2016

A Cloud Service Architecture for Analyzing Big Monitoring Data

Samneet Singh  
*the Department of Electrical and Computer Science, Concordia University, Montreal, Canada.*

Yan Liu  
*the Department of Electrical and Computer Science, Concordia University, Montreal, Canada.*

Follow this and additional works at: [https://tsinghuauniversitypress.researchcommons.org/tsinghua-science-and-technology](https://tsinghuauniversitypress.researchcommons.org/tsinghua-science-and-technology)

Part of the Computer Sciences Commons, and the Electrical and Computer Engineering Commons

**Recommended Citation**

A Cloud Service Architecture for Analyzing Big Monitoring Data

Samneet Singh* and Yan Liu*

Abstract: Cloud monitoring is of a source of big data that are constantly produced from traces of infrastructures, platforms, and applications. Analysis of monitoring data delivers insights of the system’s workload and usage pattern and ensures workloads are operating at optimum levels. The analysis process involves data query and extraction, data analysis, and result visualization. Since the volume of monitoring data is big, these operations require a scalable and reliable architecture to extract, aggregate, and analyze data in an arbitrary range of granularity. Ultimately, the results of analysis become the knowledge of the system and should be shared and communicated. This paper presents our cloud service architecture that explores a search cluster for data indexing and query. We develop REST APIs that the data can be accessed by different analysis modules. This architecture enables extensions to integrate with software frameworks of both batch processing (such as Hadoop) and stream processing (such as Spark) of big data. The analysis results are structured in Semantic Media Wiki pages in the context of the monitoring data source and the analysis process. This cloud architecture is empirically assessed to evaluate its responsiveness when processing a large set of data records under node failures.

Key words: cloud computing; REST API; big data; software architecture; semantic web

1 Introduction

Cloud monitoring is a process of monitoring infrastructures, platforms, and custom metrics to help ensure that workloads are operating at optimum levels[1]. For example, analyzing the memory usage for each task in a cluster helps to determine usage patterns of tasks and allows setting up alerts when certain thresholds are reached to balance the workload and maintain high availability. Cloud monitoring data analysis involves operations for querying and processing a large amount of traces. For an instance, the Google trace is of production workloads running on Google clusters collected over 6.25 hours[2-5]. The dataset contains over 3 million observations (or data points) with the following features: (1) timestamp (in seconds); (2) JobID as unique job identifier; (3) TaskID as unique task identifier; (4) TaskID as unique task identifier; (4) JobType; (5) normalized task cores as normalized number of CPU cores used; and (6) normalized task memory as normalized value of the average memory consumed by the task. In such a trace, data need to be retrieved for particular attributes from an arbitrary range of granularity (such as in minute or hourly aggregation) to estimate the Probability Distribution Function (PDF) and forecast the future workload. Since the volume of traces can be large, these operations require a scalable and reliable cloud architecture as the trace data accumulate.

Existing cloud monitoring services such as CloudWatch, AzureWatch, Nimsoft, and Nimbus are delivered by cloud service providers as part of the Infrastructure as a Service (IaaS) model[1]. They can be launched within an integrated cloud management console. To some extent, these services have a limitation in supporting analysis such as workload
forecast and pattern matching beyond the simple built-in aggregation.

1.1 Enabling technologies

To allow advanced trace data analysis, the envisioned cloud architecture should accommodate the access to arbitrary sets of the trace data. In addition, the cloud architecture needs a web-based interface for building structured queries of data and displaying the results. To address aforementioned architectural needs, we consider a cloud architecture that combines (1) a search-based cluster for monitoring data storage and query; (2) an extension to accommodate large volume of data for analysis and processing; and (3) Semantic MediaWiki (SMW) for structuring the data query and organizing the analysis result display.

Apache Solr is an open source full-text-based search engine. It provides features of faceted search, hit highlighting, and real time indexing. The core of Solr consists of Apache Lucene’s search library that allows distributed search and index replication. Solr also provides a Representational State Transfer (REST) like Application Program Interface (API) that enables language binding with Solr. Based on Solr 4.0, SolrCloud further provides the functionalities of running multiple Solr nodes in a cluster to device a scalable and highly available cloud computing environment[6].

MediaWiki is an open-source application originally developed to power wikipedia. MediaWiki helps to search, organize, tag, browse, evaluate, and share information as the wiki’s content[7]. SMW is an open-source extension developed for MediaWiki. It adds semantic annotations that allow a wiki to function as a collaborative database. SMW provides a flexible means for a developer to describe the trace data he or she intends to analyze. The fields and attributes of the trace data can be defined by SMW annotations, and their relationship helps to form a structured query[8]. Instead of storing the whole set of raw trace data into SMW’s content database, we propose a cloud architecture that stores the trace data in a search cluster using SolrCloud. Queries of the raw data are issued from the SWM page and the results are stored back to SMW’s content database for display.

Apache Spark is an open source cluster computing framework designed to process large amount of data, especially big data. It extends the MapReduce paradigm for complex computation. In contrast to Apache hadoop, it can compute using main memory entailing faster computation. It requires a resource manager and distributed storage system for data processing. Currently, it supports three resource managers, Spark standalone, Apache Yarn, and Apache Mesos. It runs on various distributed storage platforms, such as HDFS, Amazon S3, Cassandra, etc. Apache Spark also provides a unified stack, i.e., integrated components such as Spark SQL, Spark streaming, Machine learning library (Mlib), and GraphX for better data processing. For instance, Mlib provides k-means clustering algorithm to efficiently divide large number of data points into k specified sets.

1.2 The contribution

The main contribution of this paper is the novel architecture and essential techniques that integrate SolrCloud, Apache Spark, and SMW for data intensive trace analysis.

Trace data are stored as shards in the SolrCloud and retrieved by any field for an arbitrary range issued from SMW pages. The data retrieval is further processed by time series analysis packages for workload forecasting and pattern matching. The results are plotted and sent back to an SWM page for display. The glue of these techniques are through REST APIs.

This architecture addresses quality attributes of scalability and availability by leveraging SolrCloud’s shard and replica configuration. The responsiveness of this architecture and its availability upon handling failover are evaluated by experiments on Google trace. The traceAnalyzer architecture also extends Apache Spark running on the top of Apache Yarn. Another copy of trace data is stored in HDFS and can be utilized by Apache Spark for efficient data clustering.

The paper is organized as follows. The next section provides a detailed insight of design and implementation of the system architecture (Section 2). We then exemplify the architecture deployment (Section 3). This is followed by a case study of time series analysis of Google traces (Section 4) and then detailed assessment of the architecture (Section 5). We then discuss our approach (Section 6) and conclude this paper (Section 7).

2 The Architecture

We present the design and implementation of a cloud architecture that is developed as a tool called TraceAnalyzer. The architecture is divided in two major systems represented in different colours as shown in Fig. 1. The red coloured portion provides
Fig. 1 TraceAnalyzer cloud architecture.

Effective data analysis methods for smaller amount of data and represents the core architecture. The other portion coloured in blue provides effective distributing computing algorithm for analysis of large datasets and acts as an extension to the core architecture.

2.1 The core architecture

The TraceAnalyzer core architecture resembles the layer architecture with three major layers, namely the TraceAnalyzer extension for SMW, the TraceAnalyzer REST-API, and a SolrCloud cluster. These three layers interact with each other to process workload traces and monitor the patterns obtained for any anomalies. The TraceAnalyzer core architecture is shown in Fig. 2.

At the top layer, the TraceAnalyzer extension of SMW provides users an interface to edit a wiki page of a certain trace analysis, configure the parameters, launch requests of data access and processing, and finally display the results. The TraceAnalyzer extension uses the feature of special pages of MediaWiki. Special pages are pages that are created on demand to perform a specific function. The TraceAnalyzer extension is a special page, which provides a source of user’s interaction with SMW and acts as a client application to communicate with TraceAnalyzer REST-API. The wiki page allows user to send service requests to the TraceAnalyzer REST-API. The REST API in response generates a wiki page on SMW client. This wiki page contains the information regarding the analysis result. Given the range of traces and the type of interested analysis methods, the TraceAnalyzer extension uses the annotations from SMW to structure the wiki pages generated from analysis results. In SWM, annotation Category provides automatic indices, similar to tables of contents. Since analysis methods, such as Kernel Density Estimation (KDE) can be applied to arbitrary range of traces, we define each analysis method as a Category. Once the result returns from the TraceAnalyzer server, a wiki page is created and is then linked to its corresponding category according to the analysis method.

The TraceAnalyzer REST-API acts as a bridge to connect the SMW and the SolrCloud that stores the traces. The core of REST-API is built on three modules, namely the search module, the analysis package, and the SMW page generator. The TraceAnalyzer Extension utilizes the REST-API’s resources to send processing requests via HTTP. In response, it invokes the analysis package, which utilizes the search module to extract datasets of interest. The search module then sends a search query to the shard of SolrCloud using SolrCloud’s API via HTTP. Upon the return of data, the analysis package produces the result and sends success response to the TraceAnalyzer. On receiving a success response, it initializes another request to plot the forecasted data for that particular dataset via HTTP POST method. Finally, the extension sends a request to generate a wiki page. The request calls the page generator module, which generates the analysis result.
and creates a new wiki page using MediaWiki’s markup language. The complete description of TraceAnalyzer REST-API is provided in later section.

At the bottom layer of the TraceAnalyzer cloud architecture, a SolrCloud cluster is placed for data storage and access. SolrCloud contains the subset of optional features in Solr for indexing and searching. SolrCloud enables horizontal scaling of a search index using sharding and replication. The structure of a SolrCloud cluster consists of the following components:

- **SolrCore** encapsulates a single physical index.
- **Node** is a single instance of Solr. A single Solr instance is bound to a specific port and can have multiple SolrCores.
- **Collection** contains a single search index distributed across multiple nodes. Collection has a name, a shard count, and a replication factor. A replication factor specifies the number of copies of a document in a collection.
- **Shard** is a logical slice of a single collection. Each shard has a name, a hash range, a leader, and a replication factor.
- **Replica** hosts a copy of a shard in a collection, implemented as a single index on a SolrCore. One replica of every shard is designated as a leader to coordinate indexing for that shard. Each shard has one and only one leader at any time and leaders are elected using ZooKeeper.
- **Cluster** includes all the nodes that host SolrCores.
- **ZooKeeper** provides a distributed coordination service that provides the cluster state management and the leader-election mechanism for fault-tolerance. ZooKeeper nodes constantly monitor the Solr nodes in a cluster.

The workflow of the architecture is very simplified as shown in Fig. 3. For instance, if a user wants to identify outlier in a dataset, it will interact with the TraceAnalyzer extension to input required parameters to instigate the process. The TraceAnalyzer extension sends a HTTP POST request with the specified parameters to the TraceAnalyzer REST-API for specific web service. The service immediately establishes a connection to retrieve required amount of data from SolrCloud. This data is processed and the result is presented in the form of wiki page to the user.

2.2 The architecture extension

The extension to the TraceAnalyzer Core Architecture incorporates an additional component to facilitate complex computation of large amount of dataset. The component itself is a cluster of multiple complex computation applications. The cluster consists of an Apache Spark application running on a Hadoop Yarn cluster, shown in Fig. 4. Hadoop Yarn is the key feature of second generation Apache Hadoop. It is a resource management platform for managing and scheduling user application in a cluster. In the
architecture, the Hadoop Yarn cluster contains a Yarn resource manager, a Hadoop namenode, and Spark running on a single node. It is also connected to other HDFS data-nodes. Running Spark jobs on Hadoop Yarn cluster benefit the user to store data in Hadoop file system (HDFS). It provides highly available and scalable storage capabilities to the user.

As shown in Fig. 5, the workflow of the extended architecture resembles the workflow of the TraceAnalyzer Core Architecture. The special wiki page accepts the required input parameters from the user. These parameters are then forwarded to the TraceAnalyzer REST-API in a form of HTTP POST request. The TraceAnalyzer REST-API comprehends the request and sends a received parameter wrapped as JSON message to Spark cluster’s main node. The node receives the JSON message and submits the job request to the Spark cluster. The message encoded in JSON format is passed to the Spark cluster’s main node using Spark Request Sender module present in the TraceAnalyzer REST-API. On the other hand, Spark Request Receiver module running on the Spark cluster’s main node receives the message and decodes it and submits the requested Spark job. The message passing between nodes is performed using Advanced Messaging Queueing Protocol (AMQP). AMQP is a standard protocol for message passing between different hosts in a cluster. The spark job after successful completion saves the result and transfers it to the TraceAnalyzer REST-API via secure-shell (ssh).

In the end of this sequence, TraceAnalyzer REST-API sends a success response to the TraceAnalyzer extension. TraceAnalyzer extension sends another request to TraceAnalyzer REST-API to generate a wiki page for the specific result. Currently, we have implemented k-means clustering algorithm using spark’s machine learning library to analyze the trace data by dividing it into required number of sets.

### 3 REST API

The TraceAnalyzer REST-API provides five non-trivial RESTful service resources in our architecture as shown in Fig. 6. These service resources collectively provide described functionalities. The four RESTful service resources are described below:

- **HWTES Processor** provides services to process an arbitrary range of dataset using Holt-Winters Triple Exponential Smoothing (HWTES) to determine workload forecast. It facilitates the client to process data stored in both file system and SolrCloud cluster. Forecast can be accessed anytime using unique dataset identifier, i.e., initial slice index.

- **KDE Processor** provides services to determine data point lying outside a particular threshold for an arbitrary dataset. It utilizes Kernel density estimation analysis method to determine workload patterns. Similar to HWTES Processor, dataset can be accessed from both file system or SolrCloud cluster. Processed data is automatically plotted and stored within the server. Workload patterns and
plots can be accessed using initial slice index.
- **HWTES Plotter** provides functionality to plot the forecast data produced by **HWTES Processor**.
- **Spark Clustering** service determines the clustering pattern in the given dataset. It also produces a basic plot for the results obtained for data visualization.
- **SMW Page Generator** creates the SMW pages to display the analysis results and it automatically uploads the pages to SMW client.

The description of RESTful services is provided in Appendix A.

4 Architecture Deployment

To deploy a prototype of the **TraceAnalyzer** architecture, we have acquired Emulab nodes provided by University of Utah[9]. We have 8 nodes to setup the testbed for **TraceAnalyzer Core Architecture**. One node hosts SMW and another node runs the **TraceAnalyzer REST-API**. The remaining six nodes are used to setup a SolrCloud cluster. SolrCloud requires setting up a ZooKeeper ensemble for monitoring the Solr nodes. We setup minimum nodes required for ZooKeeper ensemble, i.e., three ZooKeeper nodes to monitor three Solr nodes. The network configuration of the SolrCloud cluster is illustrated in Fig. 7.

Figure 8 shows the deployment of the SolrCloud with a single shard as the leader node and two replicas where copies of data are stored. It also illustrates the ZooKeeper ensemble, i.e., zk0, zk1, and zk2 to monitor the shard that consists of three Solr nodes. Initially, data are transferred to leader node. Later on, the stored data are replicated to replicas. Upon failure of the leader nodes, data are available from replicas.

Later, we deployed three additional nodes, i.e., yarn0, yarn1, and yarn2 to **TraceAnalyzer Core architecture** extension to add another component, i.e., Spark cluster. One node is specified to Hadoop Yarn resource manager. The Hadoop Yarn’s namenode contains the resource manager and Spark. It also includes the module to receive requests from the **TraceAnalyzer REST-API**. There are two more nodes connected to the Hadoop Yarn and serve the purpose of data-nodes for HDFS storage.

5 Case Study for Time Series Data

We consider a case study on analyzing the trace data to predict the workload and understand patterns in time series with levels, trends, and seasonality factors. We use the trace version1 of Google trace[3] as the input to develop the working prototype of the architecture. The Google trace provides data points for CPU and memory usage of the jobs acquired by machines in Google clusters at a particular timestamp. The data is anonymized in several ways. There are no task or job names, only numeric identifiers. Timestamps are relative to the start of data collection. The consumption of CPU and memory is obscured using a linear transformation[3]. According to Central Limit Theorem (CLT), the mean of an arbitrary dataset of a large size from an arbitrary distribution has approximately normal distribution. Thus, we assume that sample dataset taken for analysis is normally distributed.

5.1 Using core architecture components  

5.1.1 Data loading

The trace version1 of Google trace is unstructured data with no unique feature available to index. Solr requires a unique feature to index the data. Therefore we use the function of SolrCloud that automatically creates a unique id for each record being loaded. The automatically generated unique id is defined in the Solr schema file as shown below,

```json
{fieldType name = "uuid" class =
  "solr.UUIDField" indexed = "true" /
  {uniqueKey} id (/uniqueKey)
```
An example of automatically generated unique id is given as,

```
str name = "id"
```

SolrCloud supports different types of data uploading techniques such as index handlers, data import handlers for structured data, and HTTP POST for uploading data over the network. As Google trace is in the CSV format, we have used the CSV index handler to directly upload the data to SolrCloud.

5.1.2 Data query

Data queries are sent via HTTP POST requests to SolrCloud, that contain the collection name and the number of rows, i.e., the amount of data. For example, if a user intends to request 10 000 data records from SolrCloud, then a sample query is shown below,

```
se = SearchOptions();
se.commonparams.q('*:*').rows('100000')
response = trace_collection.search(se)
```

The query receives the data in the JSON format from SolrCloud. Then, the dataset queried is returned in the HTTP response. The returned dataset is further processed by the analysis module.

5.1.3 Workload prediction

The analysis package of the TraceAnalyzer implements the analysis methods include HWTES and KDE. The analysis methods are implemented as Python libraries. In this case study, the HWTES method is used to forecast the CPU usage. We consider a data set of 10 000 data points for analysis. This chunk of data is split into a number of seasons, each with duration $L$. In our case, the 10 000 data points are split into 4 seasons, i.e., $4L$, where $L$ contains 2500 data points. Each season and trend is predicted at the end of each period of $L$ based on the growth of previous season and trend, respectively. Finally, this smoothing method provides the forecasting for time series data of length $4L$. Figure 9 illustrates the exponential smoothing result for a slice of 10 000 data points of CPU usage from the trace version1 of Google trace. This particular result is predicted with smoothing constants $\alpha = 0.2$, $\beta = 0.2$, and $\gamma = 0.05$.

5.1.4 Workload pattern matching

KDE is used as the analysis method for determining the workload pattern. The kernel density is estimated for the data points lying outside the threshold, called outliers. Firstly, HWTES method is used to determine the outliers beyond the upper threshold and below the lower threshold. The upper and lower thresholds are calculated by adding/subtracting the standard deviation from its mean, respectively. The mean value is produced by the HWTES method. The smoothing analysis of the CPU usage for 10 000 data points from trace version1 is shown in Fig. 10. After calculating the outliers, the kernel density is estimated for 4 seasons as shown in Fig. 11.

By calculating the Pearson Correlation of two sequential seasons, it can be inferred if the outlier distribution changes significantly or not. The Pearson Correlation is a measure of how well two datasets are related to each other. The relation is shown in the following Eq. (1),

$$r = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{\sqrt{n\Sigma x^2 - (\Sigma x)^2][n\Sigma y^2 - (\Sigma y)^2]}$$

where Pearson’s $r$ value can range from $-1$ to $1$. An $r$ value of $-1$ indicates a perfect negative linear...
relationship between the variables. A value of 0 for $r$ indicates no linear relationship between variables and the value 1 for $r$ indicates a perfect positive linear relationship between variables. Furthermore, a value between 0.5 and 1 for $r$ indicates a high correlation. Figure 11 depicts the KDE analysis for outliers above the threshold for each season in Fig. 10. The $r$ value is 0.53, 0.74, and 0.76 for the correlation values of two sequential seasons in Fig. 11, respectively. The correlation values between the 2nd and 3rd seasons, and the 3rd and 4th seasons are high due to the low number of outliers. In both cases, zeros are default values for those spots where there are no outliers. Therefore, statistically the correlation values of both cases are high. This is intuitively meaningful as the low number of outliers also indicated the likelihood of the upper or lower threshold being across is low. Hence, the changes of workload over these last three partitions are negligible.

5.2 Using architecture extensions

5.2.1 Data loading

SolrCloud does not support interfacing with Hadoop Yarn cluster yet and it does not provide capabilities to retrieve data from Hadoop file system, i.e., HDFS. However, single Solr instance can be run over HDFS but it retains its high availability and scalability. To avoid this situation, another copy of trace data is stored in Hadoop file system, i.e., HDFS for processing. The file system is dedicated to the Spark cluster and provides highly available and scalable storage capabilities.

5.2.2 Workload clustering

We used basic $k$-means clustering algorithm for classification of the time series workload data. For this purpose, we utilized spark’s inbuilt machine learning library to run $k$-means clustering algorithm. $k$-means clustering algorithm is the most commonly used workload classification method. The algorithm divides the dataset into specified number of clusters, i.e., $k$, and assigns each datapoint to its closest cluster centroid by identifying the shortest distance between the data point and the centroids. We have used Euclidean metric to estimate the distance between data points.

Spark’s $k$-means clustering algorithm requires multiple input parameters to classify the dataset. The parameters are described below,

- $k$ decides the number of clusters required from the dataset.
- $Runs$ is the number of times $k$-means clustering algorithm should iterate to obtain convergence.
- $Max iterations$ restricts the maximum number of iterations allowed for clustering.
- $Initialization mode$ provides two initialization modes either random initialization or initialization via $k$-means $\mid$.
- $Initialization steps$ provides the estimation of number of steps in $k$-means $\mid$.
- $Epsilon$ is the distance threshold to decide
To demonstrate the working of the framework, we classified the trace data for different values of $k$. The workload classification intends to find insight of the task event in the trace data. We ran the clustering algorithm on $10^6$ data point for different values of $k$, i.e., 2, 4, 6, 8, 10, and 12. We used $k$-means || as initialization mode and kept the convergence threshold, i.e., epsilon constant as 0.6. The example results are produced using the framework illustrated in Fig. 12.

We scrutinized classification results for different value of $k$. We observed that most of the data points representing tasks CPU and memory usage in the monitoring dataset have considerably low value. These task events are utilizing approximately 0.0016 and 0.0031 CPU and memory on average, respectively. On the other hand only fewer tasks have high CPU and memory usage, i.e., 0.287 and 0.018 on average, respectively. This shows that cluster may contain similar task events. Due to anonymity of the trace data the observations are not very conclusive. However, the main focus of the paper is to implement an architecture providing such services.

5.3 Result display on SMW

The page generator in the TraceAnalyzer server uploads the results to SMW in three steps. First, the results are uploaded to SMW as image files. SMW authenticates the page generator module as a valid user. After the authentication, SMW grants the page generator module access to the upload feature. Second, a new page is created on SMW. The generated page is shown in Fig. 13a. Finally the content of the page is written using the wiki markup language to organize the page to represent the results in a comprehensive form. The markup content in Fig. 13b shows how results are organized as wiki pages. These wiki pages can be used in various way to structure the results so that it helps document and share the workloads of the system.

6 Evaluation

We evaluate the architecture for two quality attributes, namely performance and fault-tolerance.

6.1 Performance evaluation

To evaluate the performance, we designed experiments and measured end-to-end delay under various workloads for the Core Architecture. A sequence diagram in Fig. 14 shows the time broken down for operations in order, from a request being initialized in an SMW page to the result displayed to the SMW page. As annotated in Fig. 14, the ExecutionTime ($\Delta t$) is the time taken by the TraceAnalyzer REST-API to send the response back to the user, whereas, the FetchTime ($t$) is
the time taken by the TraceAnalyzer server to retrieve data from SolrCloud. Therefore, the ResponseTime \( t' \) is the total time taken as a sum of the FetchTime and the ExecutionTime, given by Eq. (2),

\[
\text{ExecutionTime} \ (\Delta t) = \\
\text{ResponseTime} \ (t') - \text{FetchTime} \ (t)
\]

The response time provides performance indication to the changes in the system behaviour in different situations. The situations include system failure and scaling as workload changes. Therefore, we conduct experiments to determine the response time to test scalability and availability of the system architecture. The experiments are explained in detail in the following paragraphs.

The aforementioned time metrics of the architecture is measured by changing the workload that is the amount of data being retrieved and processed. We increase the workload incrementally from 10,000 data records to 50,000 data records. The experiments are performed for both analysis methods. The broken down times are plotted and presented in Figs. 15 and 16 for Holt-Winters exponential smoothing and KDE, respectively.

In the case of the Holt-Winters exponential smoothing, the ExecutionTime is almost constant regardless of the workload changes. Hence, the linear growth of the ResponseTime is determined by the linear trend of the FetchTime. The FetchTime grows as the workload increases. In contrast, both the ExecutionTime and the FetchTime increase as the size of the workload changes in the case of KDE. From above observation, the size of the data determines the workload of the system and therefore the overall performance. Further improvement of the end-to-end delay requires parallelism of the analysis methods.

We also evaluated the performance aspect of the clustering functionality provided by Core architecture extension using the similar approach. In case of clustering, the trace data is stored in Hadoop file system instead of SolrCloud, and does not explicitly
require time for extraction. Hence, the FetchTime is replaced by JobExecutionTime, shown in Fig. 17. The JobExecutionTime is the total time taken by Spark cluster to process the data. The performance of $k$-means clustering is measured on the basis of the following Eq. (3).

$$\text{ExecutionTime} (\Delta t) = \frac{\text{ResponseTime} (t') - \text{JobExecutionTime} (t)}{\text{JobExecutionTime} (t)} \quad (3)$$

The response time of the $k$-means clustering methods is measured by examining processing time on different amount of data. Initially we gradually increased the size of data sets form 100 000 data points to 500 000 datapoint. Later, we jumped from 500 000 to 800 000. We obtained multiple reading and estimated the average time taken by the $k$-means clustering methods on different dataset. The result obtained is illustrated in Fig. 18.
The k-means clustering results show the similar behaviour to the Holt-Winter exponential smoothening and Kernel density estimation. The main responsibility of the TraceAnalyzer REST-API in Core Architecture extension is to submit job to Spark cluster. It takes approximately 3 s to accomplish its objective. It remains constant in the observations and shows that amount of data doesn’t effects the Execution time. However, the effect of increase in data size is clearly visible in the JobExecutionTime. It is notable that the Spark cluster’s execution time increases as the size of data increases.

6.2 Fault tolerance evaluation

We design experiments to evaluate the fault tolerance of this architecture by leveraging the fault tolerant mechanisms supported by SolrCloud. We deliberately put down the nodes of SolrCloud and the nodes of ZooKeeper. Meanwhile, we measure the FetchTime and observe any delays in data accessing. The initial measurement of the FetchTime is taken when all the nodes are alive. Then, deliberately a node is killed and its effect on the system’s behaviour is observed by measuring the FetchTime. These observations are then plotted for both Holt-Winters exponential smoothing and KDE, as shown in Figs. 19 and 20, respectively. In Fig. 19a, it shows the system is continuously responding when nodes of SolrCloud and Zookeeper are down. The node failure of SolrCloud, however, has more effect on the responsiveness of the architecture. It is observed that the FetchTime slightly increases with spikes in both cases of Holt-Winters exponential smoothing and KDE. The observed spikes make approximately 5% to 35% more of the FetchTime. This is mainly due to the election of a new leader node from the remaining nodes when a failure occurs in either the ZooKeeper ensemble or Solr nodes and there is an overhead associated to stabilize the cluster.

7 Related Work and Discussion

The survey papers\cite{1,10} of cloud monitoring abstract
the process of cloud monitoring in three main steps: the collection of relevant state, the analysis of the aggregated state, and decision making as a result of the analysis. The requirements of cloud monitoring on scalability and fault-tolerance have an inherent propensity from cloud computing. A scalable and fault-tolerant cloud monitoring system is free from a single point of failure and bottlenecks, and adapts to elasticity. Since the monitoring data can be useful long after it has been collected, the monitoring data are continuously accumulating. Therefore, failures must not interrupt monitoring.

An interesting work on SaaS architecture is described in Ref. [11]. The paper highlights the key differences between the traditional Service Oriented Architecture (SOA) and SaaS and provides an insight of the four different SaaS architecture styles. These architecture styles have been analyzed based on their capabilities to address three key challenges of SaaS architecture: its capability to address customization support to multi-tenancy architecture, and salability. In Ref. [12], two mainsteam architectures are compared. The observations led to the conclusion that new architecture designed keeping the core architecture design principles provides more stable, scalable, and sustainable architecture. They also proposed an evolutionary architecture compatible with the existing architectures.

Existing tools are mostly provided as part of IaaS or PaaS cloud services. These monitoring systems are provisioned by the cloud service providers. They often have limitations in adding analysis methods beyond simple aggregations and threshold-based settings. In this paper, we present a cloud architecture leveraging SolrCloud, the open source search-based cluster that supports large monitoring data storage, query, and processing. This architecture is integrated with Semantic MediaWiki that allows documenting, structuring, and sharing the source of cloud monitoring data as well as any analysis results. The search-based cluster shares characters with a NoSQL data storage as data are indexed and partitioned. The replication and the failover mechanisms of SolrCloud strengthen this architecture’s attribute on fault-tolerance.

The analysis methods that we demonstrate are developed as python libraries. Through experiments, we show the data size has linear effects on the end-to-end delay. In order to improve the responsiveness of this architecture, the parallelism of analysis methods is necessary. This is not a simple work and requires dedicated research. Reference [13] has developed a MapReduce algorithm to run the exponential smoothing. It remains our future work to expand the architecture with support of running MapReduce-based analysis methods.

8 Conclusions

In this paper, we propose an architecture that integrates search-based clusters and semantic media wiki by
REST APIs to support the exploration of cloud monitoring data. This architecture benefits from a Web-based MediaWiki interface and allows a user to define the access to monitoring data and organize the processing results. The search-based cluster built on SolrCloud enables indexing of large size of data, and thus makes the whole architecture suitable to explore and monitor the ever-accumulating data such as the traces produced from data centres. The architecture also includes an extension, which runs Spark on Yarn cluster for deploying efficient analysis methods for large data set. It utilizes the spark’s MapReduce paradigm to identify the cluster in the dataset using k-means clustering method. We evaluate the performance and fault-tolerance of the architecture by experiments. Our observations demonstrate the architecture is responsive under node failure. The overhead incurred to handle the node failure in the SolrCloud cluster can result in extra delays of retrieving data. Since SolrCloud is optimized for indexing and data retrieval, the overall response time of this architecture is mainly determined by running analysis methods. The architecture provides a rationale to develop cloud monitoring applications with advance algorithms for forecasting data and identifying workload patterns. We also evaluated the architecture extension for its performance and observed increase in responsivenss with the increasing amount of data. Our future work includes developing more analysis methods for the architecture extension to make the architecture generic.

Appendix A  REST API Resources

A.1 HWTES Processor

- Resource URI: http://host:port/api/process/hwtes/POST evaluates forecast data for arbitrary dataset of 10000 data points using HWTES. The dataset is extracted either from a trace-data file or SolrCloud cluster.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>Constant estimated in such a way that the mean square error (MSE) is minimum.</td>
</tr>
<tr>
<td>Beta</td>
<td>Same as above.</td>
</tr>
<tr>
<td>Gamma</td>
<td>Same as above.</td>
</tr>
<tr>
<td>State</td>
<td>“–l” or “–r” are arguments to decide to load a new dataset or reload the existing one in the system, respectively.</td>
</tr>
<tr>
<td>Path/index</td>
<td>Path variable is specified alongside load state to locate dataset. For example, /solr would request to retrieve data from SolrCloud, whereas, path some-folder/some-file.csv would extract data from specified file location. However, index variable is specified alongside reload state, to retrieve data from existing dataset.</td>
</tr>
</tbody>
</table>

- Resource URI: http://host:port/api/process/hwtes/(index)/GET returns the list of forecast data for a particular dataset in JSON format.

A.2 KDE Processor

- Resource URI: http://host:port/api/process/kde/POST determines the outliers, i.e., data points lying above and below the estimated threshold value, and plots the results.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>“–l” or “–r” arguments decide to load a new dataset or reload the existing one.</td>
</tr>
<tr>
<td>Path/index</td>
<td>Path variable is specified alongside load state to locate dataset. For example, /solr would request to retrieve data from SolrCloud, whereas, path some-folder/some-file.csv would extract data from specified file location. However, index variable is specified alongside reload state, to retrieve data from existing dataset.</td>
</tr>
</tbody>
</table>
• Resource URI: 
  http://host:port/api/process/kde/{index}/GET returns the number of outliers, i.e., data points lying above/below a specific threshold.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>It is the initial dataset index used as an id to find the forecast list request.</td>
</tr>
</tbody>
</table>

**DELETE** purges the KDE results for a particular dataset.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>It is the initial dataset index used as an id to find the KDE data lists location.</td>
</tr>
</tbody>
</table>

### A.3 HWTES Plotter

• Resource URI: 
  http://host:port/api/plot/hwtes/POST plots the forecast data for a particular dataset.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>It is the initial dataset index used as an id to find the forecast list location.</td>
</tr>
</tbody>
</table>

• Resource URI: 
  http://host:port/api/plot/hwtes/{index}/GET returns the forecast plot for particular dataset.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>It is the initial dataset index used as an id to find the forecast list location.</td>
</tr>
</tbody>
</table>

**DELETE** deletes the forecast plot for the specified index.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>It is the initial dataset index used as an id to find the forecast list location.</td>
</tr>
</tbody>
</table>

### A.4 Spark Clustering

• Resource URI: 
  http://host:port/api/sparkclustering/kmeans/POST provides the information regarding the clusters present in the given dataset.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Input path for the file location. The data can be retrieved from HDFS or local file system by specifying in the URL as hdfs:// or file://, respectively.</td>
</tr>
<tr>
<td>Output</td>
<td>To save the result obtained after processing.</td>
</tr>
<tr>
<td>k</td>
<td>It defines the number of clusters to be obtained in the dataset.</td>
</tr>
<tr>
<td>Iteration</td>
<td>Number of times data needs to be iterated to get convergence.</td>
</tr>
<tr>
<td>Size</td>
<td>The size of data to utilized for processing.</td>
</tr>
</tbody>
</table>

### A.5 SMW Page Generator

• Resource URI: 
  http://host:port/api/generate/POST structures the analysis results and creates an SMW page using MediaWiki markup language.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index/path</td>
<td>The initial dataset index used as an id to find the forecast list location required for KDE and HWTES only. However, output path of the result location is required in case of creating k-means clustering result page.</td>
</tr>
<tr>
<td>Method</td>
<td>“HWTES”, “KDE”, and “k-means” string arguments to create holt winter forecast, to generate KDE outliers estimation page and “k-means” to generate k-means clustering page, respectively.</td>
</tr>
<tr>
<td>Username</td>
<td>Semantic-Mediawiki username for access.</td>
</tr>
<tr>
<td>Password</td>
<td>Same as above.</td>
</tr>
</tbody>
</table>

**Acknowledgements**

This work was supported by the Discovery grant No. RGPIN 2014-05254 from Natural Science & Engineering Research Council (NSERC), Canada.
References


Yan Liu received the PhD degree in computer science from the University of Sydney, Australia in 2004. She is currently an associate professor at the Electrical and Computer Engineering Department, Concordia University, Canada. Her research interests are data stream processing, cloud software architecture, distributed computing, software performance engineering, and adaptive systems. She was a senior scientist at the Pacific Northwest National Laboratory in Washington State and senior researcher at National ICT Australia. She is a member of the IEEE.

Samneet Singh is a master student of applied science at Concordia University pursuing his research work under the supervision of Dr. Yan Liu. His major interest lies in cloud computing, distributed systems, big data analytic, and machine learning. Currently, he is working as a data analytics intern at Ericsson Canada Inc.