies the existence of a single monitoring unit $mu = (azure, multi-probe, C)$, running the probe p_{net} , which serves both operators OP-A and OP-B, and the probes p_{cpu} , p_{db} , which both serve operator OP-B. Consequently, C consists of the following four probe configurations: $(p_{net}, NETWORK_CONSUMPTION, OP-A)$, $(p_{net}, NETWORK_CONSUMPTION, OP-B)$, $(p_{cpu}, CPU_CONSUMPTION, OP-B)$, and $(p_{db}, DB_METRICS, OP-B)$.

Note that the monitoring units are created to have the right visibility of the target to be monitored. In fact, a virtual machine monitoring unit can be either the same virtual machine running the monitored service or a separated virtual machine with probes that query an interface exposed by the monitored service (e.g., using SNMP [18]). On the other hand, a container monitoring unit can be created as a sidecar of the container running the target service [19], to have extensive visibility of the monitored service, or as a standalone container running in the same node of the target.

Figure 2 shows the proposed **monitoring framework**, which consists of four main stateless services and three repositories. The four services are (i) an *API Service*, which offers a gateway to access and update state information about the monitoring system, (ii) a *Monitoring Claim Controller*, which is responsible for handling the life-cycle of every monitoring claim, (iii) a *Monitoring Unit Controller*, which is

responsible for handling the life-cycle of every monitoring unit, and (iv) a *Cloud Bridge*, which exploits a plug-in based architecture to interact with different cloud providers and platforms, actuating the operations decided by the other services. The three repositories consist of (i) a repository of *monitoring claims* submitted by operators, (ii) a repository with the created *monitoring units* and their configurations, and (iii) a *probe catalog* with all probes and deployable artifacts.

Automated life-cycle management of the probes is provided by the two controllers that collaborate to manage the set of monitoring units, and the deployed probes, based on the requests produced by the operators that only include the information about the indicators to be collected. The stateless nature of the controllers guarantees *scalability*, as long as sufficient resources are provided to the monitoring system. The controllers also track the status of the monitoring units to *handle and recover from errors*. Finally, the framework is built with a plug-in based architecture that allows *multiple cloud platforms* to be integrated, as long as they provide a management API. In the rest of this section, we rigorously describe how the components, and the controllers in particular, behave.

3.1 Repositories

The **Probe Catalog** is a repository $PC = \{p_1, \dots, p_n\}$ where p_i is a probe. We assume the probe catalog is organized in such a way there is a unique artifact that can be used in a given context, that is, given an index i and the execution constraints (e.g., the host environment that executes the probe, the timeseries database that must be used to store the data, etc.), there is a unique probe p that can be used to collect i in the target environment. We do not detail here the execution constraints that can be used to identify the probe, but these are represented in the metadata associated with the available artifacts and matched for equality (or inclusion in case of lists) by the framework to select the probes.

Complex matching procedures can be also implemented in the catalog if needed, such as the possibility to have multiple probes suitable for a same context, and a decision procedure that can choose among them. Defining algorithms to choose among multiple probe artifacts is however out of the scope of the presented work and we simply require the operator to populate the probe catalog with one usable artifact per execution context that must be addressed with the architecture.

1. We use the symbol \cdot when any value is allowed in a tuple.

To illustrate the matching procedure, consider the case of OP-A asking to collect user session data from PostgreSQL. Let us assume the system considered in the running example runs on Kubernetes and that Elasticsearch is used as time-series database. In this context, the monitoring system will check the probe catalog looking for a probe whose metadata specify the capability to (a) collect user session data from PostgreSQL, (b) to run within containers, and (c) to store data in Elasticsearch. The monitoring system is configured with information about the environment (e.g., how to access Elasticsearch and Kubernetes APIs) to be able to configure the probes once deployed. If a matching entry is found, the corresponding artifacts are selected, and then deployed in a container, as illustrated later in this section. Otherwise, the request is aborted and the Probe Catalog has to be extended to support new probes, as described in Section 5.

The **Monitoring Claims Repository** stores the monitoring claims and tracks their statuses while they are created, processed, and updated. Since operators can update their claims about a given target, the repository can at most include one monitoring claim for a given operator-target pair. For example, an operator may submit a first monitoring claim to collect network consumption for a running instance of PostgreSQL (corresponding to the INIT-CONF in our running example), and later update the monitoring claim asking to collect two more indicators, CPU consumption and user session data, still from PostgreSQL (corresponding to the 2-MORE-KPIS-CONF in our running example).

The **Monitoring Units Repository** tracks the status of the monitoring units and their configurations. In particular, the Monitoring Units Repository stores both the *current configuration* of a monitoring unit, which reflects the status of the software monitoring unit, and the *desired configuration* of a monitoring unit, which reflects the configuration that must be reached based on the received requests, supporting the controllers in the process of adapting the configurations.

To conveniently work with the configurations required by operators, we define the operator $|_{op}$ which discards every entry related to op from a configuration. More formally, given a configuration C , we define $C|_{op} = \{c_i \mid c_i \in C \text{ and } c_i = (p_i, I_i, op_i) \text{ with } op_i \neq op\}$.

A Monitoring Units Repository MUR stores tuples (t, mu, dc) that associate a target t with a monitoring unit mu running probes that collect data from t , to its desired configuration dc . Given a monitoring unit $mu = (host, mus, C)$, we use the notation $conf_c(mu)$ to refer to its current configuration, that is, $conf_c(mu) = C$. We instead use the notation $conf_d(mu)$ to refer to the desired configuration of a monitoring unit mu , that is, $conf_d(mu) = dc$. The level of alignment between $conf_c(mu)$ and $conf_d(mu)$ indicates how much the actual monitoring unit (i.e., the unit running in the cloud) matches the monitoring claims submitted by operators. If $conf_c(mu) = conf_d(mu)$, the current and desired monitoring configurations are the same, thus the monitoring unit mu is up to date and perfectly aligned with the existing monitoring claims. Otherwise if $conf_c(mu) \neq conf_d(mu)$, the monitoring unit mu needs to be modified to reach the desired configuration.

If MUR is handled according to the multi-probe monitoring unit strategy, given a target t , there is at most one mu such that $(t, mu, \cdot) \in MUR$ (i.e., one monitoring unit running

multiple probes per target). If MUR is handled according to the single-probe monitoring unit strategy, given a target t and a probe p , there is at most one $(t, mu, C) \in MUR$, with $(p, \cdot, \cdot) \in C$, but there might exist multiple monitoring units running different probes associated with a same target.

3.2 API Service

The API Service provides two APIs: a public API for external clients and a private API for internal use only.

The *public API* is used by operators to submit monitoring requests, receive information about the status of their requests, extract the list of the current available Targets, and upload new probes to the Probes Catalog.

The *private API* is used by the Monitoring Claim Controller and Monitoring Unit Controller to handle (i.e., to read and update) the status information about both the monitoring claims and the monitoring units, as described in Sections 3.3 and 3.4.

Note that the API Service is the only service that can directly access the three repositories. The presence of a single entry-point for accessing the persistent data drastically reduces the risk of (potentially) introducing data inconsistencies. To avoid introducing a single-point of failure in the architecture, we designed the API Service as a stateless service that can be instantiated in multiple replicas.

The API Service is accessed through synchronous API calls, to guarantee that requests are processed as quickly as possible, but status updates are delivered through a message bus, since serving a request is not always an immediate operation.

3.3 Monitoring Claim Controller

Algorithm 1 Monitoring Claim Controller

Require: a monitoring claim $mc = (I, op, t)$ to be processed
Require: mus , the monitoring unit strategy
Ensure: desired configurations are updated according to mc

```

1:  $P \leftarrow$  APIService.getProbeConfigs( $I, t$ )
2: if  $P = \emptyset$  then return

3: if  $mus = \text{multi-probe}$  then
4:   UpdateConfUnit( $P, op, t, mus$ )
5: else if  $mus = \text{single-probe}$  then
6:   for  $p_{conf} \in P$  do
7:     UpdateConfUnit( $\{p_{conf}\}, op, t, mus$ )

8: procedure UPDATECONFUNIT(Set of probe configurations
    $P$ , operator  $op$ , target  $t$ , monitoring unit strategy  $mus$ )
9:    $unit \leftarrow$  APIService.getMonitoringUnit( $t, mus, P$ )
10:  if  $unit = \emptyset$  then
11:     $unit \leftarrow$  APIService.createEmptyMonitoringUnit( $t$ )
12:  APIService.updateDesiredConf( $unit, conf_d(unit)|_{op} \cup P$ )

```

The main responsibility of the Monitoring Claim Controller is to manage the life-cycle of the submitted monitoring claims by assigning the desired configurations, derived from the received claims, with the monitoring units. In particular, every time a monitoring request is received by the API Service, the API Service stores the monitoring claims

included in the request in the dedicated repository and sends a status update message to the Monitoring Claim Controller, which will incrementally process them.

Since controllers are stateless, the capability to process monitoring claims in parallel can be increased arbitrarily, based on the available resources, by instantiating multiple Monitoring Claim Controllers.

Algorithm 1 shows in details the operations performed by the monitoring claim controller every time a monitoring claim is processed. When a monitoring claim $mc = (I, op, t)$ of an operator op is processed, the controller first identifies the set of probes necessary to collect the indicators specified in the request and their configuration (line 1). This set is computed by the API service based on the probe metadata.

The monitoring units are reconfigured differently depending on the monitoring strategy. If the *multi-probe monitoring unit strategy is used*, the UPDATECONFUNIT procedure is invoked to associate a single monitoring unit with a desired configuration that includes all the probes (line 4). If the *single-probe monitoring unit strategy is used*, the individual probes configurations are extracted and then used to update the configuration of different monitoring units (lines 6-7).

The way a set of probe configurations are associated with a monitoring unit is defined in the UPDATECONFUNIT procedure. To identify the monitoring unit that must be updated, the controller queries, through the API Service, the monitoring units repository for an existing monitoring unit (line 9). If the *multi-probe monitoring unit strategy* is used, units can conveniently run multiple probes for a same target. In this case, the service looks for any monitoring unit created to observe t , that is, it looks for an entry $unit = (t, multi-probe, \cdot)$, where t is the target reported in the monitoring claim. If the *single-probe monitoring unit strategy* is used, P can only include a single probe, and the API service looks for a monitoring unit that is already using the selected probe to monitor the target t , that is, it looks for an entry $unit = (t, single-probe, (p, \cdot, \cdot))$.

In both cases, if the unit does not exist, a new unit with an empty desired configuration is created for the target t (line 11). Finally, the existing entry (i.e., the existing desired configuration) is updated by replacing the probes associated with operator op with the new ones specified in P (if the existing configuration is empty, P is simply used).

Let us consider the running example, with operator OP-A asking to collect two more indicators (CPU consumption and user session data) from PostgreSQL, if we assume the monitoring framework is configured to use the single-probe monitoring strategy, the submitted monitoring claim would be processed as follows. The access to the probe metadata would reveal the availability of two different probes that can be configured to collect the two indicators: p_{cpu} , which can monitor CPU consumption using a Metricbeat probe, and $p_{session}$, which can use a custom probe to collect data about user sessions. That is, $P = \{(p_{cpu}, CPU_CONSUMPTION, OP-A), (p_{session}, USER_SESSION_DATA, OP-A)\}$ at line 1. Since $mus = single-probe$, the UPDATECONFUNIT procedure is invoked twice, once for each probe.

The first invocation with probe p_{cpu} leads to the identification of a running unit that is already collecting CPU_CONSUMPTION from PostgreSQL for OP-B (line 9). The current configuration of the retrieved unit is $\{p_{cpu},$

CPU_CONSUMPTION, OP-B)\}. The framework finally updates the desired configuration of the unit by replacing the probe configurations of operator OP-A (none in this case) with the input configuration $(p_{cpu}, CPU_CONSUMPTION, OP-A)$, finally obtaining the desired configuration $\{(p_{cpu}, CPU_CONSUMPTION, OP-B), (p_{cpu}, CPU_CONSUMPTION, OP-A)\}$.

The second invocation with probe $p_{session}$ returns no unit that is already running that probe. Thus, a new unit is created (line 11), and the desired configuration $\{(p_{session}, USER_SESSION_DATA, OP-A)\}$ is associated with the unit.

The time complexity of Algorithm 1 is linear with respect to the number of selected indicators (I) and the number of matched probes (P), that is, $O(|I| + |P|)$.

3.4 Monitoring Unit Controller

Algorithm 2 Monitoring Unit Controller

Require: a monitoring unit mu

Require: its current configuration $conf_c(mu) = \{(p, I, op)\}$

Require: its desired configuration $conf_d(mu) = \{(p', I', op')\}$

Ensure: the unit is updated according to the desired configuration is generated

- 1: **if** $conf_d(mu) = \emptyset$ **then** dismiss mu
 - 2: $P_{add} \leftarrow \{p \in conf_d(mu)_P \setminus conf_c(mu)_P\}$
 - 3: $P_{update} \leftarrow \{p \in conf_d(mu)_P \cap conf_c(mu)_P \text{ s.t. } I'(p) \neq I(p)\}$
 - 4: $P_{drop} \leftarrow \{p \in conf_c(mu)_P \setminus conf_d(mu)_P\}$
 - 5: **if** $P_{add} \cup P_{update} \cup P_{drop} \neq \emptyset$ **then**
 - 6: $res \leftarrow \text{Bridge.doChanges}(mu, P_{add}, P_{update}, P_{drop})$
 - 7: **else**
 - 8: $res \leftarrow \emptyset$
 - 9: $\text{UpdateConfiguration}(mu, res)$ ▷ If no error, $conf_c(mu)$ is updated with $conf_d(mu)$
-

The main responsibility of the Monitoring Unit Controller is to manage the life-cycle of the monitoring units according to the desired configurations generated by the Monitoring Claim Controller. In particular, the Monitoring Unit Controller runs a control-loop that continuously checks the Monitoring Units for changes to be actuated, as a consequence of a misalignment between the current and the desired configurations. Multiple monitoring unit controllers can be active at the same time, but two monitoring unit controllers cannot act simultaneously on a same monitoring unit, to prevent any potentially erroneous concurrent change that would introduce inconsistencies in the process.

The operations performed by a Monitoring Unit Controller are shown in Algorithm 2. It first checks if the desired configuration is empty, in such a case the entire monitoring unit is dismissed (line 1). This is an important step to avoid running phantom monitoring units with no running probes. It then computes the diff between the current and desired configuration, identifying the probes to be added (line 2), the probes to be reconfigured to collect a different set of indicators (line 3), and the probes to be dropped (line 4). If any of these sets is non empty, the Cloud Bridge receives the probe configurations corresponding to the changes that must be actuated (line 6). Passing all the changes to be actuated at once enables the Cloud Bridge to potentially optimize how these changes are actuated.

The Cloud Bridge returns a result that specifies the errors experienced during the update process, if any. This information is used to update the current and desired configuration. In case no error is experienced, the desired configuration simply replaces the current configuration (line 9). Otherwise, the update process takes the errors into consideration. We describe error handling in Section 4.

Let us consider the case of the two desired configurations generated by operator OP-A when asking to collect two more indicators (CPU consumption and user session data) from PostgreSQL with the single-probe monitoring unit strategy, as discussed at the end of Section 3.3. The desired configuration related to the already deployed probe p_{cpu} results in no changes to be operated ($P_{add} \cup P_{update} \cup P_{drop} = \emptyset$), since the existing probe will be simply shared between the two operators (this is achieved by only updating the configurations in UPDATECONFIGURATION without touching the running probes). While, the desired configuration related to the new probe $p_{session}$ to be deployed results in a probe to be added ($P_{add} \neq \emptyset$).

The time complexity of Algorithm 2 is linear with respect to the number of probes to add ($|P_{add}|$), update ($|P_{update}|$), and drop ($|P_{drop}|$) while configuring a monitoring unit. That is, if $pchanges = |P_{add}| + |P_{update}| + |P_{drop}|$, the complexity of Algorithm 2 is $O(pchanges)$.

3.5 Cloud Bridge

The main responsibility of the Cloud Bridge is to actuate plans on cloud systems using their management APIs. The Cloud Bridge also provides information about the targets and the deployment status of the probe artifacts.

In particular, the Cloud Bridge exploits a plug-in based architecture that can be extended to support additional cloud systems. A plug-in for a target environment (e.g., Kubernetes) is used to map each change requested by controllers into a concrete command for the specific management API (e.g., the Kubernetes API) or the specific configuration management tool used to interact with the platform (e.g., Ansible [11]). This approach encapsulates the technological details inside the plug-in, keeping the whole control-plane framework agnostic from technology. Once all the changes have been actuated, the list of probes resulting in an erroneous state is sent back to the controller.

4 ERROR HANDLING

The presented framework implements error handling procedures to recover from deployment errors, namely, errors that might be experienced at deployment time while creating, updating and removing either probes or monitoring units. The framework does not target the runtime errors that might be experienced after a successful deployment. These procedures are extremely important for the dependability of the monitoring framework, whose behavior may otherwise diverge from the desired behavior. We distinguish two classes of errors that can be detected and handled:

- **Soft errors.** Soft errors indicate problems in the operations performed *while preparing* for the creation, update and deletion of a unit, such as retrieving probes and preparing their configuration. All these operations are performed

before modifying any existing monitoring unit. Since those are problems that do not compromise the dependability of the running units, they are considered soft errors that have negligible consequences on the running monitoring system.

- **Hard errors.** Hard errors indicate problems in the operations performed while *changing a running monitoring unit*, such as adding, reconfiguring or removing probes. Since these problems may compromise the dependability of the running monitoring system, they are considered hard errors that timely require corrective actions to be managed.

Errors are detected by the Cloud Bridge while interacting with platform management APIs and while running commands of configuration systems. Soft errors are produced during the execution of the preparatory steps, differently from hard errors that are generated while changing the actual monitoring units. For this reason, depending on if and when an error is detected, a probe to be deployed can be in one of the following states:

- *Failed probe:* a soft error has been detected by the Cloud Bridge while preparing the probe.
- *Broken probe:* a hard error has been detected by the Cloud Bridge while deploying/undeploying the probe.
- *Stable probe:* no error detected

The errors detected for each probe configuration that is processed by the Cloud Bridge are reported in the results returned to the Monitoring Unit Controller (line 6 of Algorithm 2).

Consequently, a monitoring unit can be in any of the following states, depending on the states of its probes:

- *Stable unit:* no error is detected for the probes in the monitoring unit.
- *Unsound unit:* there is at least a failed probe and no broken probe in the monitoring unit. This status indicates a failure in the attempt to align the desired and current configurations of the monitoring unit, but no actual problem is affecting the running unit.
- *Dirty unit:* there is at least a broken probe in the monitoring unit. This status indicates that the software running in the unit might be compromised.

Errors are mostly handled in the context of the UPDATECONFIGURATION procedure whose pseudocode is shown in Algorithm 3. The UPDATECONFIGURATION procedure is invoked by the Monitoring Unit Controller to finalize the update of a monitoring unit (line 9 in Algorithm 2).

In addition to referring to a monitoring unit mu and the set of probe configurations that resulted in soft ($Pconf_{soft}$) and hard ($Pconf_{hard}$) errors, the procedure maintains two data structures. The *RetryTable* is a table that stores for every monitoring unit the number of consecutive soft failures generated by each probe configuration. The *BlackList* data structure stores for each monitoring unit the list of probe configurations that generated hard failures. The idea is that soft failures are not harmful for the monitoring unit, and thus the failed changes can be safely retried. Instead, hard failures introduce dependability problems, and thus the failed changes should not be retried. Operators can reset these tables to allow again certain operations (e.g., after a compatibility problem in a probe has been fixed).

Algorithm 3 UpdateConfiguration

Require: a monitoring unit mu to be updated
Require: $res = (Pconf_{soft}, Pconf_{hard})$, where $Pconf_{soft}$ and $Pconf_{hard}$ are the set of probe configurations that resulted in soft or hard errors
Require: $RetryTable \subseteq MUnits \times ProbeConfigs \times \mathbb{N}$, which is a table that counts how many times a given probe configuration has been retried in a monitoring unit
Require: $BlackList \subseteq MUnits \times ProbeConfigs$, which is a table that tracks the probe configurations that cause errors and should not be retried again
Ensure: mu is updated and any error is reported

- 1: **for** $pc \in Pconf_{soft}$ **do**
- 2: $RetryTable.IncRetry(mu, pc)$
- 3: **for** $pc \in Pconf_{hard}$ **do**
- 4: $BlackList.add(mu, pc)$
- 5: **if** $Pconf_{hard} \neq \emptyset$ **then** ▷ Dirty unit
- 6: $Bridge.cleanUnit(mu)$
- 7: $conf_c(mu) \leftarrow \emptyset$
- 8: **else**
- 9: $conf_c(mu) \leftarrow conf_d(mu) \setminus (Pconf_{soft} \cup Pconf_{hard})$ ▷
 $conf_d(mu)$ is unchanged, so probe configs causing soft errors are retried, while probe configs with too many retries and probe configs in blacklist are automatically ignored

In practice, the error handling routine first increases the number of retries for the probe configurations that caused soft failures (line 2) and adds to the blacklist the probe configurations that caused hard errors (line 4). When the number of retries exceeds an operator-defined threshold, the configuration is blacklisted.

If at least a hard error has been detected, the unit is *dirty* and thus the bridge is asked to clean it. This operation depends on the target environment and the implementation of the plug-in used in the Cloud Bridge. For instance, in our implementation for containers, the bridge destroys the existing container and creates a new monitoring unit to replace it. The current configuration of the newly created monitoring unit is consequently set to the empty configuration.

If no hard error is detected, the current configuration is updated by adding all the configurations that generated no errors. In all the cases, the desired configuration stays unchanged.

This process may lead to three main distinct situations:

- *the current and desired configurations are aligned:* no changes will be performed on the monitoring unit in the future, unless a new request is submitted by an operator;
- *the current and desired configuration differs only for some blacklisted configurations:* in this case again there is nothing to be done. Note that although for simplicity we have not used the blacklist when computing the set of probes to be added, reconfigured, and deleted, in reality the Monitoring Unit Controller discards the configurations that appear in the BlackList data structure when computing them (Algorithm 2, lines 2- 4)
- *there are configurations that must be retried:* in such a case the desired and current configurations do not match, and the monitoring unit controller will process them again in the next iteration of its control-loop, retrying the failed probe configurations.

The time complexity of the Algorithm 3 is linear with respect to the number of probe changes and number of

errors occurred while configuring the monitoring unit. In particular, if *errors* is the number of probe configurations that resulted in soft or hard errors. The resulting time complexity is $O(pchanges + errors)$.

5 TECHNOLOGY AGNOSTIC DESIGN

The proposed monitoring framework is designed to transparently integrate heterogeneous monitoring technologies, releasing a *technology agnostic control-plane* that can be exploited to obtain MaaS capabilities using the preferred probe technologies and target platforms. To witness this capability, this section exemplifies the integration of probes of different types and the capability to support multiple cloud platforms.

5.1 Incorporating New Probes

To demonstrate the flexibility of the monitoring framework we describe how two largely different probes can be supported: a health-check probe, which queries the health status endpoint of services exploiting the HTTP protocol, and a Prometheus exporter for Apache Kafka [20], which monitors Kafka brokers resources (topics, partitions, etc.) and exposes the collected indicators as Prometheus metrics.

Adding a new probe can be done in two steps. First, the probe artifacts have to be manipulated in such a way they can be used by the Cloud Bridge. Second, the probe is added to the catalog by passing the probe metadata, which include information about where the probe can be deployed (the probe might be compatible with certain monitoring unit strategies but not with others), the supported data outputs (i.e., the database where the collected values can be stored), and the supported indicators, to the API Service. Listings 1 (a) and (b) show an excerpt of the metadata associated with the Apache Kafka Prometheus exporter and the health-check probe, respectively. Note that the configuration of the monitoring framework (e.g., the knowledge of both the available time-series database and the type of the target platform), jointly with the requests produced by the operators, allows the framework to select and deploy the right probes. In fact, artifact ids are mapped to the concrete software artifacts and scripts that are executed for probe deployment.

Adding new probes (i.e., new artifacts and corresponding metadata) to the catalog may require a different amount of time depending on the knowledge of the involved technologies. It is however a quite convenient operation for people who know the monitoring framework. For instance, we needed 1.5 hours to develop and setup a health-check probe that can be deployed on virtual machines, and 30 minutes to add a Kafka exporter that can be deployed on Kubernetes.

5.2 Supporting New Target Cloud Platforms

Supporting multiple target cloud platforms is another capability of the framework. A platform can be supported only if it provides a management API that can be used by the Cloud Bridge to manage the monitoring units and discover targets. Developers who want to create a new Cloud Bridge plug-in have to implement the base interface in order to run execution plans and provide information about the


```

{
  "id": "5fb6337a4102891e3677b475",
  "artifactId": "kafka_exporter",
  "supportedIndicators": [
    "KAFKA_BROKERS", "...",
  ],
  "supportedDataOutputs": [
    "PROMETHEUS"
  ],
  "supportedMUStrategies": [
    "SINGLE_PROBE",
    "MULTI_PROBE"
  ]
}
(a)

```

```

{
  "id": "5fb6337a4102891e3677b476",
  "artifactId": "http_healthcheck_probe",
  "supportedIndicators": [
    "HEALTHCHECK"
  ],
  "supportedDataOutputs": [
    "ELASTICSEARCH"
  ],
  "supportedMUStrategies": [
    "SINGLE_PROBE",
    "MULTI_PROBE"
  ]
}
(b)

```

```

{
  "targetPlatform": "azure",
  "targetPlatformId": "postgres-1",
  "envType": "INACCESSIBLE_VM",
  "metadata": {
    "resourceGroup": "resource-group-vm-1",
    "ipAddress": "52.92.34.124",
    "privateIpAddress": "10.19.20.3",
    "...",
  }
}
(c)

```

Listing 1: Listings show an excerpt of metadata for the Kafka Prometheus Exporter (a), an excerpt of metadata for the HTTP Healthcheck Probe (b), and a sample JSON representation of a Target retrieved from Microsoft Azure (c).

targets to the framework. Listing 1 (c) shows an example of target information that can be retrieved by the API via the Cloud Bridge component. Plug-ins are also associated with metadata (e.g., the supported monitoring unit strategies) that can help the framework in taking some decisions.

Our prototype implementation already includes two plug-in implementations that can transparently actuate the same plans on radically different platforms: Kubernetes, a container-based platform, and Microsoft Azure Compute, a virtual-machine-based platform.

6 EMPIRICAL EVALUATION

Since probe deployment and error handling are two representative capabilities of the proposed framework, we designed an empirical evaluation to assess them. We further study scalability, to investigate how the monitoring framework scales with an increasing number of requests and operators. These points are captured by the following three research questions.

- 1) **RQ1 - Framework Efficiency: How efficiently are probes deployed?** This research question validates the framework capability of deploying probes starting from a declarative input and investigates how efficiently it is in fulfilling an operator monitoring request. This is investigated for both cloud systems based on containers and virtual machines giving evidence of the technology-agnostic capabilities of the framework. Results are studied in comparison to a solution working with pre-deployed probes that can be activated/deactivated (Figure 1, cases (d) and (e)). To this end, we selected JCatascopia [13], which is consistent with the MaaS case shown in Figure 1 (d), and it is usable with no restrictions being an open-source research prototype.
- 2) **RQ2 - Error Handling: How efficiently are errors handled?** This research question validates the framework capability of detecting and recovering from errors and investigates the time required by the framework to execute the error handling routine.
- 3) **RQ3 - Scalability: How does the framework scale for an increasing number of requests?** This research question validates the framework capability of optimizing the control-plane during the evolution of the monitoring system. It studies scalability with respect to the number of requests produced by operators.

All RQs were addressed with cloud systems based on both virtual machines and containers. In the following, we

describe the prototype we used to run experiments, we report the design of the study, and the results for each research question.

6.1 Prototype

We implemented the framework described in this paper in a publicly available prototype hosted at <https://gitlab.com/learnERC/varys>.

The services are implemented as Java standalone applications. The repositories are implemented as MongoDB [21] collections. The JSON format is used both for communication and to persist information, except for the Cloud Bridge which exposes a gRPC API that uses Protocol Buffers. The status update messages are delivered through Redis Streams [22]. The monitoring system can be deployed both on containers and virtual machines, depending on the hosting environment.

We designed a probe catalog reusing probes from Metricbeat [23], one of the most popular cloud monitoring framework. We used Elasticsearch [24], as timeseries database to store the values extracted by the probes. We implemented plug-ins for the Cloud Bridge to support both Kubernetes and Microsoft Azure as target cloud platforms.

The Microsoft Azure plug-in supports either creating virtual machine monitoring units on-the-fly within the configured Azure resource group, or accessing the same virtual machine running the target to deploy the probes internally. In our experiments, we annotated the target service as an ACCESSIBLE_VM, and made it accessible to the Cloud Bridge via SSH in order to (un)deploy the probes directly within the virtual machine running the target service.

With respect to container monitoring units, the Kubernetes plug-in deploys container monitoring units in the same platform of the target and configures the probes accordingly. Purposely, it does not implement the container sidecar pattern [19] because it would trigger the redeployment of the target service, due to how Pods work in Kubernetes, every time probes are (un)deployed, potentially causing service or monitoring interruptions unless a robust rolling update strategy is in place.

6.2 RQ1 - How efficiently are probes deployed?

The monitoring framework can work in parallel on any number of monitored targets, if enough instances of the monitoring unit controller service are created. If there are

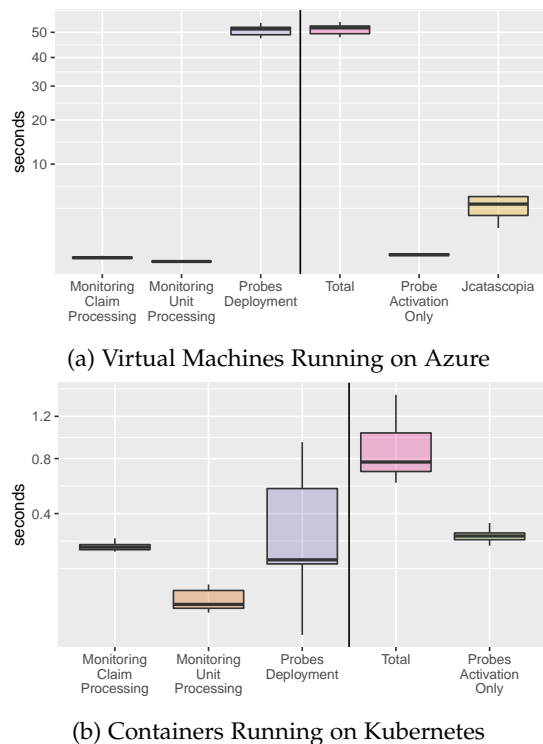


Fig. 4: Probes Deployment

more targets to modify than controller instances, some modifications will be performed sequentially. For instance, if 4 monitoring units must be modified and only 3 controller instances are available, one unit will be modified sequentially after another one. We will thus study how efficiently a monitoring unit can be managed by a single controller instance, the performance over multiple simultaneously evolving units can be straightforwardly deduced given the number of controllers available.

We consider two cases for the experiments: PostgreSQL running in a container in an on-premise installation of Kubernetes and PostgreSQL running in a virtual machine on Microsoft Azure. The two cases show how the same framework can be transparently used to address heterogeneous scenarios where the involved technologies are significantly different. We collect time figures considering the case of a request that requires the simultaneous deployment of three probes to collect the following three indicators from PostgreSQL: CPU consumption (using the CPU metricset of the Metricbeat probe), memory consumption (using the memory metricset of the Metricbeat probe), and database metrics (using the database metricset of the Metricbeat probe).

To study the efficiency of each step, we measure the time taken by the first controller to process the claim, by the second controller to compute the execution plan, and by the Cloud Bridge to actuate the plan. To have a baseline measurement, we also consider the case of a static framework, that is, a framework that does not support dynamic probe deployment, requiring operators to *deploy and configure probes in-advance*, which can be later activated and de-activated. This framework is *far less flexible* than the framework presented in this paper, but faster since it does not deploy probes

dynamically. To this end, we both use our framework with pre-deployed probes and the JCatascopia [13] state of the art monitoring solution, which allows us to collect further measurements from a third party system. We do not have measurements for JCatascopia applied to containers since it only supports virtual machines. Every experiment is repeated 10 times to collect stable measures.

Figure 4 shows the collected time figures with a semilogarithmic scale considering both virtual machines (Figure 4a) and containers (Figure 4b). The individual steps of the probe deployment process are captured by the Monitoring Claim Processing, Monitoring Unit Processing and Probes Deployment boxes. While Total represents the total time elapsed between the submission of the request and the time the deployed probes start collecting the required indicators.

Not surprisingly Probes Deployment is the most expensive step of the process for both virtual machines and containers. In the case of virtual machines it takes nearly 50 seconds, while the other steps can be completed an order of magnitude faster. In case of containers the difference is remarkably smaller, due to their computational efficiency and their ability to cache artifacts. In fact, probes deployment can be performed in at most 1 second with containers, while the remaining steps take less than 0.25 seconds.

Overall, the entire probe deployment process of the three probes (indicated with Total in Figure 4) could be completed in slightly less than a minute using virtual machines and less than 1.5s using containers, which is a nearly two orders of magnitude difference.

The box Probe Activation Only shows the time required to activate pre-deployed probes using our framework. In the case of virtual machines, exploiting dynamic probe deployment might be quite expensive compared to manually pre-deploying probes, since it increases the runtime cost by an order of magnitude. However, pre-deploying many probes can be expensive, can generate large and difficult to manage virtual machines, and is efficient only when the indicators that might be collected can be predicted. The comparison to JCatascopia shows that the presented framework is efficient, also when just used to process requests and activate pre-deployed probes. In fact, JCatascopia required several seconds to activate the probes, while our framework could activate probes in less than a second. The difference between dynamic probe deployment and pre-deployed probes for containers is indeed less significant, both in relative and especially absolute terms.

Answer to RQ1 In the case of virtual machines, the cost of flexibly deploying probes is significantly higher than working with pre-deployed probes. Thus, the trade-off between flexibility and efficiency should be carefully considered by operators to decide the monitoring solution that should be adopted. Instead, in the case of containers, the cost of flexibly deploying probes is significantly amortized by the efficiency of the cloud technology. In fact, our framework can complete the process in 0.5-1.5 seconds, while activating the pre-deployed probes requires slightly less than 0.5s, suggesting that dynamic probe deployment might be often preferable.

6.3 RQ2 - How efficiently are errors handled?

To study the capability of the framework to react to errors, we designed a variant of the experiment performed for RQ1 where we deploy a malfunctioning probe. We obtained such a probe by implementing a wrong configuration of the Metricbeat probe for PostgreSQL that makes the probe deployment to fail.

In the case of virtual machines, we study the creation of a new monitoring unit with two probes: one working probe and a malfunctioning probe. The malfunctioning probe artifact contains an Ansible role with a wrong command that leads to a hard deployment error when the Cloud Bridge executes it. Since we use the multi-probe monitoring unit strategy with virtual machines, error detection must autonomously detect the problem with the monitoring unit with two probes and automatically create a monitoring unit with the working probe only.

In the case of containers, we study the creation of a new monitoring unit with the malfunctioning probe only. The malfunctioning probe artifact contains a bugged Kubernetes manifest file that tries to deploy the probe within a non-existent Kubernetes namespace. This leads to a hard deployment error when the Cloud Bridge executes it. Since we use the single-probe monitoring unit strategy with containers, error detection should simply drop the malfunctioning monitoring unit (in this case we do not consider the deployment of two probes because the deployment strategy would simply create two different monitoring units handled independently).

To capture how error detection works, we measure the time necessary to the framework to *attempt the deployment and detect* that a monitoring unit is not working (namely Error Detection), the time necessary to *process* the error and take the decision to clean the monitoring unit (namely Error Processing), and finally the time necessary to actuate the cleaning plan (namely Error Cleaning). Error detection is performed by the cloud bridge while actuating changes (see the call in Algorithm 2, line 6), error processing consists of the operations shown in Algorithm 3, and error cleaning is again performed by the Cloud Bridge when cleaning a unit (see the call in Algorithm 3 line 6).

We repeated measurements 10 times to collect stable time figures. Figure 5 shows the collected time figures with a semilogarithmic scale considering both virtual machines (Figure 5a) and containers (Figure 5b).

In both environments, error detection and error cleaning are more expensive than error processing. In fact, error detection requires performing the deployment, at least partially, and similarly error cleaning requires disposing monitoring units and creating new stable units, when possible.

Similarly to probe deployment, error handling is significantly more efficient with containers than virtual machines. For instance, error detection requires around 21 seconds with virtual machines while it can be completed in less than 0.25 seconds with containers. Similarly, error cleaning requires around 13 seconds with virtual machines, while it can be completed in about 0.15 seconds with containers, but it is important to remark that the cleaning phase with containers does not require recreating a monitoring unit that is instead only disposed. The entire error handling process can be completed in around 35 seconds with virtual machines and less than a second with containers.

Answer to RQ2 Results show how the proposed MaaS solution that flexibly allocates and destroys resources, although usable with both virtual machines and containers, are naturally more suitable for containers where errors can be recovered in seconds.

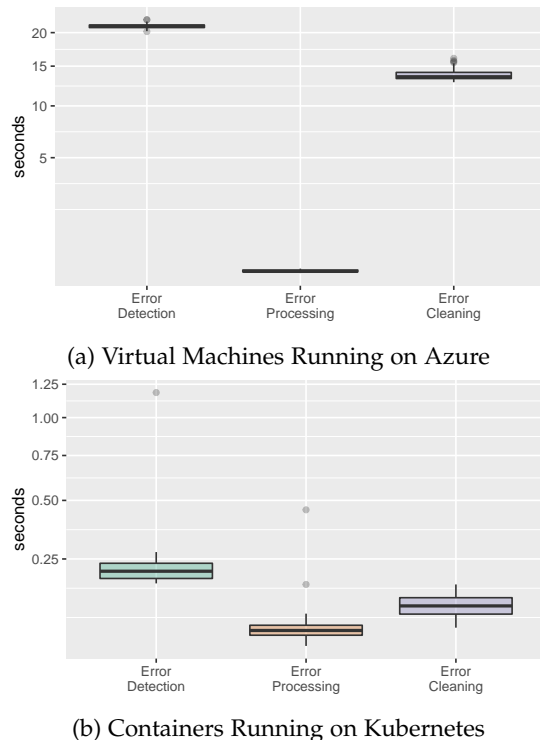


Fig. 5: Error Handling

6.4 RQ3 - How does the framework scale for an increasing number of requests?

As discussed, the framework can update multiple monitoring units in parallel as long as a sufficient number of controller instances are created. We thus focus the scalability study on measuring how the cost of collecting additional indicators grows with an increasing number of requests when single instances of the controllers are available. In particular, we consider two cases: processing requests that require *deploying new probes* and processing requests that require *reconfiguring* the monitoring system without deploying new probes. The former case corresponds to operators asking for new indicators to be collected. The latter case corresponds to operators asking for indicators already collected by other operators that the framework handles in an optimized way sharing the existing probes among operators without touching the monitoring units, but only changing the set of configurations associated with a unit.

We measure how the total deployment time grows while increasing either the number of *new indicators* or the number of *existing indicators for new operators* from 1 to 30. We submit all requests at once and we measure the total time necessary to fulfill the request. We repeated every experiment 5 times on both virtual machines and containers for a total of 160 samples collected about scalability.

Figure 6 shows the results. Again, the remarkable difference between virtual machines and containers is confirmed.

The scalability experiment gives additional evidence of how the linear growth of the total time for virtual machines is far more steep than containers. The difference is dramatic when considering the deployment of 30 probes, which requires around 10 minutes, in contrast with containers that can complete this operations in seconds.

The results show that sharing probes between multiple operators can significantly improve the efficiency of the monitoring system. This is particularly important for virtual machines where the probe deployment cost can be cut thanks to probes sharing.

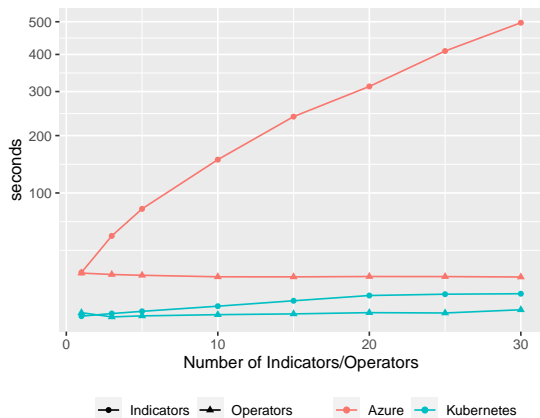


Fig. 6: Scalability results.

Answer to RQ3 Overall, results show that dynamic probe deployment can be feasible with both virtual machines and containers. However, the former environment can efficiently deal with probes only if changes are sporadic and the number of parallel requests received is limited. On the contrary, the container technology is definitely ready to support dynamic probe deployment, even in rapidly evolving contexts, based on the proposed framework.

6.5 Threats to Validity

The threats to the validity of the results mainly concern with the relationship between the technical setup of the experiment and the collected time figures. In fact, efficiency is affected by both the available computational resources and the choice of the probes. However, while changing the available computational resources and the deployed probes are likely to affect absolute figures, the trends and gaps between the different frameworks and cloud platforms are clear, despite these factors. In fact, plots for virtual machines and containers are similar, although values are on different scales. Further, the scalability trends clearly identify a single case (collecting increasingly more indicators on virtual machines) that scales remarkably worst than the others.

In our evaluation, we also selected a specific target service to be monitored (i.e., PostgreSQL) and we also used a specific malfunctioning probe (Metricbeat for PostgreSQL). Both these choices do not likely affect our results. In fact, the cost of handling a monitoring unit does not depend on the monitoring target, and similarly the error handling policy is the same for every type of error and malfunctioning probe.

Finally, the collected time figures might be affected by noise. To mitigate this issue we repeated experiments between 5 and 10 times. The reported boxplots show a low variance for the collected values, suggesting that measures are stable and meaningful, and can be used to derive valid conclusions.

7 RELATED WORK

Due to the size, complexity, and dynamicity of cloud systems, Monitoring-as-a-Service (MaaS) systems are increasingly studied to better cope with the requirements of the cloud [25], [26]. In this paper, we focused on the inability of the monitoring solutions to cope with the dynamic deployment of probe artifacts, which is required to effectively address scenarios that imply quickly or frequently changing the collected indicators.

Some MaaS systems are designed to operate *tightly coupled with the target cloud technology*. Although the knowledge of the target platform may simplify the design of the MaaS solution, the resulting architecture lacks generality and is inherently limited to the collection of platform-level indicators, missing to cope with application-level indicators. For instance, MonPaaS [27] is an open-source monitoring solution tightly coupled with OpenStack that provides a MaaS model to cloud users. It collects indicators by integrating Nagios with OpenStack and it obtains VM status information by intercepting messages in the OpenStack message queue. Moreover, several commercial cloud monitoring solutions are developed to monitor resources running on specific cloud platforms, such as Amazon CloudWatch [28] and Google Cloud Monitoring [29]. As a result, these solutions have limited interoperability and applicability. On the other hand, general purpose monitoring tools like Prometheus [10] and Zabbix [30] struggle with adapting to changing needs. In contrast, the framework proposed in this paper provides automation while relying on a technology-agnostic architecture that can operate probes of different type.

On the other hand, MaaS systems are normally limited to *interactions with pre-deployed probes*, which can be activated and deactivated by operators, but cannot be deployed/undeployed. For instance, CLAMS is a MaaS framework that can monitor, benchmark and profile applications hosted on multi-clouds environments [31], [32]. MEASURE is a Monitoring-as-a-service (MaaS) framework to monitor the cloud using stream processing [33] that relies on a publish-subscribe architecture to push data from resources through stream processors that convert and deliver data to stream subscribers. AdaptiveMon [34] is a peer-to-peer monitoring solution that exploits reconfigurable probes to address the complexity of the fog environment. Unlike our framework, these approaches rely on pre-deployed probes.

In some cases, MaaS systems use probes that can address *autoscalable services*. For instance, Amazon CloudWatch [28] uses the MaaS technology to deliver status monitoring capabilities in presence of autoscalable services. JCatascopia [13] is a monitoring solution that targets elastic tasks using automatic discovery capabilities. Compared to these solutions, in addition to managing probes that have native support to services autoscaling and discovery, our framework delivers automatic and declarative probe deployment capabilities,

allowing users to continuously and efficiently adapt the monitoring infrastructure.

Early contributions related to automatic probe deployment for cloud systems are the work by Ciuffoletti [35], who proposed a MaaS model for the deployment of monitoring components as required by users, and Anisetti et al. [36], who applied automatic probe deployment to monitor and certify security properties of services running on virtual machines. These works describe high-level design and prototype implementation targeting specific cases. The framework presented in this paper provides instead a general and applicable approach for dynamic probes deployment.

8 CONCLUSIONS

Monitoring systems must be able to cope with dynamically changing and unstable sets of monitored indicators, to effectively address scenarios that characterize modern cloud systems. Current solutions however badly adapt to these scenarios, providing little flexibility and requiring significant manual effort to deploy, undeploy and re-configure monitoring probes.

In this paper, we present a framework that can be used to dynamically work with monitoring units and probes. The operator interacts with the monitoring system in a as-a-service fashion, specifying the indicators that must be collected, and letting the framework to deal with probe deployment and configuration. The framework is also able to recover from deployment problems and integrate probes from multiple monitoring technologies.

Results show that the framework can be feasibly used with cloud systems based on both virtual machines and containers, although it is significantly more efficient with containers.

We identify three main limitations of the current framework implementation that we want to address as part of future work. First, fine-grained control of the probes configurations (e.g., changing the sampling rate of the individual probes) is not supported. This limitation can be potentially addressed by enriching monitoring claims with information about probe configurations. Second, the support to elasticity right now depends on the probe intelligence (e.g., it requires the probes to embed a discovery mechanism as the one in the Metricbeat Kubernetes module). It would be interesting to move this support at the level of the monitoring framework, so that any probe can be used to monitor elastic services. Third, error-handling is limited to the deployment phase, and it is unable to detect and repair run-time errors that occur during the regular execution of the monitoring system. Indeed, error handling requires further research to cover the full range of situations.

Other future work involves extending the framework with as-a-service anomaly detection and healing capabilities to increase the dependability of the monitored system, and caring of the vicinity of the monitored resources to further improve the deployment of the monitoring system.

REFERENCES

- [1] hostingtribunal.com. (2020) 25 must-know cloud computing statistics in 2020. <https://hostingtribunal.com/blog/cloud-computing-statistics/>. [Online; accessed 28-April-2021].
- [2] A. J. Ferrer, J. M. Marquès, and J. Jorba, "Towards the decentralised cloud: Survey on approaches and challenges for mobile, ad hoc, and edge computing," *ACM Computing Surveys*, vol. 51, no. 6, 2019.
- [3] L. Romano, D. D. Mari, Z. Jerzak, and C. Fetzer, "A Novel Approach to QoS Monitoring in the Cloud," in *2011 First International Conference on Data Compression, Communications and Processing*, Jun. 2011, pp. 45–51.
- [4] P. Cedillo, J. Jimenez-Gomez, S. Abrahao, and E. Insfran, "Towards a Monitoring Middleware for Cloud Services," in *2015 IEEE International Conference on Services Computing*, 2015, pp. 451–458.
- [5] A. Shatnawi, M. Orrù, M. Mobilio, O. Riganelli, and L. Mariani, "CloudHealth: A Model-Driven Approach to Watch the Health of Cloud Services," in *Proceedings of the 1st International Workshop on Software Health (SoHeal 2018)*. ACM/IEEE, 2018, pp. 40–47.
- [6] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf, "A Flexible Architecture Integrating Monitoring and Analytics for Managing Large-scale Data Centers," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ser. ICAC '11. ACM, 2011, pp. 141–150.
- [7] M. Kutare, K. Schwan, G. Eisenhauer, V. Talwar, C. Wang, and M. Wolf, "Monalytics: online monitoring and analytics for managing large scale data centers," in *In ICAC '10: Proceeding of the 7th international conference on Autonomic computing*. ACM, 2010, pp. 141–150.
- [8] The Linux Foundation. (2021) Kubernetes. [Online; accessed 26-April-2022]. [Online]. Available: <https://kubernetes.io/>
- [9] Elasticsearch BV. (2021) The Elastic Stack. <https://www.elastic.co/elastic-stack>. [Online; accessed 26-April-2022].
- [10] The Linux Foundation. (2021) Prometheus. <https://prometheus.io/>. [Online; accessed 26-April-2022].
- [11] Red Hat, Inc. (2021) How Ansible Works. <https://www.ansible.com/overview/how-ansible-works>. [Online; accessed 26-April-2022].
- [12] Puppet. (2021) Puppet. <https://puppet.com>. [Online; accessed 26-April-2022].
- [13] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Jcatascopia: Monitoring elastically adaptive applications in the cloud," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2014, pp. 226–235.
- [14] Hewlett-Packard Enterprise Development LP. (2021) Monasca - an OpenStack Community project. <https://monasca.io/>. [Online; accessed 26-April-2022].
- [15] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn, "A survey of cloud monitoring tools: Taxonomy, capabilities and objectives," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2918 – 2933, 2014. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0743731514001099>
- [16] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.
- [17] A. Tundo, M. Mobilio, M. Orrù, O. Riganelli, M. Guzmàn, and L. Mariani, "Varys: An agnostic model-driven monitoring-as-a-service framework for the cloud," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), tool demo*, 2019.
- [18] W. Stallings, *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Addison-Wesley Longman Publishing Co., Inc., 1998.
- [19] B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, 2016, pp. 108–113.
- [20] The Apache Software Foundation. (2021) Apache Kafka. <https://kafka.apache.org/>. [Online; accessed 26-April-2022].
- [21] MongoDB, Inc. (2021) MongoDB. <https://www.mongodb.com/>. [Online; accessed 26-April-2022].
- [22] Salvatore Sanfilippo and contributors. (2021) Redis.io. <https://redis.io/>. [Online; accessed 26-April-2022].
- [23] Elasticsearch BV. (2021) Beats: Data Shippers for Elasticsearch. <https://www.elastic.co/beats/>. [Online; accessed 26-April-2022].
- [24] ——. (2021) Elasticsearch: RESTful, Distributed Search & Analytics. <https://www.elastic.co/elasticsearch/>. [Online; accessed 26-April-2022].
- [25] S. Meng and L. Liu, "Enhanced monitoring-as-a-service for effective cloud management," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1705–1720, 2013.

- [26] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu, "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends," in *2015 IEEE 8th International Conference on Cloud Computing*, 2015, pp. 621–628.
- [27] J. M. A. Calero and J. G. Aguado, "Monpaas: an adaptive monitoring platform as a service for cloud computing infrastructures and services," *IEEE Transactions on Services Computing*, vol. 8, no. 1, pp. 65–78, 2014.
- [28] Amazon Web Services, Inc. (2021) CloudWatch. <https://aws.amazon.com/cloudwatch/>. [Online; accessed 26-April-2022].
- [29] Google. (2021) Cloud monitoring | google cloud. <https://cloud.google.com/monitoring>. [Online; accessed 26-April-2022].
- [30] Zabbix LLC. (2021) Zabbix features overview. <https://www.zabbix.com/features>. [Online; accessed 26-April-2022].
- [31] K. Alhamazani, R. Ranjan, K. Mitra, P. P. Jayaraman, Z. Huang, L. Wang, and F. Rabhi, "Clams: Cross-layer multi-cloud application monitoring-as-a-service framework," in *2014 IEEE International Conference on Services Computing*, 2014, pp. 283–290.
- [32] K. Alhamazani, R. Ranjan, P. Prakash Jayaraman, K. Mitra, C. Liu, F. Rabhi, D. Georgakopoulos, and L. Wang, "Cross-layer multi-cloud real-time application qos monitoring and benchmarking as-a-service framework," *IEEE Transactions on Cloud Computing*, vol. 7, no. 1, pp. 48–61, 2019.
- [33] M. Smit, B. Simmons, and M. Litoiu, "Distributed, application-level monitoring for heterogeneous clouds using stream processing," *Future Gener. Comput. Syst.*, vol. 29, no. 8, 2013.
- [34] V. Colombo, A. Tundo, M. Ciavotta, and L. Mariani, "Towards Self-Adaptive Peer-to-Peer Monitoring for Fog Environments," in *Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2022.
- [35] A. Ciuffoletti, "Application level interface for a cloud monitoring service," *Computer Standards & Interfaces*, vol. 46, pp. 15 – 22, 2016.
- [36] M. Anisetti, C. A. Ardagna, E. Damiani, and F. Gaudenzi, "A semi-automatic and trustworthy scheme for continuous cloud service certification," *IEEE Transactions on Services Computing*, vol. 13, no. 1, pp. 30–43, 2017.



Alessandro Tundo is a Ph.D. student at the University of Milano-Bicocca. He holds a Master Degree in Computer Science received from the same university in 2018.

His research interests include cloud and fog computing, distributed systems monitoring and software architectures.



Marco Mobilio is a post-doc at the University of Milano-Bicocca, where he got his Ph.D. in 2017 and his Master Degree in Computer Science in 2013. His main interests cover Software Architecture, Cloud Monitoring and Self-Healing, Automatic Testing for web and mobile applications and Human Activity Recognition.



Oliviero Riganelli is an Assistant Professor at the University of Milano-Bicocca. He holds a Ph.D., in Computer Science and Complex System from the University of Camerino in 2009. He is a computer scientist with a keen interest in Software Engineering. His main research interests focus on creating advanced methodologies and technologies to build better software by automatically testing, analyzing, and correcting the software itself and its development process.

He is and has been involved in several research projects, both international and national, in close collaboration with leading partners from industry and academia. He is also regularly involved in the program committees of workshops and conferences in his areas of interest



Leonardo Mariani is a Full Professor at the University of Milano-Bicocca. He holds a Ph.D. in Computer Science received from the same university in 2005.

His research interests include software engineering, in particular software testing, program analysis, automated debugging, specification mining, and self-healing and self-repairing systems. He has authored more than 100 papers appeared at top software engineering conferences and journals.

He has been awarded with the ERC Consolidator Grant in 2015, an ERC Proof of Concept grant in 2018, and he is currently active in several European and National projects. He is regularly involved in organizing and program committees of major software engineering conferences.