MONAD: Self-adaptive Micro-service Infrastructure for Heterogeneous Scientific Workflows

Phuong Nguyen
University of Illinois at Urbana-Champaign
pvnguye2@illinois.edu

Klara Nahrstedt
University of Illinois at Urbana-Champaign
klara@illinois.edu

***Abstract*—Scientific workflows have become a popular computational model in a variety of application domains, such as astronomy, material science, physics, and biology. As scientific applications are moving to the cloud to take advantage of the elasticity and service level agreement of resources, there has been a number of recent research efforts on cloud-based workflow systems that support various types of performance guarantees under resource cost constraints. However, most of the related work often requires advanced knowledge about workflow structures to perform scheduling and resource optimization. In addition, existing workflow systems usually employ a monolithic approach in workflow implementation and execution, which makes them inefficient in dealing with heterogeneous types of workflows. In this paper, we present MONAD, a self-adaptive micro-service infrastructure for heterogeneous scientific workflows. Specifically, our micro-service architecture helps improve the flexibility of workflow composition and execution, and enables fine-grained scheduling at task level, considering task sharing across different workflows. In addition, we employ a feedback control approach with artificial neural network-based system identification to provide resource adaptation without any advanced knowledge of workflow structures. Our evaluation on multiple realistic heterogeneous workflows demonstrates that our system is robust and efficient in dealing with dynamic scientific workloads.**

I. INTRODUCTION

Scientific workflows [18][7], or workflows for short, which are typically presented as Directed Acyclic Graphs (DAG), have become a popular computational model for a variety of scientific domains, such as astronomy, biology, physics, and earth sciences, due to their ability to express complex data processing applications that consist of multiple computational steps and dependencies between them. Typically, given a workflow description (e.g., in the form of a DAG of tasks), a workflow management system (WfMS) will generate a workflow execution plan, considering resource and delay constraints, that specifies how the workflow can be executed on a computational infrastructure, such as clusters or grid.

Recently, with the increasing popularity of cloud infrastructure, there have been efforts [14][6] to deploy WfMS on the cloud to take advantage of the elasticity and service level agreement of cloud resources. Specifically, different from the best-effort resource offering as in cluster and grid environments, a cloud-based WfMS is able to adjust its computational resources on-demand to cope with the dynamism of application workload. As a result, it gives users more control over the expected processing delay of requested workflows, given users' resource budget constraints. It also opens the opportunity to offer WfMS via the Software-as-a-Service model that can reach a broader range of users with different requirements and constraints.

However, the cloud-based WfMSs also pose several challenges. *First*, cloud-based WfMSs should support heterogeneous types of workflow that are often interrelated to each other (e.g., sharing common tasks among different workflow types). Since most existing WfMSs employ a monolithic approach in workflow implementation (i.e., workflows are implemented as a tightly coupled set of tasks) and execution (i.e., an execution plan is derived for each individual workflow), they are not efficient in dealing with the heterogeneous workflows. *Second*, while WfMSs on clusters and grid often consider physical resources, such as CPU and memory, and try to allocate the workflow's tasks on servers/VMs to optimize resource utilization, cloud-based systems allow users to specify high-level compute resource requirements, scale individual tasks, and abstract away the actual allocation of tasks to physical servers/VMs. As a result, cloud-based WfMSs need to consider new resource abstraction and constraints (e.g., resource cost constraint, delay guarantees of individual workflow types) in their scheduling and resource provisioning formulation. *Third*, since existing workflow scheduling and resource allocation approaches usually require advanced knowledge about workflow structures (e.g., to leverage critical paths, partitioning of tasks in workflows), they are not suitable for cloud-based WfMSs that support heterogeneous types of workflows.

To address the above challenges, we present MONAD,[1] a novel self-adaptive micro-service infrastructure for heterogeneous scientific workflows. To deal with the *heterogeneity of workflows*, we design workflow execution mechanism based on a micro-service architecture, where each task is modeled as a micro-service with its own request queue and computing capability. With this design, we can separate complex

[1] MONAD stands for **MON**itoring and **AD**aptation. *Monad* is also a concept used in functional programming to refer to "*a way to build computer programs by joining simple components (or monads) in robust ways*" (Wikipedia).

task dependencies from the implementation of individual tasks, and thus, enable more flexible and scalable workflow composition and scheduling. In addition, the micro-service model allows us to simplify the system resource abstraction as the resource allocation over the set of micro-services, and thus, enable more complex resource optimization. To handle *various constraints*, we designed a feedback control-based resource adaptation approach that incorporates performance guarantees into the adaptation objective and seeks to find resource allocation strategies that satisfy resource budget constraints. Especially, our adaptation approach uses an artificial neural network, a black-box model, to perform system identification, and thus *does not require any advanced knowledge of workflow structures* supported by the system. The micro-service based execution model, the feedback control-based adaptation, together with a non-intrusive monitoring layer form a novel three-layer architecture of the MONAD system. Our evaluation on multiple sets of heterogeneous scientific workflows helps verify the effectiveness of MONAD in dealing with dynamic workloads.

In summary, our contributions are as follows:

- A novel three-layer system architecture of MONAD - a self-adaptive micro-service infrastructure for heterogeneous scientific workflows.
- A feedback control-based resource adaptation mechanism for WfMS and control algorithms that produce resource allocations without any advanced knowledge of workflow structures.
- Evaluation of the system using state-of-the-art technologies and realistic workflow ensembles with highly encouraging results.

The paper is organized as follows. We first describe the performance metrics and resource models in Section II. We present an architectural overview of the MONAD system in Section III. In Section IV, V, and VI, we respectively describe the design of each layer of the MONAD system. We show our evaluation results in Section VII and related work in Section VIII. Finally, we conclude the paper and present some future directions in Section IX.

II. ASSUMPTIONS AND MODELS

Before presenting the MONAD system architecture, we describe various system assumptions, performance metrics, guarantees, and resource models used by the system.

Performance Metrics: Let us assume that MONAD supports processing of N different types of heterogeneous workflows, each type is denoted as i, $1 \leq i \leq N$. We are interested in the *average processing time*[2] of each workflow type i, as well as the average delay of all workflow types. The processing delay of a request of a workflow type i that arrives to the system at time t, denoted as r_t^i, is defined as the duration between t and the time when the workflow's last task finishes. If we divide the time dimension into time windows $\{(T_k, T_{k+1})\}$ of equal lengths, the average delay of workflow type i over the time window (T_k, T_{k+1}), denoted as $\mathbf{r}_k^i$, is defined as $\mathbf{r}_k^i = \frac{1}{R_k^i} \cdot \sum_{T_k \leq t \leq T_{k+1}} r_t^i$ with R_k^i being the total number of requests of workflow type i during (T_k, T_{k+1}). We denote $\mathbf{r}_k$ as the vector form of the set of all average delays of workflow types in the k-th time window: $\mathbf{r}_k = (\mathbf{r}_k^1, \mathbf{r}_k^2, ..., \mathbf{r}_k^N)$. The average delay of requests over all types of workflows in the time window (T_k, T_{k+1}), denoted as $\bar{\mathbf{r}}_k$, is defined as $\bar{\mathbf{r}}_k = \frac{1}{\sum_{i=1}^N R_k^i} \cdot \sum_{i=1}^N \sum_{T_k \leq t \leq T_{k+1}} r_t^i$.

Performance Guarantees: In this paper, we use *absolute delay guarantee* for the average processing delay of individual workflow types (i.e., $\mathbf{r}_k^i$) and of all workflow types (i.e., $\bar{\mathbf{r}}_k$). Specifically, a delay threshold $\mathcal{T}_i$ $(1 \leq i \leq N)$ is assigned to each type of workflow i, so that the average delay of workflow type i over any time window (T_k, T_{k+1}) is guaranteed to be under the threshold: $\mathbf{r}_k^i < \mathcal{T}_i$. Similarly, a delay threshold $\mathcal{T}$ is used as the performance guarantee for all types of workflows: $\bar{\mathbf{r}}_k < \mathcal{T}$. To account for the different importance of different workflow types, we use λ_i to represent the relative importance of workflow type i (to be used in resource adaptation).

Task and Resource Model: 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, or all, of J types of tasks). We model each task type j $(1 \leq j \leq J)$ as a micro-service (cf. Section IV) that handles requests of the task type j. Specifically, the micro-service consists of a *request queue* that stores the task's requests, and a set of uniform *task consumers*[3] that subscribe to the request queue to perform actual processing of the task's requests. A workflow request is processed by multiple micro-services that correspond to the tasks in the workflow. Micro-services communicate with each other via a publish/subscribe middleware[4] (more details in Section IV).

We denote the configuration of the numbers of consumers over tasks during time window (T_k, T_{k+1}) as $\mathbf{m}_k = (m_k^1, m_k^2, ..., m_k^J)$, where m_k^j is the number of consumers of task type j during the k-th time window. To account for the difference in cost of consumers of different task types (e.g., the cost can be proportional to the computational capacity of the consumer), we use μ_j to present the cost of task j's consumers. Since the more consumers subscribe to a task's request queue, the more requests can be processed in parallel (and the less time requests must wait in the queue), $\mathbf{m}_k$ influences task's and workflow's processing times. Hence, we use $\mathbf{m}$ to represent *resource allocation*

[2]In the following, we use *processing time* and *delay* interchangeably.

[3]Consumers of a task have uniform computational capacity, in terms of CPU and memory, and this low-level resource information is abstracted away by the cloud infrastructure. Hence, the WfMS only needs to control the number of consumers for each task and task consumers become the computational representation of resource.

[4]We assume that all workflow data and intermediate results between tasks are stored in a shared storage system that can be accessed by all tasks, and the data transfer times are included in the task processing time.

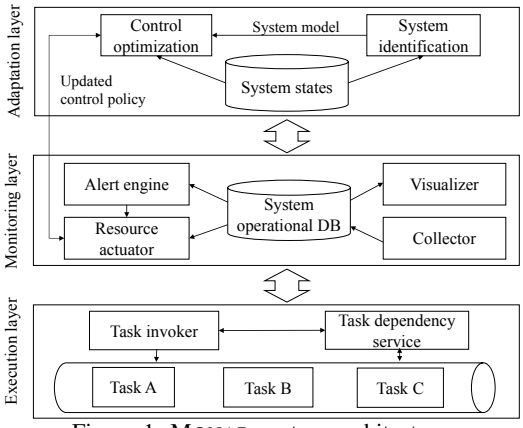


Figure 1: MONAD system architecture

decision[5] to be made by the system, so that it can adapt with the dynamism of incoming workload to satisfy various performance guarantees.

III. MONAD SYSTEM ARCHITECTURE

MONAD consists of three main layers (from bottom-up): *workflow execution layer*, *monitoring layer*, and *adaptation layer* (cf. Figure 1).

The ***workflow execution layer*** is in charge of executing requested workflows. To deal with the heterogeneity of workflows, we designed the execution layer using a *micro service-based architecture*, in which tasks are modeled as *micro-services* that interact with each other via event-based message passing mechanism. We present the design of the execution layer in detail in Section IV.

The ***monitoring layer*** monitors the performance of workflows and task's micro-services (e.g., processing times, arrival rates of workflow requests of different types). The *collector* collects performance information and stores them in a time series database, which then can be presented to system administrators via a visualization interface (i.e., *visualizer*). The *alert engine* component periodically checks the performance information in the database and makes alerts if any performance guarantee is violated. To respond to alerts, *resource actuator* will consult with the adaptation layer (to be described) for resource allocation decisions and then, perform resource reallocation to adapt system performance.

The ***adaptation layer*** provides control decisions, which are in the form resource allocations, to the monitoring layer, so that the system can adapt to the dynamism of the incoming workload. We designed adaptation layer using a feedback control-based mechanism. The layer consists of a *system identification* component that learns a system model based on historical performance data, and a *control optimization* component that leverages the learned system model to optimize the allocation of resources under performance and resource constraints.

[5]From now, we refer to *resource allocation decision* as $\mathbf{m}_k$ - the allocation of consumers over different task's micro-services.

IV. WORKFLOW EXECUTION LAYER

Traditional WfMSs 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. For example (cf. surveys in [18] and [7] for more details), *Pegasus* statically translates a workflow graph into an execution plan (e.g., including selecting sites for tasks to run and cluster tasks based on various criteria) and the plan cannot be changed once it is executed. In other systems, such as *Taverna*, *Triana*, or *Kepler*, all data movements and task submissions to grid infrastructure need to be explicitly specified and organized in the execution plan. In *Shock/AWE* [25], once being executed, the execution plan is often coordinated by a centralized server (and often, with a single task queue) that is in charge of task invocation and synchronization. This, the static, infrastructure-dependent execution plan creation, and centralized coordination mechanism make the existing systems less efficient in dealing with large-scale workloads of heterogeneous workflows.

In MONAD, we leverage the latest advances in cloud computing and virtualization technology to abstract away the infrastructure complexity (e.g., task allocation on actual servers/VMs is handled by a cluster management system, such as YARN or Kubernetes) and focus on the design of the workflow execution model. In particular, by modeling tasks as micro-services, we are able to separate the workflow's task dependencies from the implementations of individual tasks, and thus, enable more flexible and scalable workflow composition and resource scheduling (e.g., resource scaling can be done at the task level, instead of the whole workflow).

A. Micro Service-based Workflow Execution

The micro service-based architecture of MONAD's execution layer is depicted in Figure 2.

Each task is modeled as a micro-service 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*[6] (or TDS). Figure 3 shows an example of a task dependency table of workflows type 1 & 2 maintained by TDS. Whenever a workflow request arrives,[7] the *task invoker* asks the TDS which task of the workflow should be processed first. Upon receiving response from TDS, given a request of workflow type 1, for example, the task invoker will send the request to task A's

[6]Workflow's task dependencies are checked by TDS to make sure there is no cycle in the workflow.

[7]Only authorized users can send workflow requests to the system, and task invoker will check if the user has appropriate permissions to access the data required by the workflow request.

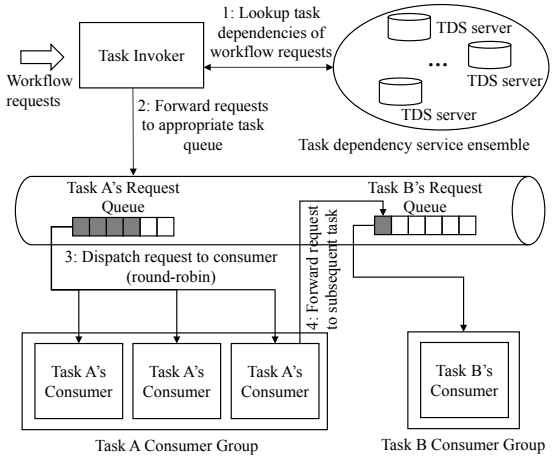
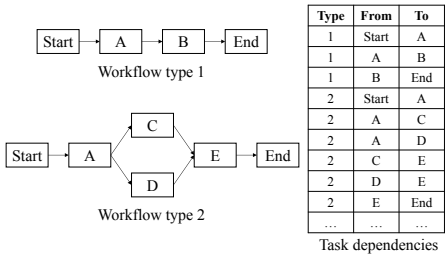


Figure 2: Design of workflow execution layer



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

Figure 3: Example of workflow types and the corresponding task dependencies

request queue (i.e., the first task of workflow type 1) so that it can be processed by one of A's consumers. 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 ask TDS about the subsequent task(s) of the workflow to "publish" the request to those tasks. For example, with a request of workflow type 1, after being processed by a task A consumer, the consumer will publish the request to task B's request queue. The processing of the request ends when task B's consumer is informed by the TDS that B is the last task of the workflow type 1.

B. Task Synchronization

In addition to answering task dependency look-ups, TDS is also responsible for synchronization between tasks that run in parallel. For example, in the workflow type 2 (Figure 3), task E must wait for both tasks C and D, which can run in parallel, to finish before E can be processed. Because of the publish/subscribe mechanism, when C and D finish their work, they will publish the workflow request to task E's queue. Let us assume that task C finishes before task D. The request published by C thus arrives first at task E's request queue and is picked up by an E's consumer. Since task E depends on the output of both C & D, the E's consumer could not perform task E yet. Therefore, the E's consumer creates a temporary *synchronization token* in TDS that holds status of E's dependencies. For simplicity, we can consider the synchronization token as a *dependency counter* that is initialized as the number of dependencies a task has minus one (in our example, the token is initialized to be 1, since E has two dependencies). When task D finishes, it publishes the workflow request to E. Another E's consumer that picks up the request published by D will check if a synchronization token for the request exists. Since a token has already been created, the consumer checks if the request is forwarded from the task's last dependency (i.e., current token value of 1). If not, the consumer decreases the token value by one and exits. In our example, since D is E's last dependency, the consumer will proceed to perform actual processing of task E.

C. Scalable Task Dependency Service

Since TDS is involved every time a task consumer is invoked (i.e., to perform synchronization), or during task dependency lookup, it is vital that the TDS is highly available and able to quickly respond to a large number of requests at the same time. To offer high availability and high performance, we designed TDS as an ensemble of multiple TDS servers and maintain a replica of task dependencies data on each server. For *read* requests (e.g., dependency lookup requests, token retrieval), any of the TDS servers can respond using its own local replica of task dependencies. Therefore, reads are quick and scalable. For *write* requests (i.e., for creating, updating synchronization tokens, and updating workflow's task dependencies for applications such as dynamic workflow composition), to guarantee consistency across multiple TDS servers, we use a *quorum-based* write mechanism with leader election. Specifically, one server from the set of TDS servers is elected as the leader. When a write request is sent to a server, the server passes on the request to the leader. This leader then issues the same write request to all other TDS servers. The write request is deemed successful only if a strict majority of the servers, or a quorum, responds successfully to this write request.

V. Monitoring Layer

To support performance guarantees, it is important to monitor system states to respond to abnormal performance in a timely manner. In the following, we present the design of our monitoring layer that captures the system performance measures in a non-intrusive way.

At a glance, the monitoring layer consists of four main components: *collector*, *visualizer*, *alert engine*, and *resource actuator* (cf. Figure 1). For each time window (T_k, T_{k+1}), the collector collects information about system performance metrics (i.e., $\mathbf{r}_k$), and the current allocation of resources (i.e., $\mathbf{m}_k$). The collected information is time-stamped and stored in a time series database, called *system operational database*. Visualizer retrieves real-time performance data from the time series database and displays it to system

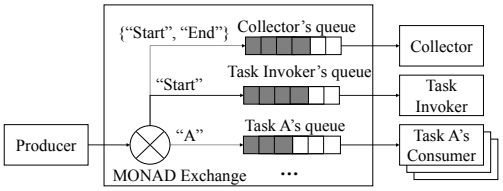


Figure 4: Leverage AMQP subscription model to perform non-intrusive monitoring

administrators via an interactive Web interface. The alert engine periodically checks on key performance metrics from the database and triggers alert if any performance guarantee is violated (e.g., when $\bar{\mathbf{r}}_k$ exceeds a processing time threshold $\mathcal{T}$). The triggered alert notifies resource actuator to consult adaptation layer to provide an updated allocation of resources (i.e., $\mathbf{m}_{k+1}$, for the next time window (T_{k+1}, T_{k+2})). Upon receiving $\mathbf{m}_{k+1}$, the resource actuator will perform re-allocation of resources on the execution layer.

The main challenge for the monitoring layer is to be able to capture performance information in a non-intrusive way and with little or no modification to the implementation of applications. Often, the monitoring feature is implemented as part of the APIs and applications have to explicitly make calls to monitoring APIs to record their performance (e.g., calling monitoring APIs when the application starts and ends to record processing time). In our MONAD system, we leverage the publish/subscribe middleware used in the execution layer to design a monitoring service that has *does not interfere with* to the performance and *does not require any modification* to the existing implementation of tasks and workflows.

Specifically, we leverage the subscription model of Advanced Message Queuing Protocol (AMQP), the open standard that has been supported by most publish/subscribe middlewares, to perform non-intrusive performance monitoring. In AMQP subscription model (cf. Figure 4), when messages arrive from publishers, they will be routed through an *exchange* to appropriate message queues. The routing decisions depend on the type of exchange used. In MONAD, we employ a *topic* type, in which the exchange routes messages to one or many queues (i.e., all queues receive the same copy of the message) based on matching between the message's *routing key* and a *pattern* that was used to bind a queue to an exchange. As shown in the design of the execution layer (cf. Section IV), each task holds its own message queue (e.g., task A's message queue is bound to the exchange to match messages with routing key "A") and has a set of consumers subscribe to its queue. We also use two pseudo tasks, i.e., "Start" and "End", to respectively represent the invocation and the completion of each workflow (i.e., the task invoker essentially becomes the consumer of "Start" message queue). Using these two pseudo tasks, we introduce

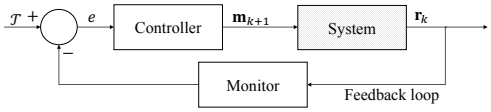


Figure 5: Feedback control system design

a separate message queue for the monitoring layer's collector that is binded to all messages with routing keys "Start" or "End". When collector processes a workflow request with routing key "Start", it creates a new entry for the request in the time series database to mark its arrival. When collector processes a workflow request with routing key "End", it updates the database entry of the started request to mark its completion and record the processing time.

To capture statistics about arrival workload (i.e., the number of arrival requests of a workflow type over a time window $\{R_k^i\}$), we simply perform an aggregation query over the time series database over that time windows to count the number entries having arrival time fall in between (T_k, T_{k+1}). Then, performance metrics $\{\mathbf{r}_k^i\}$, $\bar{\mathbf{r}}_k$ can be calculated by averaging the processing times of requests in the time window (T_k, T_{k+1}).

VI. ADAPTATION LAYER

We employ a control-theoretic approach in designing the adaptation layer (cf. Figure 5). Specifically, each time a performance guarantee is violated (e.g., $\bar{\mathbf{r}}_k$ exceeds $\mathcal{T}$), the adaptation layer is notified by the monitoring layer and the controller, as part of the adaptation layer, will generate a new allocation of resources (i.e., $\mathbf{m}_{k+1}$) based on the feedback e that captures the deviation of the performance metrics from the reference performance $\mathcal{T}$. There are typically two steps involved in developing a feedback control-based system: *system identification* and *controller design*. We present our solutions for each step in the remainder of this section.

A. System Identification

In the system identification step, we develop a mathematical model of the system that we want to control using measurements of the system's input and output signals. Particularly, given a control input[8] (i.e., $\mathbf{m}_k$ in our case) and the state of the system in the current time window (i.e., $\mathbf{r}_k$), the system model should be able to predict the performance of the system in the next time window (i.e., $\mathbf{r}_{k+1}$).

There are two common approaches to the system identification problem: *white-box* and *black-box* approach [9]. While the former assumes the understanding of how the system works internally (e.g., workflow structures) or the prior existence of a system model formulation; the latter assumes no prior model or knowledge of internal system is available. Due to its realistic assumptions, the black-box approach is more desirable and is our choice of approach in

[8]In the context of feedback control-based adaptation, we also use *control input* to refer to the allocation of resources.

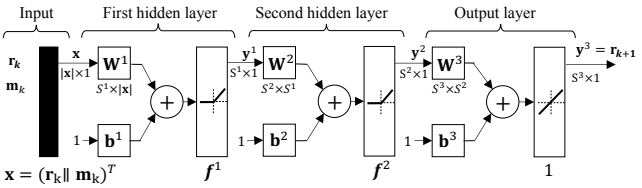


Figure 6: Neural network model of workflow system identification

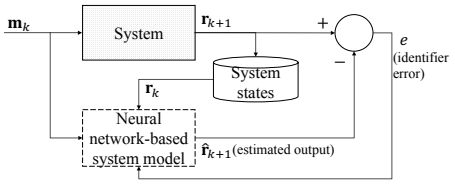


Figure 7: Training artificial neural network-based system identifier

this paper. In the following, we first present our design of the system model and then, the model training procedure.

1) System Model Design: There are many techniques to solve the system identification problem [9]. Capturing the performance model of complex dynamic systems, such as workflow systems, which are often non-linear and consist of multiple inputs, outputs with complex interactions, without knowledge of the system internals, requires techniques with good approximation power. In this paper, we use *multilayer neural networks*, which have proven approximation power and have been applied successfully in the identification of dynamic and non-linear systems with multiple inputs and outputs.

Our neural network model of the workflow system is presented in Figure 6. The network consists of two hidden layers[9] and one output layer, with the number of neurals in each layer is denoted as S^1, S^2, and S^3 respectively. Similarly, $\mathbf{W}^i, \mathbf{b}^i$ $(i = 1, 2, 3)$ represent the weight matrix and bias of each layer. Since most dynamic systems are non-linear, to introduce the non-linearity into the network model, we add a rectified linear unit (or ReLU) as the non-linear activation function (i.e., $\boldsymbol{f}^1$, $\boldsymbol{f}^2$) on the output of the two hidden layers. The neural network model takes input $\mathbf{x}$ as the combination of system states (i.e., $\mathbf{r}_k$) and control input (i.e., $\mathbf{m}_k$) in the current time window (T_k, T_{k+1}), and predicts output as the system states $\mathbf{r}_{k+1}$ of the next time window (i.e., (T_{k+1}, T_{k+2})). The neural network model can be presented as a matrix-based function of $\mathbf{r}_k$ and $\mathbf{m}_k$:

$$\begin{aligned}\mathbf{r}_{k+1} &= \boldsymbol{f}(\mathbf{r}_k, \mathbf{m}_k) \\ &= \mathbf{W}^3\boldsymbol{f}^2(\mathbf{W}^2\boldsymbol{f}^1(\mathbf{W}^1(\mathbf{r}_k \parallel \mathbf{m}_k)^T + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3\end{aligned} \quad (1)$$

2) Training System Identifier: The training procedure for system identifier's neural network model is presented in Figure 7. Specifically, the training process begins with the system model in an initial state and the control input is randomly generated. At the k-th time window (i.e., (T_k, T_{k+1})), the input of the neural network model is set as the combination of the current system states $\mathbf{r}_k$ and system's control input $\mathbf{m}_k$. The neural network model can be trained using the backpropagation algorithm for feedforward neural networks. The predicted next states of the system $\hat{\mathbf{r}}_{k+1}$ are compared with the output of the real system $\mathbf{r}_{k+1}$ (i.e., the reference values), and identifier error e is calculated for each of the neurons in the output layer. After that, the error values are propagated backwards through the network to update the model's parameters $\{\mathbf{W}^i, \mathbf{b}^i\}(i = 1, 2, 3)$.

[9]The structure of the network and its parameters are decided empirically, as currently there is not yet a theoretical foundation for the design of neural networks.

B. Controller Design

The controller design step uses the system model learned from the system identification step (i.e., function $\boldsymbol{f}(\mathbf{r}_k, \mathbf{m}_k)$) to design a controller that can produce control inputs to guide the system to follow a desired output. The three most common approaches for controller design, given a learned neural network system model are: model predictive control, NARMA-L2 control, and model reference control [13]. To incorporate various performance and cost constraints into controller design, we employ the model predictive control methodology and treat the controller design problem as an optimization problem using the receding horizon technique [8].

Specifically, at the k-th time window, when the performance constraint is violated (e.g., $\bar{\mathbf{r}}_k$ exceeds $\mathcal{T}$), the controller seeks to produce new control inputs $\mathbf{m}_{k+1}$ to guide the system back to comply with the performance constraints. Using the receding horizon technique, we solve a control optimization problem over fixed $\mathtt{T}$ future intervals, from time window $(k+1)$-th to $(k+\mathtt{T})$-th, to obtain a sequence of the next $\mathtt{T}$ control inputs $\mathbb{M} = \{\mathbf{m}_\tau\}$, $k+1 \le \tau \le k+\mathtt{T}$ that minimize the deviations from the predicted values to the reference trajectory (i.e., $\mathcal{T}$) over $\mathtt{T}$ time windows while satisfying the resource budget constraint (i.e., $\mathcal{C}$ denotes the maximum number of consumers in the system). The control optimization problem to solve at time k-th is formally defined as follows:

$$\begin{aligned} \underset{\substack{\mathbb{M}=\{\mathbf{m}_\tau\} \\ k+1\le\tau\le k+\mathtt{T}}}{\text{argmin}} \quad & \sum_{\tau=k+1}^{k+\mathtt{T}} l(\mathbf{r}_\tau, \mathbf{m}_\tau) \\ \text{subject to} \quad & \mathbf{r}_{\tau+1} = \boldsymbol{f}(\mathbf{r}_\tau, \mathbf{m}_\tau), k \le \tau \le k+\mathtt{T}-1 \\ & \sum_{j=1}^{J} m_\tau^j \le \mathcal{C}, k+1 \le \tau \le k+\mathtt{T} \\ & \mathbf{m}_\tau \in \mathbb{Z}_+^J, k+1 \le \tau \le k+\mathtt{T} \end{aligned} \quad (2)$$

where $l(\mathbf{r}_\tau, \mathbf{m}_\tau)$ is defined as:

$$l(\mathbf{r}_\tau, \mathbf{m}_\tau) = \sum_{i=1}^{N} \lambda_i \cdot (\mathcal{T}_i - \mathbf{r}_\tau^i)^2 + \sum_{j=1}^{J} \mu_j \cdot (\Delta m_\tau^j)^2 \quad (3)$$

In the above optimization problem, $l(\mathbf{r}_\tau, \mathbf{m}_\tau)$ is the instantaneous cost function at time τ $(k+1 \le \tau \le k+\mathtt{T})$ that captures the deviation of system output $\mathbf{r}_\tau$ from reference performance $\{\mathcal{T}_i\}$, while also accounting for the control increments (i.e., $\Delta m_\tau^j = m_\tau^j - m_{\tau-1}^j$) over different types

of resources. In case we use performance guarantee $\mathcal{T}$ on the average processing time across all types of workflows, we can replace the first term in Equation 3 (i.e., $\sum_{i=1}^{N} \lambda_i \cdot (\mathcal{T}_i - \mathbf{r}_\tau^i)^2$) by $(\mathcal{T} - \bar{\mathbf{r}_\tau})^2$ to capture the deviation from the reference performance $\mathcal{T}$.

The model predictive control-based adaptation algorithm, denoted as `MPCAdapt`, is presented in Algorithm 1. The algorithm starts by initializing the control sequence $\mathbb{M}$ using the same current control input $\mathbf{m}_k$ for all `T` future time horizons (Line 4-5), and the set of constraints (Line 6-9). After that, we solve the optimization problem (i.e., `minimize` function) with objective function `mpc_obj_func` described in problem (2) (Line 11).

It is easy to observe that the optimization problem (2) is a *constrained non-linear integer programming* problem (i.e., because decision variables $\{\mathbf{m}_\tau\}$ are positive integers and the objective function is in quadratic form with non-linear component $\boldsymbol{f}(\mathbf{r}_\tau, \mathbf{m}_\tau)$), whose complexity is NP-hard. To solve the problem *efficiently*, we relax problem (2) into a *constrained non-linear optimization problem* (i.e., by relaxing the integrality constraint of control inputs) and use the *Sequential Least Squares Programming optimization algorithm* [16] (or SLSQP), an iterative method, to solve the relaxed problem. The non-integer solution then can be used to approximate the integer solution for the original problem.

After finding the sequence of control input $\mathbb{M}^*$, only the first "control move" $\mathbb{M}^*[0]$ is returned and is used as the control input for the next time window $\mathbf{m}_{k+1}$. As we move to the $(k+1)$-th time window, we repeat the same control optimization process (i.e., `MPCAdapt`) for the next `T` time windows (i.e., from $(k+2)$-th to $(k+\mathtt{T}+1)$-th). The process ends when the system satisfies the performance constraints.

Algorithm 1 Model Predictive Control-based Adaptation

```
1:  procedure MPCAdapt(m_k, T, {T_i}, C)
2:      M = {}
3:      C = {}
4:      for ι in [0, T − 1] do # Initialize M
5:          M.append(m_k)
6:      for ι from [0, T − 1] do # Add cost constraints
7:          C = C ∪ {C − sum(M[ι]) ≥ 0}
8:      for ι from [0, T − 1] do # Add positive resource constraints
9:          C = C ∪ {M[ι][j] ≥ 0, ∀j ∈ [0, J − 1]}
10:     # Solve the optimization problem as described in (2):
11:     M* = minimize(mpc_obj_func_{{T_i},T}, M, C)
12:     # Only return the first control move:
13:     Return M*[0]
```

Even though we are able to solve the optimization problem (2) in an online manner by relaxation, it is still inefficient to run `MPCAdapt` on a high dimensional input and over a large number of of time windows. Therefore, we also propose a *heuristic-based dynamic control algorithm* (Algorithm 2) to efficiently produce control inputs for the system to meet performance guarantees while satisfying resource constraints. Specifically, different from `MPCAdapt`, in `HeuristicAdapt`, we only consider the next time window, instead of looking ahead `T` time windows, and we iteratively (and greedily) allocate additional resources to the task that produces minimal instant cost (i.e., `instant_cost` function, based on Equation 3), instead of trying to solve a global optimization problem.

Algorithm 2 Heuristic Adaptation Algorithm

```
1:  procedure HeuristicAdapt(m_k, r_k, {T_i}, C)
2:      while {∃i ∈ [1, N] : r_k^i > T_i} and Σ_{j=1}^{J} m_k^j ≤ C do
3:          cur_min = −1.0
4:          j_min = −1
5:          for j from 1 to J do
6:              m_k^j = m_k^j + 1
7:              if instant_cost(r_k, m_k) < cur_min then
8:                  cur_min = instant_cost(r_k, m_k)
9:                  j_min = j
10:             m_k^j = m_k^j − 1
11:         m_k^{j_min} = m_k^{j_min} + 1
12:     Return m_k
```

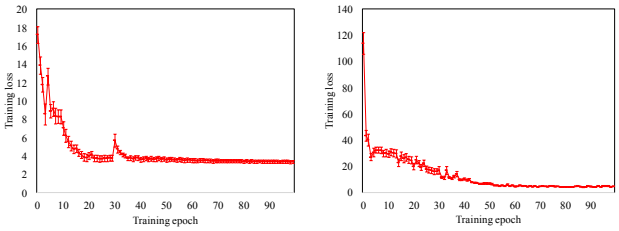
VII. Evaluation

A. MONAD *Deployment*

In the following, we describe in detail our implementation of the MONAD system. We have deployed the MONAD system on a cluster of three servers, each server is equipped with an Intel Xeon quad core processor (1.2Ghz per core) and 16GB of RAM.

For the workflow execution layer, we implement TDS service using ZooKeeper and use a quorum of 3 TDS servers to maintain tasks dependencies data. We use RabbitMQ as the publish/subscribe middleware and we implement each task's micro-service as an ensemble of a RabbitMQ's task request queue and a set of task consumers, each consumer is deployed as a Docker container. We use the round-robin dispatching mechanism for each task's request queue so that multiple requests can be processed in parallel and each consumer receives a fair share of requests. In terms of fault tolerance, to make sure a request never gets lost (e.g., because of consumer crashes), we employ a message acknowledgment mechanism between request queue and its consumers: An acknowledgement is sent back from the consumer to tell the request queue that the consumer has finished processing a request.

We use Kubernetes as the container orchestration engine for the execution layer. With Kubernetes, we can abstract the set of consumers of each task's micro-service as a Kubernetes' Replication Controller, which helps ensure that, in the event of a container crash and server failure, a specified number of containers per task (or scaling factor) is always running at any time. Upon receiving a control input (i.e., $\mathbf{m}_k$) from the adaptation layer, the resource actuator simply instructs Kubernetes to change the scaling factor of each task's replication controller.



(a) MDP (b) LIGO

Figure 8: Effectiveness of training system identifiers

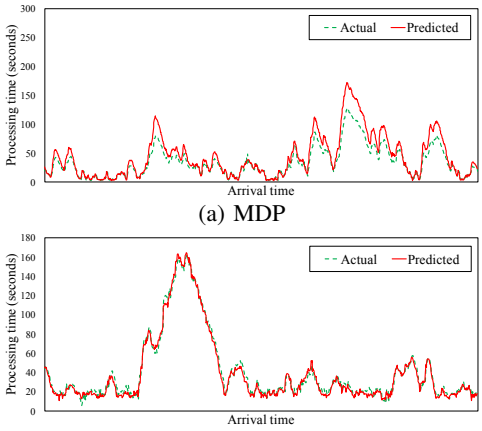
For the monitoring layer, we use InfluxDB as the time series database to store the monitoring data, and Grafana as the visualization engine to provide real-time system performance status to administrators via an interactive interface. We use Kapacitor as the alert engine that monitors the time series database and invokes adaptation process whenever a performance guarantee is violated. Other components in monitoring layer are implemented using Python.

For the adaptation layer, we use Tensorflow to train system identifiers and use Python's SciPy optimization package to solve the control optimization problem.

B. Evaluation Settings

Workflows: We use two workflow ensembles: the material science data processing workflows [24] (or MDP for short) that support processing experimental data generated by digital microscopes; and LIGO Inspiral Analysis workflows [15] that analyze data from the coalescing of compact binary systems such as binary neutron stars and black holes. The MDP ensemble consists of 3 types of workflows and 4 types of tasks, and we use 3 complex workflows from LIGO: CAT, Full, and Injection, which consist of 7 types of tasks.

System Identification: To generate training and testing data for the system identification step, we randomly generate a workload by varying the arrival rates of requests of different workflow types and vary the allocation of resources over tasks. This way of generating data to train system identifiers allows us to capture a good variety of workload and resource allocation dynamics, so that our trained models can be better prepared for future workload situations. In addition, this allows us to collect the training data without the need of bootstrapping the system. The performance data (i.e., $\mathbf{r}_k$) and resource allocation (i.e., $\mathbf{m}_k$) are then captured by the monitoring layer and stored into the time series database for training. The data are aggregated over equal-length time windows. Via our experiments, we choose the window length (i.e., $T_{k+1} - T_k$) to be 10 seconds as it helps produce the best balance between the prediction accuracy and the data collection overhead (i.e., a too long time window might not capture the dynamism of workload, while a too short one might cause data aggregation overhead). Our collected dataset is aggregated from about 60K requests of MDP and about 15K requests of LIGO workflows. The data is then split using 80:20 ratio for training and testing.



(a) MDP

(b) LIGO

Figure 9: Effectiveness of system identification on predicting average performance on different workflow ensembles

In terms of the neural network's model parameters, we set S^1 and S^2 equal to 32 neurals in each hidden layer for MDP, 64 neurals for LIGO workflows (as LIGO has a higher input dimension). To train system identifiers for both sets of workflows, we set learning rate as 0.001, batch size as 100, and used 100 training epochs. Figure 8 shows the effectiveness of training system identifier using neural network model on two workflow ensembles.

Resource Adaptation: To evaluate the effectiveness of different adaptation algorithms in handling varying and bursty workload situations (i.e., workload with abnormally high arrival rate of requests), we emulate the bursty workload situation by abnormally increasing the arrival rates of requests of different types of workflows to up to 10 and 5 times compared with the normal rates on MDP and LIGO workflows (respectively), and measure the effectiveness of our resource allocation strategy. In this evaluation, we use the absolute delay guarantee for $\bar{\mathbf{r}}_k$: $\bar{\mathbf{r}}_k < \mathcal{T}$, and set $\mathcal{T}$ equal 10 and 30 seconds for the MDP and LIGO workflows respectively. We set the resource constraint $\mathcal{C}$ for the MDP and LIGO worklows to be 15 and 90 maximum number of consumers respectively.

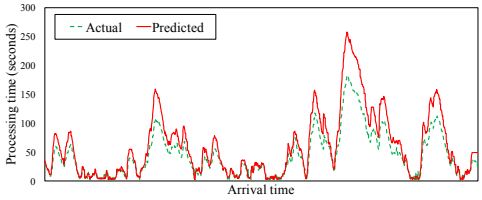
For the `MPCAdapt` algorithm, we set $\mathtt{T} = 5$, and use a uniform cost for resources $\mu_j = 1 \forall j$ and the same weight for all workflow types $\lambda_i = 1 \forall i$ [10].

C. Effectiveness of System Identification

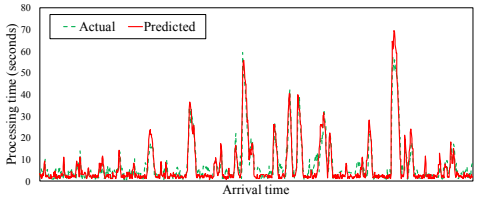
Figure 9 shows clearly the effectiveness of the trained neural network-based system identifier on accurately predicting the average processing delay over all workflow types on both MDP (cf. Figure 11a) and LIGO (cf. Figure 9b) workflow ensembles.

Further study of the results shows that our system identifier also performs well on predicting the processing time of *individual* workflow types, as shown in Figure 10 on

[10]The costs and weights can be easily set to more realistic values (if available) without affecting the performance of the algorithm.

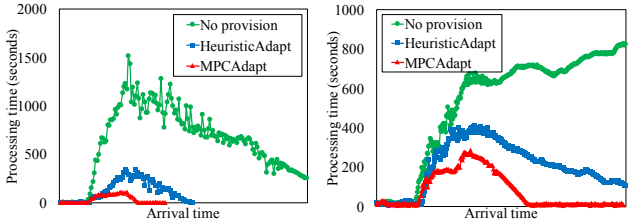


(a) MDP's DM3 file processing workflow



(b) MDP's experiment classification workflow

Figure 10: Effectiveness of system identification on predicting processing time of individual MDP workflows



(a) MDP (b) LIGO

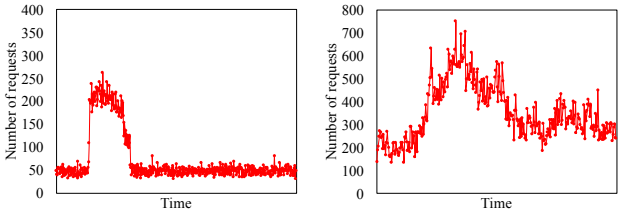
Figure 11: Effectiveness of adaptation algorithms on adapting system performance when dealing with bursty workload

two of MDP's workflows. Although these MDP's workflows pose different workload and performance characteristics, the system identifier can accurately predict the processing time of each type. These results help verify the effectiveness of using neural network-based model with multiple outputs, one for each workflow type, to capture the dynamic system model with complex interactions between tasks across different workflow types.

D. Effectiveness of Adaptation Algorithms

As we can see in Figure 11, the `MPCAdapt` algorithm outperforms `HeuristicAdapt` as it helps quickly neutralize the effect of abnormal and bursty workload on the performance of the system. This is because `MPCAdapt` can produce better resource adaptation strategies by solving the optimization problem over `T` future time windows, instead of using heuristic as in `HeuristicAdapt` which can lead to local optimum. In addition, by looking ahead a future time horizon and taking one control move at a time, the `MPCAdapt` algorithm can adjust its adaptation strategies to the external factors, such as changes in arrival workload. The `HeuristicAdapt` algorithm shows good promise[11] as it offers acceptable performance while being more efficient to compute.

[11] Although the results show that `HeuristicAdapt` is more efficient than `MPCAdapt`, we leave the evaluation on more complex workflow ensembles with higher number of dimensions (or tasks) for future work.



(a) MDP (b) LIGO

Figure 12: Incoming requests to TDS during the bursty workload (aggregated every 5 seconds)

E. Efficiency of TDS

To evaluate the efficiency of the TDS, we capture the number of requests (aggregated every 5 seconds) which arrive at TDS during the bursty workload experiment above (for both MDP and LIGO workflows). As expected, when additional consumers are allocated to tasks to process increasing workflow requests, the number of requests sent to TDS also increases (i..e., both dependency lookups and synchronization inquiries). Despite the increase in the number of requests to TDS, the maximum latency of responses by TDS is only 22ms for MDP and 37ms for LIGO workflows, which are insignificant compared with the workflow processing time. These results help verify the efficiency of TDS service in handling dynamic workload.

VIII. Related Work

Scientific workflow management systems [18][7] have traditionally employed a monolithic approach, in which 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 [3]. To generate workflow execution plans, there have been related works on scheduling and task allocation under cost and deadline constraints of individual workflow [27][1], or workflow ensembles [21]. However, techniques developed in these works mostly focus on optimizing execution order of tasks and the allocation of tasks (in order to minimize costly data transfer over highly distributed environments such as grid) and require advanced knowledge of workflow's structure (e.g., critical paths, partitioning tasks in workflows). In this paper, we consider cloud-based workflow system scenario, in which data transmission is no longer the main bottleneck in terms of processing delay due to the cloud's high-speed network infrastructure, and we focus our resource adaptation mechanism on scaling of resource over micro-services. Besides, our adaptation approach is based on black-box system identification that does not require any knowledge of workflow structures.

In addition, while related works that leverage grid and cloud-based infrastructure [14][6] often use resource models based on VMs and virtual resources, our resource model is based on the allocation of consumers over task's micro-services. We leverage the latest container technology and implement consumers as containers, which not only helps

abstract away the allocation of physical resources like CPU and memory, but also offers better server consolidation and finer-grained resource provisioning.

Real-time monitoring is important to control workflow execution [7], and is still an open issue. The common approaches for workflow monitoring are still based on analyzing execution log data [12], or provenance data [5], and often require extra implementation effort to collect such data. In this paper, we leverage our micro service-based architecture and the publish/subscribe middleware to perform seamless monitoring of workflows and tasks.

Feedback control has been applied to a variety of application domains, such as web server [20], distributed visual tracking system [17], adaptive real-time systems [19], and more recently, on auto-scaling of cloud applications [10] [26] [11]. In this paper, we employ feedback control mechanism to enable self-adaptive and QoS-aware cloud-based scientific workflow systems. In addition, while the related efforts mainly focus on scaling and adaptive reconfiguration at VM level (i.e., IaaS cloud), we focus on finer-grained and at a higher abstraction level of resources (i.e., containers and scaling of tasks).

There are also related work on developing prediction model (especially using machine learning techniques [22]) to accurately predict performance and resource demand of applications to make informed decisions on resource scaling and reconfiguration. For example, AGILE [23] uses wavelets to provide a medium-term resource demand prediction, Matrix [4] uses clustering models with probability estimates to predict the performance of new workloads, Chandra et. al. [2] uses time series analysis techniques to predict expected workload parameters. In this paper, we show the effectiveness of artificial neural network in predicting the performance of scientific workfow applications.

IX. Conclusions and Future Work

In conclusions, in this paper, we presented the design and implementation of MONAD, a self-adaptive micro-service based infrastructure for heterogeneous scientific workflows. Our micro-service based approach helps improve the flexibility and scalability of workflow composition and execution by separating task dependencies from its implementation. In addition, our feedback control-based resource adaptation approach allows us to generate resource allocation decisions without any advanced knowledge of workflow structures. For future work, we plan to take into account workload patterns and admission control mechanism to provide a more proactive approach in resource adaptation. In addition, we also plan to study resource adaptation in the scientific workflow-as-a-service scenario where workflow workloads pose much more dynamic and heterogeneous characteristics.

Acknowledgment

We would like to thank anonymous reviewers for insightful comments and our shepherd Hausi Muller for guiding the preparation of the camera-ready paper. This research was funded by the National Science Foundation NSF ACI 1443013. The opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the National Science Foundation.

References

[1] S. Abrishami et al. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Comp. Sys.*, 29(1):158–169, 2013.

[2] A. Chandra et al. Dynamic resource allocation for shared data centers using online measurements. In *Int. Workshop on Quality of Service*, pages 381–398. Springer, 2003.

[3] W. Chen and E. Deelman. Partitioning and scheduling workflows across multiple sites with storage constraints. *Parallel Processing & Applied Mathematics*, pages 11–20, 2012.

[4] R. C.-L. Chiang et al. Matrix: Achieving predictable virtual machine performance in the clouds. In *Autonomic Computing (ICAC)*, pages 45–56. USENIX, 2014.

[5] F. Costa et al. Capturing and querying workflow runtime provenance with prov: a practical approach. In *ACM Joint EDBT/ICDT Workshops*, pages 282–289, 2013.

[6] E. Deelman. Grids and clouds: Making workflow applications work in heterogeneous distributed environments. *International Journal of High Performance Computing Applications*, 24(3):284–298, 2010.

[7] E. Deelman et al. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Comp. Sys.*, 25(5):528–540, 2009.

[8] S. DI. Neural generalized predictive control: A newton-raphson implementation. 1997.

[9] G. F. Franklin et al. *Feedback control of dynamic systems*, volume 3. Addison-Wesley Reading, MA, 1994.

[10] A. Gandhi et al. Adaptive, model-driven autoscaling for cloud applications. In *Autonomic Computing (ICAC)*, pages 57–64. USENIX, 2014.

[11] H. Ghanbari et al. Optimal autoscaling in a iaas cloud. In *Autonomic Computing (ICAC)*, pages 173–178. ACM, 2012.

[12] D. Gunter et al. Online workflow management and performance analysis with stampede. In *IFIP CNSM*, pages 152–161, 2011.

[13] M. T. Hagan et al. An introduction to the use of neural networks in control systems. *International Journal of Robust and Nonlinear Control*, 12(11):959–985, 2002.

[14] C. Hoffa et al. On the use of cloud computing for scientific workflows. In *IEEE eScience*, pages 640–645, 2008.

[15] G. Juve et al. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013.

[16] D. Kraft. Algorithm 733: Tomp–fortran modules for optimal control calculations. *ACM Transactions on Mathematical Software (TOMS)*, 20(3):262–281, 1994.

[17] B. Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, 1999.

[18] J. Liu et al. A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13(4):457–493, 2015.

[19] C. Lu et al. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1):85–126, 2002.

[20] C. Lu et al. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, 2006.

[21] M. Malawski et al. Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *Future Generation Comp. Sys.*, 48:1–18, 2015.

[22] A. Matsunaga and J. A. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *IEEE/ACM CCGrid*, pages 495–504, 2010.

[23] H. Nguyen et al. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Autonomic Computing (ICAC)*, pages 69–82. USENIX, 2013.

[24] P. Nguyen and K. Nahrstedt. Resource management for elastic publish subscribe systems: A performance modeling-based approach. In *IEEE CLOUD*, pages 561–568, 2016.

[25] W. Tang et al. A scalable data analysis platform for metagenomics. In *IEEE Big Data*, pages 21–26, 2013.

[26] L. Wang et al. Qos-driven cloud resource management through fuzzy model predictive control. In *Autonomic Computing (ICAC)*, pages 81–90. IEEE, 2015.

[27] J. Yu et al. Cost-based scheduling of scientific workflow applications on utility grids. In *IEEE e-Science and Grid Computing*, pages 8–pp, 2005.