Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

Monitoring and data warehousing tools – Initial version

Deliverable 4.1

Deliverable:	D4.1
Title:	Monitoring and data warehousing tools – Initial version
Editor(s):	Daniel Pop (IEAT)
Contributor(s):	Daniel Pop (IEAT), Gabriel Iuhasz (IEAT)
Reviewers:	Darren Whigham (FLEXI), Ilias Spais (ATC)
Type (R/P/DEC):	Demonstrator
Version:	1.0
Date:	31-January-2016
Status:	Final version
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2016, DICE consortium – All rights reserved

DICE partners

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

This deliverable documents the monitoring and data warehousing platform, namely DICE Monitoring Platform (DMon), developed in T4.1. This component is central to DICE architecture being used by simulation, optimization, verification and quality enhancement tools. The initial version of the platform is powered by a suite of open-source frameworks (Elasticsearch - for storage and indexing, LogStash – for log processing, and Kibana - for visualization) and provides its services through a RESTful API. In the initial version, the platform is able to collect, store, index and query logs produced by the following popular Big Data technologies: Apache YARN, HDFS and Spark.

The document is structured as follows: the Introduction section highlights the objectives and features of DMon platform and describes the contributions of the platform to DICE objectives and DICE innovation objectives. This is followed by the presentation of the position of DMon inside overall architecture and its interfaces to other DICE tools. First section highlights in its last sub-section the achievements of the period under report. The second section connects the DICE Monitoring platform to DICE use cases and requirements identified and presented in deliverable D1.2. The third section, Architecture and design of the tool, details the constituent components of the platform and its microservices, while the deployment and validation of the DMon platform are tackled in section 4. Section 5 – Documentation – provides additional references to REST API documentation and deployment and installation guides. Last section draws final conclusions and sets the future development plans for DICE Monitoring platform.

Table of contents

List of figures

List of tables

1. Introduction

This deliverable presents the initial release of the DICE Monitoring Platform (DMon), whose main goal is to collect, store and provide access to monitoring data collected from various Big Data technologies, such as Apache Hadoop, Spark or Storm. The Monitoring Platform is being developed in the task T4.1, part of WP4 work package, and its main objectives are:

- Collect, store and query monitoring data collected from various Big Data technologies
- Scalability, extensibility and high-availability platform
- Easily deployable on Cloud environment using DICE tool chain

Main features of the platform:

- Distributed architecture (Microservices)
- Deployable on Cloud environment, as well as on bare-metal
- High availability
- Access through RESTful API
- Easy installation
- Vagrant scripts for development and testing phases
- Integrates data from multiple Big Data platforms in a unique platform
- Horizontal scaling

The remaining of this section presents the positioning of DMon relative to DICE innovation objectives, DICE objectives and relation to other tools from DICE tool-chain.

1.1. Relation to DICE innovation objectives

The focus of the DICE project is to define a quality-driven framework for developing data-intensive applications that leverage Big Data technologies hosted in private or public clouds. DICE will offer a novel profile and tools for data-aware quality-driven development. The methodology will excel for its quality assessment, architecture enhancement, agile delivery and continuous testing and deployment, relying on principles from the emerging DevOps paradigm. The DICE Monitoring Platform contributes to all core innovations of DICE, as follows:

● *Innovation 1: Tackling skill shortage and steep learning curves in quality-driven development of data-intensive software through open source tools, models, methods and methodologies.*

The Monitoring platform's automatic deployment and support of key Big Data technologies enable end-users to install and monitor existing infrastructures just by executing a Vagrant script that does all the job, or integrate the platform into Chef infrastructure. The platform provides a rich Web user interface where users can visualize the status of their resource usage.

● *Innovation 2: Shortening the time to market for data-intensive applications that meet quality requirements, thus reducing costs for ISVs while at the same time increasing value for end-users.*

Optimization, verification, simulation and iterative quality enhancement tools use the collected monitoring data in their initial step.

● *Innovation 3: Decreasing costs to develop and operate data-intensive cloud applications, by defining algorithms and quality reasoning techniques to select optimal architectures, especially in the early development stages, and taking into account SLAs.*

Thanks to Monitoring platform easy deployment, integration with Chef configuration management system, support of key Big Data technologies and open-source delivery model, the costs of monitoring data-intensive Cloud applications are highly reduced. Monitoring data can serve as a valuable starting iteration of reasoning techniques for architecture optimization.

● *Innovation 4: Reducing the number and severity of quality-related incidents and failures by leveraging DevOps-inspired methods and traditional reliability and safety assessment to iteratively learn application runtime behaviour.*

The Runtime application behaviour is collected and stored by the Monitoring platform. Connected tools of DICE toolchain, such as Enhancement tools, Trace Checking tools, Anomaly Detection tools, will query the platform to get the data they need either in real-time, or offline.

1.2. Relation to DICE objectives

The following table highlights the contributions of DICE Monitoring platform to DICE objectives.

Table 1 Relation to DICE objectives

DICE Objective Description	**Relation to Monitoring Platform**
DICE profile and methodology Define a data-aware profile and a data-aware methodology for model-driven development of data-intensive cloud applications. The profile will include data abstractions (e.g., data flow path synchronization), quality annotations (e.g., data flow rates) to express requirements on reliability, efficiency and safety (e.g., execution time or availability constraints).	None
Quality analysis tools Define a quality analysis tool-chain to support quality related decision-making through simulation and formal verification.	Simulation and verification tools (WP3) use quality metrics computed by Monitoring platform as initial iteration of their analysis.
Quality enhancement tools An approach leveraging on DevOps tools to iteratively refine architecture design and deployment by assimilating novel data from the runtime, feed this information to the design time and continuously redeploy an updated application configuration to the target environment.	Data from runtime is collected, processed and served by the Monitoring platform. Thus, enhancement tools will query the platform to obtain the input data they need.
Deployment and testing tools Define a deployment and testing toolset to accelerate delivery of the application.	In this initial version, all components of the platform (core and monitoring agents) are automatically deployed using simple Vagrant script, respectively scripts executed via `SSH` on

	monitored nodes. Its deployment will be integrated with Deployment tools (WP5).
IDE Release an Integrated Development Environment (IDE) to simplify adoption of the DICE methodology.	None
Open-source software Release the DICE tool-chain as open source software.	Monitoring platform relies on open-source software (Elasticsearch, Logstash, Kibana, collectd) and is made available as open-source.
Demonstrators Validate DICE productivity gains across industrial domains through 3 use cases on data-intensive applications for media, e-government, and maritime operations.	Applications provided by ATC and NETF will be using the Monitoring platform.
Dissemination, communication, collaboration and standardisation Promote visibility and adoption of project results through dissemination, communication, collaboration with other EU projects and standardisation activities.	Monitoring platform is being presented in Big Data and innovation events (e.g. Romanian BigData Roadshow, InnoMatch 2015), as well as in scientific publications.
Long-term tool maintenance beyond life of project. The project coordinator (IMP) will lead maintenance of tools, project website and user community beyond DICE project lifespan.	Monitoring platform source code and homepage are stored using Github publicly, as open-source software. Community is welcome to contribute to the platform, during and after DICE end.

1.3. Relation to other DICE tools

Figure 1 illustrates the interfaces between the DICE Monitoring platform (DMon), namely Monitoring in the figure, and the other DICE tools. The main goal of DMon is to provide a flexible yet lightweight platform that enables the fine grained monitoring of big data applications by various actors: developers, architects or software engineers. It must also be able to serve in near real-time monitoring metrics to other DICE tools comprising the DICE tool-chain, such as Configuration optimization, Enhancement tool and Quality testing tool.

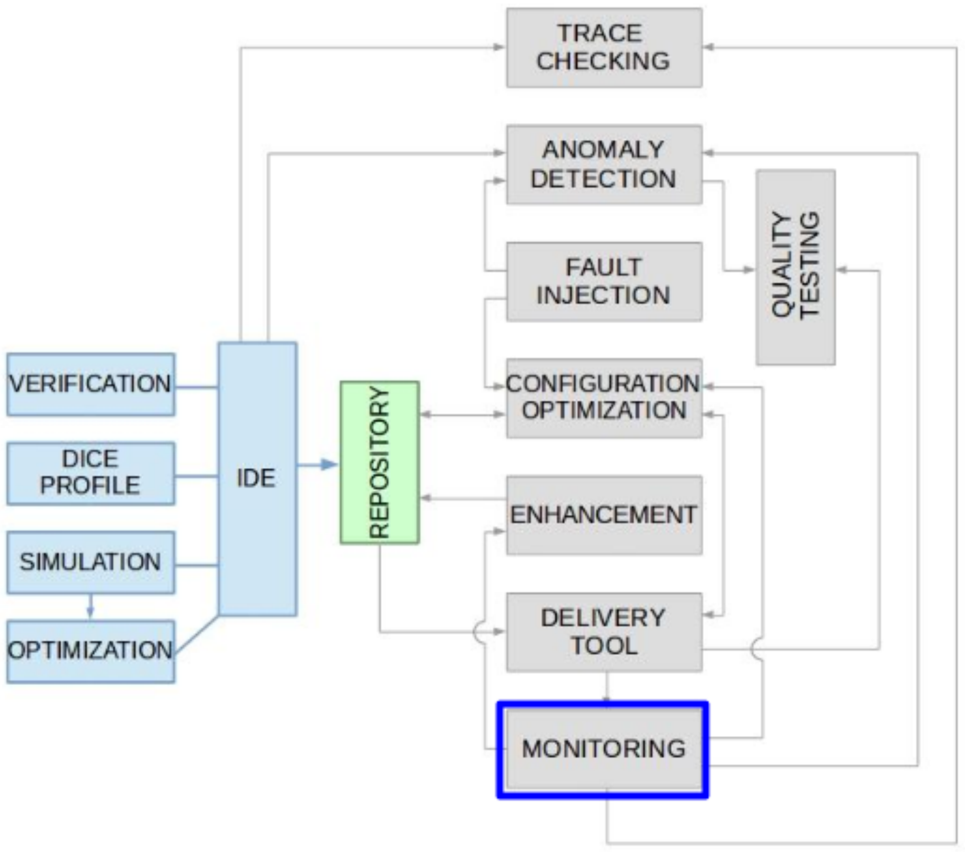


Figure 1 DICE Overall architecture

It can be easily seen from Figure 1 that DMon supplies monitoring data to various DICE tools. The deepest level of integration will be with the Anomaly Detection Tool. This is because the DMon and the anomaly detection tool will be part of a Lambda Architecture. DMon will be used as a serving layer. More details on this can be found in section 3.3.

Other tools such as the Enhancement and Configuration Optimization tools will query DMon. These tools will then use the returned metrics to monitor the current big data framework status and the impact of the modifications enacted by them. They can also use historical data available in DMon to create and validate various predictive models. Each tool will have to know the endpoint of DMon.

The trace-checking tool requires logs from big data frameworks as traces. It will parse these logs in order to extract valuable insight into the current status of the framework. These logs will be collected and indexed by DMon.

1.4. Achievements of the period under report

Overview of the main achievements in the reported period:

- Collect monitoring data from Apache YARN, Apache HDFS and Apache Spark via their metrics systems.
- Collect system resource (CPU, memory, network) usage thanks to specific `collectd` plugins

- Store collected data in a unique storage platform powered by the ELK stack (Elasticsearch, Logstash, Kibana)
- Provide access to collected data via RESTful API.
- Visualize collected data in Web GUI using Kibana dashboard extensions.

2. Requirements and Use Cases

This section reviews the requirements and use case scenarios of the monitoring platform, which were already presented the deliverable "D1.2 Requirements specifications" and its annexes [11, 12] released on month 6, and it expands on how these were addressed in the current DMon version.

2.1. Use cases

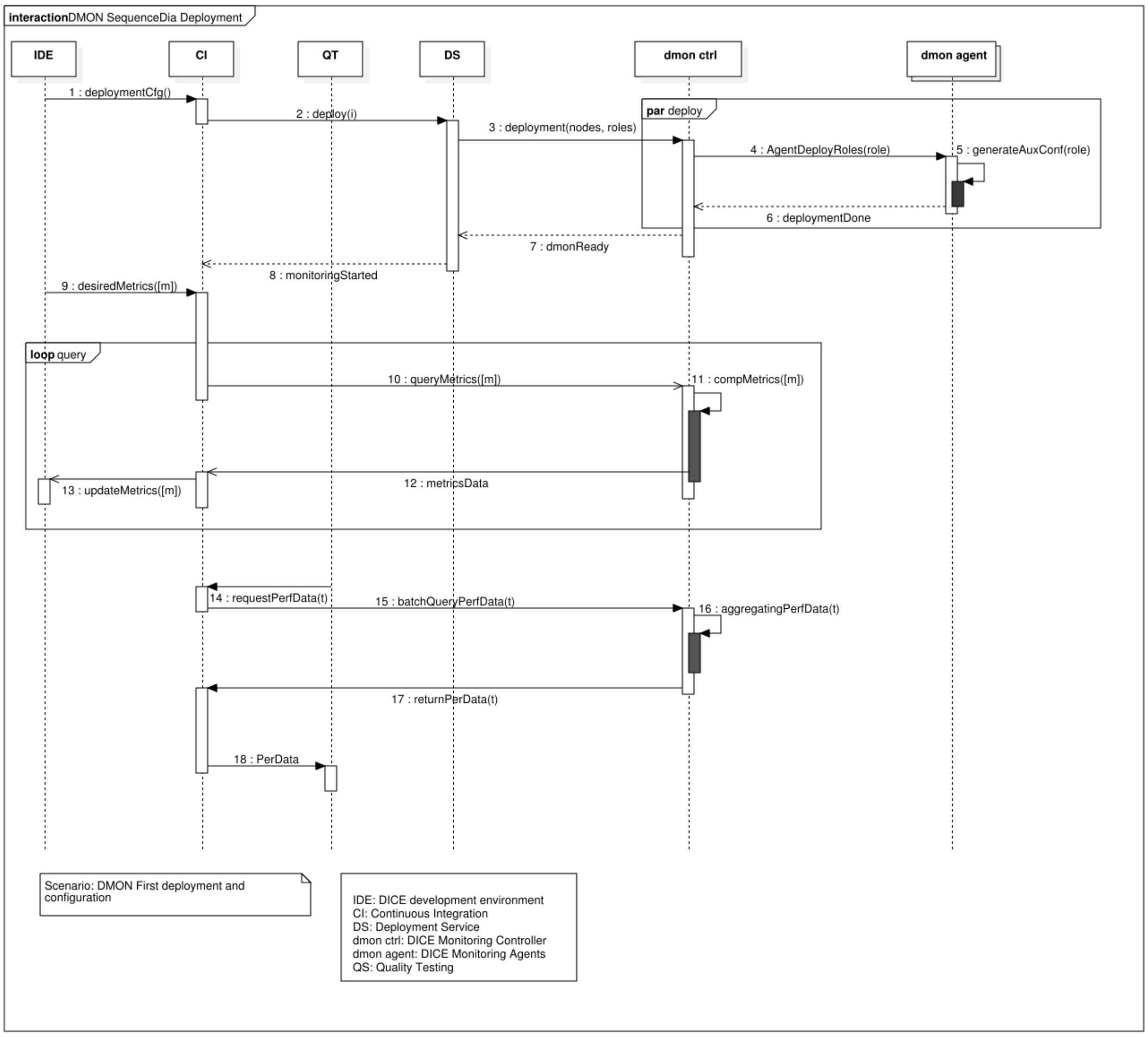

interactionDMON SequenceDia Deployment
IDE
CI
QT
DS
dmon ctrl
dmon agent
1 : deploymentCfg()
2 : deploy(i)
3 : deployment(nodes, roles)
par deploy
4 : AgentDeployRoles(role)
5 : generateAuxConf(role)
6 : deploymentDone
7 : dmonReady
8 : monitoringStarted
9 : desiredMetrics([m])
loop query
10 : queryMetrics([m])
11 : compMetrics([m])
12 : metricsData
13 : updateMetrics([m])
14 : requestPerfData(t)
15 : batchQueryPerfData(t)
16 : aggregatingPerfData(t)
17 : returnPerData(t)
18 : PerData
Scenario: DMON First deployment and configuration
IDE: DICE development environment
CI: Continuous Integration
DS: Deployment Service
dmon ctrl: DICE Monitoring Controller
dmon agent: DICE Monitoring Agents
QS: Quality Testing

Figure 2 DMon sequence diagrams

The first interaction illustrates the deployment of the Monitoring platform, scenario in which the Deployment tool (DS) is responsible to deploy the Monitoring Platform Core and the Monitoring Agent on each monitored node of the infrastructure. The entire deployment process may be triggered by different classes of end-users (DEVELOPER, ARCHITECT) via DICE IDE and Continuous Integration (CI) framework.

Once DMon is provisioned, various tools of DICE tool-chain may query the platform for needed metrics supplying the time interval of interest.

In an end-to-end scenario, the DESIGNER/ARCHITECT/DEVELOPER starts by loading the model in DICE IDE and annotates various components of the model with his/her metrics of interest (see Table 2 UC4.1.1). The DEPLOYMENT_TOOLS then deploy the application on the infrastructure, registers the monitoring agents (see Table 3 UC4.1.2) and configures them to collect the user-defined metrics. During the execution of the application, monitoring agents collect and send runtime data to the DMon platform (see Table 4 UC4.1.3). Using the Web-based user interface the end-users are able to visualize collected metrics (see Table 7 UC4.4). Alternatively, other DICE tools (such as SIMULATION_TOOLS, ENHANCEMENT_TOOLS, OPTIMIZATION_TOOLS) will query the DMon platform to retrieve; either at on-the-fly (online) or offline, data they need to perform their specific tasks (see Table 5 UC 4.2). If need, external actors may run data cleaning tasks (see Table 6 UC4.3) as a preliminary step before querying the platform.

The tables below are based on the usage scenarios included in D1.2 [11, 12] and they have been updated to include latest changes in tools terminology.

Table 2 UC4.1.1 Metrics specification on application model

Actors	DEVELOPER, ARCHITECT
Priority	REQUIRED
Flow of Events	1. The actor loads the model in DICE IDE 2. Annotates the model with metrics of interest (e.g. Arrival rate, throughout) 3. Deploys the model using DEPLOYMENT_TOOLS 4. DEPLOYMENT_TOOL configures the monitoring agents with user-defined metrics 5. Monitoring agents sends monitoring data to Monitoring Platform Core
Pre-conditions	A UML model for the application has been defined.
Post-conditions	The application UML model is annotated with requirements on the metrics to be collected.

Table 3 UC4.1.2 Monitoring agents' registration

Actors	MONITORING_TOOLS (Node Agents)
Priority	REQUIRED
Flow of Events	Each Monitoring agent, which is running on a monitored node, will: 1. Discover the the Data Warehousing component 2. Send its identifier and the list of available roles to the Data Warehousing component

	3. Negotiate the monitored metrics and acknowledge
Pre-conditions	Test application MUST be successfully deployed on test environment. Monitoring Platform Core (ELK) installed.

Table 4 UC4.1.3 Monitoring data storage

Actors	MONITORING_TOOLS
Priority	REQUIRED
Flow of Events	1. Monitoring agents connects to the Monitoring Platform Core 2. Compute the metrics by performing ETL (extract-transform-load) type jobs. 3. Store resulting metrics in the Monitoring Platform Core
Pre-conditions	Existence of data collection tools.
Post-conditions	Recorded and computed metrics stored and available in Monitoring Platform Core

Table 5 UC4.2 Querying the DMon platform

Actors	SIMULATION_TOOLS, ENHANCEMENT_TOOLS, OPTIMIZATION_TOOLS
Priority	REQUIRED
Flow of Events	1. Actors send a query to the Monitoring Platform for specific metrics and timeframe 2. MP validates the query 3. If the query is malformed then the actor receives an error message 4. If the query is correct then the actor receives a dataset as a result
Pre-conditions	Monitoring Platform available

Table 6 UC4.3 Data cleaning

Actors	ANOMALY_TRACE_TOOLS
Priority	RECOMMENDED

Flow of Events	1. ANOMALY_TRACE_TOOLS connect to the Monitoring Platform 2. ANOMALY_TRACE_TOOLS specifies an event window 3. ANOMALY_TRACE_TOOLS specifies a data cleaning algorithm to be applied on selected window 4. ANOMALY_TRACE_TOOLS launches the data cleaning task(s)
Pre-conditions	A UML model for the application has been defined.
Post-conditions	The application UML model is annotated with requirements on the metrics to be collected.

Table 7 UC4.4 Metrics visualization

Actors	DEVELOPER, ARCHITECT, ADMINISTRATOR
Priority	REQUIRED
Flow of Events	1. Actors access the Monitoring Platform WUI (Kibana) 2. Actors choose what metrics want to visualize 3. Actors choose the visualization form

Table 8 presents the status of the use cases' implementation at the end of this reporting period (M12).

Table 8 Use cases status

UC	Title	Status at M12
UC4.1.1	Metrics specification on application model	NOT STARTED
UC4.1.2	Monitoring agents registration	DONE
UC4.1.3	Monitoring data storage	DONE
UC4.2	Querying the monitoring platform	ON-GOING (Initial version of RESTful API implemented)
UC4.3	Data cleaning	NOT STARTED
UC4.4	Metrics visualization	ON-GOING (Initial visualization supported using Kibana user interface)

2.2. Requirements

This section overviews requirements expressed by the users of DICE monitoring platform and how are these addressed in the implementation of the DMon platform.

Table 9 Requirements for DMon platform

Requirement	Implementation details
R4.1: Monitoring data warehousing	Monitoring agents running on each node of the cluster forward the logfile data (exported by various Big Data frameworks or by custom applications) to the DMon platform that performs ETL.
R4.2: Monitoring data warehouse schema	DMon platform's input/output format is JSON; 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, or 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	Build number is provided by the Deployment tool as tag.
R4.3: Monitoring data extractions	Logstash server performs ETL on incoming data before sending data to Elasticsearch for storage and indexing.
R4.4: Monitoring data format transformations	Data is stored internally as JSON messages and it's transformed to requested format upon delivery.
R4.5: Monitoring data retention policy	Elasticsearch indexes are archived and deleted after some time interval.
R4.6: Monitoring data access restrictions	User authentication using user name and password, or SSL certificates
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 is supported by Kibana framework.
R4.9: Data Warehouse replication	Elasticsearch, which powers the data warehouse, is natively highly available and supports replication; Supporting high incoming rate is achieved using a queuing service handling requests addressed to Logstash.
R4.22: Time-based ordering of monitoring data entries	Domain assumption; data is consistently ordered when collected from different nodes
R4.34: Monitoring for quality tests	Using Monitoring Query API, QTESTING_TOOL queries DMon for metrics of interest (high-level metrics, such as arrival rate, throughput are supported).
R4.35: Tag monitoring data with OSLC tags	DMon exports metrics in RDF format using OSLC Perf Mon 2.0 vocabulary.

Table 10 presents the status of requirements' implementation at the end of reporting period (M12).

Table 10 Requirements status

Requirement	**Status at M12**
R4.1: Monitoring data warehousing	ON-GOING (technologies supported at M12: HDFS, YARN, Spark)
R4.2: Monitoring data warehouse schema	DONE
R4.2.1: Monitoring data versioning	NOT STARTED
R4.2.2: Supplying the version number	NOT STARTED
R4.3: Monitoring data extractions	ON-GOING
R4.4: Monitoring data format transformations	ON-GOING
R4.5: Monitoring data retention policy	NOT STARTED
R4.6: Monitoring data access restrictions	NOT STARTED
R4.7: Monitoring tools REST API	ON-GOING
R4.8: Monitoring Visualization	ON-GOING
R4.9: Data Warehouse replication	NOT STARTED
R4.22: Time-based ordering of monitoring data entries	DONE
R4.34: Monitoring for quality tests	ON-GOING
R4.35: Tag monitoring data with OSLC tags	NOT STARTED

3. Architecture and design of the DICE Monitoring Platform

The DICE Monitoring platforms (DMon) architecture is designed as a web service that enables the deployment and management of several subcomponents which in turn enable the monitoring of big data applications and frameworks. In contrast to other monitoring solutions [1, 2], DMon aims to provide as much data as possible about the current status of the big data framework subcomponents. This intent brings a wide array of technical challenges, not present in more traditional monitoring solutions, as serving near real-time fine grained metrics entails a system that should exhibit a high availability, as well as easy scalability.

Traditionally, web services have been built using a monolithic architecture where almost all components of a system ran in a single process (traditionally JVM) [30]. This type of architecture has some key advantages such as: deployment and networking are trivial while scaling such a system requires running several instances of the service behind a load-balancer instance.

On the other hand, there are some severe limitations to this type of monolithic architecture, which would directly impacts the development of DMon. Firstly, changes to one component can have an unforeseen impact on seemingly unrelated areas of the application [30], thus adding new features or any new development can be potentially costly both in time and resources. Secondly, individual components cannot be deployed independently. This means that if one only needs a particular functionality of the service this cannot be decoupled and deployed separately thus hindering reusability. Lastly, even if components are designed to be reusable these tend to focus more on readability than on performance.

Considering these limitations of a monolithic architecture, we decided to use the so called **microservice architecture** [25] for DMon, which is widely used in large Internet companies [14]. This architecture replaces the monolithic service with a distributed system of lightweight services, which are by design independent and narrowly focused. These can be deployed, upgraded and scaled individually. As these microservices are loosely coupled, it better enables code reusability, while changes made to a single service should not require changes in a different service. Integration and communication should be done using HTTP (REST API) or RPC requests. We also want to group related behaviours into separate services. This will yield a high cohesion, which enables us to modify the overall system behaviour by only modifying or updating one service instead of many.

DMon uses REST APIs for communication between different services with request payload encoded as JSON message. This makes the creation of synchronous or asynchronous messages much easier.

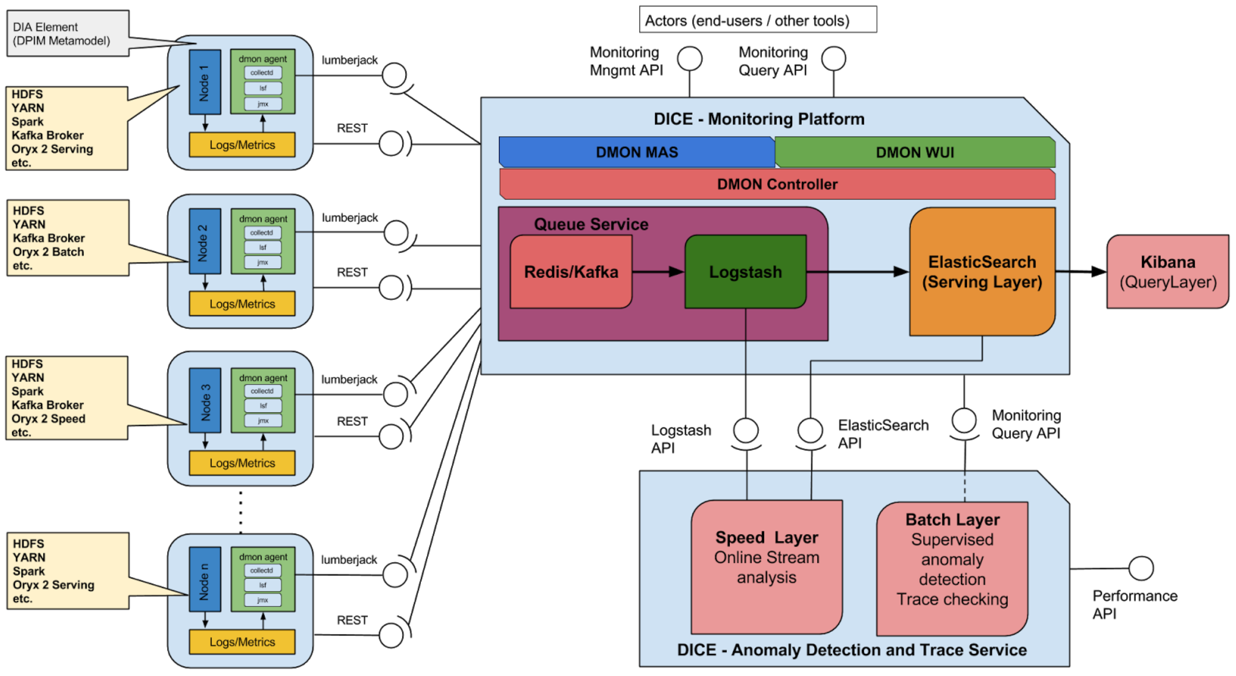


Figure 3 DMon architecture

Figure 3 shows the overall architecture of the DMon platform, which will be part of a lambda architecture [24] together with the anomaly detection and trace checking tools. In order to create a viable lambda architecture we need to create 3 layer: speed, batch and serving layers. Elasticsearch will represent the serving layer responsible for loading batch views of the collected monitoring data and enabling other tools/layers to do random reads on it. The speed layer will be used to look at recent data and represent it in a query function. In the case of anomaly detection this will mean using unsupervised learning techniques or using pre-trained models from the batch layer. The batch layer needs to compute arbitrary functions on large sections of the dataset stored in Elasticsearch. This means running long running jobs to train predictive models that than can be instantiated on the speed layer. All trained models will then be stored inside the serving layer and accessed via DMon queries.

The core components of the platform are Elasticsearch, for storing and indexing of collected data, and Logstash for gathering and processing logfile data. Kibana server provides a user-friendly graphical user interface. The main services composing DMon are the following: *dmon-controller, dmon-agent, dmon-shipper, dmon-indexer*, *dmon-wui* and *dmon-mas*. These services will be used to control both the core and node-level components.

3.1. Core Components

The core components make up the backbone of the entire monitoring platform. They will be used to collect, process, aggregate and transform all incoming monitoring data. All of these components have to be easily configurable, scalable and should support a high throughput.

Elasticsearch [15] is an open-source, RESTful search engine based on top of Apache Lucene [16]. It is an inherently (horizontally) scalable solution which can perform near real-time processing with up to 5-second latency. It also provides support for multi-tenancy, streamlined backup procedures as well as insuring data integrity. One of the most important capabilities of Elasticsearch is its ability to handle high throughput of tens or even hundreds of thousands of messages per second.

Logstash [31] is a tool developed in order to collect, process and forward events and log messages. Basically, it handles Extract, Transform and Load (ETL) operations. It uses configurable plugins for input, output and filters in order to collect, process and load data. The input plugins can be configured to accept a wide range of inputs starting from TCP/UDP to Kafka [22] topics. The input plugins then send the data for processing to the filter worker plugins. Finally, the processed data is routed to one or more output plugins such as Elasticsearch, Kafka, InfluxDB etc. One important property of Logstash is that is essentially stateless thus making it extremely scalable. For example it is possible for two Logstash instances to serve the same Elasticsearch endpoint.

Kibana [15] serves the role of browser based analytics and search interface for Elasticsearch. It was primarily developed to view Logstash processed events.

All of the above mentioned components are part of the so called ELK stack. This setup provides a very robust base for DMon and will be used as a proof-of-concept implementation.

3.2. 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 this we had to choose collectors that are flexible enough to accommodate these technologies and they also have to have a small computational footprint. By doing this we hope to limit the amount of "noise" the presence of these data collectors might have on the collected data. Also they have to be easy deployable and configurable on thousands of physical or virtual machines.

Collectd [10] is an open-source POSIX daemon that collects, transfers and stores performance and network related data. One of the main features of collectd is its wide array of available plugins [10]. In DMon we use collectd to collect **system metrics** (CPU utilization, memory, hard disk, network etc.).

The Logstash server (detailed in the previous section) is able to collect metrics and logfiles directly from the machine it is installed on. That would mean to have a Logstash instance on each monitored node. In the DICE context, this is not feasible as Logstash has a substantial computational footprint especially when using specialized filters such as grok [17]. Because of this in DMon we use *logstash-forwarder* [23] which is designed to forward logs to one or more logstash server instances, eliminating node side processing of logs and the computational cost this processing would require.

At this point it is important to note that there are several alternatives to logstash-forwarder and even collectd. Most notably there are the Beats data shippers for Elasticsearch [6]. Although today this provides an interesting alternative, they were not available when implementation started on the current DMon prototype.

Since most of Big Data frameworks are Java tools, 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* [20] tool is used in our architecture to collect attributes exported at JVM level.

It's worth mentioning that both the core and node-level components of DMon may not be final. There is a lot of tools that could be used in the DICE context. For example, it is possible to use rsyslog [29] instead of Logstash to process and load data into Elasticsearch. There are alternatives to Elasticsearch as well, such as NoSQL databases that support handling of time series data, InfluxDB [19] being an emerging technology.

Of particular interest is the collection of JMX metrics using collectd via a plugin. Although there are some available collectd plugins [21] that are able to accomplish this task, they have proven to be either slow or very resource hungry in preliminary tests.

3.3. DMon Services

All components described in the previous sections have to be deployed and configured. This is accomplished by a number of Web services wrapping the core and the node-level components. This section details these services.

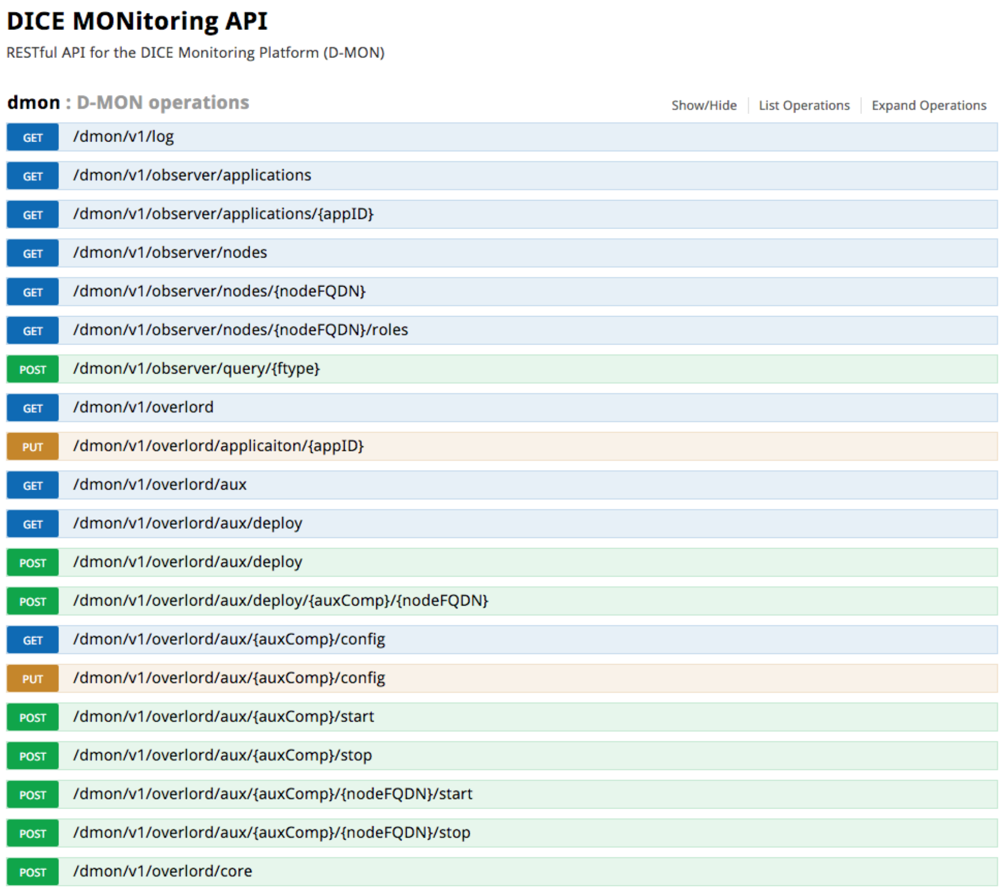
3.3.1. Core-level services

There are in total 3 core services: *dmon-controller, dmon-shipper* and *dmon-indexer*. All have been implemented using the Python programming language. Specifically with the Flask microframework [13]. The interface used by the services to communicate with each other takes the form of JSON encoded messages.

dmon-controller

The *dmon-controller* service is essentially the service with which all other components will communicate. It is in fact the main point of integration with the rest of the DICE solution. In particular, it will be used by all DICE components that require monitoring data. The REST API is split into two main parts: Monitoring Management API and Monitoring Query API. Figure 4 shows a swagger based web UI of the REST API defined for this service.

The **Monitoring Management API**, namely *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 key). If node-level components and services have already been deployed by other tools (such as WP5 delivery tools) they only have to register the already deployed node-level service endpoints. In this scenario credentials are not needed.



DICE MONitoring API

RESTful API for the DICE Monitoring Platform (D-MON)

dmon : D-MON operations

GET	/dmon/v1/log
GET	/dmon/v1/observer/applications
GET	/dmon/v1/observer/applications/{appID}
GET	/dmon/v1/observer/nodes
GET	/dmon/v1/observer/nodes/{nodeFQDN}
GET	/dmon/v1/observer/nodes/{nodeFQDN}/roles
POST	/dmon/v1/observer/query/{ftype}
GET	/dmon/v1/overlord
PUT	/dmon/v1/overlord/applicaiton/{appID}
GET	/dmon/v1/overlord/aux
GET	/dmon/v1/overlord/aux/deploy
POST	/dmon/v1/overlord/aux/deploy
POST	/dmon/v1/overlord/aux/deploy/{auxComp}/{nodeFQDN}
GET	/dmon/v1/overlord/aux/{auxComp}/config
PUT	/dmon/v1/overlord/aux/{auxComp}/config
POST	/dmon/v1/overlord/aux/{auxComp}/start
POST	/dmon/v1/overlord/aux/{auxComp}/stop
POST	/dmon/v1/overlord/aux/{auxComp}/{nodeFQDN}/start
POST	/dmon/v1/overlord/aux/{auxComp}/{nodeFQDN}/stop
POST	/dmon/v1/overlord/core

Figure 4 dmon-controller: Swagger interface for REST API

Long term storage of metrics is of course a problem that has to be dealt with in all data warehousing solutions. In the case of DMon one may use the management API to create new indexes which store monitoring data. We can also export these indexes or even dump the entire dataset into a different format. By default, DMon creates an index every 24 hours. These indexes can be queried either on at a time or all at once. The exported indexes can be at any given time be reloaded into DMon or into a different Elasticsearch deployment for truly offline processing.

Metrics version annotation is also supported by the dmon-controller. By this we mean that metrics pertaining to a specific application version can be annotated using tags. This way we can more easily query, aggregate or even compare metrics of the application. Creation of a separate index for each application version is also possible however, it is not as versatile a solution and makes comparison of different application version performance more difficult.

The dmon-controller is also responsible for generating and enacting configurations for all core components (Elasticsearch, Logstash and Kibana). The configuration is largely dependent on the data provided during the registration of each node. This data is then used to configure each component of DMon.

As already mentioned, the type of node-level component 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 which is then used to setup and manage node-level services and components.

Querying DMon is done using the **Monitoring Query API**, namely *Observer*. In contrast to the Management API, this one doesn't require authentication. A query example is included in Table 11, showing attributes that can be specified: the size of the returned response, its ordering, or start and stop dates (in UTC). The *queryString,* which follows the same format and rules as Kibana queries, actually defines the predicate to be run on the Elasticsearch and can be used to aggregate data, or perform additional operations on the stored data [15]. Query's response may be returned in several formats, currently supported are: CSV, JSON, or plain text. The next version of the DMon platform will also support RDF+XML encoding using OSLC Perf. Mon 2.0 vocabulary [27]. The CSV and RDF+XML query responses are generated using the dmon-controller service, which takes the JSON response from ElastiSearch and converts it to the target format.

Table 11 Example of query (JSON format)

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

dmon-shipper and dmon-indexer

The *dmon-shipper* microservice is meant to deploy, manage and configure logstash instances. In contrast to dmon-controller service, this service has to be located on the same machine with the controller logstash instance is located. This service is not meant to be used by external tools and services, being an internal component of DMon platform. The *dmon-indexer* microservice is used to control nodes comprising the Elasticsearch cluster.

Splitting the control of various core and node-level components into microservices we can easily separate the application logic of DMon from the code that actually drives and enacts them. Another important point is that all services besides the dmon-controller are essentially stateless. For example neither service stores the current state of the components it controls, rather it has to poll the status of the component. The dmon-controller stores some basic state and node-level information inside a relational database, which can be exported, imported, versioned or even backed up.

Queuing service

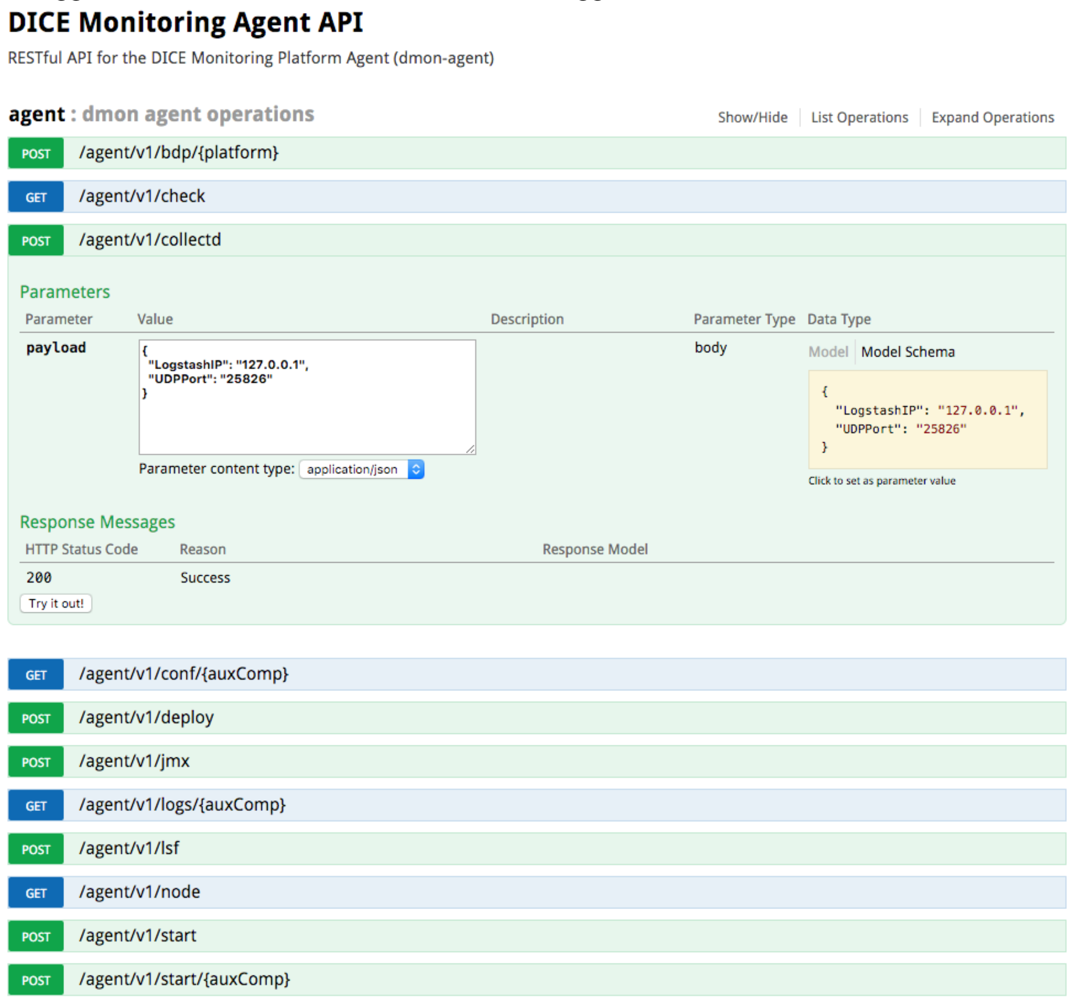
In some instances of DMon platform, metrics that are sent by the node-level components might exceed the capabilities of Logstash to process them effectively. This might lead to data loss. There are ways to mitigate this problem. First, we could increase the number of workers assigned to the filter plugin. In the Logstash documentation it is specified that some filters (specifically grok) might cause slowdown in metrics ingestion. Second, if increasing the number of workers is not an option we could create a second instance of Logstash which can handle some of the load.

The third variant is to use a queuing service that receives all metrics and from which the Logstash instances can consume data. Certainly, this will mitigate the data loss problem but could potentially increase the time it takes for a specific metrics reading to be processed and indexed inside Elasticsearch. This service is pictured in Figure 3, possible candidates for its implementation ranging from Kafka, or Redis to a combination of MongoDB [8] and RabbitMQ [7]. The full technical stack is still an open question and will be addressed in future versions of DMon.

3.3.2. Node-level services

dmon-agent

dmon-agent service is used to manage and configure all node-level components. Similarly to the dmon-shipper and dmon-indexer services, it is also stateless. The dmon-controller service issues request to each dmon-agent service with a JSON payload that contains all required information for controlling the node-level components. Each monitored node has to have a dmon-agent instance running on it. Figure 5 dmon-agent: Swagger interface for REST API shows the swagger UI of the REST API defined for this service.



DICE Monitoring Agent API

RESTful API for the DICE Monitoring Platform Agent (dmon-agent)

agent : dmon agent operations — Show/Hide | List Operations | Expand Operations

POST /agent/v1/bdp/{platform}

GET /agent/v1/check

POST /agent/v1/collectd

Parameters

Parameter	Value	Description	Parameter Type	Data Type
payload	{ "LogstashIP": "127.0.0.1", "UDPPort": "25826" }		body	Model | Model Schema: { "LogstashIP": "127.0.0.1", "UDPPort": "25826" }

Parameter content type: application/json

Click to set as parameter value

Response Messages

HTTP Status Code	Reason	Response Model
200	Success	

Try it out!

GET /agent/v1/conf/{auxComp}

POST /agent/v1/deploy

POST /agent/v1/jmx

GET /agent/v1/logs/{auxComp}

POST /agent/v1/lsf

GET /agent/v1/node

POST /agent/v1/start

POST /agent/v1/start/{auxComp}

Figure 5 dmon-agent: Swagger interface for REST API

As of writing this report, the dmon-agent supports collectd, logstash-forwarder and jmxtrans. It is able to install all of the supported components. The installation is based on the type of big data services and the roles assigned to the node where the dmon-agent service is installed. In short the *dmon-controller* is not responsible for picking what component each dmon-agent is deploying and managing, it only send the list of roles each node has. It is also able to interact with Hadoop, Spark and Storm deployments. This interaction is required to change or activate the metrics system for the supported big data services.

dmon-wui

DMon, as many monitoring solutions, will have a web user interface from which the current layout of the monitoring platform can be seen and managed. It will also have an overview of the metrics collected from the current Big Data deployment. This service is not scheduled for release in this prototype version of DMon, instead the dmon-controller service can be used to generate a dashboard in Kibana [15]. This dashboard contains systems metrics as well as some basic big data framework specific metrics. The type of metrics can vary based on the framework being monitored.

dmon-mas

The dmon-mas service will be in fact a multi-agent system that will enable automatic scaling and management of the entire DMon platform. It will monitor the current deployment of the monitoring solution and scale different components and service based on the current performance of the platform. This will aid in the overall performance and resiliency and add self-healing capabilities to the DMon platform.

There will be specialized agents that monitor the current deployment, to reason on the gather data and agents to enact the changes required by the performance analysis. Most likely there will be one or more specialized agents that will be able to provision VMs on a variety of cloud platforms (Flexiant Cloud Orchestrator (FCO), Amazon, OpenStack).

Most of the configuration and management task will be accomplished using the dmon-controller service. Because of this separation of roles the dmon-mas service is not crucial to the correct functioning of DMon. It will not be mandatory to start this service in order to use DMon.

3.4. Performance Metrics

In DICE most tools require a smaller subset of metrics that DMon can collect. One notable exception will be the Anomaly Detection Tool which, in order to detect contextual anomalies might require a significant variety of metrics. The precise number or type of metrics that are required will be discussed in the deliverables from Year 2 of the project.

Other tools such as the Enchantment and Configuration Optimization tools require a subset of the collected metrics. These metrics can be split up into 3 categories. The first category is represented by Resource-Level Metrics. They represent CPU, Memory and Network utilization at the VM level. Also in this category we can include a metric related to failure rates of specific VMs and the services running on them.

The second category is related to system level metrics. These metrics are metrics specific to each Big Data framework supported by DICE. These metrics include: Job arrival rate, job throughput, job parallelism, job response time, waiting buffer occupancy. 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. The third category focuses on worker level metrics focusing on task metrics.

As mentioned before DMon will be able to export some metrics using the OSLC Performance Monitoring v2.0 specification. This specification uses RDF+XML representation for metrics. There are several metrics categories such as: CPU, Disk, Memory, Network, Request (both Failure and Time), Resource Availability, Resource Usage, Thread pool and Virtualization metrics. All metrics can be represented in a tree structure.

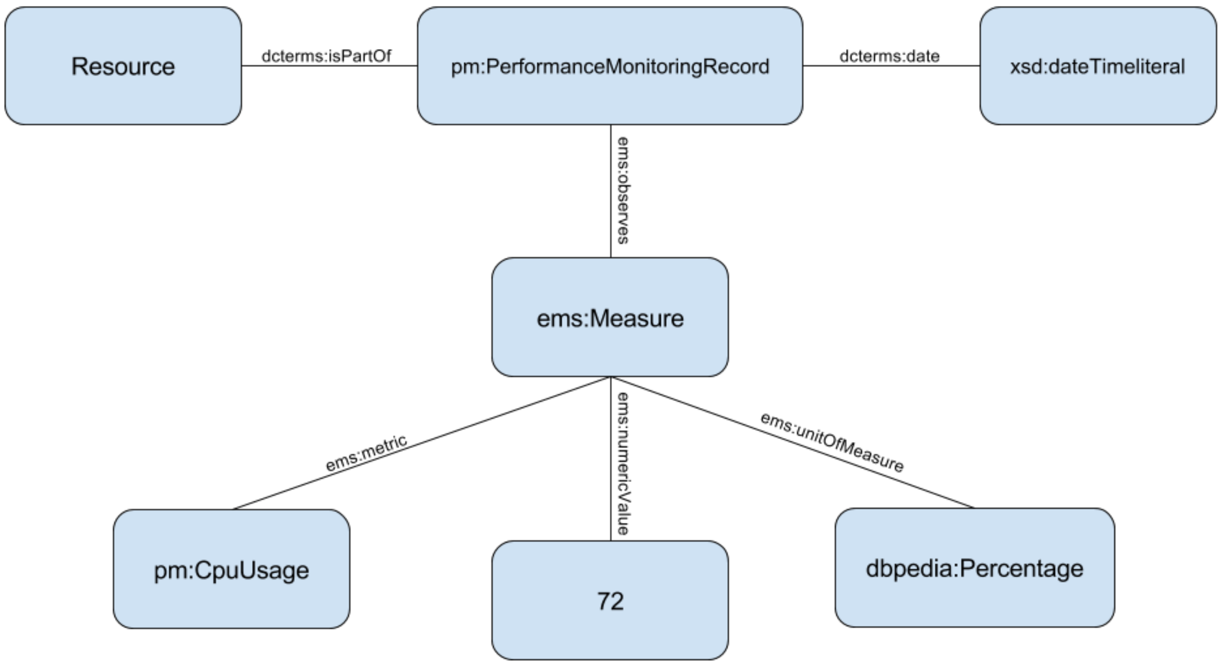


Figure 6 OSLC Perf. Monitoring v2.0 Diagram

Figure 6 represents an overview of this tree like structure. The problem we face in DICE is that some metrics, which are required by DICE tools, have no direct analogue in OSLC Perf. Mon v2.0. The most practical solution to this problem is to expand this specification (more precisely ems:metric from Figure 6) with the required metrics. This however is out of the scope of this deliverable and will be the subject of future updates of DMon.

4. Deployment and validation

During the period covered by this report, we have been focusing on the development of the platform, its deployment on Flexiant Cloud Orchestrator (FCO) and its validation against Big Data technologies available in Cloudera's Distribution for Hadoop (CDH 5.4.7) and Oryx2 [26].

Integration with other tools comprising DICE tool-chain will be defined and performed in the second year of the project.

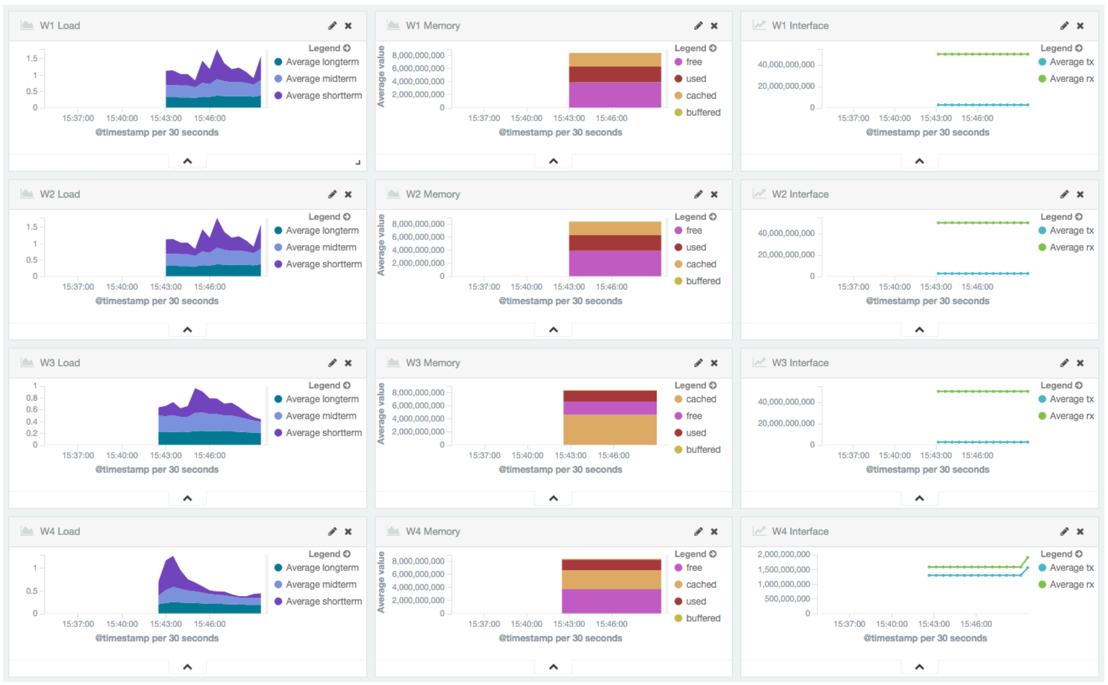
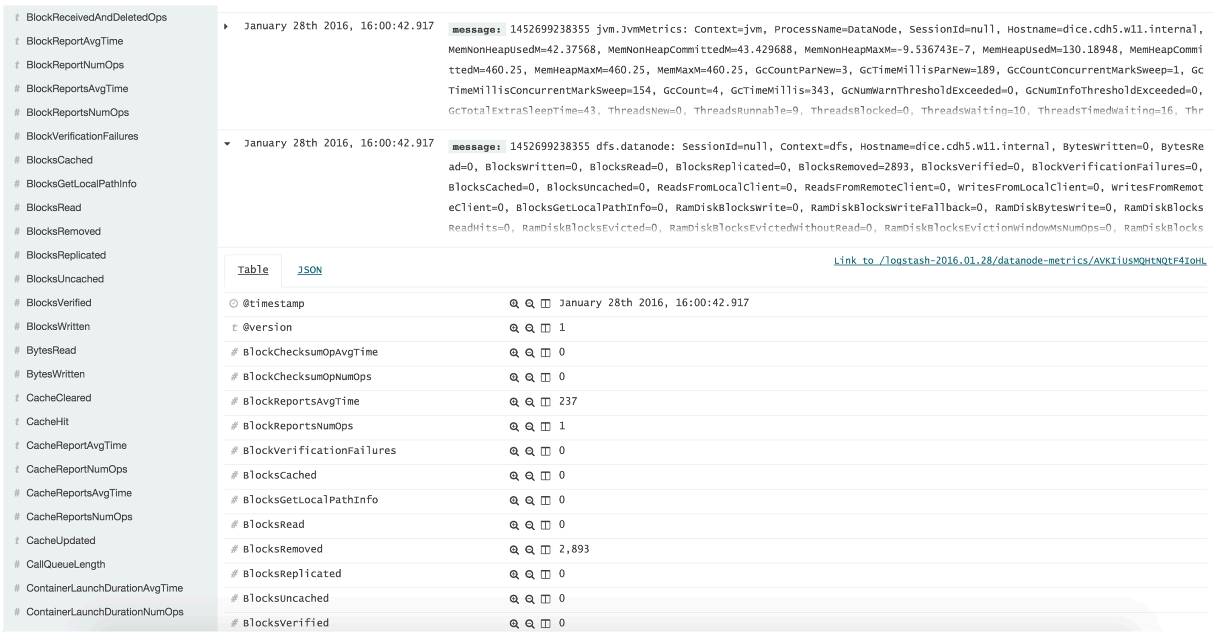


Figure 7 DMon Dashboard



```
t BlockReceivedAndDeletedOps
t BlockReportAvgTime
t BlockReportNumOps
# BlockReportsAvgTime
# BlockReportsNumOps
# BlockVerificationFailures
# BlocksCached
# BlocksGetLocalPathInfo
# BlocksRead
# BlocksRemoved
# BlocksReplicated
# BlocksUncached
# BlocksVerified
# BlocksWritten
# BytesRead
# BytesWritten
t CacheCleared
t CacheHit
t CacheReportAvgTime
t CacheReportNumOps
# CacheReportsAvgTime
# CacheReportsNumOps
t CacheUpdated
# CallQueueLength
# ContainerLaunchDurationAvgTime
# ContainerLaunchDurationNumOps
```

```
▸ January 28th 2016, 16:00:42.917   message: 1452699238355 jvm.JvmMetrics: Context=jvm, ProcessName=DataNode, SessionId=null, Hostname=dice.cdh5.w11.internal,
                                    MemNonHeapUsedM=42.37568, MemNonHeapCommittedM=43.429688, MemNonHeapMaxM=-9.536743E-7, MemHeapUsedM=130.18948, MemHeapCommi
                                    ttedM=460.25, MemHeapMaxM=460.25, MemMaxM=460.25, GcCountParNew=3, GcTimeMillisParNew=189, GcCountConcurrentMarkSweep=1, Gc
                                    TimeMillisConcurrentMarkSweep=154, GcCount=4, GcTimeMillis=343, GcNumWarnThresholdExceeded=0, GcNumInfoThresholdExceeded=0,
                                    GcTotalExtraSleepTime=43, ThreadsNew=0, ThreadsRunnable=9, ThreadsBlocked=0, ThreadsWaiting=10, ThreadsTimedWaiting=16, Thr

▾ January 28th 2016, 16:00:42.917   message: 1452699238355 dfs.datanode: SessionId=null, Context=dfs, Hostname=dice.cdh5.w11.internal, BytesWritten=0, BytesRe
                                    ad=0, BlocksWritten=0, BlocksRead=0, BlocksReplicated=0, BlocksRemoved=2893, BlocksVerified=0, BlockVerificationFailures=0,
                                    BlocksCached=0, BlocksUncached=0, ReadsFromLocalClient=0, ReadsFromRemoteClient=0, WritesFromLocalClient=0, WritesFromRemot
                                    eClient=0, BlocksGetLocalPathInfo=0, RamDiskBlocksWrite=0, RamDiskBlocksWriteFallback=0, RamDiskBytesWrite=0, RamDiskBlocks
                                    ReadHits=0, RamDiskBlocksEvicted=0, RamDiskBlocksEvictedWithoutRead=0, RamDiskBlocksEvictionWindowMsNumOps=0, RamDiskBlocks

  Table   JSON                                                           Link to /logstash-2016.01.28/datanode-metrics/AVKIiUsMQHtNQtF4IoHL

  @timestamp                          January 28th 2016, 16:00:42.917
t @version                            1
# BlockChecksumOpAvgTime              0
# BlockChecksumOpNumOps               0
# BlockReportsAvgTime                 237
# BlockReportsNumOps                  1
# BlockVerificationFailures           0
# BlocksCached                        0
# BlocksGetLocalPathInfo              0
# BlocksRead                          0
# BlocksRemoved                       2,893
# BlocksReplicated                    0
# BlocksUncached                      0
# BlocksVerified                      0
```

Figure 8 DMon Metrics Representation

The first release of DMon doesn't have bespoken metrics visualization facilities. Nevertheless we can use Kibana to create a dashboard for any number of metrics. In Figure 6 we can se an example of such a dashboard. It represents CPU, Memory and Network metrics of 4 VMs. These visualizations are based on ElasticSearch queries (y-axis), which are aggregated and plotted using a histogram (x-axis is represented by timestamps). Figure 7 shows an overview of how DMon stores the collected metrics. The before mentioned figure shows a small portion of metrics collected related to HDFS distributed data store.

4.1. Deployment on FCO

The DICE Monitoring platform has been deployed on one VM on Flexiant Cloud Orchestrator. The Table 12 details this installation.

Table 12 DMon deployment on FCO

Property	**Value**
Number of nodes / VMs	1
Total number of CPU-cores	4
Total amount of RAM in Gbytes	8 GB RAM / VM
Storage system/layer	250 GB
Operating System	Ubuntu 14.04.3
Available Services	* Elasticsearch * Logstash * Kibana * Marvel
Programmatic access details	The default ports for all services are available for programmatic access

In the second year of the project deployment using chef recipes will also be considered. These recipes enable the deployment of DMon using the Deployment Service developed in WP5.

4.2. DMon validation against Cloudera Distribution for Hadoop CDH 5.4.7 and Oryx2

In order to validate the platform against state-of-the-art Big Data technologies, we have deployed Cloudera Distribution for Hadoop (CDH) 5.4.7 and Oryx2 on a cluster of 14 nodes on FCO as well. The details of the cluster are provided in Table 13.

It is important to note that Oryx 2 is treated in this setup as a collection of different Big Data services not as a big data service. This means that Oryx2 metrics are comprised of metrics from HDFS, Yarn, Spark and Kafka (Kafka monitoring is still in early development). No Oryx2 specific metrics is monitored.

Table 13 CDH and Oryx2 deployment on FCO

Property	**Value**
Number of nodes / VMs	14
Total number of CPU-cores	56
Total amount of RAM in Gbytes	8 GB RAM / VM
Storage system/layer	2.92 TB
Operating System	Ubuntu 14.04.3
Available Services	* HDFS * YARN * Spark * Hive * Kafka (1 brokers) * Zookeeper (1 servers) * Hue * Oozie * Oryx2 - beta-2
Programmatic access details	The default ports for all services are available for programmatic access

Each time a new node is added to the DMon platform for monitoring, the platform automatically installs the monitoring agent on it. The monitoring agents collect the data from the local files and sends the data to logstash server in the DMon. Currently, logs produced by the following technologies are collected: Apache HDFS [3], Apache YARN [4], and Apache Spark [5]. System metrics (CPU, memory, network) are also collected using collectd plugins.

4.3. Deployment using Vagrant

For development purposes, Vagrant [28] deployment scripts for DMon, CDH and Storm were also created. The first script installs and configure a distribution of DMon where all core components and services are collocated on the same VM.

The second vagrant script provisions 4 VMs on which it installs the newest version of the CDH together with a version of Oryx 2 toy application. It is important to note that some additional manual configuration steps are still required. Namely the Big Data service distribution on the 4 Nodes. Lastly, we have a vagrant script that provision 4 VMs that are used to create a Storm Cluster. No additional setup steps are required. Once the installation is complete the Storm deployment can be used.

The usage of all Vagrant scripts documentation can be found in the Github repository (https://github.com/dice-project/DICE-Monitoring/blob/master/src/). Using these scripts anybody can create a standard development/demo environment, which contains not only the latest version of DMon but also a Cloudera deployment on which to test it out. By default, all VMs are provisioned with 2 CPUs, 4 GB RAM and a HDD of 50 GB. These values can be adjusted in order to accommodate less powerful computers.

5. Documentation

Further details about the platform are available online on DICE Monitoring Platform's Github repository (https://github.com/dice-project/DICE-Monitoring/blob/master/src/README.md), quick links being provided below:

1. Installation / deployment guide: https://github.com/dice-project/DICE-Monitoring/blob/master/src/README.md#installation
2. REST API: https://github.com/dice-project/DICE-Monitoring/blob/master/src/README.md#rest-api-structure

A version of the documentation can also be found at the end of this deliverable as Appendix A (REST API structure) and Appendix B (Installation). It is important to note that these appendices refer to v0.1.4-alpha of DMon. The overall REST API structure may suffer changes and additions. All changes will be detailed inside the Changelog from the official DICE github repository.

Versioning for DMon will use a 3-digit schema. The first digit represents the current major beta version of DMon. The second digit represents the current major alpha build of the platform while the last digit represents minor platform build.

The first major beta version of DMOn will be released once all required features are implemented at the end of the project. Until then only the major alpha build digit will be incremented.

6. Conclusion and future plans

This document presents the initial version of DICE Monitoring platform, which is a distributed, highly available platform for monitoring Big Data technologies. The goal of the first prototype of the platform (at M12) is to demonstrate a working Proof-of-Concept that collects, stores and processes monitoring data from state-of-the-art Big Data technologies. DMon is integrating monitoring data from a number of Big Data technologies, the first prototype of the platform supporting Apache HDFS, Apache YARN and Apache Spark. Engineered using a microservices architecture, the platform is easy to deploy, and operate, on heterogeneous distributed Cloud environments. We reported successful deployment on Flexiant Cloud Orchestrator using Vagrant scripts.

The development plan of the platform's next releases include features and extensions:

- Queuing service
- Finalize the querying options, such as aggregates
- Finalize the metrics visualization user interface, providing a customizable dashboard
- Metrics specification on application model
- Data cleaning
- Cover additional Big Data technologies
- Integrate with DICE Deployment tool (Cloudify and Chef based)

References

[1] Aceto, G.; Botta, A.; De Donato, W. & Pescapè, A. Survey Cloud Monitoring: A Survey *Comput. Netw., Elsevier North-Holland, Inc.,* **2013**, *57*, 2093-2115

[2] Alhamazani, K.; Ranjan, R.; Mitra, K.; Rabhi, F.; Jayaraman, P. P.; Khan, S. U.; Guabtni, A. & Bhatnagar, V. An Overview of the Commercial Cloud Monitoring Tools: Research Dimensions, Design Issues, and State-of-the-art *Computing, Springer-Verlag New York, Inc.,* **2015**, *97*, 357-377

[3] Apache HDFS Documentation, January 2016 - https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[4] Apache YARN Documentation, January 2016 -

https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[5] Apache Spark Documentation, January 2016 -

http://spark.apache.org/docs/latest/

[6] Beats Platform, January 2016 - https://www.elastic.co/guide/en/beats/libbeat/1.0.1/index.html

[7] Boschi S. and Santomaggio G., RabbitMQ Cookbook. Packt Publishing, 2013.

[8] Chodorow, Kristina and Dirolf, Michael. MongoDB - The Definitive Guide: Powerful and Scalable Data Storage.. : O'Reilly, 2010.

[9] Cloudera Quickstart VM, January 2016 - http://www.cloudera.com/content/www/en-us/downloads/quickstart_vms/5-5.html

[10] Collectd Documentation, January 2016 - https://collectd.org/documentation.shtml

[11] DICE Consortium, D1.2 Requirements specification, 2015. http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2015/08/D1.2_Requirement-specification.pdf

[12] DICE Consortium, D1.2 Requirements specification companion, 2015.

http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2015/08/D1.2_Requirement-specification_Companion.pdf

[13] Flask Framework, January 2016, http://13.pocoo.org/

[14] Fowler, M., Microservice overview, January 2016, http://martinfowler.com/articles/microservices.html

[15] Gormley, Clinton, and Zachary Tong. *Elasticsearch: The Definitive Guide*, ISBN 1449358543

[16] Gospodnetić, Otis, and Erik Hatcher. Lucene in Action. Greenwich, CT: Manning Publications, 2005.

[17] Grok filter, January 2016 - https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html

[18] Hortownworks Sandbox, January 2016 - http://hortonworks.com/products/hortonworks-sandbox/#install

[19] InfluxDB Documentation, January 2016 - https://influxdb.com/docs/v0.9/concepts/key_concepts.html

[20] jmxtrans Documentation, January 2016 - https://github.com/jmxtrans/jmxtrans/wiki

[21] JMX plugin for collectd, January 2016, https://collectd.org/wiki/index.php/Plugin:GenericJMX

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

[23] Logstash-forwarder, January 2016 - https://github.com/elastic/logstash-forwarder

[24] Marz N. and Warren J., Big Data: Principles and Best Practices of Scalable Realtime Data Systems (1st ed.), 2015, Manning Publications Co., Greenwich, CT, USA.

[25] Newman, S., 2015. Building microservices : Designing fine-grained Systems. http://www.worldcat.org/isbn/9781491950357

[26] Oryx 2 Documentation, January 2016 - http://oryx.io/docs/endusers.html

[27] OSLC Perf. Mon. v2.0, January 2016 - http://open-services.net/wiki/performance-monitoring/OSLC-Performance-Monitoring-Specification-Version-2.0/

[28] Michael Peacock, Creating Development Environments with Vagrant, Packt Publishing, 2013.

[29] rsyslog Documentation, January 2016 - http://www.rsyslog.com/doc/v8-stable/

[30] Stubbs, J.; Moreira, W.; Dooley, R., "Distributed Systems of Microservices Using Docker and Serfnode," in Science Gateways (IWSG), 2015 7th International Workshop on , vol., no., pp.34-39, 3-5 June 2015 doi: 10.1109/IWSG.2015.16

[31] Turnbull, J. The Logstash Book., ISBN 0988820226, dec. 2014

Appendix A. REST API Structure

There are two main components from this API:

- First we have the management and deployment/provisioning component called Overlord (Monitoring Management API).
    - It is responsible for the deployment and management of the Monitoring Core components: ElasticSearch, Logstash Server and Kibana.
    - 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 Data Warehouse represented by ElasticSearch. This component is called Observer.
    - It is responsible for the returning of monitoring metrics in the form of: CSV, JSON, simple output.

NOTE: Future versions will include authentication for the Overlord resources. This is a preliminary structure of the REST API. It may be subject to changes!

Overlord (Monitoring Management API)

The Overlord is structured into two components:

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

Monitoring Core Resources

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 status information of various sub components.

GET /v1/overlord

Returns information regarding the current version of the Monitoring Platform.

GET /v1/overlord/framework

Returns the currently supported frameworks. Currently only Apache Yarn, HDFS and Spark are supported.

```
{
  Supported Frameworks:[<list_of_frameoworks>]
}
```

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 DMON and creates a unique tag for the monitored data. The tag is defined by appID..

NOTE: Scheduled for future versions!

POST /v1/overlord/core

Deploys all monitoring core components provided they have values pre-set 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 Dmon 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 dmon. It will replace the current states with new states. The old states are backed up before applying the new ones. Database should take the form of sqlite3 database file and sent unsing the application/x-sqlite3 mimetype.

GET /v1/overlord/core/status

Returns the current status of the Monitoring platform status.

```
{
  "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 Cluster, Logstash server and Kibana status information.

GET /v1/overlord/core/chef

Returns the status of the chef-client of the monitoring core services.

FUTURE WORK: This feature will be developed for future versions. A chef server is deployed as part of Deployment Service (WP5). Because of this this resource is at this point superfluous. It could be cut in future versions.

GET /v1/overlord/nodes/chef

Returns the status of the chef-clients from all monitored nodes.

FUTURE WORK: This feature will be developed for future versions. Same situation as before.

GET /v1/overlord/nodes

Returns the current monitored nodes list. It is the same as /v1/observer/chef.

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

PUT /v1/overlord/nodes

Includes the given nodes into the monitored node pools. In essence 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 start 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: Duplicate from ../aux/.. branch

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.

POST /v1/overlord/nodes/roles

Generates metrics configuration files for each role assigned to a node and uploads it 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 installtions are not yet supported. As yarn and HDFS have the same metrics system their tags (i.e. hdfs and yarn) are interchangable 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_Systen>",
     "key":"<keyName|null>",
     "username":"<uname|null>",
     "password":"<pass|null>",
    "chefclient":"<True|False>",
    "CDH":"<active|inactive|unknown>",
    "CDHVersion":"<version>",
    "Roles":"[listofroles]"
}
```

FUTURE Work: A more fine grained node status will be implemented. Currently it is boolean - online/offline. The last three elements are not implemented. These are scheduled for future versions.

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_Systen>",
  "key":"<keyName|null>",
  "username":"<uname|null>",
  "password":"<pass|null>"
}
```

POST /v1/overlord/nodes/{NodeFQDN}

Bootstraps specified node.

DELETE /v1/overlord/nodes/{nodeFQDN}

Stops all auxiliary monitoring components associated with a particular node.

NOTE: This does not delete the nodes nor the configurations it simply stops collectd and logstash-forwarder on the selected nodes.

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 the given node. It 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.

```
{
  "ES Instances": [{
    "ESClusterName": "<clustername>",
    "HostFQDN": "<HostFQDN>",
    "IP": "<Host IP>",
    "NodeName": "<NodeName>",
    "NodePort": "<IP:int>",
    "OS": "<host OS>",
    "PID": "<ES component PID>",
    "Status": "<ES Status>",
    "Master":"<true|false>"
  }, .................. ]}
```

POST /v1/overlord/core/es

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

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

FUTURE Work: This process needs more streamlining. It is recommended to use only local deployments for this version.

GET /v1/overlord/core/es/config

Returns the current configuration file of 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 of ElasticSearch.

Input:

```
{
  "HostFQDN":"<nodeFQDN>",
  "IP":"<NodeIP>",
  "OS":"<Operating_Systen>",
  "NodeName":"<ES host name>",
  "NodePort":"<ES host port>",
  "ClusterName":"<ES cluster name>"
}
```

NOTE: The new configuration will not be generated at this step.

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

Stops the ElasticSearch instance on a given host and removes all configuration data from DMon.

GET /v1/overlord/core/ls

Returns the current status of all logstash server instances registered with D-Mon.

Response:

```
{
   "LS Instances":[
     {
         "ESClusterName":"<name>",
         "HostFQDN":"<Host FQDN>",
         "IP":"<Host IP>",
         "LPort":"<port>",
         "OS":"<Operating_System>",
         "Status":"<status>",
         "udpPort":"<UDP Collectd port>"


     },
     …………
   ]
}
```

POST /v1/overlord/core/ls

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

FUTURE Work: Better support for distributed deployment of logstash core service instances.

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:

```
{
  "HostFQDN":"<Host FQDN>",
  "IP":"<Host IP>",
  "OS":"<Operating_Systen>",
  "LPort":"<Lumberjack Port>",
  "udpPort":"<UDP Collectd port>",
  "ESClusterName":"<ES cluster Name>"
}
```

FUTURE Work: Only for local deployment of logstash server core service. Future versions will include distributed deployment.

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]",
}
```

Note: 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>
}
```

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.

Response:

```
{
 "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 who are note 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.

Response:

```
{
   {
      "NodeFQDN":"<nodeFQDN>",
      "NodeIP":"<nodeIP>",
      "Monit":"<True|False>",
      "Collectd":"<status>",
      "LSF":"<status>"
   },
   ............................
}
```

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 an it is currently running
- **Stopped** -> There is the aux component on the registered node and it is currently stopped

If the status is None than 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.

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 than the defined component (collectd or lsf) will be installed.

FUTURE Work: Currently configurations of both collectd and logstash-forwarder are global and can't be changed on a node by node basis.

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

Returns the current collectd or logstashfw configuration file.

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

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

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

Starts the specified auxiliary component on all nodes.

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

Stops the specified auxiliary components on all nodes.

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

Starts the specified auxiliary component on a specific node.

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

Stops the specified auxiliary component on a specific node.

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.

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.

Observer (Monitoring Query API)

GET /v1/observer/applications

Returns a list of all YARN applications/jobs on the current monitored big data cluster.

NOTE: Scheduled for future release.

GET /v1/observer/applications/{appID}

Returns information on a particular YARN application identified by {appID}. The information will not contain monitoring data only a general overview. Similar to YARN History Server.

NOTE: Scheduled for future release.

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 UP dynamically at VM startup. Because of this D-Mon treats the FQDN as a form of UUID. In future versions this might change, the FQDN being replaced/augmented with a hash.

Response:

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

GET /v1/observer/nodes/{nodeFQDN}

Returns information of a particular monitored node. No information is limited to non confidential information, no authentication credentials are returned.

Response:

```
{
  "<nodeFQDN>":{
   "Status":"<online|offline>",
   "IP":"<NodeIP>",
   "Monitored":"<true|false>",
   "OS":"Operating_Systen"
  }
}
```

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

Returns roles the node identified by 'nodeFQDN'.

Response:

```
{
   'Roles':['yarn','spark','storm']
}
```

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.

NOTICE

The REST API detailed in this appendix is only for v0.1.4-alpha of the DICE Monitoring Platform. Because this is only a proof of concept build changes frequent chances and/or updates are to be expected.

Up to date REST API specification can be found at the official DMon github page[1]. Also on github there is the repository change log where all changes between versions are logged.

[1] https://github.com/dice-project/DICE-Monitoring

[2] https://github.com/igabriel85/IeAT-DICE-Repository/tree/master/Vagrant CDH Cluster

Appendix B. Installation

In this version of DMon, installation is largely based on bash scripts. Future versions will most likely be based on chef recipes and/or deb or rpm packages. There are 2 types of installation procedures currently supported.

Cloud

This type of installation is for client/cloud deployment. It will install all python modules as well as the ELK (ElasticSearch, Logstash and Kibana 4) stack. Only local deployment is currently supported.

Download the installation script to the desired host and make it executable

```
wget https://github.com/igabriel85/IeAT-DICE-Repository/releases/download/v0.1-install/install-dmon.sh
&& sudo chmod +x install-dmon.sh
```

After which execute the installation script

```
sudo ./install-dmon.sh
```

Note: This script will clone the DMon repository into /opt and change the owner of this directory to ubuntu.ubuntu!

Next go inside the cloned repository and run

```
sudo ./dmon-start.sh -i -p 5001
```

The '-i' flag will install all Core components of the monitoring platform (i.e. ELK) as well as setting the appropriate permissions for all folders and files. The '-p' flag designates the port on which DMon will be deployed.

In order deploy D-Mon locally execute:

```
./dmon-start.sh -l -p 5001
```

The '-l' flag signals the service that this is a local deployment of both ElasticSearch and Logstash server. The service will start logging into stdout.

Note: Do not execute this command as root! It will corrupt the previously set permissions and the service will be inoperable.

If you do not wish to create a local deployment run the command.

```
./dmon-start.sh -p 5001
```

This will only start the service and not load the local deployment module.

Note: By default all the IP is set to 0.0.0.0. This can be change using the '-e' flag.

Observation: Kibana 4 service is started during the bootstrapping process. You can check the service by running:

```
sudo service kibana4 status
```

For starting stoping the service replace the status command with start, stop or restart.

Vagrant

There are two vagrant files in this repository. The first[2] one creates a deployment of 4 VM on which it automatically installs the Cloudera Manager suite.

The second[3] script installs DMon as well as the ELK stack, essentially taking the place of the '-i' flag in the above mentioned instructions. The procedure for creating a local deployment of D-Mon is the same as before.

Note: These vagrant scripts should only be used for development or as a small demonstration. They are not meant for an exploitable deployment of the monitoring solution.

[2]https://github.com/igabriel85/IeAT-DICE-Repository/tree/master/Vagrant CDH Cluster
[3]https://github.com/igabriel85/IeAT-DICE-Repository/tree/master/Monitoring