Online Scheduling Technique To Handle Data Velocity Changes in Stream Workflows

Mutaz Barika, Saurabh Garg, Albert Y. Zomaya, Fellow, IEEE, and Rajiv Ranjan, Senior Member, IEEE

Abstract—Many IoT applications and services such as smart parking and smart traffic control contain a network of different analytical components, which are composed in the form of a workflow to make better decisions. These workflows are also known as stream workflows. The focus of existing research works is on the streaming operator graph, which differs from stream workflow application as it involves heterogeneity, multiple data sources and multiple outputs. Considering the complexity and dynamism of stream workflow, meeting real-time data analysis requirements at deployment time is not the whole story as the velocity of data changes over time. This change is the most dynamic form of stream workflow that occurs frequently during the execution of this application. In this paper, we propose a new dynamic scheduling technique that manages cloud resources over time to handle data velocity changes in stream workflow while maintaining user-defined real-time data analysis requirements and minimising execution cost. The efficiency of the proposed technique is evaluated, and experimental results showed that this technique outperformed its competitors and is close to the lower bound.

Index Terms—IoT, Stream workflow, Dynamic scheduling, Alpha-Beta pruning, GA with Random Immigrants, Cloud environments.

1 Introduction

Several IoT applications and services such as smart city, smart parking and smart traffic control, have evolved to cope with the demand of improving our lives [1] [2]. These applications are not a monolithic application, but they contain a network of different analytical components which are composed in the form of a workflow to make better decisions. For instance, smart road traffic monitoring application as a service of smart city services utilises the true power of connected vehicles in addition to roadside infrastructure (e.g. traffic lights, cameras) to a create real-time view of road traffic conditions [3]. This type of workflow is also called stream workflow application and is becoming gradually viable for solving real-time data computation problems that are more complex.

In contrast to traditional business and scientific workflows [**?**] [5], stream workflows support the continuous processing of an infinite stream of data with each analytical component always in an active state. They have multiple data sources that inject their data streams into any analytical components and have multiple outputs. They also differ from streaming operator graphs as the source of data for the whole operator graph is one and there is a one end operator. Thus, the operator graph is just a simplified case of stream workflow. Moreover, this workflow can be highly dynamic in nature, where the data velocity may change at runtime reflecting the load at a given time, and therefore the resources should be managed over time. Furthermore, the use of a single cloud to execute this workflow could not meet user requirements due to the distribution of external data sources. Thus, multicloud environment that consolidates multiple clouds can help in utilising data locality by orchestrating analytical components included in data pipeline over different clouds. However, provisioning resources from different clouds while meeting user performance requirements is also a challenge. By combining the aforementioned challenges with the heterogeneity of compute resources available in cloud datacenters and users' quality of service requirements, managing the execution of such applications is a complex task.

In general, stream workflows have received less attention. But, the importance of making real-time decisions by analysing streaming data is rising with IoT emergence. Most of the existing research works focused on supporting the other type of big data processing which is batch processing. These works such as [6] [7] provided the ability to compose batch processing applications into pipelines to process static data at once and get final analytical insights by extending the capability of scientific workflow management systems. While the rest offered big data orchestrators (Apache YARN [8] and Apache Mesos [9]) that do not need to deal with the dynamism of stream workflow applications and meet real-time user requirements. Therefore, there are few scheduling algorithms in the literature that treat the scheduling problem of various streaming big data applications over cloud infrastructure [10] [11] [12] [13] [14] or over geo-distributed datacenters [15] [16] [17].

In respect to our previous work [18], this work investigated the problem of scheduling stream workflow at deployment time. Genetic and greedy algorithms are proposed to generate a sub-optimal provisioning and scheduling solution for executing stream workflow in a multicloud environ-

- *M. Barika and S. Garg are with Discipline of ICT — School of Technology, Environments and Design, University of Tasmania, Hobart, Tasmania, Australia.*
  E-mail: {mutaz.barika, saurabh.garg}@utas.edu.au
- *A. Y. Zomaya is with the School of IT, University of Sydney, New South Wales, Australia.*
  E-mail: albert.zomaya@sydney.edu.au
- *R. Ranjan is with the School of Computing, Newcastle University, Newcastle Upon Tyne, United Kingdom.*
  E-mail: raj.ranjan@ncl.ac.uk

Manuscript received January xx, 2020; revised xx xx, 2020.

ment. Such work does not deal with the dynamism of stream workflows. Considering the velocity of data may fluctuate over time, this paper investigates the problem of scheduling stream workflows to support runtime data fluctuations. It deals with a stream workflow application based on the fact that this workflow is an adaptive workflow application. Because this workflow serves the current-extra and future demands of changing real-time analytical requirements at runtime to make faster and better decisions.

To fill the gap of supporting dynamic scheduling under the variations of data stream rates, we design a new adaptive scheduling technique. This technique revises the scheduling plan of a stream workflow application to handle the changes that happen in the speed of data at runtime to always meet real-time analytical requirements with minimal execution cost. In other words, it is aimed at tackling data stream velocity fluctuations while maximising performance efficiency, and all of that at a minimal monetary cost. In summary, our contributions are:

- Dynamic stream workflow application model.
- Two-phase adaptive scheduling technique that incorporates two advanced optimisation algorithms (Random Immigrants Genetic Algorithm (GA for short) and two-level greedy algorithm) to handle runtime data fluctuations while meeting real time user performance constraints and minimising execution cost.

This paper is structured as follows: Section 2 reviews the related works. Section 3 presents dynamic stream workflow requirements. The problem formulation is presented in Section 4. Section 5 presents the proposed scheduling technique whose performance is evaluated in Section 6. Section 7 concludes the paper and highlights future improvements.

2 Related Work

In this section, we present the comparisons with related works from three perspectives, which are application, modelling and methods/techniques.

From the application perspective, there are a batch-oriented big data workflow (MapReduce workflow) and a stream-oriented big data workflow (stream workflow). The focus of previous studies (such as J. Wang et al. [6] [7], F. Teng [19], Y. Wang and W. Shi [20], T. Shu and C.Q. Wu [21], and X. Zeng et al. [22] [23]) were mostly on MapReduce workflows and their executions in cloud infrastructure.

From the modelling perspective, there are two stream processing models, which are a data-flow graph with micro-batch processing model (i.e. discretised streaming model) and an operator graph with continuous processing model. With discretised streaming model, streaming computations are performed on a series of small data batches called micro-batches. M. Zaharia et al. [24] followed this model and proposed a stream programming model named Discretized Streams (D-Streams). It brings together a series of Resilient Distributed Datasets (RDDs) and allows performing computations through various transformations. Apache Spark uses RDD data model and allows to perform stream computations on RDD to define data processing. While with a continuous processing model, an operator graph is used to model a data pipeline, where each node in the graph is a long-lived operator. This operator carries-out stream computation on streams as they arrive and produces a new stream. Stream-oriented big data platforms and services such as Apache Storm and IBM Streaming allow building streaming operator graphs for performing real-time data processing. As streaming operator graphs are different from dynamic stream workflows in that the source of data for the whole operator graph is one and there is one end operator, a new model is needed for dynamic stream workflow. This model should involve heterogeneity, multiple data sources and outputs.

From the scheduling perspective, scheduling techniques in the literature use heuristic and/or meta-heuristic approaches for making decisions based on different scheduling criteria (such as deadline, execution cost and performance) in order to meet user-defined SLA requirements. Research works such as D. Sun [10], T. Buddhika et al. [11] and A. Boek and F Werner [12] focused on scheduling data stream computations for performance and/or energy optimisations. However, those research works and frameworks model stream workflow as a streaming operator graph. Since streaming operator graphs are different from dynamic stream workflows, the scheduling problem of dynamic stream workflows has different assumptions and optimisation goals. This problem considers the mapping of analytical components to multiple compute resources and optimisation goals include minimising execution cost and improving performance without violating real-time user requirements.

In the same perspective, D. Sun and R. Huang [13] and D. Sun et al. [14] focused on online scheduling with guaranteed makespan and utilised single cloud as an execution environment for big data streaming applications. These scheduling strategies/methods do not consider stream workflow as a network of streaming big data workflow applications (i.e. workflow of workflows). They also do not take into consideration the dynamic nature of this workflow and its unpredictable performance. They also do not consider the various real-time decision support requirements and the powerful capability of 'cloud of clouds' as a dynamic execution environment. In the same context but for scheduling big data processing jobs/tasks and workflow in geo-distributed clouds, L. Chen et al. [15] proposed a fair job scheduler. This scheduler aimed at reducing job completion time that relied on Apache Spark. Z. Hu et al. [16] proposed a new job scheduling method named Flutter, which aimed at reducing completion time and implemented in Apache Spark. H. Chen et al. [17] proposed task-duplication based real-time scheduling method to reduce completion and execution times. However, these scheduling methods are considered a stream workflow as an operator graph and have different optimisation goals.

For scheduling techniques supported with big data application orchestrators, each one of them uses a different scheduling technique to map applications on cloud resources. Apache YARN uses a monolithic scheduler to map compute resources among competing applications in the cluster. Apache Mesos uses a dual-level scheduling mechanism called resource efforts, which provides resource offerings to a framework. This framework will either accept the offer or reject it if the offered resources do not meet its constraints and then wait for the ones they do. Con-

sequently, these orchestrators assume either that they do not need to meet real-time decision support requirements or are intended for big data workflows that have predictable performance [25]. Therefore, the scheduling mechanisms in those orchestrators consider big data workflow application as a static structure, so that they neglect the following: (1) dynamic nature of this application and its analytical components, (2) unpredictable performance of this workflow application, (3) real-time performance requirements defined by the owners of these workflows, (4) runtime changes and (5) the powerful capability of 'cloud of clouds' as a dynamic execution environment.

Accordingly, the scheduling techniques proposed in the aforementioned studies do not fit the composition needs of complex big data workflows. They also do not leverage the capability of multicloud environment to cope with the dynamic aspects of these workflows. As a result, the dynamic scheduling technique is needed for a stable and efficient execution of stream workflow over multiple cloud infrastructures. Such a technique should meet user real-time performance requirements and respond to the fluctuations of data at runtime while reducing the overall execution cost.

3 Stream Workflow and its Requirement

Stream workflow application is a network of streaming big data applications (analytical components) that can be independently executed over cloud resources while maintaining data dependencies among them. It has three main characteristics that need to be considered. These characteristics are continuous input data (from external and internal sources), continuous processing and continuous insights produced by end analytical components. Considering these characteristics, the most dynamic form of stream workflow that occurs frequently is the change in the speed of streaming data. Example of stream workflow is a smart road traffic monitoring that utilizes IoT connected vehicles and roadside infrastructure to create real-time view of road traffic and incidents. The details of this real use case for stream workflow application is provided in Appendix A.

In stream workflow, the throughputs of services under the dynamic variations of input data rates should be maintained all the time. Also, end-to-end latency (response time) is needed to be maintained or be bounded when it starts to increase. Furthermore in this workflow, there are distributed data sources that inject their data streams into a data pipeline, thus data locality approach should be utilised by leveraging multicloud environment. With this environment, we can avoid transferring large data to the corresponding resources over long-heal networks, that is not only incurring high latency and execution cost but also makes achieving real-time data analysis requirements more difficult. Additionally, each cloud in a multicloud environment offers different computing capabilities with different prices, so that changing the placement cloud for a service is possible to reduce execution cost.

Consequently, the variables of stream workflow application as pointed-out in our previous work [18] are: type of service, its data processing requirement, its data processing rate, data mode, the dynamic variations of input data rates and the dynamism of execution environment (i.e. multicloud environment). Overall, the requirements of both workflow application and real-time data analysis should be maintained while dealing with runtime data fluctuations and minimising the monetary cost.

After presenting a stream workflow and its requirements, we will discuss how this workflow application will be scheduled in a multicloud environment using a motivation example presented in Appendix A. First, we will walk through a static schedule plan and then revise this plan to cope with data velocity changes. For simplicity, let us consider a sample multicloud environment that is formed of two datacenters with different VM offers as listed in Table 1. Also, let us consider the services' configurations of a motivation workflow as listed in Table 2. Based on the configurations of workflow services and VM offers available, the following are static and dynamic reosurce assignments:

- Static assignment: As we discussed in Appendix A, S1, S3 and S4 are unmovable services, so we assume that datacenter placement constrains for these services are 1, 0 and 1 respectively. While the rest of services are movable services, they can be placed on any datacenter to minimise total execution cost. so, we assume that S2, S5, S7 and S9 are placed in Datacenter 0, while S6 and S8 are placed in Datacenter 1. Considering the configurations of workflow services and given constraints, the following is the static assignment of motivation workflow that minimises the total execution cost: S1(4 Small VMs from Datacenter 1), S2 (1 Small VM from Datacenter 0), S3(1 Large VM from Datacenter 0), S4(4 Small VMs from Datacenter 1), S5(1 Small VM and 1 Large VM from Datacenter 0), S6(1 Large VM and 1 XLarge VM from Datacenter 1), S7(1 Large VM from Datacenter 0), S8(1 Small VM and 1 Medium VM from Datacenter 1) and S9(1 Large VM from Datacenter 0).
- Dynamic assignment: Let us assume that a data velocity increase request is occurred at runtime due to the speed of one of the external sources connected to S4 is increased by 4MB/s. This change means more computing power may be required to process the increased input data rate in S4 and its downstream services (i.e. S6-S0). Thus, those affected services will be updated as listed in Table 3. Considering the updated information of those affected services and current deployment plan (i.e. static scheduling plan), the following are amendments on the current scheduling plan for dynamic assignment: For S4 and S6, extra 2 Small VMs from Datacenter 1 will be provisioned for each service to provide an additional 8000 MIPS; For S7 and S9, there is no need to provision any extra VM because the overprovisioning on the current plan (i.e. 2000 MIPS) can cover the need of additional 1000 MIPS for data increase request; For S8, extra 1 Small VMs from Datacenter 1 will be provisioned to provide 2000 MIPS with an additional 2000 MIPS.

From the above walk-through motivation example, the static assignment is only reasonable when the workflow application is static in its nature. As a stream workflow is a dynamic workflow, the dynamic assignment is needed to cope with

its dynamic aspects. Thus, managing resources and revising scheduling plan at runtime when the velocity of data stream changes, requires careful scheduling decisions to keep execution cost and computational time as minimal as possible and having no data processing disruptions. This will ensure a stable and efficient execution of stream workflow over multiple cloud infrastructures. Accordingly, it is important and worthwhile to investigate the problem of dynamically scheduling stream workflow application on various cloud infrastructures to tackle the fluctuations of input data rate at runtime while reducing the overall execution cost.

Table 1
VM offers in a sample multicloud environment

VM type	Datacenter 0	Datacenter 1
Small	MIPS:2000 Cost:0.05 ¢/s	MIPS:4000 Cost:0.06 ¢/s
Medium	MIPS:4000 Cost:0.1 ¢/s	MIPS:8000 Cost:0.15 ¢/s
Large	MIPS:8000 Cost:0.15 ¢/s	MIPS:16000 Cost:0.3 ¢/s
XLarge	MIPS:16000 Cost:0.25 ¢/s	MIPS:32000 Cost:0.4 ¢/s

Table 2
Service configurations of motivation workflow

Service	data processing requirement	Input	Output	Required MIPS
S1	1000	16	10	16000
S2	1000	2	1	2000
S3	1000	8	4	8000
S4	2000	8[1]	4	16000
S5	2000	5	8	10000
S6	4000	12	6	48000
S7	1000	6	6	6000
S8	1000	12	1	12000
S9	1000	6	2	6000

Table 3
Updated input and output of affected workflow services with MIPS

Service	Proportion of output to input	Updated input	Updated output	More MIPS required
S4	4/8=0.5	12	12*0.5=6	8000
S6	6/12=0.5	14	14*0.5=7	8000
S7	6/6=1	7	7*1=7	1000
S8	1/12=0.83	14	14*0.83=1.2	2000
S9	2/6=0.33	7	7*0.33=2.3	1000

4 PROBLEM MODELLING

In this section, we represent and extend our previous problem modelling [18] to model data velocity change at runtime. The list of all terminologies that will be used in this model is presented in Table 4.

4.1 Application Model

Stream workflow application can be represented as a Direct Acyclic Graph (DAG) with $G = (S, EX, E)$. S represents a set of N services $S = s_1, s_2, s_n, ..., s_N$, EX represents a set of P external sources $EX = ex_1, ex_2, ex_p, ..., ex_P$ and E represents a set of M edges/links between external sources and services and between services themselves $E = e_1, e_2, e_m, ..., e_M$. Each edge, e_m in workflow graph is represented as a tuple $(\psi^m, s^m_{dest}, ¥^m)$, where ψ^m denotes stream output source which is either an external source (ex^m_p) or an origin service (s^m_{org}), s^m_{dest} denotes destination service as the target of the edge e_m and $¥^m$ denotes the percentage of data generated by ψ^m that is routed towards s^m_{dest}. Each particular external source ex_p is represented as a tuple $ex_p = (\Lambda^{ex_p})$, where Λ^{ex_p} denotes the output data

Table 4
Problem Modelling Notation

Symbol / Term	Description
G	Workflow graph
S	Set of all graph services
EX	Set of all external sources in a workflow graph
E	Set of all graph edges
e_m	Edge m between to two services or an external source and service in a workflow graph
ex^m_p	External source p that is the origin source of the edge e_m
s^m_{org}	Service that is the origin source of the edge e_m
s^m_{dest}	Service that is the target of the edge e_m
ψ^m	Data source on edge e_m that can be external source ex^m_p or origin service s^m_{org} injecting its output data stream into the target of this edge s^m_{dest}
$¥^m$	Percentage of data that is routed from parent service to child service (100% in replica mode or any percent in partition mode)
ex_p	External source p in a workflow graph
Λ^{ex_p}	Output data rate of external source ex_p
s_n	Service n in workflow graph
MI^{s_n}	Number of floating-point operations required to process one MB of s_n input data (MI/MB)
λ^{s_n}	Amount of data produced by a given external source(s) and being consumed by a service s_n (MB/s)
γ^{s_n}	Proportion of output data to input data for s_n
$parent(s_n)$	Set of parent services for s_n
C	Set of all clouds in multicloud environment
c_g	Particular cloud g in multicloud environment
$c^g_{s_n}$	Placement cloud g of s_n
L	Network latency matrix
$L(c^g_{s_{n_1}}, c^g_{s_{n_2}})$	Latency between s_{n_1} placement cloud and s_{n_2} placement cloud
B	Network bandwidth matrix
$B(c^g_{s_{n_1}}, c^g_{s_{n_2}})$	Bandwidth between s_{n_1} placement cloud and s_{n_2} placement cloud
D	Data transfer cost matrix
$D(c^g_{s_{n_1}}, c^g_{s_{n_2}})$	Data transfer cost between s_{n_1} placement cloud and s_{n_2} placement cloud
VM^g	Set of all VMs in cloud g
vm^g_k	Particular VM k in cloud g
U^g	Set of all internal network links between VMs in cloud g
u^g_h	Particular internal link between vm^g_{org} and vm^g_{dest}
$MIPS_{vm^g_k}$	Rating of the capacity of VM k in cloud g
$¢_{vm^g_k}$	Provisioning cost of VM k in cloud g (cents/s)
$pro(s_n, t_i)$	Set of VMs that are provisioned from one cloud for s_n at t_i
$\varphi(s_n, vm^g_k)$	Data processing rate for s_n when mapped to vm^g_k
$\varphi'(s_n, pro(s_n, t_i))$	Total data processing rate for s_n when mapped to all VMs in $pro(s_n, t_i)$
$pro^+(s_n, t_i)$	Set of extra VMs being provisioned for s_n from its placement cloud at t_i
$pro^-(s_n, t_i)$	Set of VMs being deprovisioned from $pro(s_n, t_i)$ for s_n at t_i
$inStream(s_n)$	Input data stream rate of s_n
$outStream(s_n)$	Output data stream rate of s_n
$outStream'(s_{n_1}, s_{n_2})$	Amount of output data stream of s_{n_1} routed to s_{n_2} taking into account network bandwidth and latency
MSU	Minimum stream unit for the whole application (MB)
MSR	Minimum stream processing rate based on MSU for the whole application (MB/s)
ϑ^{s_n}	x MSUs based on percentage change from original data rate that being increased or decreased from input stream of service s_n
χ	Service unit data processing rate
ϱ	Data transfer time
$EC(S, T)$	Cost of running all provisioned VMs for all services in a workflow application during T time
$ec(s_n, t_i)$	Cost of running VMs provisioned for s_n at t_i
$CTS(S, T)$	Total data transfer costs performed by services for T time
$cts(s_n, t_i)$	Total cost of transferring data to s_n at t_i
$\check{c}(s_{n_1}, s_{n_2})$	Cost of transferring data from s_{n_1} parent service to s_{n_2} child service

rate (data velocity) of this data source.

Each particular service s_n, is represented as a tuple $s_n = (MI^{s_n}, \lambda^{s_n}, \gamma^{s_n})$, where MI^{s_n} denotes the number of floating-point operations required to process one MB of incoming data (service data processing requirement) in MI/MB, λ^{s_n} denotes the arrival rate of data streams generated by sources outside the application in MB/s (such as data streams generated by sensors) to be consumed by the service, and γ^{s_n} denotes the proportion of data generated by the service based on input streams.

Notice that, given the nature of stream workflow applications, it is possible that data generated by one service can be sent to one or more services, or can be split among different services. Thus, for service s_n, both parameters γ^{s_n} and ¥^{m} (in edges where such service is origin service) are necessary to define the whole application. Additionally, the minimum stream unit per second (denoted as MSR) should be defined to process streams that coming at different speeds, where each VM can process one or more units based on its computing power.

4.2 System Model

The cloud system is modelled as a tuple W = (C, L, B, D). A set of G clouds in the multicloud environment is denoted as $C = c_1, c_2, c_g, ..., c_G$. L, B, and D denotes matrices containing respectively the latency (in seconds), the bandwidth (in MB/s), and the data transfer cost (in cents/MB or ¢/MB) between each of the pair of clouds in C.

Each cloud, c_g is represented as a tuple (VM^g, U^g), where $VM^g = vm_1^g, vm_2^g, vm_k^g, ..., vm_K^g$ is a set of K virtual machines (compute resources) with different resource configurations deployed in c_g, and $U^g = u_1^g, u_2^g, u_h^g, ..., u_H^g, u_h^g = (vm_{org}^g, vm_{dest}^g)$, a set of H links that are part of the data center network topology.

Each VM deployed in the cloud, vm_k^g, is represented as a tuple $(MIPS_{vm_k^g}, ¢_{vm_k^g})$, where $MIPS_{vm_k^g}$ denotes floating-point operations computed by this VM according to its compute capacity per second and $¢_{vm_k^g}$ denotes the cost of provisioning such VM (in cents per second).

The data processing rate for s_n if it is mapped to vm_k^g is denoted as $\varphi(s_n, vm_k^g)$ and is calculated by dividing VM computing power by service unit data processing rate and service data processing requirement as follows:

$$\varphi(s_n, vm_k^g) = \frac{\lfloor MIPS_{vm_k^g}/\chi \rfloor * \chi}{MI^{s_n}} \text{ MB/s} \qquad (1)$$
$$Where\ \chi = MSR * MI^{s_n} and\ MIPS_{vm_k^g} \geq \chi$$

As s_n could be mapped to more than one VM to achieve user performance requirements, let $pro(s_n, t_i)$ be the set of VMs that are provisioned from one cloud for service s_n at t_i. The data processing rate for s_n if it is mapped to VMs in $pro(s_n, t_i)$ is denoted as $\varphi'(s_n, pro(s_n, t_i))$ and is calculating by summing data processing rates for s_n on all provisioned VMs at t_i as follows:

$$\varphi'(s_n, pro(s_n, t_i)) = \sum\nolimits_{v \in pro(s_n, t_i)} \varphi(s_n, v) \text{ MB/s} \qquad (2)$$

In stream workflow application, the calculation of data processing rate for each service s_n should be carried-out at runtime. This is because of the need to handle dynamic changes that result in varying the speed of input streams being injecting into this service. Thus, system should calculate this rate based on the updated input speed of a service after the occurrence of change request at runtime. Let $inStream(s_n)$ denotes the input stream of s_n and is the total rate of incoming data from external sources and internal sources (i.e. parent services) based on data modes used to route such streams toward this service:

$$\begin{aligned} inStream(s_n) = \lambda^{s_n} + \sum\nolimits_{e_m \in E | \psi^m = s_{org}^m \& s_{dest}^m = s_n} \\ (\gamma^{s_{org}^m} * \varphi'(s_{org}^m, pro(s_{org}^m, t_i))) * ¥^m \text{ MB/s} \\ Where\ \lambda^{s_n} = \sum\nolimits_{e_m \in E | \psi^m = ex_p^m \& s_{dest}^m = s_n} (\Lambda^{ex_p^m}) \end{aligned} \qquad (3)$$

The following data processing constraint of s_n is maintained:

$$\varphi'(s_n, pro(s_n, t_i)) \geq inStream(s_n) \qquad (4)$$

Each service s_n produces output stream as a result of computation. Let $outStream(s_n)$ denote the output data stream for a service s_n and is calculated by multiplying the total input rate of s_n by output data proportion/percent as follows:

$$outStream(s_n) = \gamma^{s_n} * inStream(s_n) \text{ MB/s} \qquad (5)$$

The velocity of data for given external source may change at runtime, which leads to a direct impact either an increase or decrease on the velocity of data for each s_n connected to this source. This change makes $inStream(s_n)$ and $outStream(s_n)$ be updated by the amount of data that being increased or decreased. Also, this change affects not only those services, but also has a subsequent change (i.e indirect impact) on the velocity of data for child services which have dependency-link with those services. Therefore, it is worth to note that the maximum number of velocity changes that can be sent at any instant of time is assumed to be one and such velocity change request (either increase or decrease request) is only happen via external source. Let ϑ^{s_n} denotes the amount of data stream (in MB/s) based on percentage change from original data rate that being increased or decreased to $inStream(s_n)$ as x MSUs. In case of data velocity decrease, ϑ^{s_n} should be $0 < \vartheta^{s_n} < inStream(s_n)$. The $inStream(s_n)$ will be updated by adding or subtracting x MSUs and $outStream(s_n)$ will be updated by multiplying the update total input rate of s_n by output data proportion/percent as follows:

$$\begin{aligned} inStream(s_n) &= inStream(s_n) \pm \vartheta^{s_n} \\ outStream(s_n) &= \gamma^{s_n} * inStream(s_n) \end{aligned} \qquad (6)$$

As well as the decrease in velocity of data for s_n leads to lower computing needs for maintaining the above data processing constraint, so that VM(s) that is not required will be deprovisioned. This results in cost reduction while meeting user real-time data processing requirements. While the increase in velocity of data leads to more computing demands to maintain the above data processing constraint for this high data rate, additional VM(s) will be provisioned. Let $pro^+(s_n, t_i)$ be the set of new VMs that need to be provisioned from $c_{s_n}^g$ (placement cloud of service s_n) at t_i to cope with the increase speed of data streams, and $pro^-(s_n, t_i)$ be the set of VM(s) from $pro(s_n, t_i)$ for service

s_n that will be terminated/deprovisioned at t_i in response to an decrease in the speed of data streams. Thus, $pro(s_n, t_i)$ is updated periodically at runtime by provisioning new VM(s) in case of velocity increases or deprovisioning VM(s) from the existing ones to respond to velocity decreases as follows:

$$pro(s_n, t_i) = \begin{cases} pro(s_n, t_{i-1}) \cup pro^{+}(s_n, t_i), \textit{if velocity incr.} \\ pro(s_n, t_{i-1}) - pro^{-}(s_n, t_i), \textit{if velocity decr.} \\ pro(s_n, t_{i-1}), \textit{otherwise (no change)} \end{cases} \tag{7}$$

As $pro(s_n, t_i)$ is updated at runtime, the $\varphi'(s_n, pro(s_n, t_i))$ is also updated, reflecting the new data processing rate for s_n based on the updated pro(s_n, t_i).

Given the change in velocity of data that either increases or decreases data rate which leads to provision more VMs or deprovision existing VMs at runtime, the execution cost needs to be calculated frequently. For our problem here, the calculation basis for the total execution cost of stream workflow application is per second. If T is total time duration, for cost calculations it is divided into several intervals (i.e. $t_1, t_2, ..., t_I$).

Additionally, we assume that every data stream should be processed, as unprocessed data streams lead to incorrect results. We also assume that the order of stream portions should be maintained during the distributed among the corresponding compute resources. Based on these assumptions, we maintain user specific throughputs for all services and end-to-end latency (response time) as low as possible or even bounded when it is being increased. Thus, the incoming data streams are processed as they arrive and the latency is maintained, which is a time from a stream being added to input queue until its emission from the service as output stream. Of course, in case of a child service receives two or more dependency streams from its parents services, the latency is from the time of the last stream being added to input queue until its emission from child service.

The cost of running VMs used by service s_n to process incoming streams per second t_i is denoted as ec(s_n) while the total cost of running all VMs used by all services to process incoming streams during period of time T is denoted as EC(S,T). The EC(S,T) is calculated by summing VM provisioning costs for all services for T time as follows:

$$EC(S,T) = \sum\nolimits_{t_i} \sum\nolimits_{s_n} ec(s_n, t_i) \qquad \text{cents} \tag{8}$$

The ec(s_n, t_i) is calculated by totalling the costs of all VMs provisioned for s_n at t_i as follows:

$$ec(s_n, t_i) = \sum\nolimits_{v \in pro(s_n, t_i)} \text{¢}_v \qquad \text{cents} \tag{9}$$

The data transfer cost is based on the amount of data being moved, the cost of data transfer charged by cloud provider, and network bandwidth. In a dynamic workflow application, the velocity of data determines the speed of generation, processing and analysis of data, where both input and output data are moved among different clouds. As we mentioned before, the change in velocity of data affects the data transfer cost as increasing speed leads to an increase in the cost and vice versa, so that the cost calculation needs to be carried-out per second. Let $cts(s_n, t_i)$ denotes the cost of transferring output streams of parent services to s_n at t_i, and $CTS(S,T)$ denotes the total data transfer cost for the amount of data being moved for all services during the period of time T. The $CTS(S,T)$ is calculated by summing the costs of data transfer between services for T time as follows:

$$CTS(S,T) = \sum\nolimits_{t_i} \sum\nolimits_{s_n} cts(s_n, t_i) \qquad \text{cents} \tag{10}$$

The cts(s_n, t_i) is calculated by totalling the costs of transferring data performed by parent services to s_n at t_i as follows::

$$cts(s_n, t_i) = \sum\nolimits_{s_x \in parent(s_n)} \breve{c}(s_x, s_n) \text{ cents} \tag{11}$$

$$\breve{c}(s_x, s_n) = \begin{cases} 0, & \text{if } c^g_{s_x} = c^g_{s_n} \\ outStream'(s_x, s_n) * D(c^g_{s_x}, c^g_{s_n}), & \text{otherwise} \end{cases}$$

$$outStream'(s_x, s_n) \begin{cases} outStream(s_x) * ¥^m, & \text{if } \varrho \leq 1 \\ \frac{outStream(s_x) * ¥^m}{\varrho}, & \text{otherwise} \end{cases}$$

$$\textit{Where } \varrho = \frac{outStream(s_x) * ¥^m}{B(c^g_{s_x}, c^g_{s_n})} + L(c^g_{s_x}, c^g_{s_n})$$

Overall, the objective function is to minimise the total cost of executing the dynamic workflow without violating data dependences and real-time performance requirements while dealing with changes in speed of data at runtime:

$$min\ f(S,T) = EC(S,T) + CTS(S,T) \tag{12}$$

Eq. 12 is solved for minimisation to generate a cost-efficient scheduling plan for the execution of stream workflows. Considering services' data processing requirements and the variety of resources offered by multiple clouds, each service can be mapped to more than one resource. This allows maintaining a service data processing constraint based on input data rate (refer to Eq. 2 and Eq. 4). If we relax such mapping constraint thus each service is mapped only to one resource (i.e. $|pro(s_n, t_i)| = 1$), assuming that this resource is sufficient to meet service's data processing constraint (Eq. 4). This relaxed constraint makes the problem 0-1 assignment problem. In this problem, the assignment matrix M indicates that a service i is assigned to resource j if $m_{ij} = 1$. This problem is well-known NP-hard [26]. Consequently, if we consider the mapping of service to more than one resource without any relaxation now, our problem is even harder than 0-1 assignment problem. Thus, it is an NP-hard problem. Moreover, our problem belongs to NP because if a feasible resource allocation solution is given, this solution can be tested in polynomial time using Algorithm 1. Accordingly, our problem is NP-complete problem.

5 Proposed Adaptive Scheduling Technique

As we discussed in the previous section, our scheduling problem is NP-complete problem. Thus, the problem's search spaces are complex, with large sets of VM offerings provided by various cloud infrastructures and many constraints that need to be fulfilled. Examples of these

Algorithm 1 polynomial-time algorithm for checking the feasible solution

```
totalDPRate ← 0
for each service s_n in S do
    for each VM vm_k^g from pro(s_n, t_i) do
        totalDPRate = totalDPRate + φ(s_n, vm_k^g)
    end for
    if totalDPRate < inStream(s_n) then
        return false {this is not feasible solution}
    end if
end for
```

constraints are data dependencies, user-defined real-time performance, throughput and end-to-end latency. Indeed, the search space of finding candidate solutions for the efficient execution of a stream workflow application rapidly increases with the size of the problem. Furthermore, the fluctuation of data velocity over time makes it necessary to re-explore the complex search space to find a sub-optimal solution as quickly as possible. The exhaustive search for an optimal solution is not feasible. Consequently, the goal is to find a near-optimal solution in the complex search space and revise it quickly to tackle the changes in data velocity over time without violating data dependences and real-time performance requirements while minimising the total execution cost (Eq 12). As we cope with velocity change for this workflow application, the following are cases of changing in input stream rate of a service:

- The speed of output stream of an external source connected to this service is either increased or decreased.
- The speed of the output stream of parent service(s) connected to this service is either increased or decreased. This happens when the increase or decrease in the speed of stream propagated from parent services due to the increase or decrease in the speed of stream for connected external sources

From the aforementioned goal, we have two challenges: (1) explore large search space to find candidate solution at deployment time and (2) revise this solution quickly with each velocity change request that occurs at runtime to locate a sub-optimal solution to respond to such request. For the first challenge, the genetic algorithm is a useful algorithm in exploring complex search space to enable practical implementation of the optimising problem. Thus, the objective function of Eq. 12 can be considered as a fitness function of the genetic algorithm. While for the second challenge, Greedy heuristic can be used to adopt a deployment plan generated by the genetic algorithm at runtime. Because it provides an immediate sub-optimal solution for tackling the velocity change request as it needs a relatively small time to compute. Thus, it can fulfil the need to make a scheduling decision under time constraints, enabling the practical implementation of optimising objective function at a given point. Accordingly, we propose a new dynamic scheduling technique for stream workflows.

The proposed technique is a two-phase dynamic workflow scheduling technique that incorporates two advanced optimisation algorithms (i.e. random immigrants genetic algorithm in Phase 1 and two-level greedy algorithm in Phase 2). It effectively performs dynamic scheduling of stream workflow applications in multicloud environment and intelligently response to changes happen at runtime (i.e. velocity changes) with minimal execution costs. In Phase 1, the proposed random immigrants GA is called at the begin of workflow execution to find the best global sub-optimal solution to deploy a given workflow. While in Phase 2, The proposed two-level greedy algorithm is used to adapt a scheduling plan to respond to the changes in the velocity of data streams at runtime. The flowchart of the proposed technique with its phases are provided in Appendix B.

Table 5
Time complexity of random-based immigrants GA

Name	Time complexity
Random population generation	$O(su)$
Fitness Function	$O(ps^2d)$
Roulette wheel selection with binary search	$O(plog(p))$
Crossover	$O(s)$
Mutation	$O(sv)$
Sort	$O(plog(p))$
Random-based immigrants schema	$O(s^2d)$
Total	$O(gps^2d)$

g the number of generations (as termination condition), p the size of population, s the length of candidate solution (number of services), u the maximum number of required MSUs of any service, v the number of VM offers in the placement cloud and d the maximum number of stream dependencies of any services

5.1 GA with Random Immigrants Scheme

Traditional GA has a considerable problem, which is convergence that prevents genetic diversity of the population. Therefore, to avoid such problem and to enhance the genetic diversity of the population, random immigrants schema is used [27]. This schema retains the diversity level of the population every generation via replacing a portion of candidate solutions in the current population with random candidate solutions called immigrants. Accordingly, we propose a random-based immigrants genetic algorithm that is able to find sub-optimal resource selection solution for scheduling stream workflow application in multicloud environment. It exploits data locality by selecting the most appropriate datacenter for each service, which leads to the reduction in both execution and data transfer costs. It generates the initial population randomly, and then evaluate the candidates and sort them in ascending order of fitness. During each generation, the elite candidate is selected and m random immigrants are generated then replaced the worst n candidates in the current population. Following the evaluation of m immigrants, the selection, crossover and mutation operators are applied. Finally, the elite candidate is added and the evolved population is evaluated and then sorted in ascending order of fitness before going to the next generation. Algorithm 2 shows the pseudocode of the proposed random immigrants GA provisioning and scheduling algorithm. The time complexity of this algorithm is presented in Table 5. The Watchmaker framework for evolutionary computation [28] is used to implement this algorithm.

5.2 Two-level Greedy Algorithm

We propose a new two-level greedy algorithm that uses Minimax with Alpha-Beta pruning method in game theory. It minimises the maximum resource provisioning cost by finding the best resource selection solution for services that

Algorithm 2 GA with Random Immigrants Scheme

```
P ← empty initial population
generate N candidates randomly and add them to P
calculate fitness values for candidates in P
sort candidates in P in ascending order of fitness
while condition not satisfied do
    perform elitist selection
    P' = generate m random immigrants
    replace worst m candidates in P by random immigrants in P'
    calculate fitness values for random immigrants
    select candidates using selection operator for evolving
    create new offsprings using crossover operator
    create new offsprings using mutation operator
    add elite candidates to the evovled population
    calculate fitness values for candidates of the evovled population
    sort candidates of the evolved population in the ascending order of
    fitness
end while
return best candidate (candidate with minimum cost)
```

Algorithm 3 Two-level Greedy Algorithm

```
1: min ← TreeNode(−1, ∞) {-1 is vm global id and ∞ is value}
2: max ← TreeNode(−1, −∞) {-1 is vm global id and -∞ is value}
3: depth ← 2 {depth level in game tree}
4: affectedSIDs ← get service ids affected by velocity change request
5: for each service s_n in affectedSIDs do
6:     if velocity change request is increase request then
7:         Velocity_Increase_Req_Proc(s_n, min, max, depth)
8:     else
9:         Velocity_Decrease_Req_Proc(s_n, min, max, depth)
10:    end if
11: end for
```

affected by data velocity changes. Minimax with Alpha-Beta pruning method is considered as a powerful searching and decision-making algorithm on game tree to find optimal/sub-optimal result from possible choices. Thus, this method is used in our algorithm to find the best resources with the lowest provisioning cost at runtime to achieve the updated data processing rate for each service affected by the velocity change request. The direct effect happens when the service is connected to an external source whose data velocity will be changed. While indirect effect occurs when the service is in the velocity change path.

Our proposed algorithm addresses the problem of ongoing resource scaling under the dynamic variations of data stream rates by managing resources over time. This

Algorithm 4 Velocity_Increase_Req_Proc(s_n,min,max,depth)

```
1: reqUnits ← 0
2: unitMIPS ← MSR * MI^{s_n}
3: avalVms ← get VM offers from service placement cloud c^g_{s_n}
4: avalVms = avalVms − {x ∈ VM^g | MIPS_x < unitMIPS}
5: extraAchievedUnits ← φ'(s_n, pro(s_n, t_i))/MSR − ⌈(inStream(s_n) * MI^{s_n})/unitMIPS⌉
6: incRate ← get data rate increases over service input rate
7: reqUnits ← get number of MSUs required based on incRate
8: reqUnits ← reqUnits − extraAchievedUnits
9: nodes ← create tree nodes list for avalVms list
10: while reqUnits > 0 do
11:     shuffle nodes in the list and construct tree with specified depth
12:     root ← get root of constrcuted tree
13:     best ← Minimax_AlphaBeta(depth, true, root, min, max) {best node for VM selected}
14:     VMList = VMList ∪ best.getVmgid()
15:     reqUnits = reqUnits − ⌊(MIPS_{best.getVm()}/unitMIPS)⌋
16: end while
17: add VMList of s_n to ServiceVMsMap { VMList≠ φ}
```

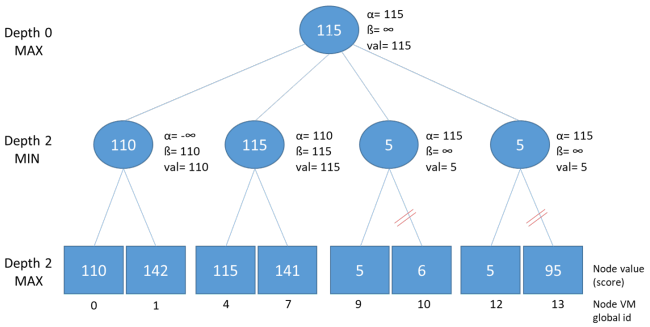


Figure 1. Search Tree Sample

algorithm at first level determines the services whose input data rates will be changed due to the received velocity change request. Then, at the second level, it finds the best resource provisioning/deprovisioning solution(s) that will be used to revise the scheduling plan. With the occurrence of a velocity change request, it dynamically and quickly updates the scheduling plan to respond to this change request while reducing the overall provisioning cost. The pseudocode of the proposed two-level greedy algorithm is shown in Algorithm 3 and the time complexity analysis of this algorithm is presented in Table 6. The pseudocode of two procedures that used in this algorithm to respond to velocity increase and decrease requests are shown in Algorithm 4 and Algorithm 5 respectively. The pseudocode of Minimax with Alpha and Beta algorithm that used in both procedures (Algorithm 4 and Algorithm 5) is shown in Algorithm 6. Algorithm 7 shows the pseudocode of evaluation function used in Algorithm 6.

Algorithm 5 Velocity_Decrease_Req_Proc(s_n,min,max,depth)

```
1: redUnits ← 0
2: unitMIPS ← MSR * MI^{s_n}
3: SPVMs ← pro(s_n, t_i)
4: extraAchievedUnits ← φ'(s_n, pro(s_n, t_i))/MSR − ⌈(inStream(s_n) * MI^{s_n})/unitMIPS⌉
5: decRate ← get data rate decreases from service input rate
6: redUnits ← get number of MSUs based on decRate
7: redUnits ← redUnits + extraAchievedUnits
8: while redUnits > 0 do
9:     remove VM(s) from SPVMs that achieved units > redUnits
10:    if SPVMs is empty then
11:        return {no provisioned VM can be deprovisioned}
12:    end if
13:    construct tree from SPVMs list with specified depth
14:    root ← the root of constrcuted tree
15:    best ← Minimax_AlphaBeta(depth, true, root, min, max)
16:    VMList = VMList ∪ best.getVmgid()
17:    redUnits = redUnits − ⌊(MIPS_{best.getVm()}/unitMIPS)⌋
18:    SPVMs = SPVMs − best.getVm()
19: end while
20: if VMList is not empty then
21:    add VMList of s_n to ServiceVMsMap
22: end if
```

Prior processing the velocity change request, the proposed technique finds the ids of service affected by this request directly or indirectly (Algorithm 3 Line 4). Then, for each service affected, it finds the best provisioning or deprovisioning solution based on the type of velocity change request. If the request is velocity increase request (Algorithm 3 Line 6), it calls Algorithm 4 to get VM of-

Algorithm 6 Minimax_AlphaBeta(depth, maximisingPlayer, node, alpha, beta)

```
1: if depth == 0 then
2:     return evaluate(node)
3: else if maximisingPlayer then
4:     for each child of node do
5:         TreeNode val =
           Minimax_AlphaBeta(depth - 1, false, child, alpha, beta)
6:         if val.getValue() > alpha.getValue() then
7:             alpha = val
8:         end if
9:         if beta.getValue() <= alpha.getValue() then
10:            break {alpha cut-off}
11:        end if
12:    end for
13:    return alpha
14: else
15:    for each child of node do
16:        TreeNode val =
           Minimax_AlphaBeta(depth - 1, true, child, alpha, beta)
17:        if val.getValue() < beta.getValue() then
18:            beta = val
19:        end if
20:        if beta.getValue() <= alpha.getValue() then
21:            break {beta cut-off}
22:        end if
23:    end for
24:    return beta
25: end if
```

Algorithm 7 Evaluation Function - evaluate(node)

Require:

1: *reqUnits, redUnits, unitMIPS*
2: $value, cost \leftarrow 0$ {value for increase request and cost for decrease request}
3: **if** *velocity change request is increase request* **then**
4: *VMboottime* $\leftarrow$ get boottime for VM node
5: *achievedUnits* $\leftarrow$ get units achieved by VM node
6: $value \leftarrow (achievedUnits/(reqUnits * \dot{\varsigma}_{vm_k^g}))/VMboottime$
7: $value \leftarrow value + \lfloor MIPS_{vm_k^g}/(unitMIPS * numServiceDependencies) \rfloor / \dot{\varsigma}_{vm_k^g}$
8: $node.value \leftarrow value$
9: **else**
10: *achievedUnits* $\leftarrow$ get units achieved by VM node
11: $cost \leftarrow (achievedUnits/(redUnits * \dot{\varsigma}_{vm_k^g}))$
12: $node.value \leftarrow cost$
13: **end if**
14: *return node*

fers of service placement cloud and then finds the extra MSUs that are achieved by the current provisioned VMs in accordance to service input data rate. In other words, extraAchievedUnits is the number of extra MSUs that are not required to achieve service input data rate due to over-provisioning. Next, such algorithm calculates the number of MSUs required for data rate being increased over service input data (i.e. reqUnits). Based on the increase in data velocity, reqUnits is the number of MSUs required to achieve this increase in data rate. For example, consider MSU for the whole application is 1MB/s, original service input data rate is 3MB/s and this rate is increased by 2MB/s at a given time to be 5MB/s, therefore reqUnits = $\frac{2MB/s}{1MB/s} = 2$ MSUs. From reqUnits, the extra achieved units is deducted. After that, it calls Algorithm 6 several times to finds best VM(s) to provision until achieving the required units. While, with velocity decrease request (Algorithm 3 Line 8), it calls Algorithm 5 to get the list of VMs provisioned for a service and then finds the extra MSUs that are achieved by these

Table 6
Time complexity of two-level greedy algorithm

Name	Time complexity
Get affected services	$O(s)$
Velocity increase request procedure	$O(ub^m)$
Velocity decrease request procedure	$O(ub^m)$
Minimax alpha-beta	$O(b^m)$
Evaluation function	$O(1)$
Total	$O(sub^m)$
s the number of services, u the maximum number of required MSUs of any service, b the branching factor and m the maximum depth of the tree	

VMs based on service input data rate. Next, such algorithm calculates the number of MSUs based on the data rate being decreased from service input data, and then increases this number by extra achieved units. Indeed, the calculation of extraAchievedUnits is the same that the one has done in Algorithm 4 and redUnits is similar to the one done in Algorithm 4 but for the amount of data decreases. After that, it removes those VMs from the list of provisioned VMs where their powers achieved units greater than the number of minDPunits that will be removed. The remaining VMs in this list will be used to find the best VM to deprovision using Algorithm 6.

Each run of the game finds the best VM to provision it in case of velocity increases or to deprovision it in case of velocity decreases. For instance, let us say we have six VMs remaining in the VM list for the game. The global ids of these VMs are 0, 1, 4, 7, 9, 10, 12 and 13, and their values evaluated using evaluation function (Algorithm 7) are 110, 142, 115, 141, 5, 6, 5 and 95 respectively. Figure 1 shows a search tree sample and the best VM selected. Since multiple VMs may be needed to achieve the updated data processing rate or may be released in response to a velocity decrease request, the game will be repeated to produce the best solution. For each VM selected, the number of minimum data processing units achieved based on the computing power of this VM in one game is deducted from the total required units (i.e. reqUnits) in case of velocity increases or from total reduced units (i.e. redUnits) in case of velocity decreases.

6 Experiments and Discussion

6.1 Experiment Methodology

6.1.1 Configuration of Workflow Application

In our previous work [18], we simulated stream workflow applications using common workflow structures (Montage, Inspiral, Epigenomics and CyberShake) and described the additional parameter configurations. These applications with their different sizes are used in our experiments.

6.1.2 Multicloud Environment

In our previous work [18], we modelled three different cloud system providers, namely Amazon EC2, Google Cloud Engine, and Microsoft Azure, to form a multicloud environment. This modelled multicloud environment is used here for our experiments. Also, to model boot time (startup time) for each VM configuration in the modelled clouds, we use the average range of VM startup time defined in [29]. For each modelled cloud, we generate random numbers from

Table 7
Percentage ranges of data velocity increase and decrease amounts

Velocity Range	Increase		Decrease	
	Minimum (Percent)	Maximum (Percent)	Minimum (Percent)	Maximum (Percent)
Low	10	30	5	15
Medium	50	70	25	35
High	90	100	45	50

the defined range and then assign these numbers to its VM configurations.

6.1.3 Configuration of Data Velocity

To model the amount of data that is being increasing or decreasing in velocity change request for one external source, we utilised future data rates given in Gartner foreseen [30]. The one connected vehicle will generate as much as 25GB/hour of data, equivalent to 8MB/s. By considering this value as the average data rate of an external source in a workflow application, we create different percentage ranges for modelling the increase and decrease in data velocity. For velocity increase, we model the value of increase in data velocity as a percentage that is increased from current data rate. Similarly, we model the data velocity decrease as percentage of decrease in the current data rate. Table 7 lists the percentages of change to increase and decrease data velocity. It is worth to note that as there is a minimum limit for a stream unit in a workflow application, the change value will be approximated/rounded to the nearest given MSU. As an instance, if the minimum stream unit per second is 1MB/s and there is a 65% increase in data velocity from 5MB/s as original data rate is chosen randomly, the approximation will be applied on the change value (3.25MB/s) to be 3MB/s (i.e. the nearest value based on MSU) so that the new data rate will be 8MB/s.

6.1.4 Workflow and Simulation Parameters

To run our experiments, we need to configure a set of parameters for both workflow application and simulator. These parameters and their values are fixed for all scenarios and listed in Table 8. For external data source rate, the value considered is from the data velocity configuration discussed in the previous subsection. For network bandwidth and latency for ingress and egress traffic, cost of data transfer and service data processing requirement, we considered the medium ranges of these parameters that presented in [18].

6.1.5 Experimental Scenarios

Our evaluations for efficiency and performance of the proposed technique are described in the below paragraphs.

Comparison with lower bound (Evaluation 1) – Study and compare the proposed dynamic scheduling technique in finding the best resource provisioning solution and adapting scheduling plan in response to velocity changes with competitors (baseline algorithm and random-based immigrants GA scheme) to approach lower bound. This comparison is in terms of the execution cost of different workflow applications for 3 minutes simulation time. A realistic baseline algorithm is created for our problem that does not need to use any advanced heuristic. It finds VM with the highest computing power and then provisiones it to respond to velocity increase requests. While with velocity decrease requests, it deprovisions one or more VMs from the available VMs to respond to these requests. The aim of comparison with baseline algorithm is to appreciate the necessity of our proposed technique to find the best resource provisioning solution and adapting the scheduling plan in response to velocity increases/decreases. The comparison with GA is aimed at evaluating the proposed technique with another meta-heuristic algorithm that is widely used in workflow scheduling research works in order to further proof its efficiency. Furthermore, the comparison with lower bound is necessary to show how the proposed approach is effective compared to its competitors. In lower bound, we relaxed the same constraints that are discussed in our previous work [18]. For the comparison, we consider the results obtained from lower bound as the base values.

Guarantee processing speed for execution time (Evaluation 2) – Study the efficiency of the proposed dynamic scheduling technique in guaranteeing the processing speed required with different workflow applications under different velocity changes. This evaluation aims to show the data processing constraint is satisfying at all times with changing data velocity. The baseline here to achieve real-time user-defined requirements and end-to-end execution time is that the computing power available should be sufficient to process all incoming data without data loss. In other words, the computing capacity should be greater than or equal the velocity of incoming data at runtime. Thus, the sufficient computing capacity should be always maintained while the velocity of data increases or decreases at runtime. The experimental results will be collected before the end of simulation due to at that time all velocity changes have

Table 8
Workflow and simulation parameters

Parameter	Value
External Source Data Rate	5 MB/s with increase-velocity experiment 10 MB/s with decrease-velocity experiment
Ingress Network Bandwidth	Range [615, 926] MB/s
Ingress Network Latency	Range [0.00064, 0.00086] second
Egress Network Bandwidth	Range [122, 218] MB/s
Egress Network Latency	Range [0.021, 0.031] second
Data Transfer Cost	Ingress traffic: 0 Egress traffic: Range [0.013 - 0.019] cents/MB
Type of Service	50% unmovable services 50% movable services
Service Data Processing Requirement	Range [1348, 2674] MI/MB
Service Data Processing Rate	System-calculated rate based on input stream(s)
Data mode type	Replica
Service Output Data Rate	Range [1, 50] % of input rate
Minimum Stream Unit (MSU)	1 MB
Minimum Stream Processing Rate (MSR)	1 MB/s
GA - Population Size	50
GA - Generation Limit	50
GA - Elitism	1
GA - Crossover Probability	0.8
GA - Mutation Probability	0.3
GA - Number of Random Immigrants	5
Number of Velocity Change Events	2
Delay between Velocity Change Events	10 seconds
Simulation Time	180 seconds (3 minutes)
Note: Detailed information about the ranges of values for simulation parameters are provided in [18].	

been made and handled by the proposed technique. This ensures the efficiency of the proposed technique to adopt a scheduling plan to handle velocity changes at runtime. Also, this allows to guarantee processing speed required to achieve end-to-end execution time.

Efficiency of handling velocity change (Evaluation 3) – Study and compare the proposed dynamic scheduling technique with random-based immigrants GA scheme based on performance matrix presented in Figure 2 for performing dynamic scheduling at runtime. The aim of this evaluation is to determine how our proposed technique is effective in maintaining the quality of solution. This comparison is in terms of the quality of solution for the revised scheduling plan. The quality of solution includes solution cost (provisioning + data transfer cost per second) after the data velocity change request is applied, and the number of allocation changes applied on the current scheduling plan to respond to this change request. The GA responses to each velocity change request by generating a totally new scheduling plan, which serves as a revised plan to replace the old one. While the proposed technique revises the current scheduling plan. When GA applied those VMs in a new plan that exist in the old plan are being excluded to avoid VM duplication. By doing so, only VMs in the new plan that do not exist in the old plan will be provisioned and those VMs in the old plan that are not part of the new plan will be deprovisioned. Therefore, the number of allocation changes includes the changes in provisioning plan (for new VMs that are not in the old plan) and deprovisioning plan (for provisioned VMs that do not exist in the new plan).

In aforementioned scenarios, data rate of each external source in a workflow application is set to be 5MB/s with velocity-increase experiment or 10 MB/s with velocity-decrease experiment at the beginning of execution. As velocity change requests being sent, the data rate of chosen external sources will be increased or decreased according to the conducted experiment.

Comparing the proposed technique with its competitors based on lower bound values can demonstrate the efficiency of the proposed technique in finding the best solution; this comparison will also show which technique or algorithm is closer to lower bound results. We can further quantify the proposed technique efficiency by assessing its ability to guarantee data processing constraint all the time. In addition, comparing and evaluating the quality of solution generated by the proposed technique in comparison with GA allows to evaluate the performance of the proposed technique in relative to the performance of GA.

6.2 Experimental Results

To evaluate the efficiency and performance of the proposed technique, we conduct our experiments in a simulation environment. This is because we need a controllable and repeatable environment to configure the parameters of each experiment scenario, and then compare the results obtained from the proposed technique with those from competitors under the same environment conditions. In real environment, some parameters like network bandwidth and latency cannot be controlled, making environment conditions change with each execution of a workflow application.

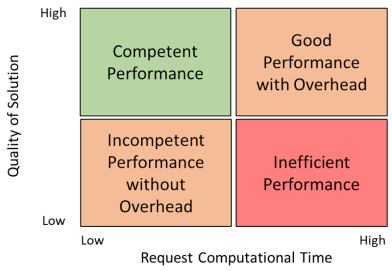


Figure 2. Performance Matrix

Thus, conducting our experiments in a real environment will produce inconsistent evaluation results, where these results cannot be used to assess the efficiency of proposed technique and the quality of solution produced. Accordingly, we conduct our experiments using IoTSim-Stream [31], our simulation toolkit for modelling and simulating stream workflow applications in multicloud environments.

The experimental scenarios are performed in a simulation environment (by using IoTSim-Stream) on a Nectar Cloud virtual machine that has 8 vCPUs, 32GB of RAM memory and running Ubuntu 16.04.1 LTS, and the experimental results are collected. Since a genetic algorithm is used in our proposed technique, each experimental scenario runs ten times, and the average value of the obtained results is taken and used in the representation of experimental results. For Evaluation 3 results, we present the average value for both solution cost and number of allocation changes since two velocity changes are made during the simulation time.

6.2.1 *Evaluation 1: Results*

We conducted experiments to record the total execution cost achieved by the proposed technique, its competitors (Baseline and GA) and lower bound for modelled workflow applications under different ranges of velocity increase and decrease. The experimental results for the first evaluation showed that the total execution costs of modelled workflow structures under different velocity change ranges for both velocity-increase and velocity-decrease have not changed significantly. Therefore, we only present those results for medium range of velocity change.

Figure 3 and Figure 4 depict the relative difference of execution cost for modelled workflow applications under medium range of velocity increase and decrease that achieved by the proposed technique and the competitors. From these results, our analysis and findings are:

- With various workflow applications, the proposed technique is efficient in finding the best solution to quickly respond to velocity change requests and then dynamically updating the current scheduling plan. By comparing the execution cost (relative difference) resulted from using the proposed technique with its competitors based on lower bound (see Figure 3 and 4), the proposed technique reduces relative execution cost with velocity increase request in comparison to Baseline by at least 73.79%, and in comparison to GA by at least 40.2%; While with velocity decrease

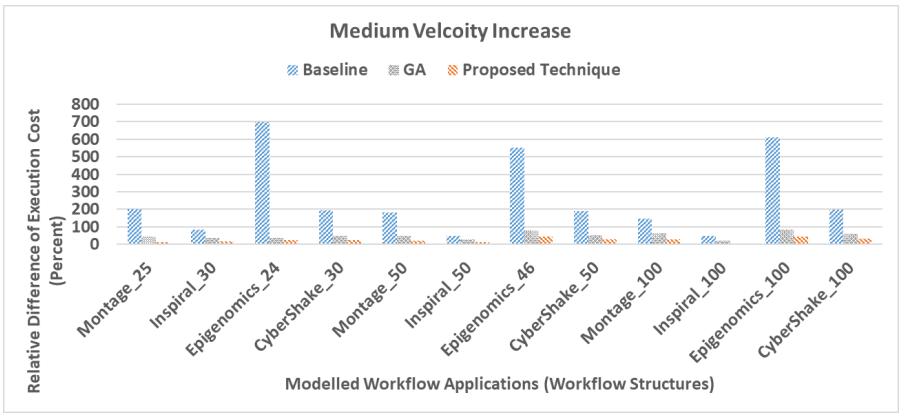


Figure 3. Relative difference of execution cost vs. Modelled workflow applications under medium range of velocity increase

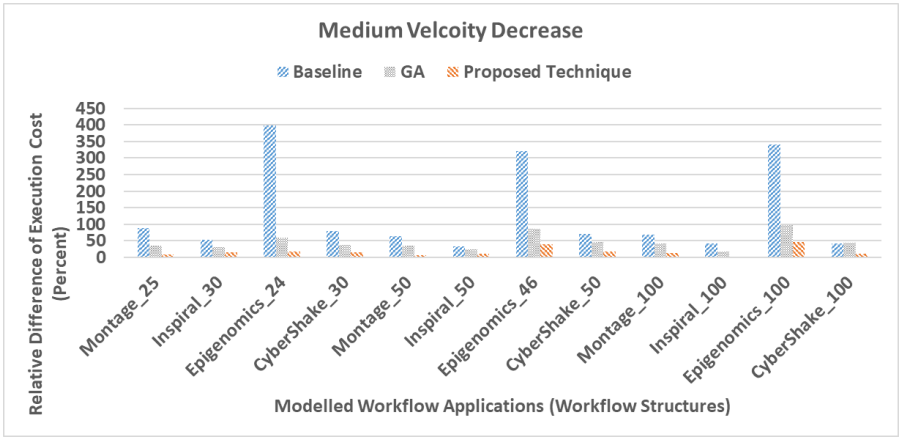


Figure 4. Relative difference of execution cost vs. Modelled workflow applications under medium range of velocity decrease

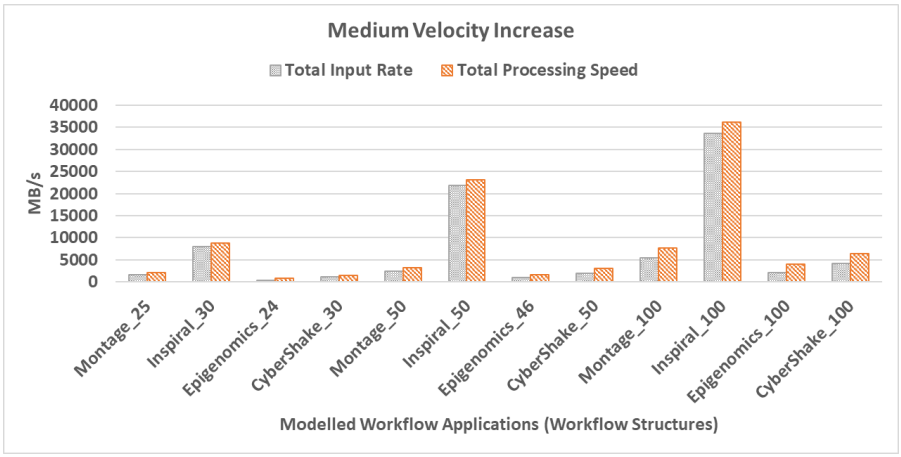


Figure 5. Total Input Rate vs. Total Processing Speed for different workflow structures (medium velocity increase range)

request, the proposed technique reduces relative execution cost in comparison to Baseline by at least 68.1%, and in comparison to GA by at least 50.3%. Therefore, the results of total execution cost obtained through the proposed technique outperformed the results obtained from both baseline and GA to approach lower bound with most workflow structures. The reason behind that is the proposed technique uses both GA and greedy heuristic. It uses GA at first phase for exploiting data locality to find near-optimal placement and scheduling plan, which reduces resource provisioning and data transfer costs. It then uses greedy heuristic at second phase to find the best provisioning plan that reduces the provisioning cost as much as possible to respond to any velocity change request. For instance, with velocity increase request, the proposed technique achieved the same total execution cost as lower bound achieved with

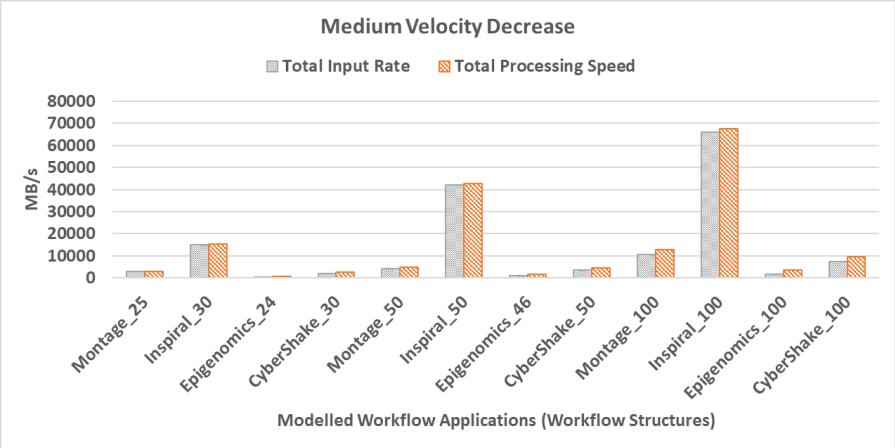


Figure 6. Total Input Rate vs. Total Processing Speed for different workflow structures (medium velocity decrease range)

Inspiral_100, 12.7% more than lower bound with Inspiral_50 and 18.3% more than lower bound with Inspiral_30; while Baseline and GA are 48.5% and 20.2% more than lower bound with Inspiral_100, 48.4% and 27.9% more than lower bound with Inspiral_50, and 82.6% and 37.1% more than lower bound with Inspiral_30 respectively. Also, with velocity decrease request, the proposed technique is 13.9%, 7.4% and 8.3% more than lower bound with Montage_100, Montage_50 and Montage_25 respectively; while Baseline and GA are 68.2% and 41.6% more than lower bound with Montage_100, 64.8% and 35.9% more than lower bound with Montage_50, and 88.5% and 34.7% more than lower bound with Montage_25 respectively.

- The total execution cost for the proposed technique is a maximum of 32% of the cost generated by lower bound from medium velocity change. The reason for this difference is due to the structure of workflow which may lead to processing less data. Thus, the provisioning cost reduction factor contributes more to the total execution cost. Based on that, lower bound produces unachievable results as VM provisioning constraint is relaxed, while the proposed technique maintains this constraint.
- As data velocity increases from low to high range, the total execution cost for modelled workflow applications is slightly increased. The reason behind that is the proposed technique is able to revise the current plan to cope with velocity increase changes with minimal cost, leading to cost reduction even with high velocity of data.
- The proposed technique is an adequate and practical dynamic scheduling method with competent accuracy. This is because it takes all the defined constraints into consideration while meeting user real-time performance requirements and reducing the overall execution cost with different workflows.

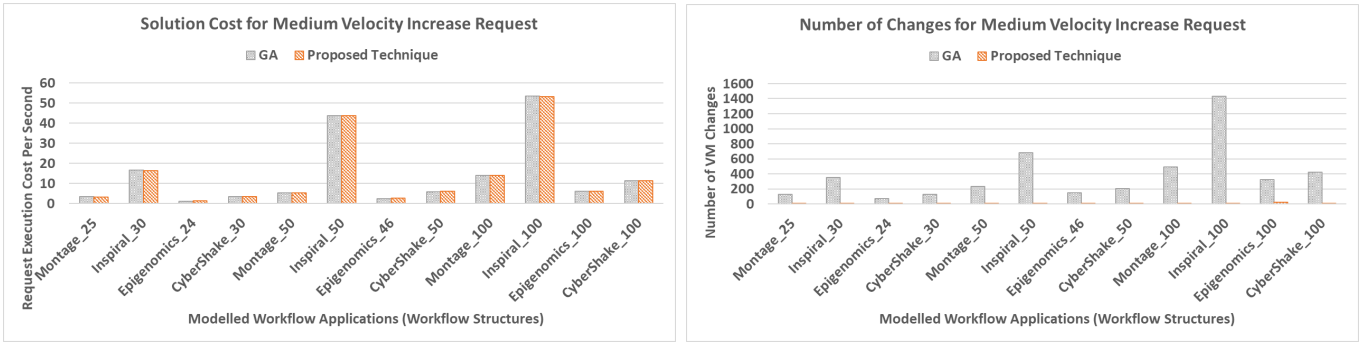
6.2.2 Evaluation 2: Results

We conducted experiments to record total input data rate (in MB/s) and total processing speed (in MB/s) achieved by proposed technique for modelled workflow applications under different ranges of velocity increase and decrease. From the results obtained, we present here those results for medium range of velocity increase and decrease since these results are enough to reach to the conclusion. Figure 5 and Figure 6 show the experimental results achieved by the proposed technique in term of total input rate and total processing speed. From the presented results, it is clear that the proposed technique always guarantees processing speed to process incoming data with all workflow applications. Even more, it also has some extra computing power to handle increase in data velocity with immediate response and without the need to reschedule the execution plan.

6.2.3 Evaluation 3: Results

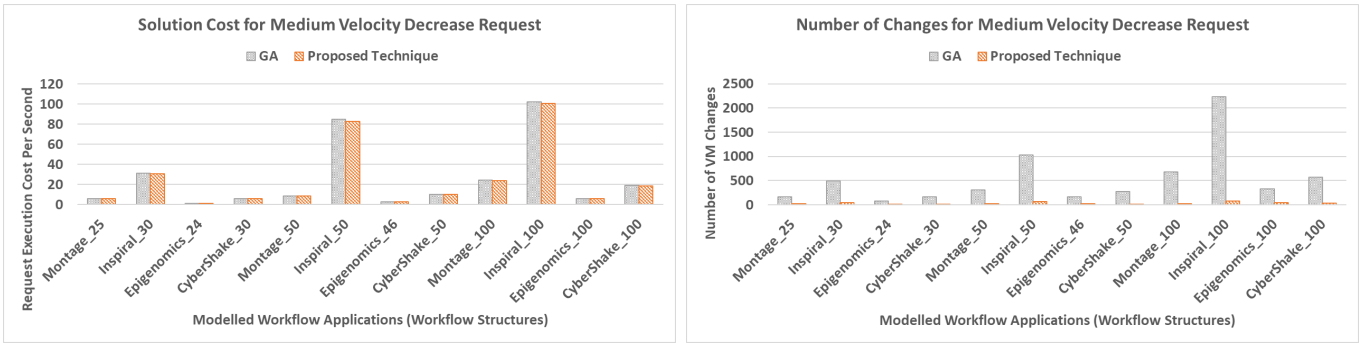
We conducted experiments to collect solution cost and number of allocation changes achieved by proposed technique and GA for modelled workflow applications under different ranges of velocity increase and decrease. It is worth noting that we do not present the experiential results for end-to-end latency because our assumption in problem modelling is that every data stream arrives will be processed as soon as the dependency is achieved. Moreover, we do not present experimental results for the computational time required to respond to velocity change requests because it is machine-dependent and GA needs more time due to its evolutionary nature. The experimental results for medium range of velocity changes (including both velocity increase and decrease requests) are presented here. These results are enough to reach to the conclusion. Figure 7 and Figure 8 show the experimental results achieved by the proposed technique compared to GA in term of solution cost and the number of allocation changes required to revise scheduling plan. From the presented results, our analysis and findings are as follows:

- In term of execution cost, GA with each request tries to find sub-optimal solution by generating a new plan. While the proposed technique just revises the current plan quickly, which may not lead to sub-optimal solution.
- In terms of the number of allocation changes for a velocity change request, the proposed technique



(a) Solution cost (b) Number of changes

Figure 7. Quality of solution for different workflow structures (medium velocity increase range)



(a) Solution cost (b) Number of changes

Figure 8. Quality of solution for different workflow structures (medium velocity decrease range)

reduces these changes in comparison to GA by at least 90.41% with increase request. While with decrease request, the proposed technique reduces these changes by at least 86.11% in comparison to GA. This is because it quickly adjusts the current scheduling plan instead of generating a completely new scheduling plan to respond to the velocity change request. In contrast, GA generates a new scheduling plan in both velocity changes. This is not only incurring more computational time but also requires a lot of VM changes to deprovision those VMs that are not in the new plan and to provision those that are in the new plan. To maintain the continuity of processing incoming streams at current data rates, unneeded VMs from the old plan must remain in use until the new VMs become ready. This causes further overhead in execution time and additional cost as both new VMs and the current VMs (that will be deprovisioned later on) are remaining in the resource pool. This also incurs more processing delays for upcoming streams when velocity change is a velocity increase request or more provisioning cost when the change is a velocity decrease request.

- From the results of the number of allocation changes presented in Figure 7b and Figure 8b, we can notice that the proposed technique achieved the most performance gain (i.e. the number of allocation changes is reduced by 99.65%) with Inspiral_100 with increase request. The reason behind that is the structure of this workflow processes huge amounts of data compared with other workflows, resulting in more computing powers are required. Thus, generating a new plan is too expensive and incurs a large number of allocation changes, while revising the existing plan incurs small number of allocation changes that leads to huge performance gain.
- Based on the presented performance matrix (Figure 2), the proposed technique achieved a competent performance with high quality of solution besides good execution cost compared to GA with most workflows. It also achieved a non-competitive number of allocation changes required to revise the scheduling plan, and little or negligible execution time. Therefore, the proposed technique outforms GA with different workflow structures.

7 Conclusion and Future Work

In this paper, we considered the dynamic scheduling problem of stream workflow application on various Cloud infrastructures. These infrastructures forming a multicloud environment, which becomes the dynamic execution environment for these applications. To this end, we proposed a new dynamic scheduling and provisioning technique that incorporates GA and two-level greedy algorithm to efficiently schedule stream workflow application in multicloud environment while meeting real time user performance

constraints under velocity changes with minimal execution cost. The experimental results showed that the proposed technique outperformed competitors in responding to data velocity changes at runtime while reducing the total execution cost for all modelled workflow applications under various data velocity ranges. It is also closer to lower bound in comparison to its compositors (Baseline and GA).

For future study, this paper reveals two new directions to enhance the performance and capability of the proposed dynamic scheduling technique. The first direction is aiming to parallelise Minimax with Alpha-Beta pruning algorithm to reduce running cost and achieve speedup. The second direction is to support more dynamism forms for stream workflow applications such as application structure and real-time data processing requirement, where coping with these changes at runtime enables the full dynamic support.

ACKNOWLEDGMENT

This research is supported by an Australian Government Research Training Program (RTP) Scholarship.

REFERENCES

[1] A. Zanella *et al.*, "Internet of things for smart cities," *IEEE Internet of Things journal*, vol. 1, no. 1, pp. 22–32, 2014.
[2] Y. Mehmood *et al.*, "Internet-of-things-based smart cities: Recent advances and challenges," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 16–24, 2017.
[3] C. Chen *et al.*, "Connected vehicular transportation: Data analytics and traffic-dependent networking," *IEEE Vehicular Technology Magazine*, vol. 12, no. 3, pp. 42–54, 2017.
[4] D. Redlich *et al.*, "Research challenges for business process models at run-time," in *Models@ run. time*. Springer, 2014, pp. 208–236.
[5] J. Liu *et al.*, "A survey of data-intensive scientific workflow management," *J. of Grid Computing*, vol. 13, no. 4, pp. 457–493, 2015.
[6] J. Wang *et al.*, "Kepler+ hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems," in *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*. ACM, 2009, p. 12.
[7] ——, "Big data applications using workflows for data parallel computing," *Computing in Science & Engineering*, vol. 16, no. 4, 2014.
[8] V. Vavilapall *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.
[9] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11.
[10] D. Sun, G. Zhang, S. Yang, W. Zheng, S. U. Khan, and K. Li, "Re-stream: Real-time and energy-efficient resource scheduling in big data stream computing environments," *Information Sciences*, vol. 319, pp. 92–112, 2015.
[11] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3553–3569, 2017.
[12] A. Bożek and F. Werner, "Flexible job shop scheduling with lot streaming and sublot size optimisation," *Int. J. of Production Research*, vol. 56, no. 19, pp. 6391–6411, 2018.
[13] D. Sun and R. Huang, "A stable online scheduling strategy for real-time stream computing over fluctuating big data streams," *IEEE Access*, vol. 4, pp. 8593–8607, 2016.
[14] D. Sun *et al.*, "Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams," *J. of Supercomputing*, vol. 74, no. 2, pp. 615–636, 2018.
[15] L. Chen, S. Liu, B. Li, and B. Li, "Scheduling jobs across geo-distributed datacenters with max-min fairness," *IEEE Transactions on Network Science and Engineering*, 2018.
[16] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling tasks closer to data across geo-distributed datacenters," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.
[17] H. Chen *et al.*, "Big data processing workflows oriented real-time scheduling algorithm using task-duplication in geo-distributed clouds," *IEEE Transactions on Big Data*, 2018.
[18] M. Barika *et al.*, "Scheduling algorithms for efficient execution of stream workflow applications in multicloud environments," *IEEE Transactions on Services Computing, In press*.
[19] F. Teng, "Scheduling real-time workflow on mapreduce-based cloud," in *Third International Conference on Innovative Computing Technology (INTECH 2013)*. IEEE, 2013, pp. 117–122.
[20] Y. Wang and W. Shi, "Budget-driven scheduling algorithms for batches of mapreduce jobs in heterogeneous clouds," *IEEE Transactions on Cloud Computing*, vol. 2, no. 3, pp. 306–319, 2014.
[21] T. Shu and C. Wu, "Performance optimization of hadoop workflows in public clouds through adaptive task partitioning," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.
[22] X. Zeng *et al.*, "Sla-aware scheduling of map-reduce applications on public clouds," in *2016 IEEE 18th International Conference on High Performance Computing and Communications (HPCC 2016)*. IEEE, 2016, pp. 655–662.
[23] ——, "Cost efficient scheduling of mapreduce applications on public clouds," *J. of computational science*, vol. 26, pp. 375–388, 2018.
[24] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Presented as part of the*, 2012.
[25] R. Ranjan *et al.*, "Orchestrating bigdata analysis workflows," *IEEE Cloud Computing*, vol. 4, no. 3, pp. 20–28, 2017.
[26] S. Martello, "An algorithm for the generalized assignment problem," *Operational research*, pp. 589–603, 1981.
[27] S. Yang, "Genetic algorithms with memory-and elitism-based immigrants in dynamic environments," *Evolutionary Computation*, vol. 16, no. 3, pp. 385–416, 2008.
[28] D. Dyer. (2010) Watchmaker framework for evolutionary computation. [Online]. Available: https://watchmaker.uncommons.org/
[29] K. Collins. (2015) When to use containers or virtual machines, and why. [Online]. Available: https://www.nextplatform.com/2015/08/06/containers-versus-virtual-machines-when-to-use-each-one-and-why/
[30] Q. Hassan *et al.*, *Internet of Things: Challenges, Advances, and Applications*. CRC Press, 2017.
[31] M. Barika *et al.*, "Iotsim-stream: Modelling stream graph application in cloud simulation," *Future Generation Computer Systems*, vol. 99, pp. 86–105, 2019.

Mutaz Barika has PhD in Information Technology from the School of Technology, Environments and Design at University of Tasmania. He was awarded an Australian Government Research Training Program (RTP) Scholarship for pursing his PhD studies. He completed his MSc. and BSc. in Computer Science from King Saud University and University of Petra respectively His current research interests include Big Data, Big Data Workflow, IoT, Cloud Computing and Data Security.

Saurabh Garg is a Senior Lecturer at the University of Tasmania, Tasmania. He is one of the few Ph.D. students who completed in less than three years from the University of Melbourne in 2010. He has gained about three years of experience in the Industrial Research while working at IBM Research Australia and India. His area of interests are Distributed Computing, Cloud Computing, HPC, IoT, BigData analytics, and education analytics.

Albert Y. Zomaya is a Chair Professor and Director of the Centre for Distributed and High Performance Computing at Sydney University. He has published more than 500 scientific papers and is an author, co-author, or editor of more than 20 books. He is the Editor-in-Chief of IEEE Transactions on Sustainable Computing and serves as an Associate Editor for 22 leading journals. He is a chartered engineer, a fellow of the AAAS, the IEEE, the IET (UK), and an IEEE Computer Society Golden Core member.

Rajiv Ranjan is a Full professor in Computing Science at Newcastle University, United Kingdom. Before moving to Newcastle University, he was Julius Fellow (2013-2015), Senior Research Scientist and Project Leader in the Digital Productivity and Services Flagship of Commonwealth Scientific and Industrial Research Organization (CSIRO) Australian Governments Premier Research Agency). Prior to that he was a Senior Research Associate (Lecturer level B) in the School of Computer Science and Engineering, University of New South Wales (UNSW). Dr. Ranjan has a PhD (2009) from the department of Computer Science and Software Engineering, the University of Melbourne.