Future Generation Computer Systems 88 (2018) 624–635

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Detecting performance anomalies in scientific workflows using hierarchical temporal memory

Maria A. Rodriguez *, Ramamohanarao Kotagiri, Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

HIGHLIGHTS

- An anomaly detection framework for scientific workflows is presented.
- HTM is used to detect anomalies on a stream of resource consumption time series data.
- The HTM-based model is unsupervised and learns incrementally.
- The framework is platform-agnostic and can be deployed on different infrastructures.
- Detected anomalies can trigger scheduling and resource provisioning actions.

ARTICLE INFO

Article history:
Received 23 November 2017
Received in revised form 18 April 2018
Accepted 9 May 2018
Available online 18 May 2018

Keywords:
Online anomaly detection
Scientific workflow
Hierarchical temporal memory
Performance anomalies

ABSTRACT

Technological advances and the emergence of the Internet of Things have lead to the collection of vast amounts of scientific data from increasingly powerful scientific instruments and a growing number of distributed sensors. This has not only exacerbated the significance of the analyses performed by scientific applications but has also increased their complexity and scale. Hence, emerging extreme-scale scientific workflows are becoming widespread and so is the need to efficiently automate their deployment on a variety of platforms such as high performance computers, dedicated clusters, and cloud environments. Performance anomalies can considerably affect the execution of these applications. They may be caused by different factors including failures and resource contention and they may lead to undesired circumstances such as lengthy delays in the workflow runtime or unnecessary costs in cloud environments. As a result, it is essential for modern workflow management systems to enable the early detection of this type of anomalies, to identify their cause, and to formulate and execute actions to mitigate their effects. In this work, we propose the use of Hierarchical Temporal Memory (HTM) to detect performance anomalies on real-time infrastructure metrics collected by continuously monitoring the resource consumption of executing workflow tasks. The framework is capable of processing a stream of measurements in an online and unsupervised manner and is successful in adapting to changes in the underlying statistics of the data. This allows it to be easily deployed on a variety of infrastructure platforms without the need of previously collecting data and training a model. We evaluate our approach by using two real scientific workflows deployed in Microsoft Azure's cloud infrastructure. Our experiment results demonstrate the ability of our model to accurately capture performance anomalies on different resource consumption metrics caused by a variety of competing workloads introduced into the system. A performance comparison of HTM to other online anomaly detection algorithms is also presented, demonstrating the suitability of the chosen algorithm for the problem presented in this work.

1. Introduction

Scientific applications enable the extraction of knowledge from vast amounts of data. The volume and underlying value of these data are continuously increasing as technological advances that support the creation of more powerful and precise scientific instruments are made. The significance and importance of the analyses performed by these applications becomes then of utmost importance for scientific progress. For instance, the Large Hadron Collider (LHC) at CERN produces approximately 30 petabytes of data per year that must be analyzed to understand the effects of particle collisions. Another example is the Laser Interferometer Gravitational Observatory (LIGO) project, which harnesses scientific workflows to process data. Since the deployment of their advanced interferometers leveraging hardware developments in optics and

* Corresponding author.
E-mail addresses: marodriguez@unimelb.edu.au (M.A. Rodriguez), kotagiri@unimelb.edu.au (R. Kotagiri), rbuyya@unimelb.edu.au (R. Buyya).

https://doi.org/10.1016/j.future.2018.05.014
0167-739X/

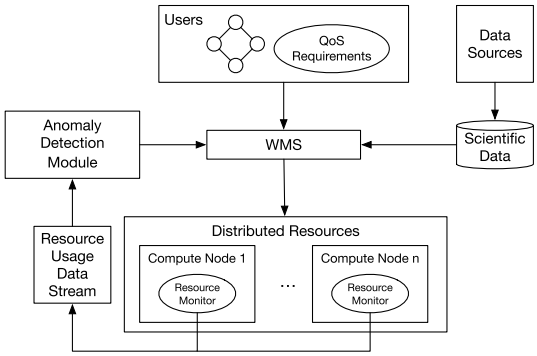

Fig. 1. High-level overview of a performance anomaly detection framework for scientific workflows.

vibration suspension systems, gravitational waves stemming from the collision of black holes were detected for the first time in 2015. This discovery is not only fundamental in supporting Einstein's General Theory of Relativity, but has also paved the way for the emerging field of gravitational wave astronomy which will eventually enable a better understanding of gravitational wave sources and the universe.

This increased ability to collect data from different sources leads to an inevitable growth in the scale and complexity of scientific applications. This is also true for workflows, which have been traditionally used as a way of structuring different scientific computations and their dependencies. They enable scientist to describe applications in a platform agnostic manner while Workflow Management Systems (WMS) hide the complexities of the underlying computing infrastructure by transparently orchestrating the execution of workflow tasks. Emerging extreme-scale workflows are becoming more widespread and so is the need to efficiently automate their execution on a variety of platforms such as high performance computers, dedicated clusters, cloud computing infrastructures, and, more recently, fog environments.

It becomes paramount then to execute large-scale complex scientific workflows in a reliable and scalable manner on distributed systems. In particular, being able to continuously monitor their performance and create models of expected behavior that adapt over time to the dynamicity of the underlying infrastructure can be of great benefit for scientists. Specifically, we argue that WMSs can use runtime resource consumption information to detect performance anomalies and mitigate their effects on the overall workflow execution and Quality-of-Service (QoS) requirements. This can be achieved by making dynamic scheduling and resource provisioning decisions that are triggered by the detection of such anomalous behavior. For example, a performance anomaly may be detected in the CPU consumption pattern of a running task. This will eventually lead to the task taking longer than expected to complete, ultimately having a negative impact on the total workflow execution time (i.e., makespan). Such effect becomes more prominent as the scale of the applications in terms of the number of tasks, the amount of data they process, and their computational requirements continue to grow. Resource management modules can attempt to correct performance issues by provisioning more resources, executing unscheduled workflow tasks on more powerful resources, migrating or replicating running tasks, or scaling the compute resources vertically (e.g., increasing the memory of a virtual machine), among other approaches.

As a result, our goal is to develop a framework that optimizes the performance of complex data-intensive workflows by detecting, diagnosing, and potentially correcting the cause of anomalous runtime performance. An overview of the key components of such a system are depicted in Fig. 1. In this work, we focus on the first challenge; that is, the early detection of performance anomalies through real-time infrastructure monitoring. In particular, we aim to identify patterns in the data that do not conform to expected behavior and we focus on analyzing time series data that contain the resource consumption details of tasks at different stages of their execution. The metrics considered in our approach are related to the CPU and I/O usage of a given task on a particular machine.

There are various requirements that arise from the nature of the problem addressed and the monitored data. Mainly, we strive to implement a model that is capable of learning in an online fashion as data becomes available. The main reason for this requirement is the nature of the computing infrastructures used for the deployment of workflows. Firstly, there are a variety of platforms currently used for this purpose; by having a model that learns as data is collected, our framework can be deployed on different computing infrastructures in a seamless manner. Secondly, there are a wide range of factors impacting the performance of running jobs in compute nodes, especially in increasingly popular multi tenant, virtualized environments such as cloud and fog platforms. By learning incrementally in an online manner, the model will dynamically adjust to environmental changes such as peak hour in a data center. Finally, online learning enables our framework to be used to detect anomalies in the execution of different scientific workflows, without the need of previously training a specific model based on data collected from a dedicated infrastructure for example. Another important requirement is for the framework to be capable of processing sequential streaming data in one pass and as the data becomes available. Finally, we are interested in detecting temporal anomalies on these data, that is, identifying a set of abnormal transitions between patterns as opposed to identifying a single data point that deviates from what is standard (i.e., spatial anomaly).

To achieve these requirements, we propose an anomaly detection model that uses Hierarchical Temporal Memory (HTM) networks. HTMs are continuous learning systems that meet our requirements by mimicking the anatomy of the mammalian neocortex and the behavior of neurons to perform learning, inference, and prediction [1]. They are efficient, tolerant to noise, capable of adapting to changes in the statistics of the data, and capable of detecting subtle temporal anomalies [2]. Our detection algorithm uses an HTM model for each monitored metric to detect CPU or I/O related anomalies. For each workflow task deployed, its resource consumption is monitored at given intervals and analyzed as soon as it becomes available. Our comprehensive evaluation demonstrates the efficiency of our method in identifying anomalies caused by different types of competing workloads on two scientific workflows from the bioinformatics field. Our solution has also been designed in such a way that it facilitates the identification of the cause of the anomalies in our future work.

The rest of this paper is organized as follows. Section 2 presents the related work followed by an overview of HTM systems in Section 3. Section 4 explains the proposed anomaly detection method and Section 5 presents the experimental setup and the evaluation of our solution. Finally, conclusions and future work are outlined in Section 6.

2. Related work

Anomaly detection in sequential data has been extensively researched. There are a variety of existing approaches that are offline and are designed to process data once a model capable of making predictions has been built based on some training data. Netflix's Robust Anomaly Detection (RAD) [3] framework is an example. It is a statistical approach based on Robust Principle Component Analysis (RPCA) [4] that relies on data having high cardinality. Another example is HOT SAX [5], an algorithm capable of finding time

series discords, defined as subsequences of a longer time series that are maximally different to all the rest of the time series. More recently, Malhotra et al. [6] addressed the anomaly detection problem in time series using long short term memory networks while Akouemo et al. [7] proposed a probabilistic approach that uses linear regression and a Bayesian maximum likelihood classifier. Finally, ARIMA [8] is another widely used, batch-based statistical time series forecasting model for temporal data with seasonality.

Our problem is more related to anomaly detection in streaming data as it involves analyzing a continuous sequence of records arriving continuously; that is, each record is processed once and learning is done in an online fashion. There are several existing approaches that meet these requirements. An example is EXPected Similarity Estimation (EXPoSE) [9], a kernel-based algorithm designed for large-scale datasets that is capable of efficiently computing the similarity between new data points and the distribution of regular data. However, contrary to our problem definition, EXPoSE was designed for datasets with high-dimensional features. Another example is the method proposed by Wang et al. [10]; a statistical technique based on relative entropy that is computationally lightweight and does not assume a particular form for the distribution of the data, that is, it is non-parametric. The Bayesian Online Checkpoint Detection [11] method is also designed to analyze streaming data, however, it focuses on spatial anomalies and assumes the underlying distribution of the data is known in advance. Contrary to this, KNN-CAD [12] is a probabilistic, non-parametric approach that relies on a density and distance based nearest neighbor algorithm.

HTM systems have been used to address various learning tasks in different domains. For example, they have been used to identify anomalous traffic in computer networks [13,14], to detect anomalies in the behavior of website users [15], to process bio-signals and predict sensory and location data in the context of smart homes [16], to model typical geospatial travel patterns and identify anomalies in movement [17], to detect anomalies in publicly traded stocks [18], to detect the optic nerve in retina images [19], and to build a commercial proactive and automatic IT incident response system called Grok [20], among other applications.

In terms of its applications, anomaly detection has been widely used in the context of distributed systems. For instance, it has been extensively applied to intrusion detection in networked systems [21], to increase the reliability of web applications [22], to monitor and diagnose faults in sensor networks [23], to prevent DDoS attacks in clouds [24], and to detect anomalies in the usage of virtual machine in clouds [25].

Performance anomaly detection has also been extensively implemented and a variety of techniques have been used for this purpose [26]. For example, statistical techniques such as regression and correlation analysis have been used to diagnose potential causes of SLA violations in virtualized systems [27] and to detect performance anomalies in multi-server distributed systems [28]. Other statistical approaches used for this purpose include Markov and probability models [29,30]. Machine learning approaches have also been used for this purpose. Examples include the work by Tan and Gu which uses Bayesian classifiers and tree augmented networks to predict and classify anomalies [31], the work by Yu and Lan which uses non-parametric clustering to detect anomalous behavior in Hadoop clusters [32], and the work by Pannu et al. which uses Support Vector Machines for detecting system failures in cloud environments [33].

Finally, Gaikwad et al. [34] are the first to propose a framework for the detection of performance anomalies on time series monitoring data for scientific workflows. Although the motivation behind their work and ours is similar, they propose the use of an anomaly detection algorithm that uses auto-regression based statistical methods. To generate anomaly triggers, they create model parameters that fit the training data and estimate an error based on the predicted and actual values. Their training data was collected on a dedicated, controlled infrastructure, and hence the assumption that the data analyzed for anomalies would stem from a similar infrastructure is made.

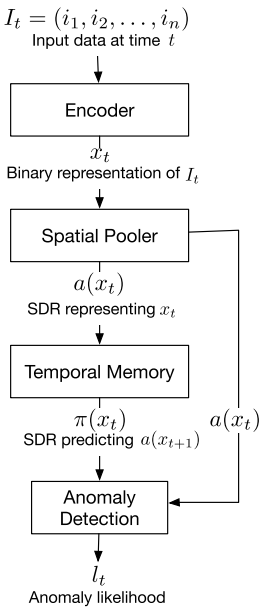

Fig. 2. Core algorithms of a region in a Hierarchical Temporal Memory model.

3. Hierarchical Temporal Memory (HTM)

HTM is an unsupervised learning technique biologically derived from the neocortical region of the brain. It aims to mimic the way in which the brain continuously processes sensory data with temporal awareness and spatial properties. In particular, HTM systems have an inherent notion of time and are capable of continuously learning the structure of unlabeled streaming data to make predictions and detect anomalies. They are capable of not only learning the spatial representation of patterns but also sequences of those patterns and the context in which they occur.

In an HTM system, sensory input data is processed to produce an inference that can be interpreted as an invariant understanding of the problem data. An overview of this process along with the main components present in an HTM system are depicted in Fig. 2. In this section, an introductory explanation of these components is given and readers are referred to the work by Hawkins and Ahmad [35] for a detailed description including examples of the intermediate data produced at different stages of the HTM process.

Broadly, the input data I_t at time t is first encoded into a binary representation that captures its semantic meaning. This representation, x_t, is the input to the spatial pooler, the component responsible for learning the spatial patterns in the data and producing a sparse distributed representation (SDR) of it. The produced SDR, $a(x_t)$, is then used as input for the temporal memory. This component learns recurrent sequences of SDRs and produces $\pi(x_t)$, another sparse binary vector predicting the next term (or terms) in the sequence, that is, $a(x_{t+1})$.

SDRs are the neocortically inspired data structure used in HTMs; they represent neurons and their state by means of a binary vector. For such a vector to be an SDR, only a small percentage, typically two percent, of its entries should be active (i.e., set to one). This is based on the fact that the relative number of neurons that are active in the neocortex at any given time is low. SDRs encode

the semantic meaning of the data and have some valuable properties such as fault and noise tolerance, can be easily compressed, and the semantic similarity between two inputs can be quickly determined via a bit comparison, among others [36].

Transforming input data (I_t) such as scalar values, dates, and categorical values into binary arrays (x_t) is achieved by using encoders. They are responsible for determining which entries in the array, or which bits, will be ones and which ones will be zero. This must be done in such a way that the semantic characteristics of the data are captured in its encoding. In this way, encodings for semantically similar data will have a larger number of overlapping bits than dissimilar ones.

Each encoded pattern x_t is transformed into an SDR representing a set of neurons arranged in columns by the spatial pooler. For the purpose of this component, all neurons in a column are treated as a unit since it is assumed that all neurons within a column detect identical feedforward input patterns [37]. Similarly to the input binary array, each column can be either active or inactive. Furthermore, the resulting SDR maintains the semantic properties of the input array; that is, SDRs of input arrays that are semantically dissimilar will also be dissimilar and vice versa. This is achieved by assigning each neuron with a set of potential connections (or synapses) to a random subset of the input array. Each potential synapse has a permanence value associated with it and only when this permanence value is greater than a threshold, the synapse is said to be connected. A neuron is then said to be active in the output SDR if the number of connected synapses to active entries in the input array is greater than a threshold. The learning process in the spatial pooler consists on adjusting these permanence values so that the model learns to represent spatial properties of the input data using SDRs in a way in which the semantic properties of the input vector are maintained.

The SDR produced by the spatial pooler, $a(x_t)$, is fed as input to the temporal memory. In this case, individual cells or neurons within a column are differentiated upon as at any given point in time, a subset of neurons in the active columns will be used to represent the temporal context of the current spatial pattern. Neurons within a column can be in three different states: active, inactive, or predictive. Neurons in a predictive state become active in the next time step if they receive sufficient feedforward input, that is, if their column becomes active. In this way, neurons in such a state are anticipating that they will be active in the next time step, and hence are predicting what the next input may be. The output of the temporal memory, $\pi(x_t)$, can then be interpreted as a prediction for $a(x_{t+1})$, that is, columns that could potentially be active in the next time step. In this way, it is possible to measure how unexpected a given input was and compute an anomaly score by comparing the columns that are active at time t and those that had neurons in a predictive state at time $t-1$. Eq. (1) shows how this score is calculated.

$$s_t = 1 - \frac{\pi(x_{t-1}) \cdot a(x_t)}{|a(x_t)|}. \tag{1}$$

This anomaly score is a measure of how well the system predicted the current pattern in the previous time step. As a result, a score of zero is given in cases in which the current input was successfully predicted, a score of one in cases in which it was not predicted, and a value in between when the pattern was partially predicted. Depending on the data, and particularly on the amount of noise present, this anomaly score may produce too many false positives. To address this issue, Ahmad et al. [2] proposed a method that uses a probabilistic model of the anomaly scores to output the likelihood that the system is in an anomalous state. In this way, rather than thresholding the prediction error directly to determine whether a pattern is anomalous or not, a window of the last W anomaly scores is maintained and the distribution is modeled as a normal distribution with the mean and variance continuously updated. The anomaly likelihood L_t is then estimated as a measure of how well the corresponding score fits the given model, it is defined as

$$L_t = 1 - Q\left(\frac{\mu'_t - \mu_t}{\sigma_t}\right) \tag{2}$$

where

$$\mu_t = \frac{\sum_{i=0}^{W-1} s_{t-i}}{W}, \tag{3}$$

$$\sigma_t^2 = \frac{\sum_{i=0}^{W-1} (s_{t-i} - \mu_t)^1}{W}, \tag{4}$$

$$\mu'_t = \frac{\sum_{i=0}^{W'-1} s_{t-i}}{W'}, \tag{5}$$

and W' is a window for a short term moving average with $W' \ll W$.

Finally, a threshold on the anomaly likelihood determines whether an anomaly was detected or not. In particular, an anomaly is detected if $l_t \geq 1 - \epsilon$ where ϵ is a user-defined parameter.

4. Performance anomaly detection using HTM

This work considers workflows that are modeled as graphs and are composed of a set of tasks and a set of dependencies between them. We propose a model in which running workflow tasks are continuously monitored by measuring the amount of computational resources they consume throughout their execution in a particular machine. Specifically, we require CPU and I/O consumption metrics and that the time interval between measurements τ be a configurable parameter. A smaller value will translate in a larger amount of resources (e.g., network, processing capacity) used to make predictions whereas a larger value may lead to a delay in the detection of performance issues. In this work, we achieve this by using the Pegasus [38] WMS. In fact, we use the monitoring functionality added to Pegasus as part of the Panorama [39] project and used by Gaikwad et al. [34] in their anomaly detection work. The system uses a job wrapper called Kickstart [40] to collect the resource usage metrics for each invocation of a task. The monitoring is done at a task level, not at a host or machine level, and hence measurements correspond to the independent consumption of resources by each executing task. The specific metrics that are collected are outlined in Table 1. For further details on the implementation of the monitoring mechanisms we refer the readers to the work by Juve at al. [41]. To meet the online and continuous processing requirements, we modified the system so that monitoring data is published to a RabbitMQ queue (instead of an InfluxDB database), from where our anomaly detection framework consumes the time series measurements.

We implemented an anomaly detection module that is independent from the WMS and the monitoring system. Although it can currently interface with RabbitMQ and InfluxDB, different adaptors can be easily plugged in and the only requirement is for a continuous input of time series data structured in the following manner. Each data point is a tuple of the form (t, e, c) where t is a timestamp, e is the time since the start of the task (or since the first measurement was taken), and c is a measure of the resource consumption at time t. The overall system architecture is depicted in Fig. 3. We used the Numenta Platform for Intelligent Computing (NuPIC) implementation of HTM, specifically, we used the Java version of it.[1]

[1] https://github.com/numenta/htm.java.

Table 1
Resource consumption metrics used to detect performance anomalies.

	Metric	Description
CPU	utime	Time spent in user code
	stime	Time spent in kernel code
I/O	read_bytes	Number of bytes read from disk
	write_bytes	Number of bytes written to disk
	wchar	Number of bytes read using any read-like system call
	rchar	Number of bytes written using any write-like system call
	iowait	Time spent waiting on I/O

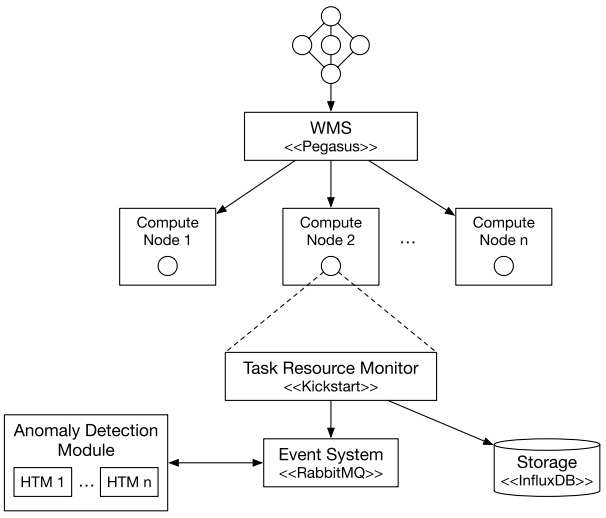

Fig. 3. Anomaly detection using multiple independent Hierarchical Temporal Memory models.

Algorithm 1 Anomaly Detection using HTM

```
1:  procedure DETECTANOMALIES
2:      L_t = empty set of all anomaly likelihoods at time t
3:      A_t = empty set of all anomalies detected at time t
4:      while there are unprocessed measurements in the message queue do
5:          M_t = fetch new measurement
6:          for each metric m ∈ M_t do
7:              HTM = find HTM model corresponding to w, task, and m
8:              e = t − t_0
9:              d = (t, e, m)
10:             inference i = HTM.process(d)
11:             s_t = i.anomalyScore
12:             l_t = anomalyLikelihood(s_t)
13:             L_t.add(l_t, m, w, task)
14:             if l_t ≥ 1 − ϵ then
15:                 A_t.add(l_t, m, w, task)
16:             end if
17:         end for
18:     end while
19: end procedure
```

Rather than combining multiple metrics to create a multi-dimensional model, we create multiple single-metric models for each task in the workflow. Although there may be some correlation between different I/O related metrics for example, we argue that (i) a single high anomaly likelihood among all the metrics is enough to identify an event that may require attention, (ii) since the number of metrics we consider is not large, the anomaly score of other metrics can be easily used to confirm this, and (iii) being able to identify the specific metric, or metrics, causing an anomaly may facilitate the process of diagnosing and creating an action plan to mitigate its effects. Furthermore, we confirmed experimentally that combining multiple metrics in a single model does not improve the ability of the system to detect anomalies but rather, has the opposite effect.

With the aim of having a single HTM model per metric for each task and of potentially reusing models across multiple equivalent workflows, we assume workflows are deployed on homogeneous compute resources. In cloud computing for example, this translates in using a single VM type to deploy the workflow tasks. The reason for this is that the consumption value for each metric is dependent on the hardware, virtualized or not, in which the task is executing. Each model consists then of an encoder, a spatial pooler, a temporal memory, and an anomaly detection module. The encoder consolidates three values: timestamp, time since the start of the task, and the resource consumption metric at that specific time. Specifically, we encode the time of the day and day of week data contained in the timestamp by using NuPIC's date encoder and the remaining two values as scalars using NuPICs random distributed scalar (RDS) encoder. The three encodings are then concatenated into a single one using NuPIC's multi encoder. Details and examples of these three encoders are explained in the work by Purdy [42].

We chose to encode the timestamp in order to capture the dynamicity of the infrastructure. In shared environments for example, it is not unusual to see an increased demand for resources in certain days or times of a day. The time since the start of the task is encoded so that the model learns the amount of a specific resource that a task normally consumes at different stages of its execution. The corresponding HTM model will also use this information to learn normal sequential resource consumption patterns as the execution of the task progresses. For example, it may be common for the amount of time spent in user code (*utime*) to rise by one minute every monitoring interval, any deviation from this pattern may indicate an issue such as resource contention due to a co-located CPU intensive load on the same compute node.

At each time step, for each executing task, the (accumulated) metrics at time t are collected by the task wrapper (Kickstart) and published to a RabbitMQ queue. The measurement data published is then a tuple of the form $M_t = (t, w, task, m_1, m_2, \ldots, m_n)$, where w is the workflow identifier, *task* the task identifier, and m_i is a metric value. The anomaly detection module consumes the new data point from RabbitMQ and processes it in the following way. Firstly, for each metric, it looks for an existing and active HTM model corresponding to w and *task*. If no corresponding HTM model has been created yet, then a new model is created. If there is an existing HTM model but it is inactive (e.g., persisted), then the model is loaded so that it can be used with the incoming data.

After the corresponding HTM model is identified, e (the time elapsed since the first measurement was taken) is estimated as the difference between the timestamp of the metric (t) and the timestamp of the first recorded metric for the given task. The tuple $(t, e, metric)$ is then passed to the HTM model to be processed. The HTM model outputs an inference with an anomaly score (s_t), which is then used to estimate the anomaly likelihood (L_t) for the given data point. Finally, the likelihood is assessed against the predefined threshold to determine whether an anomaly was detected or not. At the end of each time step then, the system outputs n anomaly likelihoods and has classified each of them as anomalous or normal. This process is depicted in Algorithm 1 . Note that the aim of the pseudo code is to detail the logic of the process but not its actual implementation. Specifically, the processing of each new measurement for different metrics is not done sequentially but rather in parallel by making use of threads.

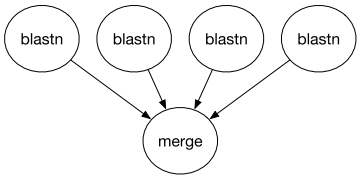

Fig. 4. BLAST workflow.

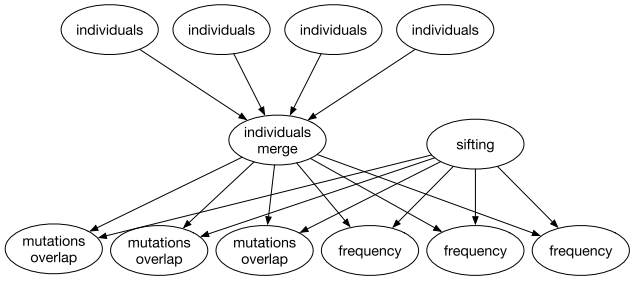

Fig. 5. Sample 1000Genome workflow.

Table 2
Summary of datasets.

Workflow	Task name	Tasks per workflow	Total tasks
BLAST	blastn	4	1052
1000Genome	Individuals	10	1600
	Individuals merge	1	160
	Sifting	1	160
	Mutations overlap	7	1120
	Frequency	7	1120

5. Performance evaluation

This section presents the evaluation of the proposed HTM-based anomaly detection approach with two workflows from the bioinformatics field. The first workflow uses Basic Local Alignment Search Tool (BLAST) to find regions of similarity between biological sequences. In particular, we used the *blastn* application, which searches a nucleotide database using a nucleotide query. The database used was env_{nt}, which contains the nucleotide sequences for metagenomes. We split the database in four equal subsets, each of which was processed by a different *blastn* task in parallel. The structure of the BLAST workflow used is depicted in Fig. 4.

The second workflow is based on the data collected by the 1000 Genomes Project [43]. The project reconstructed the genomes of 2504 individuals across 26 different populations and provides a reference for human variation. Specifically, we used an existing Pegasus workflow that analyses the project data [44], we refer to it as 1000Genome from here on. Its structure is shown in Fig. 5. The workflow measures the overlap in mutations among pairs of individuals by population and the frequency of overlapping in mutations. For our evaluation, we used a workflow to process one chromosome across five populations. Namely we analyzed the data corresponding to chromosome 21 across the African (AFR), Mixed American (AMR), East Asian (EAS), European (EUR), and South Asian (SAS) populations. The number of tasks of each type of the resulting workflow are depicted in Table 2.

5.1. Experiment setup

The evaluation was done by using resources from Microsoft Azure. For each of the applications evaluated, an HTCondor cluster composed of homogeneous VMs was set up, which was in turn used by Pegasus to execute the workflow tasks. Considering the number of tasks and the computational requirements of each workflow, we used two different cluster settings for each application. In both cases, the VMs used were of type *Standard DS13-2* with 2 virtual CPUs, 56 GB of RAM, 25600 max IOPs, and a local SSD storage of 112 GB. The clusters differed on the number of worker VMs used, 2 worker VMs were used for BLAST and 4 for 1000Genome. In both cases, two additional VMs were deployed, one with RabbitMQ and InfluxDB installed and another one that acted as the Pegasus master and had the anomaly detection module deployed on it.

Table 3
Parameter values used for the encoders in the evaluation.

Encoder	Parameter	Value
RDS encoder	Size of the array (w)	50
	Number of on bits (n)	3
	Resolution	0.01
Date encoder	Time of day width	5
	Time of day radius	4
	Day of week width	1
	Day of week radius	1

For each application, multiple identical workflows were continuously and sequentially executed. The resource consumption metrics for each running task were collected every 15 s for BLAST and every 30 s for 1000Genome. The BLAST workflow was executed 263 times and the 1000Genome one 160 times, Table 2 summarizes the total number of tasks executed for each of the workflows. We used the *stress* [45] benchmark to inject anomalies to specific VMs in our system. This benchmark is a workload generator that allows for the amount of CPU, memory, I/O, and disk stress to be configured.

Regarding the HTM models, there are a number of parameters that are configurable for each of the core algorithms. However, there is a standard set that has been demonstrated to work well for detecting anomalies in time series data of scalar metrics [2]. These parameters and their values as used in the evaluation are shown in Table 4. The encoders used also require some parameters to be configured and their values are summarized in Table 3.

Furthermore, we define $\epsilon = 10^{-4}$. As a result, the anomaly likelihoods must be equal to or greater than 0.9999 to be classified as an anomaly. For better visualization, all of the graphs presented in this section are log-scale plots of the anomaly likelihoods, with 0.4 being the threshold equivalent to 0.9999. Furthermore, from here on, the term anomaly likelihood will refer to the log-scale representation of the anomaly probabilities. This value was chosen based on Numenta's practical experience with various types of applications in which $\epsilon = 10^{-4}$ is used to classify anomalies as somewhat likely whereas $\epsilon = 10^{-5}$ is used to classify them as highly likely [2].

5.2. Evaluation results

The experiment results were evaluated in two different ways. Firstly, Sections 5.3 and 5.4 present a detailed analysis of the performance of the HTM-based algorithm for the BLAST and 1000Genome workflows respectively. This is followed in Section 5.5 by a performance comparison of the proposed solution with three other anomaly detection algorithms over the dataset generated after performing the proposed experiments.

5.3. Evaluation results for BLAST

We introduced various anomalies throughout the workflow runs; in total, four different competing loads were introduced at different stages. This can be seen in Fig. 6 depicting the anomaly

Table 4
Parameter values for the Hierarchical Temporal Memory model used in the evaluation.

Parameter	Value
Numeric value encoder number of buckets	130
Number of columns	2048
Number of active columns	40
Spatial pooler connection threshold	0.2
Spatial pooler permanence increment	0.003
Spatial pooler permanence decrement	0.0005
Number of cells per column	32
Dendritic segment activation threshold	13
Maximum number of segments per cell	128
Maximum number of new synapses at each step	32
Temporal memory initial synaptic permanence	20
Temporal memory permanence increment	0.21
Temporal memory permanence decrement	0.1
Spatial value tolerance	0.05
Anomaly detection W	8000
Anomaly detection W'	10
Anomaly detection ϵ	10^{-4}

likelihoods for the utime metric obtained throughout the executions of two of the BLAST workflow tasks. For all the monitored metrics, the anomaly likelihood is close to zero during non-anomalous runs. The first anomaly was a CPU-intensive workload introduced during the 99th workflow run, the second one an I/O intensive workload introduced in run number 162, the third load was memory-intensive, and finally the fourth one was a combination of the previous three, this is shown in Table 5 in addition to the tasks that were affected by each of the anomalous workloads.

Fig. 7 shows the utime and stime metrics before, during, and after the CPU anomaly introduction for one of the *blastn* tasks. It also depicts the corresponding anomaly likelihoods. For both metrics, the anomaly likelihoods climb to their maximum value of 1 during the anomalous run; however, the threshold of 0.4 is first reached by the anomaly probability associated withe the stime metric. This occurs 530 s into the execution of the task, which is approximately halfway through.

Fig. 8 shows the iowait, rchar, and wchar metrics during the I/O anomaly for another *blastn* task. The likelihoods quickly climb to one soon after the anomalous workload begins affecting the execution of the task and the resource consumption patterns begin to change. For the iowait metric, the anomaly threshold is first exceeded with a value of 0.87 at approximately 12:23, one minute after the I/O intensive load was introduced. The likelihood values then return to a stable pattern close to 0 at 12:51 around the time the anomalous load completed execution. As seen in the chart, both of the other metrics, rchar and wchar, exhibit very similar behaviors to iowait.

The read and write bytes metrics and corresponding anomaly probabilities during the memory-intensive workload are shown in Fig. 9. The anomaly is first detected in the read_bytes metric at time 6:53 with a likelihood of 0.43. Although not detected as early as in the case of the I/O anomaly, the detection still happens approximately halfway through the execution of the task (756 s elapsed out of a makespan of 1649 s), which can still be useful to resource management modules. Anomalies continue to be flagged until the execution of the task concludes, alerting to a resource consumption pattern different from the normal one, as can be clearly seen in the chart. The anomaly is also detected in the write_bytes metric, however this occurs almost five minutes later, at time 6:59. In the case of both metrics, false positives occur at the beginning of the workflow run 180. After analyzing the data, a possible contributing factor may be that the resource measurements were taken at different time intervals for this run. Specifically, the second measurement was taken 10 s after the first one, as opposed to the configured 15 s interval. This affects the time elapsed since the execution of the task for consecutive measurements which in turn affects the encoding of the data fed into the HTM system.

Finally, Fig. 10 shows the anomaly probabilities associated with the stime, utime, wchar, and rchar metrics around the time the combined anomaly was introduced. The anomaly likelihood for stime is the first one to exceed the threshold at approximately 1:56, with a probability of 0.77. This is 674 s into the execution of the task, which completed after 2774 s. For most metrics, as the anomalous pattern extends for close to one hour, the anomaly likelihoods gradually descend at around 2:06 as HTM learns the sequence pattern as a new normal. This leads to a brief spike in the anomaly probabilities at the beginning of a new workflow run, with false positives shortly occurring in the first measurements of run 210. However, HTM quickly recovers from this and recognizes the normal consumption pattern observed in the previous runs.

5.4. Evaluation results for 1000Genome

Similarly to the evaluation of BLAST, multiple anomalies were introduced throughout the 1000Genome workflow runs; in fact, the same types of anomalous workloads were used for both use cases. We depict the resource consumption and anomaly likelihood charts for four different metrics (utime, iowait, rchar, and write_bytes) during, before, and after the combined anomaly was introduced at the beginning of the execution of the 144th run of the 100Genome workflow. The results are shown in Fig. 11 and correspond to the execution of task *individuals_06* which was deployed on the worker VM where the anomaly was introduced.

The performance anomaly is first detected in the utime, rchar and write_bytes metrics at approximately the same time (9:49) while the anomaly likelihood associated with the iowait metric takes longer to reach the 0.4 threshold. The probability values continue to increase quickly until they reach their maximum value of one and hence the anomaly is clearly identified in the four cases. Even though the anomaly is detected during the first half of the task execution, it does take a few minutes before the likelihoods begin to increase. One of the causes contributing to this delay is the fact that the anomaly likelihoods are computed as the probability of a short-term average of the raw anomaly scores. These raw scores begin to rise earlier in the anomaly window but it takes several of these higher scores to occur before the anomaly likelihood begins to rise. Despite this, we still found the raw scores to be too unpredictable making it hard to establish a threshold for them. It would be of interest to further explore this issue in future work as well as exploring other options that may lead to a faster detection such as different HTM parameters or different encodings.

It is also observed that at the beginning (the first five measurements) of the execution of the workflow run 145, the likelihood values climb above the threshold in three out of the four metrics, but quickly drop to values close to zero after HTM recognizes the consumption pattern. This in fact has been a common occurrence observed multiple times throughout the performance evaluation and hence, it should be considered when embedding autonomous performance anomaly recovery mechanisms in schedulers and resource managers, at least for this specific type of scientific applications. Finally, another important insight from the obtained results is the potential benefit of aggregating the anomaly likelihoods of multiple metrics in order to get a more comprehensive measurement of the state of the running tasks and possibly detect anomalies earlier and better prevent false positives.

5.5. Algorithm performance comparison

To demonstrate the suitability of HTM to the problem defined in this work and to corroborate its superiority in this particular scenario over other algorithms, in this section we present

Table 5
Anomalous workloads used for BLAST.

Type	Processes	Time	Workflow run	Duration (s)	Affected tasks
CPU	10 CPU-bound	10/29 12:23–12:53	99	1800	blastn_01/02
I/O	100 I/O-bound	10/30 12:22–12:52	162/163	1800	blastn_03/04
VM	50 VM-bound	10/30 06:41–07:11	179	1800	blastn_01/02
ALL	5 CPU-, 50 I/O-, 25 VM-bound	10/31 01:45–02:45	209	3600	blastn_01/02

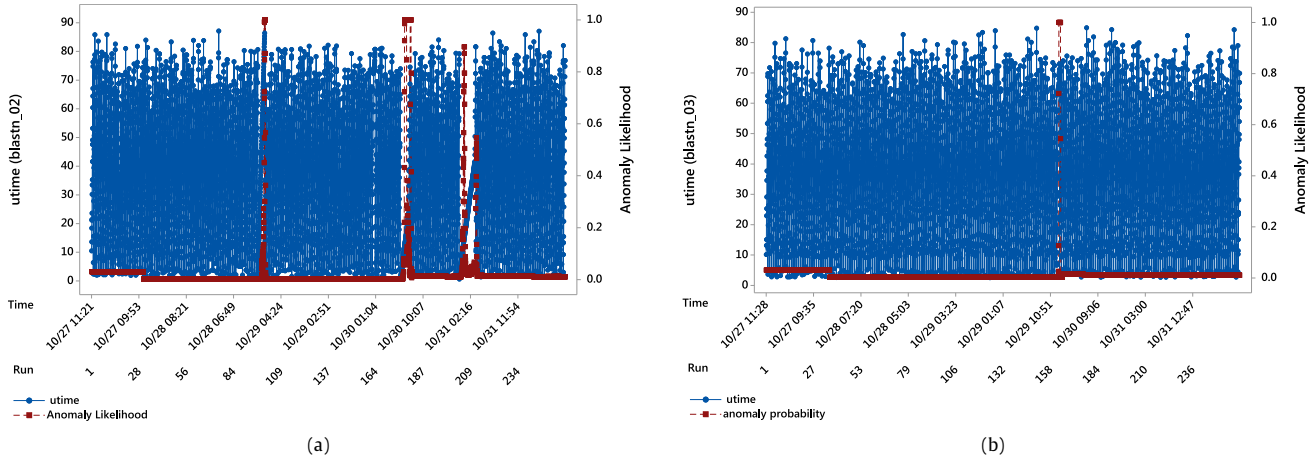

Fig. 6. Plots of the recorded utime measurements and their corresponding log-scale anomaly likelihoods for two different BLAST tasks. (a) utime for task blastn_02. (b) utime for task blastn_03.

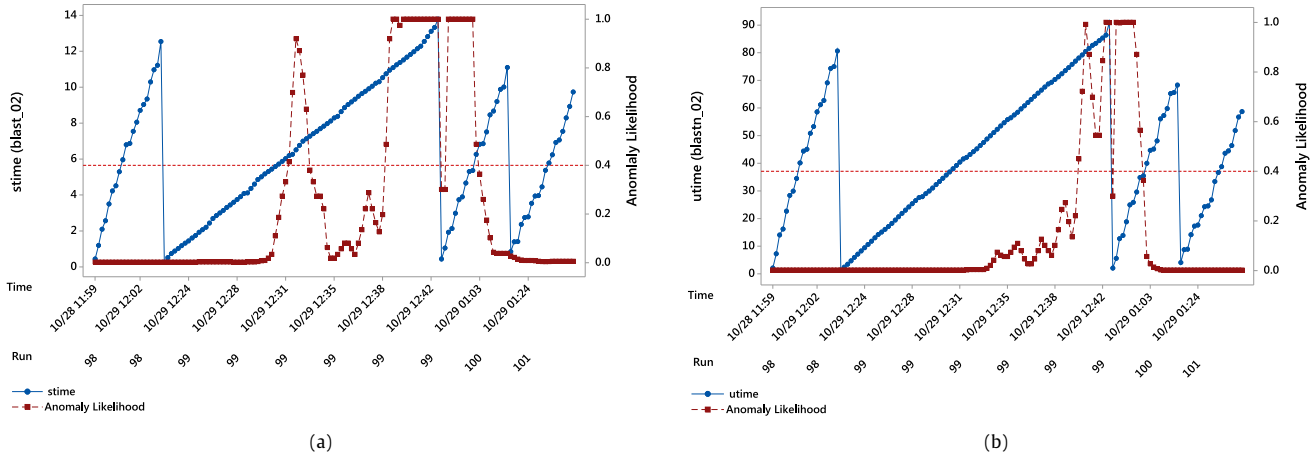

Fig. 7. Plots of the stime and utime metrics and their corresponding log-scale anomaly likelihoods for task *blastn_02* around the time the CPU anomaly was introduced during the execution of the BLAST workflows. (a) stime. (b) utime.

a performance comparison of the HTM-based solution to other online anomaly detection algorithms. In particular, the Numenta Anomaly Benchmark (NAB) [2] was used to score and compare the performance of the HTM-based anomaly detection algorithm with three other approaches, namely KNN-CAD [12], Bayesian Online Checkpoint Detection [11] (BOCD), and a Sliding Threshold [46] (ST) approach.

NAB is a framework designed to evaluate anomaly detection algorithms in streaming applications. The open source implementation of NAB [47] includes the implementation of the algorithms used for comparison in this Section as well as the implementation of a scoring mechanism. Overall, this mechanism favors detections (anomaly scores above a predefined threshold) that are within an anomaly window and penalizes those outside a window. More importantly, the scoring system in NAB rewards early detection of anomalies. This is achieved by using a set of predefined anomaly windows to identify and weight true positives (TP), false positives (FP), and false negatives (FN) (true negatives (TN) do not impact the final score). Based on these, a sigmoidal scoring function gives detections earlier in the window higher positive scores than those later in the window and detections that appear slightly after the anomaly window are penalized less (with lower negative scores) than those that occur further away. We refer readers to the work by Ahmad et al. [2] for a detailed description of this scoring function.

The dataset used to evaluate the algorithms corresponds to the data collected during the execution of the experiments as previously outlined in this Section. For each of the *blastn* tasks in the BLAST workflow and the *individuals* tasks in the 1000Genomes application, the timestamp and measurement values at each time step for each of the metrics was recorded. NAB was then used to obtain an anomaly score or likelihood for each entry in these time series data for the KNN-CAD, BOCD, and ST algorithms. The anomaly scores for the HTM-based algorithm correspond to those

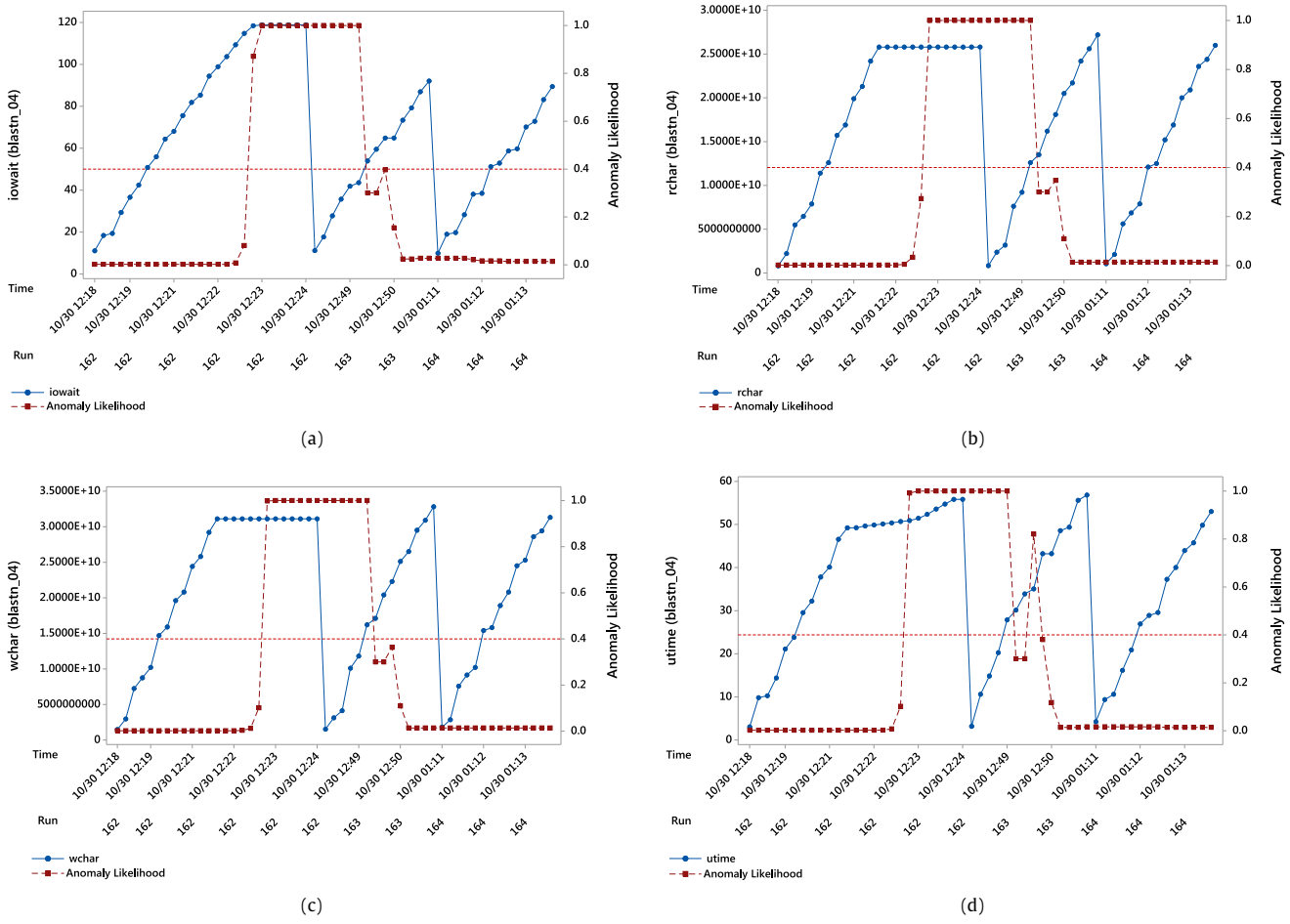

Fig. 8. Plots of the iowait, rchar, wchar, and utime metrics and their corresponding log-scale anomaly likelihoods for task *blastn_04* around the time the I/O anomaly was introduced during the execution of the BLAST workflows. (a) iowait. (b) rchar. (c) wchar. (d) utime.

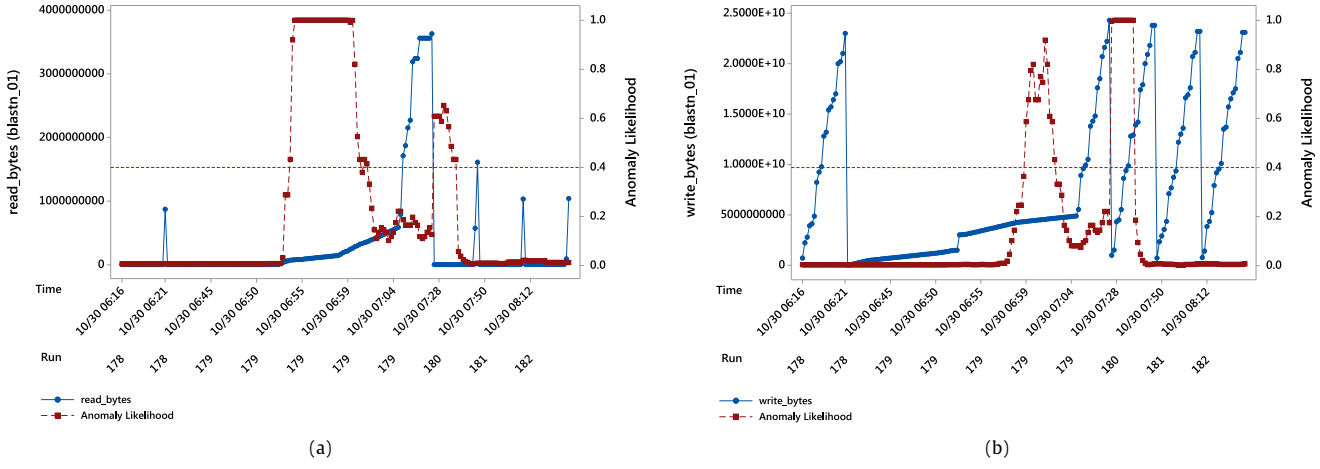

Fig. 9. Plots of the read_bytes and write_bytes metrics and their corresponding log-scale anomaly likelihoods for task *blastn_01* around the time the VM anomaly was introduced during the execution of the BLAST workflows. (a) read_bytes. (b) write_bytes.

collected online throughout the experiment lifecycle and were not estimated using NAB.

To determine whether an anomaly score is classified as a detection or not, a threshold is used for each of the algorithms. These anomaly scores along with the time windows when the anomalies were introduced as outlined in Table 5 and Section 5.4 were used by NAB to generate the overall performance scores. We used NAB to optimize the threshold values for each algorithm so that their final scores were maximized. This values are shown in Table 6. For the HTM-based algorithm we depict the NAB scores for both the optimized threshold (HTM-OT) and for the threshold used in the evaluation of the online framework (HTM). It is worthwhile mentioning that this optimized threshold was not used when evaluating our framework in the previous sections as determining

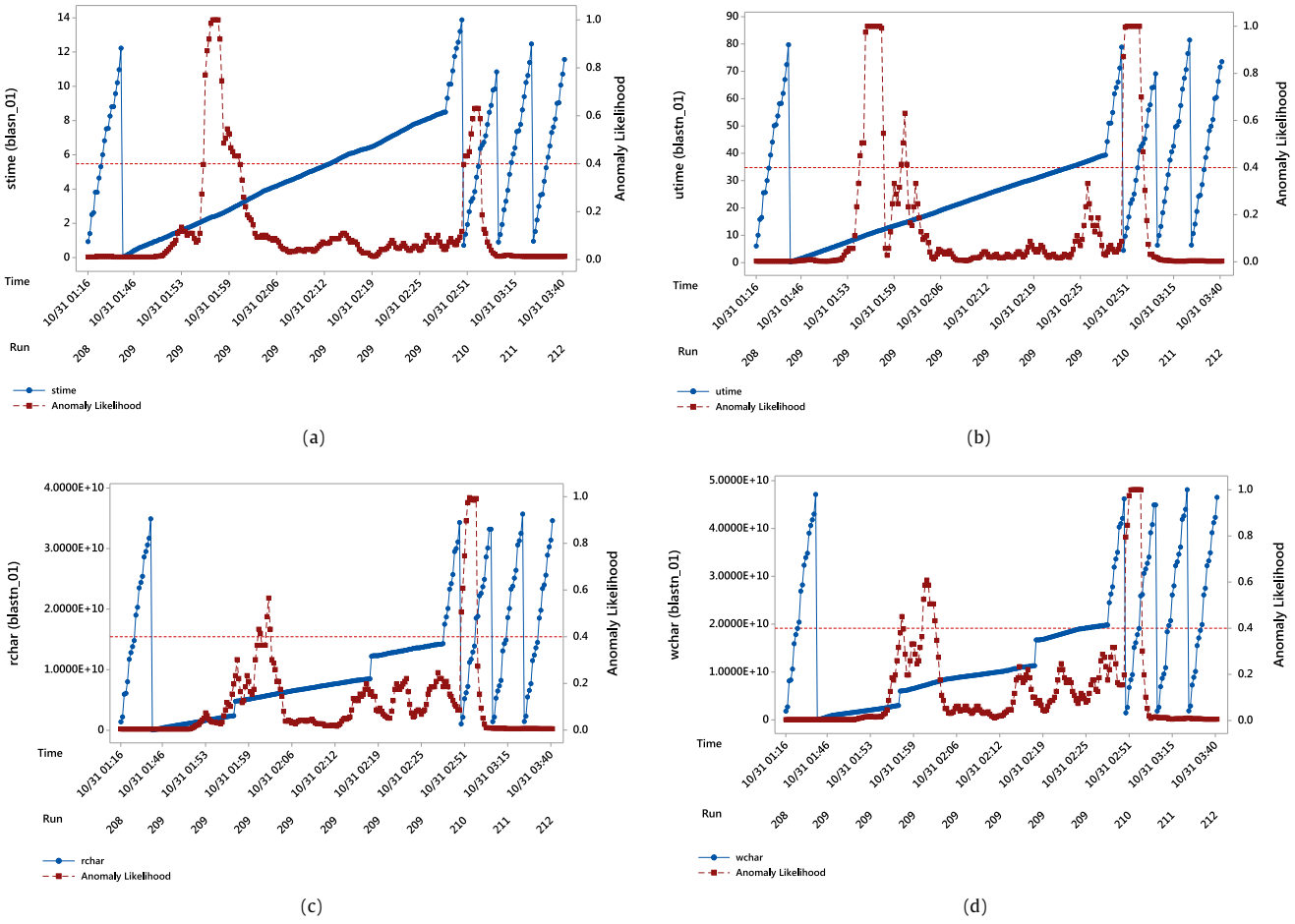

Fig. 10. Plots of the stime, utime, rchar, and wchar metrics and their corresponding log-scale anomaly likelihoods for task *blastn_01* around the time the combined (CPU, I/O, and VM) anomaly was introduced during the execution of the BLAST workflows. (a) stime. (b) utime. (c) rchar. (d) wchar.

Table 6
NAB Thresholds for the BLAST dataset.

Algorithm	Reward low FN	Reward low FP	Standard
HTM	0.40	0.40	0.40
HTM-OT	0.07	0.12	0.07
KNN-CAD	0.99	0.99	0.99
ST	1.0	1.0	1.0
BOCD	1.1	1.1	1.1

its optimal value requires a priori knowledge of the anomaly scores and windows for the entire dataset. The proposed online framework does not accommodate this assumption and such data was only available after the experiments completed their execution.

All the parameter values used for BOCD, KNN-CAD, and ST correspond to those used in the NAB benchmark. These can be found online in the algorithms implementation [47] and are also described in the supplementary material of the work by Ahmad et al. [2].

Three different NAB scores (scores range between 0 and 100) are displayed in Table 7. The first two are called *Reward Low FP* and *Reward Low FN*; they enforce higher penalties for false positives and false negatives respectively. The third one referred to as *Standard* assigns true positives, false positives, and false negatives with relative weights such that random detections made 10% of the time would get a zero final score on average [2]. Finally, the number of true positives, true negatives, false positives, and false negatives corresponding to the *Standard* score and threshold are also displayed as a reference to readers in Table 8.

The results demonstrate that HTM outperforms the other algorithms on each of the three different scores. What is more, not only is this achieved with an optimized threshold, but also with a generic threshold that was not tailored for the specified problem. KNN-CAD achieves the second largest scores, followed by ST, and BCOD respectively. The vast difference between the HTM scores and the BCOD ones may be due to the fact that BCOD assumes an underlying distribution of the data; demonstrating the benefits of a non-parametric approach for this specific problem. Finally, when compared to KNN-CAD, the number of true positives obtained with HTM is considerably larger. However, KNN-CAD has a lower incidence of false positives. This may be explained by the pattern seen in Sections 5.3 and 5.4 that show that it is common for HTM likelihoods to be beyond the threshold for a few iterations after the anomaly window.

It is worthwhile mentioning that for BOCD, the optimal threshold is larger than one and hence, the algorithm performs best on our particular dataset when in makes no anomaly detections at all. This behavior is equivalent to that of the *null* detector in NAB, which is used to normalize the scores to obtain values between 0 and 100. Threshold values equal or less than one for BOCD led to the number of false positives degrading the performance of the algorithm and hence the NAB score below 0.

Finally, the algorithms were also evaluated using the 1000Genome dataset. The performance of the HTM-based detector was again better than that of the other approaches based on the NAB scores. The Standard Scores for HTM, HTM-OT, KNN-CAD, ST, and BOCD where 67.91, 82.38, 20.78, 21.20, and 52.11 respectively.

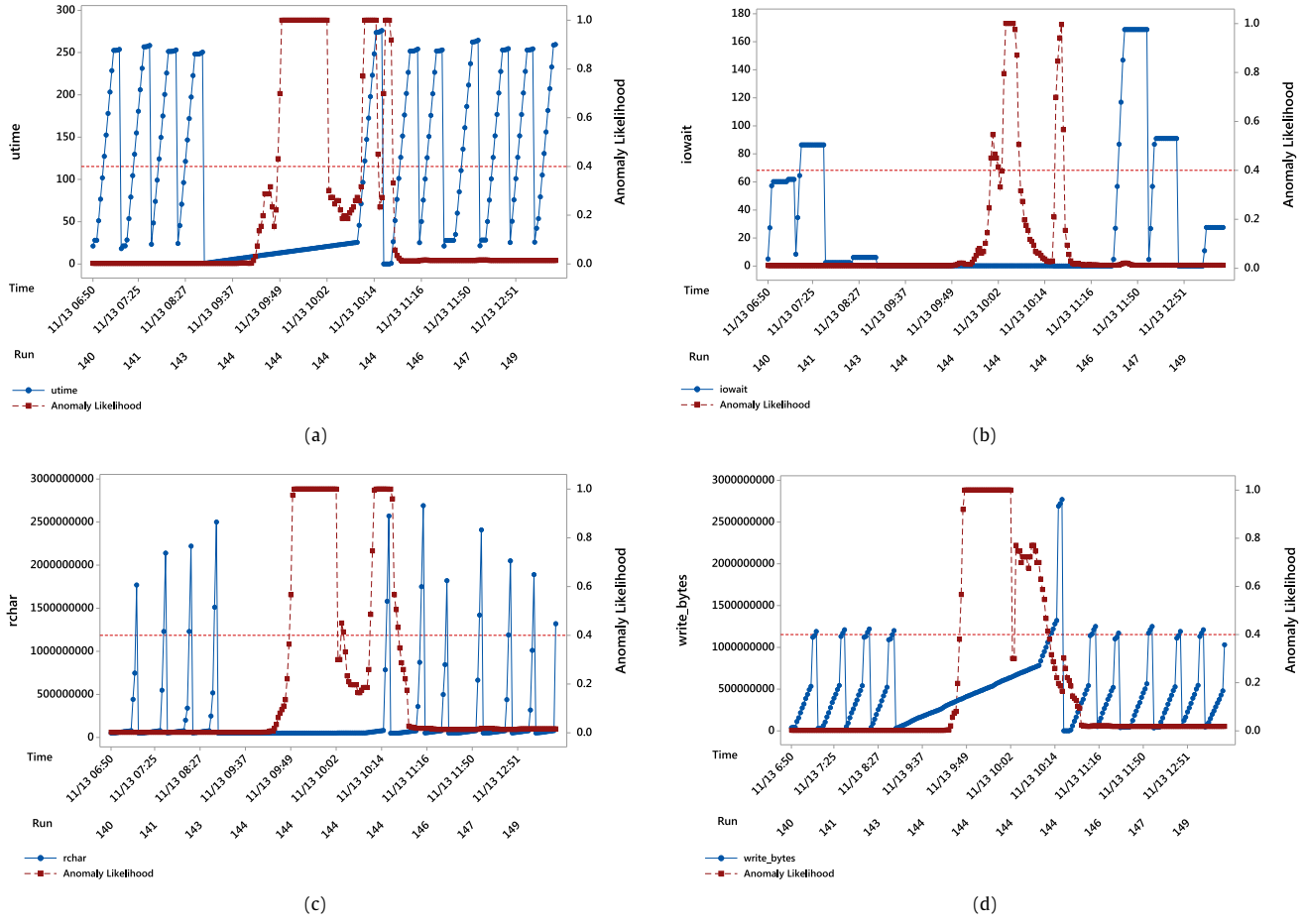

Fig. 11. Plots of the utime, iowait, rchar, and write_bytes metrics and their corresponding log-scale anomaly likelihoods for task *individuals_06* around the time the combined (CPU, I/O, and VM) anomaly was introduced during the execution of the 1000Genome workflows. (a) utime. (b) iowait. (c) rchar. (d) write_bytes.

Table 7
NAB scores for the BLAST dataset.

Algorithm	Reward low FN	Reward low FP	Standard
HTM	74.21	63.62	69.35
HTM-OT	85.82	71.67	80.51
KNN-CAD	72.43	52.92	66.67
ST	5.40	3.55	4.53
BOCD	0.0	0.0	0.0

Table 8
TP, TN, FP, and FN associated with the standard score for each algorithm on the BLAST dataset.

Algorithm	TP	TN	FP	FN
HTM	778	107 727	332	4927
HTM-OT	2034	107 570	489	3671
KNN-CAD	47	107 919	140	5658
ST	18	108 049	10	5687
BOCD	0	108 059	0	5705

6. Conclusions and future work

In this work, we used HTM to detect performance anomalies in the execution of scientific workflows in distributed computing environments such as clouds. The proposed method is an online approach that can be deployed on different infrastructures without the need of previously collecting data for training purposes. Instead, HTM enables the framework to learn incrementally and detect anomalies in an unsupervised manner while adjusting to changes in the statistics of the data. The data analyzed corresponded to resource consumption metrics of executing workflow tasks and was processed in an online manner, as it became available. We evaluated the HTM-based technique in a cloud environment using two scientific workflows from the bioinformatics domain and the Pegasus workflow management system. The results demonstrate the ability of the proposed technique to detect anomalies on different resource consumption metrics caused by external loads with different characteristics.

As future work, we plan to further investigate how anomaly likelihoods that arise from different metrics can be aggregated to detect and potentially diagnose anomalies earlier and more accurately. We will also develop scheduling and resource provisioning techniques that make use of the information collected by the anomaly detection framework; initially we will explore replicating or rescheduling tasks exhibiting anomalous behaviors on different compute nodes, avoiding the use of machines that consistently lead to anomalous behavior in tasks, and provisioning new resources to minimize the impact on the makespan of the workflow when anomalies are detected in tasks on the critical path of the workflow. Finally, we plan to explore how the proposed HTM-based anomaly detection framework can be utilized in emerging fog platforms as these environments and the latency-sensitive applications they host can greatly benefit from a lightweight framework capable of detecting anomalies in the performance of multiple tasks continuously and in an online fashion.

References

[1] M. Putic, A. Varshneya, M.R. Stan, Hierarchical temporal memory on the automata processor, IEEE Micro 37 (1) (2017) 52–59.
[2] S. Ahmad, A. Lavin, S. Purdy, Z. Agha, Unsupervised real-time anomaly detection for streaming data, Neurocomputing (2017).
[3] Netflix, Netflix's robust anomaly detection (rad). URL https://medium.com/netflix-techblog/rad-outlier-detection-on-big-data-d6b0494371cc.
[4] E.J. Candès, X. Li, Y. Ma, J. Wright, Robust principal component analysis, J. ACM 58 (3) (2011) 11.
[5] E. Keogh, J. Lin, A. Fu, Hot sax: Efficiently finding the most unusual time series subsequence, in: 5th International Conference on Data Mining, IEEE, 2005, pp. 8–pp.
[6] P. Malhotra, L. Vig, G. Shroff, P. Agarwal, Long short term memory networks for anomaly detection in time series, in: 23rd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Presses universitaires de Louvain, 2015, p. 89.
[7] H.N. Akouemo, R.J. Povinelli, Probabilistic anomaly detection in natural gas time series data, Int. J. Forecast. 32 (3) (2016) 948–956.
[8] A.M. Bianco, M. Garcia Ben, E. Martinez, V.J. Yohai, Outlier detection in regression models with arima errors using robust estimates, J. Forecast. 20 (8) (2001) 565–579.
[9] M. Schneider, W. Ertel, F. Ramos, Expected similarity estimation for large-scale batch and streaming anomaly detection, Mach. Learn. 105 (3) (2016) 305–333.
[10] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, K. Schwan, Statistical techniques for online anomaly detection in data centers, in: IFIP/IEEE International Symposium on Integrated Network Management (IM), IEEE, 2011, pp. 385–392.
[11] R.P. Adams, D.J. MacKay, Bayesian online changepoint detection, arXiv preprint arXiv:0710.3742, 2007.
[12] E. Burnaev, V. Ishimtsev, Conformalized density-and distance-based anomaly detection in time-series data, arXiv preprint arXiv:1608.04585, 2016.
[13] G. Khangamwa, Detecting network intrusions using hierarchical temporal memory, in: International Conference on e-Infrastructure and e-Services for Developing Countries, Springer, 2010, pp. 41–48.
[14] G.M. Bonhoff, Using hierarchical temporal memory for detecting anomalous network activity, Tech. rep., Air Force Institute of Technology, 2008.
[15] V. Berger, Anomaly detection in user behavior of websites using hierarchical temporal memories: Using machine learning to detect unusual behavior from users of a web service to quickly detect possible security hazards, 2017.
[16] M. Otahal, O. Stepankova, Anomaly detection with cortical learning algorithms for smart homes, Smart Homes, 20144, 24.
[17] Numenta, Geospatial tracking - learning the patterns in movement and detecting anomalies. URL https://numenta.com/assets/pdf/whitepapers/GeospatialTrackingWhitePaper.pdf.
[18] Numenta, Detecting anomalies in publicly traded stocks using trading and twitter data. URL https://numenta.com/assets/pdf/apps/htmforstocks.pdf.
[19] A.R. Boone, T. Karnowski, E. Chaum, L. Giancardo, Y. Li, K. Tobin, Image processing and hierarchical temporal memories for automated retina analysis, in: Biomedical Sciences and Engineering Conference (BSEC), IEEE, 2010, pp. 1–4.
[20] Grok. URL http://grokstream.com/.
[21] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, E. Vázquez, Anomaly-based network intrusion detection: Techniques, systems and challenges, Comput. Secur. 28 (1) (2009) 18–28.
[22] T. Wang, J. Wei, W. Zhang, H. Zhong, T. Huang, Workload-aware anomaly detection for web applications, J. Syst. Softw. 89 (2014) 19–32.
[23] S. Rajasegarar, C. Leckie, M. Palaniswami, J.C. Bezdek, Distributed anomaly detection in wireless sensor networks, in: 10th IEEE Singapore International Conference on Communication Systems (ICCS), IEEE, 2006, pp. 1–5.
[24] A. Navaz, V. Sangeetha, C. Prabhadevi, Entropy based anomaly detection system to prevent ddos attacks in cloud, arXiv preprint arXiv:1308.6745, 2013.
[25] F. Doelitzscher, M. Knahl, C. Reich, N. Clarke, Anomaly detection in iaas clouds, in: 5th International Conference on Cloud Computing Technology and Science (CloudCom), Vol. 1, IEEE, 2013, pp. 387–394.
[26] O. Ibidunmoye, F. Hernández-Rodriguez, E. Elmroth, Performance anomaly detection and bottleneck identification, ACM Comput. Surv. 48 (1) (2015) 4.
[27] H. Kang, X. Zhu, J.L. Wong, Dapa: Diagnosing application performance anomalies for virtualized infrastructures., in: 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE), USENIX, 2012.
[28] M. Peiris, J.H. Hill, J. Thelin, S. Bykov, G. Kliot, C. Konig, Pad: Performance anomaly detection in multi-server distributed systems, in: 7th International Conference on Cloud Computing (CLOUD), IEEE, 2014, pp. 769–776.
[29] H. Nguyen, Z. Shen, Y. Tan, X. Gu, Fchain: Toward black-box fault localization for cloud systems, in: 33rd International Conference on Distributed Computing Systems (ICDCS), IEEE, 2013, pp. 21–30.
[30] Y. Tan, X. Gu, H. Wang, Adaptive system anomaly prediction for large-scale hosting infrastructures, in: 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, ACM, 2010, pp. 173–182.
[31] Y. Tan, Online performance anomaly prediction and prevention for complex distributed systems (Ph.D. thesis), North Carolina State University, 2012.
[32] L. Yu, Z. Lan, A scalable, non-parametric anomaly detection framework for hadoop, in: ACM Cloud and Autonomic Computing Conference, ACM, 2013, p. 22.
[33] H.S. Pannu, J. Liu, S. Fu, A self-evolving anomaly detection framework for developing highly dependable utility clouds, in: Global Communications Conference (GLOBECOM), IEEE, 2012, pp. 1605–1610.
[34] P. Gaikwad, A. Mandal, P. Ruth, G. Juve, D. Król, E. Deelman, Anomaly detection for scientific workflow applications on networked clouds, in: International Conference on High Performance Computing & Simulation (HPCS), IEEE, 2016, pp. 645–652.
[35] J. Hawkins, S. Ahmad, Why neurons have thousands of synapses, a theory of sequence memory in neocortex, Front. Neural Circuits 10 (2016) 23.
[36] S. Ahmad, J. Hawkins, Properties of sparse distributed representations and their application to hierarchical temporal memory, arXiv preprint arXiv:1503.07469, 2015.
[37] Y. Cui, S. Ahmad, J. Hawkins, Continuous online sequence learning with an unsupervised neural network model, Neural Comput. (2016).
[38] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, et al., Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Sci. Program. 13 (3) (2005) 219–237.
[39] E. Deelman, C. Carothers, A. Mandal, B. Tierney, J.S. Vetter, I. Baldin, C. Castillo, G. Juve, D. Król, V. Lynch, et al., Panorama: an approach to performance modeling and diagnosis of extreme-scale workflows, Int. J. High Perform. Comput. Appl. 31 (1) (2017) 4–18.
[40] J.-S. Vöckler, G. Mehta, Y. Zhao, E. Deelman, M. Wilde, Kickstarting remote applications, in: 2nd International Workshop on Grid Computing Environments, 2006, pp. 1–8.
[41] G. Juve, B. Tovar, R.F. Da Silva, D. Król, D. Thain, E. Deelman, W. Allcock, M. Livny, Practical resource monitoring for robust high throughput computing, in: International Conference on Cluster Computing (CLUSTER), IEEE, 2015, pp. 650–657.
[42] S. Purdy, Encoding data for htm systems, arXiv preprint arXiv:1602.05925, 2016.
[43] I.T.I.G.S. Resource, 1000 genomes project. URL http://www.internationalgenome.org/about.
[44] Pegasus-isi, 1000genome workflow. URL https://github.com/pegasus-isi/1000genome-workflow.
[45] Stress, URL http://people.seas.harvard.edu/~apw/stress/.
[46] Numenta, Github - nab, windowed gaussian detector. URL https://github.com/numenta/NAB/tree/master/nab/detectors/gaussian.
[47] Numenta, Github - nab. URL https://github.com/numenta/NAB.

Maria A. Rodriguez is a Post-Doctoral Research Fellow in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory in the Department of Computing Information Systems, The University of Melbourne, Australia. Her research interests include resource management and scheduling in clouds and scientific computing.

Ramamohanarao Kotagiri received his Ph.D. from Monash University. He was awarded the Alexander von Humboldt Fellowship in 1983. He has held several senior positions including Head of Computer Science and Software Engineering, Head of the School of Electrical Engineering and Computer Science at the University of Melbourne and Research Director for the Cooperative Research Centre for Intelligent Decision Systems. He has served on the Editorial Boards of the Computer Journal, Universal Computer Science, TKDE, VLDBJ and International Journal on Data Privacy. He is a Fellow of the Institute of Engineers AU, a Fellow of Australian Academy Technological Sciences and Engineering and a Fellow of Australian Academy of Science. He was awarded Distinguished Contribution Award in 2009 by the Computing Research and Education Association of Australasia. He has published more than 350 articles and has over 48 Ph.D. completions.

Rajkumar Buyya is a Professor of Computer Science and Software Engineering and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in Cloud Computing. He has authored over 400 publications and four textbooks. He is one of the highly cited authors in computer science and software engineering worldwide.