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

A short, polished version of this technical report has been submitted for publication to ICAC 2017, and is currently awaiting review

Shenjun Ma Alexey Ilyushkin Alexander Stegehuis Alexandru Iosup

March 7, 2017

Abstract: Complex workflows that process sensor data are useful for industrial infrastructure management and diagnosis. Although running such workflows in clouds promises reduces 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. Reinforcement learning, based in this work on 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 (5–20%) and cost less (20–35%) than the considered alternatives.

Keywords: Portfolio Scheduling, Resource Management, Workflow Scheduling, Reinforcement Learning, and Auto-Scaling.

Contents

1 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 is often involves the usage of complex workflows designed to analyze real-time sensor data [4]. Although the management of workflows and resources has already been studied for decades [2, 13, 19, 33, 34], previous works have mostly focused on scientific workloads [3, 9, 15, 24] 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 [7, 18]. In contrast to previous body of work, in this paper 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 which can be represented as bags-of-tasks or sub-workflows. Each group (or stage) could have a predefined deadline, and could operate with certain performance requirements. Production workloads are often demonstrate notable recurrent patterns as some tasks could run periodically, e.g., when new data is acquired from sensors. Moreover, both the workloads and the 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 also trying to reduce the operational costs.

To fulfill dynamic SLOs and achieve cost savings, the cloud customer could employ dynamic scheduling techniques, such as *portfolio scheduling*. Derived from the economic field [14], a portfolio scheduler is equipped with a set of policies, each designed for a specific scheduling problem. The polices are selected dynamically, according to a user-defined rule and to a feedback machanism. 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 [28, 31], and, in contrast, that a portfolio scheduler performs well without external (in particular, manual) tuning [11, 12].

Although portfolio schedulers are promising for the context of complex workflows, previous approaches [26, 30] lack by design the ability to use historical knowledge in their selection. All these approaches prioritize the diversity of selection, and thus do not bias it towards approaches that have delivered good results in the past. This approach works well for workloads without periodic behavior, but may not deliver good results for industrial applications that focus on processing real-time sensor data. Thus, to take advantage of historical information about the system and the workload, a research question arises in the context of complex industrial workflows: *How to integrate learning techniques into portfolio scheduling?*.

To answer the research question, we propose in this work to integrate a reinforcement-learning technique, Q-learning [32], into a cloud-aware portfolio scheduler. A Q-learning algorithm interacts with a system by applying an action to it, and learns about the merit of the action from the system's feedback (reward). We explore the strengths and limitations of a Q-learning-based portfolio scheduler managing diverse industrial workflows and cloud resources. Towards answering the research question, our main contribution is threefold:

1. We design ANANKE, an RM&S architecture that integrates a reinforcement-learning technique, Q-learning, into a portfolio scheduler (Section 3). This enables portfolio schedulers operating in cloud environments to use historical information, and thus service periodic workloads.

2. We design and build a prototype of ANANKE, a scheduling system with a learning-based portfolio scheduler as its core element (Section 4). The key conceptual contribution of this design is the selection and design of scheduling policies equipped by the portfolio. The prototype is now part of the production environment at Shell, and evolves from the existing *Chronos* system [4].

3. We evaluate our learning-based portfolio scheduler through real world experiments (Section 6). Using the cloud-like experimental environment DAS-5 [5] and workloads derived from a real industrial workflow, we analyze ANANKE's user-level and system-level performance, and elasticity (metrics defined in Section 5). We also compare ANANKE with a baseline system and with a portfolio-scheduling-only approach.

2

System Model

In this section, we define the system model used in this work: workload, system architecture, and system infrastrucure. In practice, this model is already commonly used in real-time infrastructure monitoring systems, such as the Chronos [4] system in the "Smart Connect" project at Shell.

2.1. 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 among them. Each workflow is aimed for processing sensor data and has exactly 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. All the workflows in the model contain these three steps and differ only in the formula used to calculate.

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., CPU core or thread) for its execution.

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

In practice, infrastructure monitoring systems commonly use a tree-tier architecture to process sensor data. The three-tier architecture, which we depict in Figure 2.1, 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 job bucket, which is maintained by the workload manager (b) withing the master node. The client manager (c) controls the set of client nodes and monitors their statuses. At the heart of the architecture, the *scheduler* (a) is responsible for making allocation and provisioning decisions. The scheduler selects appropriate workflows from the workload manager and, through the client manager, allocates them to the client nodes.

The client nodes are responsible for running workflows. 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 job bucket. To display the states of the monitored infrastructure and the diagnostics results back to the users the framework has a web interface.

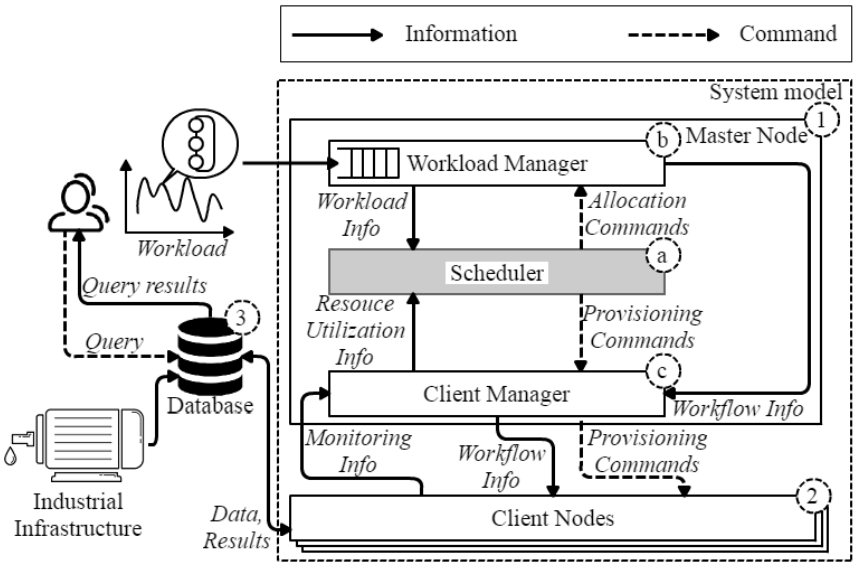

Figure 2.1: Three-tier architecture for processing workflows.

2.3. Infrastructure: Cloud-Computing Resources

We model the infrastructure as an infrastructure-as-a-service (IaaS) cloud environment, either public or private. (The Chronos system is currently deployed in a private cloud.) In this work, we assume that all resources are homogeneous, and do not consider hybrid private-public cloud scenarios. In contrast with typical cloud resource models which usually operate on a per-VM basis, our model uses the *computing thread* 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.

Compared to popular public clouds, our resource model has certain differences. The resources for our experiments are only on-demand instances, and not spot or reserved instances. In public clouds such as Amazon AWS [1], instances take some time to fully boot up [17]. However, to have a better control over the emulated environment, we use in practice preallocated nodes (zero-time booting) and do not consider node booting times in our model. 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 [31] we use in our model the total running time of all the active instances to represent the *actual resource cost* and not the charged cost.

3

ANANKE Requirements and Design

In this section, we present the design of our ANANKE system. First, we define the architectural requirements and specify the design goals. Then, we explain all the components of the system and discuss the design of our Q-learning-based portfolio scheduler. We further show how to integrate the scheduler into the architecture introduced in Section 2.2.

3.1. Architectural Requirements and Design Goals

The major architectural requirements (design goals) for ANANKE are:

1. The designed system must match the model proposed in Section 2. This allows the new system to be backwards compatible with the system currently in operation, and enables adoption in practice.
2. The system must implement elastic functionality, e.g., it must be able to automatically adjust the amount of allocated resources based on demand changes.
3. The system should use portfolio scheduling, which shows promise in managing mixed complex workloads [11].
4. The system should integrate Q-learning into the portfolio scheduler, to be able 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.

3.2. Architecture Overview

ANANKE extends the current Chronos system with the components and concepts needed to achieve the design goals 2–4. Matching the model from Section 2 (goal 1), ANANKE has a three-tier architecture, and consists of a master node, client nodes, and a database. We further present the design of the ANANKE master and client nodes, focusing on the new aspects.

3.2.1. Master Node

The master node of ANANKE consists of three major components: a scheduler, a workload manager, and a client manager. Figure 2.1 depicts these components, and their communication pathways. Addressing design goals 3 and 4, the *scheduler* (component (a) in Figure 2.1) is a *Q-learning-based portfolio scheduler* equipped with a set of policies: a *Q-learning policy* and also other, simpler, threshold-based heuristic policies. As detailed in Section 3.3, the scheduler hides the complexity of selecting the right policy, by autonomously managing its set of policies and by taking decisions periodically. 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.

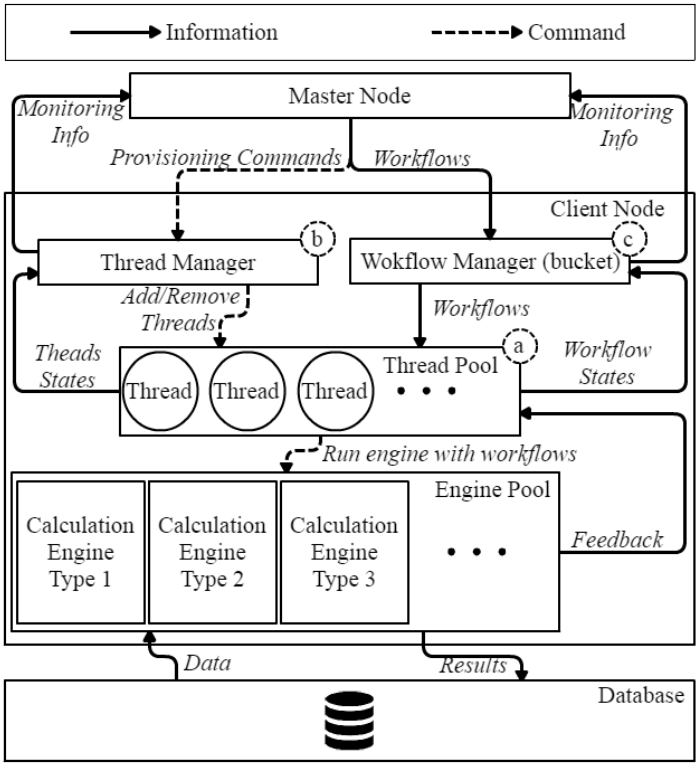

Figure 3.1: Components of the client node.

3.2.2. Client Nodes

The actual calculations are performed on client nodes. Figure 3.1 depicts the three components of the client node: a thread pool, a thread manager, and a task manager. The *thread pool* (component (a) in Figure 3.1) maintains a set of threads (smallest working units, see Section 2.3). Each thread continuously fetches workflows from the task manager and calls an external calculation engine from the engine pool. The thread logic is completely determined by the workflow tasks and is defined by developers. 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 and removes threads from the pool. It also reports the resource utilization (the CPU load, and the ratio of busy to 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 for our scheduler: the number of workflow stored in the bucket normalized by the size of the bucket and the average completion time for the last five executed workflows.

3.3. The Q-Learning-Based Portfolio Scheduler

Addressing design goals 3 and 4, in this section we design a Q-learning-based portfolio scheduler for complex industrial workflows in cloud (elastic) environments.

3.3.1. Adding a Portfolio Scheduler to the Architecture

Our design of the scheduler component of the master node is based on a portfolio scheduler. Figure 3.2 depicts the main components of this design: similarly to previous work [12], the portfolio scheduler consists of a policy *Simulator*, an *Evaluation* component, and a *Decision Maker*. The portfolio scheduler is equipped with a set (portfolio) of policies; we describe our design of the portfolio in Section 4.1. 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 Evaluation component to rank descendingly the policies, then selects the best-ranked policy. This mechanism is versatile:

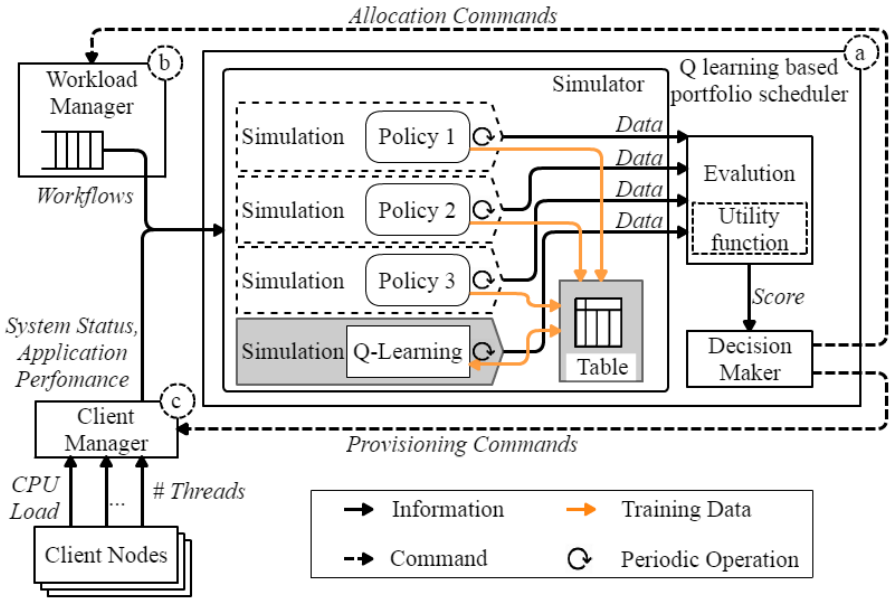

Figure 3.2: Architecture of a Q-learning-based portfolio scheduler, part of the master node. Data generated by each simulation is used as training data, to train the decision table.

different Utility functions have been used to focus on performance [12, 26], risk [30], and multi-criteria optimization [11, 30].

3.3.2. Designing a Q-Learning-Based Approach

Inspired by the work proposed by Padala et al. [25], we design non-trivially a provisioning policy based on Q-learning. We define the *state* s_t at moment t as a tuple of the current resource configuration, resource utilization, and the workflow performance: $s_t = (u_t, v_t, y_t)$, 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 further 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 set of provisioned resources, respectively. The *reward function* for the Q-learning policy is defined by user, 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 [25]. Specifically, for every moment of time t we define the reward function as:

$$r(t) = f(s_t, a_t) \times g(s_t, a_t), \tag{3.1}$$

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:

$$f(s_t, a_t) = sgn(1 - p_t) \times e^{|1-p_t|}, \tag{3.2}$$

$$g(s_t, a_t) = e^{1-max(v_t, y_t)}, \tag{3.3}$$

where p_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 y_t is the average CPU load across clients.

When calculate the score for workflow performance, we use p_t (average value of workflow's response time normalized by deadline in this work) as the main metric. According to the formula $f(s_t, a_t)$, smaller value of p_t leads to a higher score. A lower response time indicates that our system performs well in allocating workflows. To calculate the score for resource utilization, we decide to use v_t and y_t. Formula $g(s_t, a_t)$ shows that lower value of v_t and y_t can lead to a higher score. In a word, we would like to observer that our system can use less resource to process the workflows. By combining the workflow performance and resource utilization, the reword function can balance the trade off between user-experienced performance and resource utilization.

3.3.3. Integrating Q-Learning into the Portfolio Scheduler

All learning techniques use a learning and training process. Because ANANKE is a real-time system, online training is more suitable for our case than offline training. In our design, to train the Q-learning policy within a portfolio scheduler, the master node uses the simulation-based approach depicted in Figure 3.2. Compared to a standard portfolio scheduler, our design supports online training through a mechanism that feeds-back simulation data into a decision (learning) table. The Q-learning policy uses the updated decision table in its own decisions. However, using only the feed-back generated by applying the decisions of the Q-learning policy itself ignores that other policies take decisions. To avoid this problem, our Q-learning-based portfolio scheduler train its decision table with information from *all* 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).

Table 4.1: Provisioning policies in our portfolio scheduler.

Name	Operational Principle
AQTP	Lease threads for the first n waiting workflows, if their waiting time exceeds a pre-set threshold t.
ODA	Lease n threads for waiting workflows in the bucket, until the resource maximum is reached, with $n = n_w - n_i$, where n_w is the number of waiting tasks and n_i is the number of idle threads.
ODB	Lease n threads for waiting workflows in the bucket until the resource maximum is reached, with $n = n_w - n_t$, where n_w is the number of waiting tasks and n_t is the total number of threads.
Q-Learning	Make a provisioning decision according to the current status of the system, by retrieving an action from the decision table.

4

The Implementation of the ANANKE Prototype

Two ANANKE components require careful configuration and implementation, the Q-learning-based portfolio scheduler and the Q-learning policy, respectively. We explain in this section the selection of the portfolio policies, which we see as a conceptual contribution. We also detail the process for constructing the decision table for the Q-learning policy, which we see as an important technical contribution.

4.1. The Configuration of Policy Combinations

ANANKE is designed for both workflow allocation and resource provisioning. However, it is difficult to define the allocation and provisioning behavior 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 combination of allocation and provisioning policies, that 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:

4.1.1. Provisioning Policies

Provisioning policies can vary significantly in how aggressively they change the amount of resources. We select four significantly different provisioning policies, adapt them to use threads as smallest processing unit, and summarize their operation in Table 4.1. ODA [11] is the most aggressive in the set, as it always ensures

Table 4.2: Workflow-selection policies in our portfolio scheduler.

Name	Operational Principle
LCFS	Last-Come, First-Served.
SWF	Workflow with the Shortest Waiting time First.
CDF	Workflow which is Closest to its Deadline First.
SEF	Workflow with the Shortest Execution time First.

Table 4.3: Client-selection policies in our portfolio scheduler.

Name	Operational Principle
LWTF	A client with the Lowest workflow Waiting Time First.
LUF	A client with the Lowest CPU Utilization First.
HITF	A client with the Highest number of Idle Threads First.
SWWF	A client with the Smallest number of Waiting Workflows First.

that the system has enough *idle* resources for waiting workflows. ODB [11] is less aggressive, as it just ensures that the system has enough resources, whether busy or idle. AQTP [23] is the least aggressive policy in the set, because it only considers the needs of a subset of the waiting workflows. Last, our Q-learning policy provides a trade-off between the other policies, by learning from the decisions made by them.

4.1.2. Allocation: Workflow-Selection Policies

Which workflow to select next for execution is a typical question in multi-workflow scheduling systems. We select four workflow-selection policies for our portfolio, and summarize their operation in Table 4.2. Besides the typical LCFS policy, the other policies we use waiting time, time-to-deadline, or execution time as criteria to select workflows from the waiting bucket. In particular, time-to-deadline is often used in real-time systems.

4.1.3. Allocation: Client-Selection Policies

To map workflows to available resources, we select four client-selection policies for our portfolio and summarize their operation in Table 4.3. LUF and HITF use two different *system-oriented* metrics (the CPU usage and the number of idle threads) to find the "most idle" client node. In contrast, LWTF and SWWF sort the client nodes according to *user-oriented* metrics. LWTF allocates workflows to the client nodes that can start processing the workflows the earliest. SWWF allocates workflows to the client nodes which have the least workflows in their buckets.

4.2. Operational flow for the selecting the combination of policies

During the execution of a workload, the appropriate triplets of policies are chosen by the scheduler automatically and output the selected one for next step (the actual provisioning and allocation). In a single decision-making iteration, the portfolio scheduler selects the combination of policies by applying the following steps in sequence. First, the scheduler prepares the virtual environments for simulation. It uses the current system state to build the virtual environments. Note, that the scheduler runs simulations for each triplet of the policies in an isolated virtual environment. In this work, 48 same virtual environments are created at the beginning of the simulation. Then the scheduler clones the current workload and loads the triplet of policies into each virtual environment. As shown in Figure 4.1, the first three steps are the preparation for the simulation. During the simulation, the equipped policies tuple are periodically applied until the workload is

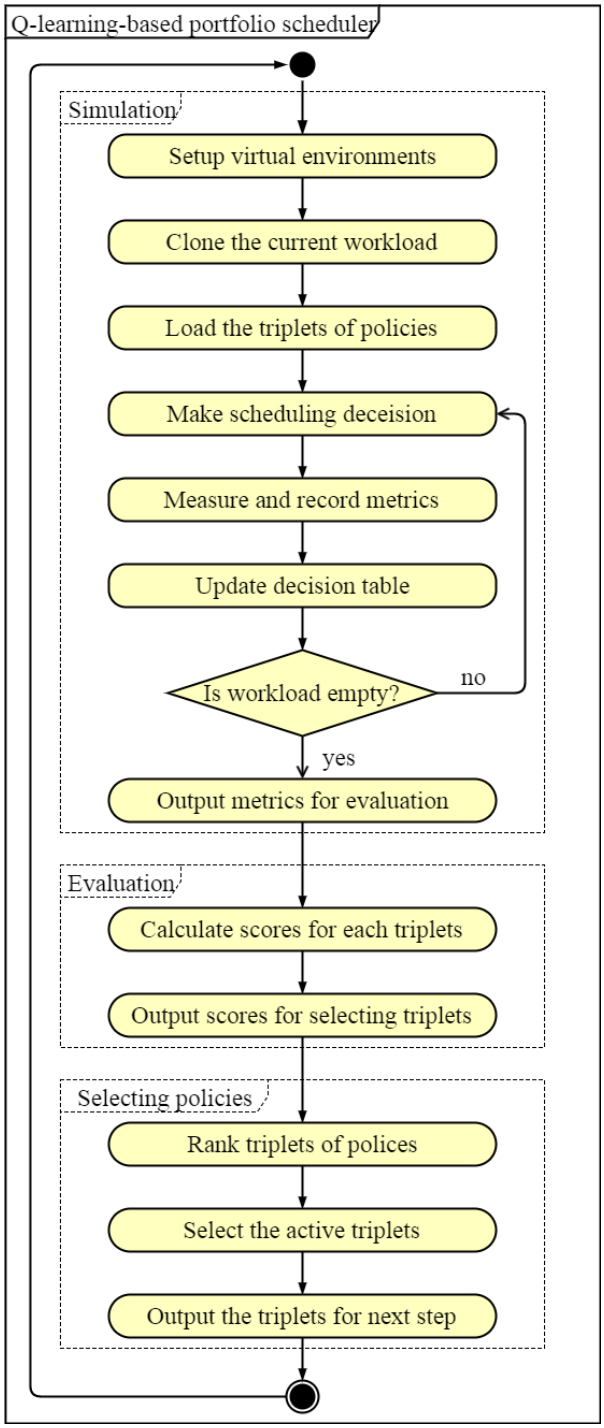

Figure 4.1: The operational flow for simulation, evaluation and policies selecting.

consumed in each virtual environment. Metrics are also measured and recorded which are used to evaluate different triplets in the evaluation step. In the evaluation, a score is calculated for each triplet based on the recorded metrics. A user-defined utility function (explained in section 4.3) is used for the calculation. The decision maker, therefore, ranks the triplets discerningly according the scores calculated by the evaluation component. The best-ranked triplet is selected. The selected triplet is used for actual resource provisioning and workflows allocation. Figure 4.2 shows how the active triplet changes during the experiment.

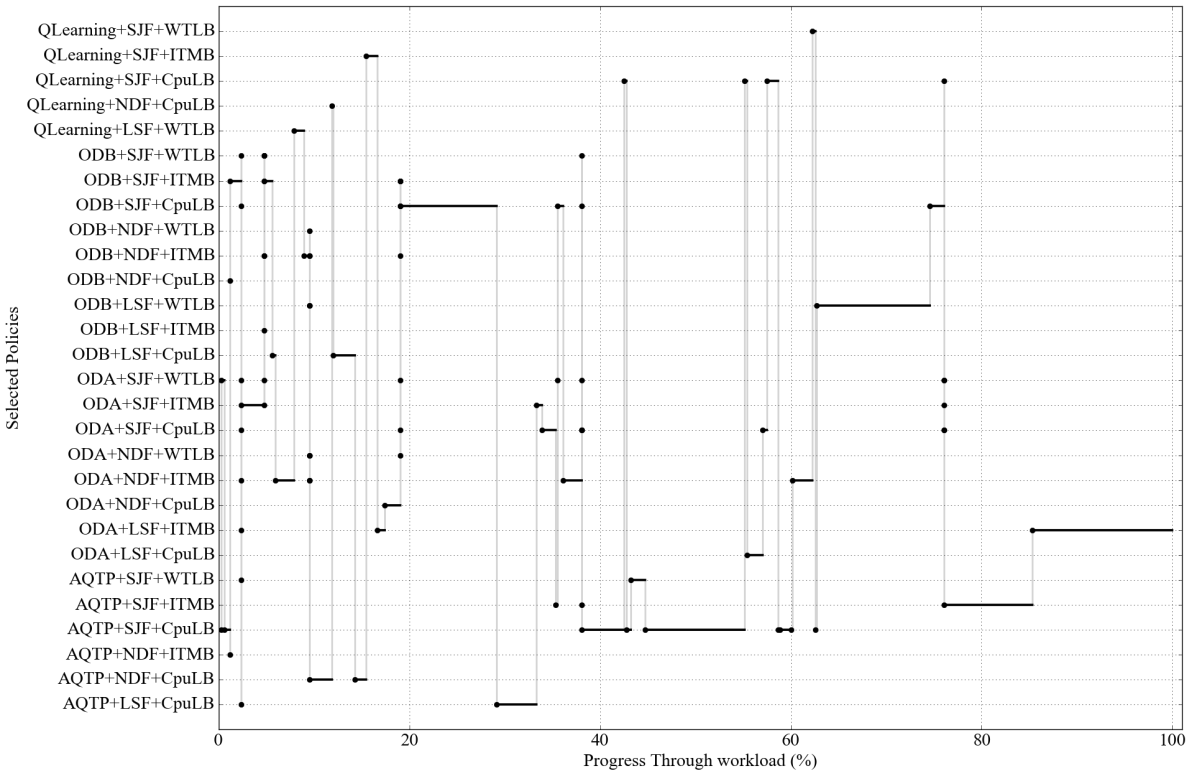

Figure 4.2: The figure gives an example of how the active triplet switches during the experiment.

4.3. Utility function as selection criteria

Users can customize utility functions used as selection criteria by the portfolio scheduler. The utility function is a formula which considers both user-oriented metrics and system-oriented metrics. In this project, our system operates in a private cloud environment. One of our goals is to consume the resources in the most efficient way. Thus we focus on task throughput and CPU utilization which are combined in the following formula:

$$S = k \times (\frac{P}{P_{max}})^{\alpha} \times (\frac{1}{1+W_{avg}})^{\alpha} \times (\frac{C}{C_{max}})^{\beta} \tag{4.1}$$

k, α, and β are scaling factors. P is the workflow throughput. W_{avg} is the average workflow waiting time. C is the CPU usage. α and β are used to balance these three metrics. Table 4.4 gives the definitions of each notation used in the utility function. The *throughput* is a standard metric which presents the capability of processing tasks. Although a higher throughput is usually desired, it may result in higher resource costs. Therefore, we also consider the CPU cost when evaluating each pair of policies. Since we have the requirement to finish each workflow before its deadline, we need to additionally consider the waiting time of a workflow to have control over the response time.

4.4. Implementation of the Decision Table

The key question when implementing the Q-learning policy is how to match actions and states. For this, we use a decision table storing in its cells state-action weights, which are dynamically updated by the Q-learning policy. Figure 4.3 depicts an example of this table, with the states and actions defined as described in Section 3.3.2. The size of the decision table is determined by the size of the action-state space (e.g., if the system has 10 actions and 10 possible states, the size of the decision table is 10×10). To limit the size of the table (otherwise, the training may take infinite time), the values of u_t, v_t, y_t are normalized between 0 to 1 and discretized (e.g., 10 possible steps), and m is given a maximum value that limits the number of columns in the table. Before the training process starts, all the state-action weights are initialized by zero. When making a decision at time t, our policy finds the row which represents the current state s_t and then, within the cell values of that row, applies the provisioning action a_t with the highest weight. The next moment, $t+1$, the scheduler updates the weight q of that cell to a new weight q':

Table 4.4: Explanation for the symbols used in utility function.

Symbol	Description
k	k is the scaling factor for the total score whose value should in $[1, 10^n]$. (n is the amount of metrics considered in the objective function)
α	α is used to emphasize the urgency of the tasks. (α is user defined)
β	β is applied to stress the efficiency of resource usage. (β is user defined)
P	Task throughput in one single simulation.
P_{max}	Estimated maximum value of task throughput in one single simulation.
W_{avg}	Estimated average of task waiting time.
C	Integration of CPU usage during one single simulation.
C_{max}	Estimated maximum value of integration of CPU usage during one single simulation.

Figure 4.3: The decision table for the Q-learning policy. Columns represent actions and rows represent states of the environment.

$$q'(s_t, a_t) = q(s_t, a_t) + \alpha(r_{t+1} + \beta h - q(s_t, a_t)), \tag{4.2}$$

$$h = \max_a q(s_{t+1}, a), \tag{4.3}$$

where α and β are the learning rate and the discount factor, respectively, r_{t+1} is the reward caused by state-change, and h is the estimate of the optimal weight.

Table 5.1: The parametrization of synthetic workflows.

Synthetic workflow	Execution time range (s)	CPU usage (%)
Workflow 1	[5, 15]	20
Workflow 2	[5, 10]	20
Workflow 3	[5, 10]	15
Workflow 4	[5, 15]	5
Workflow 5	[10, 15]	15
Workflow 6	[10, 15]	5

5 Experiment Setup

In this section, we present our real-world experimental setup: in turn, the workloads, the resource configuration, and the baselines used to compare with ANANKE.

5.1. Workload Settings

To measure and analyze the performance of the Q-learning-based portfolio scheduler in different conditions, we generate synthetic workloads that emulate the statistical features of real workloads. (We cannot use the real workloads from the Chronos production environment, due to the confidentiality agreements.) Each workload is comprised of a set of workflows and an arrival process (pattern).

Individual workflows: After analyzing the original Chronos workloads, we create six different types of synthetic workflows and parametrize them similarly to the real-world cases. As summarized by Table 5.1, each workflow differs in execution time and CPU usage, rated on a baseline client node.

Complete workloads: We create four synthetic workloads with periodic or uniform arrival patterns, matching the behavior observed in real-world workloads (i.e., in production Chronos workloads). Each synthetic workload combines the six different types of synthetic workflows, using an uniformly random distribution to select the type for each arrival. Figure 5.1 shows the recurrent patterns of workflow arrival in the synthetic workloads. For the workloads with periodic (*dynamic*) arrival patterns, `ED.5x` and `EI2x`, the workload consists of a sequential batch submission with 10 batches/minute, where, respectively, the number of workflows in a batch exponentially decreases by 0.5 and increases by 2 in geometric progression. For the workloads with uniform (*static*) arrival patterns, `PA3` and `PA6`, the arrival rate is of one workflow every 3 and 6 seconds, respectively.

5.2. Environment Configuration

Real-world infrastructure: We conduct real-world experiments with ANANKE on the DAS-5 multi-cluster system, configured as a cloud environment using the existing DAS-5 capabilities [5]. For our experiments, we use 1 cluster of the 6 available in DAS-5, and up to 50 homogeneous nodes of the 68 available nodes in this cluster. Each node has Intel E5-2630v3 2.4GHz CPUs and 64 GB of RAM; nodes are interconnected with 1 Gbit/s Ethernet links (conservatively, we do not use the existing high-speed FDR InfiniBand links).

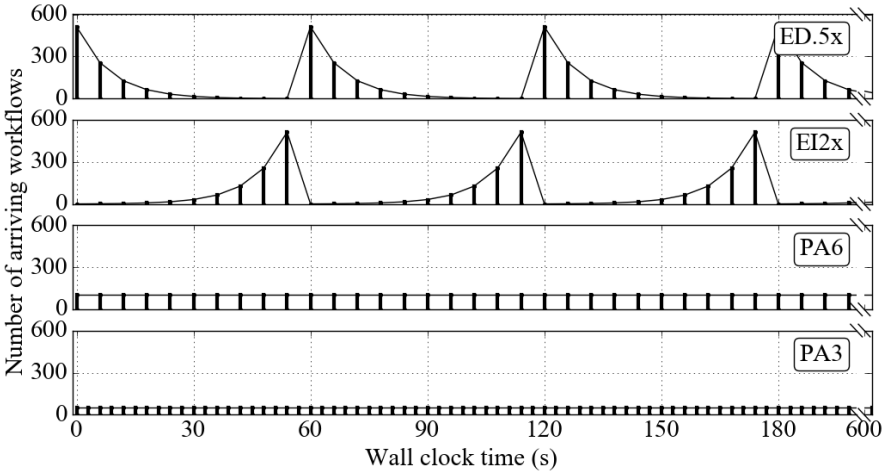

Figure 5.1: Four different patterns: ED.5x and EI2x are dynamic workloads, PA6 and PA3 are static workloads. (The horizontal axis only shows a 190 s-cut from every complete workload.)

Cloud-deployment models: We conduct experiments using 3 cloud-deployment models, each of which uses one master node, but different amounts and management of client nodes. The *private cloud mode* uses 3 statically allocated client nodes. This mode emulates the current production environment of the Chronos system and represents standard practice in the industry. The *public cloud mode* allows changing the number of client nodes during the experiment, from 1 to 5. Using this configuration, we evaluate the elasticity of our scheduler. The *scalability mode* allows changing the number of client nodes in a wider range, from 5 to 50, allowing us to conduct experiments for the what-if scenario in which the system load would increase by an order of magnitude.

Configuration of Ananke *components:* The master node and client nodes are deployed in DAS-5. We benchmark the client nodes, and determine that each client node can maintain up to 70 threads. Correspondingly, we set the maximal bucket size on a client node to 140, which means the client node can accumulate (queue) load for twice its rated capacity.

5.3. Metrics to Compare ANANKE and Its Alternatives

To analyze ANANKE and its alternatives, we use a variety of operational metrics focusing on application performance, on resource utilization, and on elasticity.

5.3.1. Application Performance

To quantify application performance, which is a user-oriented performance view, we use *throughput*, the workflow *waiting time*, and the *expiration rate*. We define throughput as the average number of completed workflows per second. The waiting time of a workflow is the time between its arrival and the start of its first task and the runtime is the time between the start of its first task and the completion of its last. We look at *slowdown in workflow response time*, which for a workflow is the fraction between the runtime, and the sum between the runtime and the wait time of the workflow. The expiration rate metric is the fraction of failed workflows, that is, workflows that did not complete before their deadlines, from the total number of eligible workflows during the entire experiment.

5.3.2. Resource Utilization

We use as metric the *number of active (used) threads*. Because each client node can run up to 70 threads, a lower amount of threads (busy and idle) in the system leads to piece-wise linearly lower operational costs.

5.3.3. Elasticity

To evaluate elasticity, we adopt the metrics and comparison approaches introduced in 2017 by the SPEC Cloud Group [16].

Supply and demand curves: The core elasticity metrics are based on the analysis of discrete supply and demand curves. In this project, we define *supply* as the current number of threads in the system. We define

the *demand* as the current number of running and waiting workflows, and when computing demand we only consider those workflows near their deadlines [20].

Elasticity metrics: We adopt the suite of metrics proposed by the SPEC Cloud Group [16], each of which characterizes a different facet of the mismatch between supply and demand of resources. We use three classes of system-oriented elasticity metrics, related to accuracy, duration of incorrect provisioning, and instability caused by elasticity decisions. The SPEC Cloud Group defines two accuracy metrics: the *under-provisioning accuracy*, defined as the average fraction by which the demand exceeds the supply; and *over-provisioning accuracy*, defined as the average fraction by which the supply exceeds the demand. The *Wrong-provisioning Timeshare* represents the fraction of time of periods with inaccurate provisioning, either under- or over-provisioning, from the total duration of the observation. The *Instability* represents situations when the supply and demand curves do not change with the same speed, and is defined as the fraction of time the supply and demand curves move in opposite directions or move towards each other.

Comparing multiple auto-scalers: To perform a comprehensive comparison including all system- and user-oriented metrics, we use the two numerical approaches proposed for use by the SPEC Cloud Group [16]: *Pairwise Comparison* [10] and the *Fractional Difference Comparison* [16]. In the pairwise comparison, for each auto-scaler we compare the value of each metric with the value of the same metric of all the other auto-scalers. The auto-scaler which has better performance for a particular comparison gains 1 point. If two auto-scalers perform equally well for the same metric, both earn a half-point. Finally, auto-scalers are ranked by the sum of points they have accumulated through pairwise comparisons. For the fractional difference comparison, from all the obtained results we construct an ideal system, that for each metric is ascribed the the best performance observed among the compared systems. (The ideal system expresses a pragmatic ideal that may be closer to what is achievable in practice than computation based on theoretical peaks, and likely does not exist in practice.) Then, we compare each auto-scaler with the ideal case, and accumulate for each metric the fractional difference between; the lower the difference, the closer the real auto-scaler is to the ideal. Last, we rank auto-scalers inversely, such that the lowest accumulated value indicates the best auto-scaler.

5.4. Auto-scalers Considered for Comparative Evaluation

We compare experimentally ANANKE with the following alternatives:

5.4.1. Existing Baseline

As baseline for ANANKE, we experiment with the current Chronos system, which is production-ready, does not meet design goals 3–4, and operates in a private cloud.

5.4.2. Elasticity Baselines

Elasticity can be achieved by both vertical scaling, that is, adding or removing threads on existing client nodes, and horizontal scaling, that is, adding or removing client nodes. Unlike ANANKE, which is elastic both horizontally and vertically, Chronos is only horizontally elastic. To better assess 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 [11]): Vertical auto-scaling by the standard portfolio scheduler, and (`PS`) no autoscaling or (`PS-VR`) horizontal auto-scaling by the React policy [8]. React is a top-performing auto-scaler for horizontal elasticity and workflows [16].
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.

5.4.3. Decision Table Configuration Alternatives

The decision table should be well-trained for the Q-learning policy to make accurate decisions. Intuitively, the bigger the size of the decision table the longer it takes to fill it with data. The large size of the table increases the training duration. Moreover, the size of the policy pool also affects the training process. A small number of policies makes the training process longer as it generates less training data during the simulation. Thus, it

Table 5.2: The policy pool configurations.

# Policies	Provisioning Policy	Workflow Selection	Client Selection
16	AQTP, ODA, ODB, QL	LCFS	LUF, HITF, LWTF, SWWF
12	AQTP, ODA, ODB, QL	LCFS	LUF, HITF, LWTF
9	AQTP, ODB, QL	LCFS	LUF, HITF, LWTF
8	AQTP, ODA, ODB, QL	LCFS	LUF, HITF
6	AQTP, ODB, QL	LCFS	LUF, HITF
4	AQTP, ODA, ODB, QL	LCFS	HITF
3	AQTP, ODB, QL	LCFS	HITF

is highly possible that a poorly trained policy can cause performance degradation. The size of decision table and the size of the policy pool are the most important factors affecting the performance of a Q-learning-based policy.

As mentioned in Section 2, the Q-learning policy uses a decision table to store and manage weights of the action-state pairs. The size of the decision table is determined by the number of possible system states and actions. Figure 4.3 shows an internal structure of the decision table maintained by the Q-learning-based portfolio scheduler. Our experiments compare the following configurations: different number of considered actions, a different number of considered system states, and different sizes of the policy pool. All the related experiments are performed in the private cloud environment. To evaluate the performance in this setup we use the workflow waiting time normalized by the execution time. In this work, we design

1. **State having more information vs State having less information**: The *State* function uses three metrics u_t, v_t, and y_t. By definition, v_t and y_t have boundaries, and u_t can take any positive integer value or 0. To define a range for u_t, we normalize it by the maximal number of threads n_{max}. The normalized configuration guarantees that the decision table will converge after a finite number of iterations. Further we call the configuration with normalized u_t as the state with less information (LD) and the opposite configuration as the state with more information (MD).

2. **10 as the maximum action value vs 2^n as the maximum action value**: More threads may be leased or terminated when the resource increases. As mentioned in Section 3.3.2, the *action* is defined as a_t=$(m, up|down|none)$. It may take a large value of if the number of threads (n_{max}) becomes large. We designed two approaches to determine the value of m which avoid it to increase linearly with n_{max}. The first approach is giving m a maximum value which is 10. It scales at most ten threads in a single scaling operation. The second approach is using Equation 5.1 to restrict the value of m in the case that large resource is needed to be scaled.

$$k = \begin{cases} k, & \text{for } k < 10 \\ 10, & \text{for } 10 \leq k < 2^4 \\ 2^i, & \text{for } 2^i \leq k < 2^{i+1} \ (i \geq 4) \end{cases} \tag{5.1}$$

3. **Setting for policy size**: The decision table is solely trained by the data generated during the simulation. Since, as mentioned in Section 2, the simulation is based on policies, the more policies the scheduler has, the more data is generated. At the beginning of this section, we also mentioned that the lack of training data might lead to a poorly trained Q-learning policy. Based on these assumptions, we investigate the dependency between the size of the policy pool and the application performance. Table 5.2 demonstrates how the policy pool is constructed in this experiment. Note, that the usage of different workflow selection policies may cause fluctuations in the workflow waiting time. Thus, to minimize possible side effects, we only use LCFS for the workflow selection.

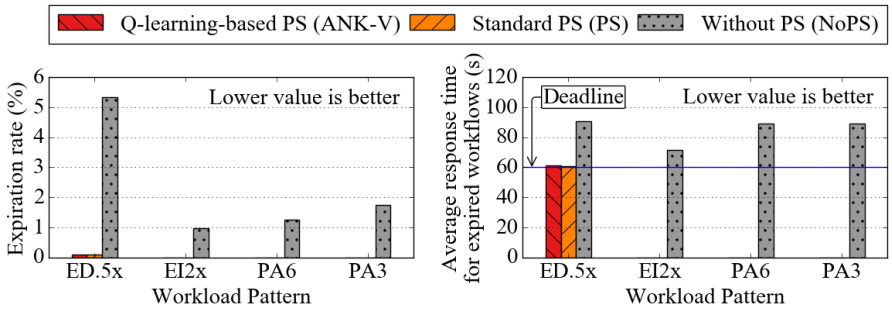

Figure 6.1: The expiration rate and the response time of expired workflows with different workloads. (Lower values are better.)

6

Experimental Results

In this section, we evaluate and validate the design choices we made for ANANKE (in Section 3). As mentioned in Section 5, we use two dynamic and two static workloads. Overall, our main experimental findings are:

1. 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.
2. Compared with the standard portfolio scheduler, which may be adopted easily by the industry, under static workloads the Q-learning-based portfolio scheduler has better user-oriented metrics.
3. Compared with the experimental `PS-VR`, the Q-learning-based portfolio scheduler reduces the performance degradation due to elasticity, and for our largest experiments it outperforms `PS-VR`, but for small experiments it shows worse overall elasticity performance when not tuned.
4. The Q-learning-based scheduler can be tuned to achieve a wide range of performance and elasticity goals.

6.1. Scheduler Impact on Workflow Performance

We conduct experiments for this part using the *private cloud mode*, and report here only the expiration rate and the workflow waiting time as the main metrics indicating application performance.

Figure 6.1 depicts the results measured for ANANKE (`ANK-V`), for the system using a standard portfolio scheduler (`PS`), and for the system without a portfolio scheduler (`NoPS`). Both the Q-learning-based portfolio scheduler (`ANK-V`) and the standard portfolio scheduler (`PS`) have very low expiration rates, much better than the system without a portfolio scheduler. The portfolio scheduler leads to a few expired tasks, with the `ED.5x` workload, but even then the average response time of the expired workflows is closer to the deadline (indicated in the figure). Accordingly, both the Q-learning-based portfolio scheduler and the standard portfolio scheduler can fulfill the deadline-constrained SLOs. Overall, we cannot observe a significant difference

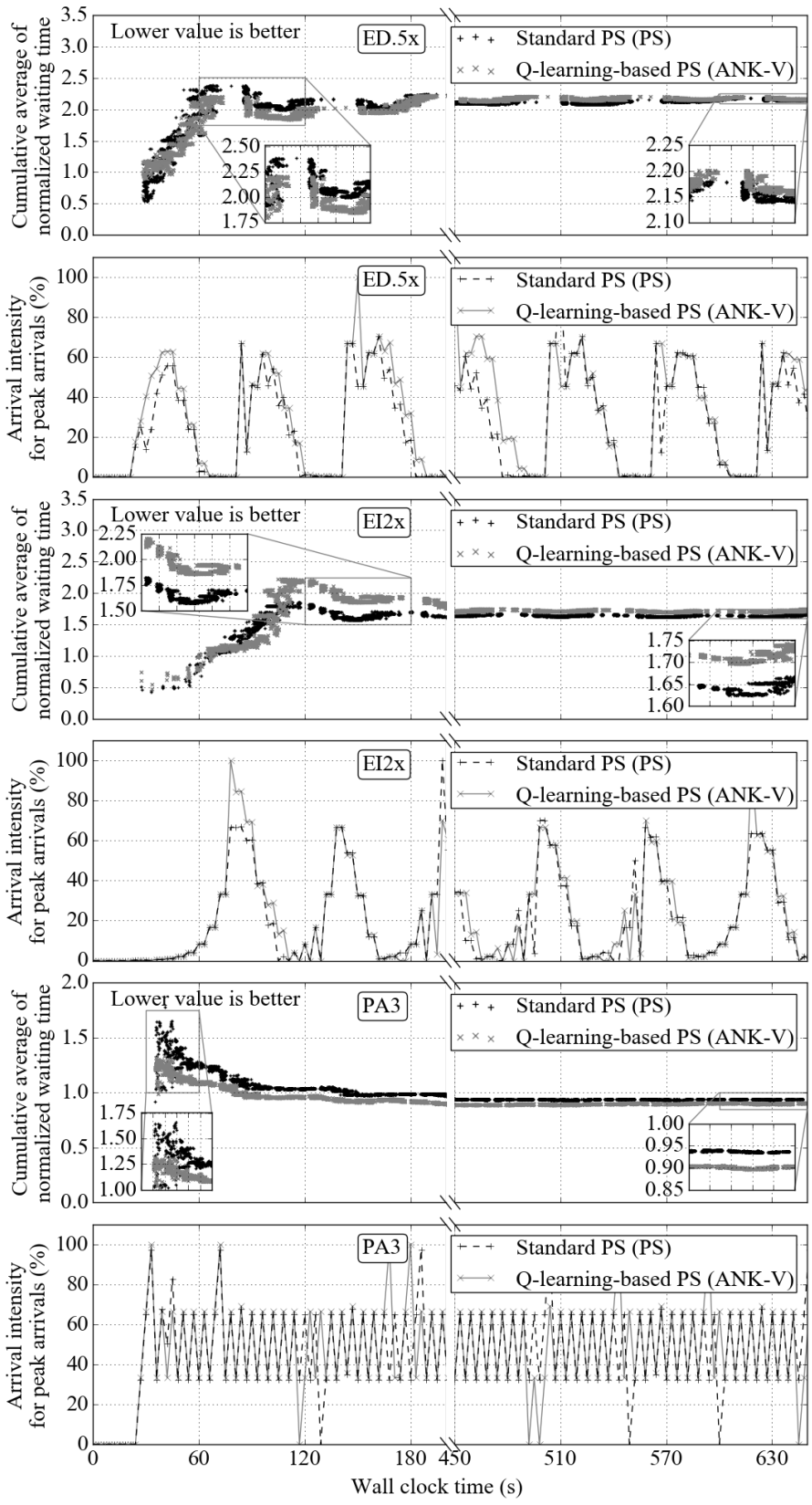

Figure 6.2: The normalized response time of the Q-learning-based portfolio scheduler (`ANK-V`) and of the standard portfolio scheduler (`PS`). Workload: (top pair of plots) dynamic (`ED.5x`),(middle part of plots) dynamic (`EI2x`), and (bottom pair of plots) static (`PA3`).

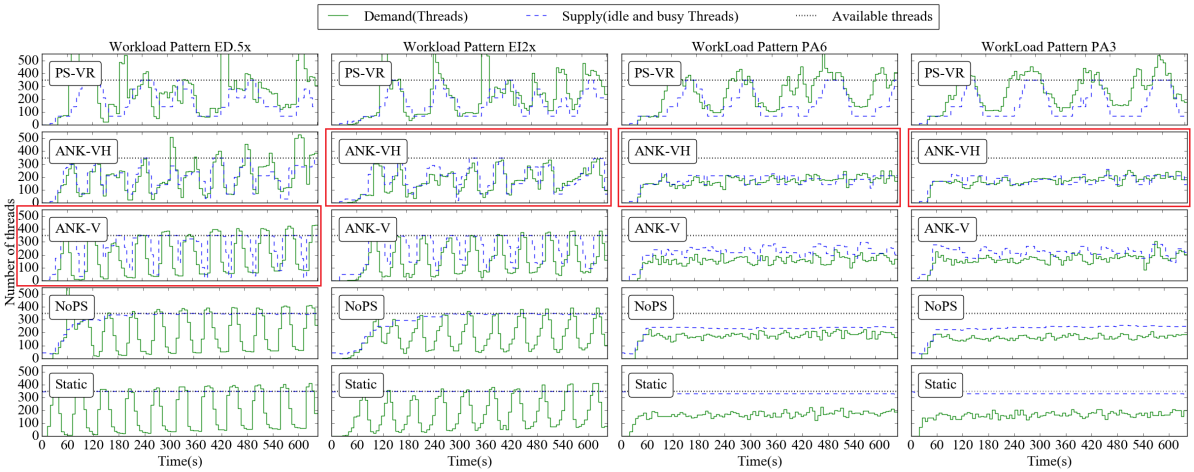

Figure 6.3: The supply and demand curves for five different auto-scalers, under (left) dynamic and (right) static workloads. The "Available threads" horizontal line indicates the resource limit of 350 threads. The best performance for each workload is highlighted with a red box.

between the Q-learning-based portfolio scheduler and the standard portfolio scheduler in the expiration rate and in the response time degradation.

Figure 6.2 depicts a deeper analysis of the performance of the Q-learning-based portfolio scheduler. For this, we compare the normalized workflow waiting times achieved by our scheduler 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 6.2 shows the performance of our schedulers and of `PS`, for static (`PA3`) and dynamic (`ED.5x` and `EI2x`) workloads, and correlated sub-plots depicting the arrival of workflows in the system. The Q-learning-based portfolio scheduler reduces the normalized waiting time by 5–20% compared with the standard portfolio scheduler, 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%. We explain this as follows. Because static workloads exhibit strong recurring patterns, the Q-learning-based portfolio scheduler 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.

6.2. Evaluation of Elasticity and Resource Utilization

To assess the elasticity, we conduct experiments in the *public-cloud mode*. As explained in Section 5, in an ideal case the supply should follow the envelope of the demand. Figure 6.3 displays the supply and demand curves for each auto-scaler under different workloads and highlights the one having the best performance (where the supply and demand curves are close to each other) with a red box. The Q-learning-based portfolio scheduler which uses only vertical scaling (`ANK-V`) performs better with the ED.5x workloads. The Q-learning-based portfolio with both horizontal and vertical scaling (`ANK-VH`) beats all the others under the other three workloads. `PS-(VR)` often under-provisions which means that it can 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. Figure 6.3 gives an overview of elastic behavior of the considered auto-scales for various configurations. To perform a comprehensive comparison including all system- and user-oriented metrics, we use two numerical methods. We rank the policies using the pairwise comparison method and the fractional difference comparison method which are introduced in Section 5.3.3.

The comprehensive comparison is based on the metrics described in 5.3.3 which include *over- and under-provisioning accuracy*, *over- and under-provisioning timeshare*, two types of *instability*, the *average job throughput*, the *average number of used threads*, and the *slowdown in job response time*. Table 6.1 shows the comparison results. The best auto-scaler for each workload is highlighted in bold. Considering all the combined results generated by the described metric aggregation approaches, ANANKE outperforms all the other config-

Table 6.1: The results of the pairwise and fractional comparisons. The winners are highlighted in bold. AS stands for auto-scaler.

AS	Pairwise (points)				Fractional (frac.)			
	ED.5x	EI2x	PA6	PA3	ED.5x	EI2x	PA6	PA3
PS-(VR)	11	13	13	13	3.00	2.88	3.65	3.58
ANK-VH	17	16	**19**	15	**2.89**	**2.70**	**1.80**	**1.94**
ANK-V	**20**	**22**	15	**19**	6.78	5.53	4.00	3.95
NoPS	19	16	17.5	17	6.02	5.82	4.08	3.92
Static	13	13	15.5	16	13.44	15.91	14.51	14.46

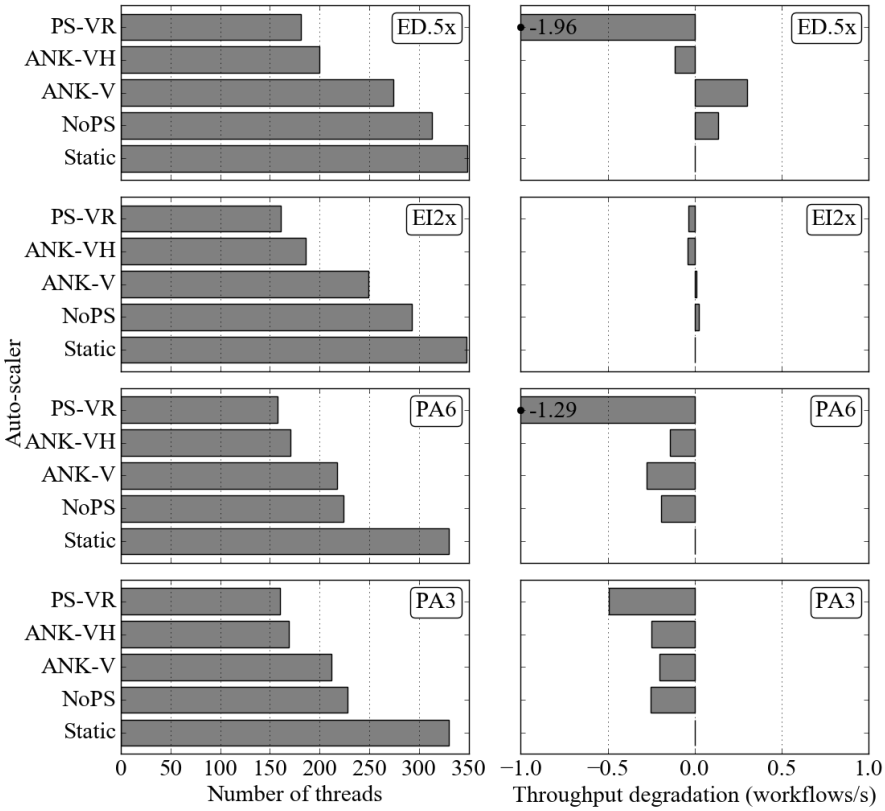

Figure 6.4: The average number of used threads during the experiment and the average throughput degradation. The baseline is the static case without an auto-scaler.

urations in both static and dynamic workloads. From the results in Figure 6.2 and Table 6.1 we can conclude that the reduction in the number of utilized resources often leads to the user-experienced performance degradation. Since our SLO requires workflows to meet their deadlines, further we focus on the influence of the number of used threads on the user-experienced performance. Because we can not observe significant difference in workflow waiting times between ANANKE and the standard portfolio scheduler, we measure changes in the throughput to evaluate the user-experienced performance. For that we selected two metrics which are the throughput degradation in workflows per second (compared with the Static case) and the number of used threads.

Figure 6.4 shows the results. Although PS-(VR) uses 48–52% less resources, it also causes higher throughput degradation (–1.96 workflows per second at most). ANK-VH has lower throughput degradation from –0.14 to –0.11 workflows per second which is 0.68–0.8% of the baseline throughput and saves from 42.8% to 48.2% of resources. From the results shown in Figure 6.4, we can conclude that taking the performance degradation and resource costs into account, the Q-learning-based portfolio scheduler (which uses both horizontal and vertical auto-scaling) allows to achieve the best user-experienced performance with the lowest resource cost among all the four auto-scalers with static and dynamic workloads.

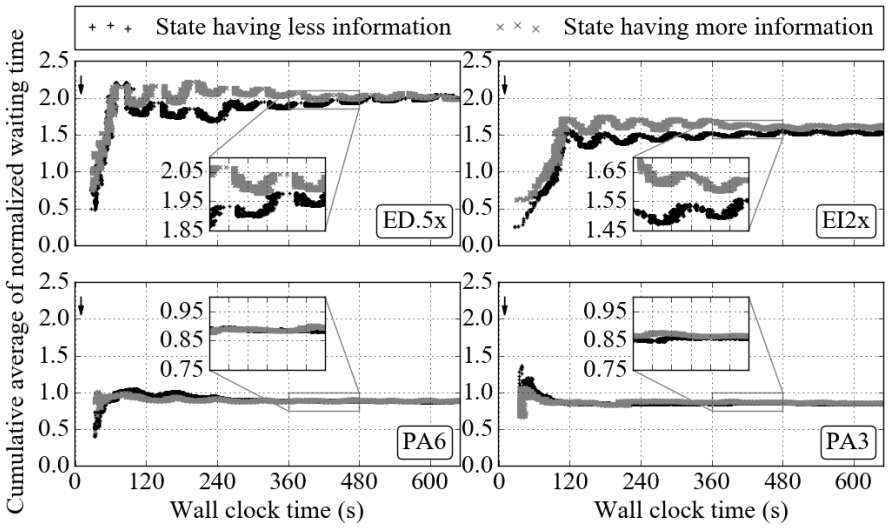

Figure 6.5: Normalized job response times for the *state with more information* (MD) and the *state with less information* (LD) configurations. The arrow at the left upper corner indicates that the lower the better. The upper two cases use dynamic workloads, the bottom two use static workloads.

6.3. Decision Table Configuration Impact on Workflow Performance

In this section, we discuss our findings about the trade-off between the configuration of the decision table and the user-experienced performance.

We study this topic varying the number of actions, varying the number of system states, and varying the size of the policy pool. All the related experiments are performed in a private cloud setting. We use job waiting time normalized by the execution time to represent the user-experienced performance.

6.3.1. Different Configuration Setting in Determining State s_t

To compare the impacts of different configurations of system states (stored in the decision table) on user-experienced performance, we use LD, and MD configuration setting mentioned in Section 5.4.3 and measure the job waiting time and normalize it by job execution time. Figure 6.5 shows the normalized job waiting time under different configuration settings. The Q-learning-based portfolio scheduler with the LD configuration performs better than the one with the MD configuration under a dynamic workload. Compared to the scheduler with the MD configuration, the LD configuration reduces job waiting times by 15.9% under the ED.5x workload. Under the EI2x workload, the LD configuration reduces job waiting times from 4.5% to 12%. However, for static workloads, the difference in the measured metrics is less obvious. Comparing the zoomed-in areas (between 360s and 480s), we can notice that the waiting times converge to the same value faster under a static workload. This observation indicates that the state configuration also influences the convergence speed of the job waiting time. We explain this as following: Dynamic workloads can cause rapid changes in system states. As a consequence, the size of the decision table with the MD configuration becomes larger as it allows more possible state values. A larger decision table requires more time for its training and thus causes more inaccurate scheduling decisions. So scheduler with LD configuration performs better under a dynamic workload. However, the situation is different if the workload is static. The system is stabler, and the number of possible states values is less than the case under dynamic workloads. No matter which state configuration is used, the decision table can always be trained in a short period. Since there is no great difference between LD and MD configuration in the training time, the user-experienced performances observed in LD and MD cases are thus similar. We can conclude that smaller value space of system state leads to better performance under dynamic workloads.

6.3.2. Different Configuration Setting in Determining Action a_t

As the size of the state value space affects the user-experienced performance, in this section, we investigate if varying the value space of action has a similar effect on workflow performance. The results are shown in Figure 6.6. In Figure 6.6 we cannot observe any significant differences in normalized job waiting times. Taking

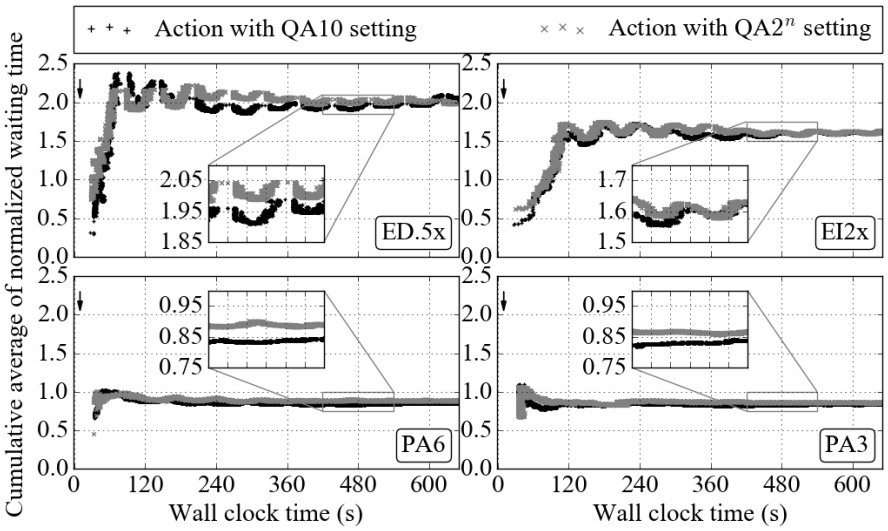

Figure 6.6: The user-experienced performance for QA10 and QA2^n configurations. The user-experienced performance is represented as job waiting times normalized by job execution times. The arrow at the right upper corner indicates that the lower the better.

four workloads into account, the QA10 configuration reduces normalized job waiting times by 5.5%–9.7%. The values of normalized waiting times with different action configurations are closer to each other for most of the experiment time. No matter which type of workload is used, both action configurations have similar user-experienced performance (normalized waiting time). Form Figure 6.6 we can conclude that there is no strong correlation between the size of the action value space and the job waiting tim

6.3.3. Policy Pool Size Impact on Workflow Performance

The final hypothesis which we would like to check is the influence of the size of the policy pool on the user-experienced performance. We use the policy pool configurations from Table 5.2 and perform experiments. The smallest configuration uses three policies and the largest configuration uses 16 polices. Figure 6.7 shows the dependency between the size of the policy pool and the normalized job response time. Under the ED.5x and EI2x workloads, no strong correlation between the normalized job response time and the size of the policy pool can be found. Under the PA6 and PA3 workloads, the normalized job response time is inversely proportional to the number of policies. The bigger size of the policy pool leads to lower values of normalized job response times. Though, a clear linear dependency is not found between the size of the policy pool and the user-experienced performance. We can conclude that for static workloads the Q-learning-based portfolio scheduler with bigger policy pool has smaller task waiting times. However, such a relation is not observed under a dynamic workload.

6.4. Analysis at 10× Larger Scale

Last, we show the performance of the Q-learning-based portfolio scheduler when managing 10 times more resources—up to 50 nodes in the *scalability mode* (see Section 5.2). Because these experiments are large, we only use `PS-(VR)` as the baseline. Figure 6.8 shows the supply and demand curves of two auto-scalers. Similarly to what we have observed from the evaluation of elasticity, `PS-(VR)` experiences problems: it fails to adjust the available number of resources quickly, to match demand changes. `ANK-VH` shows better results: compared with `PS-(VR)`, the gap between the supply and demand curves for `ANK-VH` is (visibly) smaller. Overall, Figure 6.8 indicates that the Q-learning based portfolio scheduler is able to deliver good elasticity properties, by provisioning moslty the required number of resources, even at 10 times larger scale than the current practice.

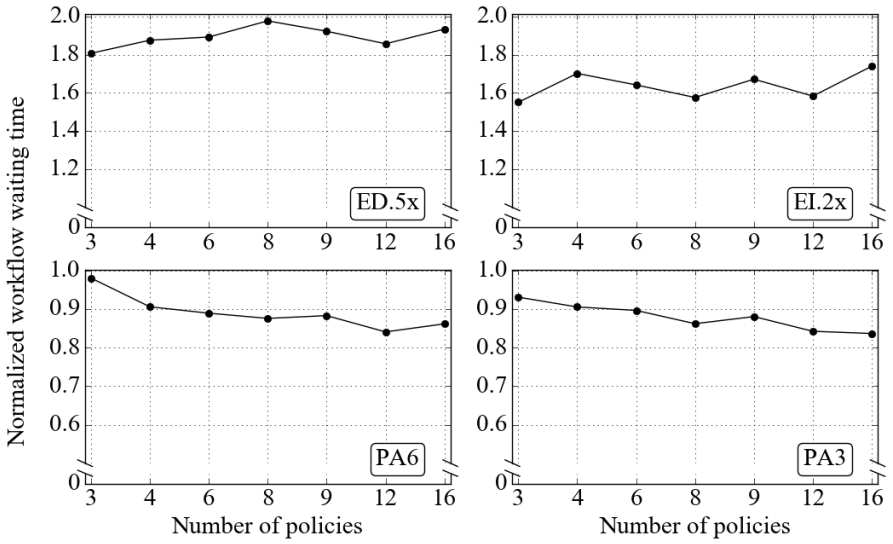

Figure 6.7: The vertical axis is the job waiting time normalized by the job execution time and the horizontal axis represents the size of the policy pool varying from 3 to 16.

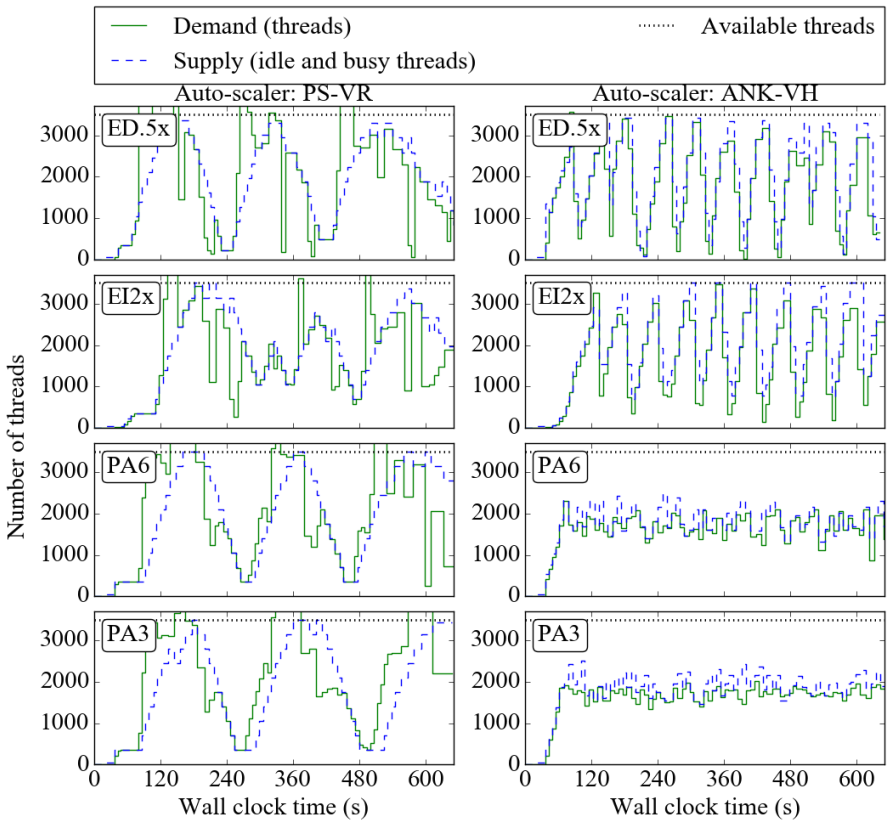

Figure 6.8: Comparison when auto-scaling at large scale between: (left) a simple reactive mechanism, and (right) ANANKE.

7

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. [29] 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. Different from the authors' approach, our work uses online training, by taking advantage of the dynamic portfolio scheduling. Padala et al. [25] use reinforcement learning to learn the application's behavior and to design a solution based on Q-learning to perform vertical scaling problems on the VMs level. Compared with the authors' work, we add portfolio scheduling, scale finer-grained resources (threads in our work vs. VMs in theirs), and take both horizontal and vertical scaling into account. Bu et al. [6] use reinforcement-learning to change the configuration of VMs and resident applications, whereas we add portfolio scheduling and solve both resource allocation and workload scheduling.

Work on portfolio scheduling: Although the general technique emerged in finance over 50 years ago [14], portfolio scheduling has been adopted in cloud computing only in the past 5 years—introduced simultaneously, by Intel [26] and by authors of this work [12]. Our current work extends a standard portfolio scheduler to support reinforcement learning and to the significantly different workload complex industrial workloads. Closest to our work, and not using reinforcement learning, Kefeng et al. [11, 12] build a standard portfolio scheduler equipped only with threshold-based policies and only focusing on scientific bags-of-tasks; and van Beek et al. [30] 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. Closest to our work, several scaling policies [21, 22, 27] take deadline constraints as their main SLO. Our work further considers complex workflows and, simultaneously, resource provisioning, and performs real-word experiments for evaluation.

Work on auto-scaling in cloud-like settings: Marshall et al. [23] present many resource provisioning policies to match resource supply with demand. 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. [16] propose a detailed comparative study of a set of auto-scaling algorithms. We use their system- and user-oriented evaluation metrics to assess the performance of our auto-scaling approach, but consider different workloads and thus supply and demand curves.

8

Conclusion and Future Work

Dynamic scheduling of complex industrial workflows is beneficial for companies when migrating to cloud environments. Current state-of-the-art approaches which are based on portfolio scheduling do not take into account periodic effects which are often observed in workflows. SLOs specific for complex industrial workflows are also rarely addressed. To fill this gap, in this work we have explored the integration of a learning technique into a cloud-aware portfolio scheduler.

We 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 implemented ANANKE as a prototype of the production environment at Shell, and conducted real-world experiments. The experimental results allowed us to analyze the interdependencies between the parametrization of Q-learning and portfolio scheduling, and discover how the size of the decision table and the size of the scheduling-policy pool affects the performance. We also compared ANANKE with its state-of-the-art and state-of-practice alternatives. In our experiments, we use a diverse set of workloads derived from real industrial workflows, and various metrics to characterize the performance and elasticity.

Our results show that the usage of the Q-learning policy in the portfolio scheduler allows to get 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.

In the future, we plan to extend our work with considering other reinforcement learning techniques, such as error-driven learning and temporal difference learning. Our ongoing work is focusing on an extended set of workloads and SLOs, and includes experiments on bigger scale, especially temporal, that are only tractable through trace-based simulation.

Methodology, Code, and Data Availability

The authors of this article are considering the SC16 Reproducibility Initiative. They have experience with releasing methodology, code, and data related to their research.

Bibliography

[1] Amazon web services (AWS). https://aws.amazon.com.

[2] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick H. J. Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Comp. Syst.*, 29(1):158–169, 2013.

[3] Younsun Ahn and Yoonhee Kim. Auto-scaling of virtual resources for scientific workflows on hybrid clouds. In *HPDC*, pages 47–52, 2014.

[4] Graham Baird et al. Upgraded online protection and prediction systems improve machinery health monitoring. *Asset Management & Maintenance Journal*, 27(2):16, 2014.

[5] Henri E. Bal, Dick H. J. Epema, Cees de Laat, Rob van Nieuwpoort, John W. Romein, Frank J. Seinstra, Cees Snoek, and Harry A. G. Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *IEEE Computer*, 49(5):54–63, 2016.

[6] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE Trans. Parallel Distrib. Syst.*, 24(4):681–690, 2013.

[7] Steve J. Chapin, Walfredo Cirne, Dror G. Feitelson, James Patton Jones, Scott T. Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *JSSPP*, pages 67–90, 1999.

[8] Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve, and Alla Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *ICEBE*, pages 281–286, 2009.

[9] Reginald Cushing, Spiros Koulouzis, Adam S. Z. Belloum, and Marian Bubak. Prediction-based auto-scaling of scientific workflows. In *(MGC*, page 1, 2011.

[10] Herbert A David. Ranking from unbalanced paired-comparison data. *Biometrika*, 74(2):432–436, 1987.

[11] Kefeng Deng, Junqiang Song, Kaijun Ren, and Alexandru Iosup. Exploring portfolio scheduling for long-term execution of scientific workloads in iaas clouds. In *SC*, pages 55:1–55:12, 2013.

[12] Kefeng Deng, Ruben Verboon, Kaijun Ren, and Alexandru Iosup. A periodic portfolio scheduler for scientific computing in the data center. In *JSSPP*, pages 156–176, 2013.

[13] Marc Frîncu, Stéphane Genaud, and Julien Gossa. Comparing provisioning and scheduling strategies for workflows on clouds. In *IPDPSW*, pages 2101–2110, 2013.

[14] Bernardo A Huberman, Rajan M Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296), 1997.

[15] Alexey Ilyushkin and Dick H. J. Epema. Towards a realistic scheduler for mixed workloads with workflows. In *CCGrid*, pages 753–756, 2015.

[16] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Roman Herbst, Alessandro Papadopoulos, and Alexandru Iosup. An experimental performance evaluation of autoscaling algorithms for complex workflows. In *ACM/SPEC ICPE*, 2017.

[17] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick H. J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE TPDS*, 22(6):931–945, 2011.

[18] Dalibor Klusácek and Simon Tóth. On interactions among scheduling policies: Finding efficient queue setup using high-resolution simulations. In *Euro-Par*, pages 138–149, 2014.

[19] Li Liu, Miao Zhang, Yuqing Lin, and Liangjuan Qin. A survey on workflow management and scheduling in cloud computing. In *CCGrid*, pages 837–846, 2014.

[20] Shenjun Ma, Alexey Ilyushkin, Alexander Stegehuis, and Alexandru Iosup. Ananke: a Q-Learning-Based Portfolio Scheduler for Complex Industrial Workflows: Extended Technical Report. Technical report, TU Delft. DS-2017-001.

[21] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. In *SC*, page 22, 2012.

[22] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *SC*, pages 49:1–49:12, 2011.

[23] Paul Marshall, Henry M. Tufo, and Kate Keahey. Provisioning policies for elastic computing environments. In *IPDPS*, pages 1085–1094, 2012.

[24] Simon Ostermann, Radu Prodan, and Thomas Fahringer. Dynamic cloud provisioning for scientific grid workflows. In *GRID*, pages 97–104, 2010.

[25] Pradeep Padala, Anne Holler, Lei Lu, A Parikh, M Yechuri, and X Zhu. Scaling of cloud applications using machine learning. *VMware Technical Journal*, 2014.

[26] Ohad Shai, Edi Shmueli, and Dror G. Feitelson. Heuristics for resource matching in intel's compute farm. In *JSSPP*, pages 116–135, 2013.

[27] Jiyuan Shi, Junzhou Luo, Fang Dong, Jinghui Zhang, and Junxue Zhang. Elastic resource provisioning for scientific workflow scheduling in cloud under budget and deadline constraints. *Cluster Comp.*, 19(1): 167–182, 2016.

[28] Omer Ozan Sonmez, Nezih Yigitbasi, Saeid Abrishami, Alexandru Iosup, and Dick H. J. Epema. Performance analysis of dynamic workflow scheduling in multicluster grids. In *HPDC*, pages 49–60, 2010.

[29] Gerald Tesauro, Nicholas K. Jong, Rajarshi Das, and Mohamed N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *ICAC*, pages 65–73, 2006.

[30] Vincent van Beek, Jesse Donkervliet, Tim Hegeman, Stefan Hugtenburg, and Alexandru Iosup. Self-expressive management of business-critical workloads in virtualized datacenters. *IEEE Computer*, 48 (7):46–54, 2015.

[31] David Villegas, Athanasios Antoniou, Seyed Masoud Sadjadi, and Alexandru Iosup. An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds. In *CCGrid*, pages 612–619, 2012.

[32] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[33] Yi Wei, M. Brian Blake, and Iman Saleh. Adaptive resource management for service workflows in cloud environments. In *IPDPSW*, pages 2147–2156, 2013.

[34] Li Yu and Douglas Thain. Resource management for elastic cloud workflows. In *CCGrid*, pages 775–780, 2012.