RLQ: Workload Allocation with Reinforcement Learning in Distributed Queues

Alessandro Staffolani, Victor-Alexandru Darvariu, Paolo Bellavista and Mirco Musolesi

Abstract—Distributed workload queues are nowadays widely used due to their significant advantages in terms of decoupling, resilience, and scaling. Task allocation to worker nodes in distributed queue systems is typically simplistic (e.g., Least Recently Used) or uses hand-crafted heuristics that require task-specific information (e.g., task resource demands or expected time of execution). When such task information is not available and worker node capabilities are not homogeneous, the existing placement strategies may lead to unnecessarily large execution timings and usage costs. In this work, we investigate the task allocation problem within the *Markov Decision Process* framework, where an agent assigns tasks to an available resource, by receiving a numerical reward signal upon task completion. This allows our solution to learn effective task allocation strategies directly from experience in a completely dynamic way. In particular, we present the design, implementation, and experimental evaluation of *RLQ* (Reinforcement Learning based Queues), i.e., our adaptive and learning-based task allocation solution that we have implemented and integrated with the popular Celery task queuing system. By using both synthetic and real workload traces, we compare *RLQ* against traditional solutions, such as Least Recently Used. On average, using synthetic workloads, *RLQ* reduces the execution time by a factor of at least $3\times$. When considering the execution cost, the reduction is around 70%, whereas for the time waited before execution, the reduction is close to a factor of $7\times$. Using real traces, we observe around 70% improvement for execution time, around 20% for execution cost and a reduction of approximately $20\times$ for waiting time. We also analyze *RLQ* performance against E-PVM, a state-of-the-art solution used in Google's Borg, showing that we are able to outperform it in the synthetic data evaluation, while we outperform it in all the three settings based on real data.

Index Terms—task allocation, reinforcement learning, distributed task queuing.

---

1 Introduction

THE problem of task scheduling concerns performing allocations to resources so as to satisfy desired, often conflicting, objectives (such as throughput, latency, or fairness) while accounting for underlying architectural properties. This problem presents itself at multiple levels in computer systems; notable examples include scheduling of threads on processors [1], [2], scheduling of packets in network infrastructure [3], [4], stream processing [5], [6], software cache management [7], and scheduling of tasks and resources in clusters [8], [9], [10], [11].

More specifically, in this paper we are interested in *application-level workload scheduling for distributed queues with zero information about the managed tasks*. The goal is to allocate tasks (units of work) to resources (entities capable of executing said work). In this context, producers place units of work on shared job queues, which are consumed and executed by distributed worker nodes. Such middleware systems are widely used since they provide resilience to connectivity or node failures. They are also easy to scale horizontally and allow loose coupling between producers and consumers. In fact, distributed task queues also allow for the execution of workloads in third-party environments, where workers are not owned by the system administrator and information about the underlying hardware utilization is not available (or expensive to obtain). A relevant use case is that of federated cloud deployments, where the cloud infrastructure belonging to several owners is leased to a client in order to satisfy its business needs. In this situation, the infrastructure owners typically limit hardware monitoring. Another example is that of citizen science projects such as SETI@home [12] and Folding@home [13], in which volunteers lend their computational resources for running simulations, but extensive and detailed real-time monitoring is avoided because of potential privacy concerns.

Notable distributed queue examples include *Celery* [14] and *RQ* [15] in the Python ecosystem; *Resque* [16] for Ruby, and *Bull* [17] for Node. Such systems are typically built on top of a message broker (such as RabbitMQ or Redis), which handles the underlying communication primitives. Tasks are allocated according to a least recently used strategy to worker nodes depending on their order of arrival, in accordance with a priority setting, or using various heuristics.

Limitations of Current Approaches. The mechanisms by which tasks are allocated in the existing solutions above are general-purpose, definitely far from optimality when tasks are long-running and the worker capabilities are strongly heterogeneous. We identify the following shortcomings present in current state-of-the-art systems and methods:

1) *Assumption of uniform worker capabilities*: most current approaches implicitly assume that workers have the same capability, while in reality they may be highly heterogeneous. Variations in hardware (e.g., disk speed, availability of a GPU) and software (e.g., optimized numerical libraries) can cause significant differences in task completion times.

- *Alessandro Staffolani, Paolo Bellavista and Mirco Musolesi are with the Department of Computer Science and Engineering, University of Bologna, Bologna, Italy.*
E-mail: alessandro.staffolani, paolo.bellavista, mirco.musolesi@unibo.it
- *Victor-Alexandru Darvariu and Mirco Musolesi are with the Department of Computer Science, University College London, London, UK and with The Alan Turing Institute, London, UK.*
E-mail: v.darvariu, m.musolesi@ucl.ac.uk

2) *Lack of notion of cost*: available task scheduling systems do not support a notion of monetary cost per unit of time associated with the execution of tasks. In the context of cloud-based infrastructures, for example, it may be desirable to strike a balance between completion time and cost spent using the hardware. Furthermore, when servers are self-managed, costs are sustained for managing and running the resources, which should be kept low.
3) *Lack of adaptability in dynamic settings*: current task schedulers require tuning in order to function well, but cannot automatically update such settings in cases where the frequencies and characteristics of incoming tasks change significantly. If such a shift is substantial, this can be detrimental to performance.
4) *Requiring task-specific information*: sophisticated task schedulers necessitate precise information about the tasks they are going to schedule, such as their estimated time of completion and the resources that are demanded. However, in cases where execution happens on third-party devices that are not under the control of the task queue user, such information may be unavailable.

Our Contributions. In this paper, we make the following two contributions in order to overcome these limitations:

1) We cast the allocation problem as a decision-making process in which tasks of various classes have to be assigned to heterogeneous types of worker nodes. A centralized agent allocates tasks to workers of a particular type, receiving a numerical reward signal based on a measure of the fitness of its assignments, adjusting its behavior so as to maximize it. Several desiderata such as the cumulative time that tasks spend waiting, total execution time, or total execution cost can be transparently captured as a reward signal. We contribute a formulation of this problem based on the Markov Decision Process (MDP) framework, for which learning is possible via contextual bandit as well as deep reinforcement learning algorithms.
2) We present the design and implementation of *RLQ* (Reinforcement Learning based Queues), a learning-based adaptive system for task allocation that is integrated with Celery, a widely-used platform for distributed task processing in Python. In our system, tasks arrive on a "main" queue in an asynchronous way and are allocated to a specific type of worker based only on a task label and the current load of the system. We have implemented two learning algorithms for this purpose, based on the LinUCB contextual bandit algorithm [18] and the DoubleDQN algorithm [19]. In addition, RLQ includes several novel architectural components that enable learning task allocation policies efficiently, while supporting *distributed asynchronous task execution with delayed rewards.*

Results. We have thoroughly evaluated RLQ using both synthetic workload and real workload traces [20] from Borg [8], [21], [22], Google's cluster management system. Our experimental evaluation shows that *RLQ* achieves significantly better performance compared to classic solutions, such as Least Recently Used, when optimizing for objectives that capture various execution time, execution cost and waiting time desiderata. In terms of learning algorithms, we find that DoubleDQN outperforms LinUCB as the complexity of the problem increases. Compared to these baselines, over the synthetic data, on average the time of execution is reduced by a factor of at least 3 ; when considering the execution cost we are able to reduce the expense around 70%; when considering the time waited before executing, the reduction is near a factor of 7 . Furthermore, our results show that RLQ is able to adapt to varying workload frequencies without any need for manual intervention, by maintaining substantially the same performance gain. Using real traces, we observe around 70% improvement for execution time, around 20% for execution cost and a reduction of approximately 20 for waiting time. Finally, we evaluate our solution against a well-established cluster management solution, namely E-PVM [23], the scheduling algorithm used by Borg, E-PVM requires knowledge about task resource requirements and workers' current load in order to allocate tasks, information that is by design unavailable to RLQ. In the experiments using synthetic workload we observe that the additional system information used by E-PVM leads to better performance when the time waited by a task is optimized while, on the other hand, it is outperformed by RLQ when time and cost of the task execution are. In the experiments based on real-traces, DoubleDQN can outperform E-PVM on all the three setups.

2 Related Work

Scheduling is a very vast area; below we only cover the recent papers that are most related and closest to our original approach for learning-based allocation in distributed task queues. Let us note that, to the best of our knowledge, our proposal is the first one that achieves application-level adaptive scheduling for distributed task queues.

Conventional Task Scheduling. Conventional schedulers typically use hand-crafted rules in order to perform allocations. Common examples of strategies include Shortest Job First and FIFO scheduling, possibly in conjunction with Backfilling (jobs that require less resources are moved to the front of the queue) [24]. Other systems allow specifying priority levels, with higher-priority tasks taking precedence. If information about the task (e.g, number of cores and memory required) is known in advance, task allocation may be seen as a bin packing problem, for which efficient heuristics are known and have been applied [9], [23]. To summarize, such approaches typically prioritize simplicity and ease of implementation at the expense of optimality of allocations. Specific job scheduling techniques for deep learning tasks, using features such as predictions of model convergence, can also be used to optimize a variety of task completion objectives [25], [26], [27].

Reinforcement Learning for Scheduling. Reinforcement learning has proven to be an effective tool for solving optimization problems, ranging from the Traveling Salesman Problem (TSP) [28], [29], the vehicle routing problem (VRP) [30], [31], as well as the Job Shop Scheduling (JSP) problem [32], [33]. JSP is a classic combinatorial optimization problem with similarities to the problem we treat, but with two important differences: firstly, tasks have an execution flow, where each task may need to be executed on more than one machine following a precise order, while in the type of task scheduling we considered, tasks are executed only once and only by one machine. Secondly,

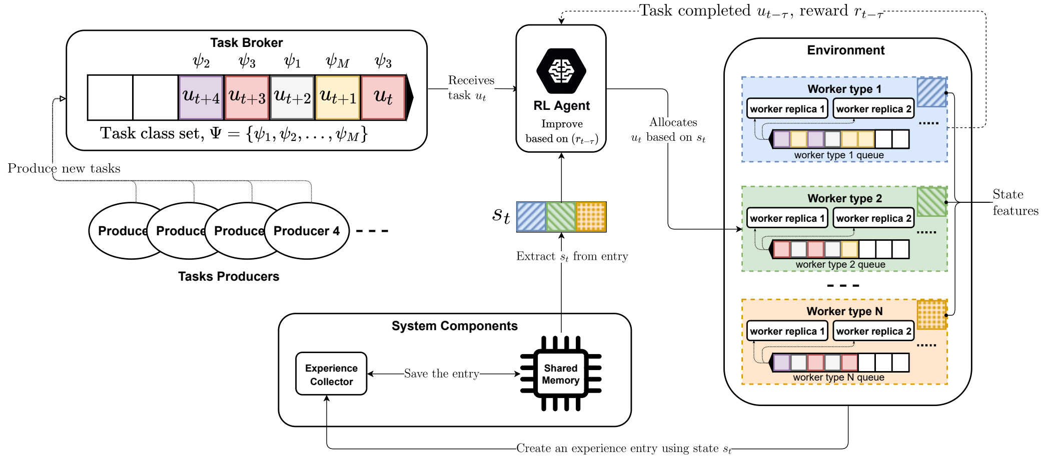

Fig. 1: Graphical summary of our learning-based scheduler for distributed task queues. Tasks of different classes arrive at a main queue of the Task Broker, and an agent is responsible for allocating them to a given type of worker (present in multiple instances). Allocation is performed based on the state and properties of worker nodes, gathered by the Experience Collector and saved in the Shared Memory. Once a task is completed, the agent receives a reward that quantifies the optimality of its allocation choice depending on time and cost measured for task execution.

while in the standard JSP the number of tasks is known in advance, in the setting we consider there is a continuous flow of tasks and decisions must be made at run-time.

Reinforcement learning has been successfully applied to workload allocation problems. The authors of [34] use reinforcement learning in order to allocate jobs for which the time of execution and resources needed are known (or can be estimated). In a subsequent work, the authors use deep RL for scheduling data processing workloads structured as a Directed Acyclic Graph (DAG), showing substantial performance gains over standard schedulers in both a simulated environment as well as in a deployment based on an Apache Spark cluster [35]. Another line of research, which addresses a different scheduling problem than that of RLQ, has focused on workflows that are specific to deep learning: for example, the authors of [36] train a policy gradient model to treat the problem of optimizing the placement of TensorFlow computational graphs on heterogeneous devices, while in [37] an improved method based on separating the device placement problem into multiple decision-making steps is presented.

All the solutions presented in [34], [35], [36], [37] share the common assumption of having access to detailed task-related information (i.e., resource requirements and / or estimated time for the execution) upon task arrival in order to make an assignment. Such detailed information is often not available in distributed task queues, in which the end user may not have full ownership of (or monitoring capabilities for) worker nodes. Indeed, none of the existing distributed task queues make use of this type of information for scheduling workloads, resorting to simpler allocation strategies that do not require it. Even in cases where such information may be available, an additional benefit of the proposed approach is that it does not require the user to go through the (time-consuming) manual or automatic step of profiling the resource usages of their tasks on a variety of heterogenous workers.

3 Scheduling in Distributed Task Queues Using Reinforcement Learning

In this section, we provide the necessary background and fix the notation for decision-making processes. We subsequently introduce two algorithms we use to learn task allocation strategies: one based on contextual bandits, and the other on the full reinforcement learning problem. Finally, we formalize scheduling in distributed task queues as a reinforcement learning problem, providing details about the state, action, and reward specifications used.

3.1 Reinforcement Learning (RL) Background

RL is a paradigm for decision-making that is reward and experience-driven. It has proven to be very effective in learning how to solve several different kinds of problems such as games [38], [39], language generation and understanding [40], [41], packet traffic control [42], [43], and many others.

The underlying formalization of decision-making used in RL is that of Markov Decision Processes (MDPs). In this paradigm, an *agent* interacts with an external *environment* by performing actions, receiving as feedback a numerical signal (*reward*) that quantifies the "goodness" of the performed action. These interactions form a trajectory, defined as $s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \ldots$ for $t = 1, 2, 3, \ldots$, where $s_t \in S$ is the state of the environment at time t, $a_t \in A$ is the action taken at time t, $r_{t+1} \in R$ is the reward associated to the execution of a_t in the current state s_t. S, A and R are the set of possible states, actions and rewards respectively. The transition between s_t and s_{t+1} is modeled by the *transition dynamics* $P(s_{t+1}|s_t, a_t)$ that determine the probability of moving from s_t to s_{t+1} when performing action a_t.

An MDP is fully characterized by the tuple $\langle S, A, P, R, \gamma \rangle$, where $\gamma \in [0, 1]$ is the discount factor. The goal of the agent is to find the policy $\pi(a|s)$, a distribution of actions over states, which maximizes the *discounted return* $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ in expectation [44]. At a high level, allocation in distributed task queues may

be viewed as an MDP as listed below (in Section 3.3 a fully specified formulation of it is provided).

- the **agent** is a centralized entity that receives tasks and is responsible for performing allocations;
- the **state** contains information about the task to be assigned and the status of worker nodes;
- the **action** that can be taken by the agent is to allocate a task to a specific worker node, not preventing assignments on different workers that are currently idle;
- the **reward** depends on the allocation optimality.

3.2 Decision-Making Algorithms

Our objective in this work is to determine whether decision-making algorithms are able to efficiently solve the allocation problem in distributed task queues. There are two main classes of algorithms that are applicable in this setting: that of *online learning*, in which the learner is only presented with (and makes decisions using) information about the current state of the system; as well as the *full reinforcement learning* problem, which explicitly accounts for transitions between states of the system, and for which the learner also considers future possible states when making allocation decisions. Since such algorithms have not been applied in this setting, and we have no a priori knowledge of how they might perform, we consider representative algorithms from both families of decision-making approaches.

LinUCB. The *LinUCB* algorithm, introduced in [18], is an approach for the online learning problem in which at each time t the learner is presented with information (called *context*) prior to selecting its action. The selection of the action at time t uses the following criterion:

$$a_t \doteq \operatorname*{argmax}_{a \in A} \hat{\theta}_a^{\top} x_{t,a} + \alpha \sqrt{x_{t,a}^{\top} A_a^{-1} x_{t,a}} \tag{1}$$

where $x_{t,a}$ represents the context of the action a at time t with size d, $\alpha = 1 + \sqrt{\frac{\ln(2/\delta)}{2}}$ is a constant which for any $\delta > 0$ controls the degree of exploration and $\hat{\theta}_a^{\top} = A_a^{-1} b_a$ is a matrix obtained applying ridge regression to the training data A and b, with A being an identity matrix of dimension d and b being a zero vector of size d.

DoubleDQN. One important family of algorithms for the full RL setting is based on the idea of a *action-value function*: given a state s, an action a, and a current policy π, the state-action value function $Q^{\pi}(s, a)$ quantifies the expected return for taking action a in state s, and subsequently following policy π. Q-learning algorithms [45] continuously estimate the Q-function via samples of interactions with the environment. During learning, the agent selects its action accordingly to an ϵ-greedy policy: the action $\operatorname{argmax}_a Q(s, a)$ is selected with probability $1 - \epsilon$, and a random exploratory action $a \in A(t)$ is picked otherwise. Once learning is completed, the agent uses a fully greedy policy.

In practice, state spaces can be considerably large, and thus *value-function approximation* methods are used, in which the lookup table is replaced with a function approximator (commonly, a deep neural network). Concretely, the lookup table $Q(s, a)$ is replaced with a deep neural network $\hat{Q}(s, a; \theta)$ parametrized by network weights θ. This helps generalize across states that are not identical but share common properties. In this work, we opt for the *DoubleDQN* algorithm proposed by [19], which is a modified version of the DQN algorithm [38], [46], created to better deal with the overestimation of Q-values of the original approach. DoubleDQN uses two sets of parameters: θ for the online network, and θ' for the target network, with online network parameters periodically copied to the target network. The agent collects experience tuples (s, a, r, s') through environment interactions, and updates online network weights by backpropagation using the learning target:

$$y = r + \gamma \hat{Q}(s', \operatorname*{argmax}_{a'} \hat{Q}(s', a'; \theta); \theta') \tag{2}$$

3.3 Task Assignment in Distributed Queues as an MDP

In this section, we construct the formal definition of the assignment problem in distributed task queues as a Markov Decision Process. We present an overview of the abstract architecture in Figure 1. Our formulation addresses the shortcomings of the current methods described in Section 1: namely, it is suitable for an environment in which worker nodes have heterogeneous capabilities, it can account for a notion of cost associated with the execution of tasks, and it can capture varying workloads. In this work, we consider a subset of characteristics as exemplar, without loss of generality. The proposed approach can be transparently extended to cases where a larger (or smaller) set of characteristics are taken into consideration since the learning process is based only on the measured time of execution.

Asynchronous Task Execution. The scheduling of a task is an asynchronous operation for which the reward can only be computed when the task completes its execution and its outcome is known. Thus, in contrast with the standard MDP framework, there exists a *delay* between when an action is taken and when the reward can be provided. Since blocking while waiting for a task to complete is wasteful, we instead assign a new task as soon as it arrives, without waiting for the previous one to complete. Formally speaking, the reward $R(s_t, a_t)$ for taking action a_t in state s_t is only revealed at timestep $t + \tau$ instead, with $\tau \geq 1$. Therefore, at each time step t, the agent receives a set H_t comprising the reward associated to all the tasks allocated in previous steps that have completed their execution at time t.[1]

Workers and Tasks. We are given a set of worker nodes of size D, which we refer to as the *worker pool* P. Each worker $w \in P$ can execute only one task at a time and belongs to a worker type ω. We denote the set of all worker types as Ω, with $|\Omega| = N$. Each worker type ω is characterized by:

- a specification of available resources (e.g., amount of RAM, CPU cores), which is identical for each worker of this type;

1. Delays in obtaining feedback occur frequently in practice. We adapt the learning mechanisms as follows to deal with this: for LinUCB, at each step we update the weights for all tasks in the set $\mathcal{H}_t$ (as also performed for the recommendation problem for which the algorithm was originally developed [18]); while for DoubleDQN we add all the (s, a, r, s') tuples in $\mathcal{H}_t$ to the experience replay buffer. Since the task execution times are significantly smaller than the full time horizon over which allocations have to be made ($\tau \ll T$), this still enables learning efficiently. For further results that characterize the ability of agents to learn in decision-making settings with delayed feedback, we refer the interested readers to [47], [48] in the online learning setting and [49] in the full reinforcement learning problem.

- the number of replicas available, denoted $d(\omega)$;
- a scalar value $_{exec}(\omega) \in \mathbb{R}$, which represents the cost per execution timestep for this worker type.

Each task u_t belongs to a task class . We denote the set of all tasks classes as , with $| \ | = M$. Each task class is characterized by its workload, which may take a variable amount of time to complete depending on its arguments and the resources of the worker that executes it.

Problem Statement. At each timestep t, the system receives a task u_t, which must be executed by a worker $w \in P$. The task is completed after $_{exec}(u_t) \in \mathbb{N}$ timesteps, potentially having spent $_{wait}(u_t) \in \mathbb{N}$ timesteps waiting. Given a cost function defined over the task, worker type, as well as waiting and execution intervals, the goal is to assign tasks such as to minimize the sum:

$$\sum_t (u_t, \omega, \ _{wait}(u_t), \ _{exec}(u_t)) \quad (3)$$

We next formalize the states (contexts), actions, and reward function that we use.

Actions. In the problem considered, tasks that arrive on the main queue must be allocated to a worker. Perhaps the most intuitive way to frame the actions available to the task allocation agent is to explicitly decide which worker node the task will be scheduled on. While straightforward, this has the drawback that often worker nodes are unavailable (since they are executing other tasks). Perhaps more importantly, if the composition of the worker pool changes (e.g., a worker node fails and is taken offline, or if a new one is added to cope with demand), the allocation strategy is no longer applicable, and must be re-learned. To deal with these two issues, we instead frame actions differently. We let the action taken by the agent to be the assignment of a task u_t to a worker type $\omega \in$. Therefore, *the set of all the possible actions is equal to the worker types set*, thus $A =$. Hence, it is reasonable to assume that the set A remains fixed during execution. Once a task is assigned to a worker type, it is moved from the main task queue to the internal queue of the worker type, from which it will be popped in a FIFO manner by one of the available worker nodes.

State and Context. In both full RL and online learning the state s_t and context $x_{t,a}$ respectively should encapsulate relevant representations of the setting in which the agent operates – specifically, for the task allocation problem, it must capture information about the task to be allocated as well as the current status of the worker nodes. We construct the state and context representations using only the following features:

- **task class**: the class (u_t) of the task to be assigned;
- **pool load**: represents the current status of the workers in the *worker pool* P. This feature captures the load at time step t for each worker type ω.

The *pool load* feature encodes two important pieces of information about the system: it captures whether worker types are idle and the status of the worker types' queues. We represent the load for worker type ω as (ω), which takes the following values:

- if $(\omega) = 0$, then the worker type ω has all its replicas idle;
- if $(\omega) = 1$, then the worker type ω has one replica busy;
- if $(\omega) = d(\omega)$, then all worker type ω replicas are busy;
- if $(\omega) > d(\omega)$, then the worker type has $(\omega) \ d(\omega)$ tasks in the queue waiting to start executing.

The task class feature is represented as a one-hot encoding vector of dimension M (the number of task classes), while the pool load feature is a real-numbered vector of dimension N (the number of worker types). To ensure comparable statistics across worker types with different numbers of replicas, we normalize the load (ω) by the sum of load statistics of all classes, $\sum_{i=1}^{N} (\omega_i)$. The state representation for the full RL problem is obtained by concatenating the task class and pool load features, resulting in a vector of size $N + M$. For the contextual bandit we require an additional one-hot feature vector of dimension N representing the worker type, resulting in a context vector of size $2N + M$.

Rewards. The reward signal is the information used by the agent for discriminating good from bad actions. We have briefly described the objective for allocation in distributed task queues as minimizing the total cumulative cost (as expressed in Equation 3) over the tasks allocated. Thus, the agent's learning process can be guided by minimizing the cost incurred for each allocation, or conversely, maximizing the negative cost:

$$R(s_t, a_t) = \ (u_t, a_t, \ _{wait}(u_t), \ _{exec}(u_t)) \quad (4)$$

We now define the specific cost functions that we consider from which reward functions are derived. As we detail below, these capture various desirable aspects of allocation strategies in distributed task queues. Note that, for brevity, we omit the arguments of in the definitions below.

Execution time $_{extime} = \ _{exec}(u_t)$, which captures the time needed for executing a task. By minimizing $_{exec}$, this objective leads to the selection of the worker type that (on average) solves a certain task the fastest.

Execution cost $_{excost} = \ _{exec}(u_t) \ _{exec}(a_t)$, which captures the cost of executing a task over a certain worker type, balancing the time for completing the task and the usage cost for executing it.

Waiting time $_{wait} = \ _{wait}(u_t)$, an important objective to be optimized since it allows to minimize the time spent waiting in the queue for each task. Hence, tasks must wait as little as possible before starting execution.

It is worth noting that our agent has no knowledge of task resource requirements (hardware or software), nor of the current resources available in the worker types. Therefore, when it is necessary to deal with failures due to the violation of hard resource requirements of tasks, it is sufficient to extend the reward function to provide a penalty (a negative value) and to requeue the task. The agent will learn, through experience, to avoid allocations to worker types that cause failures.

4 RLQ: Design and Implementation

4.1 System Design

We next present the design of RLQ, our approach for real-time task scheduling that receives tasks on a distributed queue and learns to allocate them so as to optimize the objectives discussed in Section 3. In this section, we discuss each of the components of RLQ (that we previously

presented in Figure 1) in depth, noting that they are implemented as independent entities that can only communicate using the network.

Task Broker. The Task Broker component is the main entry point of the RLQ system. It acts as a proxy, receiving all the tasks generated by Producers, and placing them on an internal queue (called *main task queue*). Every time a task arrives, the Task Broker queries the Agent about which Worker Type ω it should be allocated to, providing only the task class. Finally, it performs the allocation by publishing the task on the separate queue of the Worker Type.

Agent. The Agent component contains the adaptive decision-making functionality. In order to make an allocation decision, the Agent must gather the current state of the environment s_t from the Shared Memory (see below), and perform the necessary computations, i.e., matrix operations, to obtain an action. Having made a decision, the Agent updates the Shared Memory with the experience e_t generated at time t, inserting the selected action a_t and the context x_{t,a_t} (if a contextual bandit algorithm is used). Finally, once tasks complete, the Agent updates its policy using the reward obtained.

Worker Types. For each Worker Type ω, a queue is created and $d(\omega)$ worker replicas are instantiated, each of them subscribed only to the corresponding Worker Type queue. By default, queues are not bounded; therefore, we can assign as many tasks as desired to a Worker Type. Worker replicas of a Worker Type are identical in terms of specification (e.g., CPU, memory and disk capabilities) and can only execute one task at a time. Once a worker replica becomes available, it pops and executes tasks in a FIFO order from its queue.

Experience Collector. The Experience Collector component listens for events on the distributed queue to build the experience entries e and save them into the Shared Memory. Every experience entry is defined as the tuple $e = \langle s_t, a_t, x_{t,a_t}, r_{t+1}, s_{t+1} \rangle$, where x_{t,a_t} is the context related to the selected action a_t, which is part of the tuple only if the agent is a contextual bandit agent. The Experience Collector performs three operations. Firstly, it listens on the *main task queue* for new tasks and when they arrive it starts the generation of a new experience entry e_t by creating the state s_t and saving the partial e_t in the Shared Memory. Secondly, every time it generates the state s_t, it attaches the state as next state to the experience e_{t-1}. Thirdly, when a task u_t, completes its execution at time $t + \tau_{exec}(u_t)$, the Experience Collector computes the reward $r_{t+\tau_{exec}(u_t)}$ and adds the reward to the experience $e_{t+\tau_{exec}(u_t)}$.

Shared Memory. The Shared Memory is a centralized component that is used by the other components for saving and retrieving information, as well as an event broker for sharing messages and events for synchronization purposes.

4.2 Implementation

Our reference implementation of RLQ is built on top of *Celery*, a widely used Python framework for distributed task processing.[2] In this section, we discuss the implementation details specific to Celery. While some of the implementation details are necessarily framework-dependent, the design

2. The source code for the RLQ implementation and evaluation is available at https://github.com/AlessandroStaffolani/rlq-scheduler.

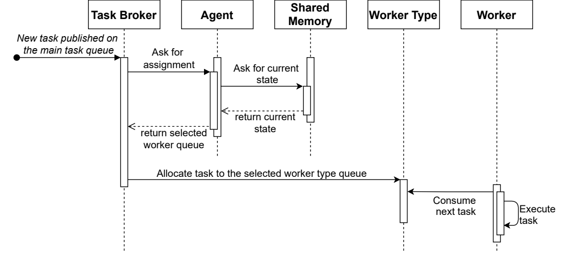

Fig. 2: Sequence diagram for task arrival and allocation.

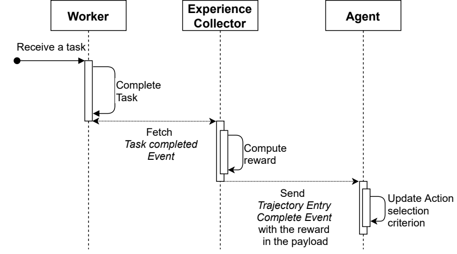

Fig. 3: Sequence diagram for task completion.

and architecture principles apply to all distributed task processing frameworks, such as those discussed in Section 1.

Distributed Task Queue. We delegate the distributed task queue aspect to Celery, which itself uses a distributed message queue implementation, with RabbitMQ and Redis available at the time of writing (Redis in our current implementation). Celery supports creating different queues and making workers subscribe to them. In RLQ, as previously mentioned, we have used several queues: the *main task queue*, part of the Task Broker, where all the tasks arrive to be allocated; additionally, we have created a different queue for each Worker Type. Celery allows to schedule tasks by sending a message to a queue. More precisely, it provides a programming interface through which a function can be called with additional parameters and annotations that are used by Celery for specifying the queue on which it will be ran, the task arguments, and other configuration options. This programming interface is used by task producers for placing tasks onto the main task queue, and is also used by the Task Broker to assign a task to a particular Worker Type according to the action selected by the Agent.

High-Level Workflows. RLQ operation can be described by two high-level workflows that rely on Celery events for their implementation:

1) *Task arrival and allocation*, illustrated in Figure 2: A new task is generated by Producers and received by the Task Broker, which contains the main task queue. Subsequently, the Task Broker queries the Agent via HTTP for the allocation decision, which corresponds to the Worker Type queue where the task needs to be assigned. The Agent ingests the current state of the system from the Shared Memory and uses it to select the action, which is sent as a response to the Task Broker. Finally, the Task Broker places the task on the selected Worker Type queue, from where it is consumed and executed by a Worker Replica.
2) *Task completion*, illustrated in Figure 3: when a Worker

Replica completes the execution of a task, a Celery *task succeeded* event is fired. The Experience Collector, upon receiving the *task succeeded event*, computes the reward based on the time needed for execution and the time waited by the task. Afterwards, the Experience Collector notifies the Agent by sending an experience entry completed event. Finally, the Agent obtains the reward from the payload of the event and uses it for updating its policy.

Task Broker. We implement all the logic of the Task Broker as a Celery task, which is executed by a special Celery worker that is the unique subscriber of the main task queue. This enables it to act as a proxy and to intercept all the tasks sent by task producers. Once a task is generated by task producers, it is wrapped into a Task Broker task, and the task class and arguments of the workload itself are passed.

Worker Types. RLQ worker types have been implemented using Celery workers and making each replica of a Worker Type subscribe to the same Worker Type queue.

Task Classes. The task classes in RLQ are mapped to Celery tasks, thus Python functions, which are scheduled and executed on the workers. While tasks of the same class share the same Python instructions, arguments may differ.

State Feature. The state feature is built using two Celery events: the *task sent* event and the *task succeeded* event. The former is fired when a task is sent to a Worker Type queue, while the latter is fired when a task completes its execution. The pool load feature is built by creating a lookup table, where each worker type is associated with a number (ω_i) (initially zero). During execution, when the *task sent* event is received, (ω_i) is increased by one. Conversely, when the *task succeeded* event is received, (ω_i) is decreased by one.

Measuring Execution and Waiting Time. Algorithms for decision-making typically use a discrete timestep t as unit of time. Recall the formulation in Section 3.3, which specifies that a *single task is received per timestep* and must be allocated. Additionally, $_{exec}$ represents the number of timesteps taken for executing a task and $_{wait}$ represents the time the task has spent waiting, with the reward observed at timestep $t + {}_{exec}$. However, measuring execution and waiting times in terms of discrete timesteps is problematic for two reasons. Firstly, it would require expensive synchronization using a global clock such that durations are reported accurately. Secondly, any variation in the number of tasks received would cause the waiting and execution times to vary, since a global clock advances quicker if more tasks are received. For these reasons, in the implementation, the waiting time $_{wait}$ and the time of execution $_{exec}$ are computed using the wall clock time measured in seconds.

5 Synthetic Workload Evaluation

In this section, we first introduce the environment used for our first set of experiments that employ synthetic workload. First we describe the characteristics of the employed deployment and generated workloads. We then detail the training and evaluation procedures. Finally, we present and discuss the results obtained by the considered methods.

5.1 Experimental Settings

Workload. We have created several different task classes, each of which stresses one or more particular resource of the workers (i.e., they may require a considerable amount of CPU clocks, memory, disk or a combination thereof). Each task class is assigned values for parameters n and d, which are ranges that control the complexity of the task. The complexity of the task is drawn from a uniform distribution over the range. Task classes are based on the following simple programs:

- **CPU**: generate a matrix M of dimension $d \quad d$ and compute $(M^{1} \quad M^{T})^{i}$ where $i = 0, 1, \ldots, n \quad 1$;
- **Disk**: write a string of n megabytes into a file, immediately close the file, then reopen it and read its contents;
- **CPU-memory**: perform the same operations as the *CPU* program but keep results of all n operations in memory;
- **CPU-disk**: perform the same operations as the *CPU* program, and additionally save the results to disk.

TABLE 1: Parameters used to configure Task Classes (top) and Worker Types (bottom).

	program	n	d
1	*CPU*	[1, 5)	[40, 60)
2	*CPU*	[3, 8)	[50, 80)
3	*CPU-memory*	[1, 5)	[40, 60)
4	*CPU-memory*	[3, 8)	[50, 80)
5	*disk*	[35, 50)	-
6	*disk*	[40, 80)	-
7	*CPU-disk*	[2, 6)	[40, 60)
8	*CPU-disk*	[3, 10)	[50, 80)

ω	CPU	Memory	Disk	$_{exec}$
ω_1	200m	100MB	128MB	0.85
ω_2	300m	128MB	256MB	1.93
ω_3	500m	180MB	512MB	3.22
ω_4	500m	128MB	128MB	2.94
ω_5	800m	256MB	256MB	4.02
ω_6	1000m	512MB	800MB	5.12

Task class parameters are shown in Table 1. For each program, two different task classes have been created, which share the same program but differ in complexity.

We model the generation of tasks by producers as a Poisson process in which, on average, tasks are generated per minute. The task class of each generated task is drawn from a uniform distribution over all the possible task classes.

Worker Types. The RLQ worker pool consists of six distinct Worker Types, each with distinct hardware allowances. In the bottom half of Table 1, we show the amount of CPU, memory and disk that we have used to configure the different workers in a heterogeneous deployment environment: ω_6 has roughly five times more resources allocated than ω_1. The relative cost per execution second $_{exec}$ of each Worker Type has been chosen by comparing prices per hour of virtual machines with various capabilities on different cloud providers. The CPU values shown in Table 1 are expressed as CPU units where 1000m of CPU is equivalent to 1vCPU/core for cloud providers and 1 hyperthread on bare-metal Intel processors. A value smaller than 1000m means that a fraction of a core is reserved.

Deployment. For deployment, we have used Kubernetes [50] version 1.2, a production-grade container orchestrator used for automated container deployment, scaling, and management. Our cluster is composed of one master and 3 worker nodes, all running Ubuntu 20.04 with reserved resources managed by an OpenStack tenant, where each node is based on 2 2.2GHz Intel Xeon Gold 5220R 24C. The

TABLE 2: Optimized hyperparameters for the synthetic workload (S.W.) and real workload (R.W.) evaluations.

Agent	Objective	Hyperparameters	*S.W.*	*R.W.*
RLQ-LinUCB	Execution Time		2	1
	Execution Cost		2	2
	Waiting Time		2	1
RLQ-DoubleDQN	Execution Time	layers, lr	$2, 0.1$	$3, 0.001$
	Execution Cost	layers, lr	$3, 0.1$	$3, 0.001$
	Waiting Time	layers, lr	$2, 0.1$	$3, 0.001$

master is configured with 8 CPUs, 8GB of RAM and 150GB of disk. The 3 worker nodes are configured with 16 CPUs, 12GB of RAM and 150GB of disk.

5.2 Training and Evaluation Procedure

To ensure statistical validity of the results, each evaluation (and training, where applicable) has been executed 20 times, each using a different random initialization for the agent initial state. The random initialization of the environment has been kept the same across the different 20 runs, ensuring the same workloads are generated. As previously discussed, we have originally implemented two variants of RLQ:

RLQ-LinUCB, based on the LinUCB contextual bandit algorithm. This method has a single hyperparameter, $\in [0, 2]$, which controls the level of exploration.

RLQ-DoubleDQN, based on the DoubleDQN deep reinforcement learning algorithm. The neural network architecture we use is a fully connected network with $layers$ hidden layers and ReLU activations, where the first hidden layer has the same number of units as the state representation, with each subsequent hidden layer having half the number of units of the previous layer. Updates are done with a batch size of 64 using the Adam [51] optimizer and a learning rate lr. For the level of exploration , we use a linearly decayed schedule over all timesteps, starting from 0.65 and finishing at 0.01. The experience replay buffer has capacity 5000, the target network update frequency is 128, and we use a discount factor $= 0.99$.

Optimized Hyperparameters. For RLQ-LinUCB, we considered $\in \{0.1, 1, 1.5, 2\}$. RLQ-DoubleDQN has many hyperparameters that can be explored, however considering the cost in terms of time, we decided to explore an important subset. We considered a learning rate $lr \in \{0.1, 0.01, 0.001\}$ as well as a number of hidden layers $layers \in \{2, 3\}$. The optimal values are reported in Table 2.

Baselines. We compare the performance of RLQ against the baselines listed below.

A **Random** policy, which randomly assigns a task to a Worker Type;

A **Least Recently Used** (**LRU**) policy, which assigns a task to each Worker Type in a round-robin fashion;

An **E-PVM** based policy, a state-of-the art solution used for cluster scheduling [23]. This method computes a single cost value that combines the heterogeneous resources required by a task, minimizing the change in cost obtained when assigning the task to a worker.

The selection of the first two baselines is motivated by the need of maintaining the initial assumptions: namely, that the system has no prior knowledge of the resources needed for running a task, of how long the tasks take to complete, or of the number of tasks to be executed. To be able to compute the cost value, E-PVM needs to access information about the task resource requirements as well as the current level of utilization of worker resources. Since this information is not available by default in our system, we estimate it. The task requirements are estimated based on the average time of execution on the different worker types, while the worker utilization is derived using the capacity of worker resources.

Training Phase. The purpose of the training phase is to let learning-based agents adjust their task allocation strategies based on experience. The results of the training phase are also used to select the hyperparameters of the learning-based agents. To speed up the training and for balancing the trade-off between initial exploration and exploitation, we execute a *bootstrapping phase* before each training run. During the *bootstrapping phase* a random policy is followed, which allows gathering initial experience. We found that this greatly speeds up the time required for training. Each training run has been performed for a total of 5000 tasks, where the first 1000 represent the bootstrapping phase, which ended only when all the 1000 tasks have completed their execution, so to observe and to learn from the reward of all of them. For the training phase, is kept fixed to 60 tasks per minute.

Evaluation Phase. The learning-based agents are initialized with the best model found during the training phase, and **continue learning** while the evaluation happens, so that we may also test scenarios in which workloads vary over time. In fact, during our evaluation, is varied every 1000 tasks. We start with $= 60$ for the first 1000 tasks. We then increase by 10 times (600 tasks per minute). Finally, for the last 1000 tasks, we reduce to 30.

Result Metrics. We measure overall performance in terms of the *total metric value*, which indicates the sum of all the values received for the optimized metric during the entire execution, and depends on the specific reward function used (recall Section 3.3). In order to analyze results during the training and evaluation phases, we consider two additional metrics: *average metric value*, which represents the mean value for each allocation over a time window of 25 time steps, as well as *cumulative metric value*, which measures the sum of the metric value received until a given timestep.

5.3 Evaluation Results

Table 3 presents the overall performance by showing the average total metric value and its confidence interval among the 20 runs performed with a different random seed. The table columns identify the corresponding agent, and the optimized metric. The plots in Figure 4 illustrate, for each optimized objective, how the average metric value per time window and cumulative metric value change over time.

Overall, the results presented in Figure 4 and Table 3 confirm that our solutions, both contextual bandit and reinforcement learning, outperform the baselines when the generation rate is the same used at training time ($= 60$), but also when it reaches values never seen before. In addition, for the execution time (Figure 4a) and execution cost (Figure 4b) objectives, the change in the workload does not

TABLE 3: Average total metric value and confidence interval using synthetic workloads. Lower values are better.

Metric	Random	LRU	RLQ-LinUCB	RLQ-DoubleDQN	E-PVM
Execution Time	10444:1 ± 180:1	10288:0 ± 202:5	3084:9 ± 156:8	**2915.9** ± **114.2**	7808:4 ± 253:4
Execution Cost	23325:9 ± 1002:2	22731:5 ± 906:4	**13144.1** ± **423.0**	**13029.5** ± **660.6**	18601:2 ± 362:6
Waiting Time	986963:0 ± 114217:2	1000484:7 ± 93778:8	305950:6 ± 81805:9	269408:6 ± 64116:3	**93149.3** ± **3091.8**

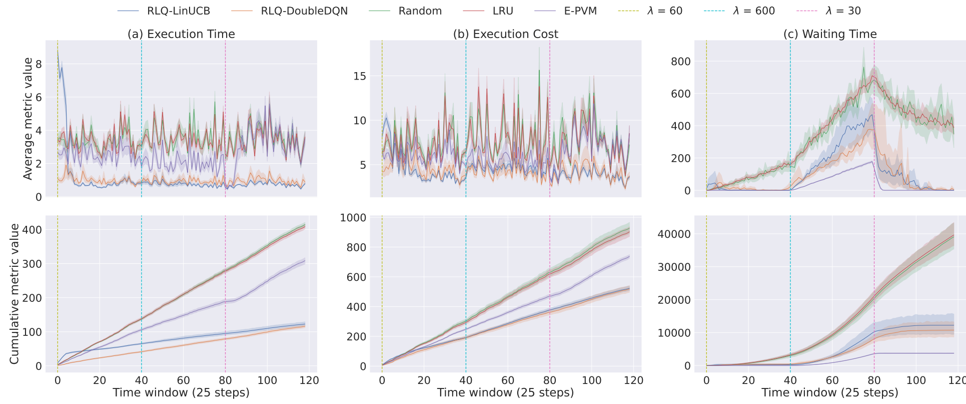

Fig. 4: Evaluation against baselines for each optimized objective using synthetic workloads. Lower values are better.

affect performance, and both *LinUCB* and *DoubleDQN* perform nearly the same and even outperform E-PVM which does not directly optimize those objectives. This confirms the ability of the agents to find the mappings that execute the tasks quickest (Figure 4a) or that execute the tasks fastest while expending the least cost (Figure 4b).

Considering instead more complex objectives such as waiting time (Figure 4c), we notice that changes in   cause noticeable changes in performance. Notwithstanding, RLQ methods outperform the baselines considerably, while, as expected, more task information leads to improved performance for *E-PVM* when optimizing for waiting time. *DoubleDQN* seems to be faster in adapting in the scenario where the rate   is increased. We believe this adaptability is due to the generalization power of its state-action value function approximator, which allows the agent to generalize between similar situations even if their concrete instances may have never been seen before. In contrast, *LinUCB* keeps no approximation of the context it receives, thus it needs to effectively change its policy, requiring more time. On the other hand, *LinUCB* seems more capable to re-adapt in situations with a smaller . The ability of *LinUCB* to change the policy quickly allows reaching a stable regime quicker. The same does not apply for *DoubleDQN*, because once it has changed its policy it requires more iterations for removing the entries gathered during the period with higher   from the experience replay. It is also worth noting how during the initial few steps, the learning-based agents (especially *LinUCB*) perform worse than the baselines for all the objectives. This behavior is due to distribution changes from training to evaluation, causing an initial wrong bias for the agent. Nevertheless, after the initial timesteps, the agents are able to adapt their policies.

System Scalability. We also stressed the scalability of RLQ to examine its capacity to manage an intense workload by increasing the generation rate   significantly, as shown in Figure 5. Results show that a bigger generation rate (up to the value of   = 18000 tasks per minute) does not affect the time needed for choosing an assignment, which remains constant among the different agents and baselines. This is mainly due to the design choice of building RLQ on top of Celery, which in turn delegates the management of received tasks to the underlying distributed queue (in our case Redis). It is worth mentioning that the increased latency incurred by the *LinUCB* agent is due to the time needed for building the additional context feature (recall Section 3.3).

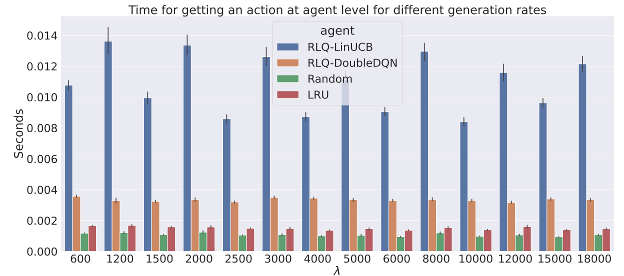

Fig. 5: Average time taken by the Agent to decide an action for different generation rates .

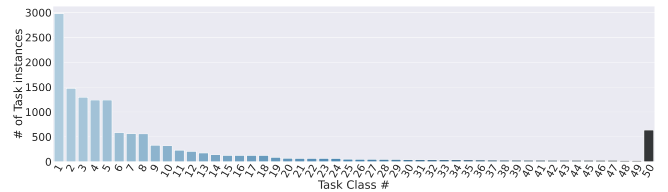

Fig. 6: Number of task instances for each task class extracted from Borg's traces.

TABLE 4: Real workload worker types configuration.

!	s(!)	d(!)	exec
$!_1$	0:5	20	1
$!_2$	0:75	14	2
$!_3$	1	8	3
$!_4$	1:5	5	4
$!_5$	2	3	5

6 Real-World Workload Evaluation

In this section, we describe the real-world traces used for our second set of experiments, as well as the training and evaluation protocol. We also present the results we obtained and the insights we derived from them.

6.1 Experimental Settings

Dataset. In this set of experiments we use the Google Borg's cluster dataset version 3 [20], which comprises traces for the month of May 2019 from 8 different cells. Traces contain information about two high-level resource request types: a *job* (composed of one or more *tasks*) describing the program / computation users want to run, and an *alloc set* (composed of one or more *alloc instances*) describing the resources reserved by jobs for their execution. Each trace is composed of several tables describing the cluster resources and their events during a given workload period. For a detailed description of the dataset, please refer to [52].

Workload. We define 50 task classes from the dataset. We group jobs that share the same *logical collection name*, which is shared by all the jobs that execute the same program. We select as candidate task classes only the jobs that require execution in isolation, since one of the assumptions of the study is that workers execute one task at a time and tasks do not depend on other tasks. Finally, we filter only jobs executed successfully with at least one task instance. This is necessary in order to obtain some statistics in terms of execution process. The result is a dataset that comprises 347 unique jobs with 13730 task instances. However, since most task instances belong to few jobs, as shown in Figure 6, we ultimately created 50 task classes. The first 49 are the jobs with the highest number of instances, while the last one contains all the remaining jobs. By doing so, we capture 95% of the total workload while keeping the number of task classes reasonably low. Tasks wait for a time that is proportional to the time of execution observed in the dataset, which fully characterizes each task class $_i$. To avoid unnecessary waiting, without loss of generality, we map 1 day of execution time in the traces to 20 seconds of real clock time.

Worker Types. Notwithstanding that workers execute all the same programs, we need to define different types with varying capabilities and costs. Therefore, for each worker type $!_i$ we define the *speed factor* $s(!)$, a value that varies the time of execution with respect to the time recorded in the traces. The final time of execution for $!_i$ becomes $^{0}_{exec} = \frac{_{exec}}{s(!_i)}$. The worker pool is composed of 5 worker types with 50 total worker instances, as summarized in Table 4.

6.2 Training and Evaluation Procedure

The procedure for our real workload experiments is the same as for the synthetic one. For the evaluation, we use the tasks as they appear in the dataset, respecting the order, time of execution, and frequency of arrival for new tasks. On the other hand, for training our model, we generate new data accordingly to the distributions observed in the dataset. We set the generation rate equal to the number of tasks observed daily in the dataset. The task classes for new tasks are drawn proportionally to their frequency in the dataset, while the time of execution is sampled uniformly from the execution times of all the instances of a certain task class. We train for a total of 13730 time steps (the same length of the dataset), with a bootstrapping phase of 3000 steps. With respect to the agents' hyperparameters, we set the capacity of the experience replay buffer to 4000 and the target network update to 400, while maintaining the others unchanged. We perform hyperparameter optimization; optimal values are shown in Table 2.

6.3 Evaluation Results

Table 5 presents the average total metric value with its confidence interval among the 20 runs performed with a different random initialization. The table columns identify the corresponding agent and the optimized metrics. The plots in Figure 7 illustrate how the average metric value per time window and cumulative metric value change over time for each optimized objective.

Despite the additional complexity of the real-world dataset, and the increased scale of the experiment, both in terms of task classes and worker pool size, the results confirm the superiority of RLQ. As shown in Figure 7a and Figure 7b, both the contextual bandit and reinforcement learning variants of RLQ outperform all the baselines, including E-PVM, in the optimization of the time and cost of task execution. In the waiting time case (Figure 7c), *DoubleDQN* also outperforms E-PVM. This is not the case for *LinUCB*. This is probably due to the use of function approximators, which are better suited for problems characterized by larger state spaces. The high variability is due to the distribution changes from training to evaluation that we also observed with the synthetic workload. Nevertheless, after the initial timesteps, the agent is able to adapt its policy.

7 Conclusion and Outlook

In this work, we have approached the problem of allocation in distributed task queues, starting from the insight that allocation strategies currently used in existing solutions are simplistic. Our paper makes two primary contributions: firstly, we have formulated the task allocation problem in distributed queues as a Markov Decision Process, and we have shown how learning-based decision-making algorithms (contextual bandit and deep reinforcement learning) can be adapted to the task allocation problem. Secondly, we have presented the design and implementation of RLQ, an adaptive system for task allocation built on top of Celery. We have also conducted an in-depth evaluation, using both synthetic and real workloads, which shows that the learning-based approaches supported by RLQ lead to significant improvements over existing baselines and that they are also able to adapt automatically to changing workloads. In general, the algorithmic and system components of RLQ can

TABLE 5: Average total metric value and confidence interval using real workloads. Lower values are better.

Metric	Random	LRU	RLQ-LinUCB	RLQ-DoubleDQN	E-PVM
Execution Time	14177:4 $\pm$ 172:8	14123:9 $\pm$ 132:9	9600:3 $\pm$ 523:7	**8505.1 $\pm$ 184.2**	10651:0 $\pm$ 43:5
Execution Cost	32754:0 $\pm$ 104:1	32751:4 $\pm$ 94:2	30882:4 $\pm$ 409:3	**27685.2 $\pm$ 376.7**	33433:9 $\pm$ 26:6
Waiting Time	3124:9 $\pm$ 938:3	2271:2 $\pm$ 694:1	3588:4 $\pm$ 2270:0	**117.2 $\pm$ 30.9**	332:2 $\pm$ 50:9

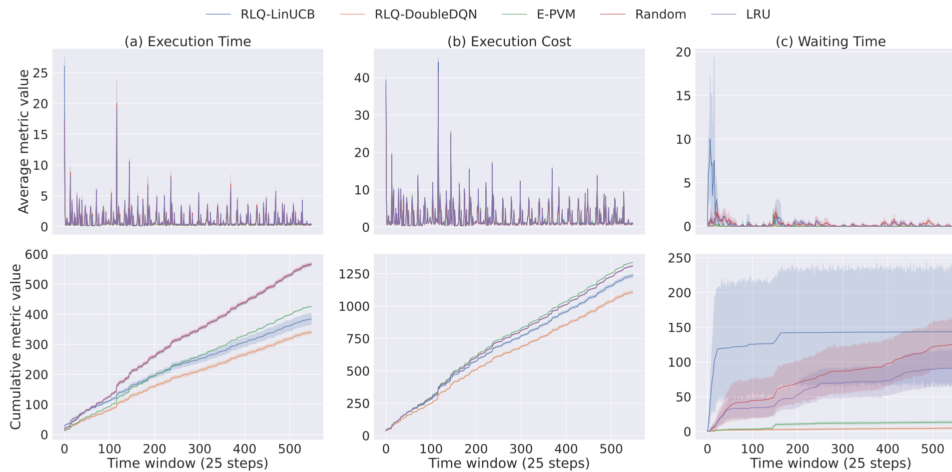

Fig. 7: Evaluation against baselines for each optimized objective using real workloads. Lower values are better.

be of wider use beyond task queues in other distributed allocation and scheduling scenarios.

One current shortcoming of our approach is the assumption of atomic tasks, which cannot capture task dependencies (for example, as opposed to [35]); our method is thus not suitable for situations in which data locality is the most important factor. We believe that our approach represents an essential building block towards the development of architectures able to deal with more complex dependencies. Finally, it is also possible to extend the problem formulation to allow dynamically shrinking and expanding the worker pool so that the system as a whole is cost-effective when the workload volume changes substantially. It is worth noting that in the Google Borg cluster traces [20] used in the version of the revised paper for evaluation, only 0:345% of tasks in the dataset have dependencies on other tasks. Hence, while this is a limitation of the implementation used in the evaluation, the current system can schedule the vast majority of tasks in a real dataset. Moreover, the system can just default to the vanilla Celery allocation mechanism when dependencies are present.

Acknowledgments

This work was partially supported by The Alan Turing Institute under the UK EPSRC grant EP/N510129/1. For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising. We thank the Big Data Innovation & Research Excellence (BI-REX) Competence Center Bologna for access to computational resources.

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[2] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys*, vol. 45, no. 1, pp. 1–28, 2012.

[3] Shang-Tse Chuang, A. Goel, N. McKeown, and B. Prabhakar, "Matching output queueing with a combined input/output-queued switch," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1030–1039, 1999.

[4] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal Packet Scheduling," in *HotNets'15*, 2015.

[5] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Middleware'15*, 2015.

[6] G. R. Russo, V. Cardellini, and F. L. Presti, "Reinforcement learning based policies for elastic stream processing on heterogeneous resources," in *DEBS'19*, 2019.

[7] G. Einziger, O. Eytan, R. Friedman, and B. Manes, "Adaptive software cache management," in *Middleware'18*, 2018.

[8] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys'15*, 2015.

[9] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *SIGCOMM'14*, New York, NY, USA, 2014.

[10] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger, "3Sigma: Distribution-Based Cluster Scheduling for Runtime Uncertainty," in *EuroSys'18*, 2018.

[11] F. Rossi, S. Falvo, and V. Cardellini, "GOFS: Geo-distributed Scheduling in OpenFaaS," in *ISCC'21*, 2021.

[12] SETI@home. [Online]. Available: https://setiathome.berkeley.edu/

[13] Folding@home. [Online]. Available: https://foldingathome.org

[14] Celery. [Online]. Available: https://docs.celeryproject.org/

[15] RQ. [Online]. Available: https://python-rq.org/

[16] Resque. [Online]. Available: https://rubydoc.info/gems/resque

[17] Bull. [Online]. Available: https://optimalbits.github.io/bull/

[18] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *WWW'10*, 2010.

[19] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *AAAI'16*, 2016.

[20] Google. (2019) Cluster Data 2019 traces. [Online]. Available: https://github.com/google/cluster-data/blob/master/ClusterData2019.md