SCUOLA
NORMALE
SUPERIORE

Classe di Scienze

Corso di perfezionamento in
Data Science

XXXV ciclo

Data-Driven Methods for Data Center Operations Support

Settore Scientifico Disciplinare **ING-INF/05**

Candidato
Dr. Giacomo Lanciano

Relatori

Prof. Tommaso Cucinotta

Prof. Davide Bacciu

Dr. Andrea Passarella

Anno accademico 2022/2023

Giacomo Lanciano: *Data-driven Methods for Data Center Operations Support*, © March 2023

SUPERVISORS:
Prof. Tommaso Cucinotta, Scuola Superiore Sant'Anna, Italy
Prof. Davide Bacciu, University of Pisa, Italy
Dr. Andrea Passarella, National Research Council, Italy

LOCATION:
Pisa, Italy

ABSTRACT

During the last decade, cloud technologies have been evolving at an impressive pace, such that we are now living in a *cloud-native* era where developers can leverage on an unprecedented landscape of (possibly *managed*) services for orchestration, compute, storage, load-balancing, monitoring, etc. The possibility to have on-demand access to a diverse set of configurable virtualized resources allows for building more elastic, flexible and highly-resilient distributed applications. Behind the scenes, cloud providers sustain the heavy burden of maintaining the underlying infrastructures, consisting in large-scale distributed systems, partitioned and replicated among many geographically dislocated data centers to guarantee scalability, robustness to failures, high availability and low latency. The larger the scale, the more cloud providers have to deal with complex interactions among the various components, such that monitoring, diagnosing and troubleshooting issues become incredibly daunting tasks.

To keep up with these challenges, development and operations practices have undergone significant transformations, especially in terms of improving the automations that make releasing new software, and responding to unforeseen issues, faster and sustainable at scale. The resulting paradigm is nowadays referred to as *DevOps*. However, while such automations can be very sophisticated, traditional DevOps practices fundamentally rely on *reactive* mechanisms, that typically require careful manual tuning and supervision from human experts. To minimize the risk of outages—and the related costs—it is crucial to provide DevOps teams with suitable tools that can enable a *proactive* approach to data center operations.

This work presents a comprehensive *data-driven* framework to address the most relevant problems that can be experienced in large-scale distributed cloud infrastructures. These environments are indeed characterized by a very large availability of diverse data, collected at each level of the stack, such as: time-series (e.g., physical host measurements, virtual machine or container metrics, networking components logs, application KPIs); graphs (e.g., network topologies, fault graphs reporting dependencies among hardware and software components, performance issues propagation networks); and text (e.g., source code, system logs, version control system history, code review feedbacks). Such data are also typically updated with relatively high frequency, and subject to distribution drifts caused by continuous configuration changes to the underlying infrastructure. In such a highly dynamic scenario, traditional model-driven approaches alone may be inadequate at capturing the complexity of the interactions among system com-

ponents. DevOps teams would certainly benefit from having robust data-driven methods to support their decisions based on historical information. For instance, effective anomaly detection capabilities may also help in conducting more precise and efficient root-cause analysis. Also, leveraging on accurate forecasting and intelligent control strategies would improve resource management.

Given their ability to deal with high-dimensional, complex data, Deep Learning-based methods are the most straightforward option for the realization of the aforementioned support tools. On the other hand, because of their complexity, this kind of models often requires huge processing power, and suitable hardware, to be operated effectively at scale. These aspects must be carefully addressed when applying such methods in the context of data center operations. Automated operations approaches must be dependable and cost-efficient, not to degrade the services they are built to improve.

PUBLICATIONS AND OTHER CONTRIBUTIONS

Some ideas and figures included in this thesis have appeared previously in the following publications:

[1] Giacomo Lanciano, Manuel Stein, Volker Hilt, and Tommaso Cucinotta. “Analyzing Declarative Deployment Code with Large Language Models.” In: *Proceedings of the 13th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2023.

[2] Tommaso Cucinotta, Giacomo Lanciano, Antonio Ritacco, Fabio Brau, Filippo Galli, Vincenzo Iannino, Marco Vannucci, Antonino Artale, Joao Barata, and Enrica Sposato. “Forecasting Operation Metrics for Virtualized Network Functions.” In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021, pp. 596–605.

[3] Giacomo Lanciano, Filippo Galli, Tommaso Cucinotta, Davide Bacciu, and Andrea Passarella. “Predictive auto-scaling with OpenStack Monasca.” In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*. Association for Computing Machinery, 2021, pp. 1–10. ISBN: 978-1-4503-8564-0.

[4] Giacomo Lanciano, Antonio Ritacco, Fabio Brau, Tommaso Cucinotta, Marco Vannucci, Antonino Artale, Joao Barata, and Enrica Sposato. “Using Self-Organizing Maps for the Behavioral Analysis of Virtualized Network Functions.” In: *Cloud Computing and Services Science*. Ed. by Donald Ferguson, Claus Pahl, and Markus Helfert. Springer International Publishing, 2021, pp. 153–177. ISBN: 978-3-030-72369-9.

[5] Tommaso Cucinotta, Giacomo Lanciano, Antonio Ritacco, Marco Vannucci, Antonino Artale, Joao Barata, Enrica Sposato, and Luca Basili. “Behavioral Analysis for Virtualized Network Functions: A SOM-based Approach.” In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2020, pp. 150–160. ISBN: 978-989-758-424-4.

[6] Giacomo Lanciano, Antonio Ritacco, Tommaso Cucinotta, Marco Vannucci, Antonino Artale, Luca Basili, Enrica Sposato, and Joao Barata. “SOM-based behavioral analysis for virtualized network functions.” In: *Proceedings of the 35th Annual ACM*

Symposium on Applied Computing. ACM, 2020, pp. 1204–1206. ISBN: 978-1-4503-6866-7.

[7] Riccardo Mancini, Antonio Ritacco, Giacomo Lanciano, and Tommaso Cucinotta. "XPySom: High-Performance Self-Organizing Maps." In: *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 209–216. ISBN: 978-1-72819-924-5.

Furthermore, some ideas are also related to the following patent applications:

[1] Enrica Sposato, Antonino Artale, Tommaso Cucinotta, Marco Vannucci, Giacomo Lanciano, Luisa Neves Pires Jorge, Filippo Galli, and Fabio Brau. "Method of managing resources of an infrastructure for network function virtualization." EP4016963A1 (Filed), IT102020000031034A (Granted). 2022.

[2] Tommaso Cucinotta, Marco Vannucci, Antonio Ritacco, Giacomo Lanciano, Antonino Artale, Joao Barata, and Enrica Sposato. "A method of identifying and classifying the behavior modes of a plurality of data relative to a telephony infrastructure for network function virtualization." EP3772833A1 (Filed), IT102019000014241A (Filed). 2021.

[3] Tommaso Cucinotta, Marco Vannucci, Antonio Ritacco, Giacomo Lanciano, Antonino Artale, Joao Barata, and Enrica Sposato. "A method of predicting the time course of a plurality of data relative to a telephony infrastructure for network function virtualization." EP3772834A1 (Filed), IT102019000014262A (Filed). 2021.

The work done for this thesis also led to the development of the following software artifacts:

- An open-source extension to the Monasca [170] monitoring framework, called *monasca-predictor* [122], that enables OpenStack operations based on predictive analytics, with a particular focus on auto-scaling.

- Several data-driven tools to support Vodafone NVI teams in their daily activities, such as monitoring the health status of the NFV infrastructure and forecasting resource utilization for capacity planning. Such tools are currently used in production.

- A tool that leverages on LLMs to support Nokia Bell Labs research teams in ensuring the quality of declarative Kubernetes deployment code, by providing QA-related recommendations based on established best practices and design patterns.

- A few small contributions to open-source projects like OpenStack *Kolla-Ansible* [169] and *Somoclu* [253].

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AD	Anomaly Detection
ANN	Artificial Neural Network
API	Application Programming Interface
AR	Auto-Regressive
ARIMA	Auto-Regressive Integrated Moving Average
ARMA	Auto-Regressive Moving Average
AST	Abstract Syntax Tree
AUC	Area Under Curve
AVX	Advanced Vector eXtensions (by Intel)
AWS	Amazon Web Services
BLAS	Basic Linear Algebra Subprograms
BMU	Best-Matching Unit
BN	Bayesian Network
CAPEX	Capital Expenditure
CART	Classification And Regression Trees
CBMG	Customer Behavior Modeling Graph
CI	Continuous Integration
CD	Continuous Deployment
CDN	Content-Delivery Network
CLI	Command-Line Interface
CNCF	Cloud Native Computing Foundation
CPS	Cyber-Physical System
CPU	Central Processing Unit
DB	Database
DT	Decision Tree
DHCP	Dynamic Host Configuration Protocol
DL	Deep Learning
DNN	Deep Neural Network
DNS	Domain Name System
DQN	Deep Q-Network
EMNIST	Extended Modified National Institute of Standards and Technology (dataset)

FFNN	Feed-Forward Neural Network
FFT	Fast Fourier Transform
FL	Fuzzy Logic
FPGA	Field Programmable Gate Array
FPR	False-Positive Rate
GBT	Gradient-Boosted Tree
GCP	Google Cloud Platform
GNN	Graph Neural Network
GP	Gaussian Process
GP-GPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HMM	Midden Markov Model
HOT	Heat Orchestration Template
HTM	Hierarchical Temporal Memory
HW	Holt-Winters
HPC	High-Performance Computing
IaaS	Infrastructure-as-a-Service
IaC	Infrastructure-as-Code
ICT	Information and Communications Technology
IMS	IP Multimedia Subsystem
IP	Internet Protocol
IR	Information Retrieval
IT	Information Technology
KPI	Key Performance Indicator
LB	Load-Balancer/Load-Balancing
LM	Language Model
LLM	Large Language Model
LR	Linear Regression
LSTM	Long Short-Term Memory
LTE	Long Term Evolution
MA	Moving Average
MAPE	Mean Average Percentage Error
MSE	Mean Squared Error
MKL	Math Kernel Library (by Intel)
ML	Machine Learning

MLP	Multi-Layer Perceptron
MPI	Message Passing Interface
MSA	Micro-Services Architecture
NBL	Nokia Bell Labs
NAR	Non-linear Auto-Regressive
NLP	Natural Language Processing
NFV	Network Function Virtualization
NSP	Network Service Provider
NSRF	Negative Sampling Random Forest
NVI	Network Virtual Infrastructure
OPEX	Operating Expenditure
OS	Operating System
PaaS	Platform-as-a-Service
PCA	Principal Components Analysis
PR	Polynomial Regression
QA	Quality Assurance
QE	Quantization Error
QoS	Quality of Service
RCA	Root-Cause Analysis
RDF	Random Decision Forest
RL	Reinforcement Learning
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
RPC	Remote Procedure Call
SARIMA	Seasonal Auto-Regressive Integrated Moving Average
SDL	self-Directed Learning
SDN	Software-Defined Network/Networking
SGD	Stochastic Gradient Descent
SSE	Streaming SIMD Extensions (by Intel)
SIMD	Single Instruction, Multiple Data
SLA	Service Level Agreement
SLO	Service Level Objective
SOM	Self-Organizing Map
SVM	Support Vector Machine
TPR	True-Positive Rate

ToR	Top-of-Rack
VM	Virtual Machine
VNF	Virtualized Network Function
VPN	Virtual Private Network
vROps	vRealize Operations Manager (by VMWare)
WAN	Wide Area Network
XaaS	Anything-as-a-Service

Part I

WHY DATA-DRIVEN OPERATIONS?

1 INTRODUCTION

1.1 MOTIVATIONS

Cloud computing is nowadays an established paradigm for effectively developing and managing enterprise applications and assets. The most evident benefit of using such technology is the possibility to have on-demand access to a diverse set of virtualized resources (e.g., computing, storage, networking, etc.) that can be configured to fit the specific needs, allowing for more elastic, flexible and resilient applications. Indeed, companies like Google, Amazon and Microsoft have been able to build successful business models leveraging this technology, such that many organizations heavily depend on their—*public*—cloud services (e.g., it is well known that Netflix's video streaming platform is largely built upon AWS technology). On the other hand, other organizations do not want to, or cannot, rely on external cloud providers, e.g., due to regulations, strict performance requirements, security concerns and so on. In such cases, they can still benefit of the advantages of such paradigm by investing in a *private* cloud, upon which they can build their IT systems. For instance, in recent years, Network Service Providers (NSP) have started to adopt this strategy to enable NFV [39, 194]. In similar cases, the choice of private cloud infrastructures is also corroborated by latency-related concerns. Indeed, since their service-chains are highly delay-sensitive (e.g. LTE, 4G, 5G, etc.), it is unpractical to rely on public cloud infrastructures, that are typically shared among multiple tenants and not necessarily deployed according to the needs of NSPs.

Either it is public or private cloud, on-premise or off-premise, eventually someone has to sustain the heavy burden of managing and maintaining the underlying infrastructure that makes cloud computing possible. In order to guarantee scalability, robustness to failure, high availability, and low latency, such systems are typically designed as large-scale distributed systems [175], often partitioned and/or replicated among many geographically dislocated data centers. The larger the scale, the more operations teams have to deal with complex interactions among the various components, such that monitoring, diagnosis and troubleshooting of possible issues become incredibly difficult tasks [82]. If such issues are not promptly handled, then the customers may be impacted, and start experiencing degradation of the offered service availability and performance.

Sometimes, we can see some major incidents hitting the news. Given the large success that cloud computing has been having during the recent years, a system outage occurring in a data center of a major provider can propagate rather quickly and cripple a huge number of services, websites or applications that depend on it. Events like this can potentially start a series of undesirable chain-reactions, that typically result in diverse issues experienced by the customers [9]. As an example, some applications could be managing vital aspects of the daily business of an organization, like granting access to email inboxes, enabling communication and interactions among co-workers, or serving important documents and spreadsheets that are required to advance critical processes and whose unavailability can severely slow down all activities company-wide. Or, for instance, imagine coming back home after work and not being able to stream your favorite TV series and films, at the point that you quit your subscription with your entertainment services provider and activate a new one with its competitor [218]. Or again, given that IoT devices are becoming more and more part of everyone's life, imagine not being able to enter your home at all, because the smart lock of your door is blocked by a service component not responding, as it happened in 2019 with Nest devices [80]. Of course, the latter are somewhat trivial examples of the effects that data center incidents may have on the end users. However, more and more organizations are adopting cloud computing, either private or public, as their main paradigm to develop applications with varying levels of criticality. Therefore, it is of the utmost importance for service providers to continuously enhance their capabilities of reacting promptly when such disruptions happen, or even to prevent them altogether, if possible. Even though in cloud environments we can usually find very advanced automations to solve really complex tasks, current approaches to data center operations are still heavily human-centric and depend on the expertise and skills of the on-call engineers. This ensures that highly specialized operators apply mitigation procedures that are usually timely and effective. However, they may still be susceptible to human errors, with non-trivial consequences. Sometimes, all it takes for crashing a very critical service is a typo, as it was the case for the AWS engineer that started a 4-hours downtime of the S3 storage service [234]. In this specific case, since AWS internal services themselves heavily depend on S3 storage, they were not even able to correctly display the status of the operations on their customer-facing dashboards, such that they had to rely on external services like Twitter to dispatch news about the status of the operations [66].

Making sure the provided services are up to the agreed level of performance is of the utmost importance for cloud providers, since such violations may result in penalties, damages to their reputation and, thus, consistent financial losses. For instance, the typical strategies employed to ensure Service Level Agreements (SLA) are static over-provisioning of virtual resources, to make a service more robust to workload fluctuations, and redundant hardware components, to tolerate individual components failures. However, these strategies are clearly cost- and energy-inefficient, and may not be sufficient to handle unforeseen events like error propagations, malicious behaviors or software bugs [209]. In addition, operational activities themselves can be the origin of a large share of system outages. Indeed, in a typical cloud environment, many maintenance tasks are executed simultaneously by multiple operators, possibly relying on separate sets of tools that are not aware of interferences generated by other operations involving the same resources. Due to the complex interactions existing in this type of environments, it is not uncommon that such errors quickly propagate and affect other parts of the system [67].

For all the aforementioned reasons, in order to minimize the risk of outages, and all the related costs, it is crucial to support system administrators introducing novel techniques that can enable a more *proactive*, data-driven, approach to data center operations.

1.2 THE CASE FOR DATA-DRIVEN OPERATIONS

The large availability of data collected at different levels of a cloud infrastructure—like physical machines metrics, virtual machines (VM) metrics, application key performance indicators (KPI), event logs, etc.—allows for employing effective data-driven methods, such as those coming from the machine learning (ML) research field. In a highly dynamic scenario like data center operations, traditional model-driven approaches alone may be inadequate at capturing the complexity of the interactions among system components. Operations teams would certainly benefit from the introduction of robust data-driven methods to support their decisions based on historical information. For instance, outages are typically preceded by failures, performance degradation and similar anomalous behaviors that, if detected on time, can be exploited to raise alerts and/or autonomously trigger suitable procedures *before* other components, or the end users, have the time to experience any issues [16, 67, 82, 209, 245]. On a similar note, also the problems related to virtual resources allocation and management, well-known complex multi-objective optimizations problems, can be efficiently addressed with similar approaches, typically trading optimal solutions for fast—*good-enough*—solutions [54, 81, 155, 179, 194, 216, 260]. Furthermore, data-driven methods can also facilitate root-cause analysis and enable a fine-grained localization of faults, performance [85, 181], energy efficiency and power management issues [60, 96, 267, 269] at scale.

A huge portion of the data available in this context comes in the form of *sequences*. Each physical component of the infrastructure—like physical hosts, top-of-rack (ToR) switches, routers, etc.—continuously generates system logs and readings. On top of that, one can also consider similar information generated by the tons of VMs and containers running on such hardware, as well as the application-level indicators of the services they implement. Cloud environments also generate data that can be intended as *graphs*. A straightforward example could be the topology according to which compute, storage and networking nodes of a data centers are connected to each other and to the public Internet. Furthermore, one could derive graphs from the specific deployments of the various services onto the data center infrastructure, such as fault graphs stating the dependencies among hardware/software components, very useful to highlight single point of failures and address correlated failures. Note that the aforementioned graphs can also be dynamic, such that their structure changes in time. This type of data is usually referred to as *spatio-temporal* graphs (or temporal networks). Finally, another important form of data largely available in cloud environments is *text*. The biggest share of such data obviously consists of the zillions of lines of code that implement the various systems and applications. However, to have a more comprehensive picture, it is important to also consider the other sources of textual data in the DevOps cycle, like: deployment scripts, configuration files, version control system history, code review discussions, post-mortem analysis documents, etc.

It is important to consider that the aforementioned data sources are refreshed with very *high frequency*. On the one hand, this aspect allows for very fine-grained analysis. On the other hand, it poses some computational challenges and requires ML approaches to be robust to distribution drifts, caused by frequent changes to the infrastructure and its configurations made by the operations teams. In this scenario, data are abundant and, in some cases, even too abundant for human operators, or simple automations, to directly consume them. Therefore, advanced data processing techniques are required to extract fundamental knowledge, that may allow operators to really understand the possible insurgence of faults and critical conditions. The overall objective of this thesis is to explore the possibilities to apply ML in this context, developing a structured methodological approach, and providing evidence of the advantages they can bring.

1.3 RESEARCH GOALS

1.3.1 *G1 - Data-driven Operations Support Framework*

The primary goal of this thesis is to devise a comprehensive framework to support data center operations, to allow for a holistic approach to the major problems that can be experienced by service providers. To this aim, data-driven techniques, such as those coming from the ML research field, are expected to be among the most effective ways to address the related research challenges. The motivation is two-fold. First, large-scale distributed cloud environments produce a humongous amount of diverse data, at each level of the technology stack. Second, given the scale of the typical underlying infrastructures that host cloud computing systems, traditional model-driven approaches alone can be inadequate at capturing the complexity of the dynamic interactions and relations among individual components. In this way, data center operators could be provided with a comprehensive set of tools to improve their analysis and to develop more intelligent automations, leveraging on the knowledge acquired from experience, so that they can better focus on those tasks that still heavily require human intervention. Therefore, the first objective of this thesis is to design reliable ML techniques to optimize and support data center operations, and to evaulate their potential benefits by implementing and evaluating them on real cloud platforms.

1.3.2 *G2 - Autonomy vs Learning from Human Interaction*

The target data-driven framework described in Section 1.3.1 is not to be interpreted as a drop-in replacement for human experts. It should be seen as an autonomous system they could safely depend on for critical maintenance tasks, while still having the possibility to provide feedback and place safeguards, filling possible gaps by leveraging on their domain knowledge. Indeed, data centers can be regarded as complex cyber-physical systems (CPS) [192], consisting of a large ensemble of multi-faceted virtual and physical resources, in which human operators play key roles. Given the scale and the performance levels at which such systems are required to operate, it is nearly impossible for humans to handle the workloads and apply the necessary corrective actions entirely manually, without any kind of automation to support them. However, on the other hand, it is still not feasible to automate all the aspects involved in data center operations, since many problems require engineering *creative* and *dependable* solutions leveraging on knowledge coming from many diverse domains, a process that it is currently hard to reproduce artificially. We expect ML-based methodologies to be among the principal and most effective enabling technologies for the realization of such *cooperative* approaches to data center operations. Indeed, they allow for dynamic acquisition of knowledge from historical data, coming from several sources and in different forms. Such capability can be used to anticipate the effect of actions, plans and interactions within the CPS and with the other systems that interface and depend on the provided services. Therefore, the second objective of this thesis is to realize data-driven methods that allow human operators to focus on those activities that inherently require expertise and creativity, by automating resource-demanding tasks and possibly learning from human experts' feedback.

1.3.3 *G3 - Performance & Efficiency*

Even though, in a typical data center scenario, there is certainly no lack of processing power, it is of the utmost importance to take into account also performance and resource-efficiency aspects in the design of the proposed data-driven support system. In particular, we need to make sure it can be operated at a data center-scale without interfering with the production workloads and the other critical services that a data center provides, that are usually regulated by SLAs. Applying state-of-the-art techniques to maximize the control of temporal interferences among the different workloads, such as the ones proposed in [48, 51], is required to deliver automated, dependable and cost-efficient approaches to support data center operations. In these regards, virtual resources allocation policies play a fundamental role. Multiple virtual components, such as VMs or containers, that are co-located on the same physical host might interfere with each other executions (i.e., so-called problem of *noisy neighbors*), compromising each other's performance stability and predictability, such that it is impossible to guarantee the completion of the tasks within the required time constraints. To address this challenge, traditional approaches employ physical resources partitioning and scheduling policies, that could be augmented by providing effective workload estimation models. This kind of issues are particularly relevant in a *private cloud* scenario, a model that is being progressively adopted by those organizations that do not want to, or cannot, rely on external cloud providers due to strict performance requirements. For instance, this is the case of modern NSPs that have started to adopt the Network Function Virtualization (NFV) paradigm to deliver network services. Therefore, the third objective of this thesis is to realize performant and efficient data-driven support tools, that can be plugged into the existing data center infrastructures without negatively affecting the overall performance.

1.4 CONTRIBUTIONS

In view of the research goals presented in Section 1.3, my main contributions generally fulfill all the three principal requisites deemed necessary for a comprehensive, and effective, data-driven approach to data center operations. Furthermore, some of these works have been developed in collaboration with industrial partners, namely *Vodafone* and *Nokia Bell Labs*, that allowed me to work on real data exported from their respective production environments. Table 1.1 offers an

Table 1.1: Summary of the research contributions included in this thesis.

Contribution	G1	G2	G3	Chapters	Publications
High-performance AD	NSPs pain-point	decision-support	GPU acceleration	3, 4	[50, 125, 126, 148]
VNF metrics forecasting	NSPs pain-point	capacity planning	training efficiency	5	[49]
Predictive auto-scaling	elasticity-control	high configurability	low end-to-end latency	6	[123], under review
Intelligent cloud operations	self-healing	recommendations	low overhead	7	under review
Deployment code analysis	quality assurance	knowledge transfer	–	8	[127]

overview of the contributions, and illustrates how they relate to the research goals:

- My work on detecting anomalous Virtual Network Functions (VNF) behaviors [50, 52, 125, 126] offers a data-driven solution to a very important problem that NSPs face in their daily operations (*G1*). However, it aims at complementing human experts assessments, rather than autonomously enacting potentially costly decisions (*G2*). Furthermore, I used self-organising maps (SOM) to implement the proposed approach, a very *low-footprint* unsupervised learning method, that I made even more efficient by leveraging on hardware acceleration [148] (*G3*).

- In my work on VNF metrics forecasting [49, 53], I provide a comprehensive overview of time-series forecasting methods that can be used by NSPs to support their capacity planning activities (*G1*, *G2*). In the experimental validation, I also assess the proposed architectures both in terms of accuracy and training efficiency (*G3*).

- My work on predictive auto-scaling [123] tackles a fundamental open problem in cloud computing (*G1*). I designed the proposed architecture such that human operators retain full-control over what underlying ML models are used and how the monitoring data are ingested, to realize a robust elasticity-control policy that can be refined over time (*G2*). In addition, I show that the approach introduces a very limited latency to the overall control mechanism (*G3*).

- My approach to *intelligent* operations, presented in Chapter 7 (unpublished work, currently under review), offers to human operators a way to embed their experience into a model able to *recommend* (*G2*) the most suitable corrective actions, when facing important issues that were already observed in the past (*G1*). This is possible, for instance, by correlating the information stored in issue tracking systems and post-mortem analysis documents to system measurements. Remarkably, I used gradient-boosted trees (GBT) to implement the proposed approach, a simple, yet very powerful, method for supervised learning, that exhibits a generally short training time and low inference overhead (*G3*).

- My work on the application of large language models (LLM) to declarative deployment code analysis [127], presented in Chapter 8, aims at providing an effective method to automate quality-assurance (QA) activities (*G1*). In this way, human experts can mitigate the impact of time-consuming activities like code-reviews and knowledge transfers, while still enforcing best practices and recommended design patterns (*G2*).

1.5 THESIS STRUCTURE

This thesis is organized as follows. Part I introduces the reader to the motivations and the scope of our work. In particular, in Chapter 2, we provide an overview of the most relevant background concepts required to understand the rest of the thesis. Our work is at the intersection of several, very broad, fields like Cloud Computing, NFV and ML. Therefore, for each field, we limit our overview to the key principles and technologies that inspired and enabled our research contributions.

Part II includes our work on high-performance SOM-based anomaly detection (AD). In Chapter 3, we describe our approach based on SOMs for detecting anomalous behaviors in VNFs. Our approach consists in a joint analysis of system-level resource consumption metrics, collected either from the physical hosts or the computational units (e.g., VMs, containers), and application-level metrics related to the performance of the individual VNF. We present the results of our validation on real data coming from a subset of the *Vodafone* NFV infrastructure, where it is currently employed to support the data center operators. Our technique is capable of identifying specific components of the infrastructure that are worth to be investigated by human operators, in order to keep the system running under expected conditions. The contents of Chapter 3 appeared previously in [50, 52, 125, 126]. In Chapter 4, we describe *XPySom*, our open-source implementation of the SOM technique, written in Python. This work was inspired by our previous experience with open-source SOM implementations, depicted in Chapter 3. Our implementation is designed to achieve high single-node performance, leveraging on the wide landscape of libraries for vector processing on multi-core CPUs and general purpose GPUs. We present the results of an extensive experimental evaluation, where we benchmarked our approach against widely used open-source SOM implementations, using the EMNIST dataset. Our results shows that *XPySom* outperforms the other available alternatives, in the single-node scenario. Indeed, under the same conditions in terms of quantization error, our implementation exhibits a speed-up of about 7x and 100x, with respect to the considered alternative open-source implementations, when multi-core and GPU acceleration are enabled, respectively. The contents of Chapter 4 appeared previously in [148].

Part III includes our work on cloud resource management approaches based on time-series analysis techniques. In Chapter 5, we present our work on time-series forecasting techniques for predicting VNF operation metrics. We investigate the performance of a number of ML-based approaches, and provide insights on how they can support the decisions of NFV operation teams. Our analysis considers both infrastructure-level and service-level metrics. To benchmark the selected forecasting techniques, in terms of forecasting accuracy and training cost, we used real data exported from a production environment deployed within the *Vodafone* NFV infrastructure. Vodafone Network Virtual Infrastructure (NVI) team currently employs these techniques to support its capacity planning activities. The contents of Chapter 5 appeared previously in [49, 53]. In Chapter 6, we describe *monasca-predictor* [122], our open-source extension to the Monasca [170] monitoring framework that enables OpenStack operations based on predictive analytics, with a particular focus on auto-scaling. Indeed, our architecture allows orchestrators to apply time-series forecasting techniques to estimate the evolution of relevant metrics, and take decisions based on the predicted state of the system, instead of simply reacting when thresholds on resource consumption metrics are breached. In this way, e.g., they can anticipate load peaks, and trigger appropriate scaling actions in advance, such that new resources are available when needed. We used our architecture to implement predictive scaling policies leveraging on linear regression (LR), autoregressive integrate moving average (ARIMA), feed-forward neural networks (FFNN) and recurrent neural networks RNN. Then, we evaluated their performance on a synthetic workload, comparing them to those of a traditional policy. Furthermore, to assess the ability of the different models to generalize to unseen patterns, we also evaluated them on traces from a real content delivery network (CDN) workload. The implementation of our architecture is open-source. The contents of Chapter 6 appeared previously in [123], and also refer to an unpublished journal paper, currently under review. In Chapter 7, we present our work on a 2-phase strategy to enable *intelligent* cloud operations. Our approach leverages on monitoring data and the information regarding the occurred anomalies, correlating them with the corresponding corrective actions. The approach consists in a ML pipeline, composed by two models in sequence, to automatically detect anomalous patterns, based on past observations of normal behavior, and recommend specific corrective actions, based on historical operational data reporting the strategies applied to heal the faulty components. We validate our approach on an OpenStack deployment, generating workloads on both a synthetic application and a Cassandra cluster, while injecting different types of faults. The contents of Chapter 7 refer to unpublished work, currently under review.

Part IV includes our work on QA-aware DevOps methodologies. In Chapter 8, we present our approach based on LLMs for analyzing declarative deployment code, to support DevOps activities. This work was conducted during an internship at *Nokia Bell Labs* (NBL) in Stuttgart, Germany. Our goal was to create a tool to support DevOps teams in their QA activities, with a focus on declarative infrastructure-as-code (IaC) specifications, like Kubernetes manifest files. Indeed, producing robust deployment specifications is not an easy feat, and for the domain experts it is time-consuming to conduct code-reviews and transfer the appropriate knowledge to other members of the team. This is particularly challenging in a context like NBL, where projects are typically developed by multidisciplinary research teams with a very diverse expertise. We propose an approach based on LLMs to automatically provide QA-related recommendations to developers, such that they can benefit of established best practices and design patterns. We developed a prototype of our proposed ML pipeline, and empirically evaluated our approach on a collection of Kubernetes manifest files exported from a repository of internal projects at NBL. The contents of Chapter 8 appeared previously in [127].

Part V concludes the thesis. In particular, in Chapter 9, we summarize our main research contributions and outline remaining open problems and possible future research directions.

2

BACKGROUND

In this chapter, we limit our presentation of the background concepts to only those key principles and technologies, in the space of Cloud Computing, NFV, and ML, that inspired and enabled the research contributions of this thesis.

2.1 CLOUD COMPUTING TECHNOLOGIES

In this section, we briefly recall basic concepts about the most popular cloud management and orchestration frameworks, both in open-source and industry.

2.1.1 *OpenStack*

OpenStack is a widely used *open-source* cloud computing platform that offers a wide range of diverse services. Many of the research contributions of this thesis have been implemented and validated using a local OpenStack cluster. Therefore, in this section, we provide details on a number of key OpenStack components, with reference to Figure 2.1.

2.1.1.1 *Nova, Cinder and Glance*

Nova [172] is the OpenStack component that provides compute resources (e.g., VMs, bare metal servers, containers) management functionalities. It leverages on Cinder [166] for block storage management, and Glance [167] for image provisioning. The core of Nova's architecture is the *compute* process, that manages the underlying hypervisor (using *libvirt*). Such process communicates with the shared central database through the *conductor* process. Finally, the *scheduler* process is the interface between the *compute* process and the instance *placement* service. All processes exchange requests via remote procedure call (RPC).

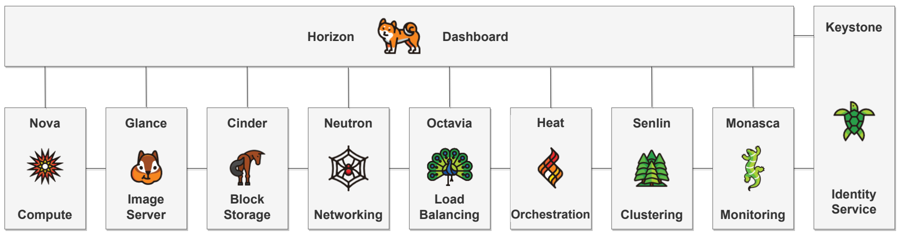


Figure 2.1: Overview of OpenStack key components.

2.1.1.2 *Neutron*

Neutron [171] is the OpenStack component that provides networking functionalities. It offers the possibility to manage per-tenant virtual networks (e.g., having their own IP numbering and DHCP settings) and can be equipped with security-related features, like firewalls and VPNs.

2.1.1.3 *Monasca*

Monasca [170] is an advanced multi-tenant, highly scalable, and fault-tolerant monitoring solution. It is designed as a collection of microservices, including: an efficient time-series DB, a streaming alarm engine, a notification engine, a message queue, etc. Monasca also provides an agent module that is to be deployed on the physical machines hosting the compute services, such that it can collect metrics and forward them to the DB through the message queue. Monasca is also compatible with Kubernetes.

2.1.1.4 *Heat*

Heat [168] is the OpenStack component that provides orchestration and automation capabilities. Indeed, it includes an IaC solution, namely the Heat Orchestration Templates (HOT), through which users can define declarative deployment specifications, and automate the creation and configuration of OpenStack resources in a repeatable and consistent way. HOTs are based on the AWS *CloudFormation* template format, that can be written either in YAML or plain JSON. Heat is designed to be highly scalable and can be used to deploy complex multi-tier applications.

2.1.1.5 *Senlin*

Senlin [174] is the OpenStack component that offers tools to effectively operate *clusters* of homogeneous OpenStack resources (e.g., Nova instances). In particular, it is possible to define and attach *policies* to such clusters, specifying how their resources must be treated under specific conditions. For instance, one can use *scaling* policies to automatically resize the cluster, load-balancing (LB) policies to distribute the workloads, or *health* policies to handle faulty instances. Compared to Heat [168], Senlin offers more effective operation support tools, and a finer-grained control over the underlying resources. Indeed, Senlin is being successfully used to operate large-scale deployments, like the on-line gaming use-case reported by [233].

2.1.1.6 *Octavia*

Octavia [173] is the former *Neutron LBaaS* and, as the name suggests, offers scalable LB functionalities. LB is crucial to enable fundamental cloud properties like *elasticity* and *high-availability*. An Octavia LB consists of a horizontally-scalable pool of Nova instances (i.e., *amphorae*), leveraging on *HAProxy*. The *controller* is the core of Octavia's architecture, consisting in a number of sub-components whose jobs include handling API requests and orchestrating the amphorae.

2.1.2 *Kubernetes*

Kubernetes [118] is an open-source orchestration platform that automates the deployment, scaling, and management of containerized services. Originally developed by Google, building on top of the experience made with its famous internal platform *Borg* [241], it is now among the *graduated* projects maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes leverages on containers as virtualization technology to package and run software applications and their dependencies. Containers belong to the class of *OS-level* virtualization solutions, offering an efficient way to develop and operate portable software. With respect to traditional alternatives like VMs (i.e., *hardware-level* virtualization), containers trade strong isolation guarantees for significantly faster boot times and less demanding resource requirements. The primary goal of Kubernetes is to facilitate operating and deploying resilient containerized applications across distributed infrastructures. Indeed, managing a large-scale production environment, composed of a huge number of containers, distributed across multiple physical nodes, is quite the challenge. Kubernetes offers a set of very effective tools, and a growing ecosystem, to easily handle large compute clusters through sophisticated *control-plane* automations and configuration management. For instance, once containers are deployed, the orchestrator (specifically, the *kube-scheduler* component) automatically takes care of scheduling them to the appropriate physical nodes, based on resource availability and the specified workload requirements. Furthermore, Kubernetes provides built-in scaling capabilities like, e.g., the *HorizontalPodAutoscaler* (HPA) component, that automatically adjusts the number of *pods* (i.e., the abstraction used to group related containers) in order to meet current demand. On a related note, Kubernetes also offers automatic service discovery and LB, that allows for seamlessly distributing the load among replicated instances of a given service. Applications running on Kubernetes can also be easily configured through *declarative* specifications, the so-called *manifest* files, to implement self-healing features, such that the underlying containers are automatically restarted, or failed-over, when their performance is degraded. Similarly, an operator can easily specify, through the same manifests, the preferred update rollout and rollback strategies. The possibility to specify, in a declarative way, virtually any aspect of an application life-cycle is the feature that mostly reflects the philosophy that Kubernetes was built with. Its approach to cloud orchestration completely eliminates the need for developers and operators to explicitly codify any kind of workflow, letting them focus on only specifying the desired state of the system. Kubernetes' composable control processes (specifically, the *kube-controller-manager* component) automatically take care of making the system continuously converge towards such state. In this way, there is no need for centralized control processes, and the resulting distributed systems are generally more robust. Being open-source software, Kubernetes is designed to be highly flexible and extensible, and can be successfully used to create private cloud deployments in on-premise data centers. However, nowadays, any major cloud provider, like AWS, Google Cloud Platform (GCP), or Microsoft Azure, offers a managed Kubernetes distribution that can be leveraged on to reduce the operational overhead.

2.1.3 *VMWare vRealize Operations Manager*

VMWare vROps [243] is an enterprise-grade software used to operate cloud infrastructures. Such framework can be deployed either on-premise or in the cloud and its main purpose is to support operations teams in automating and continuously improving their fundamental activities, also leveraging on data-driven methodologies. Indeed, the core of vROps consists of a pervasive monitoring infrastructure that collects system data at every level of the stack (e.g., physical hosts, virtual machines, networking components, etc.) and feed them to a powerful analytical engine that is able to provide useful insights and actionable feedback to the human operators, such that possible issues or anomalies can be early spotted and corrected. More than 300 system metrics, being them classical raw counters (e.g., CPU utilization, memory contention, network traffic, etc.) or more convoluted analytics computed by the engine, can be exported from the system, allowing also for the integration with third-party tools. Besides monitoring and alerting functionalities, vROps enables automated management of the VMs (or containers) that compose the deployed applications such that, for instance, the corresponding workloads can be balanced according to the optimization of specified indicators (e.g., application KPIs, licensing costs, etc.).

2.1.4 *Network Functions Virtualization*

NFV [39] is an architectural paradigm that emerged in response to the increasingly demanding performance, flexibility and resiliency requirements that modern telecommunication systems have to adhere to. Among the typical network functions managed by a NSP, we have: firewalls; intrusion detection systems; application LBs; DNS services; IP multimedia subsystems; wide-area network (WAN) accelerators; etc. NFV basically consists in virtualizing such network functions and consolidating them on commodity physical servers. Indeed, traditional approaches that rely on proprietary *dedicated* physical appliances, sized for the peak-hour, are nowadays too costly to maintain, and do not allow NSPs to conduct a business that is sustainable at scale. The adoption of NFV has been undoubtedly enabled by the tremendous evolution of *cloud computing*, that makes having on-demand access to virtualized resources unprecedentedly easy and convenient. In the NFV context, the *private* cloud model is the most widespread, given that NSPs typically already own many on-premise computing, networking and storage resources, and that they have to meet quite stringent SLAs, in terms of performance and security. In this way, NFV offers several benefits over traditional approaches. For instance, by decoupling the network functions from dedicated hardware, it allows for network services to be delivered more flexibly, and to be updated more rapidly in response to changing business needs, drastically shortening the time-to-market. Also, by consolidating functions on commodity hardware, NFV enables more efficient resource utilization strategies, including auto-scaling capabilities. The NFV framework is made of three fundamental components. First, the VNFs, i.e., the software-defined network functions, that are typically packaged and deployed onto the physical infrastructures as VMs. However, given the widespread adoption of container-based virtualization in the IT industry, we are observing that also NSP are increasingly transitioning to *cloud-native* approaches for VNFs development. Second, the NFV Infrastructure (NFVI), that is the collection of hardware and software components that NSPs have to maintain in order to provide the required resources to VNFs. An NFVI is typically a very large-scale system, whose components are distributed among many geographically dislocated data centers managed by the NSP. Such system generally consists of physical compute and storage nodes, and virtualization-aware network equipment. Third, the NFV management and orchestration (MANO) architecture, that is collection of all functions that jointly manages the lifecycle of VNFs and the underlying NFVI. Indeed, MANO functions typically include: provisioning and managing NFVI resources to be allocated to specific VNFs; instantiate, scale, and terminate individual VNFs; orchestrating all the deployed VNFs to make sure the implemented services are effectively, and efficiently, delivered.

2.2 MACHINE LEARNING METHODOLOGIES FOR OPERATIONS SUPPORT

In this section, we focus on a selected number of ML techniques that have been used to realize the research contributions presented in this thesis.

2.2.1 *Self-Organizing Maps*

A SOM [115] is an unsupervised vector quantization technique, used to produce a topology-preserving map using a competitive learning algorithm. The aim of the SOM training algorithm is to encode a data manifold (e.g., a sub-manifold $V \subseteq \mathbb{R}^N$) into a finite set $W = \{w_1, \cdots, w_M\}$ of reference vectors where $w_i \in \mathbb{R}^N$ is called *codebook*. Formally, a SOM is defined by a pair of maps (w, b). $w : \mathcal{L} \to \mathbb{R}^N$ is a discrete map from a 2D lattice into a finite vector space, also known as *features space*. Recall that a 2D lattice of dimensions $H \times K$ is a discrete set

$$\mathcal{L} = \{hA + kB \,|\, h < H,\, k < K,\, h, k \in \mathbb{N}\} \subseteq \mathbb{R}^2 \tag{2.1}$$

where $A, B \in \mathbb{R}^2$ determine its shape (e.g., $A = (1, 0)$ and $B = (0, 1)$ produce a *rectangular* grid, whereas $A = (\frac{1}{2}, \frac{\sqrt{3}}{2})$ and $B = (1, 0)$ produce an *hexagonal* grid). For the sake of simplicity, $\mathcal{L}$ is indexed with a lexicographical order (from 1 to $H \times K$), its elements $r_i \in \mathbb{R}^2$ are called *units* or also *neurons* and the images $w_i = w(r_i)$ of the neurons in the features space are called *weights*. Given a sample vector $x \in V$, $b : \mathbb{R}^N \to \mathcal{L}$ returns the BMU i.e., the unit whose weight is closest to the input sample (or any such units, if multiple ones exist) depending on a distance d in the feature space:

$$b(x) \in \operatorname*{arg\,min}_{r \in \mathcal{L}} d(x, w(r)) \tag{2.2}$$

A common choice for the distance d is the Euclidean distance, albeit alternative choices are possible (e.g., see the discussion in Section 3.3.3).

For each training epoch t, an update to the SOM weights is performed, for each input sample, as follows. At each iteration *iter*, an *input data* x is fetched (using either a random or a sequential scheduling) and its associated best matching neuron $b(x)$ is computed. Then, the weights of *all neurons* are updated according to Equation (2.3), where h is called *neighborhood function* and is defined as Equation (2.4) (assumed to be a Gaussian in what follows).

$$w_k^{(iter+1)} = w_k^{(iter)} + \alpha(t) h(b(x), r_k, t)(x - w_k^{(iter)}) \quad \forall k \tag{2.3}$$

$$h(r, s, t) = -\exp\left(\frac{\|r - s\|^2}{\delta(t)}\right) \quad \forall r, s \in \mathcal{L} \tag{2.4}$$

Here, $\alpha(t)$ and $\delta(t)$ are respectively the learning rate and the radius of the neighborhood function, which depend on the current epoch t ($\alpha, \delta : \mathbb{N} \to \mathbb{R}$), and decrease across epochs either linearly or exponentially, to make the algorithm converge. It is important to notice that, for each training sample x, not only the winning reference is modified, but the adaptation to x affects all the weights w_j depending on the proximity of r_j to $b(x)$ with a step size that decreases with the distance between the units r_j and $b(x)$ in the lattice. This way neighboring units respond to similar input patterns and each data point close in the input space is mapped to same or nearby map neurons (inducing a topology-preserving property on the codebook). The weights of the neurons w_i are typically initialized either by randomly sampling the V data set or using the well-known PCA.

A key difference between the SOM training algorithm and other vector quantization or clustering techniques is that, in the neighborhood function definition (see Equation (2.4)), the topological distance between a pair of units is declined as the Euclidean distance on the map and not in the data space. The formulation in Equation (2.3) is called the *online update* rule, that is not suitable for a parallel implementation since each iteration directly depends on the one immediately before and only processes a single data sample at a time. Therefore, a *batch parallel* implementation has been proposed: instead of updating the neuron weights for each data sample, they are updated after a batch $B \subseteq V$ of N' data samples (in the following we will assume $N' = N$, for the sake of simplicity). Essentially, the term of Equation (2.3) that depends for each $r_k \in \mathcal{L}$ on the input sample by $h(b(x), r_k, t)(x - w_k)$ is replaced by a weighted sum of the same terms computed in parallel for all samples in the batch, using the formula:

$$\frac{\sum_{x \in B} h(b(x), r_k, t)\,(x - w_k)}{\sum_{x \in B} h(b(x), r_k, t)} \quad \forall k \tag{2.5}$$

This way, one can compute in parallel all numerator and denominator parts (i.e., $h(b(x), r_k, t)(x - w_k)$ and $h(b(x), r_k, t)$, respectively) for each sample in each batch and then sum up all numerator and denominator parts and finally compute the weight update.

After training is complete, the result is that the manifold data V is divided into a finite number of subregions:

$$V_i = \left\{ x \in V | \, \|x - w_i\|_2 \leq \left\|x - w_j\right\|_2 \quad \forall j \neq i \right\} \tag{2.6}$$

called Voronoi tessellation, in which each sample vector x is described by the corresponding weight of the BMU $w(b(x))$. It is important to point out that the update rule is fundamental to the formation of the topographically ordered map. In fact, the weights are not modified independently of each other but as topologically related subsets. For each step a subset of neurons is selected on the basis of the neighborhood of the current winning unit. Hence, topological information is supplied to the map because both the winning unit and its lattice neighbors receive similar weights updates that allow them, after learning, to respond to similar inputs. After the training phase, the map can be used to visualize different features of the codebook and of the represented data, such as (i) the density of the reference vectors (e.g., with a color scale proportional to neuron hits); (ii) likewise, the distances among reference vectors, where a dark color indicates a large distance between adjacent units, and a light color indicates a small distance (i.e., the so-called U-matrix); (iii) a plot of the reference vectors for each neuron, to see at a glance all the different behaviors detected in the training dataset.

A careful choice of the SOM hyperparameters should be made in order to have a suitable trade-off in terms of quality of the clustering and computational performance (see, e.g., Section 3.4). Also, in order to mitigate the problem of having different neurons specializing on almost the same data samples (e.g., when the number of SOM neurons is large with respect to the data sample variability), it is possible to apply multi-stage clustering techniques over the SOM reference codebook (see, e.g., Section 3.4.4).

2.2.2 *ARMA, ARIMA, and SARIMA*

The auto-regressive moving average (ARMA) model provides a flexible tool for forecasting. Given the samples[1] $\{x_t\}$ up to time $t-1$, the forecasted sample $\hat{x}_t$ is computed as:

$$\hat{x}_t = \phi_0 + \sum_{i=1}^{p} \phi_i x_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} \tag{2.7}$$

where: ϕ_0 is a constant; $\{\phi_i\}_{i=1}^{p}$ are the model parameters controlling the linear dependency of $\hat{x}_t$ from the p last samples of the signal; $\{\theta_j\}_{j=1}^{q}$ are the parameters defining the linear dependency of the output from the q errors $\{\epsilon_{t-j} \triangleq \hat{x}_{t-j} - x_{t-j}\}_{j=1}^{q}$ performed by the model on the last q predictions.

From ARMA, it is possible to derive another technique, known as auto-regressive integrated moving average (ARIMA), that can be used for non-stationary time-series. ARIMA is actually an ARMA model applied to the $d-$order differenced signal (defined as $x_t^{(1)} \triangleq x_t - x_{t-1}$, for $d = 1$, and $x_t^{(d)} \triangleq x_t^{(d-1)} - x_{t-1}^{(d-1)}$, for $d > 1$), for some $d \in \mathbb{N}^+$, to obtain $d-$order differenced forecasts $x_t^{(d)}$, that need to be integrated to reconstruct the final forecast $\hat{x}_t$:

$$\hat{x}_t^{(d)} = \phi_0 + \sum_{i=1}^{p} \phi_i x_{t-i}^{(d)} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j}^{(d)} \tag{2.8}$$

1 In what follows, $\{x_t\}_{t\in\mathbb{Z}}$ with $x_t \in \mathbb{R}^n$ denotes a generic discrete, possibly multivariate, time-series whose historical evolution is known up to the current time. Its future evolution $\{\hat{x}_t\}$ is to be predicted with one of the mentioned techniques.

Equation (2.8) is referred to as an ARIMA(p,d,q) model. From such a general definition, it is possible to derive simpler models by tuning the meta-parameters (p,d,q). For instance:

- When $d = q = 0$, an AR model AR$(p) \equiv$ ARIMA$(p,0,0)$ is obtained, where x_t is a linear combination of its lagged values up to $t-p$.
- When $p = d = 0$, an MA model MA$(q) \equiv$ ARIMA$(0,0,q)$ is obtained, where x_t is a linear combination of the errors at previous timestamps up to $t-q$, not to be confused with moving average filtering.

The meta-parameter d is typically chosen to obtain a $d-$order differenced time-series $x_t^{(d)}$ that is *stationary*, i.e., whose mean, variance and auto-correlation are independent of t.

ARIMA can be further extended to deal with *seasonal* patterns by introducing additional terms in Equation (2.8). Such a variant is known as seasonal auto-regressive integrated moving average (SARIMA). Given a seasonality period of m samples and meta-parameters (P,D,Q), that are seasonal equivalents of (p,d,q), we get an additional component that corresponds to an ARIMA model, where the signal values (and errors) at $t-m, t-2m, t-3m, \ldots$ are used to compute the output at t. The parameters of the model are usually optimized via least-square optimization or likelihood maximization with Kalman filters [104].

2.2.3 *Holt-Winters*

Holt-Winters (HW), often referred to as *triple exponential smoothing* [33], is a commonly adopted method for forecasting and signal processing. Its peculiarity consists in explicitly separating predictive components into *level* (or expected value, l_t), *trend* (b_t) and *seasonality* (s_t). In its *additive* form, given knowledge of the samples $\{x_t\}$ up to the current time t, HW forecasts future samples $\hat{x}_{t+h}$ for $h \in \mathbb{N}^+$ as

$$\begin{aligned}
\hat{x}_{t+h} &= l_t + (\phi + \phi^2 + \cdots + \phi^h)b_t + s_{t+h-m} \\
l_t &= \alpha(x_t - s_{t-m}) + (1-\alpha)(l_{t-1} + \phi b_{t-1}) \\
b_t &= \beta(l_t - l_{t-1}) + (1-\beta)\phi b_{t-1} \\
s_t &= \gamma(x_t - l_{t-1} - \phi b_{t-1}) + (1-\gamma)s_{t-m}
\end{aligned} \tag{2.9}$$

where: m is the seasonality period; $0 < \alpha, \beta, \gamma < 1$ are the smoothing factors for the level, trend and seasonality forecasts; $0 < \phi \leq 1$ is

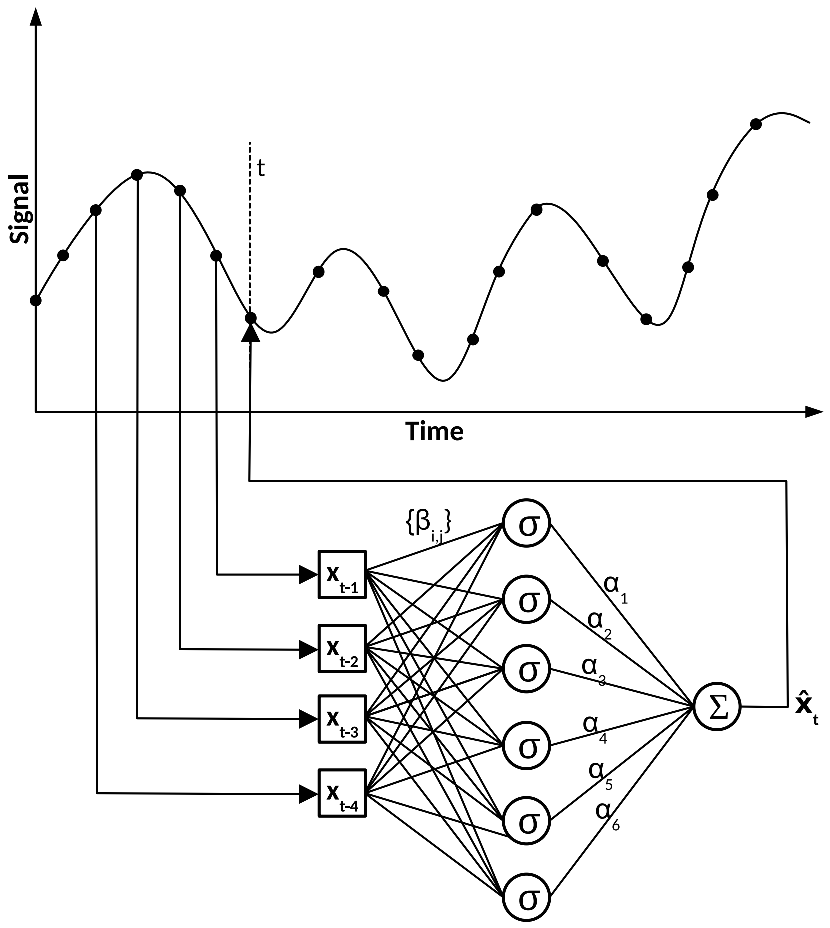


Figure 2.2: NAR neural network for time-series forecasting.

the damping factor. Alternatively, HW can be also formalized in a *multiplicative* form as

$$\begin{aligned}
\hat{x}_{t+h} &= [l_t + (\phi + \phi^2 + \cdots + \phi^h)b_t] \cdot s_{t+h-m} \\
l_t &= \alpha \frac{x_t}{s_{t-m}} + (1-\alpha)(l_{t-1} + \phi b_{t-1}) \\
b_t &= \beta(l_t - l_{t-1}) + (1-\beta)\phi b_{t-1} \\
s_t &= \gamma \frac{x_t}{l_{t-1} + \phi b_{t-1}} + (1-\gamma)s_{t-m}
\end{aligned} \tag{2.10}$$

HW parameters are optimized via sequential least-squares programming, a classical optimization technique for constrained minimization problems.

2.2.4 *Non-linear Auto-Regressive Neural Networks*

A non-linear auto-regressive (NAR) neural network is an auto-regressive model where the forecast $\hat{x}_t$ at time t is a non-linear combination of the last p observations of the input signal. In the case of the so-called *tunable-basis*, the estimated sample is obtained as a linear combination of r non-linear functions, where each function f_i processes the same input using a set of q_i specialized parameters $\beta_{i,1}, \dots, \beta_{i,q_i} \in \mathbb{R}$:

$$\hat{x}_t = \sum_{i=1}^{r} \alpha_i f_i(x_{t-1}, \dots, x_{t-p}, \beta_{i,1}, \dots, \beta_{i,q_i}) \tag{2.11}$$

The one-hidden layer *sigmoidal* neural network is a notable instance of this type of functions. In this case, the parameters are optimized via back-propagation [136, 203], using input-target pairs consisting of past and current observations. Figure 2.2 represents a NAR forecasting model with a single hidden layer in the case of $p = 4$ and $r = 6$.

2.2.5 *Recurrent Neural Networks*

Among artificial neural networks (ANN), recurrent neural networks (RNN) are commonly used for multi-variate time-series analysis (see [78], Chapter 10), and forecasting in particular. RNNs predict the one-step (or k-step) ahead value of a time-series based on the current I-dimensional input $x_t \in \mathbb{R}^I$ and a compressed history of the inputs, stored in an H-dimensional state vector $s \in \mathbb{R}^H$ computed recurrently by the hidden neurons. The model evolution is described by

$$s_t = f_s(s_{t-1}, x_t) \tag{2.12}$$

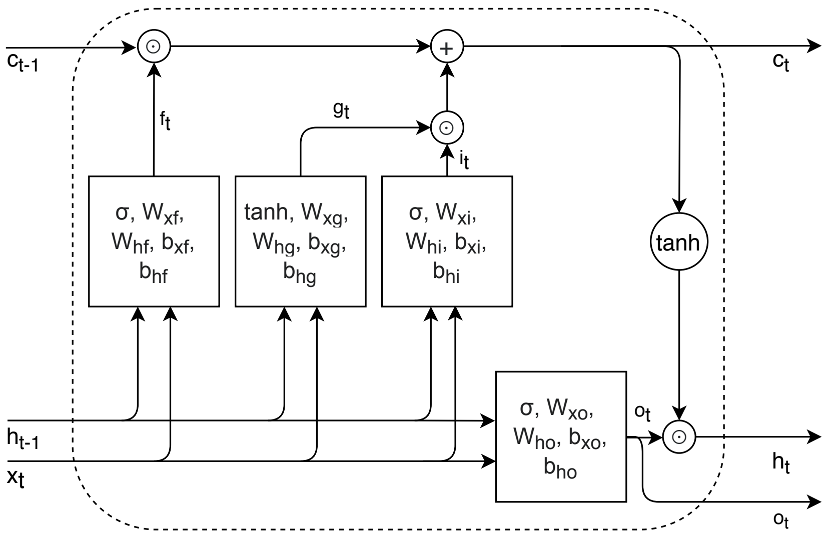
$$o_t = f_o(s_t, x_t) \tag{2.13}$$

where: *(i)* $f_s : \mathbb{R}^{H+I} \to \mathbb{R}^H$ operates on the concatenation c of s and x, and is defined, e.g., as $f_s = \tanh(W_s c + b_s)$, where W_s are the weights, b_s are the biases; *(ii)* $f_o : \mathbb{R}^{H+I} \to \mathbb{R}^O$ is the output function, similarly defined, e.g., as $f_o = \mathrm{ReLU}(W_o c + b_o)$, where ReLU denotes the *Rectified Linear Unit* function. The learnable parameters $\theta = \{\theta_j\} = \{W_s, W_o, b_s, b_o\}$ are typically trained through *gradient descent* on the loss function. In this work, we consider a stochastic update rule with momentum, so that the j-th parameter is updated at the k-th optimization step as follows:

$$\mu_{j,k} = \beta \mu_{j,k-1} + \nabla J_{\theta_{j,k}}(D) \tag{2.14}$$

$$\theta_{j,k+1} = \theta_{j,k} - \lambda \mu_{j,k} \tag{2.15}$$

where D is a dataset of input-output pairs, and $\nabla J_{\theta_{j,k}}(D)$ is the gradient of the loss function $J_{\theta_{j,k}}(D)$ with respect to parameter θ_j computed at instant k. The term β determines in which proportion the momentum $\mu_{j,k}$ is applied during the gradient descent step, and λ is the learning rate. The training process continues until the validation loss, computed over the validation dataset every K optimization steps, stops decreasing. The model corresponding to the minimum achieved on the validation loss is taken as output.



c_{t-1}
c_t
f_t
g_t
i_t
σ, W_xf, W_hf, b_xf, b_hf
tanh, W_xg, W_hg, b_xg, b_hg
σ, W_xi, W_hi, b_xi, b_hi
tanh
σ, W_xo, W_ho, b_xo, b_ho
o_t
h_{t-1}
x_t
h_t
o_t

Figure 2.3: LSTM cell processing a time-series together with the additional internal signals c_t and h_t.

2.2.6 *Long Short-Term Memory*

As to more advanced ML techniques, long short-term memory (LSTM) networks [211] represent a common architecture for time-series classification and forecasting [11, 226], and consist of an improved version of RNNs. Let $\{x_t\}$, for $x_t \in \mathbb{R}^n$ and $t \in [0, \tau]$, be a discrete multi-variate input sequence. We can formalize how an LSTM cell with d units processes its inputs at time t as

$$\begin{aligned}
i_t &= \sigma\left(W_{xi}x_t + b_{xi} + W_{hi}h_{t-1} + b_{hi}\right)\\
f_t &= \sigma\left(W_{xf}x_t + b_{xf} + W_{hf}h_{t-1} + b_{hf}\right)\\
g_t &= \tanh\left(W_{xg}x_t + b_{xg} + W_{hg}h_{t-1} + b_{hg}\right)\\
o_t &= \sigma\left(W_{xo}x_t + b_{xo} + W_{ho}h_{t-1} + b_{ho}\right)\\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t\\
h_t &= o_t \odot \tanh\left(c_t\right)
\end{aligned} \tag{2.16}$$

where each W is a (learnable) weight matrix, $\odot$ is the element-wise (or Hadamard) product and $\sigma(y) = 1/(1+e^{-y})$ is the (element-wise) sigmoidal function. We refer to $\tilde{h}_t = (h_t, c_t) \in \mathbb{R}^{2d}$ as the *latent state* at time t.

Figure 2.3 visualizes how the *gates* that compose an LSTM cell are connected with each other. In practical terms, LSTM leverages on the interactions between the *cell state* c_t, the *hidden state* h_t and the current value of the signal x_t to formulate an internal representation of the input variables, that captures long- and short-term dependencies among them. The main reason why LSTMs show an advantage when compared to standard RNNs is that the cell state can traverse the cell freely without being altered, other than linear interactions as summation and element-wise product, in backward and forward passes of the learning algorithm, thus reducing the risk of numerical problems affecting regular RNNs, such as the *vanishing gradient* [116]. The exact way $\hat{x}_t$ is computed depends on the architecture where the LSTM cell is plugged into.

2.2.7 *Gradient-Boosted Trees*

A gradient-boosted tree (GBT) is an ensemble method for supervised learning that combines the simplicity of decision trees (DT) [88] with the power of the gradient boosting algorithm [73]. GBTs are nowadays widely used to solve diverse types of learning tasks (e.g., classification, regression, ranking, etc.). In particular, due their proven capability to handle complex non-linear relationships, they exhibit remarkable performance when dealing with tabular data, often outperforming even more sophisticated models like ANNs. A GBT consists in a set of classification and regression trees (CART). However, unlike standard DTs, where leaves only contain decision values, each leaf of a CART is associated with a real-valued prediction score. This allows for a more sophisticated optimization approach, able to solve more complex learning tasks than simple classification. The final output $\hat{y}_i$ of a CART is typically computed by summing up the prediction scores of each individual tree in the ensemble:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F} \tag{2.17}$$

where K is the number of trees in the ensemble, and f_k is the function representing a specific tree in $\mathcal{F}$, the set of all possible CARTs. Looking at Equation (2.17), one may argue that GBTs are not so different from random decision forests (RDF) [88]. Indeed, both types of statistical models consist in tree ensembles. However, there exists a fundamental difference in the procedures used to train such models. For GBTs, the objective function to be optimized is usually defined as

$$\text{obj} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \omega(f_i) \tag{2.18}$$

where l is an arbitrary loss function, and $\omega(f_i)$ is the so-called *complexity* of the tree f_i. GBTs work by iteratively learning, at each time-step, a new DT to be added to the ensemble, with each subsequent tree being optimized for correcting the errors made by the previous one. The final output of the model at time t is defined as

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i) \tag{2.19}$$

The gradient boosting algorithm typically uses gradient descent to minimize the loss function—like mean squared error (MSE)—by iteratively fitting new trees to the residual errors. This process continues until the loss function is minimized, or a specified stopping criterion is met. The aforementioned complexity acts as a *regularization* term that, for instance, can be defined as follows:

$$\begin{aligned} f_t(x) &= w_{q(x)}, \quad w \in \mathbb{R}^T,\, q : \mathbb{R}^d \to \{1, 2, \ldots, T\} \\ \omega(f) &= \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \end{aligned} \tag{2.20}$$

where the tree f_t is defined in terms of w, that is the vector of prediction scores on its leaves, and q, that is a function assigning each d-dimensional data point to the corresponding leaf. T is the number of leaves of f_t, while γ and λ are parameters that control the effect of the regularization. Beside being able to handle complex non-linear relationships, and particularly robust to noisy and missing data, GBTs are also generally sufficiently easy to interpret and visualize. Furthermore, they have a quite low barrier to adoption, as there already exist several established, and highly performing, open-source implementations, like *XGBoost* [38] and *LightGBM* [108].

2.2.8 *Large Language Models*

Large language models (LLM) are a type of deep neural networks (DNN), consisting of *millions*, or even *billions*, of learnable parameters, designed for solving very complex natural language processing (NLP) tasks. The training process typically starts by *pre-training* an LLM for *language modeling*, on massive amounts of textual data. In general, the input data consists of a sequence of words (or *tokens*), that is often referred to as the *context*, and the model is trained to predict related words, by outputting a probability distribution over the considered vocabulary. However, there exist two main different language modeling approaches: *masked* and *causal*. Masked language modeling consists in training a LLM to predict *missing* tokens in the input. In fact, during training, some input tokens are randomly masked, and the model is then trained to recover them by only relying on the other tokens in the sequence. The result is that the model generally learns to understand the context and relationships between words. On the other hand, causal language modeling consists in training a LLM to predict the *next* token in the input sequence. The result of this process is that the model learns to generate coherent and realistic text. Once pre-training is complete, the model can be specialized (i.e., *fine-tuned*) for specific NLP tasks like machine translation, summarization, question-answering, sentiment analysis, etc. In general, this is accomplished by re-training the model on a smaller dataset, specific to the task at hand, using the pre-trained weights as a starting point. LLMs are generally based on the famous *transformer* architecture [240], that marked a pivotal point in NLP research. The architecture consists of two main components: the *encoder*, that is responsible for processing the input sequence, and the *decoder*, that generates the output. The fundamental building block of the transformer is the so-called *attention* function [12], that allows the model to intelligently weigh the importance of the different tokens when processing input sequences. There are several ways to implement the attention function, but the most used in practice is the *scaled dot-product* attention, that is typically faster and more space-efficient than the available alternatives, and is defined as

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.21}$$

where Q, K, and V are the so-called *queries*, *keys* and *values* matrices respectively, and d_k is the dimension of the individual query and key vectors. The function softmax : $\mathbb{R}^K \to (0,1)^K$ is defined as

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}, \quad x \in \mathbb{R}^K \tag{2.22}$$

In other words, softmax *normalizes* the input vector, such that the components of the resulting vector are real values included in $(0,1)$, and their sum is equal to 1. Given a sequence of input tokens, an *embedding* is computed for each of them, by projecting them onto the input space of the model. For each token embedding, applying the attention function basically consists in mapping a *query* and a set of *key-value* pairs, where each of these is a separate vector of *learnable parameters*, to an *output*. This terminology comes from the analogy between the attention function and an information retrieval (IR) process. Indeed, the query is just an alternative (learned) representation of the currently processed token. This representation is matched against the keys, one for each of the other tokens in the processed sequence, to compute *compatibility* scores, similarly to how an IR query is (approximately) matched against an *index*. The values associated to the keys are learned token representations as well. Finally, the output of the attention function corresponds to a weighted sum of these values, where the weight assigned to each value depends on the compatibility between the corresponding key and the query. Note that in Equation (2.21) the query, key, and value are defined as matrices. Indeed, in practice, the individual vectors, each one corresponding to an individual token in the input sequence, are typically batched for faster processing. The time complexity of the attention function is $O(n^2 \cdot d)$, where n is the sequence length and d is the dimension of the representation of a single token. While computing the attention values is theoretically expensive, such computation can be highly parallelized, resulting in an end-to-and training process that is typically significantly faster than other neural models for sequential data, like RNNs (and similar). Note that, in general, only the *encoder* part of the transformer architecture can benefit from the parallelization. Instead, the *decoder* inherently suffers from the same drawbacks of recurrent models, where the output of a computation step has to be fed to the next. In the research works that followed [240], many variations of the original transformer architecture have been proposed to improve its performance from different viewpoints, paving the way for the development of new kinds of LLMs. These variations can be broadly divided in two macro-categories: *encoder-only* and *decoder-only* architectures. In general, encoder-only models, like BERT [63] and T5 [187], exhibit remarkable performance in many diverse NLP tasks, both in terms of accuracy and efficiency, due to the aforementioned possibility to highly parallelize their computations. However, this type of models typically requires a greater amount of task-specific data to be properly fine-tuned on downstream tasks, and it is not particularly good at solving natural-language generation tasks. The latter is the main reason why decoder-only models, like the one described in [137], have been developed. Such models are indeed optimized for text generation, but they also exhibit remarkable *unsupervised multitask* learning and *few-shot* learning capabilities, that make fine-tuning them on downstream tasks generally easier and less demanding in terms of data. In fact, the architectures underlying the famous GPT models family [22, 185, 186] belongs to the decoder-only category.

Part II

HIGH-PERFORMANCE ANOMALY DETECTION

In this part, we address the AD problem, that represents a critical pain-point for NSPs. We present our proposals for anomalous VNF behavior detection using SOMs, aiming at supporting human operators in their activities, by offering a comprehensive view of the changes in the status of the monitored systems. We also provide a high-performance, open-source, implementation of SOMs, leveraging on GPU-acceleration.

3

SOM-BASED ANOMALOUS VNF BEHAVIOR DETECTION

3.1 INTRODUCTION

The NFV paradigm [39] has been progressively adopted by all the major NSPs in response to the increasingly demanding requirements they have to meet, in particular, in terms of performance, flexibility and resiliency. Indeed, traditional approaches that rely on the deployment of network functions on top of proprietary *specialized* physical appliances, typically sized for the peak-hour and very costly to maintain, are no more sustainable in the complex, fast-paced scenarios that can be found in modern telecommunication systems. Thanks to the amazing advances in the *cloud computing* space, having on-demand access to a diverse set of virtualized resources (computing, storage, networking, etc.), running on commodity hardware, has never been so easy and convenient. In the context of NFV, this kind of virtualization technologies is leveraged according to the *private* cloud computing model, where general-purpose computing, networking and storage resources owned by the operator can be dynamically and automatically managed and orchestrated, to fit the needs of time-varying workloads. This allows for cutting costs and energy consumption, as well as shortening development cycles and time-to-market [89]. For example, a virtualized network infrastructure can be easily adapted to adequately support new products of an organization or, if customers request new network functions, all it takes to handle such requests is to spin up new VMs that can be rapidly decommissioned when the functions are no longer needed. In this way, network functions can be completely decoupled from the underlying physical appliances they are deployed onto and can be effectively developed as distributed, elastic, resilient software applications. For NFV data centers, the choice of private cloud infrastructures, as opposed to the use of public cloud services, is also corroborated by latency-related concerns. Indeed, since such service-chains are highly delay-sensitive (e.g., LTE, 4G), it is unpractical to rely on public cloud infrastructures, that are usually shared among multiple tenants and non-necessarily deployed according to the network operator needs.

In order to guarantee scalability, robustness to failure, high availability, low latency, VNFs are typically designed as large-scale distributed systems [175], often partitioned and replicated among many geographically dislocated data centers. The larger the scale, the more operations teams have to deal with complex interactions among the various components, such that diagnosis and troubleshooting of possible issues become incredibly difficult tasks [82]. Also, the capacity of such systems is designed according to some technical and economical considerations, in order to support the *standard load* conditions under which the VNFs perform well, ensuring a number of diverse kinds of SLAs between network operators and their customers. However, when extraordinary events or cascade failures occur, the network is typically overloaded, and the allocated resources are not sufficient anymore to process all the incoming flows. Therefore, it is necessary to monitor the status of the data center through an efficient distributed infrastructure that continuously gathers system-level metrics from all the different levels of the architecture (e.g., physical hosts metrics, virtual machines metrics, application-level key performance indicators, event logs), in order to build a *proactive* system capable of detecting signals of system overload in advance. Such data usually drives the decisions of human operators, for instance, in terms of which actions must be taken to restore the expected conditions of the system after an outage has occurred, or how the available components should be reconfigured to prevent possible SLA violations in case of an unexpected increase in the workload.

One of the major problems of data center operators is AD, i.e., pinpointing unexpected and/or suspect behaviors of the system whenever it significantly deviates from the normal conditions. Indeed, recognizing characteristic patterns of resource consumption in early stages can be crucial to avoid resource exhaustion and to redirect critical traffic peaks so to minimize the risk of SLA violations (i.e., such that human experts can focus their efforts on the most critical activities), or at least to alert the staff to prepare the remediation/mitigation procedures in advance. Even though the amount of data usually produced by NFV infrastructures is huge, most of it is not explicitly labeled by specialized personnel, so that *unsupervised* ML algorithms (e.g., clustering or vector quantization techniques) are the easiest ones to use, especially for anomaly detection purposes. The objective of these algorithms is to group data with a similar trend in macro-categories and allow operators to keep tens or hundreds of VMs under control at the same time.

3.1.1 *Contributions*

We propose to use SOMs to perform a behavioral pattern analysis of VM metrics aiming at providing a comprehensive overview of the major behavioral patterns and detecting possible anomalies in a data center for NFV. The technique can be used to perform a joint analysis of *system-level* metrics available from the infrastructure monitoring system and *application-level* metrics available from the individual VNFs. It aims at supporting data center operations and specifically capacity and performance monitoring, by providing insightful information on the behavioral patterns, in terms of resource consumption and exhibited performance, of the analyzed VNFs. In our approach, the SOM-based behavioral analysis is leveraged to deliver a sophisticated alerting subsystem, whose output can be directly consumed by human operators or could be used as a trigger for automated remediation procedures.

3.1.2 *Chapter Organization*

This chapter is organized as follows. After discussing the related literature in Section 3.2, we present our approach in Section 3.3, along with the data processing workflow we designed for the massive data set available in the Vodafone infrastructure. In Section 3.4, we discuss some obtained experimental results that validate the approach and highlight its practical relevance. Section 3.5 concludes the chapter with our final remarks and ideas for future research in the area.

3.2 RELATED WORK

In this section, we briefly review some of the most related works that can be found in the research literature on using ML, and SOMs in particular, for classification and AD in cloud and NFV data centers.

AD can be framed as the problem of pinpointing unexpected and/or suspect behaviors of a system whenever it significantly deviates from the normal conditions. Similar problems can be found in other fields and applications such as, for instance: intrusion detection in cyber-security, machinery fault [205] and product quality issues detection [238] in industrial contexts, or fraud detection in finance [146]. It is important to stress that AD is, in general, an inherently imbalanced problem due to the scarcity of anomalous observations with respect to the ones related to the normal conditions of a system. In order to tackle this kind of challenges, a huge amount of solutions has been proposed that, depending on the scenario and the nature of the data to be processed, pose their foundations on well-established techniques coming, for instance, from the research fields of information theory and statistics.

In the recent years, ML techniques have been gaining more and more traction in the context of AD applications because of their proven effectiveness in many of the aforementioned scenarios. This is mainly due to the versatility of this kind of methods and the increasing availability of data from which they can learn from, in a continuous manner [24]. Most of the approaches to AD address the associated challenges by feeding ML models with counters like CPU utilization, memory contention and network-related metrics [77, 82, 156, 209, 245]. Others include also system-level and/or application-level event logs in the analysis to increase the amount of features and facilitate the extraction of relevant patterns [67, 251]. Embedding textual information has been in fact made easier by the advancements in NLP research [16]. Few existing works also consider the need of assisting human operators in conducting RCA to be a highly desirable feature of AD systems [85, 181].

One of the major roadblocks that can be encountered when applying ML for solving a task is the scarcity, or the complete absence, of labelled data, a very common scenario in many practical applications. Such issues can be overcome by employing so-called *unsupervised* learning techniques that, as the definition suggests, are designed to operate without a ground truth (i.e., annotated data). It is worth noticing that this characteristic of such class of learning algorithms has the side effect of increasing the amount of data that can be used for training an ML model. The principal application of unsupervised techniques is *clustering* that consists in the formation of groups of data samples that are similar, where similarity is defined according to the employed distance function.

In the context of AD, such approaches usually operate by building, starting from training data, a set of clusters of samples that are representative of the expected (normal) conditions of a system. After training, such model can be exploited to compare new patterns to known behaviors according to a predefined distance metric, in order to infer whether the observations are anomalous or not. In these applications, the aforementioned properties of retention of the original data topology and distribution give the SOM the capability of creating a suitable number of clusters for the most representative situations: this distribution of clusters allows for a more reliable characterization of anomalous patterns due to the higher granularity reserved to more common situations.

SOMs have achieved remarkable results at processing industrial data [87] in different fields. In [64], a SOM-based system for the visualization of complex process dynamics is proposed. In this application, topology conservation enables a smooth visualization of non-linear process transitions in a 2D map and favors the understanding of the influence of process parameters on process behavior. Similar approaches that exploit dimensional reduction and visualization on an easy-to-interpret 2D map are used also in [72] for process monitoring purposes and in [74] where the aspects of visualization of the evolution of process conditions are handled. In [29], SOMs are used for the grouping of electrical components based on a wide set of features that are efficiently mapped in a low dimensional space. The capability of SOMs of managing high-dimensional data and mapping them into a lower dimensional one have been exploited in medicine as well. In [34], sonographic signals are processed and grouped in order to characterize those associated to breast cancer diagnosis. Another SOM-based approach was used in [239] to allow the analysis of complex cytometry data that is hard from a point of view of human experts due to the huge amount of variables to be taken simultaneously into consideration.

For what concerns NFV applications, the existing literature reports that ML techniques have been effectively used to solve different problems. In particular, in [83] a set of ML techniques are tested for an AD application. However, only supervised methods are considered, and their performance is compared on data sets containing NFV features associated to different types of faults. Similarly, in [158], a *supervised* SOM-based method is proposed for fault detection. Here, a SOM is used to cluster *labelled* data, annotated by human experts to state which clusters correspond to faulty conditions, related to NFV performance indicators. In [164], SOM-based and other general clustering techniques are used for the same purpose in a small test-bed in the context of NFV. Likewise, in [129], the popular *K-means* algorithm is used to cluster cells traffic data in order group cells with similar through-time behavior and enable optimizations in the use of resources.

3.3 PROPOSED APPROACH

We propose the use of SOMs in order to perform a behavioral analysis of the VMs that implement VNFs within an NFV data center infrastructure. Our approach consists of the joint analysis of two classes of metrics that are usually collected and analyzed independently of one another: *system-level* metrics, reporting information related to the utilization of the underlying infrastructure, hereafter also referred to as *INFRA* metrics, which are usually available through the NFV infrastructure manager (e.g., the well-known VMWare vROps or others); and *application-level* metrics, i.e., KPIs of the individual virtualized services, collected through their own monitoring subsystems, which will be referred to as VNF metrics. Considering both types of metrics allows for gathering a comprehensive overview of the major behavioral patterns that characterize VMs and possibly identifying suspect (anomalous) behaviors.

The proposed technique relies on the capability of SOMs to preserve the topology in the projection from the input space to the SOM reference vector space. In other words, using SOMs similar input patterns are captured by same or nearby neurons (see Section 2.2.1 for details). A VM behavior can be monitored by considering the shift of its BMU,

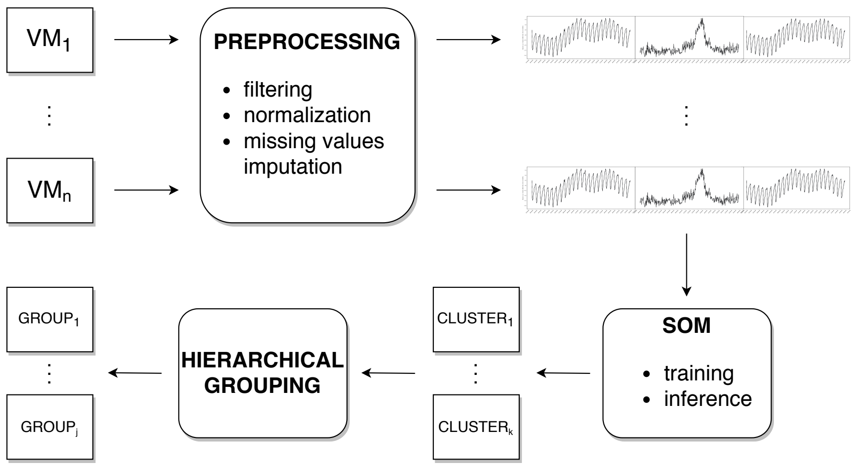


Figure 3.1: Overview of the SOM-based clustering workflow.

during the time horizon under analysis, so that any changes in a *suspect* BMU could be used to trigger an alarm.

3.3.1 *Workflow*

We realized a SOM-based clustering tool that is capable of detecting anomalies by clustering using a number of input metrics. In our experimentation, we have been applying this technique over individual monthly data available with a 5-minutes granularity (288 samples per day, per metric, per monitored VM or physical host), amounting to several GBs of data per month, for a specific region of the Vodafone network operator. Figure 3.1 summarizes the overall workflow that we applied to process the available input metrics. First, the raw data are preprocessed to address possible data-quality issues (e.g., missing values imputation and time-series detrendization) and to filter out the additional information that is not relevant for the analysis. The input samples to the SOM are constructed, for each VM, by dividing the time horizon under analysis according to a predefined period (i.e., 24 hours) and merging the contributions of the individual metrics into a single vector. Then, such data are fed to the SOM that, after a training phase, infers for each VM the neuron capturing the most similar behavior and, thus, clusters on the various behavioral patterns of all the various VMs under analysis.

The input data are filtered on the k specified metrics, and partitioned to have a sample (i.e., a time-series) for each metric, VM and period (usually a day) of the time horizon under analysis. Before being fed as input to the SOM training phase, samples are subject to a preprocessing phase, addressing possible issues such as (i) missing values and (ii) significant differences in the magnitude among the different metrics. On the one hand, to address (i), a data imputation strategy (i.e., a simple linear interpolation) is performed to mitigate the absence of data points within a sample and to retain as much data as possible for the analysis. However, in order to preserve the quality of the data set, the interpolation step has been designed not to be *aggressive*, such that a time-series can be discarded if it contains too much consecutive missing values. On the other hand, it is recommended to address (ii) when using SOM for multi-metric analysis since, due to the Euclidean distance being used as samples distance evaluation mechanism, metrics with significantly larger values (e.g., number of transmitted/received packets or bytes) tend to hide the contribution of other metrics which take on smaller values, for instance, being bounded by a predefined range that is much smaller (e.g., CPU utilization percentage). We have designed two possible strategies to tackle such problem. The first strategy, referred to as *normalized*, consists of computing the so-called *z-score*, i.e., scaling each time-series by subtracting its mean and dividing by its standard deviation. Using such a strategy hides any information regarding the magnitude of the original values and emphasizes differences in shapes. The second strategy, referred to as *non-normalized*, consists of scaling each time-series to the $[0, 1]$ range of values considering, for each metric, the historically observed minima and maxima values. Such a strategy retains information regarding the magnitude of the original values while keeping the data bounded in the same interval. However, this technique causes different metric patterns with very similar shape, but differing merely in their magnitude, to be grouped into different SOM neurons at a certain distance from each other (in the SOM grid topology). Depending on the chosen strategy, we obtain either an analysis focused on the shapes of the behavioral patterns, or we can also distinguish among the absolute values of the average levels of the metrics. In general, in the latter case one should expect more clusters to be outputted with respect to the former case, due to the possibility that the system could have experienced very diverse levels of load during its operation. Hence, one should take this possibility into account and increase the size of the SOM grid when performing a non-normalized analysis in order to avoid neurons *over-population* (i.e., too many patterns crowding within the same BMU), despite them being significantly distant from each other.

Each input sample to the SOM is constructed by concatenating k vectors (one for each of the k metrics under analysis), for each VM and period. Notice that, since INFRA metrics have been provided with a 5-minutes collection granularity, if a period of a day is considered, we typically have for each day 288 data points of each metric and for each VM. After the training phase, the SOM is used to infer the BMU for each input sample, i.e., the neuron that exhibits the least quantization error when compared with the considered input sample. Multiple VMs are expected to be associated to the same BMU and, thus, a number of different VM clusters can be derived from such process.

Such an output can be used by a data center operator to visually inspect the behaviors assumed by the different VMs during the time horizon under analysis, in order to spot possible suspect/anomalous ones. Furthermore, since the individual input samples are related to the behavior of a specific VM at specific point in time, it is also possible to visualize the evolution of the VMs throughout the time horizon, to possibly detect interesting patterns in their behavioral changes. In this way, an operator is able to focus the analysis on a restrained set of VMs (or their hosts) and to possibly trigger further, more specific, analysis that could be too time-consuming, or even unfeasible, to conduct on the whole infrastructure.

On top of the clustering mechanism described above, we have devised an approach capable of detecting possible suspect behaviors without the need for a human operator to daily inspect the status of the SOM (these aspects are described in details in Section 3.3.4). Such additional feature consists of an alerting system that is triggered whenever an input sample is firstly associated to a group of similar neurons but in the following days a sudden group change takes place. Because of the considerable distance from the BMU (i.e., the *closest* neuron) of the neurons in the two different groups, such samples are likely to depict an uncommon behavior and, thus, an alert is raised to the operator. Besides the aforementioned support that such a tool can give to data center operators in their manual operations, this feature in particular enables the possibility to deploy a fully automated AD system.

3.3.2 *SOM Implementation*

To implement our AD tool, we leveraged on an efficient open-source SOM implementation, namely `Somoclu`[253], which has been designed around the batch parallel SOM variant (see Section 2.2.1) to employ multi-core acceleration, as well as GPU hardware acceleration, to perform massively parallel computations [253]. Such accelerations have been proved to be necessary in order to reach a satisfactory performance when tackling the massive data set provided by Vodafone. Such performance requirements have also led us to realize a new SOM implementation that outperforms `Somoclu` in single-node scenarios [148] (see Chapter 4).

3.3.3 *Hierarchical Grouping*

An interesting aspect that came to our attention during the development of the aforementioned SOM-based approach is that, whenever using relatively big SOM networks, the training phase ends up with many close-by SOM neurons catching behaviors that were very similar to each other. This is in line with the topology-preservation property of the SOMs, i.e., close-by input vectors in the input space are mapped to close-by neurons in the SOM grid. This phenomenon can be controlled to some extent by acting on the neighborhood radius. However, from the viewpoint of data center operators, a set of close-by neurons with relatively similar weight vectors needs to be considered as a single behavioral cluster/group. For this reason, after the SOM processing stage, we added a step consisting of a top-down clustering strategy, based on recursively separating weight-vector's sets whose diameter is higher than a given threshold. The principal aim of this technique is to offer the possibility of collapsing similar SOM neurons, according to the distances among their representative vectors, in order to decrease the possibility to raise an alarm when it is not needed (e.g., consider very frequent movements of a VM between two similar neurons over time) and to facilitate the human operators in interpreting the results and spotting anomalous behaviors. Indeed, as shown in Section 3.4.4, this led to the overall technique outputting a reduced and more comprehensible number of behavioral clusters.

Specifically, the aforementioned technique, known as *hierarchical clustering*, can be described as follows. Let ε be a fixed threshold which provides a bound for the maximum diameter of a group. The algorithm consists of the following steps:

1. **Initialization:** The set of the groups to process is initialized with a single group $\mathcal{G}_0$ containing all the neurons $\mathcal{G}_0 = \{n_1, \cdots, n_{H\times K}\}$, and the set of the final groups is initialized to be an empty set.

2. **Distance Measure:** A group $\mathcal{G}$ is removed from the set of groups to process and its diameter D is computed by finding the two farthest away neurons:

$$(n_S, n_N) \in \operatorname*{arg\,max}_{(n,m)\in\mathcal{G}} d(w(n), w(m)), \quad D = d(w(n_S), w(n_N))$$

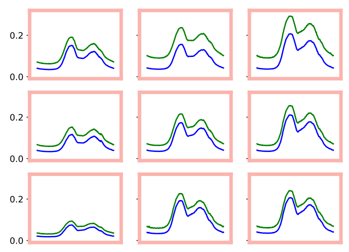
   where $w(n)$ is the weight of neuron n. In the case that $\mathcal{G}$ contains just one neuron, its diameter D is defined as zero.

3. **Splitting:** If $D \leq \varepsilon$ (i.e., the diameter is within the specified threshold), then $\mathcal{G}$ is moved to the set of final groups. Otherwise, the group is split into two smaller (non-empty by construction) groups $\mathcal{G}_1$ and $\mathcal{G}_2$ defined as:

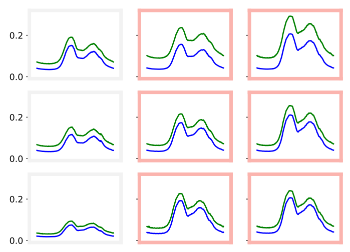
$$\begin{aligned}\mathcal{G}_1 &:= \{n \in \mathcal{G} \, : \, d(w(n), w(n_S)) \leq d(w(n), w(n_N))\} \\ \mathcal{G}_2 &:= \{n \in \mathcal{G} \, : \, d(w(n), w(n_S)) > d(w(n), w(n_N))\}\end{aligned}$$

   that are added to the set of groups to process.

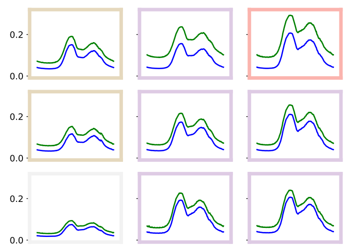
4. **Loop:** Steps 2,3 are repeatedly applied to all the elements in the set of groups to process, until it becomes empty, and the set of final groups contains only groups with a diameter lower than or equal to ε.



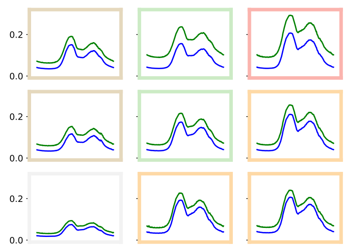
(a)



(b)



(c)



(d)

Figure 3.2: Example of grouping steps: (a) initialization; (b) first split; (c) second split; (d) final split. Neurons with same border color belong to the same group.

Figure 3.2 reports a graphical representation of how the algorithm works on a real example. From left to right, we can see all the four steps of the algorithm that bring to the final result in which each group contains only neurons with a pair-wise distance smaller than the provided threshold.

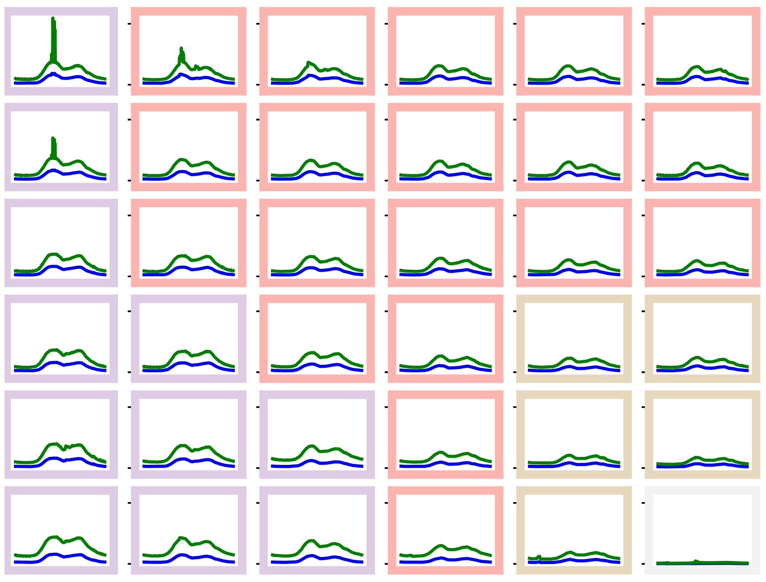
As explained above, the kernel of the hierarchical clustering technique is the measure of the diameter of a set. This implies that the definition of the distance impacts on the final result.

Definition 3.3.1. For each $p \in \mathbb{N}$, the function $d_p : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_+$ defined as

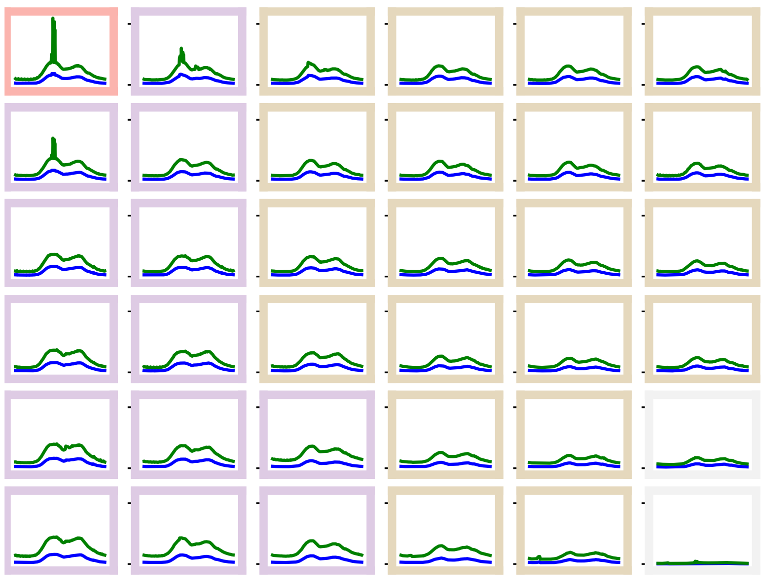
$$d_p(v, w) := \left(\sum_i |v_i - w_i|^p \right)^{\frac{1}{p}}$$

is the Minkowski distance of order p.

Note that, according to Definition 3.3.1, the Minkowski distance with $p = 2$ is the Euclidean distance, and that d_∞ degenerates into the Chebychev distance (maximum among the coordinates). Figure 3.3 shows that using $p = 4$, or in general a value higher than 2, allows for increasing the distance between neurons exhibiting a spike (i.e., neurons that are almost flat, except for an isolated huge value), so that we are able to isolate in a dedicated group such spiky neurons.



(a)



(b)

Figure 3.3: Examples of grouping using different p values. Grouping with $p = 2$ (a) makes no distinction between spiky and smooth neurons, whereas grouping with $p = 4$ (b) clusters the spiky neuron on the top-left corner in a dedicated group.

3.3.4 *Alerting*

A grouped SOM grid combined with a *calendar* representation of the VM behaviors can be used by an operator to spot possible anomalies. A calendar representation is a table containing for each couple *(VM, day)* a reference to the corresponding group. In addition, we designed a set of alerting systems based on heuristic methods, that can be used to simplify the inspection of such behaviors. We propose two main categories of alerting systems: the *calendar-view* alerting system, consisting of techniques that give a global view of the alerts over the entire period of interest, and the *dashboard-like* alerting system, consisting of techniques that give a detailed view of the behaviors that raise the alerts. In what follows, $V = \{v_1, v_2, \ldots, v_i, \ldots\}$ is the set of virtual machines under analysis, $D = \{d_1, d_2, \ldots, d_i, \ldots\}$ is the set of days that compose the time period under analysis and $\mathrm{grp} : V \times D \to G$ is the function that associates to each couple *(VM, day)* the corresponding group.

3.3.4.1 *Calendar-View Alerting System*

This category contains those alerts that generate a calendar table in which each couple *(VM, day)* is associated with a value between 0 and 1, providing a level of alerting. In what follows, we denote with p the *period* and with m the *memory*, both expressed in days (i.e., $p = 7$ days, $m = 2$ weeks).

Definition 3.3.2 (Strong)**.** Given p, m, the Alert takes one VM v and one day d and returns a boolean value raising an alert if the v is classified into a different group in at least one day among the ones at most m periods apart:

$$\mathrm{Alert}_s(v,d) \;:\; ''\exists j \in \{\pm 1, \cdots, \pm m\}, \quad \mathrm{grp}(v,d) \neq \mathrm{grp}(v, d - jp)'' \tag{SAS}$$

This alerting system is the most peaky (i.e., often producing false-positives) and, thus, performs the best when used in contexts where a few changes occur.

Definition 3.3.3 (Weak)**.** Given p, m, the Alert takes one VM v and one day d and returns a boolean value raising an alert if the v is classified into a different neuron in all the days among the ones at most m periods apart:

$$\mathrm{Alert}_w(v,d) \;:\; ``\forall j \in \{\pm 1, \cdots, \pm m\}, \quad \mathrm{grp}(v,d) \neq \mathrm{grp}(v, d - jp)`` \tag{WAS}$$

This alerting system is more loose than the previous (sometimes producing false-negatives) and, thus, performs the best in chaotic contexts, where many random changes occur.

Definition 3.3.4 (Fuzzy)**.** Given p, the Alert takes one VM v and one day d and returns a real number, between 0 and 1, defined as follows:

$$\text{Alert}_z(v,d) := \quad \#\{j \in \mathbb{Z} \,:\, \text{grp}(v,d) \neq \text{grp}(v,d-jp)\}/\#D(v,d) \tag{ZAS}$$

where $D(v,d) = \{j \in \mathbb{Z} \,:\, \exists \text{grp}(v,d-jp)\}$ is the set of all the comparable days.

This alerting system, producing real values, can be used in a wide range of situations and could be useful to understand if a change in the behavior of a VM is common or infrequent.

3.3.4.2 *Dashboard-Like Alerting System*

The aim of a dashboard-like alerting system is to provide a detailed view of the behaviors which raise the alert, providing also further information on the geometrical distance between the actual and the expected behavior in terms of weight of the SOM or also a count of the frequency of VMs/days which are clustered into *rare* groups.

Definition 3.3.5 (Expected Behavior)**.** Let v be a VM for which an alert is raised at day d, i.e., $\text{Alert}(v,d) = 1$. Let $\tilde{d}$ be the nearest day, corresponding to the same weekday, for which the most common group is taken from v and for which $\text{grp}(v,d) \neq \text{grp}(\tilde{d})$ holds. Then, we define

- `GRP`$:= \text{grp}(v,d)$
- `NEU`$:= \text{neu}(v,d)$
- `E_GRP`$:= \text{grp}(v,\tilde{d})$
- `E_NEU`$:= \text{neu}(v,\tilde{d})$
- `DIST`$:= \|w(\texttt{E\_NEU}) - w(\texttt{NEU})\|_2$

where the function $\text{neu} : V \times D \to \mathcal{L}$ returns the coordinates of the BMU associated to the behavior of a VM v during a day d and the function w, defined in Section 2.2.1, returns the weight of a neuron.

Such alerting system depends on the output of the calendar-like alerting system. Usually, we apply this method to the *weak* alerting system table (see Definition 3.3.3) in order to avoid false-positives alerts.

Definition 3.3.6 (Suspicious-Day)**.** Given a parameter K, let $\text{occ}_d : G \to \mathbb{N}$ be the function that counts the occurrences of a group in the days.

$$\text{occ}_d(g) := \#\{d \in D \,:\, \exists v \in V, \text{grp}(v,d) = g\}. \tag{3.1}$$

If a group g is such that $\text{occ}_d(g) \leq K$, then those VMs whose take the group g are stored in a table whose columns are `DAY, VM, NEU, GRP, OCC_DAY`, where `OCC_DAY`$= \text{occ}_d(g)$

Such alerting system helps in catching days in which an infrequent group appears.

Definition 3.3.7 (Suspicious-VM)**.** Given a parameter K, let $\text{occ}_v : G \to \mathbb{N}$ be the function that counts the occurrences of a group in the VMs.

$$\text{occ}_v(g) := \#\{v \in V \,:\, \exists d \in D, \text{grp}(v,d) = g\}. \tag{3.2}$$

If a group g is such that $\text{occ}_v(g) \leq K$, then those VMs whose take the group g are stored in a table whose columns are `VM, DAY, NEU, GRP, OCC_VM`, where `OCC_VM`$= \text{occ}_v(g)$

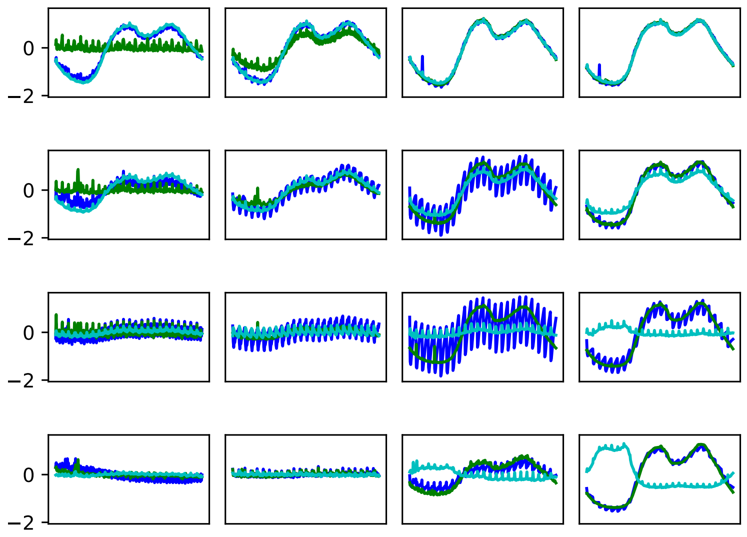
Such alerting system helps to catch VMs that are clustered into an infrequent group.

3.4 EXPERIMENTS

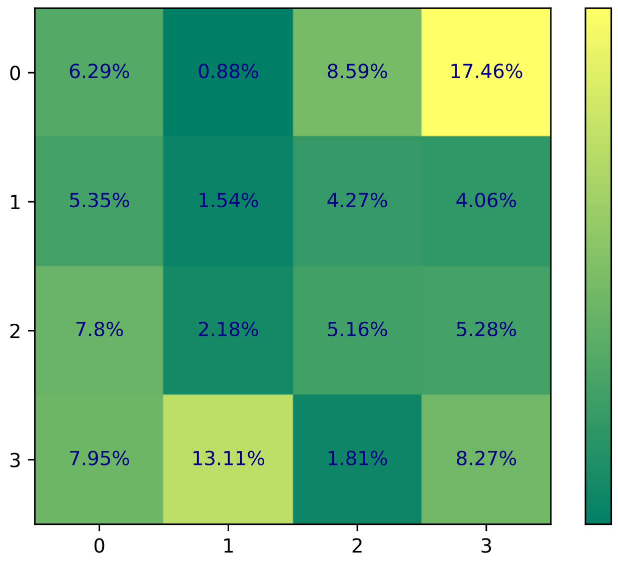
In this section, we provide an overview of the results that can be obtained using the approach proposed in Section 3.3. For the analysis, we relied on the experience of domain experts and focused our attention over a limited set of metrics that are considered the most relevant in this context: the ones related to the computational, networking and storage activity of VMs and VNFs of interest. Specifically, in the following, we highlight results obtained analyzing the following vROps metrics: `cpu|capacity_contentionPct`, `cpu|usage_average`, `net|usage_average`.

3.4.1 *Multi-metric Analysis*

The plots reported in Figure 3.4 are examples of the results that can be obtained through the multi-metric SOM-based analysis presented in Section 3.3, applied over a month worth of INFRA metrics, using the normalized strategy. The trained SOM network is visually represented in terms of the weights of its neurons. Indeed, each subplot reports the VMs daily behavior that the specific neuron specialized into. In order to simplify the representation, the weight vectors, jointly computed over the three metrics `cpu|usage_average`, `net|usage_average` and `cpu|capacity_contentionPct`, are overlapped but in different colors. For instance, one of the most recurrent patterns, occurring in 17.46% of the observations and depicted in Figure 3.5a, is the one identified by the top-right neuron. Because of the standard data normalization performed during the preprocessing phase to discard the magnitude information in favor of enhancing the behavioral information of the input samples, the values on the Y-axis can be negative. This means that VMs have been clustered based on the joint shape of their daily resource consumption patterns, not their absolute values. Notice that in this example we can observe a quite suspect output, since the `cpu|capacity_contentionPct` figure follows closely the daily traffic

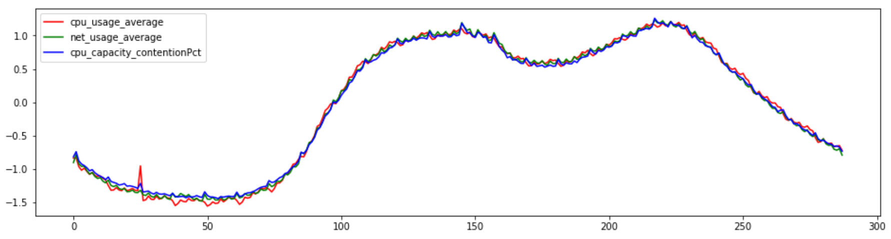


(a)

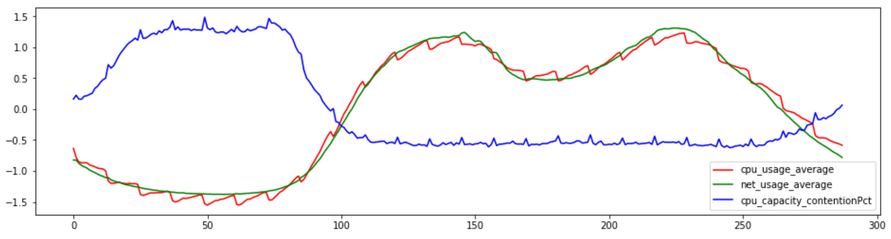


(b)

Figure 3.4: (a) INFRA resource consumption clusters identified with the multi-metric analysis. The dark blue, green and light blue curves in each plot correspond to the `cpu|usage_average`, `net|usage_-average` and `cpu|capacity_contentionPct` vROps metrics, respectively. (b) SOM grid showing the percentage of training samples captured by each neuron.



(a)



(b)

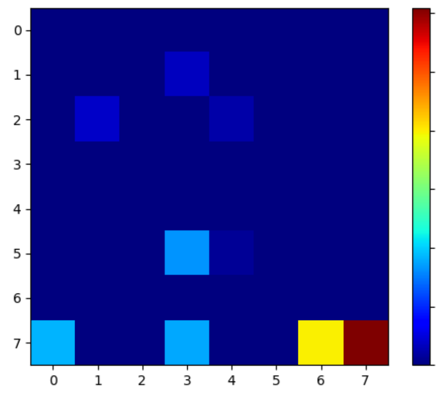
Figure 3.5: (a) The most recurrent VM cluster of Figure 3.4a and (b) a singular VM pattern captured by the bottom-right neuron of Figure 3.4a.

pattern on the involved VMs. In a normal condition of a healthy system, i.e., when VMs are provided with appropriate computational resources, we would have expected this metric to stay close to zero, or at least experience a slight increase only during the peak hours. A significantly different pattern is the one reported in Figure 3.5b, corresponding to the bottom-right neuron in Figure 3.4a. Such behavior represents the 8.27% of the observed daily patterns in the time period under analysis. As evident from the picture, there is a higher CPU contention during night, when the VM has lower traffic, than during the day.

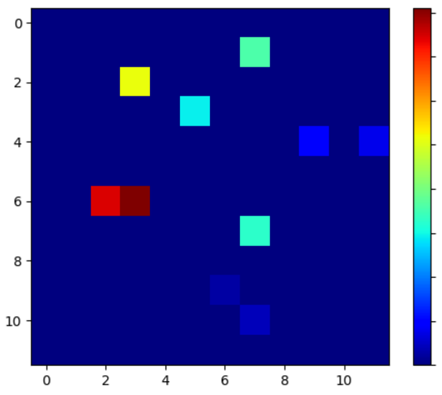
An additional remark regarding the possible presence of anomalies can be done considering the fact that the VMs included in the analysis are guaranteed to have the same role in the corresponding VNFs, i.e., they manage traffic in load-sharing mode. While it was expected to obtain an identical output for all of them, the SOM-based analysis has pointed out that a subset of such VMs exhibits daily patterns very different to the expected ones instead. This could be interpreted by human operators as a warning, that requires further monitoring and analysis of the involved components of the infrastructure. In addition, it is worth noticing that asynchronous changes among the metrics included in such analysis could be indications of anomalous behavior of the NFV environment, and not necessarily of the VNF itself.

3.4.2 *Hyperparameters Grid Search*

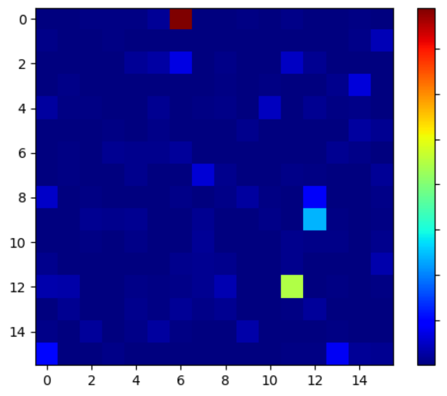
As mentioned in Section 3.3.1, different hyperparameters lead to very different clusters after training. An extensive grid search has been conducted over the search space summarized in Table 3.1. A total of



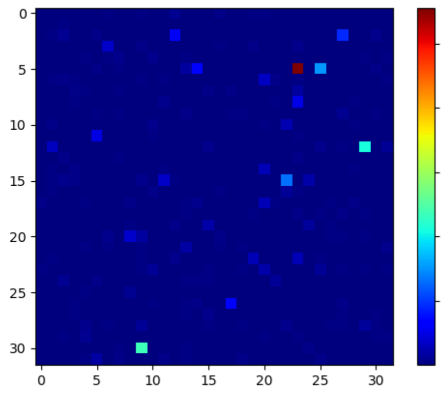
(a)



(b)

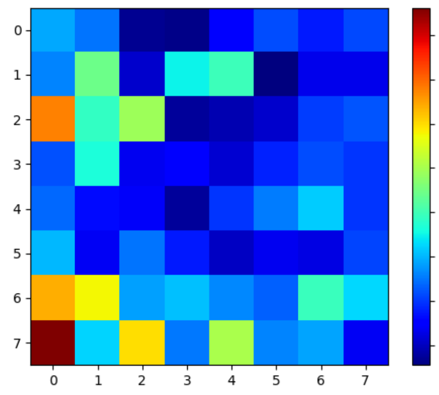


(c)

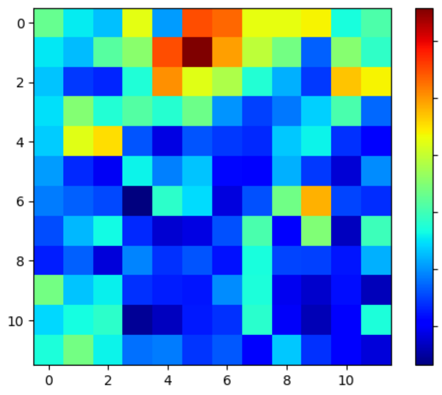


(d)

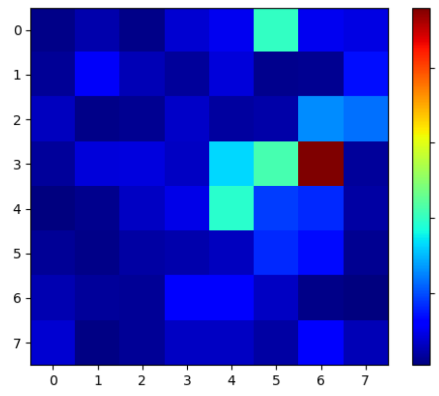
Figure 3.6: SOMs with low σ values: (a) 8×8, σ: 0.1, lr: 0.2; (b) 12×12, σ: 0.1, lr: 0.2; (c) 16×16, σ: 0.1, lr: 0.9; (d) 32×32, σ: 0.1, lr: 0.8. For confidentiality reasons, the scale has been omitted.



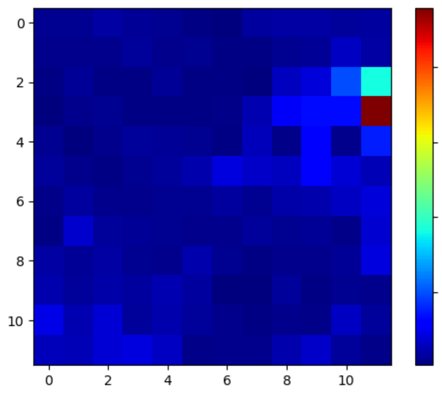
(a)



(b)



(c)



(d)

Figure 3.7: SOMs with high σ values: (a) 8×8, σ: 0.6, lr: 0.2; (b) 12×12, σ: 0.6, lr: 0.2; (c) 8×8, σ: 0.6, lr: 0.9; (d) 12×12, σ: 0.6, lr: 0.9. For confidentiality reasons, the scale has been omitted.

Table 3.1: The hyperparameters values used for the grid search.

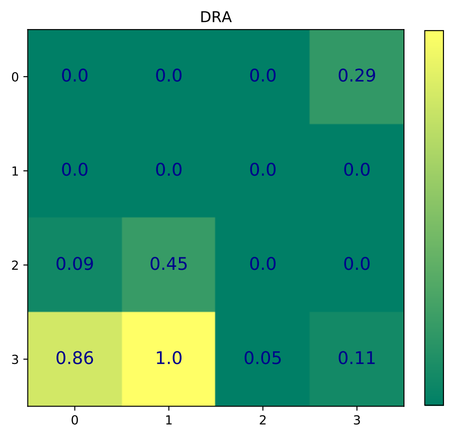
Hyperparameter	**Space**
dimensions	8×8, 12×12, 16×16, 24×24, 32×32, 48×48
learning rate	$0.1, 0.2, \ldots, 0.9, 1.0$
radius (σ)	$0.1, 0.2, \ldots, 0.9, 1.0$
epochs	$5, 10, 20$

1600 different configurations has been tested monitoring quantization error and readability of results. Figure 3.6 shows the effect of using a low σ value (0.1) in different map sizes. Using a low σ with a low learning rate gives the worst results with very few BMUs that capture more than 95% of data, resulting in higher quantization errors.

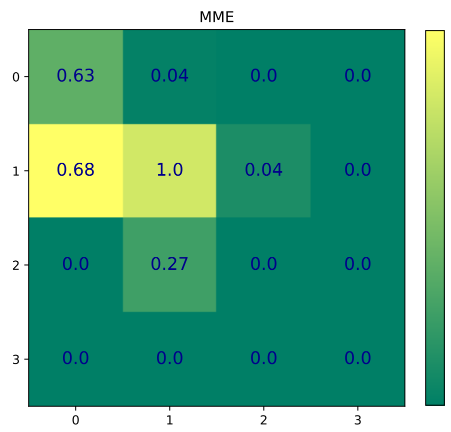
SOM maps greater than 12×12 require very high σ (> 0.8) and very low learning rate (< 0.3) in order to have low quantization errors, but in these cases the results tend to become unreadable due to the fact that too many neurons specialize on similar patterns. In Figure 3.7, the SOM maps reported in Figures 3.7a and 3.7b are trained using high σ and low learning rate, while the ones reported in Figures 3.7c and 3.7d are trained using high σ and high learning rate. Therefore, for our analysis the best combination of hyperparameters are high values of σ (> 0.6) and low values of learning rate (< 0.6) with results that are better both in terms of quantization error and readability.

3.4.3 *Per-VNF Analysis*

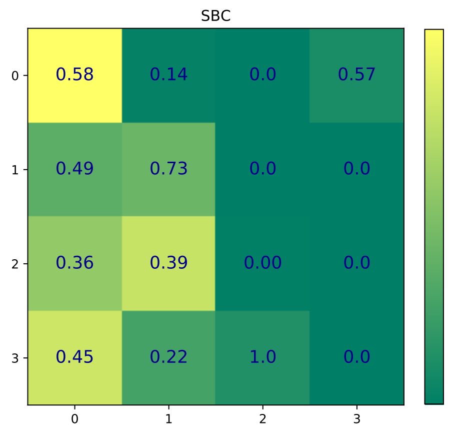
Another interesting characterization we could perform applying the SOM-based analysis, is a study of how different VNFs behave in terms of their daily resource consumption patterns. In this case, we produced hitmaps highlighting how many daily patterns of VMs of each given VNF map onto each SOM neuron. The result can be visualized as in Figure 3.8. For example, by comparing such plots with the corresponding map reporting the captured behaviors (like the one in Figure 3.4a, even though, in this case, the two figures are derived from different subsets of the available data), one can discover that both the SBC and the TAS VNFs have mostly the usual *"nightly/daily"* pattern, characterized by a low workload over nightly hours and a high workload over daily hours, with peaks around noon and 6pm. On the other hand, the DRA VNF exhibits the classical nightly/daily pattern for the `cpu|capacity_contentionPct` metric, and periodic peaks every 30 minutes for the other two metrics. Moreover, a consistent number of VTAP VMs are characterized by hourly periodic peaks.



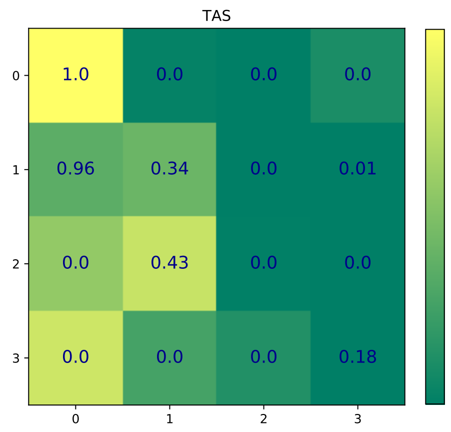
(a)



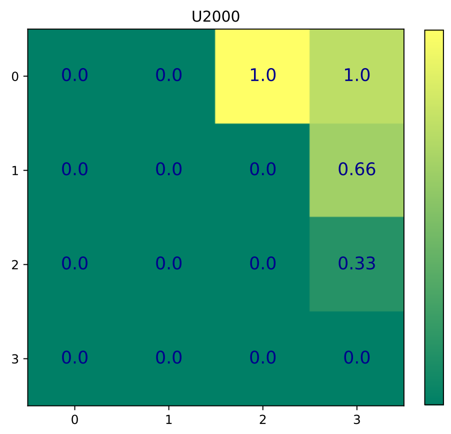
(b)



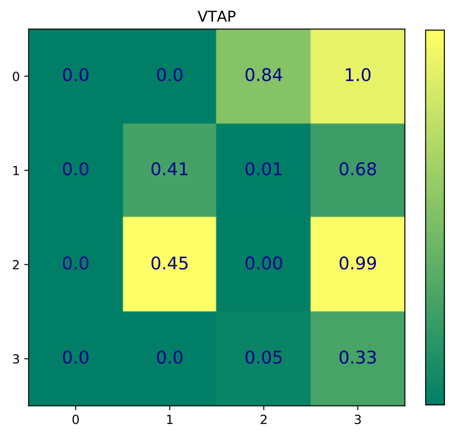
(c)



(d)



(e)



(f)

Figure 3.8: SOM clusters and corresponding per-VNF hitmaps. For confidentiality reasons, the total number of hits in the hitmap cells has been rescaled to 1.

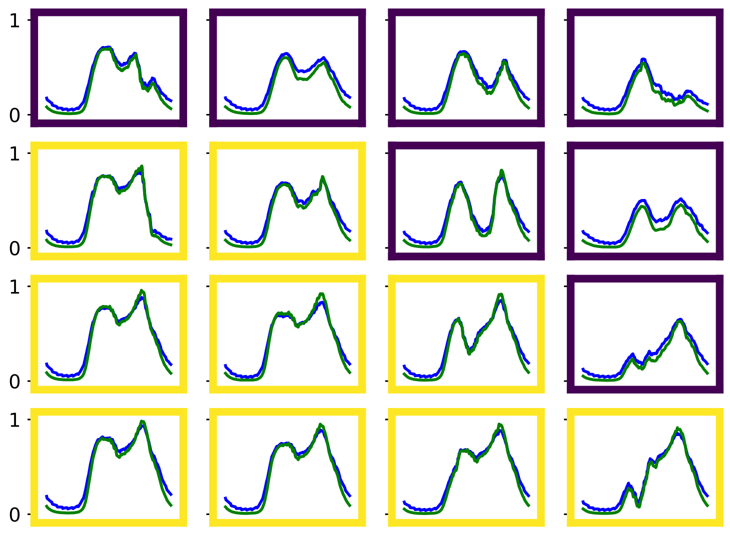
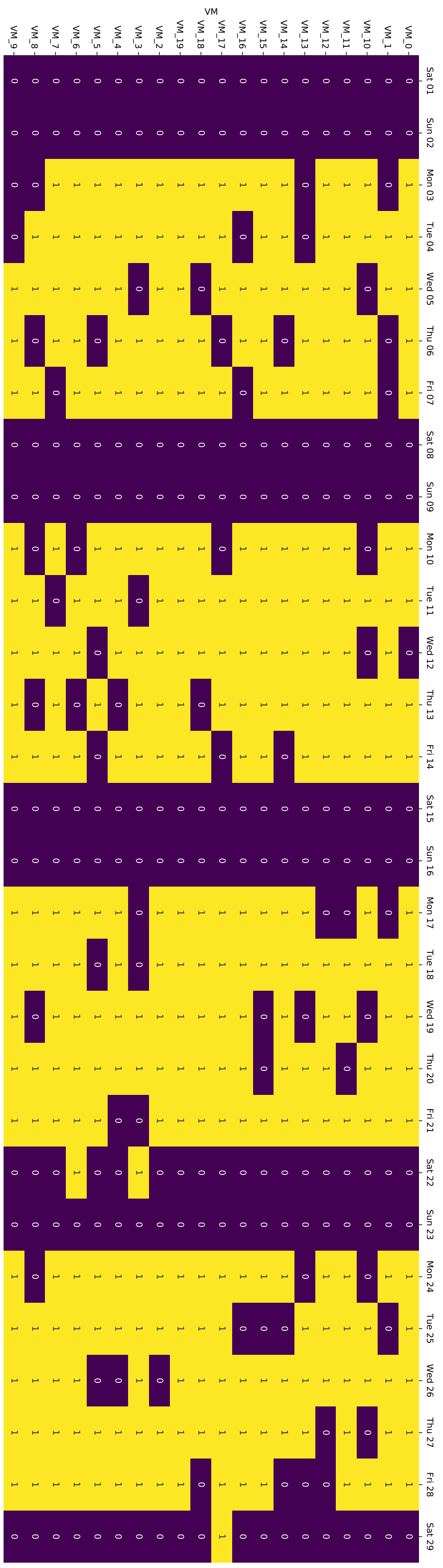


Figure 3.9: Distance-based grouping applied to a square SOM grid, 4 neurons per side.

3.4.4 *Hierarchical Grouping*

In this section, we report two examples of grouping/clustering technique described in Section 3.3.3, starting from another month of data, with respect to the experiments shown above. In the first example, we obtained the trained SOM whose weights are presented in Figure 3.9. By applying the distance-based grouping, with a group-distance threshold of 0.007, we obtained the clustering shown in the same figure, where neurons belonging to the same group have the same border color. Moreover, to facilitate a visual inspection of the behavior of each VM during the month, we produced a calendar view of the VMs, as shown in Figure 3.10, by associating to each couple *(VM, day)* the group of the BMU in the SOM grid. For instance, in this case we can notice a very common behavior: the majority of group changes take place during the weekend. The second example shows how the two outputs could be jointly used by a system operator to visually detect anomalies in the behavior of VMs. The grouped SOM grid in Figure 3.11 highlights three main (i.e, more frequent) groups. In particular, the yellow group contains all the neurons within an almost flat VM metric. By inspecting the calendar in Figure 3.12, it is evident that these behaviors are associated to those VMs that have an anomalous *constant* course, without any variations during the weekend.



	VM_9	VM_8	VM_7	VM_6	VM_5	VM_4	VM_3	VM_2	VM_19	VM_18	VM_17	VM_16	VM_15	VM_14	VM_13	VM_12	VM_11	VM_10	VM_1	VM_0
Sat 01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sun 02	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mon 03	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1
Tue 04	0	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1	1
Wed 05	1	1	1	1	1	1	0	1	1	0	1	1	1	1	1	1	1	0	1	1
Thu 06	1	0	1	1	0	1	1	1	1	1	0	1	1	0	1	1	1	1	0	1
Fri 07	1	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1
Sat 08	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sun 09	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mon 10	1	0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1
Tue 11	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
Wed 12	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
Thu 13	1	0	1	0	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1
Fri 14	1	1	1	1	0	1	1	1	1	1	0	1	1	0	1	1	1	1	1	1
Sat 15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sun 16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mon 17	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0	0	1	0	1
Tue 18	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
Wed 19	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	1	1
Thu 20	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	1
Fri 21	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
Sat 22	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Sun 23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mon 24	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1
Tue 25	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	1
Wed 26	1	1	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
Thu 27	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1
Fri 28	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0	0	1	1	1	1
Sat 29	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

Figure 3.10: VMs exhibiting common behaviors.

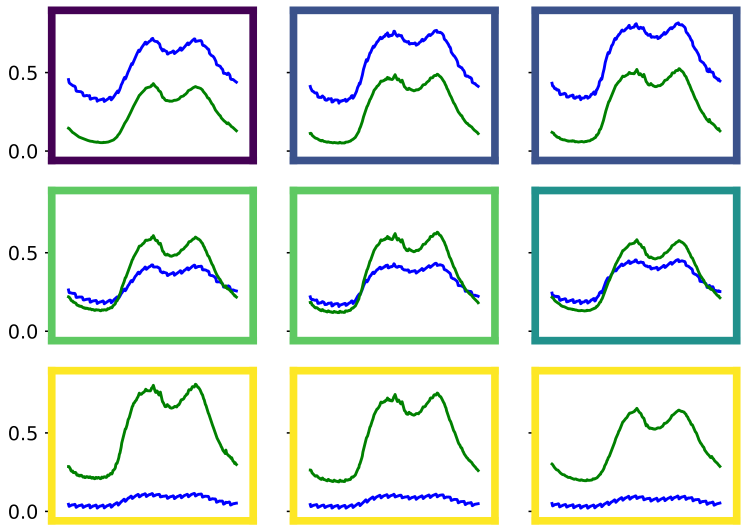


Figure 3.11: Distance-based grouping applied to a square SOM grid, 3 neurons per side.

3.4.5 *Alerting*

In this section, we provide some examples of output of the two main kind of alerting systems. All the alerting systems are applied to the behaviors captured by the SOM grid in Figure 3.13. Notice the presence of two groups with a high working-level (red, orange); two groups with a low working-level (brown, green); one group with a unique almost flat neuron (gray). Figure 3.14 reports the behavioral evolution of the VMs whose data have been used to conduct this experiment. The reference time period is April 2020 and, in particular, on April 13th (Easter Monday) many VMs change behavior, passing from their usual high working-level group to a low working-level group.

3.4.5.1 *Calendar-like Alerting Systems*

Figure 3.15a shows the *Strong Alerting System* (SAS), with $m = 3$ and $p = 7$, where dark green cells stand for a raised alert. Since the method compares the groups in same week-days and raises an alert if at least one change occurs, we can see that many alerts have been raised (some of them are obviously false positives). In contrast, Figure 3.15b shows the alerts that have been raised with the *Weak Alerting System* (WAS), with $p = 7$ and $m = 3$. Since the method raises an alert if a group appears only once in same week-days, we can see that only a few behaviors raise an alert. The output from the *Fuzzy Alerting System* (ZAS), with $p = 7$, is shown in Figure 3.15c. The higher the value (i.e., the

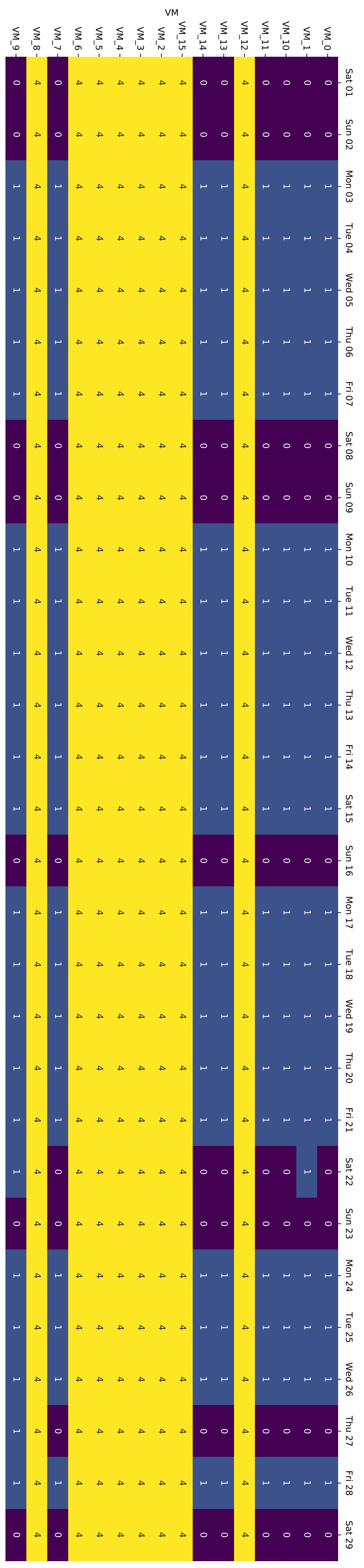


Figure 3.12: VMs exhibiting anomalous behaviors.

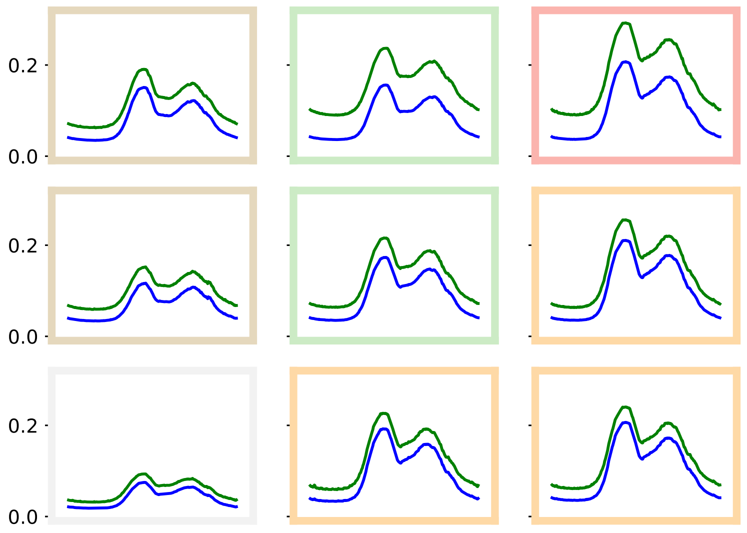


Figure 3.13: A SOM grid with neurons grouped in 5 behaviors.

darker the color), the higher the probability of an alert being significant. As expected, many of the *false-positives* reported by the *SAS*, and not by the *WAS* one, are associated to a low value.

3.5 CONCLUSIONS

We focused on the problem of analysis and classification of the behavioral patterns of VM metrics in a NFV data center. We described the technique we realized, based on SOMs, that is being used across the data centers of the Vodafone network operator. Our results highlight the capability of the proposed technique to identify interesting points in space and time (i.e., precise VMs and hosts within the infrastructure, and precise days within the analyzed time range) with potentially anomalous behaviors, thus deserving further attention and investigations by data center operators. Also, we detailed a clustering technique applied over a trained SOM codebook in order to mitigate the problem of neuron over-representation, and an alerting system built atop the SOM-based clustering, improving the AD pipeline effectively reducing the number of false positives.

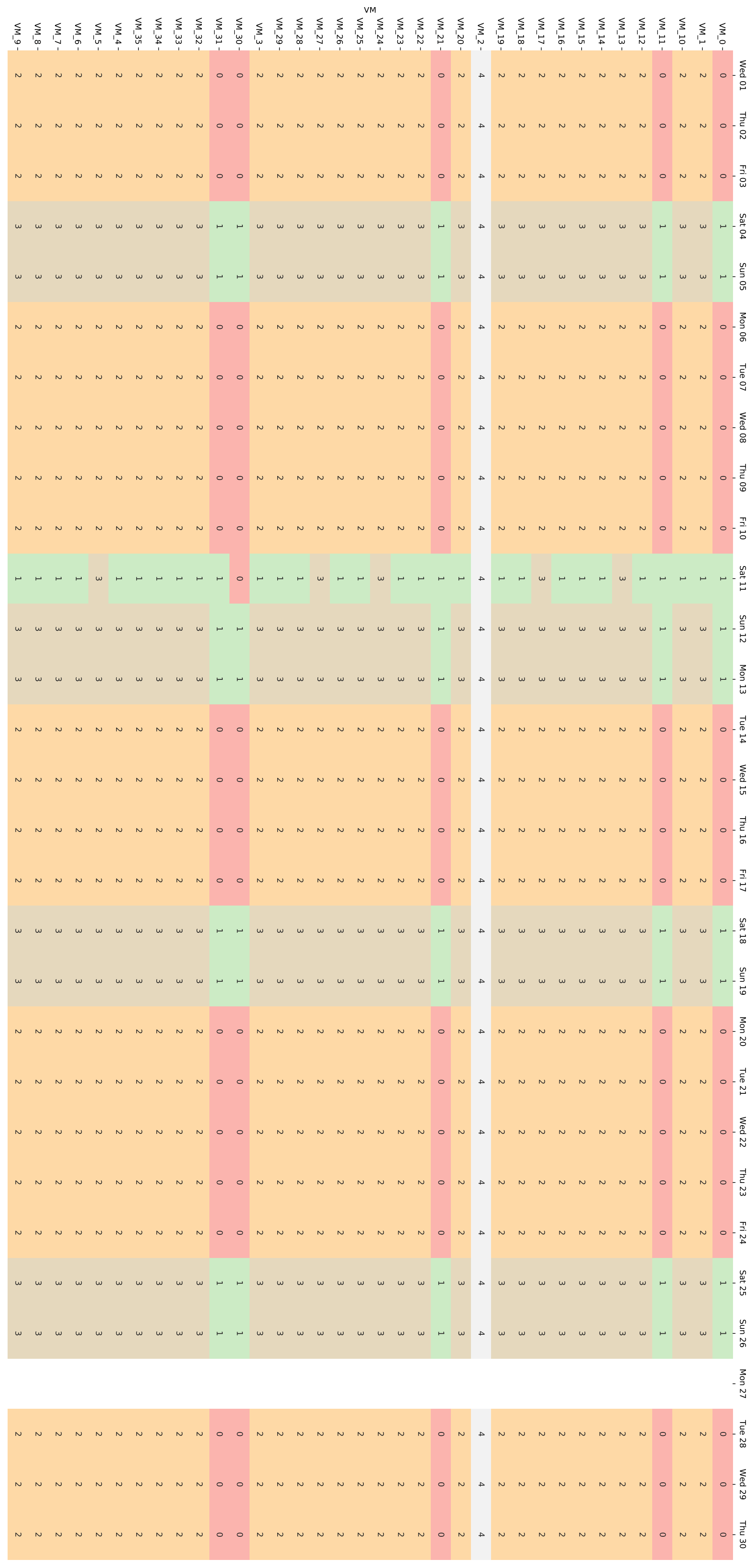
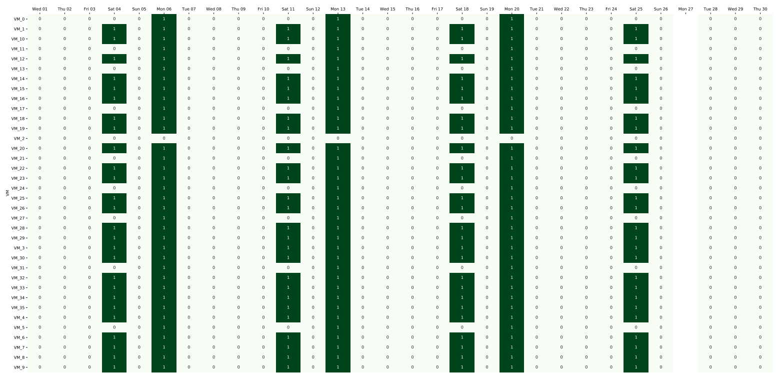
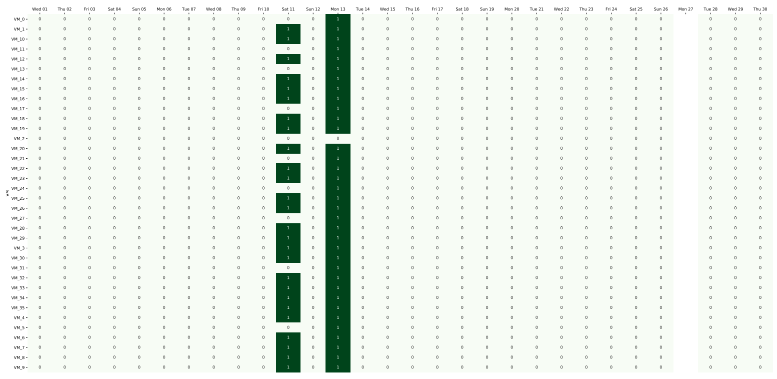


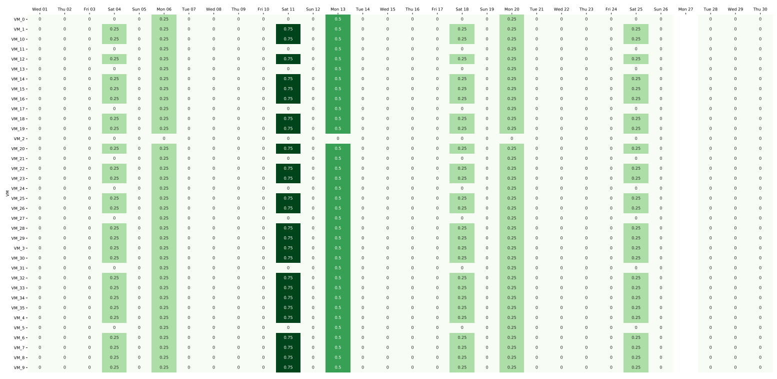
Figure 3.14: Calendar view of the set of VMs involved in the analysis.



(a)



(b)



(c)

Figure 3.15: (a) Strong Alerting System. (b) Weak Alerting System. (c) Fuzzy Alert System.

4

XPYSOM: HIGH-PERFORMANCE SOMS

4.1 INTRODUCTION

During recent years, we have been witnessing an exponential growth in the amount of data that is made available for decision-making, and a corresponding increase of the complexity of the computations carried out on such data, in order to squeeze the maximum "value" out of it. This required the development of ad-hoc software tools realizing big-data processing pipelines that make use of advanced techniques going beyond traditional statistics, relying more and more on complex computations employing ML. Indeed, ML has become a fundamental component of the software development life-cycle, allowing for building software that does not require to be explicitly programmed in order to accomplish a task.

However, for these methods to be usable and effective in practice, an efficient implementation of the algorithms is needed, exhibiting good scalability when provided with massive real-world data sets. For instance, reducing the processing time required to deliver results is not only highly desirable to increase the efficiency of production workloads, but also allows data scientists to be quicker at implementing and evaluating new ideas. GP-GPUs established as the go-to computing platform when it comes to accelerating ML algorithms, due to their extremely parallel and high-precision computing capabilities. It is well known that the advances in the technology behind such hardware accelerators have enabled researchers to show the disruptive effectiveness of DL in fields like Computer Vision [41, 117].

In this work, we focus on SOMs [115], a kind of unsupervised, shallow, ANNs, built on top of the competitive learning principle and typically employed for clustering, dimensionality reduction and high-dimensional data visualization. Indeed, they are designed for mapping high-dimensional data into a lower-dimensional space (e.g., 2D) that is better interpretable by human perception and easier to treat computation-wise, while preserving the *topology* and *distribution* of the original data at cluster-level. Given their ability to yield a data distribution in the target domain that faithfully reflects the observed relationships in the original space, SOMs have achieved remarkable results in many application fields like: image processing [57, 102], industrial data processing [29, 64], data visualization [44, 176, 258], pattern recognition [133, 259], anomaly detection in NFV infrastructures [50, 126] (see also Chapter 3).

Thanks to their simplicity, a wide variety of open-source implementations of SOMs is available. In this work, we focused on those exposing *Python* APIs, due to the raising popularity of such programming language in the ML community. The available implementations differ significantly in terms of performance under various scenarios. The differences can be caused by several factors like: their reliance on native vector processing and linear algebra libraries, that carry out most of the computations in *C/C++*, e.g., through *NumPy* [86]; their internal parallel architecture and exploitation of the underlying multi-core hardware; their capability to exploit GP-GPU acceleration, e.g., through CUDA [162]; or, the way several input samples are batch-processed so to parallelize computations and minimize the execution of slow Python `for` loops.

4.1.1 *Contributions*

We focus on the parallelization and acceleration architecture of SOM implementations, performing an extensive performance comparison of widely available open-source libraries for SOM computations, namely *MiniSom* [242], *Somoclu* [252] and *TensorFlow-SOM* [79], under various configuration options, leveraging both multi-core and GP-GPU acceleration. Also, we present for the first time *XPySom*, a novel open-source SOM implementation designed to leverage existing and widely available frameworks for accelerated processing like NumPy, particularly effective when coupled with the Intel MKL [247] or other BLAS libraries for parallel processing on multi-core CPUs, and *CuPy* [165] for GP-GPUs. We show that a proper design of the data processing operations, arranged so to perform a relatively small number of calls from Python to the natively accelerated libraries just mentioned, may result in a high-performance implementation that outperforms the others. Indeed, an experimental comparison carried out processing the EMNIST [42] dataset, highlights that *XPySom* exhibits a performance that is an order of magnitude better than the other evaluated SOM implementations, both for multi-core and GPU-accelerated platforms.

4.1.2 *Chapter Organization*

This chapter is organized as follows. In Section 4.2, we provide an overview of relevant approaches found in the research literature aiming at increasing SOM performance, mostly by parallelizing the training algorithm. In Section 4.3, we present *XPySom*, discussing the most important choices in its design regarding optimized execution and parallelization. Section 4.4 reports the results of the benchmark we have run to compare our implementation to other widely used open-source ones. Section 4.5 includes our final remarks and possible ideas for further work on the topic.

4.2 RELATED WORK

Virtually all ML algorithms can benefit from an implementation that exploits the parallelization capabilities of modern hardware architectures, being them classical statistical learning algorithms [206], visualization methods for high-dimensional spaces [32] or clustering techniques [224]. This work focuses on SOMs, an instance of the latter class of ML-based approaches.

SOMs make extensive use of vector operations, which can be accelerated at various levels, from leveraging the SIMD vector instructions of high-end processors like SSE and AVX, to multi-core processing widely available on basically all computing platforms, to the use of GP-GPUs featuring thousands of processing units able to compute in parallel the same kernel at high speeds exploiting the fast local memory on the GPU.

Several works appeared in the research literature dealing with SOM performance and optimizations. In [196], a parallel SOM implementation for interactive high-performance data analysis is proposed. Indeed, the classical (serial) training procedure for SOMs, that consists in determining the BMU and then updating the units weights accordingly for each training sample, is not feasible in contexts where massive datasets must be processed and results are expected to be returned in near real-time (e.g., interactive web searches). The approach is based on *(i)* partitioning the map over multiple processors, each one responsible for solving a local (minimization) problem of finding the BMU in its partition, *(ii)* aggregating the partial results to compute the global BMU and *(iii)* propagating the solution to allow for weights updates. The authors show that the introduced synchronization overhead is negligible with respect to the floating-point operations involved in the training process, that can be further optimized to leverage better L2 caches.

In [207, 252], attempts at distributing the SOM training algorithm using the *MapReduce* [58] framework are described. While [207] relies on a pure *Spark* [262] implementation to effectively scale on massive datasets, [252] also allows for accelerating *map* and *reduce* jobs on GPUs, by leveraging on the MapReduce-MPI [182] framework.

In [152], a thorough scalability analysis of SOMs on a GPU cluster is reported. In particular, *OpenCL-* [225] and CUDA-based single-GPU approaches, as well as a multi-GPU implementation combining CUDA and MPI, are evaluated with respect to an MPI-only baseline, considering also two different types of graphic cards. Results show that the CUDA-based implementation outperforms the OpenCL-based one, mainly due to the fact that the latter framework is designed to be compatible with a heterogeneous set of devices, and that the multi-GPU approach allows for a relatively small speed-up, because of the synchronization requirements of the training procedure.

The authors of [256] highlight that the performance of pure CUDA implementations of SOMs are poor when dealing with large neighborhoods, since many weights update operations end up being serialized. To address this limitation, the authors propose an approach in which the computation of the distances between training samples and map units is implemented as a matrix multiplication with compute shader, and the weights updates are treated as a vertex rendering problem.

In [139], a heterogeneous parallel implementation based on MPI and CUDA is proposed, which is highly scalable on multiple GPUs and multiple hosts, thanks to the employed process-level and thread-level (data) parallelism. In particular, the approach leverages on the batch version of the SOM training algorithm that performs a single weights update per epoch (i.e., after all training samples are consumed by the model), whereas the original one performs a weight update after each step of an epoch (i.e., after a single training sample is consumed). The former training strategy requires fewer computations and results in faster convergence. In addition, such version of the algorithm is a perfect candidate for highly parallel implementations, because the most computationally-intensive parts can be turned into matrix operations that can be performed with *cuBLAS*, fully exploiting the computational capacity of GPUs.

In [200], a SOM implementation that combines both data and model parallelism is described. This implementation pushes the parallelization capability of the batch training algorithm to the extreme, as not only the training set is split in chunks to be processed independently by copies of the map (data parallelism) but each copy is also partitioned at unit weights-level, so that a separated GPU thread handles the updates of a single dimension of each neuron. According to the experiments conducted by the authors, this method performs best when dealing with large data sets, but its effectiveness starts decreasing when increasing the number of features per data point. Also, by profiling the GPU usage, the authors discovered that, even though the thread processor occupancy was very close to the theoretical limit, their throughput was relatively low. This was most likely due to the time spent waiting for a CUDA core to be available, a problem that can be certainly mitigated by using a more powerful graphics card.

In [253], an efficient parallel SOM library, called *Somoclu*, is presented as an improvement on [252]. Such implementation offers high flexibility, as it can be effectively executed on a single machine as well as on a cluster, both on CPUs and GPUs, and exposes interfaces compatible with widely-used data analysis ecosystems (e.g. *Python*, *R* and *MATLAB*). The *Somoclu* core implementation is based on *OpenMP* [55] to achieve efficient single-node parallelism, and MPI, which replaces the MapReduce framework employed in [252], to distribute the computation among multiple nodes. Furthermore, a GPU kernel implemented with CUDA Thrust [14] and cuBLAS is available. Notably, due to its remarkable performance and versatility, we decided to use *Somoclu* as a reference for the evaluation of our approach.

The authors of [121] compared several SOM implementations, targeting many diverse computing platforms ranging from general-purpose CPUs to FPGAs, both in terms of performance and energy efficiency. With respect to a baseline provided by the *MATLAB* SOM toolbox, a multi-threaded CPU implementation is able to achieve a speed-up of ~200x with small networks. Although, when dealing with large, high-dimensional data, GPU and FPGA implementations performs best, with the latter being the most energy-efficient while reaching a speed-up of ~200x with respect to the multi-threaded ~CPU implementation.

Among the approaches described above, only a few of them have been made widely available as open-source projects exhibiting a Python ~API, actively maintained and well documented with online examples. We chose to compare our proposed *XPySom* with a few projects we found with the mentioned characteristics, which include: *Somoclu* [252], due to its versatility (besides multi-core parallelism, it can also exploit GP-GPUs as well as multiple nodes in a cluster) and promising performance (at the core, it is implemented in C/C++ and exposes bindings for higher-level languages such as Python); *MiniSom* [242], due to the fact that, thanks to its simplicity, our code base was realized as a modification to it; and *TensorFlow-SOM* [79], due to its seemingly promising approach based on the well-known *TensorFlow* [151] framework. *MiniSom* implements the online algorithm described in Section 2.2.1, exploiting exclusively vector-vector and vector-matrix operations (i.e., Euclidean distance, vector additions and scalar multiplications) which, albeit implemented efficiently in *Numpy* and, if possible, leveraging on multiple cores, are invoked several times from the Python language. Specifically, Python makes a number of calls to *Numpy* vector operations for each input sample, updating the neurons after each sample. Notice that such implementation pattern, with everything explicitly coded in Python, is among the easiest to implement and makes the code readable and easily modifiable. Unfortunately, this also results in a very poor implementation in terms of performance, as shown in Section 4.4. *XPySom*, on the other hand, implements the batch algorithm described in Section 4.3.1, the same used in *Somoclu* and the compared *TensorFlow* implementation, with the improvements described in Section 4.3.1, heavily exploiting higher-dimensional operations (e.g., matrix-matrix) to work on batches of input samples for each native call, resulting in a number of native calls proportional to the number of batches. Furthermore, neurons are updated once per epoch, with just the numerator/denominator accumulation required for each batch, which results in a much lower number of overall operations. Thanks to the additional implementation details provided in Section 4.3.2, *XPySom* is able to exploit either *Numpy* or *CuPy* interchangeably, leveraging on either multiple CPU cores or GPU.

4.3 PROPOSED APPROACH

This section introduces our new *XPySom* library for Python. We show the reformulated *batch* algorithm (see Section 2.2.1) that makes use of matrix-based operations to take advantage of existing BLAS libraries (e.g., *cuBLAS* and MKL).

4.3.1 *Matrix-based batch SOM*

To take advantage of BLAS libraries, that are highly optimized for execution on both GPU and CPU (e.g., making use of vector instructions), the *online* and *batch* formulation can be rewritten to operate on a batch of B data samples at a time arranged in a $B \times P$ matrix $X \in \mathbb{R}^{B \times P}$, where P is the dimension of the weights vectors. In the following, the weights are assumed as arranged in an $M \times P$ matrix $W \in \mathbb{R}^{M \times P}$, where M is the number of neurons. Also, the $M = G_w G_h$ neurons are usually arranged as a $G_w \times G_h$ grid (in the following, assumed rectangular). Therefore, each neuron W_i has also its associated coordinates $r_i = (i\, div\, G_w, i\, mod\, G_w)$ in the 2D grid (where *div* and *mod* represent the quotient and modulus integer operations, respectively).

The steps of the matrix-based batch algorithm can be summarized as follows. First, the distance matrix $D \in \mathbb{R}^{B \times M}$ is computed, consisting of all pair-wise distances between X and W (see Section 4.3.1.1). Then, for each sample, the BMU is computed, yielding a vector $BMU \in \mathbb{N}^B$ where $BMU_i = \arg\min_k D_{i,k}$ (i.e., the index of the smallest value in each row of D). The neighborhood function is then computed for each element of BMU, yielding a matrix $H \in \mathbb{R}^{B \times M}$ (see Section 4.3.1.2). Successively, numerator and denominator updates are computed, yielding a matrix $NUM \in \mathbb{R}^{M \times P}$ and vector $DEN \in \mathbb{R}^M$. Finally, after all batches of the epochs are processed, the weights of the neurons are updated (see Section 4.3.1.3).

4.3.1.1 *Distance matrix*

There exists an efficient formula for computing pair-wise distances between two matrices:

$$D^2 = X^2 - 2XW^T + W^2 \tag{4.1}$$

$$D^2_{i,k} = \sum_j X^2_{i,j} - \sum_j 2X_{i,j}W_{k,j} + \sum_j W^2_{k,j} \tag{4.2}$$

Note that, since we are interested only in finding the minimum over a row (i.e. for each sample in the batch), there is no need to compute the square root of D^2. Furthermore, if we express Equation (4.1) in terms of element-wise operations, as in Equation (4.2), we obtain that: *(i)* X^2 is constant row-wise (i.e., for equal values of i), hence we can skip its computation, since we are only interested in finding the BMU; *(ii)* W^2 is constant over an epoch and, therefore, it can be computed only once.

4.3.1.2 *Neighborhood function*

By *unraveling* the BMU_i indices we can obtain directly the coordinates of the BMU in the rectangular grid (see Equation (4.3)). Let $U \in \mathbb{N}^{B\times 2}$ be the matrix with the unraveled indices of the BMUs and $L \in \mathbb{N}^{M\times 2}$ the matrix with the locations r_i of the M neurons. We can compute H as follows:

$$U_{i,1} = BMU_i \, mod \, G_h \qquad U_{i,2} = BMU_i \, div \, G_w \tag{4.3}$$

$$H_{i,k} = -exp\left(\frac{(U_{i,1} - L_{k,1})^2 + (U_{i,2} - L_{k,2})^2}{\delta(t)}\right) \tag{4.4}$$

4.3.1.3 *Numerator, denominator and weights update*

The updates NUM' and DEN' can now be computed as:

$$NUM'_{k,j} = \sum_i H_{i,k} X_{i,j} \qquad DEN'_k = \sum_i H_{i,k} \tag{4.5}$$

The corresponding matrix operation for the numerator in Equation (4.5) is $NUM' = H^T X$ where NUM' and DEN' are accumulated in NUM and DEN. The new weights $W(t+1)$ are calculated as $W_{k,j}(t+1) = NUM_{k,j}/DEN_k$.

4.3.2 *XPySom*

XPySom is our SOM implementation in Python that uses the matrix-oriented formulation of the algorithm of Section 4.3.1. *XPySom* has been obtained as a (quite disruptive) modification to *MiniSom*, which has been chosen as a starting point due to its simplicity of implementation and richness of features. In *XPySom*, the sequential sample-by-sample operations originally in *MiniSom* have been replaced with matrix operations as detailed above. These are executed using the APIs provided by either *NumPy* or *CuPy*, which are able to exploit CPU and GPU processing, respectively. We exploit the interchangeability among *NumPy* and *CuPy*, as *CuPy* implements the same APIs as *NumPy* but it executes its operations on GPUs through cuBLAS calls or raw kernels.

The obtained data flow is exemplified in Figure 4.1, where the various steps XPySom goes through in order to perform a SOM training

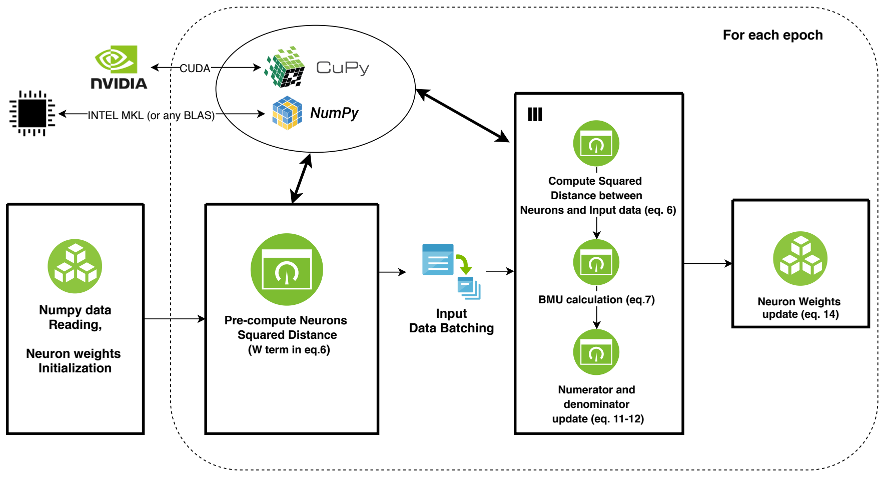

For each epoch
CUDA
CuPy
INTEL MKL (or any BLAS)
NumPy
Numpy data Reading,
Neuron weights Initialization
Pre-compute Neurons Squared Distance (W term in eq.6)
Input Data Batching
Compute Squared Distance between Neurons and Input data (eq. 6)
BMU calculation (eq.7)
Numerator and denominator update (eq. 11-12)
Neuron Weights update (eq. 14)

Figure 4.1: Visualization of the data flow for SOM training in *XPySom*.

operation are visualized, in connection with the corresponding equations defined in the previous subsections. Core features of the *XPySom* architecture are its capability to perform SOM training operations in batches of input samples, arranged as matrix/vector operations that are executed very efficiently through relatively few calls to the underlying NumPy or CuPy libraries. *XPySom* is able to use either of them interchangeably, thanks to their compatible APIs.

XPySom is an open-source project and the code is available under GPLv3 license [147]. At the moment, *XPySom* does not support execution on multiple nodes nor multiple GPUs. Moreover, when in GPU mode, *XPySom* cannot make use of CPU cores to accelerate further the processing.

4.4 EXPERIMENTS

In this section we present the results we obtained, both in terms of QE[1] and training time, from an extensive experimental comparison among our proposed *XPySom* and a few other commonly available SOM implementations: *MiniSom* [242], *SomoClu* [253] and *TensorFlow-SOM* (TF-SOM) [79]. For the experiments, we relied on a workstation equipped with 16GB of DDR4 memory, an Intel Core i7-4790K quad-core CPU (8 hyper-threads) with base frequency 4.00 GHz (turbo-boosting to 4.40 GHz) and an Nvidia GeForce GTX 1080 Ti with 11 GB of on-board memory and 3584 CUDA cores, running Ubuntu 18.04 LTS. The installed libraries and packages were Python 3.6.5, NumPy 1.18.1, *MiniSom* (at commit `0540834`), *XPySom* (at commit `6103d86`), *Somoclu* 1.7.5 (built from source at commit `7c78448`), CuPy 7.4.0, CUDA 10.2, *TF-SOM* (at commit `c0f40ed`) and TensorFlow 2.1.0.

1 The QE is defined as the average distance between each input vector and the weight vector of its associated BMU neuron.

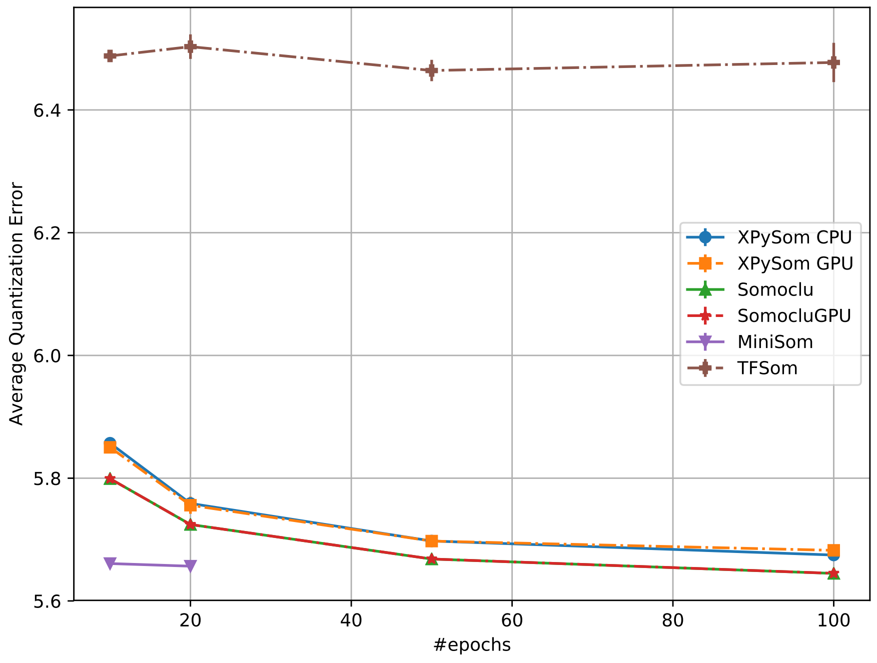


Figure 4.2: Evolution of the QE for a 10x10 SOM throughout training epochs (EMNIST, 240k samples, 784 features).

We performed five different tests each comparing different training environments. To have a fair QE evaluation, we ran each experiment five times, and used the mean QE as the *error metric* and mean training time as the *performance metric*. The input data used in the experiments is the EMNIST dataset [42] that contains 240k data samples, each composed of 784 features. For all the experiments, the input data values were divided by 255 so that all the features are scaled in the $[0, 1]$ range. In the *TF-SOM* implementation, the batch size is set to 128, which shows a nice compromise in terms of memory usage and computation time. The learning rate starts every time with a value of 0.5 and decays exponentially over epochs in all the tested implementations.

4.4.1 *Quantization error vs training epochs*

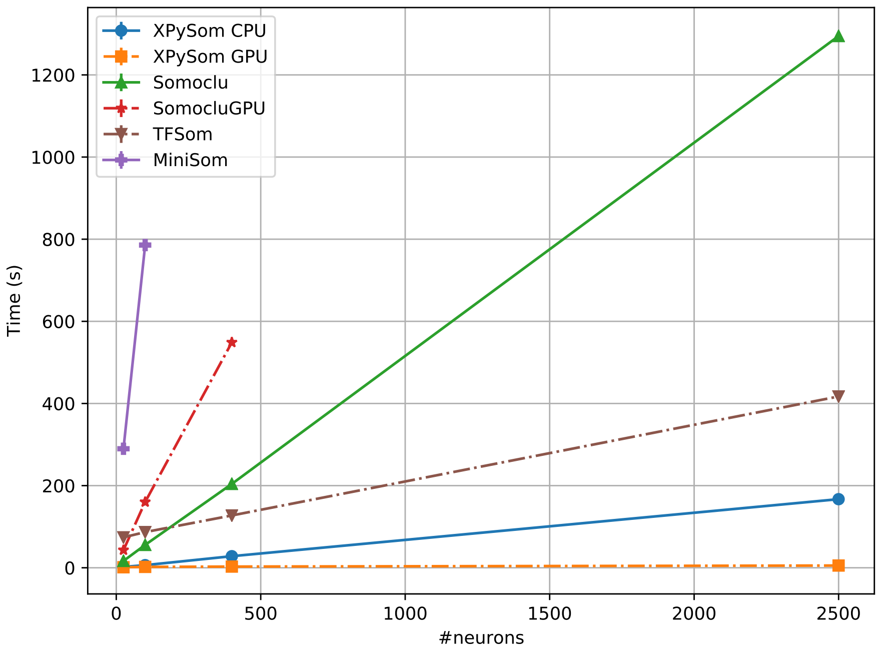
The first test aimed to exclude from further performance experiments all the SOM implementations that did not reach an acceptable QE after a fixed amount of training time, and SOM initialization techniques that did not bring performance improvements. The QE is averaged over 5 training sessions and measured after each epoch for a total of 100 epochs. The number of neurons is fixed to 100 and arranged in a 10×10 rectangular grid. Since *MiniSom* offers the possibility to initialize the weights using the PCA or the random initialization, we have chosen to run the experiment with both initialization techniques while in the *Somoclu* implementation the initialization is the random one. The training update rule is the one described in Equation (2.5) used in both *XPySom* and *Somoclu* implementations, while in the *MiniSom* implementation the training update rule is the online update rule described in Equation (2.3).

Figure 4.2 shows how both *Somoclu* and all *XPySom* implementations reach similar mean QE through all the 100 epochs. *MiniSom* reaches a lower QE in fewer epochs just because of the online update rule that updates all the SOM weights for each sample, resulting in more updates within the same number of epochs. After 20 epochs the *MiniSom* experiment is quitted since it reaches the time limit of 30 minutes. *TF-SOM* seems to be unable to lower the QE even after 100 epochs as it remains with a QE between the 15% and 20% higher than the other SOM implementations. PCA initialization of both *MiniSom* and *XPySom* seems to not help to lower the QE with respect to the random initialization, and the only difference seems to be in the slightly higher initialization time due to the initial PCA decomposition. For this reason, since the following experiments will focus only on the time performance, the faster random initialization will be used in all SOM implementations.

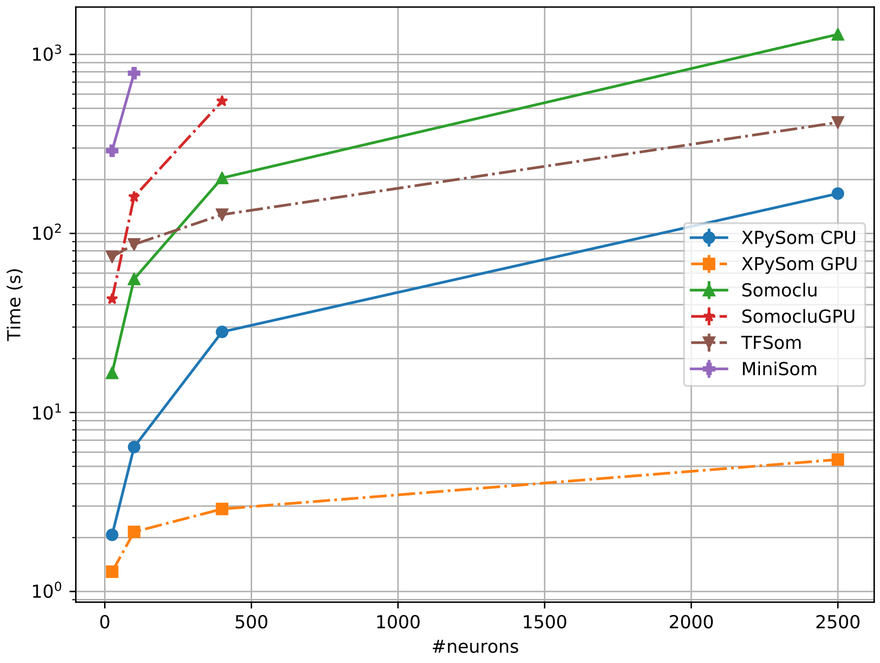
4.4.2 *Training time vs SOM grid sizes*

The results obtained in Section 4.4.1 suggest that the batch update rule used in all SOM implementations tested except the original *MiniSom* implementation does not worsen the QE compared to the online one. In this section, we focus on how the training time is affected when increasing the number of SOM neurons instead. Figure 4.3 shows that, increasing the number of SOM neurons (on the X-axis), the training time increases (in seconds on the Y-axis, averaged on 5 different training sessions, in linear and logarithmic scale in Figure 4.3a and Figure 4.3b, respectively). The number of epochs is fixed to 10, the number of training samples is fixed to 240k and the number of features is fixed to 784.

The *XPySom* implementation outperforms the *Somoclu* implementation in both OpenMP (CPU, just labelled as *"Somoclu"* in the figure) and CUDA (*"SomocluGPU"* label in the figure) compiled versions by two and three orders of magnitude. It is worth noticing how *XPySom* effectively leverages the GPU parallelization while *Somoclu* seems to perform worse when using the GPU. A quick investigation revealed that in this case *Somoclu* makes a non-intensive use of the GPU, while still keeping a significant amount of computations on the CPU (on a related note, the GPU and CPU kernels are kept in different files, in the source code). The *MiniSom* original implementation is not using any sort of explicit parallelization and the training time grows uncontrolled when increasing the number of neurons. The *Somoclu* GPU compiled version starts to become unusable when the number of neurons approaches the size of 500, while *TF-SOM* starts to be faster



(a)



(b)

Figure 4.3: Training time as a function of the number of neurons (EMNIST, 240k samples, 784 features).

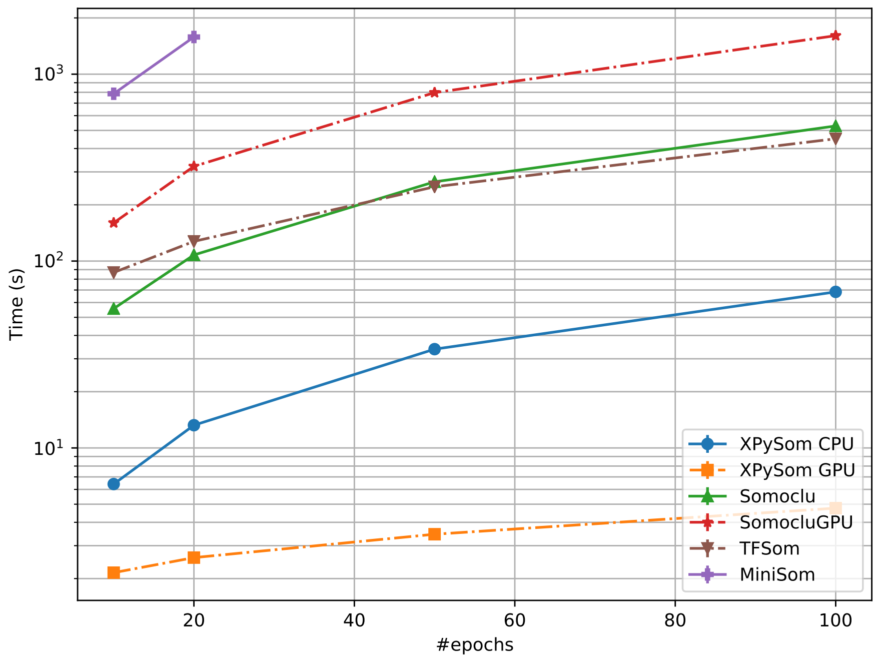


Figure 4.4: Training time as a function of the number of training epochs (EMNIST, 240k samples, 784 features).

than the *Somoclu* one when the number of neurons approaches the size of 500, but if remains two orders of magnitude slower than the *XPySom* GPU implementation.

4.4.3 *Training time vs training epochs*

Figure 4.2 shows how increasing the number of training epochs leads to a decrease in the quantization error. Figure 4.4 shows how the training time increases linearly with the number of epochs. It is worth to notice that even after 100 epochs the *XPySom* GPU implementation ends the training session in less than 10 seconds, more than two orders of magnitude faster than the second faster GPU SOM implementation (*TF-SOM*).

4.4.4 *Training time vs training samples*

The training time increases linearly with the data set size and the log-plot in Figure 4.5 follows the expected behavior in all the six tested SOM implementations. Again, the *XPySom* implementation shows the best performance with both the NumPy (CPU) and CuPy (GPU) backends.

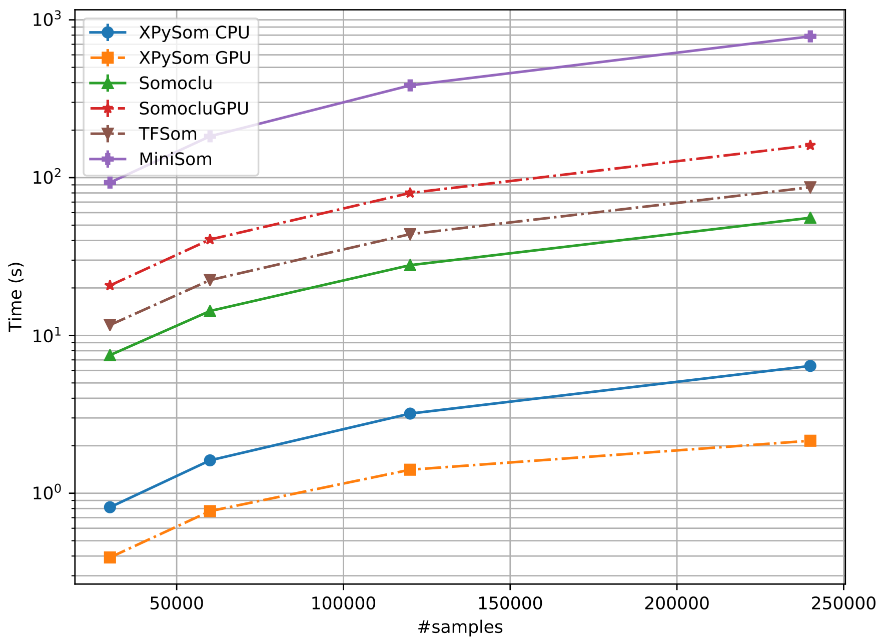


Figure 4.5: Training time as a function of the number of training samples (EMNIST, *max* 240k samples, 784 features).

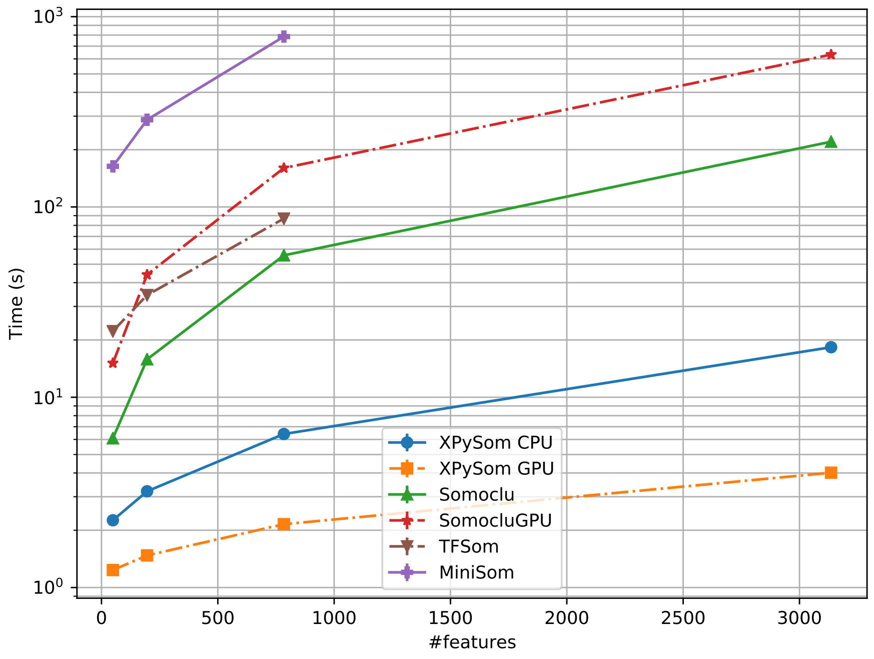


Figure 4.6: Training time as a function of the number of training features (EMNIST, 240k samples, 784 features)

4.4.5 *Training time vs input features*

To check how the training time is impacted when using more input features, we have scaled the original input samples to the following sizes using a bilinear interpolation over the pixel neighborhood:

- 7×7, resulting in a input dataset of size $240k \times 49$
- 14×14, resulting in a input dataset of size $240k \times 196$
- 28×28, resulting in a input dataset of size $240k \times 784$
- 56×56, resulting in a input dataset of size $240k \times 3136$

The log-plot in Figure 4.6 shows that the *XPySom* implementation outperforms the other SOM implementations by two or three orders of magnitude. The number of epochs is fixed to 10 while the number of training samples is fixed to 240k. *TF-SOM* cannot start the training session with 3136 features due to a memory error (exceeded available 11Gbyte GPU memory) using a batch size of 128. On the other hand, the *MiniSom* implementation cannot reach the training session ends since it exceeded the maximum conceded training time (30 minutes) for our experiments.

4.5 CONCLUSIONS

We presented *XPySom*, a variant of the popular *MiniSom* package for Python that effectively leverages the parallelization of the batch update rule for training SOMs, recurring to a massive use of matrix/vector operations optimized through the use of the well-known NumPy and CuPy libraries. We have tested our implementation on a single GP-GPU, multi-core CPU machine using different training settings. Extensive experimental results demonstrate that, even when increasing the number of neurons, the number of training samples or the number of training features, our implementation outperforms other popular open-source implementations for Python (including *Somoclu* that has a native C/C++ implementation), with a training time two or three orders of magnitude lower and a practically identical accuracy (in terms of QE). Our proposed *XPySom* implementation is certainly an interesting choice when the SOM training can be run on a single machine, being probably among the fastest SOM implementations available for Python. Moreover, the algorithm is extremely easy to be adapted to custom implementations since most of the functions are inherited from the *MiniSom* minimalistic package.

Part III

PREDICTIVE RESOURCE MANAGEMENT

In this part, we investigate on data-driven approaches for cloud resource management. In particular, we focus on time-series analysis techniques, applied to system- and application-level monitoring metrics. We analyze existing solutions, and propose novel methods, to address a number of very common problems faced by operations teams during their daily activities, like capacity planning, elasticity control and fault management.

5 VNF METRICS FORECASTING

5.1 INTRODUCTION

In recent years, the landscape of information and communication technologies has been facing an unprecedented turn into distributed computing. The wide-spread availability of high transmission bandwidths, at affordable rates, enabled multiple scenarios where computing can effectively and efficiently be distributed. Coupled with the relentless development of virtualization technologies, this led to the realization of nowadays cloud computing. Cloud technologies enabled the flexible management of pools of shared general-purpose processing, storage and communication resources. Consumers can remotely access them in an on-demand, rapid, completely automated and dynamically adaptable fashion.

At the same time, communication technologies have been evolving towards more sophisticated services, higher capacity and lower latency in both landline and mobile access networks, as well as in backbone transport segments. The new distributed computing scenarios, including big-data processing in the cloud and on-the-fly processing at the edge, exposed the incapability of traditional networks at managing the complex and fast-evolving requirements of new and emerging services.

Thanks to the increasing convergence of networking technologies towards IP-based networks (e.g., LTE), telecommunications are benefiting of well-established principles coming from the cloud computing space. This led to the recent paradigm of NFV [39], where general-purpose private cloud infrastructures allows for quickly provisioning virtualized resources (i.e., *network slices*) in which to instantiate flexible VNFs, seconding the instantaneous workload conditions and their requirements. In this way, a more *intelligent* use of physical resources can be pursued [40].

At the heart of automated management and orchestration operations (including capacity planning, performance monitoring and management), there are *monitoring systems* continuously gathering a plethora of metrics, for each individual element of the infrastructure. For instance, tens of system-level metrics may be gathered from each individual physical host, VM or networking element, with a typical time granularity of one sample every few minutes. At the same time, individual VNFs continuously collect metrics related to the status, performance and failures happening at the application level. This data is normally aggregated and analyzed in real-time by an alerting system that, under precise threshold-based conditions on said metrics, is able to trigger operators' attention (e.g., when failure rates or response latencies exceed certain thresholds), or even activate automated fault management and recovery actions (e.g., exclude a physical host from the fleet and send it to data-center operators for repair and maintenance).

Accurately forecasting how key metrics will evolve over time is an increasingly important problem, both for short-term and mid/long-term forecasts. For this reason, ML techniques have been gaining momentum as key technologies accompanying enterprises operating in pretty much any business domain. In networking, these techniques have been successfully applied in NFV infrastructures for AD [158, 164], behavioral pattern analysis [50, 126] as well as resource demand estimations [100, 157]. In particular, DL methods are among the techniques that are receiving increasing attention from both research and industry. Therefore, it is worth investigating the applicability of DL models and the trade-offs that can be achieved in terms of precision and training cost of the available techniques.

5.1.1 *Chapter Organization*

This chapter is organized as follows. Section 5.2 offers an overview of the related research literature. Section 5.3 presents the time-series forecasting approaches considered for our analysis. Section 5.4 presents the results of our experiments. We validated the performance of the considered models, in terms of accuracy and training cost, using real data exported from a production environment of the *Vodafone* network operator. Section 5.5 concludes the chapter and discusses possible ideas for future works on the topic.

5.2 RELATED WORK

Adoption of predictive techniques in NFV and SDN schemes is a long-standing approach for adapting available virtualized resources to varying loads [144, 158, 164]. In this way, *service-chains* deployed on cloud infrastructures are able to offer quick proactive measures to ensure quality of service QoS and reduction of CAPEX and OPEX costs. However, achieving such a goal comes with several challenges like, for instance: (i) employing effective predictive models that do not lead to under- or over-provisioning virtual resources; (ii) assessing which components of a service-chain should be scaled to maximize the overall efficiency; (iii) optimizing the placement of newly created virtual resources on the physical infrastructure [70, 119]. Our work focuses on (i), in particular, by investigating effective time-series forecasting techniques that can provide NFV operation teams with actionable feedbacks to support their decisions (e.g., whether a VNF needs to be scaled to accommodate traffic growth).

A standard solution for real-time forecasting and scaling of resources is *static* thresholding [30]. Despite being a straightforward heuristic, it can provide interesting results when dealing with simple systems. However, in general, different services require different threshold policies, thus a generic approach will lead to over- or undersizing the infrastructure. On the other hand, *dynamic* thresholding provides an adaptive mechanism to set thresholds, that can be implemented with, e.g., RL [5, 6]. However, the burdens related to using RL algorithms often limit their applicability to real infrastructures (e.g., the need for sophisticated simulation environments to train the agents). For instance, in [227] the authors describe a Q-learning approach for managing a real telco system. The developed agent is observed to take several unexpected decisions, before converging to an optimal policy. When deploying the approach in production, this is clearly not desirable. On the other hand, in [257], the authors propose a successful RL-based approach to deploy VNF service-chains, that works by jointly minimizing operation costs and maximizing requests throughput, and also takes into consideration heterogeneous QoS requirements.

In [188], an ML-based approach is proposed to realize an effective auto-scaling mechanism. The authors evaluate several predictive models (e.g., DT, RDF, MLP, BN) on load traces exported from a real VNF environment, also taking into account the different costs and start-up times related to different virtualization technologies. In [263], an approach based on LSTM networks, is adopted to forecast VNF requirements. However, it is not clear whether there is an actual improvement, in terms of forecasting accuracy, when compared to other methods, as the authors mainly focus on features selection aspects.

Nowadays, LSTM has been proven to be an extremely effective tool for time-series analysis, both for classification [98, 145] and forecasting [128, 193] tasks. In particular, the *sequence-to-sequence* architectural pattern, when implemented with LSTM-based *encoder* and *decoder* components [226], yields surprisingly good results. This kind of architectures are also widely adopted for machine translation and NLP tasks in general. In NFV context, they can be helpful when translating VNF metrics sequences to infrastructure metrics sequences and vice-versa. Since communication from the encoder to the decoder is limited to the hidden state values, there are no real requirements on the structure and architecture of both, thus allowing even for different types of input and output metrics.

As described in [158, 164], forecasting accuracy can also be improved by leveraging on information about the topology of the deployed VNFs

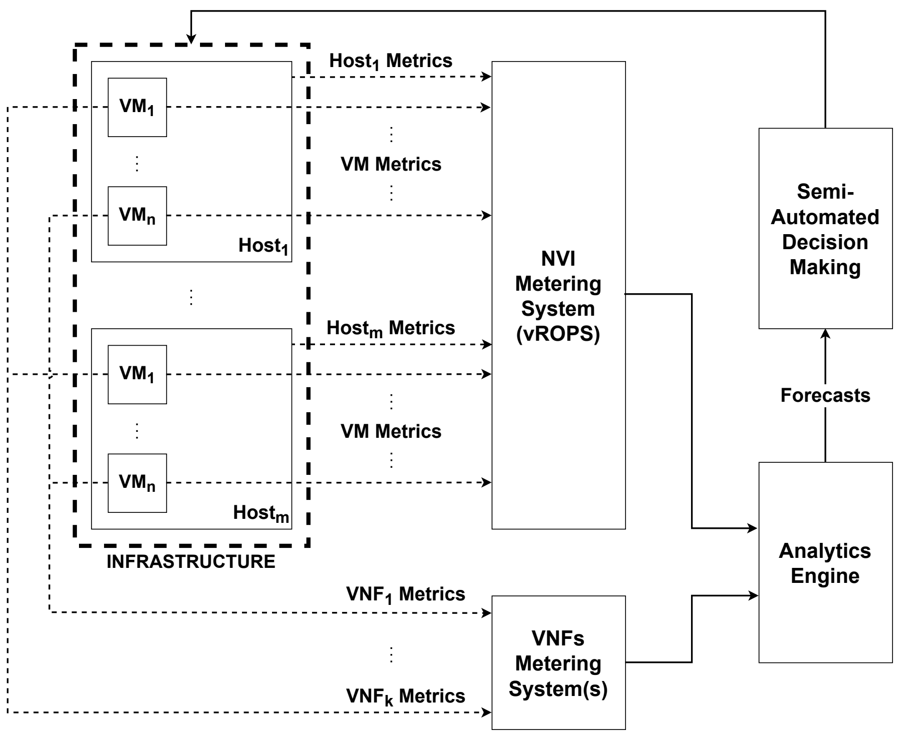


Figure 5.1: Integration of the realized analytics engine within Vodafone NFV infrastructure operations.

like, e.g., graphs characterizing interactions among the VMs belonging to the same VNFs. In [157], such *topology-aware* time-series forecasts were achieved through GNNs [10, 210].

5.3 COMPARED APPROACHES

NVI metric forecasting, both at system- and application-level, aims at providing a snapshot of the system dynamics in the future. This information is useful to operation teams to support their decisions, e.g., for capacity planning purposes. Figure 5.1 shows how forecasting capabilities enhance Vodafone NFV operations. The NVI time-series produced by each component of the infrastructure are ingested by vROps and the monitoring subsystems of the individual VNFs. Our analytics engine sits in front of these monitoring components and computes NVI forecasts. Such outputs are used in a semi-automated decision-making process, where humans consume them to get insights and possibly uncover early symptoms of system outages. The acquired knowledge provides decision-makers with actionable feedbacks that may trigger capacity planning and infrastructure management actions.

The data provided by Vodafone consists of a set of relevant indicators (metrics), each one coming in the form of a time-series that describes the evolution in time of a specific NVI metric. Dealing with time-series, especially in multi-variate settings, is an inherently complex problem. An effective model must take into account the dynamic and sequential nature of the information. Furthermore, possible high variance in the data poses an additional challenge. To mitigate the impact of such quality issues, we applied a pre-processing pipeline that includes scaling and normalizing the data using a *min-max* strategy.

We tested three classes of metric forecasting methods: (i) SARIMA, (ii) Holt-Winters and (iii) neural architectures. In order to train the latter on NVI metrics, we reshaped the data-set in the form of training samples. Namely, we built a set of pairs of input and output subsequences, fixing the length of input and output time periods upfront.

Note that supervised models rely on the data seen during training, to discover patterns and correlations among the defined variables. Although several techniques have been applied to reduce the generalization error, such models perform the best when test data and training data distributions are similar.

5.3.1 *Neural Architectures*

From a general standpoint, we aim at forecasting the future dynamics of a discrete uni-variate time-series $\{x_t\}$, for $x_t \in \mathbb{R}$, leveraging on the knowledge of its values for $t \in [0, T_{\mathtt{train}}]$. In other words, we want to compute $\{\hat{x}_t\}$ for $t \in [T_{\mathtt{train}} + 1, T_{\mathtt{test}}]$. If available, we can exploit additional *auxiliary* metrics $\{y_t\}$, for $y_t \in \mathbb{R}^m$ whose historical dynamics is known in the same time range as the prediction target $t \in [0, T_{\mathtt{train}}]$. In the context of NFV operations, for instance, $\{x_t\}$ could represent the average cpu utilization of a given subset of hosts in the infrastructure, while $\{y_t\}$ could represent a set of service-level indicators that are particularly relevant for the VNF deployed on such hosts.

All the models described in this section are examples of *encoder-decoder* architectures, a common architectural pattern for sequential data processing. An *encoder-decoder* architecture is composed of two distinct layers connected in series: (i) the *encoder* $\mathcal{E}_\theta$, whose job is to accept a (possibly variable-length) sequence in input and compute a *state* with fixed shape, and (ii) the *decoder* $\mathcal{D}_\theta$, that maps the state to an output sequence.

The parameters of the neural models can be trained by comparing their outputs with the ground truth values, provided by the training data-set, and minimizing the MSE. To this aim, in our implementation, we used the *Adam* optimization algorithm, setting its hyperparameters as recommended by the seminal paper [114].

5.3.1.1 *Baseline*

Our baseline architecture is rather simple and consists of an LSTM layer followed by a fully-connected layer, jointly trained by back-propagation. An input time-series is fed to the LSTM that is set to output only the last value of its recurrent process, for each internal unit. The resulting vector is then provided to the fully-connected layer that outputs an estimation of the target time-series in a *one-shot* fashion (i.e., the entire output is generated by a single inference pass). More formally, if φ is the desired length of the output sequence, d the number of neurons of the LSTM, m the number of auxiliary variables, the model is characterized by:

$$\begin{aligned} &\mathcal{E}_\theta : \mathbb{R}^{m+1} \times \mathbb{R}^{2d} \to \mathbb{R}^{d} \times \mathbb{R}^{2d} \\ &\mathcal{D}_\theta : \mathbb{R}^{d} \to \mathbb{R}^{1\times\varphi} \end{aligned} \tag{5.1}$$

Note that $\mathcal{E}_\theta$ is designed to process each element of the input sequence separately, not the whole sequence at once. Let $(x_{t-\lambda+1}, \dots, x_t)$ be the input sequence and $(y_{t-\lambda+1}, \dots, y_t)$ be the auxiliary sequence, both with length equal to λ. The output $\hat{x} = \left(\hat{x}_{t+1}, \dots, \hat{x}_{t+\varphi}\right)$ is generated as follows:

$$\begin{aligned} &\tilde{h}_{t-\lambda} = 0, && \tilde{h}_{t-\lambda} \in \mathbb{R}^{2d} \\ &z_s = \operatorname{vec}\left(x_s, y_s\right) && t-\lambda < s < t \\ &(o_s, \tilde{h}_s) = \mathcal{E}_\theta(z_s, \tilde{h}_{s-1}), && t-\lambda < s \leq t \\ &\hat{x} = \mathcal{D}_\theta\left(o_t\right) \end{aligned} \tag{5.2}$$

where $\tilde{h}_s$ are the latent states.

Due to the way the fully-connected layer is configured, this architecture requires the output length to be fixed upfront. Such requirement entails that if we need to increase the forecasting horizon, then we have to re-train a different model from scratch. These settings do not allow for fully leveraging on the recurrent nature of LSTMs that, in general, can be trained and perform inference on sequences of variable length. On top of that, the complexity of the model, in terms of learnable parameters, grows proportionally to the output length. This aspect is crucial not only from a scalability perspective but also for the quality of the inference. The more the parameters, the higher the risk of over-fitting the training data-set and performing poorly on unforeseen inputs. For these reasons, we decided to build upon this first attempt and devise other architectures that overcome in part the aforementioned limitations.

5.3.1.2 *Sequence-to-dense and Dense-to-dense*

The model described in Section 5.3.1.1 imposes the limitation of deciding the length of the output sequence (i.e., the number of time steps to forecast in a test scenario) upfront. To address such issue, we have introduced a change to the inference process that consists of using *smaller* forecasted sequences as inputs to the model. This way, it is possible to forecast an unlimited number of time-steps while using a model with a fixed output size. However, due to its *closed-loop* nature, the revised inference process has an important drawback: it is possible to use only the historical values of the target variable to estimate its evolution. In other words, the architectures leveraging on this method can only be used in uni-variate mode (i.e., without auxiliary variables).

In particular, we devised two architectures that employ this method at their core. The first one, called *sequence-to-dense* (seq2den), has a structure similar to the baseline described in Section 5.3.1.1. It consists of an LSTM layer followed by a fully-connected layer. Since the LSTM layer weights are shared between time-steps, the input length does not affect the number of model parameters. The second one, called *dense-to-dense* (den2den), consists of two fully-connected layers in series. When feeding the input time-series to the input layer, each input time-step has an associated weight that is independently optimized during back-propagation. In this case, the length of the input sequence contributes to the growth of the learnable parameters. Moreover, due to the output layer being fully-connected, for both these variants the number of learnable parameters is proportional to the output sequence length.

5.3.1.3 *Sequence-to-sequence with Time Embedding*

The models described in Section 5.3.1.2 improve our baseline by introducing a *closed-loop* inference strategy. However, they are not able to work in multivariate settings and, thus, we cannot leverage on contextual information to get more accurate forecasts. With the aim of taking the best from both worlds, we developed an additional model, called *sequence-to-sequence* (seq2seq), that is able to accept auxiliary variables while not being constrained in the length of the output. Additionally, as the time-series under consideration turn out to be greatly affected by timing information such as the hour of the day and the week day, including non-periodic changes occurring on holidays, we added to this model the capability to use additional *time embedding* metrics, as described below.

TEMPORAL EMBEDDING. Each sample of both input and target signals, at each time-step t, is associated with a unique date and time. This information can be encoded using additional time-series:

1. $\{\alpha_t\}$, for $\alpha_t \in \{0, 1, \dots, 23\}$, encoding the hours;
2. $\{\beta_t\}$, for $\beta_t \in \{0, 1, \dots, 6\}$, encoding the week-days;
3. $\{\gamma_t\}$, for $\gamma_t \in \{0, 1, \dots, 11\}$, encoding the months;
4. $\{\delta_t\}$, for $\delta_t \in \{0, 1\}$, encoding whether the time-steps correspond to holidays.

Note that such time-series are completely known a-priori. However, $\{\alpha_t\}$, $\{\beta_t\}$ and $\{\gamma_t\}$ cannot be fed directly to a neural network, because their values fail to encode the cyclic nature of the series (e.g., that 23:00 and 01:00 are at the same distance from 00:00). Therefore, we need to embed them properly into a vector space. A simple *one-hot* encoding (i.e., one boolean per unique value) would produce an excessive number of variables, and it would lose again the relative temporal distance among the values. To this aim, we propose a *circular* 2D embedding that can preserve such information by mapping each original k-values sequence to the 2D coordinates $\pi_k(i)$ of the vertices of a regular k-sides normalized polygon:

$$\pi_k(i) = \left(\sin\left(\frac{2\pi i}{k}\right), \cos\left(\frac{2\pi i}{k}\right)\right)^T \tag{5.3}$$

Using such an embedding we are able to produce an additional *pilot* time-series by concatenating the desired embedded temporal features. For instance, for hourly timestamps, consider

$$p_t = (\pi_{24}(\alpha_t), \pi_7(\beta_t), \pi_{12}(\gamma_t), \delta_t)^T \in \mathbb{R}^p \tag{5.4}$$

where $p = 7$.

PILOTED SEQUENCE-TO-SEQUENCE MODEL. The neural network model discussed in this section could be thought of as a modified version of a sequence-to-sequence model [226] adapted to accept an external variable, the *pilot* sequence, whose dynamics is known *a-priori*. It can be described as

$$\begin{aligned} &\mathcal{E}_\theta : \mathbb{R}^{(m+p+1)} \times \mathbb{R}^{2d} \to \mathbb{R}^{2d} \\ &\mathcal{D}_\theta : \mathbb{R}^p \times \mathbb{R}^{2d} \to \mathbb{R}^d \times \mathbb{R}^{2d} \\ &\mathcal{R}_\theta : \mathbb{R}^d \to \mathbb{R} \end{aligned} \tag{5.5}$$

where $\mathcal{E}_\theta$ and $\mathcal{D}_\theta$ are two LSTM layers with d units and $\mathcal{R}_\theta$ is a fully-connected layer acting as a *rectifier* (that reshape the result to the desired dimensions).

Let $(x_{t-\lambda+1}, \dots, x_t)$ be the input sequence, $(y_{t-\lambda+1}, \dots, y_t)$ the auxiliary sequence, and $(p_{t-\lambda+1}, \dots, p_t, \dots, p_{t+\varphi})$ the pilot sequence. The output $(\hat{x}_{t+1}, \dots, \hat{x}_{t+\varphi})$ is generated as follows:

$$\begin{aligned} &\tilde{h}_{t-\lambda} = 0, && \tilde{h}_{t-\lambda} \in \mathbb{R}^{2d} \\ &z_s = \mathrm{vec}(x_s, y_s, p_s), && t-\lambda < s \le t \\ &\tilde{h}_s = \mathcal{E}_\theta(z_s, \tilde{h}_{s-1}), && t-\lambda < s \le t \\ &(o_s, \tilde{h}_s) = \mathcal{D}_\theta(p_s, \tilde{h}_{s-1}), && t < s \le t+\varphi \\ &\hat{x}_s = \mathcal{R}_\theta(o_s), && t < s \le t+\varphi \end{aligned} \tag{5.6}$$

where $\tilde{h}_s$ are the latent states. Note that the number of learnable parameters depends only on the feature dimensions, not on the length of input and output sequences (λ and φ).

TREND-SEASONALITY DECOMPOSITION. Time-series decomposition techniques can be effectively used to improve forecasting accuracy [229]. In this case, we opted for *additive* decomposition, by decomposing time-series $\{x_t\}$ in its *trend* and *seasonality*, such that $x_t = b_t + s_t$. By assuming such a decomposition, our aim is to devise a decomposable model that produces forecasts by aggregating the contributions of two different models for trend and seasonal components. Without loss of generality, let us consider a simplified case in which there are no auxiliary variables. The workflow to train a decomposable model, and to ultimately forecast the target time-series, consists of:

1. **Fitting the trend model.** Fit a logistic function
$$a + \frac{b}{1 + e^{-ct}} \tag{5.7}$$
over $\{x_t\}$, using the Levenberg-Marquardt optimization technique [130], to obtain $\{b_t\}$ for $t \in [0, T_{\texttt{train}}]$.

2. **Data detrendization.** Remove the trend component from the data:
$$s_t := x_t - b_t \tag{5.8}$$

3. **Fitting seasonality model.** Train the neural architecture over $\{s_t\}$, for $t \in [0, T_{\texttt{train}}]$.

4. **Seasonality forecasting.** Use the neural model to infer a forecast $\{\hat{s}_t\}$ for $t \in [T_{\texttt{train}} + 1, T_{\texttt{test}}]$.

5. **Trend forecasting.** Use the fitted trend function to infer the trend $\hat{b}_t$ for $t \in [T_{\texttt{train}} + 1, T_{\texttt{test}}]$.

6. **Aggregate contributions.** Compute
$$\hat{x}_t := \hat{b}_t + \hat{s}_t \tag{5.9}$$
for $t \in [T_{\texttt{train}} + 1, T_{\texttt{test}}]$.

If needed, in steps 4 and 5, temporal embedding can be added to train the seasonality and to forecast the target time-series.

5.4 EXPERIMENTS

In this section, we report experimental results from the application of the techniques described in Sections 2.2.2 to 2.2.4, 2.2.6 and 5.3 on data provided by Vodafone. We compared the different techniques according to three main aspects: *accuracy*, *training time* and *stability*. In particular, stability refers to the capability of a model to produce similar results under different weights initial values and/or small variations of the hyperparameters.

5.4.1 *Experimental Set-up*

We focused our experimental evaluation on data coming from two distinct VNFs, namely *CSCF* and *DRA*. Both datasets report samples recorded with hourly granularity, but we also re-sampled the DRA dataset to get a daily-aggregated (average) version. The metrics under analysis was chosen by Vodafone due to their relevance in the monthly monitoring and reporting activities performed by the NFV capacity team. Due to confidentiality reasons, such datasets have not been made available to the public.

The CSCF dataset spans a time range of 16 months, with hourly frequency. It reports the dynamics of the `cpu|usage_average` system-level metric, averaged among the VMs composing the VNF, and of four application-level metrics:

- `SCSCF_REGISTERED_USERS`
- `SCSCF_SUCCESSFUL_INIT_REGIST`
- `SCSCF_RE_REGISTRATION_ATTEMPTS`
- `UNREGISTERED_IMPI_ON_SCSCF`

Such metrics exhibit a low Pearson correlation [15]. We used the last 30 days as test set, to evaluate the trained models. The reminder is divided in training and validation splits, by taking the first 90% and the last 10%, respectively. The goal is to forecast `cpu|usage_average`.

The DRA dataset includes hourly- (DRAh) and daily-aggregated (DRAd) time-series spanning a time range of 16 months. The datasets report the dynamics of the `cpu|usagemhz_average` system-level metric, averaged among the VMs composing the VNF, and of two application-level metrics:

- `DRA-DIAM-MSG.0.Max_TPS`
- `DRA-DIAM-INT.0-S6a/S6d.Res_Sent`

Such metrics exhibit a low Pearson correlation. We used the last 6 months as test set, to evaluate the trained models. The remainder is divided in training and validation splits, by taking the first 90% and the last 10%, respectively. The goal is to forecast `cpu|usagemhz_-average`.

Experiments involving neural architectures were carried out on a GCP VM, equipped with: an Intel Xeon processor (24 virtual CPU cores, 2 GHz); 120 GB of RAM; an NVidia Tesla V100 GPU (16 GB of dedicated memory, CUDA 10.0); Debian 9.9 operating system. Experiments involving classical forecasting techniques were carried out on an on-premise test-bed, equipped with: an AMD Ryzen 7 2700x processor (16 virtual CPU cores, 3.7 GHz); 64 GB of RAM; Ubuntu 18.04 operating system. Both environments were configured with Python 3.7.7, *TensorFlow* 1.14.0, *statsmodels* 0.12.0, *NumPy* 1.18.5.

Table 5.1: Neural architecture configurations for each dataset.

	CSCF	DRAh	DRAd
λ	24, 168, 720	24, 168, 720	7, 30
φ	24, 168, 720	24, 168, 720	7, 30, 182
d	25, 50,. . . , 150	25, 50,. . . , 150	25, 50,. . . , 150
b	32, 512	64, 512	16
$\mathcal{T}$	false	true	true

Table 5.2: HW configurations space for each dataset.

	CSCF	DRAh	DRAd
m	24, 48,. . . , 168	24, 48,. . . , 168	3, 7, 14, 21, 28
α	0.0, 0.1,. . . , 0.5	0.0, 0.1,. . . , 0.5	0.0, 0.1,. . . , 0.5
β	0.0, 0.1,. . . , 0.5	0.0, 0.1,. . . , 0.5	0.0, 0.1,. . . , 0.5
γ	0.0, 0.1,. . . , 0.5	0.0, 0.1,. . . , 0.5	0.0, 0.1,. . . , 0.5
ϕ	0.1, 0.2, 0.3, 0.4, 1	0.1, 0.2, 0.3, 0.4, 1	0.1, 0.2, 0.3, 0.4, 1
b_t	`add`, `mul`	`add`, `mul`	`add`, `mul`
s_t	`add`, `mul`	`add`, `mul`	`add`, `mul`

5.4.2 *Presentation of Results*

Our goal was to compare the performance of the models when provided with the same data. Therefore, we trained each model with each of the 3 datasets described above. For our evaluation, we considered the following KPIs:

- RMSE of the forecast $\{\hat{x}_t\}$ with respect to the ground truth $\{x_t\}$, for $t \in [T_{\texttt{train}} + 1, T_{\texttt{test}}]$. RMSE is defined as the square root of the MSE:

$$\sqrt{\frac{1}{T_{\texttt{test}} - T_{\texttt{train}}} \cdot \sum_t (x_t - \hat{x}_t)^2} \tag{5.10}$$

- MAPE of the forecast $\{\hat{x}_t\}$ with respect to the ground truth $\{x_t\}$, for $t \in [T_{\texttt{train}} + 1, T_{\texttt{test}}]$. MAPE is defined as the percentage of the average forecasting error:

$$\frac{100}{T_{\texttt{test}} - T_{\texttt{train}}} \cdot \sum_t \left| \frac{x_t - \hat{x}_t}{x_t} \right| \tag{5.11}$$

- Training time (TT), i.e., the time elapsed between the start of the first epoch the end of the last epoch of training, expressed in seconds.

Table 5.3: SARIMA configurations space for each dataset.

	CSCF	DRAh	DRAd
p	1, 2,. . . , 5	1, 2,. . . , 5	1, 2,. . . , 5
d	0	0	0, 1
q	0, 1, 2	0, 1, 2	0, 1, 2
P	0, 1,. . . , 7	0, 1,. . . , 7	0, 1,. . . , 7
D	0	0	0, 1
Q	0, 1, 2	0, 1, 2	0, 1, 2
m	24	24	7

Each model was tested under multiple different configurations of its parameters. If the training algorithm of the model involved a *random* weights initialization, each configuration was repeated for 10 independent runs. The performance indicators were then computed by averaging the independent scores. For neural architectures, a model configuration consists of the following hyperparameters:

- λ: the length of the input sequence
- φ: the length of the output sequence
- d: the number of units of the LSTM cell
- b: the batch size
- $\mathcal{T}$: whether the trend decomposition was used.

For the *baseline* architecture, φ corresponds to the length of the forecast (i.e., $T_{\texttt{test}} - T_{\texttt{train}}$). Also, note that trend decomposition is only available for the *seq2seq* architecture. The configurations used for the neural architectures are summarized in Table 5.1. Likewise, the configurations used for HW and SARIMA are summarized in Table 5.2 and Table 5.3, respectively.

In the remainder of this section, results will be presented by means of tables, box-plots and line-plots. For each model, tables report the mean scores of the most accurate configuration (i.e., the one with the lowest average MAPE). In other words, tables show the *lower-bounds*, in terms of accuracy, for each model. Box-plots allow for visualizing the *average* accuracy of a model: the shorter the box, the lower the variance in the accuracy; the lower the box is positioned along the y-axis, the higher the average accuracy of the model. Combined with tables, box-plots provide insights on models stability. To ease the visualization, box-plots are grouped by λ. For each value of λ, the boxes show how the model accuracy changes when the remaining hyperparameters (i.e., φ, b and d) are tuned. Note that box-plot whiskers are set such that no value is excluded from the visualization. While running such a huge amount of configurations, we noticed that the choice of λ may deeply impact on the training time of the model. Line-plots allow for visualizing how big the impact of λ is for the proposed architectures. For each value of λ, the solid line is an estimate of the central tendency, computed as φ, b and d change, while the stripe width indicates the confidence interval.

5.4.3 *Neural Architectures*

5.4.3.1 *CSCF Dataset*

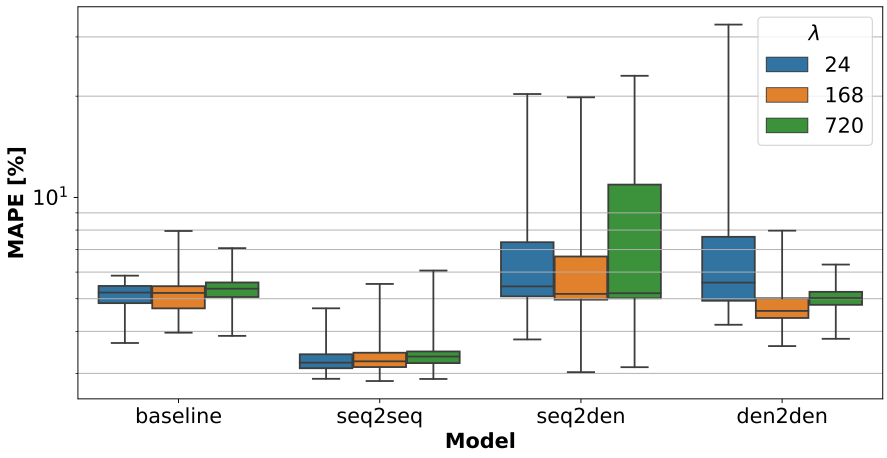
The CSCF dataset contains a stationary time-series with a strong seasonality component. Table 5.4 highlights that the seq2seq architecture outperforms the others in terms of accuracy, but the den2den has an interestingly reduced training time by an order of magnitude, at the cost of raising the accuracy from 3% to 4.2%. Figure 5.2a also shows that, in general, seq2seq provides more stable results, as 50% of the observations are densely concentrated around the average and the whiskers are relatively short (consider also the log-scale on the Y-axis). This entails that the model is very robust with respect to the hyperparameters variations. As expected, Figure 5.2b shows that the training time grows as the sequence gets longer for all models but den2den.

5.4.3.2 *DRAh Dataset*

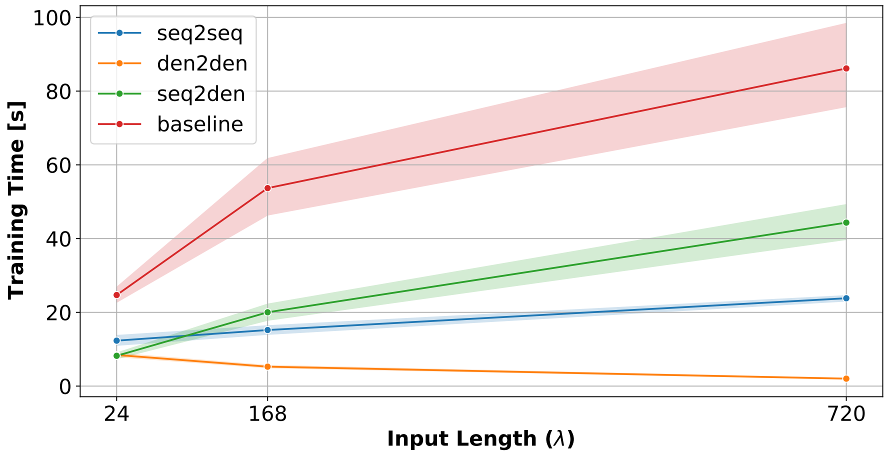
Table 5.5 shows that the seq2seq architecture achieves the best performance for the DRAh dataset. From Figure 5.3a, we can see that, in general, its accuracy is more stable with respect to variations of the hyperparameters, compared to the other models. As expected, Figure 5.3b shows that the training time grows as the sequence gets longer for all models but den2den. When applied on this dataset, with such a huge forecasting range (i.e., $T_{\texttt{test}} - T_{\texttt{train}} = 4368$), the baseline model turns out to have a decoder (i.e., a dense layer) composed by a 4368-rows weight matrix, as it is designed to provide the output sequence in one-shot at the decoding stage. In this case, we were expecting an explosion of the training time, which does not happen. This is probably due to the nice way the computations of the decoder can be parallelized on the underlying GPU. Indeed, the other architectures do not fully exploit the GPU acceleration, either because they are made entirely of recurrent units or because of their *closed-loop* nature. However, defining the root-cause of such an unexpected performance boost requires a deeper investigation.

Table 5.4: Average KPIs (± standard deviation) for the configuration with the best average MAPE (CSCF dataset, 10 repetitions).

	MAPE [%]	RMSE	TT [s]
baseline	4.39 ±0.13	1.17 ±0.03	30.23 ±4.09
den2den	4.18 ±0.09	1.22 ±0.02	2.44 ±0.25
seq2den	4.27 ±0.05	1.30 ±0.01	98.11 ±0.87
seq2seq	3.04 ±0.06	0.95 ±0.02	23.39 ±2.38



(a) MAPE distributions per input sequence length.

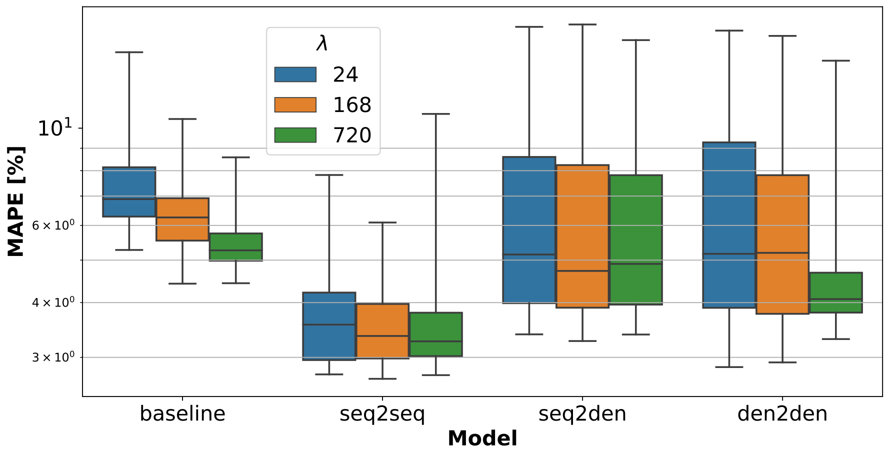


(b) Relation between training time and input sequence length.

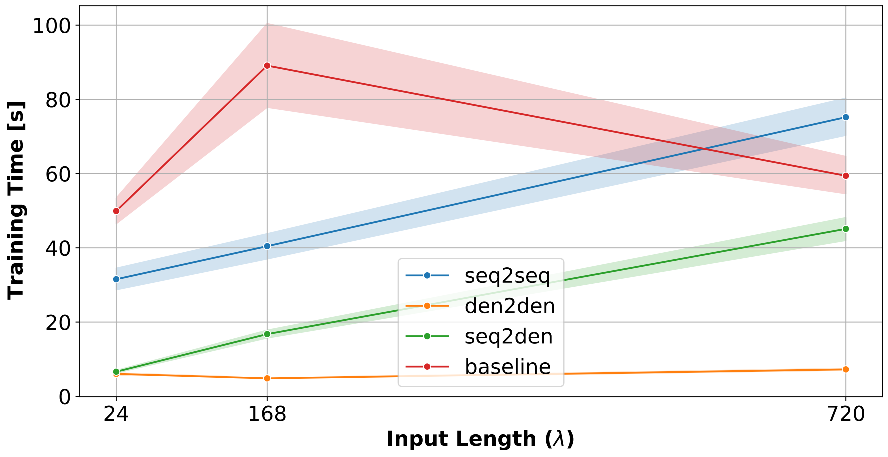
Figure 5.2: Performance measures for neural architectures applied to the CSCF hourly dataset.

Table 5.5: Average KPIs (± standard deviation) for the configuration with the best average MAPE (DRAh dataset, 10 repetitions).

	MAPE [%]	RMSE	TT [s]
baseline	5.01 ±0.25	55.69 ±3.06	40.37 ±15.86
den2den	3.38 ±0.12	41.19 ±1.19	3.12 ±0.69
seq2den	3.72 ±0.19	42.37 ±1.82	4.93 ±6.14
seq2seq	2.87 ±0.05	38.43 ±0.62	35.60 ±8.14



(a) MAPE distributions per input sequence length.



(b) Relation between training time and input sequence length.

Figure 5.3: Performance measures for neural architectures applied to the DRAh hourly dataset.

5.4.3.3 *DRAd Dataset*

Table 5.6 shows that the den2den model achieves the best performance for the DRAd dataset. However, from Figure 5.4a, we can see that the variability of the accuracy for this architecture strongly depends on the choice of the hyperparameters. As can be seen in Figure 5.4b, due to the small number of timestamps, the differences in terms of training time are not relevant for this dataset.

5.4.4 *Classical Forecasting Techniques*

Table 5.7 summarizes the best results achieved by the classical forecasting techniques, described in Sections 2.2.2 and 2.2.3. Regarding the accuracy, Figure 5.5a shows that HW outperforms SARIMA, for all the tested dataset. Looking at Figure 5.5b, We can draw a similar conclusion for what concerns the training time. In particular, for the CSCF and DRAh datasets, SARIMA generally requires more than 100 seconds per run, while HW less than a second.

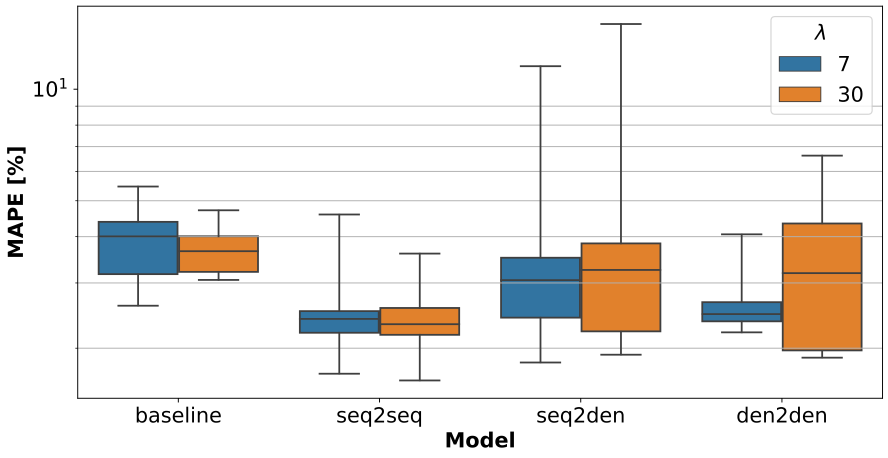
5.4.5 *Comparative Analysis*

The presented results highlight what trade-offs can be achieved between accuracy and training time with the various techniques. Among the neural architectures, in general, the *seq2seq* outperforms the others in terms of accuracy and stability (see Figures 5.2a, 5.3a and 5.4a). Such level of accuracy is achieved thanks to the time embedding and the additive decomposition. In particular, the highest stability is due to using recurrent layers in both encoding and decoding phases, that reduces the impact of the hyperparameters λ and φ. However, in general, seq2seq training time grows proportionally to the length of the input and output sequences, as the recurrent layers process samples sequentially (see Figures 5.2b and 5.3b). Note that this process cannot be improved even using GPU acceleration, as it is not possible to parallelize the computation. On the contrary, the *den2den* model makes a much better use of GPU acceleration and results to be the fastest neural model.

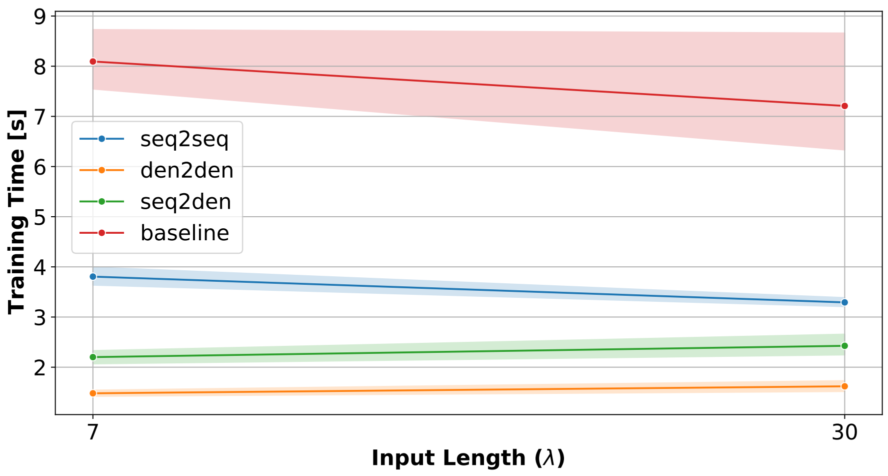
For what concerns classical forecasting techniques, there are cases such that they match neural architectures in terms of accuracy. For instance, comparing Tables 5.4 to 5.6 with Table 5.7, we can see HW *best* runs consistently scoring a—slightly—lower MAPE than seq2seq, for CSCF (2.85% vs 3.04%), DRAh (1.34% vs 2.06%) and DRAd (2.49% vs 2.87%) datasets. However, such level of performance is most likely due to a particularly *lucky shot*, in terms of hyperparameters tuning, rather than an evidence of its superior capacity at modeling complex time-series.

Table 5.6: Average KPIs (± standard deviation) for the configuration with the best average MAPE (DRAd dataset, 10 repetitions).

	MAPE [%]	RMSE	TT [s]
baseline	3.54 ±0.36	38.80 ±3.18	6.12 ±0.75
den2den	1.94 ±0.03	23.29 ±0.25	1.35 ±0.07
seq2den	2.12 ±0.07	24.72 ±0.69	1.67 ±0.12
seq2seq	2.06 ±0.07	26.26 ±0.69	3.29 ±0.46



(a) MAPE distributions per input sequence length.

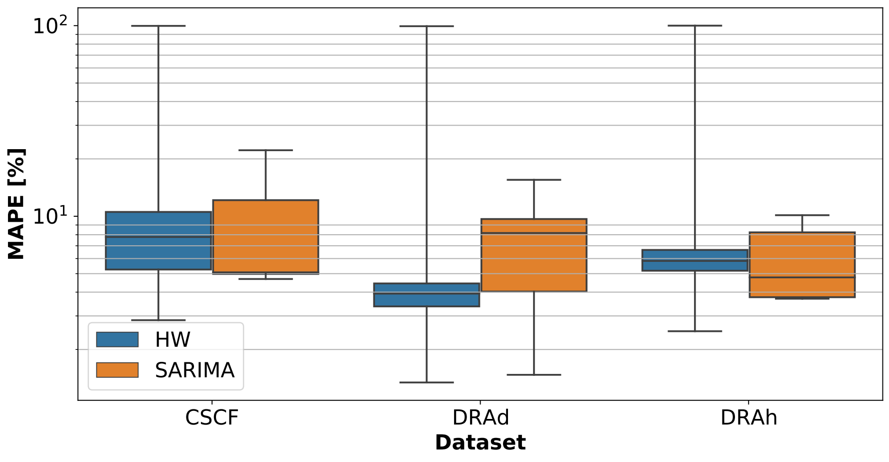


(b) Relation between training time and input sequence length.

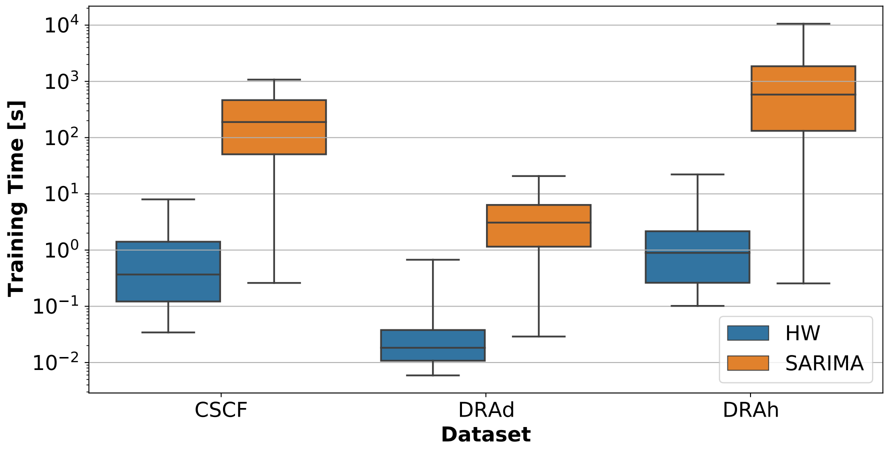
Figure 5.4: Performance measures for neural architectures applied to the DRAd daily dataset.

Table 5.7: KPIs for the configuration with the best MAPE.

		MAPE [%]	RMSE	TT [s]
Dataset	Model			
CSCF	HW	2.85	1.00	2.14
	SARIMA	4.69	1.41	832.46
DRAd	HW	1.34	17.09	0.01
	SARIMA	1.47	18.34	12.86
DRAh	HW	2.49	35.07	0.22
	SARIMA	3.69	42.08	154.43



(a) MAPE distributions.



(b) Training time distributions.

Figure 5.5: Performance measures for classical forecasting techniques.

The low stability shown by HW seems to support this hypothesis. Without any other statistical assumption (e.g., similarity of the distribution with other times-series or different time-ranges), the results reported in Section 5.4.4 suggest that HW is less likely to achieve the same accuracy of the seq2seq architecture for the tested datasets. Figure 5.5a shows that the choice of the hyperparameters has a strong impact on the performance of HW. In fact, for all the datasets, the worst HW runs score a MAPE that is very close to 100%. This is not the case for seq2seq, whose overall worst performance is around 10% (see Figure 5.3a). On hourly datasets (i.e., CSCF and DRAh), despite the best HW run achieves a MAPE lower than 3%, the first quartile is greater than 5% (see Figure 5.5a). Instead, for the seq2seq architecture, the third quartile is lower than 4% (see Figures 5.2a and 5.3a). In other words, 75% of HW runs score a MAPE greater than 5%, while 75% of seq2seq runs score a MAPE lower than 4%. On the DRAd dataset, despite the best HW run achieves a MAPE lower than 2%, the first quartile is greater than 3% (see Figure 5.5a). In this case, the seq2seq architecture exhibits a third quartile strictly lower than 3% (see Figure 5.4a). In other words, 75% of HW runs score a MAPE greater than 3%, while 75% of seq2seq runs score a MAPE lower than 3%.

5.5 CONCLUSIONS

We tackled the problem of forecasting the future evolution of metrics in an NFV infrastructure, and experimentally compared a number of techniques for time-series forecasting, using a real data-set from the production NFV infrastructure of the Vodafone network operator. Overall, our *seq2seq* neural architecture exhibited the best performance in terms of prediction accuracy. Indeed, even though the best HW runs scored a slightly lower MAPE, *seq2seq* offers a significantly greater stability than the considered classical statistical models, that are strongly impacted by the hyperparameters choice, at the cost of longer training times.

6

PREDICTIVE AUTO-SCALING

6.1 INTRODUCTION

Over the last decade, ICTs have been evolving non-stop, at an extremely rapid pace. The ever-growing availability of low-cost high-bandwidth connectivity has been one of the key enablers paving the way for the impressive growth in the adoption of distributed computing paradigms. Cloud computing [25] emerged as the de-facto standard for developing and deploying large-scale production-grade applications. This paradigm allows for completely decoupling the management of physical infrastructures from the services deployed on top of them, by heavily relying on virtualization. This enabled to make a more efficient use of physical resources, and to have a higher resiliency degree for the hosted applications. However, cloud computing has significantly evolved, and it is not only limited anymore to the IaaS provisioning model, according to which users can access compute instances (e.g., VMs) deployed on top of shared physical servers, operated by the provider. Nowadays, the so-called XaaS provisioning model enables an application developer to realize *cloud-native* services, by leveraging on a wide range of orchestration, load-balancing, storage and monitoring solutions, completely managed by the provider [23].

To operate their infrastructures 24/7, cloud providers need operation teams ready to promptly address and fix any kind of issue that might occur, including hardware faults and software defects. In production-grade cloud infrastructures, this is only feasible when such systems are designed following well-established practices (e.g., fault-independent zones; redundant powering and cooling infrastructures; multi-path networking topologies; etc.) and operated using appropriate tools (e.g., monitoring systems; resource managers; effective automation rules; etc.).

In this regard, a key enabling factor is the presence of fine-grained monitoring services, on top of which automation rules can be built, ensuring high reliability for the hosted services, and performance levels that are as stable as possible, despite sudden changes in traffic conditions. This refers to *elasticity*, that is the capability of cloud services to automatically adapt their set of allocated resources (e.g., VMs, containers, or even physical nodes) as the workload changes over

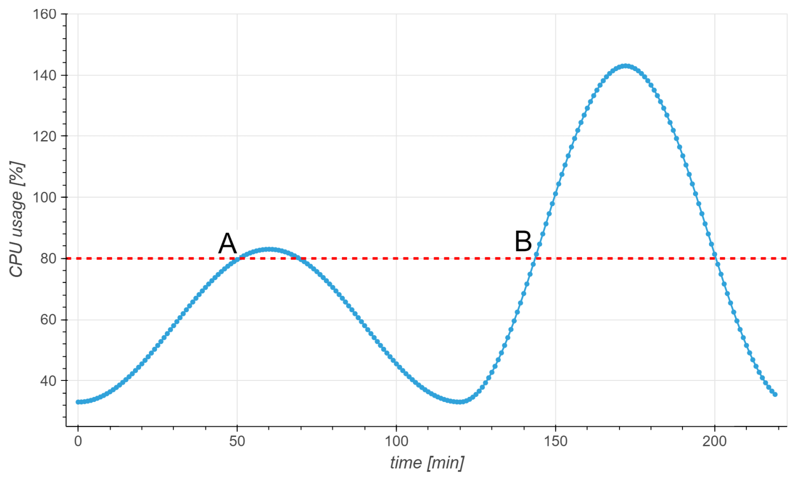


Figure 6.1: Example of load profile.

time. Elasticity is typically implemented by means of a control loop that decides which actions to take in order to keep the service running smoothly (e.g., scale-out or scale-in). Such scaling decisions are usually made on the basis of system-level resource consumption metrics (e.g., CPU utilization, network traffic, storage load), as well as KPIs at application-level (e.g., response times, connection timeouts/errors).

6.1.1 *Problem Presentation*

Classical auto-scaling mechanisms are inherently based on *reactive* automation rules that scale a service whenever some metric breaches predefined thresholds. Traditionally, after a scaling action is actually triggered, an elastic system enters a *cooldown* period that prevents further scaling actions until it expires. This is done such that the elasticity controller can take its subsequent decisions by actually considering the effects of the previous one. In addition, to make the overall mechanism more robust with respect to transient changes in the workload, it is common to require the threshold to be breached for a few consecutive observations, before triggering any action.

Developing and tuning such automation rules becomes particularly cumbersome when dealing with large-scale production environments. There are many challenges to be addressed, like: determining the KPIs to accurately estimate the status of the system; setting the frequency at which the scaling decision should be evaluated; adapting the scaling policy to changes in the workload, to prevent unnecessary scaling actions; estimating the amount and type of instances to add, to handle the new conditions; taking into account non-negligible times to set up the additional instances. For instance, in this work we put great emphasis on the latter challenge because, in large-scale production environment, spawning new instances might indeed take from a few minutes to even half an hour [197]. This is not only the time needed to instantiate and boot a new VM, but also the time needed to: configure the new instance; possibly installing any missing software, in case the same image is re-used as a base for a number of different roles requiring customized software set-ups, or just installing some minimally required security updates; sometimes copying onto the image a minimum set of information or local database needed for the software to operate correctly; starting the actual service and registering it into a load-balancing group; and finally some further time is needed for the new instance to progressively pick new traffic. Therefore, in such settings, anticipating scale-out operations becomes critical. Despite the mentioned precautions, traditional control loops are still inherently *"dumb"*, as they do not factor the rich dynamic of the observed metrics in their decisions. For instance, consider the CPU utilization evolution depicted in Figure 6.1. A classical scaling policy would treat scenarios A and B in pretty much the same way: as soon as the CPU utilization breaches the threshold (i.e., the red line), a scaling action is triggered. However, a human operator, based on their prior experience gained while operating such a fictional service, could easily distinguish between scenarios A and B. While A might be safely ignored, it is clear that B would require urgent actions to be taken.

In the context of large-scale cloud environments, given the complex relationships among their components, and the abundance of operational data they generate (e.g., event logs, application metrics, source code, etc.), treating operations as a *data science* problem [244] seems a promising approach to develop *"intelligent"* automations. In particular, time-series forecasting techniques based on ML may play a fundamental role in enhancing the capabilities of elasticity controllers to prevent services from saturating their capacity. For instance, AWS currently provides native support for predictive scaling with EC2 [93], demonstrating the suitability of this type of approaches at supporting cloud operations [214]. On a related note, ML-based approaches have also been shown to be beneficial for efficient and sustainable management of resources in cloud data centers [235].

6.1.2 *Contributions*

In this work, we propose an open-source software architecture for integrating predictive analytics within an OpenStack cloud platform. Our work provides three major contributions. First, *(i)* a general architecture for performing *predictive operations* on a cloud infrastructure, based on time-series forecasting techniques. Second, *(ii)* an open-source implementation of the forecasting component within OpenStack, leveraging on Monasca [170], that automatically computes forecasts and makes them available as additional metrics. Our implementation also includes a few reference implementations of metric predictors—i.e., LR [88], ARIMA [19], MLP and RNN—showing that the proposed architecture is flexible, as it allows for easy customization. Third, *(iii)* an extensive experimental validation of our architecture, using both synthetic and real CDN workload traces, where we set up a synthetic elastic application, exploiting the native capabilities of OpenStack, and compare the performance of several predictive elasticity controllers based on the aforementioned reference predictors. Although there are two types of scaling (i.e., *horizontal* and *vertical*), our experimentation focuses on horizontal scaling only. However, the proposed architecture may easily be leveraged for vertical scaling [236] as well, as our approach is agnostic with respect to how the actual scaling operations are implemented.

6.1.3 *Chapter Organization*

This chapter is organized as follows. Section 6.2 provides a detailed overview of the related research literature, highlighting how the proposed technique is positioned in the current landscape. Section 6.3 describes the proposed approach. Section 6.4 presents its experimental validation on an OpenStack deployment. Some final remarks are enclosed in Section 6.5, along with the discussion of possible ideas for future works on the topic.

6.2 RELATED WORK

In the research literature, a number of authors applied data-driven techniques to automated elasticity control, both for public and private cloud. In what follows, we start by reporting key research works dealing with predictive elasticity based on metric forecasting for public and private clouds. Then, we provide a brief review of elasticity-control solutions based on RL.

6.2.1 *Predictive elasticity control in cloud computing*

A variety of data-driven techniques have been proposed to provide accurate short-term predictions of workloads and resource consumption patterns of elastic clusters, to achieve a more timely and fine-tuned allocation of resources.

In [197], the authors describe a simple scaling strategy based on predicting the aggregate sum of transmitted and received bytes of a service cluster, considering resource setup delays and limited deployment throughput. The approach leverages on a workload model to estimate a percentile of the resource demand, and a probabilistic function that describes the cost of over-/under-provisioning the cluster. The authors present promising results by evaluating the technique against data from more than 40K real services deployed as auto-scaling groups on AWS.

In [202], the authors propose a model-predictive control-based approach [1], combining three major techniques: a 2nd-order ARMA filter for workload prediction; a CBMG [3], optimized on web logs, capturing the behavior of users while browsing a web application; a look-ahead optimization to trade-off between the advantages arising from dynamic elasticity and the cost of scaling decisions and cluster reconfiguration at each control period. They empirically evaluate their technique against data from the 1998 world-cup web-site traffic.

In [97], the authors tackle the problem of non-instantaneous instance provisioning when using elastic scaling in cloud environments. They propose a predictive strategy based on a resource prediction model using ANNs and LR. The method was applied on an e-commerce application scenario emulated through the TPC-W [232] workload generator and benchmarking application, deployed on AWS EC2. ANNs improve the accuracy by reducing the MAPE by roughly 50%, compared to LR.

In [18], an ANN with a single hidden layer is proposed for predicting the resource utilization and duration of continuous integration tasks, for several repositories from the Travis open data. However, the evaluation focused on predicting the task duration only, using a per-repository model using the number of files and the repository size as inputs. Results show an accuracy at least 20% and up to 89% better than a baseline LR.

In [95], authors propose supervised learning methods to tackle the problem of predictive auto-scaling for multi-tier elastic applications, considering unstable performance of individual VMs. In particular, LR is applied to the traffic arrival rate time-series to predict short-term arrival rates, that were in turn used to predict the evolution of the response times, using PR. Such estimates were then fed to an RDF, designed to learn a configurations map associating the order of configurations—required to maintain the SLO—to the experienced request arrival rates and system response times. The training data for the RDF was generated by executing a few (static) system scaling policies.

In the *RScale* framework [105], GP regression, was used to predict end-to-end tail-latency of distributed microservices workflows with generic direct acyclic graph-like topologies. *RScale* was evaluated on a NSF Chameleon test-bed, and achieved similar accuracy, but a smaller predicted uncertainty, with respect to ANNs. However, it exhibited reduced inference overheads and superior adaptability to dynamically changing workload/interference conditions.

In [13], BNs are used in a predictive framework to support automatic scaling decisions in cloud services. The method was evaluated on synthetic applications with exponentially distributed duration and workload inter-arrival patterns.

In [228], DTs are used to predict CPU, memory and network usage of Hive-based MapReduce queries over a Hadoop cluster. The authors use a 4-machines cluster to perform queries with different structures over a number of different data sets, using a per-resource decision tree to classify the query within the high or low resource-usage class. The presented results give insights as to the parameters mostly affecting the consumption level for each resource. However, as the authors used a fixed-size cluster, the technique does not seem to be useful in the context of elasticity control, albeit the investigation may be useful to design effective elasticity rules.

In [103], authors propose a proactive auto-scaling mechanism for edge computing applications on Kubernetes. The approach leverages on ARMA and LSTM to estimate the raw number of additional compute instances needed, given the observed resource utilization patterns. The authors also provide a mechanism to either automatically retrain from scratch or incrementally update the underlying model.

In [45], the authors introduce an extensive set of traces exported from Azure's internal infrastructure. They propose *Resource Central*, an approach that collects VM utilization metrics and periodically train prediction models on them offline. Such models can then be queried online by resource management systems and/or human operators. While the approach is in theory agnostic to the underlying models, the authors considers RDF, GBT and FFT for their experiments. The authors validate their approach by integrating it with Azure's VM scheduler, showing performance improvements also in over-subscription scenarios.

In [120], the authors propose a framework to forecast the workload of a cloud system, such that a resource manager can take informed scaling decisions. Their approach is based on SDL, that consists in including recent forecast errors in the input to the underlying model, such that it can be used as feedback to improve the accuracy of future predictions. The model is a feed-forward ANN, whose weights are optimized via an improved version of the *blackhole* algorithm proposed by authors. The authors validate their approach against 6 different datasets exported from real systems.

In [27], the authors propose a proactive resource scaling approach that leverages on a workload prediction module. The underlying forecasting model is based on ARIMA. The approach uses the predicted information to resize a cloud application accordingly, e.g., anticipating peaks. The authors use real traces exported from web servers of Wikipedia to train ARIMA to predict request patterns. They also validate their approach, in terms of impact on the QoS of a cloud application, by running simulations on *CloudSim*.

In [112], the authors propose *CloudInsight*, a workload prediction framework that can be used to proactively scale cloud applications. The authors leverage on an *ensembling* approach (i.e., combining the outputs from several models) to effectively handle irregular, dynamically changing workloads. The weight of an individual model is continuously re-evaluated by an SVM-based mechanism, such that the system can adapt to the current shape of the workload. The authors validate their approach against 3 different classes of workloads (exported from real systems), and compare its performance to several baseline predictors.

In [101], the authors propose *LoadDynamics*. Similarly to [112], they put the emphasis on the sensitivity to workload changes that is observed in most workload prediction frameworks. Their solution is an LSTM-based approach that is trained and evaluated on data exported from real systems, that describe requests arrival rates in different application scenarios (e.g., public cloud, HPC, web, etc.).

Predictive analytics have also been investigated in private cloud scenarios, notably for NFV and SDN [144], in order to adapt and fine-tune the allocation of virtualized resources to the conditions of the network. In this way, operators can benefit from proactive automation mechanisms to ensure QoS for their cloud-native *service-chains*. Related works in this space are already presented in Section 5.2. There exist other approaches that perform dynamic resource allocation based on instantaneous monitoring, rather than on resource estimations. For instance, the authors of [163] propose a *vertical* elasticity management approach for containers, to dynamically adapt the allocated memory in Kubernetes, to support the co-location of containers having heterogeneous QoS requirements. However, for brevity, we omit this type of approaches from our overview, as they fall within the research literature on classical reactive elasticity control.

6.2.2 *Elasticity control with Reinforcement Learning*

Straightforward heuristics like *static* thresholding [30] can yield amazing results, when dealing with elasticity control for simple systems. However, thresholds require careful ad-hoc tuning, resulting in an approach that can hardly be adopted at scale (as it will eventually lead to over- or under-provisioning). Addressing these shortcomings, *dynamic* thresholding mechanisms, like the ones based on RL, offer the capability to automatically adapt thresholds to the current status of the system.

In [6], the authors propose an adaptive mechanism to automatically learn scaling policies for NFV, based on Q-learning and GP. They leverage on GP to iteratively improve the learned policy before taking the final scaling decision, using the average response time of the system as reward signal. They evaluate their approach on a simulated NFV environment, showing that it outperforms both a standard threshold-based policy and a Q-learning-based one (not based on GP).

Table 6.1: Related works comparison (legend: G.A. = generally applicable; S.O. = spawning overhead; E. = elasticity; O.S. = open-source).

Work	Techniques	Input	Validation	G.A.	S.O.	E.	O.S.
[197]	heuristic	network data	40K AWS auto-scaling groups	Y	Y	Y	N
[202]	ARMA, CBMG	utilization data	'98 world-cup	N	Y	Y	N
[97]	ANN, LR	utilization data	e-commerce (TPC-W)	Y	Y	Y	N
[18]	LR, MLP	utilization data	CI pipeline	N	N	Y	N
[95]	PR, RDF	request rate, response times	e-commerce (RUBiS)	N	N	Y	N
[105]	GP	utilization data	Robot Shop	Y	N	Y	N
[13]	BN	utilization data	synthetic workload	Y	N	Y	N
[228]	DT	query type and structure	DB queries (Hive)	N	N	N	N
[103]	ARMA, LSTM	utilization data	Kubernetes edge app	Y	N	Y	N
[45]	RDF, GBT, FFT	utilization data	real Azure VM traces	Y	Y	Y	N
[120]	MLP, SDL, blackhole	request rate, utilization data	6 different real datasets	Y	N	Y	N
[27]	ARIMA	request rate	real Wikipedia traces	Y	Y	Y	N
[112]	ensembling	request rate	3 classes of real workloads	Y	N	Y	N
[101]	LSTM	request rate	3 classes of real workloads	Y	N	Y	N
[70]	heuristic	utilization data	Skype traces	N	Y	Y	N
[188]	DT, RDF, MLP, BN	utilization data	real VNF workload	N	Y	Y	N
[157]	GNN	utilization data, topology	real VoIP workload	N	N	N	N
[227]	Q-learning	utilization data	real telco system	N	N	Y	N
[6]	Q-learning, GP	utilization data	synthetic VNF workload	N	N	Y	N
[5]	FL, Q-learning, SARSA	utilization data	'98 world-cup, Wikipedia	Y	N	Y	N
[110]	DQN	utilization data	Twitter analytics app	N	N	Y	N
[212]	Q-learning	utilization data	synthetic serverless app	Y	N	Y	N
Ours	ARIMA, LR, MLP, RNN	utilization data	synthetic, real CDN traces	Y	Y	Y	Y

In [5], the authors propose two different novel auto-scaling strategies, based on the combination of a FL-based controller with two different RL approaches: Q-learning (i.e., *off-policy* approach) and SARSA (i.e., *on-policy* approach). According to the authors, employing RL algorithms makes the overall mechanism self-adaptive (considering, e.g., the response time as reward signal), while the FL controller enables it to work at a higher level of abstraction. Both strategies are implemented and integrated with OpenStack. The evaluation is performed on two different real web application workloads.

In [110], the authors propose a strategy based on a DQN for tuning the scaling thresholds used by the auto-scaling rules of microservices deployed in Kubernetes. The application-level response times are extracted from the log files of a Twitter analytics application, and used as reinforcement signal for the RL algorithm.

In [212], the authors investigated on using RL-based techniques to handle resource allocations and scaling in the context of a serverless computing framework. They focused on request-based scaling and developed a mechanism to automatically adapt the concurrency level of a serverless application instance (i.e., the maximum number of requests that a single instance should handle).

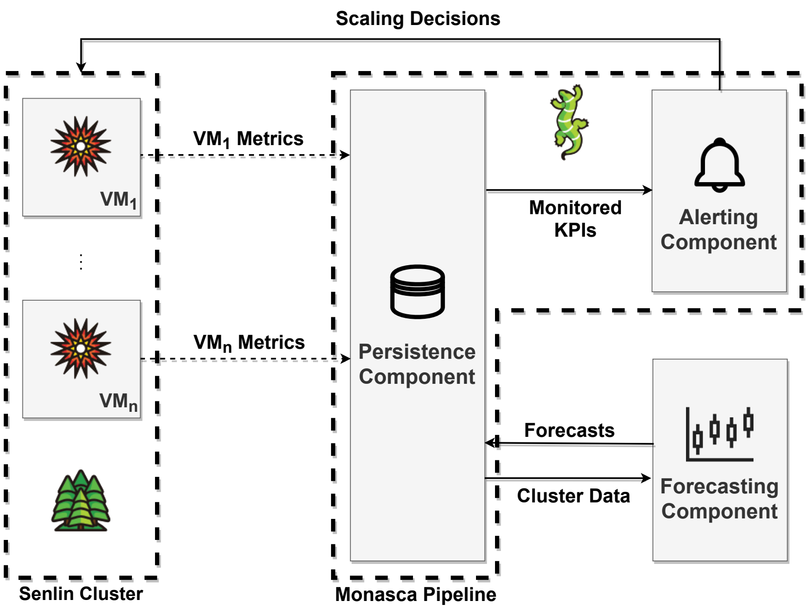


Figure 6.2: Architectural diagram of the proposed predictive auto-scaling approach.

6.2.3 *Summary*

Table 6.1 reports a schematic comparison among the major related works and our proposed approach (at the bottom of the table). The comparison takes into account the following aspects: which data-driven technique was used and which input data it was applied to; whether the method applies to generic workloads, or it is designed for specific applications; which data or use-case was used for validation; whether the work considered also overheads or delays related to spawning new instances; whether the work was actually aiming at realizing elasticity-control loops; and whether the implementation of the proposed solution is open-source, such that other researchers can reproduce the work, and possibly improve it. Overall, ours is the only work providing an open-source architecture for extending the orchestration capabilities of OpenStack with predictive analytics, enabling forecast-driven decision-making for generic elastic cloud services. Also, our implementation is modular, such that it can easily be extended with custom models developed using established modeling frameworks for the Python language, such as *Scikit-learn*, *Statsmodels*, *PyTorch* and *TensorFlow*.

6.3 PROPOSED APPROACH

As mentioned in Section 6.1, conventional scaling techniques for are *reactive*, as they adjust resources when certain metrics breach specific threshold values. They may be configured using a cautionary approach, triggering on very early degradation signs. Or, they may use an optimistic approach, by having thresholds very close to critical values. A cautionary approach is mostly necessary when scaling operations need a significant amount of time to become effective. However, the risk is, for instance, to waste resources (i.e., over-provisioning) due to unnecessary scale-out decisions. On the other hand, an optimistic approach limits resource waste, but can lead to affecting the QoS perceived by users in case scaling operations do not take effect before the system saturates its current capacity. Our approach mitigates the aforementioned issues by adopting a *predictive* auto-scaling strategy that triggers scaling actions on the basis of forecasts of one or more target metrics. For instance, given an imminent growth in the flow of requests, our approach allows for triggering a scale-out sufficiently ahead of time. We implemented our approach in OpenStack, specifically extending the Monasca monitoring system. As shown in Figure 6.2, we assume that the orchestration is performed by Senlin leveraging on our forecasting component (plus the required Monasca resources) to integrate predictions in the scaling operations. We consider a cluster of Nova VMs as the specific compute instances to be elastically scaled. However, notice that our approach is *agnostic* with respect to such implementation details.

Our approach works as follows. VMs periodically generate system-level metrics, which are ingested by Monasca. Such metrics are basically a set of time series that are indicators for the current load sustained by the system, which is the input to our forecasting component. The data are periodically fetched by the forecasting component, whose task is to generate forecasts of the input metrics over a given time window. The frequency and the time window of the forecasts, as well as the amount of historical data to be provided as input, are configurable. The generated forecasts are finally persisted, and become available to all the components of the infrastructure management system via Monasca APIs. In particular, they can also be visualized by operators through dashboards (e.g., Grafana). In our case, forecasts are fed to a threshold-based scaling policy and scale-out actions are triggered as soon as the *predicted* values of the considered KPIs breach the threshold for a given number of subsequent observations. Threshold checks are performed via the Monasca alerting pipeline and the scaling operations are actuated by Senlin.

The underlying predictors generally need to be trained before use, and constantly updated in case of dynamic changes in the statistic behavior of the time-series [111]. Based on the specifics of the metric dynamics, this may require longer or shorter training histories and periodicity of updates. In this work, for simplicity we assume that training is performed offline, but the integration of automatic model updates is planned to be handled in a short future. As discussed in Section 6.4 we compare four different predictors, i.e., a linear regressor

Listing 6.1: Example of forecasting component config file.

```
Api:
  # Fill with configs similar to monasca-agent-forwarder
Main:
  forwarder_url: ...
  hostname: ...
  inference_frequency_seconds: 60
  predictions:
    - tenant_id: # Fill with OpenStack project ID
      dimensions:
        scale_group: # Fill with scaling group ID
      metrics: [ cpu.utilization_perc ]
      group_by: [ "*" ]
      merge_metrics: false
      time_aggregation_statistics: [ avg ]
      time_aggregation_period_seconds: 60
      space_aggregation_statistics: [ sum ]
      lookback_period_seconds: 1200
      prediction_offset_seconds: 900
      out_metric: pred.group.sum.cpu.utilization_perc
      model_path: /path/to/model.dump
      scaler_path: /path/to/scaler.dump
Logging:
  enable_logrotate: true
  disable_file_logging: false
  predictor_log_file: /path/to/predictor.log
  log_level: INFO
```

(see [88], Chapter 3), an ARIMA model (see Section 2.2.2), an MLP (see [78], Chapter 6), and an RNN (see Section 2.2.5).

6.3.1 *Implementation Details*

Our forecasting component, also known as *monasca-predictor*, was implemented in Python, the main programming language used in the OpenStack ecosystem, and released under the Apache 2.0 open-source license [122]. Such component was realized in compliance with the microservice-oriented architectural pattern used by Monasca. In particular, it was designed to be eventually integrated into the *monasca-agent* (see Section 2.1.1.3).

Monasca-predictor is configured using a YAML file similar to the one shown in Listing 6.1. The `Api` block contains the information required to make authenticated calls to OpenStack APIs. As *monasca-predictor* performs tasks similar to those of the *monasca-agent*, it must be provided with similar permissions. The `Main` block contains the configurations related to the actual predictive tasks of *monasca-predictor*. Notice that the `forwarder_url` and `hostname` fields must be filled with the pointers to the *forwarder* process of the *monasca-agent*. The `inference_frequency_seconds` field specifies the frequency at which forecasted values must be generated. The `predictions` field is a list containing the individual configurations for the different forecasts. For each item in the list, additional fields can be specified as follows. The `tenant_id` field must be filled with ID of the OpenStack project containing the resources to be monitored. The `dimensions` field is a map that specify additional properties required to identify such resources (e.g., the ID of the elastic group of compute instances). The `metrics` field is a list of the metrics whose measurements are to be used as input to the predictor. In Monasca, a *metric* is identified by its name (e.g., `cpu.utilization_perc`) and the set of properties (i.e., *dimensions*) of the resource that generates measurements for said metric (e.g., `resource_id`, `hostname`, etc.). The `group_by` field is a list of such properties, to be used by Monasca API for grouping measurements in different time-series, when fetching data by metric name. For instance, one could simply specify `resource_id` (or even `*`, standing for *"all fields"*) to group measurements by resource. Depending on the boolean value specified for the `merge_metrics` field, the resulting time-series can also be merged into a single one, with measurements ordered according to their timestamp. The `time_aggregation_statistics` field is a list of operators (e.g., avg, sum, max) to be applied on the retrieved time-series individually, binning their measurements according to the resolution specified with the `time_aggregation_period_seconds` field, such that the result is possibly a multivariate—resampled—time-series. Similarly, the `space_aggregation_statistics` field is a list of operators to be applied, on top of the temporal aggregation result, in order to aggregate the measurements of a set of monitored resources (e.g., the compute instances belonging to the same elastic group). The `lookback_period_seconds` field defines the time window that measurements must fall into for them to be included in a Monasca API response, as a (backward) difference with respect to the current timestamp. The `prediction_offset_seconds` field defines the time window of the forecast, as a (forward) difference with respect to the timestamp of the most recent measurement. Notice that, when persisted, a forecast is associated with the very same timestamp, such that the forecasted metric appears backward-shifted by `prediction_offset_seconds` with respect to the input metric. The `out_metric` field is the metric name to be associated with the generated forecasted values. The `model_path` field is the path to the dump of the underlying predictive model. Similarly, the `scaler_path` field is the path to the dump of the scaler to be used for pre-processing the input data. At the moment, *monasca-predictor* only supports models built using the following frameworks: Scikit-learn [178]; Statsmodels [213]; PyTorch [177] and TensorFlow [151]. Finally, the `Logging` block allows for configuring logs management. A thorough explanation of the available tunables can also be found in the documentation within the code of our forecasting component [122].

Monasca-predictor was developed as a prototype that users can only configure via the YAML configuration file (see Listing 6.1). Such limitation implies that a user needs full administrator privileges for the OpenStack deployment, as such file contains both system-wide (e.g., authentication credentials) and application-specific configurations (e.g., the details of the resources to be monitored). We plan to improve the usability of our component by separating such different types of configurations, so that multiple—possibly unprivileged—users can leverage on the same forecasting capabilities (even though the installation must still be performed by an OpenStack administrator). Similarly to what Monasca and the other OpenStack projects offer, our idea is to develop both a command-line interface and a HOT integration for *monasca-predictor*, such that users can manage their (application-specific) configurations in the way they see fit. Independently of the chosen interface, such configurations will be eventually persisted in a database (e.g., the MySQL instance included in any OpenStack deployment) to improve the reliability of our component.

6.4 EXPERIMENTS

This section includes the results of an experimental validation of the approach described in Section 6.3. It provides a comparison of the performance of several *predictive* scaling policies with a traditional *reactive* one, considering a synthetic elastic application deployed on OpenStack.

6.4.1 *Synthetic Elastic Application*

We used *distwalk* [47] an open-source distributed processing emulation tool developed by us, to test the proposed predictive auto-scaling approach. The tool consists of a server module that accepts TCP/IP connections from one or more clients. Clients can request the server to perform computational, networking and/or I/O activities, enclosing within each request the amounts of resources to be consumed. Clients can submit requests with constant or exponentially distributed inter-arrival times, payload sizes, or I/O transfer sizes. Also, they may emulate ramp-up/ramp-down scenarios, or use a file trace specifying the requests rate to be submitted over a time horizon. For instance, we used this feature to replay traces from a real CDN workload (see Section 6.4.5). Per-request round-trip response-times can be measured and reported in a log file on experiment termination. Clients can also spawn multiple threads submitting traffic in parallel, and they can emulate the establishment of multiple sessions, by closing and re-establishing their TCP/IP connections.

6.4.2 *Experimental Set-up*

Our OpenStack deployment was hosted on a Dell R630 dual-socket test-bed, equipped with: 2 Intel Xeon E5-2640 v4 CPUs (each having 10 hyper-threaded cores, i.e., 20 hyper-threads) running at 2.40 GHz (with a turbo-boosting frequency of 3.40 GHz); 64 GB of RAM; Ubuntu 20.04.2 LTS operating system; version 5.4.0 of the Linux kernel. We used an all-in-one OpenStack deployment (*Victoria* release), installed using the tools provided by Kolla [169], resulting in each service being operated within Docker containers. As detailed in Section 6.3, we deployed an elastic control loop using the following services: *(i)* Senlin to orchestrate a horizontally-scalable cluster of Nova VMs; *(ii)* Octavia to provide the cluster with load-balancing capabilities; *(iii)* Monasca to ingest the system-level metrics and to trigger the scaling actions; *(iv)* the forecasting component, developed by us, to enable the *predictive* auto-scaling strategy.

The Senlin cluster had a minimum of 2 active instances and could expand up to 5. Each instance was provided with 1 vCPU and 2 GB of RAM available, and with an Ubuntu 20.04 cloud image including the server module of *distwalk* (see Section 6.4.1). We *artificially imposed* a delay of *~10 minutes* before starting the *distwalk* server. The purpose was to emulate a scenario with non-negligible set-up times for new instances, as it may be needed in real cloud workloads, where it is commonplace that spawning new instances may take from a few minutes to even half an hour [197]. In such a scenario, performing scale-out operations well ahead of time becomes critical. The application server instances were made reachable through an Octavia LB, set to distribute the traffic according to a *round-robin* strategy. The *distwalk* client was configured to spawn *6 threads* such that, in the beginning of our runs, each server instance had to handle the aggregated requests from 3 threads on average. Each thread followed a *~4h-long trace* reporting the operation rates (i.e., requests per second), to be maintained for an interval of *1 minute* each. Also, each thread was set to break its work in *1000 sessions*, such that a new connection was opened with the LB every *~15 seconds*, allowing for it to select a new target instance. Monasca was set such that new measurements were collected *each minute*. The forecasting component was set to output a new prediction with the same interval, leveraging on the most recent measurements. The input to the underlying forecasting model consisted in a time-series reporting the sum of the CPU usage measurements of the currently active instances. The output of the model was the estimated value of the same time-series in *15 minutes* (i.e., *~50% more* than how long a new instance takes to activate). Such output was then divided by the number of currently *active instances* to get an estimate of the *average* CPU usage expected in 15 minutes, assuming the cluster size to remain constant, and then persisted in Monasca.

The purpose of our experimentation is to show the effectiveness of the proposed architecture, and not to evaluate a novel ML model for time-series forecasting that can outperform existing predictive elasticity approaches. Indeed, the novelty of our work consists in proposing an open-source scalable architecture, compatible with Monasca, that can be easily configured by practitioners to plug virtually any type of time-series forecasting model into their data-driven control loops. Therefore, we decided to stick to a simple example where the elasticity controller uses only CPU usage as input. However, our component can be configured to predict multiple metrics per monitored instance, and/or perform multi-variate time-series forecasting.

To implement the *predictive* scaling strategy, Monasca was set to trigger a scale-out whenever the *predicted* average CPU usage of the cluster reached *80% for 3 times in a row*. On the other hand, to implement the *reactive* scaling strategy, it was set to do the same but considering the *actual* average CPU usage. In any case, a scale-in was triggered whenever the *actual* average CPU usage reached *15% for 3 times in a row*. We chose *not* to use the predicted metric to decide whether to trigger *scale-in* actions. While a cloud provider might want to anticipate traffic peaks by spawning additional resources in advance, disposing of superfluous resources can be much quicker [197], and is typically done after making sure that all residual traffic was *drained* from them. Otherwise, the risk is to overload the remaining instances in case they start taking the traffic relieving the being-released instance too early. Predicting such a condition is hard and, in practice, it is often more convenient to minimize SLA violations, rather than costs. However, our framework does not exclude this possibility. Waiting for 3 consecutive violations imposes a delay of at least *3 minutes* for an action to be triggered. However, this is a well-established practice in elasticity control loop design, as it helps with making the mechanism more robust to fluctuations. Each scaling action could adjust the size of the cluster by *1 instance* only, and could only take effect if it was triggered after a *cooldown period* of *10 minutes* since the last scaling action. The cooldown was also useful considering the 10-minutes delay forcibly added before new instances activate.

6.4.3 *Predictors Configuration*

To implement the underlying forecasting models, we used: *(i) Scikit-learn* for the LR; *(ii) Statsmodels* for ARIMA; *(iii) PyTorch* for MLP and RNN. To evaluate how the amount of past information given as input influences the prediction, we considered 3 different settings, i.e., 5, 10, and 20 minutes worth of measurements (see Section 6.4.4). Apart from

LR, that was fitted every time on a different input, all models were trained offline on the same synthetic dataset (and on the same machine where OpenStack was deployed). Such data consist of sinusoidal traffic patterns, with different frequencies and amplitudes, to provide models with expected behaviors for a wide range of operational modes (the dataset is open-source, see Section 6.4.6). Note that we did not conduct an optimal hyperparameters search, as we believe such a process goes beyond the scope of this work, whose focus is the integration of time-series forecasting techniques in the elasticity-control loop infrastructure, rather than ML models development. However, in what follows we provide some indications on why we took specific design choices, aiming at conducting a fair comparison among the implemented models.

ARIMA meta-parameters were configured such that $p = \{5, 10, 20\}$, $d = 1$ and $q = 0$. While p was somewhat constrained by the input size, we chose $d = 1$ for the stationarity assumptions, and $q = 0$ as we did not observe any benefit from using this feature of the model. For ARIMA, we observed an average training time of *2.89 seconds*. MLP consists of an input layer (with units varying in $\{5, 10, 20\}$, as per the input size constraints), two hidden layers of *10 units* each (as we wanted to keep the complexity low), and an output layer of *1 unit* (as we needed a scalar output). We also applied a *leaky* ReLU non-linearity to the output of each hidden layers, as it generally speeds up the training. MLP was trained with SGD for 1000 iterations, using a decaying learning rate (between 0.1 and 0.001), a momentum set to 0.8, and a batch size of 500 input samples. These values are generally considered sensible defaults and, given the observed performance, we did not feel the need to fine-tune them. The 3 considered variants of MLP consist of 181, 231 and 331 *learnable parameters*, when the input layer size is set to 5, 10, and 20 respectively. For MLP, we observed an average training time of *136.15 seconds*. RNN consists of 3 (stacked) recurrent layers, each one composed by 200 units and using ReLU as activation function (i.e., PyTorch's defaults), followed by a fully-connected layer of *1 unit* (as we needed a scalar output). Such model was trained with SGD for 10000 iterations, using a decaying learning rate (between 0.01 and 0.001), a momentum set to 0.5, and a batch size of 300 input samples. With respect to MLP, we had to fine-tune the latter parameters for the model to converge to an acceptable performance. RNN consists of 201601 *learnable parameters*, independently of the size of the input sequences. For RNN, we observed an average training time of *436.07 seconds*.

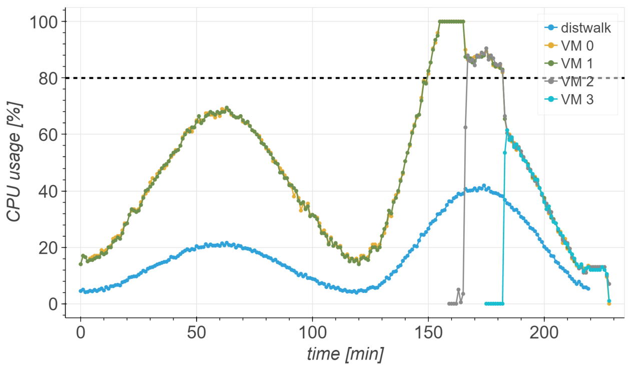
To facilitate comparison between models, we chose not to leverage on the capability of RNN to handle variable-length sequences, and trained it using fixed-length input sequences, like the other models. Also, during the runs, all forecasting models were re-loaded from disk each time they had to be queried (i.e., once per minute). In this way, we did not leverage on the hidden state of RNN to be updated after each query, that should theoretically allow the model to retain the observed dynamics and allow for computing forecasts even when provided with just a single new measurement as input.

6.4.4 *Validation on synthetic workload*

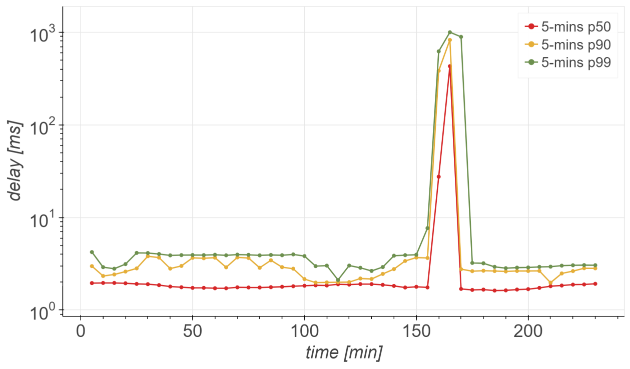
In this section we report the results obtained by applying five different scaling strategies to a synthetic workload similar to the one depicted in Figure 6.1. *Distwalk* was set such that the average CPU usage of the cluster ramps up twice during a single run. First, with a rather soft slope, peaking at ~70% (around minute 60) and progressively fading out until minute 120. Then, with a much steeper slope, (theoretically) peaking at ~120%, exceeding the cluster capacity. The first peak is designed to expose the behavior of a scaling strategy when facing a workload that *might* lead to saturation but that, instead, decreases *before* reaching the threshold (80%). In that case, a classical strategy would not react, whereas a predictive one may inaccurately forecast the evolution of the workload and trigger unnecessary actions. This scenario is useful to assess how sensitive to fluctuations and, thus, how prone to yielding *false alarms* a strategy is.

In what follows, in the CPU usage plots (e.g., Figure 6.3a), the blue curve represents the workload that each *distwalk* thread exercises on the cluster (i.e., the *ideal* usage we would observe if a single thread submitted requests to a single VM). As requests are submitted through the LB, the result is that, eventually, each VM in the cluster handles an equal share of the *cumulative* workload (see Section 6.4.2). In other words, the resource consumption curves do not closely follow the blue curve because, in the beginning of each run, there are 6 *distwalk* threads submitting requests to a total of 2 VMs through the LB. Therefore, each VM initially handles the aggregated requests coming (on average) from 3 threads. Instead, the red curve in, e.g., Figure 6.4a (left) refers to the *predicted* average CPU usage of the cluster, assuming the size of the cluster to remain constant. On the other hand, client-side response times plots (e.g., Figure 6.3b) provide a view of the system performance as observed by the *distwalk* client. Each plot reports curves that show (on the Y axis) the evolution of some response-time statistics (median, p90 and p99) over time (on the X axis), where each data point refers to a statistic aggregated over the data of a moving window of the previous *5 minutes*.

As shown in Figure 6.3a, the static scaling strategy fails to scale-out the cluster on time when facing the second peak. Starting from minute 155, it is possible to observe the system saturating its capacity (i.e., the CPU usage is 100%) and remaining in such a state for *~10 minutes*. In the meantime, the requests submitted to the cluster pile up and the client-side response times start growing up to *~1 second*, as visible



(a) CPU usage

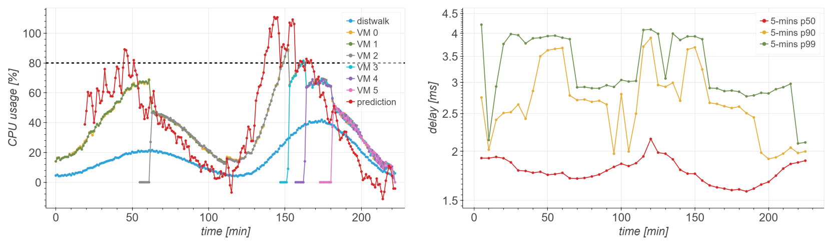


(b) Client-side response times

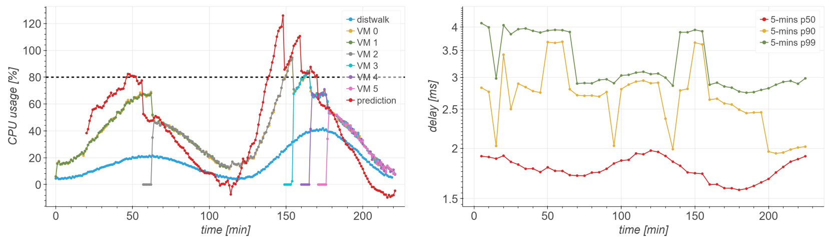
Figure 6.3: Experimental results for the traditional (static) scaling policy.

in Figure 6.3b. Such a performance degradation occurs because the static strategy waits for 3 consecutive violations of the threshold before triggering the scale-out. Furthermore, due to the *artificial* set-up delay, the new VM takes 10 minutes before starting to serve requests. Therefore, while the scale-out decision happens approximately at minute 152, the new VM starts responding only approximately at minute 166, and only for new established sessions (occurring every ~15 seconds, see Section 6.4.2). Such a scenario exposes the need for more *intelligent* strategies that are able to take scaling decisions ahead of time. We used 4 different time-series forecasting algorithms to implement different predictive scaling strategies, namely: LR, ARIMA, MLP and RNN. For each strategy, we considered 3 different values for the amount of past information to be fed to the underlying model (i.e., 5, 10, and 20 measurements, with minute granularity) when computing an estimate of the average CPU usage in *15 minutes*. In this way, we could assess how sensitive to the size of the input the predictive capability of a given model was.

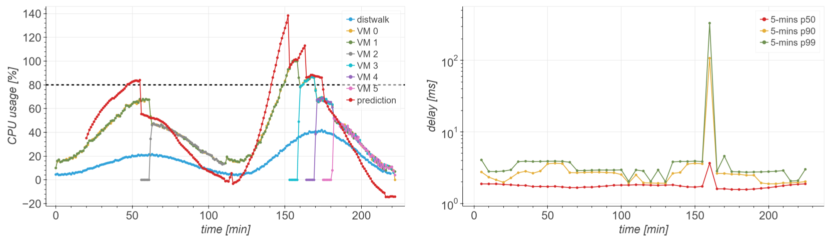
Figure 6.4 reports the results obtained using an LR-based policy, that generally performs a better job than the static policy at scaling the cluster before the second peak, as it correctly predicts the growth in the CPU usage. However, this predictor also tends to be overly sensitive



(a) input sequence length = 5



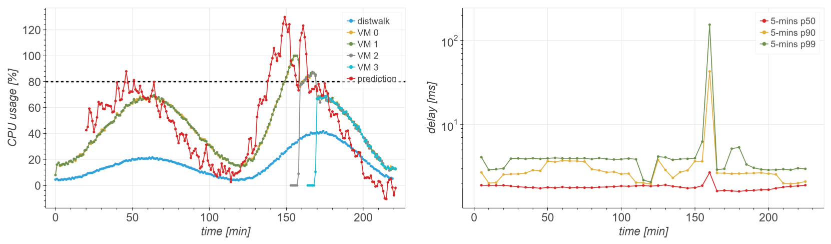
(b) input sequence length = 10



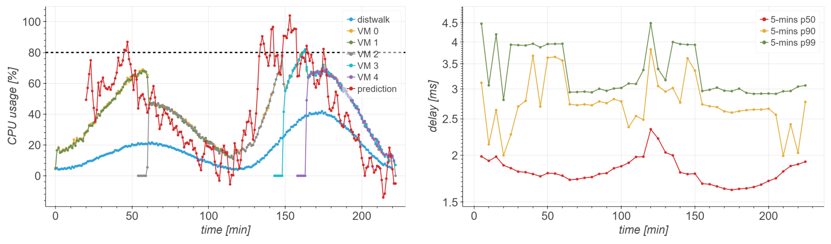
(c) input sequence length = 20

Figure 6.4: Experimental results for the LR-based scaling policy (left: CPU usage; right: client-side response times).

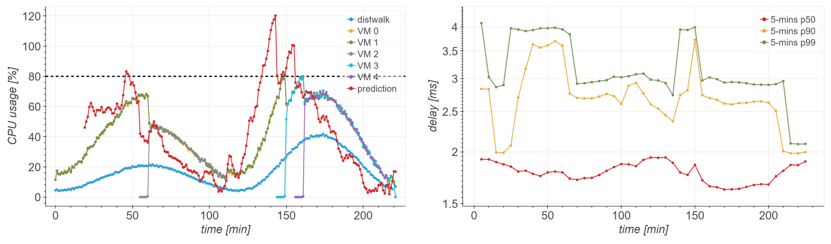
to the noise in the input, and to over-estimate. As a consequence, in all the runs, the LR-based policy triggers an unnecessary scaling action also before the first peak. For instance, when the input size is set to *5 minutes* (see Figure 6.4a), even though the cluster scales on time before the second peak, the policy seems to be overly sensitive to even small variations in the input, such that the resulting prediction is very noisy and, thus, not reliable in general. On the other end, when the input size is set to *20 minutes* (see Figure 6.4c), the cluster reaches 100% CPU usage for ~3 minutes (with client-side response times going up to ~100 ms), before the scaling action takes effect (around minute 159). This is due to the input size being too large, such that the LR cannot detect the growth soon enough. In other words, at the beginning of the ramp-up, the contribution of the newer measurements is outweighed by the older ones, generating a sort of *momentum* that delays the detection of the peak. We instead observe a nice behavior when reducing the input size to *10 minutes* (see Figure 6.4b), with the scaling action taking effect when the average CPU usage is at ~95% (around minute 155), and client-side response times going up to ~3 *ms*.



(a) input sequence length = 5



(b) input sequence length = 10

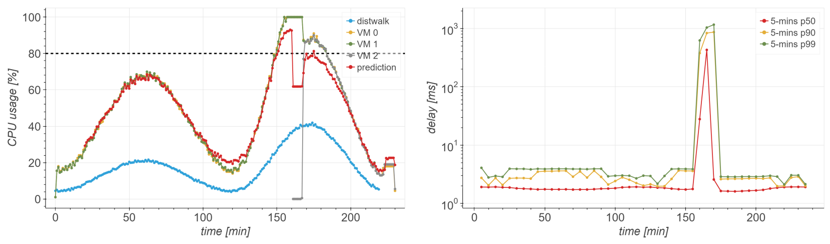


(c) input sequence length = 20

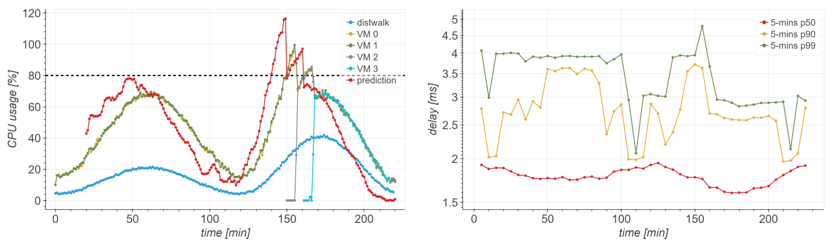
Figure 6.5: Experimental results for the ARIMA-based scaling policy (left: CPU usage; right: client-side response times).

Figure 6.5 shows the results obtained with an ARIMA-based policy. Similarly to the LR-based one, this policy seems to be generally overly sensitive to small variations in the input. In some cases (see Figures 6.5b and 6.5c), such behavior generates unnecessary scaling actions before the first peak. When the input size is set to *10* and *20 minutes* (see Figures 6.4b and 6.4c), the cluster is successfully scaled before the second peak, with the scaling action taking effect around minute 150. However, when the input size is set to *10 minutes*, the behavior of the policy appears significantly more noisy. On the other end, when the input size is set to *5 minutes* (see Figure 6.4a) the cluster reaches 100% CPU for *~3 minutes* (with client-side response times going up to *~100 ms*), before the scaling action takes effect (around minute 159).

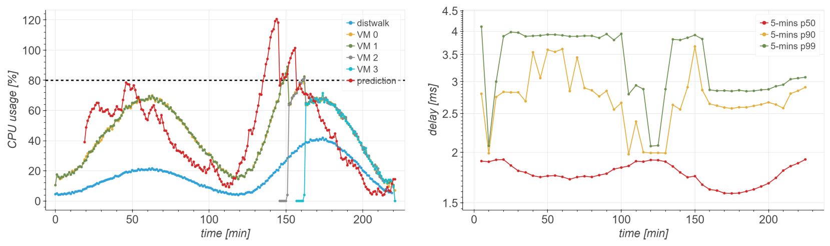
Figures 6.6 and 6.7 report the results of applying the MLP- and RNN-based policies, respectively. In contrast to the LR-based one, it is straightforward that the larger the input size, the better, in terms of overall performance. Setting the input size to *5 minutes* (see Figures 6.6a and 6.7a) results in obtaining a policy that is equivalent to



(a) input sequence length = 5



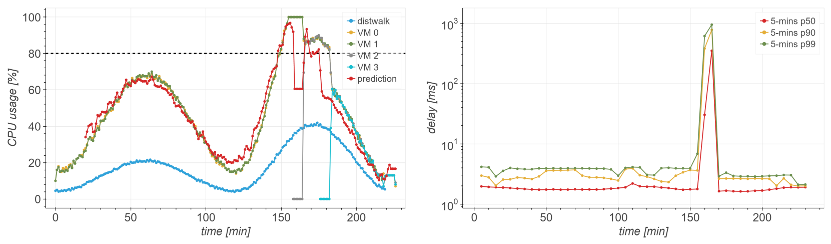
(b) input sequence length = 10



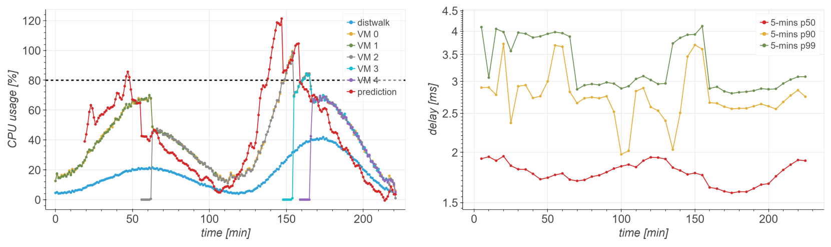
(c) input sequence length = 20

Figure 6.6: Experimental results for the MLP-based scaling policy (left: CPU usage; right: client-side response times).

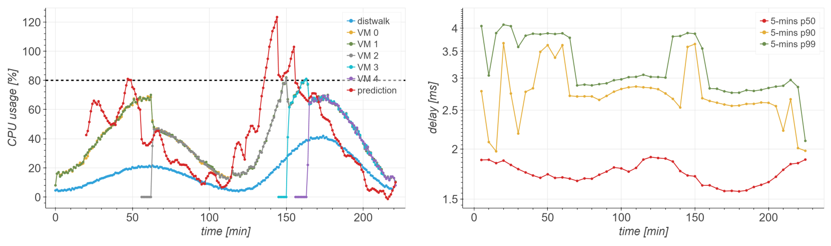
the static one (see Figure 6.3). As it is the case for the static policy, the inability to anticipate the second peak leads to a saturation of the system capacity that persists for ~*10 minutes*, with client-side response times growing up to ~*1 second*. On the other end, both MLP and RNN behave nicely with input size set to either *10* or *20 minutes*. When the input size is set to *10 minutes* (see Figures 6.6b and 6.7b), both policies scale the cluster just in time to prevent saturation, as the actions take effect around minute 155, when the average CPU usage is at ~99%. However, there is no sign of performance degradation from the client perspective, as the response times stay below ~*4 ms*. Also, both policies scale the cluster earlier when the input size is set to *20 minutes* (see Figures 6.6c and 6.7c), even though the predictions appears more noisy (that could lead to unexpected behaviors in other circumstances). In both cases, the scaling action takes effect around minute 151 and the client-side response times stay below ~*3 ms*. However, while for the MLP-based policy the scaling action takes effect when the average CPU usage is at ~89%, for the RNN-based one the same happens at ~80%. Such difference is likely the result of random fluctuations in



(a) input sequence length = 5



(b) input sequence length = 10



(c) input sequence length = 20

Figure 6.7: Experimental results for the RNN-based scaling policy (left: CPU usage; right: client-side response times).

the measured load. Remarkably, the RNN-based policy triggers an unnecessary scale-out before the first peak, as the predicted average CPU usage exceeds the threshold for the *exact* amount of times that is required. The same happens when using the ARIMA-based policy. Such behavior exposes the need for properly tuning, beside the specific forecasting model, also the other components of the scaling strategy. For instance, we could make the RNN-based policy more robust by increasing the number of times the threshold must be breached before triggering the action. Automatically adjusting a broader set of tunables (e.g., cooldown period, scaling adjustment, alarm thresholds, etc.) is among the engineering issues to be addressed in future extensions of the proposed architecture. Approaches based on neural networks are, in general, able to capture even fairly complex non-linear relations. However, in this case, an input size of *5 minutes* is clearly not enough to provide such models with the information required to output a 15-minutes forecast.

Table 6.2 reports the MAPE made by each predictor configuration during our runs. The MAPE was computed by considering the sum of

Table 6.2: Prediction errors (MAPE) observed for the considered runs.

	LR			ARIMA			MLP			RNN		
	5	10	20	5	10	20	5	10	20	5	10	20
MAPE	0.25	0.29	0.38	0.22	0.26	0.15	0.52	0.18	0.14	0.44	0.18	0.15

Table 6.3: Descriptive statistics of the client-side response times (ms) observed during the experimental runs, when the cluster was facing the first peak (minutes 0-120).

	avg	p50	p90	p95	p99	p99.5	p99.9
Static	2.12	1.81	3.18	3.73	3.96	4.08	8.12
LR (05)	3.30	1.79	2.76	3.56	3.92	4.03	9.41
LR (10)	2.02	1.79	2.81	3.57	3.90	3.99	6.59
LR (20)	1.95	1.75	2.72	2.95	3.82	3.90	5.89
ARIMA (05)	2.07	1.82	2.90	3.74	3.95	4.03	7.19
ARIMA (10)	2.03	1.80	2.78	3.16	3.92	4.17	7.79
ARIMA (20)	2.02	1.80	2.79	3.49	3.91	4.02	8.52
MLP (05)	2.03	1.80	2.82	3.68	3.90	3.97	6.88
MLP (10)	2.05	1.80	2.89	3.71	3.93	4.01	7.47
MLP (20)	2.04	1.80	2.81	3.69	3.91	4.00	7.65
RNN (05)	2.07	1.79	2.97	3.72	3.94	4.04	7.78
RNN (10)	2.01	1.79	2.79	3.57	3.90	4.00	7.80
RNN (20)	2.72	1.74	2.77	3.50	3.87	3.99	14.36

the CPU usage of all VMs, as ground truth, and the predicted values multiplied by the number of active VMs at each specific point in time (i.e., the *predicted* sum of the CPU usage of all VMs). Such results further support our conclusions regarding which configuration is best for each predictor. For instance, we can see that LR performs better when the input is set to 5 or 10 (although, the former leads to a very sensitive scaling policy). Conversely, in general, the bigger the input, the better the performance of the other predictors.

Tables 6.3 and 6.4 report the average and percentiles of the response-times observed by the client, during the peaks of our load profile. From the client perspective, when facing the first peak, all the considered policies basically guarantee the same performance level. Remarkably, leveraging on an additional compute instance (e.g., as with the RNN-based policy) does not make any difference, thus it is just a waste of computing resources. In contrast, by looking at Table 6.4, we can appreciate significant performance degradation for some specific policies. For instance, bad p99.9 are obtained with the static policy, the LR-based policy with too long inputs, and both the ARIMA and the neural-based

Table 6.4: Descriptive statistics of the client-side response times (ms) observed during the experimental runs, when the cluster was facing the second peak (minutes 121-220).

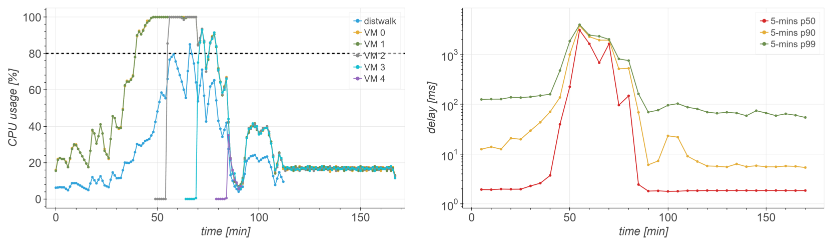
	avg	p50	p90	p95	p99	p99.5	p99.9
Static	47.06	1.80	58.12	394.05	824.53	924.59	1050.76
LR (05)	1.89	1.67	2.63	2.78	3.79	3.92	7.82
LR (10)	1.88	1.67	2.60	2.75	3.77	3.91	7.69
LR (20)	4.90	1.66	2.69	3.67	84.76	245.82	381.07
ARIMA (05)	3.42	1.72	2.76	3.72	39.48	63.80	298.10
ARIMA (10)	1.96	1.70	2.68	2.81	3.80	4.48	9.10
ARIMA (20)	1.97	1.70	2.66	2.78	3.81	4.51	11.71
MLP (05)	63.63	1.84	198.39	522.05	953.45	1051.49	1278.05
MLP (10)	1.99	1.69	2.66	2.84	3.84	4.56	11.00
MLP (20)	1.94	1.67	2.64	2.77	3.73	3.89	9.53
RNN (05)	38.04	1.78	23.17	319.82	755.57	862.19	1024.15
RNN (10)	4.50	1.68	2.65	2.82	3.78	4.03	103.55
RNN (20)	1.90	1.65	2.62	2.72	3.69	3.81	6.12

Table 6.5: Average overhead (ms) imposed by the proposed forecasting component, for the considered predictors.

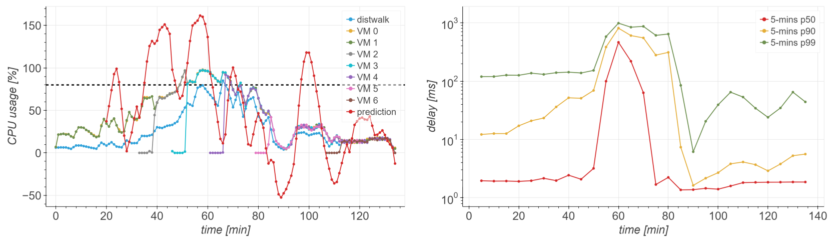
	LR			ARIMA			MLP			RNN		
	5	10	20	5	10	20	5	10	20	5	10	20
total	182.7	224.5	356.7	181.2	245.2	459.1	151.9	213.1	373.0	171.6	256.6	340.8
proc.	58.8	58.4	71.1	71.5	102.9	218.6	54.0	60.5	73.9	66.7	72.8	80.7

policies with too short inputs. On the other hand, sufficiently good results are obtained using the LR-based policy with short inputs, and with ARIMA and ANN-based models with sufficiently long inputs.

Table 6.5 reports the overhead imposed by the proposed forecasting component at each activation that, in our experiments, happened once per minute. The *total* time includes interacting with the Monasca APIs to fetch the data needed as input. Predictive policies based on LR, MLP and RNN all exhibit quite similar overheads, in the range of additional ~*60 ms* of processing time every minute, which seems acceptable in the considered scenario. On the other hand, ARIMA exhibits quite heavier overheads. Notice that RNN performance is measured in a pessimistic setting, i.e., assuming to unfold the network on the full sequence in order to obtain a prediction. The ability of RNN to keep a dynamic memory of the sequence history could be leveraged on to compute the prediction more efficiently. In that case, computing times would be roughly equivalent to those reported in Table 6.5, divided by the input sequence length.



(a) Static scaling policy



(b) RNN-based scaling policy

Figure 6.8: Experimental validation on real workload traces (left: CPU usage; right: client-side response times).

Table 6.6: Descriptive statistics of the client-side response times (ms) observed during the experimental validation on real workload traces (focus on the peak, minutes 45-90).

	avg	p50	p90	p95	p99	p99.5	p99.9
Static	879.01	426.26	2237.84	3019.86	3827.09	3940.71	4033.82
RNN (20)	163.64	20.62	549.91	687.74	883.46	943.71	1061.97

6.4.5 *Validation on real workload traces*

Our approach was also validated against a dataset exported from a real production environment. We considered the data provided by use-case D of the *RECAP* [131] EU project, reporting the requests handled by 3 nodes of a Content Delivery Network (CDN) managed by a British service provider (from 2016 to 2017). We converted a subset of this data to a *~2h-long* trace compatible with *distwalk*, such that we could generate similar traffic on our infrastructure. Similarly to the previous runs, the client was set to spawn 6 threads, each one maintaining the rates specified in the trace for an interval of 1 minute each. Each thread was also forced to break its work in 1000 sessions.

The purpose of validating our approach against a real workload is twofold: *(i)* we can assess the effectiveness of our approach in a production-like scenario, and *(ii)* we can evaluate its ability to generalize to workloads that do not quite resemble what the underlying models observed during training. Similarly to the previous runs, we considered the static policy as a baseline, and we compared its performance with the predictive ones. For brevity, we only show the plots for the RNN-based policy, that exhibited the best overall performance. Figure 6.8a (left) shows that the static policy cannot prevent the system from saturating for *~20 minutes* (from minute 48 to 69). The scaling decision is only triggered around minute 42, that is too late, considering the *artificial* ~10 minutes delay before new VMs start responding. Conversely, Figure 6.8b shows the RNN-based policy acting just in time before the system completely saturates (e.g., around minute 51 and 58). Also, the static policy run takes *~30 minutes* more to terminate because, during the saturation, the system was not able to respond to *distwalk* requests and caused a delay for the client to start the subsequent sessions. By comparing the plots reporting the client-side delays, we can see that, even though the RNN-based policy induces peaks of *~1 second*, it does not lead to a consistent increase of the delays, as the static policy does. This is also shown in Table 6.6, where, e.g., we can see the p99.9 in the static policy case being *~4 times* greater than the RNN-based policy case. Despite the workload used for this additional validation does not quite resemble what the RNN observed during training, the model exhibited a reasonable generalization ability. However, to obtain a higher prediction accuracy, one should have trained the model also on some prior segment of such new data, something we omit here for brevity.

6.4.6 *Reproducibility*

For reproducibility, this work comes with a public companion repository [124] including: the Heat templates to set up the infrastructure; the configuration files for the different tools (e.g., the synthetic *distwalk* workload); the raw results produced by our experiments; the code to generate Figures 6.3 to 6.7 and Tables 6.3 to 6.5; the synthetic dataset and the code to train the forecasting models (pre-trained models are also included).

6.5 CONCLUSIONS

We proposed an architecture that enables predictive operations in cloud infrastructures. We prototyped our approach using OpenStack, extending Monasca to ingest predictive metrics that reflect the expected evolution of the monitored system in the near future. Such metrics can be seamlessly combined with the regular ones to build operations policies that go beyond standard, *reactive*, strategies. As a case-study, we realized a predictive elasticity controller for a cluster of VMs (managed by Senlin), able to anticipate workload changes that might not be easily handled by classical threshold-based rules. The approach was validated both on synthetic and on real workload traces from a production CDN. Remarkably, it proved to be particularly useful for services with non-negligible instance spawning times, a commonplace condition in production environments [197] (e.g., creating a new VMs, and applying the required configurations, may require tens of minutes).

7

INTELLIGENT CLOUD OPERATIONS

7.1 INTRODUCTION

Cloud computing has become an essential technology in the modern distributed computing landscape [150]. Many diverse application domains [8, 106] leverage on services deployed in either public or private clouds, like smart cities, industrial factories, healthcare, e-Commerce, or even telecommunications, with the increasing adoption of NFV [39]. Cloud infrastructures and services have rapidly evolved [25] from the IaaS provisioning model to the PaaS one, that is the most widespread nowadays. PaaS enables development and deployment of modern cloud-native applications [21], deeply integrated with a plethora of APIs and services, such as: reliable relational and NoSQL databases, advanced and secure networking, LB and auto-scaling, serverless computing, integrated ML frameworks for training and operating large models, etc. Correspondingly, cloud infrastructures have significantly grown in size and complexity, having to deal with an ever-growing software stack on top of which such a wide variety of services can be made available. Guaranteeing high reliability and availability is only possible thanks to effective operations teams, that work 24/7 to keep such systems up, running and responsive.

Operating big and complex distributed infrastructures is far from trivial. Industrial practices heavily rely on monitoring metrics collected from physical and virtual elements of the infrastructure, e.g.: physical hosts, VMs, containers, networking appliances, and others. Metrics are persisted such that they can be visually, or analytically [26], inspected by operators. Also, they are typically automatically checked against a number of predefined, usually threshold-based, rules that possibly identify problems and trigger appropriate corrective actions. A classical example is a LB with self-healing capabilities, that adds new instances to an elastic compute group whenever the number of healthy instances goes below a configured amount. Other mechanisms are instead based on predefined pattern-matching rules to be checked against logs [67]. In a large-scale cloud operations scenario, there are thousands of active automation rules. However, as new scenarios occur, such rules need continuous adjustments to keep on being effective. Also, in response to an issue being identified, an operator typically starts a (non-trivial) RCA [220, 250], to understand what caused it, and ultimately what is the right fix. One of the greatest challenge for cloud providers is to *sustainably* deal with the ever-increasing size of the physical infrastructure. Ideally, without having to correspondingly increase the number of operators that continuously watch dashboards, and troubleshoot and fix infrastructure problems. In other words, they should aim at that "rapid provisioning with nearly zero human interaction", originally predicated by NIST [153].

Therefore, cloud providers are increasingly investing in developing intelligent techniques to support humans operators in their tasks, such as AD [270], resource allocation and capacity planning. Given the abundance of operational data, it is natural to seek for data-driven approaches like ML, that can augment the capabilities of operators to "navigate" the zillions of available time-series and logs. For instance, considering the AD problem, many works in the literature leverage on either supervised [65, 82] or unsupervised [2, 50, 219] ML algorithms to detect early symptoms of anomalous conditions at different levels of a cloud infrastructure. Similarly, many recent works [103, 123, 197] propose time-series forecasting techniques to anticipate the evolution of workloads and scale compute resources (e.g., VMs or containers) accordingly. However, effectively using such approaches in production is not straightforward. Indeed, there are many characteristics of the monitored system to take into account, like the topology of the physical infrastructure [157], the design patterns used to realize the individual applications [95, 105], or the in-place QoS requirements [257].

7.1.1 *Contributions*

We propose a strategy for *intelligent* cloud operations that consists of two phases: *(i)* detecting anomalous operational conditions of an application made of an elastic group of cloud instances; *(ii)* identifying the faulty component within the group, and proposing the best corrective action to restore it. Both phases rely on ML models to *learn* from the appropriate operational data to detect early symptoms of anomalous conditions and to identify the proper corrective actions to apply, without explicitly coding static rules. This work aims at closing the cloud operations loop in a totally automated fashion, envisioning a system with the ability to learn from the corrective actions applied by operators in similar previous cases. We validated the proposed approach by deploying a synthetic application and a Cassandra NoSQL cluster on an OpenStack testbed. We trained and tested the ML models on system-level monitoring data gathered while injecting different types of anomalies on the mentioned applications, including exogenous workload interferences, sudden failure of a cluster member, and saturation of CPU capacity and disk I/O bandwidth. For *(i)*, we trained an AD model such that it could generalize to variable-sized groups of instances. On the respective test sets, for the synthetic application, we obtained a ROC-AUC of 97% and an accuracy of 90.34%, while, for Cassandra, we obtained a ROC-AUC of 94% and an accuracy of 87.50%. For *(ii)*, we trained a supervised multi-label classification model, such that it could associate corrective actions to instances individually. On the respective test sets, for the synthetic application, we obtained an accuracy of 96.15%, while, for Cassandra, we obtained an accuracy of 98.75%. See Section 7.4 for more details on the performed experimentation.

7.1.2 *Chapter Organization*

This chapter is organized as follows. Section 7.2 provides a brief recall of related research. Section 7.3 presents our architecture for intelligent cloud operations. Section 7.4 reports the validation of our prototype on an OpenStack test-bed. Section 7.5 concludes the chapter and outline possible ideas for future research on the topic.

7.2 RELATED WORK

Recently, ML-based approaches have been increasingly proposed as effective solutions to a diverse set of resource management tasks [94] for both public and private cloud infrastructures. In previous chapters, we already provided a comprehensive overview of the related research works on such topics. Therefore, in this section, we limit ourselves to describing a few additional existing approaches, to highlight how our work differentiate from such alternatives.

In [65], the authors evaluate several supervised learning approaches for AD by injecting faults in a Kubernetes cluster. Similarly, in [82], the authors also evaluate supervised learning techniques for off-line AD in an NFV environment. The authors train their models on host monitoring data collected while injecting anomalies in a test-based running the ClearWater IMS system on top of OpenStack. Also, the authors of [270] provide a thorough survey where they discuss the risks, in terms of anomalous behaviors, correlated to switching to a NFV/cloud model. For instance, incurring in temporal interferences generated by virtualization and resource over-commitment. In [2], the authors propose a real-time unsupervised AD technique based on HTM. In [250], the authors describe a RCA approach for NFV anomalies, based on a digital twin. They frame the problem as a dynamic set-covering, and propose a scalable solution based on HMM. In [160], a variational autoencoder based on RNNs is proposed for AD in cloud scenarios, where the autoencoder trained on normal/healthy conditions, is expected to produce larger errors under anomalous/unhealthy conditions. This is followed by a one-dimensional **CNN!** classifier used to identify the anomaly as being either a case of process death, CPU stress, network delay or packet loss.

Compared to the above research, our proposed approach tries to bridge the gap between detecting a possible issue within a cloud system or component, identifying the exact affected element within the infrastructure, and deciding what corrective action to apply in order to return the system to a normal behavior. This work aims at closing this loop in a totally automated fashion, and with the ability to learn from the corrective actions applied by humans in similar previous cases. Most ML-based approaches focus on specific operations aspects, like auto-scaling. Instead, our scope includes a wider range of operations problems. Unlike most related works, we framed the problem of deciding a corrective action as a multi-label classification task. Typically, operations teams cater collections of procedures known to be effective at recovering their systems from (recurrent) error conditions. Also, when responding to an issue, the same teams are required to log their actions, in a ticketing system. Such information can be correlated with system-/app-level data, to learn *"intelligent"* operations models. For instance, the approach described in [160] brings an interesting resemblance with our approach, in that both include an unsupervised layer for AD, followed by an anomaly classifier. However, the previous work analyzes metrics from a single instance at a time only, and it does not consider the common case of horizontally-scalable elastic clusters. Furthermore, in our work we aim at letting the system learn what corrective action to apply to the anomaly being analyzed, imitating what was made with prior manual interventions.

7.3 PROPOSED APPROACH

In this section, we present an overview of the proposed architecture, and discuss some fundamental implementation details.

7.3.1 *General Architecture*

Traditional approaches to cloud operations ensure healthiness of applications through (often complex) automations that are, hopefully, able to detect possible abnormal conditions, send appropriate alerts and possibly trigger recovery actions. However, such mechanisms are still typically based on static rules and thresholds, that are often very easy to interpret, but quickly become cumbersome to maintain as the scale of the system grows. Using ML to solve the kind of problems mentioned above is strongly supported by the abundance of (very diverse) operational data that are produced in cloud environments. Either the infrastructure components themselves, or the on-call personnel that work around the clock to make sure that everything runs smoothly,

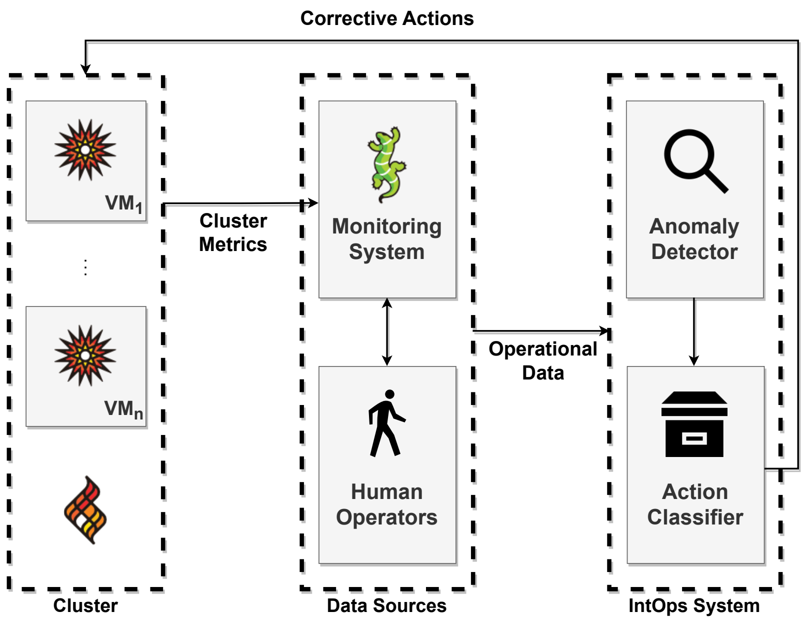


Figure 7.1: Architectural diagram of the proposed intelligent operations approach.

continuously generate useful information. Such information can be leveraged upon to devise intelligent automations, that adapt as they observe more diverse operational conditions. Figure 7.1 shows how our approach enhances the control loop of a cloud infrastructure, by ingesting the operational data coming from the aforementioned data sources and producing recommendations. In such settings, it is possible, e.g., to anticipate the occurrence of system outages, by analyzing the historical data describing the relevant system- and application-level metrics during past outages. Or, e.g., to identify their likely root-cause, and the most effective corrective actions to be applied, by looking for similarities in logs and reports associated with past incidents. At the moment, our approach allows for: detecting performance degradation due to workload co-located on the same physical hosts, recommending a relocation on healthier host; detecting faulty members of load-balanced groups of instances, that stop taking their share of the load, recommending to reboot the offending instance; detecting the shortage of allocated resources due to dynamic workload changes and its expected evolution in the short-term, recommending an elastic scaling action, to prevent serious performance degradation. A number of other anomalous scenarios are planned to be integrated into the framework, including transient failures and hardware degradation (i.e., not entire failures, but faults impairing seriously the performance of disks, memory modules, network interface cards, etc.). Remarkably, when the set of observed anomalous scenarios to consider grows, ML-based approaches like ours scale significantly better than traditional *static* rules and thresholds. Indeed, such approaches require continuous manual tuning to capture new, unforeseen, anomalous behavior, and possibly to develop a separate criterion for each possible case. Instead, for ML models, it is often sufficient to add the new observed behavior to the training set, and restart the training procedure, without explicitly coding new rules. Furthermore, provided that the resulting ML model exhibits a satisfactory generalization power, re-training might not even be necessary.

7.3.2 *Implementation Details*

To demonstrate the effectiveness of our approach, we implemented it to work with data exported from OpenStack, one of the most used open-source cloud orchestration frameworks. For simplicity, in our validation (see Section 7.4) we only considered the CPU and disk I/O activity measurements generated by the runs of our test application. However, our approach can handle a variable number of system- or application-level metrics. Then, we manually labelled such raw data to distinguish among different operational conditions, also taking into account the related response times measurements, collected client-side, as a general indication of the QoS. In other words, we emulated the information that are typically produced by operations teams after a system outage occur (e.g., *post-mortem* documents). Notice that, while the test application was running on a horizontally-scalable group of VMs, our approach is *agnostic* with respect to the used virtualization technology. Also, for both steps, we preferred ML models that guarantee a sufficient level of interpretability, as cloud automations should be highly dependable and auditable. However, explainability analysis goes beyond the scope of our work, and will be addressed in the future.

For step *(i)*, we trained an AD model on spatial aggregates of such data, so that the model can generalize to elastic groups of instances. Also, as it is continuously executed, this step acts as a *low-cost* filter that prevents the system from running the (more expensive) step *(ii)* on higher-resolution data when it is not necessary. In these settings, it is impractical to assume the availability of large amounts of *labelled* data. We opted for MADI [219], an *unsupervised* AD approach that leverages on *negative sampling* to cope with labelled data shortage. MADI works spectacularly well with high-dimensional data that capture complex multi-modal behaviors, assuming that the presence of anomalous behavior is *scarce*. It assumes all provided training data to be positive examples, and computes a *negative space* to sample from, assuming that every behavior that significantly differ from the positive examples is to be considered anomalous. As there are now two distinct classes of examples, it is possible to use any supervised classification algorithm. We used the NSRF provided by MADI, setting the hyperparameters as specified in the seminal paper. We trained NSRF on a dataset containing multiple traces of positive-only examples of expected CPU and disk I/O activity patterns. Such data were preprocessed by calculating spatial aggregations, i.e., mean (μ) and standard deviation (σ), to make the model agnostic to the actual number of instances in the cluster. After that, we applied standard scaling (i.e., subtracting to a signal its μ and dividing by its σ) and built the set of training samples by applying a rolling window of 5 observations, shifted by 1 observation at a time. Each sample consisted in a 2D vector with dimensions 5×4, i.e., a 5-minutes time-frame, partially overlapping with adjacent samples, containing 2 spatial aggregations of 2 distinct metrics. However, given that NSRF is not designed to natively work with multi-variate time-series, we reshaped each training sample to a 1D vector with dimension 20, such that the rows of the original 2D vector are stacked horizontally, and the contributions of the different signals are interleaved.

For step *(ii)*, we developed ourselves a simple, yet effective, supervised multi-label classification model using XGBoost [38], a powerful framework that offers performant implementations of GBT [73]. The job of this model is to learn to distinguish among different classes of anomalous conditions patterns, such that they can be associated with the appropriate corrective actions. This model is designed to run only when triggered by the AD model, that continuously analyze new observations as soon as they become available, and flags them if the corresponding application operates outside the expected conditions. Therefore, as it is supposed to run infrequently, we designed the model to work on instances' raw data, in an attempt to enhance its classification capabilities. Indeed, such model is designed to compare the behavior of an individual instance with the rest of the group (i.e., the other instances that implement the same application), by taking as inputs a combination of spatially-aggregated and raw data, under the assumption that all instances in the same group behave consistently. Given a flagged group of instances, that could even be fairly large, the classifier is applied to each one separately, to output a (possible) recommended corrective action for each of them. Notice that this strategy potentially allows for identifying both the root-cause and the appropriate counter-measure even when an anomaly is caused by multiple instances at once. We trained the model on data collected while injecting anomalies during runs of our test application (see Section 7.4). We applied a preprocessing similar to the one used for AD, such that each training sample consisted in a 2D vector representing a 5-minutes window on the raw data. However, for each window, we generated a number of training samples equal to the number of instances in the group. Each sample consisted in a 5×6 vector, where the columns contain the following information:

1. CPU utilization of the specific instance;
2. Disk I/O activity of the specific instance;

3. μ of the CPU utilization of the other instances;
4. μ of the disk I/O activity of the other instances;
5. σ of the CPU utilization of the other instances;
6. σ of the disk I/O activity of the other instances.

Also in this case, given that XGBoost is not designed to natively work with multi-variate time-series, we reshaped each training sample to a 1D vector with dimension 30, such that the rows of the original 2D vector are stacked horizontally, and the contributions of the different signals are interleaved. We used the metadata of our runs to *label* each sample according to the corresponding type of behavior that, in turn, is associated with a specific corrective action. If a given sample was related to an injected instance, and at least 3 observations had been collected during the injection, then the sample was labelled accordingly: 1 for *stress*, 2 for *fault*, and 3 for *saturation* (0 otherwise).

7.4 EXPERIMENTS

This section presents the results of an empirical validation of the approach described in Section 7.3, conducted by deploying both a synthetic application and the Cassandra NoSQL data store on OpenStack. We used data from such deployments to train the underlying ML models, and assess their accuracy.

7.4.1 *Experimental Set-up*

We carried out our experiments on an OpenStack installation (*Yoga* release), that was deployed using Kolla [169] (i.e., the various services run within several Docker containers). OpenStack was hosted on 3 physical hosts:

1. A Dell R630, equipped with: 2 Intel Xeon E5-2640 v4 CPUs (20 hyper-threads each) running at 2.40 GHz; 64 GB of RAM; a 3.3 TB Dell PERC H330 Mini hard disk; Ubuntu 22.04 LTS; Linux kernel 5.15.0. This host was used as *controller* and *compute* node.
2. A Dell R740xd, equipped with: 2 Intel Xeon Gold 6238R CPUs (56 hyper-threads each) running at 2.20 GHz; 126 GB of RAM; a 2.2 TB Dell PERC H740P Mini hard disk; Ubuntu 20.04 LTS; Linux kernel 5.4.0. This host was used as *compute* node.
3. A workstation, equipped with: an Intel Core i7-4790K quad-core CPU (8 hyper-threads) running at 4.00 GHz; 16GB of RAM; a 500 GB Samsung 850 SSD; Ubuntu 22.04 LTS; Linux kernel 5.15.0. This host was used as *compute* node.

These were all connected to the same switch using a 1 Gb link cable. We deployed a test application leveraging on the following services: *(i)* Heat [168], to orchestrate a horizontally-scalable cluster of Nova [172] instances; *(ii)* Octavia [173], for load-balancing; *(iii)* Monasca [170], for telemetry. The application cluster was configured to have 3 instances, each one deployed on a different physical host, such that we could better control the experiments that involved monitoring the disk I/O activity, by reducing interferences. Each instance was provided with 1 vCPU and 2 GB of RAM, and with Ubuntu 20.04 server cloud image. To better control our experiments, we disabled both the elasticity and the self-healing capabilities of the cluster, and we made sure that each instance was pinned to a different physical CPU core, that remained unchanged for the entire duration of the experiments. The instances were reachable through an Octavia LB, that was configured with a *least-connections* strategy. Monasca was configured such that new CPU and disk I/O activity measurements were collected each minute.

7.4.2 *Synthetic Workload Generator*

We used the open-source *distwalk* [47] tool, described in Section 6.4.1, to generate traffic on our deployment. We set the client such that the CPU and disk I/O activity of the instances followed a set of dynamic workload profiles. The client was configured to spawn 2 threads per instance, each one provided with a trace specifying the operation rates (i.e., requests per second), to be maintained for 1 minute each. Each thread was also configured to create a total of 5000 sessions over each run, such that a new target instance could be selected by the LB at each new session establishment.

7.4.3 *Apache Cassandra*

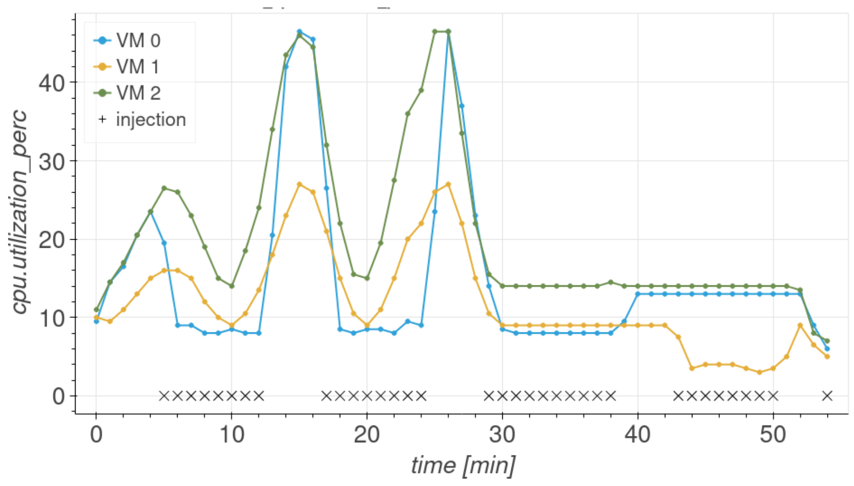
Beside the aforementioned synthetic application, we also used Apache Cassandra, a widely known open-source NoSQL data store, to also test our approach in more realistic application scenarios. Based on the design principles of Dynamo [59], Cassandra is a distributed data store characterized by a scalable and fault-tolerant peer-to-peer architecture, able to handle large amounts of data by spreading the load across the cluster. In practice, this is done by partitioning the key-space of a table primary key, spreading its shares over the peers. Cassandra offers the possibility to tune the level of write/read consistency, and the replication strategy. Such features make it a great cloud storage solution for critical big-data applications that require high scalability and availability, or for high-throughput use cases with less stringent consistency requirements. We deployed Cassandra on our OpenStack test-bed, with each peer hosted on a different physical host, and the keyspace replicated across the whole cluster to avoid data loss in case of anomalies. The traffic is generated using YCSB [43], a well-known open-source benchmarking tool for NoSQL data stores, which allows for configuring: the probability distribution of requests across the key-space; the number of pre-inserted records; the proportion of read, update, scan and insert operations to issue; and other performance-related parameters. In our case, to avoid saturating the available disk space, YCSB was configured to load into the cluster a pre-fixed amount of records (1 million, 1 KB each). Also, the traffic throughout each run included update operations only (3 millions in total, at a rate of 1000 ops/sec), such that the cluster could still perform write operations without increasing the total number of records. The cluster was also set with a *replication* level equal to 3, and a *consistency* level varying between 1 and 2.

7.4.4 *Anomaly Injection*

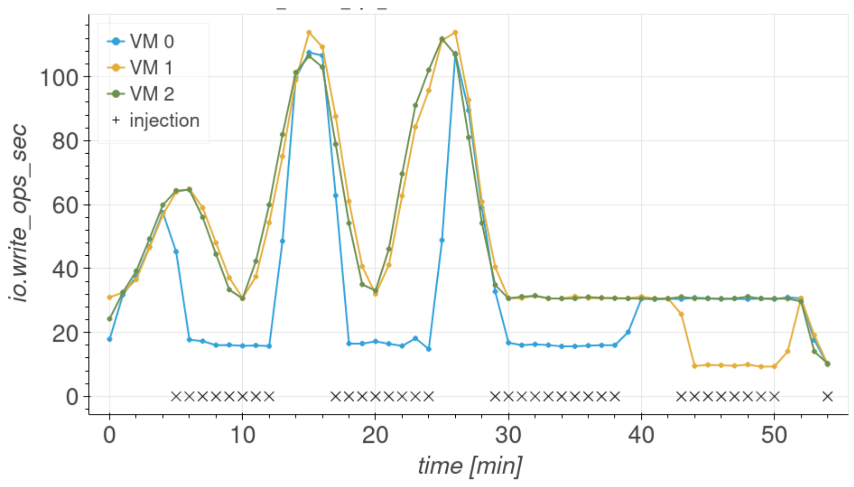
To train and evaluate the ML models underlying our approach, we needed examples of anomalous conditions to associate with the typical corrective actions described in Section 7.3. For simplicity, we considered only three of the most common anomaly types in cloud environments: *(i)* interferences generated by external load co-located on the same physical hosts; *(ii)* faulty members of load-balanced groups of instances that stop picking their traffic share; and *(iii)* saturation of the current resource capacity. To generate data describing such anomalous conditions, we artificially injected them during the execution of our runs. Specifically, for *(i)*, we used *stress-ng*[113] to simulate the interference of external processes that end up being scheduled on the same physical host of an instance. For *(ii)*, it was sufficient to kill the application process running on a specific instance to make it stop responding to requests. Whereas, for *(iii)*, we just made sure to send a workload that could not be properly handled by the currently allocated resources. We also augmented the diversity of the anomalous behaviors to be observed by our ML models by generating and enforcing schedules of randomly distributed anomalies. However, to better control our experiments, we made sure we had only one, randomly selected, unhealthy instance at a time. Also, once an instance was injected with an anomaly, we made sure it remained unhealthy for an extended period (e.g., 5-10 minutes), automatically recovering afterwards.

A few examples of the data extracted from our experimental runs with the *distwalk* application are depicted in Figures 7.2 to 7.4. By comparing such figures, one can appreciate the differences between the considered types of anomaly, in terms of observed system resource usage and client-side response times, during *distwalk* runs that follow the same workload profile. The system-level metrics (specifically, `cpu.utilization_perc` and `io.write_ops_sec`) were collected by using the Monasca monitoring system, while the client-side response times were extracted from the *distwalk* client logs. Note that, due to how the system components and the *distwalk* client are configured, if anomalies are *not* injected, then the LB continues to equally distribute the load among the available instances. In such case, the disk I/O activity level should be more or less the same for all instances. However, due to the different processors that the available physical hosts are equipped with, we can observe differences in terms of CPU utilization levels, even though the workloads follow the same profile during the run. A clear example of this scenario is depicted in Figures 7.4a and 7.4b where, during the first 10 minutes of the run, the available resources were sufficient to handle the workload. In this case, instance 2 was (randomly) scheduled on the physical host equipped with the most powerful processor (see Section 7.4.1), and exhibited a lower CPU utilization with respect to the other instances, while the disk I/O activity was more or less equivalent. Furthermore, in such *normal* cases, we can also observe particularly low client-side response times. In Figure 7.4c (and similar) we can indeed appreciate how the distribution of the response times evolve during a run, in terms of 50th, 90th and 99th percentiles. Each point in the plot refers to a specific statistic calculated over a 1-minute interval. For instance, a point at 0 refers to all the response times registered during the first minute of the run, and so on. Whenever the system did not saturate (e.g., during the first 10 minutes of the run), we generally observed a p90 *below* 35 ms. Therefore, we took this value as a rough indication of a *good* QoS.

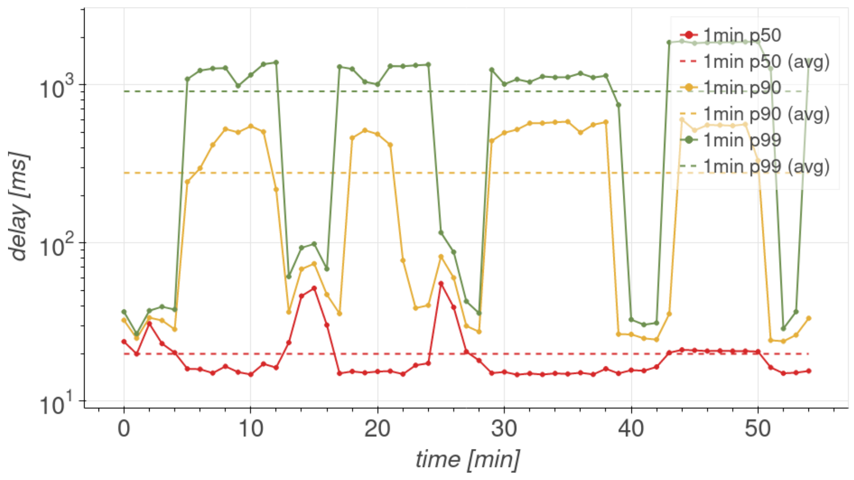
When using *stress-ng* to simulate interferences from co-located, I/O-intensive, external workloads, we observed the CPU and disk I/O activity of the affected instances significantly dropping and staying around relatively low values. For instance, in Figure 7.2, when the stress was injected around minutes 7-12 on instance 0, such instance exhibited a CPU utilization around ~10% (see Figure 7.2a) and a disk I/O activity around ~20 ops/sec (see Figure 7.2b), compared to the reference values, ~28% and ~62 ops/sec, respectively, exhibited by instance 2 during the first peak of the workload. The effect of the stress injection is even more significant around minutes 18-24, where instance 0, randomly picked again, exhibited similar resource utilization values, but this time with reference values being ~46% and ~108 ops/sec, respectively, during the third peak of the workload. The stress injection also significantly affects the response latency perceived by the client. Indeed, in Figure 7.2c, we can observe peaks of ~600 ms in the p90 curve, corresponding to the injection intervals. Due to how *distwalk* is designed, while an instance experiences interferences, but is still barely able to send responses, the client accumulates delay by waiting for responses, before triggering the subsequent requests. This is the reason why, during stress-injected runs, we typically observe longer



(a) CPU usage



(b) Disk I/O activity



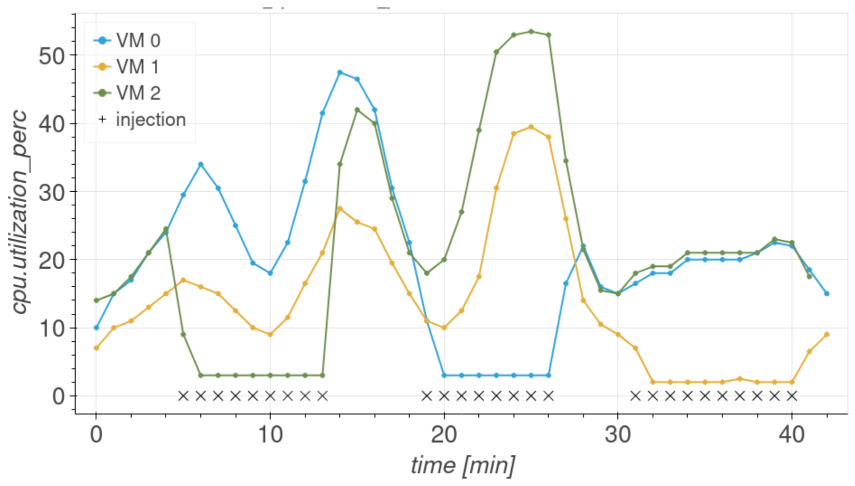
(c) Client-side response times

Figure 7.2: Interferences generated by *stress-ng* on *distwalk*.

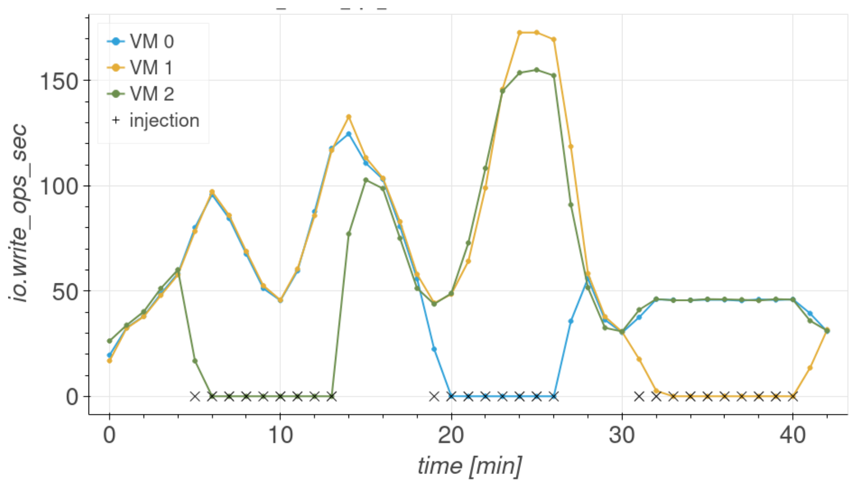
"tails" of delayed requests that keep on being sent at the last rate specified in the workload schedule (e.g., around minutes 30-50).

When killing the *distwalk* server process to simulate a fault, we observed the disk I/O activity of the affected instance dropping to 0, and its CPU usage stabilizing around 2-3% (i.e., the standard load generated by the OS background processes). On the other hand, the activity on the other instances increased accordingly, due to the LB redirecting the extra load on them. For instance, in Figure 7.3, when the fault was injected around minutes 6-13 on instance 2, we can see that the disk I/O activity of the other instances reached ~100 ops/sec during the first peak of the workload (Figure 7.3b). As we know a-priori that the workload should have closely followed an *ideal* sinusoidal pattern, we can definitely tell that it increased significantly with respect to the expectations. The effect of the fault injection, this time on instance 0, is even more significant around minutes 20-26, with instance 1 reaching ~170 ops/sec and instance 2 reaching ~150 ops/sec. Similar behaviors can be observed for the CPU usage, although it is less evident for the instances scheduled on the most powerful physical processor (see Figure 7.3a). Obviously, the fault injection also significantly affects the response latency perceived by the client. Indeed, in Figure 7.3c, we can observe peaks of ~100 ms in the p90 curve, corresponding to the injection intervals. However, overall, the impact on the QoS is significantly smaller than what we observed for the stress injection. This is likely due to the fact that, eventually, the LB detects the injected instance to be unhealthy and interrupts the current connections to redirect the load on the others. Then, when the connection is closed, the *distwalk* client ignores the remaining requests planned for the corresponding session and moves to the next, partially compensating the accumulated delay.

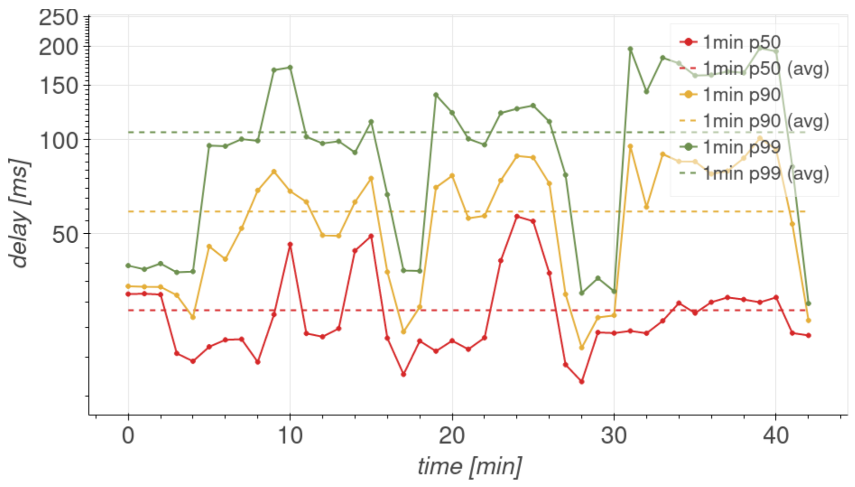
The disk bandwidth constitutes the major bottleneck in the resource saturation scenario. When an instance receives a higher volume of requests, the *distwalk* server process starts falling behind the expected schedules, the requests remain in the queue, and the response times start increasing. For instance, in Figure 7.4, we can see such a phenomenon occurring around minutes 14-16 and 23-26, during the second and third peaks of the workload, respectively. By looking at the system-level metrics in Figures 7.4a and 7.4b, we can typically observe that, in such cases, the workload profiles of the different instances do not closely follow the expected sinusoidal pattern, and start diverging. However, considering only such metrics, and assuming not to have any a-priori knowledge of the expected workload, we cannot exclude that such deviations are just noise. Furthermore, is even trickier to infer that a saturation phenomenon is occurring when the workload is mainly I/O-intensive, rather than CPU-intensive, since the actual bandwidth of traditional, rotational, hard drives depends on multiple factors, and it is not guaranteed to be always consistent. Therefore,



(a) CPU usage

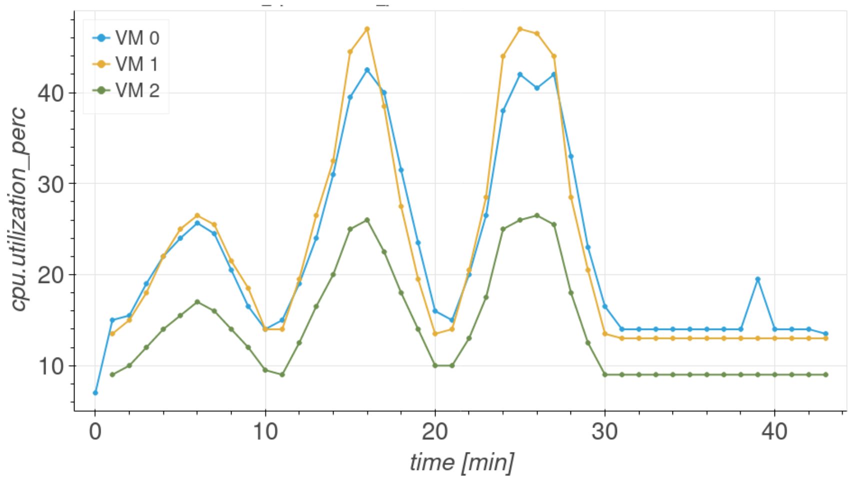


(b) Disk I/O activity

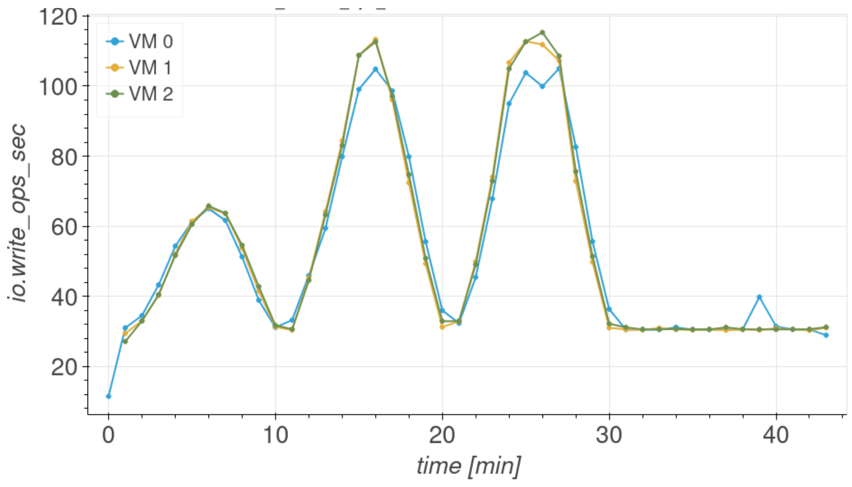


(c) Client-side response times

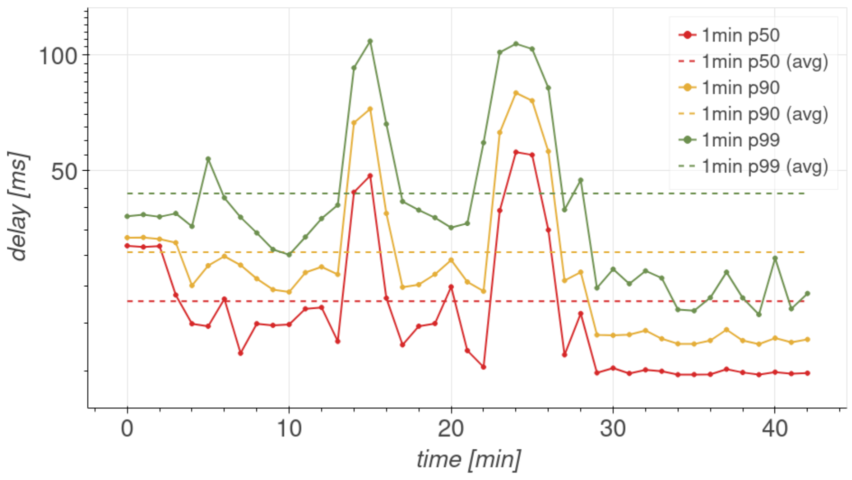
Figure 7.3: Faults generated by killing an instance of *distwalk*.



(a) CPU usage



(b) Disk I/O activity



(c) Client-side response times

Figure 7.4: Saturation of the disk bandwidth for *distwalk*.

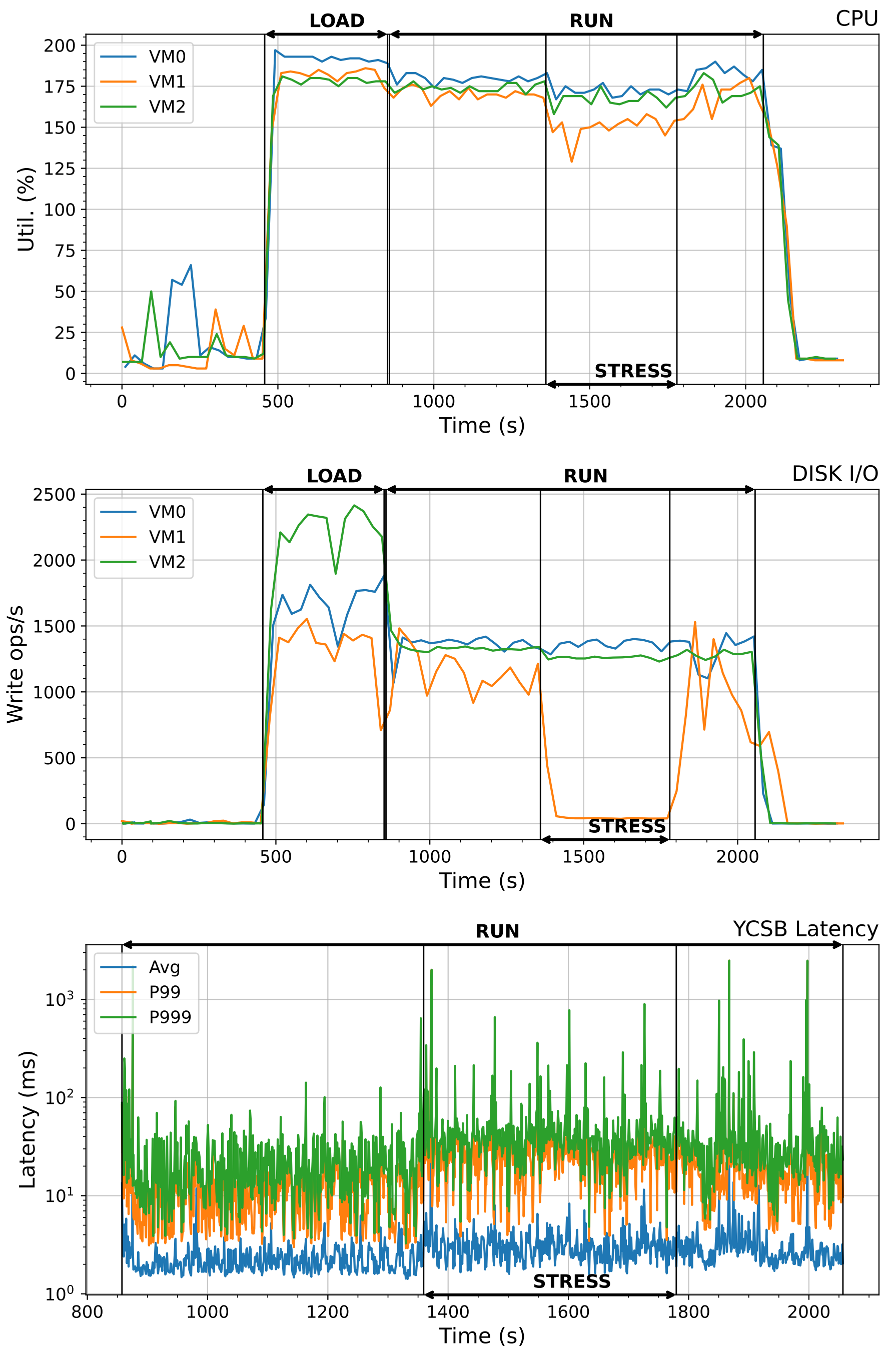
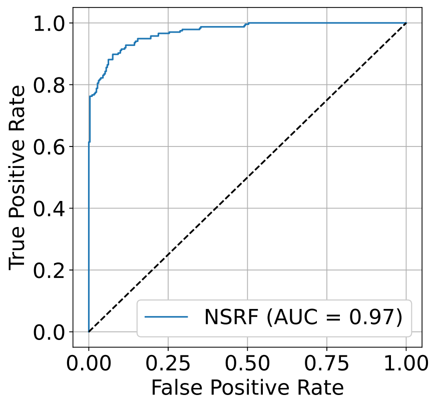


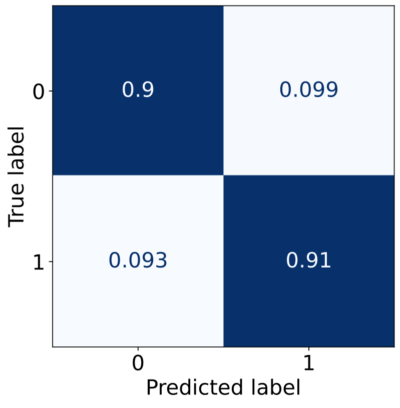
Figure 7.5: Interferences generated by *stress-ng* on Cassandra.

the most effective way to detect that a saturation phenomenon is occurring is by looking at the client-side response times. Indeed, in Figure 7.4c, we can observe peaks of ~80 ms in the p90 curve, during the aforementioned workload peaks. Similarly to the fault injection scenario, the impact of the disk saturation on the QoS is significantly smaller than what we observed for the stress injection.

Similarly to the *distwalk* application, also the Cassandra cluster was injected with anomalies during our experimental runs. For instance,



(a) ROC-AUC



(b) Confusion matrix

Figure 7.6: AD performance, computed on the *distwalk* test set (0 = anomalous; 1 = normal).

Figure 7.5 shows the measurements recorded during one of such runs, where we used *stress-ng* to generate interference, while the cluster, with replication level set to 3 and consistency level set to 2, was serving the load generated by YCSB, as explained in Section 7.4.3. In the figure, we highlighted: the LOAD phase, when YCSB loads the cluster with 1 million keys and their associated 1 KB values; the RUN phase, when YCSB imposes a constant target update throughput of 1000 ops/s; and the STRESS phase, when one of the Cassandra replicas undergoes heavy disk I/O interference from *stress-ng*. As in Figure 7.2, we can appreciate that, during the STRESS phase, the disk I/O activity of the affected instance drops significantly, with respect to the other members of the cluster. However, the effect of the interference on the CPU utilization is less evident. During the same phase, we can also observe the latency perceived by the YCSB client increasing consistently.

7.4.5 *Results*

After conducting several runs under different conditions, both with *distwalk* and Cassandra, we collected the corresponding CPU and disk I/O activity measurements and trained our models for the AD and classification steps. For both applications' datasets, separately, we held out the same portions of data to be used as training and test sets for the models. However, the two models were trained on different *views* of the same information (details in Section 7.3). In this way, the AD step can act as a filter and let the system trigger the (more costly) classification step only when it is deemed useful.

Synthetic Application - Anomaly Detection. As explained in Section 7.3, we decided to implement this step with MADI [219], using the NSRF variant. By preprocessing the collected data, we obtained a

Table 7.1: Performance metrics of the AD model, for each class of samples, computed on the *distwalk* test set (0 = anomalous; 1 = normal).

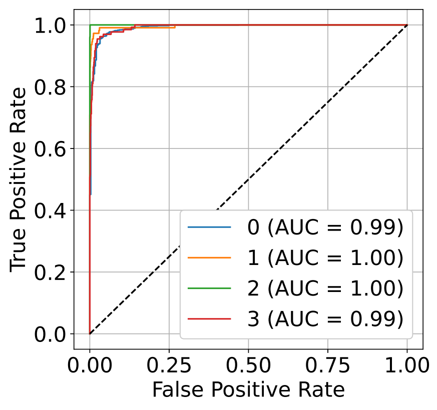
Class	Precision	Recall	F1 score
0	0.923	0.901	0.912
1	0.881	0.907	0.894

Table 7.2: Performance metrics of the classifier, for each class of samples, computed on the *distwalk* test set (0 = normal; 1 = stress; 2 = fault; 3 = saturation).

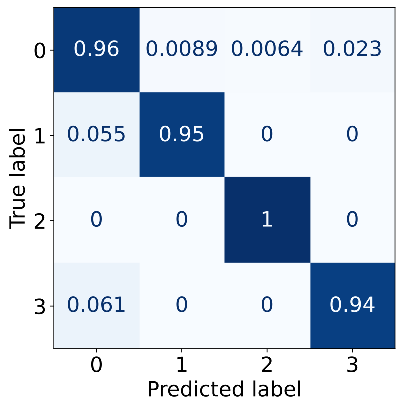
Class	Accuracy	Precision	Recall	F1 score
0	0.961	0.988	0.962	0.975
1	0.989	0.904	0.945	0.924
2	0.995	0.926	1.000	0.962
3	0.977	0.816	0.939	0.873

training set of 1087 and a test set of 528 input vectors, with shape 5×4 (as explained in Section 7.3.2). Such training and test sets contain a fraction of positive (normal) examples equal to 42.59% and 44.70%, respectively, while the rest is constituted by anomalous examples. Therefore, since NSRF is an unsupervised approach that assumes the input data to consist in mainly positive behavior, we trained it only on the 463 positive examples from the training set. After training NSRF, that typically takes just a few seconds on the CPU of our first physical host (see Section 7.4.1), without any specific acceleration settings, we ran the obtained model on the test set, this time using both positive and negative examples. Thanks to its negative sampling strategy, NSRF solves a *binary* classification task, and outputs the probability of an input belonging to the positive class. Such feature allowed us to produce the ROC curve plot [69] shown in Figure 7.6a, corresponding to an AUC of 97%. The ROC curve is a technique to visualize the evolutions of the TPR and FPR of the model, considering a variable classification threshold over the output probability of the model. In this way, we were able to select a sensible value for threshold t_p, to be applied on the output to determine the class of a given input, such that the FPR was below 10%, and the TPR was above 85% (i.e., corresponding to the upper-left region of Figure 7.6a). By setting $t_p = 0.594$, we obtained an accuracy of 90.34%, corresponding to the confusion matrix shown in Figure 7.6b. In Table 7.1, we also report other per-class performance measures.

Synthetic Application - Classification. As explained in Section 7.3, we decided to address this step, consisting in a multi-label classification task, by implemented our model using XGBoost [38]. By preprocessing



(a) "One-vs-rest" ROC-AUC

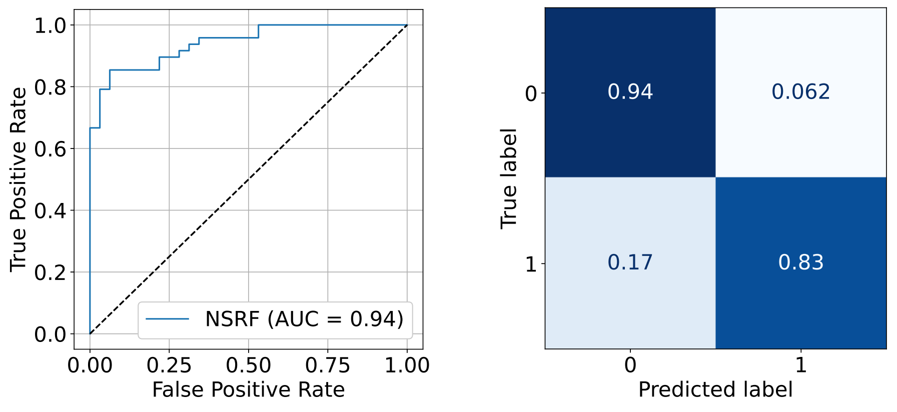


	0	1	2	3
0	0.96	0.0089	0.0064	0.023
1	0.055	0.95	0	0
2	0	0	1	0
3	0.061	0	0	0.94

(b) Confusion matrix

Figure 7.7: Classifier performance, computed on the *distwalk* test set (0 = normal; 1 = stress; 2 = fault; 3 = saturation).

the collected data, we obtained a training set of 3261 and a test set of 1584 input vectors, with shape 5×6 (as explained in Section 7.3.2). Note that the preprocessing employed for this model produces a dataset 3 times bigger than the one used for the AD model. This is due to the fact that, for each 5-minutes window on the raw data, such preprocessing produces a number of input samples equal to the number of active instances in the considered application (3, for our runs). Also, such preprocessing produces an inherently imbalanced dataset, due to the fact that, for each 5-minutes window, only one sample is marked as anomalous, given that we made sure not to inject multiple anomalies at once. Indeed, the training set is composed for the 78.44% by normal, for the 8.40% by stress-injected, for the 5.24% by fault-injected, and for the 7.91% by saturation examples. Similarly, the test set is composed for the 78.41% by normal, for the 6.94% by stress-injected, for the 6.31% by fault-injected, and for the 8.33% by saturation examples. However, XGBoost offers the capability to easily specify weights for each class, such that each one proportionally contributes to the gradient updates. After training the classifier, that typically takes less than 10 seconds on the CPU of our first physical host (see Section 7.4.1), without any specific acceleration settings, we used the test set to evaluate its performance. Given that XGBoost can be set to output the distribution of the probability of an input to belong to each of the available classes, also in this case we were able to produce a ROC curve plot, shown in Figure 7.7a. However, for the multi-label classification task, ROC curves can only be produced in a "one-vs-rest" fashion, i.e., each time considering a specific class against all the others (as they were a single one). Remarkably, all the generated ROC curves correspond to AUC values very close to 100%. Then, we applied the model on the test set and obtained an accuracy of 96.15%, corresponding to the confusion matrix shown in



(a) ROC-AUC (b) Confusion matrix

Figure 7.8: AD performance, computed on the Cassandra test set (0 = anomalous; 1 = normal).

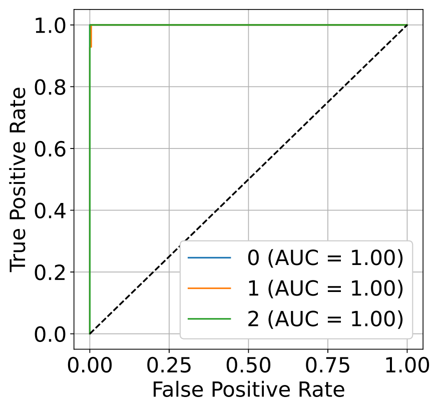
Table 7.3: Performance metrics of the AD model, for each class of samples, computed on the Cassandra test set (0 = anomalous; 1 = normal).

Class	Precision	Recall	F1 score
0	0.789	0.938	0.857
1	0.952	0.833	0.889

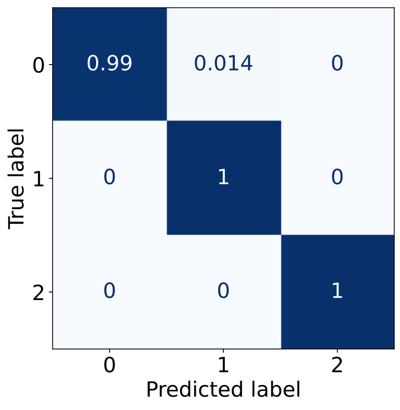
Figure 7.7b. In Table 7.2, we also report other per-class performance measures. Note that, in this case, also the accuracy can be computed in a "one-vs-rest" fashion.

Cassandra - Anomaly Detection. Similarly to the synthetic application, by preprocessing the data collected during Cassandra runs, we obtained a training set of 224 and a test set of 80 input vectors. Such training and test sets contain a fraction of positive (normal) examples equal to 57.14% and 60%, respectively, while the rest is constituted by anomalous examples. After training NSRF on positive samples only, we validated the model on the test set, and obtained the ROC curve plot shown in Figure 7.8a, corresponding to an AUC of 94%. By setting $t_p = 0.616$, we obtained an accuracy of 87.50%, corresponding to the confusion matrix shown in Figure 7.8b. In Table 7.3, we also report other per-class performance measures.

Cassandra - Classification. Similarly to the synthetic application, by preprocessing the data collected during Cassandra runs, we obtained a training set of 672 and a test set of 240 input vectors. The training set is composed for the 85.71% by normal, for the 6.25% by stress-injected, and for the 8.04% by fault-injected examples. Instead, the test set is composed for the 86.67% by normal, for the 5.83% by stress-injected, and for the 6.31% by fault-injected examples. After training the classifier, we validated it on the test set, and obtained the "one-vs-rest" ROC curve plot shown in Figure 7.9a. We also obtained an



(a) "One-vs-rest" ROC-AUC



(b) Confusion matrix

Figure 7.9: Classifier performance, computed on the Cassandra test set (0 = normal; 1 = stress; 2 = fault).

Table 7.4: Performance metrics of the classifier, for each class of samples, computed on the Cassandra test set (0 = normal; 1 = stress; 2 = fault).

Class	Accuracy	Precision	Recall	F1 score
0	0.988	1.000	0.986	0.993
1	0.988	0.824	1.000	0.903
2	1.000	1.000	1.000	1.000

accuracy of 98.75%, corresponding to the confusion matrix shown in Figure 7.9b. In Table 7.4, we also report other per-class performance measures, computed in a "one-vs-rest" fashion.

7.5 CONCLUSIONS

We proposed an ML-based strategy for *intelligent* cloud operations that consists of: *(i)* detecting anomalous conditions of a cloud application and *(ii)* identifying the corrective actions to be applied to faulty components. Both steps rely on ML models trained on operational data. Step *(i)* acts as a filter that allows the system to run the more expensive step *(ii)* on higher-resolution data only when needed. Our approach was validated using data exported from an OpenStack deployment. We used a workload generator sending traffic to a load-balanced group of Nova instances, resulting in CPU and disk I/O activity on the instances, and injected different types of anomalies that we could recover from, by applying precise corrective actions. For *(i)*, we trained an AD model (specifically, MADI [219]) on aggregated cluster data, such that it could even generalize to variable-sized groups of instances. On the respective test sets, for the synthetic application, we obtained a

ROC-AUC of 97% and an accuracy of 90.34%, while, for Cassandra, we obtained a ROC-AUC of 94% and an accuracy of 87.50%. For *(ii)*, we trained a supervised classification model, based on XGBoost [38], on a combination of spatially-aggregated and raw instances data, such that it could better compare the behavior of an individual instance with respect to its group, and associate a corrective action to instances separately. On the respective test sets, for the synthetic application, we obtained an accuracy of 96.15%, while, for Cassandra, we obtained an accuracy of 98.75%.

Part IV
QUALITY-AWARE DEVOPS

In this part, we shift our focus from operations to also consider the *development* part of the DevOps cycle. We propose a novel approach, based on LLM, to analyze declarative deployment code in order to automatically detect possible architectural problems, and provide QA-related recommendations.

8 LARGE LANGUAGE MODELS FOR DECLARATIVE DEPLOYMENT CODE ANALYSIS

8.1 INTRODUCTION

During the last decade, cloud technologies have been evolving at an impressive pace, such that we are now living in a *cloud-native* era where developers can leverage on an unprecedented landscape of advanced services to build highly-resilient distributed systems, providing compute, storage, networking, load-balancing, security, monitoring and orchestration functionality, among others. To keep up with this pace, development and operations practices have undergone very significant transformations, especially in terms of improving the automations that make releasing new software, and responding to unforeseen issues, faster and sustainable at scale. The resulting paradigm is nowadays referred to as *DevOps* [4].

QA is obviously a fundamental part of the DevOps cycle. However, the complexity of modern cloud frameworks and services makes a developer's job unprecedentedly hard. On top of that, development teams are typically composed by persons with very diverse backgrounds, and varying levels of expertise. As a team, this makes adhering to best practices everything but straightforward, because transferring knowledge from experts to novice members takes a lot of time. Therefore, in line with the DevOps philosophy, automating this process as much as possible seems the right approach. Indeed, there exist a vast amount of tools that provide (static) code analysis functionality, and that can be seamlessly integrated in existing CI/CD pipelines to address QA concerns. However, given the impressive abundance of data generated throughout the DevOps cycle, applying ML techniques in this context seems a promising path towards providing developers with high-quality feedbacks and recommendations, automatically.

When developing a cloud-native application, the definition of its *deployment* plays a fundamental role. Modern cloud management frameworks, like Kubernetes and OpenStack (two of the most well-known open-source and widely adopted projects), typically offer at least an IaC solution (e.g., deployment manifests and Heat templates, respectively). Such a mechanism allows for specifying the desired properties and the relations among the components of the deployment via *declarative* code, that can then be versioned and treated in the same way as the code that implements the actual application logic. It is obviously very important to follow best practices when, e.g., specifying a Kubernetes deployment manifest, as failing to do so may lead the applications to experience many types of issues [132, 159]. Static analysis tools for manifest files, like for instance Polaris[1] or Kubesec,[2] allow for mitigating the risk that such issues may actually occur. However, they are typically designed to run relatively simplistic checks, that do not take into account complex design patterns.

In this work, we propose an approach to declarative deployment code analysis based on LLMs, that can automatically provide QA-related recommendations to developers, based on established best practices and design patterns, building on top of standard (static) analysis approaches. To the best of our knowledge, our approach is novel, in the sense that we did not find in the research literature any other proposal to specialize LLMs on deployment code to specifically address QA-related concerns. Also, while we mainly focus on deployment code, it is interesting to consider that this information could eventually be integrated with the other available data sources in the DevOps cycle to consider a more comprehensive picture, like: version control system history, code review feedbacks, tests measurements and logs, etc.

8.1.1 *Chapter Organization*

This chapter is organized as follows. Section 8.2 provides an overview of the existing related works in the space of code analysis with LLMs. Section 8.3 presents the main features of our proposed approach and the prototype ML pipeline we implemented. Section 8.4 presents the results of our preliminary validation on a set of Kubernetes manifest files exported from a Nokia Bell Labs (NBL) repository. Section 8.5 concludes the chapter and provides indications for future research directions.

8.2 RELATED WORK

In this work, we propose the use of NLP models to detect architectural smells and issues in declarative deployment code. Indeed, LMs are nowadays extensively used in practice to analyze and generate source code [143, 215]. In particular, we focus on LLMs, that are models based on the *transformer* architecture [240], consisting of *millions*, or even *billions*, of learnable parameters. During the last years, in fact, this class of models has been gaining a lot of attention from the

1 https://www.fairwinds.com/polaris
2 https://kubesec.io/

research community, due to their fascinating emergent properties like *unsupervised multitask* [186] and *few-shot* [22] learning.

In [266], the authors propose an LLM-based approach to automatically fix textual and semantic merge conflicts in a version-controlled code-base. Their approach leverages entirely on few-shot learning, and exhibits remarkable performance without requiring fine-tuning. In [35], the authors propose *Codex*, a GPT [185] model extensively fine-tuned on open-source code retrieved from GitHub, that exhibits remarkable performance in generating source code when prompted with the corresponding textual description. Similarly, in [91], the authors propose an LLM-based approach to code generation that takes into account both the code already written by developers and their *intent*, expressed in plain natural language. In particular, such model is empirically validated on Python code generation for data science applications. In [217], *KerasBERT* is proposed. Such model is trained on a considerable amount of code examples, notebooks, blog posts and forum threads regarding the Keras deep learning framework, to provide an automatic tool to analyze and generate documentation for related code snippets. The authors of [99] propose *Jigsaw*, an approach based on program synthesis techniques, to post-process the source code generated by specialized LLMs in order to provide quality guarantees.

The work presented in [231] demonstrates how LLMs can also be used for detecting software vulnerabilities. Indeed, the authors provide the results of an empirical analysis, conducted on vulnerability datasets for C/C++ source code, showing how LLMs outperform other neural models like those based on LSTMs and GRUs. Similarly, the authors of [61] propose a malware detection mechanism that leverages on a combination of LSTM and LLMs to discover malicious instructions in assembly code.

In [141], the authors investigate on the reasons behind the emergent capability of LLMs to learn code syntax and semantic. In particular, they rely on ASTs and static analysis to deeply understand the role that the self-attention mechanism plays in learning the dependencies among code tokens. On a related note, in [246], the authors approach the problem of interpreting pre-trained LLMs for code analysis. Remarkably, their results show that, in a transformer architecture, the code syntax structure is typically preserved in the intermediate representations of each layer and, as a result, that such LLMs are able to induce ASTs.

The authors of [208] empirically demonstrated how LLMs can be successfully used to generate, and explain, code for programming exercises that is both novel and reasonable. On the other hand, in [223], the authors provide evidence that the same type of models heavily rely on contextual cues (e.g., natural-language comments, or function names) and that, by masking such information, their summarization performance drops significantly.

The works referenced in this section generally use LLMs to either provide general-purpose code generation solutions (e.g., [35, 91]), or realize code analysis tools for specific programming languages and/or frameworks (e.g., [217, 231]). However, none of them proposes an approach to detect, or recommend, the usage of specific best-practices and high-level design patterns, that are very important for QA. Furthermore, none of the aforementioned works specializes LLMs to analyze declarative deployment code that, nowadays, is ubiquitously used to configure modern cloud environments. Therefore, we believe our work addresses a very relevant problem and constitutes an innovative solution.

8.3 PROPOSED APPROACH

Our work focuses on the analysis of Kubernetes deployment manifest files. In particular, our goal is to provide non-expert developers with recommendations regarding the (mis-)usage of relevant Kubernetes architectural patterns (e.g., the *Operator* pattern). We identified a set of fundamental features that such a tool should have in order to achieve our goal:

- F1: Classifying *good*- and *bad*-quality manifests.
- F2: *Explaining* which characteristics contribute the most to the outcome of the classification.
- F3: *Pinpointing* design smells and issues, and possibly recommending a suitable fix.
- F4: Leveraging on the *relations* specified among the components to detect highly complex architectural patterns.

We assume that a (possibly small) set of *annotated* manifest examples is available. This is reasonable to assume in a scenario where DevOps teams conduct code reviews, such that useful annotations could even be automatically extracted from the platform used for such activities. Therefore, implementing F1 can be approached as a *supervised* learning problem. In this context, the notions of *good* and *bad* can be interpreted in many ways, also according to the nature of the available annotations. An expert developer can generally tell “at a glance” whether a manifest seems to be poorly written or not. Although, there are possibly many reasons why a specific manifest is problematic. Therefore, it may not be actually useful to treat this problem as a simple *binary* classification task. Indeed, both F2 and F3 are concerned with *augmenting* the quality of the recommendations. However, while F2 refers to the possibility to apply specific techniques [7, 92, 230] to better interpret the output of

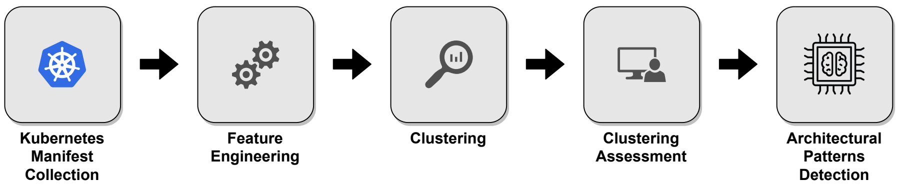

Kubernetes Manifest Collection
Feature Engineering
Clustering
Clustering Assessment
Architectural Patterns Detection

Figure 8.1: The proposed ML pipeline.

an arbitrary model, F3 entails that such a model should be able to solve a more complex task than a simple classification, in order to provide the end user with fine-grained recommendations. Implementing both F2 and F3 inherently requires a trade-off to be made between the interpretability and the power/complexity of the underlying ML model. Similarly, F4 is concerned with endowing the model with the capability of detecting more convoluted design patterns, that are not easily discoverable when looking at resources in isolation. Given the set of desired features, and the fact that the input data mainly consist in source code (or text, in general), we believe that LLMs are the most suitable tools to address our problem.

8.3.1 *ML Pipeline*

In order to realize the tools described in Section 8.3, we propose the ML pipeline that is synthetically described in Figure 8.1. Nowadays, Kubernetes is one of the most used cloud orchestration framework, and definitely among the most important projects backed by the CNCF. Therefore, it is very easy to find large open-source collections of high-quality deployment manifest files, like by considering those from CNCF graduated and incubating projects. On top of that, we have access to a vast number of (confidential) deployment manifests developed by NBL research teams and business units for their products.

In this case, the main data quality-related problem is represented by the scarcity of annotations that could be used to train supervised ML models. To overcome this limitation, we propose to use an (*unsupervised*) clustering approach to try and detect significant similarities among the manifests. For instance, it is possible to use *HDBSCAN* [28], a density-based clustering algorithm that uses a hierarchical approach to discover clusters with non-uniform densities. Unlike the majority of standard clustering approaches, HDBSCAN does not have to be provided with a number of desired clusters to be discovered, and can also automatically detect clusters with non-convex shapes and outliers. Such features make HDBSCAN very *versatile* and suitable for a diverse range of applications. However, in general, clustering approaches are not designed to handle textual data directly, so it is crucial to establish a proper feature engineering process such that manifests are transformed into appropriate feature vectors. For instance, this can be accomplished by using standard *tf-idf* scores [189] that, for each document, provide an indication of the relevance of each word in the vocabulary. Specifically, *tf-idf* considers both the *term frequency*, with respect to the specific document, and the *inverse document frequency*, with respect to the whole set of documents (i.e., the *corpus*). Such components assign more relevance to terms appearing more frequently at document-level, and to those appearing less frequently at corpus-level, respectively. In this way, *locally-meaningful* terms tend to outweigh those that are extremely frequent, and that generally carry less information. Although there exists more sophisticated one, this technique is still a very *efficient* approach to project a corpus onto a vector space, and is often sufficient to provide an overview of how the documents are related to each other. Given that the resulting vector representations are typically very high-dimensional, the results of the clustering process require embedding techniques to be properly visualized. For instance, it is possible to use *t-SNE* [142], a dimensionality-reduction algorithm that is designed to project high-dimensional spaces onto 2 or 3 dimensions, while retaining pair-wise spatial relations among the data points as much as possible. This approach works by estimating a probability distribution over pairs of data points in the original space, and then by generating a similar distribution in the target space, minimizing their Kullback-Leibler divergence via gradient descent. While this process is very computational intensive with respect to similar approaches, the capability of t-SNE to preserve local structures of the original data makes it a very effective tool for exploratory high-dimensional data visualization. In this way, an expert can manually inspect some representatives from the discovered clusters and provide initial annotations.

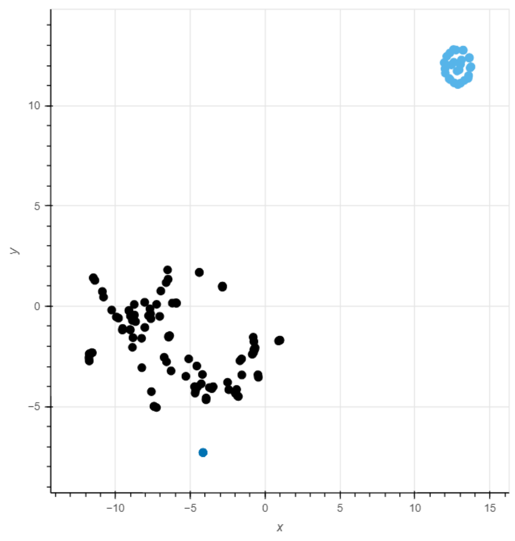
Depending on the actual task to be solved, the annotated data must then be transformed in a way that they can be consumed by a supervised learning model. In the case of LLMs, there exist two main strategies that can be used to solve a supervised learning task: *fine-tuning* or *few-shot* learning. LLMs generally require a very large amount of resources to be trained, due to their impressive size, that directly affects their computational complexity, and the (humongous) amount of textual training data needed to make them exhibit the properties they are famous for. Therefore, it is typically too expensive to train them from scratch. However, provided that a checkpoint of the weights of such a model is publicly available, it is still possible to benefit from them to solve specific tasks, even though the original training process was optimized for another type of task and/or was conducted on textual data unrelated with the application domain. Indeed, one could choose the *fine-tuning* option, that is an example of *transfer learning*, and use the original model as the initial part of a bigger architecture. The remaining part is typically optimized for solving the problem at hand (e.g., a sequence classification task), and trained using domain-specific textual data. Such an approach may generally obtain impressive performance even though the amount of available data is small. On the other hand, LLMs trained for *causal* language modeling (i.e., open-end text generation) are also capable of *few-shot* learning. This property consists in such a model being able to extrapolate how to solve a given learning task, provided that its description and *a few* input-output examples can be specified as a textual *prompt* (see the examples provided in Appendix A). In this way, one does not even need to develop (and allocate resources for) a training pipeline, as the LLM is only used in inference mode.

8.4 PRELIMINARY EXPERIMENTS

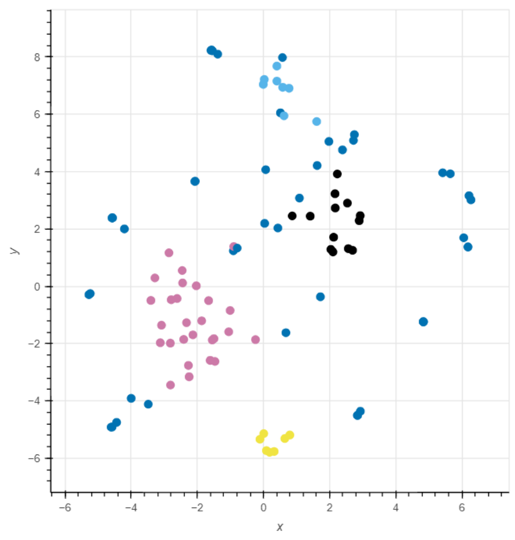
In order to validate the ideas presented in Section 8.3, we developed some prototypes of the different parts of the proposed pipeline, and conducted some preliminary experiments considering a simplified version of our problem. Specifically, we gathered a set of ~100 manifest files from internal NBL projects and ran our clustering pipeline on them. While our initial intent was to try and see whether the clustering output exposed interesting similarities that could be used to obtain a tentative data labeling, this step was particularly useful to filter out some *noise* from our data. Figure 8.2a shows our initial clustering results. As described in Section 8.3.1, we obtained *tf-idf*-based representations of the manifests and used PCA to get the *top-10* dimensions, to limit the amount of data to be fed to HDBSCAN. After using t-SNE to project the clustered vectors in a 2D space, we observed that our data included a (strangely) compact and well-delimited cluster of manifests (in the top-right corner). Upon inspecting the corresponding manifests, we realized that their contents were not adding valuable information to our analysis. Figure 8.2b reports the result we obtained by re-running the clustering pipeline after filtering out the uninteresting manifests.

Given the limited dimension of our data sample, we were not able to use the clustering results to derive interesting annotations at this stage. Therefore, we decided to run Polaris on our manifests and considered the output of the (boolean) `cpuLimitsMissing` check, that reports whether CPU usage limits were correctly specified for Kubernetes resources like `Deployment` and `Service`. In this way, we were able to quickly obtain an annotated dataset, that allowed us to reduce our problem to a binary sequence classification task and conduct some experiments with LLMs. Given the impressive computational complexity of such models, we accelerated our experiments using the following GPUs:

- NVIDIA Quadro RTX 6000 (Turing), 24 GB memory;
- NVIDIA Quadro RTX 8000 (Turing), 48 GB memory.



(a) before



(b) after

Figure 8.2: Results of the clustering process, before and after filtering out the uninteresting manifest files. The clustered manifests are projected onto a 2D space by using t-SNE (i.e., the axes do not directly refer to any specific feature).

In order to do that, we extensively leveraged on the HuggingFace *transformers* library [254] and the pre-trained model checkpoints available on the associated model hub.[3] Essentially, we focused our experiments on two LLMs: *GPT-2 (medium)*[4] and *GPT-J-6B*.[5]

The medium-sized version of GPT-2 [186], consists of 355M parameters, and accepts a maximum of 1024 tokens as input. During our few-shot learning tests, such an input token limit allowed us to provide just a couple of examples, as we had to save enough space for the actual input to be processed (see Appendix A). Furthermore, given that the kind of outputs we obtained were not related in any way to the labels we specified in the prompt, we concluded that this model is not particularly suitable for declarative code analysis via few-shot learning. This is likely due to the fact that the model was trained on English natural language only, and probably never observed any code example. However, as we were able to run fine-tuning jobs even on our smaller GPU, we believe that this model could be easily fine-tuned on a bigger declarative code training set and yield significant results, similarly to what done in [91].

As the name suggests, GPT-J-6B is instead a 6B parameters model, inspired by the success of GPT-2/GPT-3, developed and open-sourced by EleutherAI.[6] Furthermore, such model is trained on *The Pile* [76], an 800+ GB open dataset containing a very diverse set of textual documents, including source code. Given the significantly bigger size and input token limit (2048), our few-shot learning tests were rather successful. We observed that the model was indeed able to understand the specified classification task and output a correct label in most of the cases. Furthermore, we did not have any problem with running the `float16` revision of the model on our smaller GPU, as the model took only ~12 out of the available 24 GB of memory. It is also quite impressive that we obtained comparable results using the *8-bit quantized* version of the same model, that almost *halves* the memory requirements, by leveraging on a recently-added feature of the *transformers* library, whose details are described in [62]. However, using the few-shot learning strategy still imposes great limitations in terms of the amount of training examples that the model can observe. This way, the ability of the model to generalize is in turn quite limited. At the time of writing, the 8-bit quantized version seems not to support fine-tuning. Therefore, for our fine-tuning tests, we used the `float16` revision. Although, to avoid getting CUDA out-of-memory errors on our GPU setup, it was necessary to use *DeepSpeed* [195], a framework that leverages on Zero Redundancy Optimizer (ZeRO) [190, 191] to optimize model training memory footprint, either on a single or multiple GPUs, at the expense of speed. Setting up effective fine-tuning

3 https://huggingface.co/models
4 https://huggingface.co/gpt2-medium
5 https://huggingface.co/EleutherAI/gpt-j-6B
6 https://www.eleuther.ai/

jobs, and properly assessing the resulting model performance, is still a work in progress. Contrary to few-shot learning tests, they require more, and better annotated, input data then the limited sample we were able to generate from one of the repository of internal projects at NBL.

8.5 CONCLUSIONS

We proposed a method for analyzing declarative deployment code (specifically, Kubernetes deployment manifest files), such that non-expert developers can benefit from design patterns recommendations. To the best of our knowledge, our proposed approach is a novel way to address QA-related issues by specializing LLMs on declarative deployment code analysis. We conducted a preliminary validation of our ML pipeline on a simplified version of the problem, that shows that LLMs are indeed a viable and promising option for achieving our end goal.

Part V

CONCLUDING REMARKS

9 CONCLUSIONS

Over the last decade, an ever-growing number of organizations have been adopting cloud computing, either private or public, to develop many diverse types of applications and services, with varying levels of criticality. Therefore, it is fundamental for service providers to make sure they are up to the agreed level of performance, and to continuously improve their infrastructures in order to quickly react to, and possibly anticipate, system disruptions. However, the traditional over-provisioning strategies, used by service providers to ensure that SLAs are not breached by components failures and/or workload fluctuations, are inherently cost- and energy-inefficient. Furthermore, they do not provide any actual guarantee when facing unforeseen anomalous events that, due to the complex interactions existing in this type of environments, can quickly propagate and affect large portions of the entire systems. Therefore, for the operation teams that administrate such huge and complex systems, it is crucial to introduce novel and more sophisticated analysis techniques, that can enable a more *proactive* approach to data center operations. The unprecedented availability of data, collected at each level of a modern cloud infrastructure (e.g., physical hosts, networking equipments, virtualized instances, applications, etc.), indeed allows for adopting incredibly effective data-driven methods like, for instance, those proposed in the ML research field. Such techniques are expected to be extremely effective at filling the gap left by relying only on traditional model-driven approaches that, alone, quickly become inadequate at dealing with the complexity of the interactions among system components, at scale.

The various works presented in this thesis demonstrate the effectiveness of data-driven techniques in enabling more intelligent, and autonomous, data center operations. For instance, we addressed several problems that represent critical pain-points of NSPs. Indeed, in Chapters 3 and 4, we presented our proposals for implementing high-performance anomalous VNF behavior detection [50, 125, 126]. Such solutions aim at supporting human operators by providing actionable feedbacks that can complement their own analysis, offering a more comprehensive view of the evolution of the health status of the monitored system over an extended period. We used SOMs to implement the proposed approaches, such that they are sufficiently light-weight, computationally-wise, and do not require labelled data to classify anomalies. To further improve processing time, we also contributed our own, open-source, GPU-accelerated implementation of SOMs [148]. While validating this work, we identified some open questions that still need additional investigations. For instance, the proposed AD technique has a number of hyperparameters (e.g., SOM grid size, parameters, and thresholds described in Section 3.3) that have to be set. A grid search can be used for this purpose, but it requires a non-negligible processing time, as possible configurations can easily grow in the range of tens or hundreds. In order to select the best SOM hyperparameters, the various analysis runs should be compared with one another using an automated and quantitative assessment method. This cannot be simply done based on the SOM quantization error, as it would decrease increasing the SOM size, driving the choice towards excessively large networks. It would be interesting to investigate on using average silhouette width [201] to drive this type of search. Also, another promising path is to combine our approach with DL methods for time-series classification [98, 107]. An interesting approach could be using a SOM to produce a more compact and discrete representation of a time-series autoencoder, as presented in [71]. In Chapter 5, we presented our work on time-series forecasting techniques applied to NFV operational metrics [49]. Such methods are particularly important for NSPs, as they can be used to estimate the expected resource utilization over a specific time-horizon, offering a guide for capacity-planning activities. We provided a thorough analysis of several, diverse, time-series forecasting algorithms, ranging from classical statistical models to more sophisticated neural architectures, highlighting their advantages and disadvantages, both in terms of accuracy and operational costs. Our analysis can be easily extended by considering additional datasets extracted from the Vodafone NFV data centers, possibly increasing the number of considered metrics. Also, a useful extension would be to design improved ML techniques that can process additional information besides the operational metrics to be forecasted like, e.g., rough traffic volume forecasts manually produced by internal analysts quarterly. On a related note, it would be interesting to investigate the applicability of topology-aware prediction techniques [157], that look quite promising in the context of NFV metrics forecasting. The aforementioned tools for AD and forecasting have been developed and validated using real data, and according to actual operational needs, of the *Vodafone* network operator. Such collaboration highlights the relevance of our contributions to service providers that operate large-scale cloud infrastructures. Indeed, our tools are now actively used in production environments to support the daily activities of the Vodafone NVI operations team.

Another stream of works presented in this thesis focused on more general, and fundamental, problems in cloud operations. For instance, in Chapter 6, we presented our approach to predictive autoscaling [123]. Elasticity-control is a long-standing issue in the cloud computing space but, nonetheless, it is still a very active, and interesting, research area. Remarkably, while a humongous amount of diverse data-driven approaches have been proposed during the years, we observed that very few provided actual, open-source, implementations. This is not only a huge problem in terms of reproducibility, but also contributes to such data-driven techniques not being easily transferrable from research labs to actual production environments. Therefore, we decided to focus our efforts on designing a general architecture that could easily enable the usage of advanced predictive analytics in elasticity-control loops. We implemented our approach as an open-source extension to the Monasca monitoring framework for OpenStack. Such an approach allows cloud orchestrators to apply time-series forecasting techniques to estimate the evolution of relevant metrics, and take decisions based on the predicted state of the system. Human operators are able to fully customize what underlying ML models are used and how the monitoring data are ingested, to realize robust, and light-weight, elasticity-control policies. Further extensions are possible to better integrate our proposed forecasting component within OpenStack. For instance, one could provide a number of standard predictor implementations that can easily be deployed by an operator—that is not necessarily an ML expert—through the OpenStack CLI. Regarding the orchestration logics, additional ML techniques (e.g., RL) could be explored to experiment with alternative scaling policies, rather than only relying on the threshold-based approach. However, for them to be dependable, learning-based policies cannot be delivered as black-boxes. Therefore, it would be useful to also provide human operators with the ability to query the models to get explanations about their outputs [135, 199], such that they can troubleshoot and fix possible erroneous behaviors. Regarding the model training, a useful extension would be to provide means for automatic periodic re-training on fresh data and *concept drift* detection, to trigger model updates whenever the current version starts exhibiting performance drops, e.g., as the authors of [111] did. In this regard, our architecture would certainly benefit from integrating continual learning techniques [46]. It would be interesting to also conduct a deeper validation of our architecture by considering additional data sets from real production workloads. In this way, it would be possible to address scalability issues that might arise in massive deployment scenarios, with thousands of predictive elasticity loops that control different services, possibly throughout a Cloud-Fog-Edge architecture [154]. In this regard, a promising possibility is to leverage on Monasca's scalable analytics processing architecture, that is based on Apache Storm. In Chapter 7, we presented our work on *intelligent* operations (unpublished work, currently under review) where, with respect to [123], we broadened our scope and considered, beside elasticity, a set of additional anomalous scenarios that operations teams typically face during their activities. Indeed, our approach offers to human operators a way to embed their experience into a model able to *recommend* the most suitable course of action, given a particular observed unhealthy condition of the monitored system. The approach consists in an ML pipeline, composed by two models in sequence, to detect anomalous patterns, and recommend specific corrective actions that proved to be successful, in the past, at restoring normal conditions. Mapping anomalous scenarios to the corresponding corrective actions can be done, e.g., by correlating the impressive amount of information stored in issue tracking systems and/or post-mortem documents to system- and application-level measurements. Remarkably, in this way, it is possible to implement very effective autonomous operations frameworks on top of supervised learning models (e.g., GBTs) that, in general, exhibit better performance than their unsupervised counterparts, in exchange for having to deal with data labelling issues. Also in this case, we validated our proposed approach on OpenStack, also using a Cassandra deployment as a test application, to highlight the relevance of our work for modern data center operations. It would be useful to conduct a more thorough study to better (quantitatively) compare our approach to existing alternatives. In this research area, such an activity is rendered particularly difficult by the general lack of open-source implementations readily integrated within frameworks like OpenStack. For the classification, we opted for supervised-learning. However, it would be interesting to apply unsupervised or weakly-supervised approaches to our problem, to possibly weaken the dependency on labelled data. Also in this case, a very useful extension would be to integrate automatic model retraining, to counteract the disastrous effects of concept drift, and model explainability techniques in the end-to-end pipeline. Indeed, guaranteeing a sufficient level of robustness to fluctuations and interpretability is of utmost importance, as cloud operations are a scenario where any type of automation should be highly dependable and auditable. Furthermore, packaging our approach as a proper OpenStack service would be required to offer a reliable, open-source, solution for intelligent cloud operations to a wide audience.

Finally, in Chapter 8, we presented an additional stream of work, where our focus shifted from operations to also include the *development* part of the DevOps cycle. Indeed, our work consists in an approach to specialize LLMs for declarative deployment code analysis [127]. This work was conducted in collaboration with *Nokia Bell Labs* (NBL), that provided us with a real problem faced by their multidisciplinary research teams, and a collection of Kubernetes manifest files exported from a repository of internal projects. Our goal was to create a tool to support research teams in their QA activities, such that they can produce robust deployment specifications. In general, this is far from being trivial, and it is also particularly time-consuming for domain experts to enable every member of the team, with potentially very diverse expertise, to provide high-quality contributions. We developed a prototype of our proposed ML pipeline, based on LLMs, to automatically provide QA-related recommendations to developers, such that they can benefit of established best practices and design patterns. Our approach was empirically evaluated on the collection of Kubernetes manifest files provided by NBL. It would be useful to extend the proposed approach beyond recommendations that can be obtained with standard static analysis tools (e.g., Polaris), by considering more convoluted design patterns and architectural smells [31, 161], that involve a potentially large number of Kubernetes resources, possibly taking into account also security concerns [183]. In these regards, framing the problem as an *extractive question-answering* task seems a promising avenue. However, it would also be interesting to investigate the feasibility of a hybrid approach that combines LLMs with other types of models that can leverage on existing structures (e.g., relations among Kubernetes resources) in the input data, like GNNs [10]. Also, it would be interesting to conduct a more thorough comparison of different types of LLMs and their usage modes (e.g., few-shot learning vs fine-tuning vs re-training). On a related note, exploring methods for deriving more compact representations of the inputs seem a promising avenue to work around the maximum input tokens limit (e.g., YAML vs JSON encoding; optimize tokenizers for declarative code, similarly to the approach used for the natural language-guided programming model proposed by [91]). As Kubernetes is not the only cloud computing framework that leverages on declarative code for its configuration, it would be useful to generalize our approach to other forms of deployment configuration files like, for instance, OpenStack HOTs. Finally, we believe it would be interesting to integrate active learning [198] techniques into our approach, to facilitate expert architects with sharing and embedding their knowledge into the underlying model.

9.1 FUTURE RESEARCH DIRECTIONS

In what follows, we outline open problems and research directions that are potentially interesting for future studies and applications on data-driven operations support.

9.1.1 *Anomaly Detection*

AD is among the principal concerns of data center administrators and also serves as enabling technology for many other types of analysis. For instance, the capability of detecting suspect performance degradation is fundamental to the purpose of establishing automated proactive strategies to minimize the risk of SLAs violations [209]. In this space, there is a continuous need for performant and scalable algorithms that can tackle massive high-dimensional datasets [222] and return feedbacks in a timely-manner, as usually such information are only useful if one has enough time to react to such unexpected events [248]. Addressing this problem is also particularly challenging when dealing with high-dimensional and possibly incomplete sequences [17]. Also, the inherent scarcity of outliers and lack of ground truth are common issues in real scenarios, and makes the evaluation of such algorithms very difficult. Therefore, in order to improve the current state of the art, devising strategies for effective data augmentation might be included among the research items of the project. In particular, methodologies based on *unsupervised* (or *semi-supervised*) ML algorithms should be investigated. Such approaches are particularly useful in case of lack of considerable amount of labeled data and could be also implemented using *active* learning strategies, with the additional benefit of incorporating experts feedbacks in the learning process [56, 180].

DL enables the possibility of overcoming the shortage of data for a specific task by transferring the knowledge obtained solving other (*related*) tasks, i.e., *transfer learning*. Using such technique for time-series data is a promising research avenue [68, 268]. For instance, in the NFV context, it is not unusual to have the very same VNF (i.e., the same components of the particular service chain) to be deployed with (possibly very) different configurations in different data centers of the same organization. In such cases, the related performance (e.g., in terms of latency, throughput, etc.) can differ a lot from each other even though the underlying infrastructure do not. It might be interesting to investigate on the effectiveness of transfer learning in this scenario, such that the operations of an individual data center can benefit from the *lessons learned* by others.

9.1.2 *Intelligent Auto-Scaling & Resource Allocation*

Elasticity is among the primary reasons for the success of the cloud computing paradigm. Auto-scaling is in fact one of the most attractive features of a cloud environment, consisting in the capability of an application to self-adapt its resources (both at hardware- and software-level), in response to some changes in its operating conditions, to optimize QoS [37, 184] while containing costs. Classical auto-scaling mechanisms are typically based on *prescriptive* analytics and simple automation rules that, for instance, kick in as soon as a predefined threshold is reached. In this way, auto-scaling can be interpreted as a MAPE (Monitoring, Analysis, Planning, and Execution) control problem [109], where each of the phases entails a number of challenges that make it a particularly hard one. However, in order to establish proactive operations strategies it is desirable to leverage on reliable *predictive* methods [204]. Such an approach is far from trivial to implement, as it should have an extremely low footprint, not to interfere with the actual production workloads, while also being sophisticated enough to find the optimal strategy to scale complex applications, whose performance bottlenecks are not necessarily easy to identify. The latter consideration especially applies when dealing with heterogeneous applications, for instance, made in part of some components that are not scalable by design (e.g., relational databases), where acting on the elastic components could be even useless towards the goal of solving performance issues.

An important aspect that should be given more relevance when designing auto-scaling mechanisms is the possibility that cloud applications can be deployed across many geographically distributed data centers. This is nowadays a common practice in cloud-native development to address latency-related concerns (i.e., trying to minimize the response time by serving users requests in the closest data center) or just to improve the overall reliability and availability of the application, making it robust to data center-wide disruptions. In this scenario, though, the auto-scaling problem inevitably grows in complexity as a global coordination becomes necessary to take into account also the reliability assessments [138] of the individual regions in the search for an optimal scaling plan.

9.1.3 *Root-Cause Analysis*

RCA is among the toughest activities conducted by data center operators. It consists in investigating on the prime causes of failures and anomalous behaviors of a system with the aim of formulating hypothesis and (hopefully) explanations that can guide the implementation of corrective actions and prevent similar faults to occur again. This kind of tasks are extremely time-consuming, as they typically require a deep understanding of the specific infrastructure, the services that are deployed on it, and distributed systems in general. On top of that, the typical scale of the systems an operator has to deal with, including not only the components internal to the data center but also the many others distributed at the edge [90, 237], and the incredibly fast pace at which their configurations are updated make troubleshooting issues a really complex task. Such task is progressively becoming impossible to conduct with traditional tools (e.g., dashboards, threshold-based alerts) and requires higher-quality automated support to process the massive volume of diagnostic information [221]. Indeed, very often what can be easily observed and fixed by human experts are only symptoms of deeper and more complex problems, that can propagate rather quickly and cripple a huge number of services, websites or applications that depend on the faulty component. Events like this can potentially start a series of undesirable chain-reactions, that typically result in diverse issues experienced by the customers. After recovering from a system outage, even highly skilled engineers can take up to several days to complete a *post-mortem* investigation, eventually leading to meaningful insights on the actual causes. Then, deriving the required fixes to be performed, automations to be developed, or just the documentation to be produced is far from trivial, as things change pretty fast in a production environment, and new issues can be introduced by such interventions. Automated reasoning approaches could drastically simplify this kind of tasks by providing operators with reasonable hypothesis and explanations, thus reducing the scope of the operations. Even better, if designed to operate in real-time in response to detected anomalies [181, 255], they could also provide a system with a certain degree of self-healing capabilities by triggering automated remediation procedures.

An additional challenge to conducting effective RCA is posed by the growing trend in designing applications according to MSA. Such design choice offers many advantages when it comes to building large-scale, resilient, flexible distributed systems but, in these cases, performance issues are usually harder to locate and fix than in classical monolithic architectures. This is usually due to the complex relations and dependencies that exist among the individual services, that can also be implemented using heterogeneous technologies and updated with high frequency, independently of each other. In such scenario, RCA of performance issues is obviously an extremely difficult task. Many approaches rely on the applications to be properly instrumented in order to leverage on logs and metrics [75, 261], but this can lead to significant overhead. Others are based on *causality graphs* that model the dependencies among back-end and front-end components [149, 249], that are totally ineffective when the issues have no impact on the latter. With the aim of taking the best from both worlds, an additional class of methods leverages on correlating application- and system-level metrics to detect anomalous events and maintains a graph that models the propagation of such anomalies across the services (and their hosts) [20, 255], adapting very well to highly dynamic and heterogeneous MSA.

Among the nastiest types of incidents that typically occur in a cloud environment, from an RCA viewpoint, *correlated* and *partial* failures deserve a (dis)honorable mention. Correlated failures usually originate from faulty components that constitute a shared dependency among multiple services or multiple instances of an individual service, such as networking equipment or lower-level software modules. Indeed, even replication, a common practice in Cloud Computing to enhance services reliability and to implement high-availability, can be useless at preventing such failures if the replicas themselves share such dependencies, often hidden by the tons of layers that compose cloud infrastructures stacks. System outages deriving from this kind of failures are quite costly, both in terms of human effort needed to diagnose and fix the issue and in terms of financial and reputation loss. Therefore, there is a trend in shifting from a post-failure diagnosis to a more proactive approach, aiming at preventing such events from occurring. Although, current state-of-the-art methods approach this problem by checking the reliability of a cloud service only *before* its deployment [36, 265], not at runtime. Real-time monitoring is essential, though, since correlated failure are generally caused by events that occur very high frequency in a cloud environment, such as network updates, software updates, configuration changes, or even human intervention to fix pre-existing issues [84, 134]. In order to provide real-time support, one can leverage on the fact that typical updates only affect a small part of services stack and, thus, the scope of the analysis can be restricted (e.g., using *differential* fault graphs) and obtain a huge speed-up. Such problem is still NP-hard, but its solution can also be effectively accelerated by reducing it to a Boolean SAT problem, as proposed in [264]. Once the root cause of the (potential) failure has been detected, an improvement strategy should be recommended to reduce the risk of such event taking place. This is crucial to the purpose of reducing human intervention to the bare minimum, since it can be highly inefficient and error-prone. As one can expect, the accuracy of the results coming from this kind of approaches is bounded by the quality and the granularity of the data describing the dependencies among system components and their respective reliability levels. In addition, the fault graph model that is leveraged by these methods only takes into account *deterministic* failures, excluding non-deterministic and, in particular, partial ones. Indeed, the latter type of failures can be very tricky to fix, and even to detect, as it typically consists of only a restricted subset of the components of a given software not to be working properly, while the remaining parts do not exhibit any symptoms. With respect to total failures, after which a system can be restarted without any repercussion other than the downtime, partial failures can lead to severe inconsistencies and data losses caused by the healthy components not being aware of the others to be malfunctioning and keeping on working according to their programming. According to the results reported by [140]: 48% of the partial failures included in their evaluation of open-source large-scale software systems was caused by not (or poorly) handled errors; 48% of the failures caused some processes to be *stuck* but not completely unresponsive such that, for instance, health checking modules were keeping on responding in time when queried by other components; 71% of the failures were triggered by specific production environments conditions that were not exposed by the testing in staging environments; 68% of the failures required the faulty component to be manually repaired/restarted; the median diagnosis time was *6 day and 5 hours*. The latter result perfectly demonstrates the share of human effort that has to be put when conducting RCA in large-scale distributed systems.

A FEW-SHOT LEARNING EXAMPLES

Listing A.1: Example of prompt used for few-shot learning.

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  strategy: {}
  template:
    spec:
      containers:
        - image: aaa-docker-registry.com/image-name-aaa:0.1_dev
          name: image-name-aaa
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
          resources:
            requests:
              cpu: 0.1
              memory: 2Mi
            limits:
              cpu: 2
              memory: 5Gi
      restartPolicy: Always
status: {}

Answer = cpu_limit_positive

#####

apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  template:
    spec:
      containers:
      - image: bbb-docker-registry.com/image-name-bbb:1.0.0
        name: image-name-bbb
```

```
Answer = cpu_limit_negative

#####

apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  strategy: {}
  template:
    spec:
      containers:
        - env:
          image: aaa-docker-registry.com/image-name-ccc:0.1_dev
          name: image-name-ccc
          ports:
            - name: dbapp
              containerPort: 27017
              protocol: TCP
          resources:
            requests:
              cpu: 0.1
              memory: 2Mi
            limits:
              cpu: 1
              memory: 2Gi
      restartPolicy: Always
status: {}

Answer = cpu_limit_positive

#####

apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  template:
    spec:
      serviceAccountName: image-name-ddd
      securityContext:
        {}
      containers:
        - name: image-name-ddd
          securityContext:
            runAsNonRoot: true
            runAsUser: 1000
          image: "ccc-docker-registry.com:9090/image-name-ddd:0"
          imagePullPolicy: Always
          stdin: true
          tty: true
          ports:
```

```
          - name: http
            containerPort: 3000
            protocol: TCP
          resources:
            {}

Answer = cpu_limit_negative

#####

apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
      - image: aaa-docker-registry.com/image-name-eee:1.0.8-rc.5
        name: image-name-eee
        ports:
        - containerPort: 8080

Answer = cpu_limit_negative

#####

apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  template:
    spec:
      containers:
        - name: image-name-fff
          image: aaa-docker-registry.com/image-name-fff:1.0.1
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
        - name: image-name-ggg
          image: ddd-docker-registry.com/image-name-ggg:0.39.0
          imagePullPolicy: IfNotPresent
          ports:
          - containerPort: 8181

Answer = cpu_limit_negative

#####

apiVersion: apps/v1
kind: StatefulSet
spec:
  replicas: 1
  podManagementPolicy: "Parallel"
  template:
    spec:
      containers:
      - name: image-name-hhh
        image: aaa-docker-registry.com/image-name-hhh:v1.3
        imagePullPolicy: Always
        ports:
        - containerPort: 5944
        resources:
          requests:
            cpu: 0.1
            memory: 250Mi
          limits:
            cpu: 4
            memory: 10Gi
        readinessProbe:
          exec:
            command:
            - ls
            - /tmp
          initialDelaySeconds: 5
          periodSeconds: 60
      restartPolicy: Always

Answer = cpu_limit_positive

#####

apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  template:
    spec:
      initContainers:
        - name: config-data
          image: aaa-docker-registry.com/image-name-iii:1.0.1
      containers:
      - image: aaa-docker-registry.com/image-name-jjj:1.0.1
        name: image-name-jjj
        ports:
        - containerPort: 30306
        resources:
          requests:
            cpu: 0.1
            memory: 750Mi
          limits:
            cpu: 0.5
            memory: 1Gi
        readinessProbe:
            exec:
              command:
                - ls
                - /tmp
            initialDelaySeconds: 5
            periodSeconds: 60
      restartPolicy: Always

Answer = cpu_limit_positive

#####
```

BIBLIOGRAPHY

[1] Sherif Abdelwahed, Jia Bai, Rong Su, and Nagarajan Kandasamy. "On the application of predictive control techniques for adaptive performance management of computing systems." In: *IEEE Transactions on Network and Service Management* 6.4 (2009), pp. 212–225. ISSN: 1932-4537.

[2] Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha. "Unsupervised real-time anomaly detection for streaming data." In: *Neurocomputing* 262 (2017), pp. 134–147. ISSN: 0925-2312.

[3] Virgilio A. F. Almeida. "Capacity Planning for Web Services Techniques and Methodology." In: *Performance Evaluation of Complex Systems: Techniques and Tools*. Ed. by Maria Carla Calzarossa and Salvatore Tucci. Springer Berlin Heidelberg, 2002, pp. 142–157. ISBN: 978-3-540-45798-5.

[4] Ahmad Alnafessah, Alim Ul Gias, Runan Wang, Lulai Zhu, Giuliano Casale, and Antonio Filieri. "Quality-Aware DevOps Research: Where Do We Stand?" In: *IEEE Access* 9 (2021), pp. 44476–44489. ISSN: 2169-3536.

[5] Hamid Arabnejad, Claus Pahl, Pooyan Jamshidi, and Giovani Estrada. "A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling." In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017, pp. 64–73. ISBN: 978-1-5090-6611-7.

[6] Carlos Hernán Tobar Arteaga, Fulvio Rissoi, and Oscar Mauricio Caicedo Rendon. "An adaptive scaling mechanism for managing performance variations in network functions virtualization: A case study in an nfv-based epc." In: *13th International Conference on Network and Service Management*. 2017, pp. 1–7.

[7] Pepa Atanasova, Jakob Grue Simonsen, Christina Lioma, and Isabelle Augenstein. "A Diagnostic Study of Explainability Techniques for Text Classification." In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2020, pp. 3256–3274.

[8] Mohsen Attaran and Jeremy Woods. "Cloud computing technology: improving small business performance using the Internet." In: *Journal of Small Business & Entrepreneurship* 31.6 (2019), pp. 495–519.

[9] AWS. *Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region*. 2015. URL: https://aws.amazon.com/message/5467D2/.

[10] Davide Bacciu, Federico Errica, Alessio Micheli, and Marco Podda. "A gentle introduction to deep learning for graphs." In: *Neural Networks* 129 (2020), pp. 203–221. ISSN: 08936080.

[11] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate." In: *3rd International Conference on Learning Representations, San Diego, CA, USA, May 7-9*. 2015.

[12] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. URL: http://arxiv.org/abs/1409.0473.

[13] Abul Bashar. "Autonomic scaling of Cloud Computing resources using BN-based prediction models." In: *IEEE 2nd International Conference on Cloud Networking*. 2013, pp. 200–204.

[14] Nathan Bell and Jared Hoberock. "Thrust: A Productivity-Oriented Library for CUDA." In: *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 2012, pp. 359–371. ISBN: 978-0-12-385963-1.

[15] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. "Pearson correlation coefficient." In: *Noise reduction in speech processing*. Springer, 2009, pp. 37–40.

[16] Christophe Bertero, Matthieu Roy, Carla Sauvanaud, and Gilles Tredan. "Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection." In: *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 351–360. ISBN: 978-1-5386-0941-5.

[17] Ane Blázquez-García, Angel Conde, Usue Mori, and Jose A. Lozano. "A review on outlier/anomaly detection in time series data." In: (2020). URL: http://arxiv.org/abs/2002.04236.

[18] Michael Borkowski, Stefan Schulte, and Christoph Hochreiner. "Predicting Cloud Resource Utilization." In: *IEEE/ACM 9th International Conference on Utility and Cloud Computing*. 2016, pp. 37–42.

[19] George EP Box, Gwilym M Jenkins, and Gregory C Reinsel. *Time series analysis: forecasting and control*. Vol. 734. John Wiley & Sons, 2011.

[20] Álvaro Brandón, Marc Solé, Alberto Huélamo, David Solans, María S. Pérez, and Victor Muntés-Mulero. "Graph-based root cause analysis for service-oriented and microservice architectures." In: *Journal of Systems and Software* 159 (2020), p. 110432. ISSN: 01641212.

[21] Eric A. Brewer. "Kubernetes and the Path to Cloud Native." In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. Kohala Coast, Hawaii: Association for Computing Machinery, 2015, p. 167. ISBN: 9781450336512.

[22] Tom Brown et al. "Language Models are Few-Shot Learners." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[23] Sandro Brunner, Martin Blöchlinger, Giovanni Toffetti, Josef Spillner, and Thomas Michael Bohnert. "Experimental Evaluation of the Cloud-Native Application Design." In: *Proceedings of the 8th International Conference on Utility and Cloud Computing*. Limassol, Cyprus: IEEE Press, 2015, pp. 488–493. ISBN: 9780769556970.

[24] Anna L Buczak and Erhan Guven. "A survey of data mining and machine learning methods for cyber security intrusion detection." In: *IEEE Communications Surveys & Tutorials* 18.2 (2015), pp. 1153–1176.

[25] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011. ISBN: 9780470887998.

[26] Rajkumar Buyya, Kotagiri Ramamohanarao, Chris Leckie, Rodrigo N. Calheiros, Amir Vahid Dastjerdi, and Steve Versteeg. "Big Data Analytics-Enhanced Cloud Computing: Challenges, Architectural Elements, and Future Directions." In: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. 2015, pp. 75–84.

[27] Rodrigo N. Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. "Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications' QoS." In: *IEEE Transactions on Cloud Computing* 3.4 (2015), pp. 449–458. ISSN: 2168-7161.

[28] Ricardo J. G. B. Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. "Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection." In: *ACM Transactions on Knowledge Discovery from Data* 10.1 (2015), 5:1–5:51. ISSN: 1556-4681.

[29] L. Canetta, N. Cheikhrouhou, and R. Glardon. "Applying two-stage SOM-based clustering approaches to industrial data analysis." In: *Production Planning & Control* 16.8 (2005), pp. 774–784.

[30] Giuseppe Antonio Carella, Michael Pauls, Lars Grebe, and Thomas Magedanz. "An extensible autoscaling engine (ae) for software-based network functions." In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2016, pp. 219–225.

[31] Andrés Carrasco, Brent van Bladel, and Serge Demeyer. "Migrating towards microservices: migration and architecture smells." In: *Proceedings of the 2nd International Workshop on Refactoring*. Association for Computing Machinery, 2018, pp. 1–6. ISBN: 978-1-4503-5974-0.

[32] David M. Chan, Roshan Rao, Forrest Huang, and John F. Canny. "T-SNE-CUDA: GPU-Accelerated T-SNE and its Applications to Modern Data." In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2018, pp. 330–338. ISBN: 978-1-5386-7769-8.

[33] Chris Chatfield. "The Holt-winters forecasting procedure." In: *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 27.3 (1978), pp. 264–279.

[34] Dar-Ren Chen, Ruey-Feng Chang, and Yu-Len Huang. "Breast cancer diagnosis using self-organizing map for sonography." In: *Ultrasound in medicine & biology* 26.3 (2000), pp. 405–411.

[35] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. URL: http://arxiv.org/abs/2107.03374.

[36] Ruichuan Chen, Istemi Ekin Akkus, Bimal Viswanath, Ivica Rimac, and Volker Hilt. "Towards Reliable Application Deployment in the Cloud." In: *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 2017, pp. 464–477. ISBN: 9781450354226.

[37] Tao Chen, Rami Bahsoon, and Xin Yao. "A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems." In: *ACM Computing Surveys* 51.3 (2018), pp. 1–40. ISSN: 0360-0300.

[38] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System." In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 785–794.

[39] Margaret Chiosi et al. *Network Functions Virtualisation - Introductory White Paper*. Tech. rep. 2012.

[40] Ioannis P. Chochliouros, Anastasia S. Spiliopoulou, Alexandros Kostopoulos, Maria Belesioti, Evangelos Sfakianakis, Philippos Georgantas, Eirini Vasilaki, Ioannis Neokosmidis, Theodoros Rokkas, and Athanassios Dardamanis. "Putting Intelligence in the Network Edge Through NFV and Cloud Computing: The SESAME Approach." In: *Engineering Applications of Neural Networks*. Springer, 2017, pp. 704–715. ISBN: 978-3-319-65172-9.

[41] Dan Claudiu Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. “Flexible, High Performance Convolutional Neural Networks for Image Classification.” In: *Proc. 22nd International Joint Conference on Artificial Intelligence (IJCAI), Barcelona, Catalonia, Spain, July 16-22, 2011*. Ed. by Toby Walsh. IJCAI/AAAI, 2011, pp. 1237–1242.

[42] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. “EMNIST: an extension of MNIST to handwritten letters.” In: *CoRR* abs/1702.05373 (2017).

[43] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking cloud serving systems with YCSB.” In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.

[44] Emilio Corchado and Bruno Baruque. “WeVoS-ViSOM: An ensemble summarization algorithm for enhanced data visualization.” In: *Neurocomputing* 75.1 (2012), pp. 171–184. ISSN: 09252312.

[45] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms.” In: *26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 153–167. ISBN: 978-1-4503-5085-3.

[46] Andrea Cossu, Antonio Carta, Vincenzo Lomonaco, and Davide Bacciu. “Continual learning for recurrent neural networks: An empirical evaluation.” In: *Neural Networks* 143 (2021), pp. 607–627. ISSN: 0893-6080.

[47] Tommaso Cucinotta. *distwalk*. 2022. URL: https://github.com/tomcucinotta/distwalk.

[48] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. “Reducing Temporal Interference in Private Clouds through Real-Time Containers.” In: *2019 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 2019, pp. 124–131. ISBN: 978-1-7281-2708-8.

[49] Tommaso Cucinotta, Giacomo Lanciano, Antonio Ritacco, Fabio Brau, Filippo Galli, Vincenzo Iannino, Marco Vannucci, Antonino Artale, Joao Barata, and Enrica Sposato. “Forecasting Operation Metrics for Virtualized Network Functions.” In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021, pp. 596–605.

[50] Tommaso Cucinotta, Giacomo Lanciano, Antonio Ritacco, Marco Vannucci, Antonino Artale, Joao Barata, Enrica Sposato, and Luca Basili. “Behavioral Analysis for Virtualized Network Functions: A SOM-based Approach.” In: *Proceedings of the 10th*

International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications, 2020, pp. 150–160. ISBN: 978-989-758-424-4.

[51] Tommaso Cucinotta, Mauro Marinoni, Alessandra Melani, Andrea Parri, and Carlo Vitucci. "Temporal Isolation Among LTE/5G Network Functions by Real-time Scheduling." In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2017, pp. 368–375. ISBN: 978-989-758-243-1.

[52] Tommaso Cucinotta, Marco Vannucci, Antonio Ritacco, Giacomo Lanciano, Antonino Artale, Joao Barata, and Enrica Sposato. "A method of identifying and classifying the behavior modes of a plurality of data relative to a telephony infrastructure for network function virtualization." EP3772833A1 (Filed), IT102019000014241A (Filed). 2021.

[53] Tommaso Cucinotta, Marco Vannucci, Antonio Ritacco, Giacomo Lanciano, Antonino Artale, Joao Barata, and Enrica Sposato. "A method of predicting the time course of a plurality of data relative to a telephony infrastructure for network function virtualization." EP3772834A1 (Filed), IT102019000014262A (Filed). 2021.

[54] Laizhong Cui, F. Richard Yu, and Qiao Yan. "When big data meets software-defined networking: SDN for big data and big data for SDN." In: *IEEE Network* 30.1 (2016), pp. 58–65. ISSN: 08908044.

[55] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming." In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55.

[56] Shubhomoy Das, Weng-Keen Wong, Thomas Dietterich, Alan Fern, and Andrew Emmott. "Incorporating Expert Feedback into Active Anomaly Discovery." In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 2016, pp. 853–858. ISBN: 978-1-5090-5473-2.

[57] Ailing De, Yuan Zhang, and Chengan Guo. "A parallel adaptive segmentation method based on SOM and GPU with application to MRI image processing." In: *Neurocomputing* 198 (2016), pp. 180–189. ISSN: 09252312.

[58] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Commun. ACM* 51.1 (2008), pp. 107–113. ISSN: 0001-0782.

[59] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's highly available key-value store."

In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[60] Mehmet Demirci. "A survey of machine learning applications for energy-efficient resource management in cloud computing environments." In: *Proceedings - 2015 IEEE 14th International Conference on Machine Learning and Applications, ICMLA 2015*. IEEE, 2016, pp. 1185–1190. ISBN: 9781509002870.

[61] Denız Demırcı, Nazenın şahın, Melıh şirlancis, and Cengiz Acarturk. "Static Malware Detection Using Stacked BiLSTM and GPT-2." In: *IEEE Access* 10 (2022), pp. 58488–58502. ISSN: 2169-3536.

[62] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. 2022. URL: http://arxiv.org/abs/2208.07339.

[63] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[64] Ignacio Díaz, Manuel Domínguez, Abel A. Cuadrado, and Juan J. Fuertes. "A new approach to exploratory analysis of system dynamics using SOM. Applications to industrial processes." In: *Expert Systems with Applications* 34.4 (2008), pp. 2953–2965. ISSN: 0957-4174.

[65] Qingfeng Du, Yu He, Tiandi Xie, Kanglin Yin, and Juan Qiu. "An Approach of Collecting Performance Anomaly Dataset for NFV Infrastructure." In: *Algorithms and Architectures for Parallel Processing*. Ed. by Jaideep Vaidya and Jin Li. Springer International Publishing, 2018, pp. 59–71. ISBN: 978-3-030-05057-3.

[66] Darrell Etherington. *Amazon AWS S3 outage is breaking things for a lot of websites and apps*. 2017. URL: http://tcrn.ch/2mpOdPd.

[67] Mostafa Farshchi, Jean-Guy Schneider, Ingo Weber, and John Grundy. "Metric selection and anomaly detection for cloud operations using log and metric correlation analysis." In: *Journal of Systems and Software* 137 (2018), pp. 531–549. ISSN: 0164-1212.

[68] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre Alain Muller. "Transfer learning for time series classification." In: *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*. IEEE, 2019, pp. 1367–1376. ISBN: 9781538650356.

[69] Tom Fawcett. “An introduction to ROC analysis.” In: *Pattern Recognition Letters* 27.8 (2006), pp. 861–874. ISSN: 0167-8655.

[70] Xincai Fei, Fangming Liu, Hong Xu, and Hai Jin. “Adaptive VNF Scaling and Flow Routing with Proactive Demand Prediction.” In: *IEEE Conference on Computer Communications*. Vol. 2018-April. 2018, pp. 486–494. ISBN: 978-1-5386-4128-6.

[71] Vincent Fortuin, Matthias Hüser, Francesco Locatello, Heiko Strathmann, and Gunnar Rätsch. “Som-Vae: Interpretable discrete representation learning on time series.” In: *7th International Conference on Learning Representations, ICLR 2019*. 2019.

[72] Christian W Frey. “Monitoring of complex industrial processes based on self-organizing maps and watershed transformations.” In: *2012 IEEE International Conference on Industrial Technology*. IEEE. 2012, pp. 1041–1046.

[73] Jerome H. Friedman. “Greedy Function Approximation: A Gradient Boosting Machine.” In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. ISSN: 0090-5364.

[74] Juan J Fuertes, Manuel Domìnguez, Perfecto Reguera, Miguel A Prada, Ignacio Dìaz, and Abel A Cuadrado. “Visual dynamic model based on self-organizing maps for supervision and fault detection in industrial processes.” In: *Engineering Applications of Artificial Intelligence* 23.1 (2010), pp. 8–17.

[75] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. “Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices.” In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 19–33. ISBN: 9781450362405.

[76] Leo Gao et al. *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*. 2020. URL: http://arxiv.org/abs/2101.00027.

[77] Domenico Giordano, Matteo Paltenghi, Stiven Metaj, and Antonin Dvorak. “Anomaly detection in the CERN cloud infrastructure.” In: *EPJ Web of Conferences* 251 (2021), p. 02011. ISSN: 2100-014X.

[78] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[79] Chris Gorman. *TensorFlow Self-Organizing Map*. 2019. URL: https://github.com/cgorman/tensorflow-som.

[80] Michael Grothaus. *That major Google outage meant some Nest users couldn't unlock doors or use the AC*. 2019. URL: https://www.fastcompany.com/90358396/.

[81] Taylor Liles Groves, Ryan E. Grant, Aaron Gonzales, and Dorian Arnold. “Unraveling network-induced memory contention: Deeper insights with machine learning.” In: *IEEE Transactions on Parallel and Distributed Systems* 29.8 (2018), pp. 1907–1922. ISSN: 10459219.

[82] Anton Gulenko, Marcel Wallschlager, Florian Schmidt, Odej Kao, and Feng Liu. “Evaluating machine learning algorithms for anomaly detection in clouds.” In: *Proceedings - 2016 IEEE International Conference on Big Data, Big Data 2016*. IEEE, 2016, pp. 2716–2721. ISBN: 9781467390040.

[83] Anton Gulenko, Marcel Wallschläger, Florian Schmidt, Odej Kao, and Feng Liu. “A System Architecture for Real-time Anomaly Detection in Large-scale NFV Systems.” In: *Procedia Computer Science* 94 (2016), pp. 491–496. ISSN: 1877-0509.

[84] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. “Why does the cloud stop computing? Lessons from hundreds of service outages.” In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 1–16. ISBN: 9781450345255.

[85] Lav Gupta, M. Samaka, Raj Jain, Aiman Erbad, Deval Bhamare, and H. Anthony Chan. “Fault and performance management in multi-cloud based NFV using shallow and deep predictive structures.” In: *Journal of Reliable Intelligent Environments* 3.4 (2017), pp. 221–231. ISSN: 21994676.

[86] Charles R. Harris et al. “Array programming with NumPy.” In: *Nature* 585.7825 (2020), pp. 357–362.

[87] Tom Harris. “A Kohonen SOM based, machine health monitoring system which enables diagnosis of faults not seen in the training set.” In: *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*. Vol. 1. IEEE. 1993, pp. 947–950.

[88] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[89] Hassan Hawilo, Abdallah Shami, Maysam Mirahmadi, and Rasool Asal. “NFV: state of the art, challenges, and implementation in next generation mobile networks (vEPC).” In: *IEEE Network* 28.6 (2014), pp. 18–26. ISSN: 0890-8044.

[90] Ying He, F. Richard Yu, Nan Zhao, Victor C.M. Leung, and Hongxi Yin. “Software-Defined Networks with Mobile Edge Computing and Caching for Smart Cities: A Big Data Deep Reinforcement Learning Approach.” In: *IEEE Communications Magazine* 55.12 (2017), pp. 31–37. ISSN: 01636804.

[91] Geert Heyman, Rafael Huysegems, Pascal Justen, and Tom Van Cutsem. "Natural language-guided programming." In: *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Association for Computing Machinery, 2021, pp. 39–55. ISBN: 978-1-4503-9110-8.

[92] Benjamin Hoover, Hendrik Strobelt, and Sebastian Gehrmann. "exBERT: A Visual Analysis Tool to Explore Learned Representations in Transformer Models." In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Association for Computational Linguistics, 2020, pp. 187–196.

[93] Scott Horsfield and Ankur Sethi. *Introducing Native Support for Predictive Scaling with Amazon EC2 Auto Scaling*. 2021. URL: https://aws.amazon.com/blogs/compute/introducing-native-support-for-predictive-scaling-with-amazon-ec2-auto-scaling.

[94] Shashikant Ilager, Rajeev Muralidhar, and Rajkumar Buyya. "Artificial Intelligence (AI)-Centric Management of Resources in Modern Distributed Computing Systems." In: *IEEE Cloud Summit*. 2020, pp. 1–10.

[95] Waheed Iqbal, Abdelkarim Erradi, Muhammad Abdullah, and Arif Mahmood. "Predictive Auto-scaling of Multi-tier Applications Using Performance Varying Cloud Resources." In: *IEEE Transactions on Cloud Computing* (2019), pp. 1–1. ISSN: 2168-7161.

[96] Arman Iranfar, Marina Zapater, and David Atienza. "Machine Learning-Based Quality-Aware Power and Thermal Management of Multistream HEVC Encoding on Multicore Servers." In: *IEEE Transactions on Parallel and Distributed Systems* 29.10 (2018), pp. 2268–2281. ISSN: 15582183.

[97] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. "Empirical prediction models for adaptive resource provisioning in the cloud." In: *Future Generation Computer Systems* 28.1 (2012), pp. 155–162. ISSN: 0167-739X.

[98] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre Alain Muller. "Deep learning for time series classification: a review." In: *Data Mining and Knowledge Discovery* 33.4 (2019), pp. 917–963. ISSN: 1573756X.

[99] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. "Jigsaw: large language models meet program synthesis." In: *Proceedings of the 44th International Conference on Software Engineering*. Association for Computing Machinery, 2022, pp. 1219–1231. ISBN: 978-1-4503-9221-1.

[100] N. Jalodia, S. Henna, and A. Davy. "Deep Reinforcement Learning for Topology-Aware VNF Resource Prediction in NFV Environments." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks*. 2019, pp. 1–5.

[101] Vinodh Kumaran Jayakumar, Jaewoo Lee, In Kee Kim, and Wei Wang. "A Self-Optimized Generic Workload Prediction Framework for Cloud Computing." In: *2020 IEEE International Parallel and Distributed Processing Symposium*. 2020, pp. 779–788.

[102] Yuan Jiang and Zhi-Hua Zhou. "SOM Ensemble-Based Image Segmentation." In: *Neural Processing Letters* 20.3 (2004), pp. 171–178. ISSN: 1370-4621.

[103] Li Ju, Prashant Singh, and Salman Toor. "Proactive Autoscaling for Edge Computing Systems with Kubernetes." In: *10th International Workshop on Cloud and Edge Computing and Applications Management*. 2021.

[104] Rudolph Emil Kalman. "A New Approach to Linear Filtering and Prediction Problems." In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D (1960), pp. 35–45.

[105] Peng Kang and Palden Lama. "Robust Resource Scaling of Containerized Microservices with Probabilistic Machine learning." In: *IEEE/ACM 13th International Conference on Utility and Cloud Computing*. 2020, pp. 122–131.

[106] Aadharsh Kannan, Jacob LaRiviere, and R. Preston McAfee. "Characterizing the Usage Intensity of Public Cloud." In: *ACM Trans. Econ. Comput.* 9.3 (2021). ISSN: 2167-8375.

[107] Kathan Kashiparekh, Jyoti Narwariya, Pankaj Malhotra, Lovekesh Vig, and Gautam Shroff. "ConvTimeNet: A Pretrained Deep Convolutional Neural Network for Time Series Classification." In: *2019 International Joint Conference on Neural Networks (IJCNN)*. Vol. 2019-July. IEEE, 2019, pp. 1–8. ISBN: 978-1-7281-1985-4.

[108] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. "LightGBM: a highly efficient gradient boosting decision tree." In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2017, pp. 3149–3157. ISBN: 978-1-5108-6096-4.

[109] J.O. Kephart and D.M. Chess. "The vision of autonomic computing." In: *Computer* 36.1 (2003), pp. 41–50. ISSN: 0018-9162.

[110] Abeer Abdel Khaleq and Ilkyeun Ra. "Development of QoS-aware agents with reinforcement learning for autoscaling of microservices on the cloud." In: *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion*. 2021, pp. 13–19.

[111] Lidia Kidane, Paul Townend, Thijs Metsch, and Erik Elmroth. "When and How to Retrain Machine Learning-based Cloud Management Systems." In: *IEEE International Parallel and Distributed Processing Symposium Workshops*. 2022, pp. 688–698.

[112] In Kee Kim, Wei Wang, Yanjun Qi, and Marty Humphrey. "CloudInsight: Utilizing a Council of Experts to Predict Future Cloud Application Workloads." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 41–48.

[113] Colin Ian King. *stress-ng*. 2023. URL: https://github.com/ColinIanKing/stress-ng.

[114] Diederik P. Kingma and Jimmy Lei Ba. "Adam: A method for stochastic optimization." In: *3rd International Conference on Learning Representations, ICLR*. 2015.

[115] Teuvo Kohonen. "The Self-Organizing Map." In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480. ISSN: 15582256.

[116] J. F. Kolen and S. C. Kremer. "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies." In: *A Field Guide to Dynamical Recurrent Networks*. 2001, pp. 237–243.

[117] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems 25*. Ed. by F Pereira, C J C Burges, L Bottou, and K Q Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105.

[118] Kubernetes. *Kubernetes Overview*. 2023. URL: https://kubernetes.io/docs/concepts/overview/.

[119] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. "NFVnice: Dynamic backpressure and scheduling for NFV service chains." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Vol. 14. 2017, pp. 71–84. ISBN: 9781450346535.

[120] Jitendra Kumar, Ashutosh Kumar Singh, and Rajkumar Buyya. "Self directed learning based workload forecasting model for cloud resource management." In: *Information Sciences* 543 (2021), pp. 345–366. ISSN: 0020-0255.

[121] Jan Lachmair, Thomas Mieth, Rene Griessl, Jens Hagemeyer, and Mario Porrmann. "From CPU to FPGA — Acceleration of self-organizing maps for data mining." In: *2017 International Joint Conference on Neural Networks (IJCNN)*. Vol. 2017-May. IEEE, 2017, pp. 4299–4308. ISBN: 978-1-5090-6182-2.

[122] Giacomo Lanciano. *monasca-predictor*. 2022. DOI: 10.5281/zenodo.5627812. URL: https://github.com/giacomolanciano/monasca-predictor.

[123] Giacomo Lanciano, Filippo Galli, Tommaso Cucinotta, Davide Bacciu, and Andrea Passarella. "Predictive auto-scaling with OpenStack Monasca." In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*. Association for Computing Machinery, 2021, pp. 1–10. ISBN: 978-1-4503-8564-0.

[124] Giacomo Lanciano, Filippo Galli, Tommaso Cucinotta, Davide Bacciu, and Andrea Passarella. *Companion repo of the paper "Extending OpenStack Monasca for Predictive Elasticity Control"*. 2022. DOI: 10.5281/zenodo.5888406. URL: https://github.com/giacomolanciano/predictive-elasticity-monasca.

[125] Giacomo Lanciano, Antonio Ritacco, Fabio Brau, Tommaso Cucinotta, Marco Vannucci, Antonino Artale, Joao Barata, and Enrica Sposato. "Using Self-Organizing Maps for the Behavioral Analysis of Virtualized Network Functions." In: *Cloud Computing and Services Science*. Ed. by Donald Ferguson, Claus Pahl, and Markus Helfert. Springer International Publishing, 2021, pp. 153–177. ISBN: 978-3-030-72369-9.

[126] Giacomo Lanciano, Antonio Ritacco, Tommaso Cucinotta, Marco Vannucci, Antonino Artale, Luca Basili, Enrica Sposato, and Joao Barata. "SOM-based behavioral analysis for virtualized network functions." In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. ACM, 2020, pp. 1204–1206. ISBN: 978-1-4503-6866-7.

[127] Giacomo Lanciano, Manuel Stein, Volker Hilt, and Tommaso Cucinotta. "Analyzing Declarative Deployment Code with Large Language Models." In: *Proceedings of the 13th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2023.

[128] Nikolay Laptev, Jason Yosinski, Li Erran Li, Slawek Smyl, Erran Li Li, and Slawek Smyl. "Time-series Extreme Event Forecasting with Neural Networks at Uber." In: *International Conference on Machine Learning - Time Series Workshop* 34 (2017), pp. 1–5.

[129] L. Le, D. Sinh, B. P. Lin, and L. Tung. "Applying Big Data, Machine Learning, and SDN/NFV to 5G Traffic Clustering, Forecasting, and Management." In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. 2018, pp. 168–176.

[130] Kenneth Levenberg. "A method for the solution of certain non-linear problems in least squares." In: *Quarterly of Applied Mathematics* 2.2 (1944), pp. 164–168. ISSN: 0033-569X, 1552-4485.

[131] Mark Leznik et al. *RECAP Artificial Data Traces*. 2019. DOI: 10.5281/zenodo.3458559. URL: https://zenodo.org/record/3458559.

[132] Ruiyin Li, Mohamed Soliman, Peng Liang, and Paris Avgeriou. "Symptoms of Architecture Erosion in Code Reviews: A Study of Two OpenStack Projects." In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 2022, pp. 24–35.

[133] Te-Sheng Li and Cheng-Lung Huang. "Defect spatial pattern recognition using a hybrid SOM-SVM approach in semiconductor manufacturing." In: *Expert Systems with Applications* 36.1 (2009), pp. 374–385. ISSN: 09574174.

[134] Ze Li et al. "Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure." In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 389–402. ISBN: 9781939133137.

[135] Bryan Lim, Sercan Ö. Arik, Nicolas Loeff, and Tomas Pfister. "Temporal Fusion Transformers for interpretable multi-horizon time series forecasting." In: *International Journal of Forecasting* 37.4 (2021), pp. 1748–1764. ISSN: 0169-2070.

[136] Seppo Linnainmaa. "Taylor expansion of the accumulated rounding error." In: *BIT Numerical Mathematics* 16.2 (1976), pp. 146–160. ISSN: 0006-3835.

[137] Peter J. Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. "Generating Wikipedia by Summarizing Long Sequences." In: *International Conference on Learning Representations*. 2018.

[138] Xuan Liu, Bo Cheng, Yi Yue, Meng Wang, Biyi Li, and Junliang Chen. "Traffic-Aware and Reliability-Guaranteed Virtual Machine Placement Optimization in Cloud Datacenters." In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. Vol. 2019-July. IEEE, 2019, pp. 91–98. ISBN: 978-1-7281-2705-7.

[139] Yao Liu, Jun Sun, Qing Yao, Su Wang, Kai Zheng, and Yan Liu. "A Scalable Heterogeneous Parallel SOM Based on MPI/CUDA." In: *Proceedings of The 10th Asian Conference on Machine Learning*. Ed. by Jun Zhu and Ichiro Takeuchi. Vol. 95. PMLR, 2018, pp. 264–279.

[140] Chang Lou, Peng Huang, and Scott Smith. "Understanding, Detecting and Localizing Partial Failures in Large System Software." In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020. ISBN: 9781939133137.

[141] Wei Ma, Mengjie Zhao, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jie Zhang, Wenhan Wang, and Yang Liu. *Is Self-Attention Powerful to Learn Code Syntax and Semantics?* 2022. URL: http://arxiv.org/abs/2212.10017.

[142] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE." In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. ISSN: 1533-7928.

[143] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. "Generating Diverse Code Explanations using the GPT-3 Large Language Model." In: *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 2*. Vol. 2. Association for Computing Machinery, 2022, pp. 37–39. ISBN: 978-1-4503-9195-5.

[144] C. Makaya, D. Freimuth, D. Wood, and S. Calo. "Policy-based NFV management and orchestration." In: *IEEE Conference on Network Function Virtualization and Software Defined Network*. 2015, pp. 128–134.

[145] Pankaj Malhotra, T. V. Vishnu, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. "TimeNet: Pre-trained deep recurrent neural network for time series classification." In: *ESANN 2017 - Proceedings, 25th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. 2017, pp. 607–612. ISBN: 9782875870391.

[146] N. Malini and M. Pushpa. "Analysis on credit card fraud identification techniques based on KNN and outlier detection." In: *2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*. 2017, pp. 255–258.

[147] Riccardo Mancini. *XPySom*. 2022. URL: https://github.com/Manciukic/xpysom.

[148] Riccardo Mancini, Antonio Ritacco, Giacomo Lanciano, and Tommaso Cucinotta. "XPySom: High-Performance Self-Organizing Maps." In: *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 209–216. ISBN: 978-1-72819-924-5.

[149] Leonardo Mariani, Cristina Monni, Mauro Pezze, Oliviero Riganelli, and Rui Xin. "Localizing Faults in Cloud Systems." In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 262–273. ISBN: 978-1-5386-5012-7.

[150] Bernard Marr. *The 5 Biggest Cloud Computing Trends In 2022*. https://www.forbes.com/sites/bernardmarr/2021/10/25/the-5-biggest-cloud-computing-trends-in-2022/. 2021.

[151] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/.

[152] Sabine McConnell, Robert Sturgeon, Gregory Henry, Andrew Mayne, and Richard Hurley. "Scalability of Self-organizing Maps on a GPU cluster using OpenCL and CUDA." In: *Journal of Physics: Conference Series* 341 (2012), p. 012018. ISSN: 1742-6596.

[153] Peter Mell and Tim Grance. *The NIST Definition of Cloud Computing. SP 800–145*. 2011. URL: https://csrc.nist.gov/publications/detail/sp/800-145/final.

[154] Giovanni Merlino, Rustem Dautov, Salvatore Distefano, and Dario Bruneo. "Enabling Workload Engineering in Edge, Fog, and Cloud Computing through OpenStack-based Middleware." In: *ACM Transactions on Internet Technology* 19.2 (2019), 28:1–28:22. ISSN: 1533-5399.

[155] Albert Mestres et al. "Knowledge-defined networking." In: *Computer Communication Review* 47.3 (2017), pp. 1–10. ISSN: 19435819.

[156] Stiven Metaj. "End-to-end anomaly detection system in the CERN Openstack Cloud infrastructure." PhD thesis. 2022.

[157] R. Mijumbi, S. Hasija, S. Davy, A. Davy, B. Jennings, and R. Boutaba. "Topology-Aware Prediction of Virtual Network Function Resource Requirements." In: *IEEE Transactions on Network and Service Management* 14.1 (2017), pp. 106–120.

[158] M. Miyazawa, M. Hayashi, and R. Stadler. "vNMF: Distributed fault detection using clustering approach for network function virtualization." In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2015, pp. 640–645.

[159] Haris Mumtaz, Paramvir Singh, and Kelly Blincoe. "A systematic mapping study on architectural smells detection." In: *Journal of Systems and Software* 173 (2021), p. 110885. ISSN: 0164-1212.

[160] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. "Anomaly Detection and Classification using Distributed Tracing and Deep Learning." In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2019, pp. 241–250.

[161] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. "Design principles, architectural smells and refactorings for microservices: a multivocal review." In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (2020), pp. 3–15. ISSN: 2524-8529.

[162] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?" In: *Queue* 6.2 (2008), pp. 40–53. ISSN: 1542-7730.

[163] Carlos H.Z. Nicodemus, Cristina Boeres, and Vinod E.F. Rebello. "Managing Vertical Memory Elasticity in Containers." In: *IEEE/ACM 13th International Conference on Utility and Cloud Computing*. 2020, pp. 132–142.

[164] T. Niwa, M. Miyazawa, M. Hayashi, and R. Stadler. "Universal fault detection for NFV using SOM-based clustering." In: *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 2015, pp. 315–320.

[165] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations." In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017.

[166] OpenStack. *Cinder Documentation*. 2022. URL: https://docs.openstack.org/cinder.

[167] OpenStack. *Glance Documentation*. 2022. URL: https://docs.openstack.org/glance.

[168] OpenStack. *Heat Documentation*. 2022. URL: https://docs.openstack.org/heat.

[169] OpenStack. *Kolla Documentation*. 2022. URL: https://docs.openstack.org/kolla.

[170] OpenStack. *Monasca Documentation*. 2022. URL: https://docs.openstack.org/monasca.

[171] OpenStack. *Neutron Documentation*. 2022. URL: https://docs.openstack.org/neutron.

[172] OpenStack. *Nova Documentation*. 2022. URL: https://docs.openstack.org/nova.

[173] OpenStack. *Octavia Documentation*. 2022. URL: https://docs.openstack.org/octavia.

[174] OpenStack. *Senlin Documentation*. 2022. URL: https://docs.openstack.org/senlin.

[175] Per Olov Ostberg et al. "Reliable capacity provisioning for distributed cloud/edge/fog computing applications." In: *EuCNC 2017 - European Conference on Networks and Communications*. IEEE, 2017, pp. 1–6. ISBN: 9781538638736.

[176] E.J. Palomo, J. North, D. Elizondo, R.M. Luque, and T. Watson. "Application of growing hierarchical SOM for visualisation of network forensics traffic data." In: *Neural Networks* 32 (2012), pp. 275–284. ISSN: 08936080.

[177] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32*. 2019, pp. 8024–8035.

[178] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830.

[179] Jianing Pei, Peilin Hong, and Defang Li. "Virtual Network Function Selection and Chaining Based on Deep Learning in SDN and NFV-Enabled Networks." In: *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2018, pp. 1–6. ISBN: 978-1-5386-4328-0.

[180] Tiago Pimentel, Marianne Monteiro, Adriano Veloso, and Nivio Ziviani. "Deep Active Learning for Anomaly Detection." In: (2018). URL: http://arxiv.org/abs/1805.09411.

[181] Teerat Pitakrat, Dušan Okanović, André van Hoorn, and Lars Grunske. "Hora: Architecture-aware online failure prediction." In: *Journal of Systems and Software* 137 (2018), pp. 669–685. ISSN: 01641212.

[182] Steven J. Plimpton and Karen D. Devine. "MapReduce in MPI for Large-Scale Graph Algorithms." In: *Parallel Comput.* 37.9 (2011), pp. 610–632. ISSN: 0167-8191.

[183] Francisco Ponce, Jacopo Soldani, Hernán Astudillo, and Antonio Brogi. "Smells and refactorings for microservices security: A multivocal literature review." In: *Journal of Systems and Software* 192 (2022), p. 111393. ISSN: 0164-1212.

[184] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. "Auto-scaling web applications in clouds: A taxonomy and survey." In: *ACM Computing Surveys* 51.4 (2018), pp. 1–33. ISSN: 0360-0300.

[185] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. *Improving Language Understanding by Generative Pre-Training*. Tech. rep. 2018.

[186] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. *Language Models are Unsupervised Multitask Learners*. 2019.

[187] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67. ISSN: 1533-7928.

[188] Sabidur Rahman, Tanjila Ahmed, Minh Huynh, Massimo Tornatore, and Biswanath Mukherjee. "Auto-scaling VNFs using machine learning to improve QoS and reduce cost." In: *IEEE International Conference on Communications*. 2018.

[189] Anand Rajaraman and Jeffrey David Ullman. "Data Mining." In: *Mining of Massive Datasets*. Cambridge University Press, 2011, pp. 1–17. ISBN: 978-1-107-73741-9.

[190] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. "ZeRO: memory optimizations toward training trillion parameter models." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2020, pp. 1–16. ISBN: 978-1-72819-998-6.

[191] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. "ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, 2021, pp. 1–14. ISBN: 978-1-4503-8442-1.

[192] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic. "Cyber-physical systems: The next computing revolution." In: *Proceedings of the 47th Design Automation Conference - DAC '10*. ACM Press, 2010, p. 731. ISBN: 9781450300025.

[193] Syama Sundar Rangapuram, Matthias Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. "Deep state space models for time series forecasting." In: *Advances in Neural Information Processing Systems*. Vol. 2018-Decem. 2018, pp. 7785–7794.

[194] Windhya Rankothge, Jiefei Ma, Franck Le, Alessandra Russo, and Jorge Lobo. "Towards making network function virtualization a cloud computing service." In: *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015*. IEEE, 2015, pp. 89–97. ISBN: 9783901882760.

[195] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. "DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters." In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Association for Computing Machinery, 2020, pp. 3505–3506. ISBN: 978-1-4503-7998-4.

[196] Andreas Rauber, Philipp Tomsich, and Dieter Merkl. "parSOM: a parallel implementation of the self-organizing map exploiting cache effects: making the SOM fit for interactive high-performance data analysis." In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*. Vol. 6. IEEE, 2000, 177–182 vol.6. ISBN: 0-7695-0619-4.

[197] Quentin Rebjock, Valentin Flunkert, Tim Januschowski, Laurent Callot, and Joel Castellon. "A Simple and Effective Predictive Resource Scaling Heuristic for Large-scale Cloud Applications." In: *2nd International Workshop on Applied AI for Database Systems and Applications*. 2020.

[198] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B. Gupta, Xiaojiang Chen, and Xin Wang. "A Survey of Deep Active Learning." In: *ACM Computing Surveys* 54.9 (2021), 180:1–180:40. ISSN: 0360-0300.

[199] Michele Resta, Anna Monreale, and Davide Bacciu. "Occlusion-Based Explanations in Deep Recurrent Models for Biomedical Signals." In: *Entropy* 23.8 (2021).

[200] Trevor Richardson and Eliot Winer. "Extending parallelization of the self-organizing map by combining data and network partitioned methods." In: *Advances in Engineering Software* 88 (2015), pp. 1–7. ISSN: 09659978.

[201] Peter J. Rousseeuw. "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis." In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. ISSN: 0377-0427.

[202] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting." In: *IEEE 4th International Conference on Cloud Computing*. 2011, pp. 500–507.

[203] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. "Learning representations by back-propagating errors." In: *Nature* 6088.33 (1986), pp. 533–536.

[204] Krzysztof Rzadca et al. "Autopilot: workload autoscaling at Google." In: *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, 2020, pp. 1–16. ISBN: 9781450368827.

[205] R. Samrin and D. Vasumathi. "Review on anomaly based network intrusion detection system." In: *2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*. 2017, pp. 141–147.

[206] Tran Van Sang, Ryosuke Kobayashi, Rie S. Yamaguchi, and Toshiyuki Nakata. "Accelerating Solution of Generalized Linear Models by Solving Normal Equation Using GPGPU on a Large Real-World Tall-Skinny Data Set." In: *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Vol. 2019-Octob. IEEE, 2019, pp. 112–119. ISBN: 978-1-7281-4194-7.

[207] Tugdual Sarazin, Hanane Azzag, and Mustapha Lebbah. "SOM Clustering Using Spark-MapReduce." In: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 2014, pp. 1727–1734. ISBN: 978-1-4799-4116-2.

[208] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. "Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models." In: *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*. Vol. 1. Association for Computing Machinery, 2022, pp. 27–43. ISBN: 978-1-4503-9194-8.

[209] Carla Sauvanaud, Mohamed Kaâniche, Karama Kanoun, Kahina Lazri, and Guthemberg Da Silva Silvestre. "Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned." In: *Journal of Systems and Software* 139 (2018), pp. 84–106. ISSN: 0164-1212.

[210] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. "The Graph Neural Network Model." In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. ISSN: 1941-0093.

[211] Jürgen Schmidhuber and Sepp Hochreiter. "Long short-term memory." In: *Neural Comput* 9.8 (1997), pp. 1735–1780.

[212] Lucia Schuler, Somaya Jamil, and Niklas Kühl. "AI-based Resource Allocation: Reinforcement Learning for Adaptive Autoscaling in Serverless Environments." In: *IEEE/ACM 21st International Symp. on Cluster, Cloud and Internet Comp.* 2021, pp. 804–811.

[213] Skipper Seabold and Josef Perktold. "statsmodels: Econometric and statistical modeling with python." In: *9th Python in Science Conference*. 2010.

[214] Ankur Sethi. *Using EC2 Auto Scaling predictive scaling policies with Blue/Green deployments*. 2021. URL: https://aws.amazon.com/blogs/compute/retaining-metrics-across-blue-green-deployment-for-predictive-scaling.

[215] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, and Federica Sarro. *A Survey on Machine Learning Techniques for Source Code Analysis*. 2021. URL: http://arxiv.org/abs/2110.09610.

[216] Runyu Shi, Jia Zhang, Wenjing Chu, Qihao Bao, Xiatao Jin, Chenran Gong, Qihao Zhu, Chang Yu, and Steven Rosenberg. "MDP and Machine Learning-Based Cost-Optimization of Dynamic Resource Allocation for Network Function Virtualization." In: *2015 IEEE International Conference on Services Computing*. IEEE, 2015, pp. 65–73. ISBN: 978-1-4673-7281-7.

[217] Connor Shorten and Taghi M. Khoshgoftaar. "Language Models for Deep Learning Programming: A Case Study with Keras." In: *Deep Learning Applications, Volume 4*. Ed. by M. Arif Wani and Vasile Palade. Springer Nature, 2023, pp. 135–161. ISBN: 978-981-19615-3-3.

[218] Catherine Shu. *Netflix Crippled On Christmas Eve By AWS Outages*. 2012. URL: http://tcrn.ch/Y9mxXr.

[219] John Sipple. "Interpretable, Multidimensional, Multimodal Anomaly Detection with Negative Sampling for Detection of Device Failure." In: *Proceedings of the 37th International Conference on Machine Learning*. PMLR, 2020, pp. 9016–9025.

[220] Jacopo Soldani and Antonio Brogi. "Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey." In: *ACM Comput. Surv.* 55.3 (2022). ISSN: 0360-0300.

[221] Marc Solé, Victor Muntés-Mulero, Annie Ibrahim Rana, and Giovani Estrada. "Survey on Models and Techniques for Root-Cause Analysis." In: (2017). URL: http://arxiv.org/abs/1701.08546.

[222] Hongchao Song, Zhuqing Jiang, Aidong Men, and Bo Yang. "A Hybrid Semi-Supervised Anomaly Detection Model for High-Dimensional Data." In: *Computational Intelligence and Neuroscience* 2017 (2017), pp. 1–9. ISSN: 1687-5265.

[223] Ankita Nandkishor Sontakke, Manasi Patwardhan, Lovekesh Vig, Raveendra Kumar Medicherla, Ravindra Naik, and Gautam Shroff. "Code Summarization: Do Transformers Really Understand Code?" In: *Deep Learning for Code Workshop*. 2022.

[224] Matheus A. Souza, Lucas A. Maciel, Pedro Henrique Penna, and Henrique C. Freitas. "Energy Efficient Parallel K-Means Clustering for an Intel® Hybrid Multi-Chip Package." In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2018, pp. 372–379. ISBN: 978-1-5386-7769-8.

[225] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems." In: *Computing in Science & Engineering* 12.3 (2010), pp. 66–73.

[226] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks." In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[227] Pengcheng Tang, Fei Li, Wei Zhou, Weihua Hu, and Li Yang. "Efficient auto-scaling approach in the telco cloud using self-learning algorithm." In: *2015 IEEE Global Communications Conference (GLOBECOM)*. 2015, pp. 1–6.

[228] Hassan Tariq, Harith Al-Sahaf, and Ian Welch. "Modelling and Prediction of Resource Utilization of Hadoop Clusters: A Machine Learning Approach." In: *12th IEEE/ACM International Conference on Utility and Cloud Computing*. Auckland, New

Zealand: Association for Computing Machinery, 2019, pp. 93–100. ISBN: 9781450368940.

[229] Sean J Taylor and Benjamin Letham. "Forecasting at scale." In: *The American Statistician* 72.1 (2018), pp. 37–45.

[230] Ian Tenney et al. "The Language Interpretability Tool: Extensible, Interactive Visualizations and Analysis for NLP Models." In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2020, pp. 107–118.

[231] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. "Transformer-Based Language Models for Software Vulnerability Detection." In: *Proceedings of the 38th Annual Computer Security Applications Conference*. Association for Computing Machinery, 2022, pp. 481–496. ISBN: 978-1-4503-9759-9.

[232] TPC. *TPC-W Benchmark*. 2021. URL: http://www.tpc.org/tpcw/.

[233] Duc Truong and Jude Cross. *How Blizzard Entertainment Uses Autoscaling With Overwatch*. 2019. URL: https://www.openstack.org/videos/summits/denver-2019/how-blizzard-entertainment-uses-autoscaling-with-overwatch.

[234] Joseph Tsidulko. *AWS Apologizes For Cloud Outage, Blames Typo*. 2017. URL: https://www.crn.com/news/cloud/300084012/aws-apologizes-for-cloud-outage-blames-typo.htm.

[235] Shreshth Tuli et al. "HUNTER: AI based holistic resource management for sustainable cloud computing." In: *Journal of Systems and Software* 184 (2022), p. 111124. ISSN: 0164-1212.

[236] Marian Turowski and Alexander Lenk. "Vertical Scaling Capability of OpenStack." In: *Service-Oriented Computing - ICSOC 2014 Workshops*. Ed. by Farouk Toumani, Barbara Pernici, Daniela Grigori, Djamal Benslimane, Jan Mendling, Nejib Ben Hadj-Alouane, Brian Blake, Olivier Perrin, Iman Saleh Moustafa, and Sami Bhiri. Springer International Publishing, 2015, pp. 351–362. ISBN: 978-3-319-22885-3.

[237] Lorenzo Valerio, Marco Conti, and Andrea Passarella. "Energy efficient distributed analytics at the edge of the network for IoT environments." In: *Pervasive and Mobile Computing* 51 (2018), pp. 27–42. ISSN: 15741192.

[238] Frenk D Van den Berg, PJJ Kok, Haibing Yang, MP Aarnts, Philip Meilland, Thomas Kebe, Mathias Stolzenberg, David Krix, Wenqian Zhu, AJ Peyton, et al. "Product uniformity control-A research collaboration of european steel industries to non-destructive evaluation of microstructure and mechanical

properties." In: *Electromagnetic Non-Destructive Evaluation (XXI). 6 September 2017 through 8 September 2017*. 2018, pp. 120–129.

[239] Sofie Van Gassen, Britt Callebaut, Mary J Van Helden, Bart N Lambrecht, Piet Demeester, Tom Dhaene, and Yvan Saeys. "FlowSOM: Using self-organizing maps for visualization and interpretation of cytometry data." In: *Cytometry Part A* 87.7 (2015), pp. 636–645.

[240] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention is All you Need." In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017.

[241] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale cluster management at Google with Borg." In: *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*. ACM Press, 2015, pp. 1–17. ISBN: 9781450332385.

[242] Giuseppe Vettigli. *MiniSom*. 2019. URL: https://github.com/JustGlowing/minisom.

[243] VMWare. *vRealize Operations Documentation*. 2023. URL: https://docs.vmware.com/en/vRealize-Operations/index.html.

[244] Werner Vogels. *A new era of DevOps, powered by machine learning*. 2021. URL: https://www.allthingsdistributed.com/2021/05/devops-powered-by-machine-learning.html.

[245] Marcel Wallschläger, Anton Gulenko, Florian Schmidt, Odej Kao, and Feng Liu. "Automated Anomaly Detection in Virtualized Services Using Deep Packet Inspection." In: *Procedia Computer Science*. Vol. 110. Elsevier, 2017, pp. 510–515.

[246] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. "What do they capture? a structural analysis of pretrained language models for source code." In: *Proceedings of the 44th International Conference on Software Engineering*. Association for Computing Machinery, 2022, pp. 2377–2388. ISBN: 978-1-4503-9221-1.

[247] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. "Intel Math Kernel Library." In: *High-Performance Computing on the Intel Xeon Phi™: How to Fully Exploit MIC Architectures*. Springer International Publishing, 2014, pp. 167–188. ISBN: 978-3-319-06486-4.

[248] Hongzhi Wang, Mohamed Jaward Bah, and Mohamed Hammad. "Progress in Outlier Detection Techniques: A Survey." In: *IEEE Access* 7 (2019), pp. 107964–108000. ISSN: 2169-3536.

[249] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. "CloudRanger: Root cause identification for cloud native systems." In: *Proceedings - 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018*. Institute of Electrical and Electronics Engineers Inc., 2018, pp. 492–502. ISBN: 9781538658154.

[250] Weili Wang, Lun Tang, Chenmeng Wang, and Qianbin Chen. "Real-Time Analysis of Multiple Root Causes for Anomalies assisted by Digital Twin in NFV Environment." In: *IEEE Transactions on Network and Service Management* (2022), pp. 1–1. ISSN: 1932-4537.

[251] Yukihiro Watanabe, Hiroshi Otsuka, Masataka Sonoda, Shinji Kikuchi, and Yasuhide Matsumoto. "Online failure prediction in cloud datacenters by real-time message pattern learning." In: *CloudCom 2012 - Proceedings: 2012 4th IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 2012, pp. 504–511. ISBN: 9781467345095.

[252] Peter Wittek and Sandor Daranyi. "A GPU-Accelerated Algorithm for Self-Organizing Maps in a Distributed Environment." In: *Proceedings of ESANN-12, 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. 2012.

[253] Peter Wittek, Shi Chao Gao, Ik Soo Lim, and Li Zhao. "somoclu : An Efficient Parallel Library for Self-Organizing Maps." In: *Journal of Statistical Software* 78.9 (2017). ISSN: 1548-7660.

[254] Thomas Wolf et al. "Transformers: State-of-the-Art Natural Language Processing." In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2020, pp. 38–45.

[255] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. "MicroRCA: Root Cause Localization of Performance Issues in Microservices." In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9. ISBN: 978-1-7281-4973-8.

[256] Yi Xiao, Rui-Bin Feng, Zi-Fa Han, and Chi-Sing Leung. "GPU Accelerated Self-Organizing Map for High Dimensional Data." In: *Neural Processing Letters* 41.3 (2015), pp. 341–355. ISSN: 1370-4621.

[257] Yikai Xiao, Qixia Zhang, Fangming Liu, Jia Wang, Miao Zhao, Zhongxing Zhang, and Jiaxing Zhang. "NFVdeep: Adaptive online service function chain deployment with deep reinforcement learning." In: *Proceedings of the International Symposium on Quality of Service*. Vol. 19. 2019, pp. 1–10. ISBN: 9781450367783.

[258] Lu Xu, Yang Xu, and Tommy W.S. Chow. “PolSOM: A new method for multidimensional data visualization.” In: *Pattern Recognition* 43.4 (2010), pp. 1668–1675. ISSN: 00313203.

[259] Tomonari Yamagutchi, Koichi Nagata, and Pham Quang Truong. “Pattern Recognition of EEG Signal during Motor Imagery by Using SOM.” In: *Second International Conference on Innovative Computing, Informatio and Control (ICICIC 2007)*. IEEE, 2007, pp. 121–121. ISBN: 0-7695-2882-1.

[260] Song Yang, Philipp Wieder, Ramin Yahyapour, Stojan Trajanovski, and Xiaoming Fu. “Reliable Virtual Machine Placement and Routing in Clouds.” In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 2965–2978. ISSN: 10459219.

[261] Yue Yuan, Wenchang Shi, Bin Liang, and Bo Qin. “An Approach to Cloud Execution Failure Diagnosis Based on Exception Logs in OpenStack.” In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. Vol. 2019-July. IEEE, 2019, pp. 124–131. ISBN: 978-1-7281-2705-7.

[262] Matei Zaharia et al. “Apache Spark: A Unified Engine for Big Data Processing.” In: *Commun. ACM* 59.11 (2016), pp. 56–65. ISSN: 0001-0782.

[263] Zakia Zaman, Sabidur Rahman, and Mahmuda Naznin. “Novel Approaches for VNF Requirement Prediction Using DNN and LSTM.” In: *2019 IEEE Global Communications Conference (GLOBECOM)*. 2019, pp. 1–6.

[264] Ennan Zhai, Mahesh Balakrishnan, Bingchuan Tian, Ang Chen, Ruzica Piskac, Bo Song, and Haoliang Zhang. “Check before You Change: Preventing Correlated Failures in Service Updates.” In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 2020, pp. 575–589. ISBN: 9781939133137.

[265] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. “Heading off correlated failures through independence-as-a-service.” In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014*. 2014, pp. 317–334. ISBN: 9781931971164.

[266] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K. Lahiri. *Can Pre-trained Language Models be Used to Resolve Textual and Semantic Merge Conflicts?* 2021. URL: http://arxiv.org/abs/2111.11904.

[267] Kaicheng Zhang, Akhil Guliani, Seda Ogrenci-Memik, Gokhan Memik, Kazutomo Yoshii, Rajesh Sankaran, and Pete Beckman. “Machine Learning-Based Temperature Prediction for Runtime Thermal Management Across System Components.”

In: *IEEE Transactions on Parallel and Distributed Systems* 29.2 (2018), pp. 405–419. ISSN: 10459219.

[268] Xu Zhang et al. “Cross-dataset Time Series Anomaly Detection for Cloud Systems.” In: *USENIX ATC*. 2019, pp. 1063–1076. ISBN: 9781939133038.

[269] Hui Zhao, Jing Wang, Feng Liu, Quan Wang, Weizhan Zhang, and Qinghua Zheng. “Power-Aware and Performance-Guaranteed Virtual Machine Placement in the Cloud.” In: *IEEE Transactions on Parallel and Distributed Systems* 29.6 (2018), pp. 1385–1400. ISSN: 10459219.

[270] Moubarak Zoure, Toufik Ahmed, and Laurent Réveillére. “Network Services Anomalies in NFV: Survey, Taxonomy, and Verification Methods.” In: *IEEE Transactions on Network and Service Management* (2022), pp. 1–1. ISSN: 1932-4537.