ANANKE: a Q-Learning-Based Portfolio Scheduler for Complex Industrial Workflows

Shenjun Ma
TU Delft, the Netherlands
s.ma@tudelft.nl

Alexey Ilyushkin
TU Delft, the Netherlands
a.s.ilyushkin@tudelft.nl

Alexander Stegehuis
Shell, the Netherlands
alexander.stegehuis@shell.com

Alexandru Iosup
VU Amsterdam and TU Delft
a.iosup@vu.nl

***Abstract*—Complex workflows that process sensor data are useful for industrial infrastructure management and diagnosis. Although running such workflows in clouds promises reduced operational costs, there are still numerous scheduling challenges to overcome. Such complex workflows are dynamic, exhibit periodic patterns, and combine diverse task groupings and requirements. In this work, we propose ANANKE, a scheduling system addressing these challenges. Our approach extends the state-of-the-art in portfolio scheduling for datacenters with a reinforcement-learning technique, and proposes various scheduling policies for managing complex workflows. Portfolio scheduling addresses the dynamic aspect of the workload. Q-learning, allows our approach to adapt to the periodic patterns of the workload, and to tune the other configuration parameters. The proposed policies are heuristics that guide the provisioning process, and map workflow tasks to the provisioned cloud resources. Through real-world experiments based on real and synthetic industrial workloads, we analyze and compare our prototype implementation of ANANKE with a system without portfolio scheduling (baseline) and with a system equipped with a standard portfolio scheduler. Overall, our experimental results give evidence that a learning-based portfolio scheduler can perform better and consume fewer resources than state-of-the-art alternatives, in particular for workloads with uniform arrival patterns.**

I. INTRODUCTION

Many companies are currently deploying or migrating parts of their IT services to cloud environments. To take advantage of key features of cloud computing such as reduced operational costs and flexibility, the companies should effectively manage their increasingly sophisticated workloads. For example, the management of large industrial infrastructures of companies such as Shell involves the usage of complex workflows designed to analyze real-time sensor data [1]. Although the management of workflows and resources has been studied for decades [2]–[4], previous works have mostly focused on scientific workloads [5]–[7] which differ from the industrial applications. Moreover, historically, approaches which are proven to be beneficial for processing scientific workloads have rarely been proven to perform well, or have been even adopted, in production environments [8]. In contrast to the previous body of work, we focus on production industrial workloads comprised of complex workflows, and propose ANANKE, a system for cloud resource management and dynamic scheduling (RM&S) of complex workflows that balances performance and cost.

Compared to scientific workloads, production workloads are more often have detailed and complex requirements. For example, production workloads may utilize different types of task groupings (bags-of-tasks, sub-workflows) with predefined deadlines and specific performance requirements. Production workloads often demonstrate notable recurrent patterns as some tasks could run periodically e g when new data is acquired from sensors. Moreover, new workloads and processing requirements evolve over time. Such requirements translate into a rich set of Service Level Objectives (SLOs), which the RM&S system must meet while trying to reduce the operational costs.

To fulfill dynamic SLOs and save costs, dynamic scheduling techniques, such as *portfolio scheduling* [9] could be applied. A portfolio scheduler utilizes a set of policies, each designed for a specific scheduling problem. The policies are selected dynamically, based on user-defined rules and a feedback mechanism. By combining many policies, the portfolio scheduler can become more flexible and adapt to dynamic workload better than its constituent policies. Previous studies indicate that no single scheduler is able to address the needs of diverse workloads [10], [11], and, in contrast, that a portfolio scheduler performs well without external (in particular, manual) tuning [12].

Although portfolio schedulers are promising for the context of complex workflows, previous approaches [13] lack by design the ability to use historical knowledge in their selection. While portfolio schedulers work well for workloads without pronounced patterns [12], they may not deliver good results for industrial applications that focus on processing real-time sensor data. Thus, to take advantage of historical information, about both the system and the workload, a research question arises: *How to integrate learning techniques into portfolio scheduling to schedule complex industrial workflows?*

To answer this research question, we design a novel cloud-aware portfolio scheduler integrating a reinforcement-learning technique, Q-learning [14]. We explore the strengths and limitations of a Q-learning-based portfolio scheduler managing diverse industrial workflows and cloud resources. Thus, in this work, our main contribution is threefold:

1) Starting from four main requirements (Section III-A), we design ANANKE (Section III-B). The result is an RM&S architecture that integrates into a portfolio scheduler a reinforcement-learning technique, Q-learning (Section III-C) .
2) We build a prototype of ANANKE. The key conceptual contribution of this design is the selection and design of scheduling policies equipped by the portfolio (Section III-D). The prototype is now part of the production environment at Shell, and evolves from the existing *Chronos* system [1].
3) We evaluate ANANKE through real-world experiments (Section IV). Using the cloud-like experimental environment DAS-5 [15] and workloads derived from a real industrial workflow, we analyze ANANKE's performance, and elasticity. We also compare ANANKE with a baseline system and with a portfolio-scheduling-only approach.

II. System Model

We define here the system model used in this work. This model is common in practice, and is used by the Chronos system in the "Smart Connect" project at Shell [1].

A. Workload: Periodic Workflows with Deadlines

In our model, a workload is a set of jobs, where each job is structured as a *workflow* of several *tasks* with precedence constraints. Each workflow is aimed for processing sensor data and has three *chained tasks*: first, the workflow selects the *formula* for calculations from a set predefined by engineers and reads the related raw sensor data from the database. Second, it performs calculations by applying the formula to the raw sensor data. Third, the workflow writes the results back to the database and sends the completion signal.

Workflows in our model are complex due to deadline constraints and periodical arrivals, not due to task concurrency. Because such workflows are designed to process real-time raw sensor data, they have strict requirements for the execution time. Each workflow should be completed before its *assigned deadline*. The workflows which can not accomplish that are considered *expired*. Moreover, each workflow is *executed periodically*, as sensors continuously sample new data and the system needs to update the database at runtime. The chain nature of the workflow means that it does not have parallel parts, and thus requires only a single processing resource (e.g., a CPU core or a thread) for its execution.

B. Processing Sensor Data in Practice: Three-Tier Architecture

In practice, infrastructure monitoring systems commonly use a tree-tier architecture. The three-tier architecture, which depicted in Figure 1, consists of a *master node* (label 1 in the figure), *client nodes* (2), and a *database* (3). Raw sensor data is collected form the monitored facilities and stored in the database. Engineers add to the system a set of workflows for processing sensor data. These workflows are placed in the workflow bucket, which is maintained by the workload manager (b). The client manager (c) controls the set of client nodes and monitors their statuses. At the heart of the architecture, the *scheduler* (a) makes allocation and provisioning decisions. The scheduler selects appropriate workflows from the workload manager and, through the client manager, allocates them to the client nodes.

Every client node reads raw data from the database, performs certain calculations specified in the assigned workflow task, and writes the results back to the database. This model, however, can also be applied for processing other workflow types (e.g., fork-join) with tasks running in parallel. It will require an addition of a separate workflow partitioner which will convert parallel parts into a set of independent chain workflows before their addition to the workflow bucket.

C. Infrastructure: Cloud-Computing Resources

We model the infrastructure as an infrastructure-as-a-service (IaaS) cloud, either public or private. (The Chronos system is currently deployed in a private cloud.) In this work, we assume that all resources are homogeneous. In contrast to typical cloud resource models, our model uses *the computing thread* instead of *complete VMs* as the smallest working unit. Per-thread management enables fine-grained control over resources, allowing the system to perform coarse-grained vertical and fine-grained horizontal scaling. In our model, *vertical scaling* changes the number of active threads within a node, whereas *horizontal scaling* changes the number of active nodes.

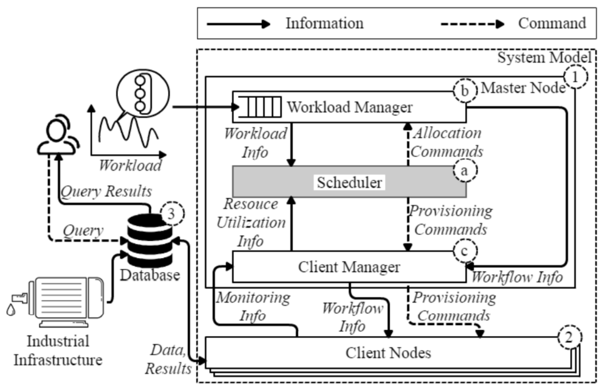

Figure 1: Three-tier architecture for processing workflows.

The resources for our experiments are only on-demand instances. In public clouds such as Amazon AWS [16], instances take some time to fully boot up [17]. However, to have better control over the emulated environment, we use in practice preallocated nodes (zero-time booting) and do not consider node booting times. Because cloud-based cost models can be diverse and likely to change over time, as indicated by the current on-demand/spot/reserved models of Amazon, and the new pricing of lambda (serverless) computation of Amazon and Google, similarly to our older work [10] we use the total running time of all the active instances to represent the *actual resource cost* and not the charged cost.

III. Ananke Requirements and Design

We present here the design of our Ananke system.

A. Architectural Requirements and Design Goals

The key requirements (design goals) for Ananke are:

R1: The designed system must match the model proposed in Section II. This allows the new system to be backward compatible with the system currently in operation, and enables adoption in practice.
R2: The system must implement elastic functionality, e.g., it must be able to automatically adjust allocated resources based on demand changes.
R3: The system should use portfolio scheduling, which shows promise in managing mixed complex workloads [12].
R4: The system should integrate Q-learning into the portfolio scheduler, to benefit from the historical information about the recurrent variability of the processed workloads.

The last two design goals are expected to improve application performance and increase resource utilization, and constitute the main conceptual contribution of this work.

B. Architecture Overview

Ananke extends the Chronos system with the components and concepts needed to achieve R2–4. Matching the model (R1), Ananke is also structured as a three-tier architecture. We further present the design of the Ananke master and client nodes, focusing on the new aspects.

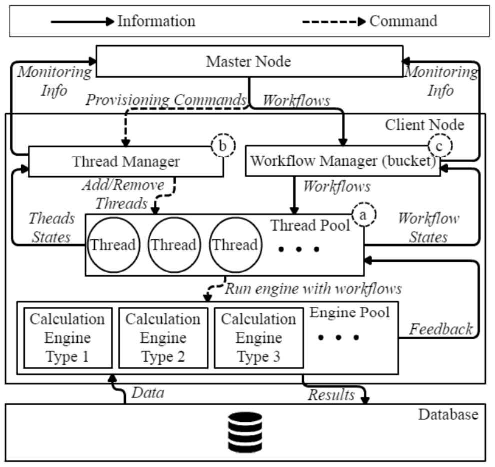

Figure 2: Components of the client node.

1) Master Node: The master node consists of three major components: a scheduler, a workload manager, and a client manager. Figure 1 depicts these components, and their communication pathways. Addressing R3-4 the *scheduler* (component (a) in Figure 1) is a *Q-learning-based portfolio scheduler* equipped with a set of policies: a *Q-learning policy* and other, simpler, threshold-based heuristic policies. As detailed later, in Section III-C, the scheduler hides the complexity of selecting the right policy, by autonomously managing its set of policies and by taking decisions periodically.

The master also uses two managers. The *workload manager* (b) maintains the bucket of workflows, collects the information about the workflow performance, and updates the status of every workflow. The workload manager uses the response and waiting time of a workflow, and the fraction of completed/expired workflows, as key performance indicators. The *client manager* (c) is designed to communicate with all client nodes, collecting continuously per-client resource utilization metrics, and sending to clients when needed commands (actions) and tasks ready to be allocated.

2) Client Nodes: Figure 2 depicts the three components of the client node: a thread pool, a thread manager, and a workflow manager. The *thread pool* (component (a) in Figure 2) maintains a set of threads (smallest working units, see Section II-C). Each thread continuously fetches workflows from the workflow manager and calls an external calculation engine from the engine pool. Each client node can execute multiple threads in parallel. At each moment, a thread executes only a single workflow. We use the number of active threads as the key metric to represent resource utilization. The *thread manager* (b) receives provisioning commands from the master node, and adds or removes threads from the pool. It also reports the resource utilization (the CPU load, and the ratio of busy vs. total number of threads) back to the master node, and continuously terminates idle threads. The *workflow manager* (c) maintains a bucket of workflows allocated to the client node. The workflow manager tracks workflow states and monitors the performance. It also computes the performance metrics and reports them back to the master node. We define two metrics: the number of workflows stored in the bucket normalized by the size of the bucket and the average completion time for the last five executed workflows.

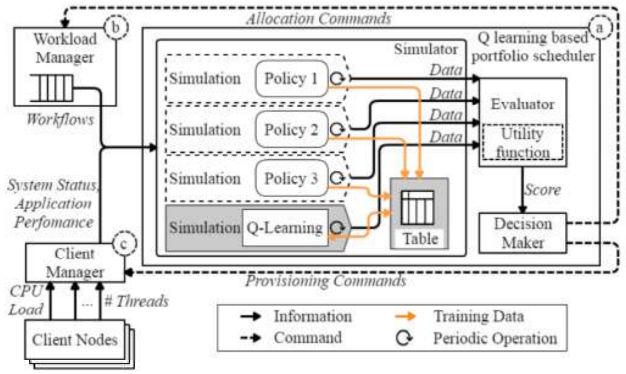

Figure 3: Architecture of a Q-learning-based portfolio scheduler, part of the master node. Data generated by each simulation is used as training data, to fill the decision table.

C. The Q-Learning-Based Portfolio Scheduler

We design an (elastic) Q-learning-based portfolio scheduler for complex industrial workflows in clouds (R3-4).

1) Adding a Portfolio Scheduler to the Architecture: The scheduler component in the master node is based on a portfolio scheduler. Figure 3 depicts the main components of our design. The portfolio scheduler consists of a policy *Simulator*, an *Evaluator*, and a *Decision Maker*. The portfolio scheduler is equipped with a set of policies; we describe our design of the portfolio in Section III-D. Periodically, the portfolio scheduler considers its constituent policies in simulation and selects from them the most promising policy. The selection is done by the Decision Maker, which uses the utility estimated by the Evaluator to rank descendingly the policies, then selects the best-ranked policy. This mechanism is versatile: different utility functions have been used to focus on performance [13], risk [18], and multi-criteria optimization [12].

2) Designing a Q-Learning-Based Approach: Inspired by the work proposed by Padala et al. [19], we design non-trivially a provisioning policy based on Q-learning. We define the *state* s_t at moment t as $s_t = (u_t, v_t, y_t)$, where u_t is the resource configuration (the total number of threads), v_t is the resource utilization, and y_t is the application performance. We define the *action* the scheduler can take as $a_t = (m, a)$, where m specifies the number of threads to be scaled and $a \in \{\text{up}, \text{down}, \text{none}\}$ is the action that grows, shrinks, and does nothing to change the provisioned resources, respectively. The *reward function* for the Q-learning policy is user-defined and used by the Q-learning algorithm to calculate the reward (the value) for the current state-action pair. In ANANKE, we design the reward function to balance workflow performance and resource usage based on previous study [19]. Specifically, for every moment of time t we define the reward function as:

$$r(t) = f(s_t, a_t) \times g(s_t, a_t), \qquad (1)$$

where $f(s_t, a_t)$ calculates the score due to workflow performance and $g(s_t, a_t)$ represents the score due to resource utilization. Higher scores indicating better user-experienced performance and resource utilization which are preferred. We define the concave functions $f(s_t, a_t)$ and $g(s_t, a_t)$ as:

$$f(s_t, a_t) = sgn(1 - y_t) \times e^{|1-y_t|}, \qquad (2)$$

$$g(s_t, a_t) = e^{1-max(v_t, p_t)}, \qquad (3)$$

where y_t is the normalized application performance according to the SLO, u_t is the number of threads, v_t is the ratio of busy to total number of threads (so, normalized by u_t), and p_t is the average CPU load across clients. We use sgn function to turn $f(s_t, a_t)$ to a negative value if the SLO is not meet (y_t exceeds 1). If y_t is less than 1, $f(s_t, a_t)$ is inversely proportional to y_t and the lower value of y_t is preferable.

3) Integrating Q-Learning into the Portfolio Scheduler: All learning techniques use a learning/training process. To train the Q-learning policy within a portfolio scheduler, the master node uses a simulation-based approach depicted in Figure 3. Our design supports online training through a mechanism that feeds back simulation data into a decision (learning) table. The values of this table are used by our Q-learning policy to make the decisions. To use the feedback from other policies as well, our Q-learning-based portfolio scheduler trains its decision table with information from *all* the policies, and from *both* real (applied decisions and real effects) and simulated (estimated decisions and effects) environments. Therefore, this method allows to generate and use more training data in a shorter time, albeit at the possible cost of accuracy (for simulated effects).

D. The Configuration of Policy Combinations

ANANKE is designed for workflow allocation and resource provisioning. However, it is not convenient to define these two behaviors in a single policy. For this reason, ANANKE maintains a policy pool (a portfolio) consisting of *combinations* of allocation and provisioning policies. An allocation policy contains a *workflow-selection policy*, which selects the workflow to schedule next, and a *client-selection policy*, which maps the selected workflows to client-nodes. The *provisioning policy* decides on adding and/or removing of the resources. A *composite-policy* is a triplet comprised of a provisioning, a workflow-selection, and a client-selection policies. At runtime, the portfolio scheduler selects from the pool a single combination of policies, as the active *composite-policy*. We design a portfolio comprised of all the unique composite policies (triplets) resulting from the policies described as following:

1) Provisioning Policies: Provisioning policies can vary significantly in how aggressively they change the resources. We select four significantly different provisioning policies and adapt them for using threads as the smallest computing unit. The On-Demand All (`ODA`) [12] policy always ensures that the system has enough *idle* resources for waiting workflows. The On-Demand Balance (`ODB`) [12] policy just ensures that the system has enough resources, whether busy or idle. The Average Queued Time Policy (`AQTP`) [20] only considers the demands of a subset of the waiting workflows. Our `Q-learning` [21] policy provides a trade-off between the other policies, by learning from the decisions made by them.

2) Allocation: Workflow-Selection Policies: Which workflow to select next for execution is a typical question in multi-workflow scheduling systems. For our portfolio, we choose four policies for selecting workflows from the waiting bucket: Last-Come First-Served (`LCFS`), Shortest Waiting time First (`SWF`), Closest to the Deadline First (`CDF`), and the Shortest Execution time First (`SEF`) policies. In particular, time-to-deadline is often used in real-time systems.

3) Allocation: Client-Selection Policies: To map workflows to available resources, we select four client-selection policies for our portfolio: Lowest CPU Utilisation First (`LUF`), Highest number of Idle Threads First (`HITF`), Lowest workflow Waiting-Time First (`LWTF`), and Smallest number of Waiting-Workflows First (`SWWF`).

IV. EXPERIMENTAL EVALUATION

We summarize our experimental setup and representative evaluation results. (Full details in our technical report [21].)

A. Experimental Setup

We create 4 synthetic workloads with periodic or uniform arrival patterns, matching the behavior of production workloads; we include *dynamic* arrival patterns (e.g., `ED.5x` in Figure 4) and *static* arrival patterns, (e.g., `PA3`). We conduct real-world experiments on the DAS-5 multi-cluster system, configured as a cloud environment using the existing DAS-5 capabilities [15], with up to 350 threads (5 nodes).

To quantify the user-oriented performance, we use *throughput*, the workflow *waiting time*, and the *expiration rate*. To evaluate the resource utilization we use the *number of active (used) threads*. A lower number of used threads in the system leads to piece-wise linearly lower operational costs.

To evaluate elasticity, we adopt the metrics and comparison approaches introduced in 2017 by the SPEC Cloud Group [11]. The elasticity metrics are based on the analysis of discrete supply and demand curves. The *supply* is the current number of threads. The *demand* is the current number of running workflows and waiting workflows which near the deadlines [21].

To analyze ANANKE's elasticity, we implement it and also a set of baselines with diverse elasticity capabilities:

1) `ANK-VH` (full ANANKE): Vertical and horizontal auto-scaling by the Q-learning-based portfolio scheduler.
2) `ANK-V` (partial ANANKE): Only vertical auto-scaling by the Q-learning-based portfolio scheduler.
3) `PS(-VR)` (standard portfolio scheduling): Vertical auto-scaling by the standard portfolio scheduler, and (`PS`) no autoscaling or (`PS-VR`) horizontal auto-scaling by the React policy [22]. React is a top-performing auto-scaler for horizontal elasticity and workflows [11].
4) `NoPS` (Chronos): Only vertical auto-scaling, by a threshold-based scaling policy.
5) `Static` (common in the industry): No auto-scaling, using only a fixed amount of client nodes.

B. Scheduler Impact on Workflow Performance

The main finding of the workflow performance evaluation can be summarized as follows: *compared with the standard portfolio scheduler, which may be adopted easily by the industry, under static workloads, the Q-learning-based portfolio scheduler leads to better user-oriented metrics.*

We conduct comparison experiments between ANANKE (`ANK-V`), the system with a standard portfolio scheduler (`PS`), and the system without a portfolio scheduler (`NoPS`). We report here only the expiration rate and the workflow response time as both the Q-learning-based portfolio scheduler (`ANK-V`) and the standard portfolio scheduler (`PS`) have very low expiration rates, much lower than the system without a portfolio scheduler.

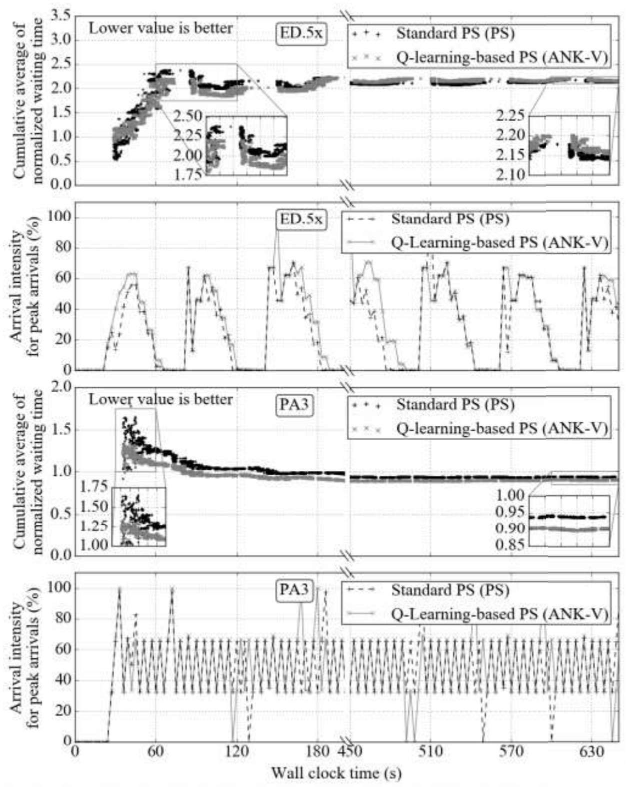

Figure 4: The normalized response time of the Q-learning-based portfolio scheduler (`ANK-V`) and of the standard portfolio scheduler (`PS`). Workload: (top pair of plots) dynamic (`ED.5x`), and (bottom pair of plots) static (`PA3`).

Notably, the `NoPS` has 1–5.5% of expired workflows under different workloads, while the system with a portfolio scheduler only has 0–1% of expired workflows. Though the portfolio scheduler configurations have a few expired workflows with the `ED.5x` workload, the average response time of the expired workflows are very close to the deadline. `NoPS` system fails to achieve deadlines for expired workflows for all the workloads, while both `ANK-V` and `PS` can fulfill the deadline-constrained SLOs. Overall, we cannot observe a significant difference between `ANK-V` and `PS` in the expiration rate and in the response time degradation.

Figure 4 depicts a deeper analysis of the performance. For this, we compare the normalized workflow waiting times achieved by `ANK-V` and `PS`. We normalize the workflow waiting time by the workflow execution time and calculate its cumulative average value; lower values indicate a better user-experienced performance. Figure 4 shows the performance of our schedulers and of `PS`, for static (`PA3`) and dynamic (`ED.5x`) workloads, and correlated sub-plots depicting the arrival of workflows. `ANK-V` reduces the normalized waiting time by 5–20% compared with `PS`, with a static workload. However, a similar performance improvement does not appear with the dynamic workload—the standard portfolio scheduler even performs slightly better, reducing the normalized waiting time by 0–8.3%. Because static workloads exhibit strong recurring patterns, `ANK-V` can make more precise scheduling decisions based on the information about previous workloads and system statuses. The results indicate that the learning technique can help the portfolio scheduler make better decisions, for workloads with strong recurring patterns.

C. Evaluation of Elasticity and Resource Utilization

The main finding of this evaluation is: *The Q-learning-based portfolio scheduler shows better elasticity results compared with the threshold-based auto-scaler common in today's practice (*`NoPS`*). Our horizontally and vertically elastic approach (*`ANK-VH`*) can save from 24% to 36% resources with at most 1.4% throughput degradation* (see technical report [21]).

Ideally, the supply should follow the envelope of the demand. We analyze the supply and demand curves for each auto-scaler under different workloads and find evidence that (see [21]): The Q-learning-based portfolio scheduler which uses only vertical scaling (`ANK-V`) performs better with dynamic workloads. The Q-learning-based portfolio with both horizontal and vertical scaling (`ANK-VH`) beats all the others when the workload is static. `PS-(VR)` often under-provisions and may cause serious performance degradation. `NoPS` and `Static`, on the contrary, significantly over-provision the resources and guarantee good user-experienced performance at the cost of many idle resources. However, according to the requirements, good performance for users with high resource costs is not our goal.

V. Related Work

We survey in this section a large body of related work. Relative to it, our work provides the first comprehensive study and real-world experimental evaluation of learning-based portfolio scheduling for managing tasks and resources.

Work on applying reinforcement learning: Closest to our work, Tesauro et al. [23] present a hybrid approach combining reinforcement-learning and queuing models for resource allocation. Their RL policy trains offline, while a queuing model policy controls the system. Our work uses online training, by taking advantage of the dynamic portfolio scheduling. Padala et al. [19] use reinforcement-learning to learn the behavior of applications and design a Q-learning solution to perform vertical scaling of VMs. While we use portfolio scheduling, scale finer-grained resources (threads vs. VMs), and take both horizontal and vertical scaling into account. Bu et al. [24] use reinforcement-learning to change the configuration of VMs and resident applications, whereas we utilize portfolio scheduling and address both resource allocation and workload scheduling problems.

Work on portfolio scheduling: Although the general technique emerged in finance over 50 years ago [9], portfolio scheduling has been adopted in cloud computing only in the past 5 years [13], [25]. Our current work extends a standard portfolio scheduler to support reinforcement learning and to the complex industrial workloads. Closest to our work, Kefeng et al. [12], [25] build a standard portfolio scheduler (without reinforcement learning) equipped only with threshold-based policies and only focus on scientific bags-of-tasks; and van Beek et al. [18] focus on different optimization metrics (for risk management) and workloads (business-critical, VM-based vs. job-based).

Work on general workflow-scheduling: This body of work includes thousands of different approaches and domains. Several scaling policies [26]–[28] take deadline constraints as their main SLO. In contrast, our work considers complex workflows, deals with resource provisioning, and presents real-word experimental results.

Work on auto-scaling in cloud-like settings: Marshall et al. [20] present many resource provisioning policies. We embed some of their policies as part of the portfolio used by ANANKE for auto-scaling, and in general extend their work through the Q-learning and portfolio scheduling structure. Ilyushkin et al. [11] propose a comparative study of a set of auto-scaling algorithms. We use their system- and user-oriented metrics to assess the performance of our auto-scaling approach, but consider different workloads.

VI. Conclusions and Future Work

Dynamic scheduling of complex industrial workflows is beneficial for companies migrating to clouds, but challenging. SLOs specific for complex industrial workflows are rarely addressed. Designing new scheduling policy is risky and ephemeral. In contrast, current state-of-the-art approaches using portfolio scheduling are promising, but do not take into account periodic effects that are common in workloads. To fill this gap, we have explored in this work the integration of a learning technique into a cloud-aware portfolio scheduler.

We have designed ANANKE, an architecture for RM&S that uses cloud resources for its operation and Q-learning as a learning technique. We further designed and selected various threshold-based heuristics as scheduling policies for the portfolio scheduler. We have implemented ANANKE as a prototype of the production environment at Shell, and conducted real-world experiments. We have also compared ANANKE with its state-of-the-art and state-of-practice alternatives, using real-world experiments and industry-derived workloads.

Our results show that the usage of the Q-learning policy in the portfolio scheduler allows getting better performance results from a user's perspective, improve the resource utilization, decrease operational costs in commercial IaaS clouds, and achieve good elasticity results for relatively static workloads. For highly dynamic workloads, the addition of Q-learning into a portfolio scheduler is less beneficial. In all our results, dynamic portfolio scheduling outperforms traditional static scheduling.

We are extending our work to better understand the interplay Q-learning/portfolio. We further plan to consider other reinforcement-learning techniques, such as error-driven learning and temporal difference learning. We are also focusing on an extended set of workloads and SLOs, and experiments at larger scale.

Acknowledgments

We thank our industry hosts, at Shell and CGI, and our academic hosts, at VU Amsterdam and TU Delft. Our work is supported by the Dutch STW/NWO personal grants Veni @large (11881) and Vidi MagnaData (14826), by the Dutch national program COMMIT and its funded project COMMissioner, and by the Dutch KIEM project KIESA.

References

[1] G. Baird *et al.*, "Upgraded online protection and prediction systems improve machinery health monitoring," *Asset Management & Maintenance Journal*, vol. 27, no. 2, p. 16, 2014.

[2] L. Yu and D. Thain, "Resource management for elastic cloud workflows," in *CCGrid*, 2012, pp. 775–780.

[3] S. Abrishami, M. Naghibzadeh, and D. H. J. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Comp. Syst.*, vol. 29, no. 1, pp. 158–169, 2013.

[4] L. Liu *et al.*, "A survey on workflow management and scheduling in cloud computing," in *CCGrid*, 2014, pp. 837–846.

[5] R. Cushing *et al.*, "Prediction-based auto-scaling of scientific workflows," in *(MGC*, 2011, p. 1.

[6] A. Ilyushkin and D. H. J. Epema, "Towards a realistic scheduler for mixed workloads with workflows," in *CCGrid*, 2015, pp. 753–756.

[7] Y. Ahn and Y. Kim, "Auto-scaling of virtual resources for scientific workflows on hybrid clouds," in *HPDC*, 2014, pp. 47–52.

[8] D. Klusácek and S. Tóth, "On interactions among scheduling policies: Finding efficient queue setup using high-resolution simulations," in *Euro-Par*, 2014, pp. 138–149.

[9] B. A. Huberman, R. M. Lukose, and T. Hogg, "An economics approach to hard computational problems," *Science*, vol. 275(5296), 1997.

[10] D. Villegas *et al.*, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," in *CCGrid*, 2012, pp. 612–619.

[11] A. Ilyushkin *et al.*, "An experimental performance evaluation of autoscaling algorithms for complex workflows," in *ACM/SPEC ICPE*, 2017.

[12] K. Deng *et al.*, "Exploring portfolio scheduling for long-term execution of scientific workloads in iaas clouds," in *SC*, 2013, pp. 55:1–55:12.

[13] O. Shai, E. Shmueli, and D. G. Feitelson, "Heuristics for resource matching in intel's compute farm," in *JSSPP*, 2013, pp. 116–135.

[14] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[15] H. E. Bal *et al.*, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *IEEE Computer*, vol. 49, no. 5, pp. 54–63, 2016.

[16] "Amazon web services (AWS)," https://aws.amazon.com.

[17] A. Iosup *et al.*, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE TPDS*, vol. 22, no. 6, pp. 931–945, 2011.

[18] V. van Beek *et al.*, "Self-expressive management of business-critical workloads in virtualized datacenters," *IEEE Computer*, vol. 48, no. 7, pp. 46–54, 2015.

[19] P. Padala *et al.*, "Scaling of cloud applications using machine learning," *VMware Technical Journal*, 2014.

[20] P. Marshall, H. M. Tufo, and K. Keahey, "Provisioning policies for elastic computing environments," in *IPDPS*, 2012, pp. 1085–1094.

[21] S. Ma *et al.*, "Ananke: a Q-Learning-Based Portfolio Scheduler for Complex Industrial Workflows: Extended Technical Report," TU Delft, Tech. Rep., DS-2017-001.

[22] T. C. Chieu *et al.*, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *ICEBE*, 2009, pp. 281–286.

[23] G. Tesauro *et al.*, "A hybrid reinforcement learning approach to autonomic resource allocation," in *ICAC*, 2006, pp. 65–73.

[24] X. Bu, J. Rao, and C. Xu, "Coordinated self-configuration of virtual machines and appliances using a model-free learning approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 4, pp. 681–690, 2013.

[25] K. Deng *et al.*, "A periodic portfolio scheduler for scientific computing in the data center," in *JSSPP*, 2013, pp. 156–176.

[26] M. Malawski *et al.*, "Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds," in *SC*, 2012, p. 22.

[27] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *SC*, 2011, pp. 49:1–49:12.

[28] J. Shi *et al.*, "Elastic resource provisioning for scientific workflow scheduling in cloud under budget and deadline constraints," *Cluster Comp.*, vol. 19(1), pp. 167–182, 2016.