

**Developing Data-Intensive Cloud  
Applications with Iterative Quality  
Enhancements**



# **Monitoring and data warehousing tools – Final version**

**Deliverable 4.2**

---

<b>Deliverable:</b>	D4.2
<b>Title:</b>	Monitoring and data warehousing tools – Final version
<b>Editor(s):</b>	Daniel Pop (IEAT)
<b>Contributor(s):</b>	Daniel Pop (IEAT), Gabriel Iuhasz (IEAT)
<b>Reviewers:</b>	Danilo Ardagna (PMI), Damian A. Tamburri (PMI), Simona Bernardi (ZAR), Giuliano Casale (IMP)
<b>Type (R/P/DEC):</b>	Demonstrator
<b>Version:</b>	1.0
<b>Date:</b>	31-January-2017
<b>Status:</b>	Final version
<b>Dissemination level:</b>	Public
<b>Download page:</b>	<a href="http://www.dice-h2020.eu/deliverables/">http://www.dice-h2020.eu/deliverables/</a>
<b>Copyright:</b>	Copyright © 2017, DICE consortium – All rights reserved

---

#### DICE partners

---

<b>ATC:</b>	Athens Technology Centre
<b>FLEXI:</b>	Flexiant Limited
<b>IEAT:</b>	Institutul E Austria Timisoara
<b>IMP:</b>	Imperial College of Science, Technology & Medicine
<b>NETF:</b>	Netfective Technology SA
<b>PMI:</b>	Politecnico di Milano
<b>PRO:</b>	Prodevelop SL
<b>XLAB:</b>	XLAB razvoj programske opreme in svetovanje d.o.o.
<b>ZAR:</b>	Unversidad de Zaragoza

---



The DICE project (February 2015-January 2018) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644869

## Executive summary

The deliverable presents the final status of the monitoring and data warehousing platform, DICE Monitoring Platform, developed in T4.1. The deliverable is a follow-up of D4.1<sup>1</sup> and for a complete picture of the DICE Monitoring Platform (DMon) the reader can refer to both documents as some aspects, such as the architecture of the platform and deployment scenarios, are solely described in D4.1. The focus of this deliverable is on the achievements of the second year efforts and on the evaluation of the platform and its integration with other components of DICE framework.

DMon is central to the DICE runtime system being used by other tools to retrieve runtime data produced by Big Data frameworks while running data intensive applications (DIA). The platform relies on a typical suite (ELK) of frameworks for Big Data monitoring (Elasticsearch - for storage and indexing runtime data, Logstash - log processing raw data processing and ingestion to Elasticsearch, and Kibana - for visualization of collected data) and provides its services through a RESTful API.

DMon is a scalable, highly-available and extensible solution able to collect, store, index and deliver system level information (CPU load, memory consumption etc.), as well as data pulled from popular Big Data technologies: Apache YARN, HDFS, Spark, Storm, MongoDB and Cassandra. Distinctive features of the platform comparing to other similar solutions are (for a detailed comparison, see Section 6):

- (1) low footprint on monitored nodes,
- (2) easy management of monitored nodes (deployment of required agents, start/stop, reconfiguration) from a central UI,
- (3) control of platform components (Elasticsearch, Logstash and Kibana) using a REST API,
- (4) automatic generation of a selected set of visualizations for collected runtime data based on the roles (technologies installed) assigned to each node,
- (5) integrates runtime data from different Big Data frameworks with system level metrics.

---

<sup>1</sup> <http://www.dice-h2020.eu/deliverables/>

## Table of contents

<b>EXECUTIVE SUMMARY .....</b>	<b>3</b>
<b>TABLE OF CONTENTS .....</b>	<b>4</b>
<b>LIST OF FIGURES .....</b>	<b>5</b>
<b>LIST OF TABLES .....</b>	<b>5</b>
<b>1. INTRODUCTION.....</b>	<b>6</b>
<b>2. DICE MONITORING PLATFORM REQUIREMENTS. STATUS AT M24 .....</b>	<b>7</b>
<b>3. ACHIEVEMENTS OF THE 2ND YEAR.....</b>	<b>10</b>
<b>4. MONITORING PLATFORM IN THE DICE ECOSYSTEM.....</b>	<b>12</b>
4.1. Integration with Anomaly Detection Tool .....	12
4.2. Integration with Configuration Optimization Tool .....	13
4.3. Integration with Enhancement Tool.....	13
4.4. Integration with Trace Checking Tool .....	13
4.5. Integration with Delivery Tool .....	14
4.6. Integration with DICE IDE .....	14
<b>5. EVALUATION OF THE PLATFORM.....</b>	<b>15</b>
5.1. Logstash tuning .....	15
5.2. Methodology .....	16
5.3. Experiment setup .....	18
5.4. Results.....	19
5.5. Conclusion .....	22
<b>6. COMPARISON WITH SIMILAR SOLUTIONS .....</b>	<b>23</b>
<b>7. CONCLUSION AND FUTURE PLANS.....</b>	<b>25</b>
<b>8. SUPPLEMENTAL RESOURCES.....</b>	<b>26</b>
<b>REFERENCES.....</b>	<b>27</b>
<b>APPENDIX A. REST API DOCUMENTATION .....</b>	<b>29</b>
Overlord (Monitoring Management API) .....	29
Monitoring Core .....	29
Monitoring auxiliary .....	40
Observer.....	43

## List of figures

Figure 1 DICE Final Architecture.....	12
Figure 2 Comparing CMS and Parallel Collector .....	19
Figure 3 Logstash throughput in 3 experiments.....	20

## List of tables

Table 1 Requirements against DICE Monitoring Platform.....	8
Table 2 Requirements status .....	9
Table 3 Logstash Redis input configuration .....	17
Table 4 GC Flags for Analyzing .....	18
Table 5 JVM Flags for Parallel Collection .....	18
Table 6 GC Statistics for CMS and parallel Collectors .....	20
Table 7 GC Statistics with baseline configuration.....	21
Table 8 Tenuring threshold value used for next experiments .....	21
Table 9 Meta space and code cache size for next experiments.....	21
Table 10 GC Statistics Young-and Old Generation (ratio=1).....	21
Table 11 JVM Optimization Flags.....	21
Table 12 GC Statistics with optimization flags .....	21
Table 13 Comparison with similar solutions .....	24

## 1. Introduction

This deliverable details the final release of the DICE Monitoring Platform (DMon), whose main goal is to collect, store and deliver runtime monitoring data collected from the nodes of a distributed cluster running Big Data technologies, such as Apache Hadoop (YARN/HDFS), Spark, Storm, Cassandra and MongoDB, plus system level information (CPU load, memory consumption etc.) The Monitoring Platform was developed in task T4.1, part of WP4 work package, and its development ended at M24.

DMon is a scalable, highly-available and extensible solution that relies on a suite of open-source frameworks: Elasticsearch [9, 14] - for storage and indexing runtime data, Logstash [9, 16] – for log raw data processing and ingestion to Elasticsearch, and Kibana [9] - for visualization of collected data. The platform provides its services through a RESTful API. More details about the architecture of DICE Monitoring Platform are available in D4.1 deliverable [1] and in Iuhasz et al. [27]. Since the enhancements introduced in the second year did not impact the overall architecture of the system, this is not repeated here.

Main characteristics of DICE monitoring platform are:

- it is architected as a distributed system composed of micro-services, each controlling core components of the platform,
- it is easy deployable on Cloud environment, as well as on bare-metal machines, using the DICE Deployment tool [6],
- it offers access to its components (core services and node agents) through a RESTful API,
- it comes with Vagrant [28] scripts for development and testing phases,
- it integrates data from multiple Big Data platforms in a unique platform.

When comparing to similar solutions, e.g., Nagios, SequenceIQ, Apache Chukwa, Sematext, DataDog, DICE Monitoring Platform's strong points are:

- low footprint on monitored nodes, thanks to collectd plugins,
- easy management of monitored nodes (deployment of required agents, their startup/shutdown/reconfiguration) from a central UI,
- automatic discovery of Storm topologies,
- automatic generation of Kibana graphs for each monitored node, customised based on node's roles,

This deliverable is structured as follows. Section 2 reviews the requirements of the platform, being followed by Section 3 where we highlight the achievements of the second year. The Section 4 positions the platform in relation with other components comprising DICE toolchain and outlines the integration of different tools with DMon. The Section 5 details the results obtained during the evaluation of the platform under different stress loads. In Section 6 we contrast DICE Monitoring Platform with other similar solutions available today. Conclusions and future research areas are presented in Section 7, while supplemental resources are bundled for an easier reader's access in Section 8. The Appendix A dumps the REST API documentation as exported from Github's Wiki.

## 2. DICE Monitoring platform requirements. Status at M24

This section reviews the requirements defined for the monitoring platform. These were initially defined in the deliverable D1.2 and its annexes [2, 3]. Table 1 expands on how these were addressed in the final version of DICE Monitoring Platform.

Requirement	Implementation details
R4.1: Monitoring data warehousing	Monitoring agents running on each node of the cluster forward the data (exported by various Big Data frameworks or by custom applications) to the DMon platform that performs ETL (Extraction-Transformation-Loading). Supported technologies: Hadoop (HDFS/YARN), Spark, Storm, Cassandra, MongoDB.  DICE Monitoring Platform is built on top of ELK open-source framework, composed of Elasticsearch, Logstash and Kibana.
R4.2: Monitoring data warehouse schema	DMon platform transfers data in JSON format; no fixed schema is possible due to the diversity of monitored technologies, but common attributes are recorded for each data item stored in the platform, such as timestamp, node id, and source.
R4.2.1: Monitoring data versioning	Application build number is handled via tags attached on collected data.
R4.2.2: Supplying the version number	Deployment ID returned by the DICE Deployment Service is set as tag in DMon platform by the Delivery tool.
R4.3: Monitoring data extractions	Logstash server performs ETL on incoming data before sending data to Elasticsearch for storage and indexing.  We have implemented a data frame for temporary storage and aggregation engine that helps clients retrieving aggregated data.
R4.4: Monitoring data format transformations	Data is stored internally as JSON messages and it's transformed to requested format upon delivery. Supported formats are JSON/CSV/TEXT/XML (using OSLC Perf 2.0 vocabulary).
R4.5: Monitoring data retention policy	This requirement has been deprecated and not addressed in the final release of the platform.
R4.6: Monitoring data access restrictions	This requirement has been deprecated and not addressed in the final release of the platform.
R4.7: Monitoring tools REST API	REST API interface implemented as a microservice in Python offers access to management and query of DMon platform.
R4.8: Monitoring Visualization	Visualization uses Kibana framework.  Diagrams are automatically generated for each monitored node based on its roles.
R4.9: Data Warehouse replication	Elasticsearch, which powers the data warehouse, is natively highly available and supports replication. Replication is supported by backup/restore of Elasticsearch indexes.

R4.22: Time-based ordering of monitoring data entries	This is a domain assumption: data is consistently ordered when collected from different nodes; nodes internal wallclock is correctly set.
R4.34: Monitoring for quality tests	Using the REST API, a number of tools dealing with quality testing and enhancement of DIA (Filling the Gap tool, Anomaly Detection tool, Configuration Optimization tool, Trace checking tool) queries DMon platform for metrics of interest (such as, arrival rate or throughput).
R4.35: Tag monitoring data with OSLC tags	DMon exports metrics in RDF format using OSLC Perf Mon 2.0 vocabulary.
R4.38 Monitoring tool integration with DICE IDE	DICE Monitoring Platform is integrated in the DICE IDE as an external service, by opening the DICE Monitoring Platform services (management and visualization) in a web view within DICE IDE.
R4.39 Discover Storm topologies	DICE Monitoring Platform automatically searches the cluster for valid Storm deployment by scanning for nodes exposing Storm end-points and connects to the first found topology.
R4.40 Collect and index raw data from Storm worker nodes log files	Log files from Apache Storm worker nodes are pulled and indexed into the platform in an asynchronous way.
R4.41 Collect and index application-specific data coming from Posidonia Operations applications	Monitoring platform is able to parse log files outputted by PRO's Posidonia Operations application; the data is stored, indexed and queried as any other piece of data collected from Big Data frameworks.

**Table 1 Requirements against DICE Monitoring Platform**

Table 2 presents the status of requirements' implementation at the end of reporting period (M24).

<b>Requirement</b>	<b>Status M12</b>	<b>Status M24</b>
R4.1: Monitoring data warehousing	IN PROGRESS	FULFILLED
R4.2: Monitoring data warehouse schema	FULFILLED	FULFILLED
R4.2.1: Monitoring data versioning	NOT STARTED	FULFILLED
R4.2.2: Supplying the version number	NOT STARTED	FULFILLED
R4.3: Monitoring data extractions	IN PROGRESS	FULFILLED
R4.4: Monitoring data format transformations	IN PROGRESS	FULFILLED
R4.5: Monitoring data retention policy	NOT STARTED	DEPRECATED (see Note 2)
R4.6: Monitoring data access restrictions	NOT STARTED	DEPRECATED (see Note 2)
R4.7: Monitoring tools REST API	IN PROGRESS	FULFILLED
R4.8: Monitoring Visualization	IN PROGRESS	FULFILLED
R4.9: Data Warehouse replication	NOT STARTED	FULFILLED
R4.22: Time-based ordering of monitoring data entries	FULFILLED	FULFILLED
R4.34: Monitoring for quality tests	IN PROGRESS	FULFILLED



R4.35: Tag monitoring data with OSLC tags	NOT STARTED	FULFILLED
R4.38 Monitoring tool integration with DICE IDE		FULFILLED
R4.39 Discover Storm topologies		FULFILLED
R4.40 Collect and index raw data from Storm worker nodes log files		FULFILLED
R4.41 Collect and index application-specific data coming from Posidonia Operations applications		FULFILLED

**Table 2 Requirements status**

Note 1: The requirements R4.38 – R4.41 have been added during second year of the project.

Note 2: The requirements R4.5 and R4.6 were not implemented in the final version of DICE Monitoring Platform because, in case of R4.5 the retention policy was not addressed anymore by other tools of the DICE toolchain, while R4.6 was deprecated since collected monitoring data was not considered sensitive by data owners.

Note 3: The gaps in requirements numbering are due to the fact that we initially collected together all the requirements for all WP4 tools. For example, the requirements R4.10 – R4.21 are related to Enhancement tool and are addressed in [7]. For a complete list of WP4 requirements, the reader can refer to D1.2 Companion document [3].

### 3. Achievements of the 2nd year

Throughout the first year we focused on the development of the platform, its deployment on Flexiant Cloud Orchestrator (FCO) and its validation against Apache Hadoop (YARN/HDFS).

The second year of the project, that is the period under report, was devoted to consolidate and extend the DICE Monitoring Platform towards a mature, stable solution. We added support for additional data intensive platforms (Apache Storm, Apache Cassandra and MongoDB), as well as features to automatically discover the Storm topologies, and pulling data from the YARN History server.

In order to enhance the availability and stability of the platform, we have implemented a daemon-based core service control schema (*systemd* and *upstart*) that ensures automatic startup of the platform and agents at bootup time.

We have implemented a data frame (based on Pandas Python [29] module) for temporary storage and aggregation engine to DMon. This helps with the querying, exporting and aggregation of data.

A unique feature of the platform, when compared to similar solutions, is the automatic generation of Kibana graphs for a selected number of system and technology-specific metrics. The visualization generation is based on the roles assigned for each monitored node. The visualizations are then saved inside a specialized index (called *.kibana*) inside the Elasticsearch core service. For more details please consult Annex 1.

We also have added functionality related to integration with the DICE toolchain by including application tagging capabilities, as well as polling period configuration. In collaboration with WP5, we have implemented the Chef cookbooks [30], which enable automatic configuration of nodes and supported services to push their runtime metrics to the DMon. Moreover, we have created an initial version of the TOSCA blueprints, which enable administrators to easily deploy the DMon services themselves.

DICE Monitoring Platform has already been integrated in ATC and PRO demonstrators, while NETF integration is being roll out. To better answer demonstrators needs, DMon is able to collect and index application-specific data coming from PRO's CEP module.

The list below outlines the additional features added in the 2<sup>nd</sup> year. Each of these features addresses specific topic of high-level requirements presented in Table 1:

- Add support for additional enhanced aggregations
- Use *systemd* and *upstart* for core component control
- Support for DICE NoSQL databases (Apache Cassandra, MongoDB)
- Implement start/stop scripts for core components
- Logstash reload config instead of stop/start
- Integrate Cloudera Manager REST API into dmon
- Integrate REST API request for Spark
- Integrate REST API request for YARN
- Create Kibana dashboard

Below is the list of enhancements of already existing functionality implemented in the second year of the project:

- Add capacity to dynamically assign Logstash servers to nodes
- Add new Elasticsearch core query engine
- Enable index definition while querying Elasticsearch
- Custom Elasticsearch index
- Add the possibility to rename collected metrics
- Update stop scripts for all micro-services
- Implement dmon-agent status

- Implement dmon-agent deployment
- Implement dmon-logstash
- Create dmon-aux flask agent
- Set logging period in metrics properties file
- Set logging granularity with env variable
- Add parameter content type to Swagger UI
- Local vs. distributed enhancement
- Create dmon-stop.sh that reads core component pids
- Create PID files for core components
- Add new column for nJMXState

## 4. Monitoring platform in the DICE ecosystem

The Monitoring Platform is central for all DICE tool-chain tools that need data from the runtime. This section describes the position of the DICE Monitoring Platform in the wider landscape of DICE ecosystem and will detail the interaction with other tools. For more elaborated details, the reader is referred to deliverable D1.4 [4] that describes the DICE framework in its entirety.

Figure 1 illustrates the position of the Monitoring Platform in the overall DICE framework. Different tools, such as Anomaly Detection or Enhancement, query the Monitoring platform in order to extract runtime data they need. On the other hand, the Delivery tool, via Repository & CI, deploys the Monitoring agents on cluster's nodes.

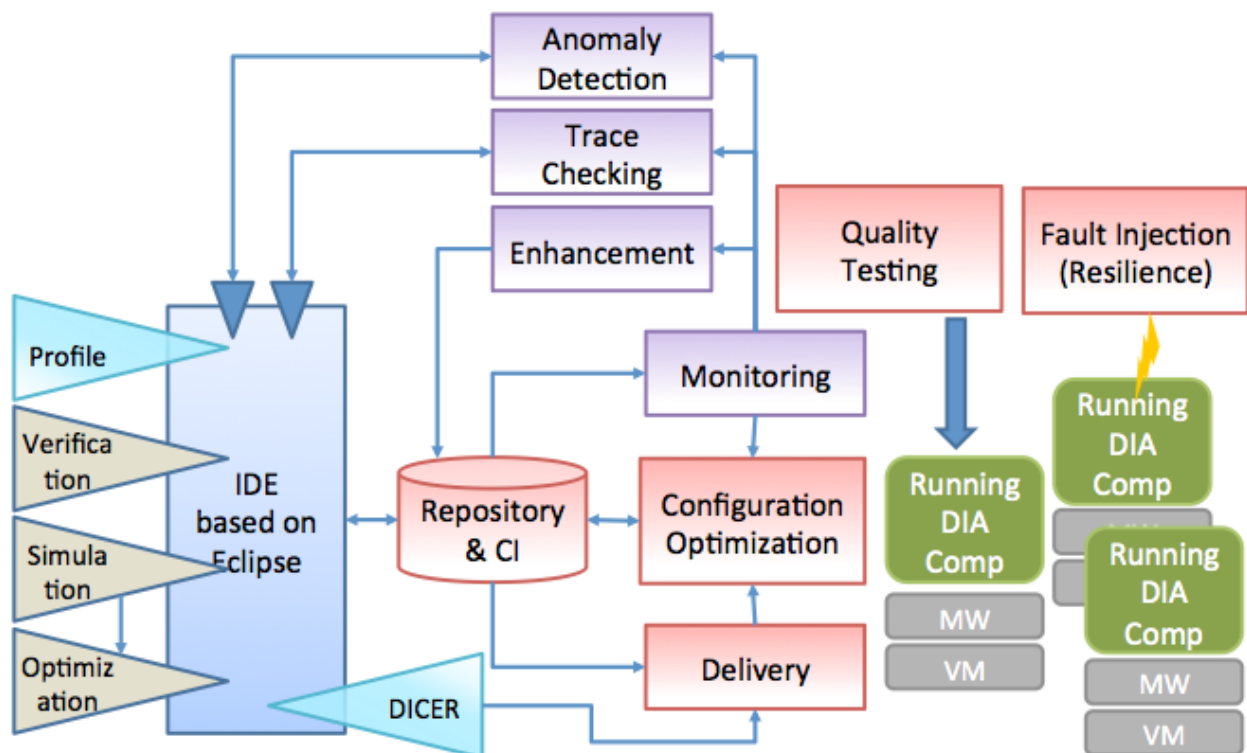


Figure 1 DICE Final Architecture

It is worth noting that DICE Monitoring Platform can be deployed and used as a **stand-alone system** by end-users. They can use the stand-alone tutorial<sup>2</sup> to learn how to install/deploy the platform in Cloud-based environments and how to use the RESTful API to control the platform. The RESTful API is exposed through a friendly Swagger User Interface.

### 4.1. Integration with Anomaly Detection Tool

The Anomaly Detection (AD) tool [5] queries the Monitoring platform for performance metrics via the DMon RESTful API, and checks for anomalies in retrieved monitoring data. The anomaly detection tool is able to collect raw as well as aggregated metrics from the platform. By offloading some pre-processing tasks to DMon, AD is able to execute data processing tasks on a method by method basis (normalization, dimensionality reduction etc.).

The integration between AD and Monitoring platform is bidirectional: the detected anomalies are pushed/stored back into a special index of DMon's Elasticsearch instance. Thus AD's clients will use the DMon's query REST calls (POST /v1/observer/query/{csv/json/plain}) to retrieve anomalies. DMon is also used to store the predictive models created by the supervised anomaly detection methods.

<sup>2</sup> <https://github.com/dice-project/DICE-Monitoring/wiki>

## 4.2. Integration with Configuration Optimization Tool

The configuration optimization (CO) tool [6] collects performance related metrics via the query REST resources from DMon. This integration has been tested for Apache Storm topologies. Special REST resources for resource polling periods are also used by CO.

## 4.3. Integration with Enhancement Tool

The Filling-the-Gap module (FG), sub-component of Enhancement tool [7], collects performance related metrics (resource utilization and throughput). Concretely, FG pulls monitoring data from DMon platform that comes from the YARN History server (e.g., number of jobs, execution time) as well as system metrics (e.g., CPU utilization).

## 4.4. Integration with Trace Checking Tool

Trace Checking (TC) tool [5] analyse events generated in an Apache Storm topology. It requires monitoring data that is not provided by the Apache Storm Metrics API, and thus not collected in the DMon platform. Therefore, the platform need to collect the raw log files from all Apache Storm worker nodes. This is implemented so that it allows fetching and serving of the required logs in an asynchronous mode. For example, fetching collected raw log files is possible by invoking POST `/v1/observer/query/json` with the following input:

```
{
  "fname": "output",
  "ordering": "desc",
  "queryString": "DComp:\"DMON\" AND cType:\"TCT\"",
  "size": 500,
  "tstart": "now-1d",
  "tstop": "None"
}
```

where fname is the JSON output file name, ordering field will order to events from newer to older, queryString is the query to be run on Elasticsearch, size specifies the max number of items to return and tstart-tstop pair defines the time window of interest. The TC is also able to fetch raw log files for Storm workers directly from DMon. This is done using:

```
POST /v1/overlord/storm/logs
```

The request starts an asynchronous process that fetches all active worker logs from all registered nodes that have the Storm role assigned to them. The current status of the process can be accessed at:

```
GET /v1/overlord/storm/logs/active
```

All collected logs are stored as *tar* containers. They contain additional *tar* containers for all worker logs from each node. It is important to note that there can be more then one worker per registered Storm node. The full list of containers can be accessed at:

```
GET /v1/overlord/storm/logs
```

The response lists all available logs:

```
{
  "StormLogs": [
    "workerlogs_2016-12-07-15:05:10.tar",
  ]
}
```

```

"workerlogs_2016-12-07-15:25:12.tar",
"workerlogs_2016-12-07-16:00:09.tar",
"workerlogs_2016-12-07-16:04:00.tar",
"workerlogs_2016-12-07-16:08:01.tar",
"workerlogs_2016-12-07-16:50:04.tar",
"workerlogs_2016-12-08-16:22:18.tar",
"workerlogs_2016-12-08-16:28:51.tar",
"workerlogs_2016-12-08-17:29:27.tar",
"workerlogs_2016-12-08-17:30:48.tar",
"workerlogs_2016-12-08-17:32:53.tar",
"workerlogs_2016-12-08-17:35:11.tar"
]
}

```

In order to download the Storm worker log container we have to issue the following request:

```
GET /v1/overlord/storm/logs/{workerlogname}
```

#### 4.5. Integration with Delivery Tool

The collection of the metrics must be automatic from the moment the DIA gets deployed and started. Thanks to a close collaboration with the WP5, in year 2 the users of the DICE Deployment Service [6], which is built on top of Cloudify and uses Chef cookbooks [30], can expect that their service and application nodes will connect automatically with the DMon. This means that the users do not have to do anything other than mark the appropriate nodes in the DDSM / TOSCA blueprint as monitorable.

The Delivery tool [6] is responsible for DIA deployment and configuration as well as the installation and initial setup of the dmon-agent component on all deployed nodes. All addresses and ports are standardized and documented and can be setup by the administrator in the Deployment Service. Application versioning is done via tags assigned to the DMon by the Delivery tool. By convention, the tag used in this process is the same as the deployment ID returned by the DICE Deployment Service. At the end of Y2, automatic monitoring support is available for Apache Storm, Apache Spark and Cassandra. More details are provided in D5.2 deliverable [6].

#### 4.6. Integration with DICE IDE

DICE Monitoring Platform is integrated in the DICE IDE [8] as an external service, by opening a web view within DICE IDE (an Eclipse-based standalone application). The DICE Monitoring Platform plug-in provides end-users with access to the platform's controller service REST API (the administration interface) and to the visualization engine. The default end-points for the two DICE Monitoring Platform services can be configured from DICE IDE's Preferences window. The DICE Monitoring Service Administration interface and DICE Monitoring Service Visualization UI are available as menu items in DICE Tools menu. More details about this integration are provided in D1.5 deliverable [8].

## 5. Evaluation of the platform

This section details the experiments aimed at evaluating the performance of the platform under different load stresses. The Logstash server is the component facing the incoming traffic in DMon platform [10, 11] and it is responsible for pre-processing the incoming data and further ingest it in the Elasticsearch. While for Elasticsearch, there is a number of publications on improving performance, including tips, guides and even profiling the Java virtual machine [12-14], to the best of our knowledge, the literature about tuning Logstash for high throughput in terms of optimizing Garbage Collection (GC) has not been reported yet. There are some guides to improve the Logstash performance by tuning the filter configuration [15], or by increasing the number of workers and buffer size [16], but the GC optimizations were not considered. Hence, the focus of our research described here is to optimize a single instance of Logstash server by tweaking various GC options and flags.

### 5.1. Logstash tuning

In the case of DMon, most of the filter configurations used in Logstash are based on regular expressions of grok filter, so the instances demand high CPU cycles. As a result, there is a requirement to reduce other impact on CPU utilization. To date, the researchers have conducted experiments on JVM and concluded that the major factor of reduction in throughput of Java applications is the GC [17], [18]. Therefore, we have looked deeply into Logstash JVM to minimize GC impact on the application.

Tuning the GC for Java applications is a tedious task and various techniques have been proposed in the last decades. We are interested to find a suitable configuration for GC in order to increase the throughput of the Logstash instances. The approach that we considered is to collect and analyze the GC activities logs. The concept and the experiments results are exposed in the following paragraphs.

Logstash is an open source data collection engine with near real-time pipelining capabilities. The Logstash pipeline consists of 3 main components. Firstly, we have the input which enables the collection of logs in a large variety of formats such as: files, TCP/UDP, Graphite, Lumberjack etc. Secondly, we have the filter plugins which enable Logstash to execute transformation on the input data. Lastly, the output plugin allows the processed and transformed data to be written in a large variety of formats ranging from JSON to Elasticsearch.

*Logstash processing pipeline:* From Logstash version 2.x onwards, output and filter are processed in the same threads. The execution model as follows:

- **Input threads:** Each `input{}` statement in Logstash configuration file is considered as its own thread. The input threads write events in a queue (default size is 20 events). This queue will transfer events to worker threads, blocking if worker threads are busy.
- **Worker threads:** Each worker thread takes a batch of events off the queue, placing events in a buffer (default size is 125 events). It then runs the batch of events based on the configuration file and it writes through the outputs. The number of workers and buffer size per each worker can be configured.
- **Scaling Logstash:** Logstash can be scaled from standalone instance to clusters [19]. The basic architecture is a Logstash instance connected directly to an Elasticsearch instance. However, there can be any number of distinct Logstash instances serving the same Elasticsearch instance. In fact each Logstash instance is configurable so that it adheres to one archetype only, meaning that it can be only used for input, filtering or output [12, 16]. Data loss prevention plays an important role in any monitoring deployment. When the incoming data to the pipeline is higher than its consumption ability, it will lead to data loss. Therefore, a message broker can be used in front of Logstash instances. The message broker is able to hold the events while Logstash is feeding data into Elasticsearch [9]. Currently, there is a wide array of technologies which can fulfill the task of a message broker: Redis [20], ZeroMQ [31], RabbitMQ [32], Kafka [21] etc.



Arguably the most interesting features of Java is the automatic memory management. This means developers do not need to handle the memory used by objects. In the HotSpot JVM that function is performed by a Garbage Collector. The Garbage Collector is responsible for allocating memory for objects, reclaiming memory of objects no longer in use and keeping referenced objects in memory. GC is the process of finding and freeing objects no longer reachable [22].

There are four types GC in JVM [22–24]: Serial Collector, Parallel Collector, Concurrent Mark Sweep Collector and Garbage First Collector (G1).

*The Serial Collector* performs minor and major collections in serial way with a single thread. It is recommended that the serial collector is used in programs with a small heap size (below 100M).

*The Throughput Collector* uses multiple threads for minor and major collections to take advantage of multi-CPU's on machines. Therefore, it is known as *Parallel Collector*. However, minor and major collection using Throughput Collector still causes a stop-the-world scenario. The old generation is compacted during major collection. This collector outperforms the serial one.

*The Concurrent Mark and Sweep Collector (CMS)* aims at low pausing collections. The CMS Collector collects Young Generation (YG) in the same way to the Throughput Collector. However, this collector does not stop application threads to perform major collections. It uses background threads to find and free unused objects in old generation. Besides, the CMS Collector does not compact the old generation, so it could make the heap to be fragmented.

*The G1 Collector* [24] is the newest collector. It aims at low pauses for application with a large heap (greater than 4G of RAM).

A Logstash instance comes with a CMS Collector as the default collector. The CMS Collector reduces pause time in processing old generation but needs extra CPU cycles. A Throughput Collector brings a high throughput for application but takes a long pause on account of Full GC. In theory, for applications demanding high CPU utilization, the Throughput Collector should be used which yields better performance, and the use of CMS collector is adequate for applications requiring low pauses or short response times, at the expense of the application's throughput [23]. However, the impact of CMS Collector on application throughput is unknown; thus, the focus of our experiments is to compare the CMS and Parallel collectors effect on Logstash.

## 5.2. Methodology

We are describing here the method to tune Logstash for high throughput. According to [17], the overall execution time of an application (T) can be described by expression:

$$T = T_{app} + T_{gc} + T_{jit} + T_{sync}$$

where  $T_{app}$  is the time for application itself;  $T_{gc}$  is the time spent on GC,  $T_{jit}$  is the time for compilation and finally,  $T_{sync}$  is for synchronization time.

For high throughput applications,  $T_{gc}$ ,  $T_{jit}$  and  $T_{sync}$  have to be minimized in order to increase  $T_{app}$ . The GC time heavily depends on the Heap size, Heap layout and GC algorithms. Consequently, if we consider GC time as a function, the function can be represented as follow:

$$T_{gc} \Leftarrow F(S, L, A)$$

where S, L, A are parameters representing the Heap Size, Heap Layout and Garbage Collection Algorithms, respectively. Each parameter has a big impact on GC time. Therefore, we investigate on GC characteristic of Logstash with different input parameter values in order to optimize  $T_{app}$ . In our experiment, we considered different heap sizes from 1G to 4G and two Garbage Collectors: CMS collector



and Parallel Collector. For the heap layouts (ratio of Young Generation to Old Generation), we chose the ratio with value of one and two.

To reduce Tjit and Tsyn, we consider the flags to optimize JVM such as `-XX:+AggressiveOpts` and `-XX:+UseFastAccessorMethods`. These flags have a positive effect on the most of Java applications, but not all. Therefore, experiment is needed to evaluate the sensitivity of the overall execution time with respect to these optimization flags. The Logstash throughput in each experiment is compared by using EWMA algorithm [25].

We conducted experiments with a Logstash instance under heavy load condition. The Logstash instance uses the filter configurations generated by DMon. A Redis [20] server is used as a buffer in front of the Logstash filter instance. There is no output plugin to write data out to Elasticsearch in the instance because we want to improve Logstash performance with filter plugin. Future work will focus on a more holistic approach in which the Logstash instance will feed the processed data into Elasticsearch.

In DMon, monitored nodes send metrics to Logstash shippers. The shippers feed data to the Redis server via output plugins. Therefore, to simulate the shippers, a simulation tool is developed to mimic Logstash shipper behaviors. The dummy data are the metrics collected from monitored nodes. Each dummy payload will mimic the metric format received by Dmon during normal operation. After constructing into *json* format, the dummy data are transferred to the buffer repeatedly. Each shipper is developed as a thread, in which, a Redis connection is used to communicate to the Redis buffer.

For input plugin of the Logstash instance, we use list as data type in Redis and the key is specified as *logstash*. The number of input threads is set equally to the number of CPU cores to get the best performance [15]. An example of input configuration is included in Table 3.

```
redis {
  host => "ip redis host"
  port => "redis port" #default : 6379 data type => "list"
  key => "logstash"
  codec => " json "
  threads => 4
}
```

**Table 3 Logstash Redis input configuration**

To avoid starvation of the filter instance due to insufficient incoming data, we set length of the list in Redis to a threshold. When the length value reaches the threshold, the simulation tool will stop sending data. If the length value is below the threshold, the simulation will trigger the sending method.

### Measured metrics

When analyzing GC logs, we measure GC throughput, minor GC interval, and average promotion per collection. With the Logstash filter, we use *metric filter plugin* to measure its throughput. This plugin is light and has low impact on system.

*Logstash metrics:* It is possible to measure total count of events, rate of events in 1-minute, 5-minute and 15-minute by using metric filter plugin. The meter in metric plugin uses EWMA. We use this metric to compare Logstash throughput in each experiment.

*GC performance metrics* are the following:

- Throughput - % of total time spent in application running time.
- GC Overhead - % of total time spent in GC
- Collection frequency - how often Collection happens.
- GC pause - Application stop time to perform each collection
- Promotion rate - the amount of data promoted from young to old generation
- Young Generation to Old Generation per time unit.

### 5.3. Experiment setup

Our experimental topology was comprised of two VMs and one physical machine. The first VM was hosting the data generation tool, the second one was hosting the Redis Server, while Logstash instance has been installed on the physical machine.

As stated in the previous section, the filter configuration was generated by the DMon platform and it was statically loaded into the Logstash instance. Swappiness (by modifying *vm.swappiness*) was set to zero.

With the metric filter configuration, *flush interval* value is configured as 60s. To ensure that we collect accurate values, the value of *clear interval* is configured to 1800s (30 minutes)<sup>3</sup>. The JVM flags shown in Table 4 were used in order to obtain GC logs. The flags in Table 4 and Table 5 are used to enable Parallel Collector and CMS Collector.

We ran 1 hour long warm-up experiments in order to ensure a steady code cache. Then, we performed Full GC using the *jcmd* tool. We collected the GC activities logs for the next 30 minutes of experiment. Then, we analyzed the GC activities to decide which JVM flags should be changed and then reran the experiments with the new flag values. After some experiments we chose the most suitable JVM flags for the Logstash instance. To report Logstash throughput, we choose 5-minute rate because the rate will converge to steady value in 30 minutes and the fluctuation around the steady value is more stable than at the 1-minute rate.

Flags	Description
-XX:+PrintGCDetails	Print details at garbage collection
-XX:+PrintGCTimeStamps	Print timestamps at garbage collection
-XX:+HeapDumpOnOutOfMemoryError	Dump heap to file when java.lang.OutOfMemoryError is thrown
-XX:+PrintCodeCache	Print detailed info on the compiled code cache when exiting
-XX:PrintFlagsFinal	Prints list of all available java parameters

**Table 4 GC Flags for Analyzing**

Flags	Description
-XX:+UseParallelGC	Enable the use of parallel garbage collector.
-XX:+UseParallelOldGC	Enable parallel garbage collector. This flag is enabled automatically when enabling -XX:+UseParallelGC
-XX:-UseAdaptiveSizePolicy	Disables the use of adaptive generation sizing. This is enabled by default

**Table 5 JVM Flags for Parallel Collection**

<sup>3</sup> The flush interval is how often the metric events are flushed from Logstash input through the metrics filter, while the clear interval defines how often the metrics from the filter metrics plugin are created.

## 5.4. Results

### Comparing CMS and Parallel collectors

First, we compare the two collectors applied for Logstash (with different heap sizes 1G, 2G and 4G) by Logstash throughput. As Figure 2 and Table 6 show, for Parallel collector, the bigger heap size yields better throughput of application. In case of CMS, the throughput of Logstash slightly decreases with the increase in heap size and, under the same heap size, it is always lower than the one with Parallel collector. Consequently, we can conclude that CMS collector is unsuitable for Logstash in the heavy load condition. It might only be an acceptable choice under light loads.

In the next experiment, we set 2G of RAM for Logstash heap size because we choose this configuration to develop application in the project.

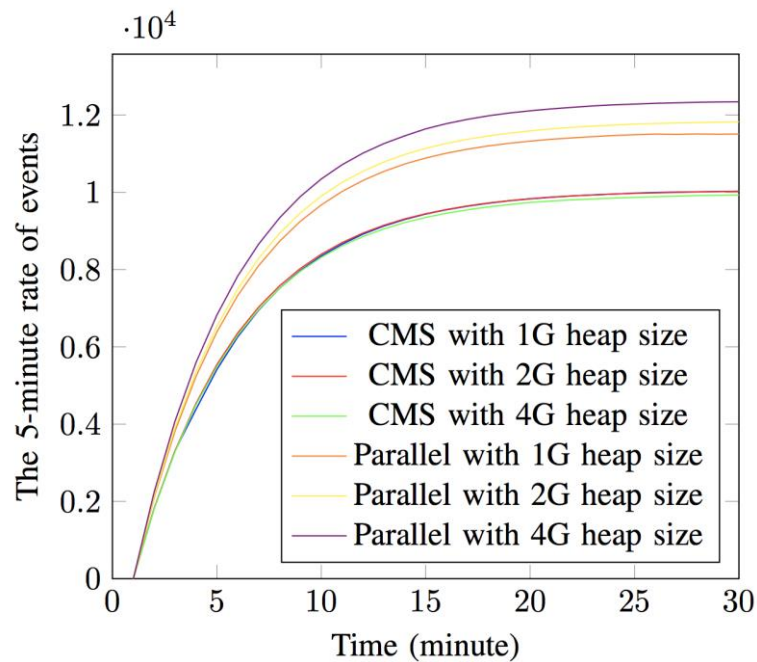


Figure 2 Comparing CMS and Parallel Collector

### Tuning GC for DMon platform

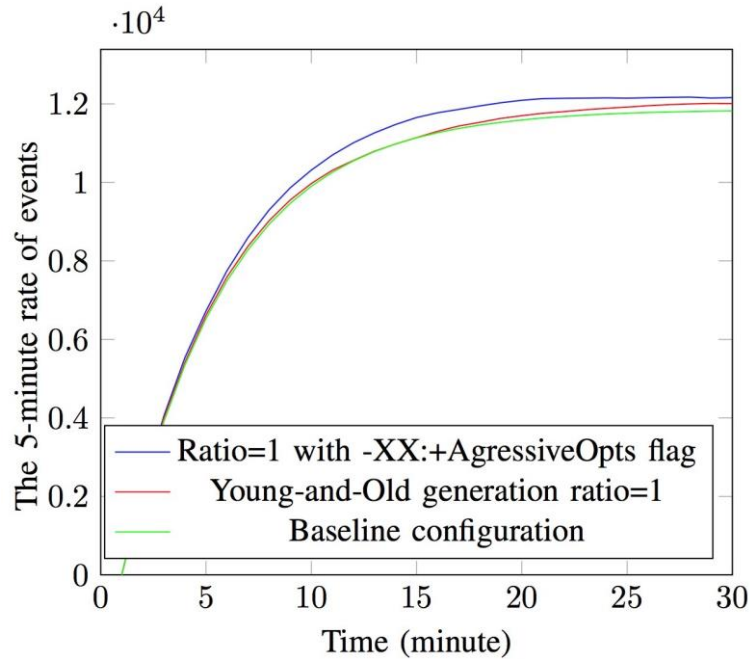
*Analyzing result of the baseline configuration:* We set Logstash to use 2G RAM and use Parallel Collector with JVM flags as Table 5. From the result in Table 7, the application running time takes up 98.89% of total time. If the application running time increases more, the application throughput will also raise. To reduce GC overhead, it is required to decrease GC frequency and GC pause time. Increasing YG can reduce GC frequency but it may lead to degraded application throughput and accounts for longer GC pauses. Because of longer duration in minor GC, more data could be promoted and copied to survivor spaces in each collection, thus having a negative impact on application throughput. On the other hand, the results in Table 7 show that the average promotion is small, just 1.3K per collection, which means that the most of new objects are discarded at each collection. Therefore, we decided to increase YG size by using flag *NewRatio=1*.

Furthermore, the default thread stack size on Linux 64 bit is 2048k, we decide to decrease this value to 512K in order to save memory. With Parallel Collector, the default values of initial tenuring threshold and maximum tenuring threshold are 7 and 15, respectively. To reduce promotion rate to Old Generation, the values for both flags are set to maximum value 15 (see Table 8). By analyzing compilation and metadata logs during the experiment, the metaspace takes up 42MB with a maximum code cache size of 22MB.

Therefore, we decide to initiate these values as shown in Table 9. With these flags in Table 9, the application start-up time is reduced.

HeapSize	1 GB		2GB		4GB	
GC	Throughput	Avg. Pause	Throughput	Avg. Pause	Throughput	Avg. Pause
CMS	97.93%	0.009s	96.95%	0.0118s	95.45%	0.02s
Parallel	97.97%	0.00742s	98.89%	0.00743s	99.6%	0.00723s

**Table 6 GC Statistics for CMS and parallel Collectors**



**Figure 3 Logstash throughput in 3 experiments**

*Analyzing result with young-and-old generation ratio=1:* With the *Young-and-Old ratio = 1* from Table 10, the GC throughput increases to 99.27% compared to 98.89% with the baseline, because the GC frequency is reduced (GC pause interval is higher) and the GC duration is nearly the same. Likewise, the Figure 3 shows that the throughput after changing the heap layout is better, from 11,800 events/5-minute for base line configuration to 12,000 events/5-minute for ratio = 1 configuration.

*Analyzing result with JVM optimization flags AggressiveOpts:* There are some JVM optimization flags that may improve application performance. We conduct an experiment with the flags as shown in Table 11. When the *AggressiveOpts* flag is enabled, there are some flags coming along with: *AutoBoxCacheMax*, *DoEscapeAnalysis*, *UseBiasedLocking*, *EliminateLocks*, *OptimizeStringConcat* and *AutoFill* [23] [26]. The *AutoFill* flag enables some better loop optimization by the compiler. This feature is disabled by default.

GC throughput	98.89%
Full GC	none
Average Promotion	1034B per collection
Average GC pause	0.00743s
Min/Max GC pause	0.00557/0/02447s

Average GC pause interval	0.72834s
---------------------------	----------

**Table 7 GC Statistics with baseline configuration**

Flags	Default value	Used value
-XX:InitialTenuringThreshold	7	15
-XX:MaxTenuringThreshold	15	15

**Table 8 Tenuring threshold value used for next experiments**

Flags	Default value	Used value
-XX:MetaspaceSize	Depends on platform	64m
-XX:MaxMetaspaceSize	unlimited	64m
-XX:ReservedCodeCacheSize	240m	32m
-XX:InitialCodeCacheSize	550k	32m

**Table 9 Meta space and code cache size for next experiments**

GC throughput	99.27%
Full GC	none
Average Promotion	1307B per collection
Average GC pause	0.00785s
Min/Max GC pause	0.00597s/0.01312s
Average GC pause interval	1.08494s

**Table 10 GC Statistics Young-and Old Generation (ratio=1)**

Flags	Description
-XX:+AggressiveOpts	Enable point performance compiler optimization
-XX:+UseFastAccessorMethods	Enable optimization for Get < <i>Primitive</i> > Field

**Table 11 JVM Optimization Flags**

GC throughput	99.2%
Full GC	none
Average Promotion	1,407B
Average GC pause	0.0080s
Min/Max GC pause	0.00653s/0.0238s
Average GC pause interval	1.084s

**Table 12 GC Statistics with optimization flags**

The *AutoBoxCacheMax* flag is set to 20,000; thus, the performance of certain applications is improved. The *BiasedLocking* flag is an optimization technique that biases an object to the thread last acquiring the lock. This flag improves uncontended synchronization performance. It is on by default in Java SE 6 or later.

The *EscapeAnalysis* flag is an optimization technique that evaluates the scope of objects. By default, this feature is off. The *OptimizeStringConcat* flag will optimize the use of `StringBuilder` objects. The *EliminateLocks* flag is on by default. It eliminates the unlock and relock in unobserved operations. This flag reduces synchronization time.

Finally, the value of *BiasedLockingStartupDelay* flag is set to 500 (default value is 2,000). This means that the biased locking will trigger sooner. As we show in Table 12, although the GC throughput value is slightly lower than the one with Young-and-Old ratio=1, the optimization flags have improved Logstash throughput, as shown in Figure 3, because of optimization techniques applied on JVM. The rate reaches to 12,150 events/5-minute interval, compared to 12,000 and 11,800 with ratio=1 and baseline configuration, respectively.

## 5.5. Conclusion

The platform evaluation shows that mostly created objects in Logstash are short-lived. Therefore, increasing YG size yields better performance. Even better, applying JVM optimization flags also brings higher throughput for Logstash.

## 6. Comparison with similar solutions

In this section, some of the most popular open-source and commercial solutions for infrastructure monitoring and Big Data frameworks, in particular, are contrasted to DMon on several dimensions. These findings were initially published in [27]. Herein, we provide some recent updates and additional information emerging from DICE project context.

In the context of monitoring tools, scalability is crucial as Big Data deployments may include thousands of nodes. Although the selected technologies (ELK stack) easily support horizontal scalability, sometimes the amount of generated monitoring data may exceed Logstash's processing capacity. In order to cope with this issue, a message queue should be employed 'in front' of Logstash server(s). See Section 6 'Evaluation of the platform' for experimental evaluation of the Logstash server.

In terms of deployment and installation approaches, platforms may be either installed manually or automatically deployed using specialized software infrastructure, namely content management systems. The reviewed platforms all require manual installation, whereas DMon provides scripts for node provision and configuration. These may be included in orchestration frameworks. The node components are transparently installed upon node addition by the DMon controller service, thus requiring no specialised skills nor personnel.

Extensibility of the platform, i.e. easy integration of new frameworks, was central to our design. The platform provides a uniform interface to a number of Big Data frameworks. Including support for a new Big Data frameworks requires proper configuration of nodes' roles and adaptation of Logstash parsers. In this way, not only Big Data frameworks can ingest data to our platform, but we can also collect log data produced by any custom data intensive application. Working closely with PRODEVELOP partner in DICE, we managed to integrate Posidonia specific log file into DMon platform. Concretely, the log files created by CEP (Complex Event Processing) module of Posidonia Operations application are parsed and data indexed in DMon, from where they are available to all tools comprising DICE solution.

In most of reviewed platforms, analytics against collected monitoring data is handled via user defined alerts. Although these provide valuable data for Ops teams, they do not provide the level of insight required by Dev teams for optimization and validation purposes. More sophisticated, contextualized methods and tools are required. The DICE Anomaly Detection tool [7] is able to detect such anomalies and with the help of the DICE Enhancement tools [6] will feedback this information into design-time models. Moreover, detected anomalies are stored back in DMon platform, thus being available for easy querying and further processing (e.g., for statistical purposes).

Table 13 summarizes the discussion and provides an overall view of this comparison.

	<b>Nagios</b>	<b>Ganglia</b>	<b>SequenceIQ</b>	<b>Apache Chukwa</b>	<b>Semantext</b>	<b>DataDog</b>	<b>D-Mon</b>
Scalability	Manual	Manual	-	Manual	Yes	-	Manual
Deployment model	VM	On-premise	As a service	On-premise	As a Service / On-premise	As a service	As a service / On-premise
Installation	-	Manual / via CMS	-	Manual / via CMS	-	-	REST API
Big Data frameworks support	Poor	Poor	Hadoop 2.x	Hadoop 2.x	Good and extensible	Good (but, no Spark / Storm)	Good and extensible
Visualization	User-defined	Pre-defined	Pre-defined	Pre-defined	User-defined	User-defined	User-defined (automatically generated)
Analytics	Alerts	-	ML support	Anomaly detection	Alerts	Alerts, correlations	ML support
Real-time data support	Yes	Yes	Yes	No	Yes	Yes	Yes
Licensing	Freemium	BSD	Commercial	Apache 2	Freemium	Freemium	Apache 2

Table 13 Comparison with similar solutions



## 7. Conclusion and future plans

DICE Monitoring Platform is an effort of 2-year development aimed at bringing monitoring process in the life of software developers/engineers. The platform is a distributed, highly available system for monitoring Big Data technologies, as well as system metrics. Aligning DMon objectives to DICE visions, that is bringing together Model-Driven Development and DevOps to enable fast development of high-quality data intensive applications, DMon features automation at numerous levels: deployment of software components the nodes of monitored cluster, easy management of the monitoring platform itself, or automatic creation of visualizations based on collected data. Thanks to close integration with DICE Deployment Service (based on Cloudify and Chef cookbooks), software engineers/architects only need to annotate appropriate nodes in the DDSM model or TOSCA blueprint as monitorable and the Deployment service will install and configure the agents on selected nodes, so that the moment the DIA is deployed on the cluster the runtime data will be flowing into the DMon platform, with absolutely no manual intervention from end-users.

Engineered using a micro-services architecture, the platform is easy to deploy, and operate, on heterogeneous distributed Cloud environments. We reported successful deployment on Flexiant Cloud Orchestrator and OpenStack using Vagrant scripts.

We have also evaluated the scalability and performance of the platform's main bottleneck component, Logstash. Experiments shows that increasing YG size and applying JVM optimization flags bring higher throughput for Logstash service.

As next research and development steps in relation to DICE Monitoring platform, we will continue to actively support DICE demonstrators and tool owners in their efforts to integrate the platform in their software systems. As immediate research goals, we intend to study the impact of increasing the number of workers and buffer size in Logstash on throughput in order to be able to tune the deployment for better performance. In the medium term, we plan to use the collected insights and data from profiling Logstash to create a machine learning based predictive model that will be able to detect event trends and, using a multi-agent based self-management module, autonomously enact the required actions on the platform (adding new Logstash/Elasticsearch nodes, or shutting down when there are not needed).

## 8. Supplemental resources

1. DICE Monitoring Platform Github repository  
<https://github.com/dice-project/DICE-Monitoring/wiki>
2. DICE Monitoring Platform installation / deployment guide:  
<https://github.com/dice-project/DICE-Monitoring/wiki/Installation>
3. DICE Monitoring Platform REST API:  
<https://github.com/dice-project/DICE-Monitoring/wiki/Getting-Started>
4. Deliverable D4.1  
[http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/02/D4.1\\_Monitoring-and-data-warehousing-tools-Initial-version.pdf](http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/02/D4.1_Monitoring-and-data-warehousing-tools-Initial-version.pdf)
5. DICE Monitoring Platform plug-in for DICE IDE  
<https://github.com/dice-project/DICE-Monitoring/wiki/DICE-IDE-Tutorial>
6. DICE Monitoring Platform Video  
<https://drive.google.com/open?id=0B5xg7vkGTDqmTDY5Z25ubnhsTWM>
7. DICE Monitoring Platform Source Code:  
<https://github.com/dice-project/DICE-Monitoring>

## References

- [1] DICE Consortium, Monitoring and data warehousing tools - Initial version (Deliverable 4.1), 2016, [http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/02/D4.1\\_Monitoring-and-data-warehousing-tools-Initial-version.pdf](http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/02/D4.1_Monitoring-and-data-warehousing-tools-Initial-version.pdf)
- [2] DICE Consortium, Requirements specifications (Deliverable 1.2), 2015, [http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2015/08/D1.2\\_Requirement-specification.pdf](http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2015/08/D1.2_Requirement-specification.pdf)
- [3] DICE Consortium, Requirement Specification M16 update (Deliverable 1.2 companion document), 2016, M16 updated version. <http://wp.doc.ic.ac.uk/dice-h2020/wpcontent/uploads/sites/75/2016/05/Requirement-Specification-M16.pdf>
- [4] DICE Consortium, Architecture definition and integration plan - Final version (Deliverable 1.4), 2017, <http://www.dice-h2020.eu/deliverables/>
- [5] DICE Consortium, Quality anomaly detection and trace checking tools - Initial Version (Deliverable 4.3), 2016, <http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/08/D4.3-Quality-anomaly-detection-and-trace-checking-tools-Initial-version.pdf>
- [6] DICE Consortium, DICE delivery tools - Initial version (Deliverable D5.2), 2017, <http://www.dice-h2020.eu/deliverables/>
- [7] DICE Consortium, Iterative quality enhancement tools – Initial version (Deliverable D4.5), 2016, <http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/08/D4.5-Iterative-quality-enhancement-tools-Initial-version.pdf>
- [8] DICE Consortium, DICE Framework - Initial version (Deliverable D1.5), 2017, <http://www.dice-h2020.eu/deliverables/>
- [9] S. Chhajed, Learning ELK Stack. Packt Publishing, 2015
- [10] G. Casale et al., DICE: Quality-driven development of data-intensive Cloud applications, in 2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering, May 2015, pp. 78–83
- [11] G. Iuhász 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), Sept 2015, pp. 363–366.
- [12] Indexing performance tips, <https://www.elastic.co/guide/en/elasticsearch/guide/current/indexing-performance.html> [last access: 26/01/2017]
- [13] J. Prante, Elasticsearch java virtual machine settings explained, Nov. 2012, <http://jprante.github.io/2012/11/28/Elasticsearch-Java-Virtual-Machine-settings-explained.html> [last access: 26/01/2017]
- [14] C. Gormley and Z. Tong, Elasticsearch: The Definitive Guide. O'Reilly Media, Jan. 2015.
- [15] Logstash configuration tuning, <https://www.elastic.co/blog/logstash-configuration-tuning> [last access: 26/01/2017]
- [16] Logstash processing pipeline, <https://www.elastic.co/guide/en/logstash/current/pipeline.html> [last access: 26/01/2017]
- [17] F. Xian et al., Investigating throughput degradation behavior of java application servers: A view from inside a virtual machine, in Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java, ser. PPPJ '06. New York, NY, USA: ACM, 2006, pp. 40–49, <http://doi.acm.org/10.1145/1168054.116806>
- [18] S.M.Blackburnetal, Mythsandrealities: The performance impact of garbage collection, in Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, ser.

SIGMETRICS '04/Performance '04. New York, NY, USA: ACM, 2004, pp. 25–36,  
<http://doi.acm.org/10.1145/1005686.1005693>

[19] Deploying and scaling logstash, <https://www.elastic.co/guide/en/logstash/current/deploying-and-scaling.html> [last access: 26/01/2017]

[20] J. L. Carlson, Redis in Action. Greenwich, CT, USA: Manning Publications Co., 2013.

[21] N. Garg, Apache Kafka. Packt Publishing, 2013.

[22] Memory management in the Java HotSpot Virtual Machine, Sun Microsystems, 2006.

[23] S. Oaks, Java Performance: The definitive guide First Edition. O'Reilly Media, 2014.

[24] D. Detlefs et al., Garbage-first garbage collection, in Proceedings of the 4th International Symposium on Memory Management, ser. ISMM '04. New York, NY, USA: ACM, 2004, pp. 37–48. [Online]. Available: <http://doi.acm.org/10.1145/1029873.1029879>

[25] N. J. Gunther, UNIX load average - Part 2: Not your average average, TeamQuest, 2010.

[26] C. Hunt and B. John, Java Performance 1st., Addison Wesley, 2011.

[27] G. Iuhasz, D. Pop and I. Dragan, Architecture of a scalable platform for monitoring multiple Big Data frameworks, in Journal of Scalable Computing: Practice and Experience, vol 17, no 4, 2016

[28] M. Peacock, Creating Development Environments with Vagrant. Community experience distilled. Packt Publishing, 2013

[29] Wes McKinney, Pandas. <https://github.com/pydata/pandas> or <http://pandas.pydata.org/>

[30] Chef cookbooks, <https://docs.chef.io/resources.html>

[31] Faruk Akgul, ZeroMQ, Packt Publishing, 2013

[32] Emrah Ayanoglu, Yusuf Aytas, and Dotan Nahum, Mastering RabbitMQ, Packt Publishing, 2016

## Appendix A. REST API Documentation

There are two main components in this API:

- First we have the management and deployment/provisioning component called **Overlord**(Monitoring Management API).
  - It is responsible for deployment and management of the Monitoring Core components: ElasticSearch, Logstash Server and Kibana.
  - Besides it is also responsible for the auxiliary component management and deployment. These include: Collectd, Logstash-forwarder.
- Second, we have the interface used by other applications to query the DataWarehouse represented by ElasticSearch. This component is called **Observer**.
  - It is responsible for returning the monitoring metrics in various formats (CSV, JSON, simple output).

### Overlord (Monitoring Management API)

The Overlord is composed from two major components:

- **Monitoring Core** represented by: ElasticSearch, LogstashServer and Kibana
- **Monitoring Auxiliary** represented by: Collectd, Logstash-Forwarder

#### *Monitoring Core*

GET /v1/log

Return the log of dmon. It contains information about the last requests and the IPs from which they originated as well as the status information from various sub components.

The D-Mon internal logging system lists 3 types of messages. INFO messages represent debug level information, WARNING is for handled exceptions and finally ERROR for caught errors.

GET /v1/overlord

Returns information regarding the current version of the Monitoring Platform.

GET /v1/overlord/framework

Returns the currently supported frameworks.

```
{
  "Supported Frameworks":["list_of_frameworks"]
}
```

GET /v1/overlord/framework/{fwork}

Returns the metrics configuration file for big data technologies. The response will have the file mime-type encoded. For HDFS,Yarn and Spark it is set to 'text/x-java-properties' while for Storm it is 'text/yaml'.

PUT /v1/overlord/application/{appID}

Registers an application with D-Mon and creates a unique tag for the monitored data. The tag is defined by *appID*. Each *appID* will be added as a tag to all performance metrics for the data intensive application it identifies.

POST /v1/overlord/core

Deploys all monitoring core components provided that they have values for the preset hosts. If not it deploys all components locally with default settings.

NOTE: Currently the '-l' flag of the start script *dmon-start.sh* does the same as the later option.

GET /v1/overlord/core/database

Return the current internal state of D-MON in the form of an sqlite2 database. The response has *application/x-sqlite3* mimetype.

PUT /v1/overlord/core/database

Can submit a new version of the internal database to D-MON by replacing the current states with new ones. The old states are backed up before applying the changes. The database should be formatted as a sqlite3 database file and sent using the *application/x-sqlite3* mimetype.

GET /v1/overlord/core/status

Returns the current status of the Monitoring platform.

```
{
  "ElasticSearch":{
    "Status":"<HTTP_CODE>",
    "Name":"<NAME>",
    "ClusterName":"<CLUSTER_NAME>",
    "version":{
      "number":"<ES_VERSION>",
      "BuildHash":"<HASH>",
      "BuildTimestamp":"<TIMESTAMP>",
      "BuildSnapshot":"<BOOL>",
      "LuceneVersion":"<LC_VERSION>"
    }
  },
  "Logstash":{
    "Status":"<HTTP_CODE>",
    "Version":"<VNUMBER>"
  },
  "Kibana":{
    "Status":"<HTTP_CODE>",
    "Version":"<VNUMBER>"
  }
}
```

NOTE: Only works for local deployments. It returns the current state of local ElasticSearch, Logstash server and Kibana status information.

GET /v1/overlord/detect/storm

Returns information regarding the currently registered Storm cluster. It will also list all available topologies.

POST /v1/overlord/detect/storm

Tries to detect if the current registered nodes have a valid storm deployment. It will first test if there are any nodes that have a Storm endpoint and port set. If this step fails it starts to scan all registered nodes. In case it finds the endpoint, the first topology is set in order to be monitored. Then, it sets all configurations necessary for monitoring storm automatically.

GET /v1/overlord/storm/logs

Returns the currently available storm logs

```
{
  "StormLogs": [
    "workerlogs_2016-12-07-15:05:10.tar",
    "workerlogs_2016-12-07-15:25:12.tar",
    "workerlogs_2016-12-07-16:00:09.tar",
  ]
}
```

```
"workerlogs_2016-12-07-16:04:00.tar",  
"workerlogs_2016-12-07-16:08:01.tar",  
"workerlogs_2016-12-07-16:50:04.tar",  
"workerlogs_2016-12-08-16:22:18.tar",  
"workerlogs_2016-12-08-16:28:51.tar",  
"workerlogs_2016-12-08-17:29:27.tar",  
"workerlogs_2016-12-08-17:30:48.tar",  
"workerlogs_2016-12-08-17:32:53.tar",  
"workerlogs_2016-12-08-17:35:11.tar"  
]  
}
```

There are only worker logs. Each logs file listed in the response can be considered as a container which contains all worker logs from all monitored Storm logs.

POST /v1/overlord/storm/logs

Starts a background process which will getch worker logs from all registered Storm nodes. This can be a long running process as each worker produces about 100MB per log and there can be more than one worker instance per node.

GET /v1/overlord/storm/logs/active

Checks if there are any running log fetch processes.

GET /v1/overlord/storm/logs/{workerlogs}

Returns the worker log files specified by *workerlogs*.

GET /v1/overlord/detect/yarn

Returns the currently registered YARN job history servers.

POST /v1/overlord/detect/yarn

Attempts to detect a YARN history server instance from the pool of monitored nodes.

PUT /v1/overlord/detect/yarn

Used to define the endpoint for a known YARN history server.

```
{  
  "NodeIP": "<node_IP>",  
  "NodePort": "<port>",  
  "Polling": 30  
}
```

Note: Logstash server instance must be restarted in order to successfully collect yarn history server related metrics.

GET /v1/overlord/history/yarn

Return all the jobs that have ever run on the registered YARN deployment.

GET /v1/overlord/history/yarn/jobs

Returns a list of YARN jobs.

GET /v1/overlord/history/yarn/jobs/tasks

Returns all jobs and their associated tasks.

GET /v1/overlord/mongo

Returns the current registered MongoDB instance information.

```
{
  "MongoDBs": "admin",
  "MongoHost": "127.0.0.1",
  "MongoPort": "27017",
  "Password": true,
  "User": true
}
```

PUT /v1/overlord/mongo

Registers a MongoDB instance to be monitored.

```
{
  "MongoDBs": "<DB_Name>",
  "MongoHost": "<IP>",
  "MongoPort": "<port>",
  "Password": "<password>",
  "User": "<user_name>"
}
```

NOTE: The registration process will only take effect after the Logstash server instance has been restarted!

GET /v1/overlord/nodes

Returns the current monitored nodes list.

```
{
  "Nodes": [
    { "<NodeFQDN1>": "NodeIP1" },
    { "<NodeFQDN2>": "NodeIP2" },
    { "<NodeFQDNn>": "NodeIPn" }
  ]
}
```

PUT /v1/overlord/nodes

Includes the given nodes into the monitored node pools. The nodes are represented as a list of dictionaries. Thus, it is possible to register one to many nodes at the same time. It is possible to assign different user names and passwords to each node.

Input:

```
{
  "Nodes": [
    {
      "NodeName": "<NodeFQDN1>",
      "NodeIP": "<IP>",
      "key": "<keyName|null>",
      "username": "<uname|null>",
      "password": "<pass|null>"
    },
    {
      "NodeName": "<NodeFQDNn>",
      "NodeIP": "<IP>",
      "key": "<keyName|null>",
      "username": "<uname|null>",
      "password": "<pass|null>"
    }
  ]
}
```

NOTE: Only username and key authentication is currently supported. There is a facility to use public/private key authentication which is currently undergoing testing.



#### POST /v1/overlord/nodes

Bootstrap of all non-monitored nodes. Installs, configures and starts collectd and logstash-forwarder on them. This feature is not recommended for testing, the usage of separate commands is preferred in order to detect network failures.

NOTE: Define one json to completely populate and set up dmon-controller. It can be then used to save and share internal state by sending the json between controller instances.

#### GET /v1/overlord/nodes/roles

Returns the roles currently held by each computational node.

```
{
  "Nodes": [
    {
      "dice.cdh5.mng.internal": [
        "storm",
        "spark"
      ]
    },
    {
      "dice.cdh5.w1.internal": [
        "unknown"
      ]
    },
    {
      "dice.cdh5.w2.internal": [
        "yarn",
        "spark",
        "storm"
      ]
    },
    {
      "dice.cdh5.w3.internal": [
        "unknown"
      ]
    }
  ]
}
```

If the node has an unknown service installed, or the roles are not specified the type is set to unknown.

#### PUT /v1/overlord/nodes/roles

Modifies the roles of each nodes.

Input:

```
{
  "Nodes": [
    {
      "NodeName": "<nodeFQDN>",
      "Roles": [
        "yarn"
      ]
    }
  ]
}
```

#### POST /v1/overlord/nodes/roles

Generates metrics configuration files for each role assigned to a node and uploads them to the required directory. It returns a list of all nodes to which a configuration of a certain type (i.e. yarn, spark, storm etc) has been uploaded.

```
{
  "Status":{
    "yarn":["list_of_yarn_nodes"],
    "spark":["list_of_spark_nodes"],
    "storm":["list_of_storm_nodes"],
    "unknown":["list_of_unknown_nodes"]
  }
}
```

NOTE: The directory structure is based on the Vanilla and Cloudera distribution of HDFS, Yarn and Spark. Custom installations are not yet supported. As yarn and HDFS have the same metrics system, their tags (i.e. hdfs and yarn) are interchangeable in the context of D-Mon.

GET /v1/overlord/nodes/{nodeFQDN}

Returns information of a particular monitored node identified by *nodeFQDN*.

Response:

```
{
  "NodeName":"nodeFQDN",
  "Status":"<online|offline>",
  "IP":"<NodeIP>",
  "OS":"<Operating_System>",
  "key":"<keyName|null>",
  "username":"<uname|null>",
  "password":"<pass|null>",
  "Roles":["listofroles"]
}
```

PUT /v1/overlord/nodes/{NodeFQDN}

Changes the current information of a given node. Node FQDN may not change from one version to another.

Input:

```
{
  "NodeName":"<nodeFQDN>",
  "IP":"<NodeIP>",
  "OS":"<Operating_System>",
  "Key":"<keyName|null>",
  "Username":"<uname|null>",
  "Password":"<pass|null>",
  "LogstashInstance": "<ip_logstash>"
}
```

POST /v1/overlord/nodes/{NodeFQDN}

Bootstraps specified node.

NOTE: Possible duplication with ../aux/.. branch. DEPRECATED.

DELETE /v1/overlord/nodes/{nodeFQDN}

Stops all auxiliary monitoring components associated with a particular node.

NOTE: This does *not* delete nodes or configurations; it only stops collectd and logstash-forwarder on the selected nodes. DEPRECATED.

PUT /v1/overlord/nodes/{nodeFQDN}/roles

Defines the roles each node has inside the cluster.

Input:

```
{
  "Roles":["list_of_roles"]
}
```

POST /v1/overlord/nodes/{nodeFQDN}/roles

Redeploys metrics configuration for a specific node based on the roles assigned to it.

DELETE /v1/overlord/nodes/{nodeFQDN}/purge

This resource deletes auxiliary tools from a given node and also removes all setting from D-Mon. This process is irreversible.

GET /v1/overlord/core/es

Return a list of current hosts comprising the ES cluster core components. The first registered host is set as the default master node. All subsequent nodes are set as workers. If status is *detached* then core Elasticsearch instance is running as a daemon.

```
{
  "ES Instances": [
    {
      "DataNode": true,
      "ESClusterName": "diceMonit",
      "ESCoreDebug": "0",
      "ESCoreHeap": "3g",
      "FieldDataCacheExpire": "6h",
      "FieldDataCacheFilterExpires": "6h",
      "FieldDataCacheFilterSize": "20%",
      "FieldDataCacheSize": "20%",
      "HostFQDN": "dice.cdh5.dmon.internal",
      "IP": "127.0.0.1",
      "IndexBufferSize": "30%",
      "MasterNode": true,
      "MinIndexBufferSize": "96mb",
      "MinShardIndexBufferSize": "12mb",
      "NodeName": "esCoreMaster",
      "NodePort": 9200,
      "NumOfReplicas": 1,
      "NumOfShards": 5,
      "OS": "ubuntu",
      "PID": 2531,
      "Status": "Running"
    }
  ]
}
```

POST /v1/overlord/core/es

Generates and applies the new configuration options for the ES Core components. During this request the new configuration will be generated.

NOTE: This version of the resource is deprecated use /v2 version.

POST /v2/overlord/core/es

Generates and applies the new configuration options for the ES Core components. During this request the new configuration will be generated. It uses init script for startup increasing performance and reliability. Can be detached from dmon-controller instance.

NOTE: If the configuration is unchanged ES Core will not be restarted! It is possible to deploy the monitoring platform on different hosts than Elasticsearch only in case that the FQDN or IP is provided.

GET /v1/overlord/core/es/config

Returns the current configuration file for Elasticsearch in the form of a YAML file.

NOTE: The first registered Elasticsearch information will be set by default to be the master node.

PUT /v1/overlord/core/es/config

Changes the current configuration options for the Elasticsearch instance defined by it's FQDN and IP.

Input:

```
{
  "DataNode": true,
  "ESClusterName": "string",
  "ESCoreDebug": 1,
  "ESCoreHeap": "4g",
  "FieldDataCacheExpires": "6h",
  "FieldDataCacheFilterExpires": "6h",
  "FieldDataCacheFilterSize": "20%",
  "FieldDataCacheSize": "20%",
  "HostFQDN": "string",
  "IP": "string",
  "IndexBufferSize": "30%",
  "MasterNode": true,
  "MinIndexBufferSize": "96mb",
  "MinShardIndexBufferSize": "12mb",
  "NodeName": "string",
  "NodePort": 9200,
  "NumOfReplicas": 0,
  "NumOfShards": 1,
  "OS": "unknown"
}
```

NOTE: The new configuration will **not** be generated at this step. Currently only ESClusterName, HostFQDN, IP, NodeName, NodePort are required. This will be changed in future versions.

GET /v1/overlord/core/es/status/<intComp>/property/<intProp>

Returns diagnostic data about the master elasticsearch instance.

DELETE /v1/overlord/core/es/<hostFQDN>

Stops the ElasticSearch (es) instance on a given host and removes all configuration data from DMON.

GET /v1/overlord/core/es/cluster/health

Return the current state of the cluster.

GET /v1/overlord/core/es/cluster/settings

Return the current ES core service settings. Some of these can be set during runtime while other only during startup.

GET /v1/overlord/core/es/cluster/state

Returns the current state of all replicas, indices and shards.

GET /v1/overlord/core/es/cluster/health

Returns the health of the current D-Mon cluster.

GET /v1/overlord/core/es/node/master/info

Returns information about the current master node of the D-Mon cluster.

GET /v1/overlord/core/es/node/master/state

Return the current state of the D-Mon cluster master node.

POST /v1/overlord/core/es/<hostFQDN>/start

Start the es instance on the host identified by *hostFQDN*. It uses the last good generated es configuration.

POST /v1/overlord/core/es/<hostFQDN>/stop

Stops the es instance on the host identified by *hostFQDN*.

POST /v1/overlord/core/halt

Stops all core components on every node.

GET /v1/overlord/core/es/index/{indexName}

Returns the current status of the desired index identified by *indexName*.

GET /v1/overlord/core/es/<hostFQDN>/status

Returns the current status (Running, Stopped, Unknown) and PID of the es instance on the host identified by *hostFQDN*.

GET /v1/overlord/core/ls

Returns the current status of all Logstash server instances registered with D-Mon. If status is *detached* then core ls instance is running as a daemon.

Response:

```
{
  "LS Instances": [
    {
      "ESClusterName": "diceMonit",
      "HostFQDN": "dice.cdh5.dmon.internal",
      "IP": "109.231.121.210",
      "LPort": 5000,
      "LSCoreHeap": "512m",
      "LSCoreSparkEndpoint": "None",
      "LSCoreSparkPort": "None",
      "LSCoreStormEndpoint": "None",
      "LSCoreStormPort": "None",
      "LSCoreStormTopology": "None",
      "OS": "ubuntu",
      "Status": "Running",
      "udpPort": 25680
    }
  ]
}
```

POST /v1/overlord/core/ls

Starts the logstash server based on the configuration information. During this step the configuration file is first generated.

NOTE: This resource is deprecated use /v2 instead

POST /v2/overlord/core/ls

Starts the logstash server based on the configuration information. During this step the configuration file is first generated. Uses init script for startup. Enabled watch dog support in order to increase performance and reliability.

DELETE /v1/overlord/core/ls/<hostFQDN>

Stops the logstash server instance on a given host and removes all configuration data from DMON.

GET /v1/overlord/core/ls/config

Returns the current configuration file of Logstash Server.

PUT /v1/overlord/ls/config

Changes the current configuration of Logstash Server.

Input:

```
{
  "ESClusterName": "diceMonit",
  "HostFQDN": "string",
  "IP": "string",
  "Index": "logstash",
  "LPort": 5000,
  "LSCoreHeap": "512m",
  "LSCoreSparkEndpoint": "None",
  "LSCoreSparkPort": "None",
  "LSCoreStormEndpoint": "None",
  "LSCoreStormPort": "None",
  "LSCoreStormTopology": "None",
  "LSCoreWorkers": "4",
  "OS": "string",
  "udpPort": 25826
}
```

NOTE: LS instances are bound by their FQDN this means that it can't change.

GET /v1/overlord/core/ls/<hostFQDN>/status

Return the status of the logstash server running on the host identified by *hostFQDN*.

POST /v1/overlord/core/ls/<hostFQDN>/start

Start the logstash server instance on the host identified by *hostFQDN*. It will use the last good configuration.

POST /v1/overlord/core/ls/<hostFQDN>/stop

Stops the logstash server instance on the host identified by *hostFQDN*.

GET /v1/overlord/core/ls/credentials

Returns the current credentials for logstash server core service.

Response:

```
{
  "Credentials": [
    {
      "Certificate": "<certificate name>",
      "Key": "<key name>",
      "LS Host": "<host fqdn>"
    }
  ]
}
```

```
}
```

NOTE: Logstash server and the Logstash forwarder need a private/public key in order to establish secure communications. During local deployment ('-l' flag) a default public private key-pair is created.

GET /v1/overlord/core/ls/cert/{certName}

Returns the hosts using a specified certificate. The certificate is identified by its *certName*.

Response:

```
{
  "Host":"[listofhosts]",
}
```

By default all Nodes use the default certificate created during D-Mon initialization. This request returns a list of hosts using the specified certificate.

PUT /v1/overlord/core/ls/cert/{certName}/{hostFQDN}

Uploads a certificate with the name given by *certName* and associates it with the given host identified by *hostFQDN*.

NOTE: The submitted certificate must use the application/x-pem-file Content-Type.

GET /v1/overlord/core/ls/key/{keyName}

Retruns the host associated with the given key identified by *keyName* parameter.

Response:

```
{
  "Host":"<LS host name>",
  "Key":"<key name>"
}
```

PUT /v1/overlord/core/ls/key/{keyName}/{hostFQDN}

Uploads a private key with the name given by *keyName* and associates it with the given host identified by *hostFQDN*.

NOTE: The submitted private key must use the application/x-pem-file Content-Type.

GET /v1/overlord/core/kb

Returns information for all Kibana instances.

```
{
  "KB Instances":[{
    "HostFQDN":"<FQDN>",
    "IP":"<host_ip>",
    "OS":"<os_type>",
    "KBPort":"<kibana_port>",
    "PID":"<kibana_pid>",
    "KBStatus":"<Running|Stopped|Unknown>"
  }
]
```

POST /v1/overlord/core/kb

Generates the configuration file and Starts or Restarts a kibana session.

NOTE: Currently supports only one instance. No distributed deployment.

GET /v1/overlord/core/kb/config

Returns the current configuration file for Kibana. Uses the mime-type 'text/yaml'.

PUT /v1/overlord/core/kb/config

Changes the current configuration for Kibana

Input:

```
{
  "HostFQDN": "<FQDN>",
  "IP": "<host_ip>",
  "OS": "<os_type>",
  "KBPort": "<kibana_port>"
}
```

GET /v1/overlord/core/kb/visualizations

Returns the current Kibana core service visualizations registered in DMON.

POST /v1/overlord/core/kb/visualizations

This will generate default visualizations and registers them inside DMON. Visualisation generations are based on each monitored node roles.

### ***Monitoring auxiliary***

GET /v1/overlord/aux

Returns basic information about auxiliary components.

GET /v1/overlord/aux/agent

Returns the current deployment status of dmon-agents.

```
{
  "Agents": [
    {
      "Agent": false,
      "NodeFQDN": "dice.cdh5.mng.internal"
    },
    {
      "Agent": false,
      "NodeFQDN": "dice.cdh5.w1.internal"
    },
    {
      "Agent": false,
      "NodeFQDN": "dice.cdh5.w2.internal"
    },
    {
      "Agent": false,
      "NodeFQDN": "dice.cdh5.w3.internal"
    }
  ]
}
```

POST /v1/overlord/aux/agent

Bootstraps the installation of dmon-agent services on nodes that are not marked as already active. It only works if all nodes have the same authentication.

GET /v1/overlord/aux/deploy

Returns monitoring component status of all nodes. This is similar to v2 of this resource.



```
{
  "NodeFQDN": "<nodeFQDN>",
  "NodeIP": "<nodeIP>",
  "Monitored": "<boolean>",
  "Collectd": "<status>",
  "LSF": "<status>",
  "LSInstance": "<ip_logstash>"
}
```

NOTE: Marked as DEPRECATED. Will be deleted in future versions.

POST /v1/overlord/aux/deploy

Deploys all auxiliary monitoring components on registered nodes and configures them.

NOTE: There are three statuses associated with each auxiliary component.

- None -> There is no aux component on the registered node
- Running -> There is the aux component on the registered node and it is currently running
- Stopped -> There is the aux component on the registered node and it is currently stopped

If the status is *None* then this resource will install and configure the monitoring components. However, if the status is *Running* nothing will be done. The services with status *Stopped* will be restarted.

All nodes can be restarted independent from their current state using the `--redeploy-all` parameter.

NOTE: Marked as DEPRECATED. Will be deleted in future versions. Use v2 version of the same resource for parallel implementation of this resource.

POST /v1/overlord/aux/deploy/{collectd|logstashfw}/{NodeName}

Deploys either collectd or logstash-forwarder to the specified node. In order to reload the configuration file the `--redeploy` parameter has to be set. If the current node status is *None* then the defined component (collectd or lsf) will be installed.

GET /v1/overlord/aux/interval

Returns the current polling time interval for all tools. This is a global setting. Future versions may be implemented for a node by node interval setting.

Output:

```
{
  "Spark": "5",
  "Storm": "60",
  "System": "15",
  "YARN": "15"
}
```

PUT /v1/overlord/aux/interval

Changes the settings for all monitored systems.

Input:

```
{
  "Spark": "5",
  "Storm": "60",
  "System": "15",
  "YARN": "15"
}
```

```
}
```

GET /v1/overlord/aux/{collectd|logstashfw}/config

Returns the current collectd or logstashforwarder configuration file.

PUT /v1/overlord/aux/{collectd|logstashfw}/config

Changes the configuration/status of collectd or logstashforwarder and restarts all auxiliary components.

POST /v1/overlord/aux/{auxComp}/start

Starts the specified auxiliary component on all nodes.

NOTE: This resource is DEPRECATED. Use v2 instead.

POST /v1/overlord/aux/{auxComp}/stop

Stops the specified auxiliary components on all nodes.

NOTE: This resource is DEPRECATED. Use v2 instead.

POST /v1/overlord/aux/{auxComp}/{nodeFQDN}/start

Starts the specified auxiliary component on a specific node.

NOTE: This resource is DEPRECATED. Use v2 instead.

POST /v1/overlord/aux/{auxComp}/{nodeFQDN}/stop

Stops the specified auxiliary component on a specific node.

NOTE: This resource is DEPRECATED. Use v2 instead.

Note: Some resources have been redesigned with parallel processing in mind. These use greenlets (gevent) to parallelize as much as possible the first version of the resources. These parallel resources are marked with ../v2/... All other functionality and return functions are the same.

For the sake of brevity these resources will not be detailed. Only additional functionality will be documented.

POST /v2/overlord/aux/deploy

Sets up the dmon-agent based on what roles are registered for each node.

POST /v2/overlord/aux/{auxComp}/{nodeFQDN}/configure

Configures dmon-agent *auxComp* on node *nodeFQDN*.

POST /v2/overlord/aux/{auxComp}/configure

Configures dmon-agent *auxComp* on all nodes.

GET /v2/overlord/aux/deploy/check

Polls dmon-agents from the current monitored cluster.

```
{
  "Failed": [],
  "Message": "Nodes updated!",
  "Status": "Update"
}
```

If nodes don't respond they are added to the *Failed* list together with the appropriate HTTP error code.

GET /v2/overlord/aux/status

Returns the current status of all nodes and auxiliary components:

Outputs:

```
{
  "Aux Status": [
    {
      "Collectd": "Running",
      "LSF": "Running",
      "Monitored": true,
      "NodeFQDN": "dice.cdh5.mng.internal",
      "NodeIP": "109.231.121.135",
      "Status": true
    },
    {
      "Collectd": "Running",
      "LSF": "Running",
      "Monitored": true,
      "NodeFQDN": "dice.cdh5.w1.internal",
      "NodeIP": "109.231.121.194",
      "Status": true
    },
    {
      "Collectd": "Running",
      "LSF": "Running",
      "Monitored": true,
      "NodeFQDN": "dice.cdh5.w2.internal",
      "NodeIP": "109.231.121.134",
      "Status": true
    },
    {
      "Collectd": "Running",
      "LSF": "Running",
      "Monitored": true,
      "NodeFQDN": "dice.cdh5.w3.internal",
      "NodeIP": "109.231.121.156",
      "Status": true
    }
  ]
}
```

POST /v2/overlord/aux/{auxComp}/start

Start *auxComp* on all nodes using parallel calls to the dmon-agent.

POST /v2/overlord/aux/{auxComp}/stop

Stops *auxComp* on all nodes using parallel calls to the dmon-agent.

POST /v2/overlord/aux/{auxComp}/{nodeFQDN}/start

Start *auxComp* on node *nodeFQDN* using parallel calls to the dmon-agent.

POST /v2/overlord/aux/{auxComp}/{nodeFQDN}/stop

Stops *auxComp* on node *nodeFQDN* using parallel calls to the dmon-agent.

## Observer

GET /v1/overlord/application

This returns all application versions currently registered with D-Mon.

```
{
  "appID-1": {
    "start": "2016-11-23 11:37:02.800271",
    "status": "STOPPED",
    "stop": "2016-12-14 09:29:05.523252",
    "ver": "3"
  },
  "appID-2": {
    "start": "2016-12-14 09:29:05.523252",
    "status": "ACTIVE",
    "stop": null,
    "ver": "1"
  }
}
```

NOTE: Each application ID can have several versions.

GET /v1/observer/applications/{appID}

Returns information about a YARN application identified by *{appID}*. The information will not contain monitoring data, only a general overview. This is similar to YARN History Server.

GET /v1/observer/nodes

Returns the current monitored nodes list. Listing is only limited to node FQDN and current node IP.

NOTE: Some cloud providers assign the IP dynamically at VM startup. Because of this D-Mon treats the FQDN as a form of UUID.

Response:

```
{
  "Nodes": [
    { "<NodeFQDN1>": "NodeIP1" },
    { "<NodeFQDN2>": "NodeIP2" },
    { "<NodeFQDNn>": "NodeIPn" }
  ]
}
```

GET /v1/observer/nodes/{nodeFQDN}

Returns information about a particular monitored node. No information is limited to non confidential information, no authentication credentials are returned. The *Status* field is true if dmon-agent has been already deployed, while *Monitored* is true if it is also started. *LSInstance* represents the Logstash instance the node is assigned to.

Response:

```
{
  "<nodeFQDN>": {
    "Status": "<boolean>",
    "IP": "<NodeIP>",
    "Monitored": "<boolean>",
    "OS": "<Operating_System>",
    "LSInstance": "<ip_of_logstash>"
  }
}
```

GET /v1/observer/nodes/{nodeFQDN}/roles

Returns roles of the node identified by '*nodeFQDN*'.

Response:

```
{
  "Roles":["yarn","spark"]
}
```

NOTE: Roles are returned as a list. Some elements represent in fact more than one service, for example 'yarn' represents both 'HDFS' and 'Yarn'.

POST /v1/observer/query/{csv/json/plain}

Returns the required metrics in csv, json or plain format.

Input:

```
{
  "DMON":{
    "query":{
      "size":"<SIZEinINT>",
      "ordering":"<asc|desc>",
      "queryString":"<query>",
      "tstart":"<startDate>",
      "tstop":"<stopDate>"
    }
  }
}
```

Output depends on the option selected by the user: csv, JSON or plain.

NOTE: The filter metrics must be in the form of a list. Also, filtering currently works only for CSV and plain output. Future versions will include the ability to export metrics in the form of RDF+XML in concordance with the OSLC Performance Monitoring 2.0 standard. It is important to note that we will export in this format only system metrics. No Big Data framework specific metrics.

From version 0.1.3 it is possible to omit the *tstop* parameter, instead it is possible to define a time window based on the current system time:

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

where s stands for second or m for minutes and h for hours.

From version 0.2.0 it is possible to specify custom index to be used in the query. The index definition supports the \* wildcard character.

```
{
  "DMON":{
    "query":{
      "size":"<SIZEinINT>",
      "index":"logstash-*",
      "ordering":"<asc|desc>",
      "queryString":"<query>",
      "tstart":"now-30s"
    }
  }
}
```

POST /v2/observer/query/{csv/json}

This resource has predefined queries for all supported technologies. The query payload is of the form:

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

The term aggregation can have the following values: system, yarn, spark, storm. It is based on dataframes and has a much better performance for large data queries. This should be used instead of */v1* query resource for large queries.