## Deliverable 3.1 Assessment on Application of Generic Data Management Technologies I

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Due Date of Deliverable</td>
<td>M4, 31/04/2015 (officially shifted to M5)</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>M5, 30/05/2015</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP3, Assessment on Application of Generic Data Management Technologies</td>
</tr>
<tr>
<td>Task</td>
<td>T3.1</td>
</tr>
<tr>
<td>Type</td>
<td>Report</td>
</tr>
<tr>
<td>Approval Status</td>
<td>Approved</td>
</tr>
<tr>
<td>Version</td>
<td>V1.0</td>
</tr>
<tr>
<td>Number of Pages</td>
<td>40</td>
</tr>
<tr>
<td>Filename</td>
<td>D3.1-final.pdf</td>
</tr>
</tbody>
</table>

**Abstract:** This document defines the vision of the Big Data Europe platform which will be used to tackle pilot cases in the societal challenges. Most importantly, the document provides an overview of the architectural building blocks of the Big Data Europe platform.

The information in this document reflects only the author’s views and the European Community is not liable for any use that may be made of the information contained therein. The information in this document is provided “as is” without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/her sole risk and liability.
**History**

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
<th>Revised by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>01/04/2015</td>
<td>Initial version</td>
<td>Bert Van Nuffelen (Tenforce)</td>
</tr>
<tr>
<td>0.1</td>
<td>05/05/2015</td>
<td>First version</td>
<td>Paul Massey, Erika Pauwels, Bert Van Nuffelen (Tenforce)</td>
</tr>
<tr>
<td>0.2</td>
<td>05/05/2015</td>
<td>Intermediate version</td>
<td>Jürgen Jakobitsch (Tenforce), Jens Lehmann (InfAI), Mohamed Mami (FHG), Axel Ngonga (InfAI)</td>
</tr>
<tr>
<td>0.3</td>
<td>22/05/2015</td>
<td>Peer review</td>
<td>Phil Archer, Rigo Wenning (ERCIM)</td>
</tr>
<tr>
<td>0.4</td>
<td>26/05/2015</td>
<td>Address peer review comments</td>
<td>Paul Massey, Erika Pauwels, Bert Van Nuffelen (Tenforce)</td>
</tr>
<tr>
<td>0.5</td>
<td>29/05/2015</td>
<td>Final version</td>
<td>Paul Massey, Erika Pauwels, Bert Van Nuffelen (Tenforce)</td>
</tr>
</tbody>
</table>

**Author List**

<table>
<thead>
<tr>
<th>Organisation</th>
<th>Name</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>TenForce</td>
<td>Paul Massey</td>
<td><a href="mailto:paul.massey@tenforce.com">paul.massey@tenforce.com</a></td>
</tr>
<tr>
<td>TenForce</td>
<td>Erika Pauwels</td>
<td><a href="mailto:erika.pauwels@tenforce.com">erika.pauwels@tenforce.com</a></td>
</tr>
<tr>
<td>TenForce</td>
<td>Bert Van Nuffelen</td>
<td><a href="mailto:bert.van.nuffelen@tenforce.com">bert.van.nuffelen@tenforce.com</a></td>
</tr>
<tr>
<td>ERCIM</td>
<td>Phil Archer</td>
<td><a href="mailto:phila@w3.org">phila@w3.org</a></td>
</tr>
<tr>
<td>ERCIM</td>
<td>Rigo Wenning</td>
<td><a href="mailto:rigo@w3.org">rigo@w3.org</a></td>
</tr>
<tr>
<td>INFAI</td>
<td>Jens Lehmann</td>
<td><a href="mailto:lehmann@informatik.uni-leipzig.de">lehmann@informatik.uni-leipzig.de</a></td>
</tr>
<tr>
<td>SWC</td>
<td>Jürgen Jakobitsch</td>
<td><a href="mailto:j.jakobitsch@semantic-web.at">j.jakobitsch@semantic-web.at</a></td>
</tr>
<tr>
<td>INFAI</td>
<td>Axel Ngonga</td>
<td><a href="mailto:ngonga@informatik.uni-leipzig.de">ngonga@informatik.uni-leipzig.de</a></td>
</tr>
<tr>
<td>FH</td>
<td>Mohamed Mami</td>
<td><a href="mailto:mohamed.nadjib.mami@gmail.com">mohamed.nadjib.mami@gmail.com</a></td>
</tr>
</tbody>
</table>
Executive Summary

Since the start of Computer Science the maintenance of knowledge in information systems has been a key aspect. Over the past 50 years knowledge management has transformed from maintaining data for the well-functioning of a single application to the management of data serving many applications and serving even more users. Data being served has reached volumes beyond human imagination and is being collected at an impressive rate. All these aspects are commonly denoted as the triple-V data aspects (volume, velocity and variety), concisely describing the major challenges in data management. Despite commonly called Big Data, the term Big Data denotes for our project data management in all its aspects, so not solely the volume aspect.

In order to tackle the Big Data challenges, technology and experience from recent and less recent past are brought together. In the recent past, a major technology leap has been taken into processing large data volumes and data that has been produced at high rate. In a slightly more distant past, the challenge of disseminating and sharing data in an unambiguous and semantic annotated way has been tackled by Linked Data. Today these technologies (and many more) create a heterogeneous and complex web of data which is full of opportunities.

This deliverable describes the approach to setting up the Big Data Europe data management platform. The platform will be established based on existing open-source Big Data technologies. The criteria used to select components for the platform will be outlined. During the project the selected components will be (minorly) enhanced and preconfigured to increase their interplay. Since one of the project ambitions is to provide an easy rollout of the software, the chosen deployment strategy is described. Based on initial feedback by our societal challenge use case partners an indicative roadmap for the functionalities of the platform is presented.
## Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMQP</td>
<td>Advanced Message Queuing Protocol</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BDE</td>
<td>Big Data Europe</td>
</tr>
<tr>
<td>CLI</td>
<td>Command-Line Interface</td>
</tr>
<tr>
<td>DEB</td>
<td>Debian Package Manager</td>
</tr>
<tr>
<td>ETL</td>
<td>Extract-Transform-Load</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LOD</td>
<td>Linked Open Data</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>QA</td>
<td>Quality Assurance</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relation Database Management System</td>
</tr>
<tr>
<td>RDD</td>
<td>Resilient Distributed Dataset</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RPM</td>
<td>Redhat Package Manager</td>
</tr>
<tr>
<td>SPARQL</td>
<td>Sparql Protocol and RDF Query Language</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
# Table of Contents

1. **INTRODUCTION** ......................................................................................... 8  
   1.1 INITIAL BDE PLATFORM ARCHITECTURE ........................................... 9  
   1.2 LAMBDA ARCHITECTURE ................................................................... 9  
   1.3 INITIAL BDE ARCHITECTURE PATTERN ........................................... 10  
   1.4 SEMANTIFICATION OF THE ARCHITECTURE .................................... 12  
   1.5 ARCHITECTURAL CONCLUSION ....................................................... 13  

2. **BDE ARCHITECTURAL COMPONENTS** .................................................. 14  
   2.1 COMPONENT SELECTION CRITERIA .................................................. 14  
   2.2 COMPONENT CLASSIFICATION ......................................................... 16  
     2.2.1 Data Acquisition ......................................................................... 16  
     2.2.2 Message Passing .......................................................................... 17  
     2.2.3 Data Processing ............................................................................ 19  
     2.2.4 Data Storage .................................................................................. 21  
     2.2.5 Data Visualization ....................................................................... 24  
     2.2.6 Monitoring .................................................................................... 24  
   2.3 FRAMEWORK SELECTION AND COMPONENT STATUS ...................... 24  
     2.3.1 Framework Component Summary ................................................. 25  
     2.3.2 Framework OS Availability Status ............................................... 26  
     2.3.3 BDE Semantic Components ......................................................... 27  
   2.4 CONCLUSIONS AND FRAMEWORK CHosen ...................................... 28  
     2.4.1 Following up the Platform Integration Status .................................. 28  

3. **DEPLOYMENT STRATEGY** ....................................................................... 29  
   3.1 BIG DATA DEPLOYMENT .................................................................... 29  
   3.2 VIRTUAL DEPLOYMENT ..................................................................... 29  
     3.2.1 Docker Engine ............................................................................. 30  
     3.2.2 Docker Registries ......................................................................... 32  
     3.2.3 Docker Compose ........................................................................... 32  
     3.2.4 Docker Swarm ............................................................................. 32  
   3.3 CLUSTER MANAGEMENT ..................................................................... 32  
     3.3.1 Definition ...................................................................................... 32  
     3.3.2 Apache Mesos ............................................................................... 33  
   3.4 DEPLOYMENT CONCLUSIONS .......................................................... 35  

4. **CONCLUSION** ......................................................................................... 36  

REFERENCES .................................................................................................. 37
APPENDICES .............................................................................................................38

APPENDIX 1: TERMINOLOGY ..................................................................................38
  Architectural Design Pattern .................................................................................38
  Lambda Architecture Design Pattern .................................................................38
  BDE Architecture Design Pattern ......................................................................38
  BDE Framework .....................................................................................................38
  BDE Platform .........................................................................................................38
  BDE Application .....................................................................................................38

APPENDIX 2: BDE ANTICIPATED USERS/ROLES ..................................................39
  Component Developers ........................................................................................39
  Application Developers ........................................................................................39
  System Deployer/QA/Testers ...............................................................................39
  Administrators/Operators .....................................................................................39
  Application Users ..................................................................................................40
List of Figures

Figure 1: Lambda Architecture ................................................................. 10
Figure 2: High-level data flow in the BDE architecture ................................ 11
Figure 3: Overview of tackled areas ...................................................... 12
Figure 4: Hortonworks view of Challenge ............................................. 16
Figure 5: Message broker architecture .................................................. 18
Figure 6: A Storm topology consisting of spouts and bolts ....................... 20
Figure 7: HDFS file storage .................................................................. 22
Figure 8: Docker containers vs. traditional virtual machines ...................... 30
Figure 9: Layers in a Docker Image ........................................................ 31
Figure 10: Cluster configuration using Mesos as Kernel ............................. 33
Figure 11: High-level flow of resource offering in Mesos .......................... 35
Figure 12: BDE Deployment Proposal ................................................... 36

List of Tables

Table 1: High level view of NoSQL database categories’ main characteristics ...... 23
Table 2: Framework/BDE Component Versions ........................................ 26
Table 3: Framework OS Availability Status .............................................. 26
1. Introduction

BDE aims at supporting the data community by providing a data management platform. Within the project this platform is used to create pilot instances demonstrating how data management can contribute to resolve and support the societal challenges the European Commission has put forward as the main objectives for 2020. Data management has been recognized as a cross-domain technology. A proper usage of data is a prerequisite in all cases in order to achieve the European wide impact of the solutions to the societal challenges put forward.

Recently, data technology has got an enormous boost: calculation power and processing technology has become a common good which is easily reachable for almost any organization. Within the BDE project the ambition is to consolidate the data technologies and make it easier to start with these technologies. Additionally, the consortium wants to collect and integrate their experience with semantic annotated data and Linked Data with the processing power of the Big Data tool stack. By bringing data technologies together, the BDE platform is able to offer support for a wide variety of data challenges described in the 7 Societal Challenges of Horizon 2020 (European Commision, 2015).

In contrast to what most people might expect, Big Data covers more aspects than just handling huge amounts of data. A Big Data problem involves one or more of the following characteristics (Big Data Technical Working Groups, 2014):

- **Volume** - The quantity of data.
- **Velocity** - The speed/rate at which data is being provided.
- **Variety** - The different formats and semantic models in which data is provided.
- **Variability** - The variation in data being provided. High variation typically corresponds to a data model with low schema guarantees. No variation means a fixed schema with an almost unique semantic interpretation.
- **Veracity** - The accuracy/truthfulness, and therefore usefulness, of the data.
- **Complexity** - Coverage of the information being provided. The information can cover a whole domain with many different entities and relations between them (e.g. the pharmaceutical domain) or it can just cover a single piece of information (e.g. the status of a traffic light).

Because the project has the ambition to support a wide area of applications, the BDE platform addresses all these characteristics. However instead of building a new data management platform from scratch, the project has decided to reuse and integrate today’s best practices and open-source data technologies. The platform will provide the necessary processing capabilities, storage potential, analytical power, visualizations, end-user data dissemination and data interoperability for all pilots.

This deliverable, which is the first of the technical deliverables, sets the foundations for the BDE platform. It reflects today's status and decisions. In particular the deliverable will present the high level design of the BDE platform, followed by the component selection process and deployment strategy. In subsequent deliverables the different aspects of the platform will be further elaborated and refined.
1.1 Initial BDE Platform Architecture

Building a (software) application requires taking into account many functional and non-functional requirements. One of the core functional requirements is how data that will drive the application is being managed, processed and produced. The presented (initial version of the) BDE platform forms in the first place a blueprint for this data processing aspect in a whole application. So when designing an application as intended in the future with the pilots for each societal challenge, the BDE platform provides a generic data processing setup which must be complemented with the functional and non-functional needs of the targeted application.

In this section, a first version of the data processing architecture pattern is presented, starting with a short introduction to the basic architectural pattern in Big Data - the Lambda architecture. This is followed by a description of the BDE architectural pattern which can be considered an extended variant of the Lambda architectural pattern. The requirements for the extensions are based on the first assessment(s) with the Societal Challenge business representatives where the need for Linked Data (and most importantly the need for the associated semantics the format carries with) has been identified. This section concludes with the outlook of the main technical extension foreseen in the future: the semantification of the Lambda architecture.

1.2 Lambda Architecture

The most prominent Big Data software architecture pattern is called the Lambda architecture (See Figure 1: Lambda Architecture). It was Nathan Marz (Marz, et al., 2015) who came up with this term to denote generic, scalable and fault-tolerant data processing architecture, based on his experience working on distributed data processing systems at Backtype and Twitter. The architecture is usually split into three aspects:

- Batch layer - This layer covers data processing which is time-consuming for which data is physically available. Data processing is expected to change very slowly. It is expected that data is provided in large chunks (files).
- Speed layer - This layer covers data processing whose outcome is expected to be available immediately. It typically works on smaller amount of data provided as a data stream.
- Data serving layer - This layer covers the software responsible for provisioning of data input and data consumption. It needs to be noted that the views, which are the result of data processing, are predefined queries that are required for the application to work. Extending the application with functionality that requires a different query impacts the implementation of the batch and speed layer.
The Lambda architecture consisting of three layers: (1) batch layer, (2) speed layer, (3) data serving layer

This architectural pattern will form the basis of the BDE architectural pattern. In section Component Classification, the typical open-source software components used to instantiate the Lambda architecture are presented.

1.3 Initial BDE Architecture Pattern

As mentioned previously, the BDE project aims to support a wide variety of data challenges (i.e. the Variety) and the requirement for Linked Data has been identified in the initial Pilot assessments. A result of this is that the pure Lambda architecture based solution is seen as not necessarily the most adequate answer to each of the data challenges. The following high level architecture (See Figure 2: High-level data flow in the BDE architecture) represents the data processing flow integrating the Lambda architecture in an extended/variant context.
The BDE architecture is designed along two axes:

From left to right the (traditional) data flow from source to end-result is presented. The flow can be divided in three primary blocks which are connected via a data bus to make sure that data is transferred reliably and efficiently from one place to another:

- The 'Source provisioning' is concerned with everything that is needed to make the source data accessible for the processing step.
- During the 'Aggregation and transformation' the provided data is transformed, aggregated and turned into the representation that is going to support the application’s needs.
- In the final 'Dissemination' stage the resulting data is being disseminated to the application using the appropriate means.

The vertical axes highlight key cases of each stage. Data can be found in many different formats, but key distinctions are: data is available in such a volume on a (single) location, data is arriving as a stream, and background data that is human constructed (e.g. ontologies, controlled vocabularies, financial reports as Excel sheet ...). As such these correspond to the typical V’s: whereas the first two can be mapped to volume and velocity, the last one is often a combination of variety, variability and veracity.

The aggregation and transformation processes have to take the key characteristics of the provided data into account in order to result in an adequate data processing solution. So for voluminous data, batch processing is required, whereas if the objective is to provide near real time results, a speed layer working on a stream of data is appropriate. Where these techniques are applicable to data that is produced by machines (e.g. sensors, robots) or by humans (submitting similar messages), they are less adequate to capture any human activity producing a wide variety of data.

Despite this limitation, the information being produced in this area is often vital for an application as it forms the semantic context of data that is being processed in the automated transformation processes. For instance: the fact that a city is part of a country is a human decision. The actual geo‐spatial location (longitude and latitude) is a physical notion that does not change. So an application that shows the amount of active, online Facebook users in a country may calculate this information in two steps: (1) from the longitude and latitude of the connection where the user has logged into Facebook the city can be retrieved; (2) once the city is known the country can be determined. Displaying the number of users on a map is stable.
to our human context, however displaying the number of users in a table per country is depending on the stability of our human context as countries appear and disappear over time. Because this contextualization is an integral and important part of any data platform, it deserves an appropriate place in the architectural overview.

The dissemination representation forms of data depends on the targeted application’s needs (e.g. RDF, XML, CSV, JSON, etc.). On the one hand, applications want to have the resulting data in a form which is ready for immediate usage. This increases responsiveness of the application, but also binds the generated output very strongly to the application. On the other hand, application designers want to have sufficient flexibility in order to quickly respond to customer change requests. Commonly used application oriented dissemination channels are databases storing generated views or data in more flexible formats like RDF, search indexes (to retrieve the appropriate view id) and REST API’s. Instead of serving a dedicated application, the objective can also be to serve other data processing applications. In that case, providing metadata about the disseminated data as dataset descriptions is required.

1.4 Semantification of the Architecture

The semantification of the architecture is concerned with several aspects. Figure 3: Overview of tackled areas, provides an overview of the tackled areas (coloured in red):

- **Provenance information collecting** - Tracking the source of data and its legal context is often an important requirement for a data management system. This work will be closely connected with the work conducted in the W3C PROV Family of Documents (W3C, 2013-2015). It is expected that an integration of Linked Data will ease the integration of the provenance collection process.

- **Data interoperability** - By reusing public vocabularies and ontologies the resulting data can be provided in a common semantic context. The vocabularies can also support the actual data processing. By having the data models explicit more reusable components can be provided.

- **Data dissemination** - Linked Data offers a wide variety of, out of the box, generic interfaces for machine readable data. This is complementary to existing Big Data
solutions which aim to create the best data structure (view) for an application. Example interfaces are dereferenceable URIs, SPARQL endpoints, JSON-LD feeds, etc.

During the project these aspects will be elaborated. The priorities will be driven by the pilots that will be developed.

1.5 Architectural Conclusion

The BDE architecture design is based on the Lambda architecture pattern, which has been previously described. This architectural design shows in a comprehensive overview the different kind of functionalities that will be foreseen in order to be able to address a wide range of data problems. As a specific ambition the semantification of the data platform has been highlighted. It is believed that doing so will combine the best of two data worlds: the Big Data community and the Linked Data community.
2. BDE Architectural Components

The BDE project aims to manage the deployment of the architectural components. For each component a number of possible choices are available. In order to aid in rationalising the choice, the following section indicates some of the criteria which will be used. Thereafter, a description is given of some of the components which form a part of the BDE framework and the additional components which the project is intending to integrate into it. Finally, a start is made in assessing the current state of the various components and frameworks which will form the start point for the work.

2.1 Component Selection Criteria

Whatever tools or components are to be actually used from the available current set, the following characteristics need to be managed (or determined):

- **Durability** - Is the component under active development? Is there an expectation that the component will be around and useable for a "reasonable" amount of time\(^1\)? This also can depend on who is sponsoring the project if open-source, for example it could be an Apache project.

- **Sustainability** - when systems, interfaces, methods are frequently changing, then the requirement to continually update their applications will become costly. In the long run, this is unsustainable since those knock-on costs of those continual changes will not be accepted, i.e. testing requirements, bug fixing, training of users, etc.

- **Accessibility** - It should be possible for the application/framework component developers to find out how to use other components not developed by them. This to a certain extent depends on the ecology around the component (i.e. documentation, mailing lists, FAQs, etc.) much of which would be present for sustainable/durable components.

- **Flexibility** - Components need to have all the functions/methods available to remove the need to modify those components to get them to work as required (i.e. hard coded dependencies should be minimal).

- **Stability** - This is to a certain extent a measure of maturity, since once the interfaces, etc. have become well designed; the components will normally become more robust. Problems will tend to become localised and easier to debug, test and optimise.

- **Reliability** - Encompasses the processes around the development and all the above points. If the interfaces are not stable then the individual parts cannot be made easily robust and frequent changes are needed to keep the software operation. This is normally not sustainable.

Selecting components with strong characteristics create a stable foundation for the BDE framework. These characteristics have to complement the following additional characteristics because the BDE framework should have components which are able to react adequately to changes in volume, velocity and variety:

- **Scalability** - This is the central point for a data project since the intention is to deal with large volumes of data. Scalability in this case meaning that the components should be

\(^1\) e.g. gcc, emacs/vi, sh, awk, sed, etc. are all standard parts of GNU/Linux which can be relied on to be present, they are also arguably some of the most successful software components available having been around since the 80's in mostly the same form. That is not to say that the components have not been updated and not changed, but that the core functionality was sufficiently solid that they are worth keeping available.
able to take advantage of additional resources (e.g. more memory disks or machines) when they are made available.

- **Fault-tolerant** - Meaning that if processing nodes fail, the framework will restore the node functioning, or (re)distribute the nodes if performance starts to degrade to a toleration point.
- **Monitorability** - This is a requirement which is, because of the scalability requirement, some way needed to allocate additional resources or to investigate whether the processes are functioning efficiently (as well as correctly, etc.).

Additional factors which need to be considered (which are often more specific business oriented questions) are:

- What is the process around the component development (open-source or commercial, active development, release cycles, etc.).
- The component versions will have to be compatible (libraries, OS details, etc.). Isolating the versions of open-source tools which work nicely together is a time consuming task.
- The operating systems that need to be supported since this will impact on the system performances and components which can be used.
- Whether the tools have different licenses and what the impact of those licenses could be on the applications to be developed?
- Are all the components available or only selected parts, with some parts being commercial - i.e. is the designed set up as a channel towards lock-in.

The above summarises some of the factors which will pose challenges when choosing components to be included in the BDE framework. Within the large Big Data community, it is a commonly mentioned lesson learned: that the open-source development is very active, with fast updates and improvements which needs to monitored. This positive atmosphere creates a potential with almost no limitations. However, since there is no overall central coordination, the interplay is tempered: interfaces and protocols change without too warning and might not be backwards compatible; there is a high overhead to find out which versions of different products really work together. The above situation is often used as a justification for choosing a framework/distribution which encompasses several components in one package as a start point. One of the Big Data distributions, Hortonworks, shows this challenge in Figure 4: Hortonworks view of Challenge (Connolly, 2012).
It is a tedious and time consuming task to select from all open-source solutions a consistent set so that they can interplay. Therefore it has been decided to base the BDE platform on a Big Data component distribution such as Hortonworks\(^3\). Using such a component distribution also brings the experience of the people behind these distributions to the consortium. They are in the core of the Big Data market, mining the requirements and watching how technologies are trending, attracting attention and gaining acceptance. Their experience should be taken into consideration when making decisions in BDE.

2.2 Component Classification

This section describes a possible classification of components identifying the main purposes of the component. The objective of this classification is to gain rapid insight in the application area of a component in the BDE platform. The classification is mainly based on existing overviews such as BIG (Big Data Technical Working Groups, 2014), enhanced with recent information about the component evolution since the previous assessment.

The classification only collects components that were considered to be a part of the BDE platform. The objective of this overview is not to be a complete overview of every possible suitable data management component that exists. It aims for providing an understanding how a component may or may not be part of the BDE platform. The decision of integrating a component in the BDE platform is determined on technical arguments but also on non-technical arguments such as existing experience in the consortium, availability of the component in the Hortonworks framework or on request by a use case.

2.2.1 Data Acquisition

Data acquisition tools support the process of gathering, filtering and cleaning data before it is put in a data warehouse or any other storage solution on which data processing can be carried out (Big Data Technical Working Groups, 2014). They should be able to handle the huge volume, high velocity, or variety of data. Data is extracted from structured, semi-structured or unstructured data sources using standard or proprietary database interfaces,
APIs provided by the source application or file-based interfaces (See Section Initial BDE Architecture Pattern).

Depending on the type of data and the source data originates from, data might be processed before it is put in the platform, i.e. next to parsing, unstructured data might need an additional information extraction step, while structured data does not. Data acquisition can also involve data filtering, e.g. to get rid of irrelevant data.

Example technologies for data acquisition are:

- **Apache Flume** - Apache Flume is a framework to populate HDFS with streaming event data. Flume agents, which are simple JVM processes, are populated throughout the infrastructure - for example inside web servers, application servers, mobile devices etc. - to collect data and integrate it into the data storage. Flume allows users to build multi-hop flows where events travel through multiple agents before reaching the final destination.

- **Apache Sqoop** - Apache Sqoop is a framework to transfer data between structured data stores such as RDBMS and Hadoop. Sqoop provisions data from external systems on to HDFS. It can also be used to populate tables in Hive and HBase. Sqoop uses a connector based architecture which supports plugins that provide connectivity to new external systems.

- **Scribe** - Scribe is a server technology developed at Facebook to aggregate log data in real-time from a large number of servers. The log data is gathered in the platform via a directed graph of Scribe servers. This graph is based on a static configuration in which each server knows only about the next server in the graph. Nowadays Scribe is no longer used by Facebook. As a consequence the code is not actively maintained and supported anymore.

- **Apache Kafka** - See Section Message Passing.

- **RabbitMQ** - See Message Passing.

### 2.2.2 Message Passing

Message passing software supports sending and receiving messages between software components. The messaging middleware decouples applications by putting a message transfer agent (message broker) in the middle that separates sending and receiving data. Messages are routed from sender to receiver according to a messaging paradigm like point-to-point, fan-out, publish-subscribe etc. The messaging middleware might be put at the start of the data processing pipeline where it takes the role of a data acquisition tool to load data into the platform from several sources.
By decoupling the sender and receiver, the message broker hides the details of the various operating systems and network interfaces from the application developer and offers a common platform for sending and receiving messages thereby reducing the complexity of application development.

Message passing is typically used in asynchronous messaging where the sending system does not wait for a response. The messaging middleware provides a temporary storage where the sender can store the message. The message receiver can retrieve the message from this temporary storage. The main advantage of this asynchronous messaging model is that the sender and receiver do not need to connect to the network at the same time. Moreover, if the receiver application fails for any reason, the sender can continue unaffected, as the messages will simply accumulate in the message queue for later processing when the receiver restarts. The main disadvantage of message-oriented middleware is the requirement of an extra component in the architecture which may reduce the performance of the system (Marz, et al., 2015).

Example technologies for message passing are:

- Apache Kafka - Apache Kafka is a distributed publish-subscribe messaging system developed at LinkedIn to support mainly persistent messaging with high throughput. Kafka maintains feeds of messages in categories called topics. Each topic behaves as a commit log, i.e. an ordered, immutable sequence of messages that is continually appended to. Message producers publish messages in a topic while message consumers subscribe to a topic to read the messages from it. Kafka brokers are stateless, i.e. consumers should keep track by themselves which messages they have consumed so far. This is what makes Kafka lightweight compared to other message passing systems. Messages are retained by the Kafka brokers for a configurable period of time, whether or not they have been consumed.

- RabbitMQ - RabbitMQ is a publish-subscribe message passing system implementing AMQP, an application layer protocol designed to efficiently support a wide variety of messaging applications and communication patterns. The RabbitMQ server is written in Erlang but client libraries to interface with the broker are available for all major programming languages. In contrast to Kafka, which is producer-centric, RabbitMQ is broker-centric. The main focus of the system is the delivery guarantees between producers and consumers.

---

4 Source: https://i-mstdn.sec.s-mstt.com/dynimg/IC171851.gif
2.2.3 Data Processing

As explained in Section Lambda Architecture, the Lambda architecture defines two layers: the batch and the speed layer. Data processing technologies used in those layers can be classified accordingly: batch processing systems and real-time or stream processing systems. Batch processing systems are high throughputs, high latency systems. They can do nearly arbitrary computations, but they may take hours or days to do so. Real-time or stream processing systems on the other hand are high throughput, low latency stream processing systems. They can't handle the range of computations a batch processing system can, but they process messages extremely quickly.

Until a couple of years ago there was a clear line between both kind of processing systems. A technology clearly focused on either batch or stream processing (e.g. Hadoop MapReduce for batch processing, Apache Storm for stream processing). Nowadays however technologies such as Apache Spark and Apache Flink start to arise claiming to handle both kinds of processing. The next sections will have a closer look at each of these technologies.

2.2.3.1 Hadoop MapReduce

Hadoop consists of two main components: HDFS and MapReduce. HDFS is a distributed file system used to store the immutable, constantly growing master dataset. It is described in more detail in Section Distributed File Systems. MapReduce is the data processing component of Hadoop. It provides a software framework to easily write applications that process large amounts of data in parallel using the MapReduce paradigm. This paradigm processes data following four steps:

1. Split the input data into independent chunks
2. Process the data chunks in parallel by a Map task
3. Sort the outputs of the Map tasks
4. Use the outputs of the Map tasks as input of the Reduce tasks

The results of the MapReduce processing are arbitrary views from the master dataset. Computing these views is a continuous operation, so when new data arrives it will be aggregated into the views when they are recomputed during the next MapReduce iteration.

Hadoop MapReduce has undergone a complete overhaul in hadoop-0.23. In this release YARN, also known as MapReduce v2 or NextGen MapReduce has been introduced. YARN divides the two major functions of MapReduce v1's JobTracker, resource management and job life-cycle management, into separate components enabling Hadoop to support more varied processing approaches and a broader array of applications than just MapReduce. Those applications range from interactive applications (Apache Tez) to in-memory applications (Apache Spark), streaming applications (Apache Storm), graph applications (Apache Graph), etc.

Hadoop MapReduce is often criticized to be too low-level. It's not easy for developers to think and develop applications in terms of the MapReduce paradigm. As a consequence numerous tools have been developed around Hadoop MapReduce to solve its limitations. Some examples of these tools are:

- **Apache Pig** - A higher-level API, developed by Yahoo, to write MapReduce jobs against data in HDFS. Complex tasks comprised of multiple interrelated data transformations can be explicitly encoded as data flow sequences, making them easy to write, understand, and maintain. The way in which tasks are encoded permits the system to optimize the execution automatically, allowing the user to focus on semantics rather than efficiency.
- **Apache Hive** - A higher-level API, developed by Facebook, to write SQL-like queries in a custom query language, called HiveQL, against data in HDFS. In the background each query is translated into a MapReduce job.

- **Apache Mahout** - A machine learning library on top of MapReduce providing algorithms like likelihood, classification and clustering. All algorithms are translated in the background to the MapReduce model.

### 2.2.3.2 Apache Storm

Apache Storm is a distributed real-time computation system on data streams. Computations are defined as a topology consisting of spouts and bolts:

- **Spout** - A source of streams to feed data in the topology. A spout reads from a queueing broker (e.g. Apache Kafka) or generates its own stream by reading data from somewhere else (e.g. Twitter streaming API).

- **Bolt** - Component that processes any number of input streams and produces any number of new output streams. The processing includes computation logic such as functions, filters, streaming joins, streaming aggregations, talking to databases, etc. The input streams a bolt processes, originate from a spout or the output stream of another bolt.

![A Storm topology consisting of spouts and bolts](http://eugenedvorkin.com/wp-content/uploads/2013/03/Stormtopology.png)

Figure 6: A Storm topology consisting of spouts and bolts

In contrast to a batch computation system where the computation ends after a finite amount of time, a Storm topology processes incoming messages forever (or until interrupted). A Storm cluster consists of two kinds of nodes: one master node running a Nimbus daemon and multiple worker nodes running a Supervisor daemon. All coordination between Nimbus and the Supervisors is done through a Zookeeper cluster. The Nimbus daemon runs on the master node and is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures. The Supervisor daemons on the other hand run on the worker nodes and listen for work assigned to their machine. If work gets assigned, they start and stop worker processes on their node as necessary.

---

2.2.3.3 Apache Spark

In contrast to most data processing engines which process data on disk, Apache Spark is an in-memory data processing engine. It provides APIs in Java, Python and Scala which try to simplify the programming complexity by introducing the abstraction of Resilient Distributed Datasets (RDD), i.e. a logical collection of data partitioned across machines. The way applications manipulate RDDs is similar to manipulating local collections of data.

On top of its core, Apache Spark provides 4 libraries:

- **Spark SQL** - Library to make Spark work with (semi-)structured data by providing a data abstraction SchemaRDD/DataFrame on top of Spark core. The library also provides SQL support and a domain-specific language to manipulate SchemaRDD/DataFrame.
- **Spark streaming** - Library that adds stream data processing to Spark core. Spark streaming makes it easy to build scalable fault-tolerant streaming applications by ingesting data in mini-batches. Moreover, application code developed for batch processing can be reused for stream processing in Spark.
- **Mlib Machine Learning Library** - Library that provides a Machine Learning framework on top of Spark core. Mlib supplies to Spark core what Apache Mahout supplies to Apache Hadoop
- **GraphX** - Library that provides a distributed graph processing framework on top of Spark core. GraphX comes with a variety of graph algorithms unifying ETL, exploratory analysis, and iterative graph computation within a single system.

2.2.3.4 Apache Flink

Like Apache Spark, Apache Flink is a data processing engine for batch and real-time data processing. Regarding the batch data processing, Apache Flink offers an abstraction on top of Hadoop MapReduce by providing more functions than just MapReduce which makes writing programs easier. Moreover, users can implement their own functions in Scala or Java. Flink transforms them into a parallel data flow, a graph of parallel operators and data streams between those operators. On real-time processing Apache Flink is comparable to Apache Spark although Flink claims to be more performant than Spark because they use real in-memory streaming (as much as possible) and not mini-batches like Spark does.

2.2.4 Data Storage

2.2.4.1 Distributed File Systems

Distributed file systems are the distributed variant of local file systems. They aim to be transparent by providing the same interface and semantics as local files systems to access files (e.g. mounting and unmounting, listing directories, system's native permission model) while the distributed file system handles locating files and transporting data over the network behind the scenes. Distributed data stores (NoSQL Databases), by contrast, require using a different API or library and have different semantics, most often those of a database.

The best known distributed file system is HDFS, an Apache Hadoop subproject. It is a distributed file system designed to run on commodity hardware on top of regular file systems like ext3/ext4. Just as a traditional file system HDFS provides the typical file system commands such as ls, rm, mkdir, tail, copy, etc. It also provides a REST API that supports the complete FileSystem/FileContext interface for HDFS.

HDFS is tuned to store large files. As illustrated in Figure 7: HDFS file storage, files are split into blocks which are then distributed and replicated across the nodes to be highly fault-tolerant as hardware failure is the norm rather than the exception. HDFS has a write-once-
read-many access model, i.e. data cannot be modified in HDFS; it can only be added. The emphasis of the file system is on high throughput of data access rather than on low latency of data access.

Figure 7: HDFS file storage

2.2.4.2 NoSQL Databases

RDBMS and the SQL programming language have long been the main and, in many cases, only choice of database technologies for organizations. Unfortunately, RDBMS technology’s biggest strength, a fixed schema design, has become its greatest weakness. Today’s data sets include structured and unstructured data originating from a variety of data types such as emails, log files, social media, sensor events etc. Data exists in high volumes and undergoes high rates of change. Moreover one of the key characteristics of big data applications is that they demand real-time or near real-time responses, i.e. data needs to be stored in such a way that it can be accessed quickly when required. RDBMS is not a good fit here (Asay, 2013).

NoSQL databases are designed with these big data needs in mind. They support dynamic schema design and offer increased flexibility, scalability and customization compared to relational databases. The term "NoSQL" does not necessarily mean "no SQL at all", it also is commonly defined as "Not Only SQL" because many NoSQL databases do support some elements of SQL. But they do not rely on RDBMS’s fixed-schema design principles, giving NoSQL users more flexibility in structuring their databases.

NoSQL databases are grouped into four categories each with its own architectural characteristics: key-value stores, document stores, column-oriented stores and graph databases. Many NoSQL platforms are tailored for specific purposes - there is no one-size-fits-all solution - so when selecting a platform it should reflect the application and its usage patterns in order to be a good fit. The following sections describe each of the NoSQL database

---

6 HDFS file storage: files are split in blocks which are distributed and replicated across the data nodes. (Source: http://blogs.infosupport.com/wp-content/uploads/2014/03/Block-Replication-in-HDFS.png)
categories in more detail and lists some example technologies per category. These lists are by no means exhaustive.

### 2.2.4.2.1 Key-Value Stores

Key-value stores are the simplest NoSQL databases. These databases are designed for storing data in a schema-less way. In a key-value store every single item in the database is stored as an attribute name or key, together with its value, hence the name. Some key-value stores allow each value to have a type (e.g. integer) which adds functionality.

Examples: Project Voldemort, Redis, Riak

### 2.2.4.2.2 Document Stores

Document stores expand on the basic idea of key-value stores but they pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, key-array pairs or even nested documents. Each document is assigned a unique key, which is used to retrieve the document. Document stores are designed for storing, retrieving and managing document-oriented information, also known as semi-structured data.

Examples: CouchDB, MongoDB

### 2.2.4.2.3 Column-Oriented Stores

Column-oriented stores are also known as wide-column stores. Instead of storing data in rows, these databases are designed for storing data tables as sections of columns of data, rather than as rows of data.

Examples: Apache HBase, Cassandra

### 2.2.4.2.4 Graph Databases

Graph databases are based on graph theory and store data in graph structures using nodes and edges connecting each other through relations. These databases are designed for data containing elements which are interconnected, with an undetermined number of relations between them. Graph databases provide index-free adjacency, i.e. every element contains a direct pointer to its adjacent elements and no index lookups are necessary.

Examples: Neo4J, FlockDB, HyperGraphDB

### 2.2.4.2.4.1 NoSQL Database Comparison

Each category of NoSQL databases has its own strengths. Even within a single category, each technology might have a different focus. A high level overview is presented in Table 1: High level view of NoSQL database categories’ main characteristics" comparing the NoSQL database categories on the aspects of performance, scalability, flexibility and complexity. As mentioned before a one-size-fits-all solution does not exist. Before selecting a database technology one should analyse the application and its usage patterns in order to find the best fit for the problem.

*Table 1: High level view of NoSQL database categories’ main characteristics*[^1]

<table>
<thead>
<tr>
<th>Performance</th>
<th>Key-value store</th>
<th>Document store</th>
<th>Column-oriented store</th>
<th>Graph database</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>Variable</td>
</tr>
</tbody>
</table>

[^1]: Comparison of NoSQL database categories’ main characteristics. Source: [http://www.planetcassandra.org](http://www.planetcassandra.org).
<table>
<thead>
<tr>
<th>Scalability</th>
<th>High</th>
<th>High</th>
<th>High</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexibility</td>
<td>High</td>
<td>High</td>
<td>Moderate</td>
<td>High</td>
</tr>
<tr>
<td>Complexity</td>
<td>None</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

2.2.4.3 Triple stores

Triple stores are database management systems for data that is modelled using RDF. RDF data can be thought of in terms of a directed labelled graph wherein the arcs start with subject URIs, are labelled with predicate URIs, and end up pointing to object URIs or scalar values. Data can be queried using SPARQL. Triple stores can be classified in three type categories:

- **Native triple stores** - Triple stores implemented from scratch exploiting the RDF data model to efficiently store and access the RDF data. Examples: Stardog\(^8\), Sesame, OWLIM
- **RDBMS-backed triple stores** - Triple stores built by adding an RDF specific layer to an existing RDBMS. Example: OpenLink Virtuoso
- **NoSQL triple stores** - Triple stores built by adding an RDF specific layer to existing NoSQL databases. Example: CumulusRDF (built on top of Cassandra),

2.2.5 Data Visualization

To be investigated in next iterations.

2.2.6 Monitoring

The architecture described in Initial Architecture (Section Initial BDE Architecture Pattern) consists of several components. A monitoring tool allows to follow-up the availability, status and performance of each of the components in the platform. If one of the components fails or exceeds a configured threshold (e.g. remaining disk space too low) a notification might be sent to an administrator to get his attention.

Most components described in this document already provide a management interface by means of a CLI or a web UI where one can monitor the state of a single component. The aim of this project however is to provide a uniform monitoring interface to the user where one can follow-up all the components in the platform. Apache Ambari, an open-source tool developed by Hortonworks, will be used as a basis.

Apache Ambari is a completely open operational framework for provisioning, managing and monitoring Apache Hadoop clusters. Ambari comes with an intuitive web interface allowing to control the lifecycle of services and components, to modify configurations and to gain instant insight into the health of the cluster. Ambari also includes a metrics system to collect metrics and an alert framework for system alerting and notifications. Although Ambari is focused on components which are part of the Hadoop ecosystem, it provides a REST API to integrate other resources with Ambari.

2.3 Framework Selection and Component Status

As indicated previously in Section 2.1 Component Selection Criteria, while discussing Figure 4: Hortonworks view of Challenge, the best starting point for the BDE development is to use an existing distribution which encapsulates several of the expected architectural components and to build upon that framework. In this section a summary is given of the current

---

\(^8\) Not an open-source component.
status of the identifier Big Data distributions and the components status they supply. Following this, a summary of the OS support availability is given, with a view to showing where the startpoint for the BDE framework can be found. The sections is concluded with a first list of semantic technology components that are considered to the added to the BDE framework.

2.3.1 Framework Component Summary

The currently identified frameworks are:

- **Cloudera**\(^9\) - provides many of the Apache Hadoop open-source components along with a set of additional proprietary components and services. Components are available as tar-balls\(^{10}\). They provide VMs and preconfigured services based around these tools, rather than the packages.

- **Hortonworks Data Platform** - is a set of tools which come with a guarantee that all the tools will be fully open-source, following the Apache project process. with no parts restricted. Packages in RPM and DEB formats are available with the RPM format being marginally better supported. The current pre-release version is 2.2 which can be downloaded as a single virtual machine (virtualbox, vmware, etc.).

- **BigTop**\(^{11}\) - is an Apache infrastructure project to build, test and package hadoop and related Apache projects (currently it is version 0.8.0 which is intended to download and build the packages required to install it on one of the selected packages. RPM and DEB have build targets)\(^{12}\).

These frameworks/distributions provide similar facilities and components to those which can be seen in the Lambda architecture (Indicated in the table below with an L). In must be noted that the BDE project is only looking for a start point for selecting compatible component versions, which are known to work together. It will still be necessary to make sure that all the selected BDE components do work correctly together in the pilot applications. The table below shows the main component versions which are associated with each of the identified Big Data frameworks\(^{13}\):

---

\(^9\) Current version is 5.4, details of the packages are on [http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/cdh_vd_cdh_package_tarball.html#topic_3_unique_2](http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/cdh_vd_cdh_package_tarball.html#topic_3_unique_2) which would have to be installed/repacked for installation.

\(^10\) [https://cwiki.apache.org/confluence/display/BIGTOP/Index](https://cwiki.apache.org/confluence/display/BIGTOP/Index)

\(^11\) The relationship between the bigtop and cloudera frameworks isn't clear.

\(^12\) This is to show the versions which should work acceptably together.
Table 2: Framework/BDE Component Versions

<table>
<thead>
<tr>
<th>BDE Framework Components</th>
<th>BigTop</th>
<th>Hortonworks</th>
<th>Cloudera</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Acquisition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** Apache Flume</td>
<td>1.5.0.1</td>
<td>1.5.2</td>
<td>1.5.0</td>
</tr>
<tr>
<td>*** Apache sqoop</td>
<td>1.99.2</td>
<td>1.4.5</td>
<td>1.4.5</td>
</tr>
<tr>
<td>Message Passing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** Apache Kafka (L)</td>
<td></td>
<td>0.8.1.1</td>
<td></td>
</tr>
<tr>
<td>Data Processing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** Apache Storm (L)</td>
<td></td>
<td>0.9.3</td>
<td></td>
</tr>
<tr>
<td>*** Apache Spark (L)</td>
<td>0.9.1</td>
<td>1.2.1</td>
<td>1.3.0</td>
</tr>
<tr>
<td>*** Apache Hadoop/HDFS (L)</td>
<td>2.3.0</td>
<td>2.6.0</td>
<td>2.6.0</td>
</tr>
<tr>
<td>*** Apache Flink</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** Apache Pig</td>
<td>0.12.1</td>
<td>0.14.0</td>
<td>0.12.0</td>
</tr>
<tr>
<td>*** Apache Hive/HCatalog (L)</td>
<td>0.12.0</td>
<td>0.14.0</td>
<td>1.1.0</td>
</tr>
<tr>
<td>*** Apache Mahout</td>
<td></td>
<td>0.9.0</td>
<td></td>
</tr>
<tr>
<td>Database components</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** Apache HBase (L)</td>
<td>0.98.2</td>
<td>0.98.4</td>
<td>1.0.0</td>
</tr>
<tr>
<td>*** Apache Cassandra (L)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** Project Voldemort (L)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other Components</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** Apache Zookeeper</td>
<td>3.4.1+</td>
<td>3.4.6</td>
<td>3.4.5</td>
</tr>
<tr>
<td>*** Apache Oozie</td>
<td>4.0.1</td>
<td>4.1.0</td>
<td>4.1.0</td>
</tr>
<tr>
<td>*** Cascading (L)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** Apache Solr</td>
<td>4.6.0</td>
<td>4.10.2</td>
<td>4.10.3</td>
</tr>
<tr>
<td>Deployment Components</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** Mesos</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The components list can be expected to change once the detailed components requirements of the BDE pilots are known. These tables will form the basis of the eventual BDE release notes.

2.3.2 Framework OS Availability Status

In terms of underlying operating system (OS), the following table shows the current OS availability situation for the three frameworks considered so far:

Table 3: Framework OS Availability Status

<table>
<thead>
<tr>
<th>OS Version</th>
<th>HORTONWORKS</th>
<th>BIGTOP</th>
<th>CLOUDDERA</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU/Linux - DEB Packages</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* Debian Jessie (8)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS Version</td>
<td>HORTONWORKS</td>
<td>BIGTOP</td>
<td>CLOUDERA</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>* Debian Wheezy (7)</td>
<td></td>
<td></td>
<td>CHD 5.4</td>
</tr>
<tr>
<td>* Debian Squeeze (6)</td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* Ubuntu (14.04)</td>
<td></td>
<td>0.8.0 (known dependency issues)</td>
<td>CHD 5.4</td>
</tr>
<tr>
<td>* Ubuntu (12.04)</td>
<td>2.2</td>
<td>0.8.0 (build dependency)</td>
<td>CHD 5</td>
</tr>
</tbody>
</table>

**GNU/Linux - RPM Packages**

<table>
<thead>
<tr>
<th>OS Version</th>
<th>HORTONWORKS</th>
<th>BIGTOP</th>
<th>CLOUDERA</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Redhat/CentOS (7)</td>
<td></td>
<td>0.8.0</td>
<td></td>
</tr>
<tr>
<td>* Redhat/CentOS (6.4,6.5,6.6)</td>
<td>2.2</td>
<td>0.8?</td>
<td>CHD 5</td>
</tr>
<tr>
<td>* RedHat/CentOs (5.7,5.105)</td>
<td>2.2 (Deprecated)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* SLES (11)</td>
<td>2.2</td>
<td></td>
<td>CHD 5.4</td>
</tr>
<tr>
<td><strong>Windows (8)</strong></td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MacOS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This table shows four main packaging streams: RPM (Redhat, CentOS, Fedora), Deb (Debian, Ubuntu), Windows and MacOS (which is not commonly used in a server setting). Of the GNU/Linux packaging formats provided by the various frameworks, RPM is the better supported (marginally). It should also be noted, that at present some of the newer operating system distribution versions are not yet supported (e.g. debian 8 (jessie) which was released in March 2015).

### 2.3.3 BDE Semantic Components

As was previously indicated, the list of expected BDE components (Table 2: Framework/BDE Component Versions) will also be supplemented by a set of semantic components which are intended to be integrated with the BDE framework. These components are not available in any of the investigated big data frameworks. These components were partially built in the LOD2 and GeoKnow projects and are currently part of the Linked Data Stack. This stack is also used by third parties and we aim at improving it further as part of the BDE platform (see also Section 4 on how the Linked Data Stack repository can be integrated into the BDE platform deployment strategy). We aim to make the following components available as packages in the packages (tentative list):

- OpenLink Virtuoso ([https://github.com/openlink/virtuoso-opensource](https://github.com/openlink/virtuoso-opensource))
- LIMES ([http://aksw.org/Projects/LIMES.html](http://aksw.org/Projects/LIMES.html))
- DL-Learner ([http://dl-learner.org](http://dl-learner.org))
- DEER ([http://aksw.org/Projects/DEER.html](http://aksw.org/Projects/DEER.html))
- FOX ([http://aksw.org/Projects/FOX.html](http://aksw.org/Projects/FOX.html))
- Facete ([http://aksw.org/Projects/Facete.html](http://aksw.org/Projects/Facete.html))
- RDFUnit ([http://aksw.org/Projects/RDFUnit.html](http://aksw.org/Projects/RDFUnit.html))
- CubeViz ([http://aksw.org/Projects/CubeViz.html](http://aksw.org/Projects/CubeViz.html))
- Strabon ([http://www.strabon.di.uoa.gr](http://www.strabon.di.uoa.gr))

---

14 Note: RPM packages can be installed on debian as alien packages so the choice isn’t really that crucial. The debian packaging description is spread over several files, while RPM is a single file with separate sections (easier to edit).
15 The Linked Data Stack ([http://stack.linkeddata.org](http://stack.linkeddata.org)) is the result of the LOD2 and the GeoKnow projects.
2.4 Conclusions and Framework Chosen

The basic conclusions from investigating the existing frameworks and the BDE architectural pattern are given in this section. Starting at the OS level, the following approach will be taken:

- All the components used or developed will be open-source,
- As BDE will be, for the most part, a server based deployment, only the GNU/Linux based packaging approaches will be supported,
- Preference should be given to the Debian packaging usage because of the intention to include and use packages from LOD2\textsuperscript{16}/GeoKnow\textsuperscript{17} projects.
- MacOS and Windows will not be directly supported, but running BDE within a GNU/Linux VM should be possible,
- Given the project duration of 3 years, the older OS versions (i.e. Ubuntu 12.04) should not be supported for BDE and the focus should be on the latest currently available GNU/Linux versions.

Of the three distributions investigated, Hortonworks best fits the requirements as a starting point for the BDE framework, for the following reasons:

- It has an active development approach (Cloudera has this as well, BigTop is unclear),
- All the components are fully open-source, no proprietary extensions,
- Debian/RPM packages are available,
- It provides all the necessary BDE architecture pattern components.

One additional reason for this whole approach is the intention to add to the BDE framework some semantic components. These components are already available as Debian packages (and can be easily used within a docker environment).

2.4.1 Following up the Platform Integration Status

As described above the BDE framework will start the existing distribution HortonWorks, enriched with additional components and with the semantification of already existing components. Because the BDE project needs to provide a coherent framework, some criteria need to be defined how the integration of a component in the framework can eventually be measured. The following provides an initial list which might be extended over time:

- The component is packaged using Docker (dockerized) so it can be deployed
- The component consumes RDF
- The component produces RDF
- The component is capable to provide provenance information
- The component preserves metadata
- The component is integrated in the general admin interface

\textsuperscript{16} http://www.lod2.eu
\textsuperscript{17} http://www.geoknow.eu
The intention is to use this list to drive the QA for the framework as a whole.

3. Deployment Strategy

One of the most critical stages in the realization of a software project is the deployment phase. During that phase the software is transferred from a development setting to a production setting in which the system stability and system administration aspects take priority over new functionality. A good deployment strategy is one with clear resource and system management already being applied during the development phase and will enforce the delivery of a robust and quality end-product. Our deployment strategy will also form an important contribution to the Big Data Open-Source community. Today often valuable time is wasted to explore deployment methodologies because of the diverse nature & status of each open-source software component. A common deployment ground is a key enabler in easing the usage of open-source Big Data technology.

3.1 Big Data Deployment

Software deployment covers a set of interrelated activities which occur when making a system available for use. Typical activities include:

- Release creation/management involves making sure that all the relevant components operate correctly, that testing is possible, that stability and reliability of the system are maintained when the additional features or enhancements are made available. Central to this is making sure that users do not lose data or information.
- Version and feature tracking (bugs as well), which basically means keep track of which components/features are available at which point and in which release.
- Installation/updating and possibly rolling back are the main reasons for the release creation and management requirements. The more critical the application the higher the QA costs on the release creation.

For Big Data applications, it is not necessarily just the changes to the software components which need to be taken into account. Changes in the data table structures or representation format could be very costly to implement (as well as backing up) depending on the volume or changes required. Additionally, each of the deployed big data system instances will have a specific configuration (number of machines, IP addresses, disk types and space available, etc.) which will have to be managed as well as maintained when updating to a new release. Configuration management is therefore an essential aspect in the deployment process.

3.2 Virtual Deployment

Nowadays application deployment is often done using virtual machines instead of native installing the application on a server. Use of a virtual machine isolates the application in a sandbox, a known and controlled environment. Virtual machines can easily be shipped from one server to another without breaking anything. Unknown factors such as mixed libraries caused by numerous installs can be eliminated. Application virtualization liberates developers and system operators from application and infrastructure dependencies, significantly accelerates the software development lifecycle, and enables substantial improvements in infrastructure cost and efficiency.

Even when creating virtual machines, there is a need to describe the packages, components and configuration settings which are part of the virtual machine(s). One
technology which has been investigated for this is Vagrant\textsuperscript{18} where the virtual machine(s) can be described in a Vagrantfile which can be used to recreate it at any time from scratch.

### 3.2.1 Docker Engine

#### 3.2.1.1 Docker Containers

Docker Engine exploits the advantages of virtual application deployment by providing a platform that enables any application to be created and run as a collection of consistent, low-overhead Docker containers that work across virtually any infrastructure. Docker containers are based on GNU/Linux Containers. In contrast to traditional virtual machines, which have a full OS with its own memory management installed, Docker containers reuse the kernel of their host and share libraries where possible. This approach makes Docker containers more lightweight than virtual machines while still running the application in a sandbox.

![Docker containers vs. traditional virtual machines](http://windowsitpro.com/site-files/windowsitpro.com/files/uploads/2015/01/docker%20overview.jpg)

Figure 8: Docker containers vs. traditional virtual machines\textsuperscript{19}

Docker containers can easily be started, stopped, removed and restarted. A virtual machine could take several minutes to create and launch whereas a Docker container can be created and launched just in a few seconds. This makes it easy to quickly launch more containers when needed and shut them down when they are no longer required. Moreover, the lightweightness of the containers and the ease of starting, stopping and removing them, makes Docker, next to deployment, an excellent tool for development, testing and experimentation.

Docker containers have several features such as:

- **Container linking** - On container startup a container can be linked to another container such that they can connect. A typical use case of container linking is a web application running in one container requiring a connection to a database running in another container.

- **Volume mounting** - A volume of the host machine can be mounted in a container on container startup. When the container is removed at a later time the mounted volume

\textsuperscript{18} [http://www.vagrantup.com](http://www.vagrantup.com)

will be preserved as it is part of the host system. A typical use case of volume mounting is a database container in which the folder where the database stores its data is mounted from the host.

- **Port binding** - A container can expose specific ports. These ports can be bound to any port on the host machine at container startup. That way services running in a Docker container become available via the host IP. E.g. a Docker container running a web application typically exposes port 80. This port can be bound to port 8080 on the host machine making the web application available at http://(host-ip):8080.

An elaborate set of features can be found in the Docker run command reference\(^\text{20}\).

### 3.2.1.2 Docker images

Docker containers are created from Docker images. A Docker image is a read-only template built by executing the instructions from a Dockerfile successively\(^\text{21}\). For example, an image could contain an Ubuntu operating system with a web application server and a web application installed. Each instruction in a Dockerfile creates a new layer in the Docker image. These layers make Docker images very hardware resource friendly. When a Docker image is changed, for example, update an application to a new version, a new layer gets built. Thus, rather than replacing the whole image or entirely rebuilding it, as might be done with a virtual machine, only that layer is added or updated making Docker image distribution faster and simpler. Layers can also be shared across images. A Docker image might be based on another Docker image. That image then forms the base layer of the new Docker image.

![Layers in a Docker Image](https://docs.docker.com/terms/layer/#layer)

---

\(^{20}\) See [https://docs.docker.com/reference/run](https://docs.docker.com/reference/run)

\(^{21}\) which partially addresses the requirement for documenting the configuration of the Docker based components. The Dockerfiles can be placed in source code repository and different versions compared when needed.

\(^{22}\) Layers in a Docker image (Source: [https://docs.docker.com/terms/layer/#layer](https://docs.docker.com/terms/layer/#layer))
3.2.2 Docker Registries

A Docker registry is a public or private service to store and distribute Docker images centrally. Images can be pulled from a repository or pushed to it. It gives team members the ability to share images. For example, one can push an image to a Docker registry which can then be pulled by other team members to deploy the image to testing, staging and production environments.

Docker provides a public Docker registry called Docker Hub\(^{23}\). It provides a huge collection of existing images. One can use those public images directly to start a container or as a base to create its own image. If images might not be published to the wide public an organization can set up and run its own registry instance to host its images privately.

3.2.3 Docker Compose

Docker Compose is a tool to define and run multi-container applications. A user might for example develop an application consisting of three containers: (1) a database container, (2) a service API connecting to the database container and (3) a front-end application consuming the service API. Without tooling the user would need to start each container individually and in the correct order to get his application running. Docker Compose lets the user define his multi-container application in a single file. This file specifies the containers that need to be created, how they are linked, which ports get published etc. The user can then start his multi-container application with one single command. Docker will make sure the containers start in the correct order and with the correct links established between them.

3.2.4 Docker Swarm

In a single node setup with Docker Engine all Docker containers are started on the same node where the Docker daemon is running. This approach is not scalable and will not satisfy in a big data environment where containers will need to be deployed and managed at massive scale. Therefore Docker Swarm offers a clustering tool which turns a pool of Docker hosts into a single virtual host, i.e. all nodes in the cluster behave as one big machine to the user on which Docker containers can be deployed. Using Swarm a Swarm node agent runs on each cluster node and communicates with the Swarm manager. Requests to start, stop and remove containers are sent to the Swarm manager. The manager then schedules the container on one of the nodes following a configured ranking strategy and complying to the filtering constraints optionally passed by the user (e.g. the container might only be deployed on a cluster node with a flash drive).

3.3 Cluster Management

3.3.1 Definition

Big data computing puts high demands on computing power and resources. Few organizations can afford a mainframe, a single large source of processing power. Instead, many organizations are now choosing to invest in computer clusters which link high-powered, low-cost desktop computers in a network in order to take advantage of the parallel processing power of those computers. Computer clusters offer a number of benefits over mainframe computers such as scalability, availability, reduced cost etc.

Because of the many nodes in a cluster, it requires some management to make the nodes work together and to get applications running on it. This is where the cluster manager comes into play. A cluster manager is a component running on one or multiple nodes in a cluster.

\(^{23}\) https://registry.hub.docker.com
providing a GUI or CLI to the user to manage, configure and monitor the nodes in the cluster and the services running on it from a central location. The cluster manager retrieves information from the cluster by communicating with cluster agents running on each node.

The purpose of the cluster manager is to facilitate the management of the nodes in a cluster by making the cluster appear as one web server to the end-user. The cluster manager dispatches work to nodes in a transparent way to the administrator and gets back work results. The cluster manager might also ensure that the work is load balanced across the nodes in the cluster. A cluster manager also provides cluster maintenance operations like adding or removing nodes and rebalancing the work on the cluster. Next to cluster maintenance, the cluster manager might also be responsible for high availability including detection and recovering of failed cluster nodes.

3.3.2 Apache Mesos

Apache Mesos is an open-source cluster manager describing itself as a kernel for a distributed operating system. As illustrated in Figure 10: Cluster configuration using Mesos as Kernel, Mesos puts a kernel layer on top of the hardware, which might be servers available in a data-centre or hosted in the cloud, making the cluster behave as one big machine to the user. The idea behind Mesos is to give the user a similar experience whether developing on his own local machine or on a cluster.

![Figure 10: Cluster configuration using Mesos as Kernel](image)

As with a regular kernel system services need to be built on top of the kernel to make the system practical to the user. In Mesos those services are called frameworks. Mesos provides an API in three languages (Java, Python and C++) so users can develop their own framework that integrates with the Mesos kernel but a lot of frameworks are already publicly available in the sectors of data storage (e.g. Cassandra, ElasticSearch), big data processing (e.g. Hadoop, Spark, Storm), long running services (e.g. Marathon, Singularity) and batch scheduling (e.g. Chronos, Jenkins).

The two most frequently used frameworks are Marathon and Chronos.

---

24 Cluster configuration using Mesos as kernel (Source: Building Web Scale Apps with Docker and Mesos by Alex Rukletsov - http://www.slideshare.net/Docker/building-web-scale-apps-with-docker-and-mesos)
Marathon\textsuperscript{26} can be thought of as the startup or init daemon for a distributed OS. It automatically handles hardware or software failures and ensures that an application is always running. Marathon provides a web interface and a REST API for starting, stopping and scaling applications.

Chronos\textsuperscript{27} can be thought of as a cron for a distributed OS. It is a fault-tolerant job scheduler that supports complex job topologies making it more flexible than a regular cron. Chronos provides a web interface and a REST API to add, delete, list, modify and run jobs.

Both frameworks integrate with Docker. Marathon supports the deployment of dockerized applications. It can run, deploy and scale Docker containers with ease. Chronos in its turn supports scheduling docker jobs which run a finite command in a Docker container.

Mesos consists of a master daemon\textsuperscript{28} managing the slave daemons running on each node in the cluster. The frameworks built on top of Mesos consist of two components: (1) a scheduler that registers with the master to be offered resources and (2) an executor process that is launched on slave nodes to run the framework’s tasks. The resource offering in Mesos is shown at a high-level in (Figure 11: High-level flow of resource offering in Mesos) and is executed as follows:

1. A slave reports to the master the resources it has available.
2. The master sends a resource offer describing what is available to one of the frameworks. The framework is determined according to a configured allocation policy.
3. The framework’s scheduler replies to the master with information about the tasks it wants to run on the slave. For each task the framework indicates the required resources. The framework might also reject the resource offer sent by the master.
4. If the framework scheduler accepts the resource offer, the master sends the tasks to the slave, which allocates appropriate resources to the framework’s executor. The framework executor will then launch the tasks on the slave node.

\textsuperscript{26} https://mesosphere.github.io/marathon
\textsuperscript{27} https://github.com/mesos/chronos
\textsuperscript{28} In high-availability mode multiple masters are available: one active master (called the leader) and several backups in case it fails.
3.4 Deployment Conclusions

A deployment strategy combining Docker and Mesos meets the requirements listed in Section Following up the Platform Integration Status

As described above the BDE framework will start the existing distribution HortonWorks, enriched with additional components and with the *semantification* of already existing components. Because the BDE project needs to provide a coherent framework, some criteria need to be defined how the integration of a component in the framework can eventually be measured. The following provides an initial list which might be extended over time:

- The component is packaged using Docker (dockerized) so it can be deployed
- The component consumes RDF
- The component produces RDF
- The component is capable to provide provenance information
- The component preserves metadata
- The component is integrated in the general admin interface

The intention is to use this list to drive the QA for the framework as a whole.

Deployment Strategy. Docker provides a simple packaging tool which can be used for development as well as testing and production deployment. Furthermore, because Docker images consist of small layers, updating and rolling back components is an easy and rather lightweight operation. As the Dockerfile, from which a Docker image is built, is just a plain text file, this file (and its different versions) can easily be managed in a version control system like Git.

Selecting Docker as container also preserves some freedom in the packaging of the software. Docker containers might contain Debian packages, RPM packages, Java services, etc. so developers are not limited by choosing Docker. Moreover, Docker also brings an entire ecosystem with it including Docker Hub, Docker Compose and Docker Swarm.

---

One focus of the BDE platform is make deployment as easy as possible and re-use existing projects and infrastructure. For this reason, the tentative plan is to use Hortonworks packages as well as Linked Data Stack packages combined with other Debian package repositories. This results in a multi repository approach as outlined in the figure below in which the BDE package repository acts as a virtual layer (proxy) above existing work and packages specifically built within BDE.

Mesos provides a user-friendly cluster management tool with a web interface and a REST API which makes deploying and scaling applications on the cluster rather easy. There are already a number of frameworks available on top of Mesos of which Marathon and Chronos are the most important ones. Given that Mesos as well as Marathon and Chronos support the deployment and scaling of Docker containers makes Mesos an ideal cluster management tool to use in combination with Docker Engine and possibly Docker Swarm.

Figure 12: BDE Deployment Proposal

BDE Deployment Architecture Proposal (Tentative): The BDE platform repository consists of several Debian package repositories on top of which Docker scripts are used to provide images. Mesos is used to distribute tasks on clusters.

4. Conclusion

This deliverable has presented the high level design of the BDE platform. The BDE platform covers all aspects of the data challenges: Volume, Velocity, Variety, Veracity,
Variability and Complexity. It has been foreseen to semantify the some of the internal workings of the BDE platform (Section 2.3) in order to address those data challenges such as provenance and generic query answering that are known to be addressed well using a Linked Data approach. The degree to which the BDE applications make the linked data available to theirs will depend on the Pilot application requirement.

In the second part of the deliverable the software component selection process is being described. Based on our hands-on investigations, literature study and provided experience, it is possible to select open-source components to fully instantiate the BDE platform. However attention has to be given to the stability and the interoperability of the components. As each component has its own independent development cycle, finding an overall consist setup in which all components smoothly integrate is a tedious task. Therefore it has been decided to base our component selection for the BDE platform on the Hortonworks Data Platform distribution. This guarantees a base interoperability level. At the same time this choice supports our deployment strategy. A deployment strategy is one of the most challenging aspects: if not tackled in advance it represents a hidden cost which only emerges at the end of the solution building. By enforcing a deployment strategy this cost can be addressed while doing development which reduces the risks in the end.

The next steps are the realization of a first instance of the BDE platform on a shared platform using the first version of the deployment strategy. Using this experience the presented context will be fine-tuned in order to be ready for the first BDE platform roll-out.

References


**Nitin Sawant and Shah Himanshu** Big Data Aplication Architecture Q&A: A problem-solving approach [Report].

Appendices

Appendix 1: Terminology

The following is a first cut at defining the common terminology used throughout this deliverable. The description can be expected to change as the project progresses.

Architectural Design Pattern

An architectural design pattern\(^{30}\), refers to a solution to a common software architectural problem (i.e. the description of the data flow, type of component and positioning within the flow of information, etc.). Examples: would be the lambda architectural pattern and the BDE architectural pattern.

Lambda Architecture Design Pattern

This is an architectural design pattern for big data. The main characteristics of the architecture being, two flows of the data within the architectural components; one flow is targeting batch processing and the other real-time information flow.

BDE Architecture Design Pattern

This is a variant of the Lambda Architecture Design Pattern, but with semantic/RDF based components added.

BDE Framework

The architectural patterns required actual components to become concrete usable systems. At present, the component configurations and usage will require software development at many different levels.

In a framework, the component development is supported but there isn't necessarily a single or set of abstraction levels shared across the tools which the application developer use (knowledge of more tools will be required to use the framework).

BDE Platform

A software platform typically provides an abstraction layer. Usually, the abstraction layer is oriented towards a set of applications types and it is on this abstraction layer that the applications are built.

Although, some of the BDE parts will provide various abstraction layers, at present this will be at different levels. However, once pilot commonalities can be abstracted then the BDE platform will start to be created.

BDE Application

The BDE application will be the pilot applications, which are to be deployed. Each will be unique applications built within the BDE framework.

\(^{30}\) http://en.wikipedia.org/wiki/Architectural_pattern
Appendix 2: BDE Anticipated Users/Roles

It can be expected that there will be several general user roles involved with the BDE platform and the following will briefly summarise them.

**Component Developers**

These will be involved with the development of specific low-level components which will be used by the BDE platform itself (e.g. semantic search components, text analysers, etc). Their needs will be dependent on the development work to be undertaken, but at the very least they will need a version which can be easily installed on a development machine (assuming their development does not involve testing with live and large volumes of data). This role will also in some cases need to create the component service container (to wrap around the running and deployed component).

**Application Developers**

These will be at the external user level and will likely be making use of the facilities provided by the BDE platform to visualise the data which has been gathered together, etc. Their needs will be the same as the component developers, in that they will very likely be develop using a reduced system and then deploy their application later on the final acceptance/production system.

**System Deployer/QA/Testers**

These will essentially be responsible for making sure that the applications and components which have been developed are deployed correctly. This is a QA/testing role, which should be performed on an acceptance system configured as close as possible to the final production system. Depending on the size of the databases/system complexity this will be very component specific. Once tested the component would be moved to the production system (e.g. Debian has unstable, testing and stable views of the various packages which could be replicated here, but that will only be for the compilation/installation stages).

The development and testing machines should be re-installable at any point, while the production environment would need to be backed-up against corruption (if needed). The BDE infra-structure choice of Vagrant/docker based solution aids in this part of the QA/testing process, because system/service configurations should be completely described in text files (which should be version controlled). This will also simplify the creation of the installation guides, etc.

**Administrators/Operators**

This role will be responsible for making sure that the system keeps running on a day-to-day basis. Tools required will be those to monitor and manage the system processing, including things such as - pausing any data ingestion, moving tasks/processes, stopping/starter specific tasks/processes, etc. As well as upgrading the system components (or even downgrading a component). Monitoring will be needed for several aspects:

- performance of the various components,
- user access/data security/etc.

Additional to this will be a need for tools to make sure that usage limits can be set or the types of access can be controlled when services are made available for external users. An example of this, would be reducing the possibility of a denial-of-service attack by disallowing...

---

31 The cost of creating robust, auto-deployable docker/vagrant descriptions should not be under-estimated though (network might go down mid-deployment).
expensive data queries such as the listing of all rows in a large database. Another example would be rights to create new information or remove information from the databases.

**Application Users**
These will be using the system developed to view the data visualisations.