A Proactive Formal Approach For Microservice-based Applications Auto-Scaling

Souheir Merkouche [1], Chafia Bouanaka [1]

[1] *LIRE Laboratory, University of Constantine 2-Abdelhamid Mehri-, Constantine, Algeria*

Abstract

Due to the emergence of cloud and containers, microservices has become widely adopted to develop large-scale applications, since deploying on the cloud provides an unlimited amount of resources to the developers. However, an uncontrolled usage of these resources leads to unnecessary costs or a non-performant system. Therefore, several researches have been carried out around an efficient resources auto-scaling, coming out with several policies. Most of the existing policies follow a reactive approach that relies on the current state of the system to adapt it. On the counterpart, proactive approaches are based on resource future usage estimation to adapt the system before it reaches a non-performant state, yet, complex and expensive methodologies are needed to ensure proactivity such as reinforcement learning. In this work, we propose a proactive approach of resource auto-scaling. We use the weak and strong dependencies concept to expect the future state of the system. To formally model the proposed approach, we combine high-level PNs and plausible PNs. The plausible PNs are suitable for decision-making, when several adaptation plans are available, they allow identifying a compromise plan when the auto-scaling concerns different qualities of the system.

Keywords

Microservice architectures, auto-scaling, containers, formal methods, Petri Nets

1. Introduction

Microservice architectures organizes the applications as a set of loosely-coupled components, evolving independently, where new versions can be added and work side by side with the old ones, hence reducing the applications downtime for maintenance and upgrades. Nowadays, microservices architectures are widely adopted, due to the emergence of cloud computing and containers technology, where the cloud provides the user with unlimited pay-per-use resources to run the application, and the containers grant a fast and easy deployment of the microservices. However, well-established policies are needed to manage the allocation/deallocation of resources to avoid supplement and unnecessary costs on one hand, and meet performance targets of the microservices on the other hand.

Auto-scaling consists of increasing/decreasing the amount of the allocated resources as a response to the workload applied on the microservices for an efficient resource utilization, a cost reduction and sustainment of the application's quality of service (QoS). Many approaches were proposed to answer those needs, where most of them are reactive, i.e. monitor the microservices and adapt them according to the current workload [5] [12] [13]. However, a reactive auto-scaling remains a long process and proactive approaches are needed to avoid the violation of the system's QoS.

Existing proactive approaches [4] require expert systems and knowledges to achieve auto-scaling, thus limiting their applicability [18]. In this paper, we propose a proactive approach for auto-scaling, in which we consider the architectural level of the system and make use of the dependency relations between microservices to establish proactive auto-scaling policies. To this end, we adopt the architecture proposed in [14] for self-adaptive systems, this architecture provides a separation between

RIF'22: The 11th Seminary of Computer Science Research at Feminine, March 10th, 2022, Constantine 2-Abdelhamid Mehri University, Algeria

EMAIL: souheir.merkouche@univ-constantine2.dz (S. Merkouche), chafia.bouanaka@univ-constantine2.dz (C. Bouanaka);

the operating system and the adaptation logic. Unlike existing approaches, our work doesn't include any complex methodologies, it is based on the strong and weak dependencies concept presented in [1] as the connected microservices to a given microservice, where strong dependencies must be provided before creating of the microservice instances, and the weak dependencies must be provided before the end of the given microservice deployment. We use this definition to determine the future state of the dependencies from the current state of a given microservice.

Application auto-scaling is a fault-sensitive operation, a wrong scaling can lead to a loss of money (resources rented without being used), or also to a reduction of the quality of service offered by the application (lack of rented resources). Therefore, we adopt formal verification to ensure that the model is reliable and thus avoiding performance degradation due to a faulty resizing. To that end, we combine High-Level Petri Nets (HLPN) with Plausible Petri Nets (PPN) where the PPNs are used to compute a compromise plan between several adaptation plans. We highlight that concurrency in our resizing model concerns the qualities of microservices and not the microservices themselves. The paper is structured as follows: in section 2, we recall basic concepts of both HLPNs and PPNs. In section 3, we present some of the existing auto-scaling approaches. Afterwards, in section 4, we propose a proactive auto-scaling approach for microservice-based applications, we introduce the adopted architecture and we present the metrics monitored to achieve auto-scaling in microservice-based applications, then we propose the policies implemented to achieve this adaptation. In section 5 we detail the PN-based model of the approach and use the email pipeline processing as a case study to model it. Finally, section 6 rounds up the paper with a summary of the presented work and ongoing.

2. Background

Petri nets have been initially proposed to model the behavior of a dynamic system with discrete events; they have then undergone several evolutions and variants. We present in what follows two types of Petri nets to be used in the present work: High level and Plausible Petri nets.

2.1. High level Petri nets

A high-level petri net (HLPN for short) [6] can be defined in several ways. In our work, we adopt the definition proposed in [7].

Definition 1 [7]: a HLPN as a directed bipartite graph $H= (P, T, A, D, Type, M_0)$, where:

- P and T are non-empty finite sets of elements called places and transitions, respectively, such that $P \cap T = \emptyset$;
- $A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs connecting places to transitions, and transitions to places;
- D is a non-empty finite set of non-empty domains where each element of D is called type;
- $Type: P \cup T \rightarrow D$ is a function used to assign types to places and transitions;
- $M_0 \in \mu PLACE$ is a multiset called initial marking of H, where $\mu PLACE$ is the set of multisets over the set $PLACE = \{(p,g) : p \in P, g \in Type(p)\}$.

In a HLPN, places are typed (see Figure 1), in order to define the collection of tokens that can be hold in each place. The collection of all the tokens associated to all the places represent the net marking. In addition, arcs can be associated with expressions, that contain constants, variables, or function images (e.g x, y, $f(x)$). To evaluate an arc expression, values are assigned to its variables, the resulting items must be inscribed in the type of the arc's place. Finally, Boolean expressions are associated to transitions and are called guards (e.g $x > y$). These features make it possible to model several data and information about the system and its different variables, so it is possible to measure the qualities of the modeled system.

A transition can fire only if it is enabled. A transition is enabled due to a net marking and a particular mode. A transition mode is defined by assigning values to all the variables in the transition's guard and the annotations of the connected arcs. After assigning values, the input arcs expressions are evaluated, the results are tokens having the same type as the input places. The transition is enabled for the current mode, if the marking of its input places is equal or superior to the multiset of the resulting tokens. The firing rule is simple; whenever a transition fires, tokens according to the multiset of the evaluated expressions are subtracted from the input places, and tokens according to the evaluation of output arcs are added to the output places.

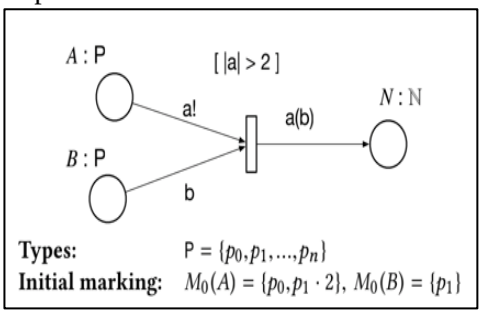

Figure 1: A HLPN example [7].

2.2. Plausible Petri nets

Plausible Petri nets (PPNs) [8] [9] are a hybrid variant of PNs composed of two types of places and transitions, namely, symbolic and numerical, in order to describe both discrete and continuous behaviors of a system. In the symbolic subnet, discrete behavior is described using regular tokens, while in the numerical subnet, continuous or numerical behavior is described with tokens that carry states of information about states of variables [10], where a state of information about a given variable $x_k \in \mathcal{X}$ is the probability density function (PDF) of x over $\mathcal{X}$[10]. In a self-adaptive system, PPNs can be used to compute the plausibility of the different possible adaption plans, so the most appropriate plan can be chosen.

Definition 2 [11]: A PPN is defined as a 9-tuple $\mathfrak{M} = (P, T, F, W, D, \mathcal{X}, \mathcal{G}, \mathcal{H}, M_0)$, where:

- *P* is the set of places partitioned into two disjoint subsets, $P^{(\mathcal{N})}$, and $P^{(\mathcal{S})}$ for numerical places, and symbolic places, respectively.
- *T* is the set of transitions partitioned into two subsets, $T^{(\mathcal{N})}$, and $T^{(\mathcal{S})}$ for numerical transitions, and symbolic transitions, respectively. Unlike places, a transition can belong to both $T^{(\mathcal{N})}$ and $T^{(\mathcal{S})}$, in this case it is referred to as a *mixed transition*.
- *F* is the set of arcs that connect transitions to places, and places to transitions.
- *W* is the non-negative set of weights applied to arcs within F, and it is partitioned into two disjoint subsets, $W^{(\mathcal{N})}$, and $W^{(\mathcal{S})}$ for arcs connected to numerical places, and those connected to symbolic places, respectively.
- *D* is the set of switching delays for both symbolic and mixed transitions.
- $\mathcal{X} \subset \mathbb{R}^d$ is the state space of a stochastic variable $\{x_k\}_{k \in \mathbb{N}}$.
- $\mathcal{G}$ is the set of density functions associated to numerical places and transitions.
- $\mathcal{H}$ is the set of equations representing the dynamics of the state variable $x_k \in \mathcal{X}$.
- M_0 is the initial marking of the net, which is given by two vectors $M_0^{(\mathcal{N})}$ and $M_0^{(\mathcal{S})}$ for numerical and symbolic places, respectively.

A PPN can be divided into two subnets, namely, the symbolic subnet, and the numerical subnet. Unlike the ordinary evolution of the symbolic subnet, the numerical subnet evolution relies on an ad-hoc information flow based on conjunction and disjunction of states of information [8][9]. For a numerical transition, it can fire when the conjunction between all its input places' states of information and the transition's states of information is possible. For a mixed transition, both conditions of symbolic

and numerical transitions must be satisfied. Firing transition's effect for the numerical places is a state of information consisting of a disjunction of the previous state of information, and the information produced after firing the transition (conjunction of state of information within the transition and its input places). Figure 2 illustrates an example of a simple PPN (given by part (a)) and its firing rules (given by (b)). For more details on PPNs, the reader is referred to [11].

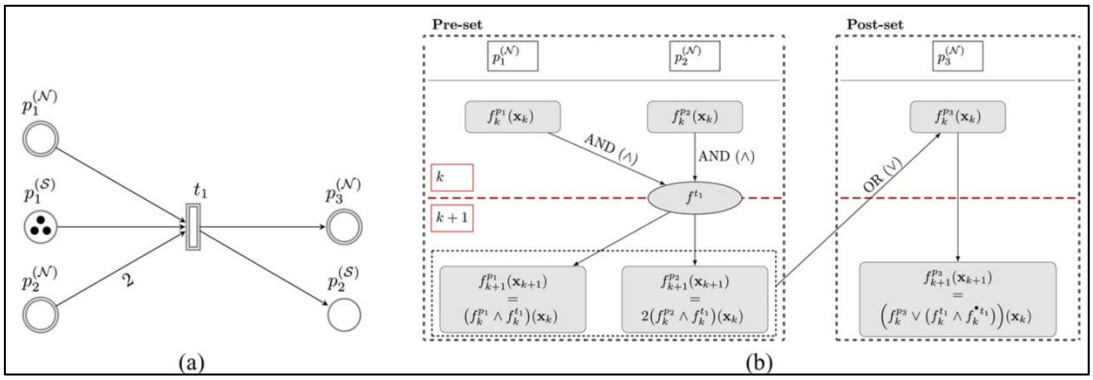

Figure 2: PPN example(a) and firing rules [11].

3. Related Work

With the expanding importance of auto-scaling, many approaches have been proposed for an efficient auto-scaling of microservices-based applications. Most of the presented works are reactive approaches; they adapt the system according to its current state. While other approaches are proactive; they adapt the system based on predicting its future state by analyzing historical data. Existing approaches generally use threshold-based policies (e.g., [19], [20], [21]), queuing theory (e.g., [22], [23],[24]) or machine learning techniques (e.g., [25], [26]).

In a threshold-based solution, static thresholds are defined for the resources usage and adaptation actions are planned according to them, allowing the definition of simple but efficient auto-scaling policies. Authors in [21] use reinforcement learning for a dynamic adaptation of the thresholds. Threshold-based policies are also used in proactive approaches such as [19] in which authors predict the CPU utilization of the microservices and then adapt them using a threshold-based policy.

In a queuing theory-based solution, each microservice is modelled as a queue of requests to predict its performance under different conditions of workload. Authors in [23] modelled a microservice-based application through a Layered Queuing Network for a dynamic scaling. However, Queuing theory solutions are considered limited due to the fact that they need to be recomputed when the workload changes.

Most of proactive approaches (e.g., [25], [26]) are machine learning-based solutions using reinforcement learning (RL). RL allows making dynamic scaling decisions after learning phases using a trial-and-error approach. However, RL techniques suffer from the long learning process and require time to converge to an optimal policy.

In the present work, we propose a proactive auto-scaling approach, with a threshold-based policy. The strong and weak dependencies concept [1] allowed us to reach proactivity without using complex and expensive methodologies (e.g., machine learning). Contrary of existing proactive approaches, our model doesn't need any learning process, since it relays on the microservice's dependencies to predict the future state of microservices.

4. A proactive auto-scaling approach

Auto-scaling approaches can be classified as proactive and reactive approaches. In a reactive approach the auto-scaler increases/decreases the resources allocated to a given microservice relying on the current state of the workload, this is achieved by monitoring the system metrics such as input data

rate and resources usage, despite the efficiency of this approach, it remains a long process due to the significant time of adaptation computing and scheduling. Conversely, in a proactive approach, the microservice is resized relying on a prediction of the future evolution of the workload and the system state using machine learning. Machine learning is a heavy and greedy process in terms of cost, also predictions can't be one hundred percent accurate which may lead to a waste of resources and unnecessary cost. [15] and [16] are examples from the few works that have considered proactive approaches. In our work, we present a proactive approach, in which we consider the architectural level of the system to achieve proactivity, by considering each microservice dependencies when scaling it, i.e. we use the monitoring result of one microservice to adapt it due to the workload applied on it, and estimate the future workload of its dependencies, and prevent an adaptation for them. In this section, we present the adopted architecture to model our approach, then we present the supervised metrics and the auto-scaling policy.

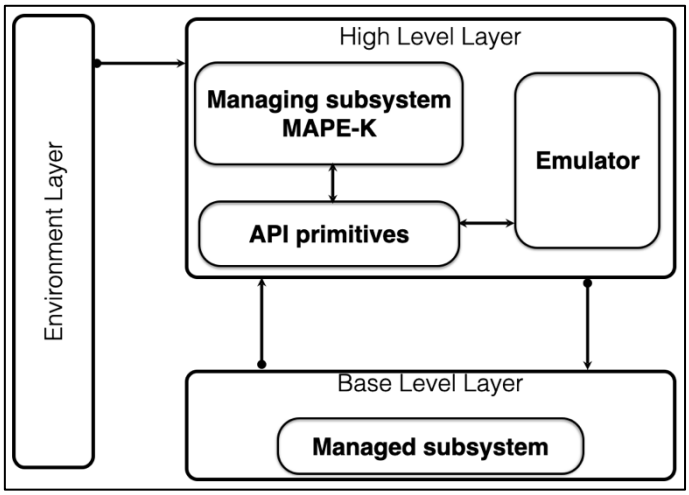

Figure 3: Layered self-adaptive system Architecture.

4.1. Modeling Architecture

We mainly adopt the architecture presented in [14], and adapt it to meet our needs. As illustrated in figure 3, this architecture is composed of a base-level layer referring to the managed subsystem, and a high-level layer representing the managing subsystem. Additionally, an emulator and a set of API primitives are defined to connect the two layers.

The logic behind this architecture is that the system process is modelled separately from its managing process, for that end, the system process, referred to by the managed subsystem is modelled in the base-level layer, while the managing logic is modelled in the high-level by means of a set of MAPE-K loops, where each loop models the adaptation process of one metric of the system. These loops use the emulator to obtain the current state of the system, and the API primitives to edit it. This feats perfectly our need to separate the application logic from the auto-scaling process. Additionally, it allows modelling the auto-scaling process independently from the application architecture, its composing microservices and distributed locations, while the managing subsystem architecture enables the modeling of multiple auto-scaling policies according to multiple objectives.

4.2. Supervised Metrics

The auto-scaling process consists of monitoring the microservices and adapting them regarding quality violations, this is measurable through a set of metrics that are directly monitored by the Cloud infrastructure and particularly the containers. For each microservice, the MAPE loop's monitor collects the following metrics:

- ***CPU usage metric:*** represents the CPU usage rate of the microservice, obtained as the sum of the microservice's replicas usage of the CPU divided by the sum of CPUs allocated to this microservice.

- ***Memory usage metric:*** it is similar to the CPU usage rate expect that the memory usage is considered.
- ***Input data rate metric:*** The input data rate is monitored at the microservice level, representing the rate of user's requests to the considered microservice.

4.3. Auto-Scaling Policy

To realize a proactive auto-scaling, we define a set of MAPE loops where each one is responsible for determining an adaptation plan that brings the system to its performant state while maintaining one of the supervised metrics. The MAPE loops share one monitor component that monitors the microservices metrics and when one of them is violated it triggers an adaptation process in the corresponding loop. After obtaining all the adaptation plans, an extended plan component is defined to make a compromise between them and execute it by one shared executor.

Authors in [1] defined the concept of strong and weak dependencies of a microservice, where the strong dependencies are microservices that must be deployed before the creation of the microservice instance, and weak dependencies are the ones that must be fulfilled before the end of its deployment. Since the auto-scaling consists on increasing/decreasing the number of instances of a microservice, then new instances of the dependencies are needed to manage them. Therefore, auto-scaling a microservice can provide us with a prediction of the necessity of resizing microservices depending on it.

The auto-scaling policy that will be adopted in our approach is as follow:

- The Shared Monitor collects metrics of each microservice from the environment layer and verifies if an adaptation is needed.
- When an adaptation is needed for a given microservice, i.e. when one of the microservice's metrics is violated, the Analyze and Plan components of the corresponding MAPE define the adaptation plan from the violated metric viewpoint.
- The Extended Plan computes a compromise adaptation plan if several metrics where violated, and computes a proactive adaptation plans for the strong dependencies, and a recommended adaptation plan for the weak dependencies.
- The Execute element applies the adaptation plan on the microservice, and the proactive adaptation plans on the strong dependencies, then it notifies the weak dependencies by their recommended adaptation plans. Figure 4 illustrate this process.

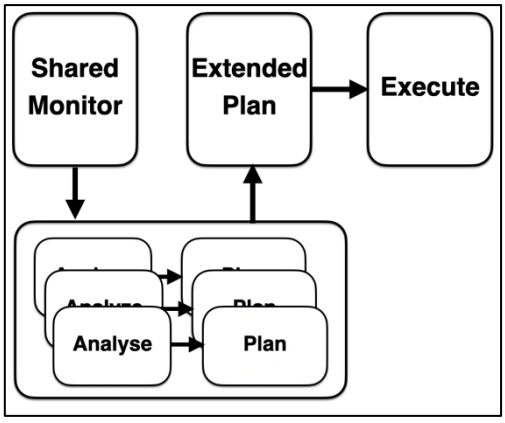

Figure 4: The Proactive Auto-Scaling Process.

The total amount of CPU and Memory allocated to the application microservices always ranges in an interval of maximal and minimal thresholds (preset at the deployment phase), to avoid under-utilization/over-utilization of resources and thus to reduce cost and avoid latency respectively.

The microservices input data rate is another supervised metric, with a maximal threshold representing the rate that the allocated resources can deal with in order to avoid the application's latency, and a minimal threshold to supervise the efficiency of usage of the allocated resource.

The adaptation plan: An adaptation plan corresponds to the number of replicas to be added or removed so that the microservice can manage the applied workload while preserving its QoS. When violations are perceived on one or multiple metrics, the corresponding loop augments/reduces the

current number of replicas and computes the new value of the metric, until it ranges again in the defined interval. This new value is computed using the following equation:

$$newV_{metric} = \frac{oldV_{metric} \times k}{k' + 1} \quad (1)$$

where $oldV_{metric}$ is the current value of the violated metric, k is the current number of replicas and k' is the new number of replicas initially set to k.

When achieving the adapted metric value, the loop returns the corresponding number of replicas as an adaptation plan, then the extended plan computes a compromise adaptation plan from all the plans as follow:

The compromise adaptation plan (CAP): The defined metrics importance and priority degrees vary from one microservice to another, therefore, a weighting function is associated to each microservice that defines a percentage foreach supervised metric according to its importance degree for that microservice. After receiving all the adaptation plans of a given microservice, the Extended Plan uses the following function to define the compromise adaptation plan:

$$CAP = \frac{\sum(P_i \times Q_i)}{nb_Q} \quad (2)$$

where P_i represents the adaptation plan proposed by the metric loop i, Q_i the priority degree of this metric and nb_Q the number of the metric loops.

***The Proactive adaptation plan*:** Foreach instance of a microservice, a number of instances of a strong dependency is needed, the Extended Plan uses this information to compute the proactive adaptation plan for each strong dependency as follow:

$$RAP = CAP_j \times N_i \quad (3)$$

where Ni is the number of instances of the microservice i that is needed to run one instance of the microservice j.

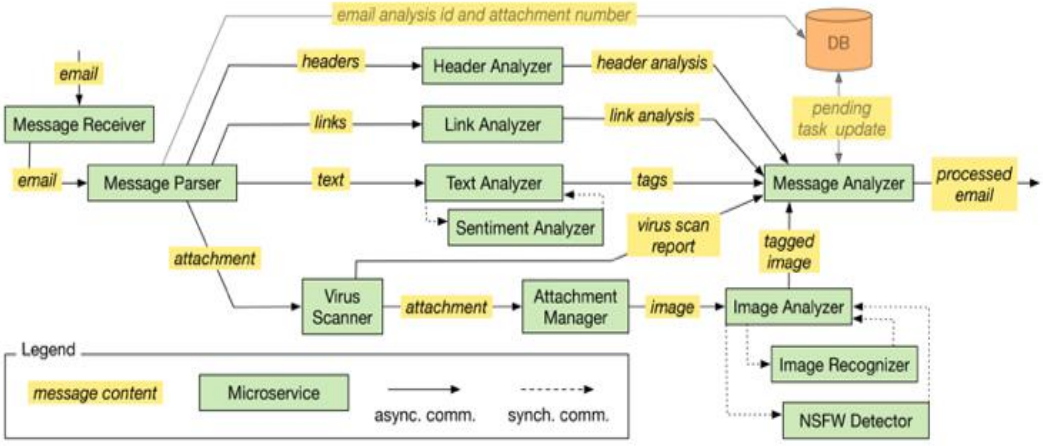

Figure 5: Microservice architecture of email processing pipeline [1].

***The recommended adaptation plan*:** is computed in a similar manner as the proactive adaptation plan, but it is not applied on the weak dependencies but sent to them, specifically to the violated metric loops. When receiving it, each loop computes the metric value associated to the plan to verify that it doesn't violate it, and if it does, it computes a new plan that preserves it.

To formally apply this policy on our model, we use a PPN in the plan component of the loops to enable computing the defined values. In the next section, we present the formal model of this approach which is based on HLPNs and PPNs. For a clear modeling of the approach we apply it on the email pipeline processing as a case study.

5. A Formal model for proactive auto-scaling

Microservice auto-scaling is a critical process in terms of cost and performance. An improper scaling can affect the system's performance or cost. Therefore, a formal approach is needed to model the auto-scaling process and validate it before its implementation. Therefore, the approach presented in section 3 for a proactive auto-scaling of microservice-based applications is formally modelled; by means of HLPNs and PPNs, in order to construct a proactive compromise auto-scaling and illustrated through the example of the email pipeline processing. We describe in what follows the PN-based model and its constituents, then we apply it on the email pipeline processing system.

5.1. The Email Pipeline Processing System

The email processing system as described in [17] is a system composed of 12 microservices working together to analyze an email as shown in Figure 5, first the *Message Receiver* receives the emails, then forwards them to the *Message Parser* which extracts data from the emails and forwards each part to the proper microservice to treat it. The treatment passes through other microservices to finally return all the results to the *Message Analyzer* (we refer the reader to [1] for a detailed description of each microservice).

In such an application, the treatment of each user request needs a processing time, so it is necessary to supervise the response time of each microservice, by monitoring the input data rate. On the other hand, due to the high-workload applied on this kind of applications, and to avoid waste of resources and extra costs, thresholds have to be defined for the resources' usage (CPU and Memory usage). In the rest of this section, we model this system by means of the proposed approach.

5.1.1. The base-level layer

The base-level layer represents the managed subsystem that we will modelled by means of a HLPN. In the managed subsystem, we are not concerned with the microservices behavior, we actually model the microservices and the connections between them. The microservices are represented by the places of the HLPN model, and their deployments are associated to those places, while the connections are modeled by the transitions and arcs connecting the places. Places in the managed subsystem are complex places. Tokens encode information representing their current state (replicas number, the amount of associated resources, strong and weak dependencies), also known as the deployment of the corresponding microservice, and when an adaptation plan is executed, the deployment information is updated.

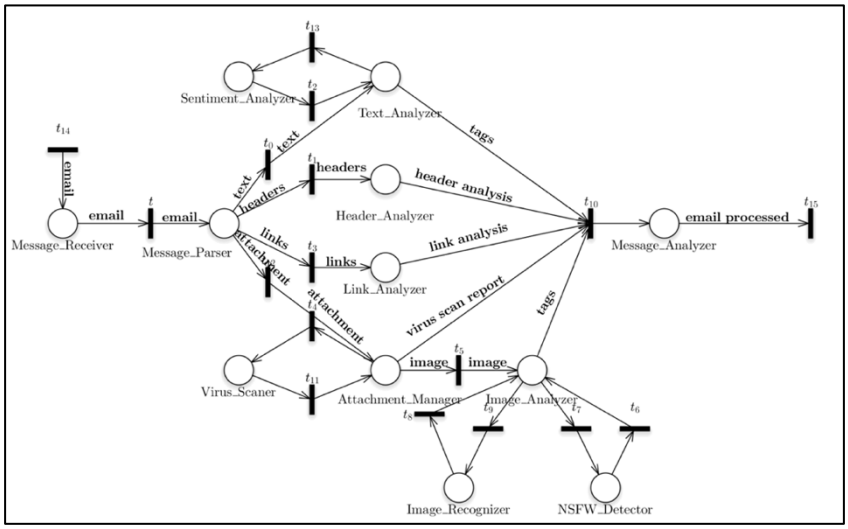

Figure 6: HLPN-based model of email processing pipeline.

The aim of this representation is to allow adding other managing subsystems to the model, that can work in parallel with the auto-scaling managing subsystem, for example a scheduler model can be added as another managing subsystem to manage allocating nodes for the new replicas and deploying them and can be warned of the adaptation made by the auto-scaling managing subsystem through the places states. What allow us of considering complex places is the encoding of this Petri net on the emulator, where places, and also all the other components of the Petri net, will be represented by tokens, then each place will be transformed to a complex token manipulated and edited by the managing subsystem. Figure 6 illustrates the HLPN model of the managed subsystem of the email pipeline processing system.

5.1.2. The environment layer

This layer refers to the physical state of the system, it contains metrics collected from the containers running the system's microservices instances. The environment layer also contains the set of available nodes where new instances can be deployed. By means of a HLPN composed of a set of places; each one representing a monitored metric of the system, and which is connected to the managing subsystem's monitor, on the other hand those places are connected to all the nodes where the system microservices instances are deployed as shown in Figure 7.

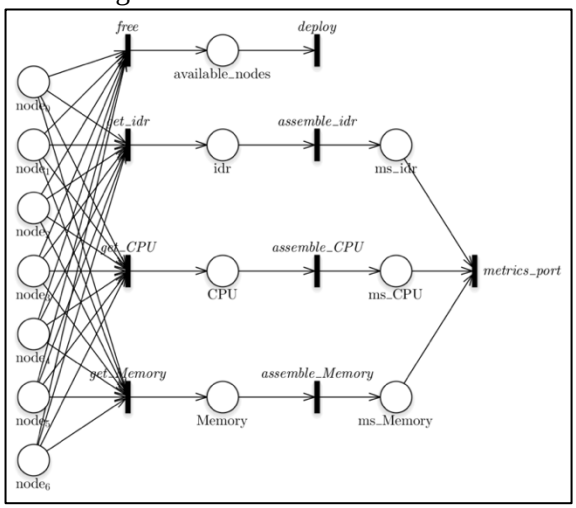

Figure 7: HLPN model of the environment layer.

Transitions *get_idr*, *get_CPU*, *get_Memory* transfer tokens from places $node_i$ to the places *idr*, *CPU*, *Memory* respectively. Transferred tokens are composed of an identifier of the corresponding microservice and a current value of the metric corresponding to one instance (*idr* for the input data rate, *CPU* for the current usage of the cpu, and *Memory* for the current usage of the memory). After collecting all the metrics, the next transitions will assemble tokens corresponding to instances of the same microservice together to obtain a single token containing the global value of the metric for one microservice. These tokens are fired to the managing subsystem monitor by the transition *metrics_port.* The *place available_nodes* contains tokens representing the set of nodes available for deploying new instances, each token is composed of the node identifier and its cpu and memory capacity, when a new instance is deployed on a certain node, the corresponding token is consumed by the transition *deploy*, whenever the instance is uninstalled, the node becomes available once again, the token is deposited in the place *available_nodes* by transition *free*. This part is not connected to the managing subsystem, since the auto-scaling concerns resize the microservices without deploying the new instances, a scheduler managing subsystem can make use of this place to find proper nodes for the new instances and schedule them.

5.1.3. The high-level layer

This layer includes three components coopering together for an efficient auto-scaling of the system:

1. *The emulator:*

An emulator is a HLPN that can encode and emulate any HLPN's behavior. The emulator set of places coincides with the set of constituents of the HLPN (places, transitions, inputs, etc.) that will be encoded into the marking of the emulator [7]. The emulator initially contains a transition fire that manipulates the encoded net, whenever this transition fires it causes the encoded net to be executed or changed. The place places is connected to the transition fire, so that when it fires tokens representing the places are obtained by the managing subsystem's monitor, in order to associate them with the metrics token obtained from the environment. The transition executes fires independently from the encoded Petri net, to edit the places' tokens by the managing subsystem. In fact, tokens flow in the managed subsystem is not considered for the auto-scaling process, it is only concerned with the places state. Figure 8 illustrates our emulator model where places of the managed subsystem are encoded as complex tokens that represent the deployment of a microservice, they are composed of:

- The microservice identifier.
- The current number of replicas.
- The amount of allocated cpu and memory.
- The needed amount of cpu and memory for one replica.
- The metrics thresholds (*max_idr, min_idr, max_cpu, min_cpu, max_memory, min_memory*).
- The sets of strong and weak dependencies.

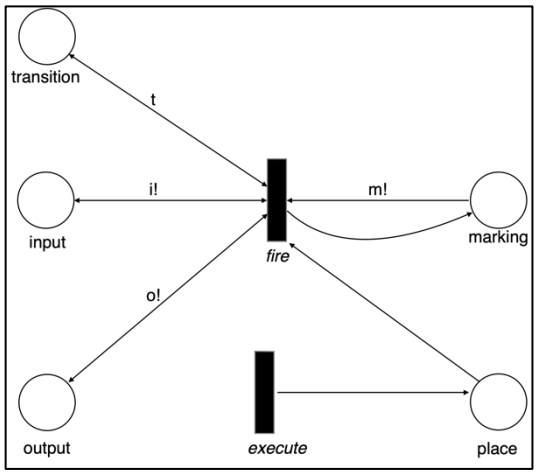

Figure 8: The emulator HLPN model.

2. *The API primitives*

The API primitives were defined by [7] as a set of transitions that allow sampling and editing the state and the structure of the managed subsystem by the managing subsystem. The API primitives are used to read and write the base-level, they allow adding new components to the net or removing existing ones (places, transitions, arcs, etc.). They represent sensors and actuators. For our model, we use the primitive *get_Tokens*, and define the primitives *set_Tokens* and *edit_Tokens*:

(a) *get_Tokens(x):=p(x)* : the primitive is connected to the place *place* of the emulator, when it fires it obtains the token *x* from *place*.

(b) *set_Tokens(x):=p(x):* also connected to *place*, it is used to put a token *x* in *place*.

(c) *edit_Tokens(x.i):=j:* the primitive updates the field *i* of the token *x* by *j*.

These primitives are used by the managing subsystem to read/edit tokens representing the places of the managed subsystem.

3. *The Managing Subsystem*

The managing subsystem models the auto-scaling policy by means of a set of MAPE loops sharing a Monitor, an Extended Plan, and the Executer, where each loop analyzes and computes an adaptation plan for the auto-scaling of the microservices. For the example of the email processing pipeline, we

considered three supervised metrics, namely *the input data rate, the CPU* and *Memory usage*. Therefore, we present a HLPN-PPN managing subsystem composed of three MAPE loops structured as follow:

Monitor: obtains from the environments the metrics values tokens, and uses the API primitive *get_Tokens* to obtain the microservices tokens from the emulator place *place*, it associates each metric token to the corresponding microservice token, then verifies if the metric values are violated by comparing it to the microservice's thresholds (*M* for maximal value of the metric, *N* for its minimal value). The violated metrics and the associated microservices are then transferred to the loops' analyzer, while the others are returned to the emulator using the API primitive *set_Tokens*. Figure 9 illustrates this Petri net.

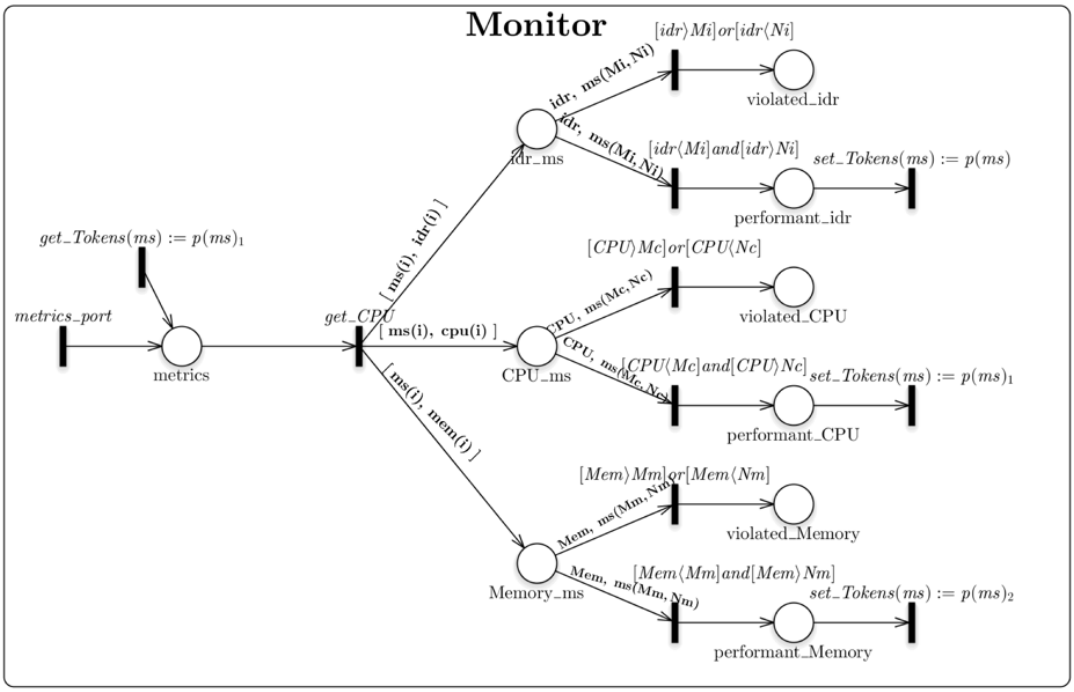

Figure 9: The Monitor HLPN model.

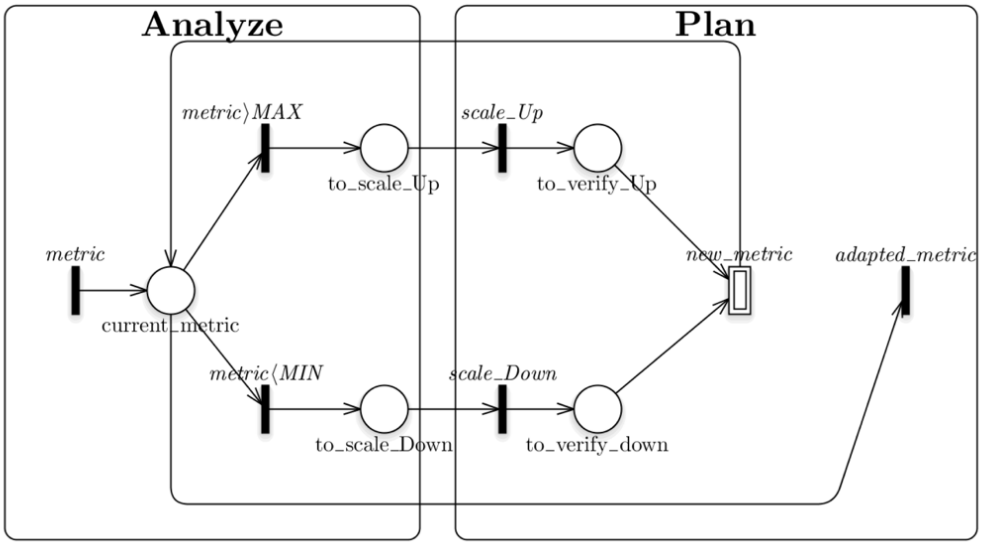

Figure 10: The metric Analyze-Plan HLPN and PPN model.

Metric Analyze and Plan : obtains tokens composed of the metric value and the corresponding microservice, verifies the auto-scaling type (Up/Down), according to which it will weather augment or reduce the replicas number, then computes the new metric value by the plausible transition *new_metric*, using equation 1 presented in section 3, it repeats the process until the metric value is adapted and the plan is fired by the transition *adapted metric*. Figure 10 illustrates a generic model of this HLPN combined with PPN.

Extended Plan: After computing the adaptation plans from the metric loops' plans, transition *combine_plans* fires a token containing a vector combining all the plans, and the corresponding microservice, then the plausible transition *compute_compromise* computes the compromise plan using equation 2 presented in section 3. Meanwhile, transitions *compute_Proactive* and *compute_Recommended* obtain the strong and weak dependencies respectively, of the microservice from the place *ms_dependencies* and compute the corresponding adaptation plans using equation 3 from section 3, then fire the strong dependencies adaptation plans to places *proactive_plans_**, and the weak dependencies adaptation plans to places *recommended_plans_**. This HLPN combined with PPN is illustrated in Figure 11.

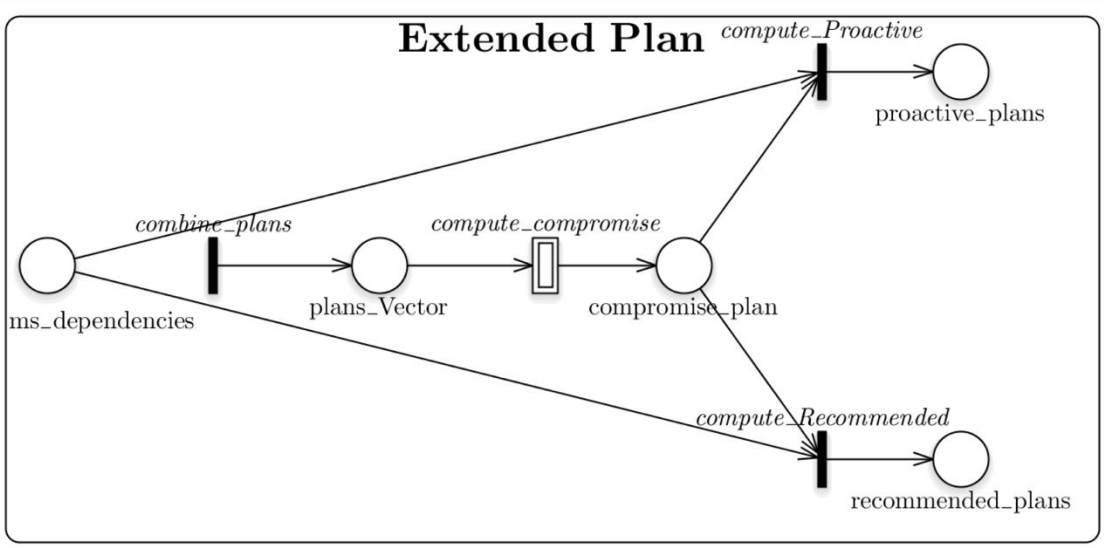

Figure 11: The Extended Plan HLPN and PPN model.

Execute: After computing all the plans, the execute apply the compromise plan using the API primitive *edit_Tokens* to edit the number of replicas, and return it to the emulator using *set_Tokens*. It does the same for the proactive plans, after obtaining the corresponding microservices tokens from the emulator using *get_Tokens*. For the recommended plans, the execute use the same process, where *edit_Tokens* is used to update the recommended plan on the token. This HLPN is illustrated in Figure 12.

6. Conclusion

In this work, we proposed a formal auto-scaling model that proactively auto-scales microservices, but still uses reactivity to define the primer scaling plan. A layered architecture was adopted to ensure separation of concerns between the application's process and the auto-scaling, it is then formally modeled using HLPNs combined with PPNs; a Petri Nets extension that allows us of computing compromise between several adaptation plans concerning the different supervising metrics of the application, enabling then auto-scaling from different point of view for the microservice-based applications.

As future work, we aim to verify and validate the presented approach to prove the efficiency of auto-scaling considering the architectural level of the system. For this purpose, we need a PN modeling tool

with the ability to model plausible functions. We also plan to model the scheduling and deployment processes to obtain a full orchestration tool for microservices-based applications.

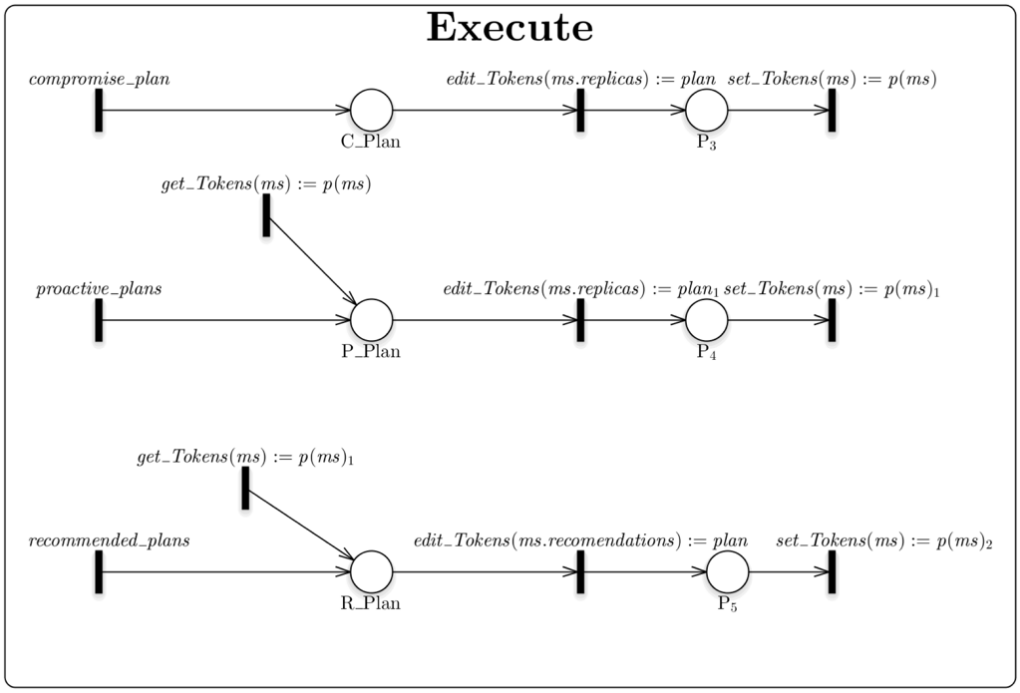

Figure 12: The Execute HLPN model.

7. References

[1] Bravetti M., Giallorenzo S., Mauro J., Talevi I., Zavattaro G. (2019) Optimal and Automated Deployment for Microservices. In: Hähnle R., van der Aalst W. (eds) Fundamental Approaches to Software Engineering. FASE 2019. Lecture Notes in Computer Science, vol 11424. Springer, Cham. https://doi.org/10.1007/978-3-030-16722-6_21

[2] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu and W. Zhou, "A Comparative Study of Containers and Virtual Machines in Big Data Environment," *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 178-185, doi: 10.1109/CLOUD.2018.00030.

[3] Lorido-Botran, T., Miguel-Alonso, J. & Lozano, J.A. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments.*J Grid Computing* **12,** 559–592 (2014). https://doi.org/10.1007/s10723-014-9314-7

[4] Rossi, Fabiana & Cardellini, Valeria & Lo Presti, Francesco. (2020). Hierarchical Scaling of Microservices in Kubernetes. 28-37. 10.1109/ACSOS49614.2020.00023.

[5] Alexander K, Hanif M, Lee C, Kim E, Helal S (2020) Cost-aware orchestration of applications over heterogeneous clouds. PLoS ONE 15(2): e0228086. https://doi.org/10.1371/journal.pone.0228086.

[6] R. Wolfgang. 1985. Petri Nets: An Introduction. Springer-Verlag New York, Inc., New York, NY, USA.

[7] Camilli, M., Bellettini, C., & Capra, L. (2018). A high-level petri net-based formal model of distributed self-adaptive systems. Proceedings of the 12th European Conference on Software Architecture Companion Proceedings - ECSA '18.

[8] Chiachío, M., Chiachío, J., Prescott, D., & Andrews, J. (2016). An information theoretic approach for knowledge representation using Petri nets. In Proceedings of the Future Technologies Conference 2016, San Francisco, 6–7 December, pp. 165–172. IEEE.

[9] Chiachío, M., Chiachío, J., Prescott, D., & Andrews, J. (2018). A new paradigm for uncertain knowledge representation by Plausible Petri nets. Information Sciences, 453, 323–345.
[10] Rus, G., Chiachío, J., & Chiachío, M. (2016). Logical inference for inverse problems. Inverse Problems in Science and Engineering, 24(3), 448–464.
[11] Chiachío, M., Chiachío, J., Prescott, D., & Andrews, J. (2018). Plausible Petri nets as self-adaptive expert systems: A tool for infrastructure asset monitoring. Computer-Aided Civil and Infrastructure Engineering.
[12] Tsagkaropoulos, A., Verginadis, Y., Papageorgiou, N. *et al.* Severity: a QoS- aware approach to cloud application elasticity. *J Cloud Comp* **10,** 45 (2021). https://doi.org/10.1186/s13677-021-00255-5
[13] A. A. D. P. Souza and M. A. S. Netto, "Using Application Data for SLA-Aware Auto-scaling in Cloud Environments,"*2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2015, pp. 252-255, doi: 10.1109/MASCOTS.2015.15.
[14] Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw (Eds.). 2013. Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers. Lecture Notes in Computer Science, Vol. 7475.
[15] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in *Proc. of IEEE ICDCS '19*, 2019, pp. 2015–2025.
[16] M. Imdoukh, I. Ahmad, and M. Alfailakawi, "Machine learning based auto-scaling for containerized applications," *Neural Computing and Applications*, vol. 32, pp. 9745–9760, 2019.
[17] Ken Fromm. Thinking Serverless! How New Approaches Address Modern Data Processing Needs. https://read.acloud.guru/thinking-serverless-how-new-approaches-address-modern-data-processing-needs-part-1-af6a158a3af1. Accessed on May, 2020.
[18] M. Wajahat, A. Gandhi, A. Karve and A. Kochut, "Using machine learning for black-box autoscaling," 2016 Seventh International Green and Sustainable Computing Conference (IGSC), 2016, pp. 1-8, doi: 10.1109/IGCC.2016.7892598.
[19] A.Bauer,V.Lesch,L.Versluis,A.Ilyushkin,N.Herbst,andS.Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in *Proc. of IEEE ICDCS '19*, 2019, pp. 2015–2025.
[20] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: Autoscaling and monitoring as a ser- vice," in *Proc. of CASCON '17*, 2017, pp. 234–240.
[21] E. Di Nitto, L. Florio, and D. A. Tamburri, "Autonomic decentralized microservices: The *Gru* approach and its evaluation," in *Microservices: Science and Engineering*. Cham: Springer, 2020, pp. 209–248.
[22] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, "DRAPS: Dynamic and resource-aware placement scheme for Docker containers in a heterogeneous cluster," in *Proc. of IEEE IPCCC '17*, 2017, pp. 1–8.
[23] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proc. of IEEE ICDCS '19*, 2019, pp. 1994–2004.
[24] F. Rossi, V. Cardellini and F. L. Presti, "Hierarchical Scaling of Microservices in Kubernetes," *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020, pp. 28-37, doi: 10.1109/ACSOS49614.2020.00023.
[25] S. Horovitz and Y. Arian, "Effificient cloud auto-scaling with SLA objective using Q-learning," in *Proc. of IEEE FiCloud '18*, Aug 2018, pp. 85–92.
[26] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. of IEEE CLOUD '19*, July 2019, pp. 329–338.