SCUOLA DI DOTTORATO
UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

Department of Informatics, Systems, and Communication

PhD program Computer Science Cycle XXXVI

Adapt and Automate: Efficiently Monitoring in the Cloud Continuum

Alessandro Tundo

Registration number: *781257*

Tutor: *Prof. Claudio Zandron*

Supervisor: *Prof. Leonardo Mariani*

Coordinator: *Prof. Leonardo Mariani*

ACADEMIC YEAR 2022/2023

ABSTRACT

Monitoring systems are increasingly being deployed throughout the cloud continuum, a distributed and heterogeneous environment with varying software and hardware stacks designed to be simultaneously accessible in a multi-tenant fashion. Its fog and edge computing layers exhibit lower network latency and greater responsiveness compared to the upper cloud layer. However, they have lower reliability due to the prevalence of wireless connectivity, and fewer computational capabilities due to limited device resources.

Managing monitoring systems in the cloud continuum poses several challenges for automation and energy consumption. In this context, this thesis addresses two main challenges: (i) automating the monitoring system configurations in response to dynamic needs and technological constraints, and (ii) efficiently utilizing available resources, which is particularly relevant in the context of fog and edge environments. Regarding the first challenge, this thesis presents two main contributions: (i) a Monitoring-as-a-Service (MaaS) framework that can fully govern the life-cycle of the probes, including error-handling, and (ii) the definition, analysis, and qualitative and quantitative evaluation of 11 possible probe deployment patterns. Regarding the second challenge, this thesis presents two additional contributions. Firstly, it presents a self-adaptive peer-to-peer monitoring system for fog environments that can abstract monitored indicators to logical states and activate countermeasures in turn. Secondly, it proposes an energy-aware approach to guide developers in implementing self-adaptive applications for edge environments. Such applications are capable of switching operation modes in response to changes in the environment, ultimately balancing energy consumption with application-level objectives, such as monitoring accuracy.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my supervisor, Prof. Leonardo Mariani, for his invaluable patience and for the knowledge and expertise he generously shared with me. His passion and dedication to research have been extremely inspiring. I am sure that he deeply shaped my *forma mentis* and taught me how to navigate the incredible, yet difficult, academic world.

Special thanks to my colleagues and co-authors, Marco Mobilio and Prof. Oliviero Riganelli, who supported and guided me throughout my Master's and Ph.D. studies. I am also grateful to Vera Colombo, who has been extremely helpful in contributing to my research with her master's thesis. I would like to express my sincere thanks to Prof. Michele Ciavotta and Marco Savi for the collaboration during these years, and to Prof. Ezio Bartocci and the entire HPC research group led by Prof. Ivona Brandić, who have been immensely welcoming since the first moment I arrived in Vienna.

I would be remiss not to mention Matteo and Ilaria, who have been my peers and friends throughout these years: their support has never been just scientific. I want to thank my flatmates Luca and Alessandro, the experience of living as a family and navigating the pandemic together is unforgettable. I must thank Daniele and Clara too, the last years in Milan were marked by your presence, and Lynn for being part of the motivations that convinced me to stay in Vienna. I am grateful to my therapist, Patrizia, because although I have had an easy doctoral experience, I am sure it is also due to the healing we have achieved together.

And last but not least, a heartfelt thank you to my parents, my brothers and my whole family, "that dear octopus from whose tentacles we never quite escape, nor, in our deepest hearts, ever quite wish to", and to Kseniia, who jumped into my life and is deeply rooting in it with her sweet heart.

CONTENTS

INTRODUCTION 1

1 THE CLOUD CONTINUUM 5
- 1.1 Introduction to Cloud Continuum . . . 6
- 1.2 Cloud Computing . . . 7
- 1.3 Fog Computing . . . 10
- 1.4 Edge Computing . . . 11
- 1.5 Monitoring Challenges in the Cloud Continuum . . . 13

2 MONITORING IN THE CLOUD CONTINUUM 15
- 2.1 Anatomy of a Monitoring System . . . 15
- 2.2 Adapting Monitoring to Evolving Requirements . . . 17
  - 2.2.1 Support to Automated Evolution . . . 18
  - 2.2.2 Support to Multi-Tenancy and Heterogeneity . . . 20
- 2.3 Adapting Monitoring to Available Resources . . . 22
  - 2.3.1 Efficiently Use Resources in the Fog . . . 22
  - 2.3.2 Efficiently Use Resources in the Edge . . . 25

I ADAPTING MONITORING TO EVOLVING REQUIREMENTS 29

3 AUTOMATING PROBE LIFE-CYCLE FOR CHANGING NEEDS 31
- 3.1 Running Example . . . 31
- 3.2 Domain Concepts . . . 32
- 3.3 Solution Architecture . . . 34
  - 3.3.1 Repositories . . . 35
  - 3.3.2 API Service . . . 37
  - 3.3.3 Monitoring Claim Controller . . . 37
  - 3.3.4 Monitoring Unit Controller . . . 40
  - 3.3.5 Cloud Bridge . . . 41
- 3.4 Error Handling Capabilities . . . 41
- 3.5 Technology Agnostic Design . . . 44
  - 3.5.1 Incorporating New Probes . . . 45
  - 3.5.2 Supporting New Target Cloud Platforms . . . 46
- 3.6 Empirical Evaluation . . . 46
  - 3.6.1 Research Questions . . . 46
  - 3.6.2 Prototype . . . 47
  - 3.6.3 RQ1.1: Framework Efficiency . . . 48
  - 3.6.4 RQ1.2: Error Handling . . . 51
  - 3.6.5 RQ1.3: Scalability . . . 53
  - 3.6.6 Threats to Validity . . . 54
- 3.7 Discussion . . . 55

4 PATTERNS FOR PROBE DEPLOYMENTS 57
- 4.1 Probe Deployment . . . 57
- 4.2 Pattern Definitions . . . 58
- 4.3 Qualitative Discussion . . . 65
  - 4.3.1 Pattern Implementation . . . 66

4.3.2 Analysis of Quality Aspects . . . 67
4.4 Empirical Evaluation . . . 69
4.4.1 Research Questions . . . 69
4.4.2 Experimental Setup . . . 70
4.4.3 RQ2.1: System-oriented Pattern Scalability . . . 73
4.4.4 RQ2.2: Application-oriented Pattern Scalability . 76
4.4.5 Threats to Validity . . . 78
4.5 Best Practices . . . 79
4.6 Usage Scenarios . . . 81
4.6.1 Monitoring a VM-based Microservice Application 82
4.6.2 Monitoring a microservice application running on Kubernetes . . . 84
4.6.3 Monitoring serverless backend functions . . . 86
4.7 Discussion . . . 88

II ADAPTING MONITORING TO AVAILABLE RESOURCES 91

5 PEER-TO-PEER SELF-ADAPTIVE MONITORING IN THE FOG 93
5.1 P2P Monitoring . . . 93
5.2 AdaptiveMon . . . 95
5.2.1 Knowledge . . . 96
5.2.2 Monitor . . . 98
5.2.3 Analyze . . . 98
5.2.4 Plan . . . 99
5.2.5 Execute . . . 100
5.3 Empirical Evaluation . . . 100
5.3.1 Research Questions . . . 101
5.3.2 Prototype . . . 101
5.3.3 Experimental Setup . . . 102
5.3.4 RQ3.1 - Monitoring Accuracy . . . 102
5.3.5 RQ3.2 - Resource Consumption . . . 106
5.3.6 Threats to Validity . . . 110
5.4 Discussion . . . 110

6 ENERGY-AWARE SELF-ADAPTIVE MONITORING IN THE EDGE 113
6.1 Motivational Scenario . . . 113
6.2 Designing Energy-Aware Self-Adaptive Applications . . 114
6.2.1 Defining the State-Based Adaptation Logic . . . 116
6.2.2 Solving the Multi-Objective Optimization Problem 117
6.2.3 Extracting the Operation Mode Configurations . 119
6.2.4 Implementing the Self-Adaptive Application . . . 121
6.3 Empirical Evaluation . . . 122
6.3.1 Research Questions . . . 122
6.3.2 Experimental Setup . . . 123
6.3.3 RQ4.1 - Meta-Heuristic VS Near-Exhaustive Search 124
6.3.4 RQ4.2 - Objectives Trade-Off . . . 126
6.3.5 Threats to Validity . . . 129
6.4 Discussion . . . 130

7 CONCLUSIONS 133

A PROBE DEPLOYMENT PATTERN PLOTS 137
B GW INSTEK GPM-8213 POWER MEASUREMENT ACCURACY 141

BIBLIOGRAPHY 143

LIST OF FIGURES

Figure 0.1 A graphical overview of the thesis context, research challenges (RCs), research questions (RQs), and main contributions. 1
Figure 2.1 Generic architecture of a monitoring system. . . 16
Figure 2.2 Automation levels introduced in monitoring systems. 19
Figure 3.1 Architecture of the MaaS framework. 34
Figure 3.2 Probe deployment time figures. 49
Figure 3.3 Error handling time figures. 52
Figure 3.4 Time to fulfilling monitoring requests for a increasing number of indicators (dot markers) and operators (triangle markers), with both VMs running in Microsoft Azure (red lines) and on containers running in a local Kubernetes cluster (light blue lines). 54
Figure 4.1 Probe deployment patterns feature diagram. . . 59
Figure 4.2 Probe deployment patterns. 62
Figure 4.3 System-oriented probe holders patterns scalability. 73
Figure 4.4 Application-oriented probe holders patterns scalability. 76
Figure 4.5 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. 83
Figure 4.6 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. 86
Figure 4.7 Network I/O consumption of the Shared-T∗P1 pattern holders with respect to an increasing number of carts-get function replicas. 87
Figure 5.1 Hierarchical P2P monitoring architecture proposed by Forti *et al.* [91]. 94
Figure 5.2 Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) loop as proposed by Kephart and Chess [140]. 95
Figure 5.3 An example of the computed states with respect to the time series values at different time instants. 99

Figure 5.4 AdaptiveMon and StaticMon Follower time series estimations for the *stable-unstable* scenario. 104
Figure 5.5 AdaptiveMon and StaticMon Leader time series estimations for the *stable-unstable* scenario. . 105
Figure 5.6 AdaptiveMon Follower time series estimation for the *spiky* scenario. The vertical dotted grey lines indicate the *sampling rate*. 105
Figure 5.7 StaticMon compared with AdaptiveMon countermeasures for each of the collected quality metrics. 107
Figure 5.8 StaticMon compared with AdaptiveMon countermeasures for the network I/O metrics when the bandwidth is not measured by the Follower. 109
Figure 6.1 A pedestrian detection scenario. 114
Figure 6.2 The steps of the proposed approach represented as a workflow diagram. 116
Figure 6.3 An abstract state machine modeling the states and the transitions of a self-adaptive application for the motivational scenario. 117
Figure 6.4 A refined version of the abstract state machine shown in Figure 6.3 with the set of weights and thresholds for each of the operation modes. . . . 121
Figure 6.5 The concrete finite state machine implementing a self-adaptive application for the motivational scenario. 122
Figure 6.6 The test-bed used to run the evaluation experiments. 123
Figure 6.7 Radar charts comparing the objective values of the four self-adaptive operation modes when employing a solution obtained with the meta-heuristic search procedure and one obtained with the near-exhaustive search procedure. The solutions are extracted with the WGRA method using the same set of weights and thresholds. . 124
Figure 6.8 Radar charts comparing the SAA and the 4 non-adaptive applications in the weekdays and weekends scenarios. 128
Figure 6.9 Box-plots comparing energy consumption for the self-adaptive and the four non-adaptive applications. 129
Figure A.1 System-oriented CPU Consumption 137
Figure A.2 System-oriented Memory Consumption 137
Figure A.3 System-oriented Network Input Consumption . 138
Figure A.4 System-oriented Network Output Consumption 138
Figure A.5 Application-oriented CPU Consumption 138
Figure A.6 Application-oriented Memory Consumption . . 139

Figure A.7 Application-oriented Network Input Consumption . . . 139
Figure A.8 Application-oriented Network Output Consumption . . . 139
Figure A.9 Monitoring a VM-based Microservice Application Usage Scenario . . . 140
Figure A.10 Monitoring a Microservice Application Running on Kubernetes Usage Scenario . . . 140
Figure A.11 Monitoring Serverless Backend Functions Usage Scenario . . . 140

LIST OF TABLES

Table 4.1 Characterization of the Patterns . . . 67
Table 4.2 Experiments Configurations . . . 71
Table 4.3 System-oriented patterns probe holder monthly costs for experiments INCREASING_KPIS_1, INCREASING_KPIS_2, and INCREASING_TARGETS_1 75
Table 4.4 System-oriented patterns probe holder monthly costs for experiments INCREASING_TARGETS_2, INCREASING_USERS_1, and INCREASING_USERS_2 75
Table 4.5 Application-oriented patterns probe holder monthly costs for experiments INCREASING_KPIS_1, INCREASING_KPIS_2, and INCREASING_TARGETS_1 77
Table 4.6 Application-oriented patterns probe holder monthly costs for experiments INCREASING_TARGETS_2, INCREASING_USERS_1, and INCREASING_USERS_2 78
Table 5.1 States definitions for categorical and numerical indicators. . . . 97
Table 5.2 Accuracy of AdaptiveMon and StaticMon for the 5 scenarios. Green (Red) cells indicate a better (worse) result obtained by AdaptiveMon compared to the StaticMon. . . . 104
Table 5.3 Statistically valid comparisons for all the quality metrics with their associated effect size. . . . 108
Table 6.1 A set of four operation modes used in the motivational pedestrian detection scenario. . . . 115
Table 6.2 The domain of the parameters used to define the search space of the multi-objective optimization problem. . . . 118
Table B.1 GW Instek GPM-8213 Power Measurement Accuracy . . . 141

LISTINGS

Listing 3.1 A metadata excerpt from an HTTP health check probe entry in the Probe Catalog. 35
Listing 3.2 A metadata excerpt from the Apache Kafka exporter probe entry in the Probe Catalog. 44
Listing 3.3 A sample JSON representation of a Target retrieved from Microsoft Azure. 45
Listing 5.1 An example rule that uses the *Change Rate* countermeasure written with the CLIPS DSL. The symbol => separates the antecedent and the consequent of the rule. The `salience` value represents the rule priority. The `bind` operator assigns the result of a function call to a variable. . 100

Monitoring is a critical activity in several fields, such as environmental sciences [126], information and communication technology (ICT) [4], healthcare [158], and engineering [174]. A *monitoring system* gathers, transmits, and archives data by using probes to sense a target, whether it is natural (e.g., water), physical (e.g., industrial machinery), or virtual (e.g., applications). This data can help understand the target behavior and potentially provide meaningful insights through both online (e.g., real-time anomaly detection [254]) or offline (e.g., root-cause analysis [129]) analyses.

Nowadays, monitoring systems are being deployed more frequently along the *cloud continuum*. This is a seamless *"continuum"* of computing services that are available from traditional clouds running in data centers located in the core network, as well as from *heterogeneous devices* such as access points, routers, gateways, and cloudlets located in the metro and access networks [176, 263].

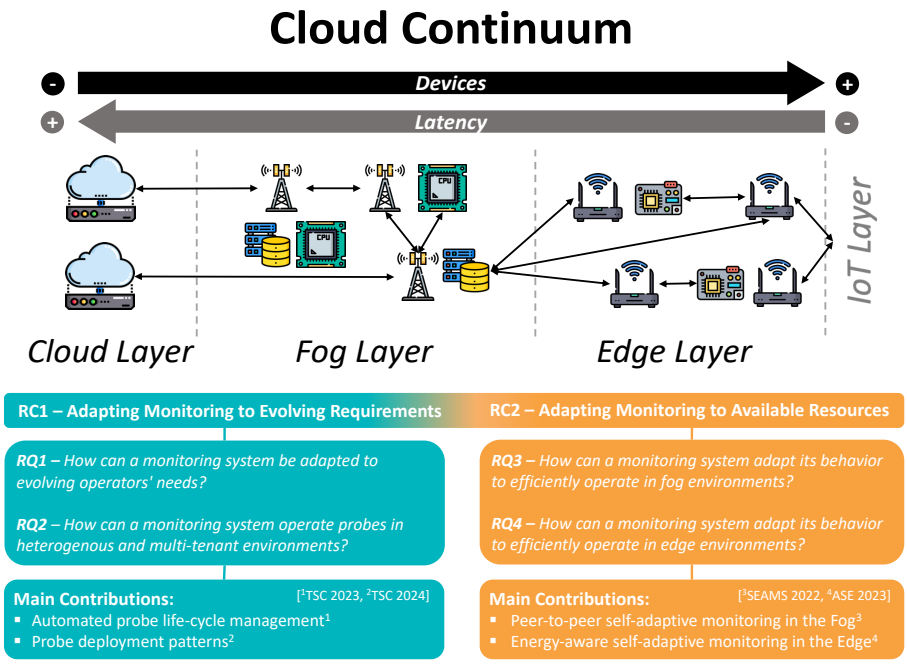


Figure 0.1: A graphical overview of the thesis context, research challenges (RCs), research questions (RQs), and main contributions.

The cloud continuum integrates the Cloud with the IoT through the *fog computing* and *edge computing* layers, as shown in the top part of Figure 0.1. Fog computing is a computing model that distributes computation, communication, control, and storage closer to the IoT at the edge of the network by using a hierarchical architecture [62,

263]. Edge computing also provides computational and storage facilities, but it is located at the very edge of the network, typically within one or two hops, with a distributed and localized architecture [258, 263]. Compared to cloud computing, both fog and edge computing exhibit lower network latency and greater responsiveness [43, 62, 263]. However, the number of devices in these layers is large, often reaching millions, and their computational capabilities are limited compared to those of cloud data centers. The cloud continuum results in a distributed and heterogeneous environment with varying software and hardware stacks. It is accessible in a multi-tenant fashion, meaning that resources are shared among tenants. Additionally, it includes fog and edge computing layers, which may be unreliable due to prevalent wireless connections and less powerful due to limited resources on the devices [263].

This thesis focuses on two challenges that impact monitoring systems operating in the cloud continuum. The challenges, research questions, and main contributions are summarized in Figure 0.1.

The first research challenge (RC1) concerns with the *adaptation of monitoring systems to evolving requirements*. For a monitoring system that operates in the cloud continuum, it is essential to support its automated evolution to accommodate changes in operators' needs due to unpredictable events such as anomalies, failures, and requests for new indicators to be collected [96, 215, 229]. For example, according to data from a survey involving 63 data centers conducted in 2016 [150], the average cost of downtime per data center increased by 38% from $500,000 in 2010 to $740,357 [214]. This remarks how enhancing monitoring systems with faster adaptation and automation capabilities can help predict anomalies and anticipate failures, ultimately impacting revenues and operational costs. Furthermore, since the cloud continuum is a heterogeneous environment used by multiple tenants, a monitoring system should abstract from underlying technologies and relieve operators from the configuration burden [1, 4].

In this challenge, the thesis delves into two research questions (RQs).
RQ1: How can a monitoring system be adapted to evolving operators' needs? RQ1 analyzes how a monitoring system can assist cloud operators and adapt its functionalities according to the their evolving needs while minimizing the number of operational changes. This thesis proposes a *Monitoring-as-a-Service (MaaS) framework* that can fully manage the life-cycle of probes, including error-handling, by starting from declarative input. The contribution has been published in the IEEE Transactions on Services Computing journal paper titled "Automated Probe Life-Cycle Management for Monitoring-as-a-Service" [242].
RQ2: How can a monitoring system operate probes in heterogeneous and multi-tenant environments? RQ2 studies possible probe deployment patterns by identifying and characterizing the components that can be used to deploy probes. This thesis provides the *definition, anal-*

ysis, and qualitative and quantitative evaluation of 11 possible probe deployment patterns. The contribution has been published in the IEEE Transactions on Services Computing journal paper titled "Monitoring Probe Deployment Patterns for Cloud-Native Applications: Definition and Empirical Assessment" [244].

The second research challenge (RC2) addressed in this thesis concerns with the *adaptation of monitoring systems to the available resources*, which is particularly relevant in the context of fog and edge environments. A monitoring system operating in these environments must efficiently use available resources to handle an increasing number of running devices, applications, and collected indicators, which produce a significant amount of data for storage and analysis [1, 228]. Additionally, it is crucial for a monitoring system to function effectively in unpredictable and possibly resource-limited conditions. This involves ensuring the system's capabilities while making efficient use of available resources, which may be limited at the network's edge [216].

Regarding this second research challenge, this thesis investigates two additional research questions.

RQ3: How can a monitoring system adapt its behavior to efficiently operate in fog environments? RQ3 studies how a monitoring system can efficiently operate in fog environments by adapting its behavior to changes in the monitored targets. This thesis proposes a *self-adaptive P2P monitoring system* that utilizes a *hierarchical P2P architecture* and incorporates adaptive behaviors based on the *MAPE-K feedback loop*. This contribution was presented at the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) and published in its proceedings with the title "Towards Self-Adaptive Peer-to-Peer Monitoring for Fog Environments" [65].

RQ4: How can a monitoring system adapt its behavior to efficiently operate in edge environments? RQ4 studies how a monitoring system can operate in resource-constrained edge environments guaranteeing its capabilities while wisely using available resources. This thesis proposes an *energy-aware approach* that can guide developers to implement an *(AI-based) self-adaptive monitoring application* able of switching its operation modes in response to changes in the environment, finally *balancing energy consumption with the application-level objectives*. This contribution was presented at the 38th International Conference on Automated Software Engineering (ASE) and published in its proceedings with the title "An Energy-Aware Approach to Design Self-Adaptive AI-based Applications on the Edge" [240].

The thesis is structured as follows. Chapter 1 presents the cloud continuum, its characteristics, and highlights the main challenges that affect the monitoring activities. Chapter 2 deeply analyzes the two RCs of monitoring in the cloud continuum, the identified research gaps, and the main thesis contributions to the four RQs. The thesis is then divided into two parts corresponding to the two research challenges of interest. Part I presents contributions about adapting monitoring sys-

tems to evolving requirements. In particular, Chapter 3 explores RQ1 and presents a monitoring framework for automated life-cycle management of probes, while Chapter 4 investigates RQ2 and describes the definition and assessment of probe deployment patterns. Part II presents contributions about adapting monitoring systems to the available resources. Specifically, Chapter 5 studies RQ3 and presents a self-adaptive monitoring approach for fog environments, while Chapter 6 investigates RQ4 and proposes an approach to design of energy-aware and self-adaptive monitoring applications for edge environments. Finally, Chapter 7 presents concluding remarks and future work.

1 THE CLOUD CONTINUUM

The rise of cloud computing is considered one of the factors that contributed to the development and spread of Internet-of-Things (IoT) applications [201]. These applications usually collect data from IoT devices (e.g., sensors, home appliances, smartphones), and rely on cloud resources for storage, data processing, and decision making [201, 263].

As reported in a recent analysis by IoT Analytics, the number of devices connected to the network should reach 16.7 billion by the end of 2023, while it is expected to be more than 29 billion by 2027 [49]. This large number of devices produces a massive amount of data, that might reach 80 billion zettabytes by 2025 according to a forecast conducted by the International Data Corporation in 2021 [117]. Moving this large amount of data from remote devices to cloud data centers can be inefficient, or in same cases, it might be even infeasible because of bandwidth limitations [263]. Despite an IoT-to-Cloud communication model can support non-latency sensitive applications, such communication model is unfeasible for (near) real-time scenarios with demanding response time constraints (i.e., milliseconds or microseconds), such as, patient monitoring, drone fleets, cognitive assistance, or autonomous driving [263].

Recently, researchers and practitioners started considering the potential benefits of locating computing resources closer to end devices, where data is generated [36, 41, 43, 62, 263]. By interconnecting large-scale cloud data centers in the core network, with servers and network devices distributed across the metro and access networks [263], it is possible to provide seamless access to a *continuum* of cloud resources, namely, the cloud continuum, to support the needs of such application scenarios [36, 176].

Many computing paradigms have been proposed in the last decade to fill the gap between the Cloud and the IoT devices [263], thus realizing the continuum. In particular, fog computing [43] and edge computing [216] emerged among many others (e.g., edge clouds [57], cloudlet [208], multi-access edge computing [203]), but multiple definitions of what the constituting tiers and the role of the cloud continuum are can be found in current literature.

This chapter is organized as follows. Section 1.1 presents the definition of cloud continuum and its recent evolution. Section 1.2 provides background information about cloud computing and its main limitations. Section 1.3 describes the fog computing characteristics and the main differences with the Cloud. Section 1.4 presents the edge computing and the main differences with the Fog. Finally, Section 1.5 in-

troduces the main challenges that affect monitoring activities in the cloud continuum environment.

1.1 INTRODUCTION TO CLOUD CONTINUUM

The definition of cloud continuum in current literature is not unique and has evolved in the last years [176]. In a recent mapping study, Moreschini *et al.* [176] consider 36 studies that propose definitions to cloud continuum dated from 2016 to 2022. The authors identify three main groups of definitions.

Studies in the first group define the cloud continuum as *an extension of the resources* and focus on their distribution related to the concept of fog and edge computing, that is, they consider it as "an aggregation of heterogeneous resources from the Edge to the Cloud" [176]. Studies in the second group define the cloud continuum as *an extension of computational capabilities* and focus on processing particularly, that is, they consider it as a "set of processing units located between the IoT and the Cloud" [176]. The third group is composed by the remaining studies not belonging to any of the two previous groups. They mainly focus on the different digital services executed across multiple physical infrastructures, without particular focus neither on the distribution of the resources nor on the computational capabilities [176].

The two earliest definitions of cloud continuum have been both presented in 2016. Chiang and Zhang [62] define the cloud continuum referring to computational aspects explicitly, highlighting where and how the computation is performed. On the other hand, Gupta *et al.* [113] define the cloud continuum as "a continuum of resources from the network edge, through the core network, to the data centers". It is worth noting the study by Gupta *et al.* [113] belongs to gray literature. However, it is referred to as the first definition of cloud continuum by many other peer-reviewed studies, and "it represents an important milestone for the definition of cloud continuum that has evolved over time with the addition/removal of other keyword" according to the authors of the mapping study [176].

In more recent studies, Dustdar *et al.* [81] define the cloud continuum as a system simultaneously operated across the cloud, fog, and edge computing tiers; while Spillner *et al.* [223] highlight it is a "novel abstraction layer to express a continuous range of capacities".

There is no complete agreement in current literature about how extensive the Cloud is, and so, about which other computing tiers (i.e., Fog, Edge, and IoT) are part of the cloud continuum. For instance, Kassir *et al.* [139] consider the terms "cloud-to-thing(s) continuum" and "Fog-to-Cloud continuum" synonyms. Similarly, both Mehran *et al.* [165] and Nezami *et al.* [180] use the terms "Cloud-fog continuum" and "fog continuum" to indicate the continuum extends the Cloud towards the fog computing. Kahvazadeh *et al.* [136] use the term "IoT continuum", but in the end they describe the same connection between

cloud and edge computing. On the other hand, Xhafa and Krause [260] define the cloud continuum as an ecosystem comprising digital services operated across Fog, Edge, and IoT.

In studies belonging to the second group, that is, those focusing on the processing capabilities, the definition of cloud continuum consider connecting any computational-enabled entities (e.g., data centers and fog/edge/IoT nodes). For instance, Beckman *et al.* [39] define the cloud continuum as "a collective of components with various capabilities and numbers in aggregate". Meanwhile, Balouek-Thomert *et al.* [35] define it as "a digital infrastructure jointly used by complex application workflows" without mentioning any specific nodes being connected by the cloud continuum.

By unifying the two main groups of definitions (i.e., extension of processing and extension of resources), Moreschini *et al.* [176] propose a new comprehensive definition of the cloud continuum as the outcome of their mapping study:

Cloud continuum *(Definition 1.1)*. Cloud continuum is an extension of the traditional Cloud towards multiple entities (e.g., Edge, Fog, IoT) that provide analysis, processing, storage, and data generation capabilities [176].

In this thesis, the cloud continuum serves as an extension of both processing and resources, as defined by Moreschini *et al.* [176] in Definition 1.1. Specifically, the Cloud, the Fog, and the Edge are considered as entities that realize the continuum of resources and processing, while the IoT is viewed as the motivation for implementing the cloud continuum, and not part of the constituting tiers.

1.2 CLOUD COMPUTING

Cloud computing became the main computing paradigm in the last decade, accelerating digital transformation and creating new business opportunities in many sectors (e.g., agriculture, healthcare, manufacturing, information and communication technology) [163, 226].

Cloud computing *(Definition 1.2)*. Cloud computing enables on-demand and ubiquitous access to (virtually) infinite shared resources (e.g., storage, servers, networks, and services) via the network. They are provided by pools of configurable and virtualized computing resources operated in data centers, that can be dynamically reconfigured with minimal effort to accommodate variable and scalable workloads [166].

The U.S. National Institute of Standards and Technology (NIST) defines the cloud model as consisting of five key characteristics [166]. Such characteristics contribute to the appealing attributes of cloud computing, such as on-demand provisioning, elasticity, ubiquitous accessibility, reduced initial investments, and accelerated time-to-market [41].

i) *On-demand self-service*, that is, a consumer can automatically provision computing resources (e.g., servers and storage) as needed without requiring any human interaction with the service providers;

ii) *Broad network access*, that is, cloud resources are available over the network and accessed via heterogeneous clients (e.g., mobile devices, laptops, and workstations) using standard mechanisms;

iii) *Resource pooling*, that is, provider's resources are pooled to offer services to multiple consumers in a multi-tenant fashion. Both physical and virtual resources are dynamically (re)allocated according to the current demand, and the customers are usually unaware of the precise location of the resources provided, despite they might be able to specify the location (e.g., data center);

iv) *Rapid elasticity*, that is, cloud resources can be elastically (and in some cases automatically) (un)provisioned to quickly adapt to the current workload demands. From the consumer's perspective, resources appear infinite and can be consumed at any time;

v) *Measured service*, that is, cloud-based systems leverage metering to control and optimize the resource usage. Usage metrics are visible to both the provider and consumer for monitoring purposes and to enable a pay-per-use cost model.

Cloud resources can be consumed according to three main service models: *Infrastructure-as-a-Service (IaaS)*, *Platform-as-a-Service (PaaS)*, and *Software-as-a-Service (SaaS)* [166]. Consumers can use a variety of these services depending on their needs [263].

Infrastructure-as-a-Service *(Definition 1.3)*. IaaS provides consumers with the capability to provision IT infrastructure for compute, storage, and network resources. Consumers have control over the provisioned infrastructure resources in terms of configurations (e.g., CPU/RAM capacity for compute services), operating systems, storage, and they are able to deploy and run arbitrary software. However, consumers do not manage or control the underlying cloud infrastructure [166, 263].

Platform-as-a-Service *(Definition 1.4)*. PaaS enables consumers to focus on software development, fully supporting the software life-cycle, often using middleware for software management and configuration. Consumers do not manage or control the underlying infrastructure (e.g., network, servers, operating systems, or storage). However, they have control over the deployed applications and possibly configuration settings for the hosting environment. In addition, PaaS providers often include tools for database management and application scaling [166, 263].

Software-as-a-Service *(Definition 1.5)*. SaaS allows consumers to use provider's applications that run on a cloud infrastructure. The applications can be accessed from a variety of client devices (e.g., web browsers or a program interface). Consumers have no control of the underlying cloud infrastructure (e.g., network, servers, operating systems, or storage) or even individual application capabilities. However, it is usually possible for consumers to access a limited user-specific application configuration settings [166].

The NIST defines also four types of cloud deployments: private cloud, community cloud, public cloud, and hybrid cloud [166].

Private cloud *(Definition 1.6)*. Private clouds provide infrastructure to a single entity exclusively and offer a high degree of privacy and configurability. They are a good choice for organizations that need infrastructure for their applications. A private cloud may be owned, managed, and operated by an organization, a third party, or a combination of the two, and may be located on or off premises [166, 263].

Community cloud *(Definition 1.7)*. Community clouds are used by specific community of users, and the cloud infrastructure is usually shared by multiple organizations that have the same concerns, for example, the same mission or compliance. A community cloud results in decentralized ownership of the cloud by multiple organizations within the community, a third party, or some combination of them, without relying on a large cloud provider for the infrastructure [166, 263].

Public cloud *(Definition 1.8)*. Public clouds are the most common deployment type of cloud computing. In a public cloud, the infrastructure is for open use by the general public and it exists on the premises of a cloud provider [166, 263].

Hybrid cloud *(Definition 1.9)*. Hybrid clouds are a combination of two or more different cloud deployment models (i.e., private, community, or public). They enable consumers to have greater control over the cloud infrastructure and streamline the capabilities of the different deployment models through standardized or proprietary technologies [166, 263].

Despite cloud computing helps to create ubiquitous computing and flexible access to resource pools, the time required to access cloud services can be prohibitive for some applications with (ultra)low-latency and high-bandwidth requirements [43, 263]. In addition, the increasing number of connected devices and the rapid growth of data generated at the edge require cloud resources to be closer to where the data is actually generated [263]. To address these needs, fog and edge computing paradigms have been proposed by both industry and academia [43, 216].

1.3 FOG COMPUTING

Fog computing *(Definition 1.10)*. Fog computing is a layered, decentralized, and hierarchical computing paradigm that enables ubiquitous access to computation, communication, control, and storage closer to the edge of the network relying on heterogeneous nodes [43, 125, 258]. It supports the deployment of distributed, latency-aware applications and services, and reduces data transfer costs, optimizing bandwidth usage, and improving user experience [43, 62, 258, 263].

The fog computing infrastructure consists of nodes, either physical or virtual, located in the metro and access network between edge devices and centralized cloud data centers in the core network [263]. Fog nodes are heterogeneous and enabled with virtualization capabilities, and they include servers, routers, switches, and access points [41, 43, 62, 263]. Moreover, they are context-aware, and can be organized in vertical clusters to support isolation, horizontal clusters to support federation, or latency-based clusters according to their distance from edge devices [125]. A wide array of industries could benefit from utilizing the fog computing paradigm, for example, energy, manufacturing, transportation, healthcare, and smart cities [263].

As with cloud computing, NIST defines the paradigm of fog computing describing its six primary characteristics [125]:

i) *Contextual location awareness and low latency*, that is, the low-latency capability is enabled due to contextual location and data exchange cost-awareness of nodes. Fog nodes, which are often positioned in closer proximity of end-devices, facilitate faster analysis and response to data generated by these devices compared to centralized cloud data centers;

ii) *Geographical distribution*, that is, geo-distributed and identifiable deployments are necessary for fog services and applications to provide high-quality services in proximity to end-devices (e.g., streaming services to vehicles on the move through nodes located along highways);

iii) *Heterogeneity*, that is, it enables the collection and processing of heterogeneous data acquired through different communication networks;

iv) *Interoperability and federation*, that is, fog components must interoperate and running services must be federated to provide seamless support for services necessitates collaboration among various providers (e.g., real-time streaming);

v) *Real-time interactions*, that is, it fosters real-time interactions rather than - or only - batch processing data-intensive applications and services;

vi) *Scalability and agility of federated clusters*, that is, it eases adaptive functions at both cluster and cluster-of-clusters level (e.g., computing elasticity, resource pooling, data-load management, and network condition variation).

In addition, they define two extra characteristics frequently linked to fog computing [125]:

i) *Predominance of wireless access*, that is, it is highly compatible with wireless IoT access networks to support the large number of wireless sensors that requires distributed analytics and compute;

ii) *Support for mobility*, that is, it must support mobility techniques to enable fog computing applications requiring direct communication with mobile devices.

Fog computing, like the conventional cloud computing model, implements the architecture in several network topology layers [125, 263]. Therefore, fog computing supports traditional cloud computing service models, that is, IaaS, PaaS, and SaaS, and enables private, community, public, and hybrid deployments [125].

Several differences exist between cloud and fog computing, particularly in the hardware infrastructure scale [263]. Cloud computing generally utilizes extensive data centers, whereas fog computing employs diverse and compact devices such as servers, routers, switches, gateways, or access points [62, 125, 263]. As these devices require less space for deployment in the field compared to the data centers used in cloud computing, they can be positioned closer to users, resulting in faster access times [263]. Moreover, cloud computing provides high availability of computing resources, albeit with comparably high power consumption, while fog computing supplies moderate resource availability with lower power usage [128].

Fog computing services and applications can be accessed through connected devices from the edge of the network (i.e., access and metro networks) to the network core, while cloud computing can only be accessed through the network core itself [263]. Additionally, fog services can operate autonomously with restricted or no Internet connectivity, and then transmit critical updates to the cloud as soon as a connection is again available. In contrast, cloud computing demands constant device connection while the cloud service is in use [263].

1.4 EDGE COMPUTING

The origin of edge computing is connected to the introduction of content delivery networks (CDNs) to accelerate web performance back in 1990 [207]. Since then, edge computing has evolved to encompass not only caching but also computation and storage assistance for mobile users and the IoT devices [207].

Edge computing *(Definition 1.11)*. Edge computing provides computational and storage facilities at the very edge of the network, typically within one or two hops [263]. It has a distributed and localized architecture consisting of heterogeneous resource-constrained devices with limited capabilities [258, 263]. Edge computing enables computation to occur at the edge of the network, processing downstream data in support of cloud services and upstream data in support of IoT services[216].

Edge computing allows for efficient access to network services with high bandwidth and ultra-low latency, making it ideal for real-time applications like surveillance, virtual reality, and traffic monitoring [141]. To support such requirements, it is characterized by:

i) *Mobility support*, that is, it decouples the location identity from the host identity and implements a distributed directory system (i.e., the Locator ID Separation Protocol (LISP)) to support mobile users [141];

ii) *Location and contextual awareness*, that is, it enables consumers to employ several technologies (e.g., cell phone infrastructure, GPS, or wireless access points) to find the location of the devices and access to the closest services to their physical location. Also, context information of the mobile device can be used to take offloading decisions [114], and it enables the providers to improve the Quality-of-Experience (QoE) [141];

iii) *Dense geographical distribution and proximity*, that is, it brings cloud-based services (i.e., computation and storage) on top of geo-distributed heterogeneous devices (e.g., access points, routers, base stations) at the edge of the network forming a very dense environment [141, 216]. The availability of the computational resources and services in the proximity of consumers allows leveraging the network context information for making offloading decisions and service usage decisions[114, 141];

iv) *Low latency*, that is, it reduces the latency in accessing the services by making available services and applications in the proximity of the end devices [141], enabling latency-sensitive application scenarios (e.g., cognitive assistance [59]);

v) *Heterogeneity*, that is, it is characterized by the existence of heterogeneous elements at different levels, for instance, end devices software and hardware technologies, service APIs and platforms, and communication protocols [141].

It is worth noting that in current literature, the terms fog and edge computing are often used interchangeably [216]. Additionally, some sources consider the Fog to encompass the Edge as well [263]. The OpenFog Consortium - now Industry IoT Consortium - tried to clarify

and make a distinction in the "OpenFog Reference Architecture for Fog Computing" white paper [258]. In particular, they describe the fog computing as hierarchical and able to provide computing, networking, storage, control, and acceleration anywhere from the Cloud to the IoT devices. On the other hand, the edge computing tends to be limited to computing and storage at the very edge of the network [263]. In this thesis, fog and edge computing are treated as separate entities, as motivated by the fog reference architecture white paper [258].

1.5 MONITORING CHALLENGES IN THE CLOUD CONTINUUM

The cloud continuum environment connects cloud data centers in the core network with fog and edge nodes in the metro and access network, resulting in a highly distributed and heterogeneous environment with varying software and hardware stacks [176]. The resources are accessible in a multi-tenant fashion, meaning that they are shared among tenants who jointly use them to deliver their services. Furthermore, the fog and edge computing tiers are less reliable and less powerful than the upper cloud tier due to prevalent wireless connections and limited resources available on the nodes providing computational capabilities [263].

These characteristics pose a challenge to the continuum monitoring systems on several dimensions. In particular, this thesis focuses two challenges.

The first challenge arises from the dynamic, heterogeneous, and unpredictable nature of the continuum environment. For example, unforeseeable circumstances affecting the monitored targets, such as service failures, or changing business requirements, may necessitate the collection of new indicators and modifications to the deployment location of monitoring components. This requires monitoring systems to adapt to changing operating requirements. The impact of this challenge is felt across all tiers of the continuum, starting from the cloud tier. Monitoring systems must handle various technological stacks, automate their configuration process, and strike a balance between technology constraints, and operators' needs in multi-tenant environments.

The second challenge pertains to the efficient utilization of computational resources in monitoring systems. Especially when monitoring activities are executed on top of fog and edge devices, the volume of collected data can be very large due to the number of devices involved. This can reach millions in the Fog and the Edge, and billions if we consider monitoring data coming from devices in the IoT tier [49]. Furthermore, when computations are performed at the network's edge, devices may be resource-constrained and powered by unreliable sources such as batteries present in the gateways deployed in remote rural areas for wildlife monitoring. Therefore, monitoring

systems must use available resources wisely and consider adapting the monitoring configuration.

The next chapter provides a deep analysis these two challenges by identifying the research gaps that resulted in the four research questions studied in this thesis.

2 MONITORING IN THE CLOUD CONTINUUM

Monitoring is a crucial activity in various fields, including environmental sciences, ICT, healthcare, and engineering. The data collected through monitoring activities can help understand the behavior of an observed target and provide meaningful insights through data analysis. According to the Oxford dictionary, the term "monitoring" is defined as follows.

Monitoring *(Definition 2.1)*. To observe, supervise, or keep under review; to keep under observation; to measure or test at intervals, especially for the purpose of regulation or control [77].

Monitoring activities are carried out by *monitoring systems*. For example, continuous emissions monitoring systems (CEMS) are used to monitor gas streams from combustion in industrial processes [126]. The data collected from CEMS is used to provide information for combustion control and to comply with air emission standards imposed by regulatory agencies [126]. Similarly, in the context of air pollution monitoring, data collected by sensors can be used to indicate air quality through colorful LEDs installed on IoT devices deployed in public areas, increasing public awareness about air quality [103]. In the ICT industry, monitoring computing and networking resources is crucial to meet Quality-of-Service (QoS) requirements, avoid Service Level Agreement (SLA) violations, and optimize capacity and resource planning, among other functionalities [4].

This chapter presents background information on monitoring systems and defines their basic components and functionalities. Additionally, it analyzes in depth the two RCs investigated in this thesis by identifying research gaps that resulted in four RQs. Finally, for each RQ, it briefly describes the main contributions.

2.1 ANATOMY OF A MONITORING SYSTEM

As monitoring systems are implemented in various domains and approached differently, literature offers several definitions. This thesis considers *active monitoring systems*, that is, monitoring systems which rely on observations collected by probes, in contrast to passive monitoring systems which rely on the passive observation and analysis of existing targets, such as network flows [47], logs [58], or execution traces [29]. This thesis defines a monitoring system with a comprehensive yet general description that includes all key elements of such a system.

Monitoring System *(Definition 2.2)*. A monitoring system is any software system operating in the cloud continuum that is designed to observe and measure any indicator of a physical or virtual target.

Target *(Definition 2.3)*. A target is any resource either physical or virtual that can be monitored.

Indicator *(Definition 2.4)*. An indicator (i.e., a metric) is an observable target behavior for which raw measurements can be collected [85, 133, 146].

Probe *(Definition 2.5)*. A probe is a component located close enough to a target responsible for collecting one or multiple indicators [238, 248], such as sampling the CPU consumption of a service or recording the temperature in a room from a sensor.

Active monitoring systems can be classified into two groups: those that deploy probes inside the targets to collect indicators directly from the target's environment and send the data to an external collector, and those that rely on external probes to retrieve indicators from interfaces exposed by the targets without adding any software to them. Although monitoring systems that rely on external probes have low maintenance costs and less risk of interference, monitoring systems that deploy probes within the targets can provide deeper and more specialized measurements than the protocols used in the first case (e.g., SNMP [54]). This is because they can access and have visibility of the same environment.

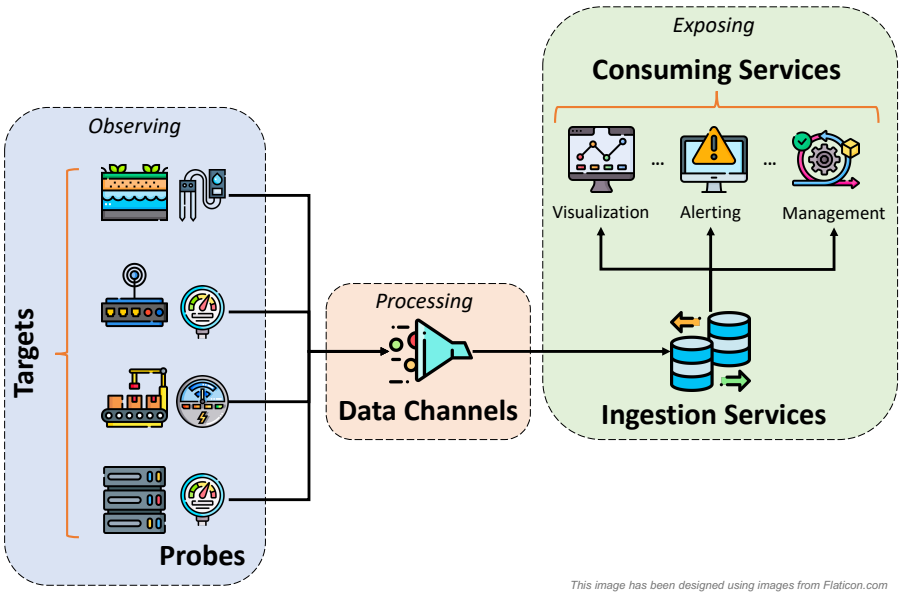


Figure 2.1: Generic architecture of a monitoring system.

A monitoring system provides three main functionality: (i) observing, (ii) processing, and (iii) exposing [1]. Generally, such functionalities are realized by mean of four key components: probes, data chan-

nels, ingestion services, and consuming services. Figure 2.1 graphically illustrates these components.

The *probes* are opportunistically placed to efficiently observe and collect indicators from the *targets*. 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 [4, 248].

The *ingestion services* can be both data repositories used to store indicator values for offline analysis, and data streams used to immediately consume incoming data for online analysis. The communication between the probes and the ingestion services can be mediated by *data channels* that are responsible for processing and transferring the data. In some cases, the data channels could be as simple as direct communication between the probes and the ingestion services. In some other cases, the data channels are pipelines devoted to data pre-processing and distribution, according to non-trivial strategies.

The *consuming services* are used to access and utilize the monitoring data provided by the ingestion services. When the collected data is used to support advanced analysis services, multiple systems may analyze the collected indicator values (e.g., alerting or management and planning systems).

2.2 ADAPTING MONITORING TO EVOLVING REQUIREMENTS

Monitoring systems operating in the cloud continuum face several challenges posed by the characteristics of such environments, and the needs of both applications and operators. The first research challenge (RC1) investigated in this thesis concerns with the *adaptation of monitoring systems to evolving requirements*.

For a monitoring system operating in the cloud continuum, supporting its automated evolution is crucial to accommodate changes in operators' needs (e.g., the request of collecting new indicators) or to react to run-time unpredictable events such as anomalies and failures. Additionally, the cloud continuum is a very heterogeneous environment utilized in a multi-tenant fashion [176], thus a monitoring system should abstract from underlying technologies and alleviate operators from the configuration burden of adapting the system to emerging requirements [1, 4].

In summary, the monitoring systems should facilitate their automated evolution while adhering to requirements from a diverse and multi-tenant environment. The following section examines the existing literature, identifies research gaps, and briefly outlines this thesis's contribution to the related research questions.

2.2.1 *Support to Automated Evolution*

A monitoring system must continuously observe the monitored resources to timely react to anomalous behaviors, generating alerts, and activating countermeasures [55, 149, 200, 215, 253]. Several cloud solutions are systematically enriched with monitoring capabilities, either natively offered by the platforms (e.g., Kubernetes [27]), or provided by external tools (e.g., Elastic Stack [31] and Prometheus [26]).

These monitoring systems are mainly designed to collect a stable set of indicators over time, being *challenged by scenarios that require rapidly modifying the set of collected indicators*. In contrast, there are many well-known causes of sudden changes to the set of collected indicators. The *goals of the operators* change with the technical and business objectives of the organization, consequently causing changes in the set of the indicators that must be collected. The *software usage patterns* that emerge from the field continuously evolve, often determining the need of adjusting the monitored indicators accordingly. The collected indicators must be adapted to changes in the *workload*, which must be carefully observed to timely reveal any symptom of stress on the services. Moreover, *service updates* normally require putting in place ad-hoc monitoring capabilities that target the updated services to measure their reliability and timely detect misbehaviors. Sometimes, the observation of *failures* generates the need of continuously observing the services that fail often, to prevent new failures and localize the causes of problems; and *dynamically deployed scenarios* (e.g., to timely react to disasters and emergencies) require quickly deploying new functional services and the corresponding monitoring components.

Relevantly, all these factors are *dynamic and cannot be entirely anticipated*. Changing the set of collected indicators often requires changing the set of probes running in the field. However, configuring and deploying new probes, as well as undeploying the existing probes, are *non-trivial and time-consuming activities*. For instance, a tech company running many cloud services needs to collect indicators at different granularity levels, taking into account both business and technical needs [215]. The needs of managers shall follow business goals and market evolution, while the needs of technicians shall follow QoS goals and software evolution. These needs evolve independently, and simultaneous changes in both business and technology may generate a rapidly increasing number of requests for the operators responsible of configuring the monitoring system. Operators may struggle adapting their monitoring systems at some point, especially when a large number of targets (e.g., devices, platforms, and services) has to be monitored. The first research question (RQ1) studies the adaptation of monitoring systems.

Research Question 1 (RQ1)

How can a monitoring system be adapted to evolving operators' needs?

This RQ analyzes how a monitoring system can assist cloud operators and adapt its functionalities according to the their evolving needs while minimizing the number of operational changes.

To address dynamicity and evolution of monitoring systems, researchers and practitioners focused on *increasing the level of automation of probe management*. Figure 2.2 shows the increasing levels of automation that have been introduced in monitoring systems.

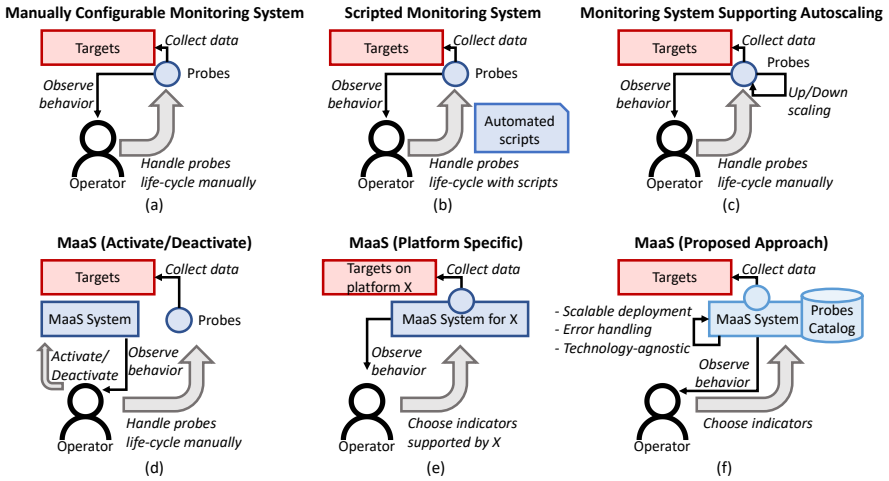


Figure 2.2: Automation levels introduced in monitoring systems.

Simple *manually configurable* monitoring systems (Figure 2.2 (a)), such as Elastic Stack [31] and Prometheus [26], require configuring and deploying probes manually, that is, the life-cycle of every component of the monitoring system must be handled manually by operators. Although useful, these monitoring systems are expensive to use in presence of frequent changes to the set of collected indicators, and badly adapt to dynamic scenarios.

Some probe deployment tasks could be implemented using *general purpose deployment systems* (Figure 2.2 (b)), such as Ansible [124] and Puppet [123]. However, these systems are not designed to specifically serve monitoring systems, and defining and controlling the deployment strategies would still be entirely on the shoulder of the operators. As discussed next in this thesis, general purpose deployment systems can be indeed used as basic building blocks of more sophisticated deployment solutions.

A simple form of automation present in some systems consists of the *support to autoscaling* (Figure 2.2 (c)), that is, probes automatically adapt to a changing number of replica of a monitored target [238]. This is a useful feature, although limited to a specific scenario, missing to

cope with the many changes that must be actuated as a consequence of changes on the set of collected indicators and monitored targets.

To obtain a sufficient level of flexibility to address the aforementioned characteristics, *Monitoring-as-a-Service* (MaaS) solutions have been studied [4, 87, 192, 238] (Figure 2.2 (d)-(f)). In fact, MaaS frameworks provide operators with the capability to flexibly decide the set of indicators to be collected, alleviating them from the burden of configuring and handling the life-cycle of the probes. In principle, an operator using a MaaS framework can simply specify the set of indicators that must be collected, while the operational aspects are automated by the framework.

Unfortunately, in many cases, automation is *limited to the activation of manually pre-deployed probes* [238] (Figure 2.2 (d)), that is, probes that have been already installed and configured *manually*. Adding probes to collect new indicators and removing existing probes must still be done manually by operators.

A higher degree of automation is provided by some *specific platforms* (Figure 2.2 (e)) that natively offer monitoring capabilities (e.g., Monasca [192]). These solutions are effective but significantly limit both the range of platforms and indicators that can be used. *So far, there is no general MaaS solution that can be used to collect virtually any indicator on any platform*. Note that a MaaS system that fully handles the life-cycle of probes is *the only solution that can entirely free operators from the burden of handling probe deployment*. In fact, they would be able to control the monitoring system by simply specifying the set of indicators to be collected.

Contribution

To address the aforementioned gap in MaaS solutions, this thesis proposes a *MaaS framework* (Figure 2.2 (f)) that exploits both a catalog of probes annotated with metadata and access to the API of the environment running the monitored resources, to deliver *full MaaS capabilities including error-handling*. The empirical results show the *effectiveness* of the framework with both *containers and VMs*, the *efficiency of error-handling*, and the *scalability* for an increasing number of operators' requests. This contribution is presented in Chapter 3.

2.2.2 *Support to Multi-Tenancy and Heterogeneity*

To deal with the multi-tenancy and the diverse number of technologies that characterize the cloud continuum environments, a monitoring system must also consider how to distribute the probes in order to respect and optimize both technological and operators' requirements. In fact, multiple probes serving different operators in a multi-tenant environment can be deployed within a same VM to save computa-

tional resources, at the expense of a reduced degree of privacy and security. On the other hand, one probe per container or VM can be deployed to preserve privacy, at the expense of more computational resources allocated to the monitoring system.

The flexibility of monitoring systems 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.

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 literature discusses the characteristics of monitoring systems, without investigating the many possible probe deployment patterns and their impact [4, 13, 87].

The second research question refers to monitoring systems and probe deployments.

Research Question 2 (RQ2)

How can a monitoring system operate probes in heterogeneous and multi-tenant environments?

This RQ studies possible probe deployment patterns by identifying and characterizing the components that can be used to deploy probes.

In software engineering, patterns are used to document knowledge about how to solve recurring problems [209]. With the rise of the cloud computing paradigm, the community has begun working on cloud computing patterns [88, 143, 221]. Although their development is still in the early stages, several online catalogs have been published, providing both specific [16, 169] and agnostic [88] solutions. Specific cloud 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. Agnostic pattern definition is a valuable means of improving portability and interoperability between different cloud environments [76]. However, none of these work specifically address the issue of probe deployment.

Burns and Oppenheimer [50] propose design patterns for distributed systems based on containers. Despite their work does not address the issue of probe deployment, some of their patters can be used for monitoring purposes (i.e., adapter single-node, multi-container patterns).

Albuquerque *et al.* [9] present proactive monitoring design patterns for cloud-native applications, basing their definitions on existing lit-

erature and tools. In particular, they present three patterns that can generate events according to the event-based monitoring paradigm, that is, Liveness Endpoint, Readiness Endpoint, and Synthetic Testing. However, they do not focus on the placement of monitoring probes and the possibility to share monitoring resources among users, and they do not provide any empirical assessment about pattern scalability as well.

Contribution

To address the research gap mentioned and answer RQ2, this thesis provides the *definition, analysis, and qualitative and quantitative evaluation of 11 possible probe deployment patterns*. The results demonstrate the *trade-offs between patterns* that require more resources to ensure good separation between users in multi-tenant environments and patterns that make better use of resources while reducing the degree of separation. The results have been cross-validated by addressing three realistic monitoring scenarios. *Best practices* have been distilled from the findings to guide engineers in implementing and configuring their monitoring systems. This contribution is presented in Chapter 4.

2.3 ADAPTING MONITORING TO AVAILABLE RESOURCES

The second research challenge (RC2) investigated by this thesis concerns with the *adaptation of monitoring systems to the available resources*, that is particularly relevant in the context of fog and edge environments. A monitoring system operating in these environments must efficiently use available resources to handle an increasing number of running devices, applications, and collected indicators, which produce a significant amount of data for storage and analysis [1, 228]. Additionally, it is crucial for a monitoring system to function effectively in unpredictable and possibly resource-limited conditions. This entails ensuring its capabilities while utilizing available resources wisely, which may be scarce at the edge of the network [216].

In summary, monitoring systems in fog and edge computing environments must efficiently utilize available resources by adapting their configuration and balancing their capabilities in resource-limited conditions. The following section discusses current literature, identifies research gaps, and briefly describes this thesis's contribution to related research questions.

2.3.1 *Efficiently Use Resources in the Fog*

In the last decade, a large number of cloud monitoring solutions, both commercial and academic, have been proposed [31, 37, 51, 182, 191, 215, 238, 241, 245]. However, they are seriously challenged by sev-

eral characteristics of the Fog, such as its massively distributed infrastructure characterized by frequent changes to the topology, and the presence of resource-constrained devices [228, 263].

Taherizadeh *et al.* [228] investigated the requirements that must be satisfied by monitoring systems specialized for adaptive applications orchestrated upon the cloud continuum. The survey reveals that none of the solutions available for the Cloud can satisfy all the requirements, identifying decentralization and resource optimization via self-adaptation as two of the main open challenges. Similar conclusions have been reported by Abderrahim *et al.* [1] who explicitly identify the adaptability of the granularity of the reported measures as one of the key properties for monitoring systems that operate in the Fog.

Peer-to-Peer (P2P) architectures have been investigated as viable approaches to effectively address monitoring in the Fog [1, 91, 108, 263]. P2P systems are "self-organizing systems of equal, autonomous entities (peers) which aim for the shared usage of distributed resources in a networked environment avoiding central services" [186]. Therefore, they represent a legitimate option to address the dynamism of the Fog without imposing strong constraints on the stability of the operating environment.

Unfortunately, although these monitoring systems show some degree of adaptivity thanks to the features provided by P2P architectures (e.g., they can tolerate node disconnections and broken communication links), they lack adaptation mechanisms that take into account the monitored indicators [1, 91, 108]. In fact, the collected indicators reveal important information about the monitored resources and their environment, and can be exploited to increase the awareness and adaptability of the monitoring system itself. For example, a monitoring component running in a device exhausting its battery may stop monitoring the non-essential indicators. Similarly, the trend of a monitored indicator can be used to optimize the sampling rate to avoid wasting resources (e.g., increasing/decreasing the sampling rate based on the degree of stability of the indicator). Based on these considerations, the following third research question is formulated.

Research Question 3 (RQ3)

How can a monitoring system adapt its behavior to efficiently operate in fog environments?

This RQ studies how a monitoring system can efficiently operate in fog environments by adapting its behavior to cope with a growing number of running devices, applications, and collected indicators that produce a large amount of data to store and analyze.

Monitoring approaches specifically designed for the fog environment have been recently investigated [46, 91, 109, 222]. In particular, FMonE [46] is a monitoring system that relies on a container orchestra-

tion system to build monitoring pipelines, addressing the distinctive features of a fog infrastructure. It provides users with the flexibility to define their monitoring pipelines and operate them across the active regions.

PyMon [109] is a lightweight prototypical monitoring system available for relevant Docker-enabled architectures such as ARM, AARCH64 and x86_64, and particularly suitable for single board computers (SBC). It extends the host-based monitoring tool Monit with capabilities to inspect running Docker containers.

Souza *et al.* [222] proposed a monitoring system that extends the CLABS model [12] and it is capable of monitoring targets by deploying services along the cloud continuum.

Unfortunately, none of these solutions implement adaptive policies to adapt the behavior of the monitoring system to the collected data. Furthermore, they are not based on a P2P architecture, so that, they struggle to cope with some of the fog distinctive traits such as heavily distributed infrastructures, rapid changes in the topology, and communication links failures.

FogMon [91] is a fog-oriented monitoring system that collects and aggregates data about resource consumption, network conditions, and IoT devices directly connected to fog nodes. It exploits a two-tier (Leader-Follower) P2P architecture and gossip protocols to reduce the network overhead. Also, it adapts the number of Leader nodes in the P2P overlay and the underlying Followers topology based on current network conditions. However, it does not provide any self-adaptive behaviors to govern the internal functioning of the monitoring system. For instance, FogMon cannot be used to dynamically change the set of the collected indicators or the sampling rate.

Among the works that are not specifically designed for monitoring in fog environments, it is worth mentioning some that still relate to it. In particular, ADMin [239] is an IoT-specific monitoring framework designed to reduce the energy consumption of the devices and the volume of data sent over the network. This is achieved essentially by adapting the rate at which devices disseminate monitoring streams based on run-time knowledge (e.g., stream evolution, variability, seasonality).

Also, Tangari *et al.* [230] propose a self-adaptive and decentralized system for resource monitoring in the scope of Software Defined Networks (SDN). It enables indicators collection through a self-tuning and adaptive monitoring technique that adjusts its settings based on traffic dynamics to balance operation costs with monitoring accuracy while reducing network overhead. However, the proposed system lacks generality since the adaptation capabilities are limited to some predetermined aspects, and it is not designed to support the capability to run multiple and diverse adaptation rules.

SkyEye [108] is a monitoring solution operating on structured P2P overlay networks. It provides continuous monitoring for a wide range

of indicators for all peers in the network. It is characterized by a tree structure, which enables peer partitions in a hierarchical fashion. The aggregated monitoring information received by the upper layers of the tree describe the information of the peers in the corresponding subtrees. Messages are used to disseminate the global monitoring data retrieved from the top levels and maintain the tree topology. Nevertheless, it is not explicitly designed for the Fog, and it completely lacks monitoring adaptivity (e.g., changing the set of collected indicators).

Contribution

To address the research gap mentioned and answer RQ3, this thesis proposes a *self-adaptive P2P monitoring system* that utilizes a *hierarchical P2P architecture* and incorporates adaptive behaviors based on the *MAPE-K feedback loop* [140]. The system can abstract monitored indicators and activate countermeasures based on their status. Countermeasures are defined using a *lightweight rule-based system* embedded in the peers. The empirical evaluation compares the accuracy and effectiveness of the adaptive version of the monitoring system with the non-adaptive version. The results show that adaptive behaviors can increase the accuracy of collected data and save network and power consumption, but at the cost of higher memory consumption. This contribution is presented in Chapter 5.

2.3.2 *Efficiently Use Resources in the Edge*

The Edge is the last tier of the cloud continuum, and it is particularly characterized by resource-constrained devices that cannot indefinitely supply a constant amount of power, such as, battery-powered devices and computing devices powered by renewable energy sources (e.g., photovoltaic panels or wind turbines) [52, 84, 190].

AI-based monitoring systems are particularly resource-intensive applications that are increasingly deployed along the cloud continuum and especially on the Edge, thus, carefully using energy is a key requirement to feasibly run AI services within these environments. For example, critical monitoring services for smart cities (e.g., pedestrian detection and traffic analysis [66, 160, 178]), environmental monitoring applications (e.g., wildfire detection [15, 156], and wildlife monitoring [80, 210]), all require fast data processing and high accuracy, with cost-effective energy consumption.

These scenarios require consuming a large volume of data generated from Internet-of-Things (IoT) sensors in various forms (e.g., time series values, video streams, images) with resource-greedy machine learning models (e.g., exploiting TPUs or GPUs) [95, 132, 197]. In contrast, the feasibility of scenarios that involve battery-powered devices [3, 22]

depends on the capability of reducing energy consumption to extend the battery life.

For these reasons, reducing energy consumption is a high-priority objective and a key technical challenge to wisely use the available resources [132]. The issue is exacerbated by the significant amount of energy consumed by ICT services and the increasing energy costs [90, 119, 152, 187], but also by initiatives like the European Green Deal [67] that accounts for "prioritizing energy efficiency" in its key principles. Based on these observations and critical aspects, the fourth research question for this thesis is formulated.

Research Question 4 (RQ4)

How can a monitoring system adapt its behavior to efficiently operate in edge environments?

This RQ studies how a monitoring system can operate in resource-constrained edge environments guaranteeing its capabilities while wisely using available resources.

Researchers have investigated several approaches to design systems with a controllable and programmable trade-off among quality, efficiency, and energy consumption. Energy-awareness and efficiency research mainly targets low-level tasks such as scheduling and provisioning [14, 18, 98, 179, 219], routing [206], data storage and processing [246], and machine learning models optimization [45]. Although valuable, only optimizing the low-level tasks may result in hardly-predictable performance of the applications. Thus, it becomes challenging or even impossible to balance competing application-level objectives (e.g., accuracy, energy consumption, and efficiency) working only on low-level features.

Other approaches targeted code optimizations [204], analysis of software energy consumption [83, 237], and architectural tactics to contain energy utilization [63] and costs [251]. Analyzing energy consumption retrospectively to take corrective actions (e.g., code or architectural refactoring) can be expensive and difficult to control in the long term.

In the context of IoT architectures and edge oriented systems, self-adaptation and optimization technologies have been used to address a range of aspects. For instance, adaptation capabilities have been engineered to achieve auto-scaling and task offloading [10], introducing flexibility in the computation at the cost of some jitter in the quality of service and, often, not optimized energy consumption shifts among the nodes [132].

Multiple approaches have been defined to modify the behavior of the components at the edge. The most common examples of self-adaptive edge components are those related to adaptive sampling. Adaptive sampling refers to the idea of dynamically modifying the sampling rate of sensors and software probes as well as the inference

rate of the components that process such data, according to the context [99, 168, 264]. Collecting and transmitting less data can save energy and computational resources [127].

Similarly, adaptive filtering focuses on reducing the number of samples transmitted. For example, if a sensor value is considered similar to a previously collected value or evolves in a predicable way, a monitoring node can avoid the transmission of such information to save the transmission cost. Since filtering usually results in sub-optimal performance, the filters must adapt at run-time to guarantee a consistent behavior [99].

Adaptive compression has been also extensively exploited at the edge. Adaptive compression solutions aim at reducing the data traffic in the network by reducing the size of the data packets with minimal loss, for instance using strategies that consider the importance of the processed data [159]. Different compression algorithms may also be used dynamically based on the shape of the data, enabling higher compression without inducing significant losses in the accuracy of the data [60].

Self-adaptive behaviors to improve energy consumption have been also studied at the architectural level [132]. For instance, a number of approaches have been proposed to target specific aspects of energy-awareness such as memory handling [131], networking [34], storage [246], and scheduling and provisioning [18]. Furthermore, the ever growing interest in machine-learning based solutions lead to specific optimized models for the edge [45]. These solutions can address specific dimensions but lack both the state-based adaptation capabilities, and the definition of a practical empirical procedure to determine the concrete configurations that must be used by the self-adaptive applications. Conversely, Da Silva *et al.* [218] proposed a framework for the automatic generation of application processes. Such processes represent the goals and capabilities of the application in the form of application workflows. This level of adaptation is not usually suitable for edge applications, since the run-time generation of the application processes requires extensive computational capabilities and introduces significant computational overhead [53], which may not be available at edge.

Mobile applications is another domain of self-adaptation where energy consumption is pivotal [111]. While adaptation mechanisms designed for mobile applications are not directly comparable to applications running on the Edge, they share some key aspects, such as the presence of a resource-constrained and battery-powered devices. For instance, Ardito *et al.* [20, 21] proposed an architectural paradigm in which the operating system or the middleware is able to offer energy-related information to running applications. This enables the implementation of energy-aware self-adaptation strategies based on energy levels, but it assumes run-time information about the available energy, that it may be not always available.

Contribution

To address RQ4 and the aforementioned limitations, this thesis proposes an *energy-aware approach* that can guide developers to implement an *(AI-based) self-adaptive application* able of switching its operation modes in response to changes in the environment, finally *balancing energy consumption with the application-level objectives*. The configuration of the operation modes are determined empirically, based on a *meta-heuristic search procedure* that can identify useful configurations by sampling a small portion of the configuration space. Experimental results show how the proposed approach can outperform non-adaptive baseline configurations, behaving as optimally as configurations selected with a nearly exhaustive exploration of the configuration space. The approach has been studied in the context of a Smart Traffic Monitoring (STM) scenario, in particular for a pedestrian detection task. This contribution is presented in Chapter 6.

Part I

ADAPTING MONITORING TO EVOLVING REQUIREMENTS

3

AUTOMATING PROBE LIFE-CYCLE FOR CHANGING NEEDS

This chapter presents a Monitoring-as-a-Service (MaaS) framework that fully automates the life-cycle of probes, including error-handling. The proposed framework enables operators to easily automate probe deployments required by changing needs, as discussed in Section 2.2.1. Moreover, it is designed to integrate with different monitoring technologies (i.e., probes and ingestion services) and cloud platforms (e.g., IaaS, PaaS, or SaaS solutions) without binding the operators to a single technical solution. The empirical evaluation examines the framework's capabilities and scalability using both VMs provided by a IaaS solution and a container-based platform. The contribution presented in this chapter has been published in the IEEE Transactions on Services Computing journal paper titled "Automated Probe Life-Cycle Management for Monitoring-as-a-Service" [242].

The chapter is organized as follows. Section 3.1 presents a running example used throughout the chapter to exemplify the proposed framework. Section 3.2 introduces the domain concepts. Section 3.3 presents the architecture of the MaaS framework, its main components and their algorithms. Section 3.4 describes the error-handling capabilities. Section 3.5 describes the technology-agnostic design and how is it possible to use the framework with different monitoring technologies. Section 3.6 presents the empirical evaluation. Finally, Section 3.7 concludes the chapter with closing remarks.

3.1 RUNNING EXAMPLE

This section introduces a running example to illustrate and exemplify how the proposed MaaS framework works. The example consists of a PostgreSQL instance [110] TARGET-PSQL running as part of a larger cloud system. Such an instance is of interest for two operators: operator OP-A and operator OP-B. Operator OP-A is mostly interested in infrastructure indicators and is collecting network consumption data related to TARGET-PSQL. Operator OP-B is interested in both infrastructure and application indicators, and is collecting 3 indicators: network consumption data, CPU consumption data, and database metrics. This initial configuration is referred as INIT-CONF.

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

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

The rest of the chapter refers to these sample scenarios and configurations to explain how the set of probes necessary to collect the indicators required by operators OP-A and OP-B can be adjusted automatically and transparently to the operators.

3.2 DOMAIN CONCEPTS

The proposed MaaS framework exploits a few relevant domain concepts to organize the responsibilities of the components. In the following, domain concepts are introduced, both informally and rigorously, and then the framework architecture is discussed.

In the running example, the target is a PostgreSQL instance that can be identified with the label TARGET-PSQL in both a Kubernetes cluster (as deployment name) and Microsoft Azure Compute Services (as VM name).

Probe Artifact *(Definition 3.1)*. A probe artifact represents a *deployable artifact* that can be used to collect indicators from targets in different environments. Probe artifacts (N.B., hereinafter in the chapter referred as probe for simplicity) are annotated with metadata that describe how they can be deployed and configured. More rigorously, a probe p is a tuple $p = (I, meta, artifact)$, where $I = \{i_1, \dots i_n\}$ is a set of indicators that can be collected with the probe, *meta* is a set of key-value pairs that represent the metadata associated with the probe, and *artifact* is a reference to the artifacts that implement the actual software probe.

The notation p^I, p^{meta} and $p^{artifact}$ refers to the individual components of a probe p.

Monitoring Claim *(Definition 3.2)*. A monitoring claim specifies the indicators that an operator may want to collect for a specific target. More rigorously, a monitoring claim *mc* is a tuple $mc = (I, op, t)$ where $I = \{i_i, \dots i_k\}$ is the set of indicators to be collected from the target t for the operator *op*. The claim is intended as a complete specification for the specified target, thus if the operator is already monitoring an indicator i for a given target t and the newly submitted claim does not include the indicator, the monitoring system will stop collecting i from t.

For example, operator OP-A shall submit a monitoring claim ({NETWORK_CONSUMPTION, CPU_CONSUMPTION, USER_SESSION_DATA}, OP-A, TARGET-PSQL) to start collecting CPU consumption and user session data, in addition to network consumption. Similarly, operator OP-B shall submit a monitoring claim ({}, OP-B, TARGET-PSQL) to stop collecting data.

Monitoring Request *(Definition 3.3)*. A monitoring request is a collection of monitoring claims submitted with a single request by an operator. More rigorously, a monitoring request *mr* submitted by operator op is a set $mr = \{mc_1, \ldots mc_m\}$ where $mc_i = (I_i, op, t_i)$.

For example, operator OP-A shall submit a monitoring request consisting of two monitoring claims [({NETWORK_CONSUMPTION}, OP-A, TARGET-PSQL), ({CPU_CONSUMPTION}, OP-A, TARGET-MARIADB)] to start collecting network consumption from the PostgreSQL instance and CPU consumption from a MariaDB instance.

Monitoring Unit *(Definition 3.4)*. A monitoring unit is an execution unit (e.g., a virtual machine or a container) that runs one or more probes. When needed, the monitoring framework dynamically creates and destroys monitoring units to collect the indicators specified by the operators in their monitoring claims. A monitoring unit is also characterized by a hosting platform, which represents the environment where the unit is executed, and a configuration, which captures how the probes in the monitoring unit are configured. More rigorously, a monitoring unit *mu* is a tuple $mu = (host, mus, C)$, where $host$ identifies the platform that provides the unit, mus indicates the strategy used to configure the unit (i.e., single probe or multi-probe), and C is the configuration of the unit, which consists of zero or more *probe configurations*, depending on the number of probes installed.

Probe Configuration *(Definition 3.5)*. A probe configuration $c \in C$ is a tuple $c = (p, I, op)$, where p is a probe, $I \subseteq p^I$ represents the set of indicators that p is configured to collect, and op is the operator who asked for the probe configuration c.

The notation mu_P refers to the set of probes in the current configuration of mu, that is, $mu_P = \{p | \exists (p, \cdot, \cdot) \in C\}$[1]. Finally, given a probe configuration (p, I, op), The notation $I(p)$ refers to the indicators that p is configured to monitor, that is, $I(p) = I$.

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

For instance, the initial configuration of the running example, where virtual machines running on Microsoft Azure are used, implies the

1 The symbol $\cdot$ means any value is allowed in a tuple.

existence of a single monitoring unit $mu = (azure, multi\text{-}probe, C)$, running the probe p_{net}, which serves both operators OP-A and OP-B, and the probes p_{cpu}, p_{db}, which both serve operator OP-B. Consequently, C consists of the following four probe configurations:

1. $(p_{net}, \text{NETWORK\_CONSUMPTION}, \text{OP-A})$,
2. $(p_{net}, \text{NETWORK\_CONSUMPTION}, \text{OP-B})$,
3. $(p_{cpu}, \text{CPU\_CONSUMPTION}, \text{OP-B})$,
4. $(p_{db}, \text{DB\_METRICS}, \text{OP-B})$.

Note that the monitoring units are created to have the right visibility of the target to be monitored. In fact, a virtual machine monitoring unit can be either the same virtual machine running the monitored service or a separated virtual machine with probes that query an interface exposed by the monitored service (e.g., using SNMP [54]). On the other hand, a container monitoring unit can be created as a sidecar of the container running the target service [50], to have extensive visibility of the monitored service, or as a standalone container running in the same node of the target. In the next chapter, a broader set of strategies to configure monitoring units (i.e., probe deployment patterns) are presented.

3.3 SOLUTION ARCHITECTURE

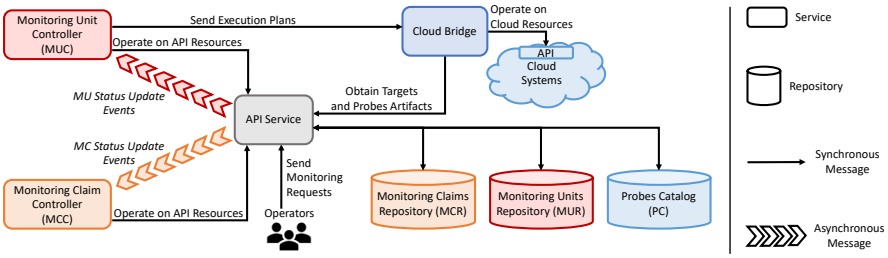


Figure 3.1: Architecture of the MaaS framework.

Figure 3.1 shows the proposed *monitoring framework*, which consists of four main stateless services and three repositories. The four services are (i) an *API Service*, which offers a gateway to access and update state information about the monitoring system, (ii) a *Monitoring Claim Controller*, which is responsible for handling the life-cycle of every monitoring claim, (iii) a *Monitoring Unit Controller*, which is responsible for handling the life-cycle of every monitoring unit, and (iv) a *Cloud Bridge*, which exploits a plug-in based architecture to interact with different cloud providers and platforms, actuating the operations decided by the other services. The three repositories consist of (i) a repository of

monitoring claims submitted by operators, (ii) a repository with the created *monitoring units* and their configurations, and (iii) a *probe catalog* with all probes and deployable artifacts.

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

3.3.1 *Repositories*

PROBE CATALOG The *Probe Catalog* is a repository $PC = \{p_1, \dots p_n\}$ where p_i is a probe. I assume the Probe Catalog is organized in such a way there is a unique artifact that can be used in a given context, that is, given an index i and the execution constraints (e.g., the host environment that executes the probe, the time series database that must be used to store the data, etc.), there is a unique probe p, that can be used to collect i in the target environment. The execution constraints that can be used to identify the probe are not detailed here, but these are represented in the metadata associated with the available artifacts (e.g., Listing 3.1) and matched for equality (or inclusion in case of lists) by the framework to select the probes.

Listing 3.1: A metadata excerpt from an HTTP health check probe entry in the Probe Catalog.

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

Complex matching procedures can be also implemented in the catalog if needed, such as the possibility to have multiple probes suitable

for a same context, and a decision procedure that can choose among them. Defining algorithms to choose among multiple probe artifacts is however out of the scope of the presented contribution and the framework simply requires the operator to populate the Probe Catalog with one usable artifact per execution context that must be addressed with the framework.

To illustrate the matching procedure, consider the case of OP-A asking to collect user session data from PostgreSQL. Let us assume the system considered in the running example runs on Kubernetes and that Elasticsearch is used as time-series database. In this context, the monitoring system will check the Probe Catalog looking for a probe whose metadata specify the capability to (a) collect user session data from PostgreSQL, (b) to run within containers, and (c) to store data in Elasticsearch. The monitoring system is configured with information about the environment (e.g., how to access Elasticsearch and Kubernetes APIs) to be able to configure the probes once deployed. If a matching entry is found, the corresponding artifacts are selected, and then deployed in a container, as illustrated later in this section. Otherwise, the request is aborted and the Probe Catalog has to be extended to support new probes, as described in Section 3.5.

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

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

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

A Monitoring Units Repository *MUR* stores tuples (t, mu, dc) that associate a target t with a monitoring unit mu running probes that

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

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

3.3.2 *API Service*

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

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

The *private API* is used by the Monitoring Claim Controller and Monitoring Unit Controller to handle (i.e., to read and update) the status information about both the monitoring claims and the monitoring units, as described in Sections 3.3.3 and 3.3.4.

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

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

3.3.3 *Monitoring Claim Controller*

The main responsibility of the Monitoring Claim Controller is to manage the life-cycle of the submitted monitoring claims by assign-

Algorithm 1 Monitoring Claim Controller

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

```
1: P ← APIService.getProbeConfigs(I, t)
2: if P = ∅ then return
3: end if

4: if mus=multi-probe then
5:     UpdateConfUnit(P, op, t, mus)
6: else if mus=single-probe then
7:     for p_conf ∈ P do
8:         UpdateConfUnit({p_conf}, op, t, mus)
9:     end for
10: end if

11: procedure UpdateConfUnit(Set of probe configurations P, operator op,
    target t, monitoring unit strategy mus)
12:     unit ← APIService.getMonitoringUnit(t, mus, P)
13:     if unit = ∅ then
14:         unit ← APIService.createEmpyMonitoringUnit(t)
15:     end if
16:     APIService.updateDesiredConf(unit, conf_d(unit)|_op ∪ P)
17: end procedure
```

ing the desired configurations, derived from the received claims, with the monitoring units. In particular, every time a monitoring request is received by the API Service, the API Service stores the monitoring claims included in the request in the dedicated repository and sends a status update message to the Monitoring Claim Controller, which will incrementally process them.

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

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

The monitoring units are reconfigured differently depending on the monitoring strategy. If the *multi-probe monitoring unit strategy is used*, the UpdateConfUnit procedure is invoked to associate a single monitoring unit with a desired configuration that includes all the probes (line 5). If the *single-probe monitoring unit strategy is used*, the individual probes configurations are extracted and then used to update the configuration of different monitoring units (lines 7-8).

The way a set of probe configurations are associated with a monitoring unit is defined in the UPDATECONFUNIT procedure. To identify the monitoring unit that must be updated, the controller queries, through the API Service, the monitoring units repository for an existing monitoring unit (line 12). If the *multi-probe monitoring unit strategy* is used, units can conveniently run multiple probes for a same target. In this case, the service looks for any monitoring unit created to observe t, that is, it looks for an entry $unit = (t, \textit{multi-probe}, \cdot)$, where t is the target reported in the monitoring claim. If the *single-probe monitoring unit strategy* is used, P can only include a single probe, and the API service looks for a monitoring unit that is already using the selected probe to monitor the target t, that is, it looks for an entry $unit = (t, \textit{single-probe}, (p, \cdot, \cdot))$.

In both cases, if the unit does not exist, a new unit with an empty desired configuration is created for the target t (line 14). Finally, the existing entry (i.e., the existing desired configuration) is updated by replacing the probes associated with operator op with the new ones specified in P (if the existing configuration is empty, P is simply used).

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

The first invocation with probe p_{cpu} leads to the identification of a running unit that is already collecting CPU_CONSUMPTION from PostgreSQL for OP-B (line 12). The current configuration of the retrieved unit is {p_{cpu}, CPU_CONSUMPTION, OP-B)}. The framework finally updates the desired configuration of the unit by replacing the probe configurations of operator OP-A (none in this case) with the input configuration (p_{cpu}, CPU_CONSUMPTION, OP-A), finally obtaining the desired configuration {(p_{cpu}, CPU_CONSUMPTION, OP-B), (p_{cpu}, CPU_CONSUMPTION, OP-A)}.

The second invocation with probe $p_{session}$ returns no unit that is already running that probe. Thus, a new unit is created (line 14), and the desired configuration {($p_{session}$, USER_SESSION_DATA, OP-A)} is associated with the unit.

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

3.3.4 *Monitoring Unit Controller*

Algorithm 2 Monitoring Unit Controller

Require: a monitoring unit mu
Require: its current configuration $conf_c(mu) = \{(p, I, op)\}$
Require: its desired configuration $conf_d(mu) = \{(p', I', op')\}$
Ensure: the unit is updated according to the desired configuration is generated

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

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

The operations performed by a Monitoring Unit Controller are shown in Algorithm 2. It first checks if the desired configuration is empty, in such a case the entire monitoring unit is dismissed (line 1). This is an important step to avoid running phantom monitoring units with no running probes. It then computes the diff between the current and desired configuration, identifying the probes to be added (line 3), the probes to be reconfigured to collect a different set of indicators (line 4), and the probes to be dropped (line 5). If any of these sets is non empty, the Cloud Bridge receives the probe configurations corresponding to the changes that must be actuated (line 7). Passing all the changes to be actuated at once enables the Cloud Bridge to potentially optimize how these changes are actuated.

The Cloud Bridge returns a result that specifies the errors experienced during the update process, if any. This information is used to update the current and desired configuration. In case no error is experienced, the desired configuration simply replaces the current config-

uration (line 11). Otherwise, the update process takes the errors into consideration. Error handling is described in Section 3.4.

Let us consider the case of the two desired configurations generated by operator OP-A when asking to collect two more indicators (CPU consumption and user session data) from PostgreSQL with the single-probe monitoring unit strategy, as discussed at the end of Section 3.3.3. The desired configuration related to the already deployed probe p_{cpu} results in no changes to be operated ($P_{add} \cup P_{update} \cup P_{drop} = \varnothing$), since the existing probe will be simply shared between the two operators (this is achieved by only updating the configurations in UPDATECONFIGURATION without touching the running probes). While, the desired configuration related to the new probe $p_{session}$ to be deployed results in a probe to be added ($P_{add} \neq \varnothing$).

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

3.3.5 *Cloud Bridge*

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

In particular, the Cloud Bridge exploits a plug-in based architecture that can be extended to support additional cloud systems. A plug-in for a target environment (e.g., Kubernetes) is used to map each change requested by controllers into a concrete command for the specific management API (e.g., the Kubernetes API) or the specific configuration management tool used to interact with the platform (e.g., Ansible [124]). This approach encapsulates the technological details inside the plug-in, keeping the whole control-plane framework agnostic from technology. Once all the changes have been actuated, the list of probes resulting in an erroneous state is sent back to the controller.

3.4 ERROR HANDLING CAPABILITIES

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

Soft Errors *(Definition 3.6)*. Soft errors indicate problems in the operations performed *while preparing* for the creation, update and deletion of a unit, such as retrieving probes and preparing their configuration. All these operations are performed *before* modifying any existing monitoring unit. Since those are problems that do not compromise the dependability of the running units, they are considered soft errors that have negligible consequences on the running monitoring system.

Hard Errors *(Definition 3.7)*. Hard errors indicate problems in the operations performed while *changing a running monitoring unit*, such as adding, reconfiguring or removing probes. Since these problems may compromise the dependability of the running monitoring system, they are considered hard errors that timely require corrective actions to be managed.

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

Failed Probe *(Definition 3.8)*. A probe is failed when a soft error has been detected by the Cloud Bridge while preparing the probe.

Broken Probe *(Definition 3.9)*. A probe is broken when a hard error has been detected by the Cloud Bridge while deploying/undeploying the probe.

Stable Probe *(Definition 3.10)*. A probe is stable when no error is detected.

The errors detected for each probe configuration that is processed by the Cloud Bridge are reported in the results returned to the Monitoring Unit Controller (line 7 of Algorithm 2).

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

Stable Unit *(Definition 3.11)*. A monitoring unit is stable when no error is detected for the probes in the monitoring unit.

Unsound Unit *(Definition 3.12)*. A monitoring unit is unsound when there is at least a failed probe and no broken probe in the monitoring unit. This state indicates a failure in the attempt to align the desired and current configurations of the monitoring unit, but no actual problem is affecting the running unit.

Dirty Unit *(Definition 3.13)*. A monitoring unit is dirty when there is at least a broken probe in the monitoring unit. This state indicates that the software running in the unit might be compromised.

Algorithm 3 UpdateConfiguration

Require: a monitoring unit mu to be updated
Require: $res = (Pconf_{soft}, Pconf_{hard})$, where $Pconf_{soft}$ and $Pconf_{hard}$ are the set of probe configurations that resulted in soft or hard errors
Require: $RetryTable \subseteq MUnits \times ProbeConfigs \times \mathbb{N}$, which is a table that counts how many times a given probe configuration has been retried in a monitoring unit
Require: $BlackList \subseteq MUnits \times ProbeConfigs$, which is a table that tracks the probe configurations that cause errors and should not be retried again
Ensure: mu is updated and any error is reported
1: **for** $pc \in Pconf_{soft}$ **do**
2: RetryTable.IncRetry(mu, pc)
3: **end for**
4: **for** $pc \in Pconf_{hard}$ **do**
5: BlackList.add(mu, pc)
6: **end for**
7: **if** $Pconf_{hard} \neq \varnothing$ **then** ▷ Dirty unit
8: Bridge.cleanUnit(mu)
9: $conf_c(mu) \leftarrow \varnothing$
10: **else**
11: $conf_c(mu) \leftarrow conf_d(mu) \setminus (Pconf_{soft} \cup Pconf_{hard})$ ▷ $conf_d(mu)$ is unchanged, so probe configs causing soft errors are retried, while probe configs with too many retries and probe configs in blacklist are automatically ignored
12: **end if**

Errors are mostly handled in the context of the UpdateConfiguration procedure whose pseudocode is shown in Algorithm 3. The UpdateConfiguration procedure is invoked by the Monitoring Unit Controller to finalize the update of a monitoring unit (line 11 in Algorithm 2).

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

In practice, the error handling routine first increases the number of retries for the probe configurations that caused soft failures (line 2) and adds to the blacklist the probe configurations that caused hard errors (line 5). When the number of retries exceeds an operator-defined threshold, the configuration is blacklisted.

If at least a hard error has been detected, the unit is *dirty* and thus the bridge is asked to clean it. This operation depends on the target

environment and the implementation of the plug-in used in the Cloud Bridge. For instance, in the implementation for containers, the bridge destroys the existing container and creates a new monitoring unit to replace it. The current configuration of the newly created monitoring unit is consequently set to the empty configuration.

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

This process may lead to three main distinct situations:

- *the current and desired configurations are aligned*: no changes will be performed on the monitoring unit in the future, unless a new request is submitted by an operator;
- *the current and desired configuration differs only for some blacklisted configurations*: in this case again there is nothing to be done. Note that although for simplicity I have not used the blacklist when computing the set of probes to be added, reconfigured, and deleted, in reality the Monitoring Unit Controller discards the configurations that appear in the BlackList data structure when computing them (Algorithm 2, lines 3- 5)
- *there are configurations that must be retried*: in such a case the desired and current configurations do not match, and the monitoring unit controller will process them again in the next iteration of its control-loop, retrying the failed probe configurations.

The time complexity of the Algorithm 3 is linear with respect to the number of probe changes and number of errors occurred while configuring the monitoring unit. In particular, if *errors* is the number of probe configurations that resulted in soft or hard errors. The resulting time complexity is $O(pchanges + errors)$.

3.5 TECHNOLOGY AGNOSTIC DESIGN

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

Listing 3.2: A metadata excerpt from the Apache Kafka exporter probe entry in the Probe Catalog.

```
{
    "id": "5fb6337a4102891e3677b475",
    "artifactId": "kafka_exporter",
```

```
    "supportedIndicators": [
        "KAFKA_BROKERS", "..."
    ],
    "supportedDataOutputs": [
        "PROMETHEUS"
    ],
    "supportedMUStrategies": [
        "SINGLE_PROBE",
        "MULTI_PROBE"
    ]
}
```

Listing 3.3: A sample JSON representation of a Target retrieved from Microsoft Azure.

```
{
    "targetPlatform": "azure",
    "targetPlatformId": "postgres-1",
    "envType": "INACCESSIBLE_VM",
    "metadata": {
        "resourceGroup": "resource-group-vm-1",
        "ipAddress": "52.92.34.124",
        "privateIpAddress": "10.0.0.2",
        "..."
    }
}
```

3.5.1 *Incorporating New Probes*

To demonstrate the flexibility of the monitoring framework I describe how two largely different probes can be supported: a health-check probe, which queries the health status endpoint of services exploiting the HTTP protocol, and a Prometheus exporter for Apache Kafka [92], which monitors Kafka brokers resources (topics, partitions, etc.) and exposes the collected indicators as Prometheus metrics.

Adding a new probe can be done in two steps. First, the probe artifacts have to be manipulated in such a way they can be used by the Cloud Bridge. Second, the probe is added to the catalog by passing the probe metadata, which include information about where the probe can be deployed (the probe might be compatible with certain monitoring unit strategies but not with others), the supported data outputs (i.e., the database where the collected values can be stored), and the supported indicators, to the API Service. Listing 3.2 and Listing 3.1 show an excerpt of the metadata associated with the Apache Kafka Prometheus exporter and the health-check probe, respectively. Note that the configuration of the monitoring framework (e.g., the knowledge of both the available time-series database and the type of the target platform), jointly with the requests produced by the operators, allows the framework to select and deploy the right probes. In fact,

artifact ids are mapped to the concrete software artifacts and scripts that are executed for probe deployment.

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

3.5.2 *Supporting New Target Cloud Platforms*

Supporting multiple target cloud platforms is another capability of the framework. A platform can be supported only if it provides a management API that can be used by the Cloud Bridge to manage the monitoring units and discover targets. Developers who want to create a new Cloud Bridge plug-in have to implement the base interface in order to run execution plans and provide information about the targets to the framework. Listing 3.3 shows an example of target information that can be retrieved by the API via the Cloud Bridge component. Plug-ins are also associated with metadata (e.g., the supported monitoring unit strategies) that can help the framework in taking some decisions.

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

3.6 EMPIRICAL EVALUATION

This section quantitatively evaluates the efficiency of the proposed MaaS framework with respect to both probe deployments and error-handling routines. Further, it studies the scalability for an increasing number of requests. I discuss the sub-research questions (Section 3.6.1), the implemented prototype (Section 3.6.2), the results of the experiments to answer the sub-research questions (Section 3.6.3, Section 3.6.4, and Section 3.6.5), and the threats to validity of the evaluation (Section 3.6.6).

3.6.1 *Research Questions*

This work responds to RQ1 and is assessed with the following three sub-research questions that capture the two representative capabilities of the proposed framework (i.e., probe deployment and error handling), and study how the it scales with an increasing number of requests and operators.

1. **RQ1.1 - *Framework Efficiency*: How efficiently are probes deployed?** This research question validates the framework capability of deploying probes starting from a declarative input and investigates how efficiently (i.e., in terms of time) it is in fulfilling an operator monitoring request. This is investigated for both cloud systems based on containers and virtual machines giving evidence of the technology-agnostic capabilities of the framework. Results are studied in comparison to a solution working with pre-deployed probes that can be activated/deactivated (Figure 2.2, cases (d) and (e)). To this end, I selected JCatascopia [238], which is consistent with the MaaS case shown in Figure 2.2 (d), and it is usable with no restrictions being an open-source research prototype.

2. **RQ1.2 - *Error Handling*: How efficiently are errors handled?** This research question validates the framework capability of detecting and recovering from errors and investigates the time required by the framework to execute the error handling routine.

3. **RQ1.3 - *Scalability*: How does the framework scale for an increasing number of requests?** This research question validates the framework capability of optimizing the control-plane during the evolution of the monitoring system. It studies scalability with respect to the number of requests produced by operators.

All RQs were addressed with cloud systems based on both virtual machines and containers. In the following, I describe the prototype used to run experiments, the design of the study, and the results for each research question.

3.6.2 *Prototype*

I implemented the MaaS framework described in this chapter in a publicly available prototype hosted at `https://gitlab.com/learnERC/varys`.

The services are implemented as Java standalone applications. The repositories are implemented as MongoDB [122] collections. The JSON format is used both for communication and to persist information, except for the Cloud Bridge which exposes a gRPC API that uses Protocol Buffers. The status update messages are delivered through Redis Streams [68]. The monitoring system can be deployed both on container-based and VM-based technologies, depending on the hosting environment.

I designed a probe catalog reusing probes from Metricbeat [33], a popular monitoring solution part of ElasticStack [31]. The prototype uses Elasticsearch [32], as ingestion service to store the values extracted by the probes. I implemented plug-ins for the Cloud Bridge to support both Kubernetes and Microsoft Azure as target cloud platforms.

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

With respect to container monitoring units, the Kubernetes plug-in deploys container monitoring units in the same platform of the target and configures the probes accordingly. Purposely, it does not implement the container sidecar pattern [50] because it would trigger the redeployment of the target service, due to how Pods work in Kubernetes, every time probes are (un)deployed, potentially causing service or monitoring interruptions unless a robust rolling update strategy is in place.

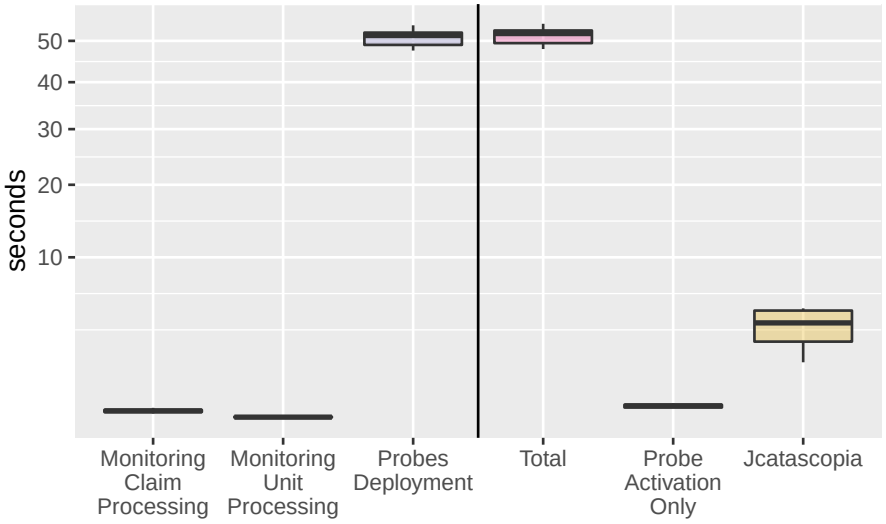
3.6.3 *RQ1.1: Framework Efficiency*

The monitoring framework can work in parallel on any number of monitored targets, if enough instances of the monitoring unit controller service are created. If there are more targets to modify than controller instances, some modifications will be performed sequentially. For instance, if 4 monitoring units must be modified and only 3 controller instances are available, one unit will be modified sequentially after another one. I will thus study how efficiently a monitoring unit can be managed by a single controller instance, the performance over multiple simultaneously evolving units can be straightforwardly deduced given the number of controllers available.

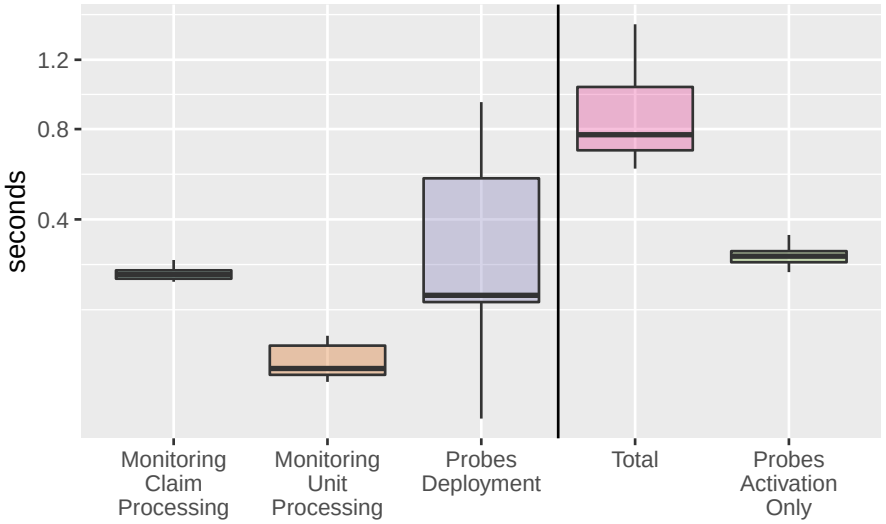
Two cases are considered for the experiments: PostgreSQL 9.5 running in a container in an on-premise installation of Kubernetes and PostgreSQL 9.5 running in a virtual machine on Microsoft Azure (Standard B2s flavor, 2 vCPU, 4 GB of RAM, OS Debian 10). The on-premise Kubernetes installation is run by a Minikube VM with 4 vCPU and 8 GB of RAM executed on a MacBook Pro 2017 (3.1 GHz Quad-Core Intel Core i7, 16 GB of RAM). The two cases show how the same framework can be transparently used to address heterogeneous scenarios where the involved technologies are significantly different. I collect time figures considering the case of a request that requires the simultaneous deployment of three probes to collect the following three indicators from PostgreSQL: CPU consumption (using the CPU metricset of the Metricbeat probe), memory consumption (using the memory metricset of the Metricbeat probe), and database metrics (using the database metricset of the Metricbeat probe).

To study the efficiency of each step, I measure the time taken by the first controller to process the claim, by the second controller to compute the execution plan, and by the Cloud Bridge to actuate the plan. To have a baseline measurement, I also consider the case of a

static framework, that is, a framework that does not support dynamic probe deployment, requiring operators to *deploy and configure probes in-advance*, which can be later activated and de-activated. This framework is *far less flexible* than the framework presented in this chapter, but faster since it does not deploy probes dynamically. To this end, I both use the framework with pre-deployed probes and the JCatascopia [238] state of the art monitoring solution, which allows us to collect further measurements from a third party system. I do not have measurements for JCatascopia applied to containers since it only supports virtual machines. Every experiment is repeated 10 times to collect stable measurements.



(a) Virtual Machines running on Microsoft Azure.



(b) Containers running on a local Kubernetes cluster.

Figure 3.2: Probe deployment time figures.

RESULTS OF RQ1.1 Figure 3.2 shows the collected time figures with a semi-logarithmic scale considering both virtual machines (Figure 3.2a) and containers (Figure 3.2b). The individual steps of the probe deployment process are captured by the Monitoring Claim Processing, Monitoring Unit Processing and Probes Deployment boxes. While Total represents the total time elapsed between the submission of the request and the time the deployed probes start collecting the required indicators.

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

Overall, the entire probe deployment process of the three probes (indicated with Total in Figure 3.2) could be completed in slightly less than a minute using virtual machines and less than 1.5s using containers, which is a nearly two orders of magnitude difference.

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

ANSWER TO RQ1.1 In the case of VMs, the cost of flexibly deploying probes is significantly higher than working with pre-deployed probes. Thus, the trade-off between flexibility and efficiency should be carefully considered by operators to decide the monitoring solution that should be adopted. Instead, in the case of containers, the cost of flexibly deploying probes is significantly amortized by the efficiency of the cloud technology.

3.6.4 *RQ1.2: Error Handling*

To study the capability of the framework to react to errors, I designed a variant of the experiment performed for RQ1.1 where I deploy a malfunctioning probe. I obtained such a probe by implementing a wrong configuration of the Metricbeat probe for PostgreSQL that makes the probe deployment to fail.

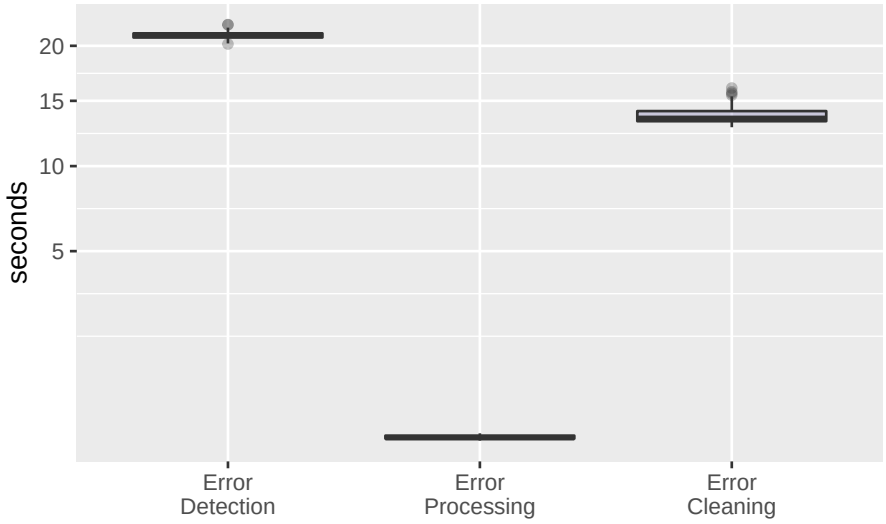
In the case of virtual machines, I study the creation of a new monitoring unit with two probes: one working probe and a malfunctioning probe. The malfunctioning probe artifact contains an Ansible role with a wrong command that leads to a hard deployment error when the Cloud Bridge executes it. Since I use the multi-probe monitoring unit strategy with virtual machines, error detection must autonomously detect the problem with the monitoring unit with two probes and automatically create a monitoring unit with the working probe only. The VM used as monitoring unit is created with a Standard B1s flavor (1 vCPU, 1 GB of RAM, OS Debian 10).

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

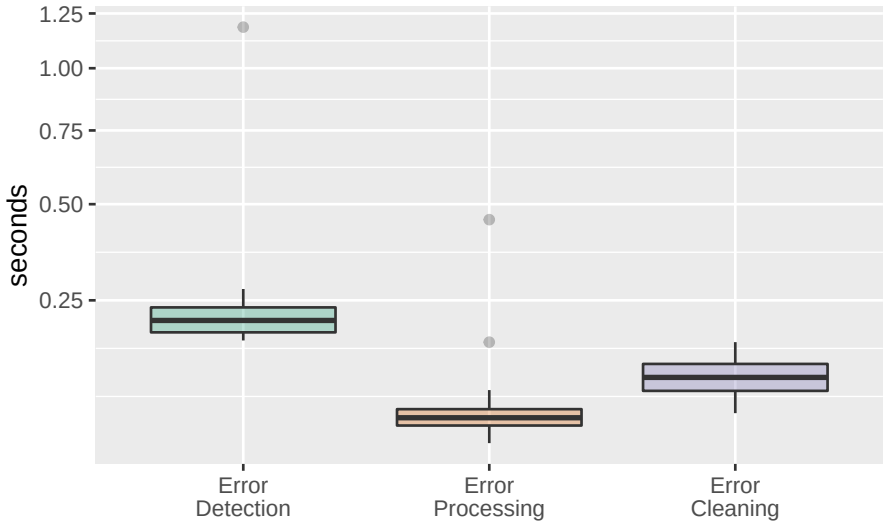
To capture how error detection works, I measure the time necessary to the framework to *attempt the deployment and detect* that a monitoring unit is not working (namely Error Detection), the time necessary to *process* the error and take the decision to clean the monitoring unit (namely Error Processing), and finally the time necessary to actuate the cleaning plan (namely Error Cleaning). Error detection is performed by the Cloud Bridge while actuating changes (see the call in Algorithm 2, line 7), error processing consists of the operations shown in Algorithm 3, and error cleaning is again performed by the Cloud Bridge when cleaning a unit (see the call in Algorithm 3 line 8). I repeated measurements 10 times to collect stable time figures.

RESULTS OF RQ1.2 Figure 3.3 shows the collected time figures with a semi-logarithmic scale considering both virtual machines (Figure 3.3a) and containers (Figure 3.3b).

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



(a) Virtual machines running on Microsoft Azure.



(b) Containers running on a local Kubernetes cluster.

Figure 3.3: Error handling time figures.

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

ANSWER TO RQ1.2 Results show how the proposed MaaS solution that flexibly allocates and destroys resources, although usable with both VMs and containers, are naturally more suitable for containers where errors can be recovered in seconds.

3.6.5 *RQ1.3: Scalability*

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

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

RESULTS OF RQ1.3 Figure 3.4 shows the results. Again, the remarkable difference between virtual machines and containers is confirmed. The scalability experiment gives additional evidence of how the linear growth of the total time for virtual machines is far more steep than containers. The difference is dramatic when considering the deployment of 30 probes, which requires around 10 minutes, in contrast with containers that can complete this operations in seconds.

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

ANSWER TO RQ1.3 Overall, results show that dynamic probe deployment can be feasible with both virtual machines and containers. However, the former environment can efficiently deal with probes only if changes are sporadic and the number of parallel requests received is limited. On the contrary, the container technology is definitely ready to support dynamic probe deployment, even in rapidly evolving contexts, based on the proposed framework.

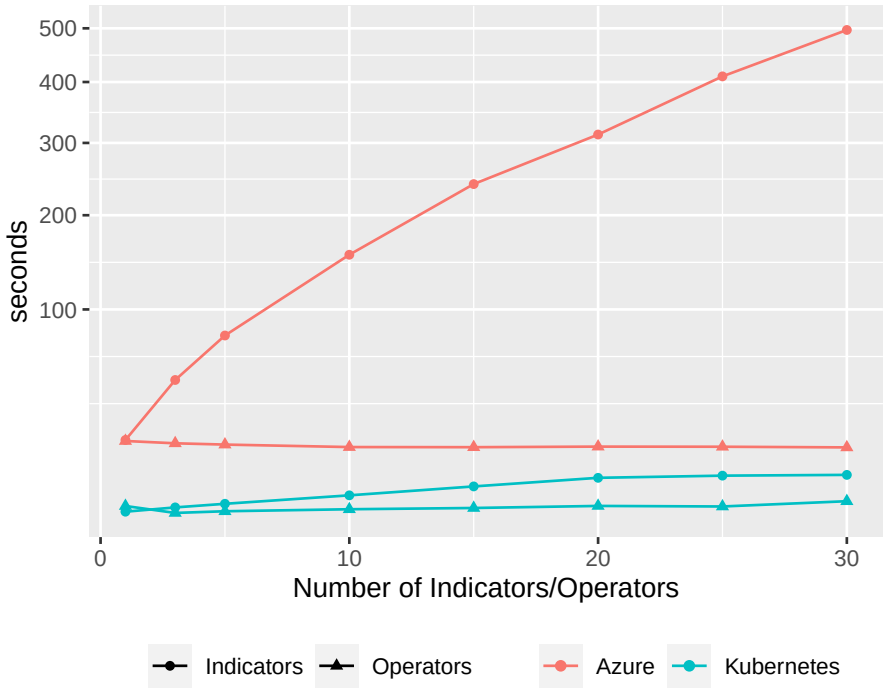


Figure 3.4: Time to fulfilling monitoring requests for a increasing number of indicators (dot markers) and operators (triangle markers), with both VMs running in Microsoft Azure (red lines) and on containers running in a local Kubernetes cluster (light blue lines).

3.6.6 *Threats to Validity*

The threats to the validity of the results mainly concern with the relationship between the technical setup of the experiment and the collected time figures. In fact, efficiency is affected by both the available computational resources and the choice of the probes, For example, the indicator type, and so its probe artifact implementation, used in the experiment performed to study RQ1.3 can affect the collected values. However, while changing the available computational resources and the deployed probes are likely to affect absolute figures, the trends and gaps between the different frameworks and cloud platforms are clear, despite these factors. In fact, plots for virtual machines and containers are similar, although values are on different scales. Further, the scalability trends clearly identify a single case (collecting increasingly more indicators on virtual machines) that scales remarkably worst than the others.

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

Finally, the collected time figures might be affected by noise. To mitigate this issue I repeated experiments between 5 and 10 times. Although the statistical significance and effect size of the collected data have not been computed, the reported box plots show a low variance for the collected values, suggesting that measures are stable and meaningful, and can be used to derive valid conclusions.

3.7 DISCUSSION

The proposed framework is capable of fully automating the deployment and undeployment of arbitrary probes starting from declarative inputs (i.e., the list of indicators to be collected) entered by the operators, thus supporting automated evolution of the monitoring system to adapt to changing requirements. Moreover, it embeds routines to handle deployment errors, because error handling capabilities received little attention so far, with approaches mostly focusing on error-free deployment scenarios or relying on the direct intervention of operators, despite the importance of error handling for long-running systems, such as a monitoring system. Lastly, the presented framework can be integrated with existing technologies (e.g., probe technology, ingestion engines, and target environments) without the need of using ad-hoc solutions. Results show that the framework can be feasibly used with cloud systems based on both virtual machines and containers, although it is significantly more efficient with containers. In the latter case, the proposed framework can complete the probe deployment process in 0.5-1.5 seconds, while activating the pre-deployed probes requires slightly less than 0.5s, suggesting that dynamic probe deployment might be often preferable for containers, while working with pre-deployed probes might be helpful for VMs. Regarding the error-handling capability studied by RQ1.2, the entire procedure can be completed in around 35 seconds with VMs and less than a second with containers. Finally, the results about the framework's scalability investigated by RQ1.3 give additional evidence of how sharing probes between operators can significantly improve the monitoring system efficiency. This is crucial for virtual machines where the probe deployment cost can be reduced by sharing probes. In fact, the difference is particularly relevant when considering the deployment of 30 probes, which requires around 10 minutes, in contrast with containers that can complete this operations in few seconds.

The current framework implementation has three main limitations. First, fine-grained control of the probes configurations (e.g., changing the sampling rate of the individual probes) is not supported. This limitation can be potentially addressed by enriching monitoring claims with information about probe configurations. Second, the support to elasticity right now depends on the probe intelligence (e.g., it requires the probes to embed a discovery mechanism as the one in the Metricbeat Kubernetes module). It would be interesting to move this sup-

port at the level of the monitoring framework, so that any probe can be used to monitor elastic services. Third, error-handling is limited to the deployment phase, and it is unable to detect and repair run-time errors that occur during the regular execution of the monitoring system. Indeed, error handling requires further research to cover the full range of situations.

4 PATTERNS FOR PROBE DEPLOYMENTS

This chapter presents the definition, analysis, and both qualitative and quantitative evaluation of 11 possible probe deployment patterns. They are specified by means of a feature model that captures several key features and logical constraints relevant to probe deployment. The proposed patterns help in understanding how to distribute the probes in order to respect and optimize both technological and operators' requirements, as discussed in Section 2.2.2. Results show trade-offs between patterns that require more resources to ensure good separation between users in multi-tenant environments, and those patterns that use less resources, while reducing the degree of separation at a cost of less privacy and risk of interference. In addition, I distilled a set of best practices from the findings to guide engineers in implementing and configuring their monitoring systems. Finally, the results have been cross-validated by addressing three realistic monitoring scenarios involving different technologies, software architectures, and monitoring requirements. The contribution presented in this chapter as been published in the IEEE Transactions on Services Computing journal paper titled "Monitoring Probe Deployment Patterns for Cloud-Native Applications: Definition and Empirical Assessment" [244].

The chapter is structured as follows. Section 4.1 briefly introduces what probe deployment is and why it is a relevant problem for monitoring systems. Section 4.2 presents and defines the probe deployment patterns. Section 4.3 provides a qualitative evaluation of the patterns. Section 4.4 presents the empirical evaluation. Section 4.5 discusses the best practices derived from the empirical findings. Section 4.6 demonstrates the application of probe deployment patterns to three realistic usage scenarios. Finally, Section 4.7 concludes the chapter with closing remarks.

4.1 PROBE DEPLOYMENT

Monitoring systems can deploy probes using various strategies that take into account features such as technology, efficiency, and privacy. The previous chapter illustrated how the proposed MaaS framework uses two different strategies to deploy probes based on the technology used to host the probe artifacts, namely VMs and containers. For further illustration, let us consider a multi-tenant environment. In this case, multiple probes can be deployed within the same virtual machine to save resources. However, this comes at the cost of reduced privacy and security. Alternatively, a single probe can be deployed per container or virtual machine to preserve privacy, but this requires

more resources allocated to the monitoring system. Therefore, the effectiveness and efficiency of a monitoring system can be impacted by the chosen probe deployment strategy.

Despite the many possible strategies, there has been no systematic analysis and assessment. Engineers designing monitoring systems are left to make their own decisions, which can have implications on the monitoring efficiency and the flexibility of the monitoring system. The next section introduces the concept of probe deployment patterns and defines a set of key features for probe deployment that led to the definition of 11 patterns.

4.2 PATTERN DEFINITIONS

Probe deployment patterns capture how probes can be deployed to monitor the targets. The possible deployment depends on several key features that I discuss in this section and are represented with the feature diagram shown in Figure 4.1.

A feature diagram is a graphical representation of a feature model that defines features and their dependencies in a tree structure [137]. 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 the defined diagram. A combination of features is a configuration. A configuration is admissible if it satisfies its feature diagram.

A feature model 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. The 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.

I started from the papers that propose monitoring and probe deployment approaches for multi-tenant and technology-heterogeneous cloud environments [2, 50, 115, 173, 191, 235], the most used cloud monitoring tools [25, 30, 151, 177, 195], and my personal experience, to identify and distill a set of relevant features for probes deployment.

A feature model can be used to automatically generate the space of all the admissible configurations. In fact, the configurations admitted by the model obtained by the diagram in Figure 4.1 represent all the probe deployment patterns that can be identified by using the consid-

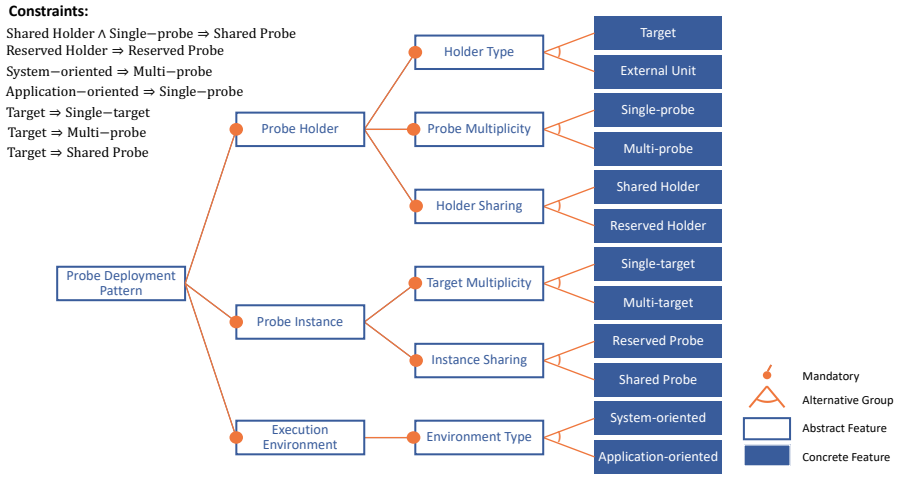


Figure 4.1: Probe deployment patterns feature diagram.

ered features and constraints. The semantics of the considered features are discussed below.

- **Probe Holder**: it represents the objects that hosts probe instances (N.B., hereinafter referred as holder). It can be a separate Virtual Machine or container, or can overlap with the target execution environment. With respect to the monitoring unit concept (Definition 3.4) defined in Chapter 3, the probe holder is the solely hosting object without any running probe.
  - *Holder Type*: it represents the holder type [173].
    - 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 [25, 30, 50, 151, 173, 177, 195]).
  - *Probe Multiplicity*: it defines the number of probes that can be executed within the unit [50].
    - 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 [2, 115, 191, 235].
    - Reserved Holder: the holder is reserved to a single user.
    - Shared Holder: the holder can be shared among users.
- **Probe Instance**: it represents the running instance of a probe artifact (Definition 3.1) executed within the holder to collect data.
  - *Target Multiplicity*: it defines the number of targets that a single probe can monitor simultaneously [25, 30, 50, 151, 177, 195].

    * 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 [2, 115, 191, 235].
    * 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*:
    * 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 [70], OpenVz [93], and Linux-VServer [71].
    * 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 [121] or containerd [94].

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 [157, 265]. 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 [50],

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* ⇒ *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 [73, 191], 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 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) indicators, 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.

All these concepts and constraints are encoded in the feature diagram in Figure 4.1, which has been implemented using FeatureIDE [234], a tool for feature-oriented software development based on Eclipse. All the admissible configurations from the model are derived 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, I 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.

Figure 4.2 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,

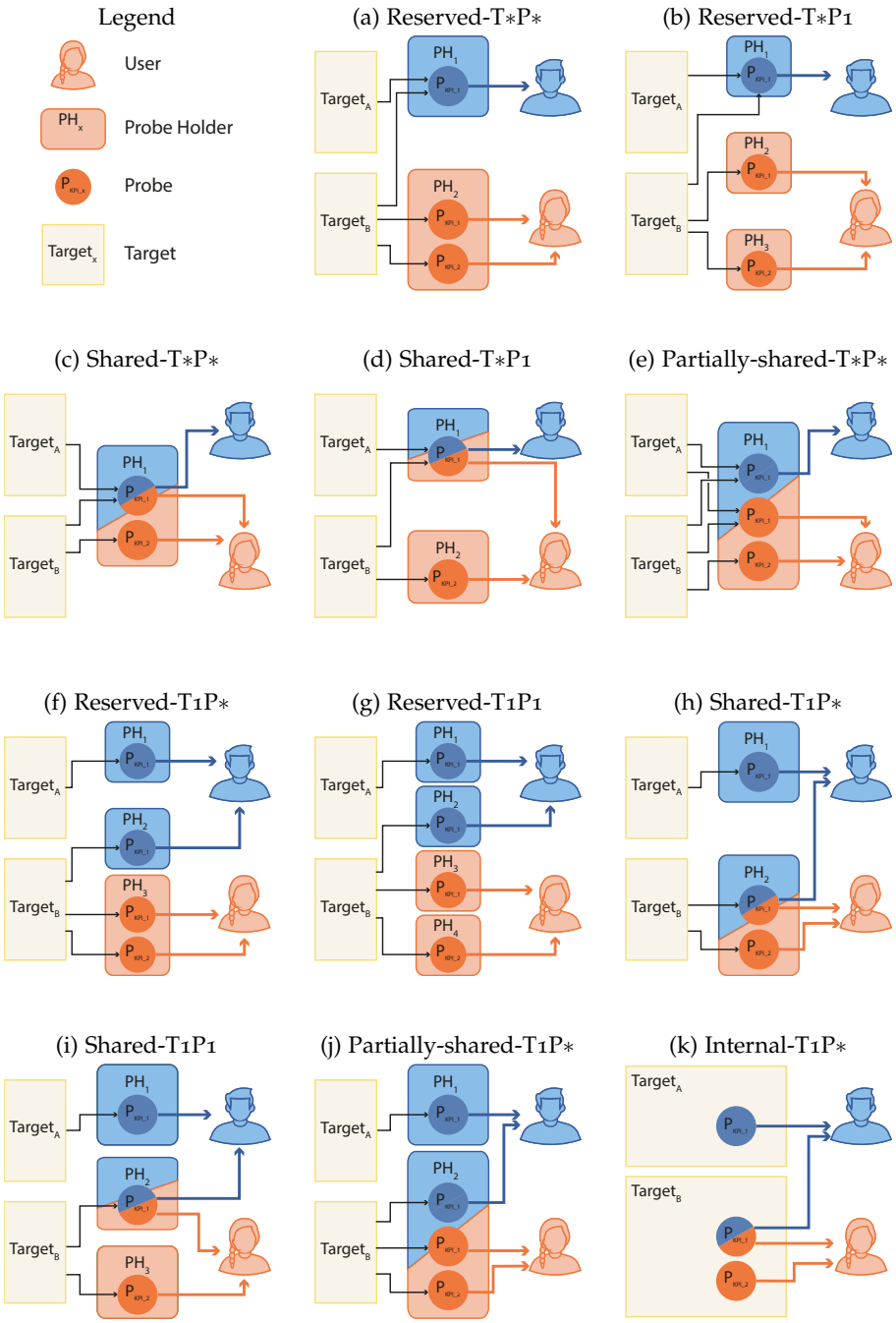


Figure 4.2: Probe deployment patterns.

which is sufficient to exemplify the differences among the various patterns. The number of holders and probes varies according to the configuration. I use colors to represent ownership (a probe holder or a probe of the same color of a user indicates the ownership of the user, while multicolored elements represent shared resources).

I adopt a same schema to illustrate each pattern. In particular, I 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.

A naming convention is defined 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. I use *T** for probes that can monitor multiple targets, while I use *T1* for probes that monitor a single target.
- The third element represents the probe multiplicity. I use *P1* for holders that run a single probe. While I 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.

Reserved-T∗P∗ (Figure 4.2a) *(Definition 4.1)*. Reserved-T∗P∗ pattern 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 indicator from multiple targets.
Target Technology: Since each holder can host multiple probes, this pattern is usually applied to system-oriented virtualization technologies.

Reserved-T∗P1 (Figure 4.2b) *(Definition 4.2)*. Reserved-T∗P1 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.
Target Technology: 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∗ (Figure 4.2c) *(Definition 4.3)*. Shared-T∗P∗ 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 indicators from multiple targets.
Target Technology: Due to the presence of multiple processes within the same holder, this pattern is specific to system-oriented environments.

Shared-T∗P1 (Figure 4.2d *(Definition 4.4).*] Shared-T∗P1 shares holders among users and they contain a single probe that is able to acquire data from multiple targets and for multiple users, if needed.
Target Technology: 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∗ (Figure 4.2e) *(Definition 4.5).* Partially-shared-T∗P∗ uses a single holder, shared among users that contains multiple probes able to acquire the indicators from multiple targets. In case the same indicator is requested by multiple users, the probe is instanced multiple times within the same holder, one for each user that requested the indicator.
Target Technology: 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∗ (Figure 4.2f) *(Definition 4.6).* Reserved-T1P∗ 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 indicators from the same target, multiple probes can be placed within the same holder.
Target Technology: Due to the fact that a holder may contain multiple probes, this pattern is suited for system-oriented environments.

Reserved-T1P1 (Figure 4.2g) *(Definition 4.7).* Reserved-T1P1 uses a reserved holder for each user and contains a single probe. Every probe is dedicated to the collection of indicators from a single target, which means that if a single user requests the same indicator from a given number of targets, an equal number of holders and probes will be deployed to fulfill such request.
Target Technology: This pattern is dedicated to application-oriented environments as it fulfills the requirement of having a single process in each holder.

Shared-T1P∗ (Figure 4.2h) *(Definition 4.8).* Shared-T1P∗ uses a holder for each target. Such holder may contain multiple probes that can acquire indicators 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.
Target Technology: Given the fact that each holder can contain multiple probes, this pattern is tailored for system-oriented environments.

Shared-T1P1 (Figure 4.2i) *(Definition 4.9)*. Shared-T1P1 uses multiple holders for each target, where each holder contains only one probe that acquires indicators from the target and shares the data among all the users.
Target Technology: 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∗ (Figure 4.2j) *(Definition 4.10)*. Partially-shared-T1P∗ 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 indicator, there will be an equal number of instances of the same probe deployed, each one dedicated to a single user.
Target Technology: 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∗ (Figure 4.2k) *(Definition 4.11)*. Internal-T1P∗ 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.
Target Technology: This pattern is specific to system-oriented environments since its nature implies multiple processes running in the target.

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 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.

4.3 QUALITATIVE DISCUSSION

This section discusses the qualitative aspects related to the presented patterns. I first discuss how the patterns can be implemented with different technologies. I then discuss the trade-off between separation and resource consumption. I conclude by discussing interoperability, portability, robustness, affordability and security of the patterns. Table 4.1 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*).

4.3.1 *Pattern Implementation*

To show the practical applicability of the patterns, in this section I 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 4.6, 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 [25] or Beats [30], such as SNMP [54, 195] or HTTP based probes. This is also the case of tools such as Zabbix [151] 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 [2, 235]. These patterns are well supported also by commercial tools, such as Nagios [177] and Dynatrace [82], and in scientific articles, such as [115, 191].

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 [82, 151, 177], and in scientific articles, such as [51, 116, 167, 220, 227]

The *Internal-T1P∗* pattern can be found in many agent-based solutions [11, 164, 238]. 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 [192, 238]. For instance, the MaaS framework proposed in Chapter 3 that has the ability to automatically govern the entire life-cycle of the probes from declarative inputs, thus relieving operators of any configuration burden.

4.3.2 *Analysis of Quality Aspects*

Table 4.1: 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

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 4.1, 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 4.4.

INTEROPERABILITY AND PORTABILITY The proposed patterns are cloud agnostic and thus are interoperable and portable across cloud environments [76]. 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 I 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.

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∗).

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.

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.

4.4 EMPIRICAL EVALUATION

In this section, I 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. I discuss the sub-research questions (Section 4.4.1), the experimental plan that was carried out and the experimental setup that I used to perform the experiments (Section 4.4.2), the results of the experiments that I executed to answer the sub-research questions (Section 4.4.3 and Section 4.4.4), and threats to validity of the evaluation (Section 4.4.5).

4.4.1 *Research Questions*

This work responds to RQ2, and in addition to the qualitative discussion reported in Section 4.7, it is quantitatively assessed with the investigation of the cost-effectiveness of the probe deployment patterns. This concerns with the scalability of the pattern with the respect to 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. I 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, I generate two distinct sub-research questions for each target environment as follows.

RQ2.1 - *System-oriented Patterns Scalability*: How do patterns for *system-oriented* environments scale with the amount of monitored data?

RQ2.2 - *Application-oriented Pattern Scalability*: How do patterns for *application-oriented* environments scale with the amount of monitored data?

RQ2.1 and RQ2.2 study how probe deployment patterns scale with respect to an increasing number of users, indicators and targets for system-oriented and application-oriented execution environments, respectively.

4.4.2 *Experimental Setup*

To answer the two research questions, I 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 indicators requested per user. To measure scalability, I 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, I 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, I 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 4.2 summarizes the experiments I 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 indicators* by a single user for a given target and by two users for a same target, respectively. That is, I investigate the impact of an increasing number of indicators 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 indicator, and two users interested in collecting a same indicator. That

is, I 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 indicator 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 indicators for a same single target. That is, I study how a growing number of users impact on the patterns, also considered in combination with an increasing number of indicators 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).

Table 4.2: Experiments Configurations

Experiment	**Sequence of Exp. Configurations (Users - Targets - Indicators)**
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)

When collecting data, I run each experimental configuration 3 times for 10 minutes to collect stable results. Since I 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 I study 30 configurations, to support the 6 experiments shown in Table 4.2, and I repeat the experiments for the 11 patterns, a total of 330 configurations is obatined. I 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, I 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*).

I ran the experiments on both virtual machines (VM) and containers. To automate experiments, I implemented Ansible playbooks [124] that interact with the Azure Compute Platform [170] and with a managed Azure Kubernetes Cluster [171] 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. I deployed container holders and targets by mean of single-replica Kubernetes Deployments [232].

I used NGINX [86], 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 [32] and can be configured to collect tailored metrics. I configured the Metricbeat NGINX module to probe the target, while I 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, I run a dedicated Metricbeat instance on both the targets and holders. In Kubernetes, I deployed Metricbeat to measure the targets and holders resource consumption as a Kubernetes DaemonSet [231].

I 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.

I 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, I 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, I referred to the monthly cost of a Microsoft Azure Standard B1s VM operated in the West Europe zone (€8.18/month in July 2023), 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 in July 2023). I 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 [243].

4.4.3 *RQ2.1: System-oriented Pattern Scalability*

Figure 4.3 shows how the resource consumption growths for the various metrics, considering the system-oriented patterns implemented with VM holders. I 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. I 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 in Appendix A.

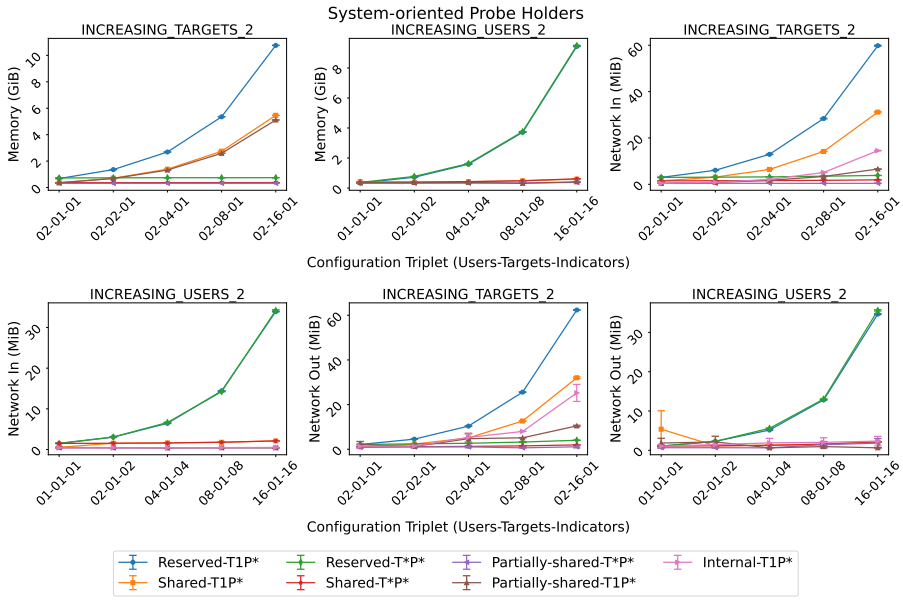


Figure 4.3: System-oriented probe holders patterns scalability.

RESULTS OF RQ2.1 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).

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 Figure 4.3 (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 Figure 4.3 (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 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. I observe in particular that both an increasing number of users requesting different indicators (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 Figure 4.3. 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 indicators.

Based on this evidence, depending on the expected scalability trend, there are patterns that should be preferred or avoided. It is however useful to remark that the overall resource network consumption that has been 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.

I report in Table 4.3 and Table 4.4 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.

Table 4.3: System-oriented patterns probe holder monthly costs for experiments INCREASING_KPIS_1, INCREASING_KPIS_2, and INCREASING_TARGETS_1

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

Table 4.4: System-oriented patterns probe holder monthly costs for experiments INCREASING_TARGETS_2, INCREASING_USERS_1, and INCREASING_USERS_2

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

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 indicators 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.

ANSWER TO RQ2.1 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.

4.4.4 *RQ2.2: Application-oriented Pattern Scalability*

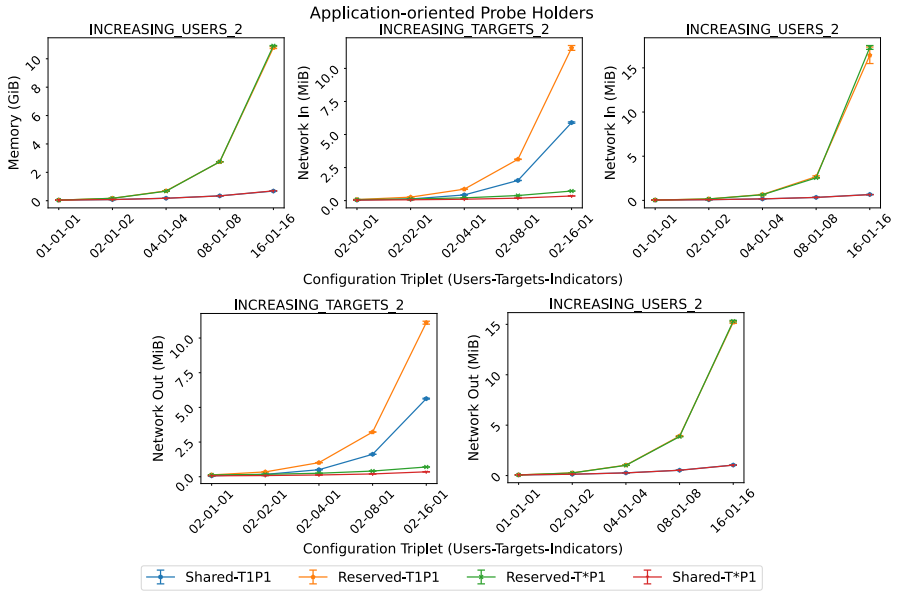


Figure 4.4: Application-oriented probe holders patterns scalability.

Figure 4.4 shows how the resource consumption growths for the various metrics, depending on the application-oriented patterns. I 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 Appendix A.

RESULTS OF RQ2.2 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 differ-

ent indicators (Figure 4.4 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 indicator to be collected. For instance, the last configuration triplet (16 Users - 1 Target - 16 Indicators) 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 is also negligible for targets, less than 1 MiB in all the experiments except for the INCREASING_USERS_2 experiment where I 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 I observed that both an increasing number of users requesting different indicators (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 Figure 4.4. 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 indicators.

Table 4.5 and Table 4.6 summarize the monthly cost of executing application-oriented holders in the experiments. I 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.

Table 4.5: Application-oriented patterns probe holder monthly costs for experiments INCREASING_KPIS_1, INCREASING_KPIS_2, and INCREASING_TARGETS_1

Pattern	**Experiment**		
	INCREASING_KPIS_1	INCREASING_KPIS_2	INCREASING_TARGETS_1
*Reserved-T*P1*	[0.1752, 2,8401]	[0.3519, 5.6497]	[0.1752, 0.1939]
Reserved-T1P1	[0.1732, 2.8022]	[0.3397, 5.7135]	[0.1732, 3.1300]
*Shared-T*P1*	[0.1755, 2.8495]	[0.1652, 2.7965]	[0.1755, 0.1980]
Shared-T1P1	[0.1814, 2.8513]	[0.1765, 2.8226]	[0.1814, 3.1658]

Table 4.6: Application-oriented patterns probe holder monthly costs for experiments INCREASING_TARGETS_2, INCREASING_USERS_1, and INCREASING_USERS_2

Pattern	Experiment		
	INCREASING_TARGETS_2	INCREASING_USERS_1	INCREASING_USERS_2
*Reserved-T*P1*	[0.3519, 0.3910]	[0.1752, 2.7764]	[[0.1752, 43.9768]
Reserved-T1P1	[0.3397, 6.2549]	[0.1732, 2.8346]	[0.1732, 43.5852]
*Shared-T*P1*	[0.1652, 0.1970]	[0.1750 0.1755]	[0.1755, 2.8446]
Shared-T1P1	[0.1765, 3.1053]	[0.1712, 0.1814]	[0.1814, 2.8193]

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 indicators 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 indicators) 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.

ANSWER TO RQ2.2 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.

4.4.5 *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.

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 the evaluation, I also selected a specific target service to be monitored (i.e., NGINX) and I used a specific probe technology (i.e., Metricbeat module for NGINX). Moreover, I deployed the same probe when experiments required to increment the number of requested indicators, and a probe is configured to collect a single indicator. In a real-world scenario probes may be configured to collect several indicators, potentially lowering the resource consumption. While using a single target application (i.e., NGINX) in the evaluation may raise concerns about the generalization of the results, it is important to remark that the monitored application was not a factor in the study. The monitored application has no impact on the cost and effectiveness of the deployment patterns. I thus intentionally used a single application in the 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 I report results about the experience with three real-world applications of the patterns in Section 4.6.

Finally, the collected resource consumption values might be affected by noise. To mitigate this issue I 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. Although the statistical significance and effect size of the collected data have not been calculated, I computed the mean and the standard deviation by all the data samples, thus, stabilizing the results to derive more solid conclusions.

4.5 BEST PRACTICES

This section discusses a distilled list of best practices for probe deployment derived from the 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 indicators increase (e.g., see experiments INCREASING_TARGETS_2 and INCREASING_USERS_2). Indeed, the case of a large number of users asking for many indicators 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 indicators to lower the consumption (see trend results for the INCREASING_USERS_2 experiment in Figure 4.3).

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 Figure 4.3 and Figure 4.4). 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 highly-dynamic indicator collection requirements. In the case of indicators 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 Figure 4.3 and in Appendix A for further details). Again, probe instances should be configured to collect multiple indicators 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 the experiments (see Figure 4.3 and the Appendix A for further details). The same cannot be usually achieved with application-oriented execution environments (e.g., due to the single main container process practice [40]). 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 [233], Podman [69]) 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.

4.6 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, I 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.

I first describe the application architecture, the technology stack, and the monitoring requirements for each scenario. Second, I discuss how patterns are selected based on monitoring requirements and probe deployment best practices. I also describe how the selected patterns would be impacted by an increase in the number of the collected indicators, the number of target instances, and the number of users interested in the collected data. Finally, I 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. I 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 [243]. In

the chapter, I report a selection of the plots, the complete set of plots is available in Appendix A.

4.6.1 *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 [107]). For each service instance, the engineers spin up a VM following the Service-as-a-VM deployment pattern [198]. 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 [31], since this monitoring service is used in several other company products.

The outsourced services expose indicators using the Prometheus format (i.e., running the node exporter [194]), so the engineers need to collect these indicators to obtain insights about the behavior of the outsourced service instances. In addition, they need to monitor the Redis database and some infrastructure indicators (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 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 indicators 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 indicators; and (ii) the system module to monitor CPU load, memory, and filesystem.

SCALING IMPACT

- *Increasing Indicators*: dealing with an increasing number of indicators requires the reconfiguration of the Metricbeat probe instances, activating new modules, or deploying new probes in the case the indicators 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.

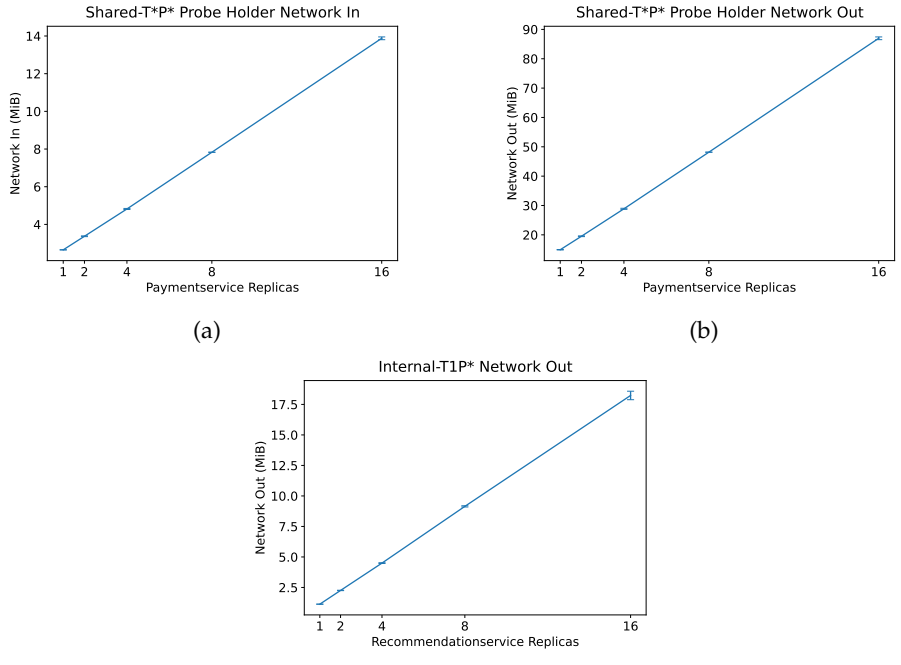


Figure 4.5: Shared-T∗P∗ pattern holder network I/O consumption and Internal-T1P∗ pattern network output consumption with respect to an increasing number of payment service and recommendation service replicas, respectively.

QUANTITATIVE EVALUATION FOR INCREASING TARGET INSTANCES

I exploit the 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. I 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 indicators 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 indicator values from the targets, and then pushing them to the Elasticsearch instance for storage. Figure 4.5a and Figure 4.5b 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). Figure 4.5c 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 4.4 for both the implemented patterns.

4.6.2 *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 [26] and the application services.

In this case, the service development teams want to monitor HTTP and gRPC indicators for their application services, and some specific indicators 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 indicators 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 indicators 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 [25] as probe instance technology, resulting in the following deployment:

- *Shared-T∗P1* pattern: a Kubernetes Pod hosting a Prometheus Blackbox exporter [193] instance to collect HTTP and gRPC indicators from the application services.
- *Reserved-T∗P1* pattern: two Kubernetes Pods hosting the Prometheus Redis exporter [100] instance configured to collect Redis indicators from all the available replicas for the database ops team and the service development teams, respectively.

SCALING IMPACT

- *Increasing Indicators*: increasing the number of indicators 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 indicators 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 collect indicators 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 I 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.

I 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

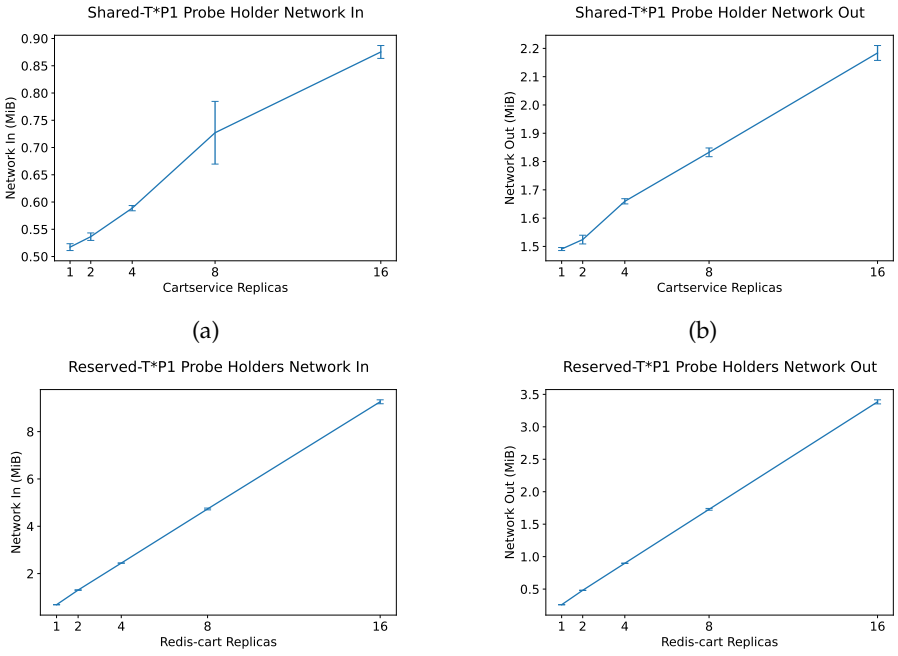


Figure 4.6: Network I/O consumption of the Shared-T∗P1 and Reserved-T∗P1 pattern holders with respect to an increasing number of cart service and Redis replicas, respectively.

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 indicator 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 Figure 4.6.

Also in this usage scenario, it is possible to observe trends consistent with the ones obtained in the controlled evaluation reported in Section 4.4.

4.6.3 *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 [101]) exploiting OpenFaaS [183] 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

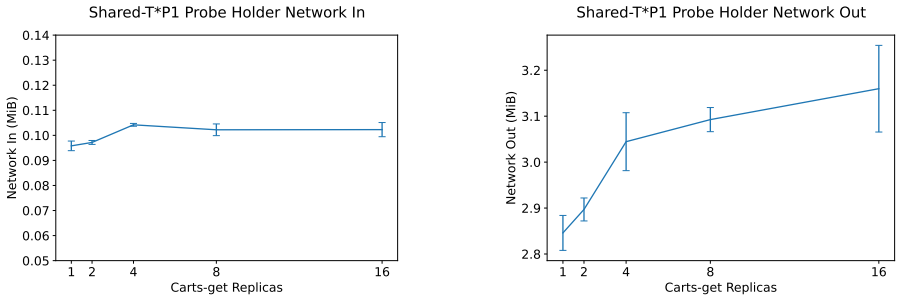
cluster nodes size. Moreover, the company has a strong knowledge on using Prometheus to monitor the Kubernetes cluster nodes and the application services.

APPLYING PROBE DEPLOYMENT PATTERNS It is possible to 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 [102] 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 indicators at container-level.

SCALING IMPACT

- *Increasing Indicators*: increasing the number of indicators requires the reconfiguration of the cAdvisor probe instances activating new indicators, or deploying new holders and instances in the case the indicators 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.



(a) (b)

Figure 4.7: Network I/O consumption of the Shared-T∗P1 pattern holders with respect to an increasing number of carts-get function replicas.

QUANTITATIVE EVALUATION FOR INCREASING TARGET INSTANCES I 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 Figure 4.7a, while on average network output consumption reaches 3.1 MiB. Network output scales linearly with an increasing number of function replicas, as shown in Figure 4.7b.

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 4.4.

4.7 DISCUSSION

The proposed patterns address the knowledge gap regarding how to efficiently distribute the probes in order to respect and optimize both technological and operators' requirements, dealing with the multi-tenancy and the diverse number of technologies that characterize the cloud continuum environments. The results obtained with the empirical evaluation 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 the performed experiments. The 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 due to technical limitations. In particular, the use of multi-target (T∗) probe deployment patterns (i.e., Reserved-T∗P∗, Shared-T∗P∗, Partially-shared-T∗P∗, Reserved-T∗P1, Shared-T∗P1) is advised when many different targets or target instances have to 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. On the other hand, the Shared-T∗P1 pattern is the most scalable pattern in the context of application-oriented virtualization, 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. The best practices distilled by the results may help engineers in designing and configuring their monitoring systems, and they generate a set of reusable solutions that people can refer to. Finally, the showcase of the application of certain patterns through three practical usage scenarios

cross-validates the findings and give evidence of the concrete outcome of the work.

The main limitation of this study concerns the relevant features and constraints used to define the presented patterns. Although they were extracted from both white and gray literature, as well as my personal experience with monitoring systems in cloud environments, a systematic protocol for reviewing and analyzing the literature references is missing. This might have introduced both conscious and unconscious biases in the pattern definitions.

Part II

ADAPTING MONITORING TO AVAILABLE RESOURCES

5

PEER-TO-PEER SELF-ADAPTIVE MONITORING IN THE FOG

This chapter presents a self-adaptive Peer-to-Peer (P2P) monitoring system for the Fog that incorporates adaptive behaviors based on the MAPE-K feedback loop [140]. The monitoring system abstracts monitored indicators by using logical states that represent their trend over time and, if necessary, activates countermeasures based on such indicator trends. Countermeasures are defined by means of a lightweight rule-based system that is directly embedded in the peers. The empirical evaluation compares the accuracy and effectiveness of the adaptive version of the monitoring system with the non-adaptive version. The results indicate that adaptive behaviors can increase the accuracy of collected data and save both network and power consumption, but at the cost of higher memory consumption. The contribution reported in this chapter was presented at the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) and published in its proceedings with the title "Towards Self-Adaptive Peer-to-Peer Monitoring for Fog Environments" [65].

The chapter is organized as follows. Section 5.1 introduces the P2P monitoring and provides background information. Section 5.2 presents the proposed self-adaptive P2P monitoring system for fog environments by defining its internal functioning for each phase of the MAPE-K feedback loop. Section 5.3 presents the empirical evaluation. Finally, Section 5.4 concludes the chapter with closing remarks.

5.1 P2P MONITORING

A P2P architecture consists of a network of autonomous self-organizing entities (i.e., peers) that employ distributed resources to accomplish a common task in a decentralized fashion, thus, without relying on central services [172, 186, 225].

The P2P architecture provides the applications with the capability to deal with some of the highly dynamic traits of fog computing, increasing the tolerance to both network failures and nodes joining and leaving the system [1, 64, 91]. Furthermore, it provides autonomy, scale, and robustness, which are critical capabilities to operate in such an environment [236]. Finally, P2P architectures make monitoring data available across the network without relying on a single centralized component, but rather on a set of peers constituting a self-organized overlay network. This is particularly beneficial when the connectivity to the Cloud is limited, such as during disasters or severe network outages [263].

This work uses the two-tier hierarchical P2P monitoring architecture [91, 145, 262] proposed by Forti *et al.* [91] that is shown in Figure 5.1, with *Followers* at the lower tier and *Leaders* at the higher tier.

The benefit of employing such an architecture in the Fog is twofold. First, it implies different roles for peers running in different tiers, depending on the available resources. Followers are designed to collect data by running on the edge, within nodes and devices with limited resources. Leaders are designed to consume more resources to store the data received from the Followers while creating and operating the P2P network. Followers are connected to a single Leader and work in a classic client-server fashion [262]. These distinct roles can be used to opportunistically exploit the available resources, including the possibility to adapt to changing conditions (e.g., bandwidth or resource degradation) through dynamic peer promotion/demotion.

Second, it helps reducing the network overhead by limiting the amount of data transferred between the peers. Actually, Followers can forward data to their Leader only, leaving the thinner upper-tier with the responsibility of building a global state of the monitored resources by exchanging monitoring data among Leaders.

I refer to FogMon as reference implementation for this architecture [91]. In FogMon, the Followers monitor their own deployment node by probing hardware resources (i.e., CPU, memory, and hard disk) and collecting end-to-end network QoS data (i.e., latency and bandwidth). Data is collected and sent to Leader nodes at a *fixed rate*. To limit network overhead, Followers send differential updates, that is, they only send data whose average or variance differ more than a *sensitivity* threshold (i.e., 10% by default) from the last value sent [91]. Leaders periodically aggregate monitoring data received from Followers, and share the aggregated data with the other Leaders through a gossip protocol [130].

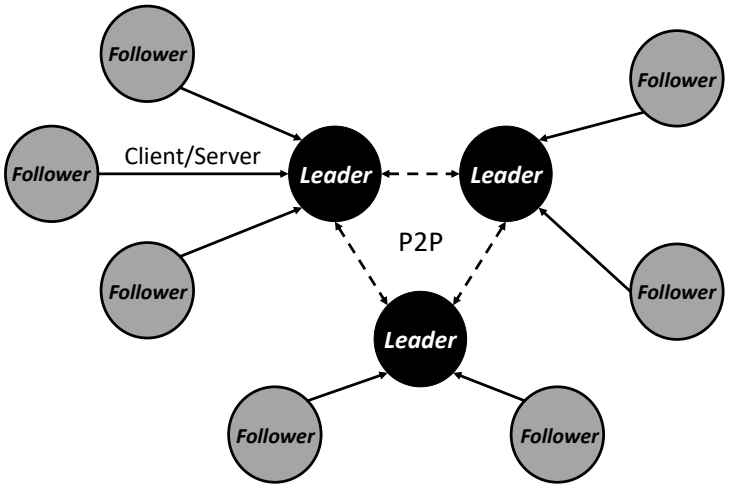


Figure 5.1: Hierarchical P2P monitoring architecture proposed by Forti *et al.* [91].

5.2 ADAPTIVEMON

This section describes how P2P monitoring can be enhanced with self-adaptive capabilities to both make a better use of the available resources and enable the capability to promptly react to run-time events. I refer to the self-adaptive version of the P2P monitoring solution presented in this thesis as AdaptiveMon, in contrast with the non-adaptive version that I refer to as StaticMon.

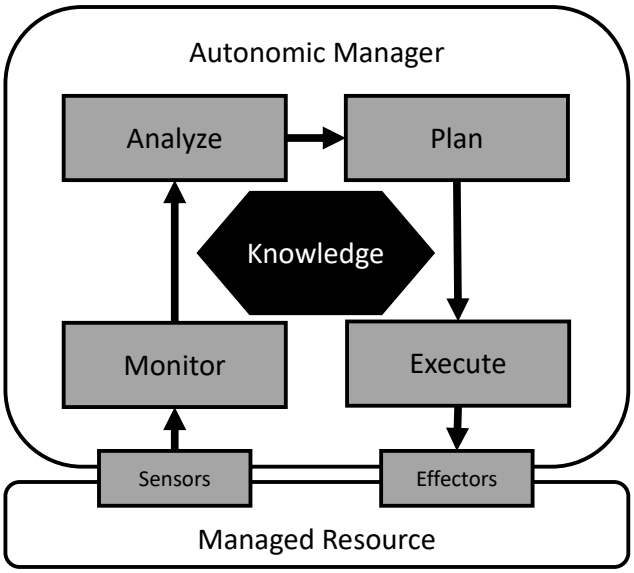


Figure 5.2: Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) loop as proposed by Kephart and Chess [140].

Self-Adaptive Application *(Definition 5.1).* A self-adaptive application (SAA) is an application capable of modifying itself or other connected resources in response to a continuously changing operational environment [48, 205]. A SAA consists of a pair (AL, MR), where AL is the *adaptation logic*, and MR represents the *managed resources* [147], which are a group of resources, such as robotics, vehicles, and generic hardware with software, that the SAA can control [147]. The adaptation logic is composed by all those items responsible for monitoring the environment (M), analyzing the data (A), planning (P), and executing the adaptation (E).

This basic feedback framework proposed by Kephart and Chess [140] is named MAPE loop, and it is often extended by a knowledge component (K) responsible for managing content (e.g., monitoring values and adaptation policies). AdaptiveMon enriches the capabilities of the monitoring system by embedding the MAPE-K control framework [140], shown in Figure 5.2, within each peer.

The *monitor* component of the MAPE-K loop collects data about a managed resource through *sensors*. In this case, this is achieved by the monitoring probes running within the peers. The *analyze* and *plan*

steps analyze the collected data and plan for the countermeasures to be activated. Finally, the *execute* step exploits *effectors* to run the selected countermeasures. In this case, the countermeasures reconfigure the monitoring systems according to the observations collected from the managed resource. The *knowledge* about the managed resource is shared among all the components. Note that the MAPE-K loop runs withing each peer, independently of the overall architecture, which gives peers the capability to run self-adaptive behaviors regardless of their role within the architecture.

In the following, I describe how the components of the MAPE-K loop embedded in the peers are defined, and present two countermeasures that have been experienced in the prototype implementation, namely (i) *Change Rate*, which adjusts the rate Followers sample and forward data to their Leader, and (ii) *Select Indicators*, which dynamically activates and deactivates the set of monitored indicators.

5.2.1 *Knowledge*

The knowledge exploited in AdaptiveMon consists of the monitored *indicators*, which represent the raw knowledge about the monitored resource, and the associated *logical states*, which capture the semantics of the values of an indicator. To this end, I introduce here below a formal definition of what an indicator value, a time series, and the logical states are.

Indicator Value *(Definition 5.2)*. Given a monitored indicator I and a domain D of values for I, $v_t^I \in D$ denotes the value of the indicator I at time t.

Time Series *(Definition 5.3)*. A sequence of values for a same indicator generates a time series, that is, $v_i^I, v_{i+1}^I, \ldots, v_k^I$ is a time series for indicator I.

Logical States *(Definition 5.4)*. Given a monitored indicator I and a finite set of abstract states S, $S_t^I \subseteq S$ represents a potentially empty set of logical states associated with the indicator I at time t. The sequence of states sets $S_i^I, S_{i+1}^I, \ldots, S_k^I$ associated with an indicator also forms a time series. For sake of notation, I is omitted when the indicator is obvious from the context.

While time series of values simply reflect the sequence of probed samples, the corresponding time series of logical states captures the state of an indicator at a specific time, revealing information about the monitored resource. For instance, an indicator might be unstable, too high, or within a normal range of values. These states can be derived from the time series of raw values and used to fire countermeasures, as explained below.

I defined a set of logical states useful for the presented countermeasures, but this set can be clearly extended depending on the countermeasures to be implemented. Table 5.1 summarizes the rigorous

Indicator Type	State	Definition
Categorical	stable	$s_t = \textit{stable} \iff v_t = v_{t-x} \quad \forall x \in [1,k]$
	unstable	$s_t = \textit{unstable} \iff \exists x \in [1,k] \mid v_t \neq v_{t-x}$
Numerical	stable	$s_t = \textit{stable} \iff \lvert Stab_t\rvert \geqslant p \cdot k \wedge \lvert v_t - v_{t-1}\rvert \leqslant \Delta_{max}$, $Stab_t = \{\lvert v_x - v_{x-1}\rvert \leqslant \Delta_{max}\}_{x\in[1,k]}$
	unstable	$s_t = \textit{unstable} \iff \lvert Stab_t\rvert < p \cdot k \vee \lvert v_t - v_{t-1}\rvert > \Delta_{max}$, $Stab_t = \{\lvert v_x - v_{x-1}\rvert < \Delta_{max}\}_{x\in[1,k]}$
	too high	$s_t = \textit{too high} \iff \lvert Too\_High_t\rvert \geqslant p \cdot k \wedge v_t \in I_{\textit{too\_high}}$, $Too\_High_t = \{v_x \in I_{\textit{too\_high}}\}_{x\in[0,k]}$
	high	$s_t = \textit{high} \iff \lvert High_t\rvert \geqslant p \cdot k \wedge v_t \in I_{\textit{high}}$, $High_t = \{v_x \in I_{\textit{high}}\}_{x\in[0,k]}$
	normal	$s_t = \textit{normal} \iff \lvert Normal_t\rvert \geqslant p \cdot k \wedge v_t \in I_{\textit{normal}}$, $Normal_t = \{v_x \in I_{\textit{normal}}\}_{x\in[0,k]}$
	low	$s_t = \textit{low} \iff \lvert Low_t\rvert \geqslant p \cdot k \wedge v_t \in I_{\textit{low}}$, $Low_t = \{v_x \in I_{\textit{low}}\}_{x\in[0,k]}$
	too low	$s_t = \textit{too low} \iff \lvert Too\_Low_t\rvert \geqslant p \cdot k \wedge v_t \in I_{\textit{too\_low}}$, $Too\_Low_t = \{v_x \in I_{\textit{too\_low}}\}_{x\in[0,k]}$

Symbols	Definition
$\lVert\rVert$	Cardinality of a set.
k	Number of samples considered in the recent history of an indicator.
$p \in [0,1]$	Tolerance parameter that indicates the percentage of k samples. It must satisfy the constraint that characterizes the state definition.
Δ_{max}	Maximum delta allowed to consider an indicator as stable.
$I_{\textit{too\_high}} = [\textit{too\_high}, +\infty)$	Interval of indicator values considered *too high*.
$I_{\textit{high}} = [\textit{high}, \textit{too\_high})$	Interval of indicator values considered *high*.
$I_{\textit{normal}} = (\textit{low}, \textit{high})$	Interval of indicator values considered *normal*.
$I_{\textit{low}} = (\textit{too\_low}, \textit{low}]$	Interval of indicator values considered *low*.
$I_{\textit{too\_low}} = (-\infty, \textit{too\_low}]$	Interval of indicator values considered *too low*.

Table 5.1: States definitions for categorical and numerical indicators.

definitions of the logical states, while they are discussed informally below.

In case the domain of a metric is *categorical*, I defined two logical states: *stable* and *unstable*. A categorical indicator is stable at a time t if its value has been constant in the recent history of the execution, otherwise it is unstable.

In case the domain of an indicator is *numerical*, I defined seven states: *stable*, *unstable*, *normal*, *high*, *low*, *too high*, *too low*. A numerical indicator is stable at a time t if its value is close to its value at time $t-1$, and the indicator had small variations in the recent history of the execution. Similarly, a numerical indicator is unstable at a time t if its value differs significantly from the value at time $t-1$, and the indicator had significant variations in the recent history of the execution.

The remaining five states detect values that steadily stay in a given region of the domain. In particular, an indicator is too high or high if its value is above given thresholds and the indicator has been mostly above those thresholds in its recent execution history. Similarly, an indicator is too low or low if its value is below given thresholds and the indicator has been mostly below those thresholds in its recent execution history. Finally, an indicator is normal if its value is in the normal range, and the indicator has been mostly normal in its recent history of the execution. The thresholds for the various levels are defined on a per-indicator basis since they depend on both the indicator and the application domain. For instance, threshold values for memory and CPU consumption are clearly different, and threshold values of memory consumption for a memory-intensive application and a lightweight application are also different.

The set of the collected indicators, along with their raw values and state values represent the knowledge available to AdaptiveMon.

5.2.2 *Monitor*

Monitoring is rather natural and inexpensive in AdaptiveMon since Followers collect data from a monitored resource by construction, and thus the same data sent to Leaders is also available to the MAPE-K loop. If needed, extra indicators can be collected for the only purpose of controlling the adaptive behavior of the peers, even if not needed by the applications accessing the data produced by the monitoring system.

The monitoring behavior is controlled by a *sampling rate* parameter that determines how frequently values v_t^I are collected and forwarded to Leader peers.

5.2.3 *Analyze*

The analysis mainly consists of a data processing routine that converts the raw values collected for every indicator into its logical state representation. In particular, the analysis process accesses the collected values and applies the definitions reported in Table 5.1 to generate a time series of logical states for every monitored indicator.

Figure 5.3 visually exemplifies the logical states that can be associated with a time series, according to the definitions in Table 5.1. The logical states are represented as annotations on the X-axis. Note that in the example, depending on the shape of the curve, a same point may have 0, 1 or up to 2 logical states associated.

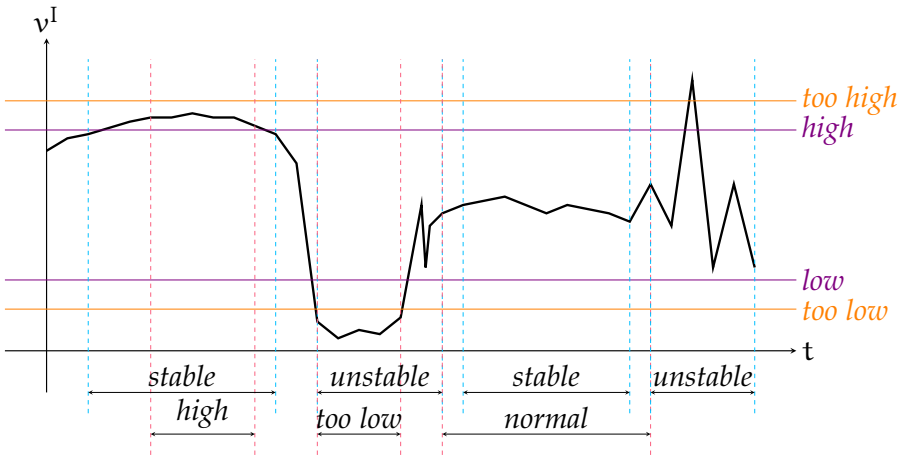


Figure 5.3: An example of the computed states with respect to the time series values at different time instants.

5.2.4 *Plan*

AdaptiveMon embeds the lightweight CLIPS expert system [199], which is responsible for determining the countermeasures that have to activate according to the accumulated knowledge based on a set of adaptation rules.

An adaptation rule consists of two parts: an *antecedent*, that is a set of conditions on the logical states of the indicators that must be satisfied to fire the rule, and a *consequent*, that is a countermeasure to be executed.

Functions that can be used to evaluate the logical states of the indicators are implemented by extending CLIPS, such as functions that can check specific conditions on the last few samples of an indicator. These functions can be used as part of the adaptation rules specified using the CLIPS Domain-Specific Language (DSL). For instance, Listing 5.1 shows an adaptation rule defined to fire the *Change Rate* countermeasure when the CPU consumption has a stable trend. The symbol `=>` separates the antecedent and the consequent of the rule.

It is worth noting that the example adaptation rule uses some of the functions I defined in AdaptiveMon. It checks if the CPU consumption is in a stable state with the `is_indicator_in_state` function and it computes the new rate for such indicator by executing the `compute_indicator_rate` function. The new rate value is in turn used, along with other variables retrieved from the knowledge base, by the `change_rate` function that implements the *Change Rate* countermeasure.

During the plan phase, AdaptiveMon uses the CLIPS expert system to take adaptation decisions, that is, CLIPS evaluates the antecedents of every rule and inserts the countermeasures activated by the consequent of the fired rules in a priority-based queue. Also CLIPS handles the activation of the rules by preventing their simultaneous execution.

To demonstrate the self-adaptive capabilities of AdaptiveMon, two specific countermeasures are defined, namely, *Select Indicators* and *Change Rate*.

The *Select Indicators* countermeasure can change the set of monitored indicators, either activating or deactivating some of them. The *Change Rate* countermeasure changes the rate used to sample and send data to Leaders based on the current trends of the indicators. The goal of the countermeasure is to gradually increase (decrease) the rate while the monitored indicator is less (more) stable. In particular, the countermeasure updates the sampling rate of the indicator I within predetermined boundaries proportionally to the number of consecutive samples with the same logical state out of the last k samples collected.

These countermeasures are exploited in the context of several adaptation rules. For instance, two rules that can enable/disable monitoring for every indicator different from power consumption if the power is above/below a given threshold are defined, to limit the chance a battery-powered device is abruptly shut down. In addition, I defined two rules to adapt the sampling and forwarding rate of CPU indicator to its trend.

Listing 5.1: An example rule that uses the *Change Rate* countermeasure written with the CLIPS DSL. The symbol => separates the antecedent and the consequent of the rule. The `salience` value represents the rule priority. The `bind` operator assigns the result of a function call to a variable.

```
(defrule adapt_cpu_rate_when_stable (declare (salience 10))
 (is_indicator_in_state (indicator cpu) (state stable))
 (has_parameter (rate ?current_rate))
    =>
 (bind ?num_of_states (count_indicator_states_in "cpu" "stable"))
 (change_rate "cpu" (compute_indicator_rate "stable" ?num_of_states ?
     current_rate))
)
```

5.2.5 *Execute*

In this phase, AdaptiveMon merely executes countermeasures by running their implementation according to their priority of activation. The actual countermeasures I defined act on the configuration of the peers adapting their behavior to the evolution of the indicators. The actuation interface is straightforward since a peer can directly access the internal variables that govern its behavior.

5.3 EMPIRICAL EVALUATION

In this section, I quantitatively evaluate the effectiveness (monitoring accuracy) and the efficiency (resource consumption) of AdaptiveMon. I discuss the sub-research questions (Section 5.3.1), the imple-

mented prototype (Section 5.3.2), the experimental setup used to perform the experiments (Section 5.3.3), the results of the experiments to answer the sub-research questions (Section 5.3.4 and Section 5.3.5), and the threats to validity of the evaluation (Section 5.3.6).

5.3.1 *Research Questions*

This work responds to RQ3 and it is assessed with the following two sub-research questions that investigate its effectiveness (i.e., the monitoring accuracy) and its efficiency (i.e., the resource consumption).
RQ3.1 - *Monitoring Accuracy*: Can AdaptiveMon improve the monitoring accuracy of StaticMon? This research question investigates whether the *Change Rate* policy of AdaptiveMon can provide a better monitoring accuracy than StaticMon, considering multiple representative trends of the monitored indicators.
RQ3.2 - *Resource Consumption*: Can AdaptiveMon save node resources compared to StaticMon? This research question studies whether the adaptive behavior of AdaptiveMon reduces resource utilization compared to StaticMon. The impact of the *Change Rate* and *Select Indicators* countermeasures on resource consumption are assessed, both individually and jointly.

5.3.2 *Prototype*

AdaptiveMon is implemented by extending the open-source C++ FogMon P2P monitoring tool [91] along multiple dimensions.

In particular, (i) the structure of the peer's local storage (based on the SQLite[1] database) has been extended to store the logical states used to classify of the monitored indicators; (ii) the CLIPS rule-based expert system [199] has been embedded to support the implementation of adaptation rules; (iii) the peers have been extended to incorporate adaptive behaviors; (iv) helper functions that can be used as part of the adaptation rules to interact with the knowledge have been added; and (v) the *Select Indicators* and *Change Rate* countermeasures have been implemented to dynamically change the set of the collected indicators and the sampling rate parameters. AdaptiveMon does not extend the set of monitored indicators, since the indicators already collected by FogMon to monitor the environment were already sufficient to control the activation of the proposed countermeasures.

The resulting prototype is publicly available with an open-source license at `https://github.com/veracoo/FogMon/tree/adaptive-fogmon`.

1 https://www.sqlite.org

5.3.3 *Experimental Setup*

All the experiments are executed on a Linux virtual machine (Intel i7-9700 CPU @ 3.00GHz x 4, 4 GB RAM, 13 GB SSD, Ubuntu 20.04 LTS 64-bit, Docker v20.10.0). All the peers run inside dedicated Docker containers, deployed on the same host, and communicate over a bridged network. The computational and network resources of the container executing the Follower agent are limited to reproduce a scenario involving resource-constrained and battery-powered devices. Reference devices are single-board computers (SBC) and micro-controller units (MCU) [79, 134, 211]; thus, container resources are upper-bounded at 5% of one CPU core, 20 MB of RAM, and 1 Mbps of bandwidth.

I measure accuracy and resource consumption at the level of individual peers to obtain results that do not depend on the number of co-deployed peers. Thus, each experiment involves one Leader and one Follower (of AdaptiveMon or StaticMon, respectively). Cumulative resource consumption for multiple nodes can be derived by scaling the results proportionally.

5.3.4 *RQ3.1 - Monitoring Accuracy*

This RQ investigates the accuracy of the collected data for both AdaptiveMon and StaticMon considering synthetic indicators following different representative trends. More in detail, a probe reporting readings from such indicators has been implemented. This allowed us to test the correctness of AdaptiveMon's adaptive behavior and verify its effectiveness. I defined 5 scenarios (also referred to as time series in the following) mimicking different key cases for an indicator conventionally ranging between 0 and 1:

1. *stable* is a time series representing a regular and stable, almost constant, trend. It is generated by alternating two close values (0.8 and 0.83), each of which remains stable for 14 seconds;

2. *unstable* is a time series that represents an irregular and erratic indicator with fluctuating values in the range $[0.5, 0.85]$; a real-life trace on CPU utilization was used as a base.

3. *stable-unstable* is a time series that alternates phases of stability with phases of instability, with each phase lasting for about 150 seconds;

4. *random* is a time series with chaotic and totally unpredictable values; it is generated by a sequence of random values uniformly distributed in the range $[0, 1]$;

5. *spiky* is a time series with mostly regular values interleaved with rare spikes; it is generated by alternating stable values for 28 seconds, unstable values for 12 seconds, and then a spike value for 4 seconds.

Every time series has a duration of 10 minutes, except for the stable-unstable time series that lasts 20 minutes since it is a combination of the stable and unstable time series.

StaticMon and AdaptiveMon's Follower peers are configured to forward the average of the last 20 measurements to their respective Leaders at each probing point. The StaticMon Follower probes a new value from the monitored metric at a fixed interval: every 30 seconds. AdaptiveMon exploits the *Change Rate* countermeasure to adjust the sampling rate to handle the variability of the monitored indicator. The hypothesis to test is that this can lead to improved monitoring accuracy because the Leader should have access to a higher number of samples when the monitored indicator is highly dynamic and fewer samples in the presence of more stable indicators.

I investigate the capability of the monitoring system to reconstruct the shape of the monitored indicators at the level of both the Follower, which directly samples the indicator, and the Leader, which collects a sequence of average values. The *Root Mean Square Error* (RMSE), which measures the differences between the original and the reconstructed indicator, is used as the primary quality metric. A smarter sampling strategy should achieve a lesser error. To appreciate the activity of the peers in relation to the monitored indicator, I also gauge the *messages/second* ratio, that is, the ratio of the messages sent by the Follower to the Leader. Finally, for the spiky time series, I also computed the *percentage of detected spikes*, which measures the capability of a monitoring technique to spot rare but significant events.

RESULTS OF RQ3.1 Table 5.2 summarizes the results obtained by both AdaptiveMon and StaticMon for the considered 5 scenarios. The last two columns show the absolute (Abs) and relative (Rel) deviations between the AdaptiveMon and StaticMon results for any of the presented quality indicators. Green (Red, respectively) cells indicate a better (worst) result obtained by AdaptiveMon compared to the StaticMon baseline. It is possible to observe that AdaptiveMon estimates the observed indicator more accurately than StaticMon at both levels of the Leader-Followers hierarchy in 4 out 5 scenarios (viz. *stable*, *unstable*, *stable-unstable*, *random*). The reduction in the RMSE reached 33.6% at Follower level (*unstable* scenario) and 82.7% at Leader level (*random* scenario). A higher accuracy, however, comes at the cost of a higher number of messages exchanged in the 4 scenarios where the monitored indicator is more erratic (nearly 5 times more than StaticMon in the worst case), and fewer messages produced when the indicator is stable (saving nearly one third of the messages).

Figures 5.4 and 5.5 exemplify the results of the comparison between AdaptiveMon and StaticMon for the *stable-unstable* scenario. It is possible to observe that the time series reconstructed by AdaptiveMon (solid orange line) is closer to the reference indicator (dotted green line) than the StaticMon baseline (dashed blue line). It is also

Scenario	Quality Metric	AdaptiveMon	StaticMon	Abs	Rel
Stable	RMSE (Follower)	0.019	0.020	- 0.181	- 5.0 %
	RMSE (Leader)	6.696	8.200	- 1.504	- 18.3 %
	Messages/second	0.027 m/s	0.040 m/s	- 0.013	- 32.5 %
Unstable	RMSE (Follower)	0.087	0.131	- 0.044	- 33.6 %
	RMSE (Leader)	2.428	5.033	- 2.605	- 51.7 %
	Messages/second	0.217 m/s	0.037 m/s	+ 0.180	+ 486.5 %
Stable-unstable	RMSE (Follower)	0.108	0.122	- 0.014	- 11.5 %
	RMSE (Leader)	5.269	6.546	- 1.277	- 19.5 %
	Messages/second	0.103 m/s	0.035 m/s	+ 0.068	+ 194.3 %
Random	RMSE (Follower)	0.235	0.321	- 0.086	- 26.8 %
	RMSE (Leader)	1.899	10.683	- 8.784	- 82.7 %
	Messages/second	0.217 m/s	0.037 m/s	+ 0.180	+ 486.5 %
Spiky	RMSE (Follower)	0.092	0.087	+ 0.005	+ 5.8 %
	RMSE (Leader)	5.713	6.251	- 0.538	- 8.6 %
	Messages/second	0.062 m/s	0.037 m/s	+ 0.025	+ 67.6 %
	Detected spikes	30 %	0	+30 %	-

Table 5.2: Accuracy of AdaptiveMon and StaticMon for the 5 scenarios. Green (Red) cells indicate a better (worse) result obtained by AdaptiveMon compared to the StaticMon.

interesting to notice how the rapid change in the observed trend is not immediately handled by AdaptiveMon, which shows some delay in sensing the drift and adjusting the sampling rate. In contrast, StaticMon always fails to follow the observed time series, confirming the importance of adaptivity in similar contexts.

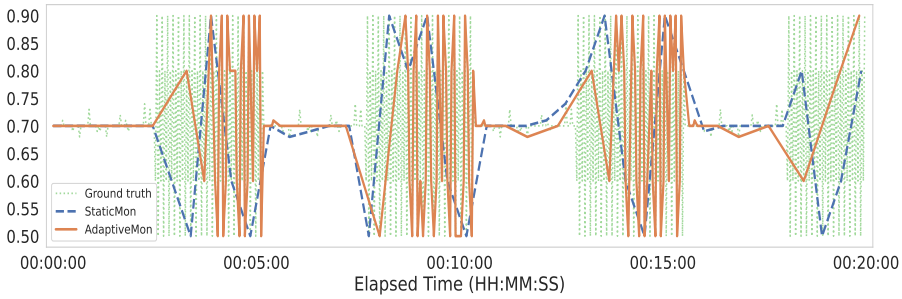


Figure 5.4: AdaptiveMon and StaticMon Follower time series estimations for the *stable-unstable* scenario.

The *spiky* scenario is the only one resulting in an increment of the RMSE metric for the adaptive Follower (+5.8%). However, this increment is a consequence of the capability to (partially) follow the trend of the indicator. In fact, the StaticMon configuration could detect

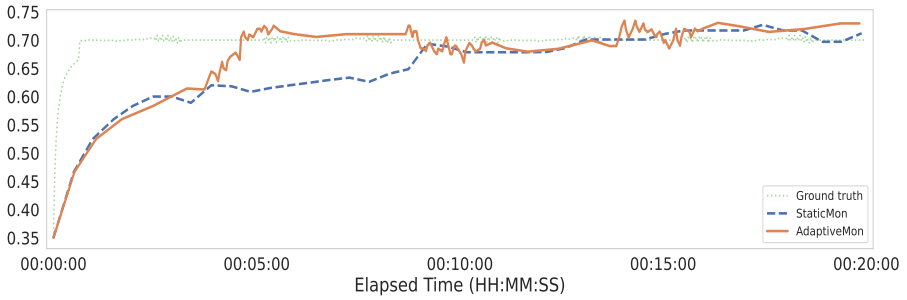


Figure 5.5: AdaptiveMon and StaticMon Leader time series estimations for the *stable-unstable* scenario.

spikes only incidentally, while AdaptiveMon could change its sampling rate to increase the chance to capture them. Figure 5.6 illustrates a representative execution of StaticMon and AdaptiveMon for the spiky scenario, with some spikes successfully detected by AdaptiveMon only. Although successfully capturing some spikes, the reconstructed time series generates a higher error compared to the flat time series reconstructed by StaticMon. Indeed, this is a challenging scenario for both approaches (rare short events are hard to detect by monitoring techniques), and more work is required to design cost-effective monitoring techniques that can accurately address spikes.

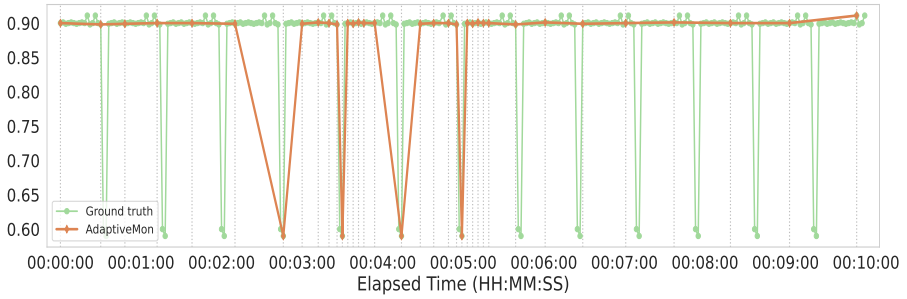


Figure 5.6: AdaptiveMon Follower time series estimation for the *spiky* scenario. The vertical dotted grey lines indicate the *sampling rate*.

ANSWER TO RQ3.1 AdaptiveMon estimates the observed indicator more accurately than StaticMon at both levels of the Leader-Followers hierarchy in 4 out of 5 scenarios. The results show that AdaptiveMon is capable of adjusting resource consumption as necessary by sending more messages only when the monitored indicator requires more precise sampling, and conserving bandwidth otherwise. However, further investigation is required to design an effective monitoring solution that can accurately detect spikes.

5.3.5 *RQ3.2 - Resource Consumption*

This RQ investigates resource consumption considering the two countermeasures currently implemented in AdaptiveMon, both in isolation and jointly. Again, StaticMon is used as the baseline for the comparison.

For the experiments reported in this section, I defined a probe (at the follower level only) that exploits the Docker Engine API[2] and PowerTOP[3] to collect the following quality metrics:

- CPU and memory consumption: the percentage of the host's CPU and memory used.
- NET I/O (MB): the cumulative amount of data sent and received over its network interface from the beginning of the experiment.
- PIDs: the number of processes or threads spawned by the peer.
- PW (mW): the estimated instantaneous power consumption.

The three AdaptiveMon configurations assessed in this RQ exploit the two strategies defined in Section 5.2. *Change Rate* adjusts the sampling and forwarding rates of all the collected indicators from 30 to 60 seconds based on the monitored values. *Select Indicators* disables the collection of all indicators except of power consumption if the battery level drops below a threshold. *Combined Countermeasures* uses both strategies. I study the impact of these configurations, along with the StaticMon baseline, on resource consumption: a total of four possible configurations are therefore considered. In each experiment, the Follower peer is configured to collect the indicators from its node for 30 minutes and to apply the countermeasures at the beginning of the execution, in such a way the impact of the countermeasures can be accurately measured (in fact, more than 300 hundreds samples per metric have been collected).

RESULTS OF RQ3.2 Figure 5.7 shows a series of five box plots (one for each quality metric) where each plot compares the four compared configurations visually.

I checked the significance of the differences between distributions with the non-parametric Mann–Whitney U test [162], as it was possible to observe (via the Shapiro-Wilk test [213]) that such differences are not normally distributed. I specifically checked if the observed differences between the baseline and any other configuration are statistically significant and if *Combined Countermeasures* is significantly better than the individual adaptations strategies (*Change Rate* and *Select Indicators*). I considered a significance level $\alpha = 0.05$, and I also computed the effect size of the observed phenomenon using the Wendt's

2 https://docs.docker.com/engine/reference/commandline/stats/
3 https://github.com/fenrus75/powertop

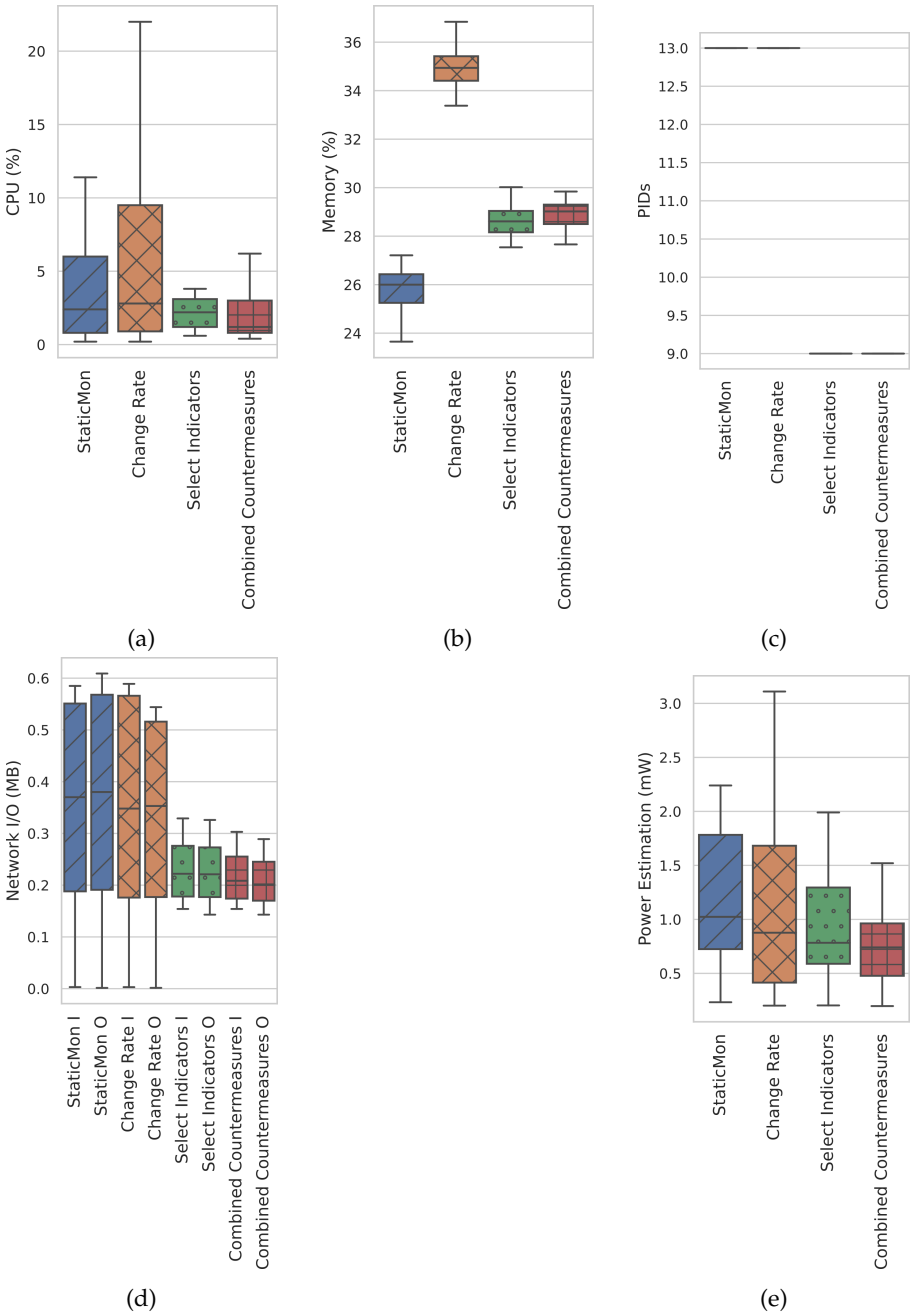


Figure 5.7: StaticMon compared with AdaptiveMon countermeasures for each of the collected quality metrics.

formula [256]. Table 5.3 shows the significant cases only with their corresponding effect size using the conventional categories *small* (less than 0.3), *medium* (between 0.3 and 0.5), and *large* (greater than 0.5).

Only observe marginal differences in CPU consumption are observed. In particular, differences between AdaptiveMon and StaticMon are not significant, and *Combined Countermeasures* introduces significant but small differences compared to employing the other two adaptive strategies individually.

Quality Metric	Comparison	Effect Size
CPU consumption	Change Rate vs Combined Countermeasures	small (0.197)
	Select Indicators vs Combined Countermeasures	small (0.260)
Memory consumption	StaticMon vs Select Indicators	large (0.858)
	StaticMon vs Change Rate	large (0.993)
	StaticMon vs Combined Countermeasures	large (0.861)
	Change Rate vs Combined Countermeasures	large (0.990)
	Select Indicators vs Combined Countermeasures	small (0.279)
# spawned sub-processes	StaticMon vs Select Indicators	large (0.980)
	StaticMon vs Change Rate	small (0.027)
	StaticMon vs Combined Countermeasures	large (0.963)
	Change Rate vs Combined Countermeasures	large (0.962)
Network Input	StaticMon vs Select Indicators	medium (0.449)
	StaticMon vs Change Rate	small (0.101)
	StaticMon vs Combined Countermeasures	medium (0.483)
	Change Rate vs Combined Countermeasures	medium (0.431)
	Select Indicators vs Combined Countermeasures	small (0.145)
Network Output	StaticMon vs Select Indicators	medium (0.473)
	StaticMon vs Change Rate	small (0.245)
	StaticMon vs Combined Countermeasures	large (0.521)
	Change Rate vs Combined Countermeasures	medium (0.460)
	Select Indicators vs Combined Countermeasures	small (0.203)
Battery power estimation	StaticMon vs Combined Countermeasures	medium (0.376)

Table 5.3: Statistically valid comparisons for all the quality metrics with their associated effect size.

The memory consumption results show statistical significance for all cases with a large effect size for all comparisons, except for *Select Indicators* compared to *Combined Countermeasures* where the effect size is small. The impact of the adaptive strategies on the memory indicator is antithetic: while it is possible to notice an increase in memory consumption of about 10% for *Change Rate* (compared to StaticMon), memory overhead decreases to about 3% when *Select Indicators* or *Combined Countermeasures* are used. These results can be easily explained considering that although the MAPE-K control loop increases the amount of memory used by AdaptiveMon when all probes are active, while when these are disabled (freeing the associate resources) the overall average memory usage decreases.

Since limiting the number of processes can be particularly important when the underlying device platform is resource constrained I measured the number of sub-processed spawned by all compared configuration. In this regard, the number of spawned sub-processes show significant reduction with large effect size when *Select Indicators* and *Combined Countermeasures* are used.

Limiting bandwidth consumption is also extremely important in fog environments. As a matter of fact, limiting I/O operations is crucial

when the network bandwidth is limited and shared by multiple devices and thus can be quickly saturated. Moreover, intensive communication implies high power consumption, a threat to energy efficiency and batteries lifespan in portable devices. The results for network I/O show statistically significant reduction for all adaptive configurations compared to StaticMon, with an effect size ranging from small to medium. Note that results for Input (I) and Output (O) present a similar behavior for the same configuration.

More in detail, results show a small effect size for *Change Rate* versus StaticMon comparison. Since I expected a stronger impact of rate adaptation in this context, I analyzed the behavior of the probes, and discovered that the bandwidth gauge exploits iPerf[4], which measures the bandwidth by saturating it with packets. Such an invasive behavior nullifies the potential benefits of a dynamic sampling rate. Therefore, to further investigate this dimension, I repeated the experiments by disabling the bandwidth monitoring probe for both StaticMon and AdaptiveMon. Results are presented in Figure 5.8. The impact of *Change Rate* is now remarkable, with a reduction on transmitted data ranging between 31% and 34%, with an even higher reduction (between the 37% and the 49%) when both countermeasures are simultaneously active.

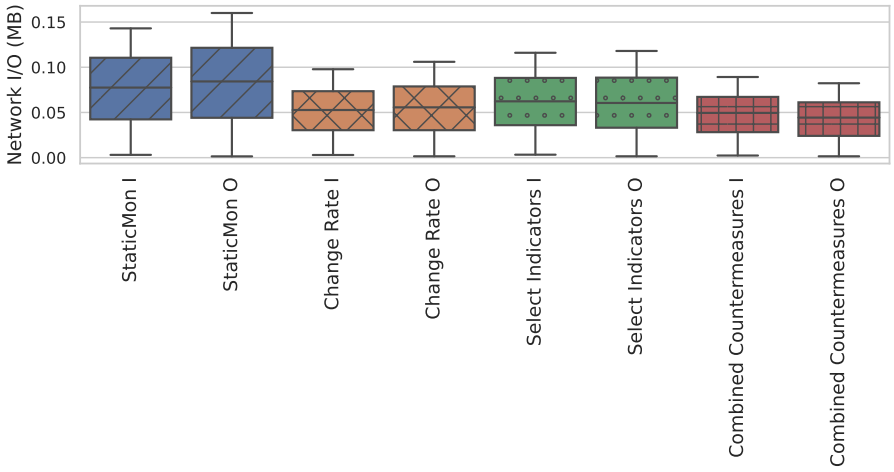


Figure 5.8: StaticMon compared with AdaptiveMon countermeasures for the network I/O metrics when the bandwidth is not measured by the Follower.

Finally, the results on power consumption show meaningful differences only for StaticMon versus *Combined Countermeasures*, suggesting that the individual countermeasures may introduce limited benefits. Still, their combination can significantly improve battery lifetime (with an estimated reduction in power usage of about 36%).

4 https://iperf.fr/

ANSWER TO RQ3.2 AdaptiveMon can help reduce network I/O and device battery usage without affecting CPU utilization. This is achieved by allocating extra memory to store the necessary data for running the adaptive mechanisms. Such result is cost-effective in fog environments, because the impact of limited extra memory is mitigated by the availability of memory, even on resource-constrained devices deployed in the Fog.

5.3.6 *Threats to Validity*

The presented study is affected by both internal and external threats to validity. The main internal threats to validity concern with the design of the scenarios used to study RQ3.1. I proposed five scenarios to mimic different trends. Although indicators collected in real scenarios may behave differently than the ones I investigated, the results obtained with the stereotyped trends are still informative, at least locally (e.g., it is possible to refer to the results reported in the thesis for an indicator that becomes unstable or too high).

The definition of the logical states for an indicator depends on several parameters, which are application-dependent. In this thesis, I studied how AdaptiveMon can be used to obtain self-adaptive capabilities relevant to monitoring, focusing on the assessment of simple countermeasures that do not strongly depend on the domain. Assessing AdaptiveMon in the context of dedicated application scenarios is part of future work and is out of the scope of this thesis.

The generality of the results obtained about efficiency (RQ3.2) might depend on the specific implementation used and the size of the experiment. I used an independent implementation for StaticMon (i.e., FogMon) to minimize any implementation bias, and the adaptive behavior is added to this implementation. To further reduce any implementation threat, the solution is publicly released. In principle, additional experiments may lead to different results. However, I obtained quite clear evidence and I checked the statistical significance of the results to mitigate the risk of overgeneralizing.

5.4 DISCUSSION

The self-adaptive monitoring system proposed in this thesis can abstract monitored indicators and activate countermeasures based on their logical states. Empirical results demonstrate that adaptive behaviors can enhance monitoring accuracy while optimizing resource utilization, compared to non-adaptive solutions. AdaptiveMon can help reduce network I/O and device battery usage without affecting CPU utilization. This can be achieved by allocating extra memory to store the necessary data. This result can be beneficial in the Fog since monitoring systems are called to reduce network overhead without impacting on power consumption, especially when devices

are battery-powered. Even resource-constrained devices at the edge of the network are typically well-equipped with memory, which mitigates the impact of the monitoring system's limited extra memory consumption. In particular, the reduction in the RMSE reached 33.6% at Follower level in the *unstable* scenario and 82.7% at Leader level in the *random* scenario. A higher accuracy, however, comes at the cost of a higher number of messages exchanged in the 4 scenarios where the monitored indicator is more erratic (nearly 5 times more than StaticMon in the worst case), and fewer messages produced when the indicator is stable (saving nearly one third of the messages). With respect to the resource consumption, the results for network I/O show statistically significant reduction for all adaptive configurations compared to StaticMon, while the combination of both the countermeasures can improve battery lifetime with an estimated reduction of the power usage of about 36%.

The presented work has three main limitations. First, defining the adaptive rules and their parameters is a manual activity, subject to the application knowledge of the operators managing the monitoring system. Additionally, it is possible that multiple instances of the monitoring system require different rules and configurations, necessitating a non-negligible configuration effort. To potentially mitigate this limitation, MaaS solutions can be adopted to automate the configuration process. Secondly, while the obtained results have been statistically checked and threats to validity have been discussed, AdaptiveMon has not been evaluated on a realistic scale involving a fog infrastructure test-bed. A more exhaustive experimental campaign can demonstrate the contribution of work on a larger scale. Third, each peer is autonomous at both the Follower and Leader tiers. Therefore, aggregated information obtained by Leaders may differ if some peers stop collecting certain indicators or change their sampling and forwarding rates. It would be worthwhile to investigate the impact of this architectural choice or to distribute the MAPE-K components differently, with Leaders responsible for governing the adaptive behavior of their Followers.

6

ENERGY-AWARE SELF-ADAPTIVE MONITORING IN THE EDGE

This chapter presents an energy-aware approach to design and implement self-adaptive applications for edge environments. Specifically, this work focuses on AI-based monitoring systems, which are increasingly deployed in the Edge. These systems represent a real-world and challenging scenario that requires the delivery of effective and sustainable AI edge services. The proposed approach can guide developers in implementing applications that can switch operation modes in response to environmental changes, balancing energy consumption with application-level objectives. The empirical evaluation shows how the approach can outperform non-adaptive baseline configurations, performing as optimally as configurations selected with a nearly exhaustive exploration of the configuration space. The contribution reported in this chapter was presented at the 38th International Conference on Automated Software Engineering (ASE) and published in its proceedings with the title "An Energy-Aware Approach to Design Self-Adaptive AI-based Applications on the Edge" [240].

The chapter is structured as follows. Section 6.1 presents a Smart Traffic Monitoring (STM) motivational scenario. Section 6.2 describes the proposed approach, with specific reference to the motivational scenario. Section 6.3 presents the empirical evaluation. Finally, Section 6.4 concludes the chapter with closing remarks.

6.1 MOTIVATIONAL SCENARIO

According to the latest report released by Governors Highway Safety Association (GHSA), "nearly 3.500 pedestrians died in the United States in the first six months of 2022 (+5% from the same period in 2021)" [23]. "In three years, pedestrian deaths raised about 18%, that is, nine times faster than U.S. population growth" [175]. Similarly, the European Transport Safety Council (ETSC) reported "20.600 road deaths in the EU last year, with vulnerable road users (pedestrians, cyclists, and users of powered two-wheelers) representing just under 70% of total fatalities within urban areas" [72, 212]. Addressing this critical issue of preventing accidents not only depends on social education [56] but also requires developing Smart Traffic Monitoring (STM) systems that enable digital monitoring of urban traffic [8, 42, 217], real-time analytics [17, 38], and intelligent driver assistants [153, 160, 250].

An STM system requires continuous monitoring of the traffic scenarios to identify potential incidents (e.g., the presence of pedestrians in blind spots) through video streams and processing frames, and alert-

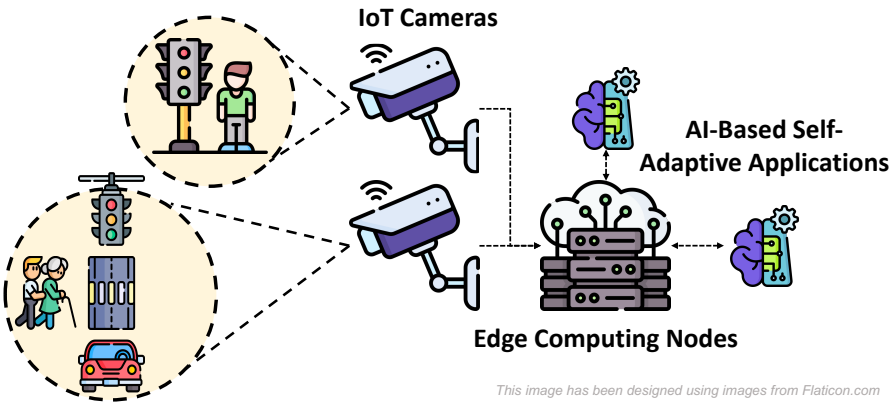


Figure 6.1: A pedestrian detection scenario.

ing the nearby vehicles through the use of 5G-enabled edge nodes [160]. Such an STM system can host hundreds of cameras and sensors deployed to roads in cities and countryside areas [78].

The edge devices processing video streams are in always-on mode and potentially powered by batteries or renewable energy sources at the edge, which is the basis for limited and unreliable power supply. Hence, reducing energy consumption and executing critical emergency applications become extremely important. On the other hand, such critical applications expect a minimum QoS for safety and reliability (e.g., inference time and ML model accuracy). Therefore, they require continuous monitoring of resources (e.g., energy budget) and workload (e.g., number of detected pedestrians in time intervals), and when needed, employing self-adaptive applications and adapting hardware and software configurations (e.g., camera resolution, ML model, and hardware acceleration).

Figure 6.1 depicts a pedestrian detection scenario where an application can employ different operation modes according to pedestrian traffic volumes. For instance, this scenario could be addressed with four operation modes as defined in Table 6.1. A self-adaptive application for this scenario can autonomously balance resource (e.g., energy consumption) and application requirements (e.g., frame processing speed and accuracy) by switching among the different operation modes.

On the contrary, using a single operation mode for a whole day cannot adapt to a changing environment. Considering a smart-city scenario with hundreds of IoT cameras and dozens of application instances deployed across several edge nodes, the benefits of such an approach are exponential.

6.2 DESIGNING ENERGY-AWARE SELF-ADAPTIVE APPLICATIONS

A *self-adaptive application (SAA)* (Definition 5.1) is an application capable of modifying itself, or other connected resources, in response to

Table 6.1: A set of four operation modes used in the motivational pedestrian detection scenario.

Operation Mode	Runtime Context	Desirable Characteristics		
		Energy Consumption	*Detection Accuracy*	*Frames Processing Rate*
power-saving	no pedestrians detected	very low	low	moderate
low-energy	few pedestrians detected	low	moderate	moderate
high-accuracy	small group of pedestrians detected	moderate	high	high
high-rate	crowd detected	high	moderate	very high

changes detected in the operational environment. SAAs are particularly effective in resource-constrained environments. Let us consider here the case of an AI-based application that implements the pedestrian detection use-case described in Section 6.1 and that is hosted on an embedded device (e.g., a Raspberry Pi) equipped with a video camera and a hardware accelerator (e.g., a TPU). The device executes an application capturing frames from the camera and processing them with an object detection model to detect pedestrians.

The hardware accelerator boosts the processing speed by lowering the ML model inference time. In this context, three main objectives must be considered: achieving high detection accuracy, processing frames at a high rate, and reducing energy consumption.

Optimizing these objectives at the same time for every possible operational condition is generally infeasible. Interestingly, a SAA can dynamically balance the degree of satisfaction of these objectives depending on the run-time context. However, engineers designing SAAs need to identify *suitable configurations* for the run-time to balance the chosen objectives. Further, SAAs have to implement the logic to automatically switch between configurations (e.g., the four operation modes reported in Table 6.1), to adapt to changes in the operational environment (e.g., the pedestrian traffic volumes).

Identifying the configurations that implement the intended operation modes is also challenging, especially for AI-based applications running on heterogeneous and resource-constrained nodes. Indeed, simply using a simulator may lead to results largely diverging from the real behavior of these applications. On the other hand, taking empirical measures by running the real devices and applications can be extremely expensive, especially when large configuration spaces must be explored [189].

I propose here an approach that combines the benefits of the *empirical identification* of the configurations and those of an intelligent *exploration of the configuration space* to yield suitable solutions to design an effective and energy-aware SAA.

Figure 6.2 describes the proposed approach with a workflow diagram. An engineer provides the adaptation logic (A) as a finite-state machine (FSM) whose states represent the SAA operation modes and whose transitions encode the switching conditions between them. In parallel, the engineer identifies the configuration space to explore, and

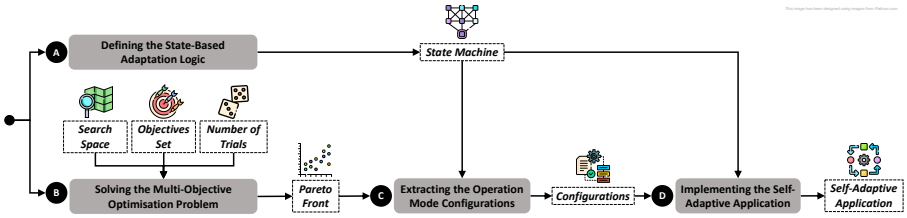


Figure 6.2: The steps of the proposed approach represented as a workflow diagram.

defines a Multi-Objectives Optimization Problem (MOOP) that can be solved automatically (B) using a meta-heuristic search procedure. Furthermore, the engineer specifies weights and thresholds for the objectives to guide the (C) extraction of the configurations to set in each operation mode. The workflow terminates (D) with the implementation of the final FSM.

In the next subsections, I describe each step of the workflow in detail and exemplify the approach with the pedestrian detection scenario described in Section 6.1.

6.2.1 *Defining the State-Based Adaptation Logic*

The first step of the proposed approach requires an engineer, supported by domain experts, to define, in a rigorous way, the *behavioral model* of the self-adaptive application [97].

As specification I use a *Finite-State Machine (FSM)*, since it allows to explicitly represent the adaptation logic of an SAA [19, 138, 154]: the states represent the operational modes of the SAA, and the transitions represent the conditions triggering a change in the operation mode of the application.

Finite-State Machine *(Definition 6.1)*. A FSM M is defined by a tuple (S, Σ, δ, s_0), where S is the set of states, Σ is the set of the input symbols, that is, the set of events that may trigger state transitions, δ is the set of all the possible transitions from a state $s_1 \in S$ to a state $s_2 \in S$ caused by an event $\sigma \in \Sigma$, s_0 is the initial state.

Let us consider the pedestrian detection scenario again. Here an engineer may want to define a SAA that can self-adapt across four operation modes (see Table 6.1) to address the four possible run-time contexts in the area where the camera shall be deployed, defined for instance according to the available studies [24, 78, 144]. Each operation mode, for example *low-energy*, represents the working condition of the software that is best suited for the corresponding run-time context, for example *few pedestrians detected*. Each operation mode must satisfy certain characteristics in terms of energy consumption, detection accuracy and frames processing rate. These characteristics are used to identify the exact software configurations at step (C) Extracting the Operation Mode Configurations by providing the corresponding sets of ob-

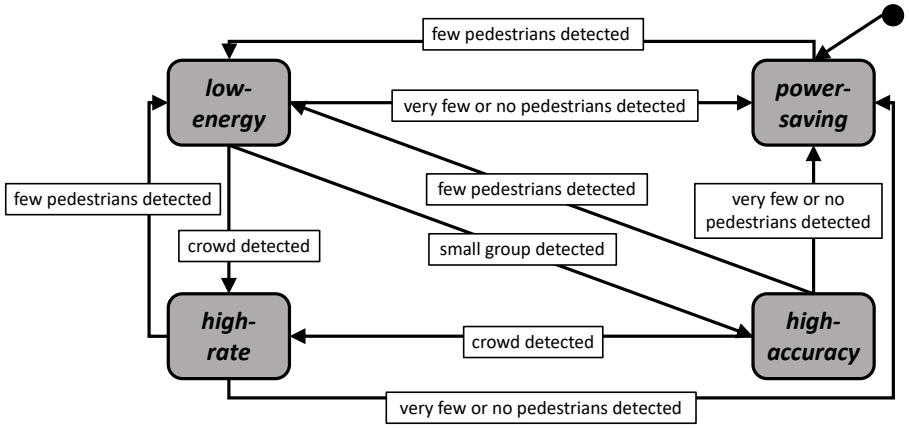


Figure 6.3: An abstract state machine modeling the states and the transitions of a self-adaptive application for the motivational scenario.

jective weights and thresholds. Figure 6.3 shows an abstract FSM, with the four identified abstract states and 9 transitions that capture when the software must self-adapt. Please note that the domain-knowledge is exploited here to determine the transitions that must be encoded in the FSM, among the full set of the possible state transitions.

6.2.2 *Solving the Multi-Objective Optimization Problem*

Finding high-quality software configurations that correspond to the operation modes identified by the engineer (e.g., the four states shows in Figure 6.3) is a hard problem. AI-based applications can be configured according to several parameters (see for instance the list of parameters that may influence pedestrian detection listed in Table 6.2), generating a huge exploration space that cannot be exhaustively explored. Computer-simulated experiments can reduce the time and effort, but they are usually inaccurate, especially in Cyber Physical Systems and other domains that include real-world metrics [202].

To address this challenge, I defined a *Multi-Objective Optimization Problem (MOOP)* that is able to discover the configurations that deliver the best results for the considered set of objectives, and that can be exploited to find the actual configurations that effectively implement the operation modes represented as states of the FSM.

An optimization process aims to find a set of input values for a problem to obtain the "optimal" output values. The definition of optimality is problem-specific, and formally, it refers to minimizing or maximizing one or more objective functions by varying the input values. Hence, a MOOP requires the satisfaction of a number of different and often conflicting objectives at the same time [181, 224]. Intuitively, there is no single best solution for all the objectives, but rather there exist several optimal solutions representing the best trade-offs among all the objectives [224].

Search Space *(Definition 6.2)*. The search space is the set of all possible solutions, that then also contains the set of input values revealing optimal outputs. The search space X is here defined as a set of configurations.

Configuration *(Definition 6.3)*. A configuration *conf* is n-tuple $(c_1, \ldots, c_n)$, where c_k is the value of the k-th configurable parameter $p_k \in P$ assuming values in its domain D_{p_k}. The size of X is $|X| = \prod_{k=1}^{n} |D_{p_k}|$.

Pareto Front *(Definition 6.4)*. The set of solutions X^* is called the Pareto front, which contains all the solutions where no improvement is possible in any objective function without sacrificing at least one of the other objective functions [181]. This is also referred to as the non-dominated solutions set.

In the pedestrian detection scenario there are three objectives: (i) maximize the pedestrians detection accuracy (*acc*), (ii) minimize the energy consumption (*eng*), and (iii) maximize the number of processed frames in a time window (*rate*). Hence, I define a MOOP with these three objectives (depending on the specific case, it might be possible to have a different number of objectives):

$$\begin{aligned} \min \quad & -acc(conf) \wedge eng(conf) \wedge -rate(conf) \\ \text{s.t.} \quad & conf \in X \end{aligned} \tag{1}$$

The search space X is defined as a set of configuration quintuples with five configuration parameters for the example application, that is, the camera resolution (R), the camera frame rate (*FPS*), the object detection model (M), the detection threshold (T), and whether to use the external hardware accelerator (*TPU*). Each parameter domain has a different cardinality (see details in Table 6.2). Accordingly, $|X| = |R| \times |FPS| \times |M| \times |T| \times |TPU| = 3402$ configuration quintuples.

Table 6.2: The domain of the parameters used to define the search space of the multi-objective optimization problem.

Parameter	Parameter Type	Domain
Camera Resolution (R)	Categorical	{1920x1080, 1280x720, 640x480}
Camera Frame Rate (*FPS*)	Categorical	{1, 5, 10, 15, 20, 25, 30}
Object Detection Model (M)	Categorical	{SSD MobileNet V1, SSD/FPN MobileNet V1 TF2, SSD MobileNet V2, SSD MobileNet V2 TF2, SSDLite MobileDet, EfficientDet-Lite0, EfficientDet-Lite1, EfficientDet-Lite2, EfficientDet-Lite3}
Detection Threshold (T)	Numerical (low: 0.1, high: 0.9, step: 0.1)	{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}
Use HW Accelerator (*TPU*)	Categorical	{true, false}

Solving the Equation 1 results in a Pareto front with non-dominated solutions, that is, configurations that *fulfill the three objectives by a different, but relevant, degree*. I use a strategy derived from NSGA-II to compute the Pareto front.

NSGA-II is a solid and widely used optimization algorithm in real-world applications [249]. I use the approach defined by Deb *et al.* [74]

for the exploration of the search space: it is explored by searching for dominant solutions (i.e., the fitness of a solution is defined by computing its non-domination level) in less populated areas of the space (i.e., determined by computing the crowding distance) guaranteeing the diversity of the identified solutions; mutations randomly change parameter values with a probability that is computed according to the number of parameters in the configuration, and uniform crossover recombines configurations with a probability of 0.9.

During the search space exploration, the presented procedure records all the evaluated objective values, and at the end it extracts the Pareto front from the whole results set. In the empirical evaluation, I show how this strategy can be used to explore only 10% of the search space to select nearly optimal configurations. Note this is particularly relevant, since assessing how a single configuration fulfills the three objectives requires collecting empirical measures by repeating a same experiment multiple times.

6.2.3 *Extracting the Operation Mode Configurations*

The Pareto front obtained by solving the MOOP usually contains a large number of non-dominated solutions, compared to the operation modes needed by the self-adaptive application. The decision-making process to identify the actual solutions from the Pareto front involves comparing multiple criteria, trading-off certain objectives for others [148, 255]. To address this problem, I use the *weighted gray relational analysis (WGRA)* [148] method, a weighted version of the GRA introduced by Ju-Long [135] and employed in multiple application domains [61]. This is a very robust method [161], preferable to other multi-criteria decision making (MCDM) methods as it inherently incorporates uncertainty in data, and it is simple to calculate [161, 259] and to integrate into existing software.

GRA combines into a single value all the objectives. This simplifies the original MCDM problem into a single-criterion decision-making problem [148], making Pareto front solutions easily comparable. To let engineers extract states that fulfill the objectives by different degrees, I employ the weighted version of the algorithm that uses a set of weights W to give more importance to certain objectives [61].

The WGRA algorithm consists of three main steps: (i) data normalization, (ii) reference network computation, and (iii) gray relational grade (GRG) computation [255].

The *data normalization* step consists of the normalization of the objective values in the Pareto front according to two cases: larger-the-better for maximization, and smaller-the-better for minimization. The nor-

malized value F_{ij} is calculated by Equation 2 and 3 for maximization and minimization cases, respectively:

$$F_{ij} = \frac{f_{ij} - \min_{i \in n} f_{ij}}{\max_{i \in n} f_{ij} - \min_{i \in n} f_{ij}} \tag{2}$$

$$F_{ij} = \frac{\max_{i \in n} f_{ij} - f_{ij}}{\max_{i \in n} f_{ij} - \min_{i \in n} f_{ij}} \tag{3}$$

with f_{ij} as the i-th value of the j-th objective in the matrix O, a matrix $n \times m$ composed of n Pareto front solutions and m objectives. F_{ij} is the value of f_{ij} after normalization.

The *reference network computation* step consists in forming the reference network F_j^+, that is, an ideal network obtained by choosing the best value of each of the objectives as follows:

$$F_j^+ = \max_{i \in n} F_{ij} \tag{4}$$

Finally, the *gray relational grade (GRG) computation* step consists in calculating the similarity between each candidate network (i.e., the objective values of each optimal solution in the Pareto front) and the reference network F_j^+. The GRG for each i-th value in the Pareto Front is computed as follows:

$$GRG_i = \frac{1}{n} \sum_{j=1}^{m} w_j \frac{\Delta\min - \Delta\max}{\Delta_{ij} + \Delta\max} \tag{5}$$

where w_j is the weight of the j-th objective value (with $\sum_{j=1}^{m} w_j = 1$); $\Delta ij = |F_j^+ - F_{ij}|$ is the absolute value of the difference of between the j-th objective value in the reference network and the one in the candidate network; $\Delta\max = \max_{i \in n, j \in m}(\Delta ij)$ and $\Delta\min = \min_{i \in n, j \in m}(\Delta ij)$ are the maximum and minimum deltas, respectively.

The $conf \in X$ with the largest GRG_i is the recommended optimal solution outputed by the WGRA process. Depending on the set of weights used to extract the configuration from the Pareto front, the configuration shall map to a different state of the FSM, that is, it implements a different operation mode of the AI-based edge service.

To illustrate further, let us focus on two operation modes in the example application, namely, *power-saving* and *high-rate*. The engineer, jointly with domain experts [118], may provide the following sets of weights for the two operation modes, respectively: $W_{power\text{-}saving} = \{0.05, 0.9, 0.05\}$ and $W_{high\text{-}rate} = \{0.6, 0, 0.4\}$. The specific weights could be derived from a Service Level Agreement (SLA) defining the QoS, and the costs the application service provider to sustain and deliver the application.

Engineers could also define a set of objective thresholds t_j for each objectives O_j to reduce the size of the Pareto front given in input to the WGRA algorithm, filtering out solutions that might be unreasonable for a given operation mode op. In particular, a solution is filtered from the Pareto front if the value it achieved on objective O_j is above the threshold t_j.

For example, let us consider the *power-saving* and the *high-rate* operation modes again. The weights assigned to the $W_{power\text{-}saving}$ set must give a large importance to the energy consumption objective in order to extract an energy-efficient configuration. However, this may lead to the identification of a very poor but still non-dominated solution for the other two objectives. To prevent this risk, the engineer can filter all the solutions that do not provide a minimum detection accuracy level and/or number of processed frames. For instance, they can define a set of thresholds $T_{power\text{-}saving} = \{t_{acc}, t_{eng}, t_{rate}\} = \{0.2, 0, 60\}$ to exclude solutions with a detection accuracy lower than 0.2, and a number of processed frames lower than 60. A completely different set of thresholds could be defined for the *high-rate*, that is, $T_{high\text{-}rate} = \{0.3, 0, 0\}$. In this case, solutions with a detection accuracy lower than 0.3 are filtered out in order to provide a minimum detection accuracy level, when compared to the *power-saving* mode.

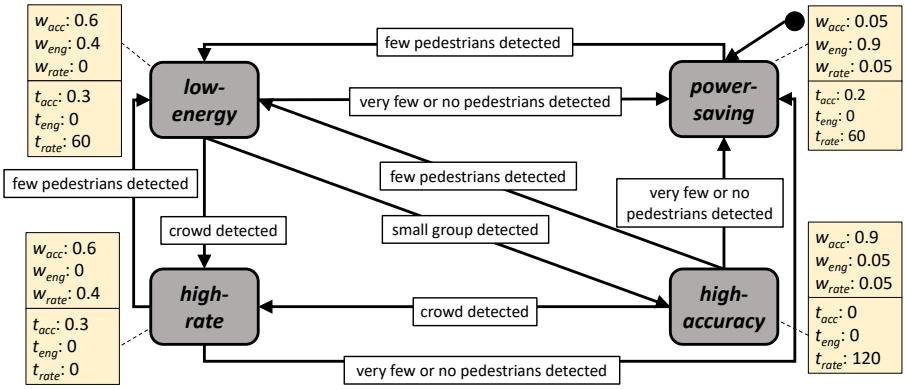


Figure 6.4: A refined version of the abstract state machine shown in Figure 6.3 with the set of weights and thresholds for each of the operation modes.

Figure 6.4 shows the refined version of the abstract FSM previously shown in Figure 6.3 with the weights and thresholds for WGRA analysis defined by the engineers attached to states. The chosen weights and thresholds represent the actual specification of the *desirable characteristics* of the operation modes listed in Table 6.1. The execution of the WGRA algorithm for each of the FSM state extracts a configuration $conf_{op}$ with the actual configuration parameter values that can be used by the SAA application to self-adapt the operation mode.

6.2.4 *Implementing the Self-Adaptive Application*

In the last step, the engineer is required to implement the self-adaptive application according to the output of the analysis. The abstract state machine is transformed into a concrete one in two steps: first, each of the transitions must be turned into an actual triggering condition; second, the operation mode configurations extracted in the previous step are mapped into a piece of logic able to set these config-

urations at runtime. Figure 6.5 shows the final FSM for the pedestrian detection scenario, with actual conditions and operation modes.

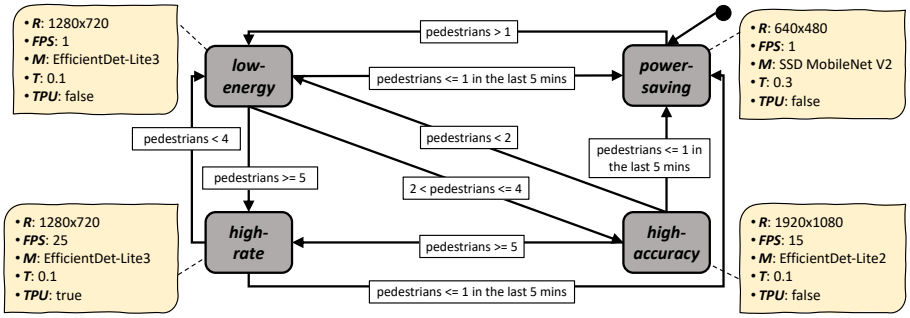


Figure 6.5: The concrete finite state machine implementing a self-adaptive application for the motivational scenario.

The FSM can be translated into working code using generators [196, 247] or when this is not possible or too difficult [6], the SAA can be obtained semi-automatically or manually [5, 6, 261]. The proposed approach outputs a concrete FSM encoding the SAA and does not bind the engineer to use any specific method to implement the SAA.

6.3 EMPIRICAL EVALUATION

In this section, I quantitatively evaluate the effectiveness of the meta-heuristic strategy and the capability of the proposed approach to release a better trade-off between application-level objectives and the energy consumption. I discuss the sub-research questions (Section 6.3.1), the experimental setup and the test-bed used to perform the experiments (Section 6.3.2), the results of the experiments to answer the sub-research questions (Section 6.3.3 and Section 6.3.4), and the threats to validity of the evaluation (Section 6.3.5).

6.3.1 *Research Questions*

This work responds to RQ4 and it is assessed with the following two sub-research questions in the context of the pedestrian detection scenario described in Section 6.1.

RQ4.1 - *Meta-Heuristic VS Near-Exhaustive Search*: Can the meta-heuristic search approach discover solutions whose quality is comparable to those obtained by a near-exhaustive search? This research question investigates the effectiveness of the proposed meta-heuristic strategy. In particular, it studies whether the heuristic exploration of a small portion of the search space can lead to results comparable to a near-exhaustive exploration.

RQ4.2 - *Objectives Trade-Off*: Can a self-adaptive pedestrians detection application better balance energy consumption and application objectives compared to a non-adaptive application? This research question investigates whether the self-adaptive application resulting

from the proposed methodology can release a better trade-off among accuracy, energy, and processing speed compared to four baseline non-adaptive applications.

6.3.2 *Experimental Setup*

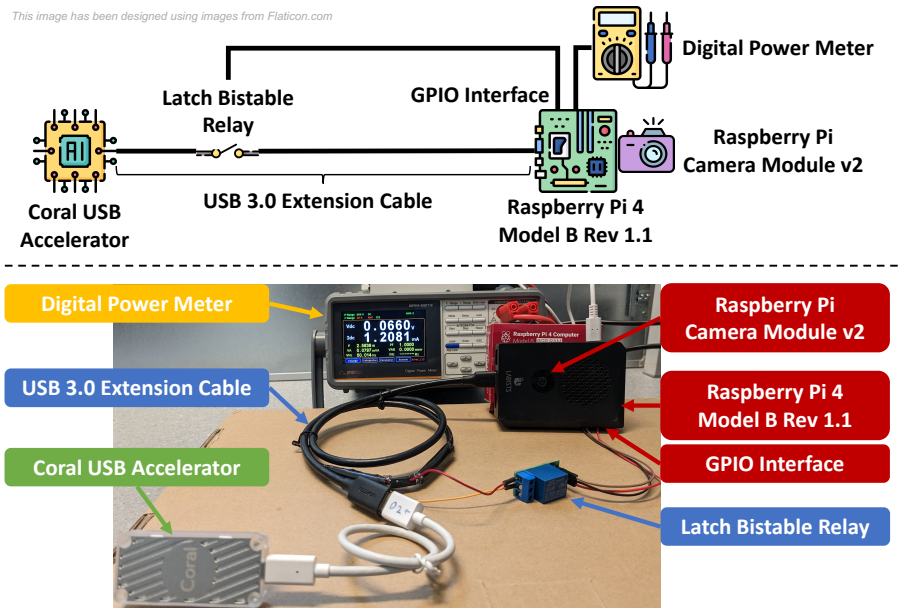


Figure 6.6: The test-bed used to run the evaluation experiments.

Figure 6.6 shows the test-bed I used to run the case study evaluation, first schematically (above), then its concrete in-lab implementation (below).

I employ a Raspberry Pi (RPi) 4 Model B Rev 1.1 (64-bit quad-core ARMv8, 4GB of RAM, RPi OS Lite 64-bit Debian GNU/Linux 11) equipped with the RPi Camera Module v2 and boxed in a LABISTS case with a 5V fan connected to the RPi General Purpose Input/Output (GPIO) interface. The RPi is powered by a USB-C AC adapter connected through a GW Instek GPM-8213 digital power meter [104] that I use to collect instant power values[1].

To reduce the idle energy consumption of RPi, I disable the unnecessary components: all the LEDs (i.e., activity, power, and Ethernet port), the Wi-Fi antenna, the Bluetooth, and the HDMI port. Internet and private network connectivity is provided via network cable. A Coral USB Accelerator (Edge TPU) [106] is plugged-in for those experiments that require hardware accelerator. The accelerator is automatically powered-on when connected to the USB port.

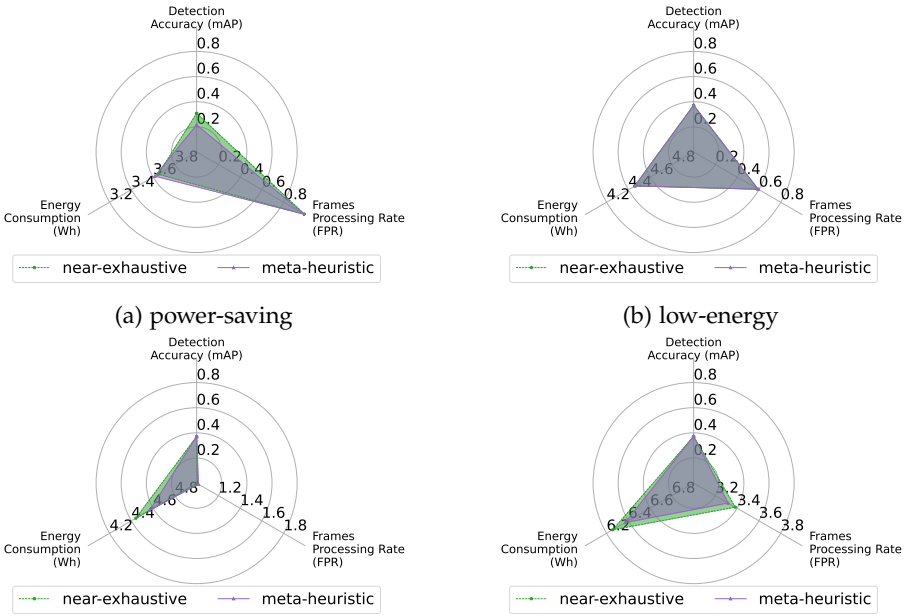
Since it not always possible to easily enable and disable a single USB port on-the-fly via software, a self-adaptive application running on such device would not be capable to completely power-off the accelerator when not in use, reducing the potential benefits of switching

1 The accuracy of the power measurements is reported Appendix B.

to an energy-efficient operation mode. To overcome this limitation, I realize a software-level power switch by employing a latch bi-stable relay (SONGLE SRD-05VC-SL-C) connected to the GPIO interface and a USB 3.0 extension cable. This enables us to turn it on and off by triggering the relay through software to close or open the circuit using a GPIO pin. For pedestrian detection, I employ state-of-art object detection models pre-trained on the COCO dataset [155]. The models are publicly available at the Coral.ai website [105], and they are already compiled for both CPU and Edge TPU execution.

The experimental material to fully reproduce the study, including instructions to recreate the test-bed based on Raspberry Pi, is available in a publicly accessible repository[2].

6.3.3 *RQ4.1 - Meta-Heuristic VS Near-Exhaustive Search*



(a) power-saving

(b) low-energy

(c) high-accuracy

(d) high-rate

Figure 6.7: Radar charts comparing the objective values of the four self-adaptive operation modes when employing a solution obtained with the meta-heuristic search procedure and one obtained with the near-exhaustive search procedure. The solutions are extracted with the WGRA method using the same set of weights and thresholds.

This research question aims to investigate whether exploring a small portion of the search space efficiently can lead to comparable results with a near-exhaustive exploration.

To answer RQ4.1, I first compute the Pareto front of the MOOP as defined in Equation 1 with the meta-heuristic search procedure by

2 https://gitlab.com/sustainable-continuum-monitoring/self-adaptive-moop/-/tree/ASE_2023?ref_type=tags

only exploring 10% of the search space reported in Table 6.2 (i.e., 340 unique trials out of 3402 trials), and then I explore more than 80% of the same space (i.e., 2790 unique trials out of 3402 trials) with a random search procedure. Second, I extract the four operation modes needed to address the pedestrian detection scenario (according to the weights and thresholds reported in Figure 6.4) from the two Pareto fronts: the one computed with the meta-heuristic search procedure and the one obtained with the near-exhaustive procedure. Finally, I compare the objective values achieved with the two SAAs that derive from the two sets of selected states. A good meta-heuristic procedure should be able to achieve results as good as the near-exhaustive exploration.

The whole optimization procedure is implemented with the Optuna framework [7], a state-of-art hyperparameter optimization framework with MOOP capabilities. The meta-heuristic search procedure with memory capabilities is realized by using the NSGAIISampler [184] implementing the NSGA-II algorithm and the results database provided by Optuna. I use the default framework values to configure the sampler and I repeat the search 10 times with a different seed value recorded for reproducibility. The near-exhaustive search procedure, instead, employs the RandomSampler [185].

At each optimization round, when a sampler selects a point `conf` from the search space, two experiments must be executed to determine the objective values for the selected `conf`.

The first experiment computes the *detection accuracy* by employing a pedestrian street scene belonging to the Multiple Object Tracking benchmark dataset [75] (i.e., the ADL-Rundle-6 video). Both the frame size and the ground truth have been properly adjusted to match the camera resolution (R) parameter values defined by the search space.

I use the Mean Average Precision (mAP) as detection accuracy metric, a popular metric for object detection algorithms [188], and I evaluate the model predictions by using the open-source FiftyOne COCO-style evaluator [252].

The second experiment, instead, computes both the achieved *Frames Processing Rate* (FPR) and the *energy consumption*. I run the pedestrian detection application on the device (i.e., the Raspberry Pi described in Section 6.3.2 for 120 seconds configured according to *conf*). I collect both the consumed energy in Watt-hours (Wh) and the FPR computed as the ratio between the number of processed frames and the experiment duration.

RESULTS OF RQ4.1 The near-exhaustive search executed for about 18 days sampling 2790 unique trials and discovered a Pareto front with 131 solutions. The meta-heuristic search executed for about 54 hours sampling 340 unique trials (10% of the entire space) and discovered a Pareto front with 83 solutions on average. Note that the saving, when

the sampling involves running experiments, is significant in both relative and absolute terms (more than 2 weeks of computing saved).

Since each run of the meta-heuristic search may return a slightly different configuration for a given state, I selected the configuration that occurred most frequently in the 10 repetitions to derive the corresponding SAA. When multiple solutions have the same highest frequency, I excluded the solution matching the one extracted from the near-exhaustive Pareto front to avoid any bias, and consider a worst case scenario.

Figure 6.7 shows four radar charts - one per each operation mode in the SAA - comparing the three objective values of the solution extracted with the near-exhaustive search (green, dashed, dot mark), and the one extracted from meta-heuristic search (purple, solid line, triangle mark), respectively. Each of the axes has its own scale, but for all the objectives, the higher is the value the better it is.

The plots clearly indicate that the states identified by the meta-heuristic search procedure and the ones obtained with the near-exhaustive search result in highly similar performance. The *low-energy* operation mode (Figure 6.7b) resulted in exactly the same solution returned by the two procedures. While the near-exhaustive search identified solutions performing comparably to the ones identified by the meta-heuristic search in the remaining three operation modes.

In the case of the *power-saving* operation mode (Figure 6.7a), the two solutions perform with the same FPR and with negligible difference in energy consumption ($< 1\%$). The difference is slightly larger for the detection accuracy (0.307 mAP VS 0.215 mAP), whose relevance in the power-saving mode is however limited.

In the case of the *high-accuracy* operation mode (Figure 6.7c), the two solutions perform with the same detection accuracy, and with negligible differences for FPR($< 1\%$) and energy consumption (4.442 Wh VS 4.570 Wh).

Finally, the two solutions obtained for *high-rate* (Figure 6.7d) perform with the same detection accuracy, and with negligible differences for both FPR and the energy consumption ($< 2\%$).

ANSWER TO RQ4.1 The search procedure has been as effective as the near exhaustive procedure for the pedestrian detection scenario, despite an empirical exploration of only 10% of the search space.

6.3.4 *RQ4.2 - Objectives Trade-Off*

This research question aims to investigate whether a self-adaptive application changing its operation mode can better balance the fulfillment of multiple objectives compared to a non-adaptive application using a single operation mode.

I study this research question in the context of two pedestrian traffic scenarios, namely, *weekdays* and *weekends*, derived from real-world

traffic shapes reported by Dobler *et al.* [78] in their work about urban pedestrians dynamic in the borough of Manhattan. In particular, the *weekdays* scenario has a 3-peaks structure aligned with the "9-to-5" workday time, in which the peaks correspond to commuting to work, exiting buildings at lunch time, and leaving the work place. The *weekend* scenario does not show a peaked structure, but rather a steady increase of pedestrians until the night.

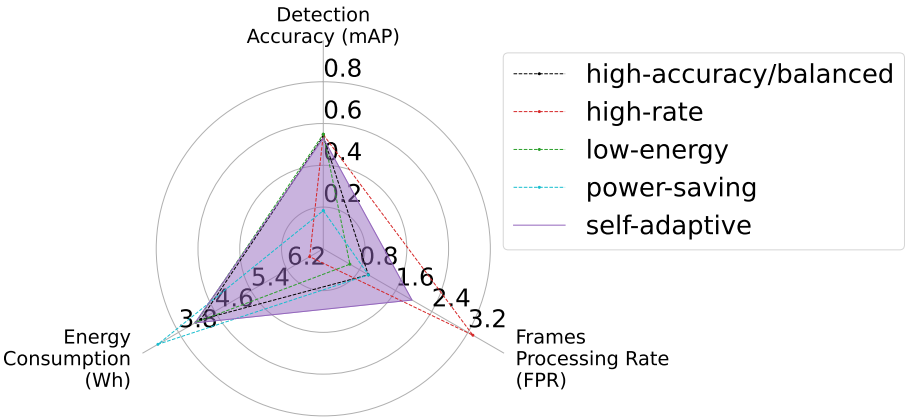
I create a scenario by selecting 1440 frames, that is, 60 frames per hour, from a pool 115 of manually annotated frames containing between 0 and 5 pedestrians. Each hour of the day is labeled as 0 pedestrians, 1 to 3 pedestrians, and 4 to 5 pedestrians. The frames used for the experiment are taken from a study about real-time analytics for traffic safety [160].

I implement a self-adaptive pedestrian detection application according to the FSM depicted in Figure 6.5 using the Python State Machine library [112]. Then, I use the same pedestrian detection logic to obtain the non-adaptive baseline application. The four operation mode configurations obtained by the meta-heuristic search procedure in RQ4.1 are used to configure both the self-adaptive application and the non-adaptive baselines, obtaining four non-adaptive applications. Figure 6.5 shows the configuration parameter values. Further, I include in the study a non-adaptive configuration, namely the *balanced* configuration, that assigns the same weight (0.33) to the three objectives and uses the thresholds ($t_{acc} = 0.3, t_{eng} = 0, t_{rate} = 120$) that filter out the same unsatisfactory configurations collectively filtered out by the four operation modes of the adaptive approach. This configuration implements the best attempt to balance all the objectives without introducing any self-adaptation logic. Interestingly, the *balanced* configuration matches the *high-accuracy* configuration, that is, high-accuracy can be released maintaining a good level of energy consumption and frame rate.

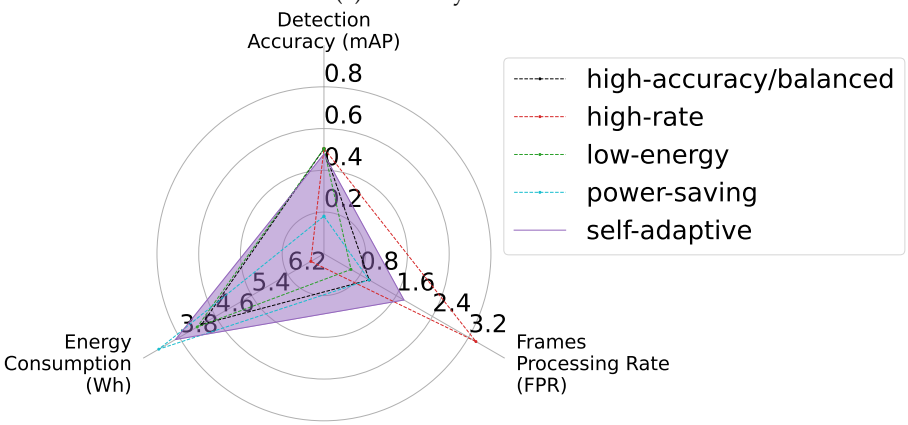
I evaluate the resulting self-adaptive and non-adaptive applications by using the same set of metrics used for RQ4.1, that is, the MOOP objectives: detection accuracy (mAP), energy consumption (Wh), and FPR.

RESULTS OF RQ4.2 Figure 6.8 compares the performance of the SAA (purple, solid line) with the four non-adaptive applications (black/red/green/cyan, dotted lines) in both the *weekdays* (Figure 6.8a) and *weekends* (Figure 6.8b) scenarios. As for the radar charts in Figure 6.7, the higher the better.

The shape of the triangle in both the radar charts visually shows how the adaptive behavior guarantees the achievement of a better trade-off among the three objectives compared to the non-adaptive behavior. It outperforms three out of four non-adaptive applications regarding both energy consumption (i.e., *low-energy*, *high-accuracy*/*balanced*, *high-rate*) and FPR (i.e., *power-saving*, *low-energy*,



(a) Weekdays Scenario



(b) Weekends Scenario

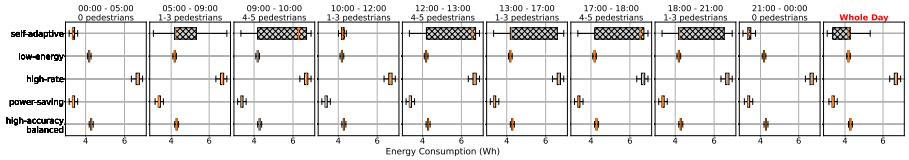
Figure 6.8: Radar charts comparing the SAA and the 4 non-adaptive applications in the weekdays and weekends scenarios.

high-accuracy/balanced), and one out of four w.r.t. the detection accuracy (i.e., *power-saving*). Notably, it is still able to guarantee a similar accuracy when compared to the other three non-adaptive applications (i.e., *low-energy*, *high-accuracy/balanced*, *high-rate*).

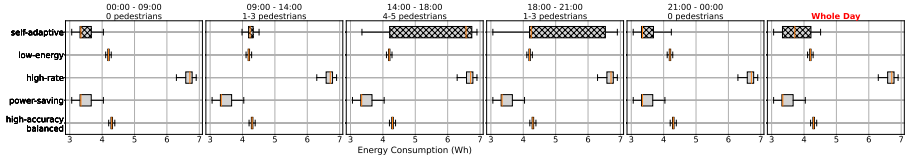
In particular, compared to the best/worst non-adaptive operation mode, the SAA is able to save between 0.5% and 61% of energy in the *weekdays* scenario, and between 13% and 81% in the *weekends* scenario. The improvement on the FPR is between 96% and 233% in the *weekdays* scenario, and between 77% and 196% in the *weekends* scenario. The accuracy loss is between 2% and 4% in the *weekdays* scenario, and between 5% and 6% in the *weekends* scenario, but the SAA outperforms the *power-saving* application with a gain in the accuracy between 62% and 189%.

The SAA performed slightly differently in the two scenarios. In fact, the presence of a 3-peaks structure with a higher number of pedestrians in the *weekdays* scenario makes the self-adaptive application to use more accurate and faster operation modes (i.e., *high-accuracy* and *high-rate*) for a larger amount of time, resulting in a higher FPR at the cost of a higher energy consumption. On the other hand, the traffic shape

of the *weekends* scenario fosters the usage of energy efficient operation modes (i.e., *power-saving* and *low-energy*), resulting in a lower energy consumption and slower processing speed.



(a) Weekdays Scenario



(b) Weekends Scenario

Figure 6.9: Box-plots comparing energy consumption for the self-adaptive and the four non-adaptive applications.

This behavior is also confirmed by the energy consumption box plots shown in Figure 6.9a and Figure 6.9b. The two figures show the energy consumption of the self-adaptive application and the four non-adaptive applications in different time windows of the day for both the scenarios. The vertical orange line in the boxes indicates the median value.

It is possible to observe how the self-adaptive application captures correctly the 3-peaks structure in the *weekdays* scenario (Figure 6.9a) and uses the *high-rate* in these three time windows. At the same time, it employs energy efficient operation modes (i.e., *power-saving* and *low-energy*) when the pedestrians traffic is less intense (e.g., 00:00 - 05:00 and 21:00 - 00:00). A similar behavior is obtained in the *weekends* scenario shown in Figure 6.9b.

ANSWER TO RQ4.2 The SAA application can employ more accurate operation modes when the pedestrians workload is higher, using less accurate operation modes (i.e., *power-saving*) when the pedestrians workload is less demanding. In a nutshell, the self-adaptive solution is consuming energy only when it is worth doing it.

6.3.5 *Threats to Validity*

First, the design of the FSM requires the definition of a set of operations modes characterized by weights and thresholds, and the definition of state transition conditions. This is a manual and non-trivial operation guided by domain-expert knowledge that can limit the feasibility of the approach and lead to different results. Nevertheless, the reported results show how a SAA can largely outperform non-adaptive baselines, regardless of the specific configuration used.

Second, the design of the pedestrian traffic shapes may have an impact on the results. To mitigate this threat, I referred to real scenarios to achieve realistic and informative results.

Finally, the results may not generalize to other application domains. Indeed, a *case study* evaluation focusing on AI-services for pedestrian detection running at the edge has been proposed, and the design of a SAA addressing a different problem may produce different results. Although the methodology and the approach are general, it is not possible to claim the results shall straightforwardly generalize to other contexts. The illustrated case study nevertheless provides evidence that the proposed approach can generate useful results in non-trivial domains such as pedestrian detection, which requires to balance high-speed computations (e.g., video-processing) with energy saving requirements.

6.4 DISCUSSION

The presented approach can guide developers in implementing energy-aware, AI-based, self-adaptive monitoring systems that can switch operation modes in response to changes in the environment. Balancing application-level objectives, such as accuracy, with energy consumption is a particularly relevant problem for systems deployed at the Edge, where resource-constrained devices are in place and may be powered by unreliable sources. Practitioners should consider applying the proposed approach if their goal is to obtain trade-offs that provide the optimal compromise for multiple application scenarios, rather than always striving for the best outcome for any individual application objective in all scenarios, which is usually unfeasible in real-world problems. The operation mode configurations of the resulting SAA are determined empirically through a meta-heuristic search procedure that samples a small portion of the configuration space to identify useful configurations. Empirical results demonstrate that the proposed approach can outperform non-adaptive baseline configurations and behave as optimally as configurations selected with a nearly exhaustive exploration of the configuration space in a pedestrian detection scenario. Specifically, the study reveals that the self-adaptive application can save up to 81% of energy compared to non-adaptive baseline configurations, with a minimal loss of accuracy ranging from 2% to 6%. Additionally, the meta-heuristic search procedure only needed to sample 10% of the configuration space to identify the operation mode configurations.

The main limitation of the proposed approach is the manual definition of the FSM. The FSM design is a critical part of the approach that can influence the final outcome, and therefore deserves specific attention and engagement of all parties involved in the application development. Therefore, practitioners should always involve domain experts and stakeholders in the design process to discover application re-

quirements and capture the information that can be translated into the technical implementation of the FSM (i.e., the operation modes, transitions, and optimal set of weights and thresholds). To overcome this limitation, automating the FSM design and synthesis through data-driven methods would be beneficial in reducing the effort required by engineers.

7

CONCLUSIONS

Monitoring is a key task performed in several fields as it can aid in understanding the behavior of the monitored targets. Monitoring systems are increasingly being deployed throughout the cloud continuum, a distributed and heterogeneous environment with varying software and hardware stacks designed to be simultaneously accessible in a multi-tenant fashion [176]. Its fog and edge computing layers exhibit lower network latency and greater responsiveness when compared to the upper cloud layer, balanced by lower reliability, due to the prevalence of wireless connectivity, and fewer computational capabilities, due to limited device resources [263].

Managing monitoring systems in the cloud continuum presents several challenges to automation and energy consumption. This thesis investigates how to automate monitoring system configurations in response to dynamic needs and technological constraints, and how to efficiently use available resources.

In particular, the first challenge concerns with *adapting monitoring systems to evolving requirements* based on operators' needs and the characteristics of the cloud continuum. It is essential to support the automated evolution of the monitoring system to accommodate changes in operators' needs due to unpredictable events such as anomalies, failures, and requests for new indicators. Furthermore, as the cloud continuum is a heterogeneous environment used by multiple tenants, a monitoring system should abstract from underlying technologies and relieve operators from the configuration burden [1, 4].

To address the dynamicity and evolution of monitoring systems, researchers and practitioners have focused on increasing the level of automation of probe deployments by studying Monitoring-as-a-Service (MaaS) solutions [4, 87, 192, 238]. However, there is currently no general MaaS solution that can be used to collect virtually any indicator on any platform. Current approaches significantly limit both the range of platforms and indicators that can be used.

Contribution 1: ***Automated Probe Life-Cycle Management*** To address the aforementioned gap, this thesis proposes a *Monitoring-as-a-Service framework that utilizes a catalog of probes annotated with metadata and access to the API of the environment running the targets to monitor*. This framework provides full MaaS capabilities, including error-handling. The results indicate that the framework is effective for both containers and VMs, with virtual machines taking slightly less than a minute and containers taking less than 1.5 seconds. The framework also demonstrates efficient error-handling, with containers taking only a few sec-

onds to recover for erroneous configurations. Additionally, the framework is scalable for an increasing number of operators' requests.

The monitoring systems and probe technologies offer flexibility in deploying probes, allowing for diverse probe deployment patterns. These patterns consist of probe deployment architectures that target specific environments, such as container-based environments, and satisfy specific constraints, such as the need for probes to be shared among multiple operators. The effectiveness and efficiency of the resulting monitoring system can be impacted by the choice of probe deployment pattern. However, there has not been a systematic analysis and assessment of the many possible patterns available.

Contribution 2 ***Probe Deployment Patterns*** *This thesis presents the definition, analysis, and qualitative and quantitative evaluation of 11 possible probe deployment patterns* to fill this knowledge gap. The results indicate the trade-offs between patterns that require more resources to ensure good separation between users in multi-tenant environments and patterns that make better use of resources while reducing the degree of separation. The findings have been cross-validated by addressing three realistic monitoring scenarios. Furthermore, results helped in distilling probe deployment best practices with valuable insights that can guide engineers in implementing and configuring their monitoring systems.

This thesis investigates a second research challenge, which concerns the *adaptation of monitoring systems to the available resources*. This is particularly relevant in the context of fog and edge environments, where a monitoring system must efficiently use available resources to handle an increasing number of running devices, applications, and collected indicators that produce a significant amount of data for storage and analysis [1, 228].

Recently, monitoring approaches specifically designed for the fog environment have been investigated [46, 91, 109, 222]. However, none of these solutions implement a comprehensive adaptive solution that can support a diverse range of adaptive behaviors, such as changing the set of collected indicators or updating the configuration parameters based on current data trends.

Contribution 3: ***Peer-to-Peer Self-Adaptive Monitoring in the Fog*** This thesis presents a *self-adaptive P2P monitoring system for fog environments that utilizes a hierarchical P2P architecture and incorporates adaptive behaviors based on the MAPE-K feedback loop*. The monitoring system can abstract monitored indicators and activate countermeasures based on their status. Countermeasures are defined using a lightweight rule-based system embedded in the peers. The findings indicate that utilizing adaptive behaviors can enhance the precision of gathered data (up to -82.7% RMSE in unstable scenarios) while also reducing network usage (up to -51.7% messages/second in stable scenarios) and power consumption (up to approximately -36%). However, this comes at the expense of increased memory consumption (up to +10%).

In addition, it is crucial for a monitoring system to function effectively in unpredictable and possibly resource-limited conditions. This requires ensuring its capabilities while utilizing available resources wisely, which may be scarce at the edge of the network [216]. Researchers have investigated several approaches to optimize the energy consumption, including low-level task optimization such as scheduling and provisioning [14, 18, 98, 179, 219], architectural tactics [63], and specific adaptive behaviors like sampling, filtering, and compression [99, 159].

Contribution 4: ***Energy-Aware Self-Adaptive Monitoring in the Edge*** In contrast to prior work, this thesis proposes *an approach that considers energy consumption and guides developers to implement a self-adaptive application capable of switching operation modes in response to changes in the environment*. This ultimately balances energy consumption with application-level objectives, such as monitoring accuracy. The configuration of operation modes is determined empirically through a metaheuristic search procedure that samples a small portion of the configuration space to identify useful configurations. The experimental results indicate that the proposed approach outperforms non-adaptive baseline configurations and behaves optimally, similar to configurations selected with a nearly exhaustive exploration of the configuration space. This is achieved by saving up to 81% of energy while losing only between 2% and 6% in accuracy.

Practitioners can exploit the contributions provided by this thesis either independently or in combination to create a more comprehensive, automated, and adaptive monitoring process. The proposed approaches can be opportunistically combined in a cloud continuum monitoring framework that automates probe deployment according to different patterns and adapts monitoring configurations (i.e., probe deployment pattern, probe configurations, and system-wide configurations) in response to changes in collected data, available resources, and the operational environment. Optimization techniques and finite-state machines are recommended for discovering a discrete set of monitoring configurations and designing adaptive behavior in resource-constrained environments. This ensures that the monitoring system uses available resources wisely while maintaining regular its functionality.

OPEN CHALLENGES AND RESEARCH DIRECTIONS The results reported in this thesis advanced knowledge in monitoring in the cloud continuum, but also opened new challenges for the community.

Probe-Level Adaptation and Configurability The contributions on automated probe deployments and deployment patterns demonstrated that MaaS solutions can aid operators in configuring monitoring systems and managing the configuration burden. However, in some scenarios, the centralized and human-centered redeployment of probes can be time-consuming. Although a centralized decision point is un-

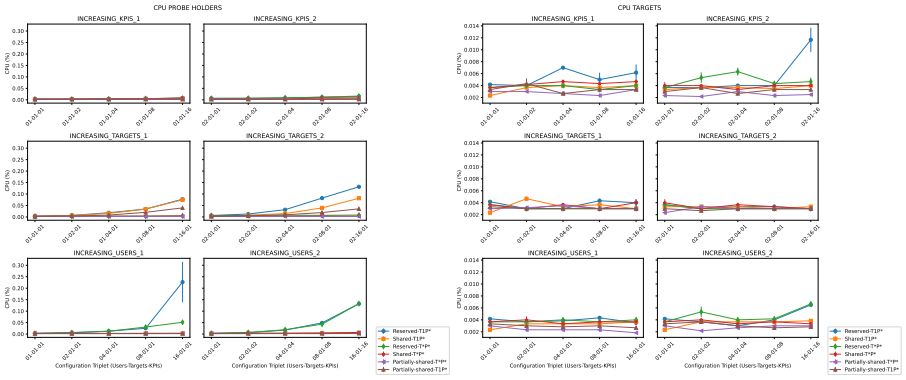
doubtedly useful, it can also be too slow for some cloud continuum scenarios, such as volatile edge environments [28, 120]. Thus, the first challenge pertains the design of configurable and adaptive monitoring behaviors at the probe-level for a rapid local adaptation. To offer controllable and configurable self-adaptive capabilities, probes have to be carefully designed, incorporating managers, which can reconfigure the components in the probe, and variability points, which offer multiple options to flexibly fit the various use cases that can be faced at run-time. This would enable monitoring systems to adapt to a wide range of scenarios without predefined configurations and offer fine-grained configurability for probes deployed in the field.

Semi-Automatic Adaptive Behavior Definition The studies on self-adaptive monitoring systems for fog and edge environments demonstrated the effectiveness of adaptive behaviors in responding to changes in the operating environment and optimizing resource utilization. However, the definition and configuration of these behaviors still rely on expert knowledge. This is a critical aspect because defining triggering events, countermeasures, and configuration parameters can be a cumbersome and application-dependent task [257]. Relying solely on engineers' knowledge can be difficult and may not work in all application scenarios due to the unpredictability of the cloud continuum environment. Therefore, the second challenge concerns with the definition of adaptive behaviors exploiting (semi-)automatic techniques. A potential approach to investigate involves utilizing continual reinforcement learning techniques [142] to capture current behavioral patterns of the application and environment. These patterns can then be utilized to define and adjust adaptive behaviors in a (semi-)automated manner.

Accurate Software-Level Power Models Power estimations can be obtained through simulations or relevant software executions in the field, as demonstrated in the last contribution presented in this thesis. However, continuous changes to the monitoring software may impact the accuracy of previous measurements, resulting in an outdated solution that require additional executions to collect new energy data. The third challenge concerns with deriving precise power models that help in defining optimal trade-offs between energy consumption and application-level objectives, such as monitoring accuracy. Unfortunately, implementing energy-aware solutions at the software level is challenging due to the difficulty of obtaining accurate energy consumption information at run-time for specific sections of the software code. Improvements have been made in this area [44, 89], but the solutions are still limited to measuring hardware-level APIs and do not extend beyond the process level. This limitation hinders the ability to gain a clear understanding of how the execution of a specific part of the code, for example, as a result of an adaptive behavior, will impact energy consumption.

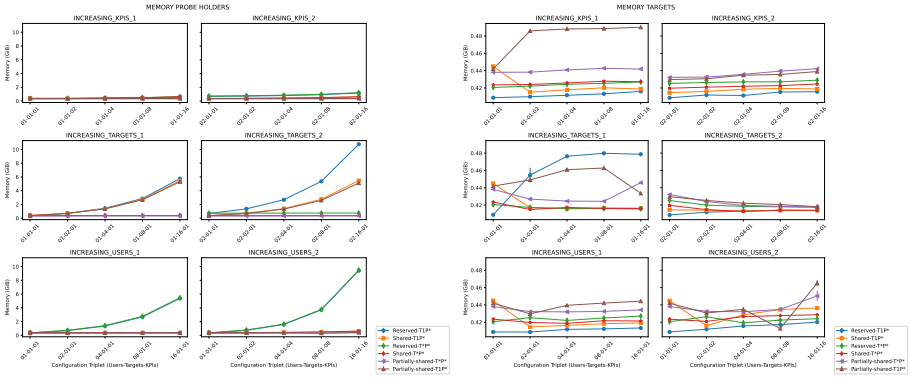
A

PROBE DEPLOYMENT PATTERN PLOTS



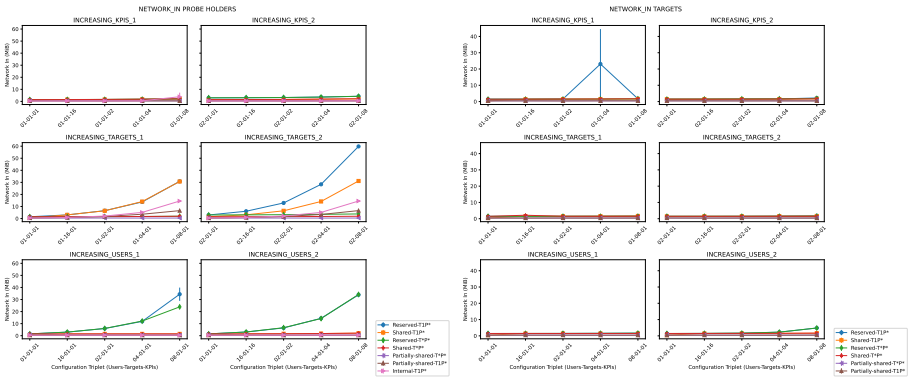
(a) (b)

Figure A.1: System-oriented CPU Consumption



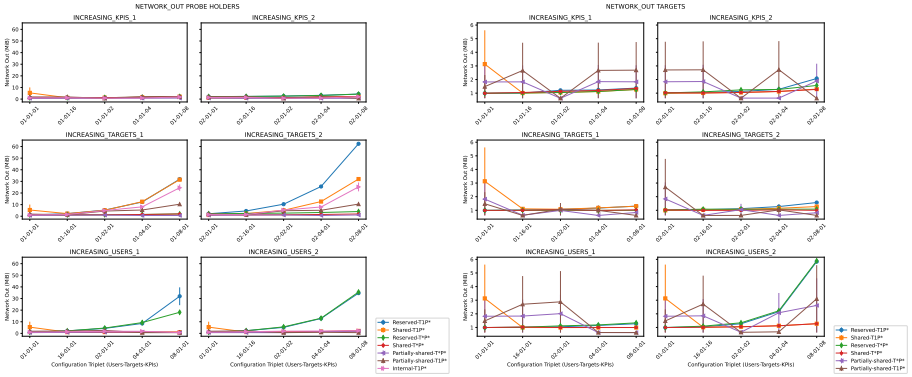
(a) (b)

Figure A.2: System-oriented Memory Consumption



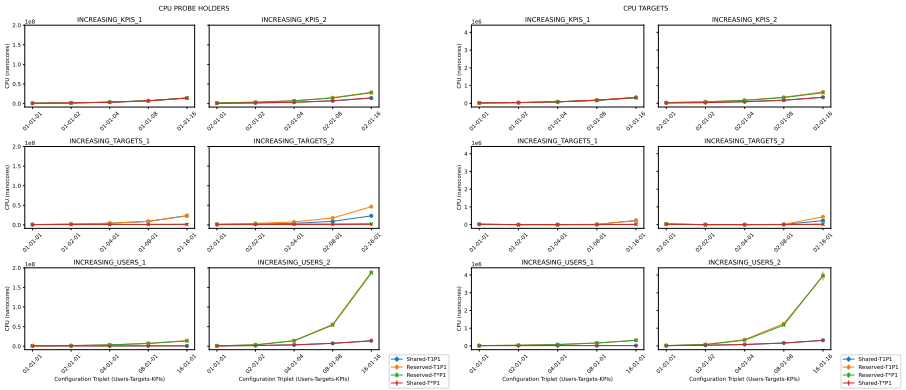
(a) (b)

Figure A.3: System-oriented Network Input Consumption



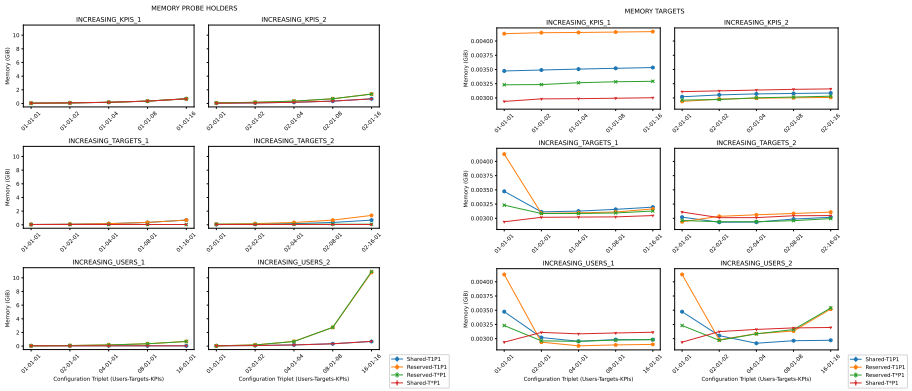
(a) (b)

Figure A.4: System-oriented Network Output Consumption



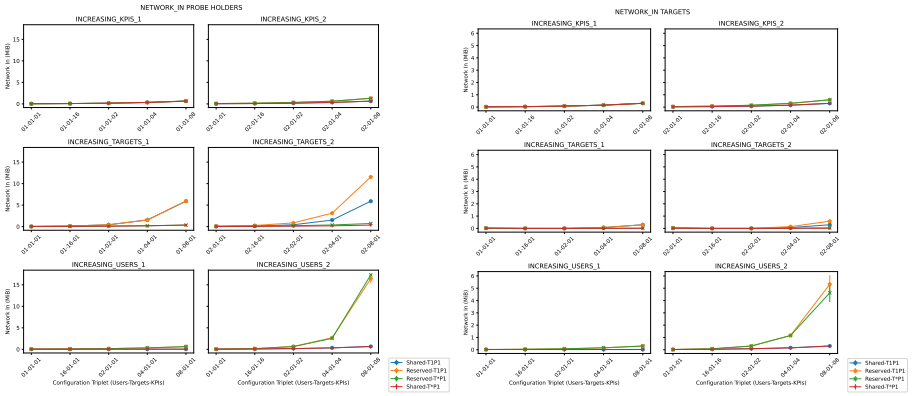
(a) (b)

Figure A.5: Application-oriented CPU Consumption



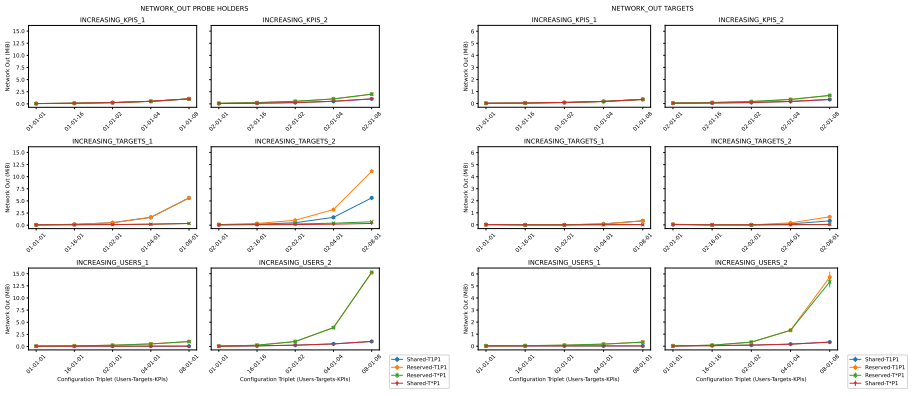
(a) (b)

Figure A.6: Application-oriented Memory Consumption



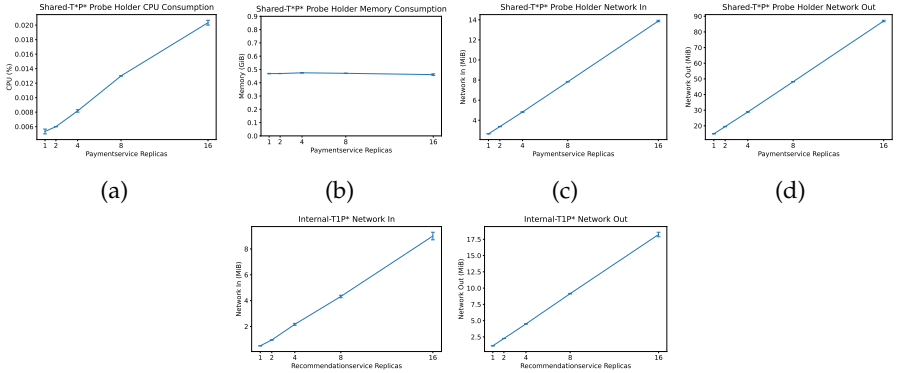
(a) (b)

Figure A.7: Application-oriented Network Input Consumption



(a) (b)

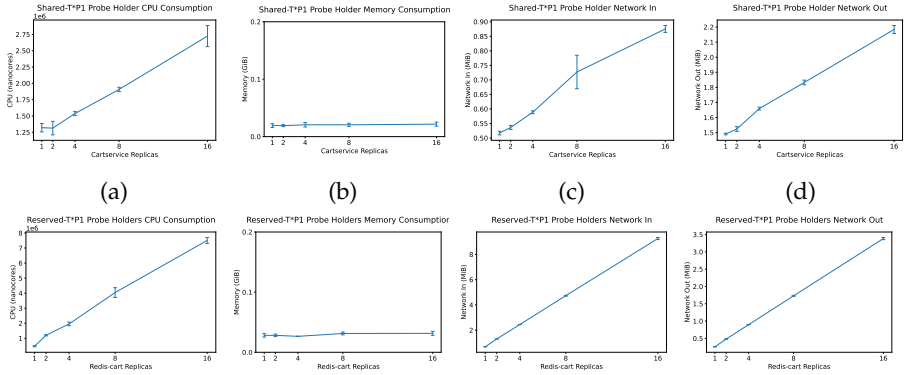
Figure A.8: Application-oriented Network Output Consumption



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

(e) (f)

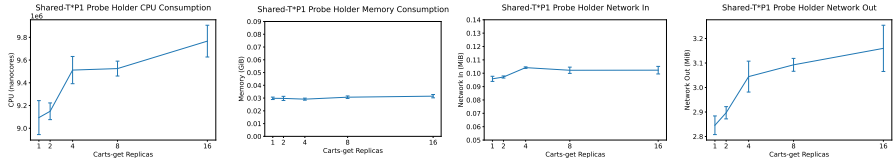
Figure A.9: Monitoring a VM-based Microservice Application Usage Scenario



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

(e) (f) (g) (h)

Figure A.10: Monitoring a Microservice Application Running on Kubernetes Usage Scenario



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

Figure A.11: Monitoring Serverless Backend Functions Usage Scenario

B

GW INSTEK GPM-8213 POWER MEASUREMENT ACCURACY

Table B.1: GW Instek GPM-8213 Power Measurement Accuracy

Effective Range	1% to 110% of range
DC	±(0.2% reading + 0.2% range)
45 Hz < f ⩽ 66 Hz	±(0.1% reading + 0.1% range)
66 Hz < f ⩽ 1 kHz	±(0.1% reading + 0.2% range)
1 kHz < f ⩽ 6 kHz	± 3% of range
The filter is turned on	Increase 0.3% reading @ 45Hz to 66Hz

BIBLIOGRAPHY

[1] Mohamed Abderrahim, Meryem Ouzzif, Karine Guillouard, Jerome Francois, and Adrien Lebre. "A holistic monitoring service for fog/edge infrastructures: a foresight study." In: *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE. 2017.

[2] Mohamed Abderrahim, Meryem Ouzzif, Karine Guillouard, Jérôme François, Adrien Lebre, Charles Prud'homme, and Xavier 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.

[3] Muhammad Abrar, Ushna Ajmal, Ziyad M Almohaimeed, Xiang Gui, Rizwan Akram, and Roha Masroor. "Energy efficient UAV-enabled mobile edge computing for IoT devices: A review." In: *IEEE Access* 9 (2021), pp. 127779–127798.

[4] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. "Cloud monitoring: A survey." In: *Computer Networks* 57.9 (2013), pp. 2093–2115.

[5] Paul Adamczyk. "The anthology of the finite state machine design patterns." In: *The 10th Conference on Pattern Languages of Programs*. 2003.

[6] Paul Adamczyk. "Selected patterns for implementing finite state machines." In: *The 11th Conference on Pattern Languages of Programs*. 2004.

[7] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. "Optuna: A Next-generation Hyperparameter Optimization Framework." In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[8] Saravanan Alagarsamy, S Ramkumar, Kartheeban Kamatchi, Hari Shankar, Ajith Kumar, Sanjeev Karthick, and Praveen Kumar. "Designing a Advanced Technique for Detection and Violation of Traffic Control System." In: *Journal of Critical Reviews* 7.8 (2020), pp. 2874–2879.

[9] Carlos Albuquerque, Kadu Relvas, Filipe Figueiredo Correia, and Kyle 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.

[10] Iván Alfonso, Kelly Garcés, Harold Castro, and Jordi Cabot. "Modeling self-adaptative IoT architectures." In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion*. IEEE. 2021, pp. 761–766.

[11] 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: *Proceedings of the IEEE International Conference on Services Computing*. 2014, pp. 283–290.

[12] Khalid Alhamazani, Rajiv Ranjan, Prem Prakash Jayaraman, Karan Mitra, Chang Liu, Fethi Rabhi, Dimitrios Georgakopoulos, and Lizhe Wang. "Cross-layer multi-cloud real-time application QoS monitoring and benchmarking as-a-service framework." In: *IEEE Transactions on Cloud Computing* 7.1 (2015), pp. 48–61.

[13] Khalid Alhamazani, Rajiv Ranjan, Karan Mitra, Fethi Rabhi, Prem Prakash Jayaraman, Samee Ullah Khan, Adnene Guabtni, and Vasudha Bhatnagar. "An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art." In: *Computing* 97.4 (2015), pp. 357–377.

[14] Abdullah Fawaz Aljulayfi and Karim Djemame. "A Novel QoS and Energy-aware Self-adaptive System Architecture for Efficient Resource Management in an Edge Computing Environment." In: *Proceedings of the 35th UK Performance Engineering Workshop*. 2019, p. 39.

[15] Robert S Allison, Joshua M Johnston, Gregory Craig, and Sion Jennings. "Airborne optical and thermal remote sensing for wildfire detection and monitoring." In: *Sensors* 16.8 (2016), p. 1310.

[16] Amazon Web Services. *AWS Prescriptive Guidance Patterns*. 2023. URL: https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/. (accessed: 26.05.2023).

[17] Sasan Amini, Ilias Gerostathopoulos, and Christian Prehofer. "Big data analytics architecture for real-time traffic control." In: *2017 5th IEEE international conference on models and technologies for intelligent transportation systems*. IEEE. 2017, pp. 710–715.

[18] Atakan Aral, Vincenzo De Maio, and Ivona Brandic. "ARES: Reliable and sustainable edge provisioning for wireless sensor networks." In: *IEEE Transactions on Sustainable Computing* 7.4 (2021), pp. 761–773.

[19] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. "Formal design and verification of self-adaptive systems with decentralized control." In: *ACM Transactions on Autonomous and Adaptive Systems* 11.4 (2017), pp. 1–35.

[20] Luca Ardito. "Energy aware self-adaptation in mobile systems." In: *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 1435–1437.

[21] Luca Ardito, Marco Torchiano, Marco Marengo, and Paolo Falcarin. "gLCB: an energy aware context broker." In: *Sustainable Computing: Informatics and Systems* 3.1 (2013), pp. 18–26.

[22] Amna Arouj and Ahmed M Abdelmoniem. "Towards energy-aware federated learning on battery-powered clients." In: *Proceedings of the 1st ACM Workshop on Data Privacy and Federated Learning Technologies for Mobile Edge Network*. 2022, pp. 7–12.

[23] Governors Highway Safety Association. *Pedestrian Traffic Fatalities by State: 2022 Preliminary Data*. 2023. URL: https://www.ghsa.org/resources/Pedestrians23. (accessed: 08.01.2024).

[24] Lisa Aultman-Hall, Damon Lane, and Rebecca R Lambert. "Assessing impact of weather and season on pedestrian traffic volumes." In: *Transportation research record* 2140.1 (2009), pp. 35–43.

[25] Prometheus Authors. *Exporters and integrations*. 2023. URL: https://prometheus.io/docs/instrumenting/exporters/. (accessed: 08.01.2024).

[26] Prometheus Authors. *Prometheus*. 2024. URL: https://prometheus.io/. (accessed: 08.01.2024).

[27] The Kubernetes Authors. *Kubernetes*. 2024. URL: https://kubernetes.io. (accessed: 08.01.2024).

[28] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. "Adaptive Management of Volatile Edge Systems at Runtime With Satisfiability." In: *ACM Transactions on Internet Technology* 22.1 (2021).

[29] Pavel Avgustinov, Julian Tibble, and Oege de Moor. "Making Trace Monitors Feasible." In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM. 2007, 589–608.

[30] Elasticsearch B.V. *Beats: Data Shippers for Elasticsearch*. 2023. URL: https://www.elastic.co/beats. (accessed: 08.01.2024).

[31] Elasticsearch B.V. *ElasticStack*. 2024. URL: https://www.elastic.co/elastic-stack. (accessed: 08.01.2024).

[32] Elasticsearch B.V. *Elasticsearch*. 2024. URL: https://www.elastic.co/elasticsearch. (accessed: 08.01.2024).

[33] Elasticsearch B.V. *Metricbeat: Lightweight Shipper for Metrics*. 2024. URL: https://www.elastic.co/beats/metricbeat. (accessed: 08.01.2024).

[34] Ahmet Cihat Baktir, Atay Ozgovde, and Cem Ersoy. "How can edge computing benefit from software-defined networking: A survey, use cases, and future directions." In: *IEEE Communications Surveys & Tutorials* 19.4 (2017), pp. 2359–2391.

[35] Daniel Balouek-Thomert, Eduard Gibert Renart, Ali Reza Zamani, Anthony Simonet, and Manish Parashar. "Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows." In: *The International Journal of High Performance Computing Applications* 33.6 (2019), pp. 1159–1174.

[36] Luciano Baresi, Danilo Filgueira Mendonça, Martin Garriga, Sam Guinea, and Giovanni Quattrocchi. "A unified model for the mobile-edge-cloud continuum." In: *ACM Transactions on Internet Technology (TOIT)* 19.2 (2019), pp. 1–21.

[37] Wolfgang Barth. *Nagios: System and network monitoring*. No Starch Press, 2008.

[38] Johan Barthélemy, Nicolas Verstaevel, Hugh Forehead, and Pascal Perez. "Edge-computing video analytics for real-time traffic monitoring in a smart city." In: *Sensors* 19.9 (2019), p. 2048.

[39] Pete Beckman, Jack Dongarra, Nicola Ferrier, Geoffrey Fox, Terry Moore, Dan Reed, and Micah Beck. "Harnessing the Computing Continuum for Programming Our World." In: *Fog Computing*. John Wiley & Sons, Ltd, 2020. Chap. 7, pp. 215–230.

[40] D. Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes." In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[41] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz DaSilva, Craig Lee, and Omer Rana. "The internet of things, fog and cloud continuum: Integration and challenges." In: *Internet of Things* 3 (2018), pp. 134–155.

[42] Marina Bolsunovskaya, Alexander Leksashov, Svetlana Shirokova, and Vladimir Tsygan. "Development of an information system structure for photo-video recording of traffic violations." In: *E3S Web of Conferences*. Vol. 244. EDP Sciences. 2021, p. 07007.

[43] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. "Fog Computing and Its Role in the Internet of Things." In: *Proceedings of the 1st edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, 13–16.

[44] Aurélien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "Powerapi: A software library to monitor the energy consumed at the process-level." In: *ERCIM News* 2013.92 (2013), pp. 43–44.

[45] Ivona Brandic. "Sustainable and Trustworthy Edge Machine Learning." In: *IEEE Internet Computing* 25.5 (2021), pp. 5–9.

[46] Álvaro Brandón, María S Pérez, Jesus Montes, and Alberto Sanchez. "Fmone: A flexible monitoring solution at the edge." In: *Wireless Communications and Mobile Computing* 2018 (2018).

[47] Pierre-Olivier Brissaud, Jérôme François, Isabelle Chrisment, Thibault Cholez, and Olivier Bettan. "Passive Monitoring of HTTPS Service Use." In: *2018 14th International Conference on Network and Service Management (CNSM)*. 2018, pp. 219–225.

[48] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. "Engineering self-adaptive systems through feedback loops." In: *Software engineering for self-adaptive systems* (2009), pp. 48–70.

[49] Fernando Brügge, Mohammad Hasan, Matthieu Kulezak, Knud Lasse Lueth, Eugenio Pasqua, Satyajit Sinha, Philipp Wegner, Kalpesh Baviskar, and Anand Taparia. *State of IoT – Spring 2023*. Tech. rep. IoT Analytics, 2023.

[50] 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.

[51] Jose M Alcaraz Calero and Juan Gutierrez Aguado. "Monpaas: An adaptive monitoring platformas a service for cloud computing infrastructures and services." In: *IEEE Transactions on Services Computing* 8.1 (2014), pp. 65–78.

[52] Gilles Callebaut, Guus Leenders, Jarne Van Mulders, Geoffrey Ottoy, Lieven De Strycker, and Liesbet Van der Perre. "The Art of Designing Remote IoT Devices—Technologies and Strategies for a Long Battery Life." In: *Sensors* 21.3 (2021), p. 913.

[53] Javier Cámara, David Garlan, Bradley Schmerl, and Ashutosh Pandey. "Optimal planning for architecture-based self-adaptation via model checking of stochastic games." In: *Proceedings of the 30th annual ACM symposium on applied computing*. 2015, pp. 428–435.

[54] Jeffrey D Case, Mark Fedor, Martin Lee Schoffstall, and James Davin. *Rfc1157: Simple network management protocol (snmp)*. 1990.

[55] P. Cedillo, J. Jimenez-Gomez, S. Abrahao, and E. Insfran. "Towards a Monitoring Middleware for Cloud Services." In: *Proceedings of the 12th International Conference on Services Computing*. IEEE, 2015, pp. 451–458.

[56] Centers for Disease Control and Prevention. *Pedestrian Safety*. 2022. URL: https://www.cdc.gov/transportationsafety/pedestrian_safety/index.html. (accessed: 08.01.2024).

[57] Hyunseok Chang, Adiseshu Hari, Sarit Mukherjee, and TV Lakshman. "Bringing the cloud to the edge." In: *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2014, pp. 346–351.

[58] Boyuan Chen and Zhen Ming (Jack) Jiang. "A Survey of Software Log Instrumentation." In: *ACM Computing Surveys* 54 (2021).

[59] Zhuo Chen et al. "An Empirical Study of Latency in an Emerging Class of Edge Computing Applications for Wearable Cognitive Assistance." In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. 2017, pp. 1–14.

[60] Jie Cheng, Qiang Ye, Hongbo Jiang, Dan Wang, and Chonggang Wang. "STCDG: An efficient data gathering algorithm based on matrix completion for wireless sensor networks." In: *IEEE Transactions on Wireless Communications* 12.2 (2012), pp. 850–861.

[61] Goh Chia Yee, Chin Jeng Feng, Mohd Azizi Bin Chik, and Mohzani Mokhtar. "Weighted grey relational analysis to evaluate multilevel dispatching rules in wafer fabrication." In: *Grey Systems: Theory and Application* 11.4 (2021), pp. 619–649.

[62] Mung Chiang and Tao Zhang. "Fog and IoT: An Overview of Research Opportunities." In: *IEEE Internet of Things Journal* 3.6 (2016), pp. 854–864.

[63] Katerina Chinnappan, Ivano Malavolta, Grace A Lewis, Michel Albonico, and Patricia Lago. "Architectural Tactics for Energy-Aware Robotics Software: A Preliminary Study." In: *Software Architecture: 15th European Conference, ECSA 2021, Virtual Event, Sweden, September 13-17, 2021, Proceedings*. Springer. 2021, pp. 164–171.

[64] Michele Ciavotta, Davide Motterlini, Marco Savi, and Alessandro Tundo. "DFaaS: Decentralized Function-as-a-Service for Federated Edge Computing." In: *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*. 2021, pp. 1–4.

[65] Vera Colombo, Alessandro Tundo, Michele Ciavotta, and Leonardo Mariani. "Towards Self-Adaptive Peer-to-Peer Monitoring for Fog Environments." In: *2022 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2022, pp. 156–166.

[66] Tabitha S Combs, Laura S Sandt, Michael P Clamann, and Noreen C McDonald. "Automated vehicles and pedestrian safety: exploring the promise and limits of pedestrian detection." In: *American journal of preventive medicine* 56.1 (2019), pp. 1–7.

[67] Directorate-General for Communication. *The European Green Deal*. URL: https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/european-green-deal_en. (accessed: 02.04.2024).

[68] Redis Community and Redis Ltd. *Redis Streams*. 2024. URL: https://redis.io/docs/data-types/streams/. (accessed: 08.01.2024).

[69] Containers Organization. *What is Podman?* 2023. URL: https://docs.podman.io/en/latest/. (accessed: 08.01.2024).

[70] Linux Containers Contributors. *Linux Containers - LXC*. 2024. URL: https://linuxcontainers.org/lxc/introduction/. (accessed: 08.01.2024).

[71] Linux-VServer Contributors. *Linux-VServer*. 2024. URL: http://linux-vserver.org/. (accessed: 08.01.2024).

[72] European Transport Safety Council. *Five ways Europe can tackle road deaths*. 2023. URL: https://etsc.eu/five-ways-europe-can-tackle-road-deaths/. (accessed: 08.01.2024).

[73] Shirlei Aparecida De Chaves, Rafael Brundo Uriarte, and Carlos Becker Westphall. "Toward an architecture for monitoring private clouds." In: *IEEE Communications Magazine* 49.12 (2011), pp. 130–137.

[74] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II." In: *IEEE transactions on evolutionary computation* 6.2 (2002), pp. 182–197.

[75] Patrick Dendorfer, Aljosa Osep, Anton Milan, Konrad Schindler, Daniel Cremers, Ian Reid, Stefan Roth, and Laura Leal-Taixé. "Motchallenge: A benchmark for single-camera multiple target tracking." In: *International Journal of Computer Vision* 129 (2021), pp. 845–881.

[76] Beniamino Di Martino, Giuseppina Cretella, and Antonio 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.

[77] Oxford English Dictionary, ed. *monitor, v., sense 2.c*. Oxford University Press, 2023.

[78] Gregory Dobler, Jordan Vani, and Trang Tran Linh Dam. "Patterns of urban foot traffic dynamics." In: *Computers, Environment and Urban Systems* 89 (2021), p. 101674.

[79] Dalibor Dobrilovic, Francesco Flammini, Andrea Gaglione, and Daniel Tokody. *Open-source hardware performance in Edge Computing*. 2021. URL: https://smartcities.ieee.org/newsletter/july-2021/open-source-hardware-performance-in-edge-computing. (accessed: 11.01.2024).

[80] Juan P Dominguez-Morales, Lourdes Duran-Lopez, Daniel Gutierrez-Galan, Antonio Rios-Navarro, Alejandro Linares-Barranco, and Angel Jimenez-Fernandez. "Wildlife monitoring on the edge: A performance evaluation of embedded neural networks on microcontrollers for animal behavior classification." In: *Sensors* 21.9 (2021), p. 2975.

[81] Schahram Dustdar, Victor Casamayor Pujol, and Praveen Kumar Donta. "On Distributed Computing Continuum Systems." In: *IEEE Transactions on Knowledge and Data Engineering* 35.4 (2023), pp. 4092–4105.

[82] Dynatrace LLC. *Dynatrace*. 2024. URL: https://www.dynatrace.com. (accessed: 08.01.2024).

[83] Kerstin Eder, John P Gallagher, G Fagas, L Gammaitoni, and DJ Paul. "Energy-aware software engineering." In: *ICT-energy concepts for energy efficiency and sustainability* (2017), pp. 103–127.

[84] Hassan Elahi, Khushboo Munir, Marco Eugeni, Sofiane Atek, and Paolo Gaudenzi. "Energy harvesting towards self-powered IoT devices." In: *Energies* 13.21 (2020), p. 5528.

[85] Justin Ellingwood. *An Introduction to Metrics, Monitoring, and Alerting*. URL: https://www.digitalocean.com/community/tutorials/an-introduction-to-metrics-monitoring-and-alerting. (accessed: 20.12.2023).

[86] F5, Inc. *NGINX*. 2024. URL: https://www.nginx.com/. (accessed: 08.01.2024).

[87] Kaniz Fatema, Vincent C. Emeakaroha, Philip D. Healy, John P. Morrison, and Theo Lynn. "A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives." In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2918 –2933.

[88] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014.

[89] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. "Smartwatts: Self-calibrating software-defined power meter for containers." In: *Proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE. 2020, pp. 479–488.

[90] Alcides Fonseca, Rick Kazman, and Patricia Lago. "A manifesto for energy-aware software." In: *IEEE Software* 36.6 (2019), pp. 79–82.

[91] Stefano Forti, Marco Gaglianese, and Antonio Brogi. "Lightweight self-organising distributed monitoring of Fog infrastructures." In: *Future Generation Computer Systems* 114 (2021), pp. 605–618.

[92] The Apache Software Foundation. *Apache Kafka*. 2024. URL: https://kafka.apache.org/. (accessed: 08.01.2024).

[93] The Linux Foundation. *OpenVz*. 2024. URL: https://openvz.org/. (accessed: 08.01.2024).

[94] The Linux Foundation. *cotainerd*. 2024. URL: https://containerd.io/. (accessed: 08.01.2024).

[95] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. "Estimation of energy consumption in machine learning." In: *Journal of Parallel and Distributed Computing* 134 (2019), pp. 75–88.

[96] Peter Garraghan, Renyu Yang, Zhenyu Wen, Alexander Romanovsky, Jie Xu, Rajkumar Buyya, and Rajiv Ranjan. "Emergent failures: Rethinking cloud reliability at scale." In: *IEEE Cloud Computing* 5.5 (2018), pp. 12–21.

[97] Carlo Ghezzi, Andrea Mocci, and Mario Sangiorgio. "Runtime monitoring of component changes with Spy@ Runtime." In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 1403–1406.

[98] Rajrup Ghosh, Siva Prakash Reddy Komma, and Yogesh Simmhan. "Adaptive energy-aware scheduling of dynamic event analytics across edge and cloud resources." In: *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2018, pp. 72–82.

[99] Dimitrios Giouroukis, Alexander Dadiani, Jonas Traub, Steffen Zeuch, and Volker Markl. "A survey of adaptive sampling and filtering algorithms for the internet of things." In: *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 2020, pp. 27–38.

[100] GitHub Contributors. *Redis Exporter*. 2024. URL: https://github.com/oliver006/redis_exporter. (accessed: 08.01.2024).

[101] Github Contributors. *Serverless Sock Shop*. 2024. URL: https://github.com/deib-polimi/serverless-sock-shop. (accessed: 08.01.2024).

[102] Github Contributors. *cAdvisor*. 2024. URL: https://github.com/google/cadvisor. (accessed: 08.01.2024).

[103] e Gminy. *EcoClou*. URL: https://ecoclou.eu/. (accessed: 18.12.2023).

[104] Good Will Instrument Co. Ltd. *GPM-8213-Gwinstek*. URL: https://www.gwinstek.com/en-GB/products/detail/GPM-8213. (accessed: 08.01.2024).

[105] Google LLC. *Models - Object Detection*. URL: https://coral.ai/models/object-detection/. (accessed: 08.01.2024).

[106] Google LLC. *USB Accelerator*. URL: https://coral.ai/products/accelerator. (accessed: 08.01.2024).

[107] GoogleCloudPlatform. *Online Boutique*. 2023. URL: https://github.com/GoogleCloudPlatform/microservices-demo. (accessed: 08.01.2024).

[108] Kalman Graffi and Andreas Disterhöft. "SkyEye: A tree-based peer-to-peer monitoring approach." In: *Pervasive and Mobile Computing* 40 (2017), pp. 593–610.

[109] Marcel Großmann and Clemens Klug. "Monitoring container services at the network edge." In: *2017 29th International Teletraffic Congress (ITC 29)*. Vol. 1. IEEE. 2017, pp. 130–133.

[110] The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. 2024. URL: https://www.postgresql.org/. (accessed: 08.01.2024).

[111] Eoin Martino Grua, Ivano Malavolta, and Patricia Lago. "Self-adaptation in mobile apps: a systematic literature study." In: *Proceedings of the IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE. 2019, pp. 51–62.

[112] Piotr Gularski. *Python State Machine*. URL: https://pysm.readthedocs.io/. (accessed: 08.01.2024).

[113] Harshit Gupta, Shubha Brata Nath, Sandip Chakraborty, and Soumya K Ghosh. "SDFog: A software defined computing architecture for QoS aware service orchestration over edge devices." In: *arXiv preprint arXiv:1609.01190* (2016).

[114] Bin Han, Stan Wong, Christian Mannweiler, Marcos Rates Crippa, and Hans D. Schotten. "Context-Awareness Enhances 5G Multi-Access Edge Computing Reliability." In: *IEEE Access* 7 (2019), pp. 21290–21299.

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

[116] Christopher B Hauser and Stefan Wesner. "Reviewing cloud monitoring: Towards cloud resource profiling." In: *International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 678–685.

[117] Jeffrey Hojlo. *Future of Industry Ecosystems: Shared Data and Insights*. URL: https://blogs.idc.com/2021/01/06/future-of-industry-ecosystems-shared-data-and-insights/. (accessed: 17.11.2023).

[118] Chao-Jung Hsu and Chin-Yu Huang. "Comparison of weighted grey relational analysis for software effort estimation." In: *Software Quality Journal* 19 (2011), pp. 165–200.

[119] IEA. *Data Centres and Data Transmission Networks*. Tech. rep. IEA, 2022.

[120] Shashikant Ilager, Jakob Fahringer, Samuel Carlos de Lima Dias, and Ivona Brandic. "DEMon: Decentralized Monitoring for Highly Volatile Edge Environments." In: *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2022, pp. 145–150.

[121] Docker Inc. *Docker*. 2024. URL: https://www.docker.com/. (accessed: 08.01.2024).

[122] MongoDB Inc. *MongoDB*. 2024. URL: https://www.mongodb.com. (accessed: 08.01.2024).

[123] Perforce Software Inc. *Puppet*. 2024. URL: https://www.puppet.com/. (accessed: 08.01.2024).

[124] Red Hat Inc. *How Ansible works*. 2024. URL: https://www.ansible.com/overview/how-ansible-works. (accessed: 08.01.2024).

[125] Michaela Iorga, Larry Feldman, Robert Barton, Michael Martin, Nedim Goren, and Charif Mahmoudi. *Fog Computing Conceptual Model*. 2018.

[126] James A. Jahnke. *Continuous emission monitoring*. John Wiley & Sons, 2022.

[127] Ankur Jain and Edward Y Chang. "Adaptive sampling for sensor networks." In: *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*. 2004, pp. 10–16.

[128] Fatemeh Jalali, Kerry Hinton, Robert Ayre, Tansu Alpcan, and Rodney S. Tucker. "Fog Computing May Help to Save Energy in Cloud Computing." In: *IEEE Journal on Selected Areas in Communications* 34.5 (2016), pp. 1728–1739.

[129] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. "Performance Monitoring and Root Cause Analysis for Cloud-Hosted Web Applications." In: *Proceedings of the 26th International Conference on World Wide Web*. 2017, 469–478.

[130] Márk Jelasity. "Gossip." In: *Self-organising software*. Springer, 2011, pp. 139–162.

[131] Gangyong Jia, Guangjie Han, Jiaxin Du, and Sammy Chan. "A maximum cache value policy in hybrid memory-based edge computing for mobile devices." In: *IEEE Internet of Things Journal* 6.3 (2018), pp. 4401–4410.

[132] Congfeng Jiang, Tiantian Fan, Honghao Gao, Weisong Shi, Liangkai Liu, Christophe Cérin, and Jian Wan. "Energy aware edge computing: A survey." In: *Computer Communications* 151 (2020), pp. 556–580.

[133] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A.S. Ward. "System Monitoring with Metric-Correlation Models: Problems and Solutions." In: *Proceedings of the 6th International Conference on Autonomic Computing*. ACM. 2009, 13–22.

[134] Steven J Johnston, Philip J Basford, Colin S Perkins, Herry Herry, Fung Po Tso, Dimitrios Pezaros, Robert D Mullins, Eiko Yoneki, Simon J Cox, and Jeremy Singer. "Commodity single board computer clusters and their applications." In: *Future Generation Computer Systems* 89 (2018), pp. 201–212.

[135] Deng Ju-Long. "Control problems of grey systems." In: *Systems & control letters* 1.5 (1982), pp. 288–294.

[136] Sarang Kahvazadeh, Xavi Masip-Bruin, Eva Marín-Tordera, and Alejandro Gómez Cárdenas. "Securing Combined Fog-to-Cloud Systems: Challenges and Directions." In: *Proceedings of the Future Technologies Conference (FTC) 2019*. 2020, pp. 877–892.

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

[138] Gabor Karsai and Janos Sztipanovits. "A model-based approach to self-adaptive software." In: *IEEE Intelligent Systems and Their Applications* 14.3 (1999), pp. 46–53.

[139] Saadallah Kassir, Gustavo de Veciana, Nannan Wang, Xi Wang, and Paparao Palacharla. "Service Placement for Real-Time Applications: Rate-Adaptation and Load-Balancing at the Network Edge." In: *2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*. 2020, pp. 207–215.

[140] Jeffrey O Kephart and David M Chess. "The vision of autonomic computing." In: *Computer* 36.1 (2003), pp. 41–50.

[141] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. "Edge computing: A survey." In: *Future Generation Computer Systems* 97 (2019), pp. 219–235.

[142] Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. "Towards continual reinforcement learning: A review and perspectives." In: *Journal of Artificial Intelligence Research* 75 (2022), pp. 1401–1476.

[143] Foutse Khomh and S Amirhossein Abtahizadeh. "Understanding the impact of cloud patterns on performance and energy consumption." In: *Journal of Systems and Software* 141 (2018), pp. 151–170.

[144] Taehyun Kim, Dong-Wook Sohn, and Sangho Choo. "An analysis of the relationship between pedestrian traffic volumes and built environment around metro stations in Seoul." In: *KSCE Journal of Civil Engineering* 21 (2017), pp. 1443–1452.

[145] Michael Kleis, Eng Keong Lua, and Xiaoming Zhou. "Hierarchical peer-to-peer networks using lightweight superpeer topologies." In: *Proceedings of the 10th IEEE Symposium on Computers and Communications*. IEEE. 2005, pp. 143–148.

[146] Ales Komarek, Jakub Pavlik, Lubos Mercl, and Vladimir Sobeslav. "Metric based cloud infrastructure monitoring." In: *Proceedings of the 12th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2017)*. Springer. 2018, pp. 391–400.

[147] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. "A survey on engineering approaches for self-adaptive systems." In: *Pervasive and Mobile Computing* 17 (2015), pp. 184–206.

[148] Yiyo Kuo, Taho Yang, and Guan-Wei Huang. "The use of grey relational analysis in solving multiple attribute decision-making problems." In: *Computers & industrial engineering* 55.1 (2008), pp. 80–93.

[149] M. Kutare, K. Schwan, G. Eisenhauer, V. Talwar, C. Wang, and M. Wolf. "Monalytics: online monitoring and analytics for managing large scale data centers." In: *Proceeding of the 7th International Conference on Autonomic computing*. ACM, 2010, pp. 141–150.

[150] Ponemon Institute LLC. *Cost of Data Center Outages*. Tech. rep. Ponemon Institute LLC, 2016.

[151] Zabbix LLC. *Zabbix*. 2023. URL: https://www.zabbix.com. (accessed: 08.01.2024).

[152] Steffen Lange, Johanna Pohl, and Tilman Santarius. "Digitalization and energy consumption. Does ICT reduce energy demand?" In: *Ecological Economics* 176 (2020), p. 106760.

[153] Igor Lashkov and Alexey Kashevnik. "Smartphone-based intelligent driver assistant: context model and dangerous state recognition scheme." In: *Intelligent Systems and Applications: Proceedings of the 2019 Intelligent Systems Conference Volume 2*. Springer. 2020, pp. 152–165.

[154] Euijong Lee, Young-Duk Seo, and Young-Gab Kim. "Self-adaptive framework based on mape loop for internet of things." In: *sensors* 19.13 (2019), p. 2996.

[155] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. "Microsoft coco: Common objects in context." In: *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer. 2014, pp. 740–755.

[156] Zhongjie Lin, Hugh HT Liu, and Mike Wotton. "Kalman filter-based large-scale wildfire monitoring with a system of UAVs." In: *IEEE Transactions on Industrial Electronics* 66.1 (2018), pp. 606–615.

[157] Ashish Lingayat, Ranjana R Badre, and Anil Kumar Gupta. "Performance evaluation for deploying docker containers on baremetal and virtual machine." In: *Proceedings of the 3rd International Conference on Communication and Electronics Systems (ICCES)*. IEEE. 2018, pp. 1019–1023.

[158] Jaime Lloret, Lorena Parra, Miran Taha, and Jesús Tomás. "An architecture and protocol for smart continuous eHealth monitoring using 5G." In: *Computer Networks* 129 (2017), pp. 340–351.

[159] Tao Lu, Wen Xia, Xiangyu Zou, and Qianbin Xia. "Adaptively Compressing IoT Data on the Resource-constrained Edge." In: *3rd {USENIX} Workshop on Hot Topics in Edge Computing*. 2020.

[160] Ivan Lujic, Vincenzo De Maio, Klaus Pollhammer, Ivan Bodrozic, Josip Lasic, and Ivona Brandic. "Increasing traffic safety with real-time edge analytics and 5G." In: *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. ACM. 2021, pp. 19–24.

[161] Amin Mahmoudi, Saad Ahmed Javed, Sifeng Liu, and Xiaopeng Deng. "Distinguishing coefficient driven sensitivity analysis of GRA model for intelligent decisions: application in project management." In: *Technological and Economic Development of Economy* 26.3 (2020), pp. 621–641.

[162] Henry B Mann and Donald R Whitney. "On a test of whether one of two random variables is stochastically larger than the other." In: *The annals of mathematical statistics* (1947), pp. 50–60.

[163] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. "Cloud computing — The business perspective." In: *Decision Support Systems* 51.1 (2011), pp. 176–189.

[164] M. L. Massie, B. N. Chun, and D. E. Culler. "The Ganglia distributed monitoring system: design, implementation, and experience." In: *Parallel Computing* 30.7 (2004), pp. 817–840.

[165] Narges Mehran, Dragi Kimovski, and Radu Prodan. "A Two-Sided Matching Model for Data Stream Processing in the Cloud – Fog Continuum." In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021, pp. 514–524.

[166] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. 2011.

[167] Shicong Meng and Ling Liu. "Enhanced monitoring-as-a-service for effective cloud management." In: *IEEE Transactions on Computers* 62.9 (2012), pp. 1705–1720.

[168] Jhonny Mertz and Ingrid Nunes. "Software runtime monitoring with adaptive sampling rate to collect representative samples of execution traces." In: *Journal of Systems and Software* (2023), p. 111708.

[169] Microsoft. *Cloud Design Patterns*. 2023. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/. (accessed: 26.05.2023).

[170] Microsoft. *Azure Compute*. 2024. URL: https://azure.microsoft.com/en-gb/products/category/compute/. (accessed: 08.01.2024).

[171] Microsoft. *Azure Kubernetes Service (AKS)*. 2024. URL: https://azure.microsoft.com/en-gb/services/kubernetes-service/. (accessed: 08.01.2024).

[172] Dejan S Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. *Peer-to-peer computing*. 2002.

[173] Angelos Mimidis-Kentis, Jose Soler, Paul Veitch, Adam Broadbent, Marco Mobilio, Oliviero Riganelli, Steven Van Rossem, Wouter Tavernier, and Bessem Sayadi. "The Next Generation Platform as A Service: Composition and Deployment of Platforms and Services." In: *Future Internet* 11.119 (2019).

[174] Mayank Mishra, Paulo B Lourenço, and Gunturi Venkata Ramana. "Structural health monitoring of civil engineering structures by using the internet of things: A review." In: *Journal of Building Engineering* 48 (2022), p. 103954.

[175] Tanya Mohn. *Pedestrian Deaths On The Rise Again, A Walker Dies Every 75 Minutes On America's Roads*. 2023. URL: https://www.forbes.com/sites/tanyamohn/2023/02/28/pedestrian-deaths-on-the-rise--again-a-walker-dies-every-75-minutes-on-americas-roads/. (accessed: 08.01.2024).

[176] Sergio Moreschini, Fabiano Pecorelli, Xiaozhou Li, Sonia Naz, David Hästbacka, and Davide Taibi. "Cloud Continuum: The Definition." In: *IEEE Access* 10 (2022), pp. 131876–131886.

[177] Nagios Enterprises. *Nagios*. 2024. URL: https://www.nagios.com. (accessed: 08.01.2024).

[178] Attila M Nagy and Vilmos Simon. "Survey on traffic prediction in smart cities." In: *Pervasive and Mobile Computing* 50 (2018), pp. 148–163.

[179] Yucen Nan, Wei Li, Wei Bao, Flavia C Delicato, Paulo F Pires, Yong Dou, and Albert Y Zomaya. "Adaptive energy-aware computation offloading for cloud of things systems." In: *IEEE Access* 5 (2017), pp. 23947–23957.

[180] Zeinab Nezami, Kamran Zamanifar, Karim Djemame, and Evangelos Pournaras. "Decentralized Edge-to-Cloud Load Balancing: Service Placement for the Internet of Things." In: *IEEE Access* 9 (2021), pp. 64983–65000.

[181] Patrick Ngatchou, Anahita Zarei, and A El-Sharkawi. "Pareto multi objective optimization." In: *Proceedings of the international conference on, intelligent systems application to power systems*. IEEE. 2005, pp. 84–91.

[182] Rihards Olups. *Zabbix Network Monitoring*. Packt Publishing Ltd, 2016.

[183] OpenFaaS Ltd. *OpenFaaS*. 2024. URL: https://www.openfaas.com. (accessed: 08.01.2024).

[184] Optuna Contributors. *optuna.samplers.NSGAIISampler*. URL: https://optuna.readthedocs.io/en/stable/reference/samplers/generated/optuna.samplers.NSGAIISampler.html. (accessed: 08.01.2024).

[185] Optuna Contributors. *optuna.samplers.RandomSampler*. URL: https://optuna.readthedocs.io/en/stable/reference/samplers/generated/optuna.samplers.RandomSampler.html. (accessed: 08.01.2024).

[186] Andy Oram. *Peer-to-Peer: Harnessing the power of disruptive technologies*. " O'Reilly Media, Inc.", 2001.

[187] UK Parliament POST. *Energy Consumption of ICT*. Tech. rep. UK Parliament, 2022.

[188] Rafael Padilla, Sergio L Netto, and Eduardo AB Da Silva. "A survey on performance metrics for object-detection algorithms." In: *2020 international conference on systems, signals and image processing*. IEEE. 2020, pp. 237–242.

[189] Jacopo Panerati, Donatella Sciuto, and Giovanni Beltrame. "Optimization strategies in design space exploration." In: *Handbook of Hardware/Software Codesign*. Springer, 2017, pp. 189–216.

[190] Vincenzo Pecunia, Luigi G Occhipinti, and Robert LZ Hoye. "Emerging indoor photovoltaic technologies for sustainable internet of things." In: *Advanced Energy Materials* 11.29 (2021), p. 2100698.

[191] Javier Povedano-Molina, Jose M Lopez-Vega, Juan M Lopez-Soler, Antonio Corradi, and Luca Foschini. "DARGOS: A highly adaptable and scalable monitoring architecture for multi-tenant Clouds." In: *Future Generation Computer Systems* 29.8 (2013), pp. 2041–2056.

[192] OpenStack Project. *Monasca*. 2024. URL: https://www.openstack.org/software/releases/antelope/components/monasca. (accessed: 08.01.2024).

[193] Prometheus Authors. *Blackbox Exporter*. 2024. URL: https://github.com/prometheus/blackbox_exporter. (accessed: 08.01.2024).

[194] Prometheus Authors. *Node Exporter*. 2024. URL: https://github.com/prometheus/node_exporter. (accessed: 08.01.2024).

[195] Prometheus Authors. *SNMP exporter for Prometheus*. 2024. URL: https://github.com/prometheus/snmp_exporter. (accessed: 08.01.2024).

[196] Codrin Pruteanu and Cristian-gyözö Haba. "GenFSM: A finite state machine generation tool." In: *Proc. 9th Int. Conf. Dev. Applicat. Syst*. 2008, pp. 165–168.

[197] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. "Survey and benchmarking of machine learning accelerators." In: *Proceedings of the 8th IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2019, pp. 1–9.

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

[199] Gary Riley. "Clips: An expert system building tool." In: *Proceedings Second National Technology Transfer Conference and Exposition*. Vol. 2. 1991, pp. 149–158.

[200] L. Romano, D. De Mari, Z. Jerzak, and C. Fetzer. "A Novel Approach to QoS Monitoring in the Cloud." In: *Proceedings of the 1st International Conference on Data Compression, Communications and Processing*. 2011, pp. 45–51.

[201] Karen Rose, Scott Eldridge, and Lyman Chapin. "The internet of things: An overview." In: *The internet society (ISOC)* 80 (2015), pp. 1–50.

[202] Timothy Rupprecht and Yanzhi Wang. "A survey for deep reinforcement learning in markovian cyber-physical systems: Common problems and solutions." In: *Neural Networks* (2022).

[203] Dario Sabella, Alessandro Vaillant, Pekka Kuure, Uwe Rauschenbach, and Fabio Giust. "Mobile-edge computing architecture: The role of MEC in the Internet of Things." In: *IEEE Consumer Electronics Magazine* 5.4 (2016), pp. 84–91.

[204] Hareem Sahar, Abdul A Bangash, and Mirza O Beg. "Towards energy aware object-oriented development of android applications." In: *Sustainable Computing: Informatics and Systems* 21 (2019), pp. 28–46.

[205] Mazeiar Salehie and Ladan Tahvildari. "Towards a goal-driven approach to action selection in self-adaptive software." In: *Software: Practice and Experience* 42.2 (2012), pp. 211–233.

[206] Arun Kumar Sangaiah, Ali Shokouhi Rostami, Ali Asghar Rahmani Hosseinabadi, Morteza Babazadeh Shareh, Amir Javadpour, Shirin Hatami Bargh, and Mohammad Mehedi Hassan. "Energy-Aware Geographic Routing for Real-Time Workforce Monitoring in Industrial Informatics." In: *IEEE Internet of Things Journal* 8.12 (2021), pp. 9753–9762.

[207] Mahadev Satyanarayanan. "The Emergence of Edge Computing." In: *Computer* 50.1 (2017), pp. 30–39.

[208] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. "The case for vm-based cloudlets in mobile computing." In: *IEEE pervasive Computing* 8.4 (2009), pp. 14–23.

[209] Douglas C Schmidt, Mohamed Fayad, and Ralph E Johnson. "Software patterns." In: *Communications of the ACM* 39.10 (1996), pp. 37–39.

[210] Daniel Schwartz, Jonathan Michael Gomes Selman, Peter Wrege, and Andreas Paepcke. "Deployment of Embedded Edge-AI for Wildlife Monitoring in Remote Regions." In: *Proceedings of the 20th IEEE International Conference on Machine Learning and Applications*. IEEE. 2021, pp. 1035–1042.

[211] Seedstudio.com. *All about cpus: Microprocessor, microcontroller and Single Board Computer*. 2020. URL: https://www.seeedstudio.com/blog/2020/10/27/all-about-cpus-microprocessor-microcontroller-and-single-board-computer/. (accessed: 11.01.2024).

[212] Commission Spokesperson's Service. *Road safety in the EU*. 2023. URL: https://ec.europa.eu/commission/presscorner/detail/en/ip_23_953. (accessed: 08.01.2024).

[213] Samuel Sanford Shapiro and Martin B Wilk. "An analysis of variance test for normality (complete samples)." In: *Biometrika* 52.3/4 (1965), pp. 591–611.

[214] Yogesh Sharma, Bahman Javadi, Weisheng Si, and Daniel Sun. "Reliability and energy efficiency in cloud computing systems: Survey and taxonomy." In: *Journal of Network and Computer Applications* 74 (2016), pp. 66–85.

[215] Anas Shatnawi, Matteo Orrù, Marco Mobilio, Oliviero Riganelli, and Leonardo Mariani. "Cloudhealth: a model-driven approach to watch the health of cloud services." In: *Proceedings of the IEEE/ACM 1st International Workshop on Software Health (SoHeal)*. IEEE. 2018, pp. 40–47.

[216] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. "Edge Computing: Vision and Challenges." In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646.

[217] R Shreyas, BV Pradeep Kumar, HB Adithya, B Padmaja, and MP Sunil. "Dynamic traffic rule violation monitoring system using automatic number plate recognition with SMS feedback." In: *2017 2nd International Conference on Telecommunication and Networks*. IEEE. 2017, pp. 1–5.

[218] Carlos Eduardo da Silva and Rogério de Lemos. "A framework for automatic generation of processes for self-adaptive software systems." In: *Informatica* 35.1 (2011).

[219] Joaquim Silva, Eduardo RB Marques, Luís Lopes, and Fernando Silva. "Energy-aware adaptive offloading of soft real-time jobs in mobile edge clouds." In: *Journal of Cloud Computing* 10.1 (2021), pp. 1–21.

[220] Michael Smit, Bradley Simmons, and Marin Litoiu. "Distributed, Application-Level Monitoring for Heterogeneous Clouds Using Stream Processing." In: *Future Generation Computer Systems* 29.8 (2013), pp. 2103–2114.

[221] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. "A Survey on the Adoption of Patterns for Engineering Software for the Cloud." In: *IEEE Transactions on Software Engineering* 48.6 (2022), pp. 2128–2140.

[222] Arthur Souza, Nélio Cacho, Ayman Noor, Prem Prakash Jayaraman, Alexander Romanovsky, and Rajiv 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)*. IEEE. 2018, pp. 758–765.

[223] Josef Spillner, Juliana Freitag Borin, and Luiz Fernando Bittencourt. "Intent-based placement of microservices in computing continuums." In: *Future Intent-Based Networking: On the QoS Robust and Energy Efficient Heterogeneous Software Defined Networks*. Springer, 2021, pp. 38–50.

[224] Wolfram Stadler. *Multicriteria Optimization in Engineering and in the Sciences*. Vol. 37. Springer Science & Business Media, 1988.

[225] Ralf Steinmetz and Klaus Wehrle. *Peer-to-peer systems and applications*. Vol. 3485. Springer, 2005.

[226] Ali Sunyaev. "Cloud Computing." In: *Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies*. Springer International Publishing, 2020, pp. 195–236. ISBN: 978-3-030-34957-8.

[227] Hassan Jamil Syed, Abdullah Gani, Fariza Hanum Nasaruddin, Anjum Naveed, Abdelmuttlib Ibrahim Abdalla Ahmed, and Muhammad Khurram Khan. "Cloudprocmon: A non-intrusive cloud monitoring framework." In: *IEEE Access* 6 (2018), pp. 44591–44606.

[228] Salman Taherizadeh, Andrew C Jones, Ian Taylor, Zhiming Zhao, and Vlado Stankovski. "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review." In: *Journal of Systems and Software* 136 (2018), pp. 19–38.

[229] Damian A Tamburri, Marco Miglierina, and Elisabetta Di Nitto. "Cloud applications monitoring: An industrial study." In: *Information and Software Technology* 127 (2020), p. 106376.

[230] Gioacchino Tangari, Daphne Tuncer, Marinos Charalambides, Yuanshunle Qi, and George Pavlou. "Self-adaptive decentralized monitoring in software-defined networks." In: *IEEE Transactions on Network and Service Management* 15.4 (2018), pp. 1277–1291.

[231] The Kubernetes Authors. *Kubernetes DaemonSet*. 2024. URL: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/. (accessed: 08.01.2024).

[232] The Kubernetes Authors. *Kubernetes Deployment*. 2024. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. (accessed: 08.01.2024).

[233] The Kubernetes Authors. *Kubernetes Pod*. 2024. URL: https://kubernetes.io/docs/concepts/workloads/pods/. (accessed: 08.01.2024).

[234] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. "FeatureIDE: An extensible framework for feature-oriented software development." In: *Science of Computer Programming* 79 (2014), pp. 70–85.

[235] Daniel Tovarňák and Tomáš Pitner. "Towards multi-tenant and interoperable monitoring of virtual machines in cloud." In: *Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2012, pp. 436–442.

[236] David Tracey and Cormac Sreenan. "How to see through the Fog? Using Peer to Peer (P2P) for the Internet of Things." In: *Proceedings of the IEEE 5th World Forum on Internet of Things*. IEEE. 2019, pp. 47–52.

[237] Anne E Trefethen and Jeyarajan Thiyagalingam. "Energy-aware software: Challenges, opportunities and strategies." In: *Journal of Computational Science* 4.6 (2013), pp. 444–449.

[238] Demetris Trihinas, George Pallis, and Marios D Dikaiakos. "Jcatascopia: Monitoring elastically adaptive applications in the cloud." In: *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2014, pp. 226–235.

[239] Demetris Trihinas, George Pallis, and Marios D Dikaiakos. "ADMin: Adaptive monitoring dissemination for the Internet of Things." In: *IEEE INFOCOM 2017-IEEE conference on computer communications*. IEEE. 2017, pp. 1–9.

[240] Alessandro Tundo, Marco Mobilio, Shashikant Ilager, Ivona Brandić, Ezio Bartocci, and Leonardo Mariani. "An Energy-Aware Approach to Design Self-Adaptive AI-based Applications on the Edge." In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2023.

[241] Alessandro Tundo, Marco Mobilio, Matteo Orrù, Oliviero Riganelli, Michell Guzmàn, and Leonardo Mariani. "VARYS: An Agnostic Model-Driven Monitoring-as-a-Service Framework for the Cloud." In: *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, 1085–1089.

[242] Alessandro Tundo, Marco Mobilio, Oliviero Riganelli, and Leonardo Mariani. "Automated Probe Life-Cycle Management for Monitoring-As-a-Service." In: *IEEE Transactions on Services Computing* 16.2 (2023), pp. 969–982.

[243] Alessandro Tundo, Marco Mobilio, Oliviero Riganelli, and Leonardo Mariani. *Monitoring Probe Deployment Patterns for Cloud-Native Applications: Definition and Empirical Assessment (Experimental Material)*. 2024. URL: https://gitlab.com/learnERC/monitoring-patterns-experiments. (accessed: 08.01.2024).

[244] Alessandro Tundo, Marco Mobilio, Oliviero Riganelli, and Leonardo Mariani. "Monitoring Probe Deployment Patterns for Cloud-Native Applications: Definition and Empirical Assessment." In: *IEEE Transactions on Services Computing (Early Access)* (2024), pp. 1–19.

[245] James Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018.

[246] Ruben Vales, Jose Moura, and Rui Marinheiro. "Energy-aware and adaptive fog storage mechanism with data replication ruled by spatio-temporal content popularity." In: *Journal of Network and Computer Applications* 135 (2019), pp. 84–96.

[247] Bertrand Vandeportaele. "A Finite State Machine modeling language and the associated tools allowing fast prototyping for FPGA devices." In: *2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics*. IEEE. 2017, pp. 1–6.

[248] Yiannis Verginadis. "A Review of Monitoring Probes for Cloud Computing Continuum." In: *Proceedings of the International Conference on Advanced Information Networking and Applications*. Springer. 2023, pp. 631–643.

[249] Shanu Verma, Millie Pant, and Vaclav Snasel. "A comprehensive review on NSGA-II for multi-objective combinatorial optimization problems." In: *Ieee Access* 9 (2021), pp. 57757–57791.

[250] Gheorghe-Daniel Voinea, Cristian Cezar Postelnicu, Mihai Duguleana, Gheorghe-Leonte Mogan, and Radu Socianu. "Driving performance and technology acceptance evaluation in real traffic of a smartphone-based driver assistance system." In: *International journal of environmental research and public health* 17.19 (2020), p. 7098.

[251] Sophie Vos, Patricia Lago, Roberto Verdecchia, and Ilja Heitlager. "Architectural Tactics to Optimize Software for Energy Efficiency in the Public Cloud." In: *2022 International Conference on ICT for Sustainability*. IEEE. 2022, pp. 77–87.

[252] Voxel51 Inc. *Evaluating Models*. URL: https://docs.voxel51.com/user_guide/evaluation.html. (accessed: 08.01.2024).

[253] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf. "A Flexible Architecture Integrating Monitoring and Analytics for Managing Large-scale Data Centers." In: *Proceedings of the 8th International Conference on Autonomic Computing*. ACM, 2011, pp. 141–150.

[254] Tao Wang, Jiwei Xu, Wenbo Zhang, Zeyu Gu, and Hua Zhong. "Self-adaptive cloud monitoring with online anomaly detection." In: *Future Generation Computer Systems* 80 (2018), pp. 89–101.

[255] Zhiyuan Wang and Gade Pandu Rangaiah. "Application and analysis of methods for selecting an optimal solution from the Pareto-optimal front obtained by multiobjective optimization." In: *Industrial & Engineering Chemistry Research* 56.2 (2017), pp. 560–574.

[256] Hans W Wendt. "Dealing with a common problem in social science: A simplified rank-biserial coefficient of correlation based on the statistic." In: *European Journal of Social Psychology* (1972).

[257] Danny Weyns. *An introduction to self-adaptive systems: A contemporary software engineering perspective*. John Wiley & Sons, 2020.

[258] OpenFog Architecture Workgroup. *OpenFog Reference Architecture for Fog Computing*. Tech. rep. OpenFog Consortium, 2017.

[259] Hsin-Hung Wu. "A comparative study of using grey relational analysis in multiple attribute decision making problems." In: *Quality Engineering* 15.2 (2002), pp. 209–217.

[260] Fatos Xhafa and Paul Krause. "IoT-Based Computational Modeling for Next Generation Agro-Ecosystems: Research Issues, Emerging Trends and Challenges." In: *IoT-based Intelligent Modelling for Environmental and Ecological Engineering: IoT Next Generation EcoAgro Systems*. 2021, pp. 1–21.

[261] Sherif M Yacoub and Hany Hussein Ammar. *Pattern-oriented analysis and design: composing patterns to design software systems*. Addison-Wesley Professional, 2004.

[262] B Beverly Yang and Hector Garcia-Molina. "Designing a super-peer network." In: *Proceedings 19th international conference on data engineering*. IEEE. 2003, pp. 49–60.

[263] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. "All one needs to know about fog computing and related edge computing paradigms: A complete survey." In: *Journal of Systems Architecture* 98 (2019), pp. 289–330.

[264] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. "Characterizing logging practices in open-source software." In: *2012 34th International Conference on Software Engineering*. IEEE. 2012, pp. 102–112.

[265] Zhaoning Zhang, Dongsheng Li, and Kui Wu. "Large-scale virtual machines provisioning in clouds: challenges and approaches." In: *Frontiers of Computer Science* 10.1 (2016), pp. 2–18.