Hierarchical Auto-Scaling Policies for Data Stream Processing on Heterogeneous Resources

GABRIELE RUSSO RUSSO, VALERIA CARDELLINI, and FRANCESCO LO PRESTI, University of Rome Tor Vergata, Italy

Data Stream Processing (DSP) applications analyze data flows in near real-time by means of operators, which process and transform incoming data. Operators handle high data rates running parallel replicas across multiple processors and hosts. To guarantee consistent performance without wasting resources in face of variable workloads, auto-scaling techniques have been studied to adapt operator parallelism at run-time. However, most the effort has been spent under the assumption of homogeneous computing infrastructures, neglecting the complexity of modern environments.

We consider the problem of deciding both how many operator replicas should be executed and which types of computing nodes should be acquired. We devise heterogeneity-aware policies by means of a two-layered hierarchy of controllers. While application-level components steer the adaptation process for whole applications, aiming to guarantee user-specified requirements, lower-layer components control auto-scaling of single operators. We tackle the fundamental challenge of performance and workload uncertainty, exploiting Bayesian optimization and reinforcement learning to devise policies. The evaluation shows that our approach is able to meet users' requirements in terms of response time and adaptation overhead, while minimizing the cost due to resource usage, outperforming state-of-the-art baselines. We also demonstrate how partial model information is exploited to reduce training time for learning-based controllers.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Computer systems organization** → Distributed architectures; Self-organizing autonomic computing; • **Information systems** → **Stream management**.

Additional Key Words and Phrases: Auto-Scaling, Data Stream Processing, Resource Management, Reinforcement Learning

ACM Reference Format:
Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. 2023. Hierarchical Auto-Scaling Policies for Data Stream Processing on Heterogeneous Resources. *ACM Trans. Autonom. Adapt. Syst.* 1, 1, Article 1 (January 2023), 44 pages. https://doi.org/10.1145/3597435

1 INTRODUCTION

Data analytics applications have become increasingly important in several domains, such as smart manufacturing or remote healthcare, empowered by the increasing availability of rich data sets and the ongoing development of powerful data analysis techniques. *Data Stream Processing* (DSP) has emerged as a *de facto* standard to process high-volume data flows in (near) real-time [17]. Data *streams* are unbounded, ordered sequences of data units (i.e., *tuples* or *records*) emitted by one or more sources. DSP applications are usually defined as directed acyclic graphs (DAGs), whose

Authors' address: Gabriele Russo Russo, russo.russo@ing.uniroma2.it; Valeria Cardellini, cardellini@ing.uniroma2.it; Francesco Lo Presti, lopresti@info.uniroma2.it, University of Rome Tor Vergata, Rome, Italy.

1556-4665/2023/1-ART1 $15.00
https://doi.org/10.1145/3597435

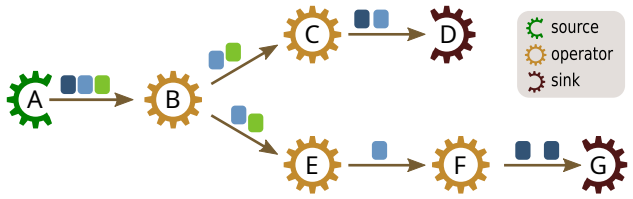

Fig. 1. Example of a DSP application graph, composed of *sources*, which emit data streams, *operators*, which apply some processing logic to the streams, and *sinks*, final consumers (e.g., dashboards, external databases).

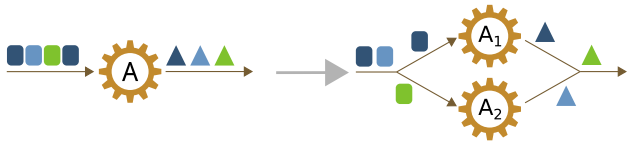

Fig. 2. Horizontal scaling of a DSP operator, where multiple parallel replicas are executed to sustain higher input rates.

vertices are data sources and operators, and the edges represent streams flowing between them (see, Figure 1). *Operators* receive one or more streams as input, apply their processing logic (e.g., filtering, aggregation) and emit a new stream as the result, possibly updating an internal state (*stateful* operators). Data streams eventually reach *sink* vertices, which act as consumers of the produced results (e.g., dashboards, databases).

In practice, parallel replicas (or, instances) of each operator are executed across multiple – and possibly heterogeneous – CPUs and hosts to sustain higher data rates. In this context, *elasticity* is an essential feature for modern DSP systems [17, 50] that enables dynamic resource provisioning in response to workload and condition changes. Elasticity is mainly implemented by means of horizontal operator auto-scaling that is, modifying the number of operator replicas during execution, as illustrated in Figure 2. Scaling decisions should be taken at run-time to provision enough computational capacity to meet users' requirements (e.g., maximum processing latency), while avoiding resource wastage due to over-provisioning. This problem, which has been widely investigated for years in the field of Cloud computing [4], is particularly challenging in the context of DSP, because parallelism reconfigurations come at the cost of significant overhead. Indeed, to preserve stream and state integrity, specific reconfiguration protocols must be adopted, slowing down or (more likely) interrupting normal data processing [24].

As surveyed in [5, 10, 50], auto-scaling DSP applications has been widely investigated in the literature, but there are still open issues that motivate research on the topic. Indeed, existing solutions mostly consist of either best-effort heuristics (e.g., threshold-based policies), which do not allow users to fine-tune their requirements, or model-based approaches, which are hard to adopt in practice because of the difficulty of obtaining accurate models as new applications are deployed. Moreover, state-of-the-art solutions mostly neglect the existence of heterogeneous computing nodes, assuming that any host will provide the same computational capacity. Unfortunately, this assumption is very far from the reality of modern Cloud and Fog/Edge infrastructures, where providers offer computing nodes (e.g., virtual machines or containers) of multiple *types*, characterized by different computational capacity, energy consumption and price.

In this paper, we present an autonomic hierarchical approach to derive auto-scaling policies for DSP applications deployed over heterogeneous computing infrastructures. We tackle the problem

of minimizing the cost due to resource usage while meetings users' Quality-of-Service (QoS) requirements concerning application performance and scaling overheads. To this end, we rely on a two-layered hierarchy of controllers, where an *Application Manager* (AM) steers the adaptation process for each running application and coordinates lower-layer decentralized *Operator Managers* (OM), which control auto-scaling of single operators. Our control architecture is based on the *Hierarchical Control* pattern for decentralized self-adaptation [68], which avoids the scalability limitations of fully centralized controllers as well as the lack of coordination of fully decentralized schemes.

To cope with uncertainty about application performance, infrastructure conditions as well as workload dynamics, we design Operator Managers as *reinforcement learning* (RL) agents. RL is a collection of learning methods for sequential decision-making [62], where agents (i.e., the OMs) must choose suitable scaling decisions over time so as to minimize the long-term value of a cost function, which – in our formulation – accounts for operator performance, resource usage and scaling overhead. A key challenge with RL methods is the possibly long time required to learn a good policy. This is especially important when no historical information is available to train agents off-line and, instead, the whole training happens on-line (i.e., while the application runs). We tackle this challenge (i) exploiting neural networks (NNs) and deep RL [43], which allow agents to learn over a reduced parameter space; and (ii) integrating partial model knowledge in the learning algorithm, by means of the *post-decision state* [39] abstraction, which reduces the amount of information to learn at run-time.

To make sure that decentralized OMs contribute to the satisfaction of application-level requirements, AMs must suitably tune the local multi-objective cost function of each OM, specifying a performance target to guarantee for each operator and properly weighting the different objective terms (i.e., resource usage, adaptation overhead, performance). We tackle this challenge by means of black-box optimization techniques and, in particular, *Bayesian optimization* [18]. By constructing an approximate application model during execution, the AM adapts the configuration of each OM according to user's QoS requirements. In such a way, our solution allows the user to express high-level QoS requirements (e.g., the application response time must not exceed 500 ms for more than 5% of the time), without the need of manually translating them into lower-level parameters (e.g., utilization thresholds).

The main contributions of this paper can be summarized as follows:

- RL-based auto-scaling algorithms for DSP operators in a heterogeneous computing infrastructure, which combine DNNs for value function approximation and post-decision states to incorporate partial knowledge about the system.
- An approach based on Bayesian optimization to steer application adaptation by dynamically tuning the local objectives of lower-layer decentralized auto-scaling controllers, relieving users from tuning low-level control knobs.
- An extensive numerical evaluation that considers different heterogeneity scenarios, workloads and applications. Our results demonstrate the benefits of the RL-based policies over state-of-the-art heuristic policies and show how our two-layered control hierarchy manages to minimize resource usage while guaranteeing the desired service level.

While we focus on a particular class of distributed applications (i.e., DSP), our approach for hierarchical learning-based adaptation is relevant and suitable for other service-based applications, which are also typically modeled as DAGs (e.g., microservices and serverless applications).

The remainder of this paper is organized as follows. In the next section, we review related works from the literature. In Sec. 3, we give an overview of the scenario we consider and the architecture of our solution. Then, following a bottom-up approach, we illustrate the RL-based policies for the

Operator Managers in Sec. 4, before presenting the upper-layer policy for the Application Manager in Sec. 5. We evaluate our solution in Sec. 6 and conclude in Sec. 7.

2 RELATED WORK

DSP applications deal with unbounded data sets and, as such, are usually long-running. Therefore, working conditions and workloads may vary during execution and applications may need to adapt to the new conditions to keep a consistent service level. Not surprisingly, researchers have spent a lot of effort investigating strategies to make DSP systems capable of *self-adaptation*. As surveyed in [10], a variety of mechanisms have been considered to modify one or more system aspects during execution, including, e.g., operator migration, dynamic stream routing, load shedding.

Operator scaling is the most investigated mechanism so far [10], as elasticity is fundamental to ingest data at dynamic rates and process them in a timely and cost-efficient manner. Elasticity may be achieved by means of either horizontal or vertical operator scaling. The former consists in changing the number of parallel operator replicas, whilst the latter modifies the amount of resources allocated to the existing replicas. Vertical operator scaling is generally easier to implement, because it does not require operator or state movements across different nodes, but at the same time it enables limited scalability (i.e., limited by the capacity of the currently used processor or node). For this reason, with a few exceptions (e.g., [14, 27, 40, 53, 59]), the majority of the existing solutions focus on horizontal auto-scaling, as we do in this paper. Therefore, in the following we will use the term "scaling" referring to horizontal operator scaling, unless differently specified.

Given their popularity in the last decade, solutions for operator auto-scaling have been the subject of specific surveys, such as [5, 50]. An essential limitation we found in most of the existing solutions regards the underlying computing infrastructures, which are assumed to comprise homogeneous computing nodes, providing identical performance. An exception to this trend is represented by [55], where a heuristic policy is proposed for auto-scaling on heterogeneous nodes. However, their policy only considers throughput as the key objective metric, whereas we are interested in a more general QoS optimization scenario, involving additional metrics such as resource cost and adaptation overhead. Beyond auto-scaling, resource heterogeneity has been considered in the field of DSP when studying other problems, such as operator placement and migration (see, e.g., [31]), as well as query processing on heterogeneous CPU/GPU systems (see, e.g., the works on DSP surveyed in [51]).

Operator auto-scaling solutions differ from each other along various dimensions and, in particular, the methodology used to decide when and how scaling operations should be performed (i.e., the policy). Several works, e.g., [16, 20, 22, 66], adopt simple threshold-based policies, where scaling actions are triggered based on the monitored resource utilization. These policies benefit from their simplicity and the lack of centralized controllers, which might limit scalability. While thresholds must be usually hand tuned by users, approaches to automatically update them have also been proposed, e.g., in [26, 36]. Given the popularity of threshold-based policies, we will compare our solution to them.

Various techniques have also been used to devise more complex auto-scaling policies, such as integer linear programming (ILP) [9], control theory [13], queueing theory [19, 35], game theory [41]. These approaches aim to optimize one or more metrics (e.g., processing latency, throughput, resource usage), often relying on some kind of system model. While most of them rely on centralized controllers, in [41] a fully decentralized game-theoretic approach has been proposed, where the control logic is distributed on each operator.

As the use of machine learning (ML) techniques in self-adaptive systems have become increasingly popular in the last decade, as surveyed in [21], their adoption for DSP operator auto-scaling has been fostered as well (e.g., [28, 30, 36, 44]). ML approaches have been mainly used for load prediction,

thus enabling proactive scaling decisions. For instance, in [30] multiple regression techniques are exploited to forecast the operator input rate and adjust the parallelism accordingly. For similar purposes, Lombardi et al. [36] rely on artificial neural networks, whose predictions are used as input for a threshold-based scaling policy.

Reinforcement learning (RL) is a branch of ML that suits well the problem of adaptation control and has been applied to drive DSP adaptation in, e.g., [8, 12, 25, 32, 36, 54, 58]. Focusing on operator scaling, in [25] RL is used considering a reward function that accounts for operator resource utilization: the closer to a pre-defined target value, the higher the agent reward. They use the SARSA algorithm, which is a model-free RL algorithm. RL-based operator scaling has also been investigated by our group in [8], where we focused on model-based RL algorithms, which allow for significant reduction of the training phases. Operator scaling is also considered by [36], where RL is only used to learn the optimal utilization thresholds for their policy. None of these solutions deals with heterogeneous computing nodes.

A work similar to ours has been recently proposed in [69], where the auto-scaling problem is formulated as a finite-horizon Markov Decision Process (MDP) and solved using RL. While they study a heterogeneous computing environment, decisions on the type of node to use (i.e., Edge, Fog or Cloud) are made at deployment time and not incorporated in the auto-scaling problem, differently from our solution. Their MDP optimizes multiple weighted objective terms. As the task of tuning these weights is left to the users, the policy we propose for the AM in Sec. 5 could be integrated in their work to support higher-level QoS specifications.

RL has also been used to control different DSP adaptation mechanisms, such as operator placement in [32] and [58] and micro-batch scheduling in [12]. In particular, Li et al. [32] tackle the placement problem and, to deal with very large state spaces, resort to DNNs for function approximation. Clearly, the adoption of RL methods for system management has gained popularity beyond the field of DSP systems, with applications to virtual machine (VM) and container auto-scaling [47, 52, 64], VM consolidation [15], container migration [63], power management [33], network-level adaptation [38], scheduling [23]. In particular, Tesauro et al. [64] use a hybrid RL approach for Cloud resource allocation, combining queueing models and learning. Mastronarde and van der Schaar [39] rely on the concept of *post-decision state* (PDS) to devise a RL algorithm that exploits the available knowledge about the system in their solution for energy-efficient wireless communication. Similarly to these works, we will try to incorporate partial knowledge and models into the learning algorithms, leveraging the PDS abstraction as in [39].

We complement the RL-based auto-scaling agents with a higher-layer Application Manager, which steers the adaptation process by tuning their local objectives (i.e., the cost function used in their local optimization problem). A similar latency apportioning problem, where the response time bound for each DSP operator must be identified, has been previously tackled by [49], relying both on an ILP formulation and a heuristic approach. However, their work assumes that the quality of different latency bound configurations can be analytically evaluated by means of a cost function, formulated in terms of estimated response time only. We take the more general perspective where such cost function is not given, as it happens in practice. Therefore, we will resort to black-box optimization and, specifically, Bayesian optimization (BO).

The adoption of BO for configuration tuning of computing systems is not novel, as emerges from [56]. Popular applications of BO include hyperparameter tuning for ML systems [60], parameter optimization for high-performance computing applications [42], automated configuration of serverless functions [2]. In [29] and [65], BO has also been applied for configuration tuning of DSP applications and frameworks. For example, in [29] BO is applied to configure Apache Storm applications, where it leads to performance improvements of at least an order of magnitude with a limited experimental budget.

3 OVERVIEW

We present a hierarchical auto-scaling solution for DSP applications, which adapts the parallelism level of operators at run-time in order to keep the desired service level in face of workload fluctuations while, at the same time, minimizing the costs due to resource usage. We target DSP systems deployed in *heterogenous* computing infrastructures where computing nodes (e.g., virtual machines, containers) can be acquired on-demand and differ in price and computational capacity.

3.1 DSP Application Model

A DSP application is defined as a directed acyclic graph $G_{dsp} = (V_{dsp}, E_{dsp})$, where V_{dsp} is a set of vertices (i.e., data sources, operators and sinks) and E_{dsp} a set of edges (i.e., data streams flowing between vertices). Data sources are vertices with no incoming edges and generate data streams. Sinks are vertices with no outgoing edges and represent final information consumers (e.g., dashboards, database systems). Processing operators instead take one or more streams as input, apply some transformation or function to the data, and output a new stream. By chaining and interconnecting several operators, DSP applications can implement complex processing functions.[1] For instance, the example application shown in Figure 1 is represented as a DAG with $V_{dsp} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}\}$ and $E_{dsp} = \{(\mathsf{A}, \mathsf{B}), (\mathsf{B}, \mathsf{C}), (\mathsf{B}, \mathsf{E}), (\mathsf{C}, \mathsf{D}), (\mathsf{E}, \mathsf{F}), (\mathsf{F}, \mathsf{G})\}$.

In practice, modern DSP frameworks (e.g., Apache Flink[2], Apache Storm[3]) provide developers with high-level programming libraries to define the application logic using general-purpose programming languages (e.g., Flink supports Java, Scala and Python). These libraries at least allow developers to define the processing logic of each operator and specify how different operators are inter-connected (i.e., how they exchange data streams). Some frameworks also feature higher-level interfaces and libraries that ease the development of specific classes of applications (e.g., *FlinkCEP* for *Complex Event Processing*). Once written using any of these libraries, applications are submitted to a DSP engine, which parses them and builds an internal representation based on the graph model introduced above (e.g., called *JobGraph* in Flink).

We consider applications concerned with QoS requirements revolving around (i) *performance* aspects and (ii) *adaptation overheads*. The former capture performance-based expectations in terms, e.g., of application response time or throughput. Since DSP applications are usually latency-sensitive [61], in this work we focus on requirements formulated in terms of maximum application response time. For instance, users may require application response time to exceed a given value R_{π}^{max} no more than 5% of the time.

Besides performance requirements, we are concerned with the overheads due to parallelism adaptation, which are usually significant for DSP applications (see, e.g., [8, 57]), as specific reconfiguration protocols must be used to preserve state and stream integrity, possibly forcing application execution to be paused in the process. For these reasons, users may require parallelism reconfiguration to be limited within a given fraction of time (e.g., no more than 10% of the time).

3.2 Computing Infrastructure Model

For application execution, we consider computing infrastructures managed by an external resource provider and offered under a pay-as-you-go cost model. As often the case in cloud computing, the provider offers different types of computing resources where operator replicas can run, possibly including different VMs (e.g., c4.large, t2.small in AWS EC2) or different container configurations

[1] An in-depth discussion of the DSP paradigm is beyond the scope of this paper. We refer interested readers to [3, 7] for more details about the stream processing paradigm and its semantics.
[2] https://flink.apache.org/
[3] https://storm.apache.org/

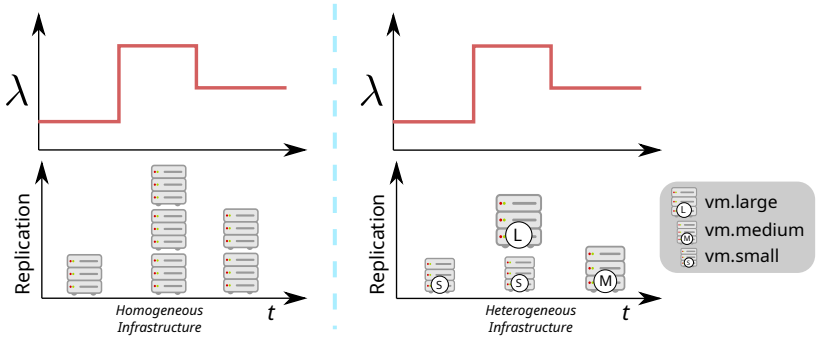

Fig. 3. In presence of heterogeneous computing nodes, the auto-scaling controller must not only decide how many replicas to run, but also which type of node to use for them.

(e.g., with different CPU shares and memory reservations). Hereafter, we will use the expression "computing node" (or, simply, node) to refer to any instance of the computing resources that can be acquired and can host application operators, regardless of the underlying technological aspects.

Specifically, we consider a resource provider offering N_{res} types of computing nodes exhibiting a trade-off between performance and price. We will denote by T_{res} the set of the available node types, i.e., $N_{res} = |T_{res}|$. Each node type $\tau \in T_{res}$ is associated with a monetary cost c_τ, which is paid for deploying an operator replica on a node of type τ for a reference time period.[4]

Note that our model is general enough to support fine-grained resource reconfiguration through, e.g., vertical scaling. Indeed, we do not make assumptions about the infrastructure-level operations required to switch between different node types in our model (e.g., spawning a new VM or scaling up an existing VM).

Example: VM-based infrastructure. We may have $T_{res} = \{\texttt{vm.small}, \texttt{vm.medium}, \texttt{vm.large}\}$, where `vm.small` has 1 CPU core and 0.5 GB of memory `vm.medium` has 2 cores and 1 GB of memory, and `vm.large` has 4 cores and 2 GB of memory. The usage cost for `vm.small`, `vm.medium` and `vm.large` is, respectively, 0.005 \$/h, 0.01 \$/h and 0.02 \$/h. Figure 3 illustrates how an auto-scaler can exploit the heterogeneous VMs introduced in this example to handle load variations.

Example: container-based infrastructure. We may have $T_{res} = \{32, 64, 128, 256, 512, 1024\}$, where each $i \in T_{res}$ represents a container configuration where the container is given i shares of CPU and a fixed amount of memory. The usage cost for container i is computed as $i \times 0.001$\$/h. Such a model may represent a scenario where a container orchestration platform is used (e.g., Kubernetes). Such platforms allow for fine-grained resource allocation to containers and, thus, support heterogeneous computing nodes, in the sense explained above. While platforms like Kubernetes provide mechanisms to acquire and release heterogeneous containers, our solution provides a DSP system with the control logic to exploit such heterogeneity.

3.3 Proposed Auto-Scaling Solution

DSP applications scale their execution over the available computing nodes and, as already explained, exploit parallel operator replicas to handle high data rates. The operator auto-scaling problem we consider aims to minimize the monetary cost paid for resource usage, while guaranteeing application QoS requirements.

[4]In practice, a monetary cost may be associated with the usage of a computing node independently of what it hosts. In this case, the cost c_τ still guides the auto-scaler towards better configurations, while actual money savings may be achieved through consolidation.

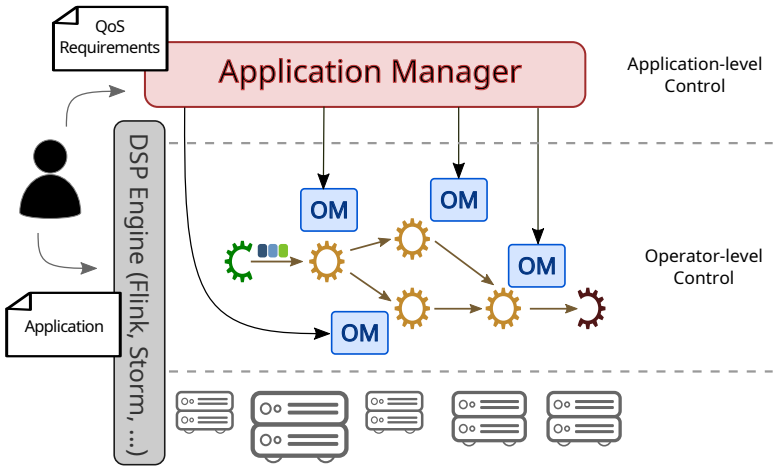

Fig. 4. Our two-layered auto-scaling framework exploits an application-level Application Manager, which supervises adaptation and aims to meet user-given requirements, and decentralized Operator Managers, which control auto-scaling of single operators. The Application Manager is responsible for translating global requirements into local requirements for each operator.

We tackle this problem by means of a two-layered hierarchical auto-scaling solution, whose architecture is adapted from our previous work [8]. Leveraging such hierarchical approach, we aim to overcome both the limitations of fully centralized controllers, which suffer from limited scalability, and fully decentralized schemes, which do not easily provide guarantees on application-level QoS. Our solution, which is illustrated in Figure 4, comprises two main components. At the higher layer, an *Application Manager* (AM) supervises the adaptation of the application as a whole, taking into account user-specified QoS requirements. At the lower layer, each operator is associated with an *Operator Manager* (OM), which controls auto-scaling for that operator. Since OMs have a local vision of the system (i.e., limited to a single operator), they are concerned with local adaptation objectives, specified – as explained below – through a multi-objective cost function that accounts for resource usage, performance requirements and adaptation overhead. Therefore, a key responsibility of the AM in our solution consists of properly tuning the local objective of each OM, by means of weights and performance targets, according to the global application-level objective (i.e., minimizing resource usage cost and meeting application QoS requirements).

Both the AM and the OMs are organized according to the well-known *Monitor, Analyze, Plan and Execute* (MAPE) pattern for autonomous software systems. As such, our auto-scaling control architecture represents an instance of the *Hierarchical Control* pattern for decentralized self-adaptation, as presented in [68]. According to this pattern, illustrated in Figure 5, decentralized controllers are organized in a hierarchy, with separation of concerns and time scales. Specifically, loops at lower layers guarantee timely adaptation concerning the part of the system under their direct control (i.e., an operator, in our solution). Higher layers operate at a longer time scale with a more global vision that accounts for the overall adaptation objectives. Note also that, on a control-theoretic perspective, our approach resembles an instance of *cascade control* [6], where an *outer* controller (i.e., the AM) attempts to keep the output close to the reference by manipulating the reference input of an *inner* controller (i.e., the OM). Cascade control is especially useful when the inner loop involves less delay and dynamics, and its tight feedback reduces effects of disturbances and uncertainties.

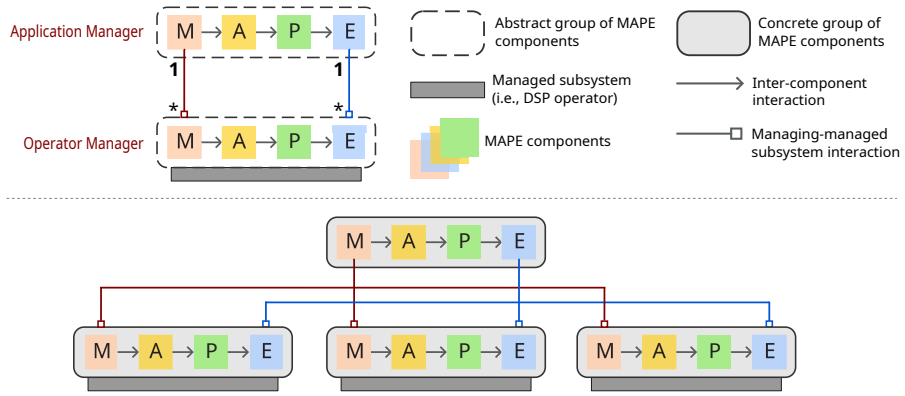

Fig. 5. Our two-layered framework is an instance of the hierarchical control pattern [68] for decentralized adaptation, where MAPE-based controllers at different layers act with separation of concerns and time scales. Top: hierarchical control pattern. Bottom: example of a concrete instance of the pattern.

OMs directly interact with operators and act independently from one another. The key responsibilities of their internal MAPE components can be summarized as follows:

- *Monitor*: collection of metrics regarding the operator activity and deployment (e.g., CPU usage, data arrival rate, current number of replicas);
- *Analyze*: application of models and/or policies to the available metrics to identify auto-scaling actions to perform. The OM aims to optimize a multi-objective cost function that comprises (i) its local cost due to resource usage, which depends on how many replicas it run and which computing nodes host them; (ii) adaptation overheads it incurs triggering scaling operations; (iii) performance penalties, based on a local response time requirement R_u^{max}, for each operator u;
- *Plan*: based on the analysis result, planning of a specific auto-scaling operation (e.g., choosing the exact replica to terminate in case of scale-in);
- *Execute*: activation of new operator replicas or termination of existing ones, possibly by means of specific reconfiguration protocols.

The cost function and the performance requirement for each OM are periodically updated by the higher-layer AM, which relies on the following MAPE components:

- *Monitor*: collection of metrics regarding the overall application performance (e.g., response time) as well as monitoring data coming from OMs (e.g., average arrival rate per operator);
- *Analyze*: application of models and/or policies to the available metrics to identify suitable weights for the local OM cost functions as well as local performance requirements to optimize the overall objective;
- *Plan*: based on the analysis result, identification of the local updates to be communicated to OMs;
- *Execute*: communication of new objective configuration to OMs.

According to the high-level description provided above, it is clear that MAPE components at different layers do interact with each other – as also illustrated in Figure 5. In particular, (i) monitoring information directly collected by OMs for each operator can be aggregated and shared with the AM-level Monitor component; (ii) adaptation actions executed at the higher layer directly impact on the OMs, as they alter their objective configuration.

It is worth remarking that in this work we focus on the definition of control policies for both the AM and the OM and we do not investigate issues related to the implementation of monitoring functionalities and adaptation execution. In the next sections, we will follow a bottom-up approach and we will first describe in Sec. 4 how the auto-scaling problem is tackled at operator-level by the OMs, before discussing in Sec. 5 how the AM adapts the local objectives to optimize the overall cost and QoS.

4 OPERATOR AUTO-SCALING THROUGH RL

The OM controls auto-scaling for a single DSP operator, aiming to satisfy a response time requirement, while also minimizing resource usage and scaling overheads. In this section, we formulate this problem as a discrete-time Markov Decision Process and, then, we discuss the challenges related to its resolution and how we tackle them by means of RL.

4.1 Operator Model and Problem Formulation

We assume that the operator scaling controller is activated periodically and, hence, we consider a slotted time model with intervals of length T_{OM} (i.e., the i-th time slot corresponds to the time interval $[iT_{OM}, (i+1)T_{OM}]$). The operator deployment at the beginning of time slot i is defined by the vector $\mathbf{k}_i$, with $k_{\tau,i} \geq 0$ being the number of replicas deployed on nodes of type τ. At any time, $1 \leq \sum_\tau k_{\tau,i} \leq K^{max}$, as (i) at least one instance must be running for each operator, and (ii) we consider a maximum parallelism level K^{max}. We denote by λ_i the average input data rate measured during slot $i-1$ (i.e., the previous slot).

We formulate the operator auto-scaling control problem as a discrete-time Markov Decision Process (MDP). An MDP is defined by a 5-tuple $\langle \mathcal{S}, A, p, c, \gamma \rangle$, where $\mathcal{S}$ is a finite set of states, $A(s)$ a finite set of actions for each state s, $p(s'|s,a)$ are the transition probabilities from state s to state s' given action $a \in A(s)$[5], $c(s,a)$ is the immediate cost[6] when action a is executed in state s, and $\gamma \in [0,1]$ a discount factor that weights future costs.

The state of the system at time i is defined by the pair $s_i = (\mathbf{k}_i, \lambda_i)$, that is the current operator deployment and the input data rate. For the sake of analysis, we consider a discrete state space, where $\lambda_i \in \{0, \bar{\lambda}, 2\bar{\lambda}, \ldots, N_\lambda \bar{\lambda}\}$ and $\bar{\lambda}$ is a suitable quantum.

For each state s, the set of the available actions $\mathcal{A}(s)$ is defined as follows:

$$\mathcal{A}(s) = \{\omega\} \cup \left\{ (\delta, \tau) : \begin{smallmatrix} \delta \in \Delta(s) \\ \tau \in T_{res} \end{smallmatrix} \right\} \tag{1}$$

where ω is the "do nothing" action that leaves the operator deployment unchanged, and reconfiguration actions are denoted by pairs (δ, τ), which specify the number $\delta \in \Delta(s)$ of replicas to add/remove on nodes of type τ. Specifically, we let $\Delta(s) = \{+1, -1\}$ for all states, except for those where the operator parallelism is 1 and $\Delta(s) = \{+1\}$ (we cannot have less than one replica), or the parallelism is equal to K^{max} and $\Delta(s) = \{-1\}$ (we cannot add more instances).

State transitions occur as a consequence of reconfiguration decisions and tuple arrival rate variations. Following an action a, the operator deployment $\mathbf{k}$ is (possibly) updated according to a into $\mathbf{k}'$. In the following, we will describe a deployment reconfiguration as $\mathbf{k}' = \mathbf{k} \oplus a$. Let us denote

[5]To improve readability, we will omit the time subscripts where possible, denoting state s_i simply as s, and the next state s_{i+1} as s'.

[6]In this section – with a slight abuse of notation – we adopt the MDP terminology and use the term "cost" to indicate the penalty value incurred by the agent after performing any action. This cost must not be confused with the application resource usage cost that represents the objective of our problem, as presented in the previous section.

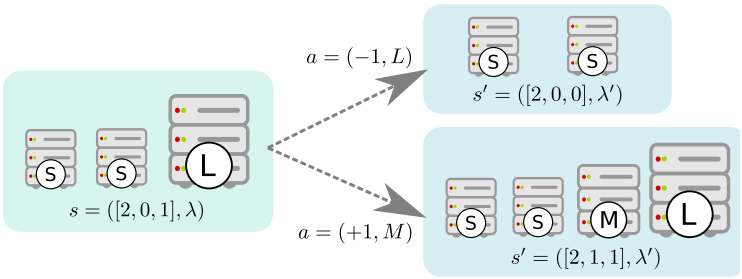

Fig. 6. Example of states and actions from the MDP model used by the OM.

by $p(s'|s, a)$ the transition probability from state s to state s' given action a. We readily obtain:

$$\begin{aligned} p(s'|s,a) &= P[s_{i+1} = (\mathbf{k}', \lambda') | s_i = (\mathbf{k}, \lambda), a_i = a] = \\ &= \begin{cases} P[\lambda_{i+1} = \lambda' | \lambda_i = \lambda] & \mathbf{k}' = \mathbf{k} \oplus a \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

It is easy to realize that the system dynamics comprise a stochastic component due to the tuple rate variation, which we assume exogenous, captured by the transition probabilities $P[\lambda_{i+1} = \lambda' | \lambda_i = \lambda]$, and a deterministic component due to the fact that, given action a, the new deployment configuration $\mathbf{k}'$ is $\mathbf{k} \oplus a$.

Figure 6 provides a simple example of the resulting state-action space, based on the example VM-based infrastructure presented in the previous section, where three types of VMs are available and $T_{res} = \{\texttt{vm.small}, \texttt{vm.medium}, \texttt{vm.large}\}$. In the example, the current state of the operator is $s = ([2, 0, 1], \lambda)$, that is two small instances and one large instance are in use, and the input rate level is λ. Possible actions from the set $\mathcal{A}(s)$ are $(-1, L)$ and $(+1, M)$, which, respectively, correspond to removing a large instance, and adding a medium instance.

A cost is associated with the execution of actions and state transitions. The function $c(s, a, s')$ captures the cost of operating the system in state s', after carrying out action a in state s. The function accounts for the three different cost metrics we have been considering so far: resource usage, performance violation, adaptation overhead.

The *resources cost* $\mathbf{c_{res}}$ is the cost paid for using the computing resources on which the operator is deployed throughout the next interval. Formally, we have:

$$c_{res}(s, a, s') = \sum_{\tau \in T_{res}} k'_\tau c_\tau$$

where c_τ is the cost paid for deploying an operator replica on a resource of type τ for the next time interval.

The *reconfiguration penalty* $\mathbf{c_{rcf}}$ captures the impact of deployment adaptation (i.e., state movement, application downtime). We model such a penalty with a constant value, because the large reconfiguration overhead imposed by most DSP frameworks dominates the variable impact of state movements for many applications.[7] Therefore, we have $c_{rcf}(a) = \mathbb{1}_{\{a \neq \omega\}}$, where $\mathbb{1}_{\{\cdot\}}$ is the indicator function.

The *performance requirement violation* cost $\mathbf{c_{perf}}$ captures the penalty incurred whenever the operator violates its performance requirement in the next interval. Specifically, as we consider response

[7] More accurate and detailed models of the adaptation overhead can be formulated, as we did, e.g., in [9], and integrated in our auto-scaling model. However, as we have observed in practice that the reconfiguration time is often dominated by the overhead imposed by DSP frameworks to pause-and-resume the whole application upon parallelism changes, we decided to adopt a simple adaptation penalty model and rely on a constant penalty for scaling actions.

time requirements assigned by the AM to each operator, we have $c_{perf}(s,a,s') = \mathbb{1}_{\{\tilde{R}(s')>\tilde{R}_{max}\}}$, where $\tilde{R}(s)$ is the average[8] operator response time in state s and $\tilde{R}_{max}$ is the per-operator latency bound. Note that the same approach may be used in presence of different performance requirements (e.g., throughput), updating the penalty evaluation as needed.

We combine the different costs into a single cost function using the *Simple Additive Weighting* (SAW) technique. According to SAW, we define the cost function $c(s,a,s')$ as the weighted sum of the normalized costs:

$$c(s,a,s') = w_{res}\frac{c_{res}(s,a,s')}{C_{max}} + w_{rcf}c_{rcf}(a) + w_{perf}c_{perf}(s,a,s') \tag{2}$$

where w_{res}, w_{rcf} and w_{perf} are non negative weights (with $w_{res} + w_{rcf} + w_{perf} = 1$), and C_{max} is a normalization parameter defined as $\max_{\tau \in T_{res}} c_\tau K^{max}$.

It is worth observing that we might drop the explicit adaptation cost term, as it could be indirectly captured by other terms (e.g., increased response time). Having an explicit adaptation cost has a twofold motivation. First, it makes the formulation more general, as it allows us to capture the impact of adaptation beyond response time degradation (e.g., with respect to network usage or application unavailability, due to state migrations possibly required by auto-scaling). A second advantage will be evident when presenting the learning algorithms in the following. Since the adaptation impact is known, it can be excluded from the dynamics that algorithms have to learn.

The ultimate goal of the OM, which corresponds to the *decision making agent* in the MDP parlance, is to minimize the total cost incurred by its actions over time. To this end, the OM seeks for a good *policy*, that is a function $\pi : \mathcal{S} \to \mathcal{A}$ which associates each state with an action to perform.

Formally, given a policy π, the *value function* $V^\pi(s)$ denotes the expected discounted cost paid over the infinite time horizon when following π from the initial state s, that is:

$$V^\pi(s) = E^\pi\left[\sum_{i=0}^{\infty} \gamma^i c(s_i, a_i, s_{i+1}) \middle| s_0 = s\right]$$

We are interested in determining the optimal policy π^* that minimizes the expected discounted cost. The optimal policy satisfies the *Bellman optimality equation* for every state s:

$$V^{\pi^*}(s) = \min_{a \in \mathcal{A}(s)} \left\{ \sum_{s' \in \mathcal{S}} p(s'|s,a) \left[c(s,a,s') + \gamma V^{\pi^*}(s') \right] \right\} \tag{3}$$

where the expected future costs starting from state s' are recursively expressed by means of $V^{\pi*}(s')$. It is also convenient to define the *action-value* function $Q^\pi : \mathcal{S} \times A \to \mathbb{R}$ (also known as "Q function"), which is the expected infinite-horizon discounted cost incurred by taking action a in state s and then following the policy π:

$$Q^\pi(s,a) = \sum_{s' \in \mathcal{S}} p(s'|s,a) \left[c(s,a,s') + \gamma V^\pi(s') \right] \tag{4}$$

It is easy to realize that the value functions V and Q are closely related in that $V^\pi(s) = \min_a Q^\pi(s,a)$. More importantly, the knowledge of Q provides the associated policy π, as $\pi(s) = \arg\min_a Q(s,a)$, for every state s.

[8]While we consider average response time in this work, the approach can be readily updated to work with different requirements, e.g., in terms of percentiles. Indeed, we only require the ability to evaluate whether the requirement is met in each time slot.

Algorithm 1 Auto-Scaling Policy based on Q-learning (QL)

1: $i \rightarrow 0$
2: Initialize the Q function
3: **loop**
4: choose a scaling action a_i based on current estimates of Q, e.g., ϵ-greedy
5: observe the next state s_{i+1} and the incurred cost c_i
6: $Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \left[c_i + \gamma \min_{a' \in \mathcal{A}} Q(s_{i+1}, a') - Q(s_i, a_i)\right]$
7: $i \leftarrow i + 1$
8: **end loop**

4.2 Reinforcement Learning: Background on Tabular and Deep Q-learning

Given a MDP instance whose parameters, including the cost function and the transition probabilities, are known, *planning* algorithms can compute the optimal policy (e.g., through *Value Iteration*). However, in many scenarios, information about the system dynamics are not available. A natural choice in these scenarios is to use *reinforcement learning* (RL) techniques which learn the optimal policy through experience and direct interaction with the system [62]. The core idea behind RL algorithms is to improve the policy over time based on feedback, that is the incurred costs, through direct interaction with the system.[9] This high-level idea has led to the development of many different RL approaches and algorithms. Hereafter, we will focus on value function-based algorithms, whose general scheme is shown in Algorithm 1: value function entries (e.g., Q entries) are first initialized (setting all to 0 will often suffice) (line 1); then, by direct interaction with the system, the RL agent (i.e., the OM) at each step t chooses an action a_t based on current estimates of Q (line 3), observes the incurred cost c_t and the next state s_{t+1} (line 4), and updates Q based on what it just experienced, that is the tuple (s_i, a_i, c_i, s_{i+1}) (line 5). Existing algorithms mainly differ in the way actions are selected and Q is updated at each step.

As regards the choice of actions to be performed while learning, RL algorithms often face the so-called *exploration-exploitation* dilemma. To expand their knowledge, agents need to *explore* new states and actions. At the same time, in order to minimize the incurred cost, agents would need to *exploit* actions that have proven to be good in the past. Therefore, most RL algorithms seek for a suitable balance between exploration and exploitation.

4.2.1 Q-learning. The simplest RL algorithm is the well-known *Q-learning* algorithm, which is a *model-free* algorithm (i.e., it does not require any knowledge of the system dynamics). Q-learning [67] (denoted as **QL** for short in figures and tables), is a learning algorithm that estimates the optimal action value function Q^* by its sample averages. At any decision step (line 4 of Algorithm 1), Q-learning either: 1) *exploits* its current knowledge and *greedily* selects the action expected to minimize future costs, i.e., $a_i = \text{argmin}_{a' \in A(s_i)} Q(s_i, a')$; alternatively, 2) it *explores* the environment by choosing a random action to improve its knowledge of the system. A common approach to mix exploration and exploitation is the so-called ϵ-*greedy* action selection policy, which chooses either a random action with probability ϵ or the greedy action with probability $1 - \epsilon$.

The algorithm performs simple one-step updates at the end of each time slot (line 6), as follows:

$$Q(s_i, a_i) \leftarrow (1 - \alpha) Q(s_i, a_i) + \alpha \left[c_i + \gamma \min_{a' \in \mathcal{A}(s_{i+1})} Q(s_{i+1}, a') \right] \tag{5}$$

[9] RL terminology usually considers agents aiming to maximize action *rewards*. To be consistent with our MDP formulation, we instead consider an equivalent cost minimization problem.

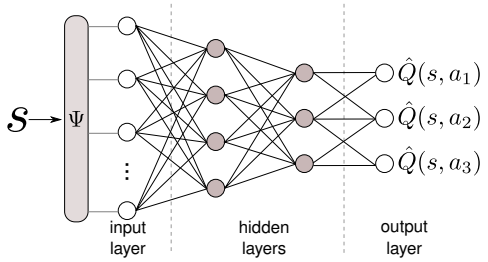

Fig. 7. Example of neural network used by deep Q-learning.

where $\alpha \in (0, 1)$ is the *learning rate* parameter. It is worth remarking the simplicity of this rule, which involves the updating of a single entry, $Q(s_i, a_i)$, with the latest observed cost c_i plus the expected cost of following the greedy policy onward, that is $\min_{a' \in \mathcal{A}} Q(s_{i+1}, a')$.

4.2.2 Deep Q-learning. Algorithms that rely on tabular representations of the value function as Q-learning, although effective in many cases, may suffer from limited scalability as the size of the state space grows. To overcome these issues, *Function Approximation* (FA) techniques have been largely adopted in the literature. When using FA, the Q table is replaced by a parametric function $\hat{Q}(s, a, \theta)$ that is used to approximate $Q(s, a)$. By so doing, rather than storing and updating every single entry $Q(s, a)$, we only have to store and update the - usually much smaller - parameters vector θ. This approach can dramatically reduce the computational and storage demand of learning algorithms.

The quality of the computed policy clearly depends on how well $\hat{Q}$ approximates Q. For this purpose, a large number of approximation models may be used, ranging from linear models to (deep) neural networks. Artificial neural networks and, in particular, *deep* neural networks (DNNs) have gained enormous popularity in the last decade, thanks to the outstanding advancements they enabled in various areas of research. The field of RL has not ignored this trend, with the development of deep RL methods (see, e.g., *deep Q Networks* [43]), where DNNs support nonlinear FA and enable learning over continuous or very large state spaces.

The core idea behind deep RL is using an approximate value-function $\hat{Q}(s, a; \theta)$ obtained as the output of a DNN, where θ denotes the weights of the network. For this purpose, the network has to be trained using samples of Q collected through experience, using, e.g., stochastic gradient descent (SGD). It is worth noting that such training often requires a lot of data and time and, thus, it is performed off-line by means, e.g., of system simulators. In our setting, instead, we aim to perform the training on-line, assuming that new applications can be submitted to the system at any time, preventing us from computing an auto-scaling policy in advance.

An example architecture of a DNN used within RL is given in Figure 7. Given a state s, the input of the network consists of a vector $\psi(s)$, which is a suitable encoding of s (we discuss possible state encoding strategies in Appendix B.3). One or more hidden layers contribute to the computation of the network output, which comprises one element for each available action a, yielding the approximate value $\hat{Q}(s, a)$.

In principle, DNN-based FA can be integrated in Q-learning by simply replacing the traditional Q update rule in Equation (5) with an iteration of network training that uses the newly collected experience. However, to reduce the risk of overfitting due to correlation between consecutive observations, DRL approaches usually rely on *experience replay*. The agent does not immediately use its experience for learning and instead stores the associated information (i.e., state transitions, performed actions and paid costs) in an *experience buffer*. Periodically, when a learning step has to

Algorithm 2 Auto-Scaling Policy based on Deep Q-learning (DQL)

1: Initialize θ and the experience buffer $\mathcal{B}$
2: **loop**
3: pick scaling action a_i (e.g., with ϵ-greedy)
4: observe the next state s_{i+1} and the incurred cost c_i
5: add the experience tuple $\langle s_i, a_i, s_{i+1}, c_i \rangle$ to $\mathcal{B}$
6: **if** $shouldUpdate(i)$ **then**
7: sample batch of $\langle s_j, a_j, s_{j+1}, c_j \rangle$ tuples from $\mathcal{B}$
8: $y_j \leftarrow c_j + \gamma \min_{a' \in \mathcal{A}(s_{i+1})} \hat{Q}(s_{i+1}, a'; \theta), \forall j$
9: train the network using samples (s_j, y_j)
10: **end if**
11: $i \leftarrow i + 1$
12: **end loop**

be performed, a mini-batch of experience items is randomly drawn from the buffer. In addition to reducing correlation between samples, this approach promotes reuse of past observations to avoid forgetting.

The resulting deep Q-learning (**DQL**) algorithm is reported in Algorithm 2. As for the standard Q-learning algorithm, the agent picks an action at every time step (line 3), observing a transition and paying a cost (line 4). As mentioned, the latest observation is stored into an experience buffer $\mathcal{B}$ (line 5), which is usually a fixed-capacity FIFO buffer. When a learning iteration has to be done (e.g., periodically every k steps), a batch of experience tuples are drawn from $\mathcal{B}$ (line 7). A training iteration is then performed (using, e.g., SGD) against a batch of samples y_j obtained through the available experience (lines 8-9).

4.3 Operator Autoscaling: Learning Policies with Post Decision States

The aforementioned learning techniques, as Q-learning and DNN, are based on the underlying assumption that we have no knowledge about the system and everything must be learnt through direct interaction with the system itself. This is clearly a feature when indeed no knowledge about the system dynamics is available, but it can become a liability otherwise, since - by using Q-learning and DNN as previously described - we are basically using the learning algorithm to also learn what we already know, which in turn results in slower converging algorithms.

In our domain as well as in many others, the dynamics of the system are not completely unpredictable. In particular, in our setting, the impact of actions on the system state is known and deterministic (e.g., scale-out actions increase the operator parallelism by one level). Therefore, it seems desirable to provide the learning agent with such basic knowledge, rather than letting it discover it by itself, in order to provide faster convergence and, as a result, better scaling algorithm performance.

In this paper, in order to integrate the partial knowledge we have in the learning algorithm, we use the concept of *post-decision state* (PDS) (also known as *afterstate*), exploiting the generalized definition given in [39]. A PDS refers to the state of the system *after* the known dynamics takes place, but *before* the unknown dynamics does. Formally, we denote a PDS as $\tilde{s} \in \mathcal{S}$. At any time i, we logically split the state transition $s_i \rightarrow s_{i+1}$ into two distinct transitions: $s_i \rightarrow \tilde{s}_i$ and $\tilde{s}_i \rightarrow s_{i+1}$. The first transition $s_i \rightarrow \tilde{s}_i$ represents the impact of the decision a_i and the known dynamics, hence the name *post-decision state*; the second transition $\tilde{s}_i \rightarrow s_{i+1}$ represents the impact of the

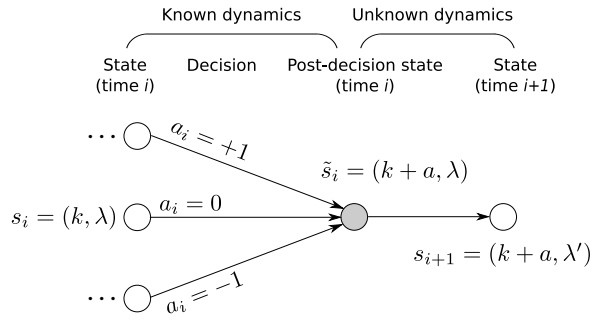

Fig. 8. Relationships between current state, actions, PDS, and next state, assuming $N_{res} = 1$ and, thus, only three actions exist in the model. (Adapted from the diagram reported in [39].)

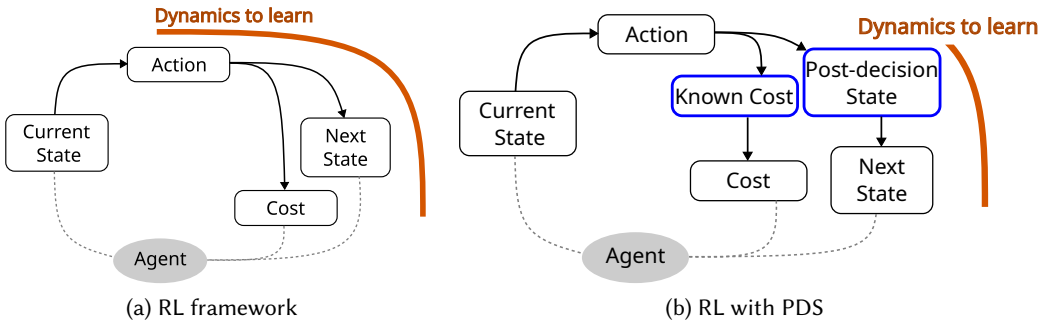

Fig. 9. The post-decision state (PDS) abstraction allows us to separate known and unknown dynamics. By doing so, we can design enhanced RL algorithms where we only have to learn the unknown components, possibly achieving faster convergence.

unknown dynamics, which brings the system from the *post-decision state* $\tilde{s}_i$ to the next state s_{i+1} (see, Figure 8).

In our setting, given the current state $s_i = (k_i, \lambda_i)$ and the selected action a_i, we have:

$$\tilde{s}_i = (\mathbf{k}_i \oplus a_i, \lambda_i) = (\mathbf{k}_{i+1}, \lambda_i) \tag{6}$$

$$s_{i+1} = (\mathbf{k}_{i+1}, \lambda_{i+1}) \tag{7}$$

where $\mathbf{k}_i \oplus a_i$ indicates the deployment reconfiguration dictated by action a_i. The *post-decision state* $\tilde{s}_i$ reflects the consequences of action a_i (and, in general, any other known dynamics), whilst the next state s_{i+1} incorporates the unknown system dynamics (i.e., the input rate variation).

Correspondingly to the splitting of the state transition, we have to separate the cost function (2) in known and unknown components as follows:

$$c(s, a, s') = c_k(s, a) + c_u(\tilde{s}, s') \tag{8}$$

where $c_k(s, a)$ accounts for the *known* deterministic cost associated to the scaling action a in state s, and $c_u(\tilde{s}, s')$ incorporates the *unknown* unpredictable impact of the rate variation on the system performance when transitioning from the PDS $\tilde{s}$ to the new state s'. Comparing the expression above to (2), we have $c_k(s, a) = w_r \frac{c_{res}(s,a,s')}{C_{max}} + w_a c_{rcf}(a)$ which accounts for the costs due to

adaptation and resource usage, and the unknown cost $c_u(\tilde{s}, s') = w_s c_{perf}(s, a, s')$ which accounts for the performance penalty, which depends on the input data rate in the next time state.

Separating known and unknown dynamics through the PDS abstraction allows us to design an enhanced version of the Q-learning and DNN algorithms. Intuitively, once we have separated the system dynamics in known and unknown components, we can restrict ourselves to only learning the unknown part, as illustrated in Figure 9, possibly leading to a faster learning process. Following the approach presented in [39], we introduce the PDS value function $\tilde{V}(\tilde{s})$, which represents the expected discounted cost paid when starting from a PDS $\tilde{s}$ [10].

The resulting learning algorithm (**QL+PDS, for short**) updates at each step the Q-function along with the post decision state value function $\tilde{V}$. More precisely, we replace the Q-learning update step (5), with the following two steps:

$$Q(s_{i+1}, a) \leftarrow c_k(s_{i+1}, a) + \tilde{V}(\tilde{s}_{i+1}) \; \forall a \in \mathcal{A} \tag{9}$$

$$\tilde{V}(\tilde{s}_i) \leftarrow (1-\alpha)\tilde{V}(\tilde{s}_i) + \alpha\left[c_{u,i} + \gamma \min_{a' \in \mathcal{A}(s_{i+1})} Q(s_{i+1}, a')\right] \tag{10}$$

Comparing them to (5), we can observe that in QL+PDS: (i) we first update $Q(s_{i+1}, a)$ but, differently from (5), we only take into account the known dynamics, that is the known cost $c_k(s_{i+1}, a)$ and the PDS value $\tilde{V}(\tilde{s}_{i+1})$; (ii) then we update $\tilde{V}(\tilde{s}_i)$ by taking into account the unknown dynamics, that is the incurred cost $c_{u,i}$ and the input rate variation from λ_i to λ_{i+1}. As shown in [39], the update equations (9)-(10) ensure convergence as $i \to \infty$.

As we will demonstrate in Sec. 6, separating known and unknown parts of the model results in significant boosts to the learning speed. In this context, a legitimate question is if we can do more: intuitively, the less we need to learn, the faster the learning speed. To this end, we now also consider the idea of estimating the unknown part of our model, that is the unknown cost $c_u(\tilde{s}, s')$: if we manage to estimate the performance violation cost over the next interval, $\hat{c}_u(\tilde{s})$, which depends on the workload dynamics and the operator response time model, we reduce the unknown part of our model to the performance violation prediction error.

The resulting enhanced PDS-based algorithm (**QL+PDS$^+$**, for short) replaces (9)-(10) with

$$Q_i(s_{i+1}, a) \leftarrow c_k(s_{i+1}, a) + \hat{c}_u(\tilde{s}_{i+1}) + \tilde{V}_i(\tilde{s}_{i+1}) \; \forall a \in \mathcal{A} \tag{11}$$

$$\tilde{V}_{i+1}(\tilde{s}_i) \leftarrow (1-\alpha)\tilde{V}_i(\tilde{s}_i) + \alpha\left[\delta_{c,i} + \gamma \min_{a' \in \mathcal{A}(s_{i+1})} Q_i(s_{i+1}, a')\right] \tag{12}$$

where $\delta_{c,i} = \hat{c}_{u,i} - \hat{c}_u(\tilde{s}_i)$ represents the performance violation estimation error. Comparing the updating rules, it is worth observing that QL+PDS$^+$ differs from QL+PDS in that: (i) in (11) we replace the known cost $c_k(s_{i+1}, a)$ with the sum of the known cost and the estimate of the unknown cost $c_k(s_{i+1}, a) + \hat{c}_u(\tilde{s}_{i+1})$; and (ii), in (12) we replace the unknown cost $c_{u,i}$ with the estimation error $\delta_{c,i}$.

4.3.1 Deep Q-learning with Post-Decision States. In order to improve the performance and responsiveness of the learning algorithm we also propose a PDS-based DQL algorithm (**DQL-PDS**, for short) by analogous reasoning.

In QL-PDS we introduced the PDS value function $\hat{V}(\tilde{s})$ to capture the unknown system dynamics that we must learn. Here, we approximate $\tilde{V}$ by $\hat{V}(\tilde{s})$ using the DNN and to update over time, whilst Q can be readily computed from $\tilde{V}$ and the known dynamics. To do so, we need a different network architecture, shown in Figure 10. Network input still consists of a single state, which is now a PDS $\tilde{s}$. The main difference with respect to the DNN presented above regards the output layer, which is now composed of a single neuron, associated with the value $\hat{V}(\tilde{s})$.

[10] Observe that we do not have a corresponding action-value function associated to $\tilde{s}$ since there are no actions in post-decision states.

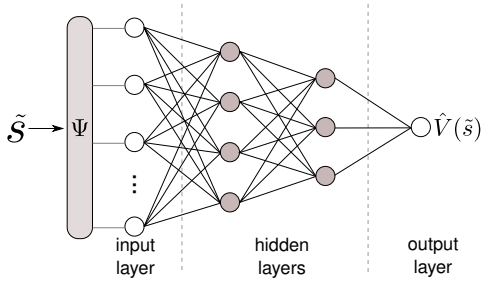

Fig. 10. Example of neural network used by deep Q-learning to exploit the PDS concept.

The resulting algorithm is very similar to the standard DQL from Algorithm 2, with the notable exception that only the unknown cost terms are stored in the experience buffer (lines 4-5) and used to compute the target values for training (line 8). Furthermore, as we did for the tabular case, we can also define an enhanced version of the algorithm (denoted as **DQL-PDS$^+$**), where we bootstrap the learning process with estimates of the unknown costs. In this case, we observe the difference between the expected and the observed costs on-line and use it for learning (see Sec. 4.3).

5 APPLICATION MANAGER POLICY

The Application Manager (AM) is the top-layer entity of our hierarchical auto-scaling framework, as illustrated in Figure 4. The AM is responsible for steering application adaptation by supervising the lower-layer decentralized OMs.

Instead of directly making auto-scaling decisions, the AM aims to optimize application QoS by configuring OM local objectives. In particular, the role of the AM is illustrated in Figure 11. Given a set of user QoS requirements, the AM aims to minimize the application operating costs while guaranteeing the desired service level (e.g., in terms of processing latency). To this end, the AM runs its own control loop, which comprises three main components: Application Monitor, QoS Analyzer and Optimizer.

The *Application Monitor* collects key performance metrics from the lower-layer OMs about the running application (e.g., data arrival rate, operator response times). This information is passed to the *QoS Analyzer* to build a model of the system, which is used to evaluate the achieved QoS under various configurations. To this end, we equip the QoS Analyzer with simulation-based models of the system, which are used asynchronously with respect to application execution. As further explained below, these models are exploited by an *Optimizer* to adjust OM configuration.

We formalize the problem tackled by the AM in the next subsection, before presenting the resolution strategy we propose.

5.1 Problem Formulation

The goal of the AM is to minimize the monetary application operating costs due to acquisition and usage of computing resources while satisfying the applications QoS requirements. Without lack of generality, we assume QoS requirements to be specified in terms of three target values, as follows:

- R_π^{max}, the maximum response time for each source-to-sink path π in the application graph; source-to-sink paths may correspond to different queries on the input data streams and, thus, be associated with different latency requirements.
- η_V, the maximum fraction of time when the response time requirement is allowed to be violated (e.g., no more than 5% violations).

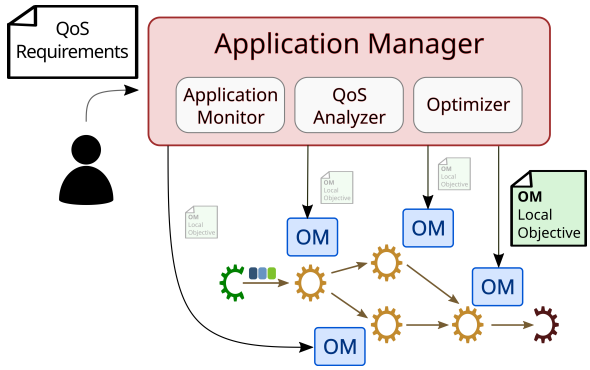

Fig. 11. Application Manager architecture and its interaction with the Operator Managers.

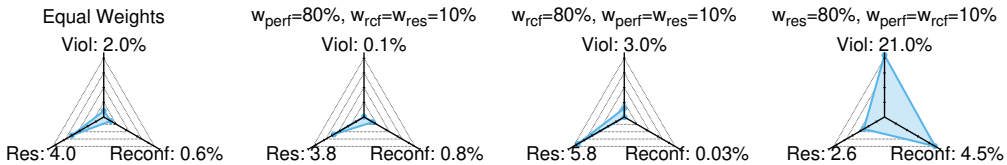

Fig. 12. Example results showing the impact of changing the weights used in the OM cost function, in terms of resource usage, as well as percentage of performance violations and parallelism reconfigurations.

- η_R, the maximum fraction of time when the application deployment can be reconfigured, possibly incurring overheads (e.g., no more than 10% reconfigurations).

The AM aims to minimize operating costs and meet the requirements introduced above, while the lower-layer OMs independently optimize a different local cost function (i.e., Equation 2). Therefore, the key knobs available to the AM correspond to the parameters of the local OM objectives, which can be tuned so as to optimize the application-level QoS. Specifically, the OM cost function can be adjusted by tweaking the objective weights $\mathbf{w} = (w_{perf}, w_{rcf}, w_{res})$ and the operator response time bounds $R_u^{max}, \forall u \in V_{dsp}$.

As regards the response time requirement, the AM is only given the values R_π^{max}, which specify the maximum response time at level of end-to-end paths in the application graph. From these values, the AM has to derive the response time requirement for each operator, so as to satisfy the application-level requirement through a set of local requirements. As operators have different resource demands, it is clear that local bounds should be set accordingly to optimize resource usage (e.g., highly loaded operators would benefit the most from larger response time "budgets").

As regards the weights $\mathbf{w}$, they express the relative importance of the different objective terms at the operator level. Figure 12 reports an example of the impact of setting different objective weights to an OM, in terms of performance violations, resource usage and reconfigurations. It can be seen that tuning the weights leads to a different trade-off between the three key metrics. Such tuning should be abstracted away from users concerned with higher-level QoS requirements and should be delegated to the AM. Therefore, the AM has to suitably tune them according to the user-given target performance violation and reconfiguration requirements (i.e., η_V and η_R).

The optimization problem tackled by the AM can be formulated as follows:

$$\min_{\mathbf{w}, R_u^{max}} \quad \sum_{t=0}^{T} C_{res}(t) \tag{13}$$

$$\text{s.t.} \quad \frac{1}{T} \sum_{t=0}^{T} Viol(t) \leq \eta_V \tag{14}$$

$$\frac{1}{T} \sum_{t=0}^{T} Rcf(t) \leq \eta_R \tag{15}$$

$$w_{perf}, w_{rcf}, w_{res} \in (0, 1) \tag{16}$$

$$R_u^{max} \geq 0, \forall u \in V_{dsp} \tag{17}$$

where $C_{res}(t)$ denotes the application resource usage cost at time t, which accounts for the resource usage of each operator in the application:

$$C_{res}(t) = \sum_{u \in V_{dsp}} c_{res}^u(t)$$

and $Viol(t)$ and $Rcf(t)$ are binary functions indicating whether, respectively, a performance violation or a reconfiguration happens at time step t.

Unfortunately, we cannot tackle this problem by standard methods as we cannot rely on an analytical expression of the objective function, which depends on the policies adopted at run-time by the OMs, not known *a priori*, and their interaction.

5.2 Problem Resolution via Bayesian Optimization

In order to resolve the problem presented above, we resort to *black-box* optimization methods. These techniques aim to optimize the value of objective functions that are not analytically available (and neither are their derivatives), but can be evaluated on given candidate solutions. Among this group of methods, we turn our attention to *Bayesian optimization* (BO) [56], which searches for the global optimum of an objective f given a limited number of steps in which the function can be evaluated on arbitrary points. To do so, BO builds a probabilistic *surrogate model* of f, which captures prior belief about the function and guides the choice of the most "promising" points to sample.

5.2.1 Basics of Bayesian Optimization. Algorithm 3 shows the pseudocode for a generic BO method, which works as follows. To initialize the surrogate model of f, we first evaluate the objective function at n_0 arbitrary points, which can be chosen, e.g., randomly (lines 1–3). Then, we iteratively sample new points until allowed (i.e., until the desired number of iterations N_{BO} is reached or, alternatively, the available time budget expires). At each iteration, the point to sample $\bar{\mathbf{x}}$ is chosen so as to maximize a so-called *acquisition function* (line 6). The acquisition function relies on the surrogate model and usually balances exploitation (i.e., sampling where the surrogate model predicts a high value) and exploration (i.e., sampling points where uncertainty is high). After sampling $f(\bar{\mathbf{x}})$, the surrogate model is updated accordingly (lines 7–8). At the end, BO yields the "best" point $\mathbf{x}$, which – depending on the requested behavior – is either the point evaluated with largest $f(\mathbf{x})$, or the point that maximizes the surrogate model.

Figure 13 illustrates the main concepts involved in BO. A popular choice for the surrogate model in this context is represented by Gaussian Processes [48], a statistical tool for stochastic modeling of functions. As regards the acquisition function, several options exist, including, e.g., expected improvement, or upper confidence bounds.

Algorithm 3 Basic Bayesian optimization approach.

Input: Black-box objective f, domain D, allowed iterations N_{BO}

choose arbitrarily $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_{n_0}} \in D$, $n_0 < N_{BO}$
evaluate $f(\mathbf{x_1}), f(\mathbf{x_2}), \ldots, f(\mathbf{x_{n_0}})$
build a surrogate model of f using the available samples
$n \leftarrow n_0$
while $n < N_{BO}$ **do**
    pick $\bar{\mathbf{x}} \in D$ that maximizes the acquisition function
    evaluate $f(\bar{\mathbf{x}})$
    update the surrogate model using the new sample
    $n \leftarrow n + 1$
end while
return $\mathbf{x}$ that either is the point evaluated with largest $f(\mathbf{x})$ or maximizes the surrogate model

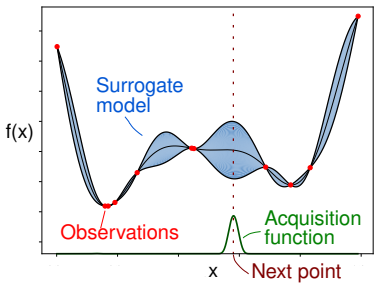

Fig. 13. Bayesian optimization searches for the global optimum of an objective $f(x)$ given a limited number of *observations* (i.e., function evaluations on arbitrary points). To do so, BO builds a probabilistic *surrogate model* of f, which captures prior belief about the function. An *acquisition function* uses this surrogate model to guide the choice of the most "promising" points to sample (e.g., those where uncertainty is maximum).

5.2.2 Using BO within the AM. We exploit BO to minimize the objective function (13), i.e., the resource usage cost, given the local objective weights $\mathbf{w}$ and the operator response time bounds. To do so, we must be able to evaluate (13) over an observation period $\bar{T}$ given a candidate solution. We fulfill this requirement equipping the AM with a system *simulator*, which can be used in parallel with the real system. In order to build such a simulation model, the AM needs to collect data about the application, so this method cannot be used to bootstrap the auto-scaling framework (for this purpose, it suffices to use the naive approach described above). In particular, the AM needs information about both the workload and the operators. For the workload, the AM needs to collect a sequence of input data rate values, which will be used for trace-driven simulation. As regards the operators, the AM needs to collect basic information to build an estimated performance model. Specifically, we monitor the average operator service time and its variance and use them to instantiate an M/G/1 queueing model of each operator instance.

To limit the number of variables involved in the optimization, we separately optimize the objective weights and the response time bounds, as described in the next two subsections. Then, we will present an iterative algorithm that wraps the two phases.

5.2.3 Latency apportioning. To determine the response time bound for each operator, we let BO search for the vector $\mathbf{x} = (x_1, x_2, \ldots, x_n), x_i \in (0, 1)$, such that $R_i^{max} = x_i R^{max}$ is the time apportioned

Algorithm 4 Iterative response time bounds and objective weights tuning.

Input: application G_{dsp}, QoS requirements $\zeta = \langle R^{max}, \eta_V, \eta_R \rangle$, max BO iterations N_{BO}
Output: obj. weights $\boldsymbol{w}$, operator max. response times $\boldsymbol{\rho}$

1: initialize $\mathbf{w}$ and $\boldsymbol{\rho}$ using the naive approach ▷ see Sec. 6.3
2: **repeat**
3: $\quad \boldsymbol{\rho}_0 \leftarrow \boldsymbol{\rho}$
4: $\quad \boldsymbol{\rho} \leftarrow \text{optimizeLatency}(G_{dsp}, \zeta, N_{BO}, \boldsymbol{\rho}_0, \boldsymbol{w})$
5: $\quad \boldsymbol{w}_0 \leftarrow \boldsymbol{w}$
6: $\quad \boldsymbol{w} \leftarrow \text{optimizeWeights}(G_{dsp}, \zeta, N_{BO}, \boldsymbol{w}_0, \boldsymbol{\rho})$
7: $\quad$ reduce N_{BO} if needed
8: **until** max iterations or no improvement

to the i-th operator.[11] Therefore, the number of variables equals the number of operators in the application. Since a limited number of objective evaluations are available to the algorithm, it is important to carefully select the points to sample. For this reason, we guide the search by introducing constraints, which are taken into consideration when maximizing the acquisition function. Specifically, we introduce two types of constraints. We enforce the variable domain:

$$0 < x_i < 1, \ \forall i \in V_{dsp}$$

Moreover, we require that the total latency allocation along any source-to-sink path roughly corresponds to R^{max}, with a margin $\epsilon > 0$:

$$1 - \epsilon \leq \sum_{i \in \pi} x_i \leq 1 + \epsilon, \ \forall \pi \in \Pi_{dsp}$$

5.2.4 Weights tuning. In this setting we only have three variables corresponding to the weights w_{res}, w_{rcf} and w_{perf}. As for the latency apportioning problem, we introduce constraints, starting with the variable domain enforcement:

$$0 < w_X < 1, \ \forall X \in \{res, rcf, perf\}$$

An additional constraint requires the sum of the weights to equal 1:

$$w_{res} + w_{rcf} + w_{perf} = 1$$

5.2.5 Iterative AM algorithm. The AM tunes both the response time bounds and the objective weights through an iterative procedure, where the two sets of parameters are alternatively updated through BO. The resulting Algorithm 4 works as follows. Both the weights and the response time bounds are first initialized using the naive approach described above (line 1). Then the algorithm iteratively exploits BO to optimize the bounds or the weights in an alternate fashion (lines 3–6). When response time bounds (or, respectively, the weights) are to be tuned, the weights (response time bounds) are kept fixed using the best solution found so far. In each optimization round, the BO is initialized with the current solution to speedup the process. The algorithm stops iterating when a stop condition is met (i.e., when the desired number of iterations is reached or no improvement has been provided by the latest iteration). We will show in the evaluation that in practice a couple of iterations are enough to significantly improve the objective value compared to a baseline configuration and the algorithm converges.

[11] To simplify the formulation, we assume that the maximum response time requirement is the same for all the source-to-sinks paths in G_{dsp} and it is equal to R^{max}. The algorithm can be readily generalized to deal with different requirements.

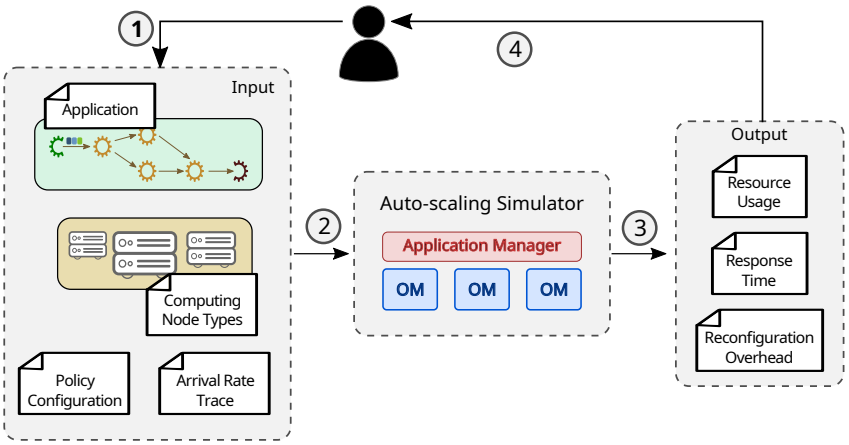

Fig. 14. Overview of the experimental workflow built around the auto-scaling simulator that implements all the policies presented.

6 EVALUATION

We numerically evaluate the presented policies, aiming to answer the following key questions:

- **Q1**: How do auto-scaling policies for DSP operators computed through RL compare to heuristic or model-based baselines?
- **Q2**: Which is the impact of function approximation (i.e., DNN-based RL) and injected model knowledge (i.e., post-decision states) on RL policies?
- **Q3**: How effective is the Application Manager in configuring independent RL-based Operator Managers to optimize application-level auto-scaling?

In this section, we first describe the software used for evaluation and then present the experiments performed to answer the questions listed above.

6.1 Evaluation Software

We implemented an evaluation software in Java[12]. Our software consists both of the implementation of all the auto-scaling policies presented in the paper and the required logic to simulate their execution against a given workload and infrastructure configuration.

A high-level view of the experimental workflow is provided in Figure 14. The input of each experiment comprises (i) the specification of a DSP application (i.e., the operators and their interconnections), including the service time distribution of each operator, (ii) a trace of data arrival rates, (iii) the specification of the computing infrastructure (i.e., types of available computing nodes, their speedup and monetary cost), and (iv) all the parameters related to the auto-scaling policy to simulate. Based on the provided input, our software initializes the simulated computing infrastructure (i.e., the set of available node types ans associated instances) and operator deployment (i.e., a single replica for each operator is created using the cheapest available node type). Then, a simulation is started where the following key tasks are performed periodically at the beginning of fixed-length time slots until the arrival trace is fully consumed:

(1) the mean application response time is evaluated given the current arrival rate and the operator response time models; if the response time exceeds the user-given maximum response time, a violation penalty is paid in the time slot;

[12]https://github.com/grussorusso/dsp-elasticity-sim

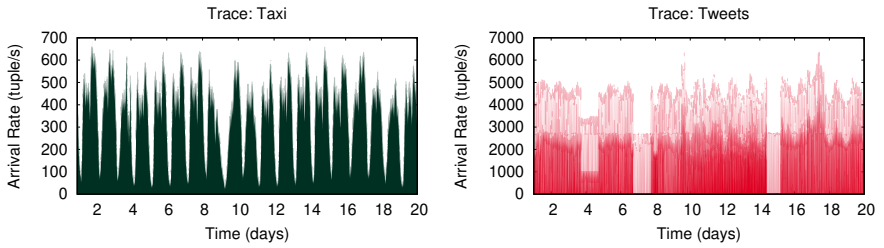

Fig. 15. Illustrative segments of the two arrival traces used in the experiments.

(2) the auto-scaling policy is provided with the current arrival rate data and computes a reconfiguration decision for each operator;
(3) operator deployment is updated based on auto-scaling decisions;
(4) resource usage cost and reconfiguration count are updated based on auto-scaling decisions, and the overall MDP cost function is computed according to (2);
(5) the MDP cost is communicated to the OMs, which can perform a learning iteration;
(6) the new application arrival rate is read from the trace.

The output of the simulation consists of statistics about the metrics of interest, mostly revolving around response time performance violations, cost due to resource usage and number of performed reconfigurations.

In principle, the logic of the simulation and, thus, the output of each experiment is deterministic. However, since RL-based policies might take randomized decisions (e.g., because of the ϵ-greedy action selection), we repeated all the experiments 10 times using different seeds for random number generation.

As regards the policies integrated within the evaluation software, we implemented all the RL-based algorithms described above as well as heuristic baselines that will be used in the following for comparison. DNN-based algorithms have been implemented exploiting *Deeplearning4j*[13], an open-source library for deep learning in Java. As regards the AM and the Bayesian optimization, we used *GPyOpt*[14], an open-source library for Gaussian Process-based optimization.

6.2 Operator Manager Policies (Q1, Q2)

In these experiments, we focus on the computation of auto-scaling policies at operator-level and, thus, evaluate single Operator Managers, according to the following setup.

Workloads. We consider data arrival rates retrieved from real traces at the beginning of each time interval. In particular, we use two traces. The first trace (**Taxi**, for short) is extracted from a data set of taxi trips in New York City throughout 1 year[15]. The resulting trace shows a daily pattern with average arrival rate equal to 330 tuple/s.[16] The second trace (**Tweets**) is extracted from a data set containing tweets related to the COVID-19 pandemic collected from Twitter in the first half of 2020.[17] This trace is, thus, shorter than the previous one and has a higher average rate equal to 2200 tuple/s. A sample of the two traces is shown in Figure 15.

[13]https://deeplearning4j.org/
[14]https://github.com/SheffieldML/GPyOpt
[15]https://chriswhong.com/open-data/foil_nyc_taxi/
[16]We scaled the event arrival rate of the original data set by 60x to achieve a more demanding workload.
[17]https://github.com/lopezbec/COVID19_Tweets_Dataset_2020

Table 1. Numerical speedup of nodes in the computing infrastructures A and B. Note: node type ordering in the table is such that, when $N_{res} = n$ node types are considered, only nodes with index $i \leq n$ are selected.

	Node Type Index									
	1	2	3	4	5	6	7	8	9	10
Infrastructure	**Speedup**									
A	1.0	0.7	1.3	0.9	1.7	0.8	1.8	2.0	1.65	1.5
B, B2	1.0	0.05	30.0	0.1	0.2	0.4	0.8	2.0	5.0	7.0

Operators. We consider a DSP operator which runs up to $K^{max} \in \{10, 20\}$ parallel replicas. Each replica is modeled as a G/G/1 queue, with mean service time μ and service time variance equal to $\frac{\mu^2}{2}$. When using the Taxi trace we set $\mu = 180$ tuple/s, whereas for the Tweets trace we set $\mu = 1600$ tuple/s.

Infrastructure. We define different computing infrastructures for the experiments (**Infr. A**, **B** and **B2**, for short), where heterogeneous nodes can be allocated.[18] As explained above, nodes are characterized by (i) a scalar speedup value, and (ii) a monetary price per unit of time. The speedup values for the infrastructures are reported in Tab. 1, where it is evident that nodes in infrastructure B are more heterogeneous compared to A. As regards the price, it is assumed to be equal to the speedup in Infrastructure A and B. Nodes in Infrastructure B2 have the same speedup of Infrastructure B, but their price does not grow linearly with the speedup. Specifically, in Infrastructure B2, the cost c_τ is computed as: $\sigma_\tau^{1.1}$, where σ_τ is the speedup of node type τ. In the experiments, we consider settings where a different number of node types are available, with $N_{res} \in \{3, 6, 10\}$.

Problem parameters. The OM objective terms in the cost function are weighted so as to give more importance to performance over resource usage and reconfiguration overhead, as follows $w_{perf} = 0.6$, $w_{res} = w_{rcf} = 0.2$. We will show how these weights can be automatically set by the AM in the following. The input rate is discretized using $N_\lambda = 30$ levels. The maximum response time R_{op}^{max} is set to 50 ms. The discount factor γ in the MDP is set to 0.99, which is a typical choice.

RL and DNN parameters. For tabular Q-learning, we use decaying learning rate and exploration probability – as suggested in [62] – and, specifically, we set $\alpha(t) = 0.98^{\lfloor \frac{t-1}{10} \rfloor}$, and $\epsilon(t) = 0.95^{t-1}$. For DNN-based algorithms, we use a fully connected network with 3 hidden layers, using ReLU as activation function. The number of neurons in each layer is set as $\frac{3}{4}$ of its inputs. We use an experience buffer with capacity 10,000. Every 5 time steps, we randomly draw a batch of 32 items from the buffer and perform a training iteration using SGD.

For QL-PDS$^+$and DQL-PDS$^+$, as they need to estimate the immediate action costs, we let them use an approximate system model, where (i) the operator service process is assumed to follow an exponential distribution and its rate is randomly under-estimated or over-estimated with a relative error comprised between 5% and 10% with respect to the true value; (ii) the computing node speedups are estimated with a uniform error not larger than 20%; (iii) the input rate is assumed to remain unchanged in the next time step.

6.2.1 Comparison against Baseline Policies (Q1). We compare our solution against baseline approaches from the literature:

- **Threshold-based (Thr) policies** increase operator parallelism whenever the average resource utilization U exceeds a given threshold $\bar{U}$. Conversely, if $n > 1$ instances are active

[18]We assume that the resource capacity in the infrastructure – as usual in a Cloud scenario – is much larger than the application demand and, thus, nodes can always be provisioned as requested.

and the utilization without one of them would be sufficiently smaller than the threshold (i.e., $\frac{n}{n-1}U < \beta\bar{U}$), a scale-in decision is made. This approach has been frequently adopted in the literature (e.g., [11, 16, 20]), not only for DSP auto-scaling [37]. In the experiments, we set $\beta = 0.75$ and $\bar{U} = 0.7$, after preliminary experiments.

- **DRS** (Fu et al. [19]): a queueing-based approach to scale DSP applications with latency constraints, based on the theory of Jackson open queueing networks.
- Lohrmann et al. [35] (**Lohrmann**, for short): a queueing-based approach to auto-scale DSP applications with latency constraints, which exploits Kingman's approximation for GI/GI/1 queues.
- Heinze et al. [25] (**Heinze**, for short): a model-free RL-based solution to auto-scale DSP systems. The goal of the agent is to keep the system resource utilization as close as possible to a target value, avoiding both over- and under-utilization. While the original implementation in [25] performs scaling operations at level of computing nodes, we implement it to scale operator parallel replicas, enabling a comparison with our solution.

Note that all these policies have been usually adopted in presence of homogeneous computing nodes. To cope with resource heterogeneity, we combine them with simple node selection strategies. The first one (denoted with suffix **-C**) always picks the cheapest available node. The second one (denoted with **-S**) instead picks the node with maximum speedup. The third one (denoted with **-0**) only considers the first type of node as listed in Table 1.

As regards the model-based policies, i.e., Lohrmann and DRS, we consider their "best case" and assume that they have exact knowledge of the operator service time process and the speedup provided by each computing node. While in practice they would hardily have such information, we will show that they struggle with heterogeneity even in this ideal scenario.

Figure 16 shows the average operating cost[19] achieved by the baseline policies and our solution, represented by QL-PDS. In Infrastructure A, which shows the least heterogeneity degree, all the considered baseline policies and RL perform well, with the only exceptions of Heinze-C and Lohrmann-C showing slight higher costs due to a suboptimal choice of nodes.

However, the situation changes significantly when looking at Infrastructure B, where more heterogeneous nodes are available. In this setting, our QL-PDS policy achieves a significantly lower average cost. The model-based policies by Heinze and Lohrmann do not make good use of the available nodes and incur higher costs. The threshold-based policy shows acceptable performance when always picking the most powerful nodes (i.e., Thr-S), but completely fails to satisfy the response time requirement when using the cheapest nodes (i.e., Thr-C). In Infrastructure B2, where a nonlinear cost-speedup relationship is used, things are even worse for baseline policies, which fail to optimize the trade-off between resource capacity and price. In this third and last setting, the RL policy is the only one delivering acceptable results, when looking at configurations with more than a single type of node.

6.2.2 Comparison of RL algorithms (Q2). We now extend the comparison to all the RL algorithms presented above, under a variety of settings. In addition to the algorithms presented above, we include in the comparison the solution we have previously presented in [54] (denoted as QL+FA), where linear function approximation is used in conjunction with Q-learning.

Complete tabular reporting of this comparison can be found in Appendix B. The results associated with a relevant subset of the configurations (i.e., those where $K^{max} = 20$) are illustrated in Figure 17. Note that, although the threshold-based policies are included in this comparison, the corresponding

[19]In these experiments, where we focus on the OM policy, the average cost refers to the average MDP cost incurred by the agent, i.e., computed according the cost function (2).

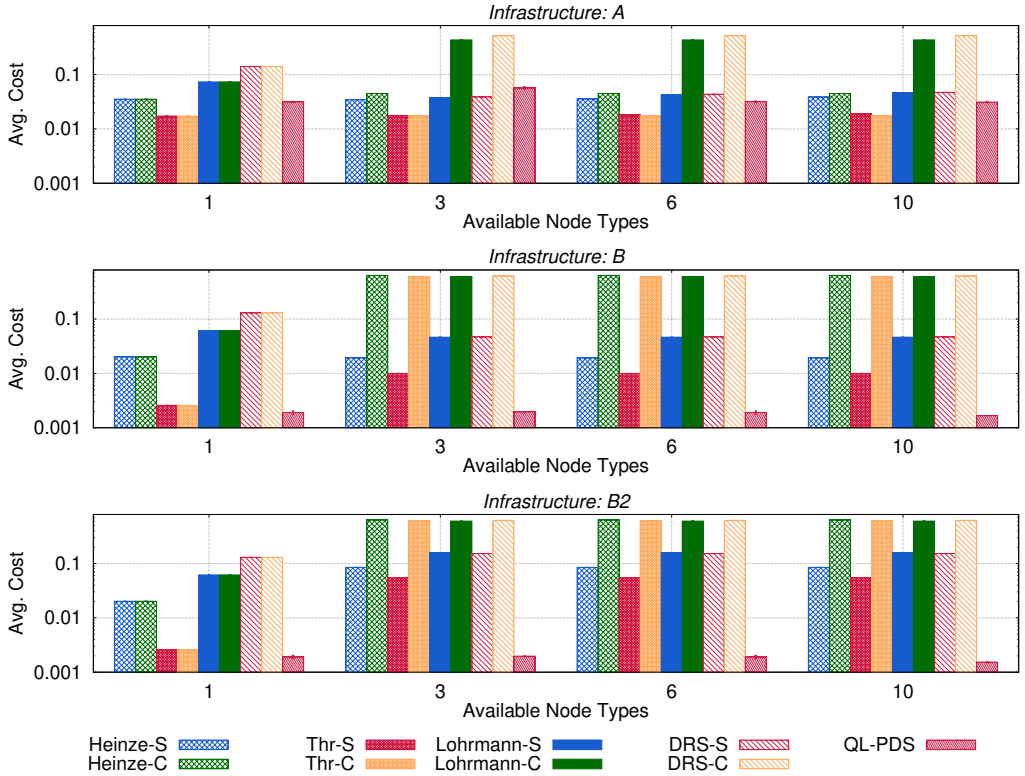

Fig. 16. Comparison of state-of-the-art and RL policies with different computing infrastructures.

results are not shown in the figure and we will not further discuss them, since they basically confirm what we observed in the previous subsection. The related results can be found in Appendix B.

Looking at the tabular RL algorithms, we first note that plain Q-learning has poor behavior in all the considered settings, especially as regards the large number of performed reconfigurations due to the need of exploration. The PDS-based Q-learning clearly outperforms the base version when considering infrastructure A, managing to exploit the knowledge provided to the agent to simplify learning. In this setting, QL+PDS keeps performance violations below 0.5%, reconfiguring the operator less than 1% of the time. Unfortunately, QL+PDS does not work well in infrastructure B, where the difference across node types is more significant. We observed that the OM fails to pick suitable types of nodes at the beginning and keeps them for most the experiment while it learns which deployment configurations are good. As such, it performs even worse than QL, which avoids this issue because of exploratory actions. This issue is also fixed by QL-PDS$^+$, which leverages reasonable cost estimates to quickly rank node types. It consistently shows very good performance in both the considered infrastructures and under both the workloads, emerging as the best auto-scaling solution.

The only drawback of QL-PDS$^+$, as well as the other tabular methods, consists in the memory requirement to store the value function. As shown in Figure 18, the memory required by tabular methods grows significantly as we consider more types of computing nodes in the infrastructure. While not challenging on its own (e.g., tens of megabytes with 10 types of nodes), such memory requirement may be undesirable in presence of many applications and, thus, many operators to

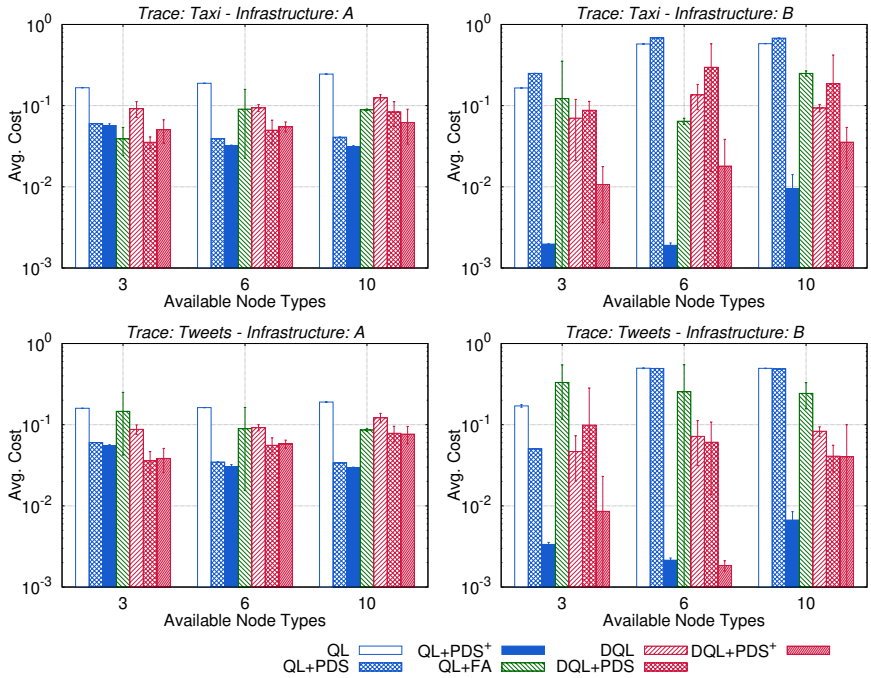

Fig. 17. Comparison of RL algorithms with maximum operator parallelism $K^{max} = 20$.

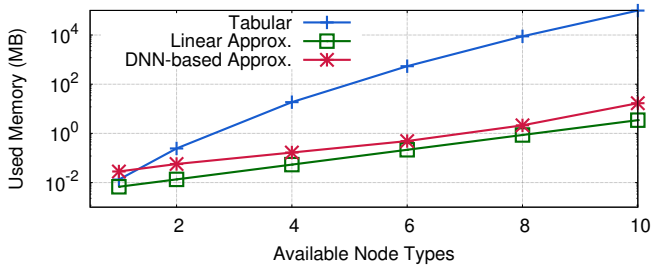

Fig. 18. Memory required by standard (i.e., tabular) RL algorithms vs. linear and DNN-based function approximation.

control. For this reason, it is worth considering function approximation techniques, starting with the linear **QL+FA** that we presented in [54]. Linear FA dramatically reduces memory demands (see, Figure 18) and also leads to faster learning convergence compared to plain QL in several configurations. Unfortunately, it also suffers when considering infrastructure B, where sub-optimal decisions about node types can have a huge impact. In this setting, QL+FA leads to unacceptable performance violations (up to 50% on average).

To overcome these limitations, in this work we have turned our attention towards nonlinear FA by means of DNNs. Experimental results show that DQL performs consistently better than plain QL, exploiting the generalization ability of the underlying DNN. It leads to less than 1% of performance violations in infrastructure A, with slightly worse accuracy in infrastructure B, where in some

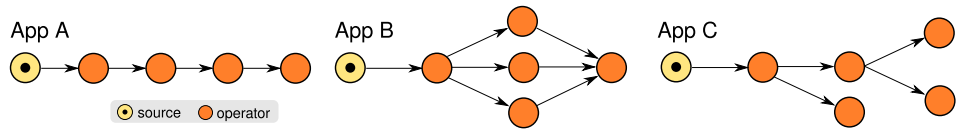

Fig. 19. DSP applications used in the experiments.

configurations violations increase up to 9%. When we introduce the PDS in the algorithm, we note that DQL+PDS has a similar behavior to what we observed with QL+PDS. It clearly outperforms DQL in infrastructure A, under both the workloads, whilst the benefits decrease (and even disappear in some settings) in the more challenging infrastructure B. However, differently from the tabular case, DQL+PDS still provides acceptable results in the worst case, with performance violations exceeding 10% in few configurations, but staying below 5% in the vast majority. Again, by exploiting cost estimates, DQL-PDS$^+$overcomes this issue and consistently performs well in all the considered configurations, blending together the benefits of nonlinear FA and model-based initialization.

6.3 Application Manager Policies (Q3)

We now turn our attention to the whole auto-scaling solution, where multiple OMs are supervised by an AM at the first layer of the control hierarchy. Therefore, we consider three DSP application graphs (i.e., A, B and C), which are illustrated in Figure 19. In particular, applications A and B have been frequently used in the literature as reference application topologies (e.g., in [30, 34, 45]) and are often indicated as, respectively, *pipeline* and *diamond* topologies. The third application C instead features multiple sinks and end-to-end paths with different length. For all of them, we consider a maximum response time R_π^{max}, equal for all their end-to-end paths. Unless differently stated, we will set $R_\pi^{max} = 40$ ms and require response time violations and reconfigurations not to exceed, respectively, $\eta_V = 5\%$ and $\eta_R = 10\%$. While most of the setup is left unchanged with respect to the previous section, in this experiment we consider the Taxi workload only, as it does not directly impact the AM policy. Furthermore, as the OM scalability has already been discussed, we focus on the case where $N_{res} \in \{1, 3\}$. We equip the OMs with the QL-PDS$^+$algorithm, which has been shown to work well.

As regards BO, we used the following settings. After preliminary experiments, we selected the Matérn kernel with parameter $\nu = 5/2$ (see, [18]) as the surrogate function, and *expected improvement* for the acquisition. We limit the number of iterations N_{BO} to 25. At each iteration, the objective function is evaluated running a simulation of the system with length 50,000 steps.

Baseline. As a baseline technique, we consider a model-free naive approach to apportion latency to operators and tune their objectives. According to this approach, the objective weights are simply set as equal, i.e., $w_{res} = w_{rcf} = w_{perf} = \frac{1}{3}$. To set operator response time bounds, we exploit a simple idea: apportioning an equal share of R_π^{max} to each operator along any source-to-sink path π, independently of the operator characteristics. Such baseline can be used both to (i) compare more complex resolution techniques as we do in this section, and (ii) bootstrap the auto-scaling solutions when new applications are deployed and no information is available about them. For the sake of completeness, we report the detailed pseudocode in Appendix A.

Results. We start the evaluation focusing on the latency apportioning problem, where the objective weights are assumed to be given. We report in Figure 20 the monetary cost incurred using the baseline and the BO-based approach, when R^{max} is equal to 50 and 40 ms. We observe that the BO-based AM leads to cost savings in all the considered settings and for all the applications. While not dramatic, the cost reduction is particularly evident in the homogeneous infrastructure (i.e.,

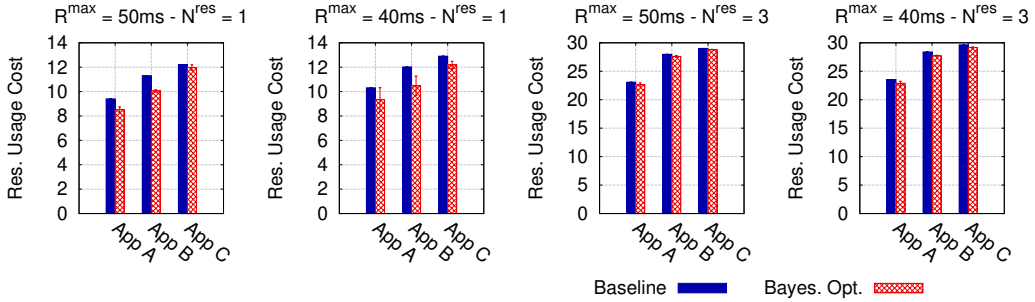

Fig. 20. Monetary operating cost with and without the BO-based AM policy for latency apportioning.

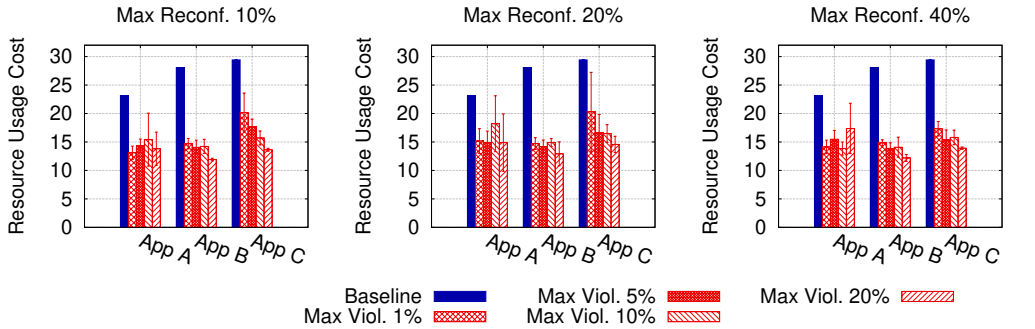

Fig. 21. Monetary operating cost with and without the BO-based AM policy for weights tuning.

$N_{res} = 1$), where the scarce variety of the computing nodes makes it critical to optimally tune the OM objectives to exploit the best trade-off between cost and performance.

We then focus on the problem of tuning the objective weights, given a fixed assignment of the response time bounds (i.e., computed using the baseline approach). Figure 21 shows the monetary cost paid considering different requirements η_V and η_R for all the applications. Compared to the previous experiment, in this case the cost savings provided by the BO-based AM are more evident, as the baseline weights assignment can lead up to double resource usage expenses. More importantly, looking at the QoS requirements, we note that the baseline algorithm actually satisfies the constraints in 42% of the considered runs, whilst the BO approach satisfies the requirements in 96% of the settings. Based on these results, it also emerges that increasing the maximum allowed reconfiguration percentage does not lead to significant cost savings. Conversely, in most the cases the BO approach manages to further reduce costs when allowed to violate maximum response time for more time. This is not possible to evaluate for the baseline, as it is not sensible to different requirements.

As a final experiment, we evaluate the AM when optimizing both the response time bounds and the weights, using Algorithm 4. We show in Figure 22a the results obtained using the baseline and the BO-based AM for different requirements η_V, which again demonstrate how the BO approach does not only reduce costs, but keeps reconfigurations and violations within the desired bounds, as required by users. Figure 22b instead shows the value of the candidate solutions identified by

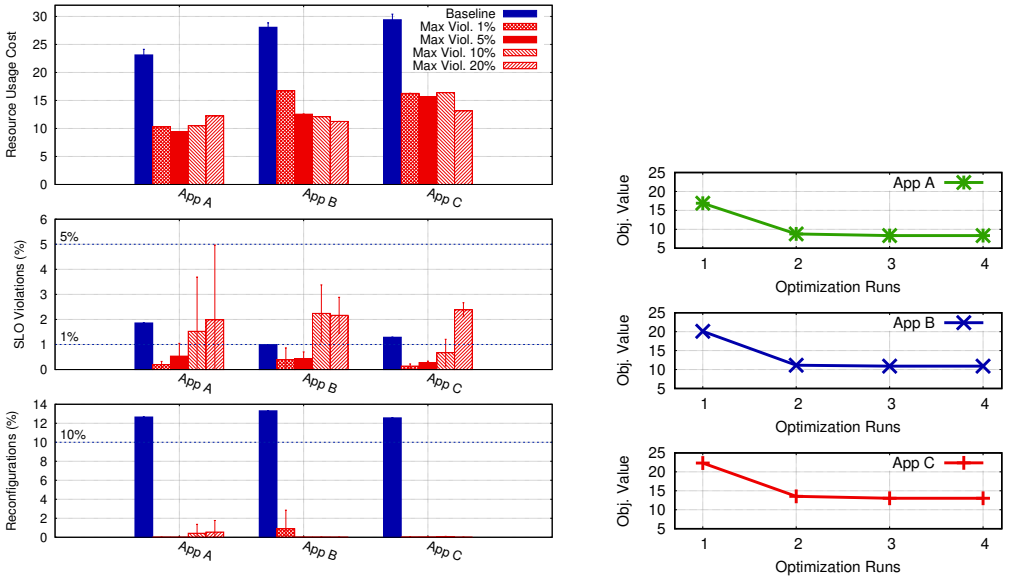

Fig. 22. Results with the complete BO-based AM policy for iterative latency apportioning and weights tuning.

BO throughout a single run of the iterative Algorithm 4. We can observe that the objective value quickly converges to the final value after a couple of iterations.

As regards the execution time, a whole run of the optimization process with $N_{BO} = 25$ takes between 3 and 5 minutes, depending on the application in use. As the optimization can be performed asynchronously with respect to application execution and on a larger time scale compared to the OM auto-scaling, we consider the AM performance acceptable.

6.4 Summary of Evaluation Findings

We can summarize the results we have presented as answers to the three key questions introduced at the beginning of the section.

- **Q1**: While both model-based and model-free auto-scaling approaches from the state of the art perform well in computing infrastructures characterized by limited heterogeneity or no heterogeneity at all, in presence of heterogeneous node types and, hence, performance-cost trade-offs, our RL-based solution achieves superior results, as it learns the best type of node to use given the current scenario.
- **Q2**: Compared to plain Q-learning policies, DRL can deal with larger problem instances (i.e., infrastructures with a larger number of node types). The use of PDS to inject partial model information into the learning process leads to dramatic reduction of the training time necessary to learn a near optimal policy.
- **Q3**: Introducing AM-level policies, we achieve a fully adaptive solution that manages to translate user-given high-level QoS requirements to internal configuration parameters, as needed by the OM policies. This marks an advancement compared to similar RL-based models from the literature (e.g., [69]), where performance objectives can only be specified at local operator level.

6.5 Threats to Validity

Although our experimental analysis covers different scenarios, a few must be considered concerning the validity of these results and their generalization to other types of applications or environments.

Internal Validity. Our evaluation relies on simple performance models for the operators, whose replicas are modeled as single-server queues. By means of analytical results for these models, we computed the response time of each operator at the end of every time slot in the experiments. While we remark that the proposed improvement (i.e., especially the use of partial model information within the RL framework) does not depend on the model used to obtain performance measures, we will consider the integration of a more accurate DSP operator simulator or a real prototype for future work.

We also made assumptions on the time scale of auto-scaling decisions. Specifically, we assumed that the control interval is long enough to allow OMs to observe the effects of reconfiguration decisions on the system after transient overheads have disappeared. As scaling operations in a DSP system can often require tens of seconds, the operation frequency of our OMs should be in the order of one or more minutes. In practice, this issue can also be addressed by letting auto-scalers run more frequently and skipping a few iterations after a scaling operation has been triggered.

External Validity. Focusing on the auto-scaling problem, we assumed that replicas of the same operator all run in the same geographical region, i.e., we are not concerned about network-level considerations when picking the type of node to use. Placement [31] decision at level of operators, which may run in different locations, are possibly taken at higher level by a distinct scheduler module, as assumed in similar works (e.g., [69]). Regarding the operator replicas, we also expect the incoming data streams to be evenly distributed among them or, at least, the load imbalance to be limited. If this is not the case, scale-out operations may be unable to actually reduce the load on the bottleneck replica (e.g., in the extreme case where all the incoming data must be routed to a single replica) and different mechanisms should be adopted. Several strategies have been studied in the literature to ensure such load balancing even in presence of stateful operators (see, e.g., [46]), and we expect the DSP system to exploit them.

Our experiments consider heterogeneous nodes that deliver different performance levels. However, we do treat different computing nodes of the same type as identical. In practice, a real computing infrastructure (especially if virtualized) may exhibit unpredictable performance variability across different computing nodes, even though created with identical specifics (e.g., due to performance interference). Such performance variability represents an additional source of uncertainty, which could make it harder for agents to learn an optimal policy and, thus, require a longer training phase.

Moreover, we relied on the assumption that the resource provider (e.g., a public Cloud provider) has enough computational capacity to always accommodate acquisition requests coming from the auto-scaler. While we find this assumption reasonable for Cloud- or cluster-based deployments, it would not necessarily hold in resource-constrained environments (e.g., Edge computing). In the future, we will consider an extension of our model that can handle constrained resource capacities and possibly make decisions based on current availability.

When presenting the AM policies in Sec. 5, we adopted the simplifying assumption of having identical response time requirements along all the paths in the application DAG. This assumption allowed us to slightly simplify the presentation of the algorithms, using a single symbol to denote the maximum response time. Nonetheless, all the ideas discussed in the section can be applied to the general case, evaluating the SLO satisfaction along each path according to its specific requirement.

7 CONCLUSION

In this paper, we have presented an auto-scaling solution for DSP applications in heterogeneous infrastructures, based on a two-layered control architecture. An application-level manager steers adaptation, aiming to minimize the cost due to resource usage while guaranteeing the satisfaction of user-given QoS constraints, specified in terms of response time and adaptation overhead. At the operator-level, a RL agent controls operator auto-scaling resolving a local multi-objective MDP.

We devised policies to deal with model uncertainty both at the top and lower layer of the hierarchy, exploiting, respectively, Bayesian optimization and reinforcement learning. The former allows the Application Manager to turn user requirements into weights for the local multi-objective cost function of each Operator Manager. Reinforcement learning is then exploited by Operator Managers to learn a local auto-scaling policy, according to the given cost function. Overall, our approach reduces the number of "knobs" that are directly exposed to users, who can instead focus on the definition of few requirements (i.e., desired maximum response time and tolerated percentage of reconfigurations and response time violations).

Our numerical experiments demonstrate the efficacy of our solution, where we exploit partial knowledge about the system to increase learning velocity. It is worth remarking that throughout the paper we took the worst-case perspective where RL agents can only be trained against on-line collected data, assuming that no historical data sets are available for off-line training. Clearly, if any data is available for such pre-training, our algorithms can leverage it to further reduce on-line training time.

As already mentioned, the approach we presented is applicable beyond the considered domain (i.e., data stream processing). Indeed, the concepts we have presented can be exploited to control run-time adaptation of similar distributed systems, as long as we can (i) define a set of adaptation actions that deterministically alter system configuration, and (ii) formulate a cost function (based, e.g., on the satisfaction of performance requirements and resource usage). For instance, similar RL-based adaptation solutions have been investigated in the literature for microservice-based applications [52] and serverless applications [1].

For future work, we plan to extend our solution and drop some of the assumptions we relied on in this work. In particular, we plan to adopt a more general notion of resource heterogeneity, not limited to the cost-performance trade-off, and possibly account for additional aspects such as different cost models (e.g., *Spot* instances, acquired through bidding) or ethical/strategical differences among providers. We will also consider resource-constrained deployments, where the desired type of computing node may be not always available for acquisition.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments that definitely contributed to the improvement of the paper.

This work has been partially supported by the Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 Componente 2 Investimento 1.4: Potenziamento strutture di ricerca e creazione di "campioni nazionali" di R&S (M4C2-19) - Next Generation EU (NGEU).

REFERENCES

[1] Siddharth Agarwal, Maria Alejandra Rodriguez, and Rajkumar Buyya. 2021. A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency. In *Proc. of IEEE CCGRID '21*. 797–803. https://doi.org/10.1109/CCGrid51090.2021.00097

[2] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: Configuring Serverless Functions using Statistical Learning. In *Proc. of IEEE INFOCOM '20*. 129–138. https://doi.org/10.1109/INFOCOM41043.2020.9155363

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[4] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. 2018. Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Trans. Serv. Comput.* 11, 2 (2018), 430–447. https://doi.org/10.1109/TSC.2017.2711009

[5] Marcos D. de Assunção, Alexandre da Silva Veith, and Rajkumar Buyya. 2018. Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions. *J. Netw. Comput. Appl.* 103 (2018), 1–17. https://doi.org/10.1016/j.jnca.2017.12.001

[6] Karl Johan Åström and Richard M Murray. 2021. *Feedback Systems: An Introduction for Scientists and Engineers* (2 ed.). Princeton University Press.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *Proc. of ACM PODS '02*. 1–16. https://doi.org/10.1145/543613.543615

[8] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Decentralized Self-Adaptation for Elastic Data Stream Processing. *Future Gener. Comput. Syst.* 87 (2018), 171–185. https://doi.org/10.1016/j.future.2018.05.025

[9] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Optimal Operator Deployment and Replication for Elastic Distributed Data Stream Processing. *Concurr. Comp. Pract. Exp.* 30, 9 (2018), 20 pages. https://doi.org/10.1002/cpe.4334

[10] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2022. Run-Time Adaptation of Data Stream Processing Systems: The State of the Art. *ACM Comput. Surv.* 54 (2022), 36 pages. Issue 11s. https://doi.org/10.1145/3514496

[11] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. 2016. Elastic Stateful Stream Processing in Storm. In *Proc. of HPCS '16*. IEEE, 583–590. https://doi.org/10.1109/HPCSim.2016.7568388

[12] Dazhao Cheng, Xiaobo Zhou, Yu Wang, and Changjun Jiang. 2018. Adaptive Scheduling Parallel Jobs with Dynamic Batching in Spark Streaming. *IEEE Trans. Parallel Distrib. Syst.* 29, 12 (2018), 2672–2685. https://doi.org/10.1109/TPDS.2018.2846234

[13] Tiziano De Matteis and Gabriele Mencagli. 2017. Elastic Scaling for Distributed Latency-Sensitive Data Stream Operators. In *Proc. of 25th Euromicro Int'l Conf. on Parallel, Distributed and Network-based Processing, PDP '17*. IEEE Computer Society, 61–68. https://doi.org/10.1109/PDP.2017.31

[14] Tiziano De Matteis and Gabriele Mencagli. 2017. Proactive Elasticity and Energy Awareness in Data Stream Processing. *J. Syst. Software* 127 (2017), 302–319. https://doi.org/10.1016/j.jss.2016.08.037

[15] Fahimeh Farahnakian, Tapio Pahikkala, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. 2014. Multi-agent Based Architecture for Dynamic VM Consolidation in Cloud Data Centers. In *Proc. of 40th Euromicro Conf. on Software Engineering and Advanced Applications*. 111–118. https://doi.org/10.1109/SEAA.2014.56

[16] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. 2013. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *Proc. of 2013 ACM Int'l Conf. on Management of Data, SIGMOD '13*. 725–736. https://doi.org/10.1145/2463676.2465282

[17] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2023. A Survey on the Evolution of Stream Processing Systems. *CoRR* abs/2008.00842 (2023), 30 pages. arXiv:2008.00842 https://arxiv.org/abs/2008.00842

[18] Peter I. Frazier. 2018. A Tutorial on Bayesian Optimization. *CoRR* abs/1807.02811 (2018), 22 pages. arXiv:1807.02811 http://arxiv.org/abs/1807.02811

[19] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2017. DRS: Auto-Scaling for Real-Time Stream Analytics. *IEEE/ACM Trans. Netw.* 25, 6 (2017), 3338–3352. https://doi.org/10.1109/TNET.2017.2741969

[20] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1447–1463. https://doi.org/10.1109/TPDS.2013.295

[21] Omid Gheibi, Danny Weyns, and Federico Quin. 2021. Applying Machine Learning in Self-Adaptive Systems: A Systematic Literature Review. *ACM Trans. Auton. Adapt. Syst.* 15, 3, Article 9 (2021), 37 pages. https://doi.org/10.1145/3469440

[22] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martinez, Claudio Soriente, and Patrick Valduriez. 2012. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Trans. Parallel Distrib. Syst.* 23, 12 (2012), 2351–2365. https://doi.org/10.1109/TPDS.2012.24

[23] Wenxia Guo, Wenhong Tian, Yufei Ye, Lingxiao Xu, and Kui Wu. 2021. Cloud Resource Scheduling With Deep Reinforcement Learning and Imitation Learning. *IEEE Internet Things J.* 8, 5 (2021), 3576–3586. https://doi.org/10.1109/JIOT.2020.3025015

[24] Thomas Heinze, Leonardo Aniello, Leonardo Querzoni, and Zbigniew Jerzak. 2014. Cloud-based Data Stream Processing. In *Proc. of 8th ACM Int'l Conf. on Distributed Event-Based Systems, DEBS '14*. 238–245. https://doi.org/10.1145/2611286.

2611309

[25] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling Techniques for Elastic Data Stream Processing. In *Proc. of 2014 IEEE Int'l Conference on Data Engineering Workshops*. 296–302. https://doi.org/10.1109/ICDEW.2014.6818344

[26] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. 2015. Online Parameter Optimization for Elastic Data Stream Processing. In *Proc. of 6th ACM Symp. on Cloud Computing, SoCC '15*. 276–287. https://doi.org/10.1145/2806777.2806847

[27] Mohammad R. Hoseiny Farahabady, Ali Jannesari, Javid Taheri, Wei Bao, Albert Y. Zomaya, and Zahir Tari. 2020. Q-Flink: A QoS-Aware Controller for Apache Flink. In *Proc. of 20th IEEE/ACM Int'l Symp. on Cluster, Cloud and Internet Computing, CCGRID '20*. 629–638. https://doi.org/10.1109/CCGrid49817.2020.00-30

[28] Shigeru Imai, Stacy Patterson, and Carlos A. Varela. 2018. Uncertainty-Aware Elastic Virtual Machine Scheduling for Stream Processing Systems. In *Proc. of 18th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing, CCGRID '18*. 62–71. https://doi.org/10.1109/CCGRID.2018.00021

[29] Pooyan Jamshidi and Giuliano Casale. 2016. An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems. In *Proc. of 24th IEEE Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS '16*. 39–48. https://doi.org/10.1109/MASCOTS.2016.17

[30] Roland Kotto Kombi, Nicolas Lumineau, and Philippe Lamarre. 2017. A Preventive Auto-Parallelization Approach for Elastic Stream Processing. In *Proc. of 37th IEEE Int'l Conf. on Distributed Computing Systems, ICDCS '17*. 1532–1542. https://doi.org/10.1109/ICDCS.2017.253

[31] Geetika T. Lakshmanan, Ying Li, and Robert E. Strom. 2008. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Comput.* 12, 6 (2008), 50–60. https://doi.org/10.1109/MIC.2008.129

[32] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. 2018. Model-free Control for Distributed Stream Data Processing using Deep Reinforcement Learning. *Proc. VLDB Endow.* 11, 6 (2018), 705–718. https://doi.org/10.14778/3199517.3199521

[33] Ning Liu, Zhe Li, Jielong Xu, Zhiyuan Xu, Sheng Lin, Qinru Qiu, Jian Tang, and Yanzhi Wang. 2017. A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning. In *Proc. of 37th IEEE Int'l Conference on Distributed Computing Systems, ICDCS '17*. 372–382. https://doi.org/10.1109/ICDCS.2017.123

[34] Xunyun Liu, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Chenhao Qu, and Rajkumar Buyya. 2018. A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications. *ACM Trans. Auton. Adapt. Syst.* 12, 4 (2018), 24:1–24:33. https://doi.org/10.1145/3132618

[35] Björn Lohrmann, Peter Janacik, and Odej Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *Proc. of 35th IEEE Int'l Conf. on Distributed Computing Systems, ICDCS '15*. 399–410. https://doi.org/10.1109/ICDCS.2015.48

[36] Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. 2018. Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems. *IEEE Trans. Parallel Distrib. Syst.* 29, 3 (2018), 572–585. https://doi.org/10.1109/TPDS.2017.2762683

[37] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. 2014. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *J. Grid Comput.* 12, 4 (2014), 559–592. https://doi.org/10.1007/s10723-014-9314-7

[38] Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. 2019. Applications of Deep Reinforcement Learning in Communications and Networking: A Survey. *IEEE Commun. Surveys Tuts.* 21, 4 (2019), 3133–3174. https://doi.org/10.1109/COMST.2019.2916583

[39] Nicholas Mastronarde and Mihaela van der Schaar. 2011. Fast Reinforcement Learning for Energy-Efficient Wireless Communication. *IEEE Trans. Signal Process.* 59, 12 (2011), 6262–6266. https://doi.org/10.1109/TSP.2011.2165211

[40] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. 2020. Turbine: Facebook's Service Management Platform for Stream Processing. In *Proc. of 36th IEEE Int'l Conf. on Data Engineering, ICDE '20*. 1591–1602. https://doi.org/10.1109/ICDE48307.2020.00141

[41] Gabriele Mencagli. 2016. A Game-Theoretic Approach for Elastic Distributed Data Stream Processing. *ACM Trans. Auton. Adapt. Syst.* 11, 2 (2016), 13:1–13:34. https://doi.org/10.1145/2903146

[42] Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. 2020. Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization. In *Proc. of 2020 IEEE Int'l Parallel and Distributed Processing Symp., IPDPS '20*. 831–840. https://doi.org/10.1109/IPDPS47924.2020.00090

[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, et al. 2015. Human-Level Control Through Deep Reinforcement Learning. *Nat.* 518, 7540 (2015), 529–533. https://doi.org/10.1038/nature14236

[44] Weimin Mu, Zongze Jin, Junwei Wang, Weilin Zhu, and Weiping Wang. 2019. BGElasor: Elastic-Scaling Framework for Distributed Streaming Processing with Deep Neural Network. In *Proc. of 16th IFIP WG 10.3 Int'l Conf. on Network and Parallel Computing, NPC '19*, Vol. 11783. Springer, 120–131. https://doi.org/10.1007/978-3-030-30709-7_10

[45] Matteo Nardelli, Valeria Cardellini, Vincenzo Grassi, and Francesco Lo Presti. 2019. Efficient Operator Placement for Distributed Data Stream Processing Applications. *IEEE Trans. Parallel Distrib. Syst.* 30, 8 (2019), 1753–1767.

https://doi.org/10.1109/TPDS.2019.2896115

[46] Muhammad A. U. Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. 2015. The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines. In *Proc. of 2015 IEEE Int'l Conference on Data Engineering, ICDE '15*. 137–148. https://doi.org/10.1109/ICDE.2015.7113279

[47] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. 2009. VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-Configuration. In *Proc. of 6th ACM Int'l Conference on Autonomic Computing, ICAC '09*. 137–146. https://doi.org/10.1145/1555228.1555263

[48] Carl Edward Rasmussen and Christopher K. I. Williams. 2006. *Gaussian Processes for Machine Learning*. MIT Press.

[49] Henriette Röger, Sukanya Bhowmik, and Kurt Rothermel. 2019. Combining it All: Cost Minimal and Low-Latency Stream Processing Across Distributed Heterogeneous Infrastructures. In *Proc. of 20th ACM Int'l Middleware Conf., Middleware '19*. 255–267. https://doi.org/10.1145/3361525.3361551

[50] Henriette Röger and Ruben Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *ACM Comput. Surv.* 52, 2 (2019), 36:1–36:37. https://doi.org/10.1145/3303849

[51] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2023. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 1, Article 11 (2023), 38 pages. https://doi.org/10.1145/3485126

[52] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. 2019. Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning. In *Proc. of 12th IEEE Int'l Conference on Cloud Computing, CLOUD '19*. 329–338. https://doi.org/10.1109/CLOUD.2019.00061

[53] Gabriele Russo Russo, Valeria Cardellini, Giuliano Casale, and Francesco Lo Presti. 2021. MEAD: Model-Based Vertical Auto-Scaling for Data Stream Processing. In *Proc. of 21th IEEE/ACM International Symp. on Cluster, Cloud and Internet Computing, CCGRID '21*. 314–323. https://doi.org/10.1109/CCGrid51090.2021.00041

[54] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. 2019. Reinforcement Learning Based Policies for Elastic Stream Processing on Heterogeneous Resources. In *Proc. of 13th ACM Int'l Conf. on Distributed and Event-based Systems, DEBS '19*. 31–42. https://doi.org/10.1145/3328905.3329506

[55] Jyoti Sahni and Deo Prakash Vidyarthi. 2021. Heterogeneity-aware Elastic Scaling of Streaming Applications on Cloud Platforms. *J. Supercomput.* 77 (2021), 10512–10539. https://doi.org/10.1007/s11227-021-03692-w

[56] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (2016), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

[57] Anshu Shukla and Yogesh Simmhan. 2018. Toward Reliable and Rapid Elasticity for Streaming Dataflows on Clouds. In *Proc. of IEEE 38th Int'l Conf. on Distributed Computing Systems, ICDCS '18*. 1096–1106. https://doi.org/10.1109/ICDCS.2018.00109

[58] Alexandre da Silva Veith, Felipe R. de Souza, Marcos D. de Assunção, Laurent Lefèvre, and Julio C. Santos dos Anjos. 2019. Multi-Objective Reinforcement Learning for Reconfiguring Data Stream Analytics on Edge Computing. In *Proc. of 48th Int'l Conf. on Parallel Processing, ICPP '19*. ACM, 106:1–106:10. https://doi.org/10.1145/3337821.3337894

[59] Rayman Preet Singh, Bharath Kumarasubramanian, Prateek Maheshwari, and Samarth Shetty. 2020. Auto-Sizing for Stream Processing Applications at LinkedIn. In *Proc. of 12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud '20*. 8 pages. https://www.usenix.org/conference/hotcloud20/presentation/singh

[60] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf

[61] Michael Stonebraker, Ugŭr Çetintemel, and Stan B. Zdonik. 2005. The 8 Requirements of Real-Time Stream Processing. *SIGMOD Rec.* 34, 4 (2005), 42–47. https://doi.org/10.1145/1107499.1107504

[62] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2 ed.). MIT Press, Cambridge, MA, USA.

[63] Zhiqing Tang, Xiaojie Zhou, Fuming Zhang, Weijia Jia, and Wei Zhao. 2019. Migration Modeling and Learning Algorithms for Containers in Fog Computing. *IEEE Trans. Serv. Comput.* 12, 5 (2019), 712–725. https://doi.org/10.1109/TSC.2018.2827070

[64] Gerald Tesauro, Nicholas K. Jong, Rajarshi Das, and Mohamed N. Bennani. 2007. On the Use of Hybrid Reinforcement Learning for Autonomic Resource Allocation. *Cluster Comput.* 10, 3 (2007), 287–299. https://doi.org/10.1007/s10586-007-0035-6

[65] Michael Trotter, Guyue Liu, and Timothy Wood. 2017. Into the Storm: Descrying Optimal Configurations Using Genetic Algorithms and Bayesian Optimization. In *Proc. of 2017 IEEE 2nd Int'l Workshops on Foundations and Applications of Self* Systems (FAS*W)*. 175–180. https://doi.org/10.1109/FAS-W.2017.144

[66] Ke Wang, Avrilia Floratou, Ashvin Agrawal, and Daniel Musgrave. 2020. Spur: Mitigating Slow Instances in Large-Scale Streaming Pipelines. In *Proc. of 2020 Int'l Conf. on Management of Data, SIGMOD '20*. ACM, 2271–2285. https://doi.org/10.1145/3318464.3386142

[67] Chris Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3-4 (1992), 279–292. https://doi.org/10.1007/BF00992698

[68] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, et al. 2013. On Patterns for Decentralized Control in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II*. LNCS, Vol. 7475. Springer, 76–107. https://doi.org/10.1007/978-3-642-35813-5_4

[69] Jinlai Xu and Balaji Palanisamy. 2021. Model-based Reinforcement Learning for Elastic Stream Processing in Edge Computing. In *Proc. of IEEE 28th Int'l Conf. on High Performance Computing, Data, and Analytics, HiPC '21*. 292–301. https://doi.org/10.1109/HiPC53243.2021.00043

A DETAILS ON BASELINE APPLICATION MANAGER POLICY

We report below the pseudocode for the baseline AM policy used in the evaluation. This policy equally weights the objective terms and computes the operator response time requirements according to the following algorithm. Equal latency shares are assigned to operators along any source-to-sink path π in the application graph G_{dsp} (lines 2–4). Clearly, for operators belonging to multiple paths, the minimum requirement is selected, possibly leaving spare latency to allocate on some paths. For the sake of simplicity, the latter can be re-distributed to operators proportionally to their current allocation (line 5).

Algorithm 5 Naive R_{op}^{max} computation.

Input: application G_{dsp}, $\mathbf{R}^{max}$
Output: operator max. response times $\boldsymbol{\rho}$

1: $\rho_u \leftarrow \infty, \forall u \in V_{dsp}$
2: **for all** $\pi \in \Pi_{dsp}$ **do**
3: $\quad \rho_u \leftarrow \min\left\{\frac{1}{|\pi|} R_{\pi}^{max}, \rho_u\right\}, \forall u \in \pi$
4: **end for**
5: allocateSpareLatency(G_{dsp}, $\boldsymbol{\rho}$)

B SUPPLEMENTARY EXPERIMENTAL RESULTS

B.1 Supplementary Tables

We report here the full results of the comparison of various operator auto-scaling policies presented in Sec. 6.2.2. Each table shows average cost, percentage of performance violations, reconfigurations and resource cost for both the considered infrastructures and different number of resource types N_{res}. In particular, Table 2 and Table 3 consider the experiments performed using the Taxi trace, whilst Table 4 and Table 5 consider the experiments performed using the Tweets trace.

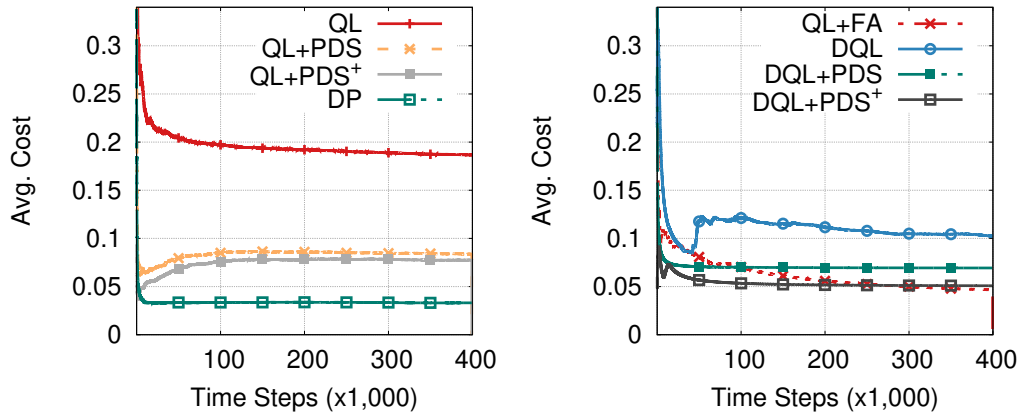

Fig. 23. Cumulative average cost during an experiment with maximum parallelism $K^{max} = 10$ and $N_{res} = 3$.

Table 2. Results of the experiments with the Taxi trace and maximum parallelism $K^{max} = 10$.

		Infrastructure A				*Infrastructure B*			
N_{res}	**OM Policy**	**Avg. Cost**	**Viol. (%)**	**Rcf. (%)**	**Res.**	**Avg. Cost**	**Viol. (%)**	**Rcf. (%)**	**Res.**
3	TB-C	0.0366	< 0.1	1.1	3.44	0.6024	100.0	1.0	0.50
3	TB-S	0.0415	< 0.1	0.8	4.00	0.0200	< 0.1	< 0.1	30.00
3	QL	0.1866	1.1	54.7	7.05	0.1530	13.9	23.6	33.31
3	QL+PDS	0.0836	0.6	4.6	7.07	0.0525	5.5	3.2	19.51
3	QL+PDS$^+$	0.0772	0.3	3.9	6.78	0.0089	< 0.1	0.2	12.51
3	QL+FA	0.0468	0.5	3.2	3.74	0.0779	6.2	14.9	16.27
3	DQL	0.0936	2.5	1.8	7.48	0.0753	2.7	1.3	85.02
3	DQL+PDS	0.0599	0.3	0.1	5.80	0.0430	0.3	0.2	61.35
3	DQL+PDS$^+$	0.0494	0.2	0.2	4.80	0.0294	< 0.1	< 0.1	43.26
6	TB-C	0.0366	< 0.1	1.1	3.44	0.6024	100.0	1.0	0.50
6	TB-S	0.0507	< 0.1	0.8	4.92	0.0200	< 0.1	< 0.1	30.00
6	QL	0.2306	5.0	71.9	5.69	0.6281	78.9	69.0	25.23
6	QL+PDS	0.0756	0.6	3.5	6.49	0.5404	76.2	31.9	29.26
6	QL+PDS$^+$	0.0645	< 0.1	2.0	5.99	0.0522	5.9	2.6	17.64
6	QL+FA	0.1145	1.8	18.7	6.62	0.0318	3.6	2.0	9.59
6	DQL	0.0959	0.6	2.4	8.74	0.1620	16.2	2.1	91.03
6	DQL+PDS	0.0660	0.1	< 0.1	6.52	0.0756	3.6	0.1	80.79
6	DQL+PDS$^+$	0.0548	< 0.1	0.3	5.38	0.0117	0.2	< 0.1	15.35
10	TB-C	0.0366	< 0.1	1.1	3.44	0.6050	100.0	1.0	1.45
10	TB-S	0.0576	< 0.1	0.8	5.61	0.0496	< 0.1	0.8	24.04
10	QL	0.2753	12.0	75.8	5.15	0.6137	74.7	73.6	9.24
10	QL+PDS	0.0740	1.2	3.4	6.00	0.6769	97.9	38.8	5.91
10	QL+PDS$^+$	0.0600	0.2	1.7	5.56	0.0317	2.1	1.0	8.56
10	QL+FA	0.1085	0.7	6.1	9.21	0.2840	35.8	18.5	16.01
10	DQL	0.1260	< 0.1	3.2	11.92	0.0962	4.8	2.1	31.66
10	DQL+PDS	0.0749	< 0.1	< 0.1	7.47	0.0591	3.3	< 0.1	19.69
10	DQL+PDS$^+$	0.0753	< 0.1	0.1	7.51	0.0416	1.1	< 0.1	17.40

B.2 RL Convergence

Figure 23 provides additional insight about the different learning approaches discussed in the paper, with respect to the time they require to learn the optimal behavior. In particular, the figure shows the cumulative average cost incurred by an OM throughout a single simulated experiment using the various learning algorithms. We can note that QL shows very slow convergence, remaining far from the optimal cost achieved by dynamic programming techniques (denoted as **DP**, which determines the optimal MDP policy off-line) even at the end of the experiment. Tabular PDS-based methods instead exploit the available knowledge since the beginning and achieve definitely better results. On the right-hand side of Figure 23, we can see how FA-based algorithms generally perform better than tabular ones. In particular, with the only exception of DQL, which converges slowly towards the optimum, the other algorithms all converge to small cost values, close to the level of DP.

B.2.1 Non-stationary Environment. We verify how learning convergence is impacted by non-stationary working conditions. In particular, we conduct two experiments. In the first one, we consider the case where the workload suddenly changes at run-time. Specifically, we scale the arrival rate of the Taxi trace by 2x after the first half of the experiment. In the second scenario, we simulate a change in the resource offering and, specifically, we let the cost of all the node types

Table 3. Results of the experiments with the Taxi trace and maximum parallelism $K^{max} = 20$.

		Infrastructure A				*Infrastructure B*			
N_{res}	**OM Policy**	**Avg. Cost**	**Viol. (%)**	**Rcf. (%)**	**Res.**	**Avg. Cost**	**Viol. (%)**	**Rcf. (%)**	**Res.**
3	TB-C	0.0194	< 0.1	1.1	3.44	0.6063	100.0	3.0	0.95
3	TB-S	0.0215	< 0.1	0.8	4.00	0.0100	< 0.1	< 0.1	30.00
3	QL	0.1663	0.7	55.7	10.15	0.1651	16.1	25.7	51.44
3	QL+PDS	0.0599	0.1	2.2	10.95	0.2498	31.1	17.1	86.65
3	QL+PDS$^+$	0.0569	< 0.1	1.9	10.56	0.0020	< 0.1	< 0.1	5.86
3	QL+FA	0.0391	0.5	4.7	5.36	0.1222	14.4	12.9	29.61
3	DQL	0.0918	1.1	1.6	16.41	0.0703	1.1	1.5	181.47
3	DQL+PDS	0.0354	< 0.1	< 0.1	7.05	0.0877	0.4	< 0.1	254.93
3	DQL+PDS$^+$	0.0506	< 0.1	< 0.1	10.12	0.0107	< 0.1	< 0.1	31.91
6	TB-C	0.0194	< 0.1	1.1	3.44	0.6063	100.0	3.0	0.95
6	TB-S	0.0261	< 0.1	0.8	4.92	0.0100	< 0.1	< 0.1	30.00
6	QL	0.1886	5.0	65.4	5.58	0.5752	73.1	64.0	25.42
6	QL+PDS	0.0392	0.4	1.0	6.96	0.6824	99.1	41.0	17.82
6	QL+PDS$^+$	0.0319	< 0.1	0.4	6.17	0.0019	< 0.1	< 0.1	5.66
6	QL+FA	0.0907	0.7	15.8	10.97	0.0643	8.5	2.8	23.35
6	DQL	0.0942	0.5	2.8	17.15	0.1359	7.0	2.1	269.94
6	DQL+PDS	0.0499	0.1	< 0.1	9.79	0.2971	44.3	0.1	93.67
6	DQL+PDS$^+$	0.0551	< 0.1	< 0.1	11.01	0.0180	0.4	< 0.1	46.21
10	TB-C	0.0194	< 0.1	1.1	3.44	0.6079	100.0	3.0	1.90
10	TB-S	0.0296	< 0.1	0.8	5.61	0.0256	< 0.1	0.8	24.04
10	QL	0.2449	12.0	73.6	5.10	0.5807	71.8	70.3	9.19
10	QL+PDS	0.0409	0.6	1.0	7.08	0.6786	99.1	40.1	3.91
10	QL+PDS$^+$	0.0312	< 0.1	0.3	6.03	0.0094	0.3	0.2	7.04
10	QL+FA	0.0890	0.2	5.2	15.45	0.2497	34.7	6.8	27.92
10	DQL	0.1251	< 0.1	3.4	23.65	0.0936	3.7	2.5	66.76
10	DQL+PDS	0.0835	< 0.1	< 0.1	16.65	0.1864	20.2	< 0.1	65.14
10	DQL+PDS$^+$	0.0620	< 0.1	< 0.1	12.38	0.0355	0.2	< 0.1	34.52

except one become 100 times higher at run-time (e.g., because of resource scarcity). Figure 24 shows the cumulative average cost in the first experiment. We include in the comparison a dynamic programming (DP, for short) resolution algorithm, which determines the optimal MDP policy offline. We can note that this approach results in the minimal cost in the first part of the experiment and in the highest cost in the end, as it does nothing to adapt to the workload change. Conversely, both the threshold-based and the RL-based algorithms keep acceptable performance after the arrival rate increase.

Figure 25 provides similar insight for the second experiment, where the cost of the available node types is modified. Again, offline DP leads to unacceptable performance after the changes and so does the threshold-based policy, as its resource picking logic is not adaptive. Conversely, RL-based approaches manage to adapt their policies to the new scenario.

B.3 Neural Network Configuration

As explained in Sec. 4, DRL uses a neural network to predict the value function. For this purpose, the network is given a state s as input and processes it through one or more hidden layers. In practice, we have some freedom in deciding (i) how many hidden layers $\mathcal{L}$ to use and (ii) how any state should be turned into an input vector (i.e., picking a function ψ, as in Figure 7). We provide here some results of preliminary experiments we performed to select the parameters to use in our

Table 4. Results of the experiments with the Tweets trace and maximum parallelism $K^{max} = 10$.

		Infrastructure A				*Infrastructure B*			
N_{res}	**OM Policy**	**Avg. Cost**	**Viol. (%)**	**Rcf. (%)**	**Res.**	**Avg. Cost**	**Viol. (%)**	**Rcf. (%)**	**Res.**
3	TB-C	0.1096	5.7	24.1	2.75	0.5361	79.8	28.6	0.43
3	TB-S	0.0861	3.3	17.4	3.13	0.0200	< 0.1	< 0.1	30.00
3	QL	0.1795	0.8	52.8	6.94	0.1557	13.4	25.9	35.27
3	QL+PDS	0.0864	0.6	6.5	7.00	0.1145	12.9	7.6	32.95
3	QL+PDS$^+$	0.0779	0.3	5.4	6.55	0.0079	0.1	0.2	10.21
3	QL+FA	0.1316	1.5	40.8	4.11	0.3884	43.8	57.9	14.48
3	DQL	0.0980	0.7	2.2	8.93	0.0895	1.5	1.6	116.39
3	DQL+PDS	0.0479	0.3	0.1	4.57	0.0201	0.6	< 0.1	24.42
3	DQL+PDS$^+$	0.0467	< 0.1	0.2	4.62	0.0046	< 0.1	< 0.1	6.41
6	TB-C	0.1096	5.7	24.1	2.75	0.5361	79.8	28.6	0.43
6	TB-S	0.0927	3.3	17.4	3.79	0.0200	< 0.1	< 0.1	30.00
6	QL	0.2054	1.8	69.6	5.56	0.5390	63.8	69.4	26.22
6	QL+PDS	0.0764	0.4	5.3	6.36	0.3912	55.6	22.0	20.74
6	QL+PDS$^+$	0.0723	0.1	4.9	6.18	0.0084	0.6	0.3	6.24
6	QL+FA	0.0812	1.0	7.1	6.07	0.4154	49.1	54.3	18.34
6	DQL	0.1036	0.2	3.3	9.59	0.0990	9.3	3.5	54.24
6	DQL+PDS	0.0517	0.3	0.1	4.98	0.1411	20.0	0.4	30.29
6	DQL+PDS$^+$	0.0477	< 0.1	0.2	4.69	0.0121	0.2	< 0.1	16.51
10	TB-C	0.1096	5.7	24.1	2.75	0.5054	75.2	25.8	1.36
10	TB-S	0.0976	3.3	17.4	4.28	0.0896	3.3	17.4	17.42
10	QL	0.2223	4.2	73.0	5.08	0.5207	59.6	72.3	9.32
10	QL+PDS	0.0743	0.7	5.2	5.97	0.5032	72.7	27.7	5.93
10	QL+PDS$^+$	0.0673	0.2	4.4	5.73	0.0180	1.1	0.5	5.12
10	QL+FA	0.0997	0.4	5.2	8.71	0.3246	35.5	39.0	16.79
10	DQL	0.1275	< 0.1	3.6	12.02	0.0965	5.3	3.3	28.93
10	DQL+PDS	0.0581	0.2	0.2	5.67	0.0264	0.6	< 0.1	11.47
10	DQL+PDS$^+$	0.0588	< 0.1	0.2	5.82	0.0195	0.1	< 0.1	9.27

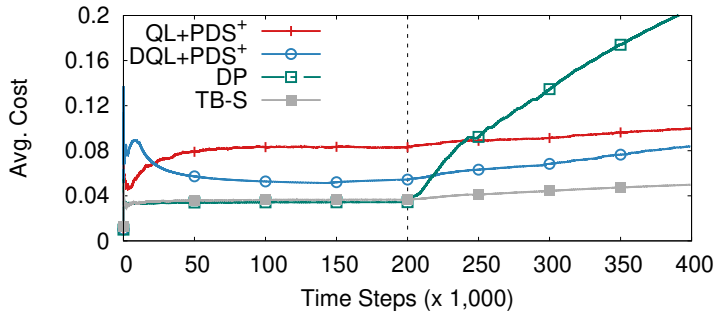

Fig. 24. Average cost with a sudden workload change.

solution, with the aim of showing the impact of picking different configurations for the neural network.

As regards state encoding, a `Minimal` approach is associating each component of the state vector with exactly one component of the input vector after a simple normalization (an example is given in Figure 26a). In this case, it is totally up to the network to extract relevant and meaningful

Table 5. Results of the experiments with the Tweets trace and maximum parallelism $K^{max} = 20$.

		Infrastructure A				*Infrastructure B*			
N_{res}	**OM Policy**	**Avg. Cost**	**Viol. (%)**	**Rcf. (%)**	**Res.**	**Avg. Cost**	**Viol. (%)**	**Rcf. (%)**	**Res.**
3	TB-C	0.0959	5.7	24.1	2.75	0.5108	74.3	32.5	0.83
3	TB-S	0.0705	3.3	17.4	3.13	0.0100	< 0.1	< 0.1	30.00
3	QL	0.1592	0.4	53.8	9.80	0.1704	15.7	28.9	54.52
3	QL+PDS	0.0602	0.1	4.0	10.29	0.0505	6.7	2.3	16.19
3	QL+PDS$^+$	0.0553	< 0.1	3.4	9.66	0.0033	< 0.1	< 0.1	9.74
3	QL+FA	0.1462	0.8	53.3	6.91	0.3315	28.2	75.0	36.26
3	DQL	0.0876	0.5	2.2	15.99	0.0467	0.9	1.6	113.83
3	DQL+PDS	0.0359	0.3	< 0.1	6.86	0.0988	15.2	0.1	21.58
3	DQL+PDS$^+$	0.0384	< 0.1	< 0.1	7.65	0.0086	< 0.1	< 0.1	24.90
6	TB-C	0.0959	5.7	24.1	2.75	0.5108	74.3	32.5	0.83
6	TB-S	0.0738	3.3	17.4	3.79	0.0100	< 0.1	< 0.1	30.00
6	QL	0.1624	1.8	62.2	5.41	0.4962	59.5	65.2	26.84
6	QL+PDS	0.0347	0.3	1.6	5.96	0.4932	71.8	28.6	15.50
6	QL+PDS$^+$	0.0305	< 0.1	1.2	5.52	0.0021	< 0.1	< 0.1	5.53
6	QL+FA	0.0894	0.9	17.2	9.94	0.2551	29.0	35.7	29.83
6	DQL	0.0920	0.4	2.9	16.77	0.0719	6.3	3.1	83.86
6	DQL+PDS	0.0557	< 0.1	< 0.1	11.03	0.0608	1.6	0.1	152.07
6	DQL+PDS$^+$	0.0579	< 0.1	< 0.1	11.52	0.0018	< 0.1	< 0.1	5.17
10	TB-C	0.0959	5.7	24.1	2.75	0.4956	72.6	29.2	1.75
10	TB-S	0.0762	3.3	17.4	4.28	0.0722	3.3	17.4	17.42
10	QL	0.1902	4.3	69.7	5.01	0.4945	57.6	69.9	9.42
10	QL+PDS	0.0338	0.4	1.5	5.65	0.4863	71.1	27.6	4.38
10	QL+PDS$^+$	0.0295	< 0.1	1.1	5.33	0.0067	0.3	0.1	4.71
10	QL+FA	0.0861	< 0.1	4.2	15.44	0.2431	27.9	23.7	28.28
10	DQL	0.1217	< 0.1	4.1	22.68	0.0830	4.4	3.6	49.37
10	DQL+PDS	0.0785	< 0.1	0.1	15.57	0.0409	0.9	< 0.1	35.49
10	DQL+PDS$^+$	0.0766	< 0.1	0.1	15.26	0.0404	4.4	0.1	13.87

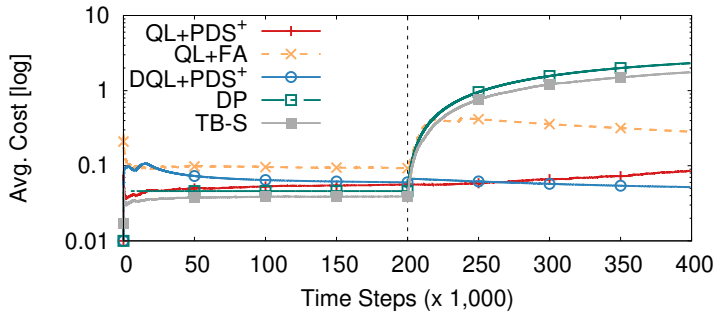

Fig. 25. Average cost with a sudden change in the cost of resources.

information from the input. While this is possible and desirable, because we do not want to rely on human-driven feature engineering, better results in terms of learning speed can be achieved by engineering the state representation.

An alternative approach (denoted as `Large`) exploits *one-hot* vectors to encode the value of the state components $\mathbf{k}$ and λ, as neural network have been shown to work well on binary inputs

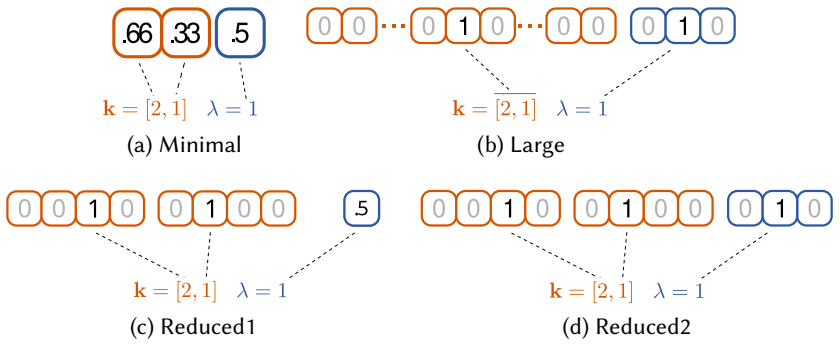

Fig. 26. Example of different state representations as network input vectors. In the example, we assume $N_{res} = 2$, $K^{max} = 3$ and $\lambda \in \{0, 1, 2\}$.

(see Figure 26b). Clearly, this strategy has the drawback of leading to a large number of input neurons and, hence, parameters as the problem scale grows. Therefore, we try to find a better balance between representation accuracy and computational demand. For this purpose, we also use a strategy that independently encodes each component of $\mathbf{k}$ (i.e., the number of instances for each node type). According to this approach, we define two configurations where (i) the input rate λ is encoded as single neuron (denoted as `Reduced1` in Figure 26c) or (ii) the input rate is encoded using a one-hot vector (`Reduced2`, Figure 26d).

Figure 27 shows the average cost paid by DQL using different state representations and $1 \leq \mathcal{L} \leq 3$, in a reference scenario with $K_{max} = 10$ and $N_{res} \in \{3, 6, 10\}$. Overall, we can note that the performance difference across the considered configuration space is not significant. This observation marks a positive point for DRL, as it manages to achieve acceptable accuracy with multiple different configurations, while identifying the optimal hyperparameters is often difficult. As regards the hidden layers, the configuration with three layers provided minor advantages over the alternative ones and we kept it for the experiments. The state coding strategy with best results is `Large`. However, this approach is also the most demanding both in terms of memory and computation to train the network. Therefore, we only tested it with $N_{res} = 3$, where it was computationally feasible. The other coding strategies led to very similar results, with `Reduced2` emerging as the most consistent one, with an acceptable memory footprint, and was selected for use in the other experiments.

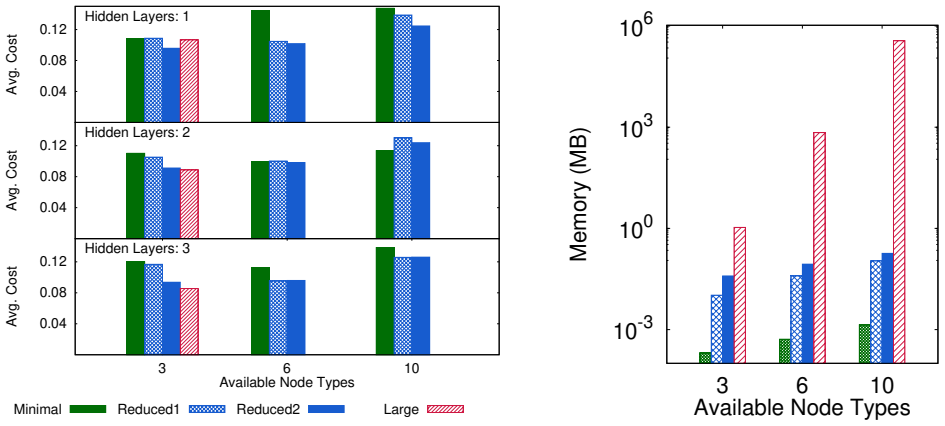

Fig. 27. (a) Average cost paid using DQL with different NN configurations and state representations, and (b) memory required to store NN weights under different state representations, in the case of 3 hidden layers.