Monitoring Probe Deployment Patterns for Cloud-Native Applications: Definition and Empirical Assessment

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

Abstract—*Monitoring* is a key feature to enhance systems with the capability to anticipate, detect, predict, and mitigate failures, while providing Quality of Service (QoS) monitoring and Service Level Agreements (SLAs) guarantee. Monitoring frameworks can serve these purposes by deploying probes according to many possible patterns that have different features, for instance in terms of efficiency and privacy. So far, these probe deployment patterns have not been systematically defined, analyzed and assessed. Thus, engineers who design and configure their monitoring systems have to take decisions only based on *partial knowledge* and *personal experience*.

This paper addresses this knowledge gap, by presenting a systematic analysis of *11 probe deployment patterns*, their known uses, and implementations. We assess these patterns qualitatively, and quantitatively using both VMs and containers. Results show the targets have negligible resource consumption (e.g., less than 1% CPU usage), while the probe holder consumption is mainly significant in relation to memory consumption, reaching up to 10 GiB in our experiments. Our findings suggest that reusing probes and holders among users can generally enhance efficiency and scalability when direct access to the monitored target is not an option. We generate a set of *best practices* that can assist engineers in configuring their monitoring systems. Finally, we showcase the application of certain patterns through three practical usage scenarios, which feature diverse technologies and requirements.

Index Terms—Monitoring Probes, Probe Deployment, Deployment Patterns, Design Patterns, Monitoring Best Practices, Container Monitoring, Cloud Monitoring, Cloud Computing.

1 Introduction

Cloud computing is the de facto standard platform for the global connectivity of actors (e.g., humans, robots, devices and sensors) and services, across many heterogeneous domains (e.g., mobile [1], health care [2], IoT [3] and telecommunication [4]). In this scenario, actors and services may have to frequently interact in ways that cannot be fully anticipated, demanding for strong configurability, adaptability and programmability requirements [5], [6], [7], [8], [9], [10].

Highly dynamic systems require appropriate tools and techniques for continuously verifying their behavior, so that deviations can be timely detected and compensated. In this context, *monitoring* is a key feature [11], [12], [13] to enhance systems with the capability to detect, predict, and mitigate failures [14], [15], [16], [17], in addition to enable more traditional operations, such as Quality of Service (QoS) monitoring [18], [19], [20] and Service Level Agreements (SLAs) guarantee [21], [22], [23].

There are several effective monitoring frameworks that can be used to implement a monitoring system for cloud applications. For instance, the Elastic Stack [24] can be used to run probes that push the monitored data into an Elasticsearch time series database, or Prometheus [25] can be used to pull data from probes into its database. Regardless of the adopted framework, the resulting cloud monitoring system is a distributed system that runs potentially many probes configured to collect several indicators for multiple operators.

The flexibility of monitoring frameworks and probe technologies allows for *diverse probe deployment patterns*, which consist of probe deployment architectures targeting specific environments (e.g., a container-based environment) and satisfying specific constraints (e.g., probes must be shared among multiple operators). The choice of a probe deployment pattern has implications on the effectiveness and efficiency of the resulting monitoring system. For instance, multiple probes serving different operators in a multi-tenant environment can be deployed within a same virtual machine to save resource consumption, at the expense of a reduced degree of privacy and security. On the other hand, one probe per container or virtual machine can be deployed to preserve privacy, at the expense of more resources allocated to the monitoring system.

The many possible probe deployment patterns have not been analyzed and assessed systematically so far, and the engineers who design their monitoring systems are called to take decisions whose implications might be relatively well-known. The existing body of work discusses the characteristics of monitoring systems, without investigating the many possible probe deployment patterns and their impact [11], [12], [13]. To address this knowledge gap, this paper *systematically presents and analyzes the possible probe deployment patterns, their known uses, and implementations*. This requires the careful identification and characterization of the components that can be used to deploy probes. Further, we assess these patterns both quantitatively and qualitatively, distilling findings that can guide engineers in

- *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*

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2024.3349648

the implementation and configuration of their monitoring systems.

In particular, we experimented with 11 patterns with a range of configurations requiring a total of about 154 hours of data collection time and more than 175 hours of total runtime for the empirical evaluation, including setup and teardown of the VMs and containers. Results show the trade-offs between the patterns that require more resources while guaranteeing good separation between users acting within multi-tenant environments, and patterns that make better use of resources while reducing the degree of separation. We cross-validate the obtained results by addressing three realistic monitoring scenarios. The experimental material containing all the software artifacts and the collected dataset is publicly available at [26].

In a nutshell, this paper provides the following contributions:

- it systematically analyzes the possible probe deployment patterns and defines a feature diagram that comprehensively captures the possible deployment strategies;
- it provides a set of empirically-derived best practices for deploying probes, which can serve as a reference for future use;
- it qualitatively and quantitatively assesses the patterns, to understand their strengths and weaknesses;
- it showcases the implementation of some patterns through three realistic usage scenarios involving different monitoring technologies, application architectures, and requirements.

The paper is organized as follows. Section 2 provides background information about monitoring systems. Section 3 defines the probe deployment patterns, which are illustrated according to a same structure. Section 4 compares probe deployment patterns qualitatively. Section 5 quantitatively assesses the probe deployment patterns with a set of comparative experiments. Section 6 discusses some best practices distilled from our experiments. Section 7 showcases the implementation of some patterns through realistic usage scenarios. Section 8 discusses related work. Section 9 provides final remarks.

2 Background Information

This section provides the background information useful to understand the probe deployment patterns introduced in Section 3.

We use the term *resource* to generically refer to any cloud element that can be monitored, including services, hardware resources, and virtualized components. A *probe* is an artifact that runs close enough to the monitored resource to observe its behavior, and it collects observations of one or more *Key Performance Indicators (KPIs)*. A *KPI* is a measurable and quantifiable metric used to track the behavior of a *resource*. We mainly refer to observations resulting in time series data (e.g., CPU consumption data), but the probe deployment patterns are also valid for observations generating other types of data (e.g., log files).

A cloud monitoring system generally consists of four key components: a set of probes, a time series database, a data transfer channel, and a dashboard. Fig. 1 graphically illustrates these components.

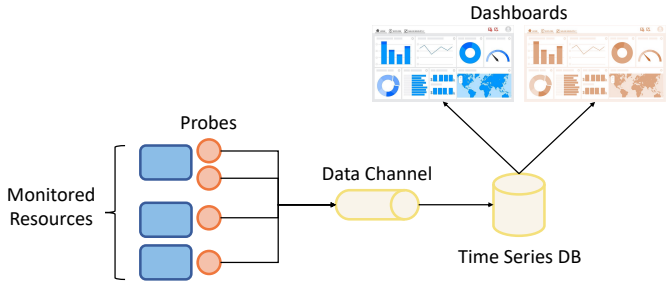


Fig. 1: Architecture of a cloud monitoring system.

The *set of probes* is opportunistically distributed within a cloud system to efficiently collect data from the *monitored resources*. Depending on the type of probes, the collected data can be shipped according to different patterns, for instance probes could push or pull data according to different policies.

A *time series database* is used to store the data obtained from the probes. The communication between the probes and the database is usually mediated by a *data transfer channel* that is responsible for processing and transferring the data. In some cases the channel could be as simple as direct communication between the probes and the database. In some other cases, the channel is a data processing pipeline that is able to pre-process and distribute data, according to non-trivial strategies.

The *dashboard* is finally used to access and visualize the data stored in the time series database. When the collected data is used to support advanced analysis routines, multiple tools may analyze the collected data (e.g., alerting and notification systems and anomaly detectors).

This work thoroughly investigates the organization of the probes, studying the possible deployment patterns, and their impact on the efficiency and effectiveness of the monitoring system.

3 Probe Deployment Patterns

Probe deployment patterns capture how probes can be deployed to monitor the target resources. The possible deployment depends on several key features that we discuss in this section and are represented with the feature diagram shown in Fig. 2.

A feature diagram is a graphical representation of a feature model that defines features and their dependencies in a tree structure [27]. In this case the model characterizes the features relevant to probe deployment patterns. The inner nodes represent abstract features (features that are not implemented but only used to group features), while the leaf nodes represent the concrete features (features that are implemented). The parent-child relationship represents the feature decomposition, from abstract to concrete features. While the default interpretation of feature decomposition is the AND relationship, other decomposition are possible, such as the alternative decomposition that indicates that only one feature can be selected among the ones that are available (see the legend in the figure). Finally, features can

be optional or mandatory. All features are mandatory in our diagram. A combination of features is a configuration. A configuration is admissible if it satisfies its feature diagram.

A feature diagram may also include logical constraints that limit the set of admissible configurations, that is, only the configurations that satisfy the specified constraints can being admitted by the model. Our model also includes several constraints that prevent that infeasible or highly inefficient configurations can be admitted by the model, thus guaranteeing the reasonableness of the result.

A feature model can be used to automatically generate the space of all the admissible configurations. In fact, the configurations admitted by the model in Fig. 2 represent every possible probe deployment pattern.

We started from the papers that propose monitoring and probe deployment approaches for multi-tenant and technology-heterogeneous cloud environments [31], [28], [32], [29], [30], [37], [46], the most used cloud monitoring tools [40], [39], [43], [44], [42], and our experience, to identify and distill a set of relevant features for probes deployment. We discuss below the semantics of the considered features.

- **Probe Holder**: it represents the object that hosts the probes that are executed (N.B., hereinafter referred as holder). It can be a separate Virtual Machine or Container, or can overlap with the target execution environment.
  - *Holder Type*: it represents the holder type [28]
    - Target: the holder is the target of the monitoring activity, that is, the holder hosts both the target and the monitoring probes
    - External Unit: the holder is an external object which monitors the target from the outside (e.g., a sidecar container [28], [29], [43], [39], [40], [44], [42])
  - *Probe Multiplicity*: it defines the number of probes that can be executed within the unit [30], [29]
    - Single-probe: only one probe can be executed
    - Multi-probe: one or more probes can be executed
  - *Holder Sharing*: it defines if the holder can be shared among multiple users [31], [32], [46], [37]
    - Reserved Holder: the holder is reserved to a single user
    - Shared Holder: the holder can be shared among users
- **Probe Instance**: it represents probe artifact executed within the holder to collect data.
  - *Target Multiplicity*: it defines the number of targets that a single probe can monitor simultaneously [40], [29], [39], [42], [43], [44]
    - Single-target: a probe can monitor only one target
    - Multi-target: a probe can monitor multiple targets
  - *Instance Sharing*: it defines if a probe instance within a holder can be shared among users [31], [32], [46], [37]
    - Reserved Probe: the probe collects data for a single user
    - Shared Probe: the probe can collect data for multiple users
- **Execution Environment**: it defines the supported execution environment.
  - *Environment Type* [30]:
    - System-oriented: monitoring is performed within a virtualized entity aimed at offering a system-level environment. This is usually the case with Virtual Machines and system-level containerization technologies, such as LXC, OPenVz, and Linux-VServer.
    - Application-oriented: monitoring is performed within a virtualized entity aimed at offering an application-level environment. This is the case of common containerization technologies, such as Docker or Containerd.

The admissible configurations are bounded by constraints that capture bad/best practices and unfeasible combinations, as follows:

a) *A shared holder that executes at most a single probe must allow for the execution of shared probes* (*Shared Holder* $\wedge$ *Single-probe* $\Rightarrow$ *Shared Probe*): If the holder must be shared but only one probe can be executed within the holder, the only way to actually share resources is to allow for probes that can be shared among multiple users.

b) *If the holder is reserved to a single user, also the probes running within that holder must be serving that user* (*Reserved Holder* $\Rightarrow$ *Reserved Probe*): Clearly, if the holder is reserved to a single user, it is impossible to install probes serving multiple-users within that holder.

c) *Each system-oriented virtualization unit should possibly run multiple probes* (*System-oriented* $\Rightarrow$ *Multi-probe*): Virtual machines are expensive units whose instantiation should be limited to prevent excessive resource consumption, due to their non-negligible size and significant bootstrapping cost [34], [35]. For this reason, reserving a virtual machine to a single probe is strongly discouraged, and it should rather be used to run multiple probes.

d) *Each Application-oriented virtualization unit should not run more than one probe* (*Application-oriented* $\Rightarrow$ *Single-probe*): Following good design practices concerning isolation and separation of concerns [29], each container should run one process at most, and thus each container should be dedicated to a distinct monitoring probe, so that any interference is prevented.

e) *A probe sharing the holder with the target should only monitor that target* (*Target* $\Rightarrow$ *Single-target*): When a probe is installed within the same holder (e.g., a virtual machine) that runs the target of the monitoring activity, the probe should not be configured to monitor something else hosted outside the target, otherwise it may interfere with the activity of the target. This follows the practice that, if needed, probes might be running within the same holder of the target to circumvent observability issues. For instance, collecting memory consumption either about a process running inside a VM, or the VM itself, may not be possible via external interfaces or at hypervisor level [36], [37], and in such cases probes are specifically configured to extract data from that target.

f) *If probes are allowed to run within the same holder of*

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2024.3349648

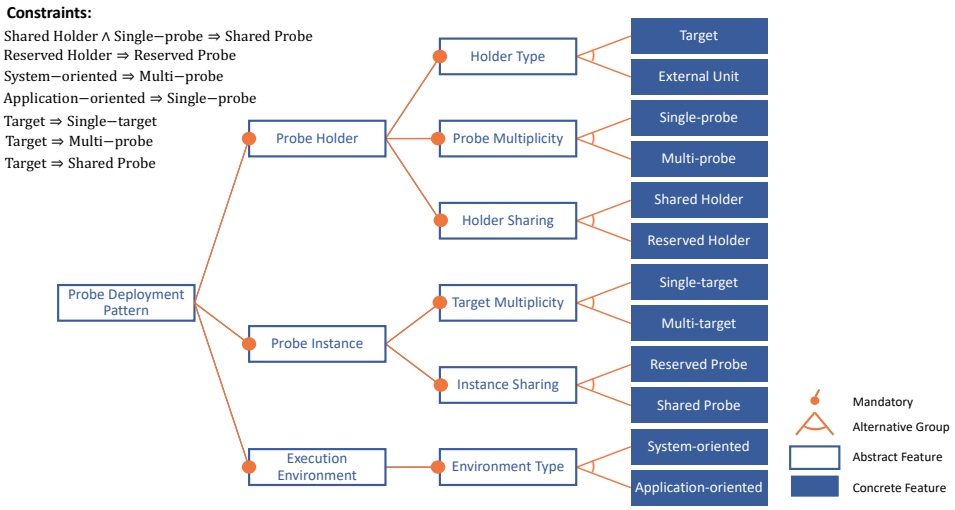


Fig. 2: Probe deployment patterns feature diagram.

the target, more than one probe should be allowed to run (*Target* ⇒ *Multi-probe*): Limiting the target to host a single probe would limit the monitoring system to the collection of a single (set of) KPIs, which would not be acceptable in the majority of practical cases.

g) *A probe sharing the holder with the target should be shared among users* (*Target* ⇒ *Shared Probe*): Using the holder space for both the target and the probes may raise interference issues. For this reason having multiple copies of functionally-equivalent probes to serve multiple users is particularly inefficient and risky, despite ownership concerns. Thus, additional probes should be installed only to collect data that are not collected by the already existing probes, which have to be shared among users.

We encoded all these concepts and constraints in the feature diagram in Fig. 2, which has been implemented using FeatureIDE [38], a tool for feature-oriented software development based on Eclipse. We generated all the admissible configurations from the model automatically, taking also into account the specified constraints. The tool created 11 admissible configurations corresponding to 11 probe deployment patterns, which are by product correct, according to the features and constraints represented in the feature model. In this work, we focused on the key features that characterize a set of monitoring probes. In the future, the model could be extended to incorporate additional features and constraints, which could be used to refine the set of probe deployment patterns.

Fig. 3 provides a graphical representation of the probe deployment patterns, and proposes names coherent with their structure. The illustrations consistently refer to a case with two targets and two users, which is sufficient to exemplify the differences among the various patterns. The number of monitoring units and probes varies according to the configuration. We use colors to represent ownership (a monitoring unit or a probe of the same color of a user indicates the ownership of the user, while multicolored elements represent shared resources).

We adopt a same schema to illustrate each pattern. In particular, we use the following fields: *name*, which defines the name of the pattern; *description*, which provides a short description of the pattern, and *target technology*, which indicates the technical environment in which the pattern is used.

We also defined a naming convention to easily recall the details of a pattern from its name. Specifically the name of each pattern is obtained by concatenating three elements:

- The first element represents the level of sharing of the pattern, which could be *Reserved*, *Shared*, or *Partially Shared*. *Reserved* is used for holders reserved to individual users. *Shared* is used for shared holders running shared probes. Finally, *Partially Shared* is used for shared holders that run reserved probes.
- The second element represents the type of executed probes. We use *T** for probes that can monitor multiple targets, while we use *T1* for probes that monitor a single target.
- The third element represents the probe multiplicity. We use *P1* for holders that run a single probe. While we use *P** for holders that can run multiple probes.

For example, the Partially-shared-T1P* pattern identifies the case of a shared holder that can run multiple probes configured to serve individual users and collect data from individual targets.

Table 1 contains a detailed description of each identified pattern, along with information regarding its target technologies.

It is worth detailing further the Internal-T1P$*$ pattern which is the only one where the holder matches with the target execution unit. In this unique instance, probes can gain the highest observability as they have the privileged viewpoint of collecting data from inside the same execution unit hosting the target. Hence, probes may easily observe indicators that would otherwise be hard or even impossible to collect.

The implementation of this pattern can be highly intrusive as the probes and target share execution unit resources. Additionally, users cannot operate reserved probes unless the monitoring system permits single-user access. There could also be challenges in precisely gathering indicators

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2024.3349648

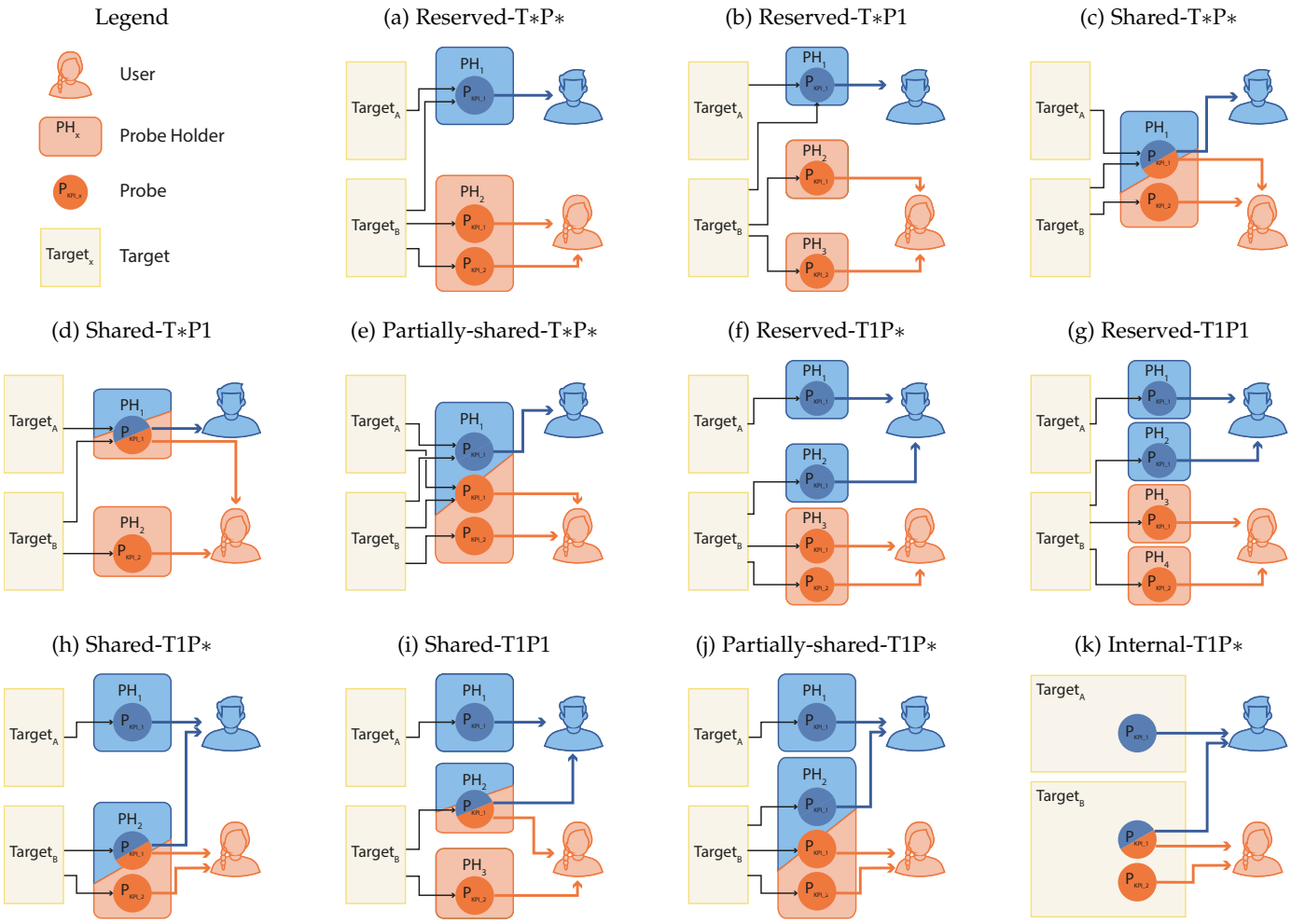


Fig. 3: Probe deployment patterns.

specific to the target. This is particularly true for resource-related metrics such as memory and CPU usage, which could be influenced by the inclusion of probes that also consume resources.

4 Qualitative Discussion

This section discusses the qualitative aspects related to the presented patterns. We first discuss how the patterns can be implemented with different technologies. We then discuss the trade-off between separation and resource consumption. We conclude by discussing interoperability, portability, robustness, affordability and security of the patterns. Table 2 summarizes how patterns can be classified according to these seven dimensions. For ease of comparison, patterns have been grouped into three groups: (i) patterns that reserve both holders and probes to individual users (column *Patterns that privilege reservation*); (ii) patterns that share both probes and holders among users (column *Patterns that privilege sharing*); and (iii) patterns that share holders among users, but run probes reserved to individual users (column *Patterns that balance the two aspects*).

4.1 Pattern Implementation

To show the practical applicability of the patterns, in this section we provide guidance on how the identified patterns can be implemented with current real-world monitoring tools. Further guidance on applying patterns in real-world contexts is provided in Section 7, where realistic usage scenarios are reported.

A common way to implement the patterns with reserved resources (*Reserved-T∗P1*, *Reserved-T∗P∗*, *Reserved-T1P∗*, and *Reserved-T1P1*) with platforms such as Prometheus or the Elastic Stack is to have multiple instances of the framework, one for each user, and then deploy their holders with *agentless* Prometheus exporters [39] or Beats [40], such as SNMP [41], [42] or HTTP based probes. This is also the case of tools such as Zabbix [43] in its agentless configuration, where it is expected to handle multi tenancy with the deployment of distinct components for each tenant. The reserved aspect of both the probes and the holders can be implemented either deploying distinct instances of the full monitoring system or employing a probe-deployment framework that can support multi-tenancy [30]. These patterns are well supported also by commercial tools, such as Nagios [44] and Dynatrace [45], and in scientific articles, such as [46], [37].

TABLE 1: Probe Deployment Patterns: Description and Target Technology

Description	Target Technology
Reserved-T∗P∗ (Fig. 3a): it uses a reserved holder for each user. A single holder hosts multiple probes able to gather information from multiple targets. Note that in this configuration, to increase data separation, both the holder and the probes are reserved to a single user, but a single probe can gather the same KPI from multiple targets.	Since each holder can host multiple probes, this pattern is usually applied to system-oriented virtualization technologies.
Reserved-T∗P1 (Fig. 3b): it uses multiple reserved holders for each user, one for each probe deployed. Although there is a one-to-one relationship between holders and probes, probes are enabled to gather data from multiple targets.	This pattern is tailored for application-oriented virtualization technologies (e.g., Docker containers) since each holder only contains the process of a single probe. It is discouraged to exploit this pattern with system-oriented virtualization technologies (e.g., VMs) since the cumulative overhead caused by holders is likely unaffordable for non-trivial settings.
Shared-T∗P∗ (Fig. 3c): it uses a single holder, shared among different users. The holder can contain multiple probes, which are also shared among the users for collecting the same KPIs from multiple targets.	Due to the presence of multiple processes within the same holder, this pattern is specific to system-oriented environments.
Shared-T∗P1 (Fig. 3d): the holders are shared among users and contain a single probe that is able to acquire data from multiple targets and for multiple users, if needed.	This pattern targets application-oriented environments because, while sharing probes among users and targets promotes optimization and reuse, having a dedicated holder for each probe can cause a significant overhead for the monitoring solution if using heavier virtualized units (e.g., VMs).
Partially-shared-T∗P∗ (Fig. 3e): it uses a single holder, shared among users that contains multiple probes able to acquire the KPIs from multiple targets. In case the same KPI is requested by multiple users, the probe is instanced multiple times within the same holder, one for each user that requested the KPI.	This pattern is specific to system-oriented environments as it creates a small number of holders with multiple processes running in each one (i.e., multiple probes).
Reserved-T1P∗ (Fig. 3f): it uses a reserved holder for each user. Moreover each holder is allowed to contain only probes that acquire data from a single target, however, if a single user requires to collect multiple KPIs from the same target, multiple probes can be placed within the same holder.	Due to the fact that a holder may contain multiple probes, this pattern is suited for system-oriented environments.
Reserved-T1P1 (Fig. 3g): it uses a reserved holder for each user and contains a single probe. Every probe is dedicated to the collection of KPIs from a single target, which means that if a single user requests the same KPI from a given number of targets, an equal number of holders and probes will be deployed to fulfill such request.	This pattern is dedicated to application-oriented environments as it fulfills the requirement of having a single process in each holder.
Shared-T1P∗ (Fig. 3h): it uses a holder for each target. Such holder may contain multiple probes that can acquire KPIs from a single target. Since there is only one holder for a specified target, multiple users interested in monitoring such target share the holder. Moreover users also share the actual probes within the holder.	Given the fact that each holder can contain multiple probes, this pattern is tailored for system-oriented environments.
Shared-T1P1 (Fig. 3i): it uses multiple holders for each target, where each holder contains only one probe that acquires data from the target and shares the data among all the users.	This pattern is designed to be applied to application-oriented environments, mainly due to the high number of holders that it can generate.
Partially-shared-T1P∗ (Fig. 3j): it uses a single holder for each target, shared among users. However. the probes are not shared among users, implying that if two or more users wish to monitor the same KPI, there will be an equal number of instances of the same probe deployed, each one dedicated to a single user.	Since this pattern involves a single holder for each target with a number of probes deployed within it, it is aimed at system-oriented environments.
Internal-T1P∗ (Fig. 3k): it is the only case in which the holder matches with the execution unit that hosts the target. The probes run within the same execution unit that runs the target (e.g., within a VM). If the same indicators are collected by multiple users for the same targets, the probes are necessarily shared, mitigating the possibility of interfering with the target.	This pattern is specific to system-oriented environments since its nature implies multiple processes running in the target.

The patterns with shared resources (*Shared-T∗P∗*, *Shared-T1P1*, *Partially-shared-T∗P∗*, *Shared-T∗P1*, *Shared-T1P∗*, *Partially-shared-T1P∗*) are easy to implement with base technologies, such as Prometheus and the Elastic Stack, as they exploit components that can be installed in a single shared holder configured to permit multiple users to access the data gathered from the deployed probes. These patterns are available also within commercial systems [45], [44], [43], and in scientific articles, such as [47], [48], [49], [50], [51]

The *Internal-T1P∗* pattern can be found in many agent-based solutions [52], [53], [54]. It could also be obtained in Prometheus, by installing its exporters directly in the target VMs, and similarly with Elastic Stack, by installing beats and custom probes directly within the target VMs.

The study of approaches to switch from one pattern to another is beyond the scope of this work. Nevertheless, designing monitoring systems that can automatically change the deployment pattern according to changes in monitoring needs would be valuable. In the context of automated deployment of holders and probes, one feasible method involves the use of Monitoring-as-a-Service (MaaS) frameworks [54], [55], [30]. For instance, Tundo et al. [30] proposed a MaaS framework that has the ability to automatically govern the entire life-cycle of the probes

from declarative inputs, thus relieving operators of any configuration burden.

4.2 Separation Versus Resource Consumption

One of the aspects relevant to the choice of the pattern is the level of separation to be achieved, in comparison to the possible resource consumption. Separation concerns with probes and holders acting for the purpose of a single user or organization in a multi-tenant environment. Separation is beneficial to privacy, security and reliability.

Some patterns require a given level of sharing to be accepted by the users in order to be used. Depending on this choice, the behavior of the probes serving a user may impact the probes serving other users. On the other hand, guaranteeing separation requires extra resources to be allocated on the monitoring system.

Patterns that privilege reservation guarantee the maximum level of separation, but resource consumption may growth quite quickly with a growing number of users. On the other hand, patterns that favor sharing probes may save resources but require sharing probe configurations (e.g., sampling rate and accuracy) among users, and this could be problematic in some use cases.

Some patterns share the holders among users while running probes reserved to individual users (Table 2, column *Patterns that balance the two aspects*). This guarantees that probes may impact one another only through the holder, which is unlikely to happen, although possible (e.g., due to a malfunctioning probe). In terms of resources, although the number of probes may still increase quickly, the number of holders is guaranteed to stay small.

Resource consumption growth rate is quantitatively studied in detail in Section 5.

4.3 Interoperability and Portability

The proposed patterns are cloud agnostic and thus are interoperable and portable across cloud environments [56]. There might be however some practical aspects that make certain patterns more suitable for an environment than another. For instance, although the proposed patterns are conceptually applicable to both containers and virtual machines, in practice we restricted the application of some of them to certain technologies only, so as not to go against well-known and widely accepted design principles of those technologies.

Patterns are beneficial to interoperability and portability also when used to describe and model existing monitoring systems. In fact, they ease the understanding of different implementations of probe deployment designs by introducing a set of reference designs. This facilitates the understanding of the responsibilities of monitoring components, which could be easily replaced with compliant ones having the same or similar characteristics of the replaced component.

4.4 Robustness

Robustness is an important aspect of monitoring systems. Among the described patterns, the ones that use a holder that is distinct from the holder of the target service provide higher robustness (Reserved-T∗P∗, Reserved-T∗P1, Reserved-T1P∗, Reserved-T1P1). In fact, the external holder provides failure containment by isolating the monitoring modules into separate units. This allows the target's functionalities to be safeguarded despite failures in the monitoring infrastructure. For example, the target can continue serving even if the probe has failed.

In addition, these external units are deployed on dedicated VMs and containers, allowing each piece of monitoring functionality to be updated, configured and, when needed, rolled back, independently from targets, and vice versa.

Shared holders may cause the propagation of failures from the probes of a user to the probes of different users through the holders.

Finally, shared probes imply sharing failures between users (Shared-T∗P∗, Shared-T∗P1, Shared-T1P∗, Shared-T1P1). Even worst, internal probes may propagate failures to the target (Internal-T1P∗).

4.5 Affordability

Patterns that promote more efficient consumption of cloud resources offer greater assurance of affordability. These are the patterns that share resources among users, such as patterns that share the holder and/or the probe instances. A further level of resource sharing is given by the pattern (Internal-T1P∗) that shares the holder with the target holder, consequently saving also the cost of sharing messages between the probes and the target, otherwise needed with the other patterns.

4.6 Security

Security concerns may derive from the definition of the patterns and their implementation. Different patterns introduce different levels of resource sharing among users, which might be a source of concerns. For example, if an attacker takes control of a holder, all the probes running in the holder might be compromised. A compromised probe may compromise the clients using the probe. In short, shared and partially-shared probe deployment patterns expose users to higher security risks compared to reserved patterns.

Pattern implementations may also be a source of security concerns. For instance, resource pooling enables the use of the same pool of resource by multiple users through multi-tenancy and virtualization technologies. Although these technologies introduce rapid elasticity and optimal resource management, they also introduce some risks into the system. Multi-tenancy carries the risk of data visibility to other users and tracking of operations. Similarly, the virtualized environment introduces its own set of risks and vulnerabilities that include malicious cooperation between virtual components and the leakage of these.

5 Quantitative Evaluation

In this section, we quantitatively evaluate the cost-effectiveness of the probe deployment patterns by measuring their cost in terms of CPU, memory and network consumption, and their monthly operating costs. We discuss the research questions (Section 5.1), the experimental plan that was carried out (Section 5.2), the experimental setup

TABLE 2: Characterization of the Patterns

	Patterns that privilege isolation	Patterns that privilege sharing	Patterns that balance the two aspects
Patterns	Reserved-T∗P∗, Reserved-T∗P1, Reserved-T1P∗, Reserved-T1P1	Internal-T1P∗, Shared-T∗P∗, Shared-T∗P1, Shared-T1P∗, Shared-T1P1	Partially-shared-T∗P∗, Partially-shared-T1P∗
Resource Consumption	number of probes and holders growths with users	scalable growth with respect to users	only number of probes growths
Separation	no interference among users	probes shared between users who have to agree on their configuration	possible interference at the level of the holder
Interoperability & Portability	no impact	no impact	no impact
Robustness	dedicated holder increases failure containment	shared probes can propagate failures among users, internal probe can propagate failures to target	shared holder can propagate failures among users
Affordability	resource utilization requires higher cost	sharing probes and holders can reduce overall cost of resources	sharing holders reduces resource cost
Security	reserved resources can mitigate security risks	sharing probe and holder can significantly pose security risks	sharing holders can pose security risks

that we used to perform the experiments (Section 5.3), the results of the experiments that we executed to answer the research questions (Section 5.4 and 5.5), and threats to validity of our evaluation (Section 5.6).

5.1 Research Questions

The quantitative investigation of the cost-effectiveness of the probe deployment patterns concerns with the following main research question

RQ - How do patterns scale with the amount of monitored data?

Investigating the scalability of the patterns is important to determine how well the patterns can fit situations asking for different amounts of data to be collected. We consider multiple scalability dimensions, including probe overhead and cost. Since the two main target environments, system-oriented (e.g., virtual machines) and application-oriented (e.g., Docker containers) environments, are significantly different in terms of elasticity and amount of resources consumed to create and run holders, and plots would be on radically different scales, we generate two distinct sub-research questions for each target environment as follows.

RQ-SO - How do patterns for *system-oriented* environments scale with the amount of monitored data?

RQ-AO - How do patterns for *application-oriented* environments scale with the amount of monitored data?

RQ-SO and RQ-AO study how probe deployment patterns scale with respect to an increasing number of users, KPIs and targets for system-oriented and application-oriented execution environments, respectively.

5.2 Experimental Plan

To answer the two research questions, we studied the scalability of the probe deployment patterns by performing *6 experiments* each one investigating a different scalability dimension with *5 experimental configurations*. An experimental configuration consists of a triplet: the number of users considered in the experiment, the number of monitored targets, and the number of KPIs requested per user. To measure scalability, we considered how patterns consume the CPU (%), memory (GiB/MiB), and network I/O (MiB) of both the holder and the target holder. To this end, we could appreciate both how probes and holders consume resources, but also how, and if, patterns may impact on the target, also estimating the performance overhead and cost.

In each experiment, we vary at least one out of the three dimensions that compose an experimental configuration to study how the patterns handle the growth of that dimension. Table 3 summarizes the experiments we did. Column *Experiment* specifies the name of the experiment, while Column *Sequence of Exp. Configurations* reports the set of experimental configurations investigated to study scalability. Note that the sequence of configurations always have at least a growing dimension. In all the cases, the growth rate corresponds to doubling a dimension at each step. As shown in the experiments, the selected values are sufficient to appreciated the trend shown by each dimension.

In particular, the *INCREASING_KPIS_1* and *INCREASING_KPIS_2* experiments investigate the scalability of the patterns with respect to an *increasing number of requested KPIs* by a single user for a given target and by two users for a same target, respectively. That is, we investigate the impact of an increasing number of KPIs collected, both for single and multiple users.

The *INCREASING_TARGETS_1* and *INCREASING_TARGETS_2* experiments investigate the scalability of the patterns with respect to an increasing number of targets, for a user interested in collecting a given KPI, and two users interested in collecting a same KPI. That is, we investigate how a growing number of targets impact on the single and multi-user scenarios.

The *INCREASING_USERS_1* experiment investigates the scalability of the patterns with respect to an increasing number users interested in monitoring a single KPI for a given target. Finally, *INCREASING_USERS_2* investigates the scalability of the patterns with respect to an increasing number of users requesting an increasing number of KPIs for a same single target. That is, we study how a growing number of users impact on the patterns, also considered in combination with an increasing number of KPIs collected.

Overall, this set of experiments can provide a clear picture about how patterns scale according to the different dimensions. All the experiments are repeated for both patterns applicable to system-oriented technologies (e.g., VMs) and patterns applicable to application-oriented technologies (e.g., Docker containers).

When collecting data, we run each experimental con-

TABLE 3: Experiments Configurations

Experiment	Sequence of Exp. Configurations (Users - Targets - KPIs)
INCREASING_KPIS_1	(1-1-1), (1-1-2), (1-1-4), (1-1-8), (1-1-16)
INCREASING_KPIS_2	(2-1-1), (2-1-2), (2-1-4), (2-1-8), (2-1-16)
INCREASING_TARGETS_1	(1-1-1), (1-2-1), (1-4-1), (1-8-1), (1-16-1)
INCREASING_TARGETS_2	(2-1-1), (2-2-1), (2-4-1), (2-8-1), (2-16-1)
INCREASING_USERS_1	(1-1-1), (2-1-1), (4-1-1), (8-1-1), (16-1-1)
INCREASING_USERS_2	(1-1-1), (2-1-2), (4-1-4), (8-1-8), (16-1-16)

figuration 3 times for 10 minutes to collect stable results. Since we sample the resource-related metrics (CPU, memory and network) every 10 seconds, each of the experimental configurations results in 60 samples for a sampled resource-related metric. Overall, the 3 repetitions sustained for 10 minutes generates 180 samples per resource-metric for a given configuration, that gives us good confidence on the stability and significance of the results. Since we study 30 configurations, to support the 6 experiments shown in Table 3, and we repeat the experiments for the 11 patterns, we obtain a total of 330 configurations. We avoid repeating the execution of 22 configurations because some pattern configurations produce the same experimental setting (e.g., same number of deployed holders and probes). As a result, we collected 166,320 samples instead of the expected 178,200 samples (60 *samples per metrics* $\times$ 3 *metrics* $\times$ 3 *repetitions* $\times$ 330 *configurations* $= 178,200$ *samples*).

5.3 Experimental Setup

We ran the experiments on both virtual machines (VM) and containers. To automate experiments, we implemented Ansible playbooks [57] that interact with the Azure Compute Platform [58] and with a managed Azure Kubernetes Cluster [59] to run VM-based and container-based experiments, respectively.

Virtual machine holders are created with the Azure Standard B1s flavor (1 vCPU, 1GiB of RAM, Ubuntu 18.04 LTS), while VM targets are created with the Standard A2 v2 flavor (2 vCPUs, 4GiB of RAM, Ubuntu 18.04 LTS). The Kubernetes Cluster consists of a single node pool with 3 workers (Standard B4ms flavor, 4 vCPU, 16GiB of RAM) and run Kubernetes v1.20.9. We deployed container holders and targets by mean of single-replica Kubernetes Deployments [60].

We used NGINX [61], a well-known web server and reverse-proxy, as the target application; and Metricbeat [33] as probing system. Metricbeat helps in monitoring servers by collecting metrics from both the system and the services running on them. It can ship the collected metrics to Elasticsearch [62] and can be configured to collect tailored metrics. We configured the Metricbeat NGINX module to probe the target, while we activated the System and Kubernetes modules to measure the resource consumption in the case of VMs and containers, respectively.

To collect CPU, memory and network metrics on virtual machines, we run a dedicated Metricbeat instance on both the targets and holders. In Kubernetes, we deployed Metricbeat to measure the targets and holders resource consumption as a Kubernetes DaemonSet [63].

We used Metricbeat also to implement the monitoring probes that are part of the monitoring patterns, either deployed within virtual machines or deployed as single-replica Kubernetes Deployments.

We compute the CPU and memory consumption of a pattern as the sum of the resource consumption of each holder activated by the pattern. The consumption of a holder is obtained as its medium resource consumption along the experiment. The CPU and memory consumption of targets is computed as the mean value of the collected samples. For network I/O consumption, since it is a cumulative metric, we simply compute the total consumption of each element per experiment as the difference between the first and last data point.

To compute the actual cost of running probes, we referred to the monthly cost of a Microsoft Azure Standard B1s VM operated in the West Europe zone (€8.18/month at the time of writing), and to an Azure Container Instance operated in the West Europe zone (€31.7762/month x 1vCPU + €3.4845/month x 1 GB of RAM at the time of writing). We calculate the cost range of system-oriented patterns by multiplying the monthly expense of one VM by the number of holders generated by the pattern. Meanwhile, the cost range of application-oriented patterns is determined by multiplying the average CPU/RAM consumption values collected during the experiments with the monthly CPU/RAM costs. As for system-oriented patterns, also for application-oriented patterns the expense of a single container instance is then multiplied by the number of holders generated by the pattern.

The experimental material containing all the software artifacts (i.e., Ansible playbooks, execution scripts, configurations, data analysis) and the collected dataset is publicly available at [26].

5.4 RQ-SO - How do patterns for system-oriented clouds scale with the amount of monitored data?

Fig. 4 shows how the resource consumption growths for the various metrics, considering the system-oriented patterns implemented with VM holders. We do not include the Internal-T1P$*$ pattern in the plots related to memory consumption because no holder is added to the system (the holder matches with the target holder). Thus the overhead is limited to the resource consumption of the probe, which is negligible compared to the resources already consumed by the target holder. We report a selection of plots that is sufficient to illustrate the results and the trends. The complete set of plots with resources consumed by the holders and the targets for all the metrics and experiments is available as an online appendix [64].

CPU and Memory Consumption CPU and memory consumption are both negligible for *targets*. In particular, it is less than 1% for CPU consumption and less than 502 MiB for memory, independently of the dimension that is increasing. This is a clear evidence that all the system-oriented patterns are non-intrusive in terms of CPU and memory consumption for the target, including the Internal-T1P$*$ pattern (which may interfere in other ways due to the holder matching with the target holder).

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2024.3349648

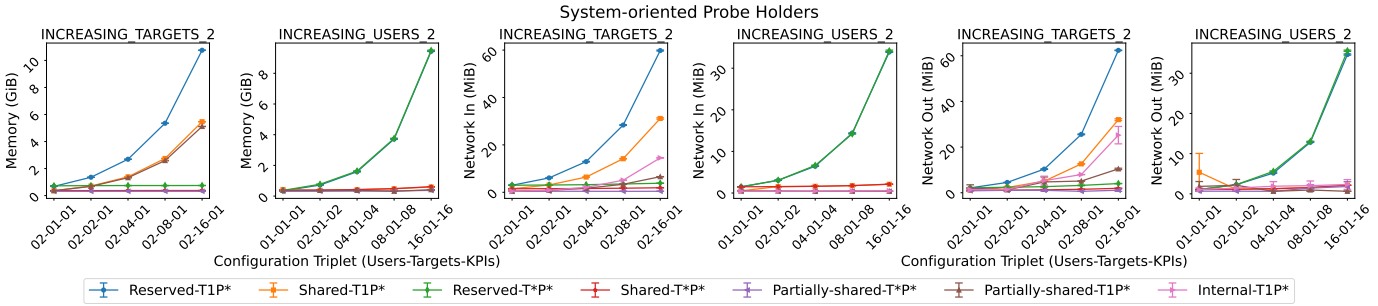


Fig. 4: System-oriented probe holders patterns scalability

CPU consumption is also negligible in the holders (less than 1%). In fact, probes are lightweight artifacts that consume little resources and even when their number increases, their impact on CPU is negligible. On the contrary, memory consumption is non-trivial in holders, for some patterns (up to 10 GiB). In fact, an increasing number of targets makes single-target ($T1$) holders used by Reserved-T1P$*$, Shared-T1P$*$, Partially-shared-T1P$*$ patterns subject to an exponential increase of memory consumption, as shown in Fig. 4 (INCREASING_TARGETS_2). On the other hand, an increasing number of users makes reserved holders used by Reserved-T1P$*$, Reserved-T$*$P$*$ patterns subject to an exponential memory consumption as shown in Fig. 4 (INCREASING_USERS_2). This trend can be expected, since all the five probes deployment patterns create new holders for an increasing number of users or targets, resulting in new VMs creation, that is, new allocated resources.

Network I/O Consumption Network I/O consumption is negligible on targets: up to 6 MiB transferred in 10 minutes for reserved patterns in the most expensive experiment (INCREASING_USERS_2). The transferred data are due to probes extracting data from the target. The limited traffic generated confirms the suitability of all patterns in terms of their interference on the target.

With respect to holders, network I/O consumption can be more significant. We observe in particular that both an increasing number of users requesting different KPIs (INCREASING_USERS_1 and INCREASING_USERS_2 experiments) and an increasing number of targets (INCREASING_TARGETS_1 and INCREASING_TARGET_2 experiments) resulted in an exponential network consumption trend, as shown in Fig. 4. In particular, single-target ($T1$) holders (Shared-T1P$*$ and Reserved-T1P$*$ patterns) and the Internal-T1P$*$ pattern are sensitive to an increasing number of targets, while reserved holders (Reserved-T1P$*$ and Reserved-T$*$P$*$ patterns) are sensitive to an increasing number of users requesting different KPIs.

Based on this evidence, depending on the expected scalability trend, we have patterns that should be preferred or avoided. It is however useful to remark that the overall resource network consumption that we observed has been limited, even for the most expensive scenarios (up to 60 MiB transferred in 10 minutes). This order of magnitude is likely relatively significant in a cloud environment, where network resources are usually abundant, while it is indeed relevant in resource-constrained environments, such as fog and edge environments.

Monthly Operating Costs We report in Table 4 how these differences may reflect in the monthly operation cost. All costs are in euros (€) and each cost interval is obtained by considering the minimum and the maximum number of employed holders for a specific scalability experiment.

Cost figures directly depend on the number of holders created, and are generally low as long as holders are not dedicated to individual service instances, which is a case that immediately generates unreasonable operation costs. Many scalability dimensions do not impact on the cost because VMs are quite large holders that can easily run several probes and their cost is not affected by the number of running probes, until the number is so large that multiple VMs have to be created. For this reason, it is difficult to estimate the cost of the Internal-T1P$*$ pattern, since probes run within the VM that hosts the target service and they do not induce a measurable costs as long as a larger VM has to be created due to the presence of the probes.

Answer to RQ-SO None of the patterns impacts on targets, thus their selection should be entirely *based on the resource consumption of holders*.

Holders are *not CPU eager*, so the choice of the pattern can focus on *memory* and *network consumption for environments where network consumption should be carefully controlled*, otherwise network consumption can be overlooked due to limited absolute consumption.

The expected resource consumption should be considered in relation to the expected growing rate of the key dimensions. If the monitoring system is employed in a multi-tenant environment where the number of users requiring different KPIs can easily increase, the patterns with reserved holders are particularly impacted (Reserved-T1P$*$ and Reserved-T$*$P$*$ deployment patterns). This may suggest that reusing probes and holders among users is advised when direct access to the target is not possible. The use of multi-target ($T*$) probe deployment patterns (i.e., Reserved-T$*$P$*$, Shared-T$*$P$*$, Partially-shared-T$*$P$*$) is advised when many different targets or instances must be monitored. Overall, since the overhead is mostly due to the holders, reducing their number increase the efficiency, making Shared-T$*$P$*$, Partially-shared-T$*$P$*$, and Internal-T1P$*$ the more scalable patterns for applications based on system-oriented virtualization, with Shared-T1P$*$ highly recommended in situations where the number of targets remains low, while the number of interested users increases.

TABLE 4: System-oriented Patterns Probe Holder Monthly Costs

Pattern	Experiment					
	INCREASING_KPIS_1	INCREASING_KPIS_2	INCREASING_TARGETS_1	INCREASING_TARGETS_2	INCREASING_USERS_1	INCREASING_USERS_2
*Internal-T1P**	≈ 0*	≈ 0*	≈ 0*	≈ 0*	≈ 0*	≈ 0*
*Reserved-T*P**	[8.18, 8.18]	[16.36, 16.36]	[8.18, 8.18]	[16.36, 16.36]	[8.18, 130.88]	[8.18, 130.88]
*Reserved-T1P**	[8.18, 8.18]	[16.36, 16.36]	[8.18, 130.88]	[16.36, 271.76]	[8.18, 130.88]	[8.18, 130.88]
*Partially-shared-T*P**	[8.18, 8.18]	[8.18, 8.18]	[8.18, 8.18]	[8.18, 8.18]	[8.18, 8.18]	[8.18, 8.18]
*Partially-shared-T1P**	[8.18, 8.18]	[8.18, 8.18]	[8.18, 130.88]	[8.18, 130.88]	[8.18, 8.18]	[8.18, 8.18]
*Shared-T*P**	[8.18, 8.18]	[8.18, 8.18]	[8.18, 8.18]	[8.18, 8.18]	[8.18, 8.18]	[8.18, 8.18]
*Shared-T1P**	[8.18, 8.18]	[8.18, 8.18]	[8.18, 130.88]	[8.18, 130.88]	[8.18, 8.18]	[8.18, 8.18]

5.5 RQ-AO - How do patterns for application-oriented clouds scale with the amount of monitored data?

Fig. 5 shows how the resource consumption growths for the various metrics, depending on the application-oriented patterns. We report a selection of plots that is sufficient to illustrate the results and the trends. The complete set of plots with resources consumed by the holders and the targets for all the metrics and experiments is available as an online appendix [64].

CPU and Memory Consumption Similarly to system-oriented patterns, also application-oriented patterns do not impact on the target. In fact, CPU and memory consumption of the target is below 0.01% and 5 MiB, respectively. Again, it confirms the suitability of the monitoring patterns to collect data from targets without interfering with their resource consumption.

CPU consumption is also negligible in holders despite patterns and growing trends (below 0.01%), while memory consumption can be significant. In fact, increasing the number of users who request for different KPIs (Fig. 5 INCREMENTING_USERS_2 experiment) results in an exponential memory consumption trend (up to more than 10 GiB for reserved holders (Reserved-T∗P1 and Reserved-T1P1 patterns). Note that these two reserved probe deployment patterns create a holder hosting one probe only for each of the users requesting a new KPI to be collected. For instance, the last configuration triplet (16 Users - 1 Target - 16 KPIs) of the INCREMENTING_USERS_2 experiment creates 256 holders to satisfy the user needs. Thus, although memory consumption may growth exponentially, the overall consumption in relation to the number of created containers is still quite good.

Network I/O Consumption Network I/O consumption is also negligible for targets, less than 1 MiB in all the experiments except for the INCREASING_USERS_2 experiment where we observed up to 6 MiB of network I/O consumption for the patterns using reserved holders (Reserved-T1P1 and Reserved-T∗P1 patterns).

With respect to holders, network I/O consumption can be still considered negligible (up to 17 MiB), but we observed that both an increasing number of users requesting different KPIs (INCREASING_USERS_2 experiment) and an increasing number of targets (INCREASING_TARGETS_1 and INCREASING_TARGET_2 experiments) resulted in an exponential network consumption trend as shown in Fig. 5. In particular, single-target ($T1$) holders (Shared-T1P1 and Reserved-T1P1 patterns) are sensitive to targets increment, while reserved holders (Reserved-T1P1 and Reserved-T∗P1 patterns) are sensitive to an increasing number of users requesting different KPIs.

Monthly Operating Costs Table 5 summarizes the monthly cost of executing application-oriented holders in the experiments. We can notice how deploying probes within an application-based environment is cheaper than in a system-oriented environment, due to the nature of the environments and the billing strategies. The Internal-T1P∗ VM-based pattern is the only exception, but such a pattern introduces non-trivial security and reliability issues, as discussed later. Interestingly, costs based on containers is often negligible, reaching a cost that could be appreciated on a monthly basis only for the most demanding configurations.

Answer to RQ-AO Although on different scale values, experiments with container-based applications resulted in trends similar to the ones obtained for VM-based applications. In fact, resource consumption on targets is negligible and the holder consumption is significantly mainly in relation to memory consumption.

Similarly, increasing the number of KPIs and increasing the number of users are the least impactful drivers for container-based holders. However, their combination (i.e., the increment of users requesting different KPIs) particularly impact reserved holders employed by Reserved-T1P1 and Reserved-T∗P1 probe deployment patterns. This suggests that an optimization and reuse of probes and holders among users is advised for application-oriented patterns too. Single-target ($T1$) holders are mostly impacted by the increase of targets, thus, the use of multi-target ($T*$) probe deployment patterns (i.e., Reserved-T∗P1, Shared-T∗P1) is advised when many different targets or instances must be monitored. Overall, Shared-T∗P1 is the most scalable pattern in the context of container-based applications, with Shared-T1P1 as a solid alternative option when there are few targets to be monitored but a potentially high number of users, and Reserved-T∗P1 yet another option when several targets must be monitored for a few users only.

5.6 Threats to Validity

The threats to the validity of the presented results mainly concern the relationship between the setup of the experiment and the collected resource consumption values. In fact, the consumption is affected by both the available computational resources and the choice of the probe technology and configuration. However, while changing the available computational resources and the deployed probes are likely to affect absolute values, the trends and differences among the probe deployment patterns are clear, despite these factors.

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2024.3349648

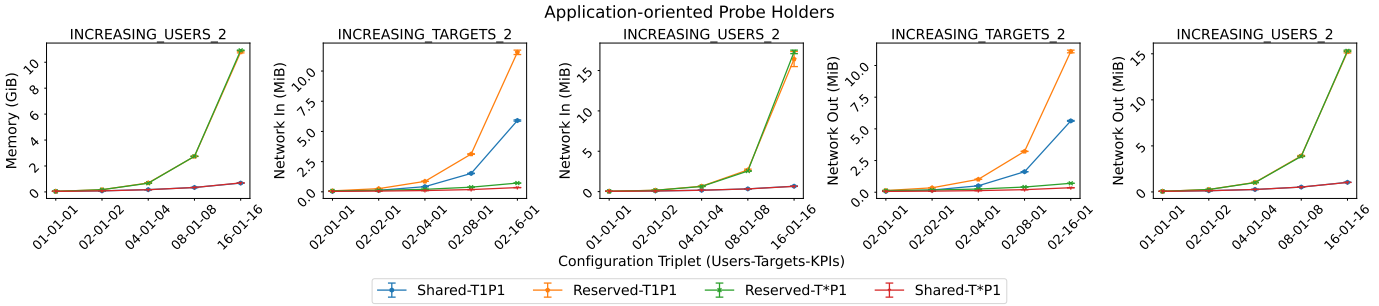


Fig. 5: Application-oriented probe holders patterns scalability.

TABLE 5: Application-oriented Patterns Probe Holder Monthly Costs

Pattern	Experiment					
	INCREASING_KPIS_1	INCREASING_KPIS_2	INCREASING_TARGETS_1	INCREASING_TARGETS_2	INCREASING_USERS_1	INCREASING_USERS_2
*Reserved-T*P1*	[0.1752, 2,8401]	[0.3519, 5.6497]	[0.1752, 0.1939]	[0.3519, 0.3910]	[0.1752, 2.7764]	[0.1752, 43.9768]
Reserved-T1P1	[0.1732, 2.8022]	[0.3397, 5.7135]	[0.1732, 3.1300]	[0.3397, 6.2549]	[0.1732, 2.8346]	[0.1732, 43.5852]
*Shared-T*P1*	[0.1755, 2.8495]	[0.1652, 2.7965]	[0.1755, 0.1980]	[0.1652, 0.1970]	[0.1750 0.1755]	[0.1755, 2.8446]
Shared-T1P1	[0.1814, 2.8513]	[0.1765, 2.8226]	[0.1814, 3.1658]	[0.1765, 3.1053]	[0.1712, 0.1814]	[0.1814, 2.8193]

In fact, plots for system-oriented and application-oriented probe deployment patterns are similar although specific values are different. Nevertheless, the relationship between increasing specific variables (e.g., the number of targets or the number of users) and the pattern characteristics (e.g., single-target or reserved patterns) are clearly identified by the resulting consumption trends.

In our evaluation, we also selected a specific target service to be monitored (i.e., NGINX) and we used a specific probe technology (i.e., Metricbeat module for NGINX). Moreover, we deployed the same probe when experiments required to increment the number of requested KPIs, and a probe is configured to collect a single KPI. In a real-world scenario probes may be configured to collect several KPIs, potentially lowering the resource consumption.While using a single target application (i.e., NGINX) in our evaluation may raise concerns about the generalization of the results, it is important to remark that the monitored application was not a factor in our study. The monitored application has no impact on the cost and effectiveness of the deployment patterns. We thus intentionally used a single application in our quantitative study to ensure that the evaluation is conducted under controlled and similar conditions, minimizing the possibility to introduce any confounding factor that could affect the results. To mitigate this issue we report results about our experience with three real-world applications of the patterns in Section 7.

Finally, the collected resource consumption values might be affected by noise. To mitigate this issue we repeated the experiments for 3 times for a total of 30 minutes of execution collecting 180 samples for each resource-related metric in any of the experiment configurations. We computed the mean and the standard deviation by all the data samples, thus, stabilizing the results to derive valid conclusions.

6 Best Practices for Probe Deployment

This section discusses a distilled list of best practices for probe deployment derived from our empirical findings. Engineers can exploit them when designing and configuring their monitoring systems, depending on the target environment and desired qualities.

BP-1: Share probe instances and holders for non-accessible targets in multi-user environments. Results show that resource consumption might grow quite quickly when the number of users and the number of monitored KPIs increase (e.g., see experiments INCREASING_TARGETS_2 and INCREASING_USERS_2). Indeed, the case of a large number of users asking for many KPIs in multi-user environments must be handled carefully, regardless of the underlying technology (e.g., system-oriented or application-oriented). This issue is exacerbated by non-accessible targets (e.g., third-party applications and inaccessible services for security concerns) that require the deployment of probes that sample the target from the outside. In such cases, the monitoring system should be configured to share as many resources as possible. This implies sharing the deployed probes, and possibly also the holders (see patterns *Shared-T∗P∗*, *Shared-T1P∗*, *Shared-T∗P1* and *Shared-T1P1*). Sometime, when probes cannot be shared, the patterns with partially-shared holders (see *Partially-shared-T∗P∗*, *Partially-shared-T1P∗* patterns) offer a valuable trade-off. When possible, probe instances must be configured to collect multiple KPIs to lower the consumption (see trend results for the INCREASING_USERS_2 experiment in Fig. 4).

BP-2: Use multi-target probe deployment in large-scale monitoring environments. Single-target patterns show that probes may consume significant amount of resources with an increasing size of the monitoring system (see for instance single-target patterns trends for INCREASING_TARGETS_2 experiment in Fig.s 4 and 5). For this reason, large-scale deployments with tens or more targets must adopt multi-target probe deployments. This is strongly advised for system-oriented environments where resource allocation for reserved holders can be resource eager (e.g., VMs), and thus also expensive. Suitable patterns for this case are: *Shared-T∗P∗*, *Shared-T∗P1*, *Partially-shared-T∗P∗*, *Reserved-*

T∗P∗ and *Reserved-T∗P1*. Single-target application-oriented holders (*Reserved-T1P1*, *Shared-T1P1* patterns) can sometime still be used thanks to the lightness of application-oriented containers.

BP-3: Privilege application-oriented holders to address high-dynamic KPI collection requirements. In the case of KPIs requirements that change often (e.g., many users with different business goals), application-oriented holders can be life-savers. Their advantage is twofold: first, their bootstrapping phase is way faster than VMs, and thus frequent creation and destruction of holders can be accomplished efficiently; second, even when probe instances (holders) cannot be shared to guarantee high configurability and isolation to multiple tenants, their allocation is still affordable in terms of resources and cost, when compared to system-oriented holders implemented with VMs (see INCREASING_USERS_2 experiment results for system-oriented holders shown in Fig. 4 and in the online appendix [64] for further details). Again, probe instances should be configured to collect multiple KPIs at once to save resources.

BP-4: Prefer container-based holders for isolation requirements. Dealing with third-party applications or strict security requirements may require satisfy isolation despite efficiency, and to deploy probe instances in dedicated holders. System-oriented holders can be resource-greedy and expensive when implemented with VMs especially. In fact, reserved patterns implemented with VM-based holders scale significantly worse for an increasing number of targets, as results for INCREASING_TARGETS experiments demonstrated. Thus container-based holders should be preferred when possible (Reserved-T∗P1, Reserved-T1P1 patterns). In the cases where VMs must be employed (e.g., due to constraints on the technology stack), the best practice is to use partially-shared holders (Partially-shared-T∗P∗, Partially-shared-T1P∗ patterns) and implement isolation at probe instance level.

BP-5: When the target is accessible and resource consumption is a concern, probes should be deployed within the same execution unit of the target. An accessible target offers the opportunity of collecting indicators efficiently, since there is not the burden of querying any monitoring interface and sharing the holder with the target increases observability. The low resource consumption has been confirmed with our experiments (see Fig. 4 and the online appendix [64] for further details). The same cannot be usually achieved with application-oriented execution environments (e.g., due to the single main container process practice [65]). Due to the side-effects that probes may introduce on targets, this choice is advice when resource consumption is a primary concern, compared to system reliability. Some specific technology stacks may offer interesting compromises. For example, engineers can exploit the concept of *pod* (i.e., Kubernetes Pod [66], Podman [67]) to obtain a setup similar to the Internal-T1P∗ pattern. In fact, thanks to pods, it is possible to execute multiple co-located containers that share storage and network resources, circumventing observability issues even though the execution unit is not the same.

7 Usage Scenarios

This section demonstrates the application of probe deployment patterns to three realistic usage scenarios that involve different technologies, software architectures, and monitoring requirements. In particular, we provide (i) a scenario for a VM-based microservice application, (ii) a scenario for a microservice application running on top of a Kubernetes cluster, and (iii) a scenario for serverless backend functions operated with the OpenFaaS platform.

We first describe the application architecture, the technology stack, and the monitoring requirements for each scenario. Second, we discuss how patterns are selected based on monitoring requirements and probe deployment best practices. We also describe how the selected patterns would be impacted by an increase in the number of the collected KPIs, the number of target instances, and the number of users interested in the collected data. Finally, we quantitatively evaluate the selected probe deployment patterns by collecting the CPU (%), memory (MiB), and network I/O (MiB) consumption for an increasing number of target instances, mimicking real-life situations that are faced in operation. We sample resource-related metrics every 10 seconds, repeating the experiment 3 times for 10 minutes to collect stable results, obtaining a total of about 180 samples.

The experimental material containing both the code to reproduce the experiment and the collected data is publicly available [26]. In the paper we report a selection of the plots, the complete set of plots is available in our online appendix [64].

7.1 Monitoring a VM-based Microservice Application

Scenario Description A company operates an e-commerce application composed of 11 microservices and a Redis database (e.g., Online Boutique[1]). For each service instance, the engineers spin up a VM following the Service-as-a-VM deployment pattern [68]. The payment, currency, and advertisement services are outsourced to an external provider that does not allow direct access to the service platform. Moreover, the company has a strong knowledge about Elastic Stack [24], since this monitoring service is used in several other company products.

The outsourced services expose KPIs using the Prometheus format (i.e., running the node exporter[2]), so the engineers need to collect these KPIs to obtain insights about the behavior of the outsourced service instances. In addition, they need to monitor the Redis database and some infrastructure KPIs (e.g., CPU and memory consumption, filesystem usage) for the VMs they are responsible for.

Applying the Probe Deployment Patterns This scenario can be effectively addressed with two patterns: the *Shared-T∗P∗* pattern and the *Internal-T1P∗* pattern. The *Shared-T∗P∗* pattern can be used to monitor inaccessible service instances, consistently with best practice BP-2. While the *Internal-T1P∗* pattern can be used to monitor the services running on their own VMs according to best practice BP-5. The probes can be implemented as Metricbeat [33] probe instances and can be configured to save data in the already

1. https://github.com/GoogleCloudPlatform/microservices-demo
2. https://github.com/prometheus/node_exporter

available Elasticsearch cluster, resulting in the following deployment:

- *Shared-T∗P∗* pattern: it consists of a VM hosting a Metricbeat probe instance configured with the Prometheus module to collect the KPIs exposed by the node exporters of the three outsourced services.
- *Internal-T1P∗* pattern:
  - for each VM running application services, it consists of a Metricbeat instance configured with the system module to monitor CPU load, memory, and filesystem.
  - for each of the Redis database replicas, it consists of a Metricbeat instance configured with (i) the Redis module to collect tailored Redis KPIs; and (ii) the system module to monitor CPU load, memory, and filesystem.

Scaling Impact

- *Increasing KPIs*: dealing with an increasing number of KPIs requires the reconfiguration of the Metricbeat probe instances, activating new modules, or deploying new probes in the case the KPIs to collect are not provided by any of the already deployed modules. Considering that both the patterns can hold multiple probes, no new holders have to be created to accommodate additional probe instances.
- *Increasing Targets*: increasing the number of targets requires to: (i) reconfigure the Metricbeat probe in the *Shared-T∗P∗* holder in the case new instances of the outsourced services are deployed; (ii) run a Metricbeat instance within any new VM they spin up to scale the internal services or the Redis database.
- *Increasing Users*: increasing the number of users accessing the monitoring system and interested in collected data do not require any new holders or probe instances because both the selected patterns allow sharing of resources among users.

Quantitative Evaluation for Increasing Target Instances

We exploit our implementation of this scenario to collect resource-related metrics for an increasing number of target instances, measuring the overhead introduced by the two implemented patterns. We increment both the number of *payment* and *recommendation* service replicas up to 16 to observe the impact on the Shared-T∗P∗ and Internal-T1P∗ patterns. All the other services are deployed with a single replica. Please note that in the case of the holder implementing the Shared-T∗P∗ pattern, it is simultaneously collecting KPIs from a single replica of the *currency*, a single replica of the *advertisement* service, and all the *payment* service replicas deployed during the experiment.

The collected data for the holder implementing the Shared-T∗P∗ pattern revealed CPU consumption is negligible (less than 1%), thus an increasing number of targets does not impact on CPU. Memory consumption was below 491.5 MiB in all the runs, and it is also not impacted by an increasing number of targets. Not surprisingly network I/O consumption is affected by an increasing number of targets (up to 13.9/87.0 MiB) due to the network traffic caused by the probes both scraping the KPI values from the targets, and then pushing them to the Elasticsearch instance for storage. Fig. 6a and Fig. 6b show the linear increment trend for an increasing number of *payment* service replicas.

When the number of *recommendation* service replicas is increased, no holder is added to the system since the holder matches with the target holder for the Internal-T1P∗ pattern. Thus the overhead in terms of CPU and memory consumption is limited to the resource consumption of the probe, which is negligible compared to the resources already consumed by the target. Network output consumption is instead affected by an increasing number of target instances due to the cumulative amount of data transferred by the probes contained in the target holders to the Elasticsearch instance (up to 18.2 MiB). Fig. 6c shows the linear increment trend for an increasing number of *recommendation* service replicas.

The trends observed in this scenario are indeed consistent with those obtained by the controlled evaluation reported in Section 5 for both the implemented patterns.

7.2 Monitoring a microservice application running on Kubernetes

Scenario Description A company operates the same application described in the previous usage scenario on top of a Kubernetes cluster. This time the company fully developed the application in-house. The company has a dedicated team for managing database infrastructure and several service development teams, with a strong knowledge about both Prometheus [25] and the application services.

In this case, the service development teams want to monitor HTTP and gRPC KPIs for their application services, and some specific KPIs for the Redis database. However, the requirements for monitoring Redis are different between the database ops team and the service development teams (e.g., collected KPIs and frequency).

Applying Probe Deployment Patterns This scenario can be well addressed with the *Shared-T∗P1* pattern, to monitor the application services and gather KPIs from multiple instances according to best practice BP-2, and the *Reserved-T∗P1* pattern, to monitor the Redis database replicas using a different holder to meet the conflicting requirements of the teams according to best practice BP-4. The monitoring solution exploits an already available Prometheus cluster as data storage, and Prometheus exporters [39] as probe instance technology, resulting in the following deployment:

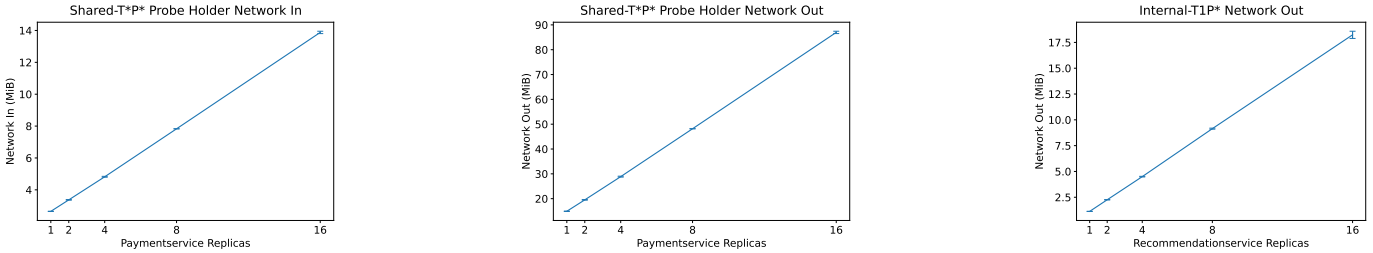
- *Shared-T∗P1* pattern: a Kubernetes Pod hosting a Prometheus Blackbox exporter instance[3] to collect HTTP and gRPC KPIs from the application services.
- *Reserved-T∗P1* pattern: two Kubernetes Pods hosting the Prometheus Redis exporter instance[4] configured to collect Redis KPIs from all the available replicas for the database ops team and the service development teams, respectively.

Scaling Impact

- *Increasing KPIs*: increasing the number of KPIs requires the engineers to reconfigure the probe instances activating new modules (e.g., TCP-level module for the Blackbox exporter), or deploying new holders hosting the probe instances for KPIs that are not already collected by any of the deployed probes.
- *Increasing Targets*: No actions are required for new service instances since both the exporters can be configured to

3. https://github.com/prometheus/blackbox_exporter
4. https://github.com/oliver006/redis_exporter

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2024.3349648



(a) (b) (c)

Fig. 6: Shared-T∗P∗ pattern holder network I/O consumption and Internal-T1P∗ pattern network output consumption with respect to an increasing number of payment service and recommendation service replicas, respectively.

collect KPIs from annotated targets (i.e., through Kubernetes annotations and Prometheus service discovery configuration).
- *Increasing Users*: increasing the number of users accessing the monitoring system and interested in the collected data may require creating new holders and instances, as for the database ops and the service development teams, because the *Reserved-T∗P1* pattern privileges isolation.

Quantitative Evaluation for Increasing Target Instances

We incremented the number of *cart* service replicas and Redis replicas up to 16 to observe the impact on the Shared-T∗P1 and Reserved-T∗P1 patterns, respectively. All the other services are deployed with a single replica.

We observed a negligible increase (less than 1%) in CPU consumption. Memory consumption does not exceed 340 MiB for any of the two patterns, and it is not impacted by an increased number of targets. Network input consumption is negligible for the Shared-T∗P1 pattern holder (i.e., less than 1 MiB), while on average network output consumption is slightly higher in terms of absolute values, reaching up to 2.18 MiB. Results are different in terms of absolute values for Reserved-T∗P1 pattern. In particular the network input consumption is higher compared to the output (i.e., up to 9.3/3.4 MiB), a scenario explained by the probe specific implementation. In fact, the Redis exporter has to query the Redis database instances to obtain the KPI values, and than it simply exposes the values as a web endpoint to Prometheus. However, both the patterns scales linearly with an increasing number of targets as shown in Fig. 7.

Also in this usage scenario, we observe trends consistent with the ones obtained in our controlled evaluation reported in Section 5.

7.3 Monitoring serverless backend functions

Scenario Description A company serves a serverless-based socks e-commerce application composed of 12 functions, 6 databases, and a message queue (e.g., SockShop Serverless[5]) exploiting OpenFaaS[6] and Kubernetes. The engineers adopt the FaaS model to exploit auto-scaling policies and obtain a flexible number of function replicas in response to the volatile workload that can affect their application (e.g., peaks of purchases during Black Friday, intense browsing and cart usage before Christmas, low demand in summer). They are particularly interested in monitoring the backend functions in terms of CPU and RAM usage in order to tweak auto-scaling policies and the cluster nodes size. Moreover, the company has a strong knowledge on using Prometheus to monitor the Kubernetes cluster nodes and the application services.

Applying Probe Deployment Patterns We can address this scenario by implementing the *Shared-T∗P1* pattern, to monitor multiple targets (i.e., functions) together, enabling less effort and resource usage in response to an increasing number of function replicas according to best practice BP-2.

The monitoring solution exploits the Prometheus cluster provided by OpenFaaS as data storage, and cAdvisor[7] as probe instance technology. The resulting deployment consists of a Kubernetes DaemonSet (i.e., a Kubernetes Pod for each of the cluster nodes) hosting a cAdvisor instance to collect the needed function KPIs at container-level.

Scaling Impact

- *Increasing KPIs*: increasing the number of KPIs requires the reconfiguration of the cAdvisor probe instances activating new KPIs, or deploying new holders and instances in the case the KPIs to collect are not provided by cAdvisor.
- *Increasing Targets*: increasing the number of targets does not require any change since cAdvisor is able to automatically detect new targets (i.e., container functions running on the Kubernetes node).
- *Increasing Users*: increasing the number of users accessing the monitoring system and interested in the collected data does not require any new holders or probe instances since the selected pattern supports sharing of resources.

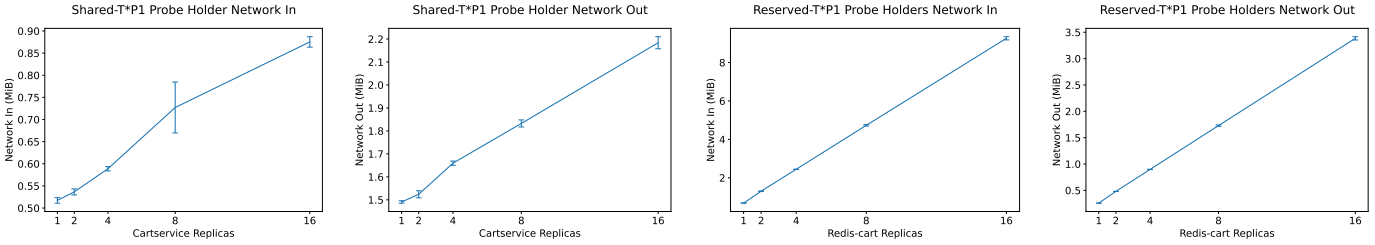
Quantitative Evaluation for Increasing Target Instances

We collect resource-related metrics for an increasing number of *carts-get* function instances (i.e., up to 16) to measure the overhead introduced by Shared-T∗P1 pattern. All the other functions are deployed with a single replica.

Collected data revealed CPU consumption is negligible (less than 1%). Memory consumption does not exceed 320 MiB, and it is not impacted by an increasing number of targets. Network input consumption is negligible (i.e., less than 0.2 MiB), as shown in Fig. 8a, while on average network output consumption reaches 3.1 MiB. Network output scales linearly with an increasing number of function replicas, as shown in Fig. 8b.

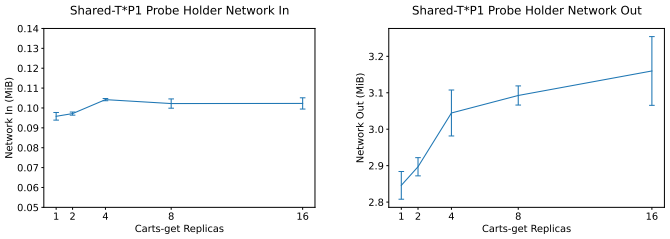
5. https://github.com/deib-polimi/serverless-sock-shop
6. https://openfaas.com
7. https://github.com/google/cadvisor

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2024.3349648



(a) (b) (c) (d)

Fig. 7: Network I/O consumption of the Shared-T∗P1 and Reserved-T∗P1 pattern holders with respect to an increasing number of cart service and Redis replicas, respectively.



(a) (b)

Fig. 8: Network I/O consumption of the Shared-T∗P1 pattern holders with respect to an increasing number of carts-get function replicas.

As for the previous scenarios, the observed trends for the Shared-T∗P1 pattern are consistent with the results obtained with the controlled evaluation reported in Section 5.

8 Related work

There are two distinct areas of research that are related to our contribution: cloud patterns and cloud monitoring.

Cloud patterns. In software engineering, patterns are used to document knowledge about how to solve recurring problems [69]. With the rise of the cloud computing paradigm, the community has begun working on cloud computing patterns [70], [71], [72]. Although their development is still in the early stages, several online catalogs have been published, providing both specific [73], [74] and agnostic [70] solutions. Specific patterns refer to particular cloud providers, are customized for a target environment, and provide solutions optimized for it. In contrast, agnostic patterns are more generic solutions that are not tied to a particular technology, are flexible, and can be applied to different platforms. The patterns presented in this article are agnostic since they are not tied to a specific cloud technology and can be applied to any execution environment. Agnostic pattern definition is a valuable means of improving portability and interoperability between different cloud environments [56]. However, none of these work specifically address the issue of probe deployment. Burns and Oppenheimer [29] propose design patterns for distributed systems based on containers. Despite their work does not address the issue of probe deployment, some of their patters resemble our Shared-T1P1 or Reserved-T1P1 patterns. Albuquerque et al. [75] present proactive monitoring design patterns for cloud-native applications, basing their definitions on existing literature and tools. In particular, they present three patterns that can generate events according to the event-based monitoring paradigm. Compared to our work, there are three main differences: (i) we focus on the placement of monitoring probes and the possibility to share monitoring resources among users; (ii) we extract features and constraints from existing literature and bad/best practices, to then define more generic and agnostic patterns; and (iii) we both qualitatively and empirically assess scalability of the proposed patterns, and further showcase their implementation in diverse usage scenarios involving different monitoring technologies.

Cloud Monitoring. We can broadly classify the methods for cloud monitoring into two main classes: active monitoring [76], in which measurements are based on probes injected into the system, and passive monitoring, in which measurements are not based on probing, but rather on the passive observation and analysis of existing resources, such as network flows [77] or logs [78]. The advantage of passive monitoring is that it does not add any communication overhead. It makes sense for some components where important indicators can be observed, but is less useful for others where it is not sufficient to observe existing flows to capture the state of the monitored component. In our work, we focus on the case of active monitoring, which requires probe deployment.

We can further distinguish two sub-classes of approaches in active monitoring: agent-based solutions [76] and agent-less solutions [79]. Agent-based solutions install software agents inside the monitored components. These agents calculate measurements on the component and then send the data to an external collector. In contrast, agent-less solutions rely on external components to retrieve monitoring data from interfaces exposed by the monitored components, without adding any software to the latter. Although the agent-less solution has low maintenance costs and less risk of interference, agent-based monitoring systems provide deeper and more specialized measurements than the protocols used by agent-less protocols, such as SNMP [41]. In this paper we presented monitoring patterns for both agent-based (i.e., Internal-T1P∗) and agent-less solutions (i.e., all the other ten patterns presented in the paper).

Recently, fog and edge computing paradigms have emerged to create a continuum of cloud services that extend from centralized data centers to end devices. In his review, Verginadis [80] compares twenty prominent moni-

toring technologies for the cloud continuum, encompassing both active and passive solutions across various dimensions, such as monitoring level, output types, supported metrics, and more. The analysis indicates that Netdata [81] is a suitable solution for fog-edge environments, which often involve resource-constrained devices. Netdata offers internal and external plugins for gathering KPIs, and these plugins can be deployed according to the presented probe deployment patterns (e.g., Internal-T1P$*$ or Shared-T$*$P$*$). Other works focusing on fog monitoring [82], [83], [84], [85] rely on monitoring agents for collecting KPIs. FMone [83] employs agents executed in separate Docker containers similarly to the Shared-T$*$P1 and Shared-T1P1 patterns. Souza et al.'s approach [84] involves the utilization of both internal and external agents, which can be deployed using the Internal-T1P$*$ pattern or external unit patterns, such as Shared-T$*$P1 and Partially-shared-T$*$P$*$, depending on the execution environment or sharing policies. Additionally, both FogMon [82] and its self-adaptive extension [85] employ internal monitoring agents to gather multiple KPIs, following the Internal-T1P$*$ pattern.

9 Conclusion

The flexibility of monitoring frameworks and probe technologies for the Cloud allows for diverse probe deployment strategies, which may have implications on the effectiveness and efficiency of the resulting monitoring system. For instance, multiple probes serving different operators in a multi-tenant environment can be deployed within a same virtual machine to save resources, at the expense of a reduced degree of privacy and security. On the other hand, one probe per container or virtual machine can be deployed to preserve privacy, at the expense of more resources allocated to the monitoring system.

This paper systematically derives, presents, and analyzes possible probe deployment strategies, resulting in 11 probe deployment patterns that are described and assessed empirically. The results of this work may help engineers in designing their monitoring systems, and generate a set of reusable solutions that people can refer to. We also released publicly all the experimental material containing the software artifacts and the collected dataset at [26].

Future work mainly concerns with exploiting and empirically assessing the presented patterns in the design of novel monitoring systems for a range of environments, including fog and edge systems. An additional area of research is the identification of the conditions that may necessitate shifting from one pattern to another, in order to develop intelligent, self-adaptive monitoring solutions, for instance using rule-based expert systems and machine learning techniques.

Acknowledgments

This work has been partially funded by the Centro Nazionale HPC, Big Data e Quantum Computing (PNRR CN1 spoke 9 Digital Society & Smart Cities).

References

[1] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[2] M.-H. Kuo, "Opportunities and challenges of cloud computing to improve health care services," *Journal of Medical Internet Research*, vol. 13, no. 3, p. e1867, 2011.

[3] Z. Benomar, F. Longo, G. Merlino, and A. Puliafito, "Cloud-based network virtualization in iot with openstack," *ACM Transactions on Internet Technology*, vol. 22, no. 1, pp. 1–26, sep 2021.

[4] D. Breitgand, V. Eisenberg, N. Naaman, N. Rozenbaum, and A. Weit, "Toward true cloud native nfv mano," in *Proceedings of the 12th International Conference on Network of the Future (NoF)*, 2021, pp. 1–5.

[5] J. D. Pereira, R. Silva, N-Antunes, J. L. M. Silva, B. de França, R. Moraes, and M. Vieira, "A platform to enable self-adaptive cloud applications using trustworthiness properties," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Association for Computing Machinery, 2020, pp. 71–77.

[6] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. K. Ports, and A. Panda, "Multitenancy for fast and programmable networks in the cloud," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[7] A. Caulfield, P. Costa, and M. Ghobadi, "Beyond smartnics: Towards a fully programmable cloud," in *International Conference on High Performance Switching and Routing (HPSR)*, 2018, pp. 1–6.

[8] M. Filho, E. Pimentel, W. Pereira, P. H. M. Maia, and M. I. Cortes, "Self-adaptive microservice-based systems - landscape and research opportunities," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2021, pp. 167–178.

[9] Y. Zhou and D. Wentzlaff, "The sharing architecture: Sub-core configurability for iaas clouds," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 559–574.

[10] J. T. A. Ali-Eldin and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *IEEE Network Operations and Management Symposium*, 2012, pp. 204–212.

[11] 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.

[12] K. Alhamazani, R. Ranjan, K. Mitra, F. Rabhi, P. P. Jayaraman, S. U. Khan, A. Guabtni, and V. Bhatnagar, "An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art," *Computing*, vol. 97, no. 4, pp. 357–377, Apr. 2015.

[13] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.

[14] A. Brogi, J. Carrasco, F. Duran, E. Pimentel, and J. Soldani, "Self-healing trans-cloud applications," *Computing*, pp. 1–25, 2021.

[15] O. Riganelli, P. Saltarel, A. Tundo, M. Mobilio, and L. Mariani, "Cloud failure prediction with hierarchical temporary memory: An empirical assessment," in *Proceeding of the IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2021, pp. 785–790.

[16] L. Mariani, M. Pezzé, O. Riganelli, and R. Xin, "Predicting failures in multi-tier distributed systems," *Journal of Systems and Software*, vol. 161, p. 110464, 2020.

[17] C. Sauvanaud, K. Lazri, M. Kaaniche, and K. Kanoun, "Anomaly detection and root cause localization in virtual network functions," in *Proceeding of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 196–206.

[18] K. Alhamazani, R. Ranjan, P. P. 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, Jan. 2019.

[19] 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.

[20] A. Shatnawi, M. Orrú, 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.

[21] S. Singh, I. Chana, and R. Buyya, "Star: Sla-aware autonomic management of cloud resources," *IEEE Transactions on Cloud Computing*, vol. 8, no. 4, pp. 1040–1053, 2020.

[22] S. Mubeen, S. A. Asadollah, A. V. Papadopoulos, M. Ashjaei, H. Pei-Breivold, and M. Behnam, "Management of service level agreements for cloud services in iot: A systematic mapping study," *IEEE access*, vol. 6, pp. 30 184–30 207, 2017.

[23] R. Maeser, "Analyzing csp trustworthiness and predicting cloud service performance," *IEEE Open Journal of the Computer Society*, vol. 1, pp. 73–85, 2020.

[24] Elasticsearch B.V., "Elastic stack," https://www.elastic.co/elastic-stack/, 2023, [Online; accessed 25-July-2023].

[25] Prometheus Authors. (2023) Prometheus. https://prometheus.io/. [Online; accessed 25-July-2023].

[26] A. Tundo, M. Mobilio, O. Riganelli, and L. Mariani. (2023) Monitoring Probe Deployment Patterns for Cloud-Native Applications: Definition and Empirical Assessment (Experimental Material). https://gitlab.com/learnERC/monitoring-patterns-experiments. [Online; accessed 25-July-2023].

[27] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University - Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[28] A. Mimidis-Kentis, J. Soler, P. Veitch, A. Broadbent, M. Mobilio, O. Riganelli, S. Van Rossem, W. Tavernier, and B. Sayadi, "The next generation platform as a service: Composition and deployment of platforms and services," *Future Internet*, vol. 11, no. 5, 2019.

[29] 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*, 2016.

[30] A. Tundo, M. Mobilio, O. Riganelli, and L. Mariani, "Automated probe life-cycle management for monitoring-as-a-service," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 969–982, 2023.

[31] M. Abderrahim, M. Ouzzif, K. Guillouard, J. François, A. Lebre, C. Prud'homme, and X. Lorca, "Efficient resource allocation for multi-tenant monitoring of edge infrastructures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2019, pp. 158–165.

[32] D. Tovarňák and T. Pitner, "Towards multi-tenant and interoperable monitoring of virtual machines in cloud," in *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012, pp. 436–442.

[33] Elasticsearch B.V., "Metricbeat: Lightweight Shipper for Metrics," https://www.elastic.co/beats/metricbeat, 2023, [Online; accessed 25-July-2023].

[34] A. Lingayat, R. R. Badre, and A. K. Gupta, "Performance evaluation for deploying docker containers on baremetal and virtual machine," in *2018 3rd International Conference on Communication and Electronics Systems (ICCES)*. IEEE, 2018, pp. 1019–1023.

[35] Z. Zhang, D. Li, and K. Wu, "Large-scale virtual machines provisioning in clouds: challenges and approaches," *Frontiers of Computer Science*, vol. 10, no. 1, pp. 2–18, 2016.

[36] S. A. De Chaves, R. B. Uriarte, and C. B. Westphall, "Toward an architecture for monitoring private clouds," *IEEE Communications Magazine*, vol. 49, no. 12, pp. 130–137, 2011.

[37] J. Povedano-Molina, J. M. Lopez-Vega, J. M. Lopez-Soler, A. Corradi, and L. Foschini, "Dargos: A highly adaptable and scalable monitoring architecture for multi-tenant clouds," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2041–2056, 2013.

[38] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "Featureide: An extensible framework for feature-oriented software development," *Science of Computer Programming*, vol. 79, pp. 70–85, 2014.

[39] Prometheus Authors, "Exporters and Integrations," https://prometheus.io/docs/instrumenting/exporters/, 2023, [Online; accessed 25-July-2023].

[40] Elasticsearch B.V., "Beats: Data Shippers for Elasticsearch," https://www.elastic.co/beats/, 2022, [Online; accessed 05-September-2022].

[41] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin, "Simple network management protocol (snmp)," Tech. Rep., 1989.

[42] Prometheus Authors, "SNMP exporter for Prometheus," https://github.com/prometheus/snmp_exporter, 2023, [Online; accessed 25-July-2023].

[43] Zabbix, "Zabbix," https://www.zabbix.com, 2023, [Online; accessed 25-July-2023].

[44] Nagios Enterprises, "Nagios," https://www.nagios.com, 2023, [Online; accessed 25-July-2023]].

[45] Dynatrace LLC, "Dynatrace," https://www.dynatrace.com, 2023, [Online; accessed 25-July-2023]].

[46] P. Hasselmeyer and N. d'Heureuse, "Towards holistic multi-tenant monitoring for virtual data centers," in *2010 IEEE/IFIP Network Operations and Management Symposium Workshops*, Apr. 2010, pp. 350–356.

[47] 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, pp. 2103–2114, 2013.

[48] J. M. A. Calero and J. G. Aguado, "Monpaas: An adaptive monitoring platformas a service for cloud computing infrastructures and services," *IEEE Transactions on Services Computing*, vol. 8, no. 1, pp. 65–78, 2014.

[49] C. B. Hauser and S. Wesner, "Reviewing cloud monitoring: Towards cloud resource profiling," in *International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 678–685.

[50] H. J. Syed, A. Gani, F. H. Nasaruddin, A. Naveed, A. I. A. Ahmed, and M. K. Khan, "Cloudprocmon: A non-intrusive cloud monitoring framework," *IEEE Access*, vol. 6, pp. 44 591–44 606, 2018.

[51] 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, 2012.

[52] 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.

[53] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[54] 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*, 2014, pp. 226–235.

[55] The OpenStack Project, "Monasca," https://docs.openstack.org/monasca-api/latest/, 2023, [Online; accessed 21-Nov-2023].

[56] B. Di Martino, G. Cretella, and A. Esposito, "Semantic and agnostic representation of cloud patterns for cloud interoperability and portability," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 2. IEEE, 2013, pp. 182–187.

[57] Red Hat, Inc., "How Ansible Works," https://www.ansible.com/overview/how-ansible-works, 2023, [Online; accessed 25-July-2023].

[58] Microsoft. (2023) Azure compute. https://azure.microsoft.com/en-gb/products/category/compute/. [Online; accessed 25-July-2023].

[59] ——. (2023) Azure kubernetes service (aks). https://azure.microsoft.com/en-gb/services/kubernetes-service/. [Online; accessed 25-July-2023].

[60] The Kubernetes Authors. (2023) Kubernetes deployment. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Online; accessed 25-July-2023].

[61] F5, Inc. (2023) Nginx. https://www.nginx.com/. [Online; accessed 25-July-2023].

[62] Elasticsearch B.V., "Elasticsearch: The Official Distributed Search & Analytics Engine," https://www.elastic.co/elasticsearch/, 2023, [Online; accessed 25-July-2023].

[63] The Kubernetes Authors. (2023) Kubernetes daemonset. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/. [Online; accessed 25-July-2023].

[64] A. Tundo, M. Mobilio, O. Riganelli, and L. Mariani, "Monitoring Probe Deployment Patterns for Cloud-Native Applications: Definition and Empirical Assessment (Online Appendix)," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7081889

[65] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[66] The Kubernetes Authors. (2023) Kubernetes pod. https://kubernetes.io/docs/concepts/workloads/pods/. [Online; accessed 25-July-2023].

[67] Containers Organization. (2023) What is podman? https://docs.podman.io/en/latest/. [Online; accessed 25-July-2023].

[68] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[69] D. C. Schmidt, M. Fayad, and R. E. Johnson, "Software patterns," *Communications of the ACM*, vol. 39, no. 10, pp. 37–39, 1996.

[70] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014.

[71] F. Khomh and S. A. Abtahizadeh, "Understanding the impact of cloud patterns on performance and energy consumption," *Journal of Systems and Software*, vol. 141, pp. 151–170, 2018.
[72] T. B. Sousa, H. S. Ferreira, and F. F. Correia, "A survey on the adoption of patterns for engineering software for the cloud," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2128–2140, 2022.
[73] Microsoft. (2023) Cloud design patterns. https://docs.microsoft.com/en-us/azure/architecture/patterns/. [Online; accessed 26-May-2023].
[74] Amazon Web Services. (2023) Aws prescriptive guidance patterns. https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/. [Online; accessed 26-May-2023].
[75] C. Albuquerque, K. Relvas, F. F. Correia, and K. Brown, "Proactive monitoring design patterns for cloud-native applications," in *Proceedings of the 27th European Conference on Pattern Languages of Programs*. Association for Computing Machinery, 2023, pp. 1–13.
[76] A. Meera and S. Swamynathan, "Agent based resource monitoring system in iaas cloud environment," *Procedia Technology*, pp. 200–207, 2013, 1st International Conference on Computational Intelligence: Modeling Techniques and Applications (CIMTA).
[77] P.-O. Brissaud, J. François, I. Chrisment, T. Cholez, and O. Bettan, "Passive monitoring of https service use," in *2018 14th International Conference on Network and Service Management (CNSM)*, 2018, pp. 219–225.
[78] B. Chen and Z. M. J. Jiang, "A survey of software log instrumentation," *ACM Comput. Surv.*, vol. 54, no. 4, 2021.
[79] M. Brattstrom and P. Morreale, "Scalable agentless cloud network monitoring," in *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*. IEEE, 2017, pp. 171–176.
[80] Y. Verginadis, "A review of monitoring probes for cloud computing continuum," in *International Conference on Advanced Information Networking and Applications*. Springer, 2023, pp. 631–643.
[81] N. Inc. (2023) Netdata. https://www.netdata.cloud/. [Online; accessed 23-Nov-2023].
[82] S. Forti, M. Gaglianese, and A. Brogi, "Lightweight self-organising distributed monitoring of fog infrastructures," *Future Generation Computer Systems*, vol. 114, pp. 605–618, 2021.
[83] Á. Brandón, M. S. Pérez, J. Montes, and A. Sanchez, "Fmone: A flexible monitoring solution at the edge," *Wireless Communications and Mobile Computing*, vol. 2018, pp. 1–15, 2018.
[84] A. Souza, N. Cacho, A. Noor, P. P. Jayaraman, A. Romanovsky, and R. Ranjan, "Osmotic monitoring of microservices between the edge and cloud," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2018, pp. 758–765.
[85] 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*, 2022, pp. 156–166.

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 an Assistant Professor 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 Associate 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.