Future Generation Computer Systems 112 (2020) 348–361

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

# Cost effective stream workflow scheduling to handle application structural changes

Mutaz Barika [a,*], Saurabh Garg [a], Rajiv Ranjan [b]

[a] Discipline of ICT — School of Technology, Environments and Design (TED), University of Tasmania, Hobart, Tasmania, Australia
[b] School of Computing, Newcastle University, Newcastle upon Tyne, United Kingdom

ARTICLE INFO

*Article history:*
Received 21 January 2020
Received in revised form 10 April 2020
Accepted 20 May 2020
Available online 22 May 2020

*MSC:*
00-01
99-00

*Keywords:*
IoT
Stream workflow
Dynamic scheduling
Pluggable technique
Cloud computing

ABSTRACT

Stream workflow is a network of big data streaming applications that acts as key enabler for real-time analysis from Internet of Things data. Smart traffic management and smart grid are examples of stream workflow. The focus of existing work is on streaming operator graphs which differs from stream workflow and handling data fluctuations without significant consideration of different dynamic forms that could happen in the structure of data pipelines. This paper investigates the scheduling problem of stream workflow to support runtime alterations of stream workflow deployment, so that scheduling plans will be revised to handle stream workflow applications with continuously changing characteristics. It proposes a pluggable dynamic scheduling technique that accepts user-defined algorithms to handle stream workflow runtime changes. It also presents three different plug-in algorithms and methods to enable auto-scaling of this workflow in a Multicloud environment. The experimental results of the quality of the solution showed that the proposed plug-in optimisation technique is more efficient than baseline and dynamic fair-share techniques to handle runtime changes.


## 1. Introduction

The current advances in the Internet of Things (IoT) ecosystem [1,2] increase the interest in developing applications and services that aim to analyse data generated from IoT devices to obtain real-time analytics. In addition to the need for stream processing, integrating multiple analytical components into a data analysis pipeline forming a workflow is necessary for future smart applications [3] such as smart retail [4] and smart transportation. In stream processing, an operator graph is provided to represent a data pipeline that contains a set of operators on data streams, where the whole graph has one feeding source and one end operator. This graph can be generated by several streaming systems such as Apache Flink and Storm, to extract fresh data analytics from an infinite data sequence. However, a more complex data pipeline that is highly dynamic in nature involves analytical components that have different user and infrastructure requirements, has multiple data sources that feed their data into any analytical components and has multiple outputs. This type of workflow is called a stream workflow application.

With stream workflow, the fluctuation of data velocity is not the whole story of dynamism. This adaptive workflow is gradually more complex as its active analytical components can be adjusted over time according to the changes in a user-specific scenario and runtime environment. For instance, when vehicle environment or traffic conditions are changed in a smart traffic control service, a new analytical component may be added to the in-progress workflow, or an existing analytical component might be changed or deleted, to respond to the changes. Therefore, it is clear that these kinds of runtime changes are causing structural amendments in workflow application and even data velocity change.

The focus of existing research works in the literature (such as Liu et al. [5], Liu and Buyya [6], Kombi et al. [7], Sun and Huang [8] and Sun et al. [9]) is on streaming operator graphs and these works mainly handle the fluctuation of data stream velocity and adjust resources to meet the needs of data processing. Also, some of these works consider the availability of computational resources or guaranteeing makespan while others offer big data orchestrators (Apache YARN [10] and Apache Mesos [11]) that do not need to deal with the dynamism of stream workflow applications and meet real-time user requirements. However, the existing research gap in this field of study is to support the dynamic-nature of analytical components involved in stream workflow and deliver dynamic scaling to improve performance under structural and non-structural changes of this workflow.

The aforementioned research gap drives us to investigate the dynamic scheduling of a complex stream workflow for tackling its dynamic aspects at runtime, so that the stability of this application is maintained and achieved over time. To fill this gap, we

* Corresponding author.
*E-mail addresses:* mutaz.barika@utas.edu.au (M. Barika), saurabh.garg@utas.edu.au (S. Garg), raj.ranjan@ncl.ac.uk (R. Ranjan).

https://doi.org/10.1016/j.future.2020.05.036
0167-739X/

propose a fully-pluggable dynamic scheduling and provisioning technique that supports dynamic scaling by managing computing resources at runtime to respond to structural and non-structural changes of stream workflows in order to maintain real-time data analysis requirements. It amends the current scheduling plan according to runtime changes by using the plugged-in algorithms and methods that users define to always maintain application stability. This proposed technique is considered as a future scheduling module in stream workflow management systems, where its significance comes from the flexibility provided to handle both structural and non-structural changes of in-progress workflows. In summary, our contributions are:

- Structural change model for stream workflow.
- Pluggable dynamic scheduling technique that incorporates user-defined algorithms to handle stream workflow runtime changes. It provides a hassle-free way to have an elastic scheduling technique to handle different dynamic forms of stream workflow by simply implementing heuristic decisions.
- Three heuristic methods that can be plugged in separately into the pluggable dynamic scheduling technique to produce three different elastic scheduling techniques to handle dynamic forms of stream workflow.

## 2. Related work

IoT and cloud computing technologies are interrelated and complement each other. The former enables connected real-world objects (also referred to as IoT devices) to exchange real-time data with other connected objects over the Internet, while the latter provides a distributed, dynamic and on-demand environment to store, process and manage the data generated from IoT devices over the Internet. The related works can be categorised into two parts. The first part generally focuses on IoT and cloud computing technologies including their applications and services as well as the relevant IT trends. The second part specifically focuses on big data processing.

For IoT, research works such as Liu et al. [12], Al-Fuqaha et al. [13] and Li et al. [14] survey the landscape of IoT technology, while other works investigates specific integrated techniques like Li et al. [15] focusing on 5G networks into IoT and Sharma et al. [16] focusing on the integration of communication technologies (i.e. energy harvesting wireless sensor network and hybrid LiFi/WiFi). Also, Zhang and Chen [17] presented the state-of-the-art survey for IoT and other emerging technologies (Industry 4.0, blockchain and business analytics), while Kim [18] explored the Cyber–Physical System (CPS) technologies and their relevance to different emerging IT trends such as IoT, Industry 4.0, big data and cloud computing. By considering the importance of on-demand delivery of unlimited resources in a distributed and dynamic manner, IoT is coupled with cloud computing. This integration allows building of cloud-based IoT applications and services, which can be implemented in different sectors and industries. In the manufacturing sector, for example, Wang et al. [19] discussed the influence of IoT and cloud computing technologies in this sector from the perspective of assembly planning for complex products and then proposed an automated assembly modelling system. In the same sector but from a resource sharing perspective, Xie et al. [20] proposed a semantic resource service model for sharing data-oriented manufacturing resources in the cloud. Also, Bi and Cochran [21] discussed big data analytics applications in cloud manufacturing in order to emphasise the need for intelligent manufacturing systems. For storing IoT data that is huge in volume, rapid in generation rate and heterogeneous in types, Jiang et al. [22] proposed a new data management framework. This framework enables the storing and accessing of such data efficiently in cloud computing environments. In the context of Quality of Service (QoS) and Service Level Agreement (SLA), Zheng et al. [23] investigated QoS for general cloud services and proposed a new quality model consists of different quality dimensions (i.e. usability, availability, reliability, responsiveness, security, and elasticity), while Zheng et al. [24] discussed SLA negotiation for these services and proposed a new mixed negotiation mechanism that utilises concession and tradeoff strategies in order to balance utility and success rates.

With the focus on big data processing, this processing can be divided into two main categories, which are batch processing (to process complex and high volumes of data at once to gain insights) [25] and stream processing (to process infinite data as they arrive and produce incremental insights) [26,27]. As we investigate the efficient execution of dynamic stream workflow applications, the scope of this paper is on stream (real-time) processing. To deploy streaming big data workflow applications, a scheduling problem is raised. Scheduling methods or techniques can be divided into two based on scheduling time and the availability of information, i.e. static (at deployment time) and dynamic (at runtime).

With a static scheduling problem, resource provisioning and scheduling plan is generated at deployment time as all information is known prior to the execution of workflow application beginning. Cardellini [28] presented an optimal replication and placement scheduler for data stream processing applications that takes into consideration the heterogeneity of application requirements and computing resources. Barika et al. [29] presented two static scheduling algorithms for stream workflow, which are greedy and genetic algorithms. These algorithms take into account user-defined QoS requirements and different cloud resources offered by multiple clouds at different costs while minimising the total execution cost including provisioning and data transfer costs.

With a dynamic scheduling problem, the scheduling decisions are made at runtime as the information is updated according to the occurrence of dynamic changes. Liu et al. [5] proposed a runtime-aware dynamic scheduling technique that considers the fluctuation of input data rate and the variation in the availability of computational resource for a streaming operator graph. This technique redistributes the operator's tasks at runtime to minimise latency requirements including operator processing latency and the latency difference between different worker nodes. Liu and Buyya [6] proposed a dynamic resource-aware scheduling algorithm for stream applications and validated its efficiency by implementing a prototype scheduler named D-Storm on top of Apache Storm. This scheduler adjusts a scheduling plan under various sizes of inputs. Kombi et al. [7] proposed DABS-Storm, a data-aware approach that handles the fluctuations of input data streams (in terms of data volume and distribution) in a streaming operator graph and adjusts resources to meet the needs of data processing. However, the aforementioned research works consider streaming operator graphs, which is different from stream workflow as well as they lack the ability to handle application-level changes that may occur at runtime. Additionally, in our previous work [30], we proposed a dynamic scheduling technique for stream workflow, but the runtime change that is handled by this technique is limited to data fluctuations.

In the same context of dynamic scheduling problem but with considering a single cloud infrastructure as an execution environment, Sun and Huang [8] proposed a Stable Online Scheduling Strategy (SOMG). This strategy addresses the problem of system stability for real-time data processing over stream flow fluctuations while guaranteeing makespan. Sun et al. [9] proposed an elastic online scheduling method that aims to achieve fairness scheduling of multiple big data streaming applications

while guaranteeing makespan. However, these research works do not take into consideration that stream workflow is a workflow of workflows and its structure could be changed at runtime (i.e. various application structural changes may occur during the execution of this workflow).

For scheduling methods in big data application orchestrators (i.e. Apache YARN [10] and Apache Mesos [11]), these methods implemented a fair-share model. In Apache YARN, the default fair scheduling method equally shares cluster resources among applications over time. In Apache Mesos, the default scheduling decision used by the master process to determine how resources will be assigned to each framework is the Dominant Resource Fairness algorithm (fair-share model to multiple resource types). However, these scheduling methods do not take into consideration user-defined real-time requirements and different dynamic forms at application-level that may occur at runtime as well as the unpredictable performance of such workflow applications.

Accordingly, the aforementioned scheduling techniques do not consider that stream workflow has different dynamic forms at application-level, so that its structure could be changed at runtime. Also, they do not utilise the capability of 'cloud of clouds' as a dynamic and heterogeneous execution environment.

## 3. Problem definition and modelling

Since road traffic is under strain with the continued increase of the number of vehicles and population, smart road traffic monitoring as a service of smart city services can utilise the true power of connected vehicles in addition to roadside infrastructure (e.g. traffic lights, cameras). Collecting and analysing the streaming data generated by these vehicles allows us to create a real-time view of road traffic and incidents, and recommend runtime adjustments based on live traffic events and road conditions (e.g. average running speed, traffic density [31]). Fig. 1 shows an exemplar workflow of this real use case, which is more comprehensive than the one presented in our previous work [30]. The description and requirements of dynamic stream workflow are provided in our previous work [30].

This instant feedback use case shows the growing importance and value of real-time analytical insights in the future of smart city services (here road traffic monitoring service as a real-world application). Such service application is a real-world dynamic big data pipeline that uses sensor data from connected vehicles, uploads such data to cloud datacenters for analysis, and produces a real-time view of road traffic as continuous insights. From this application use case, we can outline the following forms of dynamism:

- Velocity of Streaming Data — As smart city is a dynamic environment, the speed of streaming data changes greatly based on time or traffic alert. As an example of the variation, during peak hour traffic, a large number of connected vehicles operate on the road transmitting their data, whilst at night time, few vehicles operate [31]. Therefore, Dynamic Form 1 is a runtime action to change the velocity of streaming data, either increase or decrease as a consequence of the change happening via external source (i.e. data rate is changed) or parent service (i.e. the velocity of output data is changed).
- Real-time Data Processing Requirement of Connected Vehicle — The data processing requirement for analytical components may change overtime, reflecting the complexity of computations that will be carried out on data (from simple to complex aggregation functions and vice versa). This will affect the computing power needed according to the changes in data processing requirement. Therefore, Dynamic Form 2 is a runtime action to change the existing service by amending either its data processing requirements (Case 1) or the velocity of output stream (Case 2).
- Structure of Application — In smart city, the analysis requirements of an application change over time to reflect new amendments to control and/or data flows. Thus, active connected vehicles and/or existing nodes may be connected to the nodes being added or their communications may be cut off from the nodes being deleted. According to the aforementioned changes, the structure of this application becomes dynamic which means the requirements of resources will vary as application structure changes. Accordingly, we have two dynamic forms for the structure of application. Dynamic Form 3 is a runtime action to amend the structure of application by adding a new service. With this action, there are five cases that will be discussed in this section. Dynamic Form 4 is an runtime action to amend the structure of application by deleting an existing service. With this action, there are two cases that will be discussed in this section.

Accordingly, the execution of a stream workflow application is crucial. The dynamic scaling of stream workflow application should be treated carefully to tackle these changes while achieving real-time performance requirements. For our problem modelling here, we consider single service change at any instant of time. In other words, we assume that only one runtime change can be made at any instant of time and such a change request requires one response action. We present here a structural change model that is an extension to our previous problem modelling presented in [30].

### 3.1. Dynamic Form 1: Change the streaming data velocity

In our previous work [30], we proposed a two-phase adaptive scheduling technique to efficiently reschedule dynamic workflow application in cloud infrastructures to respond to changes in the velocity of data at runtime. The details of this technique and how it was used to handle such dynamic forms are provided in that paper.

### 3.2. Dynamic Form 2: Change of existing service

Considering stream application as an adaptive workflow, any service (analytical component) may be changed over time. This kind of dynamic form occurs as a change in data processing requirement for a service $MI^{S_n}$ or velocity of output data stream for a service $outStream(S_n)$ (i.e. as a consequence of changing output proportion $\gamma^{S_n}$).

**Case 1: Change data processing requirement of existing service.** This change happens in the real world when a new version of existing service is available and is deployed in place of the current version. For instance, in the presented Fig. 1, the new release of traffic analysis service is available. With this case, the user-provided input and impact are as follows:
**User input:** The service $S_n$ selected, and the new value for $MI^{S_n}$ denoted as $nMI^{S_n}$.
**Impact:** The $MI^{S_n}$ and $pro(S_n)$ are updated as follows:

$$MI^{S_n} = nMI^{S_n}$$
$$pro(S_n) = \begin{cases} pro(S_n) + exVM(S_n), \text{ if } MI^{S_n} \text{ increases} \\ pro(S_n) - rmVM(S_n), \text{ otherwise} \end{cases} \quad (1)$$
$$\text{where } MIPS_v \geq unitDPRate * MI^{S_n} |\ v \in pro(S_n)$$

*and data processing constraint is maintained* (*Eq. 4 in* [30]).

**Case 2: Change output stream velocity of existing service.** This change happens in the real world when a new release of service is deployed in place of the current version, which produces

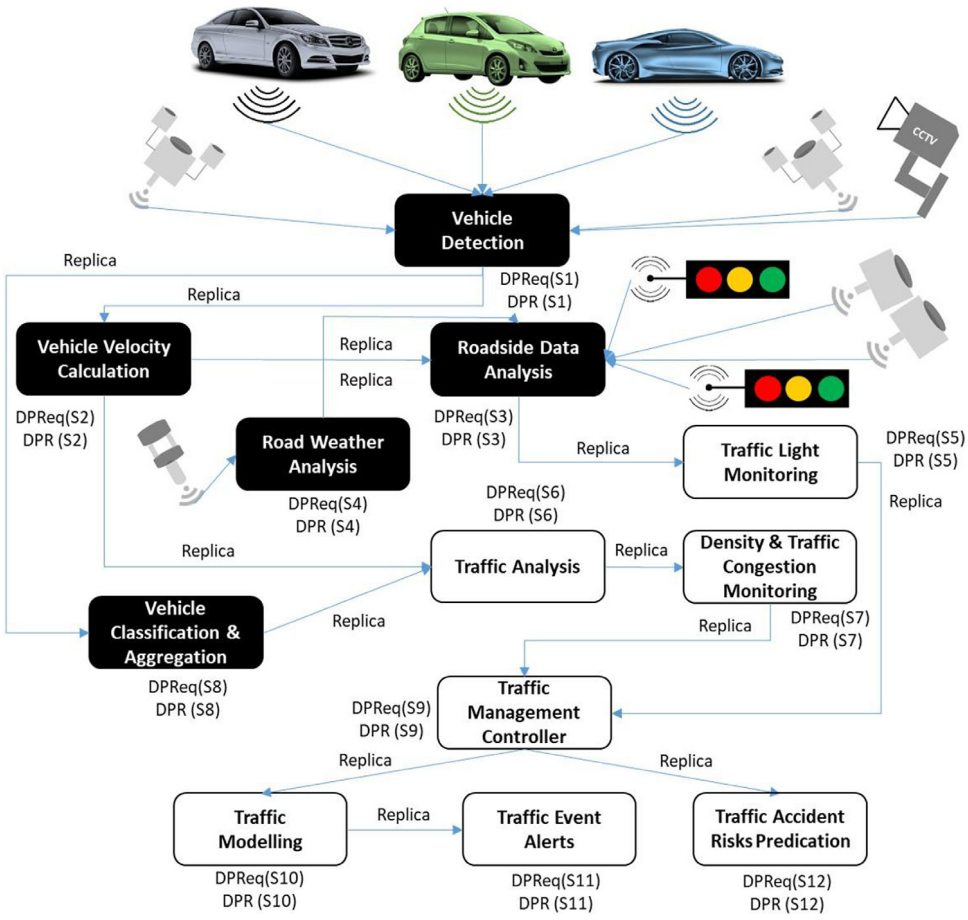



**Fig. 1.** Exemplar workflow for real-time view of road traffic and incidents.

less or more output data from the processed input data based on the new processing logic. For instance, in Fig. 1, the new release of traffic analysis service is being deployed that changes the velocity of output stream sp that it either increases or decreases, so that downstream services (density and traffic congestion monitoring, traffic management controller, traffic modelling, traffic event alerts and traffic accident risks predication services) receive more or less streaming data. With this case, the following are user-provided input and impact:

**User input:** The service $S_n$ selected, and the new value for $\gamma^{S_n}$ denoted as $n\_\gamma^{S_n}$.

**Impact:** The $\gamma^{S_n}$ and $pro(S_n)$ are updated as follows:

$$\vartheta^{S_n} = \begin{cases} (n\_\gamma^{S_n} - \gamma^{S_n}) * inStream(S_n), \text{ if } \gamma^{S_n} \text{ increases} \\ (\gamma^{S_n} - n\_\gamma^{S_n}) * inStream(S_n), \text{ otherwise} \end{cases}$$

$$\vartheta^{S_n} = \lceil \vartheta^{S_n} / minDPUnit \rceil * minDPUnit$$

$$\gamma^{S_n} = n\_\gamma^{S_n}$$

$$outStream(S_n) = \gamma^{S_n} * inStream(S_n)$$

*For each downstream service affected from this runtime change, perform Eq. 6 and Eq. 7 in* [30]. (2)

### 3.3. Dynamic Form 3: Add a new service

Adding a new service to a stream workflow application at runtime means performing a change in the structure of the application. This change can be in various forms depending on where a new service is being added to the stream workflow, what is/are the input link(s) for such new services and where the output stream of this new service is being routed. Considering these aspects, we support five different cases under this dynamic form, where each one of them has its own input requirements and consequences. Nevertheless, the common impact of all cases is the creation of a new service $S^*$ with its type, data processing requirement $MI^{S^*}$, placement cloud $c_g^{S^*}$ and output data proportion $\gamma^{S^*}$, and the update of S set as follows:

$$S^* = (MI^{S^*}, 0, \gamma^{S^*})$$
$$S = S^* \cup S \quad (3)$$

**Case 1: Add a new service with input from service and output to sink.** This change happens when there is a need to process the output of a parent service further, and provides new analytical insight or performs verification on the result by repeating the processing. For instance, in the presented Fig. 1, traffic optimisation or weather prediction service can be added based on the additional processing of input streaming data from a traffic management controller service. With this case, the following are user-provided input and impact:

**User input:** A new service $S^*$ as well as the input link from parent service $S_n$ to new service $S^*$.

**Impact:** The $S^*$ and $e_{m+1}$ are created, and S and E sets are updated as follows:

*Eq. (3) is applied*
$$e_{m+1} = (S_n, S^*, 100\%)$$
$$E = e_{m+1} \cup E \quad (4)$$

**Case 2: Add a new service with input from two or more sources (external source and/or parent service) and output**

**to sink.** This change happens when a new service is needed to process streaming data combined from several sources and produces new analytical insight. For example, in the presented Fig. 1, traffic prediction service that is based on inputs from traffic management controller (dynamic flows) and traffic modelling (static flows) can be added. With this case, the following are user-provided input and impact:

**User input:** A new service $S^*$ as well as $SS(S^*)$ denotes a set of J input sources for $S^*$, which is a subset of sets EX and S, $SS(S^*) \subseteq EX \cup S$. Thus, each source $x_j$ of set $SS(S^*)$ for $j = 1, 2, \ldots, n$ is either external source $x_j \in EX$ or parent service $x_j \in S$.

**Impact:** The $S^*$ is created, and S and E sets are updated as follows:

$$\begin{aligned} &Eq.\ (3)\ is\ applied \\ &\forall x_j \in SS(S^*),\ e_{m+j} = (x_j, S^*, 100\%) \\ &E = e_{m+j} \cup E \end{aligned} \tag{5}$$

**Case 3: Add a new service with input from external source and output to one service.** This change happens when streaming data from external sources can be processed with different logics to provide additional analytics that improve the data analysis performed by an existing service. It also could happen when data preprocessing analytical component/service is required for streaming data coming from an existing external source prior to carrying out data analysis processing on such data by an existing service. For example, in Fig. 1, a road weather data preprocessing service is needed to perform major data preprocessing steps on road weather sensor data before road weather analysis in order to significantly reduce data processing time and cost. With this case, the following are user-provided input and impact:

**User input:** A new service $S^*$ as well as one of the existing services is selected as a destination service $S_n$ such that this service has input from an external source and output to one service. Let $EX_p$ be the input source for $S_n$ where $e_m = (EX_p, S_n, 100\%)$ exists.

**Impact:** The $S^*$ and two new edges $e_{m+1}$&$e_{m+2}$ are created, $e_m$ is deleted, and S and E sets are updated as follows:

$$\begin{aligned} &Eq.\ (3)\ is\ applied \\ &E = E - e_m \\ &e_{m+1} = (EX_p, S^*, 100\%) \\ &e_{m+2} = (S^*, S_n, 100\%) \\ &E = e_{m+1} \cup e_{m+2} \cup E \\ &\forall x \in S\ where\ x\ is\ affected\ by\ this\ change,\ inStream(x), \\ &outStream(x)\ and\ pro(x)\ are\ updated\ according\ to \\ &Eq.\ 6\ and\ Eq.\ 7\ in\ [30]. \end{aligned} \tag{6}$$

**Case 4: Add a new service with input from service and output to one service.** This change happens when input streaming data of an existing service should be preprocessed before carrying out data analysis processing at this service, so that a new data preprocessing analytical component/service before the existing service is needed. For example, in Fig. 1, a new service to transform streaming data from the vehicle detection service is needed prior to being classified and aggregated by the vehicle classification and aggregation service. With this case, the following are user-provided input and impact:

**User input:** A new service $S^*$ as well as one of the existing services is selected as a destination service $S_n$ such that this service has input from one service and output to one or more services. Let $S_{n-1}$ be the input source (parent service) for $S_n$ where $e_m = (S_{n-1}, S_n, 100\%)$ exists.

**Impact:** The $S^*$ and two new edges $e_{m+1}$&$e_{m+2}$ are created, $e_m$ is deleted, and S and E sets are updated as follows:

$$\begin{aligned} &Eq.\ (3)\ is\ applied \\ &E = E - e_m \\ &e_{m+1} = (S_{n-1}, S^*, 100\%) \\ &e_{m+2} = (S^*, S_n, 100\%) \\ &E = e_{m+1} \cup e_{m+2} \cup E \\ &\forall x \in S\ where\ x\ is\ affected\ by\ this\ change,\ inStream(x), \\ &outStream(x)\ and\ pro(x)\ are\ updated\ according\ to \\ &Eq.\ 6\ and\ Eq.\ 7\ in\ [30]. \end{aligned} \tag{7}$$

**Case 5: Add a new service with input from two or more sources and output to one service.** This change happens when input streaming data of an existing service needs to be preprocessed and then enriched with additional stream output sources, so that data analysis processing at this service will be carried-out on the analytical result instead of the original streams. To apply this change, a new data analytical component/service before the existing service is necessary. For example, in Fig. 1, input data regarding vehicle velocity classification and aggregation service from vehicle detection service is preprocessed and enriched with roadside cameras by using a new service. This new service performs filtering and error correction on such data, where the output stream of this service will then be injected into the velocity classification and aggregation service instead of the original stream from the vehicle detection service. With this case, the following are user-provided input and impact:

**User input:** A new service $S^*$, one of the existing services is selected as a destination service such that this service has input from one service and output to one or more services, and $SS(S^*)$ denotes a set of J input sources for $S^*$, which is a subset of sets EX and S excluding $S_n$ and $S_{n-1}$, $SS(S^*) \subseteq EX \cup S - \{S_n, S_{n-1}\}$. Thus, each source $x_j$ of set $SS(S^*)$ for $j = 1, 2, \ldots, n$ is either external source $x_j \in EX$ or parent service $x_j \in S$. Let $S_{n-1}$ be the input source (parent service) for $S_n$ where $e_m = (S_{n-1}, S_n, 100\%)$ exists.

**Impact:** The $S^*$ and new edges $e_{m+1\ldots(m+|SS(S^*)|)}$ are created, $e_m$ is deleted, and S and E sets are updated as follows:

$$\begin{aligned} &Eq.\ (3)\ is\ applied \\ &E = E - e_m \\ &SS(S^*) = SS(S^*) \cup S_{n-1} \\ &\forall x_j \in SS(S^*),\ e_{m+j} = (x_j, S^*, 100\%) \\ &E = e_{m+j} \cup E \\ &e_{m+1} = (S^*, S_n, 100\%) \\ &\forall x \in S\ where\ x\ is\ affected\ by\ this\ change,\ inStream(x), \\ &outStream(x)\ and\ pro(x)\ are\ updated\ according\ to \\ &Eq.\ 6\ and\ Eq.\ 7\ in\ [30]. \end{aligned} \tag{8}$$

### 3.4. Dynamic Form 4: Delete an existing service

Like adding a new service to a workflow application, deleting an existing service also changes the structure of this application. However, the deletion of service varies in cases depending on the output link(s) of this service, where in case of the output link(s) is not to sink, the sub-tree of this service will be impacted. Under this dynamic form, we support two cases for performing application structure change by means of deleting an existing service.

**Case 1: Delete a service with output to sink**. This case happens in the real world when one of the ending services in the

stream workflow application as an analytical component is no longer required in the data analysis pipeline. For instance, the traffic accident risks predication service in Fig. 1 is no longer wanted. With this case, the following are user-provided input and impact:

**User input:** Existing service $S_n$ selected. Let $SS(S_n)$ denote a set of J input sources for $S_n$, which is subset of sets EX and S excluding $S_n$, $SS(S_n) \subseteq EX \cup S - S_n$.

**Impact:** The $S_n$ is deleted, and S and E sets are updated as follows:

$$\begin{aligned} &\forall x_j \in SS(S_n), E = E - (x_j, S_n, 100\%) \\ &\qquad pro(S_n) = \emptyset \\ &S = S - S_n \end{aligned} \tag{9}$$

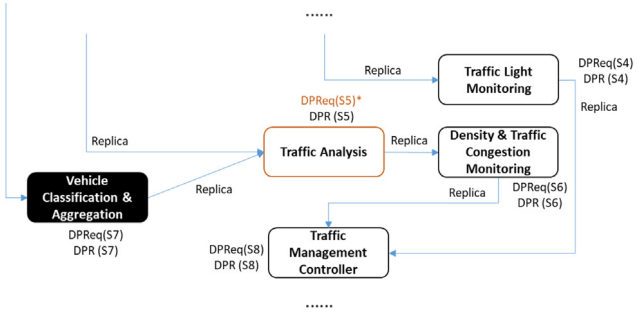
**Case 2: Delete service with output link(s) to one or more services** – This change happens when the analytical processing carried out by existing service and all subsequent analysis performed by services in the sub-tree of this service are not needed. For example, the traffic modelling service and its sub-tree in Fig. 1 may not be required as analytical components in the data pipeline. With this case, the following are user-provided input and impact:

**User input:** Existing service $S_n$ selected. Let $SS(S_n)$ denote a set of J input sources for $S_n$, which is subset of sets EX and S excluding $S_n$, $SS(S_n) \subseteq EX \cup S - S_n$, and $ST(S_n)$ denote a set of services in the sub-tree of $S_n$, where each child service $S_j$ of set $ST(S_n)$ for $j = 1, 2, \ldots, n$ is a descendant of the current tree service $S_n$.

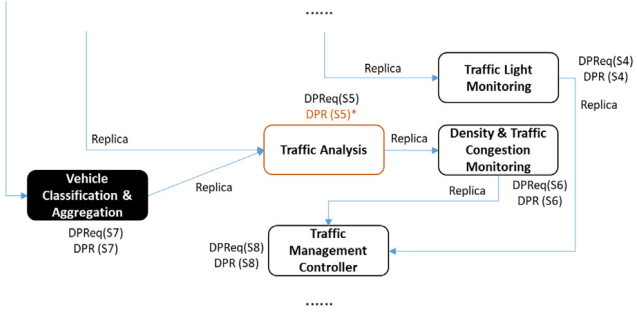
**Impact:** The $S_n$ is deleted, and S and E sets are updated as follows:

$$\begin{aligned} &\forall x_j \in SS(S_n), E = E - (x_j, S_n, 100\%) \\ &\qquad pro(S_n) = \emptyset \\ &\forall S_j \in ST(S_n), S = S - S_j \end{aligned} \tag{10}$$

Figs. 2 to 4 show the illustrative case examples of Dynamic Forms 2, 3 and 4 respectively.




(a) Case 1: Change data processing requirement of Traffic Analysis Service




(b) Case 2: Change output stream velocity of Traffic Analysis Service

**Fig. 2.** Illustration of Dynamic Form 2 with their cases after applying each case on Fig. 1.

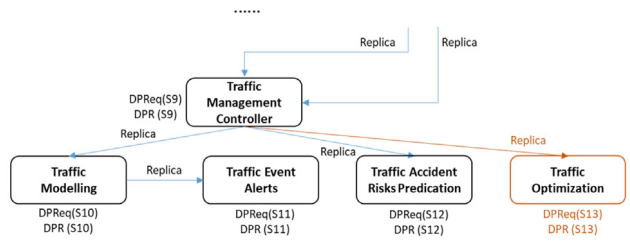
## 4. Proposed pluggable scheduling technique

For adaptive workflows like stream workflows, finding an optimal or near-optimal solution at deployment time is not the whole story. This is because the dynamic nature of these workflows cause different types of changes to occur at runtime. The problem of handling runtime changes of in-progress stream workflows needs to be investigated. In this paper, our problem is to reschedule stream workflow applications in cloud infrastructures to respond to different dynamic forms that occur at runtime. The revised scheduling plan should be generated as quickly as possible, be cost-effective, and maintain performance requirements. In other words, maintaining the quality of the revised scheduling solution is what matters here.

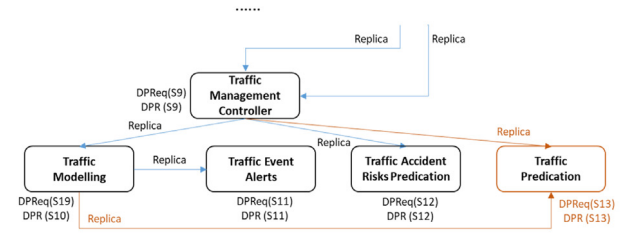
Considering various dynamic forms of stream workflows, each one of them requires a different type of response and this response should be efficient. Consequently, we propose a pluggable dynamic scheduling technique that supports runtime changes of in-progress stream workflows. It handles application-level changes during the execution of this workflow to always guarantee user-defined performance requirements while minimising the execution cost. The proposed technique copes with four dynamic forms with their different cases as described in our structural change modelling (see Section 3). The pseudocode of the proposed technique is presented in Algorithm 1. This algorithm at the beginning calls the plugged-in scheduling method to generate a scheduling plan for deploying the given stream workflow application and then waiting for the occurrence of runtime change. When such change happens, it calls the appropriate method to handle this change.

Algorithm 2 presents the pseudocode of the handling method that is used with Dynamic Form 2. This algorithm firstly checks whether the change event is an increase or decrease change. Then, it retrieves the service to be changed $S_n$ and calculates the change value based on the change percent from original value. After that, it checks the dynamic case, updates the original value and calls the plugged-in algorithm to amend the scheduling plan based on the dynamic case. It is worth noting that with increased change of data processing requirement for an existing service, the plugin algorithm must evaluate the computing power of the provisioned VM(s) if they are able to maintain the minimum data processing based on the updated data processing requirement for this service. This is because the absence of this check may lead to violating real-time data processing requirements for that service.

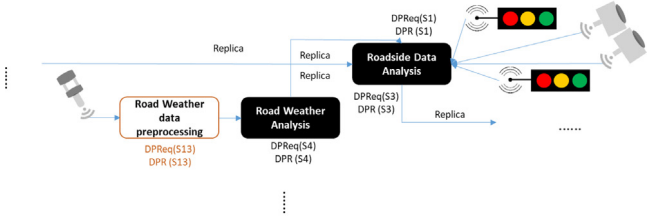
Algorithm 3 presents the pseudocode of the handling method that is used with Dynamic Form 3 Case 1 and Case 2, while Algorithm 4 presents the pseudocode of the handling method for the rest of Dynamic Form 3 cases (i.e. Case 3, Case 4 and Case 5). Both algorithms create a new service $S^*$ with its type, data processing requirement, placement cloud and output data proportion. But, the main difference between these algorithms is that the former adds a sink service, while the latter adds a non-sink service which has impact on downstream services. Thus, Algorithm 3 retrieves the input data sources selected for the service to be added, adds these input data sources based on the dynamic case to this service and updates parent–child relationships. It then adds the output stream and calls handleNewServiceChange method with only one affected service (i.e. the service to be added) in order to deploy it. Algorithm 4 firstly retrieves the destination service $S_n$, where the new service $S^*$ will be added before that service. Based on the dynamic case, this algorithm retrieves the input data sources selected for the service to be added and adds these input data sources to this service. Then, it updates the stream



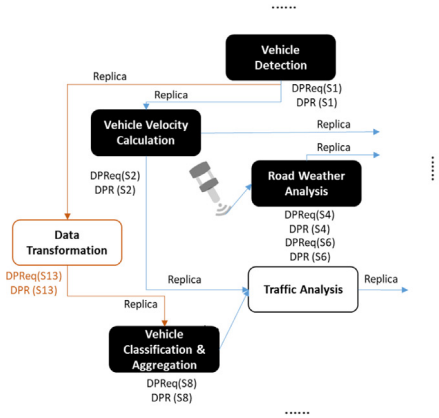

(a) Case 1: Add Traffic Optimisation Service



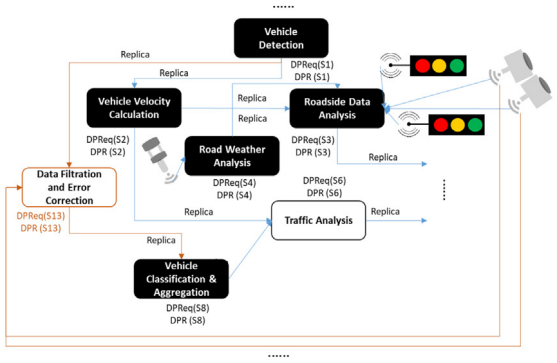

(b) Case 2: Add Traffic Predication Service




(c) Case 3: Add Road Weather Data Preprocessing Service

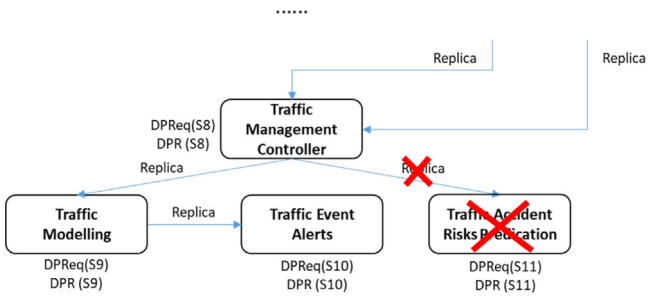



(d) Case 4: Add Data Transformation Service

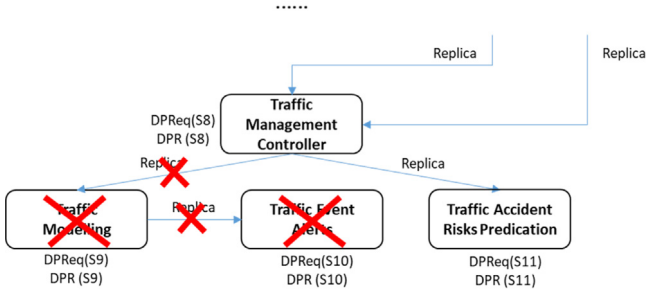



(e) Case 5: Add Data Filtration and Correction Service

**Fig. 3.** Illustration of Dynamic Form 3 with their cases after applying each case on Fig. 1.




(a) Case 1: Delete Traffic Accident Risks Predication Service




(b) Case 2: Delete Traffic Modelling Service

**Fig. 4.** Illustration of Dynamic Form 4 with their cases after applying each case on Fig. 1.

dependencies and parent–child relationships for input source of destination service $S_{n-1}$, new service $S^*$ and destination service $S_n$. After that, it retrieves the list of downstream services that will be affected by the addition of a new service. Lastly, it calls handleNewServiceChange method with this list to amend the scheduling plan.

Algorithm 5 presents the pseudocode of the method that is used to handle all cases of Dynamic Form 3. This algorithm first retrieves the service to be added $S^*$ and deploys this service using the plugged-in algorithm. Then, it calls the plugged-in algorithm with each service affected from runtime change to amend the provisioning plan of this service. After that, it retrieves those VMs that will be deprovisioned and those VMs that will be provisioned. Lastly, it performs provisioning and deprovisioning requests.

Algorithm 6 presents the pseudocode of the handling method that is used with Dynamic Form 4 Case 1. For Dynamic Form 4 Case 2, the handling method is presented in Algorithm 7. Both algorithms delete an existing service $S_n$, but the main difference between these algorithms is that the former removes a sink service which has impact on the parent service that could become a sink service, while the latter removes a non-sink service in which the sub-tree of this service should be removed. Algorithm 6 first retrieves the existing service to be deleted $S_n$. Then, it removes stream dependencies of this service and updates the child relationships of input data sources of this service. After that, it deprovisions VM(s) allocated to that service and lastly deletes the service. Similarly, Algorithm 7 performs the same actions, and in addition to the need to retrieve the sub-tree of $S_n$, it deletes those services in this sub-tree by deprovisioning their VMs, removing their stream dependencies and parent–child relationships.

**Algorithm 1** Pluggable Dynamic Scheduling Technique

call Scheduling Algorithm for deployment of stream workflow {a plugin algorithm that finds resource selection and scheduling solution at deployment time for executing stream workflow}
**for** each runtime change that occurs **do**
  *appChangeType* ← *get dynamic form*
  *appChangeCase* ← *get dynamic case in this dynamic form*
  **if** *appChangeType == 1* **then**
    call Velocity Change Response Algorithm {a plugin algorithm to revise the current scheduling plan to cope with data velocity changes}
  **else if** *appChangeType == 2* **then**
    call processExistingServiceChange(appChangeCase);
  **else if** *appChangeType == 3* **then**
    **if** *appChangeCase == 1 or appChangeCase == 2* **then**
      call newServiceChange_Case1&2(appChangeCase);
    **else if** *appChangeCase == 3 or appChangeCase == 4 or appChangeCase == 5* **then**
      call newServiceChange_Case3&4&5(appChangeCase);
    **end if**
  **else if** *appChangeType == 4* **then**
    **if** *appChangeCase == 1* **then**
      deleteService_Case1();
    **else if** *appChangeCase == 2* **then**
      deleteService_Case2();
    **end if**
  **end if**
**end for**

**Algorithm 2** processExistingServiceChange( appChangeCase)

*changeEvent* ← *get change event type* {increase or decrease}
$S_n$ ← *get the existing service that is selected*
get provisioned VMs for a service
*changePercent* ← *get increase/decrease percent from original value*
*value* ← *changePercent*/100
**if** *appChangeCase == 1* **then**
  **if** *changeEvent == 'increase'* **then**
    $MI^{S_n} = MI^{S_n} + (MI^{S_n} * value)$
  **else**
    $MI^{S_n} = MI^{S_n} * value$ $\{0.01 < value < 0.99\}$
  **end if**
  call Data Processing Requirement Change Response Algorithm with given service $S_n$ {a plugin algorithm that assesses each provisioned VM as still satisfying minimum data processing based on the updated $MI^{S_n}$ and provisioning more computing power if needed in case of increase event or deprovisioning the provisioned VMs that are not required to achieve the updated $MI^{S_n}$ in case of decrease event}
**end if**
**if** *appChangeCase == 2* **then**
  **if** *changeEvent == 'increase'* **then**
    $\gamma^{S_n} = \gamma^{S_n} + \gamma^{S_n} * value$)
    call Velocity Change Response Algorithm with increase event {a plugin algorithm to amend scheduling plan to handle data velocity changes}
  **else**
    $\gamma^{S_n} = \gamma^{S_n} * value$ $\{0.01 < value < 0.99\}$
    call Velocity Change Response Algorithm with decrease event {a plugin algorithm to revise scheduling plan to handle data velocity changes}
  **end if**
**end if**

**Algorithm 3** newServiceChange_Case1&2( appChangeCase )

Create service $S^*$
InputSources = $\phi$
Add input source(s) selected to $S^*$ in InputSources {Case 1: one input source, Case 2: two or more input sources; selection constraints are applied}
**for** each input source in InputSources **do**
  Add input dependency to $S^*$ from this source
**end for**
Update parent and child relationships for input source(s) and $S^*$
Add output stream for $S^*$ based on input stream dependency
Add $S^*$ to affectedSIDs
call handleNewServiceChange(affectedSIDs)

**Algorithm 4** newServiceChange_Case3&4&5(appChangeCase)

Create service $S^*$
$S_n$ ← get one of the existing services as a destination service {selection constraint is applied according to dynamic case}
InputSources = $\phi$
**if** *appChangeCase == 3 or appChangeCase == 4* **then**
  Add input source of $S_n$ (i.e. $S_{n-1}$) in InputSources
**else**
  Add input source(s) selected to $S^*$ in InputSources {input source of $S_n$ (i.e. $S_{n-1}$) + other input sources that are selected}
**end if**
**for** each input source in InputSources **do**
  Add input dependency to $S^*$ from this source
**end for**
Remove the dependency link of $S_n$
Add output stream for $S^*$ based on input stream dependency
Add dependency link for $S_n$ {source: $S^*$}
Update parent and child relationships between service(s) in InputSources and $S^*$, and between $S^*$ and $S_n$
affectedSIDs = get ids of services affected by adding request starting from $S^*$
call handleNewServiceChange(affectedSIDs)

**Algorithm 5** handleNewServiceChange(affectedSIDs)

$S^*$ ← get and remove the new service from affectedSIDs
call Resource Selection Algorithm for $S^*$ {a plugin algorithm that finds near-optimal resource selection solution for given service}
**if** affectedSIDs is not empty **then**
  call Velocity Change Response Algorithm with the list of affected services (affectedSIDs) {a plugin algorithm to revise the current scheduling plan to cope with data velocity changes for the list of services provided}
**end if**
*BeProVMs* ← get VMs that need to be provisioned $S^*$
*BeDeproVMs* ← get VMs that need to be deprovisioned $S^*$
**if** BeProVMs is not empty **then**
  provision VMs in the list
**else**
  **if** BeDeproVMs is not empty **then**
    deprovision VMs in the list
  **end if**
**end if**

## 5. Experiment setup and configuration

The aim of our experiments is to assess the quality of the response solution that is generated when a dynamic change happens at runtime. For this purpose, we simulate a Multicloud environment with the proposed pluggable dynamic scheduling technique using our simulator named IoTSim-Stream [32], which enables the execution of stream workflow applications in this environment. This simulation environment facilitates our evaluations

**Algorithm 6** deleteService_Case1()

1: $S_n \leftarrow$ get existing service that will be deleted {selection constraint is applied}
2: InputSources $\leftarrow$ get input sources of $S_n$
3: Remove dependency link(s) to $S_n$ from those sources in InputSources
4: Update child relationships for those sources in InputSources
5: Deprovision the provisioned VM(s) of $S_n$
6: Delete $S_n$

**Algorithm 7** deleteService_Case2()

1: $S_n \leftarrow$ get existing service that will be deleted
2: subtree $\leftarrow$ get services in sub-tree of $S_n$
3: **for** for each service in subtree **do**
4: Deprovision the provisioned VM(s) of this service
5: Remove this service with its dependency link(s) and parent-child relationships
6: **end for**
7: InputSources $\leftarrow$ get input sources of $S_n$
8: Remove dependency link(s) to $S_n$ from those sources in InputSources
9: Update child relationships for those sources in InputSources
10: Deprovision the provisioned VM(s) of $S_n$
11: Delete $S_n$

as we can compare experimental results obtained from different scheduling algorithms under the same environment conditions.

### 5.1. Workflow application and simulation environment

Prior to setting up a simulation environment, we need to look at different real structures for workflow applications to simulate stream workflows. Similar to our previous work [29], we use stream workflow applications modelled using common workflow structures (Montage, Inspiral, Epigenomics and CyberShake). For the execution environment, the modelled Multicloud environment presented in our previous work [29] is used. This Multicloud environment is made up of three different clouds, where each cloud offers different VM configurations. Additionally, a set of parameters for both workflow application and simulator should be configured to run our experiments. These parameters and their values are fixed for all scenarios and listed in Table 1 (more information in [29] and [30]).

### 5.2. Configuration changes in service data processing requirement

To model the amount of increase or decrease in service data processing requirement, we not only consider the percentage increase of the original value up to 100%, but we also add 50% as an additional margin to make it 150%. With the highest increase percentage (i.e. up to 150%), the data processing requirement for simple or medium aggregation function will be transformed to represent complex aggregation function. Thus, it is valuable to take into consideration the additional margin to increase the data processing requirement. We also need to note that the updated value of the data processing requirement after the increase request is capped at 4000 MI/MB, which is the maximum value of data processing requirement as listed in [29]. For decreasing data processing requirements, we model the decrease percentage of up to 75%, where 25% is considered as additional margin to transform data processing requirement of complex aggregation function into a simple aggregate function. Accordingly for increase requests, low range is from 10% to 40%, medium range is from 60% to 90% and high range is from 110% to 150%. For decrease requests, low range is from 5% to 25%, medium range is from 35% to 55% and high range is from 65% to 75%.

**Table 1**
Workflow and simulation parameters.

| Parameter | Value |
|---|---|
| External Source Data Rate | Range [5, 10] MB/s considering 5 MB/s as the minimum and 10 MB/s as the maximum based on [30] |
| Ingress Network Bandwidth | Range [615, 926] MB/s |
| Ingress Network Latency | Range [0.00064, 0.00086] s |
| Egress Network Bandwidth | Range [122, 218] MB/s |
| Egress Network Latency | Range [0.021, 0.031] s |
| Data transfer cost | Ingress traffic: 0<br>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 Data Processing Unit | 1 MB |
| Minimum Data Processing Rate | 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 s |
| Simulation time | 180 s (3 min) |

### 5.3. Configuration changes in service output data rate

Considering Rizou et al.'s research work [33], the selectivity of the operator is the percentage of output data to input data and this selectivity varies between 0 and 1. Selectivity close to 1 means that the operator generates output data rate equal to the input data rate, while selectivity close to 0 means that the operator generates very low output data rate and acts as high selective filter in the network. In this research work, the output data rate is up to 100% of input data rate and the data rate unit is kilobit per second. The advancements of networking and IoT technologies lead to increase in the speed of data being exchanged. For example, in Fischer and Bernstein [34], the size of tuple used in experiments varies from bytes to KB to MB, so that the data rate unit in megabit/megabyte per second is being considered. These advancements are continuing, which means the data rate will continue to increase. Therefore, it is valuable to consider additional 50% beyond 100% as the future margin of increase. Accordingly, the output data rate is considered to be up to 150% of input data rate as defined in [29].

Considering output data rate is up to 150% of input data rate (including 50% additional margin), we consider the percentage change in increasing service output data rate is up to 100% (low range [10–30%], medium range [50–70%] and high range [90–100%]). While for decreasing service output data rate is up to 75% (low range [5–15%], medium range [25–35%] and high range [45–50%]).

### 5.4. Experimental scenarios

To evaluate the quality of response solution for revising the scheduling plan at runtime when application-level change happens, we examine different experiment scenarios for different dynamic forms. All these experiments are with regard to cost, change and time. The cost is the solution cost after the change is applied. This cost includes data provisioning cost and data transfer cost. Regarding change, we consider the number of changes

applied to the current scheduling plan in terms of compute resources in order to respond to any runtime change. In other words, it is a number of VM provisioning and/or deprovisioning changes that are made to revise the current scheduling plan. For time, we consider the request execution time (computational time) required to process and complete this request. This time is a sum of the request's processing time, algorithm running time and highest boot time among the VMs provisioned.

For each dynamic case, we conduct an experiment to study the quality of solutions being generated in response to this dynamic change. Thus, we will perform 11 experiments as follows: two experiments (one for increase request and the other for decrease request) for each case in Dynamic Form 2 (with a total of four experiments), one experiment for each case in dynamic 3 (with a total of five experiments) and one experiment for each case in dynamic 4 (with a total of two experiments). Then, we record experimental results of the quality of solution and compare these results to examine the scale of performance quality.

The aforementioned experiment scenarios are used to examine and evaluate the performance and service quality of three different techniques under different application-level changes.

Baseline Technique (BT): our pluggable dynamic scheduling technique with proposed realistic and straightforward algorithm (i.e. baseline algorithm) that does not need to use any complicated heuristic. This algorithm handles different dynamic changes by merely provisioning the VM with the highest computing power and achieving service minimum data processing unit when a new service is deployed or more computing power is needed, and deprovisioning some of the VMs available when existing service is deleted or less computing power is needed. It worth noting that this technique can, if possible, deprovision part of VMs available based on the amount of computing power being decreased.

Dynamic Fair-Share Technique (DFST): Fair sharing model, to one resource type or multiple resource types, is a default scheduling decision used by Apache YARN and Mesos to equally share the resources of a cluster among applications over time. This default scheduler cannot handle dynamic forms of stream workflows by managing the resources at runtime. Therefore, we have extended and implemented this model to support elasticity and adjust scheduling plan at runtime to cope with stream workflows and its dynamic forms. Accordingly, DFST is our pluggable dynamic scheduling technique with the proposed dynamic fairness heuristic method. This technique provisions the same type of VM (with high-medium computing power) when a new service is deployed or there is a need for more computing power, and deprovisions any available VM when less computing power is needed or releases all available VM(s) when the service is deleted.

Optimisation Technique (OT): our pluggable dynamic scheduling technique with proposed plugin algorithms and methods presented in Section 5.5.

By comparing the quality of solution being generated by our techniques (BT, DFST and OT) in response to various dynamic changes, we can evaluate the efficiency of each technique in respect to the others and find the most efficient technique that produces the best response solution. The comparison between OT and BT is aimed at figuring out whether the complex heuristic-based method is necessary to improve the quality of solution being generated to respond to application-level runtime change. The comparison between OT and DFST is aimed at evaluating the quality of solution generated by the dynamic version of fair-share scheduling decision since a fair share model is used in big data application orchestrators (i.e. Apache YARN and Mesos).

Note that we will not conduct an experiment for Dynamic Form 1 (change the streaming data velocity) as we have investigated this form in detail in our previous work [30].

### 5.5. Plugin scheduling algorithms and techniques

As the proposed technique is a pluggable method to dynamically schedule stream workflow at runtime, customisable or plugin scheduling algorithms are needed to make scheduling decisions. To run our experiments, different types of plugin algorithms/methods are used with different techniques to perform quality of solution evaluations according to the aforementioned experimental scenarios.

With OT, we used our previous scheduling algorithm and techniques presented in [29,30].

For scheduling stream workflow at deployment time, the proposed algorithms in [29] or GA with Random Immigrants Scheme in [30] can be used as plugin algorithms. GA with Random Immigrants Scheme is the advanced version of traditional GA and performed better even with dynamic scheduling according to the results presented in [30]. Thus, we use GA with Random Immigrants Scheme as plugin algorithm in Line 1 of Algorithm 1.

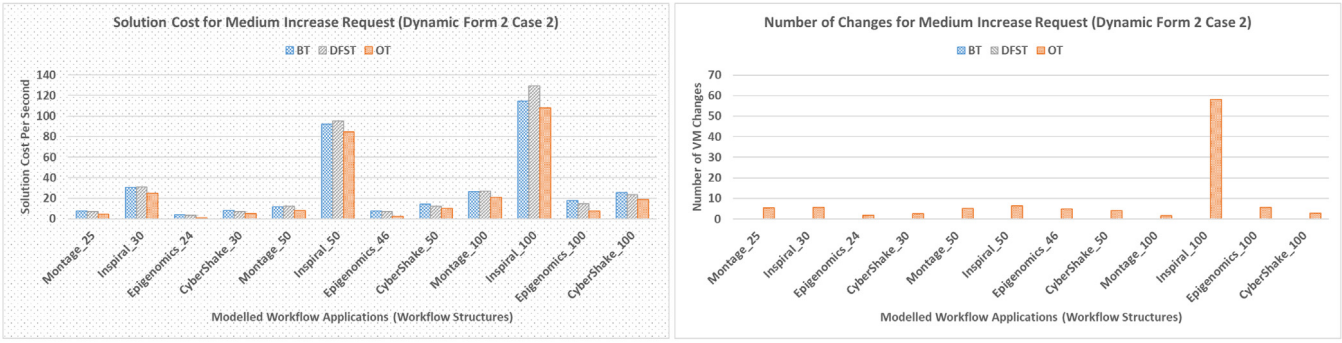
For adaptive scheduling with Dynamic Form 1 and 2, Two-level Greedy Algorithm [30] is used as a plugin algorithm. For Dynamic Form 1, this algorithm is used in Line 6 of Algorithm 1 to dynamically respond to the data velocity changes for services by finding the best resource selection solution for those services affected by such changes. For Dynamic Form 2, it is used in Line 17 and 20 of Algorithm 2 to revise the scheduling plan with both data velocity events. With a velocity increase event, this algorithm calls for provisioning more computing power while with velocity decrease events, it finds those VMs that are not needed any more to deprovision them. In Line 12 of Algorithm 2, a new heuristic technique is proposed as a plugin algorithm (see Algorithm 8). It assesses all provisioned VMs of a given service to ensure they still achieve minimum data processing based on the updated data processing requirement. Then, it provisions more computing power if needed in the case of increase events or deprovisioning those provisioned VMs that are not needed to achieve the updated data processing requirement in cases of decrease events.

For adaptive scheduling with Dynamic Form 3, two algorithms are used as plugin algorithms. Greedy Selection algorithm [29] is used in Line 2 of Algorithm 5 to find computing resources (resource selection solution) for the new service. Note that this algorithm originally works on all services, but for our purpose here, we made a minor modification to make it run only with a given service (i.e. new service). Moreover, in Line 4 of Algorithm 5, Two-level Greedy Algorithm [30] is used as a pluggable algorithm to revise the scheduling plan based on the list of services affected by dynamic change.

With BT, the proposed technique is plugged in with a simple schedule model to provision the highest VM when more computing power is needed and deprovision VM(s) if applicable when the provisioned VM(s) is unnecessary. With DFST, the proposed technique is plugged in with the extended fair-share model to support dynamic scheduling. If more computing power is needed (such as adding new service or modifying the existing service with increase requests), it provisions the same type of VM, while deprovisioning VM(s) if applicable when the provisioned VM(s) is unnecessary. For both BT and DFST, if changes occur in the existing service, they provision and/or deprovision VM(s) according to whether the change is an increase request or decrease request after checking the current computing power for this service.

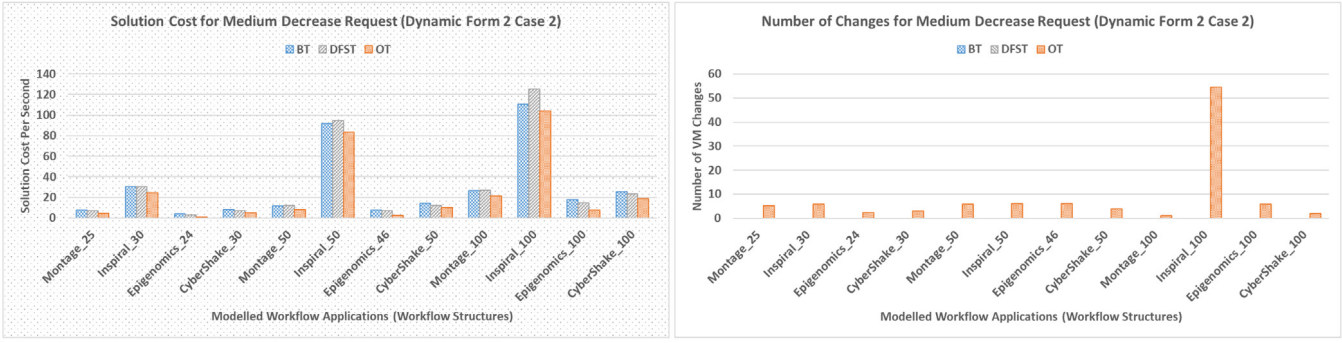
## 6. Experimental results

The use of a real environment to conduct our experiments would produce inconsistent evaluation results as the parameters



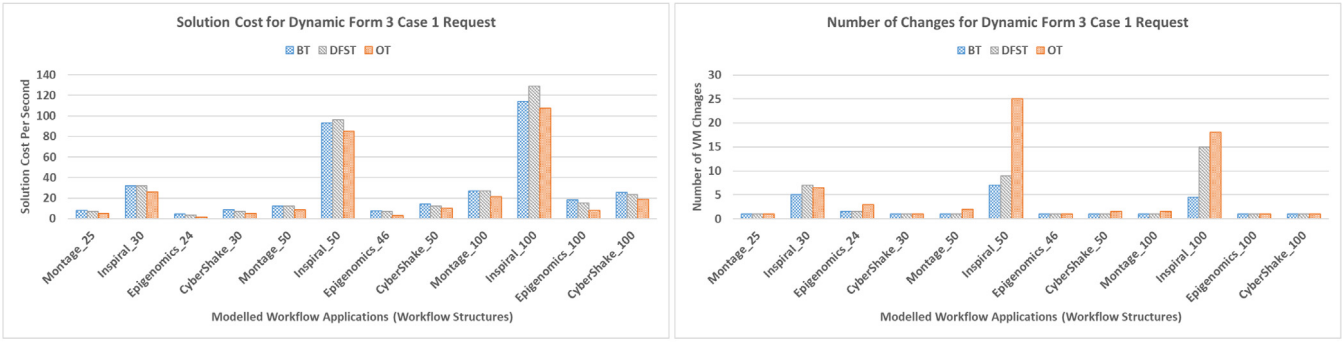

(a) Solution cost (b) Number of changes

**Fig. 5.** Quality of solution for different workflow structures under Dynamic Form 2 Case 2 Increase (medium range).



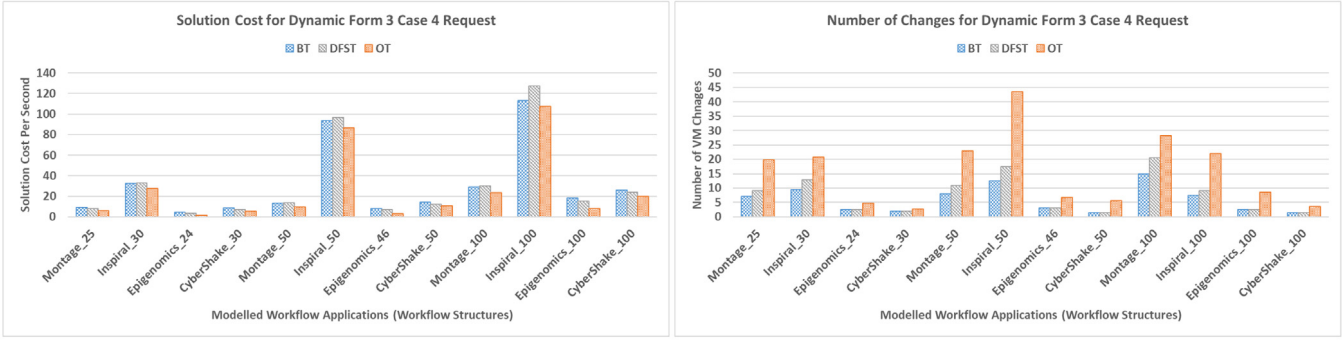

(a) Solution cost (b) Number of changes

**Fig. 6.** Quality of solution for different workflow structures under Dynamic Form 2 Case 2 Decrease (medium range).



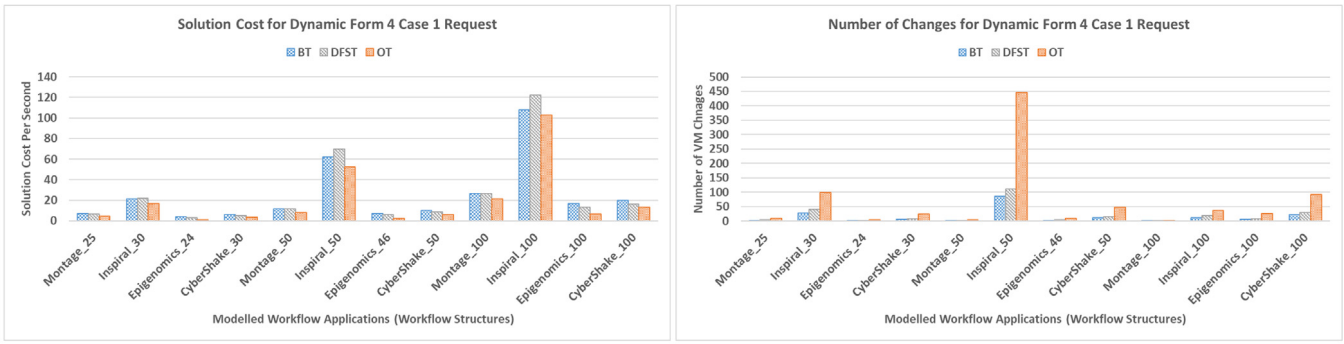

(a) Solution cost (b) Number of changes

**Fig. 7.** Quality of solution for different workflow structures under Dynamic Form 3 Case 1.




(a) Solution cost (b) Number of changes

**Fig. 8.** Quality of solution for different workflow structures under Dynamic Form 3 Case 4.




(a) Solution cost (b) Number of changes

**Fig. 9.** Quality of solution for different workflow structures under Dynamic Form 4 Case 1.

**Algorithm 8** ResourceSelection_DPReqChange(Service)

```
unitMIPS ← MI^{S_n} * unitDPRate
reqUnits ← get number of units required based on updated MI^{S_n}
changeEvent ← get velocity change event {increase or decrease}
S_n ← get the existing service that is selected
pro(S_n) ← provisioned VMs for a service
for each vm in pro(S_n) do
   if MIPS_vm >= unitMIPS then
      if reqUnits > 0 then
         reqUnits = reqUnits − ⌊(MIPS_vm/unitMIPS)⌋
      else
         add vm in rmVM(S_n) {deprovision vm}
      end if
   else
      add vm in rmVM(S_n)
   end if
end for
while reqUnits > 0 do
   selectedVM, VMList = ϕ
   VMOffers ← VM offers of S_n placement cloud order by comp. power
   for each vm_offer in VMOffers do
      achievedUnits = ⌊(MIPS_vm_offer/unitMIPS)⌋
      if achievedUnits >= reqUnits or vm_offer is last offer then
         selectedVM = vm_offer
         break
      end if
   end for
   VMList = VMList ∪ selectedVM
   reqUnits = reqUnits − ⌊(MIPS_selectedVM/unitMIPS)⌋
end while
```

of this environment cannot be controlled and they continue to change with every workflow execution. The use of simulation environment approach is a visible solution. Therefore, we conduct our experiments using IoTSim-Stream [32], a validated simulator for modelling and executing stream workflow in Multicloud environments. It provides a controllable real-world simulation environment to conduct and repeat experiments for evaluating scheduling algorithms and techniques. By using this simulator, large-scale simulation experiments can be conducted to assess the quality of solution produced by different scheduling techniques to respond to application-level changes. These simulations are so close to reality the obtained results are valid in real-life. In addition, conducting these simulations on simulation environments is cost-effective, effortless and various conditions are reproducible to reproduce the results.

To perform our experiments, IoTSim-Stream [32] is used on a Nectar Cloud virtual machine with 8 vCPUs, 32 GB of RAM memory and running Ubuntu 16.04.1 LTS, and the results of these experiments are collected. For OT, each experimental scenario runs 10 times since random-based immigrants genetic algorithm is used in this technique, and then the average value of the obtained results is taken and used in the representation of experimental results. Moreover, for the quality of solution results, we present the average value for both the cost of solution and number of changes as two runtime changes are made during simulation time. Regarding the results of Dynamic Form 2 scenarios, we only present medium-range results as these results are sufficient to reach the conclusion.

We have examined the experimental results looking for key results that allow us to reach the conclusion. We found that the results of experiment scenarios for Dynamic Form 2 Case 2 (with increase change), Dynamic Form 2 Case 2 (with decrease change), Dynamic Form 3 Case 1, Dynamic Form 3 Case 4 and Dynamic Form 4 Case 1 are enough for our discussion. Figs. 5 to 9 depict the quality of solution result for the aforementioned dynamic forms and cases. From these results, our analysis and findings are:

- In terms of solution cost, OT achieved the best results in comparison with BT ad DFST. This is because OT applied genetic algorithm at deployment time to find the best plan to place services over multiple cloud infrastructure to minimise not only provisioning cost but also data transfer by utilising data locality. With an efficient service placement and scheduling plan, OT is able to maintain the lowest cost when handling all dynamic changes during the execution of workflow. On the other hand, BT and DFST do not have that capability, so they cannot minimise the solution cost after handling dynamic changes. In addition, the highest cost saving is achieved by Epigenomics workflow structure. This is because such workflow processes a lower amount of data compared with other workflow structures, so that less than average computing power is needed. Based on this, provisioning VM with high-medium or highest computing power does not lead to achieving the best execution cost. Instead finding a near-optimal scheduling plan at deployment allows maximisation of savings in this step and later when amending the scheduling plan.
- Based on the scheduling decision of BT and DFST, the conclusion from Fig. 5b is that these techniques did not make any VM changes as over-provisioning is sufficient to cope with the increased need for computing power in the changed service and downstream services. OT needs to make some VM changes as it needs to provision VM(s) with suitable computing power to handle the change request efficiently while maintain minimal execution cost.
- From Fig. 6b, Both BT and DFST are unable to deprovision VM(s) with decrease change request. This is because the

scheduling decision of these techniques is based on provisioning VMs with high-medium to high computing power, so that when there is no substantial decrease in data speed or MIPS value, they cannot avoid over-provisioning and thus additional computing resources are wasted. Moreover, it is worth noting that by having various VM types in the generated scheduling plan as an OT does, the opportunity becomes very high to find suitable VM(s) to deprovision when any change happens in output stream velocity, leading to avoidance of over-provisioning and reduce the total execution cost.

- From Fig. 7b, it is clear that the results of OT are close to BT and DFST in most cases. This is because adding a new service that outputs to sink has no effect on the other services and it only requires provisioning VM(s) to deploy this service. Thus, OT is able to provision suitable VM(s) while maintaining minimal number of changes. On the other hand, when the runtime change has effect on the other services (i.e. downstream services), OT needs to make more VM changes in comparison with BT and DFST (see Fig. 8b) in order to maintain lower execution cost. Moreover, OT in some cases (such as with Epigenomics_24, CyberShake_30 and CyberShake_100) achieved results similar to those achieved by BT and DFST, as the marginal over-provisioning is sufficient to handle the consequence of adding a new service on the downstream services.
- From Fig. 9b, the straightforward conclusion is that OT deprovisions more VMs when the service is deleted as it generally provisioned more VMs at deployment time to maintain execution cost to be as low as possible by finding near-optimal/optimal scheduling plan. Moreover, we can notice that the highest number of changes is achieved with Inspiral_50 as this workflow processes a large amount of data streams and then requires large computation power, so that OT at deployment time provisioned more VMs to achieve the required computing powers and deprovision them when the service is deleted.
- In terms of request execution time, this time includes computational time required by scheduling techniques to amend the current scheduling plan, VM boot time when more computing power is needed and the time for performing the updates such as updating services, stream dependencies and parent–child relationships. For computational time, all techniques achieved negligible time to amend the current scheduling plan since their scheduling decisions were made using a heuristic approach. Also, the time required for performing the update is also negligible. Based on that, the request execution time will remain negligible when the technique only needs to deprovision VM(s) to respond to runtime change (such as Dynamic Form 4 Case 1). Request execution time will be increased when there is a need to provision new VMs. This time is mainly determined by the maximum VM boot time among the provisioned VMs. DFST incurred constant time (approx. 35 simulation time) since this technique provisions the same type of VM all the time. BT incurred on average 36 simulation time, while OT incurred on average 40 simulation time. Accordingly, there is no significant difference in request execution time between those techniques.

## 7. Conclusion and future work

In this paper we investigated dynamic scheduling problems of stream workflow in the cloud under different dynamic forms including fluctuations of input data rate, the change of workflow structure and the change of real-time data processing requirement. To enable the full dynamic support for in-progress workflow, we proposed a scalable and pluggable dynamic scheduling technique that allows the user to plug her/his algorithms and methods in place to respond to runtime changes with the focus on scheduling decisions rather than the complexity of dealing with these changes. We also presented three different plug-in techniques that can be used to handle the aforementioned runtime changes, so the user can use these built-in techniques. These techniques vary based on their complexity, from simple heuristic to multi-heuristic algorithm to tackle different dynamic forms of stream workflow. The quality of solutions generated by those techniques are evaluated to determine the most efficient technique. The experimental results showed that OT outperformed BT and DFST in quality of solution evaluations.

For future studies, we would like to extend our pluggable scheduling technique with resource fault tolerance capability. We are also interested in extending our proposed technique with intelligence capability by integrating multiple plug-in algorithms and methods, so that the improved technique can make intelligent decisions about choosing and changing the plug-in algorithm on the fly based on application monitoring and change analysis.

## CRediT authorship contribution statement

**Mutaz Barika:** Conceptualization, Methodology, Software, Investigation, Writing - review & editing. **Saurabh Garg:** Supervision, Writing - review & editing. **Rajiv Ranjan:** Writing - review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

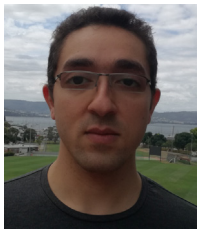
## Acknowledgement

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

## References

[1] I. Lee, The internet of things for enterprises: An ecosystem, architecture, and iot service business model, Internet Things (2019) 100078.
[2] R. Woodhead, et al., Digital construction: From point solutions to iot ecosystem, Autom. Constr. 93 (2018) 35–46.
[3] C. Alexopoulos, et al., A taxonomy of smart cities initiatives, in: 12th International Conference on Theory and Practice of Electronic Governance, ACM, 2019, pp. 281–290.
[4] F. Caro, R. Sadr, The internet of things (iot) in retail: Bridging supply and demand, Bus. Horiz. 62 (1) (2019) 47–54.
[5] Y. Liu, et al., Runtime-aware adaptive scheduling in stream processing, Concurr. Comput.: Pract. Exper. 28 (14) (2016) 3830–3843.
[6] X. Liu, R. Buyya, D-storm: Dynamic resource-efficient scheduling of stream processing applications, in: 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2017, pp. 485–492.
[7] R. Kombi, et al., DABS-storm: A data-aware approach for elastic stream processing, in: Transactions on Large-Scale Data-and Knowledge-Centered Systems XL, Springer, 2019, pp. 58–93.
[8] D. Sun, R. Huang, A stable online scheduling strategy for real-time stream computing over fluctuating big data streams, IEEE Access 4 (2016) 8593–8607.
[9] D. Sun, et al., Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams, J. Supercomput. 74 (2) (2018) 615–636.
[10] V. Vavilapall, et al., Apache hadoop yarn: Yet another resource negotiator, in: 4th Annual Symposium on Cloud Computing, ACM, 2013, pp. 1–16.
[11] B. Hindman, et al., Mesos: A platform for fine-grained resource sharing in the data center, in: NSDI, Vol. 11.

[12] F. Liu, et al., Traversing knowledge networks: An algorithmic historiography of extant literature on the internet of things (iot), J. Manage. Anal. 4 (1) (2017) 3–34.
[13] A. Al-Fuqaha, et al., Internet of things: A survey on enabling technologies, protocols, and applications, IEEE Commun. Surv. Tutor. 17 (4) (2015) 2347–2376.
[14] S. Li, et al., The internet of things: a survey, Inf. Syst. Front. 17 (2) (2015) 243–259.
[15] S. Li, et al., 5g internet of things: A survey, J. Ind. Inf. Integr. 10 (2018) 1–9.
[16] P.K. Sharma, et al., EH-HL: effective communication model by integrated EH-WSN and hybrid lifi/wifi for iot, IEEE Internet Things J. 5 (3) (2018) 1719–1726.
[17] C. Zhang, Y. Chen, A review of research relevant to the emerging industry trends: Industry 4.0, IoT, Block Chain, and Business Analytics, J. Ind. Integr. Manage. 5 (1), 165–180.
[18] J.H. Kim, A review of cyber-physical system research relevant to the emerging it trends: industry 4.0, iot, big data, and cloud computing, J. Ind. Integr. Manage. 2 (03) (2017) 1750011.
[19] C. Wang, et al., Iot and cloud computing in automation of assembly modeling systems, IEEE Trans. Ind. Inf. 10 (2) (2014) 1426–1434.
[20] C. Xie, et al., Linked semantic model for information resource service toward cloud manufacturing, IEEE Trans. Ind. Inf. 13 (6) (2017) 3338–3349.
[21] Z. Bi, D. Cochran, Big data analytics with applications, J. Manage. Anal. 1 (4) (2014) 249–265.
[22] L. Jiang, et al., An iot-oriented data storage framework in cloud computing platform, IEEE Trans. Ind. Inf. 10 (2) (2014) 1443–1451.
[23] X. Zheng, et al., Cloudqual: A quality model for cloud services, IEEE Trans. Ind. Inform. 10 (2) (2014) 1527–1536.
[24] X. Zheng, et al., Cloud service negotiation in internet of things environment: A mixed approach, IEEE Trans. Ind. Inf. 10 (2) (2014) 1506–1515.
[25] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[26] M. Hirzel, et al., IBM streams processing language: Analyzing big data in motion, IBM J. Res. Dev. 57 (3/4) (2013).
[27] H. Hu, et al., Toward scalable systems for big data analytics: A technology tutorial, IEEE Access 2 (2014) 652–687.
[28] V. Cardellini, et al., Optimal operator replication and placement for distributed stream processing systems, ACM SIGMETRICS Perform. Eval. Rev. 44 (4) (2017) 11–22.
[29] M. Barika, et al., Scheduling algorithms for efficient execution of stream workflow applications in multicloud environments, IEEE Trans. Serv. Comput. (2019) (early access).
[30] M. Barika, et al., Adaptive scheduling for efficient execution of dynamic stream workflows, arXiv preprint arXiv:1912.08397.
[31] C. Chen, et al., Connected vehicular transportation: Data analytics and traffic-dependent networking, IEEE Veh. Technol. Mag. 12 (3) (2017) 42–54.
[32] M. Barika, et al., Iotsim-stream: Modelling stream graph application in cloud simulation, Future Gener. Comput. Syst. 99 (2019) 86–105.
[33] S. Rizou, et al., Solving the multi-operator placement problem in large-scale operator networks, in: 19th International Conference on Computer Communications and Networks, 2010, pp. 1–6.
[34] L. Fischer, B. Abraham, Workload scheduling in distributed stream processors using graph partitioning, in: IEEE International Conference on Big Data (Big Data), 2015, pp. 124–133.



**Mutaz Barika** has obtained his BSc. and MSc. in Computer Science from University of Petra and King Saud University respectively. He is currently a Ph.D. Candidate at University of Tasmania, Australia. He has been awarded an Australian Government Research Training Program (RTP) Scholarship for supporting his studies. His current research interests include Big Data, Big Data Workflow, Cloud Computing, IoT and Data Security.

**Saurabh Garg** is a Senior Lecturer at the University of Tasmania, Hobart, Tasmania. He is one of the few Ph.D. students who completed in less than three years from the University of Melbourne in 010. 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.

**Rajiv Ranjan** is a Chair Professor in Computing Science and Internet of Things at Newcastle University, United Kingdom. He has received two IEEE research excellence awards 018 IEEE TCCPS Early Career Award and 016IEEE TCSC Award for Excellence in Scalable Computing), which recognised his leading expertise in algorithms, resource management models and distributed system architectures for Cloud computing, Internet of Things (IoT) and Data Science. Another testimonial of his international research leadership is his appointment by IEEE Computer Society as the Advisory Board Chair and Lead Editor 01 019) for the Blue Skies department of IEEE Cloud Computing. In this appointment, Prof Ranjan's main role is to develop a vision for the research community to guide future research at the intersection of Cloud computing, IoT and Data Science. Additionally, he also serves on the editorial boards of top quality international journals including IEEE Transactions on Cloud computing, ACM Transactions on Internet of Things, IEEE Transactions on Computers 014 016), IEEE Cloud Computing, Springer Computing, The Computer Journal (Oxford University Press), among many others. His research outcomes include 40+ academic peerreviewed articles and multiple opensource software toolkits-stemming from funded research projects worth over $12 Million AUD (£6 Million GBP). He is one of the highly cited authors (top 0.05%) in computer science and software engineering worldwide (h-index=46, g-index= 121, and 12700+ google scholar citations).