BRAHMA^{+}: A Framework for Resource Scaling of Streaming and ASAP Time-Varying Workflows

Ankita Atrey, Gregory Van Seghbroeck, Bruno Volckaert, and Filip De Turck

***Abstract*—Automatic scaling of complex software-as-a-service application workflows is one of the most important problems concerning resource management in clouds. In this paper, we study the automatic workflow resource scaling problem for *streaming* and ASAP workflows, and its time-varying variant where the workflow resource requirements change over time. Service components of *streaming* workflows execute concurrently while those of ASAP workflows execute sequentially. We propose an intelligent framework, BRAHMA^{+}, which possesses the capability to learn the workflow behavior and construct a knowledge base that serves as its decision making engine. The proposed resource provisioning algorithms leverage this learned information curated in the knowledge base to perform informed and intelligent scaling decisions. Additionally, BRAHMA^{+} employs the use of *online-learning* strategies to keep the knowledge base up-to-date, thereby accommodating the changes in the workflow resource requirements over time. We evaluate the proposed algorithms using *CloudSim* simulations. Results on streaming and ASAP workflows, with both static and time-varying resource requirements show that the proposed algorithms are effective and produce good cost-quality trade-offs. The proactive and hybrid algorithms meet the service level agreements and restrict deadline violations to a small fraction (3%–5% in the considered scenarios), while only suffering a marginal increase in average cost per component compared to the described baseline algorithms.**

***Index Terms*—Cloud resource provisioning, workflows, cloud scalability, adaptive clustering, knowledge base, deadline-constraints, SLA, cloud simulation.**

I. Introduction

Cloud enabled services have become an integral part of the day-to-day life of almost every Internet user. Cloud users enjoy *flexible* and *cost-effective* usage of various cloud services, however, providing *quality of service* – meeting SLAs, scalability, and deadline constraints – while maintaining cost-effectiveness with *highly dynamic resource requirements* exhibited by end-user requests, is the paramount concern of various service providers.

Having said that, *resource management* continues to be one of the most fundamental and important areas of research in the field of cloud, distributed, and grid computing. While there exists a plethora of research in devising industry scale resource management systems especially by Internet giants like Google [42], Facebook [5], Microsoft [10], [21], Alibaba [48] etc., the focus of these systems have been on execution environments like grids and clusters, and such efforts have been relatively *scarce for SaaS application workflows in cloud-based systems* [22]. In this article, we propose a unified framework, BRAHMA^{+}, for resource scaling in clouds.

Manuscript received June 26, 2017; revised December 23, 2017 and April 9, 2018; accepted April 15, 2018. This research is partly funded by the IWT SBO DeCoMAdS project. The associate editor coordinating the review of this paper and approving it for publication was L. Y. Chen. *(Corresponding author: Ankita Atrey).*

The authors are with the Internet Technology and Data Science Laboratory, Ghent University–imec, 9052 Gent, Belgium (e-mail: ankita.atrey@ugent.be; gregory.vanseghbroeck@ugent.be; bruno.volckaert@ugent.be; filip.deturck@ugent.be).

Digital Object Identifier 10.1109/TNSM.2018.2830311

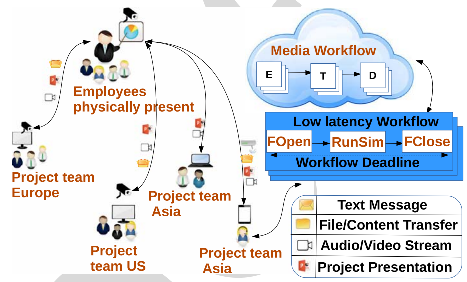


Fig. 1. Use case: An online collaborative meeting room service.

A. Use Case: Online Collaborative Meeting

The use case under investigation (Fig. 1) is an elastic, multi-tenant online meeting room offering an interactive collaboration service. This use case is inspired by the EMD project [4], which investigates scalable A/V collaboration applications (streams) and deadline-critical jobs (such as decision support, data analysis, etc.) triggered by the end-user during these collaborations. The project leader is presenting an interactive media (A/V streaming) session, where some colleagues are present in the meeting room, while others are connected remotely. Every stream consists of an encoder, a transcoder and a decoder, all of which have different SLA requirements in an attempt to provide a flawless service (no A/V interruptions, stuttering, etc.). The A/V stream encapsulates a *streaming workflow* (Definition 4), which is similar to the long-running services presented in Fig. 2 that should not experience any downtime. Each attendee can additionally trigger low-latency/deadline-critical workflows during the meeting to, e.g., run analytical simulations (file-open→run-simulation→file-close workflow in Fig. 1) and show the results to the meeting audience or render high quality graphics. We use the term *ASAP workflows* (Definition 7) to denote these low latency jobs. While streaming workflows usually constitute one or more service components executing concurrently, the components of ASAP workflows are executed sequentially,

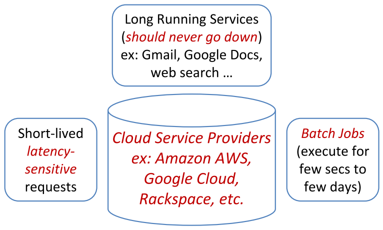


Fig. 2. Types of jobs in cluster/grid/cloud computing environments [42].

thereby exhibiting runtime characteristics which differ from that of the former. Users can join or leave these online meetings at any point in time, leading to potentially large fluctuations in terms of number of tenants using the system, and each user can trigger multiple ASAP workflows during the session.

In the context of the use-case discussed above, meeting the strict deadline constraints of ASAP workflows and SLAs of streaming workflows (for A/V quality), while keeping cloud resource cost as low as possible, is the focus of this work. This scenario presents a plethora of challenges stated as follows:

(1) Scaling the resulting application up or down in order to keep the SLAs, no longer becomes an issue of scaling resources for a single service, but instead results in a complex problem of scaling all individual service endpoints in the workflow, depending on their runtime monitored behavior [9]. (2) As described above, heterogeneity of application workflows (Fig. 2) leads to a host of characteristics: ranging from execution flows being either *sequential* or *concurrent* [8] to the nature of the deadlines associated with the workflows being either strict or fuzzy. (3) The resource requirements of the jobs submitted to a cloud environment are usually not static. Rather, in real-world cloud environments the type of workflows submitted by users and their resource requirements change over time [31], [47]. For instance, given the meeting room use-case, a user might trigger a high-quality graphic render as an ASAP workflow during a weekly-update meeting, while during a technical deep-dive session, she might trigger an analytical simulation. Thus, the resource management algorithms should be adaptive thereby enabling effective resource provisioning for such *time-varying* workflows.

To effectively address the challenges besetting the resource management problem in real-world scenarios for clouds, there is a need for a framework, tailored to serving *cloud* service workflow requests, that (1) possesses the capability to handle different types of workflows, (2) intelligently performs resource provisioning tasks under specified deadlines or SLAs, and (3) possesses algorithms that adapt to the temporally changing resource requirements posed by these workflows.

This article presents a framework, *BRAHMA*$^+$, which incorporates the use of mathematical models (classification and clustering) to *learn the workflow behavior* and curates a *knowledge base (KB)* that aids in taking informed future resource provisioning decisions. These models analyse the resource request patterns of workflows to predict whether a new workflow will meet its deadline or not, and clusters the workflows into groups possessing similar resource requirements. Moreover, for time-varying workflows we design and implement online versions of the proposed learning algorithms, where the learned models are updated with temporally changing data. To further enhance the efficacy and efficiency of the time-aware learning process, we use a sliding window, which controls the amount of historical data to be used for learning at any given time instant, thereby improving both the quality (helps ignoring irrelevant historical data) and efficiency (learning from a relevant fraction of the complete data).

In sum, in this article we address the *automatic workflow resource scaling* problem (Section III) under the combined presence of *streaming and ASAP workflows*, called *AWS-SA*, and its time-varying variant, called *AWS-tSA* where the workflow resource requirements change over time.

Key contributions are as follows:

- A novel framework, *BRAHMA*$^+$ (Section IV), which learns workflow behaviour over time and stores this information in a KB. BRAHMA$^+$ possesses the capability to predict workflow deadlines and cluster these workflows into semantically meaningful groups. Additionally, the online learning algorithms of BRAHMA$^+$ are capable of adapting to the changes in resource request patterns exhibited by time-varying workflows.
- Resource provisioning algorithms (Section V) that leverage BRAHMA$^+$ to maintain SLAs and deadlines for streaming and ASAP workflows respectively, as well as their time-varying variant, while keeping the cost in line.
- Empirical analysis (Section VII) portraying the effectiveness of the proposed algorithms. Our algorithms keep the SLAs and restrict deadline violations to 3–5%, while only suffering a marginal increase in the average resource utilization cost of 5–8% over the baselines.

II. Related Work

Resource management and scheduling [22] is a fundamental and one of the most extensively studied problems in the field of cloud computing. Here, we provide an overview of the existing works that overlap with the work presented in this article.

Workflows provide a natural and attractive choice for representing a host of SaaS applications, thus, *automatic workflow resource scaling and scheduling* [13] with focus on maintaining quality of service parameters, like *SLAs* [29] and *deadline-constraints* [6], [38], has been a hot topic of research in the broad area of cloud resource management. The readers are referred to [43] for a detailed and recent survey.

SLA-aware resource provisioning: focusses on strategies for resource scaling to keep the SLAs in line while minimizing cost. Wu *et al.* [44], [45], presented SLA-aware resource provisioning algorithms for SaaS providers. The authors propose maximum and minimum available space based resource reservation and request rescheduling strategies, while using customer profiles to handle dynamic and changing customer requests. Later, the authors developed a method for admission control of user requests [46], thus facilitating prevention of additional user requests that would lead to SLA violations from being accepted.

Serrano *et al.* presented a new model: SLA aware service (SLAaaS) [37], proposed a language for describing SLAs and an approach using control-theory for keeping the SLA of cloud applications. Focussing on *application workflows*, Atrey *et al.*

TABLE I
SUMMARY OF NOTATIONS USED

Item	Definition
$\mathcal{V}$	The pool of VMs; $\forall i, V_i \in \mathcal{V}$.
$\mathcal{W}_s$	Set of streaming workflow requests.
$\mathcal{W}_a$	Set of ASAP workflow requests.
$\mathcal{W}$	Set of workflow requests; $\forall j, W_j \in \mathcal{W} = \mathcal{W}_s \cup \mathcal{W}_a$.
$\mathcal{D}_s(t)$	Time varying distribution for streaming worfklow requests.
$\mathcal{D}_a(t)$	Time varying distribution for ASAP worfklow requests.
C_{kj}	A service component for the workflow W_j; $\forall k, C_{kj} \in W_j$.
$MI_{C_{kj}}$	The number of instructions (in MI) required to execute C_{kj}.
DC_{W_j}	The deadline constraint of $W_j \in \mathcal{W}_a$.
$SLA^{C_{kj}}_{status}$	The status (binary) of the SLA of C_{kj}, i.e. met or violated.
$DEADLINE^{W_j}_{status}$	The status (binary) of the Deadline of W_j, i.e. met or violated.
η	Fraction of ASAP workflows with violated deadlines.
$MIPS_i$	The processing power in MIPS of VM V_i.
$\mathcal{CC}$	Set of identified clusters and their cluster centers, $\rho = \lvert\mathcal{CC}\rvert$.
$\mathcal{CM}$	Classification model.
$\mathcal{N}^i_{running}$	Number of components currently running on VM V_i.
$\mathcal{N}^i_{max}$	Maximum number of components allowed on VM V_i.
τ	Parameter controlling how quickly the *Proactive* algorithm intervenes in terms of scaling resources up or down.
$sw[sw_{start}, sw_{end}]$	The sliding window sw with window size = $\lvert sw_{end} - sw_{start}\rvert$.
λ	The rate of information decay over time.
$T^{V_i}_{reserve}$	Time required for reservation of a new VM V_i.
$T_{migrate}$	Time required for migrating a service component to V_i.
$P^{V_i}_{reserve}$	Penalty incurred due to reservation of a new VM V_i.
$P_{migrate}$	Penalty incurred owing to migrating components to V_i.

proposed a *pro-active* algorithm [9] that uses a monitoring mechanism to track the run time behavior of each workflow component and horizontally scales resources accordingly, thereby avoiding SLA violations. Singh *et al.* [39] studied the effect of various QoS parameters on the rate of SLA violations, and proposed an autonomic pipeline along with a knowledge store to devise effective resource provisioning strategies.

In addition to research on devising strategies for SLA-aware resource provisioning, a few studies have performed benchmarking and validation [7], [17] of various SLA-aware models and resource provisioning algorithms.

Deadline-aware resource provisioning: focusses on devising strategies to minimize the deadline violations of jobs submitted to clouds. Genez *et al.* [16] present an Integer Linear Programming (ILP) based algorithm for scheduling SaaS workflows in IaaS clouds, which finds the mapping between workflow tasks and VMs provided by the IaaS providers to minimize the overall cost and achieve deadline constraints.

Poola *et al.* present robust and fault-tolerant resource scheduling algorithms with three multi-objective resource allocation policies in [31]. Moving ahead, Rodriguez and Buyya [34] applied a genetic algorithm (Particle Swarm Optimization) in order to obtain an optimized solution in terms of cost, deadline and elasticity, highlighting resource provisioning techniques for scientific workflows on IaaS. Luo *et al.* propose a resource provisioning algorithm [26] for hybrid settings comprising both grids and clouds. The idea is to estimate the probability of deadline violation of a sub-task in a workflow, and then later redirect certain sub-tasks from grids to intelligently selected virtual resources on clouds, in order to achieve strict workflow deadline-constraints. Recently, Atrey *et al.* [8] presented a framework called BRAHMA (which has been *significantly* extended to BRAHMA$^+$ in this article) that used workflow clustering and a curated KB to perform resource provisioning for workflows with strict deadline-constraints.

Resource provisioning using machine learning: is a relatively recent paradigm in clouds, where researchers have incorporated the use of various classification and clustering methods for learning and characterizing workflow behaviour [23], [25]. Mon *et al.* [28] proposed workflow clustering based on task dependencies with an aim to minimize the data transfer overhead of data-intensive scientific workflows. Moving ahead, Peng *et al.* [30] presented a machine learning framework that used a radial basis function based neural network to estimate application resource requirements, and a k-means based genetic clustering algorithm for performing multi-objective optimization to solve the resource provisioning problem. Atrey *et al.* [8] proposed a machine learning framework that curates all the learned information in a KB. Very recently, Li *et al.* [24] present a resource scheduling algorithm that uses fuzzy clustering methods to identify different resource clusters thereby simplifying their allocation to jobs.

Resource provisioning for dynamic workflows: addresses workflows with dynamically changing resource requirements. To the best of our knowledge, research in this context has been scarce. Zhang *et al.* [47] presented ROSA, an online randomized algorithm that stacks the execution of multiple jobs submitted to the cloud environment, thereby achieving spatial multiplexing. This helps minimize cost with the capability to leverage volume discounts offered by cloud service providers, while also keeping the job-level constraints in line. Poola *et al.* [32] presented an adaptive resource provisioning algorithm that is capable of incorporating the use of both spot and on-demand instances, thereby minimizing the total cost: as it leverages the price benefit from spot instances, and ensuring fault-tolerance by meeting workflow level deadlines using on-demand instances whenever necessary. Recently, adaptive resource scheduling [14] has also been studied in the context of software defined networks [41]. Rodriguez and Buyya [35] proposed a container-based algorithm that can adapt to changes in the workload, while mitigating inefficiencies in resource utilization and meeting workflow level deadlines.

Despite wide-spread research in the broad area of workflow resource scheduling [43], to the best of our knowledge, none of the existing state-of-the-art methods are capable of solving this problem in a *holistic* manner. Specifically, the existing works have focused on the two problems, i.e., scaling streaming and deadline-critical workflows independently, however, a *unified* and *generic framework* possessing capabilities of collectively scaling both types of workflows is non-existent. Additionally, research on devising algorithms that adapt to *temporally changing* resource requirements of workflows has been scarce. To this end, this article presents an enabling, unified, and adaptive framework, BRAHMA$^+$, with algorithms for provisioning cloud resources to streaming (SLA-aware) and ASAP workflows (with strict deadline-constraints), whose resource requirements change over time.

III. PROBLEM STATEMENT

This section provides a concise model of streaming and ASAP workflows, with an introduction of their basic concepts followed by a formal description of the *AWS-SA* problem, and its variant *AWS-tSA*, focussing on time-varying workflows. Table I summarizes the notations used in the rest of the article.

Definition 1 (Resource): A resource corresponds to a processing unit with specifications defined in terms of processing power (in MIPS), memory (in GB), storage (in GB), and network bandwidth (in Mbps).

Cloud computing environments offer virtual resources in the form of virtual machines (VMs), containers etc. In this study, we consider a pool of resources called a VM pool $\forall i,\, V_i \in \mathcal{V}$.

Since cloud-based applications are usually built as *workflows* integrating multiple existing services (albeit with custom glue code tying all of them together), an application workflow is defined as follows:

Definition 2 (Application Workflow): An application workflow $W_j(C, E)$ is defined as a directed acyclic graph (DAG) comprising a set of service components C_j and a set of edges E_j. Each component $\forall k, C_{kj} \in C_j$ represents an atomic task in the application workflow W_j, while each directed edge $\forall k, l, e = (C_{kj}, C_{lj}) \in E_j$ defines the dependency of the component C_{lj} on component C_{kj}.

Definition 3 (Workflow Resource Requirements): The resources required by a component C_{kj} of a workflow W_j are defined in terms of number of instructions to be executed (measured in millions of instructions ($MI_{C_{kj}}$)), the memory and storage space required (($Mem_{C_{kj}}$) in GB), and the number of bits to be transferred over the network (($BW_{C_{kj}}$) measured in millions of bits (Mb)).

Application workflows can possess a wide-variety of characteristics in terms of execution flow, resource requirements etc. Based on these characteristics we next define the two types of workflows considered in this article.

Definition 4 (Streaming Workflow): A streaming workflow $W_j \in \mathcal{W}_s$ is an application workflow where service components $C_{lj} \in C_j$ continuously receive streaming data from other components $C_{kj} \in C_j$ via directed edges $e = (C_{kj}, C_{lj}) \in E_j$, while they themselves stream their output data to other workflow components following their execution.

For instance, if the workflow in Fig. 3 is a streaming workflow, C_{21} would continuously stream output data to C_{31} and C_{41}, who in turn would process that input data and stream it to C_{51}. All service components are processing information in parallel. Each streaming workflow service component possesses a separate SLA, which defines its minimal resource requirements (in terms of processing power, memory, storage etc.) to ensure proper working of the workflow according to its specifications. Using this component-level minimal resource requirement and the resource specification (processing power, memory, storage etc.) of the assigned VM $V_i \in \mathcal{V}$, we define the maximum number of components $\mathcal{N}^i_{max}$ that can simultaneously run on $\mathcal{V}_i$ while ensuring SLAs are met. For example, given a VM with processing power as 1500 MIPS and the minimal resource requirement per component to be 50 MI, the maximum number of components that can be scheduled on this VM is 30. Note that for simplicity each VM is assigned to components of one specific type, i.e., those possessing the same minimal resource requirement. Using $\mathcal{N}^i_{max}$ we further define the SLA status of a component and the average SLA violation duration.

Definition 5 (SLA Status): The SLA status for each service component C_{kj} of a workflow W_j running on a VM V_i is

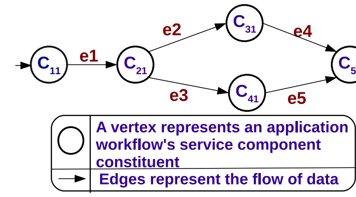


Fig. 3. An application workflow W_1 composed of multiple service components and inter-component data flows.

defined as a binary variable which assumes the value of *false* if the SLAs are violated, or *true* otherwise. Mathematically,

$$SLA^{C_{kj}}_{status} = \begin{cases} false, & \text{if } \mathcal{N}^i_{running} > \mathcal{N}^i_{max} \\ true, & \text{otherwise} \end{cases}$$

where, $\mathcal{N}^i_{running}$ denotes the number of components currently running on the VM V_i.

Definition 6 (Average SLA Violation Duration): The SLA violation duration of a service component C_{kj} is defined as the amount of time for which its $SLA^{C_{kj}}_{status}$ is violated over its runtime duration. Thus, for a simulation with w workflow requests $W_j \in \mathcal{W} \mid 1 \leq j \leq w$, c_j service components $C_{kj} \in W_j \mid 1 \leq k \leq c_j$, and $T^{C_{kj}}_{slaviolate}$ being the duration for which the SLAs remain violated for a component C_{kj}, we mathematically state the average SLA violation duration as:

$$\frac{1}{w}\left(\sum_{j=1}^{w}\left(\frac{1}{c_j}\sum_{k=1}^{c_j}\left(T^{C_{kj}}_{slaviolate}\right)\right)\right). \tag{1}$$

Definition 7 (ASAP Workflow): An ASAP workflow $W_j \in \mathcal{W}_a$ is an application workflow where the execution flow between service components is sequential. More specifically, the execution control moves from one component $C_{kj} \in C_j$ to the subsequent workflow component(s) $C_{lj} \in C_j$, once the former finishes processing thereby passing its full output to the latter.

Again as an example, if the workflow in Fig. 3 would be sequential, C_{21} would, once it has finished processing, send all its output data in parallel along the edges *e2* and *e3* to C_{31} and C_{41} respectively. At that point in time, C_{31} and C_{41} start processing. Additionally, each ASAP workflow possesses a deadline-constraint (DC_{W_j}) which is used to identify a VM $\mathcal{V}_i$ that possesses the desired resources to ensure proper working of the workflow according to its specifications.

Definition 8 (Deadline Status): The deadline status of a workflow W_j, running on a VM V_i, is defined as a binary variable which assumes the value of *false* if its deadline-constraint DC_{W_j} is not met, and *true* otherwise. The fraction of the workflows whose deadlines are violated is denoted by η. Mathematically,

$$\eta = \frac{1}{w}\left(\sum_{j=1}^{w} I\right) \tag{2}$$

where I is the indicator function: $I = 1$ if $DEADLINE^{W_j}_{status} = false$; and 0 otherwise.

Definition 9 (VM Cost): VM cost is defined as the sum of all costs related to resource usage when running streaming and ASAP workflow service components. Thus, for a simulation with *w* workflow requests, each one with c_j service components, and M_{kj}, S_{kj}, CPU_{kj}, representing, memory, storage and CPU costs respectively for a component C_{kj}, we mathematically define VM cost and average VM cost as follows:

$$\sum_{j=1}^{w}\left(\sum_{k=1}^{c_j}\left(M_{kj}+S_{kj}+CPU_{kj}\right)\right) \tag{3}$$

$$\frac{1}{w}\left(\sum_{j=1}^{w}\left(\sum_{k=1}^{c_j}\left(M_{kj}+S_{kj}+CPU_{kj}\right)\right)\right). \tag{4}$$

Definition 10 (Penalty): The Penalty is the cost spent on components, while waiting for (1) a new VM reservation $P_{reserve}$ and (2) migration of components from one VM to another $P_{migrate}$. We mathematically state the Penalty and the average Penalty as follows:

$$\sum_{j=1}^{w}\left(\sum_{k=1}^{c_j}\left(P_{reserve\,kj}+P_{migrate\,kj}\right)\right) \tag{5}$$

$$\frac{1}{w}\left(\sum_{j=1}^{w}\left(\frac{1}{c_j}\sum_{k=1}^{c_j}\left(P_{reserve\,kj}+P_{migrate\,kj}\right)\right)\right) \tag{6}$$

The technical problem studied in this work is inspired by the use case of online collaborative A/V meetings where both streaming and ASAP workflows co-exist. Note that tenant requests for streaming and ASAP workflows follow time-varying distributions $\mathcal{D}_s(t)$ and $\mathcal{D}_a(t)$ respectively. While streaming workflows do not benefit from assigning more resources to them than required, as one cannot *'speed up'* an online collaborative meeting, ASAP workflows definitely benefit from finishing early (meeting their deadlines), when allocated to more powerful resources. Owing to this significant difference in characteristics, scaling the service end-points of applications where streaming and ASAP workflows co-exist is a challenging problem. We name this problem *AWS-SA*, and formally define it as:

Problem 1 (AWS-SA): Given a VM pool $\mathcal{V}$, a set of workflow requests ($\mathcal{W}$) consisting of a combination of streaming ($\mathcal{W}_s$) and ASAP ($\mathcal{W}_a$) workflow requests, following time varying distributions $\mathcal{D}_s(t)$ and $\mathcal{D}_a(t)$ respectively, the maximum number of allowed requests ($\mathcal{N}^i_{max}$) and the processing power ($MIPS_i$) for each VM ($\forall V_i \in \mathcal{V}$), perform automatic resource provisioning to keep the SLAs ($SLA^{C_{kj}}_{status} = true$) and the deadline-constraints ($DEADLINE^{W_j}_{status} = true$) for all the workflow components $C_{kj} \mid \forall k, C_{kj} \in W_j, \forall j, W_j \in \mathcal{W}$, while simultaneously minimizing the vm cost and penalty.

In addition to the number of streaming and ASAP workflows changing over time (denoted by $\mathcal{D}_s(t)$ and $\mathcal{D}_a(t)$), the actual resource requirements of workflows change as well. To this end, we address the temporal variant of the AWS-SA problem called the *AWS-tSA* problem.

Problem 2 (AWS-tSA): The AWS-SA problem where the resource requirements ($MI_{C_{kj}}, Mem_{C_{kj}}, BW_{C_{kj}}$) of workflow components, $C_{kj} \mid \forall k, C_{kj} \in W_j, \forall j, W_j \in \mathcal{W}$, change over time.

IV. BRAHMA$^+$ Framework

In this section, we present the BRAHMA$^+$ framework and provide a description of its core components. In this article, BRAHMA [8] has been significantly extended using *online learning* strategies as BRAHMA$^+$ to *learn* workflow request behavior in an *online* manner, i.e., without the need for training data to bootstrap the learning process. This enables BRAHMA$^+$ to handle workflows whose resource requirements change over time. We also provide insights about the way in which BRAHMA$^+$ facilitates development of effective resource provisioning strategies.The building blocks of the BRAHMA$^+$ framework are detailed next.

- *Classification (Build Classifier):* analyses the resource requirements and request patterns exhibited by ASAP workflows, and learns a decision boundary, using historical resource requirement data of workflows submitted to a cloud environment, capable of predicting whether the deadline of a previously unseen workflow would be met or violated. The main benefit that the classifier module provides is the ability to predict the $DEADLINE_{status}$ of (previously unseen) incoming ASAP workflows, facilitating more informed resource provisioning decisions.
- *Clustering (Identify Clusters):* allows for fine-grained analysis of the behaviour exhibited by ASAP workflows. Here, the resource request patterns are clustered, thereby creating semantically meaningful groups of ASAP workflows, with each group possessing similar resource requirements. The advantage of clustering is that once these clusters are identified, it is easier to devise customized and informed resource provisioning strategies pertaining to each cluster. Moving ahead, any previously unseen ASAP workflow request can then be assigned to its most similar group, and hence utilize the already devised resource provisioning strategy for that group.
- *Online Clustering:* extends the clustering module by making it flexible and adaptive to effectively accommodate changes in data distributions originating from time-varying workflows. More specifically, since the resource requirements of ASAP workflows may change over time, the identified clusters have to change as well, as the clusters generated from older data will have been invalidated. Unlike conventional methods, where clustering is performed as a single-shot process comprising two steps: (1) cluster identification, and (2) cluster assignment; online clustering methods continuously learn from the data, i.e., the identified clusters are refined as and when newer data-points are ingested by the system. Moreover, to ensure consistency the identified clusters are updated regularly in the knowledge base.
- *Knowledge Base (KB):* is one of the most important components of the BRAHMA$^+$ framework as it curates all the information learned from the classifier and the clustering modules. More specifically, the KB stores an updated copy of the classifier model and the set of identified cluster centres. For each submitted ASAP workflow, the resource provisioning algorithms probe the KB to identify the cluster closest (most similar) to this workflow, thereby assigning it to an appropriately sized VM with

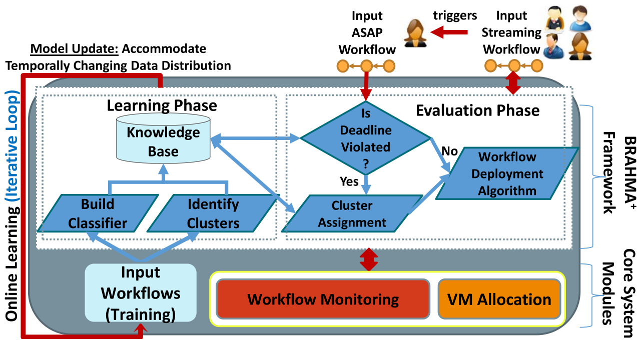


Fig. 4. Overview of the BRAHMA$^+$ Framework.

the aim to meet its deadline constraints. The KB thus serves as the decision making body for the entire framework, and hence, is kept *up-to-date* with all the changes resulting from the online learning process.

- *Workflow Monitoring:* keeps track of the progress for each component C_{kj} of a workflow W_j. More specifically, it continuously probes the VMs and the workflow components to monitor the number of components running on a particular VM and the time remaining for the component to finish execution respectively. As will be explained later in Section V, the monitoring capability plays a central role in the design of the more involved *pro-active* and *hybrid* resource provisioning algorithms for streaming and ASAP workflows respectively.
- *VM Allocation:* facilitates on-demand creation of new VM instances based on a specific VM template from the pool of VMs $\mathcal{V}$. The core function of this module is to perform VM allocations based on the events triggered by the workflow monitoring module and the information retrieved from the KB. VM allocations broadly happen in two ways: (1) VMs are reserved in the beginning and remain fixed throughout; (2) VM reservations happen dynamically and their specifications are adapted based on the submitted workflow resource requirements.

A. Learning Phase

BRAHMA$^+$ (Fig. 4), operates in two phases. Firstly, in the *learning phase*, BRAHMA$^+$ takes as input workflow requests submitted to the cloud environment, which serves as its training data. To facilitate robustness and generalizability of the learned models, BRAHMA$^+$ keeps on updating its models incrementally to ensure modelling a proper mix of workflows with varying number of components, component types etc. Each workflow W_j, possesses resource usage statistics (in MI) for each of its constituent component $\forall k, C_{kj}$, while also containing information about its deadline status (i.e., was the deadline violated or met).[1] The first task of BRAHMA$^+$'s learning phase is that of building a classifier. Here, we use the classification module (described earlier in this section) to analyse the generated training data and learn a classifier model $\mathcal{CM}$, based on the resource requirements and request patterns, for answering the binary question: *whether the deadline of an ASAP workflow is met or violated?*

[1]The workflow generation process is described in detail in Section VI.

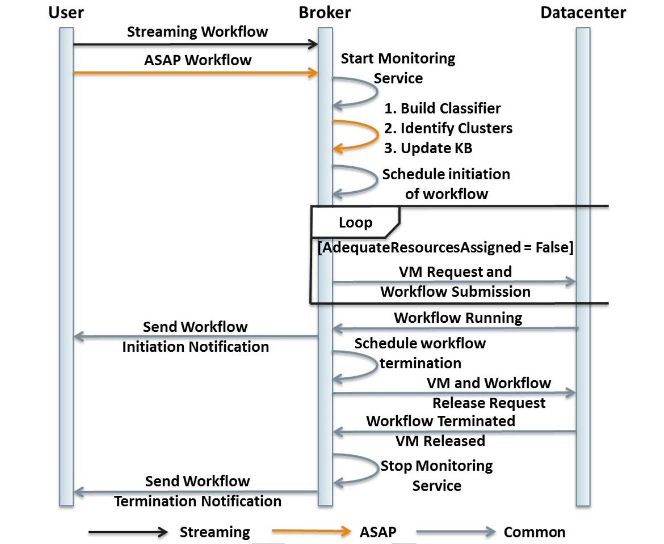


Fig. 5. Sequence diagram portraying the execution flow of both Streaming and ASAP workflows.

BRAHMA$^+$ independently clusters similar workflows (based on the resource requirements and request patterns of its constituent components) from the training data to form semantically meaningful groups or clusters. As explained above, the (online) clustering module allows analysis of the workflow behavior at a finer level of granularity, facilitating appropriate resource provisioning decisions with the aim to avoid deadline violations. Eventually both the classifier model $\mathcal{CM}$ and the created set of clusters along with their cluster centers $\mathcal{CC}$, are curated in the *Knowledge Base (KB)*.

The learning process described till now lacks the capability to tackle time-varying workflows. Hence, to effectively solve the *AWS-tSA* problem where workflow resource requirements vary over time, BRAHMA$^+$ employs the use of online learning strategies. As is clear from Fig. 4, the learning phase does not represent a single-shot conventional machine learning pipeline, rather it is iterative and continuously refines the learned classifier model $\mathcal{CM}$ and the identified cluster centers $\mathcal{CC}$. More specifically, as and when BRAHMA$^+$ receives newer training data, it is ingested in the learning phase, the models are updated, and eventually these updates are propagated to the KB thereby facilitating the evaluation phase to employ the most recently learned models for resource provisioning.

B. Evaluation Phase

In the *evaluation phase*, new streaming requests along with triggered ASAP requests are submitted to BRAHMA$^+$ for inferring their execution behavior, resource requirements and deadline status. As a first step, BRAHMA$^+$ probes the classifier model $\mathcal{CM}$ saved in the KB to predict the $DEADLINE_{status}$ of the workflow under consideration, i.e., whether its deadline would be met or not. If the deadline is going to be met, then there is no need to perform any specialized resource scaling, as the already assigned resources will be sufficient to meet the deadline-constraint of the workflow. However, if a violation is predicted, we

Algorithm 1 Workflow Deployment Algorithm

Input: $\mathcal{V}$, τ, *windowSize*, λ, $\mathcal{N}^i_{max} \mid \forall V_i \in \mathcal{V}$, $DC_{W_j} \mid \forall W_j \in \mathcal{W}$, $\mathcal{W} \sim \mathcal{D}(t)$, $provisionType$, $workflowType$
Output: SLA_{status}, $DEADLINE_{status}$, η, *AvgCost*
1: $numRunning \leftarrow 0$
2: **for** each $V_i \in \mathcal{V}$ **do**
3: $\quad \mathcal{N}^i_{running} \leftarrow 0$
4: **for** $t = 0$ to t_{max} **do**
5: $\quad \mathcal{W}^t \sim \mathcal{D}(t)$; $numStreamDeploy \leftarrow |\mathcal{W}^t_s| - numRunning$
6: $\quad KB^t = \{\mathcal{CM}^t, \mathcal{CC}^t\} \leftarrow OnlineLearn(t, \mathcal{W}^t \sim \mathcal{D}(t), windowSize, \lambda)$
7: $\quad$ **if** $numStreamDeploy \geq 0$ **then**
8: $\quad\quad$ **if** $workflowType = Streaming$ **then**
9: $\quad\quad\quad \{SLA_{status}, AvgCost_s\} \leftarrow ProactiveDeploy(\mathcal{W}^t_s, \mathcal{V}, \tau)$
10: $\quad\quad$ **else** //workflowType = *ASAP*
11: $\quad\quad\quad numASAPDeploy \leftarrow |\mathcal{W}^t_a|$
12: $\quad\quad\quad$ **if** $numASAPDeploy \geq 0$ **then**
13: $\quad\quad\quad\quad \{DEADLINE_{status}, \eta, AvgCost_a\} \leftarrow KBQuery(\mathcal{W}^t_a, \mathcal{CM}^t, \mathcal{CC}^t, \mathcal{V})$
14: $\quad$ **else** //Terminate Streaming Workflows
15: $\quad\quad$ **for** each $W_j \in \mathcal{W}^t_s$ **do**
16: $\quad\quad\quad$ **for** each $C_{kj} \in W_j$ **do**
17: $\quad\quad\quad\quad$ Cancel C_{kj} and free its resources on V_i
18: $\quad\quad\quad\quad \mathcal{N}^i_{running} \leftarrow \mathcal{N}^i_{running} - 1$
19: $\quad\quad\quad\quad$ **if** $\mathcal{N}^i_{running} \leq \mathcal{N}^i_{max}$ **then**
20: $\quad\quad\quad\quad\quad SLA_{status} \leftarrow false$
21: $\quad numRunning \leftarrow numRunning + numStreamDeploy$
22: $\quad AvgCost \leftarrow AvgCost_s + AvgCost_a$

Algorithm 2 Online Learning Algorithm

Input: t, $\mathcal{W}^t \sim \mathcal{D}(t)$, $windowSize$, λ
Output: $KB^t = \{\mathcal{CM}^t, \mathcal{CC}^t\}$
1: **procedure** OnlineLearn$(t, \mathcal{W}^t \sim \mathcal{D}(t), windowSize, \lambda)$
2: $\quad sw_{end} \leftarrow t - 1$
3: $\quad sw_{start} \leftarrow sw_{end} - windowSize$
4: $\quad$ **if** $sw_{start} < 0$ **then**
5: $\quad\quad sw_{start} \leftarrow 0$
6: $\quad$ **for** $t' = sw_{start}$ to sw_{end} **do**
7: $\quad\quad \alpha \leftarrow \lambda^{(sw_{end} - t')}$
8: $\quad\quad \mathcal{W}^{t'} \sim \mathcal{D}(t')$
9: $\quad\quad \mathcal{CM} \leftarrow$ Update the classifier model using $\alpha * \mathcal{W}^{t'}$
10: $\quad\quad \mathcal{CC} \leftarrow$ Update the identified clusters using $\alpha * \mathcal{W}^{t'}$
11: $\quad$ **return** $KB^t = \{\mathcal{CM}^t, \mathcal{CC}^t\}$

query the KB's cluster centers $\mathcal{CC}$ to assign this workflow to the cluster closest/most-similar to it in terms of exhibited resource requirements, thereby guiding the resource provisioning algorithms. While the predictions from the classifier module facilitate the decision: whether to scale the resources provisioned to a workflow up or down, the cluster assignments from the clustering module (if the workflow was predicted to violate its deadline) provide information about the resource requirements of a workflow, thereby providing guidance on ways to scale the resources effectively. This information is further employed to predict the deadline status of newly incoming ASAP workflows.

Having described the key components, two phases: learning and evaluation, and the overall execution flow of the BRAHMA$^+$ framework in detail, we present a sequence diagram of BRAHMA$^+$ in Fig. 5. The sequence diagram explains the execution flow of streaming and ASAP workflows, while also providing an in-depth explanation of the interaction between various components of the BRAHMA$^+$ framework using the entities in context of CloudSim [12].

V. Resource Provisioning Algorithms

To effectively perform resource provisioning for both time-varying and static workflows, we present a generic workflow deployment algorithm with pseudo-code listed in Algorithm 1. The number of workflow requests submitted to BRAHMA$^+$ follow a time-varying distribution $\mathcal{D}(t)$. To this end, we sample requests at different discrete time-instants (line 5). If the workflow under consideration is a *streaming* workflow, we invoke the proactive algorithm (Section V-B) to scale its services, with the objective of completely mitigating SLA violations and maintaining high cost-efficiency (lines 8 and 9). On the other hand, for *ASAP* workflows the resource provisioning is performed using the hybrid algorithm (Section V-C), which in turn probes the curated information from the KB to take appropriate decisions (lines 10–13). Note that as indicated in Fig. 6 streaming and ASAP workflows are provisioned on separate resource pools and thus, the resource provisioning decisions made for one will not interfere with that of the other and vice-versa. Once workflows have been successfully executed, the resources allocated to them are freed and corresponding bookkeeping information is updated (lines 14–20).

As discussed previously, the resource requirements exhibited by the submitted workflows may change over time. Since the KB acts as an important decision making unit of the workflow deployment algorithm, the information curated here should be consistent and up-to-date with the latest monitored workflow requirements and older (stale) information about now-defunct workflow requirements should in time be phased out. This is achieved using online learning (line 6), where newly arriving workflows are used to update the classifier model and identified clusters. Specifically, we incorporate the use of a *sliding window* based approach, which is described in the subsequent sub-section.

Note that the workflow deployment algorithm is capable of handling both time-varying and static workflows. With the value of $windowSize = 0$, there is no window constructed and the algorithm works for static workflows, while on the other hand, any non-zero value of the $windowSize$ enables the algorithm to work for time-varying workflows.

A. Online Learning Algorithm

Algorithm 2 presents the pseudo-code for the *online learning* algorithm. To keep the KB up-to-date with the changing workflow resource requirement patterns, we need to update the learned mathematical models – classification model $\mathcal{CM}$ and identified cluster centers $\mathcal{CC}$. Therefore, the models need to be updated in an *online* manner. Since the changes in data can be large, updating models for every new incoming request is highly inefficient and impractical. To this end, we use a sliding window based approach to handle all the updates. For every newly arriving request at time *t*, a window $sw[sw_{start}, sw_{end}]$, where $sw_{end} = t - 1$ and $sw_{start} = sw_{end} - windowSize$, is constructed (lines 2–5). The intuition is that the window captures resource requirements exhibited by workflows that are temporally close to the newly incoming workflow. Later, the learned models $\mathcal{CM}$ and $\mathcal{CC}$ are updated using the workflows pertaining to the constructed sliding window *sw* and the updates are translated to the KB (lines 6–11).

To effectively incorporate new workflow resource requirements and simultaneously phase out defunct requirements we

Algorithm 3 Proactive Algorithm

Input: $\mathcal{V}$, $\mathcal{N}^i_{max} \mid \forall V_i \in \mathcal{V}$, $\mathcal{W}^t_s \sim \mathcal{D}_s(t)$

Output: SLA_{status}, $AvgCost_s$, $AvgPenalty_s$, $AvgSLABreakDuration$

1: $SLABreakDuration \leftarrow 0$, $AvgSLABreakDuration \leftarrow 0$

2: **procedure** ProactiveDeploy($\mathcal{W}^t_s, \mathcal{V}, \tau$)

3: **for each** $W_j \in \mathcal{W}^t_s$ **do**

4: $Cost_{W_j} \leftarrow 0$, $Penalty_{W_j} \leftarrow 0$

5: **for each** $C_{kj} \in W_j$ **do**

6: $\mathcal{N}^i_{running} \leftarrow \mathcal{N}^i_{running} + 1$

7: **if** $\mathcal{N}^i_{running} = \lfloor \tau . \mathcal{N}^i_{max} \rfloor + 1$ **then**

8: Identify VM V_l, with $\mathcal{N}^l_{running} < \mathcal{N}^l_{max}$

9: $StartVM^{V_l} \leftarrow t$

10: **if** $\mathcal{N}^i_{running} > \mathcal{N}^i_{max}$ **then**

11: **if** $t - StartVM^{V_l} < T^{V_l}_{reserve} + T_{migrate}$ **then**

12: $SLA_{status} \leftarrow true$

13: $extraDelay \leftarrow T^{V_l}_{reserve} + T_{migrate} - t + StartVM^{V_l}$

14: $SLABreakDuration \leftarrow SLABreakDuration + extraDelay$

15: $Penalty_{W_j} \leftarrow Penalty_{W_j} + \frac{extraDelay}{T^{V_l}_{reserve} + T_{migrate}} (P^{V_l}_{reserve} + P_{migrate})$

16: Deploy C_{kj} on VM V_l

17: $\mathcal{N}^l_{running} \leftarrow \mathcal{N}^l_{running} + 1$

18: $\mathcal{N}^i_{running} \leftarrow \mathcal{N}^i_{running} - 1$

19: $Cost_{W_j} \leftarrow Cost_{W_j} + M_l + CPU_l + S_l$

20: **else**

21: $Cost_{W_j} \leftarrow Cost_{W_j} + M_i + CPU_i + S_i$

22: $AvgCost_s \leftarrow AvgCost_s + Cost_{W_j}/|W_j|$

23: $AvgPenalty_s \leftarrow AvgPenalty_s + Penalty_{W_j}/|W_j|$

24: $AvgSLABreakDuration \leftarrow SLABreakDuration/|W_j|$

25: $AvgCost_s \leftarrow AvgCost_s/|\mathcal{W}^t_s|$, $AvgPenalty_s \leftarrow AvgPenalty_s/|\mathcal{W}^t_s|$

26: $AvgSLABreakDuration \leftarrow AvgSLABreakDuration/|\mathcal{W}^t_s|$

incorporate the use of information decay. Workflows that are temporally farther from the newly incoming workflows would have relatively less contribution towards learning their resource requirements when compared to the workflows that are temporally closer [20]. To model this effect, we use λ ($<= 1$) as the rate of information decay over time. More specifically, given a sliding window *sw*, the contribution of workflows pertaining to a time-instant t' is scaled using $\alpha = \lambda^{(sw_{end} - t')}$ (line 7). Later α is used to weigh the relative importance of $\mathcal{W}^{t'}$ for updates to $\mathcal{CM}$ and $\mathcal{CC}$ (lines 9–10).

B. Proactive Algorithm

Algorithm 3 describes the pseudo-code for the *proactive* algorithm. In this algorithm, the *SLA monitoring module* continuously monitors the number of service components $\mathcal{N}^i_{running}$ and checks how far this is from the maximum permissible limit $\mathcal{N}^i_{max}$, for each VM $V_i \in \mathcal{V}$ (lines 6–9). The *proactive* algorithm incorporates the use of a parameter τ, which enables triggering of new VM reservations (line 8) for service components running on a VM V_i once $\mathcal{N}^i_{running} = \lfloor \tau \mathcal{N}^i_{max} \rfloor + 1$ (line 7). By using the parameter τ, a VM is proactively started, which when ready accepts the new requests for this session. More specifically, the parameter τ facilitates the reservation of a new VM V_l and the migration of service components from V_i to V_l, while there is still room for more components to be executed on VM V_i without breaking the SLAs.

Note that since we preach maximum resource utilization, although new VM reservations are triggered once the above

Algorithm 4 Hybrid Algorithm

Input: $\mathcal{V}$, $DC_{W_j} \mid \forall W_j \in \mathcal{W}^t$, $\mathcal{W}^t_a \sim \mathcal{D}_a(t)$, $\mathcal{CM}^t$, $\mathcal{CC}^t$

Output: $DEADLINE_{status}$, η, $AvgCost_a$

1: **procedure** KBQuery($\mathcal{W}^t_a, \mathcal{CM}^t, \mathcal{CC}^t, \mathcal{V}$)

2: $AvgCost_a \leftarrow 0$; $Penalty_a \leftarrow 0$; $\eta \leftarrow 0$

3: **for each** $W_j \in \mathcal{W}^t_a$ **do**

4: $MI_{W_j} \leftarrow 0$; $assignedMIPS_{W_j} \leftarrow 0$; $DEADLINE^{W_j}_{status} \leftarrow true$

5: $DEADLINE^{W_j}_{status} \leftarrow \mathcal{CM}^t_{predict}(\{C_{1j}, C_{2j}, \ldots, C_{kj}\} \in W_j)$

6: **if** $DEADLINE^{W_j}_{status} = true$ **then**

7: Deploy W_j on a pre-reserved "medium" VM V_i

8: $AvgCost_a \leftarrow AvgCost_a + (M_i + CPU_i + S_i) \times |W_j|$

9: **else**

10: Assign W_j to the closest cluster center $cc \in \mathcal{CC}^t$

11: **for each** $C_{kj} \in W_j$ **do**

12: $MI_{W_j} \leftarrow MI_{W_j} + MI_{C_{kj}}$

13: Deploy C_{kj} on VM V_l with $MIPS_l \geq MI_{C_{kj}}$

14: Monitor the progress of C_{kj} for every Δt; $t_{cur} \leftarrow t_{cur} + \Delta t$

15: **if** $t_{C_{kj}} - t_{cur} = T^{V_l}_{reserve} + T^{V_l}_{migrate}$ **then**

16: Initiate reservation for VM V_l

17: $assignedMIPS_{W_j} \leftarrow assignedMIPS_{W_j} + MIPS_l$

18: $AvgCost_a \leftarrow AvgCost_a + (M_l + CPU_l + S_l)$

19: **if** $(MI_{W_j}/assignedMIPS_{W_j}) > DC_{W_j}$ **then**

20: $DEADLINE^{W_j}_{status} \leftarrow false$; $\eta \leftarrow \eta + 1$

21: $AvgCost_a \leftarrow (AvgCost_a + Penalty_a)/|\mathcal{W}^t_a|$; $\eta \leftarrow \eta/|\mathcal{W}^t_a|$

22: **return** $DEADLINE_{status}, \eta, AvgCost_a$

condition is met, the service components are migrated only after the VMs currently running them are utilized to their maximum capacity, i.e., once for a VM V_i $\mathcal{N}^i_{running} = \mathcal{N}^i_{max}$. Thus, once V_i is fully utilized (line 10), the workflow components are migrated to the newly reserved VM V_l, and the corresponding costs are updated accordingly (lines 16–19).

Next, we describe the effect of new VM reservations and component migrations on workflow SLAs (lines 11–15). The SLAs of all the components remain violated for the time required to reserve new VMs and the time required to migrate them from one VM to another, *discounting* the time duration corresponding to the start of the reservation process and the time instant at which the SLA actually got violated. Mathematically, $SLABreakDuration^{C_{kj}} = extraDelay = T^{V_l}_{reserve} + T_{migrate} - t + StartVM^{V_l}$ (line 13); $\forall W_j \in \mathcal{W}^t_s, \forall C_{kj} \in W_j$ and $\forall V_l \in \mathcal{V}$. Thus, with a careful selection of τ, $T_{reserve} + T_{migrate}$ would get subsumed by the difference in time at which the SLAs actually got violated and the time at which the reservation process was triggered. This will enable SLAs to be always met while the waiting time on VMs that need to be started will also be 0. Additionally, a penalty proportional to the duration for which the SLAs were violated is added to the costs (line 15), on top of the usual VM utilization costs.

The proactive algorithm prevents SLA violations by closely monitoring the behavior of service components. If the parameter τ is too low, additional VMs will be reserved rapidly which will in turn drive up the cost. Likewise, if τ is too high, new deployments will be queued until a new VM is instantiated.

C. Hybrid Algorithm

Algorithm 4 presents the pseudo-code for the hybrid algorithm. It incorporates the use of the curated information from the KB updated and constructed by the online learning

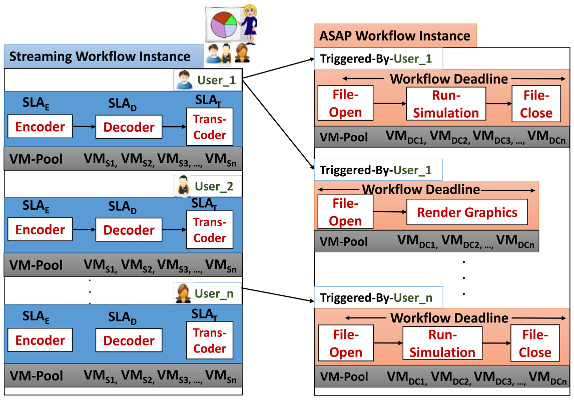


Fig. 6. Streaming workflows spawning ASAP workflows.

algorithm (Algorithm 2) of BRAHMA$^+$. As a first step, the hybrid algorithm invokes the classification model $\mathcal{CM}^t$ stored in the KB, to predict the $DEADLINE_{status}$ of each ASAP workflow (line 5). The workflows with the predicted $DEADLINE_{status} = true$ do not need any specialized scaling and thus, they are assigned to a "medium" (detailed in Section VI) VM (lines 6–8). For workflows whose deadlines are predicted to be violated, we query the identified cluster centers $\mathcal{CC}^t$ stored in the KB and try to identify the cluster, which possesses workflows with the most similar resource requirement patterns (line 10). Next, with this derived information, the workflow components are assigned appropriate resources accordingly (lines 11–22).

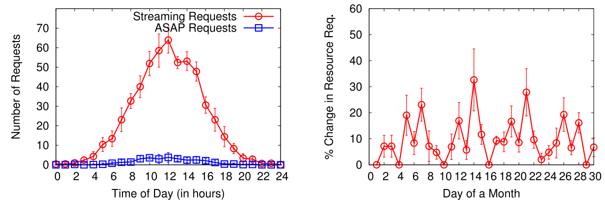
Specifically, each service component $C_{kj} \in W_j$ is assigned to a VM V_l that is large enough to honor the component resource requirements (line 13). A notable limitation of the hybrid algorithm is that, the resources are not pre-reserved, and hence, it is prone to suffer from various penalties incurred owing to new VM reservations and migration of workflow components from one VM to another. To mitigate or minimize these penalties, the *hybrid* algorithm incorporates the use of *monitoring* (similar to the proactive algorithm described in Section V-B) to continuously track the progress of an executing component. More specifically, for every clock tick Δt, a monitor event tracks the execution status of a currently running component C_{kj} (line 14), and as soon as the time left for its execution to complete, crosses the VM reservation and migration time $T^{V_l}_{reserve} + T^{V_l}_{migrate}$ (line 15), a new VM reservation is triggered. This enables timely reservation of new VMs and migration of components, thereby mitigating the incurred penalties completely (lines 14–16).

VI. Experimentation Setup

A. Media Workflows

The media workflows (Fig. 6) used in this study are inspired by the EMD project[2] and the online meeting room use-case

[2]The EMD project is an *imec* funded project aimed at design and development of an elastic platform for media distribution in the context of online collaborative services. The research done in this article is inspired by the diverse workflow types observed in EMD, which in addition to presenting a real-world scenario also possess high affinity to the use-case of online collaborative meeting discussed in this article. Additional details about EMD are available at: https://www.imec-int.com/en/what-we-offer/research-portfolio/emd.



(a) (b)

Fig. 7. Variation in the (a) number of Streaming and ASAP worfklow requests based on the time of day, and (b) change of workflow resource requirements with day of the month.

discussed in Section I. Each user participating in the meeting represents an instance of a *streaming* workflow, and can additionally trigger multiple *ASAP* workflows.

Components corresponding to streaming workflows possess an SLA, which, if not met, may cause unwanted side-effects. Staying with the use case at hand this could cause A/V synchronization issues, stuttering, etc. Each ASAP workflow possesses a deadline-constraint, which can be either met or violated, and if violated causes simulation results to be delayed, or high-quality graphics not to be rendered properly. Additionally, the *resource requirements* exhibited by individual components of the submitted workflows *vary over time*. Note that even though much more elaborate workflows exist, these particular workflows have been chosen to showcase the strength of BRAHMA$^+$ and the presented algorithms in an easy-to-grasp manner. Moreover, BRAHMA$^+$ and its associated algorithms are able to work with generic workflows, and are not constrained by the above assumptions.

B. Evaluation Scenario

As shown in Fig. 7a, 200 user requests for streaming workflows are generated following a normal distribution, with the time 12 noon set as mean and 3.5 hours as standard deviation. At every time instant, each user further possesses a 5% chance of triggering an ASAP workflow. This graph portrays that the number of requests for both streaming and ASAP workflows will vary in between the start of the day up to the end of the day: high load during office hours and negligible load during evening and night time.

Each streaming workflow possesses 3 components, that execute continuously during the course of the meeting. ASAP workflows, on the other hand, can be of varying characteristics: number of components, resource requirements etc. To this end, a request generator module generates a variety of 3, 4, and 5 component workflows (obtained from our industrial partners in the EMD [4] project.), where the resource requirements of each component correspond to the templates as shown in Table II. For example, the file-open→run-simulation→file-close workflow corresponds to the <low→high/very-high→low> template. A map-reduce job could correspond to the <low→high/very-high→low→high/very-high→low> template. The deadline-constraint of each ASAP workflow is estimated using the expected resource requirement of a component. Specifically, the deadline-constraint, represented in terms of MI requirements, of a workflow W_j possessing k components is calculated as k times the expected component resource requirement.

TABLE II
RESOURCE REQUIREMENT TEMPLATES

Template Name	Very-Low	Low	Medium	High	Very-High
Resource requirement range (MI)	100 – 300	250 – 550	500 – 800	750 – 1050	1000 – 1200

TABLE III
PARAMETERIZED VM TEMPLATES

Template	**CPU**	**RAM**	**Storage**	**Hourly Cost**
Template_{01}	150 MIPS	4 GB	128 GB	\$0.154
Template_{02}	300 MIPS	8 GB	256 GB	\$0.308
Template_{03}	450 MIPS	12 GB	384 GB	\$0.462
Template_{04}	600 MIPS	16 GB	512 GB	\$0.616
Template_{05}	750 MIPS	20 GB	640 GB	\$0.77
Template_{06}	900 MIPS	24 GB	768 GB	\$0.924
Template_{07}	1050 MIPS	28 GB	896 GB	\$1.078
Template_{08}	1200 MIPS	32 GB	1024 GB	\$1.232

To better evaluate BRAHMA$^+$ in real-world scenarios, in addition to workflows where resource requirements of components are assumed to be static, we also conduct experiments with *time-varying workflows*: where resource requirements of the constituent service components vary over time. We simulate this temporal change using the hypothesis that the resource requirements of service components may undergo a change on a daily basis, with the change being small (of the order of 5%) during the weekdays and large (of the order of 25%) during the weekends. Fig. 7b portrays this behaviour. The days represented as numbers start from Monday, thus, Day 1 represents Monday, Day 12 represents Friday, and so on. As can be seen, the highest change occurs on Friday, portraying transitioning of resource request patterns from weekdays to weekends, and on Sunday, which presents the reverse effect, i.e., the change from weekends to weekdays.

For each user request a new instance of the workflow W_j is created and the constituent service components $C_{kj}, \forall k \mid C_{kj} \in W_j$ are provisioned on different VMs V_i, available from the VM pool $\mathcal{V}$ (the choice of which VM and how this VM pool grows / shrinks is driven differently depending on the choice of algorithm). To deploy VMs in the resource pools, eight types of VM images were defined as detailed in Table III. The costs for the VM templates used were parametrized based on the Amazon EC2 image *c*3.8*x*large [1], with a monthly price of \$1.680 to provide 32 vCPUs (17476 MIPS [2]), 60 GB of RAM and 2*320 GB of storage. This cost was divided equally between secondary-storage, main-memory and CPU, and the converted unit prices (per MB/hour and MIPS/hour [3]) were used to calculate the costs for the VM templates used in this article. As mentioned in Section V, the time required to reserve new VMs differs significantly from the time required to migrate one component from an existing VM to another. To this end, we define two variables, $T^{V_i}_{reserve}$ and $T_{migrate}$, that determine the duration for instantiating new VMs and the duration for migrating components from an existing VM instance to another respectively. For the simulations, the values of ($T^{V_i}_{reserve}$) and ($T_{migrate}$) were defined as uniform distributions between [40s,55s] and [0.5s,2s] respectively using recommendations provided in [27] and [40]. Additionally, the specific values were extrapolated to correspond to the VM images used in this study. All the parameters mentioned above are not constrained to the stated fixed values, and can be tuned as needed.

C. Evaluation Metrics

- *Efficacy:* We adopt SLA status (Definition 5), average SLA violation duration (Definition 6), and deadline status (Definition 8) [8], [9] to evaluate the quality of the discussed methods.
- *Cost:* We use the VM cost (Definition 9), and penalty (Definition 10) [8], [9] to measure the incurred cost.

D. Methods Benchmarked

We compare the *cost* and *efficacy* of the *Proactive* and *Hybrid* algorithms, proposed under BRAHMA$^+$, against a number of carefully designed baselines and heuristics.

For streaming workflows, we employ the use of *passive* and *reactive* algorithms [9] for comparison. Under the passive algorithm, all resources are reserved in the beginning of the application session, and do not undergo any change even if their capacity is reached. On the other hand, the reactive algorithm allows new resources to be reserved once the pre-reserved resources reach their capacity.

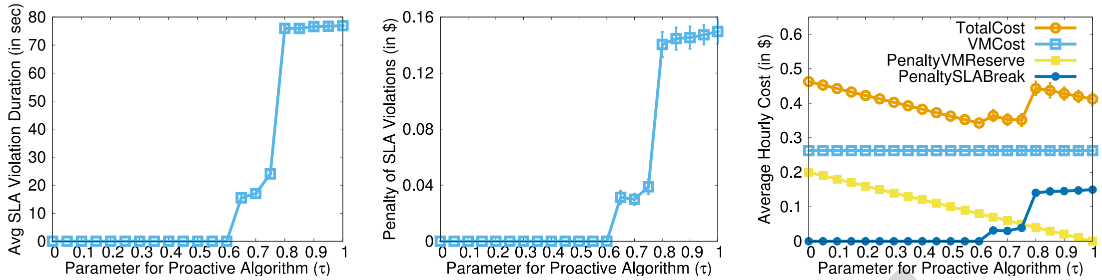
For ASAP workflows, we use the baseline and advanced algorithms [8] as benchmarks. The baseline algorithm is similar to the passive algorithm in design: it reserves all the resources at the beginning of the application session. Every incoming ASAP workflow is assigned to a pre-reserved "medium" (Template04 in Table III) sized VM. An intuitive approach to define the MIPS of a medium-sized VM is using the expected MI requirement of a workflow component. The reason being that in expectation, this VM would be able to meet the deadline constraints of half of the ASAP workflows. The advanced algorithm on the other hand allows new resources to be provisioned when the pre-reserved VM is not sufficient.

Lastly, for time-varying workflows, we use the non time window enabled versions of the proposed algorithms as benchmarks. These algorithms ignore the capability of BRAHMA$^+$ to adapt to the changing resource requirements of workflows. More specifically, after the initial learned models are populated in the KB, they are not updated as the workflow requirements change over time, and the benchmarks work with this non-updated copy of the KB instead.

VII. EVALUATION RESULTS

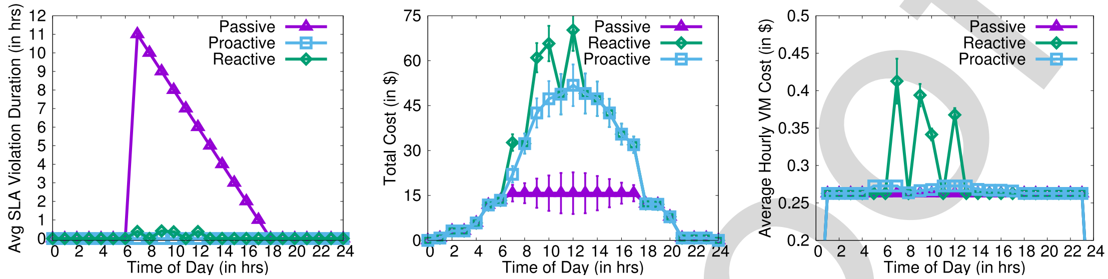
All simulations were performed using the CloudSim simulator [12] and its extensions[3] proposed in this article, on an Intel(R) Core i5 4-core machine with 1.7 GHz CPU and 8 GB RAM running Linux Ubuntu 16.04. We use the publicly available implementations of the classification and clustering models from the WEKA [18] data mining software. Results are averaged over 10 simulation runs. Note that all the parameter values/ranges recommended in the following section(s) are a result of fine-tuning based on the workload and experimental setup employed in this study. The recommended values/ranges are thus, not generic, and subject to change on new workloads.

[3]The code (along with a description of the CloudSim extensions) will be open sourced to the research community via GitHub.



(a) (b) (c)

Fig. 8. Variation in the (a) average SLA violation duration, (b) average penalty, and (c) average cost as a function of τ for the proactive algorithm.



(a) (b) (c)

Fig. 9. A comparison of the (a) average SLA violation duration, (b) total cost, and (c) average cost as a function of the time of day for the passive, reactive and proactive algorithm. The reported costs are parametrized using VM templates stated in Table III.

A. Streaming Workflows

The *proactive* algorithm possesses a parameter τ that controls triggering of new VM reservations. As mentioned in Sections V and VI, once $\mathcal{N}^i_{running} > \tau\mathcal{N}^i_{max}$, a new VM reservation is triggered by the resource provisioning modules. Note that as stated in Section V-B and [9], τ should neither be too high nor too low. The former will lead to large number of SLA violations as workflows would be queued waiting for a new VM to be instantiated, while the latter would lead to high cost, which might be prohibitive. Thus, as a general guideline τ should use a moderate value, viz. $0.25 \leq \tau \leq 0.75$, for optimizing the trade-off. Next, we analyze the impact of τ on the *proactive* algorithm in the context of our experimental setup. Fig. 8a shows that the SLAs of the components are met when $\tau \leq 0.6$, beyond which the average SLA violation duration starts increasing. $0 \leq \tau \leq 0.6$ serves as a good range with respect to minimizing the SLA violation duration.

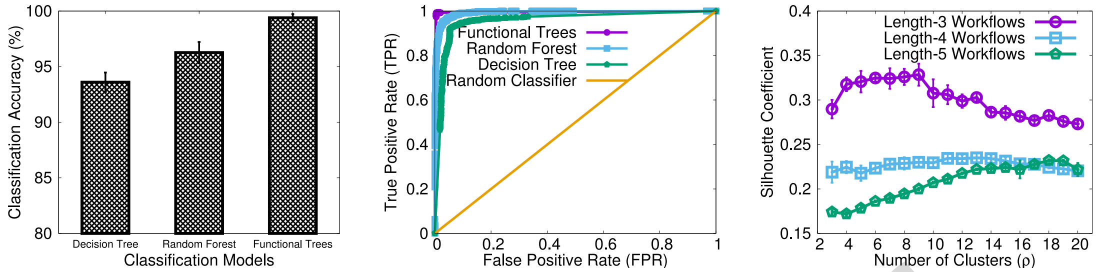
The average penalty incurred during the time when SLAs are violated is shown in Fig. 8b. Since the penalty is incurred due to SLA violations, it is not surprising that the slope of the curve in Fig. 8b is highly similar to that of Fig. 8a. Thus, even with respect to minimizing the average penalty, parameter values in the range $0 \leq \tau \leq 0.6$ are considered to be optimal.

Fig. 8c presents the average cost incurred with varying τ. The average VM cost (Eq. 4) is almost constant with the variation in τ. It is evident from Fig. 8c that the penalty incurred due to proactive reservations of VMs decreases linearly with the increase in parameter τ. More specifically, this penalty assumes its maximum value when $\tau = 0$ and its minimum value when $\tau = 1$. The total cost is the sum of the VM cost and the two penalties discussed above. It is evident that the total cost first linearly decreases till $\tau = 0.6$, becomes almost constant till $\tau = 0.75$ and then starts to increase with increasing τ. Thus, with respect to minimizing the total cost, $0.6 \leq \tau \leq 0.75$ serves as the optimal parameter range.

In sum, the value $\tau = 0.6$ serves as the best possible trade-off for minimizing the costs while also keeping the SLAs of the components in line. Note that the *proactive* algorithm will use $\tau = 0.6$ for all of the following analyses.

Fig. 9a shows a comparison of variation in the SLA violation duration depending on the time of day for the three proposed algorithms. The SLA violation duration under the *proactive* algorithm is always 0, as the SLAs are always met, while for the *reactive* algorithm it is jittery characterized by spikes where SLAs get violated. The SLA violation duration under the *passive* algorithm increases suddenly to its maximum value and then linearly decreases till it becomes 0. The reason for this phenomenon is as follows: the SLA first gets violated at 7 in the morning and remains violated until 6 in the evening. Thus, SLAs for the components arriving at 7 AM remain violated for 11 hours, those arriving at 8 AM remain violated for 10 hours and so on.

Fig. 9c presents a comparison of the variation in the average cost (per component) with the time of day for the three proposed algorithms. It is evident that the average cost of the algorithms are almost similar (except for reactive, which is characterized by spikes at some instances) at majority of the time instances. Note that the costs portrayed in Fig. 9c also include the penalties (as explained in Section V) incurred by the resource provisioning algorithms. Moreover, since no penalties are incurred by the *passive* algorithm, the cost reported equals the VM utilization cost. At certain instances, the average cost of the *reactive* algorithm is the highest, which is the result of the penalties incurred due to the VM reservation process starting only after the SLAs are violated. Since the



(a) Classifier Performance (b) ROC curve analysis (c) Impact of ρ on cluster coherency

Fig. 10. Evaluations for the learning phase of the BRAHMA$^+$ framework.

proactive algorithm triggers the new VM reservation process prior to detecting violation in the SLAs, the penalties incurred for this algorithm are significantly lower when compared to that of the *reactive* algorithm. The only penalty incurred on the *proactive* algorithm is due to the pre-reservation of VMs, which is optimized for $\tau = 0.6$ as discussed above.

B. ASAP Workflows

To simulate previously executed workflows and train the learning phase of BRAHMA$^+$, we use a request generator to generate 6000 ASAP workflow requests of varying (3, 4, and 5) lengths. Using the deadline estimation discussed in Section VI, each ASAP workflow is then assigned a class label, i.e., whether the deadline of this workflow was violated or met. If the total MI requirements of an ASAP workflow is greater than the estimated deadline-constraint (in MI), then its deadline is marked to be violated.

We use the *decision tree* (J48 algorithm) [33], *random forest* [11] and *functional tree* [15] classification methods. A grid-search is performed to choose the optimal set of internal classifier parameters. The confidence factor used for pruning the decision tree is set to 0.25, while the minimum number of instances per leaf node of the tree is set to 2. The random forest classifier is built using 50 trees, and each tree is constructed using 3 random features from the data. Lastly for functional trees, the minimum number of instances in a node for it to be considered for splitting is set to 15, while the number of boosting iterations is set to 15. The reader is referred to [18], for an in-depth understanding of these parameter terminologies and their description.

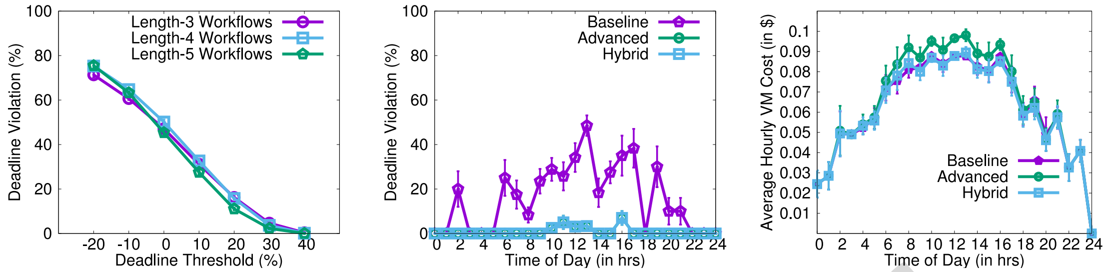
Fig. 10a portrays the classification accuracy using 10-fold cross validation, with functional tree possessing the highest accuracy ($\approx$ 99%) while decision tree possesses the least ($\approx$ 94%). Nevertheless, using any of the three classifiers, BRAHMA$^+$ possesses a reasonably high classification accuracy. Note that our contribution is not limited to the three classifiers used to portray these results, rather is based on the BRAHMA$^+$ framework which suggests the use of classification as a method in general. We also construct the receiver operating characteristic (ROC) curve for the classifiers, that plots true positive rate (TPR) against the false positive rate (FPR). Classifiers whose ROC curves approach the top-left corner of the plot are considered to be good. The line $y = x$, for a binary classification task, represents a random-classifier. Fig. 10b clearly shows that all three evaluated classifiers are significantly better when compared to a random method, and approach the top-left corner of the plot. Moreover, both functional tree and random forest possess a very good area under the ROC curve (AUROC ≈ 0.99).

We employ the use of the *k-means* clustering algorithm [19] to cluster ASAP workflows into groups with similar resource requirement patterns. Since k-means requires the number of clusters to be identified as input, we employ the use of *silhouette coefficient* [36]: a statistical metric for quantifying cluster quality, to correctly identify the optimal number of clusters. Fig. 10c plots the silhouette coefficient values for ASAP workflows of lengths 3, 4 and 5, with varying number of clusters ρ from 3 to 20. The silhouette coefficient gradually increases with the increase in ρ, stabilizes near a peak value, and then starts to decrease. The higher the silhouette value, the better the produced clustering, thus, we choose ρ as 9, 11 and 18 for the length 3, 4 and 5 workflows respectively.

We evaluate (Fig. 11a) the fraction of ASAP workflows whose deadline gets violated with varying deadline thresholds for the baseline algorithm. A large number of workflows, of the order of 50–60%, with the worst-case being up to 78%, suffer deadline violations. Moreover, only after relaxing the deadline threshold by 40%, each ASAP workflow is able to meet its deadline. This result portrays that naïvely assigning ASAP workflows to a "medium-sized" VM is not sufficient, and hence, motivates the need for a framework like BRAHMA$^+$.

Fig. 11b presents a comparison of the baseline, advanced and hybrid algorithms in terms of the percentage of ASAP workflows whose deadline gets violated with the time of day. Since the baseline algorithm does not perform any efforts to perform intelligent resource provisioning, it suffers from a large number (up to 45%) of deadline violations. On the other hand, the advanced and hybrid algorithms leverage the BRAHMA$^+$ framework to perform informed resource provisioning, and do not suffer deadline violations for a majority of the time-instances. Even when the deadlines do get violated, the percentage of violations are as low as 3–5%.

Lastly, we perform a comparison of the variation in average hourly VM costs for the proposed algorithms with the time of day. Note that this analysis includes the costs for both streaming and ASAP workflows as well as the penalties incurred, if any. The *pro-active* algorithm is used with $\tau = 0.6$, since the SLAs are always met and there are no penalties incurred due to SLA violations. Fig. 11c shows that the baseline algorithm possesses the least cost. This is mainly due to pre-assignment



(a) Deadline Violations in Baseline (b) ASAP Workflow Deadline Violation (%) (c) Average Hourly VM Cost

Fig. 11. (a) Analysing ASAP workflow deadline violation percentage under the baseline algorithm with varying deadline thresholds. (b) A comparison of the variation in the ASAP workflow deadline violation percentage and (c) the average total cost (combining costs for streaming and ASAP workflows) versus the time of day for the baseline, advanced and hybrid algorithm.

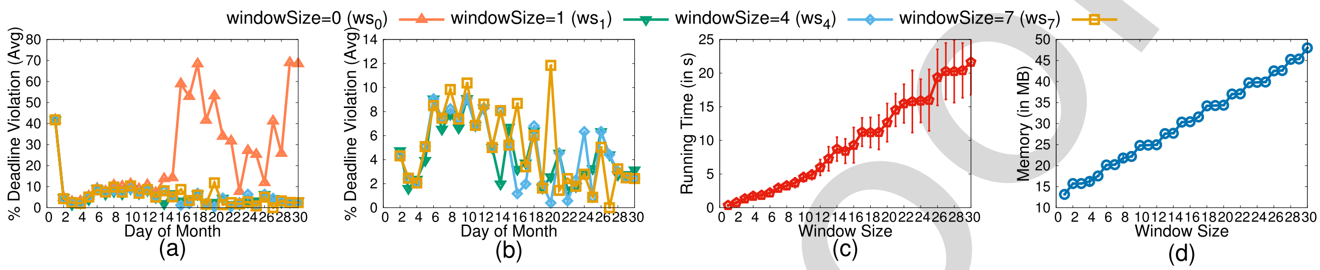


Fig. 12. (a), (b) A comparison of the variation in the average deadline violation percentage versus the day of a month for varying window sizes 0, 1, 4, and 7. Rate of growth of (c) execution time and (d) memory consumption of the clustering algorithm with increase in window size from 1 to 30.

of resources and the lack of new VM reservations even if $\forall V_i \in$ pre-reserved $\mathcal{V}$, the $MIPS_i$ is insufficient to fulfil the requirements of a workflow W_j, which results in deadline violations. On the other hand, the *advanced* algorithm possesses the highest cost, owing to penalties incurred by workflows waiting for new VM reservations and component migrations. The *hybrid* algorithm mitigates these penalties by using a *monitoring* capability similar to that of the *pro-active* algorithm, thereby closely mirroring the cost of the baseline algorithm and being as *cost-effective*.

C. *Time-Varying Workflows*

For time-varying workflows, we generate 30 different batches of workflow requests, which are sampled daily for a period of one month. We generate 6000 ASAP workflow requests of varying lengths for each day in a month. As discussed previously, the resource requirements (in MI) of the workflow components change over time (Fig. 7b).

Conventional clustering methods that assume the underlying resource request patterns to be static, are not capable of capturing the behaviour exhibited by time-varying workflows since their distribution of resource requirements vary over time. As discussed in Section V, to enable BRAHMA$^+$ perform resource provisioning for time-varying workflows (AWS-tSA), we employ the use of sliding windows $sw[sw_{start}, sw_{end}]$ that provide an effective mechanism for updating the learned mathematical models.

First, we analyse the effect of using a sliding window on the average deadline violation percentage. Fig. 12a presents the variation in average deadline violations for different window sizes. All the evaluations use the *hybrid* resource provisioning algorithm of BRAHMA$^+$, with the only change being the way in which the classification and clustering modules of BRAHMA$^+$ learn the workflow behaviour. We measure the violations in workflow deadlines by varying the sliding window size, where ws_0, ws_1, ws_4, and ws_7 represent approaches with window sizes 0, 1, 4, and 7 respectively. The window size of 1 means that for requests generated on a day t, we will consider the day $t-1$ for performing the cluster identification step; a window size of 4 means that we use the days $t-1$, $t-2$, $t-3$ and $t-4$. The procedure for any other non-zero window size follows similarly. On the other hand, a window size of 0 represents the absence of sliding windows, i.e., the conventional clustering method [8] used for workflows with static resource requirements (Section VII-B). Since the deadline violations observed under ws_0 can be as large as 70% (Fig. 12a), to better visually portray and analyse the variation of deadline violations under ws_1, ws_4, and ws_7, we plot Fig. 12b, which is a zoomed-in version of Fig. 12a, by ignoring the deadline violations observed under ws_0.

It is evident from Fig. 12a that the performance of ws_0 degrades over time. This is because the resource requirements of workflows change over time (Fig. 7b), however, ws_0 does not perform any effort to adapt to the changing resource requirements. Owing to the absence of sliding windows, the models are not updated and hence, the identified clusters perform worse over time, and no longer remain a representative of the resource requirements exhibited by the currently submitted workflows. Consequently, there is an increase in deadline violations. It is interesting to note that the same hybrid algorithm (with ws_0) is near-optimal (Fig. 11b shows that most of the deadlines were met) when the resource requirements of workflows were static.

Figs. 12a and 12b show that the performance of the approaches in terms of preserving the deadline constraints of time-varying workflows follows the following order: $ws_4 \approx ws_1 > ws_7 >> ws_0$. A careful analysis of Fig. 12b shows that in the majority of the cases, ws_7 is worse than ws_1 and ws_4. A probable explanation of this is that ws_7 captures too much historical information, i.e., it learns a lot of noise as well owing to the large window size. Moreover, on average ws_4 provides smaller deadline violations when compared to ws_1. Thus, in terms of avoiding deadline violations, ws_4 provides a good choice for the window size in our scenarios.

Having analysed the effect of window size on deadline violations, we also study its effect from a computational standpoint. Figs. 12c-d present the impact of variation in window size on execution time and memory consumption of the clustering module respectively. It is evident that both execution time and memory consumption increase with increase in the window size. However, as can be seen from Fig. 12c, the rate of growth of execution time is super-linear since the worst case complexity of *k*-means is super-linear. On the other hand, the rate of growth in memory-consumption is close to linear (Fig. 12d).

We recommend 4 as the choice for window size as it minimizes deadline violations, while being only marginally more expensive than ws_1 on computational fronts. To summarize, for the presented use-cases, BRAHMA$^+$ with its associated *proactive* algorithm using $\tau = 0.6$, the *hybrid* algorithm, and the sliding window approach with window size 4 serves as the best possible trade-off for minimizing the costs while also keeping the SLAs and the deadline-constraints of the workflows in line for both static (AWS-SA) and time-varying (AWS-tSA) workflows.

VIII. Conclusion and Future Work

In this article, we addressed the problem of *Automatic Workflow resource Scaling* under the combined presence of *Streaming* and *ASAP* workflows, called *AWS-SA*, and its time-varying variant called *AWS-tSA*. Consequently, we devised a holistic solution for both the problems; by coming up with a framework *BRAHMA*$^+$ that curates a KB of learned workflow behavior(s), the *proactive* algorithm for streaming workflows, and the *hybrid* KB driven resource provisioning algorithm that leverage BRAHMA$^+$ for effective scaling of ASAP workflows. We also portrayed the capability of BRAHMA$^+$ to *adaptively learn* the workflow behavior of time-varying workflows, thereby facilitating *online updates* to the KB and effective resource provisioning where resource requirements change over time. Our empirical studies show that the proposed algorithms are *effective* and provide good cost-efficacy trade-offs. The proposed *hybrid* algorithm – combining learning and monitoring, is able to restrict deadline violations to a very small fraction (3–5%), while only suffering a marginal increase in average cost per service component of 1–2% over the baseline algorithm, which, although possesses the least cost, suffers from a large number (up to 45%) of deadline violations. Additionally, for time-varying ASAP workflows, the online clustering approach with a window size of 4 is able to restrict average deadline violations (per day) to 5–8% in comparison to that of (up to) 60% when the identified clusters were not updated over time. In the future, we will implement a BRAHMA$^+$ prototype running on real-world cloud platforms and evaluate its runtime behavior while scaling an elastic A/V collaborative cloud-based service.

References

[1] *Amazon EC2 Images*. Accessed: May 16, 2017. [Online]. Available: http://aws.amazon.com/ec2/instance-types/

[2] *Amazon EC2 MIPS*. Accessed: May 16, 2017. [Online]. Available: http://www.cmips.net/category/cup-results/

[3] *Benchmarking the New Amazon C4 Instances*. Accessed: May 16, 2017. [Online]. Available: http://www.cmips.net/tag/intel-xeon-e5-2666-v3-2-90ghz/

[4] *The EMD Project*. Accessed: May 16, 2017. [Online]. Available: https://www.imec-int.com/en/what-we-offer/research-portfolio/emd

[5] *Tupperware: Containerized Deployment at Facebook*. Accessed: May 16, 2017. [Online]. Available: http://bit.ly/2sD46ng

[6] S. Abrishami and M. Naghibzadeh, "Deadline-constrained workflow scheduling in software as a service cloud," *Scientia Iranica*, vol. 19, no. 3, pp. 58–169, 2012.

[7] A. F. Antonescu and T. Braun, "SLA-driven simulation of multi-tenant scalable cloud-distributed enterprise information system," in *Proc. ARMS-CC@PODC*, 2014, pp. 91–102.

[8] A. Atrey, H. Moens, G. V. Seghbroeck, B. Volckaert, and F. D. Turck, "BRAHMA: An intelligent framework for automated scaling of streaming and deadline-critical workflows," in *Proc. CNSM*, Montreal, QC, Canada, 2016, pp. 216–222.

[9] A. Atrey, H. Moens, G. V. Seghbroeck, B. Volckaert, and F. D. Turck, "Design and evaluation of automatic workflow scaling algorithms for multi-tenant SaaS," in *Proc. CLOSER*, Rome, Italy, 2016, pp. 221–229.

[10] E. Boutin *et al.*, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *Proc. OSDI*, Broomfield, CO, USA, 2014, pp. 285–300.

[11] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[12] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exp.*, vol. 41, no. 1, pp. 23–50, 2011.

[13] J. Espadasa *et al.*, "A tenant-based resource allocation model for scaling software-as-a-service applications over cloud computing infrastructures," *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 273–286, 2013.

[14] G. Fan, H. Yu, and L. Chen, "A formal aspect-oriented method for modeling and analyzing adaptive resource scheduling in cloud computing," *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 2, pp. 281–294, Jun. 2016.

[15] J. Gama, "Functional trees," *Mach. Learn.*, vol. 55, no. 3, pp. 219–250, 2004.

[16] T. A. L. Genez, L. F. Bittencourt, and E. R. M. Madeira, "Workflow scheduling for SaaS / PaaS cloud providers considering two SLA levels," in *Proc. NOMS*, 2012, pp. 906–912.

[17] M. H. Ghahramani, M. Zhou, and C. T. Hon, "Toward cloud computing QoS architecture: Analysis of cloud systems and cloud services," *IEEE/CAA J. Automatica Sinica*, vol. 4, no. 1, pp. 6–18, Jan. 2017.

[18] M. Hall *et al.*, "The WEKA data mining software: An update," *SIGKDD Explor. Newslett.*, vol. 11, no. 1, pp. 10–18, 2009.

[19] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-means clustering algorithm," *Appl. Stat.*, vol. 28, no. 1, pp. 100–108, 1979.

[20] R. Hyndman and G. Athanasopoulos, *Forecasting: Principles and Practice*. Melbourne, VIC, Australia: OTexts, 2013. [Online]. Available: https://www.otexts.org/fpp/

[21] M. Isard, "Autopilot: Automatic data center management," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 60–67, 2007.

[22] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *J. Netw. Syst. Manag.*, vol. 23, no. 3, pp. 567–619, 2015.

[23] H.-J. Jiang, K.-C. Huang, H.-Y. Chang, D.-S. Gu, and P.-J. Shih, "Scheduling concurrent workflows in HPC cloud through exploiting schedule gaps," in *Proc. ICA3PP*, Melbourne, VIC, Australia, 2011, pp. 282–293.

[24] J. Li, T. Ma, M. Tang, W. Shen, and Y. Jin, "Improved FIFO scheduling algorithm based on fuzzy clustering in cloud computing," *Information*, vol. 8, no. 1, p. 25, 2017.

[25] H. Lu, J. Cao, S. Lv, X. Wang, and J. Liu, "A comparative study of DAG clustering," in *Proc. i-Soc.*, London, U.K., 2015, pp. 84–89.

[26] H. Luo, C. Yan, and Z. Hu, "An enhanced workflow scheduling strategy for deadline guarantee on hybrid grid/cloud infrastructure," *J. Appl. Sci. Eng.*, vol. 18, no. 1, pp. 67–78, 2015.

[27] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *Proc. IEEE CLOUD*, Honolulu, HI, USA, 2012, pp. 423–430.

[28] E. E. Mon, M. M. Thein, and M. T. Aung, "Clustering based on task dependency for data-intensive workflow scheduling optimization," in *Proc. MTAGS*, Salt Lake City, UT, USA, 2016, pp. 20–25.

[29] H. Morshedlou and M. R. Meybodi, "Decreasing impact of SLA violations: A proactive resource allocation approach for cloud computing environments," *IEEE Trans. Cloud Comput.*, vol. 2, no. 2, pp. 156–167, Apr./Jun. 2014.

[30] G. Peng, H. Wang, J. Dong, and H. Zhang, "Knowledge-based resource allocation for collaborative simulation development in a multi-tenant cloud computing environment," *IEEE Trans. Services Comput.*, vol. 11, no. 2, pp. 306–317, Mar./Apr. 2018.

[31] D. Poola, S. K. Garg, R. Buyya, Y. Yang, and K. Ramamohanarao, "Robust scheduling of scientific workflows with deadline and budget constraints in clouds," in *Proc. IEEE-AINA*, Victoria, BC, Canada, 2014, pp. 858–865.

[32] D. Poola, K. Ramamohanarao, and R. Buyya, "Enhancing reliability of workflow execution using task replication and spot instances," *Trans. Auton. Adapt. Syst.*, vol. 10, no. 4, pp. 1–21, 2016.

[33] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Burlington, MA, USA: Morgan Kaufmann, 1993.

[34] M. A. Rodriguez and R. Buyya, "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds," *IEEE Trans. Cloud Comput.*, vol. 2, no. 4, pp. 222–235, Apr./Jun. 2014.

[35] M. A. Rodriguez and R. Buyya, "Scheduling dynamic workloads in multi-tenant scientific workflow as a service platforms," *Future Gener. Comput. Syst.*, vol. 79, pp. 739–750, Feb. 2017.

[36] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, vol. 20, pp. 53–65, Nov. 1987.

[37] D. Serrano *et al.*, "SLA guarantees for cloud services," *Future Gener. Comput. Syst.*, vol. 54, pp. 233–246, Jan. 2016.

[38] J. Shi, J. Luo, F. Dong, J. Zhang, and J. Zhang, "Elastic resource provisioning for scientific workflow scheduling in cloud under budget and deadline constraints," *Cluster Comput.*, vol. 19, no. 1, pp. 167–182, 2016.

[39] S. Singh, I. Chana, and R. Buyya, "STAR: SLA-aware autonomic management of cloud resources," *IEEE Trans. Cloud Comput.*, to be published.

[40] S. Toyoshima, S. Yamaguchi, and M. Oguchi, "Storage access optimization with virtual machine migration and basic performance analysis of Amazon EC2," in *Proc. WAINA*, Perth, WA, Australia, 2010, pp. 905–910.

[41] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou, "Adaptive resource management and control in software defined networks," *IEEE Trans. Netw. Service Manag.*, vol. 12, no. 1, pp. 18–33, Mar. 2015.

[42] A. Verma *et al.*, "Large-scale cluster management at Google with Borg," in *Proc. EuroSys*, Bordeaux, France, 2015, p. 18.

[43] F. Wu, Q. Wu, and Y. Tan, "Workflow scheduling in cloud: A survey," *J. Supercomput.*, vol. 71, no. 9, pp. 3373–3418, 2015.

[44] L. Wu, S. K. Garg, S. Versteeg, and R. Buyya, "SLA-based resource provisioning for hosted software-as-a-service applications in cloud computing environments," *IEEE Trans. Services Comput.*, vol. 7, no. 3, pp. 465–485, Jul./Sep. 2014.

[45] L. Wu, S. K. Garg, and R. Buyya, "SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments," in *Proc. CCGrid*, Newport Beach, CA, USA, 2011, pp. 195–204.

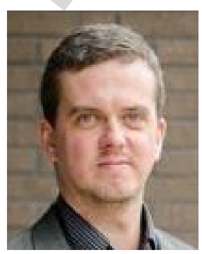
[46] L. Wu, S. K. Garg, and R. Buyya, "SLA-based admission control for a software-as-a-service provider in cloud computing environments," *J. Comput. Syst. Sci.*, vol. 78, no. 5, pp. 1280–1299, 2012.

[47] R. Zhang, K. Wu, and J. Wang, "Online resource scheduling under concave pricing for cloud computing," in *Proc. IWQoS*, Hong Kong, 2014, pp. 51–60.

[48] Z. Zhang *et al.*, "Fuxi: A fault-tolerant resource management and job scheduling system at Internet scale," *Proc. VLDB Endowment*, vol. 7, no. 13, pp. 1393–1404, 2014.

Ankita Atrey received the master's degree in computer science from the Vellore Institute of Technology, Vellore, India. She is currently pursuing the Ph.D. degree with the Department of Information Technology (INTEC), Ghent University, Belgium, and imec. She has internship experience from CNRS, France, and the Indian Institute of Technology Kanpur, India. Her research interests include cloud computing, resource scheduling and provisioning, data-placement, service management, and service oriented architectures. She is working on research problems encircling intelligent resource provisioning in multi-tenant multi-component applications with INTEC. She has published her research in cloud and service management conferences like CNSM and CLOSER, while also serving as a reviewer for CLOSER, *Journal of Network and Systems Management* and the IEEE TRANSACTION ON NETWORK AND SERVICE MANAGEMENT.

Gregory Van Seghbroeck received the graduation degree from Ghent University in 2005 and the Ph.D. degree in computer science engineering in 2011. After a brief time as an IT Consultant, he joined the Department of Information Technology (INTEC), Ghent University (currently IDLab). In 2007, he received the Ph.D. grant from IWT, Institute for the Support of Innovation Through Science and Technology, to work on theoretical aspects of advanced validation mechanism for distributed interaction protocols and service choreographies. His main research interests focus on big data engineering and complex scalable cloud platforms.



Bruno Volckaert received the Master of Computer Science degree and the Ph.D. degree in data intensive scheduling and service management for grid computing from Ghent University, in 2001 and 2006, respectively. He is a Professor of advanced programming and software engineering with the Department of Information Technology (INTEC), Ghent University and a Senior Researcher with imec. His current research deals with reliable and high performance distributed software systems for City-of-Things (IoT for Smart Cities), distributed decision support systems for UAVs, intelligent railway transportation applications and autonomous optimization of cloud-based applications. He has worked on over 35 national and international research projects and has authored or co-authored over 80 papers published in international journals and conference proceedings.

Filip De Turck leads the Network and Service Management Research Group, Department of Information Technology, Ghent University, Belgium, and imec. He has (co-)authored over 450 peer reviewed papers. His research interests include telecommunication network and service management, and design of efficient virtualized network and cloud systems. He is involved in several research projects with industry and academia in the above areas. He serves as the Chair of the IEEE Technical Committee on Network Operations and Management, and is on the TPC of many network and service management conferences and workshops. He serves as a Steering Committee Member of the IEEE Conference on Network Softwarization.