A Scalable Platform for Monitoring Data Intensive Applications

Gabriel Iuhasz
Institute eAustria Timisoara and West University of Timisoara
iuhasz.gabriel@e-uvt.ro

Ioan Dragan
Institute eAustria Timisoara and West University of Timisoara
ioan.dragan@e-uvt.ro

Dana Petcu
Institute eAustria Timisoara and West University of Timisoara
dana.petcu@e-uvt.ro

Abstract

Latest advances in information technology and the widespread growth in different areas are producing large amounts of data. Consequently, in the past decade a large number of distributed platforms for storing and processing large datasets have been proposed. Whether in development or in production, monitoring the applications running on these platforms is not an easy task, dedicated tools and platforms were proposed for this task yet none are specially designed for Big Data frameworks. In this paper we present a distributed, scalable, highly available platform able to collect, store, query and process monitoring data obtained from multiple Big Data frameworks. Alongside the architecture we experimentally show that the solution proposed is scalable and can handle a substantial quantity of monitoring data.

Keywords Exascale, H2020, Monitoring, data-intensive, machine learning, anomaly detection, distributed
Formal publication `https://doi.org/10.1007/s10723-019-09483-1`

I. Introduction

Big Data technologies have become an ever more present topic in both academia and industrial world. These technologies enable businesses to extract valuable insight from their available data, hence more and more SMEs are showing increasing interest in using these types of technologies. Distributed frameworks for processing large amounts of data, such as Apache Hadoop[1], Spark [43], or Storm[2] gained in popularity and applications developed on top of them are more and more prevalent. However, developing software that meets high-quality standards expected for business-critical Cloud applications remains a challenge for SMEs. In this case model-driven development (MDD) paradigm and popular standards such as UML, MARTE, TOSCA hold strong promises to tackle the challenges [13].

During development of Big Data applications it is important to monitor performance for each version of the application. Information obtained can be used by software architects and developers to track the evolution and performance of the developed data intensive application. Monitoring is also useful in determining main factors that impact the quality and performance of different application versions [7]. Throughout the development stage, running applications tend to be more verbose in terms of logged information so that developers can get insights about the developed application. Due to verbosity of logs, data-intensive applications produce large amounts of monitoring data, which in turn need to be collected, pre-processed, stored and made available for high-level queries and visualization.

In this paper we provide details regarding our monitoring solution that is tailor made in order to solve the aforementioned issues. We present a scalable, highly available and easy deployable platform for monitoring multiple Big Data frameworks. Currently there are no solutions that support simultaneously system and Big Data framework related metrics. The proposed solution currently integrates resource-level metrics, such as CPU, memory, disk or network, together with framework level metrics collected from Apache HDFS, YARN, Spark and Storm. The platform is easily extensible to other Big Data frameworks. In addition it is capable of ingesting large quantities of metrics from various sources and serve them to end users and other tools. This is done such that it provides a clear and complete overview of the current data intensive application and underlying system states. Most currently available monitoring solutions are not able to provide such a fine grained view of a data intensive application during the development phase. Our solution also possesses a simplified querying scheme not present with other monitoring tools. It allows for the generation and serving of monitoring data in a variety of formats which can later be used by both the end-users directly and other tools that require such information. Furthermore, we also provide automatic detection of monitorable services and the creation of visualization for Big Data frameworks. These features are not present in most if not all current monitoring solutions and are meant to help developers during data intensive application design and implementation.

The rest of this paper is structured as follows. In Section II we present the current state of the art for monitoring applications. Section III presents the architecture and design choices made for our distributed monitoring platform. Next, in Section IV, some important considerations on the proposed monitored metrics are presented. Finally, in Section VI, we present the initial validation of our monitoring solution, while Section VII discusses ongoing and future work.

[1]http://hadoop.apache.org/
[2]http://storm.apache.org/

II. Related work

In general when monitoring big data platforms on Cloud based deployments a cross layer monitoring scheme is employed. This is due to the fact that components of individual applications can be distributed on various cloud layers as well as on multiple Virtual Machines (VMs). For this purpose the monitored parameters should be transmitted across all the Cloud layers used by the applications. Only by doing so one can have a complete picture of the current status of individual applications at a given time. It is common to have applications running various components on SaaS (Software as a Service), PaaS (Platform as a Service) or/and IaaS (Infrastructure as a Service). In [24] some problems of monitoring public Clouds are identified and an architecture for a monitoring platform is proposed.

II.1 Cross layer monitoring

On IaaS typically we want to monitor resource utilization such as CPU usage, states for Hard Disk utilization, Memory usage and status, as well as additional network parameters. In contrast at PaaS and SaaS level parameters include byte throughput metrics, status of system services, uptime, availability etc. For example, in the case of a Hadoop deployment we have metrics such as MapReduce processing time, Job Turnaround, Shuffle operations etc. The difficulty in deciding what the optimum settings are for a given application is considerably made worse by lacking or even missing run-time information. For example in the case of a job and reducer task scheduling [4] monitoring data is crucial. Another approach can be seen in [30] where minimalist monitoring is used. A solution for monitoring wireless devices is proposed WiGriMMA and presented in [12].

The type of resources that are monitored is highly dependent on the application type. For example data transfer quality and rate is important for any video streaming application while a batch processing application will only be interested in basic process and network latencies.

There are several types of monitoring solutions currently in use or under development. In the case of centralized monitoring, all resource states and metrics are sent to a centralized monitoring server. These metrics are continuously pulled from each monitored component. It is easy to see that this approach, while allowing a more controlled management of any Cloud application, has several drawbacks. First, it has a single point of failure and lacks scalability. This means that at a certain stage if the monitored application will exceed the capability of the central monitoring server, the only solution applicable would be vertical scaling of the system. Moreover high network traffic can also lead to bottlenecks which in turn can lead to faulty or incomplete monitoring data. A decentralized approach can alleviate most if not all of these problems.

In a decentralized architecture no component is considered more important than another. In structured Peer-to-Peer systems the central authority is defused thus eliminating the central point of failure. Unstructured Peer-to-Peer network overlay is meant to be distributed, however, the search directory is not centralized. Besides the aforementioned we also have the hybrid Peer-to-Peer systems in which *super peers* can serve as localized search hubs for small network portions [3].

In [26] some of the more popular monitoring tools and platforms for Cloud computing and Big Data are presented. Part of these platforms have been adopted from High-performance computing (HPC) scenarios while others have been designed specifically for this task.

II.2 Monitoring tools and platforms

NewRelic[3] provides a solution for monitoring both the infrastructure in a traditional, hybrid or Cloud setup as well as the applications running on them. In a nutshell it provides a "serverless" monitoring solution that can handle all sorts of tasks by means of code instrumentation. In this situation, code refers both to actual code that runs as well as the VM's on which the code is running. By doing so one can observe the effects of deploying new features on the existing infrastructure. Similar to most enterprise level applications NewRelic comes with a price tag attached.

Honeycomb[4] is designed as a platform driven by events intended to debug systems, applications and databases. The aggregation operations of data is done at read-time in order to support fast analytics over big datasets without having to worry about the underlying schemas or indexes. Another important feature advertised for Honeycomb is collaboration. For this purpose it has integration with Slack for ease of communication and every member of the development team can explore the system without superuser rights. One of its major drawbacks of the system is the pricing scheme, the system cannot be used for free.

DataDog[5] is yet another monitoring solution that provides full stack monitoring. It handles infrastructure monitoring, application monitoring as well as log management. Any of these components can be purchased individually but also as a single solution. DataDog advertises that it is a turn-key solution for aggregating metrics and events for full devops stack. Pricing for DataDog is per million trace events or in case of application monitoring a flat rate is applied plus the rate for event monitoring, resulting in a high price to pay for application and infrastructure monitoring.

Hadoop Performance Monitoring UI [40] is designed as a built-in solution for finding bottlenecks in Hadoop set-ups as well as providing visual representation of the available tunable parameters for better performance of Hadoop. In essence, one can view the tool as being a lightweight monitoring UI for Hadoop servers. The fact that it is built-in the Hadoop ecosystem and easy to use, one can consider this solution for minor system tuning. Although built-in, it lacks performance, a good example where it laggs can be considered the time spent in garbage collection by each of the tasks.

SequenceIQ[6] is yet another solution for monitoring Hadoop clusters. The architecture proposed by SequenceIQ[7] and used in order to do monitoring is based on the ELK stack, that is, Elasticsearch[8], Logstash[9] and Kibana[10].

SequenceIQ uses an architecture based on Docker containers in order to obtain a clear separation between the Hadoop deployment and the monitoring tools. In a nutshell the monitoring solution

[3]https://newrelic.com/

[4]https://honeycomb-analytics.com/

[5]https://www.datadoghq.com/

[6]http://sequenceiq.com/

[7]http://blog.sequenceiq.com/blog/2014/10/07/hadoop-monitoring/

[8]https://www.elastic.co

[9]http://logstash.net

[10]https://www.elastic.com/producs/kibana

consist of client and server containers. The server container takes care of actual monitoring tools. In this particular deployment Kibana is used for visualization and Elasticsearch for consolidation of the monitoring metrics. Through the capabilities of Elasticsearch one can horizontally scale and cluster multiple monitoring components. The client container contains the actual deployment of the tools that have to be monitored. This instance contains Logstash, Hadoop and the collectd modules. Logstash connects to the Elasticsearch cluster as a client and stores the processed and transformed metrics data there.

Due to its design the proposed solution is created by using multiple tools that are used in order to monitor metrics from different layers [28]. Because of containerization one can easily add or remove components from the system without affecting the overall monitoring platform. Also, interrogating the system for various data becomes an easy task to perform.

Hadoop Vaidya[11] is a rule based performance diagnostic tool for MapReduce jobs. The mechanism behind Vaidya performs post analysis steps for map-reduce jobs. In order to achieve this goal it parses various execution statistics, or configuration files, from job history and stores them for later interrogation and usage.

Finding the performance problems that arise in the execution steps is done by application of individual rules implemented by Vaidya on the stored job execution statistics. Due to its design, rules are applied one by one and only known problems might be detected. By application of the rules, this system is able to provide users with advice for future executions such that some of the already occurring problems to be avoided. The output produces is an XML report based on the evaluation of individual test rules.

Ganglia[12] is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. The design of Ganglia is hierarchical and targets federation formation for clusters. Because of the use of hierarchical approach, it manages preserve a low per-node overhead and high concurrency. By design it is robust and easy to be ported on various operating systems. One of its major benefits is that it can easily scale-up to handle thousands of nodes. Nowadays it is being used in lots of setups around the globe.

Apache Chukwa[13] is an open source data collection system for monitoring large distributed systems. At its core, Chukwa uses HDFS system and the Map/Reduce framework. By doing so, it provides mechanisms for easily scale. Although it is designed to collect the monitored data, it provides the users with a toolkit able to better understand the collected data. That is, it is capable of analyzing and display the results of different runs of the monitored software. Unlike other solutions, it is released under Apache license.

Nagios[14] monitoring platform supports multi-layer monitoring. Due to its plugin based architecture, Nagios provides a way of monitoring both Cloud based resources as well as in-house infrastructure. In order to achieve this it uses SNMP monitoring network resources. The architecture of Nagios requires a centralized server in order to collect the monitoring data. However, it is possible to create a hierarchy of Nagios servers that mitigate the disadvantages of a centralized server.

[11]http://hadoop.apache.org/docs/r1.2.1/vaidya.html
[12]http://ganglia.info
[13]http://chukwa.apache.org
[14]https://www.nagios.org/

OpenNebula[15] can be viewed as a solution for monitoring and management of data-centers. The tool is designed to work over SSH in order to connect to all of the monitored machines. As advertised it was mainly designed to monitor physical infrastructures and also private Cloud deployments.

II.3 Tools appropriate for development stage

There are also monitoring and management solutions customized for various instances. For example *OpsCenter*[16] provides a dedicated tool that can be used to monitor and administrate Cassandra. Another relevant tool that can be used in monitoring both Cassandra and do administration work on the nodes in the cluster is *Applications Manager*[17]. Application Manager is not bound to only monitoring Cassandra and provides ways of monitoring also NoSQL DB systems as MongoDB and others. When speaking strictly about database deployment we have a tool developed inside MongoDB *MMS*[18] that runs as a service and handles the inner monitoring of the application.

The monitoring solutions presented in this section can be split up into two main categories. The first category contains solutions that are geared towards monitoring distributed application and/or platforms such as web applications. Although they can theoretically be used to monitor Big Data frameworks and the data intensive applications developed on top of these, this task is not as straight forward as it first appears. They either require significant modifications and configuration for collecting performance data relevant during the development stages (in particular polling periods for the metrics) or, even more problematic, they might not support the entire Big Data stack on which a data intensive application is built on. Furthermore, most solutions are offered at SaaS level thus making any modification that might be necessary especially cumbersome and in some cases impossible. The second category are those monitoring tools which rely on post-mortem analysis. These solutions can't give a insight into the current performance and internal state of the application being developed during execution.

III. Platform Architecture

In a nutshell, DICE Monitoring (DMon) platform[19] is designed as a Web service, enabling the deployment and management of several sub-components. Each of the sub-components is responsible for enabling monitoring of Data Intensive (Big Data) applications and frameworks. In contrast to other monitoring solutions [2, 3, 20], our platform aims at providing the user with as much monitoring data as possible about the current status of the Big Data sub-components. Its main aim is to give performance and quality related metrics to application architects and developers during the development phase. By doing so, a wide range of new technical challenges arise. Due to the fact that DMon is serving near real-time fine grained metrics it must exhibit high availability and easy scalability.

DMon provides a distributed, high availability monitoring service. It is tailor made for Big Data technologies and it is easily extensible

[15]http://opennebula.org/
[16]http://www.datastax.com/what-we-offer/products-services/datastax-opscenter
[17]https://www.manageengine.com/
[18]https://mms.mongodb.com/
[19]This platform has been developed in the framework of the DICE project, http://www.dice-h2020.eu

to collect metrics from a wide range of frameworks. Big Data service integration into monitoring platforms is still very much an open issue. Ingesting large amounts of data in a timely manner is also an open question. During production, only warning or error level logs and metrics are important. However, during development this level of detail is not sufficient. In order to consume and present the metrics, in a useful and timely manner, is a key problem. This problem is one of the main reasons for the necessity of designing and implementing a specialized big data ready distributed monitoring solution.

In general web services are build using a monolithic architecture where most of the components of the system run in a single process. That process is usually hosted on a Java Virtual Machine (JVM). In case one uses this architecture type there are major advantages when it comes to deployment and networking. On the other hand, scaling such a system is a non-trivial task. In order to achieve the goal of scalability, one has to organize the several instances that run behind a load-balancer instance.

Although the monolithic approach seems to be the way to go when deploying monitoring platforms, there are some severe limitations to it. One of the side effects that can happen is the appearance of unforeseen impact on various components of the application when changing one of the components. Due to this situation adding new features to the platform can become potentially very expensive (in both time and resources). Another issue is related to individual components, that is, they cannot be deployed independently. Meaning that in case one is interested in only partial functionality of the service, this cannot be achieved unless the entire service is up and running. By doing so one is ensured that most of the platform is loosing reusability. As observed in practice this is not what is happening, but rather the focus is on readability of code hence performance loss is a side effect.

Taking into consideration both the advantages and limitations exposed by a monolithic design for a monitoring platform, we decided to use another fundamentally different approach for DMon. That is, we are deploying a widely used approach in the Internet companies [18], the so called *microservice architecture* [34]. By using the microservice architecture we replace a monolithic service with a distributed system of lightweight services. Each of the service being designed independent and narrowly focused. The approach allows us to deploy, upgrade and scale individual services rather than entire monolithic components. Due to the loosely coupled manner of microservices, code reusability is much higher and changes made to individual services do not necessarily require changes in other ones. In the microservice architecture, integration and communication between services should be done either by using HTTP (REST APIs) or using RPC requests. The use of microservice architecture allows us to group related behaviors into separate services, thus enabling us to easily modify the overall system without the inconvenience of modifying multiple services.

By design DMon is based on REST APIs that enable communication between individual services and the requested payload is encoded as JSON messages. By doing so, DMon is empowered to create and send both synchronous and asynchronous messages in a much easier manner.

Figure 1 presents the overall architecture of the DMon platform. We can see from this figure the main services and workflow which make up DMon. Agents installed on the monitored nodes which send metrics through the queuing service into the serving layer. The controller is responsible for deploying and configuring all other services. Services such as the anomaly detection and trace checking consume data from the serving layer and controller. It is important to note that the controller is also responsible for some of the preprocessing of the data define through its query API.

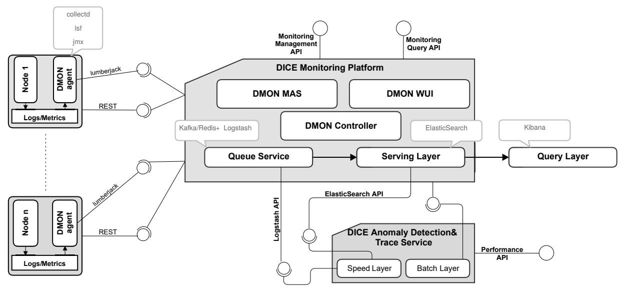

Figure 1: DMon architecture

The context in which DMon is developed relies on the so called lambda architecture [32]. The lambda architecture consists of three layers: *speed*, *batch* and *serving layer*. By design DMon follows the aforementioned guidelines and uses for the *serving layer* Elasticsearch. This component is responsible for loading batch views of the collected data providing other tools and layers with random access to read it. For looking at recent data and its representation in a query function DMon makes use of the *speed* layer. Whereas the *batch* layer is used in order to compute arbitrary functions on large sections of the dataset that is stored in Elasticsearch.

The design and implementation of individual components is based on several core technologies. For storing and indexing data we use Elasticsearch. The job of gathering and processing of log file data is performed by Logstash. For the purpose of presenting the users with a friendly graphical user interface we chose to use Kibana server. In terms of services the core services used in the development of DMon are: the *dmon-controller, dmon-agent, dmon-shipper, dmon-indexer, dmon-wui* and *dmon-mas*. Each of these services will be used to control both the *core* and *node-level components*.

III.1 Core components

As backbone for the monitoring platform we use the *Core components*. They are used for collecting, processing, aggregating and transforming all the incoming monitoring data. One of the most essential features that is common for these components are: ease of configuration, scalability and support for high throughput.

Elasticsearch [21] is an open-source, RESTful search engine based on top of Apache Lucene [33]. By design Elasticsearch is a scalable solution that is capable of performing near real time processing of data. Besides providing near real time processing speed it also allows to do streamlined backup while insuring data integrity. But one of the core capabilities of Elasticsearch is that it can handle high throughput of messages.

For collecting, processing and forwarding events and log messages in the monitoring tool we make use of Logstash [39]. In a nutshell, Logstash is designed to perform *E*xtract, *T*ransform and *L*oad (*ETL*) operations. For performing all these operations it uses a variety of plugins that can be configured as needed either for input-output or for collecting, processing and loading data. As input data the plugins are configured for accepting multiple sources ranging from TCP/UDP to Kafka [31] topics. Data received by the input plugins is later on sent for processing to the worker plugins (filter). The output from these plugins is further sent to one or more Elasticsearch, Kafka, InfluxDB plugin the can handle these. Because of its stateless design, Logstash becomes a easily scalable tool, as an example we can consider two instances of Logstash serving the same Elasticsearch endpoint.

In our design we use Kibana [21] as a web user interface that provides analytics and as a search interfaces for Elasticsearch. By doing so we match the scope of Kibana, that is viewing processed events from Logstash.

By combining the above components we obtain the so called *Elasticsearch ELK* stack. This setup provides a very robust base for DMon.

III.2 Node-level components

DMon has to monitor a wide range of Big Data technologies each of which have different metrics and metrics systems. Because of these constraints we had to use collectors that are flexible enough to accommodate a wide variety of technologies. Besides input restrictions the collectors must have a small computational footprint. By taking into account all the previous restrictions we limit the amount of "noise" produced by the presence of different collectors. Another critical feature that all the deployed collectors must obey is ease of deployments. That is, it should be easy to deploy and configure them for thousands of physical or virtual machines (VM).

Collectd [20] is an open-source POSIX daemon that collects, transfers and stores performance and network related data. Being a widely used tool, collectd provides the users with a great range of options for collector plugins. In DMon we use collectd to collect system metrics, such as CPU utilization, memory, hard disks, networking and other system related metrics.

As previously presented, the Logstash server is able to collect metrics and log files directly from the machine it is installed on. In this particular case it would mean that we need to have a Logstash instance on each of the monitored nodes, which in most cases is not a suitable approach because Logstash has a substantial computational footprint when using specialized filters such as grok. Instead, we decided to use logstash-forwarder [21] to do the job of metrics forwarding. logstash-forwarder is designed for the purpose of log forwarding to one or more logstash server instances. Using this approach inside DMon we are eliminating node-level side effects caused by local processing of logs.

Since most of Big Data frameworks are Java-based complex frameworks, we can use Java Management Extensions (JMX) to extract valuable metrics related to the JVM. In fact, a large number of Big Data frameworks already support exporting metrics via JMX. Thus, jmxtrans [22] tool is used in our architecture to collect attributes exported at JVM level. We should note at this point that jmxtrans is only supported for backwards compatibility within the DICE project. All JMX metrics are now collectable via a collectd plugin called FastJMX.

It is also worth mentioning that although there are new versions of data shippers for the ELK stack called Beats[23]. These shippers are able to process data and send it directly into Elasticsearch in the appropriate format for indexing. We currently do not use them in DMon as during some initial test we have found that they produce significant "noise" in the collected metrics which in a production environment where metrics might be collected every 30 seconds or more is negligible but when dealing with debug level metrics (every second) the "noise" can have a negative impact on performance and quality related metrics.

III.3 Platform services

In this section we present a number of web services that wrap the core and node-level components. The wrappers are developed in order to allow easy deployment and configuration of all the previously described components.

III.3.1 Core-level services

The web services developed are divided into three major categories: *dmon-controller*, *dmon-shipper* and *dmon-indexer*; all of the services are implemented in Python using the Flask microframework [22]. In order for the services to communicate with each other we encode the messages using JSON.

The core component is *dmon-controller*, all the other services and components communicate with it. In fact it acts as the main point of integration with all the services developed in the DICE project. The REST API is split into two main parts: *Monitoring Management API* and *Monitoring Query API*. A swagger[24] based web UI is used for all developed services for easy of use and a form of interactive documentation of all REST APIs.

Monitoring Manager API

The Monitoring Management API, code named *Overlord*, is used to register nodes, change configuration parameters and current status of all node-level components. It can also be used to deploy and configure node-level metrics on to registered nodes. Because of this, when registering nodes it is required that credentials for each node be supplied (username, password or ssh key). If node-level components and services have already been deployed by other tools they only have to register the already deployed node-level service endpoints. In this scenario credentials are not needed.

The *dmon-agents* are also deployable using the DICE Deployment Service [6] Chef cookbooks [25]. This allows the deployment of the developed application with all of the monitoring components already in place without any additional user intervention using default parameters which can be latter changed if necessary using the Management API.

Long term storage of metrics is a problem that has to be dealt with in all data warehousing solutions. For addressing this problem we use DMon management API for creating indexes that store the monitoring data. By default, in DMon we create indexes every 24

[20]https://collectd.org/
[21]https://github.com/elastic/logstash-forwarder
[22]http://www.jmxtrans.org/
[23]https://www.elastic.co/products/beats
[24]http://swagger.io/
[25]https://github.com/dice-project/DICE-Chef-Repository/tree/master/cookbooks

hours, all of them can be exported and even dumped into a variety of formats. We also provide with means of querying them either individually or all at once. Besides, the exported or dumped indexes can be at a later point in time uploaded back in DMon or in other deployments that perform offline processing of monitored data.

In the dmon-controlled we implemented support for version annotation of metrics. That is, metrics collected from specific application version are annotated using tags, providing a easy way of querying, aggregating or even metric comparison for a target application. Another way of dealing with the problem of having multiple versions of an application is by creating separate indexes for each of the versions. However this approach is not as versatile as the former solution because it makes it hard to do comparison by introducing a second query in order to obtain the differences in metrics for two versions of the application.

Another important role for dmon-controller is to generate and enact configurations for all the core components. These configurations are mainly dependent on the data provided at registration time for each node, resulting in an automatic configuration of each of the DMon components.

As already mentioned, the type of node-level components needed for monitoring is based on the Big Data service that run on each machine. During registration a list of services that are deployed on each node can be defined, the list is used in order to setup and manage node-level services and components. In some instances developers are not overly familiar with how Big Data platforms work or what services are deployed and need to be monitored. In the Management API we have the ability to auto detect what services (or roles) each processing node has. We accomplish this by scanning all nodes for Big Data platform specific services. Good examples of this are the YARN and Spark history servers. These history servers contain generic information about the cluster such as the available nodes, the role assigned, submitted application related information (execution time, execution attempts/failures etc.). Once all nodes have a role assigned to them DMon will automatically generate the appropriate configuration for the node level sub-components (see section III.3.2). It is important to note that this automatic discovery of roles is available for all supported Big Data services (YARN, Spark, Storm, Kafka, MongoDB, Cassandra).

Monitoring Query API

The *Observer* is used in order to query DMon by making use of the Monitoring Query API. One example of such a query is presented in Listing 1, and it does not require any kind of authentication to be performed. The Listing 1 presents various attributes such as: size of the returned response, its ordering, or start and stop dates (in UTC). We use a similar format for the *queryString* as in Kibana, defining the predicates that run on Elasticsearch [21] and can be further used to aggregate data and perform additional operations on data. As response from the query we get answers encoded as CSV, JSON or in plain text format. Support for RDF+XML encoding using OSLC Perf. Mon 2.0 vocabulary [29] was also developed but only system metrics are currently exportable using this format. The CSV and RDF+XML query responses are generated using the dmon-controller service. As input the dmon-controller takes a JSON response and converts it into the desired target format.

Internally DMon uses dataframes to store incoming queries and is able to perform some post processing on them if necessary. Initially we used a simple associate array type storage of incoming queries however it suffered from major performance problems when trying to apply any post processing to the queried data. For the dataframes we used the Pandas[26] library.

It is also possible to issue an aggregated query in which the user (or tool) specifies what technologies are being monitored. The resulting query will be an aggregation of the metrics collected for the specified technologies (or applications). The metrics index is set to the collection time of said metric. In the example query found in listing 2 the field *aggregation* contains the name of three services which comprise the applications and will yield a time-series where each row represents metrics from all platforms at a certain time (index). This will aggregation is especially helpful with novice users of Big Data platforms for developing data intensive applications as it is a significant simplification of the standard querying mechanism.

It is important to note that we can also specify the interval of queried metrics. For example if we collect system metrics every 5 seconds and YARN specific metrics every 10 seconds we will have an issue in that for every YARN metric we will have 2 system metrics. In order to mitigate this issue we can set the interval to 10s which will instruct DMon to create an average of all metrics with a step size of 10 seconds, meaning that instead of two system metrics we will have an average of those two. This interval can be set to seconds, minutes, hours etc. Of course this type of post processing might result in substantial data loss. A better solution would be to set the polling period of all metrics to the same value. This polling period can be set at runtime from DMon Monitoring Management API.

Listing 1: User defined DMon query

```
{
"DMON":{
  "query":{
  "size":"<SIZEinINT>",
  "ordering":"<asc|desc>",
  "queryString":"<query>",
  "tstart":"now−30s",
  "tstop":"<stopTime>"
   }
 }
}
```

[26]http://pandas.pydata.org/

Listing 2: Aggregated DMon query

```
{
 "DMON": {
  "aggregation":
  "system;spark;mongodb",
  "fname": "output",
  "index": "logstash-*",
  "interval": "10s",
  "size": 0,
  "tstart": "now-1d",
  "tstop": "now"
 }
}
```

The dmon-controllers synchronous endpoint is suitable only for small queries. Long running queries should be executed using the asynchronous query endpoint. This endpoint is able to start sub-processes which handle the issued queries. The number of sub-processes can be set during the deployment (default is 5). We opted to create a bespoke asynchronous task queue for DMon as this allowed us more control over the implementation and behavior while at the same time avoiding the bloating with unnecessary dependencies such as Celery for Python which requires an external broker such as RabbitMQ.

In order to deploy, manage and configure logstash instances, in our implementation we make use of the *dmon-shipper* microservice. Unlike the dmon-controller, that can be located on different machine than the one containing the Logstash instance, the dmon-shipper must be collocated on the machine. Its main purpose is to control nodes that are configured with the Elasticsearch cluster, allowing external tools to interact with the inner components of DMon.

The use of microservices in this context allows by design split of control for the various core and node-level components and the application logic of DMon. Doing so we make sure that we separate the application logic form code that drives and enacts individual components of the system. Using this design schema we managed to implement most of the components to be stateless, with the exception of dmon-controller. This implies that the current state of the controlled component is not stored but rather polled at a moment in time. In the case of dmon-controller service only basic state and node-level information is stored in a relational database. Exporting, importing, versioning or even backing up the state of dmon-controller can be performed directly from the Management API.

Queuing service

There are some cases where the metrics sent by node-level components might exceed Logstash capabilities of efficient processing, causing data-loss. In order to mitigate this issue we found three viable solutions. The first one suggests increasing the number of assigned workers for the filter plugin. As a second solution, one could create another instance of Logstash to handle part of the load, this should be done only if increasing the number of workers is not an option. Thirdly we propose to use a queuing service that receives all the metrics and from where Logstash can consume them in a timely manner. By using the third option we address data loss problem but it could potentially increase the time spent by a metric from when it was collected until it was processed and indexed inside DMon. Figure 1 presents the general framework design of this solution, where possible implementation candidates range from Kafka, or Redis to a combination of MongoDB [15] and RabbitMQ [38] are present. Section VI details on the behavior of Logstash in these situations.

III.3.2 Node-level Services

Management and configuration of all node-level components is delegated to the *dmon-agent* service. Each of the nodes that is monitored using the DMon solution must have a dmon-agent installed and running locally. The service is designed to be stateless, in a similar manner to dmon-shipper and dmon-indexer services. For the control of node-level components, dmon-controller issues requests to each dmon-agent service. Each of the requests is formulated as JSON containing all the needed information for the control of node-level components.

Each time a new node is added to the platform for monitoring, the platform automatically installs the monitoring agent on it. The monitoring agents collect data from local files and sends the data to the platform.

As in the case of most monitoring solutions, DMon presents a web user interface that allows user to easily interact with the platform for viewing and management purpose. All the metrics collected form the Big Data deployments are also presented. Being a separate component, the web user interface can be deployed on a separate machine provided that it has access to the appropriated DMon endpoints.

III.4 Visualization interface

Although the platform doesn't implement any bespoken metrics visualization facilities, one can easily use Kibana dashboards [21] including graphics for any number of metrics, such as CPU, memory or network, as well as Big Data specific metrics. All these visualizations are based on Elasticsearch queries that can be aggregated and plotted using a histogram based on their timestamps. Some visualizations are created automatically by DMon and then saved into a special index inside Elasticsearch, from where they are loaded into Kibana. This feature has proven extremely useful for developers and end users who are not yet familiar with Big Data frameworks. It is also possible to add additional visualizations manually to fit the end-users needs.

IV. Performance Metrics

The metrics platform collects can be classified into three major categories. First category is represented by resource-level metrics, such as CPU, memory and network utilization at the VM level. We can also include in this category metrics related to failure rates of specific VMs and the services running on them.

Second category is related to system level metrics. Among these metrics we mention those specific to each supported Big Data framework. Some examples are: job arrival rate, job throughput, job parallelism, job response time, waiting buffer occupancy and so on. We can easily see that these metrics are designed to monitor how the framework executes and schedules jobs. Each job is decomposed into different tasks. Third and smallest category of metrics is focused on worker level task metrics [41].

The overwhelming number of metrics that is exportable for Big Data frameworks is an important problem that has to be addressed in all monitoring solutions. There are well over 300 metrics that can be collected for YARN, HDFS, Spark, Kafka and Zookeper. All metrics have to be pre-process by Logstash and finally loaded into Elasticsearch for indexing. The metrics are usually received in a raw format that is processed by Logstash and then transformed into JSON codification that is indexable by Elasticsearch.

When we are talking of potentially thousands of VMs each sending 30 to 40 messages every 15 seconds, it is easy to see that the sheer volume of data can be a potential problem. This issue has several possible solutions such scaling both Logstash and Elasticsearch instances to handle the increased throughput or decrease the polling period for each VM from 15 seconds to a more manageable rate.

During the development stages of a Data intensive application (DIA) usually it is not clear what metrics are crucial so filtering of what metrics are indexed is not a viable solution in a DevOps environment. Once a suitably stable version of an application has been developed DMon can be setup in such a way that it indexes only certain types of metrics.

Another common way of reducing the load on Logstash instances would be to handle some transformation of the metrics before they reach the monitoring platform. Although, this solution is tempting it is not usable in DevOps. The act of in situ transformation may cause unwanted load on the monitored nodes thus introducing noise in the monitoring data.

Some metrics are strictly linked to the overall topology of the application being developed. This has to be addressed in the codification of metrics that are to be introduced into the monitoring solution. A good example of this issue are the bolts and spouts that make up a Storm topology. There are metrics related to each individual bolt and spout. If a global spout or bolt encoding is used there is the potential of losing valuable performance insight. Because, of this in DMon we collect metrics related to each bolt and spout individually. Moreover, we are able to automatically detect all running topologies on any given Storm instance and then extract all metrics related to each individual bolt and spout.

Trace checking is also an invaluable source of performance data. Log files contain timestamped data related to the internal state of any data intensive application. Logs are collected and then analyzed in order to check whether they satisfy a property which is usually specified in a logical language or rule. Returning to the Storm topology from before a property can be something like; all the emit events of a certain bolt occur not more than ten milliseconds after the latest receive event. This rule can be true or false.

V. Anomaly Detection

Anomaly Detection is an important component involved in performance analysis of data intensive applications. We define an anomaly as an observation that does not conform to an expected pattern [14, 19]. Most tools or solutions, such as Sematex[27], Datadog[28] etc. are geared more towards a production environment in contrast to this the DICE Anomaly Detection Tool (ADT) is designed to be used during the development phases of Big Data applications.

[27]https://sematext.com/spm/

[28]https://www.datadoghq.com/

V.1 Big Data framework metrics data

In DICE most data preprocessing activities are done within DMon [27]. However, some additional preprocessing such as normalization or filtering will have to be applied at method level.

In anomaly detection the nature of data is a key issue. There can be different types of data such as: binary, categorical or continuous. We usually deal with continuous data types although categorical or even binary values could be present. Most metrics data relate to computational resource consumption, execution time etc.. There can be instances of categorical data that denotes the status/state of a certain job or even binary data in the form of Boolean values. This makes the creation of data sets on which to run anomaly detection an extremely crucial aspect of ADT, because some anomaly detection methods don't work on categorical or binary attributes.

It is important to note that most, if not all, anomaly detection techniques and tools, deal with point data, meaning that no relationship is assumed between data instances [19]. In some instances this assumption is not valid as there can be spatial, temporal or even sequential relationships between data instances. This in fact is the assumption we are basing ADT on with regard to the DICE context.

Data used by the anomaly detection techniques are queried from DMon. This means that some basic statistical operations (such as aggregations, median etc.) can already be integrated into the DMon query. In some instances this can reduce the dataset size on which to run anomaly detection.

V.2 Types of anomalies

An extremely important aspect of detecting anomalies in any problem domain is the definition of anomaly types that can be handled by the proposed method or tool. In the next paragraphs we will give a short definition of anomaly classes.

First we have point anomalies which are the simplest types of anomalies, represented by data instances that can be considered anomalous with respect to the rest of data [14]. Because this type of anomaly is simple to define and check a big part of research effort will be directed towards finding them. We intend to further investigate this type of anomalies and consider them for inclusion in DICE ADT. However, as there are a lot of existing solutions already on the market this will not be the main focus of ADT instead we will use the Watcher[29] solution from the ELK stack to detect point anomalies.

A more interesting type of anomalies are the so called contextual anomalies. These are considered anomalous only in a certain context and not otherwise. The context is a result of the structure from the data set. Thus, it has to be specified as part of the problem formulation [37, 14]. When defining the context we consider: contextual attributes which are represented by the neighbours of each instance and behavioural attributes which describe the value itself. In short, anomalous behaviour is determined using the values for the behaviour attributes from within the specified context [14].

[29]https://www.elastic.co/guide/en/watcher/current/index.html

We consider the case of time-series data which is the most common type of data in which contextual anomalies can occur. Also, the meaningfulness of contextual anomalies is heavily dependent of the target application domain. Because of this in the context of our tool we must have a set of anomalies for each of the Big Data services covered. In this paper, the main focus is on creating a basic framework that enables ad-hoc definition of context rather than an exhaustive list of predefined ones. Future work will also feature some instances of these predefined contexts and anomalies.

The last types of anomalies are called collective anomalies. These anomalies can occur when a collection of related data instances are anomalous with respect to the entire data set. In other words, individual data instances are not anomalous by themselves. Typically collective anomalies are related to sequence data and can only occur if data instances are related.

There are a variety of methods currently implemented for ADT from unsupervised methods such as IsolationForest, DBSCAN to supervised methods such as Decision Trees, Naive Bayes etc. In order to have the flexibility to use as many methods as possible we have integrated into ADT both the Weka [25] and scikit-learn [35] libraries. As of writing this article we are in the final stages of integrating Googles Tensor-flow deep learning library [1]. This will allow us to use state of the art deep learning techniques for anomaly detection.

V.3 Anomaly detection implementation

The ADT is made up of a series of interconnected components that can be controlled using a simple Eclipse Plugin or a command line interface.

In total there are 8 components that make up ADT. The general architecture can be seen in Figure 2. These are meant to encompass each of the main functionalities and requirements identified in the requirements identified during DICE [17].

First we have the *dmon-connector* component which is used to connect to DMon. It is able to query the monitoring platform and also send it new data. This data can be details related to detected anomalies or it can also save trained predictive models. For each of these types of data *dmon-connector* creates a different index inside DMon.

After the monitoring platform is queried the resulting dataset can be in JSON, CSV or RDF/XML. However, in some situations additional formatting is required. This is done by the *data formatter* component. It is able to normalize the data, filter different features from the dataset or even window the data. The type of formatting the dataset may or may not need is highly dependant on the anomaly detection method which is used.

The *feature selection* component is used to reduce the dimensionality of the dataset. Not all features of a dataset may be needed to train a predictive model for anomaly detection. So, in some situations it is important to have a mechanism that allows the selection of only the features that have a significant impact on the performance of the anomaly detection methods. Currently only two types of feature selection is supported. The first is *Principal Component Analysis*[30] (from Weka) and *Wrapper Methods*.

The next two components (see Figure 2) are used for training and then validating anomaly detection predictive models. For training

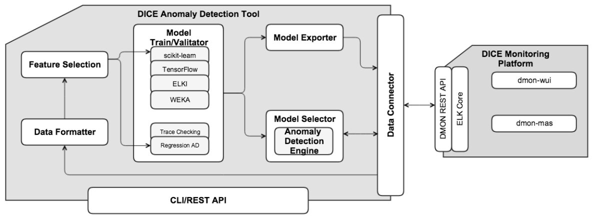

Figure 2: General overview of Anomaly Detection Stack.

a user must first select the type of method desired. The dataset is then split up into training and validation subsets and later used for cross validation. The ratio of validation to training size can be set during this phase. Parameters related to each method can also be set in this component.

Validation is handled by a specialized component which minimizes the risk of overfitting the model as well as ensuring that out of sample performance is adequate. It does this by using cross validation and comparing the performance of the current model with past ones.

Once validation is complete, the *model exporter* component transforms the current model into a serialized loadable form. We use the PMML [23] format wherever possible in order to ensure compatibility with as many machine learning frameworks as possible. This also makes the use of ADT in a production like environment much easier.

The resulting model can be fed into DMon. In fact the core services from DMon (specifically Elasticsearch) have to role of a serving layer from a lambda architecture. Both detected anomalies and trained models are stored in the DMon and can be queried directly from the monitoring platform. In essence this means that other tools from the DICE toolchain need to know only the DMon endpoint in order to see what anomalies have been detected.

Furthermore, the training and validation scenarios is in fact the batch layer while unsupervised methods and/or loaded predictive models are the speed layer. Both these scenarios can be accomplished by ADT. This integration will be further detailed in later sections.

The last component is the *anomaly detection engine*. It is responsible for detecting anomalies and for pushing them to the serving layer (i.e. DMon) via the *dmon-connector*. The *anomaly detection engine* is also able to handle unsupervised learning methods. We can see this in Figure 2 in that the Anomaly detection engine is in some ways a sub-component of the *model selector* which select both pre-trained predictive models and unsupervised methods.

As the ADT is out of the scope of this article we will not give any more details regarding the validation and experiments done using different types of anomaly detection methods. The most important thing to remember is the fact that ADT is tightly coupled with DMon and together they for a Lambda Architecture. Each instance of ADT can have the role of batch or speed layer while DMon has the role of a serving layer.

[30]http://weka.sourceforge.net/doc.dev/weka/attributeSelection/PrincipalComponents.html

V.4 Trace checking tool

Trace checking is also a valuable source of detecting sequential anomalies. In the case of DICE this job done by a tool called DICE-TraCT [9] based on Soloist [11]. The trace checking tool is based on 3 main components; trace checking engine which is in charge of performing the actual trace analysis, log merger aggregates all events relevant to a particular event based on each events timestamp, finally we have Tractor which coordinates the other two components.

The trace checking tool issues a query to DMon requesting log files from all pertinent nodes for a given data intensive application. DMon then fetches (via *dmon-agetns*) the raw log files and creates a archive which is returned to the log merger component for processing.

Similarly to the ADT tool the trace checking tool is beyond the scope of this paper. It has been extensively covered in other publication [9].

VI. Platform Validation

Data intensive applications pose some difficult and unique problems. In almost all instances these are based on Big Data frameworks and libraries which are geared towards large scale distributed workloads. From the point of view of monitoring this entails the collection, processing, storing and querying of data from hundreds potentially thousands of processing nodes. As we have seen in section IV the available metrics from each of the supported technologies add up to hundreds of distinct metrics.

Furthermore, as DMon is a bespoked DevOps monitoring solution for data intensive applications the collection and processing of performance relevant metrics is one of the most important requirements. Polling period of metrics in development and performance optimization scenarios can be extremely low, once every second for all available metrics. During the development of DMon we have used what we consider a minimum YARN and Spark deployment comprised of 4 VMs. This number was chosen because it allows us to start YARN and Spark specific services as recommended in their documentation. In the case of YARN we have deployed Node Managers and Data Nodes on all VMs, Resource Manager, Name Node and Secondary Name Node on distinct VMs, minimizing the collocation of key services and ensuring a balanced role distribution. In other words interference/noise between monitored performance metrics for Yarn and Spark services are reduced.

These 4 VMs and the services deployed on them generate (when running a YARN/Spark app) 23 monitoring events per second with a polling period of 5 seconds. If the polling period is set to 1 second the number of events increases to 115 events. It is important to consider that most of these events have more than 1800 characters and have to be parsed and then formatted into the correct JSON format for indexing in Elasticsearch. Furthermore, the Grok plugin from Logstash performs substring matching by default which have a significant impact on performance which is mitigated by the use of anchor characters.

We can safely conclude that in many instances the challenges and requirement for DMon closely mirror the challenges found in the data intensive applications it is designed to monitor both in the volume, variety and velocity dimensions. These facts have let us to devise a comprehensive set of experiments and load testing scenarios to see how DMon and its core service can handle (both process and index) the incoming monitoring data as well as how other tools can get access to this data in near-real time.

All experiments have been executed on a private OpenStack Mitaka deployment. The VMs used all have 4 vCPUs 8 GB of RAM and 250 GB of storage tuning Ubuntu 14.04.3 LTS on them. Resource contention is not an issue as these experiments where the only VM instances running on the cluster.

VI.1 Core Service Experiments

Phase 1

The first set of experiments was done on the Logstash based core service which, as mentioned in Section III.1 is responsible for ETL operations of incoming events. All events are unprocessed when entering Logstash. Once they go through the custom filters (where all ETL operations are contained) the resulting processed events can be ingested by Elasticserach. There are a wide range of raw event format which required us to create custom filters.

It should be noted at this time that Logstash supports input codecs[31] for a wide range of formats. However, we have identified two problems when dealing with Big Data framework metric events. First, almost no framework outputs metrics in a supported codec. Second, we have found that some frameworks such as Storm and Spark can output in JSON format but this output is nested. This causes significant problems when trying to index nested fields in Elasticsearch. Our solution is to flatten the input JSONs as much as possible using mutate operations in the Logstash filters.

The first set of experiments are meant to show the maximum throughput (measured in events per second) of the Logstash based core service when running custom framework specific DMon filters. For these experiments we have captured raw events from all of the supported Big Data Technologies (see section VI.3). We have identified all of the unique messages and created a test suite that is sending these messages into Logstash without delay, effectively setting the polling period to the lowest possible value.

There are a number of configuration options and parameters which we had to take into consideration. Logstash allows the setting of the number of pipeline workers. This option will allow the execution in parallel of the filter and output sections of the filter pipeline. By default this is set to the number of CPU cores. This option sets the number of workers that will, in parallel, execute the filter and output stages of the pipeline. If you find that events are backing up, or that the CPU is not saturated, consider increasing this number to better utilize machine processing power. The default is the number of hosts CPU cores.

Another important configuration option is the batch size parameter. This option defines the maximum number of events an individual pipeline worker before it attempts to executes the filters and outputs the result. By default this is set to 125 events. Increasing it usually results in better efficiency at the cost of increase in memory overhead. In [16] we have investigated different JVM settings and garbage collection methods in order to optimize throughput. In these experiments we set the heap size 2 GB with the default JVM garbage collector.

The experiments where conducted by setting batch and the number of pipeline workers using the DMon generated filter configu-

[31] https://www.elastic.co/guide/en/logstash/current/codec-plugins.html

Experiment	Number of workers	Batch size
Base-line(no filters)	4	125
Exp-0	1	125
Exp-1	2	125
Exp-2	4	125
Exp-3	6	125
Exp-4	12	125
Exp-5	4	250
Exp-6	4	500
Exp-7	4	1000
Exp-8	4	2000
Exp-9	4	10000

Table 1: Logstash Experiment Settings

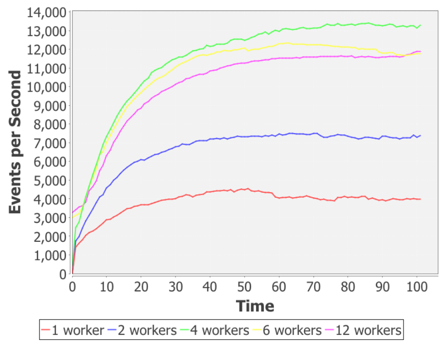

Figure 3: Different pipeline worker settings event output

rations. All test where run for 30 minutes collecting performance metrics every second. In order to ensure that these empirical experiments are deterministic we have repeated the experiments a total of 5 times. Table 1 shows the experimental setting used for each experimental run. Before we started to test the DMon filter we run our experiments with no filters set. Effectively passing the input directly to the output with no ETL processing with a batch size of 125 and 4 pipeline workers. In this case we got a maximum throughput of 16550 events per second.

Figure 3 shows the total amount of events processed for each worker configuration. For these first experiments we have set only the pipeline worker number and left the batch size default value. The result show that the best configuration is the one where the number of workers is equal to the number of cores. Figure 4 also shows that although the number of events processed is somewhat lower when over-saturating the worker count, impact on overall CPU load is quite significant.

Heap usage percentage is exemplified in figure 5 and shows that for the first 4 experimental runs the heap consumption is fairly stable jumping considerably when executing 12 pipeline workers on our deployment.

The next phase of our experiment dealt with seeing how the batch size influences the event throughput. We have run a total of 5 experiments with batch sizes; 250, 500, 1000, 2000 and 10000. The number of workers was set to 4 as this has yielded the best

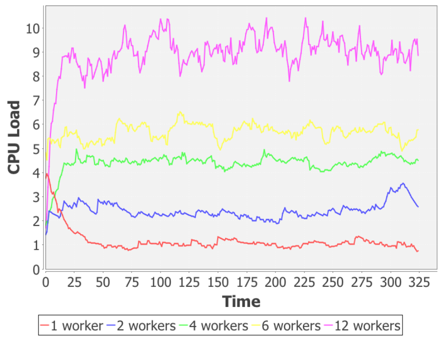

Figure 4: Short term CPU load

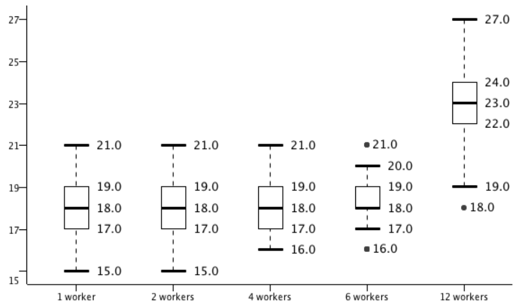

Figure 5: Heap used per experimental run

performance in the previous battery of experiments.

We can see in figure 6 that the best performing setting was the one using a batch size of 250. It is surprising that an increased batch size does not yield a better throughput. Even more surprising is that the worst performing experiment had a batch size of 1000 while the second best had the biggest batch size. Figure 7 show the percent of heap usage during these experiments. As expected there is a strong correlation between batch settings and heap usage. The biggest batch size consumed 65% of the available heap.

Figure 8 shows quite clearly that the increased heap usage affects the garbage consumption significantly. The box plot shows that

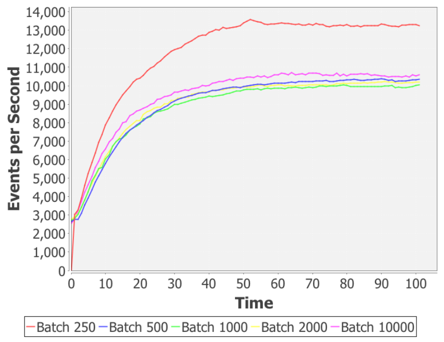

Figure 6: Events per second for batch size comparison

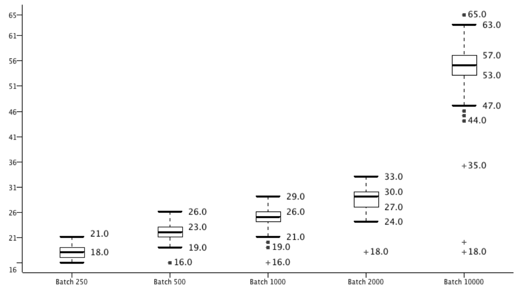
Figure 7: Heap consumtion for batch size experiments

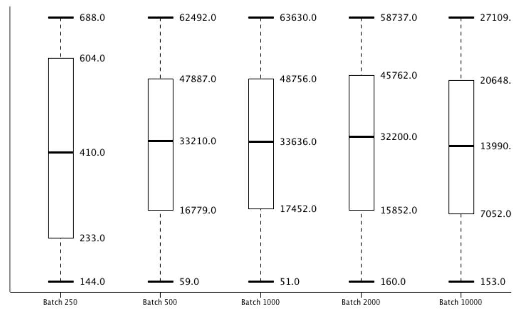
Figure 8: Garbage collections in milliseconds (normalized)

Field Data Cache	20%
Cache Filter	20%
Memory Index Buffer	30%
Memory Min Shard Index Buffer	12 MB
Memory Min Index Buffer	96 MB

Table 2: Elasticsearch memory settings

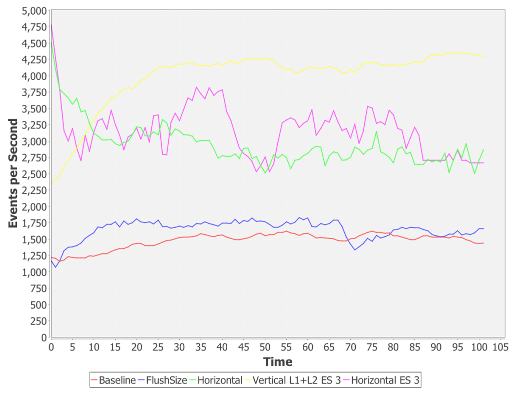
Figure 9: Processed Events

there is almost a 10 fold increase in garbage collection times which negatively effects performance. In some situation all incoming events can be stored in memory (using the batch size setting) and then flushed all at once into then DMon generated filter. However, this would mean a greater garbage collection time. A balance between heap size and garbage collection is important when dealing with applications running on JVM. In the case of our experiments we have found that for the current event types the best configuration is to use 1 pipeline worker per vCPU core and to calculate the batch size based on the current heap value. In our experiments for a heap size of 1GB the best performing batch size is 250 for the DMon generated events.

At this point it is important to mention that during the experiments detailed so far we used a simple file output, meaning that all processed messages where written to a file in their final form (flat JSON representation of the events). This was done so that we could verify the correct parsing of each event. We only considered valid those events which where correctly parsed. During the experiments only 12 parsing failures have been detected making them statistically insignificant.

Phase 2

Once the data has went through ETL processing it is fed into Elasticsearch to be indexed. This introduces a new variable into our experiments. We ran a series of experiments to help us maximizes the throughput of indexed events.

Memory usage and management is extremely important. In Elasticsearch once a string is analyzed on aggregated for a query it is loaded into field data which resides in memory. Field data structure is created at query time not during indexing because of this it is cheaper to load all of the data once into memory than piece by piece. For example if you have a complex aggregated and selective query which returns 20 hits filed data will not be loaded only for those 20 hits but form the entire index for the fields queried. This results in very good querying speed but is costly when it comes to heap allocation.

Table 2 show the field data and buffer policy used during our experiment. Field data cache size for example is the percentage of heap reserved for field data. All listed parameters effect the query and indexing performance of Elasticsearch. These parameters where set using the Management API from DMon. Heap size can also be defined and enforced from DMon however we have found that setting it to half the available system memory yields the best results. In our case the heap was set to 2 GB.

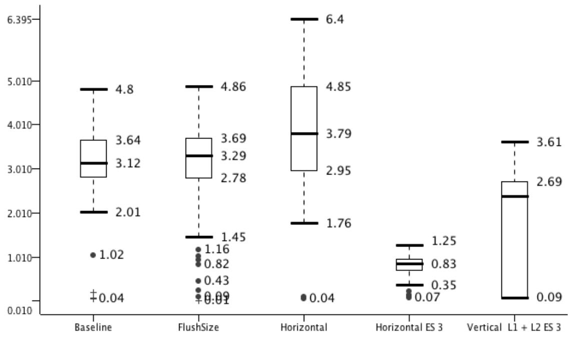
Figure 10: Elasticsearch cluster CPU load

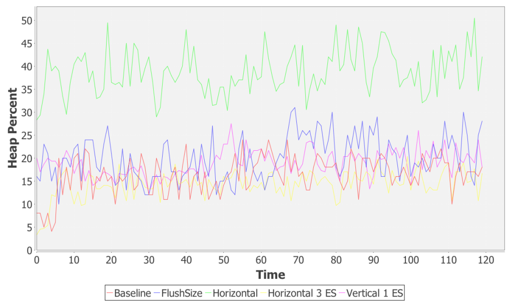

Figure 11: Elasticsearch cluster Heap usage

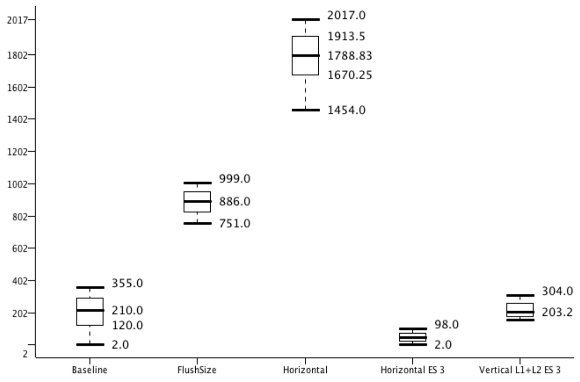

Figure 12: Garbage Collection count

Because both core service from DMon are horizontally scalable we designed the next set of experiments to test how scaling effects the performance measured in event indexing rate. First we ran an experiment where all settings were set to default. We called this run as "Baseline". In particular we left the flush size for the Elasticsearch output from Logstash to the default value. For the next experiment we set this value to be the same as the batch size from Logstash ("FlushSize").

For vertical scaling experiments we decided to run a Logstash instance on a bare metal server having 12 CPU cores (running the same number of pipeline workers and heap size) feeding data to a single Elasticsearch instance ("Horizontal"). For the next step we ran the same experiment with a 3 VM Elasticsearch cluster ("Horizontal 3 ES"). Finally we ran 2 Logstash instances feeding data into a 3 VM Elasticsearch cluster. The Logstash instances used the same settings as the one found in the second experiment ("Vertical L1+L2 ES 3")

Figure 9 show the number of events indexed for each experimental run. The first thing that we have noticed is that the amount of events indexed is much lower that the amount of events processed by Logstash. The Logstash setting which performed the best during the first phase of the experiments, processing on average 13,500 events leads to an indexing rate in the mid thousand range. We have also seen that matching the batch size settings with the flush size we got an additional 500 to 600 events indexed every second.

One of the conclusions we arrived early in the development is that the major performance bottleneck is Elasticsearch. This conclusion is enforced by the Elasticsearch CPU load, seen in figure 10, heap usage seen in figure 11 and JVM garbage collection count from figure 12. This is very obvious when we see how much more garbage collection is being done when running the horizontal scaling experiment.

The best performing setup was the vertical scaling of both core services. We decided to use 2 Logstash and 3 Elasticsearch instances (in order to avoid a "split brain" scenario). We manage to index over 4500 events. An interesting observation was that the horizontal scaling experiments yielded some interesting results. At first it seemed surprising of how much less events we where able to process compared to the vertical scaling. However, this behavior was explained when checking the Elasticsearch heap usage (Fig. 11) CPU load (Fig. 10). We can see how Elasticsearch is using a lot more computational resource while indexing a lot less events.

We can also see that for DMon, horizontal scaling works best for both Elasticsearch and Logstash. Although we could get better performance in some instances by raising the heap size for Logstash and Elasticsearch the benefits are not as immediate.

VI.2 REST Service Experiments

The scaling experiments that we have detail this far where only for the cores services on which DMon relies. In this next section we describe the testing done for the REST service. In the DICE toolchain all tools that require run-time monitoring data information relies on this service such as the verification tool called D-Vert[10]. As mentioned in section III we have split up the REST API into two components. For these experiments we re focused only on those resources which have to be accessible by other tools and users (i.e. Monitoring API) not the resources required for configuration and setup (i.e. Management API).

For these experiments we deployed DMon on a workstation which has an Intel Xeon E5-2630v3 processor with 8 physical cores clocked at 2.4 Ghz with a max turbo clock at 3.2 Ghz, 32 GB of ECC RAM and two 512 GB Toshiba SATA SSDs in RAID 0. The operating system used was Ubuntu 14.04.3 LTS.

Monitoring API

For the Monitoring API experiments we decided to use a load testing tool called Locust[32] written in Python. Locust allowed us to create a custom DMon user profile which simulates the interaction of DICE tools with DMon. We where able to create separate tasks for each resource and then create different user scenarios based on these called task sets which describe DICE tool behaviors Basically assigning a weight to each task from a task set.

DMon was implemented using Flask so during much of the development we have used the on board WSGI library called Werkzeug[33]. Because this default CGI library is only meant for development and tasting we have looked into running DMon using several different WSGI implementations such as:

- *Gunicorn*: Is a Python based WSGI HTTP server ported from the Unicorn Ruby Project. One of it's main strengths is that it is compatible with a wide array of web frameworks and that it has automatic worker process management

[32]locust.io
[33]http://werkzeug.pocoo.org/

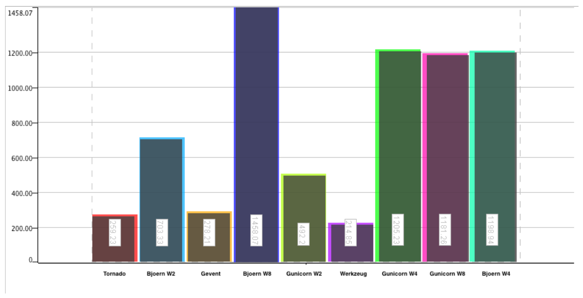

Figure 13: WSGI Experiment results using Locust

- *Tornado*: Is a scalable non-blocking web server written in Python.
- *Gevent*: Is a Python networking library that uses greenlets to provide asynchronous API on top of libev event loop.
- *Bjoern*: Is a small WSGI server written in C with a shallow memory footprint.

Both Bjoern and Gunicorn allow for the configuration of the number of background workers used. We have decided on running both of them with 2, 4 and finally 8 background workers.

All experiments where run 5 times to ensure that the result where repeatable. In contrast to the experiments detailed in section VI.1 each experimental run contained 1 million request for each setup.

Figure 13 shows the results of these experiments. We can see that the best performing by far was the Bjoern deployment managing an impressive 1480 request per second. Even the worst performing, Werkzeug managed a respectable 214.85 requests. We can also see that although Gunicorn performed extremely similar to Bjoern when 4 background workers where set this did not scale as well as in the case of Bjoern.

Query API

The experiments mentioned in the previous sections used 4 sets of DMon queries constructed in such a way that they returned monitoring data from the past 15 minutes. These queries are extremely quick and take well under 1 second. In this section we focus on queries which are longer running to see how DMon can handle such long running intensive workloads. There are a number of different query interfaces in DMon. These were detailed in section III.3. For these experiments we use the asynchronous query implementation.

Figure 14 shows the performance of using asynchronous querying versus sequential querying. For these experiments we have set the maximum allowed parallel query to 50 however this can be modified when DMon is first deployed. We have observed that we get almost 5 fold speedup when issuing concurrent queries.

VI.3 Use-cases

During the DICE project several use cases have been defined on which all of the developed tools were evaluated. Because DMon is at the core of the run time part of the DICE solution all use cases have evaluated this tool. The three use cases detailed in the following sections have been used to validate DMon with respects to the following key performance indicators; reduction in the time required to configure a monitoring solution and small monitoring agent overhead [5]. For the sake of completeness we will detail in the next few subsections an outline of the use cases and how they utilized and evaluated DMon.

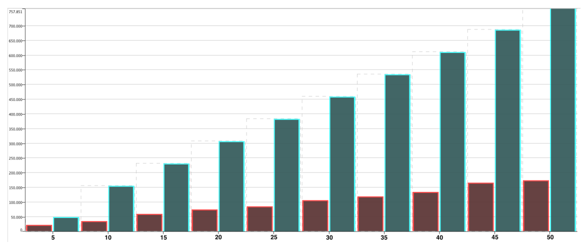

Figure 14: DMon Query Experiment, a value on vertical axis represents the amount of time it takes the queries to finish, a value on the horizontal axis represent the number of concurrent queries.

VI.3.1 NewsAsset

NewsAsset[34] is a commercial product developed and maintained by Athens Technology Center (ATC) that handles large volumes of news and media, providing the users with solutions for management, storage and delivery of sensitive information. The application incorporates state of the art technologies and software engineering practices facilitating selection of interesting media events present in the digital world. These technologies can deliver efficient processing and can increase the added value to journalists.

Alongside NewsAsset a combination of social, sensor and several other networks that are connected to Internet continuously provide end users and the entire ecosystem with a variety of real data at a tremendous pace. As more of those sources enter the digital era, journalists will be able to access data from such events, helping not only with accurate information about disasters but also in order to generate news stories. As that trend plays out, when a disaster is happening somewhere in the world, it is the social networks like Twitter, Facebook, Instagram, etc. that people are using to watch the news ecosystem and try to learn what damage is where, and what conditions exist in real-time. In many of the cases people directly involved in the event will most probably snap photos of the disaster and post it on social media. Subsequently, news agencies have realized that social-media content are becoming increasingly useful for disaster news coverage and can benefit from this future trend only if they adopt the aforementioned innovative technologies. Thus, the challenge for NewsAsset is to catch up with this evolution and provide services that can handle the developing new situation in the media industry.

The NewsAsset is integrated in the DICE framework and uses most of the technologies developed in the project. It uses the tools in order to design and deploy Data Intensive Applications for gathering big streams data released in the media. Data collected by NewsAsset come from a heterogeneous environment that mixes social networks, websites, RSS feeds and so on. Before data can be used it is processed in order to reduce its dimensionality by removing duplicated information; also the tool is capable to extract the trends.

[34] http://ilab.atc.gr/projects/dice

In this use case DMon was utilized in monitoring performance related metrics for the underlying Big Data platforms on which NewsAsset is built. In particular DMon collected information regarding the Storm topologies used as well as Solr search engine and MongoDB. It also reduced configuration time in comparison by 40% with the results presented in [5].

VI.3.2 Prodevelop usecase

Posidonia Operations[35] is an Integrated Port Operation Management System highly customizable that allows a port to optimize its maritime operational activities related to the flow of vessels in the port service area, integrating all the relevant stakeholders and computer systems.

In technical terms, Posidonia Operations is a real-time and data intensive platform able to connect to AIS (Automatic Identification System), VTS (Vessel Traffic System) or radar, and automatically detect vessel operational events like port arrival, berthing, unberthing, bunkering operations, tugging, etc.

Posidonia Operations is a commercial software solution that is currently tracking maritime traffic in Spain, Italy, Portugal, Morocco and Tunisia, thus providing service to different port authorities and terminals.

Having this scenario, several business and technical goals have been identified as a result of the future application of the DICE methodology and tools to the Posidonia Operations use case [8].

In contrast to the other use cases here DMon was utilized in monitoring newly developed components for the Posidonia solution. In particular it was used to monitor a Complex Event Processor (CEP) component which is designed for identifying different types of maneuvers that ships make in or near ports. In order to accomplish this we have utilized the ability of DMon to incorporate newly defined message formats with minimal user intervention. It required the addition of two user defined parsing instruction in the configuration file of DMon. It also reduced configuration time in comparison by 94% [5].

VI.3.3 Netfective use case

Within the frame of DICE, Netfective Technology built a minimal viable product (MVP) to appraise the capabilities of Big Data in e-government applications, especially for tax fraud detection [42]. Big Data technologies have already proven how much they are valuable to industries. Many businesses that have taken advantage of Big Data management and processing systems have increased their effectiveness and efficiency; whether it be for healthcare, advertising, or retail.

Fraud recognition requires a holistic approach, and a combined use of tactical or strategic methods and state-of-the-art Big Data solutions. Traditional fraud detection practices have not been particularly successful largely because they come into play after the fact. Big Data intelligence software can perceive the deviant behavior at real time, thereby enabling fiscal agencies to get better outcomes. Big Blu, the MVP, shall send an alert whenever a suspicious tax declaration enters the information system. In this instance DMon was utilized to collect Spark streaming and system performance metrics.

[35]https://www.prodevelop.es/en/posidonia-operations-0

VII. Conclusions and Future Work

In this paper we presented the architecture of the DICE Monitoring platform, which is a distributed, highly available platform for monitoring Big Data technologies.

The main goal of DMon is to create a monitoring platform that collects, stores and pre-processes monitoring data from state-of-the-art Big Data technologies. DMon is integrating monitoring data from a number of Big Data technologies, it supports a wide range of the big data platforms platform such as; HDFS, YARN, Spark, Storm, MongoDB, Cassandra. Engineered using a microservices architecture, the platform is easy to deploy, and operate, on heterogeneous distributed Cloud environments.

It has several advanced features and extensions such as advanced aggregated querying which enables a full overview of all of the sub systems that comprise a data intensive application. This exposes to developers a holistic view of all the performance metrics which can be not only visualized in human readable format (using DMon generated visualizations) but also exported in a variety of formats (JSON, RDF+XML, CSV) for consumption by other tools. Also, DMon features service auto-detection which allows it to self-configure and start collecting performance metrics with little to no user intervention.

We have show in this paper that the microservice based DMon architecture is easily scalable and can handle querying and post-processing of said queries for thousands of events. Also, in conjunction with the anomaly detection and trace checking tool it forms part of a larger lambda architecture, DMon having the role of serving layer which is responsible with storing not only monitoring data but also predictive models and detected anomalies.

Further research and development will focus on creating a more autonomous platform. In particular self-configuration will be a key issue that needs to be addressed by developing the multi-agent system based *dmon-mas* service. Currently we only support self-configuration on a limited scale (auto-detection of services to be monitored) we would like to extend this capability so that user intervention and setup is at a minimum.

As Big Data platforms are constantly changing and new ones are being developed we aim to support new technologies such as Apache Samza[36] and Apache Flink[37]. Exascale[36] systems are also just around the corner and monitoring such complex and large systems is a great challenge. In our opinion DMon could be used for such systems and we aim to develop functionalities that enables this.

Acknowledgments

This work has received funding from the EC-funded project H2020 DICE (Agreement 644869), which aims at providing a toolchain that makes the task of developing Big Data applications less daunting and the H2020 ASPIDE project (Agreement 801091). This work was partially supported by grants from Romanian Ministry of Research and Innovation, grant Acronim (PNIII-P4-ID-PCE-2016-0842) and grant BID (PNIII-P1-PDI-PFE-2018-028).

[36]http://samza.apache.org/

[37]https://flink.apache.org/

References

[1] Martín Abadi, Ashish Agarwal, and Paul Barham et. al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[3] Khalid Alhamazani, Rajiv Ranjan, Karan Mitra, Fethi Rabhi, Prem Prakash Jayaraman, Samee Ullah Khan, Adnene Guabtni, and Vasudha Bhatnagar. An overview of the commercial cloud monitoring tools: Research dimensions, design issues, and state-of-the-art. *Computing*, 97(4):357–377, April 2015.

[4] Vaggelis Antypas, Nikos Zacheilas, and Vana Kalogeraki. Dynamic reduce task adjustment for hadoop workloads. In *Proceedings of the 19th Panhellenic Conference on Informatics*, PCI '15, pages 203–208, New York, NY, USA, 2015. ACM.

[5] Danilo Ardagna, Laurie-Anne PARANT, Ismael Torres, and et. al. Final assessment report and impact analysis (d6.4). Technical report, H2020 DICE, 2018.

[6] Matej Artac, Tadej Borovsak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. Model-driven continuous deployment for quality devops. In *Proceedings of the 2nd International Workshop on Quality-Aware DevOps, QUDOS@ISSTA 2016, Saarbrücken, Germany, July 21, 2016*, pages 40–41, 2016.

[7] Luis Eduardo Bautista Villalpando, Alain April, and Alain Abran. Performance analysis model for big data applications in cloud computing. *Journal of Cloud Computing*, 3(1):1–20, 2014.

[8] Simona Bernardi, José Ignacio Requeno, Christophe Joubert, and Alberto Romeu. A systematic approach for performance evaluation using process mining: The posidonia operations case study. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*, QUDOS 2016, pages 24–29, New York, NY, USA, 2016. ACM.

[9] Marcello M. Bersani, Francesco Marconi, and Matteo Rossi. Trace checking of streaming applications through dice-tract. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 159–160, New York, NY, USA, 2018. ACM.

[10] Marcello M. Bersani, Francesco Marconi, Matteo Rossi, and Madalina Erascu. A tool for verification of big-data applications. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*, QUDOS 2016, pages 44–45, New York, NY, USA, 2016. ACM.

[11] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. Smt-based checking of soloist over sparse traces. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering*, pages 276–290, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[12] Mahantesh N. Birje and Sunilkumar S. Manvi. Wigrimma: A wireless grid monitoring model using agents. *Journal of Grid Computing*, 9(4):549–572, Dec 2011.

[13] Giuliano Casale, Danilo Ardagna, Matej Artac, and et. al. Dice: Quality-driven development of data-intensive cloud applications. In *7th IEEE/ACM International Workshop on Modeling in Software Engineering, MiSE 2015, Florence, Italy, May 16-17, 2015*, pages 78–83, 2015.

[14] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, 2009.

[15] K Chodorow and M Dirolf. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, 2010.

[16] D. N. Doan and G. Iuhasz. Tuning logstash garbage collection for high throughput in a monitoring platform. In *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 359–365, Sept 2016.

[17] Giuliano Casale et. al. D1.2 dice requirement specification. Technical report, 2016.

[18] M. Fowler. Microservice overview, Jan. 2016.

[19] Matthias Gander, Michael Felderer, Basel Katt, Adrian Tolbaru, Ruth Breu, and Alessandro Moschitti. Anomaly detection in the cloud: Detecting security incidents via machine learning. In Alessandro Moschitti and Barbara Plank, editors, *Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*, volume 379 of *Communications in Computer and Information Science*, pages 103–116. Springer Berlin Heidelberg, 2013.

[20] Elmer Garduno, Soila P. Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Theia: Visual signatures for problem diagnosis in large hadoop clusters. In *Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques*, lisa'12, pages 33–42, Berkeley, CA, USA, 2012. USENIX Association.

[21] C. Gormley and Z. Tong. *Elasticsearch: The Definitive Guide*. O'Reilly Media, 2015.

[22] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc., 1st edition, 2014.

[23] Robert L. Grossman, Stuart Bailey, Ashok Ramu, Balinder Malhi, Philip Hallstrom, Ivan Pulleyn, and Xiao Qin. The management and mining of multiple predictive models using the predictive modeling markup language. *Information & Software Technology*, 41(9):589–595, 1999.

[24] Juan Gutierrez-Aguado, Jose M. Alcaraz Calero, and Wladimiro Diaz Villanueva. Iaasmon: Monitoring architecture for public cloud computing data centers. *Journal of Grid Computing*, 14(2):283–297, Jun 2016.

[25] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[26] G. Iuhasz and I. Dragan. An overview of monitoring tools for big data and cloud applications. In *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 363–366, Sept 2015.

[27] Gabriel Iuhasz and Daniel Pop. Monitoring and data warehousing tools – initial version. *DICE EU H2020 Project Deliverable*, 2016.

[28] Matthew Jacobs. Challenges and lessons learned building monitoring anddiagnostics tools for hadoop. In *Proceedings of the 2012 Workshop on Management of Big Data Systems*, MBDS '12, pages 33–34, New York, NY, USA, 2012. ACM.

[29] Sean Kennedy and Lin Jiu. Facilitating collaboration and interaction across the enterprise with oslc. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 374–375, Riverton, NJ, USA, 2013. IBM Corp.

[30] A. Kertesz, G. Kecskemeti, M. Oriol, and title= et. al.

[31] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[32] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications, 2015.

[33] M. McCandless, E. Hatcher, and O. Gospodnetić. *Lucene in Action*. Manning Pubs Co Series. Manning, 2010.

[34] S. Newman. *Building Microservices: Designing fine-grained Systems*. O'Reilly Media, Incorporated, 2015.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[36] Swann Perarnau, Rajeev Thakur, Kamil Iskra, Ken Raffenetti, Franck Cappello, Rinku Gupta, Pete Beckman, Marc Snir, Henry Hoffmann, Martin Schulz, and Barry Rountree. *Distributed Monitoring and Management of Exascale Systems in the Argo Project*, pages 173–178. Springer International Publishing, Cham, 2015.

[37] Hesam Sagha, Hamidreza Bayati, José Del R. Millán, and Ricardo Chavarriaga. On-line anomaly detection and resilience in classifier ensembles. *Pattern Recogn. Lett.*, 34(15):1916–1927, November 2013.

[38] Gabriele Santomaggio and Sigismondo Boschi. *RabbitMQ cookbook*. Packt Publ., Birmingham, 2013.

[39] J. Turnbull. *The Logstash Book:*. James Turnbull, 2013.

[40] J. Venner, S. Wadkar, and M. Siddalingaiah. *Pro Apache Hadoop*. Apress, 2014.

[41] X. Wu, Y. Liu, and I. Gorton. Exploring performance models of hadoop applications on cloud architecture. In *2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 93–101, May 2015.

[42] Alexis HENRY Youssef RIDENE. Legacy data migration and fraud detection using blu age and big data. Technical report, BluAge, 2015.

[43] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.