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

Shenjun Ma


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

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Shenjun Ma

16th June 2017

**Author**
Shenjun Ma

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

**MSc presentation**
22nd June 2017

**Supervisors**

Alexandru Iosup
Alexander Stegehuis

**Graduation Committee**

| | |
|---|---|
| Prof. dr. ir. Dick H.J. Epema | Delft University of Technology |
| Prof. dr. ir. Alexandru Iosup | VU University Amsterdam |
| Alexander Stegehuis | Shell and CGI |
| Assistant Prof. dr. ir. Alberto Bacchelli | Delft University of Technology |

**Abstract**

Complex workflows that process sensor data are useful for industrial infrastructure management and diagnosis. Although running such workflows in clouds promises to reduce 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.

# Preface

This thesis has been produced as my final piece of work for my study at Delft University of Technology as a master student in Computer Science. It was a very gratifying journey and I am grateful that I made it.

Firstly, I would like to thank my university supervisor Prof. dr. ir. Alexandru Iosup for his constant guidance during my research on the master thesis. It is the knowledge and methodology he imparts inspired me to conduct my research in this field. Moreover I want to thank Alexandru for encouraging me to strive for great results and helping finish my master thesis in the end.

Secondly I want to thank Alexander Stegehuis who co-supervised me in Shell. He gave me the opportunity to work on a very interesting project in a very pleasant working environment. His guidance and knowledge helped me solve the technical issues that I encountered. During my project I had many opportunities to consult the experts in Shell, I want to thank Lennard Bakker for his patience and time. To all other engineers that helped me getting started, and during the project many thanks for your time.

In addition, many thanks to all the people in Distributed System group and @Large research group. I would like to thank Ir. Laurens Versluis, Ir. Wing Ngai and so many others for their valuable suggestions and feedback. I also want to thank Ir. Alexey Ilyushkin and Ir. Vincent van Beek for their previous works that my research relies on.

Furthermore, I want to thank the other members of the committee, Prof. dr. ir. Dick H.J. Epema and Assistant Prof. dr. ir. Alberto Bacchelli, for their interest in my research project.

Finally my thanks would go to my beloved family for their loving considerations and great confidence in me all through last two years.

Shenjun Ma
Delft, The Netherlands
16th June 2017

# Contents

# Chapter 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 is often involved the usage of complex workflows designed to analyze real-time sensor data [6]. Although the management of workflows and resources has already been studied for decades [46, 19, 45, 2, 27], previous works have mostly focused on scientific workloads [13, 34, 22, 3] which differ from 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 industrial production environments [10, 26]. In contrast to the previous body of work, in this 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.

## 1.1 Problem Statement

Compared to scientific workloads, production workloads are more often to 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 also 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.


Figure 1.1: Seven Levels of the Workload in Production. By the time this project is completed, workflows for level 4–6 are under developed.

### 1.1.1 Workloads of Smart Connect Project

The Shell company has managed the "Smart Connect" project for decades. The Shell engineers develop and maintain a real-time automated diagnosis and monitoring framework for the large assets (compressors) owned by the Shell company. By analyzing the sensor data collected from the assets, the diagnosis and monitoring system can send alerts to the engineers if any mechanical failure happens. Chronos is a sub-system of the entire diagnosis and monitoring framework and is designed for managing resources and scheduling workloads. To correctly detect any mechanical failure of the assets, several calculation jobs need to be produced. These jobs can be grouped into seven levels.

In Figure 1.1, different levels of calculation jobs (workflows) are presented (starting from Level 0). The calculations are dependent on the results of the upper level calculations. Currently, Level 1, Level 2, and Level 3 calculations are being used for the majority of assets. The resource consumption and make-span for each level are not equal. Most of these calculations can be completed in a very short space of time. However, some calculations may take significantly more time and the completion time is unpredictable.

There are some patterns in the job's arriving behaviors. A big amount of calcula-

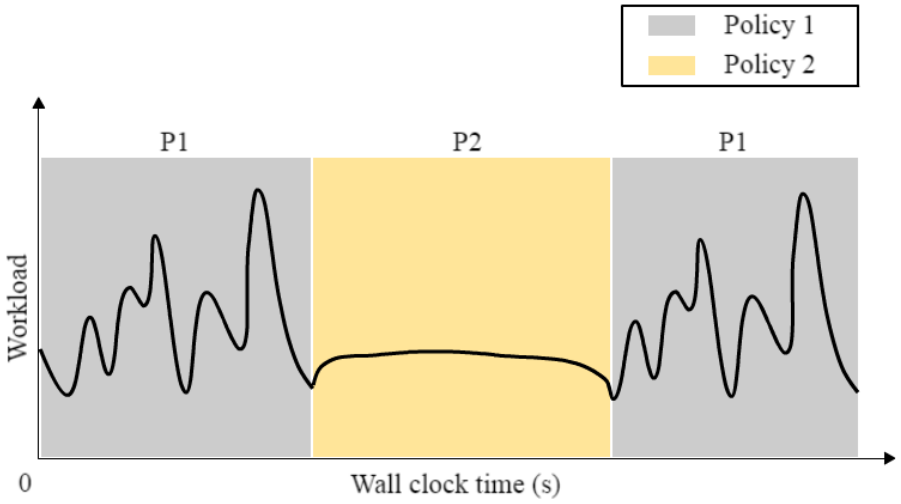


Figure 1.2: Portfolio scheduler select different policy according to the workload modes. [16]

tions has to be performed periodically and completed in a certain period (deadline).

### 1.1.2 Portfolio Scheduling

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 [20], a portfolio scheduler is equipped with a set of policies, each designed for a specific scheduling problem. The policy is selected dynamically, according to a user-defined rule and a feedback mechanism. Figure 1.2 depicts how a portfolio scheduler switches policies dynamically and responses to the workload changing. 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 [39, 43], and, in contrast, that a portfolio scheduler performs well without external (in particular, manual) tuning [16, 17].

### 1.1.3 Research Questions

Although portfolio schedulers are promising for the context of complex workflows, previous approaches [37, 42] lack by design the ability to use historical knowledge in their selection. All these existing 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, three research questions arise in the context of complex industrial workflows:

1. How to adapt the concept of portfolio scheduling to the RM&S framework used on complex industrial workflows in production environment?

2. How to use learning technique based on historical information when performing portfolio scheduling?

3. How to evaluate the learning-based portfolio scheduler, experimentally, through a prototype?

## 1.2 Approach

To answer the research questions, we propose in this work to integrate a reinforcement-learning technique, Q-learning [44], into a cloud-aware portfolio scheduler. A Q-learning policy interacts with a system by applying an action to it, and learns about the merits of the action from the system's feedback (reward).

The Q-learning policy is trained by all the previous data including system states, actions already made, and the reward value accordingly. The well-trained Q-learning policy thus has the knowledge of historical information of the system and can use the knowledge to make actions.

To evaluate our approach, we implement a prototype which can be integrated into the existing RM&S frameworks and conduct real-world experiments. We explore the strengths and limitations of a Q-learning-based portfolio scheduler managing diverse industrial workflows and cloud resources.

## 1.3 Main Contribution

Towards answering the research question, our main contribution is fourfold:

1. We design ANANKE, an RM&S architecture that integrates a reinforcement-learning technique, Q-learning, into a portfolio scheduler (Chapter 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 (Chapter 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 [6].

3. We evaluate our learning-based portfolio scheduler through real world experiments (Chapter 6). Using the cloud-like experimental environment DAS-5 [7]

and workloads derived from a real industrial workflow, we analyze ANANKE's user-level and system-level performance, and elasticity (metrics defined in Chapter 5). We also compare ANANKE with a baseline system and with a portfolio-scheduling-only approach.

4. The material in this thesis, condensed, has been published as: S.Ma, A. S. Ilyushkin, A. Iosup. ANANKE: a Q-Learning-Based Portfolio Scheduler for Complex Industrial Workflows, International Conference on Autonomic Computing 2017, Columbus, Ohio, USA, July 17-21, 2017.

## 1.4 Reading Guidelines

This thesis report has the following structure: In Chapter 2, we present the background knowledge and related works on industrial workflows, portfolio scheduling, and Q-learning. In Chapter 3, we investigate the requirements and show a design for using a learning-based portfolio scheduler with the current framework (Chronos) in production. In Chapter 4, we elaborate the configuration and implementation details of the ANANKE prototype. In Chapter 5, we present all the environment, metrics and baselines configurations for the evaluation experiments. In Chapter 6, we show the results of the experimental evaluation of ANANKE and explain our findings in turns. In Chapter 7, we summarize the entire work and list several possible future works.

# Chapter 2

# Background

In this chapter, we define the system model used in this work: workload, system architecture, and system infrastructure. In practice, this model is already commonly used in real-time infrastructure monitoring systems, such as the Chronos [6] 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.

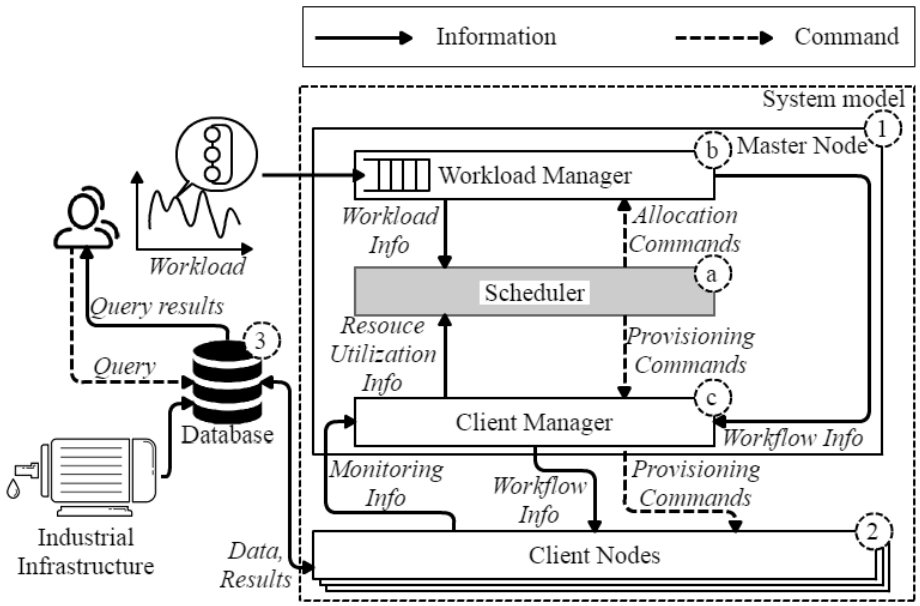


Figure 2.1: Three-tiers architecture for processing workflows.

## 2.2 Processing Sensor Data in Practice: Three-Tiers Architecture

In practice, infrastructure monitoring systems commonly use a tree-tiers 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.

## 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. 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 [24]. 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 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 [43] 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.

## 2.4 Q-Learning

Q-learning [44] is a typical reinforcement learning technique. In this work, we study a reinforcement learning-based portfolio scheduler and implement a provisioning policy according to the Q-learning algorithm. Figure 2.2 generally shows the operational flow of the Q-learning algorithm. The Q-learning policy interacts with the environment by applying an action and learning from the reward awarded by the environment. The environment provides its state $s_t$ at each time interval $t$. The system applies an action $a_t$ and receives a reward $r_{t+1}$. At the same time, the environment changes its state to $s_{t+1}$. The objective of the Q-learning technique is to choose an optimal action to achieve the maximum reward in the long run. The reward value is calculated by a user defined reward function. The Q-learning policy maintains a decision table which maps actions on states. It considers an old pair of state-action values and makes a correction based on the new feedback form the environment.

In short, the core of the algorithm is a simple iterative value update. Each time when a decision needs to be made, the Q-learning policy selects an action and thus observes a reward along with a new state that depends on both the previous state and the selected action. The decision table is updated by replacing the old value with the new value.

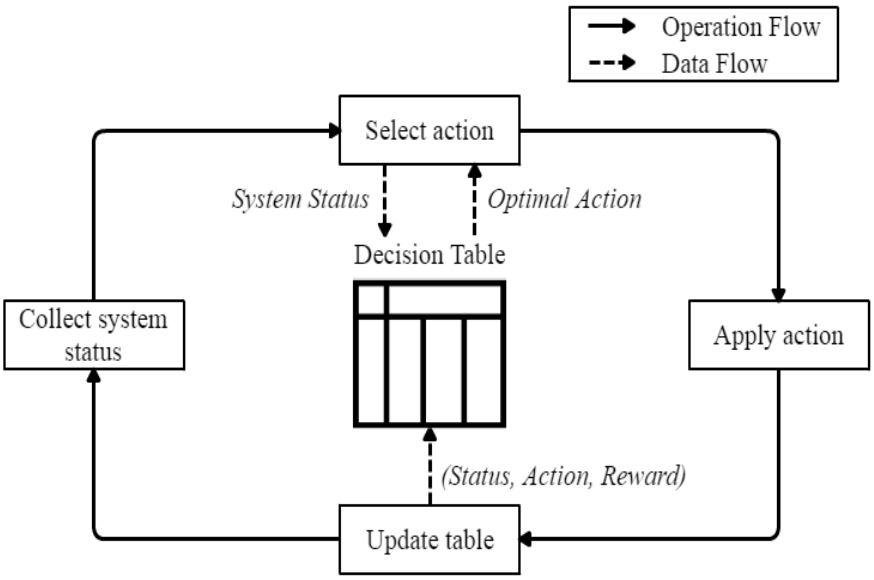


Figure 2.2: Operational Flow of Q-Learning

## 2.5 Portfolio Scheduling

Portfolio scheduling is one of the strategies of optimizing workload scheduling. It can dynamically select and use a scheduling policy, depending on the current system and workload conditions, from a portfolio of multiple policies. In this section, we present the necessary background knowledge of portfolio scheduling.

### 2.5.1 Four Steps of Portfolio Scheduling

As show in Figure 2.3, portfolio schedulers has four main steps [16]:

1. Creation: In this step, a set of policies is created (prepared) for the portfolio scheduler.
2. Selection: In this step, the portfolio scheduler selects one of the scheduling policies as the active policy.
3. Application: In this step, real actions are performed based on the active policy (from the previous step).
4. Reflection: In this step, the portfolio scheduler analyzes the operation of the last selection and application steps. The system parameter and the policy set may also change.

### 2.5.2 Main Components of the Portfolio Scheduler

Figure 2.3 provides a high-level model of the portfolio scheduler. The workloads are stored in a workflow queue. The workflows are executed by allocating re-

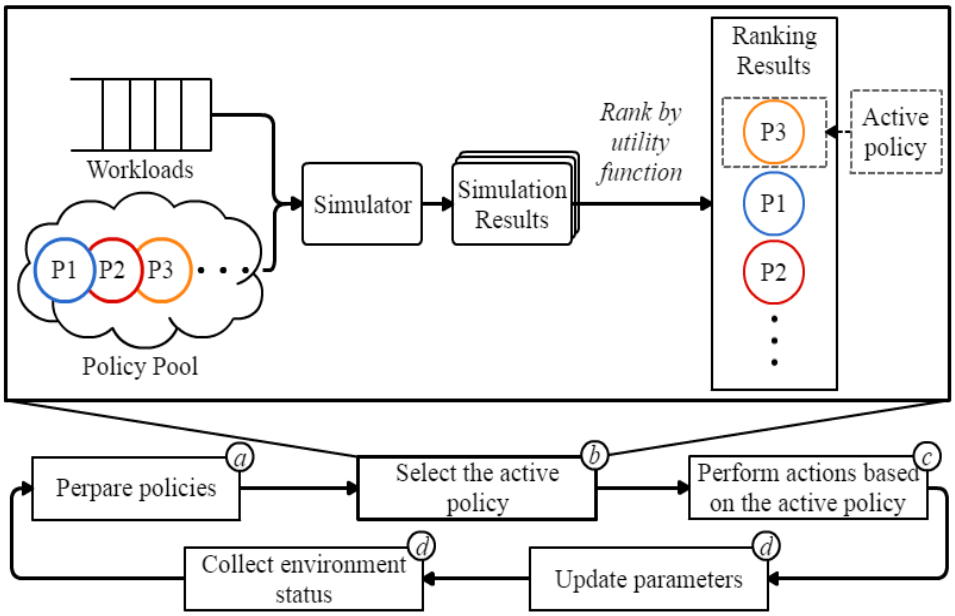


Figure 2.3: Portfolio schedulers follow a traditional process with four steps, creation (*a), selection (b), application (c), and reflection (d).*

sources to them; resources are provisioned using a combination of both allocation policies and provision policies. For each period a combination of policies is used during execution. The scheduler runs simulations based on the current system state and workloads and determines which combination of policies to be used next. The most important components in the model are the Simulator and the Selection criteria.

**The Simulator**

The simulator is designed to generate simulation results (data) for each combination of policies. It needs to balance the accurate of the simulated results with the runtime of the simulator. Deng et al. [17] use the DGSim [25] simulator which is an event-driven simulator. Similar to previous work, we implement an event-driven simulator based on CloudSim [9] to meet our requirements.

**The Selection Criteria**

The selection criteria usually is defined as a utility function. The utility function needs to balance user-oriented and system-oriented performance. The utility function is used to evaluate each combination of policies with the simulated results (data).

## 2.6 State of the Art in Tasks and Resource Management

We survey in this chapter a large body of related works on scheduling with reinforcement learning, on portfolio scheduling, on scheduling workflows, and on autoscaling in cloud computing settings. Relative to it, our work provides the first com-

prehensive study and real-world experimental evaluation of learning-based portfolio scheduling for resource management and dynamic scheduling.

### 2.6.1 Reinforcement Learning

In this work, we use the Q-Learning technique (a typical and widely used reinforcement learning) to do both horizontal and vertical scaling and integrates Q-learning algorithm into an extended portfolio scheduler.

Closest to our work, Tesauro et al. [40] present a hybrid approach combining reinforcement-learning and queuing models for resource allocation. Their RL (reinforcement learning) 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. [35] use reinforcement learning technique 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. [8] use reinforcement-learning to change the configuration of VMs and resident applications (manage the resource in vertical), whereas we add portfolio scheduling and solve both resource allocation and workload scheduling.

Cui et al. [12] propose a workflow scheduling algorithm (policy) based on reinforcement learning for cloud computing platforms. The proposed scheduling algorithm (policy) define the number of VMs as state space and the runtime of task as immediate reward. The experimental evaluation shows the algorithm (policy) can schedule multiple DAGs with multiple SLOs and improve resources utilization. Peng et al. [36] apply reinforcement learning in job (workflow) scheduling to optimize the makespan and average waiting time under the VM resource and deadline constraints. Moreover, the authors propose an advanced queuing model based on reinforcement learning [36] to optimize the job (workflow) response time. Tong et al. [41] introduce reinforcement learning into task scheduling and propose a Q-learning based policy for independent tasks. The proposed policy optimize the tasks allocation by learning task arrival and execution patterns. Different from our approach, Tong's policy is designed for independent tasks instead of workflows.

### 2.6.2 Portfolio Scheduling

Although the general portfolio technique emerged in finance over 50 years ago [20], portfolio scheduling has been adopted in cloud computing only in the past five years—introduced simultaneously, by Intel [37] and by Kefeng et al. [17]. Our current work extends a standard portfolio scheduler to support reinforcement learning and to the significantly different workload—complex industrial workloads (processing sensor data). Closest to our work, and not using reinforcement learning, Kefeng et al. [16, 17] build a standard portfolio scheduler equipped only with

threshold-based policies and only focusing on scientific bags-of-tasks; and van Beek et al. [42] focus on different optimization metrics (for risk management) and workloads (business-critical, VM-based vs. job-based).

### 2.6.3 General Workflow-Scheduling

This body of work includes thousands of different approaches and domains. Closest to our work, several scaling policies [29, 30, 38] take deadline constraints as their main SLO.

Maciej et al. [29] develop and assess task scheduling and resource provisioning algorithms which consider the deadline- and budget-based constraints. The authors produce the evaluation via simulations using scientific workflow ensembles. Mao et al. [30] present a VM-level approach for auto scaling resource and scheduling general workflows. Shi et al. [38] also propose a resource provisioning and task scheduling mechanism to process scientific workflow in the cloud. The presented mechanism also takes budget and deadline constraints into account. Ilyushkin et al. [23] create and analyze policies for scheduling of workflows with and without know task running-time. Relative to our work, the authors focus on a different SLO related with the task running-time instead of deadline constraints.

José Durillo et al. [18] propose MOHEFT, a Pareto-based list scheduling heuristic that provides the user with a set of trade-off optimal solutions, and apply the proposed approach on the commercial public cloud (Amazon EC2) for multi-objective workflow scheduling. Different from our approach, MOHEFT let users determine the scheduling solution that better suits the requirements. Deelman et al. [15] invesigate the design, development and evolution of the Pegasus Workflow Management System, which maps abstract workflow descriptions onto distributed computing infrastructures. Pegasus has been heavily for scientific workflows in a wide variety of domains. Compared with our system, Pegasus is design to map scientific workflows on grid resource and doesn't provide functions of resource scaling. Masdari et al. [33] conduct a comprehensive survey and analysis of schemes scheduling simple and scientific workflows on the cloud resource. However, schemes for schedule complex industrial workflows are not considered in the authors' work.

Our work considers complex industrial workflows (chain based workflows with the deadline as the main SLO) and, simultaneously, resource provisioning, and performs real-world experiments for evaluation.

### 2.6.4 Auto-Scaling in Cloud Computing Setting

In this work, we evaluate the elasticity of ANANKE and compare it with other auto scalers. The related work in auto-scaling are listed as follow:

Marshall et al. [32] 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. [21] propose a de-

tailed 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. Mao et al. [31] present an auto-scaling mechanism to automatically scale computing instances based on workload information and performance desire and apply the proposed mechanism on commercial public cloud resource. The focused SLOs are workflow deadline and resource budget. Different to our approach, the author's mechanism achieves the requirements by scaling different types of VM instances. Atrey et al. [5] propose a framework called BRAHMA, that learns workflow behavior to build a knowledge-base and leverages this information to perform intelligent automated scaling decisions. Relative to our work, the author takes deadline constrained requirement as the main SLO. However, the evaluation of BRAHMA is through simulation and empirical observation which is a limitation. Arabnejad et al. [4] design and implement a fuzzy rule-based system combined with a reinforcement learning algorithm for learning optimal elasticity policies. The proposed system can efficiently scale VM-level cloud resources to meet QoS requirements while reducing cloud provider costs by improving resource utilization. Different to the author's approach, our approach focus on thread level resource scaling.

# Chapter 3

# ANANKE Requirements and Design

In this chapter, 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 Chapter 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 Chapter 2.2. This allows the new system to be backward 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 number of allocated resources based on demand changes.

3. The system should use portfolio scheduling, which shows promise in managing mixed complex workloads [16].

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.

To determine these requirements, we investigate the characteristic of the workload in production and the performance of the current system. Although a private cloud environment is used for production, the develop team plan to migrate the system to the public cloud. Considering the compatibility and the migration in the near future, we derive the first two requirements. We also find there is a strong recurrent

pattern in the workloads due to the constant sampling rate of sensors. Considering the characteristic of workloads, we define the last two requirements. 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 Chapter 2.2 (goal 1), ANANKE has a three-tier architecture, and consists of a master node, client nodes, and a database.
The master node is designed to monitor client nodes and make allocating/provisioning decisions. The client node is mainly used to processes the calculation tasks. Figure 2.1 also indicates how the master node and the client node interact with each other. The client node needs to continuously report its own status back to the master node to synchronize master node with the latest status of the whole system. The monitoring information sent by the client nodes contains both client's resource utilization and the workflow performance. Before the master node making a scheduling decision, it checks all client nodes' states. The master node then makes scheduling/provisioning decision and allocate workflows to suitable clients.
After the client node receiving the provisioning command and allocated workflows, it starts executing the workflows. Each client node retrieves related data for a certain workflow from the database and writes the results back to the database as long as the calculation is completed. The details of components and operational flow inside the master and client nodes will be discussed in the following sections.

## 3.3 Components in the 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.

### 3.3.1 Scheduler

Scheduler (component a in Figure 2.1) is the most substantial component in master node. Addressing design goals 3 and 4, the *scheduler* 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.
The portfolio scheduler has comprehensive knowledge about the whole system and makes scheduling decision based on suitable policy combination. The output of the portfolio scheduler is a set of policies defining how to provision resource and allocate workflow. In principle, a standard portfolio scheduler maintains a policy pool

and a utility function predefined by users. To select the optimal policy combination (provisioning and allocation policies), the scheduler running simulation for each policy combination in the policy pool. Metrics of workflow performance and resource utilization are collected during the simulation. Collected metrics are then used to calculate a numerical value (score) for each policy combination according to the utility function. The score indicates how suitable the policy combination is. Portfolio scheduler selects the combination with the highest score as the optimal one.
Q-learning-based portfolio scheduler is different with standard portfolio scheduler in being equipped with a Q-learning provisioning policy. It adds a policy based on the Q-learning algorithm in the policy pool and trains it during the simulation process. Compared with standard portfolio scheduler, Q-learning-based portfolio scheduler has the capability of learning from historical data.

### 3.3.2 Workload manager

The workload manager (component b in Figure 2.1) maintains the bucket of workflows, collects the information about the workflow performance, and updates the status of every workflow. The workload manager maintains the bucket of workflows, collects the information about the workflow performance and updates the status of every workflow. The workloads are predefined workflows. Engineers deploy model files on the master node. All workflows are generated according to these predefined model files. The workflow is continuously generated and stored in the bucket. Workflows in the bucket are ready to be scheduled. The workload manager uses the response and waiting time of a workflow, and the fraction of completed/expired workflows, as key performance indicators when collecting the metrics of workflows. The collected information along with resource utilization information are sent to the scheduler. Workload manager then takes the policy combination from the portfolio scheduler and selects eligible workflows according to the allocating policy.

### 3.3.3 Client manager

The client manager (component c in Figure 2.1) is designed to communicate with all client nodes and has two main functions.
First one is to collecting continuously per-client status (meta-data). The meta-data can be customized by users according to allocating/provisioning policy's requirements. In this project, the client manager collects resource utilization metrics (current CPU load and the number of busy threads) and workflow performance metrics (calculation result, the amount of allocated workflows which have not been executed). Client manager sends the per-client status to the portfolio scheduler.
The second function is sending to clients when needed commands (actions) and tasks ready to be allocated. The client manager generates provisioning commands (actions) according to provisioning policy given by the portfolio scheduler. The

command(action) along with eligible workflows will then be sent to the target client node. So basically, the client manager is the interface connected to client nodes.

## 3.4 Operational Flow in Master Node

The master node makes a scheduling decision every 6 seconds. The client manager, the workload manager, and portfolio scheduler cooperate with each other to ensure the master node works functionally. How do all these components interact is described as follow:

1. The workload manager and client manager first check resource utilization metrics and workload information. These two components ensure that master node has the latest knowledge of the whole system and send data to Q-learning based portfolio scheduler for the next step.

2. As shown in Figure 2.1, resource utilization, and workload information are the input of the Q-learning-based portfolio scheduler. Simulation and evaluation are then performed based on the input data. To improve performance, simulations are run in parallel. The output is the optimal policy combination which is suitable for the current situation. The provisioning policy in the output is used by client manager, and allocation policy is sent to the workload manager.

3. The workload manager selects workflows and assigns them to target client according to the allocation behavior defined in the policy combination. At the same time, the client manager generates command messages according to the provisioning policy. The command instructs the target client how to change the current resource (lease, release or keep unchanged). Along with the allocated workflow from the workload manager, action (provisioning) command is sent to the target client.

## 3.5 Components in 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 Chapter 2.3). Each thread continuously fetches workflows from the workflow 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.

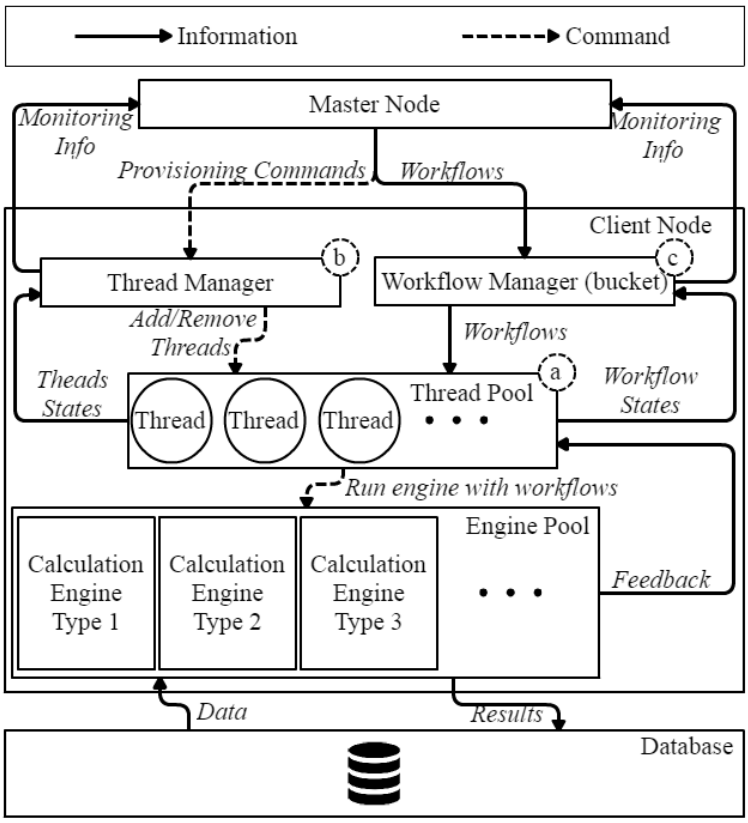


Figure 3.1: Components of the client node.

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 the 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 workflows stored in the bucket normalized by the size of the bucket and the average completion time for the last five executed workflows. We discuss the details of these components in this section.

### 3.5.1 Thread pool

Thread pool (component a in Figure 3.1) maintains a set of threads (smallest working units, see Section 2.3). Changing resource configuration thus is modifying the number of threads maintained by the threads pool. Each thread continuously fetches workflows from the workflow manager and calls an external calculation

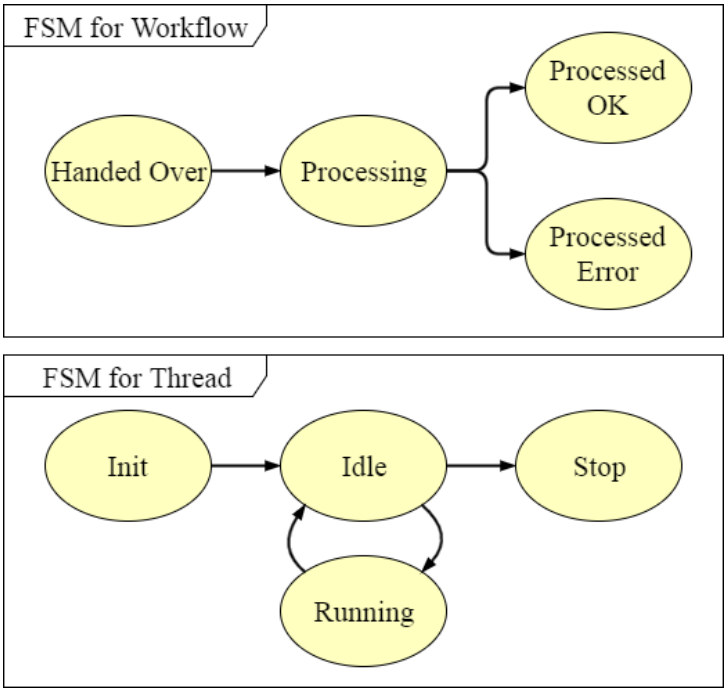


Figure 3.2: Finite-State Machines of the Workflow and Thread.

engine (CCE) from the engine pool.

The calculation engine (CCE) is just an application designed to calculate sensor data; it defines all calculation steps. Different types of workflows are calculated by different calculation engine. Currently, the Shell engineers have developed six calculation engines and deployed them on each client node. After the calculations, it writes the result to the database and returns a feedback message to the thread: *Processed OK* means the calculation is completed successfully and *Processed Error* means the calculation is failed.

In this project, the thread logic is completely determined by the workflow tasks and is defined by developers. Once a thread is created, it will continuously ask the task manager for eligible workflows and call CCE until it is terminated (stopped). The life cycle of one thread has several states: *Init* indicates the current thread is under initialization. *Idle* and *Run* shows if the current thread has called a CCE. *Stop* means the current thread is about to be terminated. Figure 3.2 shows how one state jumps to another state.

Each client node can execute multiple threads in parallel. At each moment, a thread executes only a single workflow. The client node is assigned with a ”max threads” property indicating the max amount of threads the threads pool can maintain. We use the number of active threads as the key metric to represent resource utilization.

### 3.5.2 Thread manager

The thread manager (component b in Figure 3.1) 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 the total number of threads) back to the master node, and continuously checks the active threads and terminates idle threads.

As shown in Figure 3.1, the thread manager receives provisioning commands form the master node. The command contains two pieces of instructions. First, it defines whether the client should lease new threads, release current idle threads or remain unchanged. Second, it defines the number of threads should be leased or terminated if changes are needed. The thread manager performs the real modification on the thread pool according to the command. Another feature of the thread manager is that it continuously checks active threads and terminate those have been idle for more than a user-defined threshold value (we define the threshold to be 30 seconds in this work).

The thread manager also monitors the status of all active threads and resource utilization metrics and send them back to master. The thread manager reports the client's status every 6 seconds. In this project, resource utilization metrics are CPU-load and the ratio of the number of busy threads to the number of total threads.

### 3.5.3 Workflow manager

The workflow manager (component c in Figure 3.1) maintains a bucket of workflows allocated to the client node. The bucket has a "maximum size" property which represents the maximum amount of workflow it can stores. The value of this property is set to two times of the size of the thread pool. The workflow manager has two main functions: tracks the workflow states and monitoring the workflow performance.
Each workflow has a "state" property which can be *Processed OK*, *Processed Error*, *Processing*, *Handed*. The first two states mean the execution is finished. *Processed OK* indicates the workflow is completed successfully. *Processed Error* indicates the workflow is terminated because of errors. In this work, workflows exceeding the deadline and having exceptions during the execution are considered as failed workflows. *Processing* means the workflow is being executed and *Handed* means the task is waiting to be executed. Figure 3.2 shows the relations between different states.

To clean up and maintain the buckets, the workflow manager removes workflows whose status is *Processed OK* or *Processed Error* from the bucket. The workflow manager also drops all tasks which have not been executed for more than 4.5 minutes and notify the master node to reschedule these workflows. In this project, task manager is set to do the cleaning every 6 seconds.

The task manager also computes the performance metrics and reports them back to the master node. We define two metrics for our scheduler: the number of work-

flows stored in the bucket normalized by the size of the bucket and the average completion time for the last five executed workflows.

## 3.6 Operational Flow in Client Node

To achieve better workflow performance, client node processes allocated workflows in parallel. To achieve that, components inside client node interacts as follows:

1. When the client node receives workflows and commands form the master node, the thread manager creates or terminates threads according to the provisioning command. At the same time, the workflow manager adds all the allocated task to the local bucket and update related metrics.

2. To execute a task, idle threads will first ask the task manager for eligible workflows. If the local bucket is not empty, the idle thread will be given a workflow and then call the CCE to execute it. If the local bucket is empty, the idle thread will sleep for 6 seconds and ask the workflow manager again. At the time-point when the calculation is started and finished, the thread changes its state and notify the thread manager. Each thread performs the execution process continuously until it is terminated by the thread manager.

3. Each thread will receive a feedback form the CCE when the calculation is done and change the state of the processed workflow accordingly. The state change is reported to workflow manager.

4. Every 6 seconds, both the workflow manager and the thread manager send the monitoring metrics back to the master node.

## 3.7 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.

### 3.7.1 Adding a Portfolio Scheduler to the Architecture

Our design of the scheduler component of the master node is based on a portfolio scheduler. Figure 3.3 depicts the main components of this design: similarly to previous work [17], 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 4.2. Periodically, the portfolio scheduler considers its constituent policies in simulation and selects from them the most promising policy. The selection is done

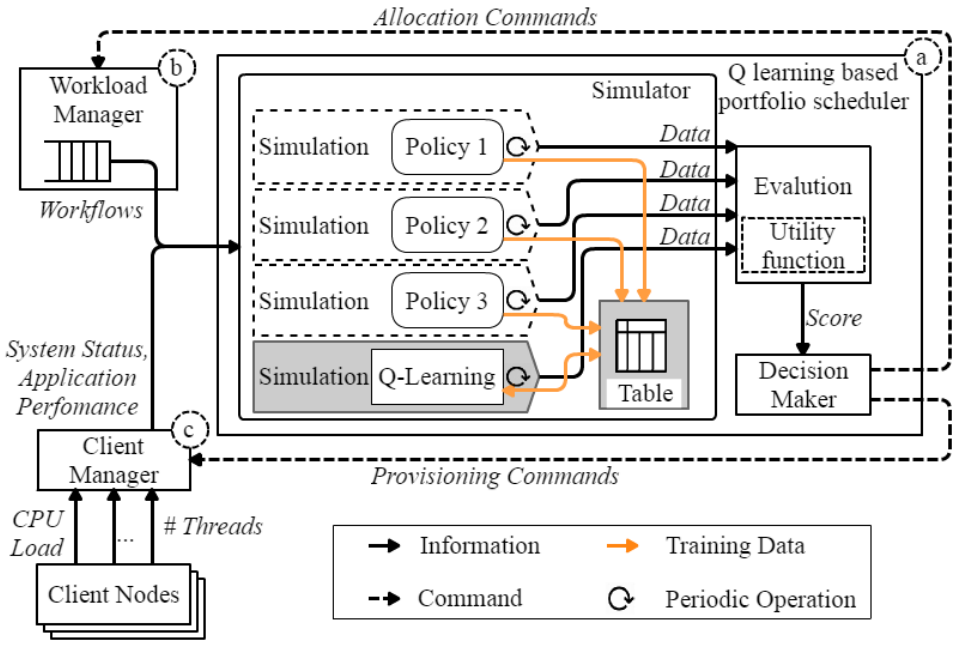


Figure 3.3: Architecture of a Q-learning-based portfolio scheduler, part of the master node. Data generated by each simulation is used as training data, to train the decision table.

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: different Utility functions have been used to focus on performance [37, 17], risk [42], and multi-criteria optimization [16, 42].

### 3.7.2 Designing a Q-Learning-Based Approach

Inspired by the work proposed by Padala et al. [35], we design non-trivially a provisioning policy based on Q-learning. Figure 3.4 shows the operational flow of the Q-learning policy 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, or does nothing to change the set of provisioned resources, respectively. The *reward function* for the Q-learning policy is defined by the users 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 a previous study [35]. Specifically, for every moment of time $t$ we define the reward function as:

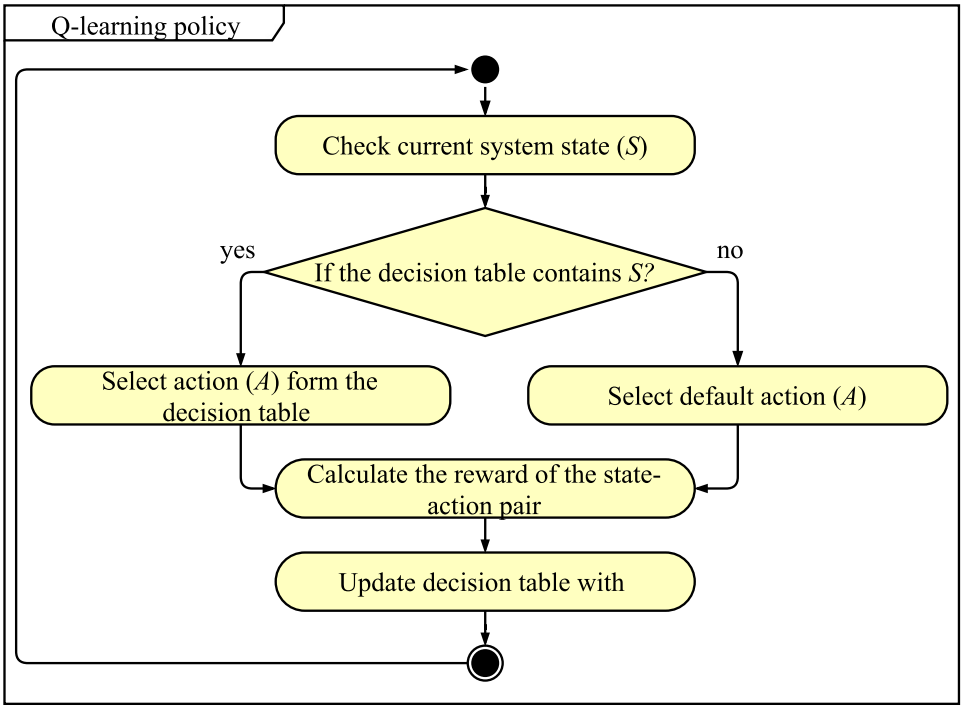


Figure 3.4: The operational flow for Q-learning policy.

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

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 Padala's work [35] 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 calculating the score for workflow performance, we use $p_t$ (average value of workflow's response time normalized by the deadline in this work) as the main metric. According to the formula $f(s_t, a_t)$, a 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 short, we would like to observe that our system can use less resource to process the workflows. By combining the workflow performance and resource utilization, the reword func-

tion can balance the trade-off between user-experienced performance and resource utilization.

### 3.7.3 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 3.3. 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 decisions. However, using only the feedbacks 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 trains 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).

# Chapter 4

# The Configuration and 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 chapter 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 Goals

Portfolio scheduler is an advanced scheduler which can be equipped with a variety of scheduling policies. ANANKE can also be tuned on a set of parameters to meet different performance requirements. To meet the deadline constraint requirement, we tune ANANKE on several parameters and elaborate these configurations which in the following sections. The configurations decision includes policy combinations equipping the portfolio scheduler, the utility function used in Q-learning based portfolio scheduler. Besides the decision on configurations, there are details need to be demonstrated when implementing the decision table which is the core component of the Q-learning policy.

## 4.2 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,

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. |

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 policy, a workflow-selection policy, and a client-selection policy. 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:

### 4.2.1 Provisioning Policies

Provisioning policies can vary significantly in how aggressively they change the number of resources. We select four significantly different provisioning policies, adapt them to use threads as the smallest processing unit, and summarize their operation in Table 4.1. ODA [16] is the most aggressive in the set, as it always ensures that the system has enough *idle* resources for waiting workflows. ODB [16] is less aggressive, as it just ensures that the system has enough resources, whether busy or idle. AQTP [32] 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.

### 4.2.2 Allocation: Workflow-Selection Policies

Which workflow to be selected next for execution is a typical question in multi-workflow scheduling systems. We select four workflow-selection policies for our

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. |

portfolio and summarize their operations in Table 4.2. Besides the typical LCFS policy, the other policies 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.

### 4.2.3 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.

## 4.3 Operational Flow for Selecting the Combination of Policies

During the execution of workloads, 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

Table 4.4: Explanation for the symbols used in the 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 utility function) |
| $\alpha$ | $\alpha$ is used to emphasize the urgency of the tasks. ($\alpha$ is user defined) |
| $\beta$ | $\beta$ is applied to stress the efficiency of resource usage. ($\beta$ is user defined) |
| $P$ | Workflow throughput in one single simulation. |
| $P_{max}$ | Estimated maximum value of workflow throughput in one single simulation. |
| $W_{avg}$ | Estimated average of workflow 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. |

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 is periodically applied until the workload is 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 Chapter 4.4) is used for the calculation. The decision maker, therefore, ranks the triplets discerningly according to 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.

## 4.4 Utility function as selection criteria

Users can define selection criteria as a utility function. The utility function is a formula which evaluates the combination of policies by considering both user-oriented metrics and system-oriented metrics. Thus we focus on task throughput and CPU utilization which are combined in the following formula:

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

Table 4.5: Constant Values used in the Numerical Analysis

| Metric | Value |
|---|---|
| $P$ | 16.78 (workflows/s) |
| $P_{max}$ | 17.1 (workflows/s) |
| $W_{avg}$ | 7.83 (s) |
| $C$ | 51.14 (%) |
| $C_{max}$ | 82.94 (%) |

$k$, $\alpha$, and $\beta$ are scaling factors. $P$ is the workflow throughput. $W_{avg}$ is the average workflow waiting time. $C$ is the CPU usage. $\alpha$ and $\beta$ 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 consider the waiting time of a workflow to have control over the response time.

To analyse the relationship between $S$ and the scaling factors ($k$, $\alpha$ and $\beta$), we conduct the numerical analysis and show the result in Figure 4.2. We set $P$, $P_{max}$, $W_{avg}$, $C$ and $C_{max}$ to the certain constant values listed in Table 4.5. Under this condition, $S$ is directly proportional to $k$ and inversely proportional to $\alpha$ and $\beta$.

## 4.5 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 stored 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 3.7.2. 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 to zero. When making a decision at time $t$, our policy finds the row which represents the current state $s_t$ and , 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 $q'$:

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

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

where $\alpha$ and $\beta$ 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.

### 4.5.1 Decision Table implementation

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 larger 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 is 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 2.4, 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 apply the following table configurations.

1. **State having more information (MD) vs. State having less information (LD)**
   The *State* function uses three metrics $u_t$, $v_t$, and $y_t$. By definition, $v_t$ and $y_t$ have upper boundaries, and $u_t$ doesn't. $u_t$ in principle can take any positive integer value or 0. To avoid $u_t$ becoming an infinite value, we normalize it by the maximal number of threads $n_{max}$. The normalized configuration guarantees that the decision table converges after a finite number of iterations. Therefore, we call the configuration with normalized $u_t$ as the state with less data set (LD) and the opposite configuration (without the normalization) as the state with more data set (MD).

2. **10 as the maximum action value (QA10) vs. $2^n$ as the maximum action value (QA$2^n$)**
   More threads may be leased or terminated when the resource increases. As mentioned in Chapter 3.7.2, 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 avoid $m$ increasing 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 4.4 to restrict the value of $m$ in the case that large resource is needed to

be scaled.

$$k = \begin{cases} k, & \text{for } k < 10 \\ 10, & \text{for } 10 \leq k < 2^4 \\ 2^i, & \text{for } 2^i \leq k < 2^{i+1} \ (i \geq 4) \end{cases} \tag{4.4}$$

3. **Setting for policy size**
   The decision table is solely trained by the data generated during the simulation. Since 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 4.6 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.

Table 4.6: 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 |

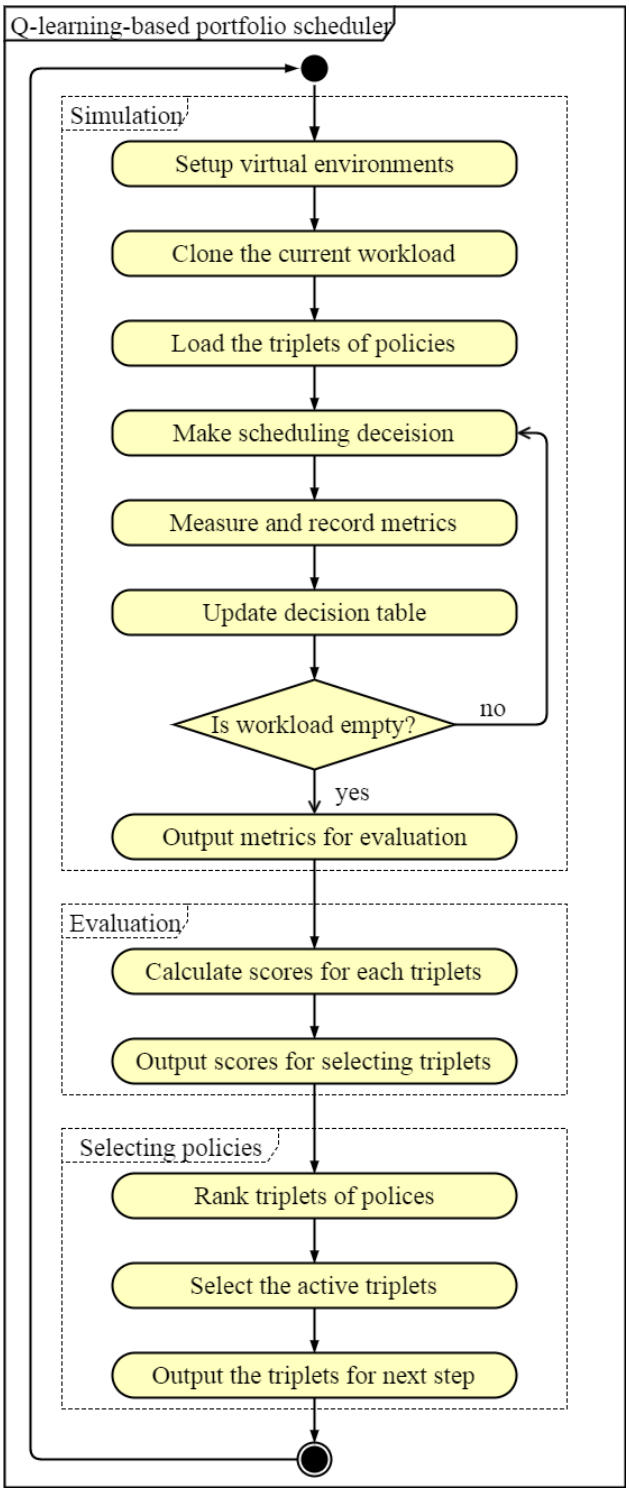


Figure 4.1: The operational flow for simulation, evaluation and policies selecting.

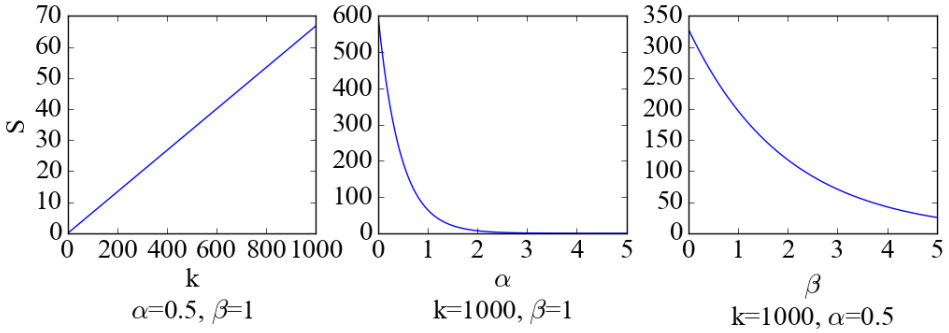


Figure 4.2: We set the $P, P_{max}, W_{avg}, C, C_{max}$ to be constant values and conduct numerical analysis for $k$, $\alpha$, and $\beta$.

| State \ Action | (0, none) | (1, up) | (3, down) | ... | ($m$, down) |
|---|---|---|---|---|---|
| (0, 0, 0) | X | X | X | ... | X |
| (0.1, 0.1, 0.1) | X | X | X | ... | X |
| ... | ... | ... | ... | ... | ... |
| ($u_t$, $v_t$, $y_t$) | X | X | X | ... | X |

Figure 4.3: The decision table for the Q-learning policy. Columns represent actions and rows represent states of the environment. Xs stand for the state-action weights which are generated and updated at runtime.

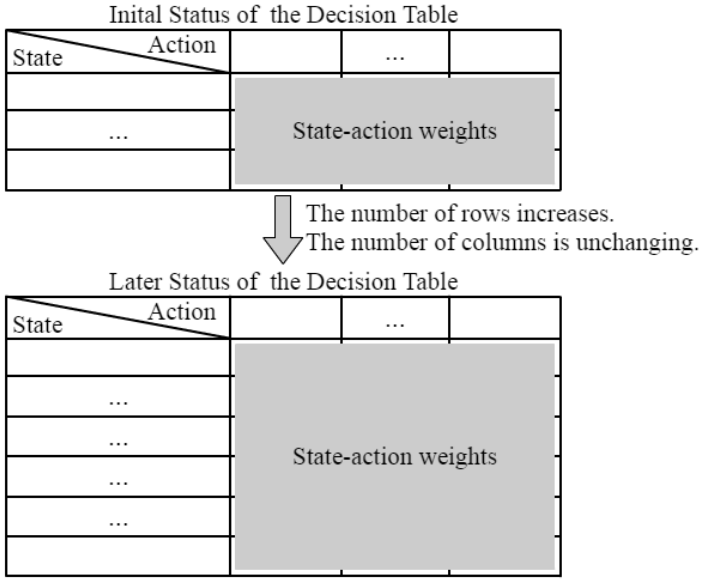


Figure 4.4: The number of the decision table's rows increases during the training under the MD configuration.

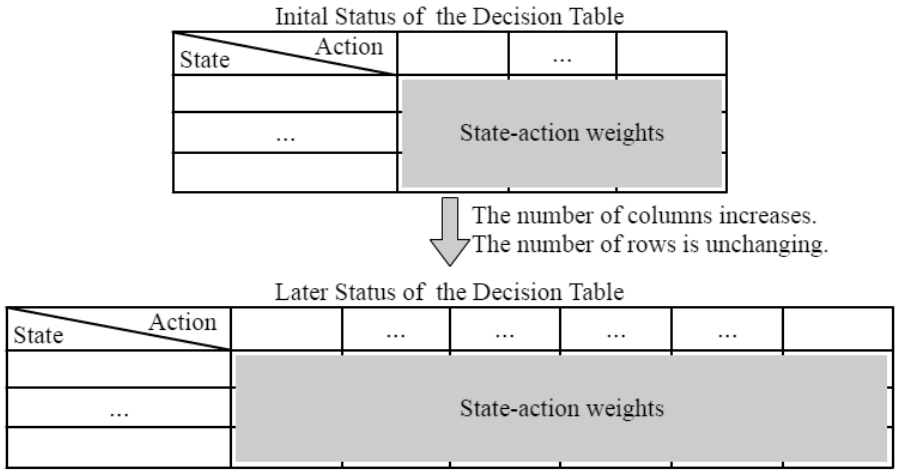


Figure 4.5: The number of the decision table's column increases during the training under the QA2$^n$ configuration.

# Chapter 5

# Experiment Setup

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

## 5.1 The Goals

During the experiment, we evaluate the comprehensive performance of ANANKE under different cloud-based environment and study the impact of different decision table configurations on workflow performance. To cover the study goals, we design several experiments and catalog them into four groups. The first group contains comparative experiments between ANANKE and current Chronos system for evaluating the user-interesting performance. The second group consists of comparative experiments between 5 different auto-sclaers (including ANANKE) for elasticity studying. The third group is composed of comparative experiments between 3 sets of decision tables configurations and study the impact of different configurations on workflow performance. The last group consists of comparative experiments on large scale resource and evaluate ANANKE's performance when managing larger scale resources. The main goal of this chapter is to provide the detailed configuration used to conduct the designed experiments.

## 5.2 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

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 |

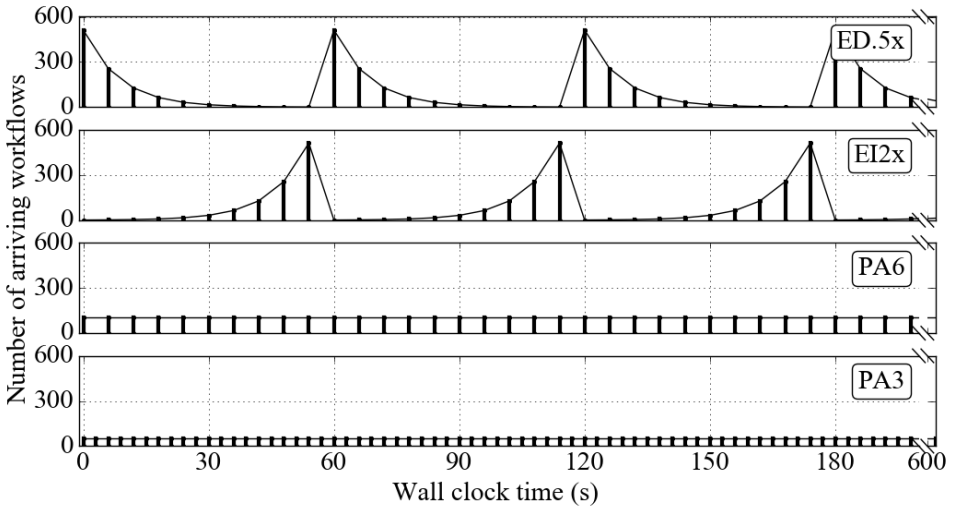


Figure 5.1: Four different patterns: ED.5x and EI2x are dynamic workloads, PA6 and PA3 are static workloads. (The horizontal axis only shows a 190 s-cut from every complete workload.)

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 a 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 ten 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 one workflow every 3 and 6 seconds, respectively.

## 5.3 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 [7]. For our experiments, we use 1 cluster of the six 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).

*Cloud-deployment models:* We conduct experiments using three cloud-deployment models, each of which uses one master node, but different amounts and management of client nodes. The *private cloud mode* uses three 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.

## 5.4 Metrics to Compare ANANKE and Its Alternatives

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

### 5.4.1 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. 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.

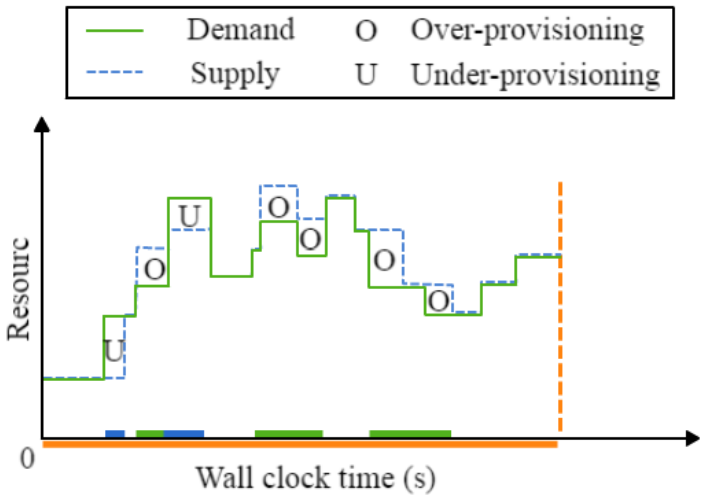


Figure 5.2: The supply and demand curves illustrating the under- and over-provisioning periods [21].

### 5.4.2 Resource Utilization

Resource utilization is an important aspect when evaluating a resource management and dynamic scheduling (RMS) system. We use as metric the *number of active (used) threads*. Because each client node can run up to 70 threads in ANANKE, a lower amount of threads (busy and idle) in the system leads to piece-wise linearly lower operational costs. We sample and collects the number of active threads at every decision-making time point. An overview analysis of the resource utilization during the experiment is based on these data.

### 5.4.3 Elasticity

To evaluate elasticity, we adopt the metrics and comparison approaches introduced in 2017 by the SPEC Cloud Group [21].

*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 [28]. (In particular, the near-deadline workflows are those whose sum of waiting time and estimated running time is close to the deadline).

*Elasticity metrics:* We adopt the suite of metrics proposed by the SPEC Cloud Group [21], 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 met-

rics: 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 the 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. According to the definition, all the elasticity metrics mentioned above are between 0 to 1. Figure 5.2 shows how the mismatches between the supply and demand curves cause under- and over-provisioning states.

*Comparing multiple auto-scalers:* To perform a comparison including all system- and user-oriented metrics, we use the two numerical approaches proposed for use by the SPEC Cloud Group [21]: *Pairwise Comparison* [14] and the *Fractional Difference Comparison* [21]. 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, which for each metric is ascribed 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 the ideal. Last, we rank auto-scalers inversely, such that the lowest accumulated value indicates the best auto-scaler.

## 5.5 Auto-scalers Considered for Comparative Evaluation

In our experiments, we analyze the elasticity of ANANKE and other auto-scalers. We compare experimentally ANANKE with the following alternatives:

### 5.5.1 Existing Baseline

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

### 5.5.2 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 a prototype and 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 [16]): Vertical auto-scaling by the standard portfolio scheduler, and (`PS`) no autoscaling or (`PS-VR`) horizontal auto-scaling by the React policy [11]. React is a top-performing auto-scaler for horizontal elasticity and workflows [21].
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.

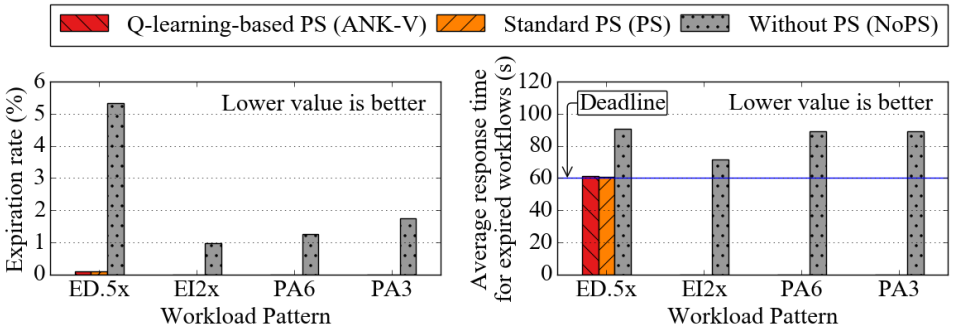


Figure 6.1: The expiration rate and the response time of expired workflows with different workloads. (Lower values are better.)

# Chapter 6

# Experimental Results

In this chapter, we evaluate and validate the design choices we made for ANANKE (in Chapter 3). As mentioned in Chapter 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 Overview

We conduct experiments from four aspects including evaluation of scheduler impact, elasticity, decision table configuration impact and large scale performance. The configurations of the experiments are listed in Table 6.1

Table 6.1: Overview of the experiments

| Environment | Workload | Measured metrics | #Policies | Baselines |
|---|---|---|---|---|
| Scheduler Impact on Workflow Performance | | | | |
| private cloud mode | ED.5x, EI2x, PA6, PA3 | expired rate, workflow waiting time, workflow response time | 36 | ANK-V, PS, NoPS |
| Evaluation of Elasticity and Resource Utilization | | | | |
| public cloud mode | ED.5x, EI2x, PA6, PA3 | supply, estimated demand, workflow response time | 36 | PS-(VR), ANK-VH, ANK-V, NoPS, Static |
| Decision Table Configuration Impact on Workflow Performance | | | | |
| public cloud mode | ED.5x, EI2x, PA6, PA3 | workflow waiting time | 36, 3–16 | ANK-VH |
| Analysis at 10× Larger Scale | | | | |
| scalability mode | ED.5x, EI2x, PA6, PA3 | the number of threads | 24 | ANK-VH, PS-(VR) |
| Analysis of Transition in the Selected Combination of Policies | | | | |
| private cloud mode | ED.5x, EI2x, PA6, PA3 | the selected combination of policies | 36 | ANK-VH |

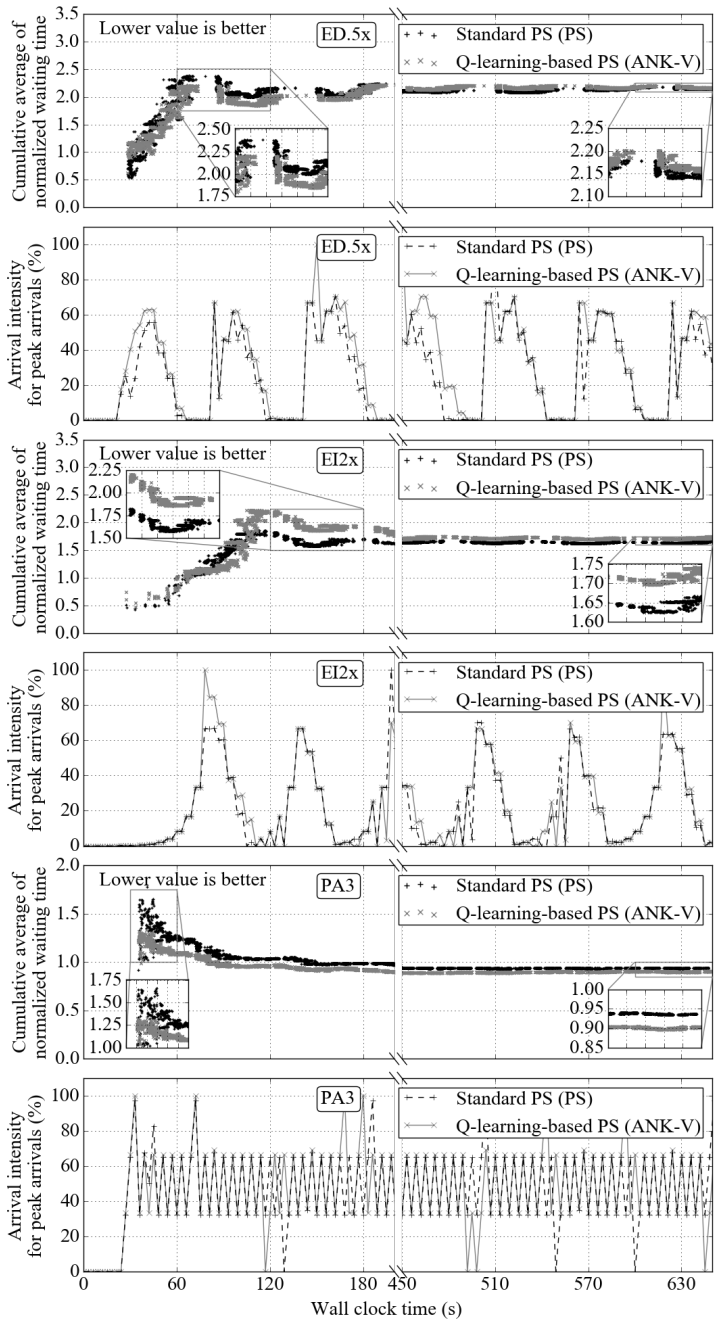


Figure 6.2: The normalized response time of the Q-learning-based portfolio scheduler (`ANK-V`) and of the standard portfolio scheduler (`PS`). Workload: (top pair of plots) dynamic (`ED.5x`),(middle part of plots) dynamic (`EI2x`), and (bottom pair of plots) static (`PA3`).

## 6.2 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 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 the standard portfolio scheduler. 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, especially at the begining of the experiment, 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.

## 6.3 Evaluation of Elasticity and Resource Utilization

To assess the elasticity, we conduct experiments in the *public-cloud mode*. As explained in Chapter 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)

Table 6.2: Calculated metrics for all of the considered auto scalers under ED.5x workload.

| ED.5x | $\overline{a}_U$ | $\overline{a}_O$ | $t_U$ | $t_O$ | $i$ | $i'$ | $S_e$ | $\overline{V}$ | $\overline{T}$ |
|---|---|---|---|---|---|---|---|---|---|
| PS-(VR) | 0.10 | 1.18 | 0.48 | 0.49 | 0.28 | 0.23 | 2.22 | 200 | 17.23 |
| ANK-VH | 0.28 | 0.30 | 0.71 | 0.26 | 0.44 | 0.27 | 2.69 | 181 | 15.39 |
| ANK-V | 0.05 | 4.94 | 0.31 | 0.67 | 0.38 | 0.25 | 1.69 | 274 | 17.65 |
| NoPS | 0.03 | 3.99 | 0.27 | 0.73 | 0.46 | 0.21 | 1.35 | 313 | 17.48 |
| Static | 0.02 | 11.06 | 0.28 | 0.73 | 0.68 | 0.30 | 1 | 349 | 17.35 |
| Ideal | 0.02 | 0.30 | 0.27 | 0.26 | 0.28 | 0.21 | 1 | 181 | 17.65 |

Table 6.3: Calculated metrics for all of the considered auto scalers under EI2x workload.

| EI2x | $\overline{a}_U$ | $\overline{a}_O$ | $t_U$ | $t_O$ | $i$ | $i'$ | $S_e$ | $\overline{V}$ | $\overline{T}$ |
|---|---|---|---|---|---|---|---|---|---|
| PS-(VR) | 0.05 | 1.01 | 0.49 | 0.58 | 0.25 | 0.20 | 2.04 | 186 | 16.19 |
| ANK-VH | 0.22 | 0.39 | 0.65 | 0.28 | 0.36 | 0.24 | 2.77 | 161 | 16.19 |
| ANK-V | 0.01 | 3.51 | 0.13 | 0.83 | 0.25 | 0.39 | 1.73 | 249 | 16.24 |
| NoPS | 0.01 | 3.48 | 0.10 | 0.90 | 0.38 | 0.46 | 1.31 | 293 | 16.25 |
| Static | 0.01 | 13.08 | 0.13 | 0.88 | 0.33 | 0.65 | 1 | 348 | 16.23 |
| Ideal | 0.01 | 0.39 | 0.10 | 0.28 | 0.25 | 0.20 | 1 | 161 | 16.25 |

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 Chapter 5.4.3.

The comprehensive comparison is based on the metrics described in Chapter 5.4.3 which include *over- and under-provisioning accuracy* ($\overline{a}_U$, $\overline{a}_O$), *over- and under-provisioning timeshare* ($t_U$, $t_O$), two types of *instability* ($i$, $i'$), the *average workflow throughput* ($\overline{T}$), the *average number of used threads* ($\overline{V}$), and the *slowdown in workflow response time* ($S_e$). Table 6.6 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 configurations in both static and dynamic workloads. From the results in Figure 6.2 and Table 6.6 we can conclude that the reduction in the number of utilized resources often leads to the user-experienced performance de-

Table 6.4: Calculated metrics for all of the considered auto scalers under PA6 workload.

| PA6 | $\overline{a}_U$ | $\overline{a}_O$ | $t_U$ | $t_O$ | $i$ | $i'$ | $S_e$ | $\overline{V}$ | $\overline{T}$ |
|---|---|---|---|---|---|---|---|---|---|
| PS-(VR) | 0.04 | 0.42 | 0.35 | 0.60 | 0.20 | 0.37 | 1.41 | 171 | 16.86 |
| ANK-VH | 0.33 | 0.04 | 0.78 | 0.16 | 0.33 | 0.34 | 3.06 | 158 | 15.71 |
| ANK-V | 0.02 | 1.95 | 0.02 | 0.96 | 0.29 | 0.35 | 1.48 | 218 | 16.73 |
| NoPS | 0.00 | 2.21 | 0.02 | 0.98 | 0.32 | 0.44 | 1.12 | 224 | 16.81 |
| Static | 0.00 | 11.28 | 0 | 1.01 | 0.41 | 0.54 | 1 | 330 | 17.00 |
| Ideal | 0.00 | 0.04 | 0 | 0.16 | 0.20 | 0.34 | 1 | 158 | 17.00 |

Table 6.5: Calculated metrics for all og hte considered auto scalers under PA3 workload.

| PA3 | $\overline{a}_U$ | $\overline{a}_O$ | $t_U$ | $t_O$ | $i$ | $i'$ | $S_e$ | $\overline{V}$ | $\overline{T}$ |
|---|---|---|---|---|---|---|---|---|---|
| PS-(VR) | 0.06 | 0.40 | 0.44 | 0.50 | 0.30 | 0.37 | 1.39 | 169 | 16.78 |
| ANK-VH | 0.35 | 0.01 | 0.85 | 0.10 | 0.33 | 0.31 | 2.90 | 160 | 16.54 |
| ANK-V | 0.01 | 2.05 | 0.07 | 0.90 | 0.27 | 0.34 | 1.39 | 212 | 16.82 |
| NoPS | 0.00 | 2.06 | 0.02 | 0.99 | 0.23 | 0.43 | 1.09 | 228 | 16.78 |
| Static | 0 | 11.17 | 0 | 1.01 | 0.49 | 0.50 | 1 | 330 | 17.03 |
| Ideal | 0 | 0.01 | 0 | 0.10 | 0.23 | 0.31 | 1 | 160 | 17.03 |

Table 6.6: The results of the pairwise and fractional comparisons. The winners are highlighted in bold.

| Auto Scaler | 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 |

gradation. 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 a 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.

## 6.4 Decision Table Configuration Impact on Workflow Performance

In this section, we discuss our findings of 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 workflow waiting time normalized by the execution time to represent the user-experienced performance.

### 6.4.1 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 Chapter 4.5.1 and measure the workflow waiting time and normalize it by the execution time. Figure 6.5 shows the normalized workflow 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 the workflow waiting time by 15.9% under the ED.5x workload. Under the EI2x workload, the LD configuration reduces the workflow waiting time 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 workflow waiting time converge to the same value faster under a static workload. This observation indicates that the state configuration also influences the convergence speed of the the workflow waiting time.

We explain this as follows: 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 potential state values. A larger decision table requires more time and data for its training and thus causes more inaccurate scheduling decisions. Scheduler with LD configuration performs better under a dynamic workload. However, the system is more stable and the number of potential state values is less if a static workload is given. No matter which state configuration is used, the decision table can always be trained in a short period with a static workload. 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 from the observations that smaller value space of system state leads to better performance under dynamic workloads.

### 6.4.2 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 the workflow performance. The results are shown in Figure 6.6. We cannot observe any significant differences in the normalized workflow waiting time. Taking four workloads into account, the QA10 configuration reduces the normalized workflow waiting time by 5.5%–9.7%. The normalized waiting time with different action configurations has similar values 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 workflow waiting time

### 6.4.3 Policy Pool Size Impact on Workflow Performance

The final hypothesis we would like to investigate is the influence of the policy pool size on the user-experienced performance. We use the configurations from Table 4.6. 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 workflow waiting time. Under the ED.5x and EI2x workloads, no strong correlation between the normalized workflow waiting time and the size of the policy pool can be found. Under the PA6 and PA3 workloads, the normalized workflow waiting time is inversely proportional to the number of policies. The bigger size of the policy pool leads to lower values of normalized workflow waiting times. We can conclude that for static workloads the Q-learning-based portfolio scheduler with bigger policy pool has a smaller workflow waiting time. However, such a relation is not observed under a dynamic workload.

## 6.5 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 Chapter 5.3). We only use `PS-(VR)` as the baseline in this experiment. 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 mostly the required number of resources, even at 10 times larger scale than the current practice.

## 6.6 Analysis of Transition in Selected Combination of Policies

As mentioned in Chapter 3.7, the Q-learning-based portfolio scheduler dynamically select the combination of policies. To analyze the transition in selected combination of policies, we track the selected combination during the experiment and present one particular result (under ED.5x workload) in Figure 6.9. According to the observations, the Q-learning-based portfolio scheduler selects different combination in runtime. We also find that not all the equipped policies have been used in the experiment. As Figure 6.9 shows, 28 out of 36 policies-combinations have been selected during the experiment. Similar results are also observed in the experiments with EI2x, PA3, and PA6 workloads.

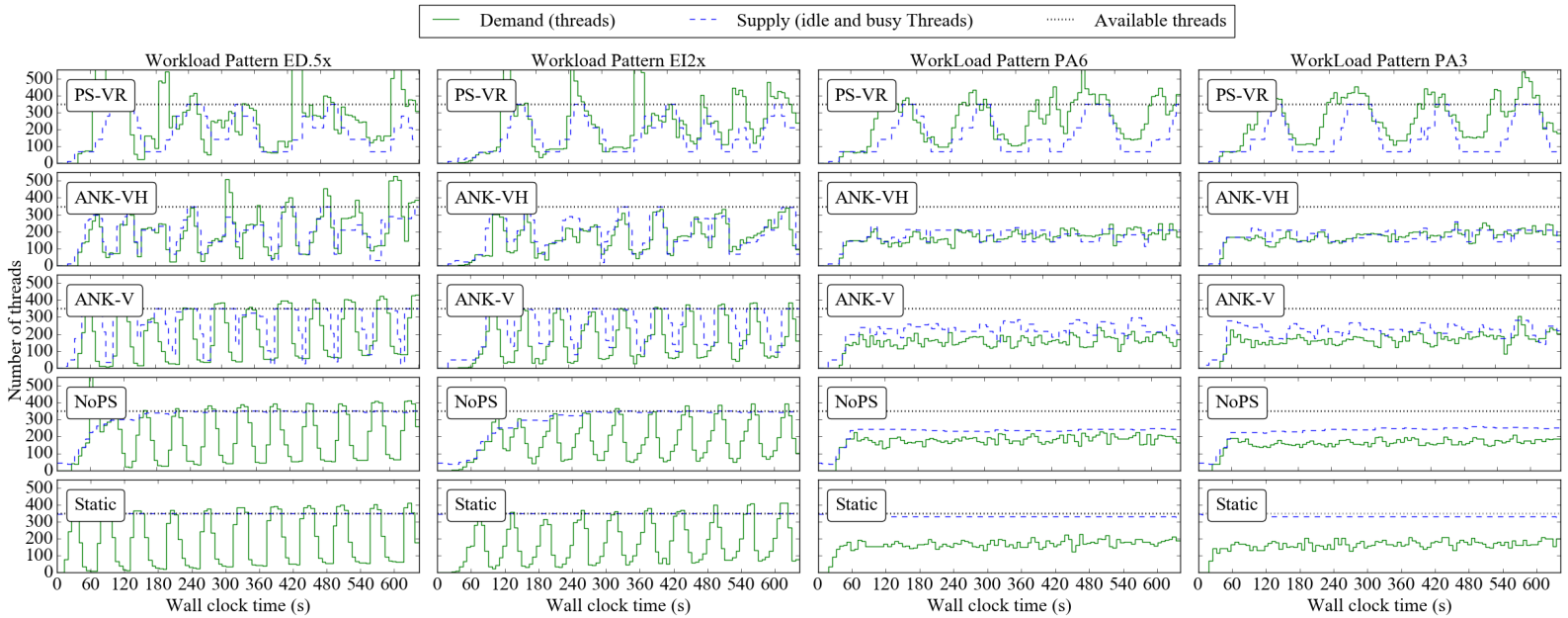


Figure 6.3: The supply and demand curves for five different auto-scalers, under (left) dynamic and (right) static workloads. The "Available threads" horizontal line indicates the resource limit of 350 threads. The best performance for each workload is highlighted with a red box.

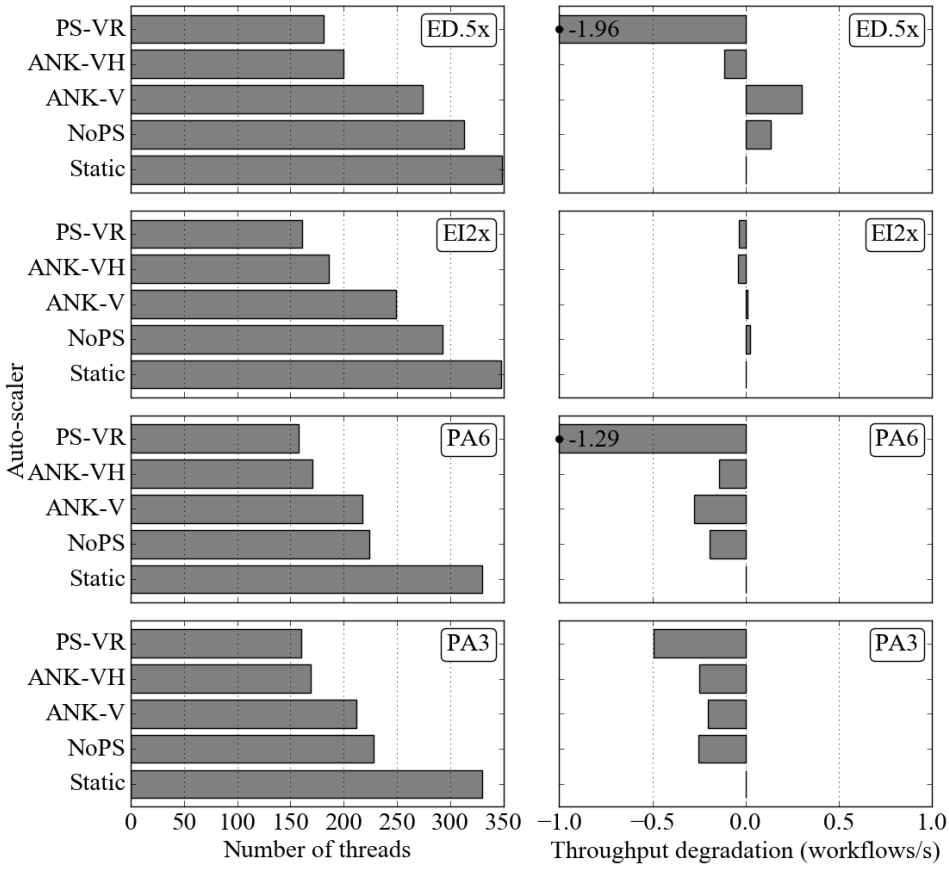


Figure 6.4: The average number of used threads during the experiment and the average throughput degradation. The baseline is the static case without an auto-scaler.

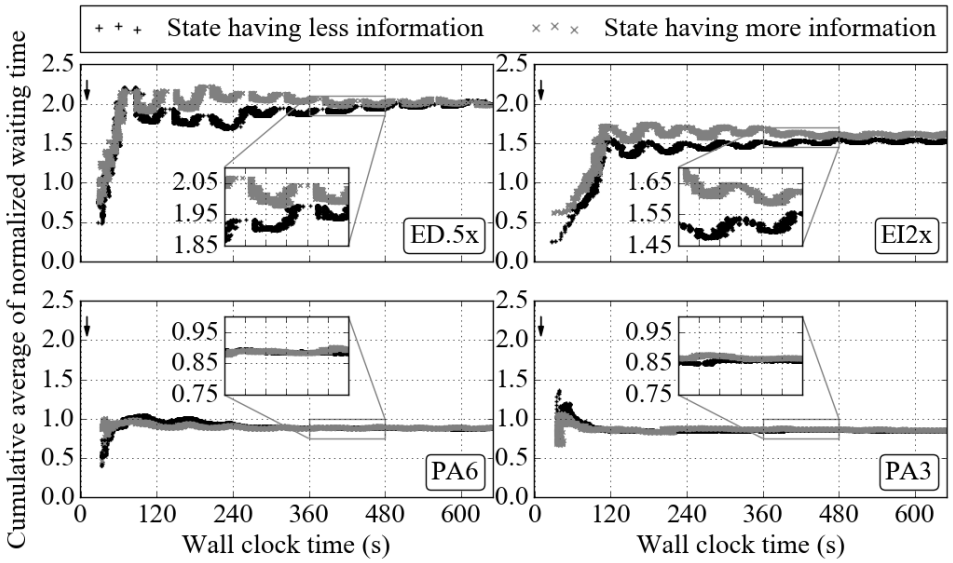


Figure 6.5: Normalized workflow waiting times for the *state with more information* (MD) and the *state with less information* (LD) configurations. The arrow at the left upper corner indicates that the lower value is better. The upper two cases use the dynamic workloads; the bottom two use the static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

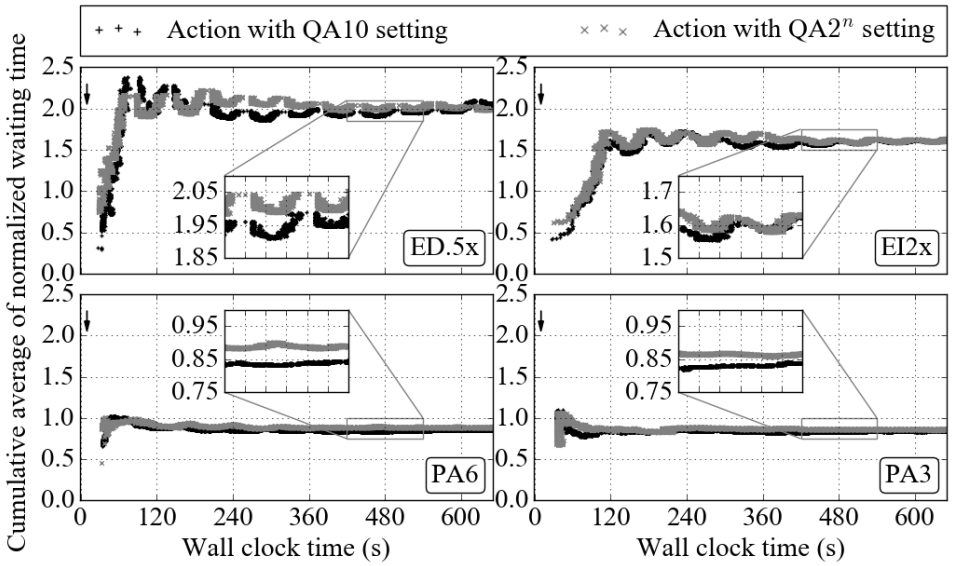


Figure 6.6: Normalized workflow waiting time for QA10 and QA2$^n$ configurations. The arrow at the right upper corner indicates that the lower value is better. The upper two cases use the dynamic workloads; the bottom two use the static workloads.

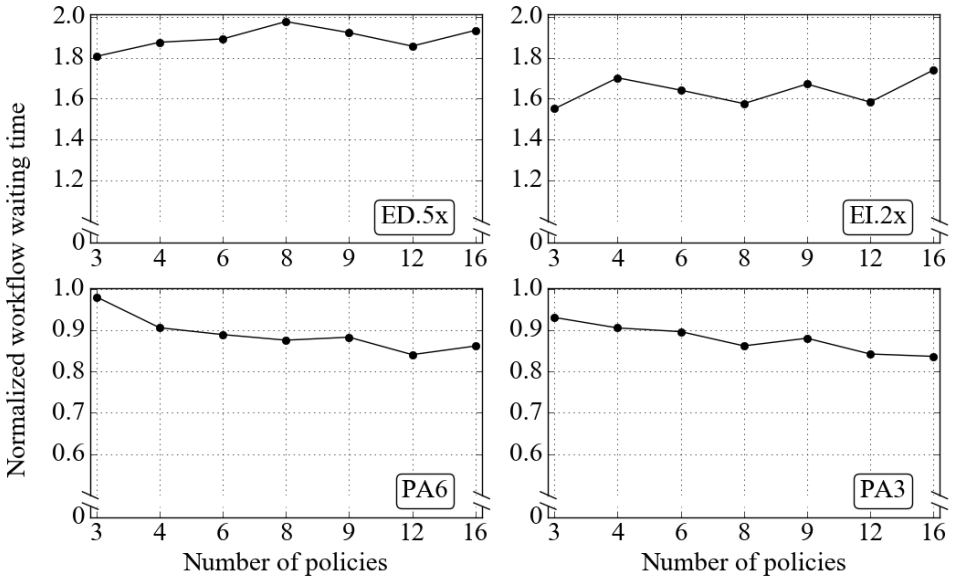


Figure 6.7: The vertical axis is the workflow waiting time normalized by the execution time and the horizontal axis represents the size of the policy pool varying from 3 to 16.

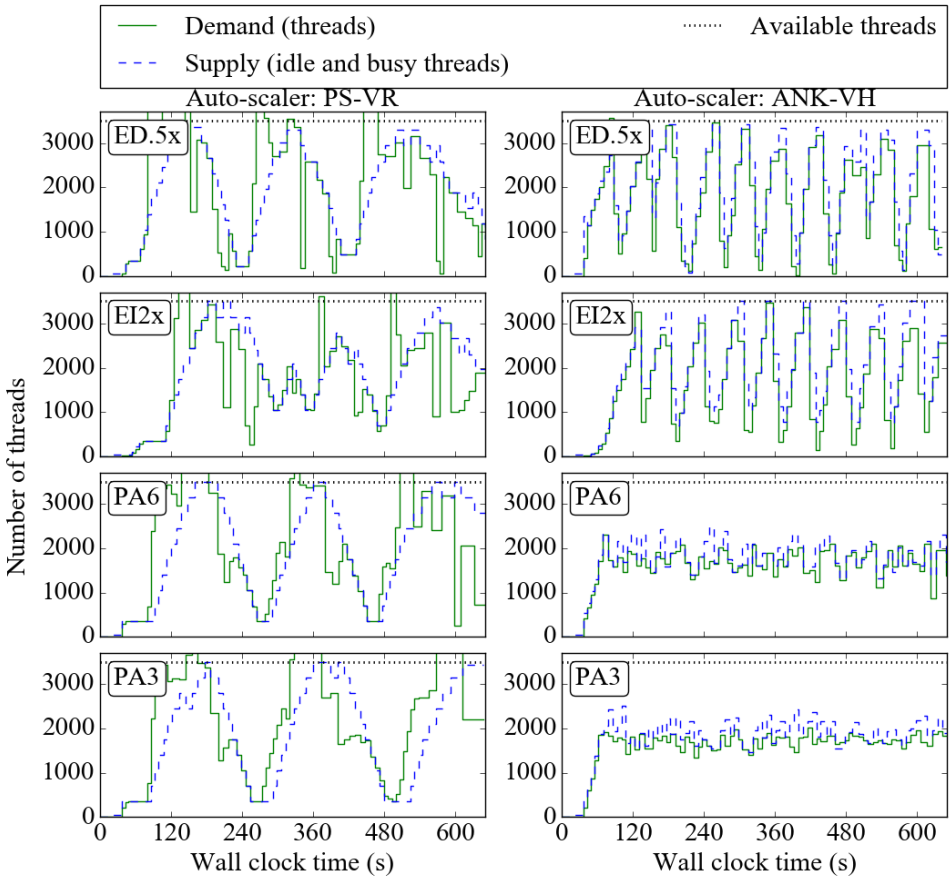


Figure 6.8: Comparison when auto-scaling at large scale between: (left) a simple reactive mechanism, and (right) ANANKE.

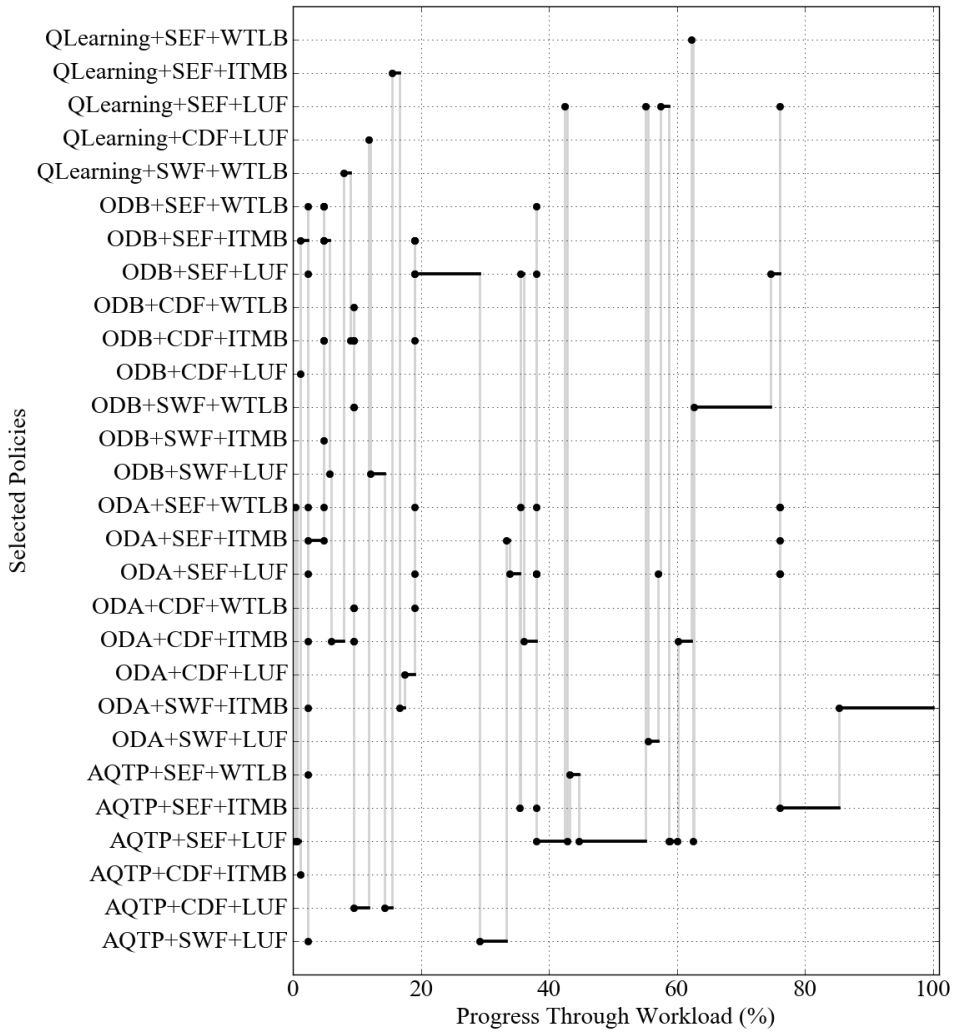


Figure 6.9: Transition in Selected Combination of Policies when preforming scheduling.

# Chapter 7

# Conclusion and Future Work

After all the elaborations of our design and the experimental evaluations. We present the summary of our study and the research findings. In this Chapter, we also discuss the potential future works.

## 7.1 Conclusion

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 the 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. In the problem statement we present three main questions that we investigate and answer throughout this work:

1. **How to adapt the concept of portfolio scheduling to the RM&S framework used in production environment?**
   We study the current approach used by The Smart Connect Team and investigate the functional and non-functional requirements. To answer the question, We thus designed ANANKE, an architecture for RM&S that uses cloud resources for its operation. ANANKE is a variant of the current framework (Chronos) which can use portfolio scheduler to provisioning resource and allocate workflows.

2. **How to use the historical information when performing portfolio scheduling?**
   To take advantage of the historical information, learning techniques can be applied when performing scheduling. We investigate many learning techniques used in task and resource management. We find Q-learning meet our requirements and design a learning-based scheduling policy using the Q-learning technique. We further present a design of a learning-base portfolio

scheduler which can be equipped with the Q-learning policy.

3. **How to evaluate the adapted learning-based portfolio scheduler, experimentally, through the implementation of a prototype?**
   We implement ANANKE as a prototype of the production environment at Shell and conducted real-world experiments. We also design and select various threshold-based heuristics as scheduling policies for the Q-learning based portfolio scheduler. The experimental results allow us to analyze the inter-dependencies between the parametrization of Q-learning and portfolio scheduling, moreover, discover how the size of the decision table and the size of the scheduling policy pool affects the performance. We also compare 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 demonstrate that using Q-learning policy in portfolio scheduler can achieve better performance in user experience, resource utilization, and auto-scalability under a static workload. When given workload with high fluctuation the difference between the performance of learning based portfolio scheduler and stander portfolio scheduler is not distinct. Moreover, we show our finding in the relation between parameter setting and user-interested performance. The space size of decision table and the size of policy pool influence the performance under a static workload. The study focusing on the internal setting provides some instructions on using learning based portfolio scheduler for various workload and different SLOs.

## 7.2 Future Work

In this work, We design and implement a learning-based portfolio scheduler adapting the concept of portfolio scheduling and reinforcement learning. There are four different topics where we can extend our work.

1. **Learning techniques**
   In this work, we only focus on Q-learning as the main learning technique for a specified workload type (sensor data processing). However, other learning techniques are preferable if the workload or requirements change. In the future, we plan to extend our work with considering other reinforcement learning techniques, such as error-driven learning and temporal difference learning. We tend to apply the learning-based portfolio scheduler with different learning techniques on an extended set of workloads and SLOs and find the limitation of the learning based portfolio scheduler in different scenarios.

2. **Simulator**
   The learning-base portfolio scheduler designed and implemented in this work

only use an event-driven based simulator. However, other types of simulator such as time-driven based and data-driven based simulator can be adapted to meet different time-constraint and accuracy requirements. In the further work, we tend to apply other types of the simulator in portfolio scheduling with learning techniques. By comparing the simulator impact on system-oriental and user-oriental performance, we want to explore the use cases of learning-based portfolio scheduler equipped with different type of simulator.

3. **Optimizations of simulation**
   One of the limitations of the event-driven based simulator is that the simulating time increase with the size of workload linearly. In the $10\times$ Larger Scale experiment, we find that the simulation process becomes time-consuming when intensive workloads are given due to the limitation of event-driven based simulator. An additional mechanism is needed to control the simulating time to meet time-constraint SLOs. We apply a simple mechanism (reducing the size of police pool) to control the simulation time. There are other advanced mechanisms can be applied to optimize the simulation and reduce simulation time. However, reducing the simulation time may influence the accuracy of the final scheduling decision since fewer metrics and data are considered during the simulation. As a consequence, the trade-off between accuracy and simulating time need to be carefully considered. In the future, we plan to apply different optimization mechanisms on portfolio scheduler and speed up the decision-making time to meet more SLOs of real-time systems.

4. **Hybrid cloud environment**
   In this work, we evaluate ANANKE either on private cloud environment or public cloud environment. However, hybrid environment combining private and public cloud resource are also adapted in the industry to ensure the confidentiality and reduce operational costs at the same time. Workflows involving confidential data can be processed on the private cloud and other workflows and be allocated on the public cloud resource. In the future, we tend to adapt the learning based portfolio scheduler on hybrid cloud resource and explore the limitation of the scheduler in this scenario.

# 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] Hamid Arabnejad, Pooyan Jamshidi, Giovani Estrada, Nabil El Ioini, and Claus Pahl. An auto-scaling cloud controller using fuzzy q-learning - implementation in openstack. In *ESOCC*, pages 152–167, 2016.

[5] Ankita Atrey, Hendrik Moens, Gregory van Seghbroeck, Bruno Volckaert, and Filip De Turck. BRAHMA: an intelligent framework for automated scaling of streaming and deadline-critical workflows. In *CNSM*, pages 216–222, 2016.

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

[7] 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.

[8] 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.

[9] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *SPE*, 41(1):23–50, 2011.

[10] 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.

[11] 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.

[12] Delong Cui, Wende Ke, Zhiping Peng, and Jinglong Zuo. Multiple dags workflow scheduling algorithm based on reinforcement learning in cloud computing. In *ISICA*, pages 305–311, 2015.

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

[14] Herbert A David. Ranking from unbalanced paired-comparison data. *Biometrika*, 74(2):432–436, 1987.
[15] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and R. Kent Wenger. Pegasus, a workflow management system for science automation. *FGCS*, 46:17–35, 2015.
[16] 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.
[17] 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.
[18] Juan José Durillo and Radu Prodan. Multi-objective workflow scheduling in amazon EC2. *Cluster Computing*, 17(2):169–189, 2014.
[19] Marc Frîncu, Stéphane Genaud, and Julien Gossa. Comparing provisioning and scheduling strategies for workflows on clouds. In *IPDPSW*, pages 2101–2110, 2013.
[20] Bernardo A Huberman, Rajan M Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296), 1997.
[21] 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.
[22] Alexey Ilyushkin and Dick H. J. Epema. Towards a realistic scheduler for mixed workloads with workflows. In *CCGrid*, pages 753–756, 2015.
[23] Alexey Ilyushkin, Bogdan Ghit, and Dick H. J. Epema. Scheduling workloads of workflows with unknown task runtimes. In *CCGrid*, pages 606–616, 2015.
[24] 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.
[25] Alexandru Iosup, Omer Ozan Sonmez, and Dick H. J. Epema. Dgsim: Comparing grid resource management architectures through trace-based simulation. In *Euro-Par*, pages 13–25, 2008.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] 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.
[31] Ming Mao, Jie Li, and Marty Humphrey. Cloud auto-scaling with deadline and budget constraints. In *GRID*, pages 41–48, 2010.
[32] Paul Marshall, Henry M. Tufo, and Kate Keahey. Provisioning policies for elastic computing environments. In *IPDPS*, pages 1085–1094, 2012.
[33] Mohammad Masdari, Sima ValiKardan, Zahra Shahi, and Sonay Imani Azar. Towards workflow scheduling in cloud computing: A comprehensive analysis. *J. Network and Computer Applications*, 66:64–82, 2016.

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

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

[36] Zhiping Peng, Delong Cui, Yuanjia Ma, Jianbin Xiong, Bo Xu, and Weiwei Lin. A reinforcement learning-based mixed job scheduler scheme for cloud computing under SLA constraint. In *CSCloud*, pages 142–147, 2016.

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

[38] 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.

[39] 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.

[40] 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.

[41] Zhao Tong, Zheng Xiao, Kenli Li, and Keqin Li. Proactive scheduling in distributed computing - A reinforcement learning approach. *JPDC*, 74(7):2662–2672, 2014.

[42] 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.

[43] 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.

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

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

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