2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows

Zhe Yang*, Phuong Nguyen*, Haiming Jin†, Klara Nahrstedt*
*University of Illinois at Urbana-Champaign
†Shanghai Jiao Tong University
*{zheyang3, pvnguye2, klara}@illinois.edu
†{jinhaiming}@sjtu.edu.cn

***Abstract*—Microservice, an architectural design that decomposes applications into loosely coupled services, is adopted in modern software design, including cloud-based scientific workflow processing. The microservice design makes scientific workflow systems more modular, more flexible, and easier to develop. However, cloud deployment of microservice workflow execution systems doesn't come for free, and proper resource management decisions have to be made in order to achieve certain performance objective (*e.g.*, response time) within constraint operation cost. Nevertheless, effective online resource allocation decisions are hard to achieve due to dynamic workloads and the complicated interactions of microservices in each workflow. In this paper, we propose an adaptive resource allocation approach for microservice workflow system based on recent advances in reinforcement learning. Our approach (1) assumes little prior knowledge of the microservice workflow system and does not require any elaborately designed model or crafted representative simulator of the underlying system, and (2) avoids high sample complexity which is a common drawback of model-free reinforcement learning when applied to real-world scenarios. We show that our proposed approach automatically achieves effective policy for resource allocation with limited number of time-consuming interactions with the microservice workflow system. We perform extensive evaluations to validate the effectiveness of our approach and demonstrate that it outperforms existing resource allocation approaches with read-world emulated workflows.**

I. Introduction

Microservice design, which is often referred as a way to structure software applications as collections of loosely coupled services, has become increasingly popular in modern software system architectures. Different from the traditional monolithic approach, the microservice architecture offers many benefits, such as the ability to independently deploy, scale, and develop individual services in different programming languages. As a result, the applications become more modular, easier to understand, develop, and test. This also enables continuous delivery and deployment of applications (*e.g.*, upgrading and deploying individual services will not affect other services).

One example of the adoption of microservice architecture is in scientific workflow systems. Scientific workflow systems [20][12] have traditionally employed a monolithic approach, in which each workflow, represented by a directed acyclic graph of tasks, is implemented as a tightly coupled set of tasks and has its own workflow execution plan that specifies how to run the workflow on a distributed computation infrastructure. With the transition to the microservice design [27][26], each task is modeled as a microservice with its own request queue and computing capacity. Complex task dependencies are separated from the implementation of individual tasks, and thus, the microservice design enables more flexible and scalable workflow composition (*i.e.*, new workflows can be composed from existing tasks) and scheduling (*i.e.*, resources can be scheduled at the fine-grained level of tasks which could potentially take advantage of the parallelism in workflow structure to speed up workflow execution). A finer-grained workflow decomposition could also benefit load balancing among execution machines.[1]

With the increasing popularity of cloud infrastructure, there have been efforts to deploy microservice-based applications on the cloud to take advantage of the elasticity of cloud infrastructure (*e.g.*, cloud-based workflow systems [1][5]). While public cloud offerings often support the ability to scale microservices elastically and on-demand using their unlimited resources, real-world deployments of microservice applications are often constrained by limited operation cost. On the other hand, applications deployed on private cloud infrastructure are likely faced with bounded resource capacity. Therefore, in these scenarios it is important to be able to properly allocate constraint resources of microservices to meet certain performance objective (*e.g.*, response time, utilization).

However, resource allocation for microservice infrastructure is challenging due to (1) the variability of dynamic workloads, (2) complex interactions between microservices that can cause cascading effects when provisioning resources for interrelated microservices, and (thus) (3) the demand for either an elaborately crafted model depicting the microservice system, which necessitates sophisticated prior knowledge of the system, or a data-driven identification model requiring a lot of training data (*i.e.,* resource allocation feedback) from the real microservice system, which potentially consumes tremendous amount of resources. There has been extensive related work on resource allocation for cloud-based system (see Section VII). However, these related approaches either (i) are rule-based, heuristics

[1]Another example of microservice is in serverless computing [11].

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00021

approaches, which depend on complete understanding and/or an elaborate model of the cloud system, or (ii) do not tackle with dynamic system status.

In this paper, we propose a novel self-adaptive resource adaptation approach, MIRAS, for microservice system that assumes little prior knowledge of its infrastructure. We take the *microservice workflow system* (to be introduced in Section II) as the target environment where we perform resource adaptation and leverage recent advances in *model-based reinforcement learning* to design a control policy for microservice workflow system using past experience of interacting with the microservice infrastructure. We choose reinforcement learning because it directly optimizes long-term reward in dynamic environment.

Recently, there have been some related publications that applied reinforcement learning (RL) to various applications. However, these efforts are mostly based on the *model-free* RL technique that tends to suffer from high sample complexity, and thus, hinders its use in real-world domains where it is often time-consuming and costly to obtain samples of interaction between RL agent and environment, distributed and cloud-based systems being an example. As a result, their applications have been mostly limited to where simulated environments were available. In contrast, MIRAS utilizes a learnt model of the microservice environment to assist policy learning, achieving much higher sample efficiency than the model-free approaches. We are *the first* to use model-based RL in the distributed and cloud-based system, and we demonstrate its effectiveness in tackling real-world tasks where low sample complexity is crucial.

MIRAS consists of two main components: environment model learning and policy optimization. In model learning, we train a neural network model to capture the behavior of the microservice infrastructure using samples collected from the real interactions between RL agent and the environment. We further refine the model to facilitate policy learning. In policy optimization, we leverage the learnt environment model to train an RL policy model and optimize long-term reward to improve performance of microservices, subject to resource constraints. We use vanilla RL algorithm, actor-critic, equipped with a novel technique parameter space noise exploration. To prevent the environment model from overfitting, we employ an iterative procedure that alternates between model learning and policy optimization and uses learnt policy to collect additional data from its interaction with the environment to further improve the accuracy of environment model.

We demonstrate through our extensive evaluation on the microservice workflow system that we are able to learn an accurate model of the environment and effectively use the learnt model to search for good policy. More importantly, we show that MIRAS outperforms state-of-the-art model-free RL technique and other resource allocation approaches in the actual task of adapting performance of microservice workflow system under dynamic workloads. It is worth mentioning that although we implement the learning based resource adaption approach for microservice workflow system, this approach could also be easily adapted to other microservice systems or microservice-like systems.

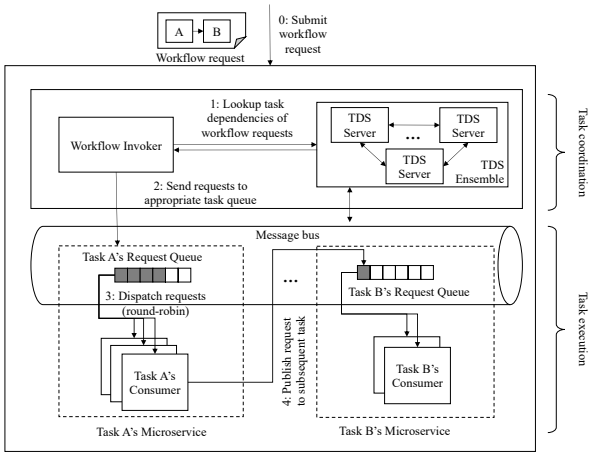


Fig. 1: Microservice workflow infrastructure.

In summary, our main contributions of this paper are as follow:

- We are *the first* to use model-based RL for resource adaptation in distributed and cloud-based system, and to apply this powerful learning technique in real-world environments where low sample complexity is crucial.
- We fully implement and integrate model-based RL resource adaptation mechanism into our microservice workflow system. To achieve this goal, we design or utilize several novel techniques including model refinement and parameter space noise exploration.
- Through extensive evaluation on real-world emulated scientific workflow workloads, we demonstrate that our proposed resource allocation approach achieves good control policy which outperforms existing algorithms.

II. Background: Microservice Workflow Infrastructure

In this section, we introduce the microservice workflow infrastructure, which is our targeted microservice environment. We briefly describe the architecture, resource model, and various challenges of resource adaptation in this infrastructure.

A. Microservice-Based Workflow Management

Traditional workflow system often employ a monolithic approach in workflow implementation and execution. In particular, each workflow is implemented as a tightly coupled set of tasks and has its own workflow execution plan that specifies how to run the workflow on a distributed computation infrastructure. On the contrary, the microservice-based cloud workflow management approach [27][26] models tasks as microservices and separate workflow's task dependencies from implementations of individual tasks. This design enables more flexible and scalable workflow composition, fine-grained resource scheduling (*i.e.,* resource scaling can be done at the task level, instead of at the workflow level), and load balancing

among computing machines. An architectural overview of this infrastructure is presented in Figure 1.

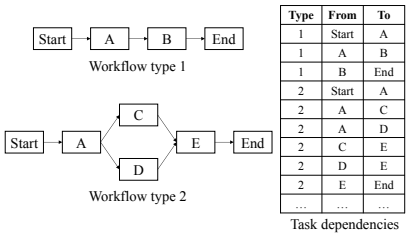
In this architecture, each task type is modeled as a microservice that consists of a request queue and a set of consumers subscribing to the queue to handle requests. Task dependencies are maintained by a separate *task dependency service* (or TDS). Several TDS servers form an ensemble to increase availability. Figure 2 shows an example of a task dependency table of workflows type 1 and 2 maintained by TDS. Whenever a workflow request arrives, the *workflow invoker* asks the TDS which task of the workflow should be processed first (step 1 of Figure 1). Upon receiving response from TDS, given a request of workflow type 1, for example, the task invoker sends the request (step 2) to task A's request queue (*i.e.*, the first task of workflow type 1) so that it can be processed by one of A's consumers (step 3). Besides being a *subscriber* to its task request queue, each task consumer also acts as a *publisher* for other types of tasks following the workflow's task dependency graph. After a task consumer finishes processing a request, it will query TDS about the subsequent task(s) of the workflow to "publish" the request to those tasks, which is step 4.

It is worth mentioning that although this paradigm shares some characteristics with stream processing (*e.g.*, both could be modeled as DAG), they have some key differences. Stream processing focuses on online distributed processing of large chunks of data, while the microservice workflow system deals with on-demand, (potentially) long tail and computationally heavier data processing jobs. These discrepancies lead to different resource adaptation considerations. Stream processing emphasizes resource placement while microservice workflow system emphasizes allocation of constraint resources.

B. Resource Models

In the following, we describe the microservice workflow system's resource model and notations used in the remaining of the paper.

Let us assume that the supported N workflow types compose of J types of tasks (i.e., each workflow type corresponds to a DAG of a subset of J types of tasks). As discussed earlier, we model each task type j ($1 \leq j \leq J$) as a microservice consisting of a *queue* that stores the task's requests, and a set of *task consumers* that subscribe to the queue to perform actual processing of the task's requests. To abtract away low-level resource information, we assume that the consumers of a task have identical computational capacity in terms of CPU and memory. Hence, the workflow system only needs to control the number of consumers for each microservce. We divide time into discrete time windows with identical time interval, and set time windows to be time units of collecting data and making resource management decisions. Namely, data collection and resource management are done at the beginning of each time window. Note that other options also exist, such as setting the beginning of each time window to be the time when a new workflow request arrives. But such a setting tends to capture the transient behavior of workflows instead of a more global view and suffers from high randomness originated from random workflow request arrival. We use (T_k, T_{k+1}) to represent the k-th time window. We denote the configuration of the numbers of consumers over microservices during the time window (T_k, T_{k+1}) as $\mathbf{m}(k) = (m_1(k), m_2(k), ..., m_J(k))$, where $m_j(k)$ is the number of consumers of task type j during the k-th time window. Since $\mathbf{m}(k)$ determines processing time, it represents *resource allocation decision* of the microservice workflow system.



Type	From	To
1	Start	A
1	A	B
1	B	End
2	Start	A
2	A	C
2	A	D
2	C	E
2	D	E
2	E	End
...	...	...

Task dependencies

Fig. 2: Example of workflow types and the corresponding task dependencies.

In terms of performance metrics, at workflow level, we are interested in the *average processing time* (or average delay) of each workflow type i ($1 \leq i \leq N$), as well as the average processing time over all types of workflows. The *processing time* of a workflow request is defined as the duration between its arrival time and the time when the workflow's last task is finished. The average delay of workflow type i over the time window (T_k, T_{k+1}), denoted as $d_i(k)$, is calculated by averaging delays of all requests of type i that arrive during (T_k, T_{k+1}). We denote $\mathbf{d}(k)$ as the vector form of the set of average delays of all workflow types in the k-th time window: $\mathbf{d}(k) = (d_1(k), d_2(k), ..., d_N(k))$. The average delay of requests over all types of workflows at k-th time window, $\mathbf{d}(k)$, is the most straightforward metric of system performance. However, workflow processing time could not be immediately observed in our paradigm, because the processing of one workflow could stretch over multiple time windows. Therefore, we propose a task-level performance metric.

At task level, the processing delay of a task request when it is processed by a microservice is measured from the time the task arrives at microservice's request queue until the task departs the microservice system after being processed by one of the consumers. According to the Little's law [10], we represent the processing delay via the number of task requests in the microservice (including tasks waiting in the queue and tasks being processed by consumers), or the number of *work-in-progress* (WIP for short), denoted by $w_j(k)$ for task j ($1 \leq j \leq J$) at time window (T_k, T_{k+1}). The more work-in-progress a microservice has, the longer delay is to be expected. We use $\mathbf{w}(k)$ as the vector representation of WIP over all microservices at k-th time window.

C. Challenges of Resource Adaptation

To meet certain performance goals, such as minimizing workflow processing time under constraint resources, it is necessary to design a resource adaptation algorithm which elastically scale the numbers of consumers assigned to each microservice type. However, due to the intrinsic properties of

the microservice-based workflow system, the design of such an adaptation algorithm faces the following challenges:

1) Workflow requests arrive online, with variant number of requests in different time windows. This is especially true for long tail scientific data processing (*e.g.*, material science microscopic image processing), where the size of each job is not very large but requests come in at random time points. Moreover, although one workflow has fixed microservice set and microservice composition structure, the processing time of each microservice is not fixed, due to variant sizes of input data. Some simple policies, *e.g.*, Earliest Deadline First, might mitigate this unpredictability, but such solutions are not able to adapt quickly to vast condition changes, and thus perform poorly on variant environment conditions which is often the real-world scenario.
2) Resource adaptation decisions have cascading effects on microservices. Workflows might share microservices, thus a resource adaptation decision on a single microservice type impacts the processing of several workflows at near future time windows. What's worse, within a workflow the tasks have dependency relationship among them. The completion of one task could immediately produce some other tasks according to their workflow topology, or the completed task could wait for synchronization signal. Therefore, it is very difficult to predict the effect of a resource allocation decision on microservice work-in-progress of the next and future time windows.
3) Cloud services are still costly when scaling up, so the resources users have access to are often constraint. In our scenario, the "constraint" has two embodiments. First, resource adaptation algorithm needs to consider operation cost and does not use more consumers than what we allocate to them. Second, if a resource adaptation approach requires training data obtained from giving the microservice workflow system control inputs and receiving feedback in order to profile the microservice system, it has access to limited number of such interactions (*i.e.,* sample complexity should not be high).

III. Resource Adaptation Framework for Microservice Infrastructure

To tackle the aforementioned challenges, we present MIRAS, a model-based reinforcement learning framework for the microservice workflow infrastructure, whose purpose is to dynamically adapt system resources to meet certain performance guarantees under changing workloads and limited resource capacity.

Our resource adaptation framework (Figure 3) consists of three main components. The first component, the *microservice workflow system* component which we discussed in Section II, is the real-world microservice workflow infrastructure that we work on. To tackle dynamic microservice system status and microservice cascading effect challenge, we propose to use a *reinforcement learning model* to dynamically make decisions

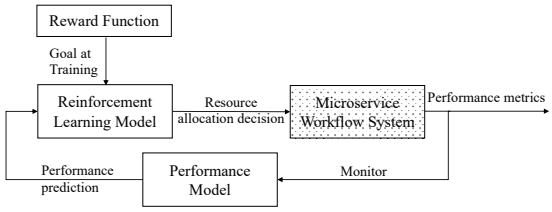


Fig. 3: Model-based reinforcement learning framework for microservice infrastructure.

on microservice resource allocation in order to improve certain performance metrics. To tackle high sample complexity introduced by reinforcement learning, we use a *performance model* that acts as a predictive model of the microservice workflow system to train the reinforcement learning model.

Reinforcement learning [32], with its closed-loop feedback from the environment, is an ideal candidate for implementing resource adaptation framework. In particular, the role of agent in RL is to learn from experience of interacting with the real environment to search for improved resource allocation policy. In fact, RL has been applied to resource adaptation of various types of applications, including resource management in data centers [38], real-time scheduling [15], or video streaming [23], just name a few (c.f. Section VII). These works are all based on *model-free* version of RL, in which the agent relies only on the feedback from actual environment as it learns to produce better policies. However, model-free technique suffers from high sample complexity (a tremendous number of interactions is needed to learn an RL policy), and thus, hinders its use in real-world application domains where it is extremely time-consuming (and thus cost-ineffective) to obtain samples. As a result, the applications of model-free RL have been often limited to where an elaborate simulated model of environment is available to facilitate the learning process.

In this work, we propose to use *model-based* RL to implement the resource adaptation framework for microservice workflow system. With model-based approach, instead of using only actual interactions with the real environment like in model-free approach, we first train a performance model of the microservice environment and use this model to generate synthetic interactions with the environment to assist policy learning. As a result, this approach has significantly smaller sample complexity compared to model-free approach. This benefit makes model-based approach suitable for applications such as microservice workflow system, where obtaining actual interactions with the real environment is time-consuming (in the microservice workflow system, one interaction takes tens of seconds, or even minutes) or where an elaborate simulated model of environment is difficult to achieve.

IV. Model-based Reinforcement Learning Approach

In the following, we first present some preliminaries on reinforcement learning and how state, action, and reward are defined in the context of microservice infrastructure. Then, we show how we train and refine a model for the environment

(*i.e.*, microservice execution environment), how we leverage the model to train a policy, and finally, how we integrate model learning and policy learning in an iterative framework.

A. Preliminary

In general setting of reinforcement learning [32], an *agent* interacts with an *environment*. At k-th time step (*i.e.*, at the beginning of time interval (T_k, T_{k+1})), the agent observes some *state* $\mathbf{s}(k)$ and takes an *action* $\mathbf{a}(k)$ sampled from a *policy* $\pi(\mathbf{a}(k)|\mathbf{s}(k))$. In general, a policy $\pi(\mathbf{a}(k)|\mathbf{s}(k))$ specifies the probability of taking action $\mathbf{a}(k)$, given current state $\mathbf{s}(k)$. After applying the action, the environment transitions to a new state $\mathbf{s}(k+1)$ with probability $P[\mathbf{s}(k+1)|\mathbf{s}(k), \mathbf{a}(k)]$, and provides a *reward* $r(k)$. The goal of reinforcement learning is to learn a policy to maximize the expected cumulative discounted reward $\mathbb{E}_\pi[R(k)]$, where $R(k) = \sum_{t=k}^{\infty} \gamma^{t-k} r(t)$, $\gamma \in [0, 1]$ is the discount factor.

In model-free RL, the state transition is the actual interaction with real environment. In model-based RL, we train a model of the environment, denoted as $\hat{f}_\Phi$ (with Φ being the parameters of the model), and use the model to predict state transition $\hat{\mathbf{s}}(k+1) = \hat{f}_\Phi(\mathbf{s}(k), \mathbf{a}(k))$. Reward is predicted in a similar way.

Policy learning includes two main approaches: Q-learning and policy optimization. With Q-learning, the agent learns an estimate of the optimal action-value function (or Q-function) and obtains estimated optimal action by maximizing Q-function. On the other hand, policy optimization methods try to learn the optimal policy by directly optimizing on the policy space. Specifically, if we denote Θ as the parameters of the policy model that we are trying to learn, given observations $(\mathbf{s}(k), \mathbf{a}(k), r(k), \mathbf{s}(k+1))$, policy optimization tries to update Θ so that the policy model will generate better actions. It is generally believed that policy optimization method is capable to a wider range of problems (*e.g.*, control problems with continuous states) and tends to converge faster than Q-learning method. Therefore, we use *actor-critic* method, a type of policy optimization method, for policy learning.

B. Definition: State, Action, and Reward

Before presenting model and policy learning, in this section, we show how to map various RL concepts, including state, action, and reward, to microservice workflow system's resource model and performance metrics introduced in Section II.

A natural choice of microservice workflow system *state* is the average delays of different workflow types $\mathbf{d}(k)$, since such state definition can be used conveniently to measure the reward of actions made by the agent. However, average delays at k-th time step $\mathbf{d}(k)$ are only partially observable when new action is made by the agent at time window (T_k, T_{k+1}). This is because workflow requests that arrive in (T_k, T_{k+1}) might not be finished during that time window.

For MIRAS, instead of using $\mathbf{d}(k)$, we use work-in-progress $\mathbf{w}(k)$ as state of microservice workflow system: $\mathbf{s}(k) = \mathbf{w}(k)$ [2]. Since WIP of a microservice is (in the long term) proportional to its average delay, WIP across all microservices can be used to capture average delays of workflow types that the system supports. In addition, different from $\mathbf{d}(k)$ that is partially observable, $\mathbf{w}(k)$ is completely observable at the end of every k-th time window.

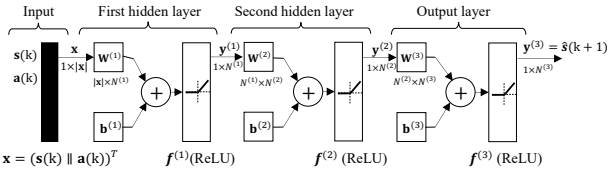


Fig. 4: Neural network model of microservice environment.

The fewer WIP of microservices, the shorter processing delays microservices will have and the shorter processing delays the system can support for different workflows. We define *reward* to be negative of the aggregated WIP across microservices observed at the end of each time window:

$$r(k) = 1 - \sum_{j=1}^{J} w_j(k). \tag{1}$$

In this way, the cumulative discounted reward $R(k)$ reflects the total number of finished microservices starting from the current time window.

As described in Section II, the number of consumers allocated to microservices $\mathbf{m}(k)$ can be used to represent resource allocation decision of the microservice workflow system. Therefore, $\mathbf{m}(k)$ represents the *action* that the RL agent makes at k-th time window: $\mathbf{a}(k) = \mathbf{m}(k)$ [3].

C. Performance Model Generation

For model learning, we assume little prior knowledge about the microservice workflow system. We use a neural network to model the microservice system, because neural networks have strong function approximation capability. We further refine the model by adjusting its behavior at boundary point. These steps are prerequisites for MIRAS policy learning.

*1) **Model Learning**:* Our neural network-based environment model takes input $\mathbf{x}$ as the combination of system states $\mathbf{s}(k)$ and action $\mathbf{a}(k)$ at the beginning of the current time window (T_k, T_{k+1}): $\mathbf{x} = (\mathbf{s}(k) \parallel \mathbf{a}(k))^T$, and predicts output as the system states $\mathbf{s}(k+1)$ of the next time window (T_{k+1}, T_{k+2}). The neural network model consists of L layers, with the number of neurons in each layer denoted as $N^{(l)}$ $(1 \le l \le L)$. Correspondingly, $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ $(1 \le l \le L)$ represent the weight matrix and bias of layer l. Each layer l also includes a non-linear activation function $\boldsymbol{f}^{(l)}$ (we use ReLU) to introduce the non-linearity into the network model. We use $\mathbf{y}^{(l)}$ to denote the vector of outputs from layer l ($\mathbf{y}^{(0)} = \mathbf{x}$). The output of the model is the predicted state of the environment in the next time window: $\hat{\mathbf{s}}(k+1) = \mathbf{y}^{(L)}$. Figure 4 shows an actual neural network architecture of our environment model.

If we denote Φ as the set of parameters of the environment models: $\Phi = \{\{\mathbf{W}^{(l)}\}, \{\mathbf{b}^{(l)}\}\}$, the environment model can

[2] In this paper we use $\mathbf{s}(k)$ and $\mathbf{w}(k)$ interchangably.

[3] In this paper we use $\mathbf{a}(k)$ and $\mathbf{m}(k)$ interchangably.

be represented by a function $\hat{f}_\Phi$: $\hat{\mathbf{s}}(k+1) = \hat{f}_\Phi(\mathbf{s}(k), \mathbf{a}(k))$. And the objective of the environment model learning is to find a parameter set Φ that minimizes the square error of one-step prediction:

$$min_\Phi \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{s}(k),\mathbf{a}(k),\mathbf{s}(k+1))\in\mathcal{D}} ||\mathbf{s}(k+1) - \hat{f}_\Phi(\mathbf{s}(k), \mathbf{a}(k))||^2, \tag{2}$$

where $\mathcal{D}$ is the collected training data set, the element of which is tuple $(\mathbf{s}(k), \mathbf{a}(k), \mathbf{s}(k+1))$. The model is trained using gradient descent optimizer with backpropagation.

*2) **Model Refinement**:* The aforementioned neural network model is able to well capture the performance characteristics of microservice workflow systems, for the most part. When at least one dimension of WIP $\mathbf{w}(k)$ is small, or close to 0 ($\mathbf{w}(k)$ should be non-negative), neural network model gives "inappropriate" outputs which hinders MIRAS from learning a good policy at small state. Specifically, when WIP approaches 0, the microservice system itself has high randomness caused by online workflow arrival and varying microservice processing time, and no clear connection between $\mathbf{w}(k)$ and $\mathbf{m}(k)$ could be observed from the environment or from the model, for the model is trained based on interactions with the environment. On the contrary, when WIP is sufficiently large, the learnt neural network model suffers less from randomness. We call 0 the boundary of WIP. This boundary effect harms policy learning because the RL agent would not know that 0 should be the termination point of learning; on the contrary, the agent would try to assign more consumers to a task type to bring its WIP down. Therefore, it is ideal for the microservice workflow system to maintain similar performance characteristics at the boundary as at a higher state instead of outputting almost random WIP predictions.

To alleviate the issue at boundary, we take inspiration from the fact that different microservice types are loosely coupled. For the j-th dimension of WIP, at next time window $w_j(k+1)$ is majorly determined by $w_j(k)$ and $m_j(k)$, and the impact of other dimensions $w_{n,n\neq j}(k)$ and $m_{n,n\neq j}(k)$ is smaller. Therefore, we "lend" some tasks to a certain task type to get it far from boundary, calculate the number of microservices of the same task type at next time window, and take back the lent tasks from the predicted WIP at next time window to stay consistent. This *Lend - Giveback* procedure encapsulates the aforementioned environment model. The process is shown in Algorithm 1.

In this algorithm, initially we obtain threshold parameters through simple statistical analysis on the collected dataset $\mathcal{D}$, which consists of tuples $(\mathbf{s}(k), \mathbf{a}(k), \mathbf{s}(k+1))$. After initialization, we perform a *Lend - Giveback* mechanism to deal with the boundary effect. In *Lend* phase, we "lend" some tasks to task type j and feed the adjusted WIP to the predictive model. In *Giveback* phase, we take back the "lent" tasks. For each dimension we perform an independent operation of *Lend - Giveback*, minimizing the impact of adjusted dimension on other dimensions. The idea of the model refinement algorithm comes from our prior knowledge of the system, but we deem that the prior knowledge is limited and intuitive.

Algorithm 1 Model Refinement

//Initialization:
for each dimension j of $\mathbf{w}(k)$ in dataset $\mathcal{D}$ **do**
  $\tau_j \leftarrow p$-percentile of $w_j(k)$ in $\mathcal{D}$
  $\omega_j \leftarrow (100-p)$-percentile of $w_j(k)$ in $\mathcal{D}$
for each dimension j of $\mathbf{s}(k)$ **do**
  $\mathbf{t}(k) \leftarrow \mathbf{s}(k)$
  if $s_j(k) < \tau_j$ **then**
    //Lend:
    generate ρ_j, uniformly sampled from (τ_j, ω_j)
    $t_j(k) \leftarrow t_j(k) + \rho_j$
    $\mathbf{t}(k+1) \leftarrow \hat{f}_\Phi(\mathbf{t}(k), \mathbf{a}(k))$
    //Giveback:
    $t_j(k+1) \leftarrow t_j(k+1) - \rho_j$
    $t_j(k+1) \leftarrow max(t_j(k+1), 0)$
$\hat{\mathbf{s}}(k+1) \leftarrow \mathbf{t}(k+1)$

D. Policy Learning

After we obtain the predictive environment model of the microservice workflow system, we train a deep reinforcement learning agent by letting it interact with the learnt environment model $\hat{f}_\Phi$ instead of the actual real environment, and observe rewards and state transitions. To boost RL performance, we integrate a novel technique, parameter space exploration noise, to a vanilla RL algorithm.

We use actor-critic method, a policy gradient method, for policy learning. With this method, we train simultaneously two neural network-based models: an *actor network* and a *critic network*. The actor network π_Θ, where Θ is the set of neural network's parameters, essentially represents a policy network, and it produces the probability of selecting an action $\mathbf{a}(k)$ given the current state of the environment $\mathbf{s}(k)$. On the other hand, the critic network is used to evaluate the value of action made by the actor network. In other words, the critic network essentially represents function $Q_{\pi_\Theta}(\mathbf{s}(k), \mathbf{a}(k))$ and its output is simply the estimated Q-value of the current state and of the action given by the actor network. To train the critic network, we can use standard methods as in Q-learning.

In general, the key idea of policy gradient methods is to estimate the gradient of $\mathbb{E}_\pi[R]$, the expected total discounted reward, by observing the trajectories of executions, or state transitions, obtained by following the policy. The gradient of the cumulative discounted reward with respect to the policy parameters Θ can be computed as $\nabla_\Theta \mathbb{E}_{\pi_\Theta}[R] = \mathbb{E}_{\pi_\Theta}[\nabla_\Theta \log \pi_\Theta(\mathbf{s}, \mathbf{a}) Q_{\pi_\Theta}(\mathbf{s}, \mathbf{a})]$ [30]. If we model action as deterministic decision $\mathbf{a}(k+1) = \mu_\Theta(\mathbf{s}(k))$, Silver *et al.* [30] prove that the above gradient can be written as $\nabla_\Theta \mathbb{E}_{\pi_\Theta}[R] = \mathbb{E}[\nabla_\Theta \mu_\Theta(\mathbf{s}) \nabla_\mathbf{a} Q_{\mu_\Theta}(\mathbf{s}, \mathbf{a})|_{\mathbf{a}=\mu_\Theta(\mathbf{s})}]$. As a result, the updating rule of deterministic policy's parameter Θ is $\Theta^{t+1} = \Theta^t + \alpha \mathbb{E}[\nabla_\Theta \mu_\Theta(\mathbf{s}) \nabla_\mathbf{a} Q_{\mu_\Theta^t}(\mathbf{s}, \mathbf{a})|_{\mathbf{a}=\mu_\Theta(\mathbf{s})}]$, where α is learning rate.

While we can leverage vanilla neural network structure for the critic network, the design of the actor network requires more considerations to ensure that the output of the actor

network satisfies the resource constraint (*i.e.*, the total number of consumers across all microservices is bounded). To enforce such constraint, we design output of actor network as a categorical distribution over J different possible categories, by applying a softmax activation function at the output layer. The categorical distribution can then be translated into numbers of consumers by multiplying with the total number of consumers $\mathcal{C}$: $m_j(k) = \lfloor \mathcal{C} * a_j(k) \rfloor, \forall 1 \leq j \leq J$.

In our scenario, both state $\mathbf{s}$ and action $\mathbf{a}$ are integers. However, we choose continuous action space because of the extremely large search space ($O(\mathcal{C}^J)$). In this paper, we use deep deterministic policy gradient method (DDPG) [19] to train the actor and critic networks. DDPG is an actor-critic algorithm that operates over continuous action spaces and explores the action space by adding an exploration noise to the output action sampled from current policy. The exploration lets the agent try new behaviors to discover the ones with higher cumulative rewards.

Directly imposing exploration noise to the output action actually performs poorly in our system. The reason is that actions added by exploration noise often violate our constraints on total number of consumers, leading to invalid exploration. Our approach to tackle with this problem is to use parameter space noise in exploration [29] instead of action space noise. Parameter space noise approach adds adaptive noise to the parameters of the policy neural network model rather than to the output action. Specifically, this method perturbs policy network parameters with additive Gaussian noise to generate explorations. The scale of this parameter noise depends on the variance in action space it induces. The introduction of parameter noise solves the action constraint issue of action space noise based DDPG, makes RL converge fast, and exhibits a greater variety of behaviors in the learnt policy.

E. Iterative Model-based Reinforcement Learning

Model-based reinforcement learning approach has a limitation: the learnt policy network often explores regions where scarce training data is available for the environment model. To overcome this limitation, MIRAS switches iteratively between i) using the current learnt policy to interact with real environment and collect more training data of the actual interactions between agent and the real environment, ii) training environment model incrementally with newly collected training data, and iii) using the updated environment model to improve the current policy. As a result, we can avoid being stuck in local optima of environment model and gradually build more accurate model of environment with the help of newly collected training data, as well as improve the policy.

The iterative training procedure is presented in Algorithm 2. The outer loop represents training environment model using interactions with real environment, and the inner loop represents policy training using environment model.

Algorithm 2 Iterative Model-based Reinforcement Learning Procedure

```
Initialize μ_Θ, f̂_Φ, and D
repeat
    Collect interactions with real environment using μ_Θ & add to D
    Train environment model f̂_Φ using D
    repeat
        Collect synthetic samples from refined f̂_Φ
        Update policy μ_Θ using parameter noise DDPG
    until Performance of the policy stops improving
until The policy performs well in real environment
```

V. Implementation

We have implemented the aforementioned microservice workflow system on Google Cloud Platform [4]. The system is built upon a cluster of three Google Cloud virtual machines (nodes), each of which has 1 vCPU (virtual CPU), $16GB$ RAM, and $100GB$ secondary storage. We use 3 Zookeeper [2] nodes to act as TDS servers to maintain microservice dependency relationship. RabbitMQ [8] serves as the message queue engine. We use Docker containers [3] as microservice consumers which subscribe to RabbitMQ's message queues. Whenever a consumer of a specific task type is idle, it consumes the next request of that type from RabbitMQ's message queue. We employ an acknowledgement mechanism between RabbitMQ message queues and consumers to guarantee that task requests (and the workflows they belong to) do not get lost in the system. We use Kubernetes [6] as the container orchestration engine. Kubernetes abstracts consumers as Replication Controllers so that the number of consumers for each microservice could be conveniently scaled. Kubernetes also manages load balancing of containers among the three machines in our cluster.

As for predictive model training, we use Python and its Tensorflow [9] library. We employ OpenAI's Baselines project [7] as implementation of the DDPG algorithm.

VI. Evaluation

We perform extensive evaluations to validate the performance of MIRAS. Specifically, we (1) evaluate how accurate the learnt predictive model is to depict the microservice workflow system, (2) validate the convergence of model-based reinforcement learning, and (3) compare our proposed approach with existing workflow management algorithms.

A. Experimental Setup

1) Workflow workloads: To evaluate the performance of MIRAS under different workloads and compare it to existing workflow processing algorithms, we use two real-world scientific workflow computing ensembles as requests. The first workflow ensemble is Material Science Data processing workflows (MSD) [26][27], which consists of 3 workflows — Type1 to Type3 — and 4 task types. The second workflow ensemble is Laser Interferometer Gravitational Wave Observatory (LIGO) [17], which consists of 4 workflows — DataFind, CAT, Full, and Injection — and 9 task types. We use Poisson process to emulate request traces for both workflow datasets.

2) Time window length setting: As we mention in Section II, we divide time into discrete time windows and set one time window to be one interaction of resource allocation. We set 30 seconds to be the length of one time window. This window length is a trade-off. Firstly, we find that it usually takes 5 to 10 seconds for Kubernetes to generate a new container or destroy an existing container (container generation and destruction can be parallelized). Time window length should be large enough so that the container "start-up" time has small influence. Secondly, time window length should not be very high in order to react responsively to environment feedback. We have tested $5s$, $15s$, and $30s$, and $30s$ is the best option.

3) MIRAS parameters for models: For MSD dataset, we use a 3-layer neural network as the predictive model, each layer has 20 neurons. Its Actor network has 3 layers, each of which has 256 neurons. We use the same parameters for the Critic network, except that we insert one of Critic's inputs — action — to the second layer. For LIGO, we use a one-layer 20-neuron neural network as the predictive model. [4] The RL networks of the LIGO datasets are similar to those of MSD, except that both networks of LIGO have 512 neurons at each layer. When we collect training data of interactions with the MSD environment, we let the agent interact with the environment for around $1,000$ steps and then we train the predictive model based on collected samples. After every 25 steps during that process we reset the environment. "Reset" means to provision sufficient consumers of each microservice to reduce WIP close to 0. When training the RL agent with the predictive model, the number of steps within one rollout (one episode before resetting the predictive model) is also 25. For LIGO we set the number of steps before training the predictive model to be $2,000$, and rollout step to be 10.

4) Consumer constraints: As we deal with resource allocation problem under constraints, it's important to find the correct constraints for the microservice systems. A good constraint means that we don't have redundant resources so that good resource allocation policies are unnecessary, and also resources should be sufficient so that feasible resource allocation solutions can be found. In our experiments we use 14 and 30 to be consumer constraints for MSD dataset and LIGO dataset respectively. In all following experiments we make sure that the constraints are enforced.

B. Model Evaluation

In this subsection we evaluate the accuracy of our predictive model on MSD and LIGO. As mentioned in Section IV-E, we iteratively learn the predictive model and the RL policy. The final policy to be used in microservice workflow system control is obtained from interacting with the predictive model learnt with all collected data. Thus we evaluate the accuracy of predictive models learnt with all collected data. For MSD we collect $14,000$ data entries and for LIGO we collect $37,000$ data entries. After we obtain predictive models as discussed in Section IV-C, we use another 100 data points as testing data. We collect real data trace from the microservice workflow system and we compare it with predictive data trace. Actions are randomly selected and vary every 4 steps.

We evaluate the learnt model in two ways. First, like in traditional supervised learning, we fix the input of the model — current state and action — and compare the model output — reward and next state — with ground truth. Second, as the model we train is expected to have the look-forward capability, we examine if the model gives consistently reasonable results when running standalone, as what we do in policy learning. To do this, we give the model an initial state, which is the first state in testing data points; we predict subsequent states and rewards using the predicted state of the last time window.

Model accuracy evaluation result is shown in Figure 5. Here we compare the prediction accuracy of immediate reward (average of next state WIP) and the first dimension of next WIP (other dimensions show similar shape in figure). We can observe that the prediction on fixed input is considerably accurate. The iterative prediction trace shows a slightly higher divergence with the ground truth due to cumulative prediction error. Also we can observe that LIGO has a higher divergence than MSD, due to its higher dimension of microservice types (9 vs 4) and its higher complexity of topology. But predictions on both datasets have the same trend with ground truth data, and the models are sufficient for policy learning.

C. Policy Evaluation

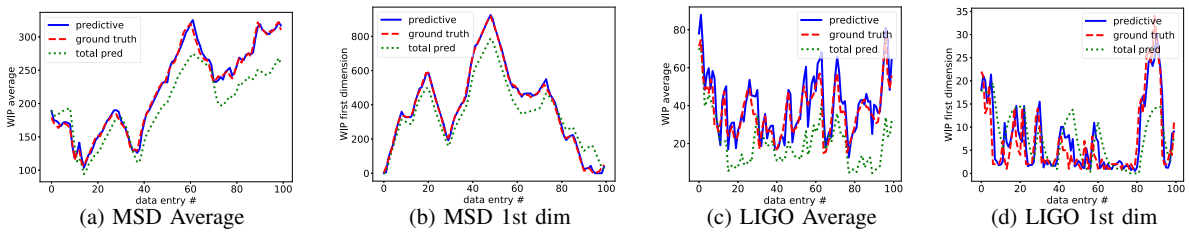
In this part we evaluate MIRAS policy training. Namely, we test whether MIRAS converges to yield good policies. As mentioned earlier, we continuously switch between running the RL agent on the real environment for some steps ($1,000$ for MSD and $2,000$ for LIGO), training the predictive model, and training RL agent based on the predictive model. At the end of each iteration, we evaluate the learnt policy by letting it interact with the environment for some steps and observe aggregated rewards. As in sample collection phase, we evaluate MIRAS agent for MSD dataset by letting it run 25 steps on the real microservice system environment. For LIGO agent, we let it run 100 steps instead of 10 steps as in training because LIGO is a more complicated workflow, and we want to eliminate the effect of obtaining small WIP from throttling upstream microservices.

Resulting training traces of MSD and LIGO are shown in Figure 6a and Figure 6b. We can observe that at the beginning iterations policies are not very good. After about 11 iterations ($11,000$ steps for MSD and $22,000$ steps for LIGO) both policies begin to converge.

D. Comparing MIRAS with existing algorithms

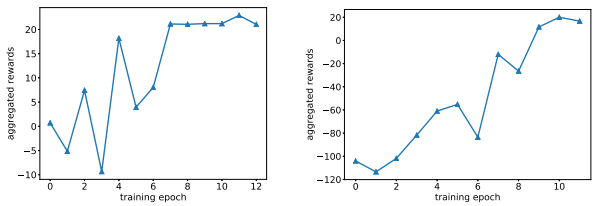
In this part we use trained MIRAS policies discussed above and compare them with existing workflow scheduling/resource allocation algorithms. As our goal of resource allocation is

[4] For the LIGO workflow which is more complex than MSD, we use a predictive model with fewer layers, which is counterintuitive. In practice we find that predictive model training suffers from overfitting if insufficient training data is provided, which is the case for LIGO, where the input space of the predictive model is extremely large. Therefore, we use a smaller neural network to tackle the overfitting problem.



(a) MSD Average (b) MSD 1st dim (c) LIGO Average (d) LIGO 1st dim

Fig. 5: Predictive model accuracy. Red dashed line represents ground truth of the testing data. Blue line represents model predictive data on fixed input. Green dotted line represents iterative prediction data when we fix actions and the initial state.



(a) MSD policy training trace (b) LIGO policy training trace

Fig. 6: Training traces for MSD and LIGO. The vertical axises represent aggregated rewards over 25 steps for MSD or 100 steps for LIGO and the horizontal axises represent training iterations.

to reduce response time of workflow requests, we compare response time in this part instead of WIP (comparison of WIP would yield similar results).

We compare MIRAS with 4 other algorithms. The $1st$ algorithm is *DRS* [14], a resource allocation algorithm based on Jackson open queuing networks proposed for stream processing. The $2nd$ algorithm is *heft* [37], a scheduling algorithm for scientific workflow processing. This algorithm gives each task a priority and assigns machines to tasks according to their priorities. This is different from our resource allocation problem, where we focus on a higher level of resource assignment and determine only the number of machines allocated to each task at one time window. We adapt *heft* into our system. Specifically, we assign tasks with priorities using their proposed method. At the beginning of each time window we make resource allocation decisions based on both task number and task priority. The $3rd$ algorithm is *MONAD*, a model predictive control based resource allocation algorithm for microservice systems. The $4th$ algorithm is DDPG with no predictive model, or model-free DDPG. That is, we directly train DDPG models by interacting with the real environment. To guarantee fairness, we train DDPG models using the same number of interactions with MIRAS.

We evaluate the performances of MIRAS and 4 other algorithms under variant environments by generating bursts of workflow requests and testing how these algorithms adapt to vast environment changes. For each of the two workflow ensembles we generate three categories of bursts. For MSD dataset, the bursts are 300 requests, 200 requests, 300 requests for Type1, Type2, and Type3; 1000, 300, 400 for Type1 to Type3; and 500, 500, 500. For LIGO dataset the bursts are 100, 100, 50, 30 for DataFind, CAT, Full, Injection; 150, 150, 80, 50; and 80, 80, 80, 80 for the 4 workflows. These request bursts are fed into the system at the beginning of each evaluation. We also feed the system with continuous workflow requests sampled from Poisson process.

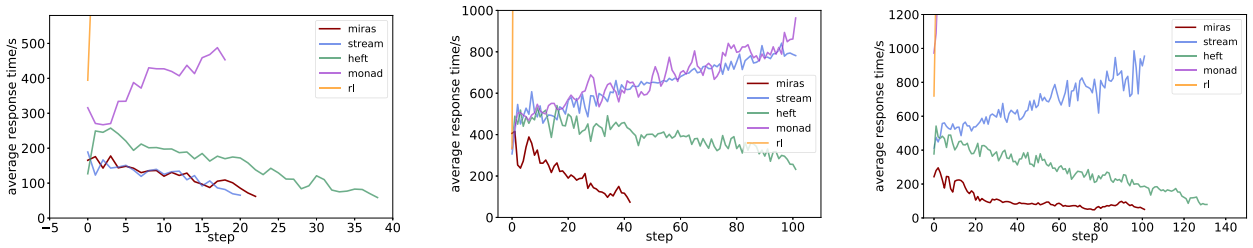
The results are shown in Figure 7 and Figure 8. As we can see, MIRAS is better than or at least as good as the other algorithms under the testing conditions, especially in long-term returns. For the MSD dataset, which is less complicated, we observe that MIRAS is significantly better than other algorithms. For the LIGO dataset, MIRAS performs well under small burst. When larger bursts, burst 2 and burst 3, are fed into the system, we can see an increase followed by a decrease of response time for MIRAS. In fact, when we look into WIP variation at high burst scenarios, we observe that MIRAS is very smart and puts aside certain tasks, *e.g.,* Coire in workflows CAT, Full, and Injection, at the beginning and focuses on other tasks. This causes the response times of CAT, Full, and Injection workflows to increase. At a later time when some other task queue is low, MIRAS turns back to deal with Coire queues. Therefore, although at 20-th step in Figure 8b and Figure 8c the response times of MIRAS are higher than *stream* and *heft*, MIRAS is able to recover quickly to a low response time level afterwards. We think that this is the advantage of using reinforcement learning in resource allocation. Reinforcement learning does not only consider immediate return, more importantly, it considers all future returns. Therefore, it considers long-term returns and achieves a better global solution.

It's worth mentioning that we find algorithms *DRS*, *MONAD*, and *DDPG* fail to control the system under the constraints. *DRS* is designed for stream processing and it does not react responsively to condition changes. *MONAD* focuses on short-term returns and is not suitable to yield a global optimal solution. *DDPG* without predictive model could perform well when supplied with sufficient training data, but with limited interactions with the real environment it doesn't converge to a good policy, showing its poor sample efficiency.

VII. Related Work

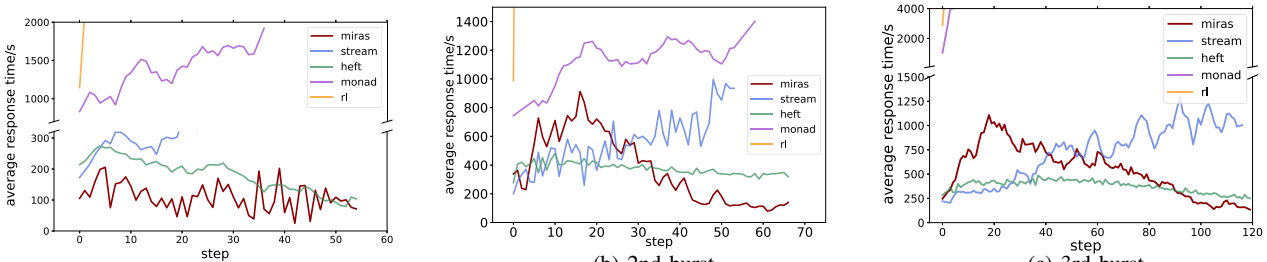
A. Resource Allocation for Cloud-based Systems

Cloud system based autonomous resource allocation or scheduling has been intensively studied in literature. In [16], Grandl *et al.* propose to adapt heuristics for multidimensional bin packing problem to cluster scheduling. In [37] the authors propose a scheduling algorithm for grid computing workflow processing, but they focus on resource placement instead



(a) 1st burst (b) 2nd burst (c) 3rd burst

Fig. 7: Performance comparison under different conditions for MSD. In this figure and the next figure, *stream* refers to DRS algorithm, *rl* refers to the original DDPG algorithm trained with the same number of interactions as MIRAS.



(a) 1st burs (b) 2nd burst (c) 3rd burst

Fig. 8: Performance comparison under different conditions for LIGO. We crop the y-axis in (a) and (c) for better visualization. Note that part of *stream* trace is discarded in (a).

of resource allocation. For cloud system or grid computing scheduling, several algorithms have been proposed based on heuristics [34][36][28]. Their works differ from ours in that they emphasize on data transmission improvement, and often fail to effectively incorporate future rewards. Suresh *et al*. [31] propose a cloud-based resource management framework for workflow processing, but they focus on large-scale Web applications and adopt a distributed approach, which makes it difficult to achieve fine-grained policies like MIRAS.

Meanwhile, there have also been works on cloud systems using reinforcement learning. Zhou *et al*. present a RL-based algorithm for adaptive resource management to optimize QoS and power consumption goals [38]. In [15], RL is applied in scheduling strategies which is learned online through interaction with the system. Wang *et al*. explore the application of deep RL to the problem of provisioning cloud resources [33], but they don't deal with the workflow problem as we do. Mao *et al*. present DeepRM, a system that translates the problem of packing tasks with multiple resource demands into a deep reinforcement learning problem [22]. However, the experiment was simulated on a synthetic dataset and hasn't been employed in real systems. In addition, it doesn't take workflow management into account. Liu *et al*. proprose an RL-based hierarchical control framework for cloud resource allocation and power management in [21], but they don't consider workflow management either. Besides, they adopt the model-free approach and thus their approach suffers from expensive interactions with the real system, limiting their evaluation to only simulation.

B. Deep Reinforcement Learning Based Systems

Apart from task scheduling or resource management, reinforcement learning, especially deep deinforcement learning (DRL) [24], has been adopted gradually in building computer or networking systems to fully take advantage of its outstanding performance in decision making. In [35], Xu *et al*. propose a DRL-based approach model-free control framework that tackles traffic engineering. In [23], the authors implements a system which uses DRL to select appropriate video bitrates to improve quality of service. Li *et al*. implement an Apache Storm based DRL system for scheduling in distributed stream data processing systems [18].

VIII. Conclusion

Resource allocation for microservice workflow systems is difficult due to dynamic workloads and complicated interactions between microservices in each workflow. In this paper, we propose to make resource allocation decisions based on deep reinforcement learning control policy. To tackle the high sample complexity problem of reinforcement learning, we propose MIRAS, a model-based approach. Specifically, we let a DDPG agent interact with the real microservice workflow system and collect interacting profiling data. We use the collected data to train a predictive model of the environment and use this model to further train the DDPG agent. We iterate among these three phases until the learnt agent is able to make good resource allocation decisions in the microservice workflow system. Our evaluations confirm the performances of the learnt policy.

Acknowledgement

This work is supported by the National Science Foundation under grant NSF 1827126.

REFERENCES

[1] Amazon Simple Workflow Service. https://aws.amazon.com/swf/. Accessed: 2018-05-10.

[2] Apache ZooKeeper. https://zookeeper.apache.org. Accessed: 2019-01-10.

[3] Docker. https://www.docker.com. Accessed: 2019-01-10.

[4] Google Cloud Platform. https://cloud.google.com. Accessed: 2019-01-10.

[5] Goolge Cloud Composer. https://cloud.google.com/composer/. Accessed: 2018-05-10.

[6] Kubernetes. https://kubernetes.io/. Accessed: 2019-01-10.

[7] OpenAI Baselines. https://github.com/openai/baselines/. Accessed: 2019-01-10.

[8] RabbitMQ. https://www.rabbitmq.com. Accessed: 2019-01-10.

[9] Tensorflow. https://www.tensorflow.org/. Accessed: 2019-01-10.

[10] ALLEN, A. O. *Probability, statistics, and queueing theory*. Academic Press, 2014.

[11] BALDINI, I., CASTRO, P., CHANG, K., CHENG, P., FINK, S., ISHAKIAN, V., MITCHELL, N., MUTHUSAMY, V., RABBAH, R., SLOMINSKI, A., ET AL. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[12] DEELMAN, E., GANNON, D., SHIELDS, M., AND TAYLOR, I. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems 25*, 5 (2009), 528–540.

[13] DEISENROTH, M. P., RASMUSSEN, C. E., AND FOX, D. Learning to control a low-cost manipulator using data-efficient reinforcement learning.

[14] FU, T. Z., DING, J., MA, R. T., WINSLETT, M., YANG, Y., AND ZHANG, Z. Drs: dynamic resource scheduling for real-time analytics over fast streams. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on* (2015), IEEE, pp. 411–420.

[15] GLAUBIUS, R., TIDWELL, T., GILL, C., AND SMART, W. D. Real-time scheduling via reinforcement learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence* (2010), ACM, pp. 201–209.

[16] GRANDL, R., ANANTHANARAYANAN, G., KANDULA, S., RAO, S., AND AKELLA, A. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review 44*, 4 (2015), 455–466.

[17] JUVE, G., CHERVENAK, A., DEELMAN, E., BHARATHI, S., MEHTA, G., AND VAHI, K. Characterizing and profiling scientific workflows. *Future Generation Computer Systems 29*, 3 (2013), 682–692.

[18] LI, T., XU, Z., TANG, J., AND WANG, Y. Model-free control for distributed stream data processing using deep reinforcement learning. *Proceedings of the VLDB Endowment 11*, 6 (2018), 705–718.

[19] LILLICRAP, T. P., HUNT, J. J., PRITZEL, A., HEESS, N., EREZ, T., TASSA, Y., SILVER, D., AND WIERSTRA, D. Continuous control with deep reinforcement learning. In *International conference on learning representations* (2016).

[20] LIU, J., PACITTI, E., VALDURIEZ, P., AND MATTOSO, M. A survey of data-intensive scientific workflow management. *Journal of Grid Computing 13*, 4 (2015), 457–493.

[21] LIU, N., LI, Z., XU, J., XU, Z., LIN, S., QIU, Q., TANG, J., AND WANG, Y. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017), IEEE, pp. 372–382.

[22] MAO, H., ALIZADEH, M., MENACHE, I., AND KANDULA, S. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (2016), ACM, pp. 50–56.

[23] MAO, H., NETRAVALI, R., AND ALIZADEH, M. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 197–210.

[24] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., ET AL. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529.

[25] NAGABANDI, A., KAHN, G., FEARING, R. S., AND LEVINE, S. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. *arXiv preprint arXiv:1708.02596* (2017).

[26] NGUYEN, P., AND NAHRSTEDT, K. Resource management for elastic publish subscribe systems: A performance modeling-based approach. In *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on* (2016), IEEE, pp. 561–568.

[27] NGUYEN, P., AND NAHRSTEDT, K. Monad: Self-adaptive micro-service infrastructure for heterogeneous scientific workflows. In *Autonomic Computing (ICAC), 2017 14th IEEE International Conference on* (2017), IEEE.

[28] PANDEY, S., WU, L., GURU, S. M., AND BUYYA, R. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *Advanced information networking and applications (AINA), 2010 24th IEEE international conference on* (2010), IEEE, pp. 400–407.

[29] PLAPPERT, M., HOUTHOOFT, R., DHARIWAL, P., SIDOR, S., CHEN, R. Y., CHEN, X., ASFOUR, T., ABBEEL, P., AND ANDRYCHOWICZ, M. Parameter space noise for exploration. In *International conference on learning representations* (2018).

[30] SILVER, D., LEVER, G., HEESS, N., DEGRIS, T., WIERSTRA, D., AND RIEDMILLER, M. Deterministic policy gradient algorithms. In *ICML* (2014).

[31] SURESH, L., BODIK, P., MENACHE, I., CANINI, M., AND CIUCU, F. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 611–623.

[32] SUTTON, R. S., AND BARTO, A. G. *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[33] WANG, Z., GWON, C., OATES, T., AND IEZZI, A. Automated cloud provisioning on aws using deep reinforcement learning. *arXiv preprint arXiv:1709.04305* (2017).

[34] WIECZOREK, M., PRODAN, R., AND FAHRINGER, T. Scheduling of scientific workflows in the askalon grid environment. *Acm Sigmod Record 34*, 3 (2005), 56–62.

[35] XU, Z., TANG, J., MENG, J., ZHANG, W., WANG, Y., LIU, C. H., AND YANG, D. Experience-driven networking: A deep reinforcement learning based approach. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications* (2018), IEEE, pp. 1871–1879.

[36] YU, J., AND BUYYA, R. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Scientific Programming 14*, 3-4 (2006), 217–230.

[37] YU, J., BUYYA, R., AND RAMAMOHANARAO, K. Workflow scheduling algorithms for grid computing. In *Metaheuristics for scheduling in distributed computing environments*. Springer, 2008, pp. 173–214.

[38] ZHOU, X., WANG, K., JIA, W., AND GUO, M. Reinforcement learning-based adaptive resource management of differentiated services in geo-distributed data centers. In *Quality of Service (IWQoS), 2017 IEEE/ACM 25th International Symposium on* (2017), IEEE, pp. 1–6.