



City Research Online

City, University of London Institutional Repository

Citation: Mantzoukas, K. (2020). Runtime monitoring of security SLAs for big data pipelines: design implementation and evaluation of a framework for monitoring security SLAs in big data pipelines with the assistance of run-time code instrumentation. (Unpublished Doctoral thesis, City, University of London)

This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: <https://openaccess.city.ac.uk/id/eprint/25619/>

Link to published version:

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

Runtime Monitoring of Security SLAs for Big Data Pipelines

**Design, implementation and evaluation of a framework
for monitoring security SLAs in Big Data pipelines with
the assistance of run-time code instrumentation**



Konstantinos Mantzoukas

Supervisor: Prof. George Spanoudakis

Dr. Christos Kloukinas

Department of Computer Science

City University of London

This dissertation is submitted for the degree of

Doctor of Philosophy

November 2020

I would like to dedicate this thesis to my loving wife Anna and beautiful son Orestis

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Konstantinos Mantzoukas

November 2020

Acknowledgements

I would like to express my gratitude to my supervisor Professor George Spanoudakis for his unabating support and invaluable guidance throughout my research that led up to the authoring of this PhD thesis. I also wish to sincerely thank my second supervisor Dr. Chrtistos Kloukinas for all the assistance that he offered me during the conception, design and implementation of this dissertation. Finally I would like to assert my heartfelt appreciation to my family, friends and colleagues who never stopped believing in me and constantly encouraged me to keep going even in the darkest of hours.

Abstract

The Big Data processing ecosystem has been constantly growing in recent years. This has been significantly reinforced by the advent of cloud computing platforms where Big Data analytics can be offered on an as-a-service basis. The ease with which users can leverage the capabilities of Big Data processing frameworks in the cloud has made them a popular solution with low up-front expenditure and a flexible deployment model. In spite of their cost benefits and flexibility of use, Big Data services in cloud platforms present us with an array of new challenges compared to traditional web services especially in the domain of data security and privacy. Their distributed nature makes them more dynamic with regards to deployment and execution but at the same time it exacerbates challenges related to data and operation security since both data and operations are shared across multiple nodes. Inevitably, distributing data and operations on multiple nodes leads to an increase in the attack surface. Given the need for systems that react fast and produce results as quickly as possible, more emphasis has been placed on performance and less so on security. Having said that, as the use of cloud computing is becoming more widespread, concerns with regards to non-functional properties such as data security are becoming more pronounced for the users.

Runtime security monitoring is a mechanism that can be employed to alleviate some of the issues that emerge with respect to the activity of security monitoring for Big Data analytics services that are outsourced in the cloud. In this thesis we make the case for a monitoring framework where monitoring events are collected and evaluated against a set of monitoring rules that describe monitorable security properties of the system. The framework that we put forward can be used to assess the level of security of Big Data analytics pipelines at runtime. For our proof of concept we examine three security properties namely the service response time, the location of execution of service operations and the integrity of the intermediate data produced during the service execution.

Table of contents

List of figures	xv
List of tables	xxiii
1 Introduction	1
1.1 Overview	1
1.2 Motivation and Research Challenges	1
1.3 Summary of Research Aims and Objectives	3
1.3.1 Review the literature	4
1.3.2 Identify the monitoring framework's components	4
1.3.3 Identify monitorable security properties	4
1.3.4 Automate the translation of SLAs into monitoring rules	5
1.3.5 Automate the deployment of the event captors	5
1.3.6 Create an integrated SLA manager platform	5
1.4 Research Assumptions	6
1.5 Research Contributions	6
1.6 Publications	9
1.7 Thesis Outline	9
2 Literature Review	11
2.1 Overview	11
2.2 Security and Privacy Properties for Big Data	12
2.2.1 Data Availability	13
2.2.2 Data Privacy	18

2.2.3	Data Integrity	23
2.2.4	Data Confidentiality	28
2.3	Monitoring Service Level Agreements	33
2.4	Metrics for Service Level Agreement	41
2.5	Monitoring Frameworks for the Cloud	50
2.5.1	Commercial monitoring frameworks	53
2.5.2	Open source monitoring frameworks	57
2.6	Big Data Processing Frameworks	64
2.7	Big Data Workflow Definition Tools and Frameworks	80
2.8	Gap Analysis	87
2.9	Summary	89
3	Monitoring Framework for Big Data Security SLAs	91
3.1	Introduction	91
3.2	Framework Architecture	92
3.2.1	Composite Service Definition	97
3.2.2	Security Requirements Specification	100
3.2.3	Translation of Security Requirements into Monitoring artefacts . . .	100
3.2.4	Installation of Monitoring Rules on the monitor	101
3.2.5	Definition and Installation of Event Captors on Apache Spark . . .	101
3.3	Monitoring Rules	110
3.3.1	Monitoring Rules for Response Time	111
3.3.2	Monitoring Rules for Location of Execution	120
3.3.3	Monitoring Rules for Data Integrity During Service Execution . . .	132
3.4	Summary	173
4	SLA Management Web Dashboard	175
4.1	Application Architecture Overview	175
4.2	Application Repository	175
4.3	Application REST API	178
4.4	Energy producer use-case	182
4.5	Screenshots for the energy provider use-case	185

4.6	Summary	198
5	Framework Evaluation	199
5.1	Experimental setup	199
5.2	Quantitative Evaluation	202
5.2.1	Event captor deployment overhead	203
5.2.2	Event captor execution overhead	217
5.3	Evaluation Summary and Discussion	246
5.4	Summary	249
6	Conclusions and Future Work	251
6.1	Overview	251
6.2	Summary of Research Work	251
6.3	Contributions	252
6.4	Limitations	253
6.5	Future Work	253
	References	257
Appendix A	Composed Task Runner for Spark Submit Command	271
A.1	Spring Cloud Data Flow	280
A.1.1	Overview	280
A.1.2	Application Types	281
A.1.3	Workflow Specification Language	281
A.1.4	Application for the Execution of Apache Spark Jobs	286
A.2	Apache Spark	288
A.2.1	Overview	289
A.2.2	Framework Architecture	293
A.2.3	Execution Model	297
A.2.4	Deployment Model	299
A.3	EVEREST	299
A.3.1	Event Calculus	299
A.3.2	Framework Architecture	301

A.4	Apache Velocity	302
A.4.1	Overview	303
A.4.2	Velocity Template Language	303
A.4.3	Velocity Template Engine	306
A.5	Byte Buddy	308
A.5.1	Overview	308
A.5.2	Java's Instrumentation API	309
A.5.3	Runtime code instrumentation and Code Generation in Byte Buddy	311

List of figures

2.1	Lifecycle stages of data in the Cloud	19
2.2	QoSMONaaS system architecture	40
2.3	Apache Hadoop architecture overview	66
2.4	Map Reduce algorithm overview	67
2.5	An example of a Apache Strom topology	71
2.6	Worker processes for the topology presented in figure 2.5	72
2.7	Overview of stream in Apache Samza	75
2.8	An example of a Samza dataflow graph	76
2.9	Overview of the task state persistence mechanism in Apache Flink	77
2.10	Architecture of the Pinball workflow manger	83
2.11	State diagram for job statuses in Pinball	84
3.1	Big Data Pipeline Monitoring Framework Architecture	93
3.2	Use Case UML diagram of the Big Data monitoring framework	96
3.3	Sequence diagram of the Big Data monitoring framework	98
3.4	Spring Cloud DataFlow pipelines	99
3.5	UML class diagram of the factory pattern for the implementation of the different emitter types supported by the event captors	107
3.6	Visual representation of events for monitoring response time	113
3.7	List of actions supported by the event captor for response time	118
3.8	Example of events that occur over time during the monitoring activity of the location of execution of computations	124
3.9	Monitoring data integrity for transformations with narrow dependencies . . .	135
3.10	Monitoring data integrity for transformations with wide dependencies . . .	138

3.11	Example of events that occur over time during the monitoring activity of data integrity for actions and transformations with narrow dependencies	142
3.12	Example of events that occur over time during the monitoring activity of data integrity for transformations with wide dependencies	143
3.13	Interception component for HadoopRDD	161
3.14	Interception component for MapPartitionsRDD	162
3.15	Interception component for <i>runJob()</i> method in <i>SparkContext</i> class	163
4.1	SLA Manager web application architecture	176
4.2	SLA Manager database repository	178
4.3	Solar panel energy production use-case	183
4.4	Example of a measurement from a household	184
4.5	Spring Cloud DataFlow UI - Empty list of available applications	185
4.6	Spring Cloud DataFlow UI - Load applications from the command line . . .	186
4.7	Spring Cloud DataFlow UI - Populated list of available applications	186
4.8	Spring Cloud DataFlow UI - Create a new composite task from a drag-n-drop menu	187
4.9	Spring Cloud DataFlow UI - View of the composite task pipeline without the edges	187
4.10	Spring Cloud DataFlow UI - View of the composite task pipeline with the edges	188
4.11	Spring Cloud DataFlow UI - Edit the properties for the <i>LoadAndAnonymizeDataService</i>	189
4.12	Spring Cloud DataFlow UI - Properties for the <i>LoadAndAnonymizeDataService</i>	190
4.13	Spring Cloud DataFlow UI - Properties for the <i>PrepareDataService</i>	190
4.14	Spring Cloud DataFlow UI - Properties for the <i>ComputeAverageService</i> . .	191
4.15	Spring Cloud DataFlow UI - Type-in a name for the composite task	192
4.16	SLA Manager - Login to the SLA Manager	192
4.17	SLA Manager - List of SLA projects of the user	193
4.18	SLA Manager - View of the list of the service assets and security property pairs	193
4.19	SLA Manager - View of the asset/property pairs	194
4.20	SLA Manager - Type-in the parameter values for the SLO templates	195
4.21	Spring Cloud DataFlow UI - Launch the composite task	195

4.22	SLA Manager - Inspect the monitoring results for the location of execution security property	196
4.23	SLA Manager - Inspect the monitoring results for the data integrity security property for transformations with narrow dependencies	197
4.24	SLA Manager - Inspect the monitoring results for the data integrity security property for transformations with wide dependencies	197
5.1	Deployment time of data privacy event captor on the Spark master over the number of workers for 500K data points	204
5.2	Deployment time of data privacy event captor on the Spark workers over the number of workers for 500K data points	204
5.3	Deployment time of data privacy event captor on the Spark master over the number of workers for 1M data points	205
5.4	Deployment time of data privacy event captor on the Spark workers over the number of workers for 1M data points	205
5.5	Deployment time of data privacy event captor on the Spark master over the number of workers for 2M data points	206
5.6	Deployment time of data privacy event captor on the Spark workers over the number of workers for 2M data points	206
5.7	Overlay graph for the deployment of the data privacy event captor on the Spark master for different data sets size	207
5.8	Overlay graph for the deployment of the data privacy event captor on the Spark workers for different data sets size	207
5.9	Deployment time of data integrity event captor on the Spark master over the number of workers for 500K data points	208
5.10	Deployment time of data integrity event captor on the Spark workers over the number of workers for 500K data points	209
5.11	Deployment time of data integrity event captor on the Spark master over the number of workers for 1M data points	209
5.12	Deployment time of data integrity event captor on the Spark workers over the number of workers for 1M data points	210

5.13	Deployment time of data integrity event captor on the Spark master over the number of workers for 1M data points	210
5.14	Deployment time of data integrity event captor on the Spark master over the number of workers for 2M data points	211
5.15	Overlay graph for the deployment of the data integrity event captor on the Spark workers for different data sets sizes	212
5.16	Overlay graph for the deployment of the data integrity event captor on the Spark workers for different data sets sizes	212
5.17	Deployment time of data availability event captor on the Spark master over the number of workers for 500K data points	213
5.18	Deployment time of data availability event captor on the Spark workers over the number of workers for 500K data points	213
5.19	Deployment time of data availability event captor on the Spark master over the number of workers for 1M data points	214
5.20	Deployment time of data availability event captor on the Spark workers over the number of workers for 1M data points	214
5.21	Deployment time of data availability event captor on the Spark master over the number of workers for 2M data points	215
5.22	Deployment time of data availability event captor on the Spark workers over the number of workers for 2M data points	215
5.23	Overlay graph for the deployment of the data availability event captor on the Spark master for different data sets sizes	216
5.24	Overlay graph for the deployment of the data availability event captor on the Spark workers for different data sets sizes	216
5.25	Service execution time with and without data privacy monitoring on a cluster with 1 worker node for multiple data set sizes	217
5.26	Service execution time with and without monitoring data privacy on a cluster with 2 worker nodes for multiple data set sizes	218
5.27	Service execution time with and without data privacy monitoring on a cluster with 3 worker nodes for multiple data set sizes	218

5.28	Service execution time with and without data privacy monitoring on a cluster with 4 worker nodes for multiple data set sizes	219
5.29	Service execution time with and without data privacy monitoring on a cluster with 5 worker nodes for multiple data set sizes	219
5.30	Service execution time with and without data privacy monitoring on a cluster with 6 worker nodes for multiple data set sizes	220
5.31	Service execution time with and without data privacy monitoring on a cluster with 7 worker nodes for multiple data set sizes	220
5.32	Service execution time with and without data privacy monitoring on a cluster with 8 worker nodes for multiple data set sizes	221
5.33	Overlay graph for the service execution overhead of the data privacy event captor for different data sets on clusters with different number of workers .	222
5.34	Service execution time with and without data availability monitoring on a cluster with 1 worker node for multiple data set sizes	223
5.35	Service execution time with and without data availability monitoring on a cluster with 2 worker nodes for multiple data set sizes	223
5.36	Service execution time with and without data availability monitoring on a cluster with 3 worker nodes for multiple data set sizes	224
5.37	Service execution time with and without data availability monitoring on a cluster with 3 worker nodes for multiple data set sizes	224
5.38	Service execution time with and without data availability monitoring on a cluster with 4 worker nodes for multiple data set sizes	225
5.39	Service execution time with and without data availability monitoring on a cluster with 5 worker nodes for multiple data set sizes	225
5.40	Service execution time with and without data availability monitoring on a cluster with 6 worker nodes for multiple data set sizes	226
5.41	Service execution time with and without data availability monitoring on a cluster with 7 worker nodes for multiple data set sizes	226
5.42	Service execution time with and without data availability monitoring on a cluster with 8 worker nodes for multiple data set sizes	227

5.43	Overlay graph for the service execution overhead of the data availability event captor for different data sets on clusters with different number of workers	228
5.44	Service execution time with and without data integrity monitoring using MD5 on a cluster with 1 worker node for multiple data set sizes	229
5.45	Service execution time with and without data integrity monitoring using MD5 on a cluster with 2 worker nodes for multiple data set sizes	230
5.46	Service execution time with and without data integrity monitoring using MD5 on a cluster with 3 worker nodes for multiple data set sizes	230
5.47	Service execution time with and without data integrity monitoring using MD5 on a cluster with 4 worker nodes for multiple data set sizes	231
5.48	Service execution time with and without data integrity monitoring using MD5 on a cluster with 5 worker nodes for multiple data set sizes	231
5.49	Service execution time with and without data integrity monitoring using MD5 on a cluster with 6 worker nodes for multiple data set sizes	232
5.50	Service execution time with and without data integrity monitoring using MD5 on a cluster with 7 worker nodes for multiple data set sizes	232
5.51	Service execution time with and without data integrity monitoring using MD5 on a cluster with 8 worker nodes for multiple data set sizes	233
5.52	Service execution time with and without data integrity monitoring using MD5 on a cluster with 8 worker nodes for multiple data set sizes	233
5.53	Overlay graph of the overhead(%) over different number of workers for data integrity monitoring using MD5	234
5.54	Service execution time with and without data integrity monitoring using SHA-1 on a cluster with 1 worker node for multiple data set sizes	235
5.55	Service execution time with and without data integrity monitoring using SHA-1 on a cluster with 2 worker nodes for multiple data set sizes	235
5.56	Service execution time with and without data integrity monitoring using SHA-1 on a cluster with 3 worker nodes for multiple data set sizes	236
5.57	Service execution time with and without data integrity monitoring using SHA-1 on a cluster with 4 worker nodes for multiple data set sizes	236

5.58	Service execution time with and without data integrity monitoring using SHA-1 on a cluster with 5 worker nodes for multiple data set sizes	237
5.59	Service execution time with and without data integrity monitoring using SHA-1 on a cluster with 6 worker nodes for multiple data set sizes	237
5.60	Service execution time with and without data integrity monitoring using SHA-1 on a cluster with 7 worker nodes for multiple data set sizes	238
5.61	Service execution time with and without data integrity monitoring using SHA-1 on a cluster with 8 worker nodes for multiple data set sizes	238
5.62	Overlay graph of the overhead(%) over different number of workers for data integrity monitoring using SHA-1	239
5.63	Service execution time with and without data integrity monitoring using SHA-256 on a cluster with 1 worker node for multiple data set sizes	240
5.64	Service execution time with and without data integrity monitoring using SHA-256 on a cluster with 2 worker nodes for multiple data set sizes	240
5.65	Service execution time with and without data integrity monitoring using SHA-256 on a cluster with 3 worker nodes for multiple data set sizes	241
5.66	Service execution time with and without data integrity monitoring using SHA-256 on a cluster with 4 worker nodes for multiple data set sizes	241
5.67	Service execution time with and without data integrity monitoring using SHA-256 on a cluster with 5 worker nodes for multiple data set sizes	242
5.68	Service execution time with and without data integrity monitoring using SHA-256 on a cluster with 6 worker nodes for multiple data set sizes	242
5.69	Service execution time with and without data integrity monitoring using SHA-256 on a cluster with 7 worker nodes for multiple data set sizes	243
5.70	Service execution time with and without data integrity monitoring using SHA-256 on a cluster with 8 worker nodes for multiple data set sizes	243
5.71	Overlay graph of the overhead(%) over different number of workers for data integrity monitoring using SHA-256	244
5.72	Overlay graph of the overhead(%) over different number of workers for data integrity monitoring using MD5, SHA-1 and SHA-256	245

5.73 Overlay graph of the average overhead(%) for all the security properties over different data set sizes	248
A.1 Pipeline of tasks executed in sequence	283
A.2 Pipeline of tasks executed in sequence	283
A.3 Pipeline of tasks with transitions for simple tasks	284
A.4 Pipeline of tasks with transitions before a sequence of tasks	285
A.5 Pipeline of tasks with wildcards	285
A.6 Pipeline of tasks launched in parallel	286
A.7 Pipeline of tasks launched in parallel that are connected to a sequence of tasks	286
A.8 Add a new Spring Cloud Data Flow application of type task	287
A.9 List of all the installed Spring Cloud Data Flow applications in the application registry	287
A.10 Example of a Spring Cloud Data Flow pipeline	288
A.11 RDD with its partitions	291
A.12 Parent and child RDD with applied operation and dependencies	291
A.13 <i>map()</i> operation - transformation with narrow dependencies	291
A.14 <i>groupByKey()</i> operation - transformation with wide dependencies	293
A.15 <i>count()</i> operation - return the number of items on an RDD to the user . . .	294
A.16 Apache Spark overall architecture	294
A.17 Example of a Directed Acyclic Graph (DAG)	295
A.18 Example of a Directed Acyclic Graph (DAG) with jobs, stages, tasks, RDDs and partitions	298
A.19 EVEREST framework architecture [84]	302
A.20 Overview of the code instrumentation process from Bute Buddy	309

List of tables

2.1	SLA metrics defined in the Cloud Service Level Agreement Standardisation Guidelines report in [135]	50
2.2	Evaluation of commercial monitoring tools and frameworks across a set of monitor attributes	56
2.3	Evaluation of open source monitoring tools and frameworks across a set of monitor attributes	63
2.4	Types of grouping in Apache Storm	74
2.5	Comparison of Big Data processing frameworks	79
2.6	Comparison of workflow management frameworks	86
3.1	Events collected for monitoring response time - start job event	111
3.2	Events collected for monitoring response time - end job event	112
3.3	Event Calculus rule for monitoring response time	112
3.4	Example of events for monitoring response time	113
3.5	Events collected for monitoring the location of execution - compute event .	122
3.6	Example of events collected for monitoring the location of execution	122
3.7	Event calculus assumption and rule for monitoring the location of execution of computations	123
3.8	Writerdd events for monitoring data integrity for transformations with narrow dependencies	136
3.9	Read events for monitoring data integrity for transformations with narrow dependencies	137
3.10	Example of writerdd events collected for monitoring data integrity of transformations with narrow dependencies	137

3.11	Example of readrdd events collected for monitoring data integrity of transformations with narrow dependencies	137
3.12	Write shuffle events for monitoring data integrity for transformations with wide dependencies	139
3.13	Read shuffle events for monitoring data integrity for transformations with wide dependencies	140
3.14	Event calculus assumption and rule for monitoring the runtime data integrity for actions and transformations with narrow dependencies	141
3.15	Event calculus assumption and rule for monitoring the runtime data integrity for transformations with wide dependencies	142
4.1	Operations of the SLA Manager RESful API	182
5.1	Hardware information for the Google VM instance host machine	201
5.2	Average overhead for monitoring data privacy for different data set sizes and number of worker nodes	221
5.3	Average overhead for monitoring data availability for different data set sizes and number of worker nodes	227
5.4	Average overhead for monitoring data integrity using MD5 for different data set sizes and number of worker nodes	234
5.5	Average overhead for monitoring data integrity using SHA-1 for different data set sizes and number of worker nodes	239
5.6	Average overhead for monitoring data integrity using SHA-256 for different data set sizes and number of worker nodes	244
5.7	Summary table of the average deployment time for the event captors on clusters with different number of workers	246
5.8	Summary table of the average overhead for availability, privacy and integrity monitoring for different data set sizes	247
A.1	Status values for Spring Cloud Data Flow tasks	282
A.2	Parameters for the Apache Spark submit application registered in Spring Cloud Data Flow	289
A.3	Event Calculus list of predicates	300

A.4	Event Calculus axioms	301
A.5	Velocity template language references	304
A.6	Velocity template language directives	306
A.7	Options for manifest file of Java agents	311

Chapter 1

Introduction

1.1 Overview

In this thesis we argue that it is possible to design, implement and evaluate an end-to-end monitoring framework that automates the monitoring activity of security properties at runtime for Big Data analytics pipelines. The proposed framework will enable the automatic translation of high-level security requirements for Big Data pipelines into low-level monitorable artefacts that can be automatically deployed and monitored. The thesis presents the state of the art in the domain of service level agreement (SLA) monitoring frameworks and uses it to describe the architecture of the proposed framework and its constituent components. In addition, we assess the framework's ability to monitor real-life applications by means of using a use-case from the domain of Internet of Things where we examine the framework's capacity to perform runtime monitoring of three non-functional properties pertaining to data availability, data privacy and data integrity. Finally, we conduct a quantitative evaluation on the basis of the time and computational overhead that the event capturing activity imposes on potential monitoring targets.

1.2 Motivation and Research Challenges

The requirement for the analysis of large datasets has become a challenging problem in recent years [73]. An ever increasing amount of data is produced both from autonomous

agents and humans alike. This development has made Big Data processing a priority for most businesses and organisations. IT practitioners, researchers and policy makers can tap into new insights that they can only gain by analysing Big Data [126]. In that regard, users are allowed to make inferences that would otherwise be impossible.

Moreover, the commoditisation of computational resources as a result of the improvements in cloud computing, has streamlined the development and deployment of Big Data applications [158] [20] in cloud infrastructures. Users that otherwise would not be able to afford, both financially and in terms of human resources, the computational power of thousands of virtual machines, now can have them provisioned as a service easily and quickly through web dashboards. Having said that, to meet those increased requirement for Big Data processing, a plethora of processing frameworks have been designed and developed that parallelise the execution of operations of large datasets [155] [131] [94] [51].

Market competition and the need for the analysis of very large datasets has placed a disproportionate emphasis on building systems that focus almost exclusively on performance and less so in non-functional properties such as data security and privacy. However, as Big Data processing is offered more and more as a service, it becomes imperative that the challenges of assessing security properties system and application are becoming more pronounced and need to be addressed from the scientific community [47] [57]. Using Big Data services that are outsourced in the cloud mandate that they are executed in an environment that the users can trust to be secure from data breaches that can potentially lead to degradation of reputation or financial losses.

A significant body of literature has made an attempt to address the importance of security for cloud deployed services [71]. The definition of service level agreements(SLAs) is a common theme that we have been able to identify in the state of the art [26]. However, there is no concensus in the literature with regards to standards for security SLAs for applications executed in the cloud. Also, most of the security SLA frameworks focus on the operation and negotion phases of the lifecycle of SLAs and do not get into the specifics of what metrics they monitoring and how they do it. In addition, most of the security SLAs examine metrics that relate to data storage such as access control management and encryption [140] [145] [75]. However, all the proposed frameworks address the issue of safeguarding the data and not continuously monitoring the preservation of specific properties as the data gets processed

or stored. They introduce techniques such as homomorphic encryption [60] or complicated mechanism for key sharing but they do not address the challenges of runtime security monitoring. Moreover, in the state of the art, monitoring security SLAs is only limited to services that run on the cloud without necessarily being executed in a distributed fashion like Big Data applications do [35] [108] [28] [29]. Typically this point is glossed over or not mentioned at all. We argue that the nature of the execution model of the underlying service for which security properties need to be monitored, is of paramount importance and present us with a completely different set of security challenges. Distributed applications operate on data that is scattered across multiple nodes. Also, operational code needs to be transmitted over the network to get executed on different locations within a cluster. Both of those unique features of distributed applications pose a set of research questions that according to our literature review have not been studied in the context of runtime security monitoring.

To address some of the limitations that we have been able to identify in the literature with respect to runtime monitoring of security properties, in this thesis we put forward a generic monitoring framework that focuses on the runtime monitoring of security properties for Big Data pipelines. In the proposed framework we automate the continuous evaluation of an application's data availability, data privacy and data integrity. We explicitly defined manifestations of the aforementioned properties by means of using Event Calculus [91] expressions. Key features of the system that we propose are that we firstly automate the generation and installation of the monitoring rules on the monitoring engine from a set of high-level security specifications and secondly we automate the deployment of the relevant event captors that are necessary for the realisation of the monitoring process. Also, our monitoring system can adopt to the changes in the execution environment of the service, which makes it suitable for monitoring applications that run in the cloud.

1.3 Summary of Research Aims and Objectives

The aim of our thesis is to address the lack of end-to-end security SLA monitoring platforms for Big Data service pipelines by describing and developing a complete framework that deals with all the stages of the monitoring process. In the general case the monitoring process starts with the definition of the security requirements, moves on with the generation

of the monitoring artefacts that are necessary to realise the monitoring of the security requirements, proceeds to collect the monitoring data and eventually evaluates whether the security requirements has been honoured or not. To attain those objectives the following key targets were pursued:

1.3.1 Review the literature

Our first objective is to critically examine the the literature and review any existing approaches that attempt to solve the same or similar problems. More specifically, we are interested in the state of the art for SLA monitoring and Big Data processing frameworks. From our analysis we intend to provide to the reader a clear view as to what systems exist in the domain of Big Data service monitoring and what are the issues that are not addressed by the state of the art.

1.3.2 Identify the monitoring framework's components

Our second objective is to identify all the components of the monitoring framework that are necessary to realise the monitoring activity for Big Data pipelines. Having a clear view of the individual modules that are involved will help to streamline the definition of all the necessary information that will facilitate the automation of the monitoring activity.

1.3.3 Identify monitorable security properties

Our third objective is to identify a set of monitorable security properties that are appropriate to be monitored for Big Data application pipelines executed in a distributed environment. For the assessment of the proposed framework we will examine three security properties; the first one will be the response time of a Big Data service that relates to availability, the second one will be the location of execution of operations of a Big Data service that relates to data privacy and the third one will be the preservation of data integrity for the intermediate data that is produced during the execution of a Big Data service that relates to data integrity. For the definition of the monitoring rules that correspond to the monitoring conditions for each property we will be using Event Calculus formulae.

1.3.4 Automate the translation of SLAs into monitoring rules

Based on the objective listed in section 1.3.3, our fourth objective will be to describe the process and build the tools that are responsible for the automatic translation of the Service Level Objectives(SLOs) of Service Level Agreements(SLAs) into EC-Assertion expressions. EC-Assertion is the language specification that our monitoring engine is using to define the monitoring rules. The translation will be conducted with the assistance of pre-compiled templates that will describe the EC-Assertion specification for the property that will be monitored. Certain parts of the template will be fixed whereas others will be parameterised and input from the user will be required to enable their concrete instantiation.

1.3.5 Automate the deployment of the event captors

As soon as our objective to automate the translation to automate the instantiating of the monitoring rules presented in section 1.3.4 has been realised, our fifth objective is to automate the deployment of the event captors on the Big Data processing framework where the Big Data service pipeline will be executed. The automatic deployment of the event captors will be the final step for the end-to-end automation of the monitoring activity that commences with the definition of a set of service level objectives and concludes with the execution of the monitoring activity.

1.3.6 Create an integrated SLA manager platform

Finally, our sixth and final objective is to provide a comprehensive set of tools for the interaction of the end-users with the monitoring platform that will be appropriate for technical and non-technical users. In our proposal we will implement a web application with a graphical user interface that we refer to as the SLA manager, that will serve two main purposes. The first one will be to enable the collection of Service Level Objective(SLO) parameters from the users for the creation of concrete instances of the monitoring rules from the parameterised templates. The second one will be to allow the inspection of the monitoring results that are produced by the EVEREST monitor. EVEREST [123] an event reasoning engine that given a set of rules can reason about whether those rules are respected or not based on a given set of events. The rules are expressed as a set of event calculus [91] expressions before the

monitoring activity has commenced. The monitoring results will become available during the Big Data pipeline execution or after it has completed.

1.4 Research Assumptions

To conduct our research and to enable the development of the tools that will allow us to monitor security properties for Big Data pipelines, the following assumptions are to be made:

1. The events that are captured from the event captors need to be sent to the EVEREST monitor in a specific format that the monitor can understand and extract useful information from their payload. The events are stipulated as self-contained XML snippets.
2. The nodes in the cluster, where the Big Data processing framework operates on, must have their clocks synchronised as accurately as possible. This property is critical for the correct evaluation of the monitoring events that are correlated with time constraints in the monitoring rules. A potential solution to this issue could be achieved if the Network Time Protocol(NTP) protocol is used and all the nodes are synchronised with an external time server. The Network Time Protocol synchronises all participating nodes to within a few milliseconds which is accurate enough for most cases.
3. End-users must have a web browser to be able to access the SLA manager web application. Also, they need to be fairly acquainted with the navigation in a web application and must have a minimum understanding of what security properties they wish to monitor.

1.5 Research Contributions

In our research we aim to design, implement and evaluate a monitoring framework for the runtime monitoring of security properties for Big Data pipelines that we refer to in this thesis as **composite services** which are comprised of Big Data services that we refer to in this thesis as **atomic services**. Our proposal is a framework that enables the continuous runtime

monitoring of security objectives defined as parts of an SLA that is defined from the users of composite services.

The contributions of our work in this thesis are:

1. **Designed and developed a monitoring framework for the automatic translation of high-level security requirements into low-level monitorable artefacts that are then automatically monitored.** The monitoring artefacts refer firstly to the monitoring rules that need to be generated and loaded on the monitor that will evaluate the monitoring events against the rules and secondly to the event captors that will have to be deployed alongside the Big Data services to collect the appropriate monitoring events. In similar frameworks, the rules are not generated automatically but are declared manually from the users, a task that can be tedious and highly technical. In addition, the relevant event captors are not deployed automatically and require the explicit installation of monitoring agents that collect the monitoring data. We argue that the automatic generation of the monitoring rules from high-level security objectives and subsequently the automatic deployment of the event captors, is significant contributions because they do not mandate from the end-users to have any understanding of the intricacies with regards to the Big Data service that they wish to monitor. This is a key point that enables non-technical and less security-savvy users to leverage the capabilities of monitoring to inspect the security of Big Data applications at runtime. Also, our proposed system is unique because it addresses security monitoring not for Web Services, which is a topic that is widely researched, but for Big Data pipelines that are executed in a distributed fashion. The distributed nature of the execution model of Big Data services, and more specifically of pipelines of services, bring in a plethora of additional security challenges both with respect to the definition of the monitoring rules and the deployment of the event captors. Addressing this lack of tools and frameworks that automate the monitoring process of security for distributed applications is a key contribution of our research.
2. **Designed and develop a monitoring framework where the event capturing process is immune to changes both in terms of how the atomic services of a composite service are arranged and in terms of the actual code of the atomic services.** The

deployment of the event captors is performed by means of associating the monitoring target Big Data services with Java agents that perform runtime code instrumentation at the underlying Big Data processing framework. The dynamic nature of the Java agent technology that we employed to build the event captors, allows our framework to run alongside the Big Data service that it produces events for. Modifications in the pipeline will not affect the deployment of the event captors or their ability to correctly collect the appropriate runtime information since they instrument the underlying code of the Big Data processing framework and not the atomic services of the Big Data pipeline itself. Similar works have been proposed that use Java agents or code instrumentation techniques to achieve the same objective but not in the domain of distributed applications. As such, we regard our contribution novel because we address this issue in the context of Big Data services. In Big Data applications, computations are executed in parallel and across multiple nodes and therefore multiple event captors need to be installed to capture all the relevant monitoring events.

3. **Designed and developed event captors that are adaptive and elastic.** The event captors in our proposed solution can adapt to changes in the layout of the cluster or its configuration. If additional nodes are included in the cluster the right event captors will get installed at the newly available nodes automatically. By the same token, if a node becomes unavailable the previously installed event captor on it will not emit any events and will be dismissed. This feature is congruent with the nature of Big Data services that get executed in an elastic environment such as the cloud, where scaling up and down is a common practice. Having said that, adaptability and elasticity of the event capturing process is of paramount importance for the uninterrupted and accurate monitoring of security SLAs for Big Data services. Similar monitoring frameworks do not address the lack of elasticity in the way the monitoring data is collected because they do not refer to distributed applications. In non-distributed applications elasticity of the event captors is not an issue because the resources that the applications use are static and are defined when the application begins executing.

1.6 Publications

The material of this thesis has also been the topic for a publication in an IEEE conference:

Mantzoukas K., Kloukinas C., Spanoudakis G. *Monitoring Data Integrity in Big Data Analytics Services*, 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), July 2018

1.7 Thesis Outline

This thesis is composed of seven chapters including this chapter. The layout of the thesis is the following:

- **Chapter 2** presents a review of the literature in the domain of SLA monitoring and the relevant frameworks for the definition of Big Data service pipelines and Big Data processing frameworks. At the end of the this chapter we perform a gap analysis where we identify a series of issues that have not been addressed in the current state of the art.
- **Chapter 3** focuses on the technical background of our work and provides an overview of the tools and frameworks that were used for the implementation of our proof of concept of the proposed monitoring framework. More specifically, we used the EVEREST event reasoning toolkit for the monitoring, Spring Cloud DataFlow as the Big Data pipeline definition and execution framework, *Apache Spark* as the Big Data processing framework, Apache Velocity for the definition templates for the monitoring rules of the security properties that we examined and finally Byte Buddy for the development of the event captors that will instrument the code for the collection of the monitoring data.
- **Chapter 4** gives a detailed account of the monitoring framework that we propose. In this chapter we describe its architecture and how the users will interact with it. In addition, we describe the monitoring rules in event calculus for three security properties that we used to evaluate our proof of concept implementation. The security properties that we examined are response time, location of execution of operations and data integrity of the produced intermediate data. Finally, for each property we give a

detailed description of the EC-Assertion template that will be used to create instances of the the corresponding monitoring rule and we also go through the event captors that will be responsible to gather the relevant monitoring data.

- **Chapter 5** presents the SLA management web dashboard that enables end-users to interact with the monitoring platform. In particular, we give an overview of the system where we describe the data repository that it uses to persist the user’s input for the definition of the service level objectives that require to be monitored and we also describe the RESTful API that has been implemented to facilitate the integration of the web application’s front-end with the repository. Finally, we describe a use-case from the domain of Internet of Things(IoT) and we execute it. All the relevant screenshots of the example use-case are also presented for a more complete view of the dashboard.
- **Chapter 6** focuses on a quantitative evaluation of the proof of concept framework that we built. Our evaluation is performed on the basis of the overhead that the monitoring activity incurs on the Big Data pipelines that it ought to continuously monitor. Our analysis identified two separate steps that are involved for the successful monitoring activity namely the deployment of the event captors and the execution of the event captors. In our evaluation we examine both of those steps with regards to the overhead that they impose on the monitoring target. Finally, we provide a summary of the results and discuss our findings.
- **Chapter 7** is the final chapter and is a recap of the monitoring framework for security SLAs for Big Data pipelines that we put forward. Finally, in this chapter we provide a list with the limitations of our approach that we were able to identify and describe the directions for future work on security SLA monitoring for Big Data services.

Chapter 2

Literature Review

2.1 Overview

In this chapter we will provide a comprehensive view of the state of the art in the domain of security service level agreements in the Cloud. Our goal is to lay out the existing body of literature and acquaint the reader with the relevant body of work on the topics of security SLA monitoring and Big Data processing frameworks. The overarching objective of this chapter is to establish a solid scientific foundation that will enable us to highlight the originality and novelty introduced by this thesis.

More specifically, section 2.2 presents a review of the definition and specification of security properties in an attempt to elucidate the composition and monitoring of SLAs by means of using monitorable security properties. Section 2.3 reviews frameworks that have been presented in the literature for the monitoring of SLA with an emphasis on security and provides an account of their characteristics in each case. Section 2.4 surveys the literature in an attempt to present the metrics that existing systems in SLAs. Moreover, in section 2.5, we list a series of monitoring solutions that have been implemented to support the monitoring activity in distributed application that run in the Cloud. Finally section 2.6 surveys the state of the art with regards to Big Data processing frameworks and section 2.7 studies the available technologies in the space of Big Data workflow definition technologies that will enable the definition of Big Data pipelines.

2.2 Security and Privacy Properties for Big Data

Application security and privacy has always been a subject of great importance for computer scientists and IT practitioners [96]. It is a topic that has been studied thoroughly and more so in the day and age of Cloud Computing [159]. In the cloud, computational resources and software components are offered as a service to the users and therefore a significant portion of the controls over the data and the applications is shifted from the users to the service and infrastructure providers. Apart from its significance for users, the issue of security and privacy has become a requirements in many parts of the world such as the countries of the EU through the General Data Protection Regulation (GDPR) that came to effect in May 2018, Australia through the Privacy Act of 1988 and the Australian Consumer Law (ACL), China through the 2017 Cyber Security Law which controls the operations of network operators and critical information infrastructure operators, Japan through the Act on the Protection of Personal Information (APPI) which mandates that the private sector is responsible for the protection of personal information from unauthorized usage and Russia through the Roskomnadzor, the Russian Data Protection regulator which describes a legal framework that requires all data stored regarding Russian citizens should be physically located with Russia.

Even though some new approaches are starting to emerge [82] [44], the most prevalent model for the evaluation of the level of security for a system or application is performed by means of assessing the triad of Confidentiality, Integrity and Availability (CIA) [105] [149] [98] [72]. The focus in the CIA approach is the definition of security in the context of the evaluation of the simultaneous presence of those three properties i.e. a system is regarded secure if it bears all three properties at the same time. A more formal definition of the CIA model would be: *A piece of data D is secure if, and only if, all parts of D retain the properties of confidentiality, integrity, and availability*

Before we continue with our analysis we will give a high level description of each one of the properties in order to provide a common understanding of the terminology, at least the way it is being used in the literature.

- **Confidentiality:** The data is not made available or revealed to unauthorized entities that could be individuals or processes.

- **Integrity:** The data is accurate, complete and has not been modified by an unauthorized entity.
- **Availability:** The data is rendered accessible upon request by an authorized entity.

In the subsections that follow, we expand on the properties mentioned above while we also review the literature on service data privacy. Even though data privacy is not part of the CIA (Confidentiality, Integrity, Availability) triad, it still is relevant in the context of data security especially in the discussion about personally identifiable information(PII). Data privacy is a rather interesting property that addresses the issue of protection of individuals from data inferences that might result from analytics performed on a data set that can have references to them. For completeness, in our proposal we chose to use security properties from both domains i.e. data security and data privacy. More specifically, from the data security domain we examined data integrity of the intermediate data that is produced when data analytics pipelines get execute that pertains to data integrity and the time it takes for a Big Data analytics service to make its results available that pertains to data availability. Finally, from the data privacy domain, we address the challenges of monitoring the locations where data can be processed. In that way we can make sure that the privacy of individuals can be protected by allowing to be processed in a predefined set of locations.

2.2.1 Data Availability

Availability pertains to the system's ability to make accessible its resources on demand. More specifically, data availability pertains to the system's ability to make accessible the data that it will produce as a result of the system's operation. The system has to be able to continue operating under any circumstances even when parts of it misbehave or fail to execute properly. This implies in order to enhance data availability systems needs to be fault tolerant and capable to recover on the off chance of a sub-system failure. Also, data availability should not be affected by security breaches that have occurred. This becomes more pronounced in the Cloud where where data services are exposed to multiple users and the requirements for data security and availability are of paramount importance. Lee *et al.* [78] propose the usage of a *Cloud-of-Clouds* where resources are redundant across multiple Cloud providers and can be used dynamically to meet system availability requirements. The authors propose the

integration of the availability requirements included in Service Level Agreements (SLAs) with a collection of Cloud providers to ensure that the SLA objectives are respected. In the same spirit, Bowers *et al.* [33] introduce HAIL (High availability and integrity layers) which is a distributed cryptographic file system implementation that can provide quantifiable guarantees with regards to file integrity and retrievability. HAIL achieves its objective by keeping redundant copies of the files across multiple nodes and by randomly checking portions of the file for integrity violations and therefore can be retrieved by the user without any loss of corruption. The authors highlight the criticality of not only making the data available but also minimizing the possibility of it being corrupted.

Similar to HAIL, Juels *et al.* [74] describe a proof of retrievability scheme where Cloud service providers can provide concise proof to its users that they can rigorously retrieve and make available a file in its entirety. The authors make a specific case for large files where optimizations need to be made to minimize the network overhead of computing proof of knowledge for multiple partitions of the file.

A more theoretical analysis for the definition of service and data availability has been conducted by Hogben and Pannetrat in [68]. In their work they underline the challenges of defining and measuring availability in the context of a Service Level Agreement in an unambiguous manner. The motivation for their work stems from the need for users to compare service offers on the basis of service and data availability. They make the case that the way availability is monitored and evaluated can make two Cloud service providers to report 0% availability and 100% availability for the exact same service. To address this ambiguity the paper delineates the basic criteria for the definition of service attribute definition should be the following:

- **Well-defined:** Accuracy in the definition is critical for the avoidance of ambiguities. e.g. what *downtime* really means, how many failures to utilize a service render a service or its data unavailable, etc..
- **Correlated to consumer utility:** Availability measurements need to be mapped to measurable business value for its users. This means that the users' experience when interacted with the service should reflect the measurement of availability. E.g. a service is responsive for 1 minute during 5 minute windows will be 100% available if the time

scope is set to 5 minutes. The same service would offer a poor experience to users that need to make multiple service invocations within the 5 minute window intervals.

- **Standardized:** Service providers need to standardize the definition of the SLA attributes to enable both users and providers to more accurately evaluate and compare them across different providers.
- **Determinate:** Success criteria are as equally important as failure criteria. E.g. if a service is unavailable due to bad network connectivity of terminal that performs the service invocation it doesn't mean that the service is unavailable. This example make it obvious that there is a strict requirement for a thorough definition of setting under which the attribute of availability is measured. In the example presented this could be expressed as the need for the invocation the service from a specific terminal or location.

In the same spirit, Snow and Weckman in [122] argue that availability is poorly defined and that is the source for a misunderstanding between service availability guarantees. They point out that Cloud service providers casually offer service availability in what is know as the *five nines* i.e. 99.999% but they do not explain what that means and most importantly they do not provide a probabilistic distribution of service availability. The point they make is that measuring availability as a mean value does not provide enough insight to the service user and that a probabilistic distribution of availability is a more tangible measurement. In their work, the authors used Monte Carlo simulations to investigate availability distributions for the *five nines* availability use case where they used two uncertainty variables namely Mean Time to Fail (MTTF) and Mean Time to Recover (MTTR). They run 3000 simulations and they were able to demonstrate that the chances of five nines availability violations are significant and that the combination of reliability and maintainability is the most salient determinant for the risk of such a violation.

By the same token, Lorenzoli and Spanoudakis in [81] used MTTR and MTTF as expressions of service availability and provided two probability distribution functions that describe the probability of violating a specific threshold. To run their experiments they used monitoring data from the Yahoo internet page service and they gathered 5500 requests and responses respectively. They had been able to show that the measurements for both metrics

of availability improved when shorter prediction periods were selected which is expected because it increases accuracy and therefore both precision and recall improve.

Addressing service and resource availability in the domain of Cloud computing, Mateo-Fornes *et al.* [87] attempt to address the challenges of availability and response time in conjunction with performance, cost and response time. In their paper the authors propose what they call the cloud availability and response time(CART) model that is responsible for the procurement of the required computational resources in order to guarantee the predefined response times stipulated in SLAs, minimizing at the same time the cost incurred to the users. To predict service response times they use queueing theory and more specifically a Poisson traffic model. They model the Cloud as a set of N virtual machines that execute b number of tasks assigned by the users. The model that they put forward treats availability as Markov model where time between failures of virtual machines are distributed exponentially.

A practical approach for ensuring availability specifically for big data is offered by Payfair *et al.* [102]. In this work an approach for a selective partial checkpointing mechanism is presented for the execution of queries in in-memory databases. In particular, they address the challenges that are presented in column-oriented databases that run queries against large data sets. Their model attempts to select the right checkpoints during the execution plant that is produced when a query is sent for execution taking into consideration that an adversary can bring down a node that will have the worst impact in terms of execution time. Currently their solution is able to obtain only one checkpoint.

To guarantee data availability, services rely on the underlying software and hardware stack. Since the Cloud is primarily built on virtual machines, virtual machine availability is critical for the honoring of data availability guarantees stipulated in SLAs. Gonzalez *et al.* in the their work in [63] study the challenges of data availability in relation to the Cloud provider's size, redundancy and fault tolerance with regards to virtual machines. They argue that in practice there are two types of failures; i) single server downtime due to hardware or software failures and ii) multiple servers downtime where several server become unavailable or an entire rack. Also, they propose that virtual servers can be in one of the three following states:

- **Active servers:** Servers that host virtual machines already running

- **Spare servers:** Servers that are operational and can be used to host virtual machines if required
- **On-Repair servers:** Servers that are non-operational and cannot be considered for the allocation of virtual machines

Virtual servers move between those three states. In the paper the authors analyse the impact of three different types of fault tolerance techniques namely vSphere fault tolerance [12], vSphere High Availability [10] and vSphere Data Recovery [11]. Finally, the paper concludes with the description of a policy model for the reduction of risk of violation of availability in SLAs called SLA-budget. The SLA-budget refers to the current remaining amount of downtime that is acceptable for a virtual machine before the provider is obliged to reimburse the service user due to an availability breach. That is to say, that after t amount of time availability guarantees will be violated which will result in a penalty for the provider. SLA-budget prioritises failed virtual machines for recovery not in a FIFO approach where the virtual machine that has waited the most gets restored but on the basis of which virtual machine has the smallest SLA-budget i.e. will cause an availability violation the soonest.

On top of existing work, a comprehensive big data-centric analysis of application and data availability has been conducted from Mohanty *et al.* in [89]. The authors place an emphasis on the specific requirements for service availability in the domain of big data analytics applications. They explain that the two basic pillars for high availability lies on the properties of *vertical scaling* i.e. deploy applications in larger physical machines or increase the resources of the existing virtual machines and *horizontal scaling* which involves the distribution of applications on multiple physical or virtual machines based on how relevant the data is to each other or how they can be associated, to achieve a higher level of parallelization. Both properties are satisfied by the basic tenets of Cloud computing and therefore their work is relevant to the big data application in the Cloud.

Finally, an interesting proposal has been made by Rawat *et al.* in [109] that presents a predictive model for data availability in big data processing and more specifically for Hadoop. The authors make a case about the importance of data availability for systems that process large data sets and argue that in such systems the name node is the most vulnerable link in the chain of the computation. This can have a significant impact in the terms of data availability

since the name node is a single point of failure and it can take a long time to recover. In the paper a predictive model is proposed for avoiding failures and enhancing fault tolerance of the name node. A back propagation algorithm in neural networks is being used to make predictions about the expected time of failure of the name node as the available memory resources become less and less available for the name node to use.

2.2.2 Data Privacy

Data privacy in Cloud computing has been the topic of many scientific papers and textbooks [54] [129]. It is a topic that, in many cases, is conflated with the notion of security which to certain extent is justifiable given that enhanced security can lead to more robust data privacy preservation models. However, data privacy is also strongly related to data anonymisation i.e. the prevention of association of a piece of information with an individual. Even though data might not have direct references to a specific person, by means of applying data correlation techniques, one could potentially make inferences regarding private information that otherwise the individual might not have intended to disclose.

The notion of data privacy is perceived differently on the basis of culture, jurisdiction or even country. To provide some context we will examine different definitions that have been presented in recent years in the literature. According to the Organization for Economic Cooperation and Development (OECD) [6] privacy is *"any information relating to an identified or identifiable individual (data subject)"*. Respectively, the definition stipulated by the American Institute of Certified Public Accountants (AICPA) and the Canadian Institute of Chartered Accountants (CICA) in the Generally Accepted Privacy Principles (GAPP), privacy pertains to *"the rights and obligations of individuals and organizations with respect to the collection, use, retention, and disclosure of personal information"*. From the last definition we draw the conclusion that data privacy is involved in several stages of the data lifecycle that spans from data collection all the way to data disposal. Data privacy breaches can occur at any of those stages and therefore the preservation of privacy ought to be addressed in all of them if a holistic solution is to be applied. More specifically, the ordered sequence of phases the data can exist in is the following [38]:

1. **Generate** - Data generation is a critical aspect of data ownership. Who owns the data dictates who is responsible for the application of data privacy policies. This issues can become complicated due to the fact that event though data is typically generated by the users it can be stored in the Cloud.
2. **Transfer** - Data need to be transferred from user's infrastructure to one or more Cloud providers for storage or processing.
3. **Use** - Data is moved to the Cloud not only for storage but for processing as well. In such occasions, Cloud providers provide the necessary software and hardware components to enable the processing of the data and the extraction of business value for its users.
4. **Share** - Data, either with or without the consent of the data owner, can be shared with third party entities. This needs to be conducted in a privacy-aware setting where the data owner's privacy is honored.
5. **Store** - Data is stored in the Cloud and sensitive private information can leak to employees of the provider or untrusted third parties.
6. **Archive** - After the data has been processed and no further on-line processing is required, it is typical for it to be archived for historical reasons. The software and hardware components that will be used to archive and store the data must be set up in a way that data security and privacy will be respected.
7. **Dispose** - Data, when rendered redundant and no longer need to be available - not even for historical reasons - it is disposed of.

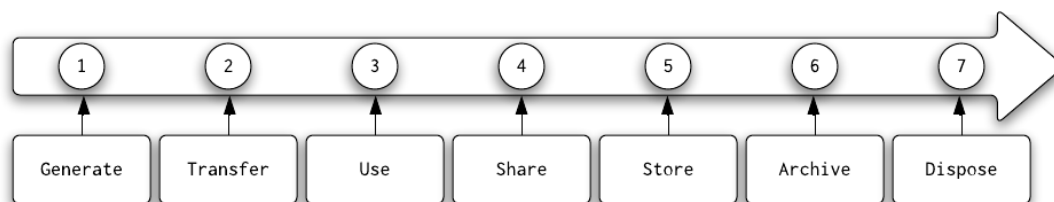


Fig. 2.1 Lifecycle stages of data in the Cloud

A significant body of work has been presented in the literature with regards to data privacy especially in recent years with the advancement of Cloud computing [127] [42] [128]. As it has been discussed in [130], the challenges of privacy and security are here to stay and that a key feature for overcoming them relies on the clear understanding of Cloud computing as a computing paradigm. This involves the a clear view of who manages the system, how it is supposed to operate it and how critical is the data for the user i.e. what would be the financial or reputational penalty in case of privacy breaches. Furthermore, a more practical analysis has been conducted in [150] by Xiao *et al.* In their work the authors argue that the preservation of privacy is directly affected by the property of data confidentiality. Any breach of data confidentiality, i.e. leakage of information to unauthorized entities entail leakage of privacy as well. They also make the point that accountability is against the objective of privacy and if accountability is enforced it can work against the preservation of privacy.

From a legal perspective, the material put forward in [103] and [70] underline the significance of the enforcement of privacy policy laws as part of the EU's strategy to address the issue of data privacy from Cloud services offered within the boundaries of the EU and identifies the relevant regulations. Additionally, in [70] the authors provide an analysis from an information privacy perspective and make a comparison between the regulatory frameworks of Europe and the United States on the basis of the protection that they offer to the users' sensitive information. They conclude their work by stressing the need for an extension of the existing definition of what constitutes sensitive data in an attempt to increase customer trust and a reduction in regulatory constraints that hinder the ability of Cloud users to fully exploit it in its full capacity. Also the importance of a set of regulations is underlined

A similar effort towards the assessment of the privacy risks that are associated with the migration of applications and systems in the Cloud is presented in a PhD thesis in [101]. In his analysis Wayne has created a measurement system using a set of standardised dimensions for the evaluation of privacy risks and provided a detailed evaluation of the existing methodologies giving an account of their inadequacies.

Given that the loss of privacy is a well-recognized issue in Cloud computing, practitioners and researchers have made a concerted effort to put to the front defense strategies to mitigate or prevent it. In [42] propose the use of three groups for classifying the privacy preserving

frameworks namely *information-centric security* i.e. data is associated with access control policies, *trusted computing* i.e. enforcement of trust by means of utilising special hardware and software enhancements and *cryptographic protocols* i.e. the use of cryptography and cryptography tools to protect data privacy.

In [61] proposes the usage of fully homomorphic encryption to facilitate the preservation of privacy. He makes the case that by applying fully homomorphic encryption one can process the data without requiring the decryption of data. By fully encrypting the data the location of storage or processing of the data poses no longer an issue and privacy is preserved by default given the level of encryption is such that cannot be violated. However, the existing homomorphic encryption schemes due to their complexity can be inefficient when it comes to applying them in real life settings. With that in mind and in an effort to improve the efficiency of fully homomorphic encryption, Naehrig *et al.* in [92] present what they call a somewhat homomorphic encryption scheme. In their scheme the authors make the case that for specific applications that they draw from the medical, financial and advertising domains, a subset of operations are sufficient for the practical use of homomorphic encryption.

Following a similar logic to homomorphic encryption schemes, data perturbation techniques have also been used for protecting the privacy of the data. The most common approaches are randomization [21] and condensation [19]. Data perturbation is appropriate for cases where data containing sensitive user information need to be stored or processed in a private Cloud. It involves the modification of the data in a way that it becomes increasingly challenging for third parties to infer the original data while at the same time preserving certain properties of the data objects that are critical for the integrity of the computations that are to be executed. Data perturbation techniques are evaluated based on two metrics; *loss of privacy* and *loss of information*. If possible, both of those metrics should be minimized for a more efficient result. Loss of privacy refers to how hard it is to guess the original data object before the perturbation has occurred. Loss of information refers to the amount of critical data that will not be recoverable as soon as the perturbation has been applied. The level both of loss of privacy and of loss of information need to be examined on a case by case basis and is task-specific. An improved version of the randomization perturbation technique is demonstrated in [39]. In this work, the authors present a rotational perturbation technique for data mining algorithms that require analysis on multiple dimensions. More specifically, they

have identified and proved that kernel methods, SVM (Support Vector Machine) classifiers and hyperplane-based classifiers are rotation-variant and therefore rotational perturbation can be used to modify the data.

Another relatively recent work that is more appropriate for Big Data analysis is shown in [22]. In this paper a modified version of the DBSCAN (Density-based spatial clustering of applications with noise) clustering algorithm for vertically partitioned data is presented where homomorphic and asymmetric encryption are used. The variant of the original DBSCAN algorithm relies on the transmission of information over a secure channel that is encrypted based on homomorphic and asymmetric encryption. Homomorphic encryption protects the data from within the system and asymmetric encryption protects the data from outside attackers. A significant drawback of the framework proposed lies on the fact that it is only appropriate for the DBSCAN algorithm.

The advent of Cloud computing has given rise to distributed applications that are a perfect fit for the scalable and elastic characteristics that it has. In [141] Wang *et al.* presented a framework for the preserving privacy by means of auditing for Big Data storage systems that run in the Cloud. Their work is different because it refers to data at rest and not data during processing. Data auditability enables users to refer to third parties to verify the integrity of the data on demand. Third party auditors (TPA) require two things; they need to be able to efficiently audit the data on-line meaning that a local copy of the data should not be required and secondly the auditing process should not increase the attack surface by means of introducing new vulnerabilities to the system. To address those two challenges the authors integrate public key based homomorphic authenticator with random masking to guarantee that the auditor will not be able to gain any knowledge with regards to the data that is being stored in the Cloud provider. The overarching objective is to take away the responsibility of auditing from the user while at the same time increasing trust. The proposed publicly auditing protocol can deal with multiple users and therefore can be applied in the Cloud where a multi-user environment is totally expected.

By the same token, addressing the issue of privacy preservation for Big Data applications that run on a distributed environment Roy *et al.* in [112] have proposed a MapReduce-based system called Airavat. To achieve its goal, Airavat utilises mandatory access control and differential privacy. Differential privacy refers to a methodology of ensuring that the

output of aggregate computations does not violate the privacy of the individual inputs. In Airavat privacy policies are governed by the data provider and can be applied even from non-expert users as well. Airavat is responsible for preventing the breaching of private information from untrusted mappers and reducers. In this context the objective is to prevent leakage of information referring to general or aggregate features when applying MapReduce computations on sensitive data. The framework's effectiveness relies on the fact that if a MapReduce computation is differentially private, the security level of its result can be safely reduced.

In the same spirit, in [157] Zhang *et al.* make the case for a privacy-aware variant of MapReduce where the computational tasks are broken down based on the privacy policy that the user wants to apply on the data that each one of those tasks will operate on. In many cases, Cloud customers keep information that contains sensitive data in private Clouds or on premises whereas data that do not contain private information is migrating in the Cloud. The proposed modified version of the MapReduce paradigm is called *sedic* and is appropriate for hybrid Cloud applications where part of the data is on a private Cloud and part of it is on a public Cloud. *Sedic* takes advantage of the underlying distributed filesystem to methodically replicate data, moving sanitized data blocks to the public cloud. In this data configuration, map tasks are launched on the public Cloud, while data containing private information remain in the private Cloud. To reduce the exchange of intermediate between the private and public Clouds during the reduce phase, *Sedic* performs an aggregation on the results of the map tasks and then submits them to the private Cloud for the final reduction phase. Given that this execution model is the execution model of the original MapReduce framework, *Sedic* supports the execution of legacy applications that have been developed with the vanilla version of the framework.

2.2.3 Data Integrity

The preservation of data integrity is of paramount importance for the production of correct and trustworthy results from software applications. This has been an issue of great concern from the research community and a significant amount of effort has been directed towards the design of data integrity preservation frameworks. In their seminal work on runtime software correctness Wasserman *et al.* [144] the authors presented a framework with checkers that use

stored randomness. Stored randomness refers to the generation, pre-processing and storage of random bits before application execution and the usage of specific hash functions to evaluate the integrity of the results that are produced. The software verifiers are pre-packaged with the application and they are in essence part of application itself. This method however suffers from the fact that the verifiers may introduce themselves bugs that can affect the integrity of the results produced. Under the same principle, Blum *et al.* [30] introduced the concept of a correctness checker which is a program that checks the integrity of the output of a computation. It uses this principle to devise program checkers for algorithms that can be execute in polynomial time. It also uses cryptography and probabilistic interactive proof. Both the work presented in [144] and [30], despite being of significant theoretical importance, in practice they address the issue of computational integrity only for a subset of specific class of problems and therefore they cannot be generalized for all types of algorithms.

In our review, we are mainly interested in performing an analysis on the state of the art with regards to methodologies for protecting the data from integrity violation that can take place in a Cloud environment. To that extend, several tools and frameworks have been proposed for the verification of integrity when data is stored or processed in the Cloud. An enhanced version BigTable [37], a popular NoSQL database introduced by Google, called iBigTable [147]. iBigTable guarantees the integrity of the data by means of using a centralized or decentralized authentication structure. The system uses two different approaches for storing authenticated data that is later used to verify its integrity and both of them rely on the construction of a Merkle Hash Tree (MHT) based authentication data configuration for the root tablet, the metadata tablet and the user tables. The first approach uses a centralised authentication structure and the second one a distributed data structure. The authors argue that the decentralised approach is more efficient because it stores the root hashes for every user tablet, metadata tablet and root tablet. This has two significant advantages. Firstly, by eliminating dependencies between authenticated data structures of tablets, updates on the tablets do not entail updates on all the root hashes but only in the one referring to the specific tablet. Secondly, the data owner is aware of all the hash roots across all the tablets which guarantees the freshness of hash values.

An extensive body of work has also been produced around the MapReduce paradigm which is very popular for batch processing of large data sets. In [146] Wei *et al.* propose SecureMR,

a decentralized replication-based integrity verification framework both of the data and the computations. They put forward an integrity assurance framework to safeguard MapReduce computations against replay and Denial of Service (DoS) type of attacks. SecureMR uses 5 additional components to verify data and computation integrity namely *secure committer*, *secure manager*, *secure verifier*, *secure task executor* and *secure scheduler*. All those additional components enhance the default version of the MapReduce framework with features such as task duplication, secure task assignment, DoS and replay attack protection, commitment-based consistency checking, data request authentication and result verification. The terminology that introduced in the paper corresponds to the default components of MapReduce prefixed with the term *secure*. The *Secure Manager* and *Secure Scheduler* run on the master node and are responsible for the duplication of tasks, the secure assignment of tasks and the consistency checking. Also, the *Secure Executors* run on all nodes i.e. mappers and reducers to fight off DoS attacks. *Secure Committers* take the intermediate results from the mappers and sends them to the *Secure Manager* for consistency checking. Finally, the *Secure Verifier* verifies the integrity of the mappers' results with the assistance of the *Secure Manager*.

In the context of Big Data and around MapReduce another replication-based approach for the evaluation of data integrity is proposed in [132]. The framework challenges the trustworthiness of the cluster's nodes and operates under the principle that one or more nodes might be compromised. The proposed system breaks down the MapReduce computation into smaller fragments. A subset of these fragments are replicated causing a reduction in the network overhead imposed from the replicated tasks. A set of experiments demonstrate the methods applicability and show that only a small subset of the fragments can provide a high degree of detection rate of integrity violations. In the same spirit, Wang *et al.* in [142] point out the significance of operational integrity from all the moving parts of a MapReduce cluster and they present VIAF a verification-based integrity assurance framework for MapReduce. In their work the authors state that mappers comprise the majority of the workers and therefore provide a larger surface for malicious attacks. The system can detect integrity violation both for non-collusive mappers i.e. mappers that are returning incorrect results and collusive mappers i.e. mappers that return incorrect results but on a consistent basis. Frameworks like SecureMR would not be able to detect those faulty mappers because the replicated tasks

for the malicious mappers would return the same incorrect results. VIAF relies on a two step process by means of integrating task replication and a non-deterministic verification where collusive mappers can be discovered with the assistance of an external trusted verifier.

A slightly different approach is analyzed by Wang *et al.* in [143] where they examine the concept of data and computational integrity for MapReduce on the assumption that the underlying virtual machines (VMs) that run on the Cloud to underpin MapReduce computations can not be trusted. The analysis in IntegrityMR is performed at two separate layers; the MapReduce task layer and the application layer. The authors propose a hybrid Cloud solution where the master and the verifiers are deployed in the private Cloud whereas the worker nodes (mappers and reducers) are deployed in one or more public Clouds. The MapReduce task layer verification process is based in the work presented in [132] and the evaluation of integrity for the application layer the creators of the framework use Pig Latin, high-level language for expressing data analysis programs where Pig Latin scripts are modified to introduce invariants to map tasks. Those invariants are checked during job execution and the system can indirectly infer if the nodes involved in the execution of the jobs have produced faulty results.

A holistic approach for addressing security in general has been introduced by Hussien *et al.* in their work in [69] where they use public auditing for ensuring security. More specifically, with regards to data integrity the authors propose the usage of a third party auditor (TPA) to apply elliptic curve cryptography on the data and the application of a cryptography hash function on the encrypted data. Encrypting the data protects it from privacy breaches as well. Xu *et al.* [151] propose a two-phase verification process, one conducted by the Cloud user and one by the arbitrator that will handle possible disputes regarding data integrity violations between Cloud users and providers. Users check the integrity of the data on a regular basis by applying a Message Authentication Code (MAC) technique and by producing the ϕ hash values that represent the hash for every file. All the ϕ values are stored as nodes in a Merkle tree that has been produced from all the files uploaded by the user to the provider's storage infrastructure. If a violation occurs the user employs an arbitrator to resolve the dispute between users and providers. Whoever loses the dispute has to pay the arbitrator.

In [138] the authors suggested a multi-agent based static and dynamic data integrity verification technique that involves the intermittent evaluation of the hash values for the files

stored in provider. Due to significant overheads imposed from evaluating data integrity for all the data in the files and in order to minimize it, authors use parts of the file that are randomly selected. Their framework also support the restoration of files whose data integrity has been compromised with the assistance of a backup storage service. Similarly, in [152] Yao *et al.* bring forward a framework to remotely check data integrity in Cloud storage systems with the use of threshold encryption. Threshold cryptography refers to a cryptography scheme that fortifies data by encrypting it and sending it to a cluster of fault-tolerant computers. Data is encrypted with a public key while the private key is shared between the the participating parties. In threshold encryption, decryption or signing of data requires multiple parties. The number of those parties is determined by a predefined threshold number. All parties need to collaborate for the successful implementation of the decryption and signature protocol. Additionally, encryption threshold is combined with secure erasure code where data is fragmented into pieces and stored across multiple locations minimizing storage requirements while offering a high degree of redundancy.

An approach with the same objective is shown by Kumar *et al.* in [124] where a data integrity verification protocol is described which does not involve the encryption of all the data stored in the Cloud but only a few random bits per block. The verifier is only responsible to safely store the cryptographic key that has been used for the encryption which makes this verification scheme appropriate for clients with limited computational capacity. Following the paradigm of third party auditing (TPA), Yu *et al.* in [154] give a description of a remote data integrity framework where no metadata need to be acquired from the auditors in order to verify the integrity of the data. This aspect is critical because it promotes the notion of *zero-knowledge privacy* and guarantees that the third party auditors will not have any information with regards to the data that they are checking. To attain the goal of building an integrity auditing protocol, Liu *et al.* in [80] use proofs of retrievability presented in [118] while using data deduplication to keep data storage requirements to a minimum. Their contribution is significant for the evaluation of data integrity in the Cloud because data deduplication is a common technique used by multiple Cloud storage services.

An improved version of Merkle Hash Tree (MHT) for the verification of integrity that supports dynamic data in the Cloud is presented by Guo *et al.* in their work in [66]. The authors use a combination of MHT and Bloom filters [32] that allows for an efficient discovery

of items within a set. Bloom filters are more efficient in terms of space and are very effective in running membership queries on large data sets. Bloom filters are used to verify if a piece of data belongs to the original data set. If it does not belong to the original data set this signifies the compromise of the integrity of the data.

2.2.4 Data Confidentiality

Keeping data confidential is a challenging problem that has been examined extensively in the literature [156] [69] [75] [76]. More specifically in this section we give an account of the literature with regards to the preservation of data confidentiality for Big Data during processing and at-rest.

A plethora of proposed framework with the intention to protect the confidentiality of Big Data from attackers has been introduced. In [106] Puthal *et al.* address the issue of data confidentiality in Big Data streams of sensing data. This is particularly interesting for sensitive data such as medical information that is gathered from medical devices. The authors in their work propose a multilevel selective encryption (SEEN) method to protect it from unauthorised access and modification. Those actions pertain to the preservation of data confidentiality and integrity. The approach they propose relies on two basic concepts namely a common shared key that is created by the data stream manager that is responsible to collect all the streams of data and a key update process that does not interfere with the encryption and decryption process of the data. The idea is that data is encrypted at the source it is produced and is associated with a specific level of confidentiality based on the sensitivity of the data. Since the data is encrypted at the source when it arrives at the data stream manager it is already encrypted. With selective encryption method different keys are applied to achieve different levels of confidentiality. Depending on the sensitivity level, multiple shared keys will be applied to encrypt the data. In the general case if n levels of confidentiality is required then $n - 1$ keys will be necessary to encrypt and decrypt the data. In their paper the authors examine three levels of confidentiality; strong confidentiality, partial confidentiality and no confidentiality. Strong confidentiality refers to the reservation of confidentiality across its complete lifecycle, partial confidentiality refers to data confidentiality until the data reaches Big Data processing engine i.e. preservation of data confidentiality when data is transmitted

over the network from the sensors to the main system and no confidentiality refers to the absence of any mechanism for preventing data confidentiality breaches.

By the same token, the issue of data confidentiality is addressed by Chen and Huang in their work presented in [40]. They examine the application of fully homomorphic encryption [60] to protect sensitive data from unauthorised entities or malicious attacks when it is being processing with the MapReduce framework. Their work is important because it addresses the data confidentiality challenges in the space of Big Data processing with MapReduce, a widely used paradigm for processing large data sets. The solution that they propose allows MapReduce to operate on encrypted data without the need to decrypt the data. To achieve this they have to modify the way MapReduce associates data items with the same key. The reducers will have to be able to discover what data items have the same key to be able to apply the reduction function. This is achieved with the addition of a module that sits between the mappers and the reducers and is responsible for applying the homomorphic encryption function on the already encrypted data. This will make sure that the ciphertext for the same keys will be the same and therefore the reducers will be able to use this property to identify data items with the same key. The theory is the following:

Lets pick two prime numbers A , B , and make $P = A * B$. Then lets pick a random positive integer A_r and A is the symmetric secret key. The encryption function can be seen in 2.1.

$$C = (M + A * A_r) \bmod P \quad (2.1)$$

The decryption function can be seen in 2.2

$$M = C \bmod A \quad (2.2)$$

In both equations M represents the message that will be processed. Now lets assume that we have two ciphertexts $c_1 \leftarrow (m_1 + A * A_{r1}) \bmod P$ and $c_2 \leftarrow (m_2 + A * A_{r2}) \bmod P$. The addition of the ciphertexts can be seen in 2.3.

$$c_1 + c_2 = (m_1 + m_2 + A * (A_{r1} + A_{r2})) \bmod P \quad (2.3)$$

Therefore applying the decrypt function on the added messages can be seen in 2.4

$$\text{Decrypt}(c_1 + c_2) = m_1 + m_2 \quad (2.4)$$

Similarly, the multiplication for ciphertexts c_1 and c_2 can be seen in 2.5.

$$c_1 * c_2 = (m_1 * m_2 + A * (A_{r1} * m_1 + A_{r2} * m_2 + A * A_{r1} * A_{r2})) \bmod P \quad (2.5)$$

Finally, the application of the decrypt function on the multiplied ciphertexts can be seen in 2.6.

$$\text{Decrypt}(c_1 * c_2) = m_1 * m_2 \quad (2.6)$$

Equations 2.4 and 2.6 meet the criteria for homomorphic encryption functions and therefore they can be used from the MapReduce algorithm. The authors proposal is one of the first attempts to use a homomorphic in the domain of Big Data. Their approach even though is feasible it lacks flexibility and cannot leverage the full potential of the MapReduce paradigm. It can perform simple operations that required the addition of values and no other reduction function can be applied.

To address access control considerations and therefore the preservation of data confidentiality in the domain of Big Data analysis, a comprehensive approach is presented in [133] that is also implemented around the MapReduce paradigm. Ulusoy *et al.* argue that the way access control is performed in MapReduce is not flexible enough for the processing of data from different stakeholders and purposes. More specifically the locate 3 main issues; firstly the files that are being processed can be and often are very large, secondly the underlying infrastructure can potentially be used by multiple users with different levels of accessibility and intentions and thirdly the data can come from different domains and sensitivity specifications. The proposal they make is based on a set of fine-grained access control predicates in framework called Vigiles. in Vigiles, users can access files only after a set of predicates that correspond to that user has been applied on the files. All the records from all the files that the user requires to access are sift through the access control predicates. This process determines if a record can be accessed or not. The predicate filters can label the records as *reject*, *grant* and *modify*. When the filter returns *reject* the record cannot be retrieved, when it returns *grant* the original record can be retrieved and finally when it returns *modify* a modified version of

the record is retrieved. Using such a fine-grained model for data access control is of great importance given the fact that in many use cases, the same data can be processed by multiple entities within an organisation and have different security permissions. A key feature of the Vigiles implementation is that it does not modify the underlying MapReduce algorithm. This makes it less invasive and easier to adopt in an existing MapReduce installation. Also it has a very light footprint on the system by imposing a 1% overhead when compared to an implementation that modifies the Hadoop source code. A solution with similar intentions from the of the same authors, is presented in [134] where key/value policy enforcement is imposed on a user basis. The second approach is different in that it supports the use of high level specification language such as OCL for the generation of the low level bytecode in Java that represent the predicates that will have to be applied on the records of the files before it gets processed from a specific user.

In addition to the work presented, Adluru *et al.* also highlight the need for data security and privacy in the Hadoop ecosystem. They recognise that when processing data with Hadoop the attack surface for data security and privacy is large due to the large number of components that are involved i.e. NameNodes, DataNodes and BackupNodes. To increase security the authors propose the use of random encryption techniques. The MapReduce framework will be responsible for applying the randomised encryption algorithms guaranteeing the preservation of scalability. This process decreases the chances of attackers to be able to get access on all the data even if they manage to intercept part of it. More specifically, the authors implemented the RSA, Rijndael, AES and RC6 encryption algorithms.

In [153] Yu *et al.* examine the challenges fine-access control in the context of cloud computing where data need to be sent to a cloud provider and be stored or processed. They make the case that, in most of the existing literature the technique that is used requires that data gets encrypted from the users and then is shipped to the provider's infrastructure. This process is slow and imposes an unreasonable computational burden on the users that need to run computationally expensive jobs on their systems to encrypt the data. In their proposed solution use a combination of attribute-based encryption, proxy re-encryption and lazy re-encryption to allow the data owners to outsource the access control verification process to the cloud providers without revealing to them the content of the data. Attribute-based encryption allows the encryption of the data based on specific attributes of the user. Uses

are associated with a tree of privileges and that is used to generate the secret key for that user. This tree of privileges represent the access control rights of a user the data and if the appropriate right are not present the user will not be able to access it. Proxy re-encryption is a technique that allows a third party entity to modify an encrypted piece of data in such a way that another entity will be able to encrypted after it had been modified. Finally, lazy re-evaluation is an optimisation technique that does not re-encrypt a file for a user whose access rights to the file has been revoked until the file is updated. This trade-off is done on the basis that users can access the contents of file that they were anyway allowed to access at some point in the past. This saves the system from having to re-encrypt files every time the access rights on that file are revoked for a user. In the proposed framework, attribute-based encryption is used to attain the fine-grained access control over files and proxy re-encryption is used from the cloud storage providers to encrypt the data in a way that the data owner will be able to decrypt.

As shown above, a significant body of literature is available for securing Big Data storage and analysis in the MapReduce ecosystem. However, many other data processing framework exist that have significant traction in the industry and academia such as Apache Spark. In [119] Shah *et al.* illustrate how they were able to achieve the application of encryption techniques to enforce security in an Apache Spark cluster for data-at-rest. In Apache Spark data is stored in memory and on the disk as well. Apache Spark tries to store as much data as possible in memory to achieve high performance but there might be cases that this is not possible. For instance this might be the case if the data does not fit in memory or it has to be serialised and sent over the network to participate in a shuffle. Also data can be stored on the disk if the Spark job that has been submitted for execution requires it explicitly in the source code. The authors propose the enforcement of security in three ways; security through secure object serialisation, security through customised *SecureRDD* and security through natively securing RDDs. Security through object serialisation applies cryptography algorithms before and after the serialisation of an RDD. Security with the use of *SecureRDD* is performed by means of a custom implementation of a special type of RDD that extends the base RDD type where the computation of its partitions require that the data is decrypted, processed and then encrypted again. A drawback of that approach is that to apply this technique users will have to modify their code to accommodate the usage of the custom *SecureRDD*. Finally, the

enforcement of cryptography is implemented at a low level where every time data is stored or retrieve it is encrypted and decrypted respectively.

2.3 Monitoring Service Level Agreements

The design and implementation of monitoring tools and frameworks that facilitate the enforcement and monitoring of SLA is a key factor for the adoption of cloud computing. As presented in [17], being able to constantly observe the state of the software and hardware stack in cloud ecosystems is a critical parameter of the smooth operation of cloud systems. Also, the monitoring activity of SLAs enhances transparency and therefore can help build a relationship of trust between the cloud users and the cloud providers. Different language specifications for the definition of SLAs have been proposed in the literature with WSLA [49] and WS-Agreement [9] being the most widely used for Web Services.

A report from the European Union for Network and Information Security in 2013 [18] provides a list with the most significant threats with respect to cybersecurity and underlines the need for the standardisation of the evaluation of security in the Cloud. One of the ways that this can be achieved is by enabling uses to define the guarantee terms of SLAs and then providing them with the necessary tools to continuously monitoring the service level objectives of the SLAs. The ability to measure certain performance and security related properties in a standardised and rigorously manner, is critical for the assessment of the quality of service offered from Cloud providers and can set the foundations for a transparent interaction from the involved parties. Most Cloud providers use natural language to define the guarantee terms of SLAs. This introduces ambiguity and in some cases leaves room for subjective interpretations of the guarantee terms both from the service users and the service providers [4] [1] [5]. SLA monitoring attempts to solve this problem by producing traces of evidence to support the satisfaction or violation of the guarantee terms of the signed SLAs. The collected metrics must be trusted from both entities namely the users and the providers. This is critical for the integrity of the monitoring activity and can prevent having one or more of the involved parties repudiate the veracity of the monitoring results.

As presented in [117], SLA management and monitoring has been a long standing requirements for businesses and organization when migrating to the Cloud. More specifically,

46% of the users did not use an SLA-oriented framework to ensure service continuity, 31% adopted a bilateral protocol i.e. an ad-hoc negotiation policy and 23% engaged with their users with a consolidated definition and monitoring protocol and more specifically WSLA [93] and WS-Agreement [9]. This highlights the fact that SLA management is still an unstructured process that is enforced on per provider basis. Additionally, the same paper brings to the forth the gap in monitoring systems for SLAs which in turn is exacerbated by the lack of adoption of SLA definition frameworks. 81% of the Cloud providers that do have some monitoring tools in place manage themselves those tools and present to the users only the information that they regards as important. In those cases, users cannot inspect the Cloud resources that they utilise beyond what is provided from the monitoring tools. Finally, performance test are regularly executed in 38% of the cases where monitoring is used whereas the in the remaining 62% the results of service performance metrics can be send off to the users only if they explicitly request them.

A plethora of frameworks have been proposed in recent years to address the monitoring of SLAs. Due to the dynamic nature of cloud applications it is imperative that application requirements can change. Such changes are difficult to be aligned with monitoring infrastructures. As a result, monitoring languages have been developed that provide the semantics required for the dynamic definition of the appropriate monitoring configurations taking into account potential amendments in the systems that are being monitored. In [65] the authors introduce Service-Centric Monitoring Language(SECMOL), a general monitoring specification language that enables the definition of monitoring requirements for dynamic deployments of services. Dynamic service composition and deployment is an integral part of the Cloud and therefore monitoring of Cloud service SLAs could benefit from such an approach. SEMCOL is comprised of two parts:

- Monitoring policies that describe how the monitoring components should be deployed, when to check the monitoring rules, how they should be monitored and how the runtime monitoring data should become available (push vs pull model)
- Monitoring rules express the conditions that the system must satisfy. These conditions can be related to either performance or other dependability properties such as security

Monitoring rules are expressed with the assistance of logical conditions directly on the runtime events they are collected during monitoring or indirectly on aggregated computations on the original runtime events.

By the same token, an SLA management and monitoring framework has been examined in [99]. The authors put forward a declarative rule-based approach to SLA representation and management. They argue that providers need to have in place rigorous processes for the management, execution and enforcement of SLAs for thousands of customers whose requirements are potentially different. They also highlight the fact that the service level objectives can contain complex logic and inferences might need to be drawn during SLA enforcement and monitoring. They employ the concept of rule-based service level agreements for the representation of user requirements. SLAs can be viewed as a compilation of logical formalisms managed by a single logical framework called ContraLog. Monitoring and enforcement of the SLAs is performed by making use of standard components of logic programming namely Horn Logic, Event Calculus and Deontic Logic.

Monitoring SLAs is a topic that has become relevant in the context of outsourcing computations from one entity to another. The first manifestation of such a model was presented in Grid computing. In [25] the authors demonstrate how complex event processing principles can be used to facilitate the monitoring of SLA metrics in near real-time. They propose the use of Event Processing Language (EPL) for the definition of composite SLA metrics that are expressed in the form of a query. Their objective is to enable firstly the on-demand specification of service level objectives in the form of metrics, secondly to calculate the metrics at near real-time and thirdly to enable the composition of higher level metrics from simpler ones. The SLA Monitoring Service is built on top of the GEMINI2 monitoring system. GEMINI2 offers the on-line monitoring engine that is comprised of the CEP-based monitoring server (GEMINI2 Monitor) and local sensors (GEMINI2 Sensors) that are installed locally on the nodes. Monitoring data is represented as events which typically contain at least a unique resource identifier (e.g. a host name), and a set of associated metrics such as current CPU load, total free memory, etc. The sensors collect the events locally and publish them to a Monitor. The Monitor has a CEP engine called Esper¹. Queries are

¹ <http://www.espertech.com/esper/>

expressed in EPL and are uploaded to the Monitor through a service. The sensors send the events and they are subsequently processed against the queries in the CEP engine.

Remaining in the same service model of Grid computing other proposals have been made for monitoring SLAs like the ones presented in [24] [56] and [31] as well. In [24] the authors present a monitoring system that is mainly intended for the supervision of the underlying hardware resources that an application is utilising. The framework uses local SMTP agents or Ganglia² agents to collect metadata about the relevant resources. In [24] Fu and Huang propose a monitoring platform that uses an improved forecast algorithm to predict performance and therefore predict possible SLA violations that are associated with performance. Finally, Boniface *et al.* in [31] showcase a framework for the dynamic provision of services with an SLA-based approach. The authors intention is to maximise the profits for service providers and minimise the possibility of quality of service (QoS) violations for the users. Ultimately the framework describes how the available resources should be distributed across services to meet the SLA QoS characteristics that are mandated by the users.

Being proactive with regards to SLA violations is a key feature for the enforcement of SLAs. To that end, there have been proposals in the literature to address the issue of violation predictions with the use of historical data. In [81] Lorenzoli and Spanoudakis have demonstrated the ability to compute the probability distribution of service availability violations for services. As a metric of availability the used was the mean time to repair and the mean time to failure. They have been able to identify a prediction model for several variable that may influence the preservation of availability.

Another key area for monitoring SLA is the automation of the monitoring activity and the ability to adopt to new SLA specifications as quickly as possible. In [114] describe a system where the collection of the monitoring data is automated and is part of the SLA specification. They also address the issue of collecting and monitoring not only information on the provider's side but also on the user's side as well. To attain that objective they authors propose a new SLA specification language. The gathering of the monitoring data is done through the instrumentation of the calls between the Web Services that are involved. The framework operates only in the context of Web Services. A similar tool has been presented

²<http://ganglia.sourceforge.net/>

in [111] where interaction between Web Services are evaluated for the assessment of SLAs. A key difference however is the ability of the system to capture performance values without any knowledge of the implementation of the service. This makes the platform presented generic and capable to be used in any setting where Web Services are involved.

Addressing the issue of SLA monitoring heterogeneous Web Services Comuzzi [45] *et al.* introduce the concept of term monitorability. In the SLA management that was designed in EU FP7 funded project SLA@SOI, they propose the establishment of SLA terms based on two requirements; firstly whether historical data is available to enable the assessment of the SLA offers that service users are presented with and secondly whether it is possible to assess the compatibility of the monitoring capabilities of the monitoring engine with guarantee terms of the SLA. These two questions are critical for the successful management and monitoring of the SLA and need to be addressed before the actual monitoring takes place.

One of biggest challenges in the domain of SLA monitoring is the association of the high level specification of guarantee terms into low level instructions that can facilitate the monitoring of the terms. This issue has been addressed in [52], [41] and [85]. In [52] introduce a framework called LoM2HiS that can assist in turning low-level resource metrics into high-level SLA objectives. The LoM2HiS framework also can detect future violations and can notify the appropriated components to avoid the emanating violations. With similar objectives work is presented in [41] from Chen *et al.* where SLAs are decomposed into its constituents and the translation of service level objectives into system level thresholds. The authors also make the case that this can help users define the service level objectives without having to rely on domain experts but they can automatically do it through the newly introduced framework. Finally, in the same spirit, in [85] Mahbub *en al.* describe a workflow for the automatic generation of SLAs into monitoring specifications that can dynamically be applied and executed to enable the monitoring activity of SLA for Web Services. The monitoring specification is comprised of monitoring rules that are monitored with an event reasoning engine named EVEREST [123].

Monitoring SLAs can also facilitate the ability to gain more insight with regards to what part of the system might be responsible for the poor performance of a runtime component. More specifically, in [148] Wetzstein *et al.* make the case that a violation in a key performance

indication might have been influenced by multiple factors that it is very difficult for users to discover with a naked eye. They propose the use of machine learning techniques to try to build dependency trees between KPIs and low-level processes. A slightly different approach is taken by Emeakaroha *et al.* in [53] who introduce an architecture called CASViD, which stands for Cloud Application SLA Violation Detection architecture. CASViD does not use machine learning to discover factors that might affect violations but it does analyse the runtime behaviour of applications and can, by means of applying a dynamic algorithm, suggest effective measurement intervals for various workloads and applications types.

Also, another thorny issue that is especially relevant for dynamic Web Service and constantly changing setups like the ones presented in the cloud, is the adaptation of the monitoring activity to changes in the service deployment. In [55] Foster and Spanoudakis address this issue by dynamically decomposing SLAs into separate components that can dynamically be monitored by the monitoring components that have the relevant capabilities. This flexible model can decompose the monitoring activity on the basis of the SLA terms and allow its monitoring from multiple monitors. This allows the monitoring of SLA that would otherwise would not be monitorable by a single monitor but can be monitored by multiple ones. Also, this setup is very flexible and can enable the continuous monitoring of SLAs despite the fact that SLAs terms or monitoring components might change. Similarly, to address the issue of SLA monitoring for cloud services that can change dynamically, Mosincat and Binder in [90] have presented a framework for the monitoring of performance in composite Web Services. Poor performance in one of the service that compose the service can potentially lead to SLA violations for the whole service. The author's intention is to automatically diagnose performance issues in composite services and take appropriate action to address them. To achieve this objective they introduce an extension to the standard BPEL specification to enable service monitoring and runtime adaptability. Those two ingredients are indispensable for the automatic detection, diagnosis and reparation of the problematic components of a composite Web Service. Both the diagnosis techniques and the reparation strategies are highly customisable and can be used in any BPEL engine that supports the standard BPEL specification.

In the cloud, storage of data is an operation that is widely used especially in public clouds. For that reason, monitoring SLAs for storage service is a critical factor for their

smooth operation and overall trust of the users towards cloud storage providers. Slota *et al.* in [121] employ the usage of ontologies to apply semantic-based monitoring for storage services. Initially they introduce a set of metrics that are appropriate for the evaluation of storage services such as average read time rate and average write time rate. The user SLA requirements are expressed in the form of ontologies. The ontologies are in essence expressions of quality of service (QoS) that can be translated into metrics that can be calculated based on the attributes of the storage resource i.e. disk arrays, local drives etc. The semantic-based approach allows the automatic mapping of QoS requirements into concrete methods of calculation. This design makes this approach applicable to multiple storage providers that support storage configurations and different software and hardware stacks. In the same scope but geared more towards the cloud infrastructures and not so much towards cloud storage services, Cicotti *et al.* in [43] leverage the capability of the cloud to offer a monitoring solutions on an as-a-service service paradigm. More specifically, they introduce the concept of Quality of Service MONitoring as a Service (QoSMONaaS) that can be provided to the users in a seamless manner through the cloud. In the QoSMONaaS delivery model of SLA monitoring, each SLA is comprised of two elements; firstly the Key Performance Indicators (KPI) that will be monitored and secondly how frequently the KPIs will be gauged. QoSMONaaS uses the notion of quality constraints to express the evaluation of the KPIs as predicates. The predicates are expressed with the assistance of two temporal operators namely *along* and *within*. The *along* operator verifies that a quality constraint QC is true for a specific period of time T - "*QC along T*" - whereas the *within* operator verifies that a quality constraint QC is true at least once in a specific period of time T - "*QC within T*". This configuration allows the use of temporal expression in Linear Temporal Logic for a special version of LTL-logic that supports only the global and eventually operators. The QoSMONaaS framework includes two components namely the **QoSMonitor** and the **QoSChecker**, comprised of the following components. This is illustrated in figure 2.2.

The *Data tier* is the layer where the complex event processing of the real time data streams are being processed and where the data regarding the SLA metadata is permanently stored in a database. The *Business Logic tier* is comprised of two modules namely the *QoSMonitor* and the *QoSChecker*.

The *QoSMonitor* is broken down into the 5 following components:

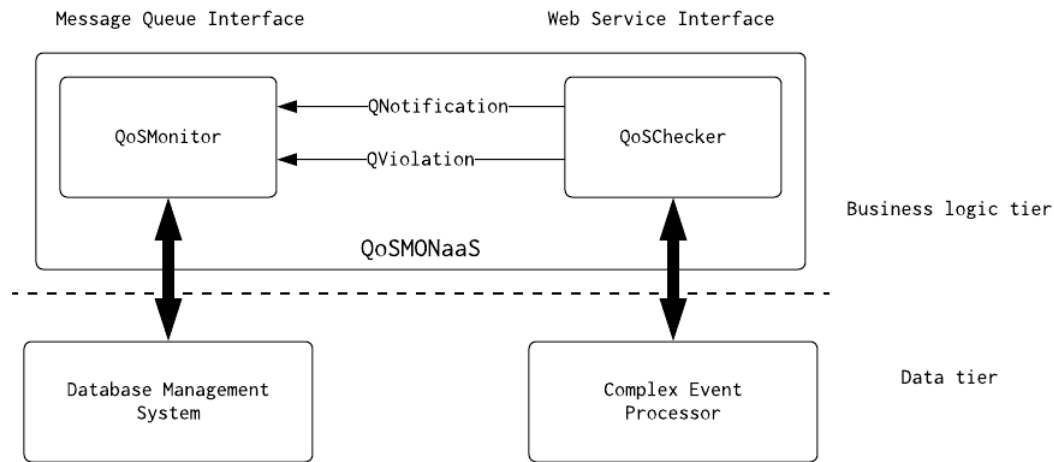


Fig. 2.2 QoSMONaaS system architecture

1. **Controller** - coordinates the rest of the components to enable the monitoring process
2. **DBManager** - centralised connector that stores and retrieves data from the database
3. **Parser** - parses the SLA and makes sure that it is syntactically sound
4. **Translator** - translate te SLA requirements into CEP queries to be executed from the montior
5. **Certifier** - adds timestamp and digital signatures on the checks that the monitor conducts to make them unforgeable

The *QoSChecker* is broken down into the 2 following components:

1. **QoSDetector** - it runs the monitoring algorithm
2. **QoSManager** - it facilitates the submission of new quality constraints to be monitored and also sends back reports and notifications if a constraint has been breached

Due of its nature, cloud resources an be scattered across multiple cloud providers. This property makes it possible to utilise a federated cloud to deploy user services. This complicates the monitoring and enforcement of SLAs because multiple providers are involved. To this end, Cascella *et al.* in [36] show a framework with the name Contrail can enable the

monitoring of SLA in federated clouds. Contrail is comprised of three main components: (i) Contrail federation, (ii) SLA manager and (iii) the Virtual Execution Platform (VEP). The Contrail federation abstracts away the cloud resources from the providers and allows users to interact with only a single component. Use applications are submitted in the Open Virtualization Format (OVF) specification and their SLA requirements are expressed based on that. Based on the requirements in terms of SLAs, the federation component decides what are the best options to deploy the services. The VEP is used to go ahead and perform the deployment step. Contrail can support cloud implementations in OpenNebula ³ and OpenStack ⁴.

2.4 Metrics for Service Level Agreement

In this section we present review of the body of literature with regards to the monitorable metrics that can be included in an SLA.

A comprehensive survey with the security parameters of SLAs, has been conducted by the European Network and Information Security Agency (ENISA) and is presented in [86]. The survey is the result of a report from 117 security officers from 17 European countries. The report highlights the fact that security officers no longer need to apply security policies on privately owned infrastructure but instead they need to be able to effectively manage contracts with service and infrastructure providers in the cloud. The report highlights the need for a clear understanding between service users and providers of what are the key artefacts of a monitorable SLA. More specifically the SLA parameters that are required are the following:

1. **Parameter definition** - what exactly is going to be measured
2. **Monitoring methodology** - the methodology that will be used for the real-time monitoring activity
3. **Independent testing** - if possible test the SLA parameters independently i.e. from the service users themselves or a third party

³<https://opennnebula.org/>

⁴<https://www.openstack.org>

4. **Incident/alerting thresholds** - contact the relevant party if a violation occurs. This requires that certain threshold have been defined for the measurable SLA parameters
5. **Regular reporting** - report the monitoring results of the SLA parameters on a regular basis
6. **Risk profile considerations** - define the incident thresholds based on the risk that entail for the organisation
7. **Penalties and enforcement** - associate incidents with financial penalties and attempt to provide incentives to providers and users to honour their contractual obligations

Sahai *et al.* in their work in [113] have also underlined the requirement for the specification of a set of guarantees that will enable us to overcome the loosely-defined principle of operation for Grid computing. We argue that Grid computing and Cloud computing have significant similarities and as such we regards their work relevant for the definition of guarantees that are monitorable in the Cloud SLAs. The authors propose an architecture for specifying and monitoring guarantee terms that will be part of a service level agreement. To achieve this they also introduce a service level agreement specification language that will allow the definition of SLAs in an unambiguous manner. According to the autor's proposal the SLA is composed of the following:

1. **Purpose** – Why create an SLA in the first place
2. **Parties** – What are the parties that are involved and what is their respective roles
3. **Validity Period** - For how long is the SLA going to be active and monitored
4. **Scope** – What services are going to be included in the SLA
5. **Restrictions** – What need to be done for the services to be provided
6. **Service-level objectives** – What are the agreed levels of service that have been decided both from the providers and the users. Services levels include a set of service indicators such as availability, performance and reliability. All service levels will have to attain a certain goal. Service levels are valid for a specific predefined period of time

7. **Service-level indicators** – How the service level objectives are gauged
8. **Penalties** – Provides an unambiguous description of what will happen is an objective has not been met. Users should be able to terminate an SLA if the desired service level objectives have not been satisfied
9. **Optional services** – provides for any services that are not normally required by the user, but might be required as an exception.
10. **Exclusions** – Define what should not be part of the SLA
11. **Administration** – provide and account of the processes created in the SLA to meet and measure the service level objectives

Note that the in points 6 and 7 in the list above, each SLA can include a series of service level objectives that are measured with the assistance of service level indicators. Therefore, a service level objective can be any aspect of the service that is measurable and can be associated with a service level indicator.

By the same token, in [100] Paschke and Schnappinger-Gerull, highlight the importance of metrics for the definition and monitoring of useful SLAs. They argue that SLA metrics sit in the heart of a successful SLA and are of great importance in the performance compliance of the services with the user's requirements. More specifically, in their work they have produced a comprehensive categorisation of SLA metrics that can help organisations to put together their SLA strategies and use the appropriate set metrics to support them. The list that is presented is the result of the collaboration of the authors with IT service providers from small and medium size enterprises and large organisations.

The intention to improve the experience for cloud users and to facilitate the improvement of the use of cloud services, the European Commission has produced a technical report with a series of standardisation guidelines for the definition of cloud service level agreements that are presented in [135]. The guidelines are proposed to assist businesses to fully leverage operational capacity of cloud computing within the boundaries of the European Union. As a result of the report's analysis four core groups of service objectives have emerged. More specifically, the groups of metrics are *Performance*, *Security*, *Data Management* and

Personal Data Protection. A complete view of all the metrics that have been associated with each group is presented in table 2.1.

Performance Service Level Objectives	
Metric	Description
<i>Availability</i>	The property of being accessible upon demand from an authorised entity. Availability can manifest as a service property in different ways and this needs to be clearly defined. For instance, it could be represented as level of uptime, percentage of successful requests, percentage of timely service provisioning requests
<i>Response time</i>	Response time describes the time interval between a request and a its corresponding correct response. This could be average response time or maximum response time
<i>Capacity</i>	Capacity refers to the maximum value for a service or system property. For instance the number of simultaneous connections, the number of simultaneous service users or the maximum value for the capacity of a resource such as computational power or memory available.
<i>Capacity Indicators</i>	Specifies capabilities of the cloud service that might be outside from the cloud system. Such system could be other cloud services or other non-cloud systems such as in-house customer systems
<i>Support</i>	The presence of an interface that allows uses to perform queries or instantly inform service providers that an issue has occurred. Typical metrics for the definition of supports could be the support hours, support responsiveness and resolution time

<i>Reversability & termination</i>	This refers to the period of time that can take the service user to retrieve their data. Also, termination refers to the process that the providers have in place to describe how the termination of the agreement will take place and for how long the backup copies of the user data is going to be retained by them. Finally, this can refer to the residual data retention that defines the period of time that user related metadata is going to be retained by the provider. Ideally, service users should have the capacity to request to have their personal information completely removed by the service providers whose services they have been users. This property is commonly mentioned as the right to be forgotten
Security Service Level Objectives	
Metric	Description
<i>Service Reliability</i>	Service reliability is associated with the service's ability to perform its function in a correct and timely fashion. In this category of metrics cloud providers could also use the level of redundancy of the constituent components of the service to describe how reliable the service can be in the unlike event of a failure of one or more of its components
<i>Authentication & Authorisation</i>	The metrics for that category refer to the presence of a mechanism for implementing an authentication process to verify the identity of the service users. Also they refer to the system's capacity to grant authorisation to authenticated users to perform a specific set of actions. Authentication and authorisation metrics can be evaluated on the basis of the mean time it takes the provider to revoke access to a user upon request, where are the user credentials stored and whether third party authentication is supported

<i>Cryptography</i>	Cryptography is used to protect the data be it at rest or in transit. The evaluation of cryptography can be performed on the basis of the strength of cryptography algorithm based on the key length and the algorithm itself. Also the level of cryptography can be assessed based on the how well protected are the cryptography keys and if any specialised hardware is used to perform the encryption of the data
<i>Incident Management & Reporting</i>	Incident management and reporting refers to parameters such as the percentage of timely incident reports, the percentage of timely incident responses and the percentage of timely incident resolutions
<i>Logging & Monitoring</i>	Logging can refer to the parameters that are logged during service execution, what log entries does service users have access to and for how long are the logs being retained from the providers
<i>Auditing & Security verification</i>	This category refers to metrics that are related to the auditing capabilities of the provider. This could be evaluated with regards to what processes does the provider has in place to allow independent evaluation from a third party auditing authority
<i>Vulnerability Management</i>	Vulnerability management is critical for the preservation of business continuity. Vulnerability management can be assesses on the basis of the percentage of timely vulnerability corrections, the percentage of timely vulnerability reports and the reports of vulnerability corrections that are produced from the providers for the their service consumers

<i>Governance</i>	Governance of service alludes to the way cloud services are directed and controlled. Governance can make references to what is the type of change that can occur on a service (SLA change or functional), how long does it take to the service provider to notify the service users about the imminent changes and what is the percentage of timely cloud service change notification. This critical because users need to be aware of all service modifications at all times
Data Management Service Level Objectives	
Metric	Description
<i>Data Classification</i>	In the context of cloud services there exist three separate classes of data - cloud service customer data, cloud service provider data and cloud service derived data. The classification of data is evaluated from users based on what policies exist in the providers to describe how they might be using cloud service customer data, what data is going to be created as a result of the execution of a cloud service and what are the right that cloud customers have upon the derived data

<i>Customer Data Mirroring, Backup & Restore</i>	<p>This group of objectives refers to the following SLA parameters:</p> <ol style="list-style-type: none"> 1. Data mirroring storage - How long does it take to mirror the data to another medium 2. Data backup method - With what method does the backup process is performed 3. Data backup frequency - Time between backup operations 4. Backup retention time - For how long does the provider keep the data before it is discarded 5. Backup generations - The number of backups that are available when restoring data 6. Max data restoration time - Time it takes for data to be restored from an available backup 7. Percentage of successful data restorations - The number of successful data restorations divided by all the attempts to restore the data
<i>Data Lifecycle</i>	<p>This is a reference to the data deletion type i.e. how the data is disposed of, the percentage of timely deletions and the percentage of tested storage retrievability. The last metric is the amount of customer data that is retrievable after it has been deleted</p>
<i>Data Portability</i>	<p>Since data might need to be migrated to another system data portability is a desirable metric that needs to be included in an SLA. It can be measured based on the data portability formats that are available, what tools are available to the users to easily export the data in a portable format and what is the transfers rate if the data is to be ported</p>

Personal Data Protection Service Level Objectives	
Metric	Description
<i>Codes of conduct, standards and certification mechanisms</i>	What are the certifications and codes of conduct that the service provider is using to operate internally and in compliant with. This is particularly appropriate in the case of the GDPR EU directive
<i>Purpose specification</i>	This is refers to the provisioning of a list of processing purposes of the data that are outside the scope of what the service customer has explicitly requested
<i>Data minimisation</i>	This gives a concrete view of what the provider does to actively retain as little personal user data as possible. Also, providers will need to explicitly define the maximum time that generated temporary data will be retained and the maximum period of time that it takes them to delete customer data upon request from their customers
<i>Use, retention and disclosure limitation</i>	This is critical in the case of legal actions that might be taken against the cloud customers where the authorities might require data to be disclosed. This metric could represent the number of customer data disclosure to law enforcement entities such as the police - if the release of such information is permitted by law - or the number of personal data disclosure notifications that is the number of customer data disclosures that have been done after the data owners have been notified within a specific period of time
<i>Openness, transparency and notice</i>	Transparency is a reference to 1 tier subcontractors that might be involved in the provisioning of resources or other services that the original service might be using

<i>Accountability</i>	This SLA metric refers to the policies that providers have in place regarding data breaches in the case that they happen. Those policies should describe in detail the protocol that will be applied to hold accountable the appropriate entities as soon as a breach has taken place. Also, accountability can refer to the documentation that should be available to customers and where all internal procedures of the cloud provider should be described and used to show the provider's compliance with standards and data protection directives
<i>Geographical Location</i>	Geographical location refers to the list of geographical locations where data can be stored and also defines whether providers offer the ability to the users to explicitly state where their data is going to be geographically located
<i>Intervenability</i>	Intervenability is the capacity of the service provider to support the customer's requirement for the exercise of data subject rights in a timely and efficient manner. This could be measured as the period of time that it takes the provider to serve such requests from its customers

Table 2.1 SLA metrics defined in the Cloud Service Level Agreement Standardisation Guidelines report in [135]

2.5 Monitoring Frameworks for the Cloud

The need for monitoring applications and infrastructure in the cloud, is of critical importance for users and providers alike. The monitoring process enhances the control that users can exercise over their applications and data when deployed in cloud platforms. It also enables providers and users to inspect certain key performance and security indicators and provision resources accordingly. Usually the quality of service and the security properties are expressed in the form of SLAs, with SLA monitoring being an integral part of the SLA enforcement process that can detect violation as they happen and notify all the involved

parties. A comprehensive list with the activities that are supported by the monitoring of cloud applications and systems are the following:

1. **Capacity and Resource planning** - To guarantee performance providers need to make a concrete plan of the resources that are required to meet user requirements [88]. Also, capacity and resource planning is an activity that is also relevant to the cloud users because they also need to know what resources to request from providers to satisfy the software and hardware needs of their applications.
2. **Resource Management** - Given the complexity of the cloud's hardware and software stack, monitoring can help in acquiring a clear view of the system's state at all times [139]. Resource management takes place both at a physical and virtual setting. Virtualized resources are a key ingredient that cloud computing relies heavily upon. Virtualized resources can refer to network, storage or computational power that might need to be modified at runtime.
3. **Data Centre Management** - Managing large data centres that host the software and hardware components of the cloud is a challenging task and requires a constant view of the state of the system. Power outages, system metrics and management of the physical infrastructure such as ventilation and cooling systems all rely upon the continuous collection of runtime information and monitoring to ensure the data centre's smooth operation.
4. **SLA Management** - Monitoring is critical for the enforcement of SLAs and the evaluation of whether performance and security guarantees of the SLAs are respected [107]. Also, in the context of SLA compliance, monitoring can be auxiliary to the audit process for the certification of SLAs and the verification of compliance with regards to regulatory constraints.
5. **Billing** - Due to the cloud's dynamic nature, billing schemes are flexible as well and can be applied on the basis of the resources that are utilised by the users. In that context, monitoring is very helpful in deciding how much users should be charged on the basis of the resources that they occupy. The billing guidelines can differ based on the provider and the type of service that the users enjoy. For Software-as-a-service

types of applications, the number of users or the total number of users can be a billing criterion. For the Platform-as-a-service model CPU utilisation or response time could be used to decide about charges and for the Infrastructure-as-a-service model the absolute number of used virtual machines could define the pricing model [115].

6. **Troubleshooting** - Typically troubleshooting requires some sort of root cause analysis. This however mandates that a series of information need to be collected and monitored to perform the root cause analysis. The information that will be collected and analysed in most cases will come from different layers of the cloud infrastructure such as applications metrics, network, storage etc. The monitoring activity can help providers spot where deviations from normal behaviour can occur and then take action to minimise the effects that they might have on the system.
7. **Performance Management** - Performance is a desirable feature that cloud users require when migrating their applications in the cloud. Performance variability and availability in general are for most applications a critical aspect that can have a significant impact on business continuity and user experience. Switching between cloud providers to attain high levels of availability or making sure that all the nodes that the application is deployed on are utilised uniformly, can contribute to improved performance. Monitoring the cloud applications and infrastructure can provide the necessary insights to enable better performance management [116] and therefore more performant services for the cloud users.
8. **Security Management** - Cloud platforms need to satisfy very strict security constraints. As such, monitoring all user activity can help in proving that the appropriate security requirements are met and that violations have or have not taken place. In that context monitoring is of utmost importance because it produces the evidence as to why the security properties have been respected or breached respectively.

Our evaluation of the existing literature and tools in the space of cloud application and infrastructure monitoring we will use the following criteria:

1. **Scalability** - how well, if at all, can the monitor handle an increase in the monitoring data that need to be collected i.e more data or probes need to be executed

2. **Elasticity** - how well, if at all, can the monitor adapt to changes at runtime of the monitoring target
3. **Adaptability** - how well, if at all, can the monitor adapt to workload policies that are related to the monitoring activity. Such policies could be the frequency of probes or how many resources the monitor can use
4. **Timeliness** - how well, if at all, can the monitor address the issue of the timely discovery of events as they happen. Usually a compromise between frequency of probes and resource availability is found in order to address timeliness.
5. **Autonomicity** - how well, if at all, can the monitor manage it self and recover in case of failures
6. **Comprehensiveness** - how well, if at all, can a monitor be used to monitor different types of resources such as physical or virtual. The ease with which users can define custom monitoring metrics pertains to comprehensiveness
7. **Unobtrusiveness** - how intrusive is the event capturing process on the system that is being monitored.

2.5.1 Commercial monitoring frameworks

Most large cloud providers of the market recognise the monitoring of their cloud stack a critical element of their business model. This is a testament to the overall importance of monitoring both from the cloud customer and cloud provider perspective. However, most providers fail to address the issue of fine grained monitoring metrics and expose to its users only a subset of the metadata that the users might require and that is a key disadvantage of proprietary solutions that are offered as-is from their respective. providers. In this section we run a survey on commercial monitoring tools that users can use to gain insight with regards to the runtime behaviour of their systems that are either deployed in the cloud or on-premises.

CloudWatch

Amazon is probably the largest player in the market of cloud services. As mentioned, Amazon's cloud monitoring service called CloudWatch [14], does not expose many low-level metrics but mostly information related to the virtual machines. The retention time of monitoring information is two weeks and after that it is discarded and no longer available for consumption from its users. An useful feature of CloudWatch is the ability to set up alarms if certain thresholds are exceeded. The monitoring service is not part of the standard set of services that is offered and can only be used with an additional charge.

AzureWatch

AzureWatch [13] is Microsoft's flagship service to address the challenges of application monitoring that are deployed in the Azure cloud platform. AzureWatch is relatively versatile and is capable of collecting information regarding virtual machines, databases, storage and web site applications. Its flexibility lies on the fact that it offers an API for the definition of custom user-defined monitorable metrics.

CloudStatus

CloudStack is a proprietary paid monitoring platform that is based on Hyperic-HQ, an open source network monitoring solution for cloud infrastructures. CloudStatus integrates smoothly with two of the most well known cloud platforms namely Amazon Web Services and Google App Engine. A plethora of metrics are available and several methodologies for root cause analysis for possible issues are available out-of-the-box from the platform.

Monitis

Monitis [16] is a distributed monitoring system that installs agents on-site to collect vital performance information. Alerts can be enabled to notify users if resources are less than expected. It has mainly been built around Amazon's EC2 services and exposes an open REST API for adding new metrics.

LogicMonitor

LogicMonitor [15] is a on-line set of tools that are specifically designed for low level monitoring of virtual infrastructures by means of a multilayer approach. LogicMonitor's monitoring model is dynamic and can automatically discover new resources as they become available. It offers integration modules for multiple hypervisor technologies such as Citrix XenServer, VMware vSphere and ESX. A comprehensive dashboard is also available for user consumption where all the monitoring data and results are consolidated.

Aneka

Aneka [137] is not simply a monitoring framework but a complete ecosystem for deploying applications in virtualized environments. It exposes an API that enables the development and deployment of .NET applications in public and private clouds. Its monitoring capabilities are baked into the software development kit (SDK) and are part of the application itself. This provides users and software engineers with all the necessary tools to define very low level metrics for their applications.

GroundWork

GroundWork is provided as service for monitoring data centres. Internally it uses an open source monitoring project called Nagios and it can take advantage of different plugins to connect to a variety of devices that might exist in a datacenter. It can collect data and monitor it both for virtual resources and applications. It also integrates with multiple virtualization technologies and cloud providers such as Amazon's EC2 service.

	Scalability	Elasticity	Adaptability	Timeliness	Autonomicity	Comprehensiveness	Unobtrusiveness
CloudWatch	No	Yes	No	Yes	No	No	No
AzureWatch	Yes	No	Yes	No	Yes	No	No
CloudStatus	Yes	Yes	No	Yes	No	No	No
Monitis	Yes	Yes	No	No	No	Yes	No
LogicMonitor	Yes	Yes	No	No	No	Yes	No
Aneka	Yes	Yes	No	No	No	No	No
GroundWork	Yes	Yes	No	No	No	Yes	No

Table 2.2 Evaluation of commercial monitoring tools and frameworks across a set of monitor attributes

In table 2.2, we present a list with all the commercial monitoring tools that we have examined in our survey against the attributes presented in section 2.5 i.e. *scalability*, *elasticity*, *adaptability*, *timeliness*, *autonomicity*, *comprehensiveness* and finally *unobtrusiveness*. As it can be seen, all the frameworks that have been examined in some way they intervene in the normal service or system execution to collect information and therefore they all score No to unobtrusiveness. This also supports the idea that the monitoring activity typically imposes some sort of overhead on the system that is being monitored in the form of additional latency, performance degradation or extra resource utilisation. An other interesting observation is that most of the tools examined do address at least one of the scalability, elasticity and adaptability properties. This seems to be a common pattern that is crucial for the current state of affairs where Cloud and distributed computing is gaining traction for SMEs and big organisations.

2.5.2 Open source monitoring frameworks

In this section we give an account of open source tools that are used from service and infrastructure providers to offer monitoring capabilities to their customers or that users use independently for their applications.

Nagios

Nagios ⁵ is a well-known monitoring solution that has stand the test of time since its first release that was in 1999. It is a powerful monitoring solution that enables organisations to detect and deal with IT infrastructure problems and prevent them from disrupting business continuity and normal system operation. Originally it was indented to be used as a standalone solution for system administrators to allow constant monitoring. However, since the advent of cloud computing, its basic functionality was enhanced to incorporate the monitoring of virtual resources and storage services. Nagios is shipped with a set of predefined agents that collect monitoring data and sends it for aggregation in a central module. The agents are open source and their implementation is extensible to enable the development of custom agents. Extensibility is a key feature of Nagios which has been a key factors for its wide adoption.

OpenNebula

OpenNebula ⁶ is set of tools geared towards the management of virtualised resources. This characteristic makes it a suitable choice for cloud providers. In its standard version OpenNebula incorporates monitoring capabilities that allow the close supervision of the resources. Through a component called the *Information Manager* it collects information about the physical and virtual infrastructure and can be very useful for users and system administrators alike. Internally it uses *collectd* ⁷, is a daemon that gathers system and application basic performance indicators at predefined time intervals and supports the storage of the collected values in an array of different file type such as Round Robin Database (RRD). The collection of the monitoring metrics is conducted over SSH and the *collectd* daemons are responsible for executing system probes on each host that they reside. A central dashboard

⁵<https://www.nagios.org/>

⁶<https://openebula.org/>

⁷<https://collectd.org/>

presents the monitoring information in an integrated manner. OpenNebula supports Xen, KVM and VMware hypervisors.

CloudStack ZenPack

CloudStack ⁸ is an Apache open source project designed to enable the deployment and management of large networks of virtual machines, as a highly available, highly scalable, Infrastructure as a Service (IaaS) cloud computing platform. CloudStack is suitable for all the standard delivery models of cloud computing and can span from public clouds on behalf of well-known cloud providers to private clouds for in-house use. CloudStack is implemented in Java and does support multiple hypervisors such as VMWare, KVM, and XenServer. CloudStack ZenPack ⁹ is an open source monitoring solution that has been developed by Zenoss a privately owned company and can be easily added in a CloudStack installation as an extension ¹⁰. CloudStack ZenPack can collect a plethora of metrics for CloudStack deployments. More specifically:

1. **Memory, CPU, Private Storage** - Allocation
2. **Public IPs** - Total and used
3. **Private IPs** - Total and used
4. **Memory** - Total (with and without over-provisioning), Allocated and used
5. **CPU** - Total, Allocated and used
6. **Primary Storage** - Total, allocated and used
7. **Secondary Storage** - Total and used
8. **Network** - Read and write

ZenosPack can be used to monitor SNMP-enabled devices or it can monitor via an HTTP API. SNMP stands for Simple Network Management Protocol and most network devices such as routers and switches can use it to communicate information about their state.

⁸<https://cloudstack.apache.org/>

⁹<https://www.zenoss.com/product/zenpacks>

¹⁰<https://github.com/zenoss/ZenPacks.zenoss.CloudStack>

LogStash

LogStash ¹¹ is an open source monitoring platform that operates on the server's side. It can consume information from multiple sources at the same time, manipulate it to fit the desirable model and then persist it. It is a project that has been implemented by ElasticSearch, a company that offers distributed real-time text analytics. LogStash is intended to be used with log files and in that regard it is different than the other monitoring solutions. The operational model of LogStash mandates that a LogStash agent runs on every monitored host and this is called a shipper. Shippers read data from log files, the standard input, etc. and ship it by means of AMQ to the LogStash indexer. The indexer parses the data and applies filters and routes to move the data forward. The LogStash indexer sends the filtered data to an instance of ElasticSearch that offers a set of tools for text analysis. From an implementation standpoint LogStash is written in JRuby ¹² and uses Redis [48] as persistence layer. LogStash has been designed with scalability in mind and can scale smoothly in thousand of nodes by leveraging the scalability features of Redis.

PCMONS

PCMONS [50] is an abstract general-purpose monitoring architecture that is mainly for the use in private clouds that use open source technologies. It encompasses three basic layers that interact with each other and are the **Infrastructure** layer, the **Integration** layer and the **View** layer. The **Infrastructure** layer consists of the basic facilities such as the hardware and network and the available software such as the operating system and the hypervisors. The **Integration** layer is providing an set of tools to facilitate the interoperability of the user's actions on the infrastructure layer and enable the use of heterogeneous hypervisor technologies. Finally the **View** layer is the presentation layer where all the monitoring information is shown. The view layer can include system alerts based on the collected monitoring data or the results of the monitoring activity for a service level agreement (SLA). PCMONS in terms of implementation, is designed in a generic fashion to avoid relying on specific tools and technologies. The system is divided in the following modules:

1. **Node Information Gatherer** - collect information at a node level

¹¹<https://www.elastic.co/products/logstash>

¹²<https://www.jruby.org/>

2. **Cluster Data Integrator** - integrate the monitoring data collected by the node information gatherer to avoid unnecessary transfer of data
3. **Monitoring Data Integrator** - collect and persist the monitoring data in the database for historical purposes. This data is propagated to the Configuration Generator
4. **VM Monitor** - insert specific scripts into VMs, execute them and collect monitoring metadata about the runtime state of VMs such as CPU usage and memory
5. **Configuration Generator** - consult the database to create the necessary configuration files to present it in the view layer
6. **Monitoring Tool Server** - receive monitoring data for the resources that are being monitored
7. **User Interface** - it can be a custom dashboard where the monitoring results are presented or Nagios
8. **Database** - permanent storage where the configuration files and the data from Monitoring Data Integrator module

PCMONS is very abstract in terms of its design. The objective of its creators is to detach it from technologies that are specific and its key feature is extensibility.

DARGOS

DARGOS is a scalable and adaptable for cloud resources that are being shared by multiple tenants that use the same physical resources. DARGOS refers to a distributed architecture for resource management platform. DARGOS is by design distributed and efficient and is tailored to be used in the cloud. It also enables the definition of new metrics in an easy and manageable way. According to the authors' analysis, DARGOS introduces a minuscule overhead when compared with similar monitoring tools. DARGOS incorporates two main modules:

1. **Node Monitoring Agent** - Multiple node monitoring agents are installed and more specifically one on each node. They are the software components that collect the

metadata regarding the physical and virtual resources. Every agent is linked with a certain zone in the cloud. Users are able to inspect monitoring information from multiple zones.

2. **Node Supervisor Engine** - The node supervisor engine is responsible for gathering all the monitoring data that has been previously collected from the local agents and make them available to the end-users or administrators via a set of visualisation tools or the REST API for easy integration with other systems.

DARGOS has been designed based on OpenStack and therefore all virtual resources and services that are monitored are defined on that basis.

Hyperic-HQ

Hyperic-HQ¹³ is a tool for performing application monitoring and performance management for virtual, physical, and cloud infrastructures. It is open source and is implemented in Java. It uses Java agents to collect data from an array of operating systems such as Unix, Linux and Windows. A very useful feature of the Hyperic-HQ monitor is its ability to dynamically discover what resources are available and allow its users to decide if they need to monitor them or not.

Sensu

Sensu¹⁴ is a cloud monitoring solution that as its backbone of communication uses RabbitMQ. It is open source and can be used as-is or a custom version with additional support and consulting can be given by Sensu, the company that maintains and continues to add features to its codebase. Sensu offers a flexible architecture, that enables the execution of service checks, the gathering of metrics, and the processing of events at scale. It is comprised of the following modules:

1. **Secure Transport** - Sensu services use RabbitMQ as way of communication
2. **Data Store** - a persistent storage repository in Redis. This setup allows Sensu services to remain stateless and store state data in Redis

¹³<https://sourceforge.net/projects/hyperic-hq/>

¹⁴<https://sensu.io/>

3. **Check Execution Scheduler** - it supports two separate check execution schedulers; the Sensu server and the Sensu client. The server schedules and runs checks requested by the users and publishes them in RabbitMQ and the client runs checks on the local system only.
4. **Monitoring Agent** - the monitoring agents are components that run scheduled check executions. Agents can self-register and clients can subscribe to receive the data they collect in a publish/subscribe model
5. **Event Processor** - an event processor that goes through the event data and can decide to take action if necessary
6. **RESTful API** - an API that can be used to perform administrative tasks on Sensu and also view monitoring results in a JSON format

Sensu has a focus on extensibility, elasticity and scalability.

	Scalability	Elasticity	Adaptability	Timeliness	Autonomicity	Comprehensiveness	Unobtrusiveness
Nagios	Yes	No	No	No	No	Yes	No
OpenNebula	Yes	No	Yes	No	No	No	No
CloudStack ZenPack	No	No	No	Yes	No	No	No
LogStash	Yes	Yes	No	Yes	No	No	No
PCMONS	Yes	Yes	No	No	No	Yes	No
DARGOS	No	No	Yes	No	No	Yes	No
Hyperic-HQ	Yes	No	No	No	No	Yes	No
Sesnsu	No	Yes	No	No	No	Yes	No

Table 2.3 Evaluation of open source monitoring tools and frameworks across a set of monitor attributes

In table 2.3, we present a list with all the open source monitoring tools that we have examined in our survey against the attributes presented in section 2.5 i.e. *scalability*, *elasticity*, *adaptability*, *timeliness*, *autonomicity*, *comprehensiveness* and finally *unobtrusiveness*. Similar to our observation for the commercial monitoring solutions that we analysed, open source tools need to collect information on the system that they monitor, therefore some level of interference with that system is necessary. Also, all monitor seem to address scalability or elasticity since they are all intended either for the cloud or they are themselves deployed in the cloud.

In the last row we have included the monitoring framework that is proposed in this thesis to demonstrate its strengths and weaknesses against similar commercial solutions. As shown,

the Big Data security SLA monitor scores in scalability, elasticity and adaptability. That is because the event captors of the monitor can be scaled out and deployed across multiple nodes automatically and without any manual intervention. Also, the monitoring activity adapts to the cluster configuration i.e. add or remove nodes and finally it is elastic since it can adapt to the addition or removal of resources in the cluster e.g. add or remove computational power (CPU) or memory respectively. With regards to timeliness our proposal does not offer any guarantees on when the event will be available and this is heavily dependent on the network infrastructure. Also there is no autonomicity since the event captors will have to be placed manually in a location where the nodes of the cluster will be able to access them. Also once the execution of the service has began, there is no way for the monitor to be reconfigured so the ability for any kind of meaningful self-government is limited. Finally, the event capturing process is intrusive since it intercept the actual code that gets executed by means for modifying it at runtime to enable the capturing of the relevant events.

2.6 Big Data Processing Frameworks

In this section we will give an account of the most common and widely used Big Data processing frameworks that are both used for research and business purposes. The analysis will be conducted on the basis of 1. programming model, programming languages, 3. input data sources, 4. ability to implement iterative algorithms, 5. ability to use machine learning libraries and 6. type of fault- tolerance. Big Data processing framework in general can be classified based on the way the consume the data i.e. does the framework receive the data in one go and no additional data is added during the computation known as batch processing or does the framework consume data on a continuous basis know as stream processing. Some of the frameworks presented are suitable for batch processing and some are appropriate for more real time type of data analysis.

Apache Hadoop - Apache Hadoop is an Apache project that was the first implementation of Google's MapReduce algorithm that allowed Google to run its PageRank algorithm utilising commodity servers. Their seminal work has been presented in [51] and relies heavily on a distributed file system called HDFS (Hadoop File System). Due to its importance and because MapReduce has set the foundations of parallel processing of large data sets in the

domain of Big Data analytics, we will give a more thorough account of the algorithm and how it achieves the level scalability that is required for the processing of large data sets. Apache Hadoop's scalability is significantly influenced by the underlying file system of HDFS. HDFS is an open source implementation of GFS (Google File System) [62] that has been designed and implemented by Google and is internally used by it. HDFS is distributed meaning that the data is scattered broken down into sizable chunks and stored across multiple nodes. In this setting it is possible to keep multiple copies for each chunk of data and therefore through redundancy to achieve fault tolerance. HDFS uses a master/slave approach. The *name node* is the master that holds all the metadata with respect to the locations where the chunks of files are stored. When a client makes a request for a file in HDFS, the name node is responsible to respond all the necessary metadata with respect to where the actual data is physically stored in the cluster. In a typical setup of HDFS, a backup name node called a *secondary name node* will be in place to take over in case the main name node fails. This is critical because name nodes are single points of failure in a Hadoop cluster. On the other hand, the *data nodes* are the nodes where the actual data is stored. They communicate with the name node on predefined time intervals to declare their availability. Typically, when the replication factor of the cluster is set to more than one, the same chunk of data will be stored on multiple nodes in the cluster. The name node will make an effort to spread the data as evenly as possible for two reasons. Firstly, to allow all nodes to contribute equally in the cluster's storage capacity - we do not want nodes to be either underutilized or overutilized - and secondly in case of a failure of a physical node the data that it hosts can be recovered from another node that is active. A visual representation of HDFS can be seen in figure 2.3.

MapReduce is the programming model that Apache Hadoop uses to parallelize the execution of the jobs across the cluster nodes. It is important to highlight that the storage and processing models in Apache Hadoop are intertwined. With that we mean that data is processed on the data nodes that they reside from processes that run on the same nodes. This allows for data locality and minimizes the transmission of data over the wire which is a time-consuming and resourceful operation. The algorithm is based on the idea of mapping data in a key/value set, group the values by key meaning that the values for the same key are combined and finally a reduction function is applied on the grouped values. The reduction function is applied on the values for every key.

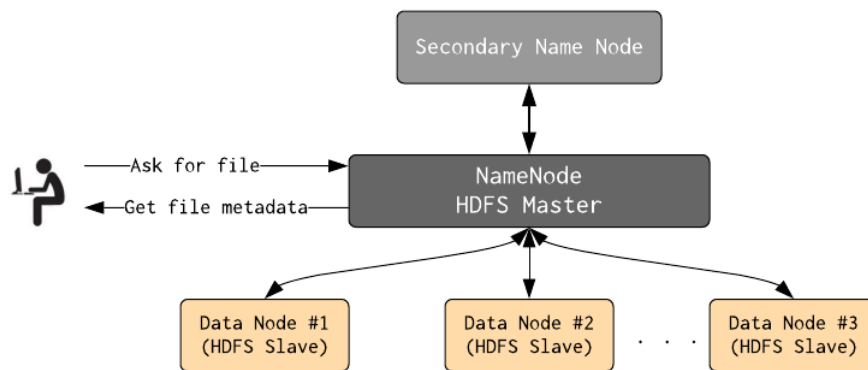


Fig. 2.3 Apache Hadoop architecture overview

Figure 2.4 showcases how the MapReduce algorithm works. Hadoop's model is broken down into 4 mandatory phases namely the load data phase, the map phase, the partition phase and the reduce phase. Optionally, depending on the type of aggregate function that needs to be computed on every key, a combine phase can also exist.

1. *Load data phase* - Read data from HDFS and transform the original data set into a set of types. Input data can be text or numeric data in tabular format.
2. *Map phase* - Map the tupled data previously loaded into key/value pairs.
3. *Combine phase* - The combine phase is optional and can be used to combine the intermediate data on every mapper to minimize the size of data that will have to be transferred over the network to the reducers.
4. *Partition phase* - In the partition phase the key/value pairs, or the combined key/value pairs if appropriate, are partitioned based on keys. Data is grouped by key so as to be sent to the same reducer.
5. *Reduce phase* - Apply a reduction function on the values of the same key and save the results to HDFS.

Hadoop's execution model uses two basic components namely the *Job Tracker* and the *Task Tracker*. The Task Tracker is a software component that oversees the execution of the map and reduce operations when the algorithm is running. The Job Tracker is a software

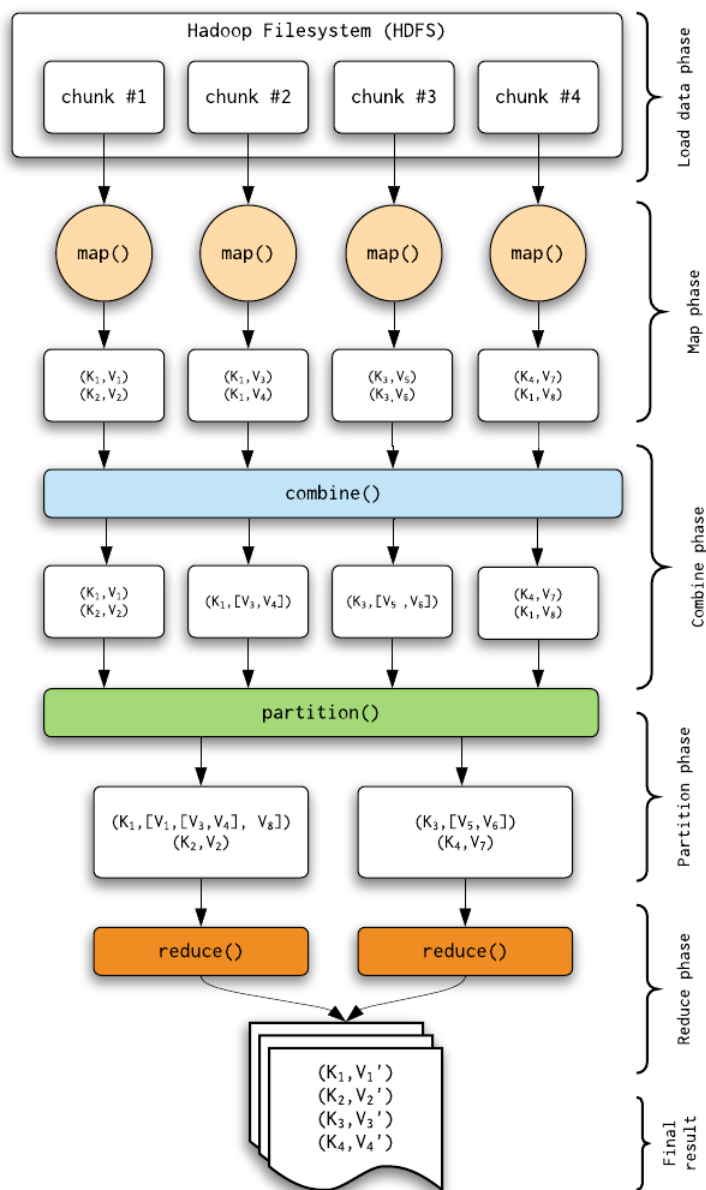


Fig. 2.4 Map Reduce algorithm overview

component that is responsible for the allocation of the necessary resources that are required to execute the tasks managed by the task tracker. Also the Job Tracker schedules and monitors the Task Trackers and makes sure that they perform their job uninterrupted.

Apache Spark - Apache Spark [155] was originally conceived at UC Berkley in 2009 and became a fully-fledged open source Apache project in 2010. Apache Spark's execution

framework is based in Resilient Distributed Datasets (RDDs). RDDs are large sets of data that are broken down into segments called partitions and are distributed across multiple nodes within a cluster. RDD bear two key features that make Apache Spark suitable for a multitude of applications. The first one is that RDDs can, and in fact in many cases, can be stored in memory during data processing. This increases the speed of processing because in-memory processing is typically faster than loading data from disk, processing it and then storing it again. Secondly, due to the fact that RDDs are by design immutable, if the processing of a partition of an RDD is not successful, Apache Spark can re-run the same computation on the original RDD and acquire the same result. Therefore, fault-tolerance in Apache Spark is baked into the framework's most fundamental structure which is RDDs. A key distinctive property of Apache Spark compared to Apache Hadoop is Spark's execution model. In Spark every program is translated into an acyclic directed graph (DAG) of operations where firstly data can be fed to more than one operations and secondly data can be provided as input to the same operation multiple times. Resubmitting intermediate data into the same operation is a property that has profound implications with regards to the applications that Apache Spark can be used for.

Apache Spark is different from Apache Hadoop in several respects. More specifically, Hadoop's execution model allows for the execution of only one pair of map and reduce functions. Spark's execution model allows the execution of multiple operations that can be stringed together in the form of a directed acyclic graph. Hadoop's operations take place in memory but the final result is stored on the disk. Contrary to that, Spark's operations take place in memory as well but it is possible and many cases advisable to keep the results of the operations in memory for further processing. Keeping intermediate results in memory in most cases result in more efficient data processing in terms of execution time. In Hadoop, at least in the default version, only a map and reduce can be applied on the a data set. That means that if for some algorithm requires that the produced results must be mapped and reduced several times this is not a feature that is part of Hadoop programming model. Opposite to that, Spark can re-feed intermediate data to operations as many times as it is required by the algorithm. The number of iteration can also be defined pragmatically which means that it can be decided at runtime. This feature makes Spark appropriate for the implementing of algorithms that require data to be processed iteratively. A prominent category of algorithms

that maintain this property are machine learning algorithms used for the creation of training models.

Apache Spark has been designed from the ground up and its creator intended to make an all around programming model for Big Data processing. To achieve that goal, Spark's API offers a unified API that looks and feels as if one is programming against collection of data and it is no coincidence that Spark's API has been designed around Java's and Scala's collection API. Writing code in Spark's API on RDDs feels like coding against collections; the important difference however is that the code is translated into a DAG that can be executed on a distributed environment. To address the different domains of interest that exist in Big Data processing Spark offers 5 programming interfaces where all of them are based on the concept of RDDs. More specifically, the 5 APIs are the following:

1. **Core** - Basic functionality around RDDs for in-memory processing.
2. **Streaming** - Set of operation that allow processing of streaming data. The API provides a set of functions that allow stream-related operations such as window operations to be executed. Spark implements stream processing in the form of micro-batching.
3. **SparkSQL** - The SparkSQL allows the creation of SQL queries to interrogate the data as if it is a database table in the traditional sense. SparkSQL comes with an SQL parser that support the full capabilities of ANSI-SQL. This makes Spark more intuitive for database administrator and database developers to use without the need to learn a new API.
4. **SparkMLib** - SparkLib exposes a suite of machine learning algorithm implementations in the form of library that make the usage of the most common machine learning algorithms an easy task.
5. **GraphX** - GraphX is an API that offers the ability to process data represented in the form of a graph in a distributed manner. Also, in the GraphX library a set of know graph algorithms have been implemented such as PageRank [97].

Spark's architecture, just like Hadoop, follows a master/slave approach. More specifically, every execution of a Spark program will require the presence of the 3 following software modules:

1. **Driver** - The driver is a program that runs on JVM and is responsible for keeping a reference on the Spark context object. The Spark context object is where the directed acyclic graph of operations is stored and submitted to the worker nodes for execution.
2. **Cluster Manager** - The cluster manager is a program that is responsible for the allocation of all the necessary resources across the available nodes of the cluster, to support the execution of tasks. The cluster manager spins up the JVMs that will host the driver and the executors on the worker nodes where the tasks will be launched for execution. Spark is bundled with a default cluster resource manager called Spark standalone or Master while it also supports Yarn [136] and Mesos [67] cluster resource manager frameworks respectively.
3. **Workers** - The workers are standalone software components where the driver can assign tasks for execution. Tasks are units of work that the DAG has been broken down to based on the types of operations that the user wants to execute and are described in a Spark program. As soon as tasks are completed, they inform the driver and the computation moves forward until the complete set of tasks has been launched and executed successfully.

Apache Storm - Apache Storm [131] is a project created by Nathn Marz in a company called Backtype that was later acquired by Twitter. Storm is a distributed, scalable and fault-tolerant data stream processing framework that can run on commodity hardware. Similar to Hadoop and Spark, Storm uses a master/slave approach when executing tasks. The master node is responsible for running a program called *Nimbus* which schedules task execution, supervises task execution and re-executes them in case of failures. Also, the Nimbus is responsible for distributing the code that needs to be executed from the worker nodes. Worker nodes are responsible for running a program called a *Supervisor* that is actively listening for new task assignment from the Nimbus. As soon as a new task is assigned to a Supervisor, the Supervisor executed the code of the task and reports back to the Nimbus.

Storm's execution model is different from the one used from Hadoop and Spark. That is because of Storm's intention is to address the activity of data stream processing and not so batch or iterative data processing. Programs in Storm are represented by a topology. A topology bears the same properties with the directed acyclic graph that Spark uses. In fact,

from an implementation point of view, topologies in Storm are implemented by means of using a DAG. So, as in Spark, a topology is a graph of computations that operation are strung together until a final result is produced. Nodes in the topology typically represent some piece of logic that we need to apply on the straming data and are called *bolts*. Data move from one bolt to another and are represented by means of a core Storm abstraction called a stream. A stream can be thought as an unbounded sequence of tuples. The topology can read data from multiple data sources that are called *spouts*. Typically spouts are an entry point for most topologies and can be a database, a distributed filesystem or a distributed mesaging framework such as Apache Kafka [77]. A visual representation of a topology can be seen in figure 2.5.

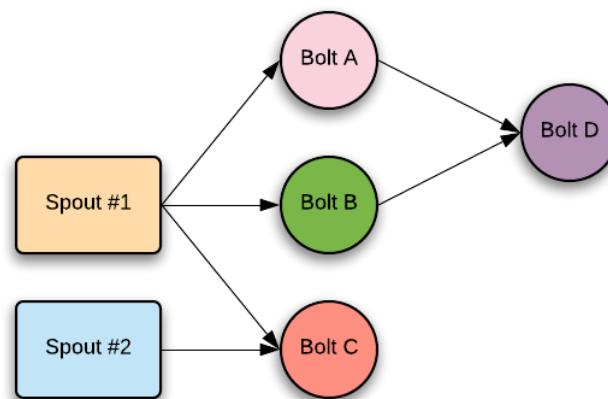


Fig. 2.5 An example of a Apache Strom topology

From an execution stand point, topologies are translated into worker processes that are launched across the cluster. It is critical to highlight, that in order for Storm to achieve parallelism, multiple instances need to be instantiated for all spouts and bolts. Once the topology has been defined, the code submitter will have to provide the necessary configuration parameters regarding the level of parallelism for each spout and bolt. Also, Storm, through its API, provides all the semantics that are required for the proper transmission of data among spouts and bolts. Storm uses different grouping strategies to decide how the data will flow between the nodes of the topology.

The topology shown in figure 2.5 is an abstraction of Storm's execution model. To gain some more insight in figure 2.6 we illustrate how the topology will be represented during

execution from Storm with regards to the worker processes that will have to be instantiated. This is a contrived example where the level of parallelism is as follows: 2 **Spout A** processes, 1 **Spout B** process, 2 **Bolt A** processes, 3 **Bolt B** processes and 2 **Bolt C** processes.

Note that the workers will not necessarily run on the same physical engine. For instance *Bolt B* worker processes can run on different nodes and from a programming stand point they are implemented as threads running on a JVM executor. In fact Storm makes a concerted effort to spread the worker processes as much as possible to take advantage of all the available resources in the cluster and therefore maximize its data processing efficiency.

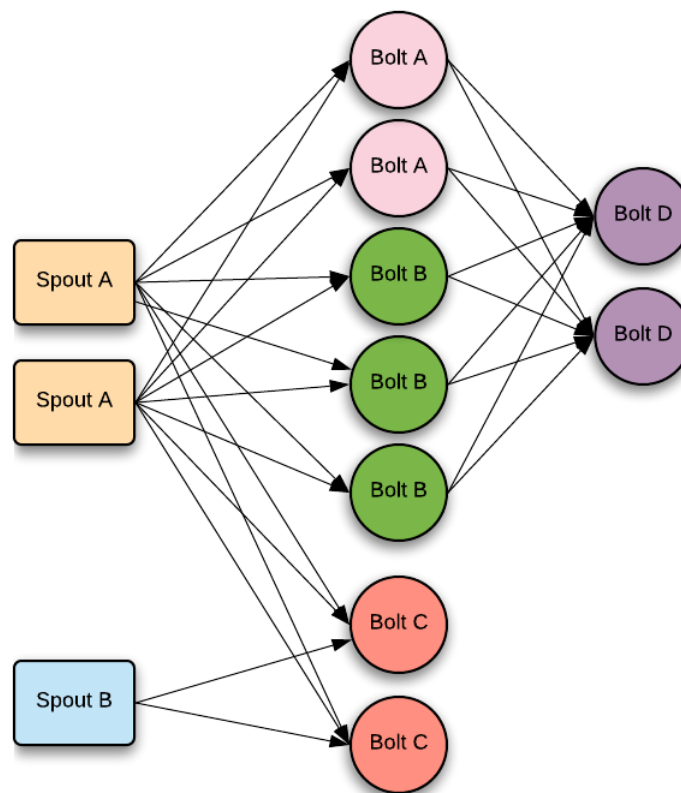


Fig. 2.6 Worker processes for the topology presented in figure 2.5

As it can be seen in figure 2.5 and figure 2.6 data is read from the spouts and is propagated to bolts. Since multiple instances for each bolt can exist, Storm needs to know how to distribute the data from spouts to bolts or from bolts to bolts. To fulfill that requirement, Storm as part of the topology specification uses the notion of grouping. Grouping defines the

strategy with which data will travel from between spouts and bolts. A list with all the built-in types of groupings that are included in the default version of Storm is shown in table 2.4.

Storm's API exposes an interface in case a custom grouping is required. One can implement the interface and use the custom grouping in the same way that the default grouping can be used.

Apache Samza - Apache Samza [94] is a stream processing framework that has been created by LinkedIn and it was out sourced as an open source incubator Apache project in 2013. Samza works closely with Apache Kafka [77], a distributed messaging broker, and Yarn [136] for the dynamic allocation of resources in a distributed environment. A key feature of Apache Samza compared to other Big Data processing frameworks is that it offers by default filesystem persistence capabilities for the state of the tasks that run across the nodes of the cluster. In Samza tasks are stateful and their state is persisted on disk. This is a critical feature because it enables the recovery of failed nodes without replaying the data but by replaying the state. Samza models the state of a task as a stream.

In Samza *streams* are immutable unbounded collections of similar data items called messages. Stream scalability in such a configuration, is attained by breaking down streams into sub-streams called *partitions*. Partitions represent chunks of data into which streams are broken down to. Also messages in partitions are ordered sequences based on the time that they arrive. Messages can be uniquely identified by an offset. Offsets are unique within partitions but not across partitions of a stream. Newly arriving messages are appended to the stream of data and more specifically to some partition of the stream at hand. Messages are appended to partitions based on a key that is decided by the writer of the message. This guarantees that messages with the same key will be sent to the same partition every time. A representation of the stream stream abstraction with its partitions can be seen in figure 2.7.

Jobs are units of work that apply a logical transformation on a collection of input streams and produce a collection of output streams. Processing scalability is achieved by breaking down jobs into multiple tasks that get executed in parallel. There is a one to one mapping between stream partitions and job tasks; a task consumes data from only one partition. The number of tasks that read data from a stream is dependant of the number of partitions of that stream. Similar to Spark's DAG and Storm's topology, Samza uses a dataflow graph as an abstraction for the composition of multi-step jobs. Jobs are independent compilations of

Grouping type	Description
<i>Shuffle</i>	Shuffle grouping sends tuples in a uniform random manner to the target worker process.
<i>Fields</i>	Field grouping makes it possible to group tuples based on specific fields of the tuples. E.g. if there is an id field in the tuples that the source tasks is reading, with the field grouping it is guaranteed that the tuples with the same id will go the same target task.
<i>Partial Key</i>	Partial key grouping is similar to field grouping with the additional benefit of load balancing between two target tasks. This grouping is appropriate when that data is skewed i.e. the values on the fields that have been used to group the tuples are over-represented. This will have some of the target tasks to work harder than others resulting in the under-utilization of certain tasks.
<i>All</i>	In all grouping tuples are sent to all target tasks in a non-discriminatory manner.
<i>Global</i>	In global grouping all the tuples of the source tasks are sent to only a single target task. Storm by default sends them to the task with the lowest id.
<i>None</i>	None grouping signifies to Storm that the user does not show a specific interest with regards to what grouping will be applied. By default Storm will do a shuffle grouping.
<i>Direct</i>	Direct grouping mandates that the source task will have to explicitly specify what task will be the recipient of the tuples after the processing.
<i>Local or shuffle</i>	Local or shuffle grouping prioritizes the transmission of tuples from source tasks to tasks that are launched on the same worker using shuffle grouping. This minimizes network traffic because tuples do not have to be sent over the network and the efficiency of the stream processing is increased. If no target tasks are co-hosted with the source tasks then a shuffle grouping takes place.

Table 2.4 Types of grouping in Apache Storm

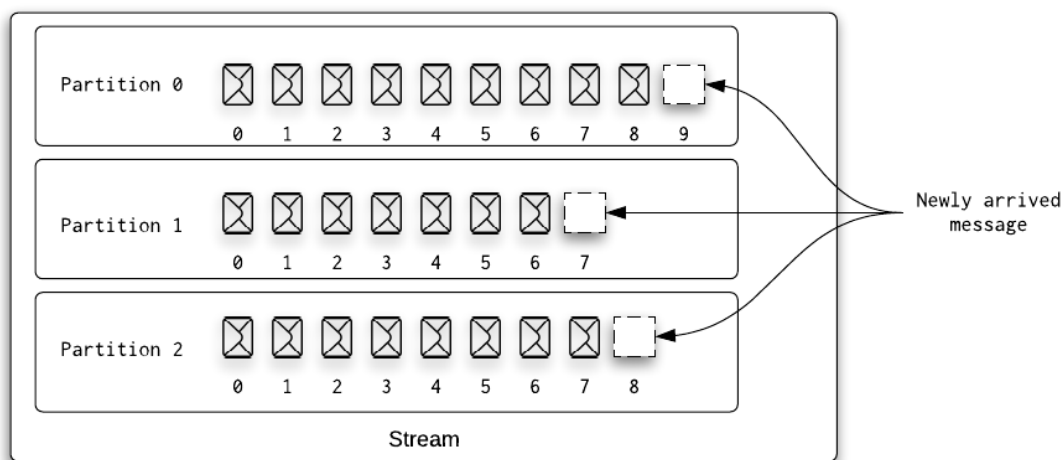


Fig. 2.7 Overview of stream in Apache Samza

tasks and they do not affect one another. If a job crashes it will not cause other jobs to crash as well. A key difference with Spark and Storm however, is that Samza allows the definition of cyclic graphs. Edges in the graph are streams containing data and nodes are jobs that apply transformations on them to meet the user's requirements. An example of a Samza graph can be seen in figure 2.8

Samza uses a coordinator for the management of tasks that get executed across the cluster nodes. Also it monitors the containers where the tasks get launched and executed while it re-executes tasks that have failed. Another useful feature of Samza is that it uses incrementally checkpoints the offset for each partition that it has processed so far. This allow Samza to guarantee that no message will get lost or not get processed. Additionally, another key feature of Samza is that it uses a permanent state store for every task where is stores meta-data about the state of the task. This storage component is co-located with the task itself to increase efficiency and avoid unnecessary network traffic. Also, if the system has to scale horizontally, since all the state of the tasks is persisted in the state store, tasks can be easily migrated to other nodes in the cluster without disrupting the normal operation of the application.

Apache Flink - Apache Flink falls into the category of distributed processing engines for stateful computations on streams of data or static sets of data i.e. data at rest. Originally Flink was designed to be a stream processing engine. However, over time is has evolved to

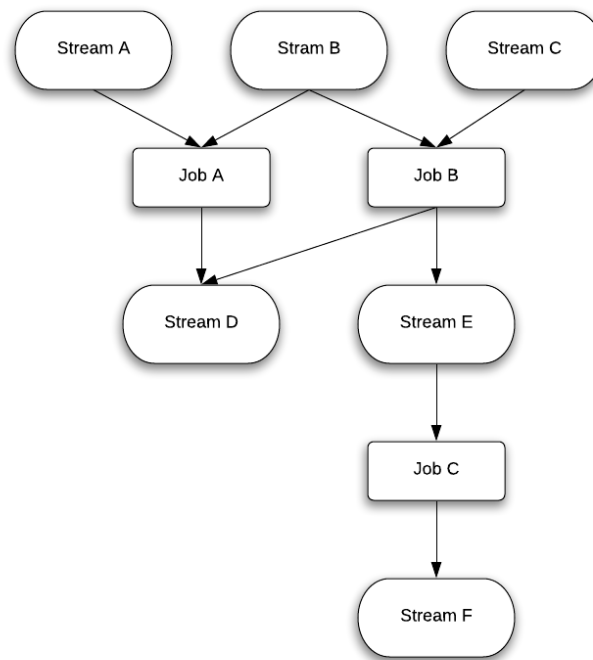


Fig. 2.8 An example of a Samza dataflow graph

handle batch processing as well. Because of that, Flink treats all incoming data as streams where in batch processing and it regards batch processing a special case of stream processing where the stream is bounded. From a deployment point of view, Flink can be deployed in Yarn [136] and Mesos [67] but is also compatible with containerized deployment models such as Kubernetes [34].

Similar to Samza, Flink supports the persistence of state for tasks on a durable medium. If the size of the state get beyond the size of the system's memory Flink stored the state on the disk. Task state is sent to the task state repository asynchronously and periodically and is offered from the framework without any additional effort from its users. An overview of the this concept can be seen in figure 2.9. Application state is a core concept in Flink and it is treated as a first-class citizen. A set of high level semantics are provided with the framework's API that makes state management easy. More specifically a series of state primitives for multiple data structures like atomic values, lists and maps. Also, the storage engine of the state is configurable and can be plugged in to meet user requirements. Also, Flink through checkpointing of state, make sure that no data is lost or unprocessed and enables it to recover from failures.

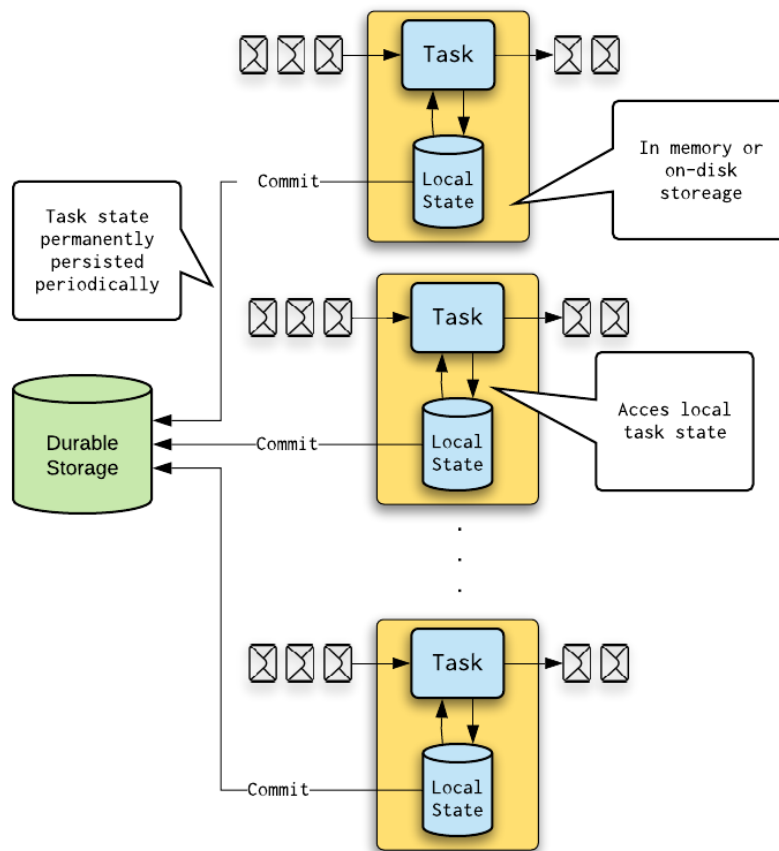


Fig. 2.9 Overview of the task state persistence mechanism in Apache Flink

Flink comes with a rich API that exposes a series of high level operations such as filters, joins and aggregate functions. It also provides a machine learning API called FlinkML that supports the definition and execution of machine learning pipelines. FlinkML API is based on the principle that all machine learning algorithms is a collection of transformations and predictors that are strung together. Transformations prepare the data on which the feature extraction will be conducted by feeding it to a predictor function to eventually create a training model.

Programs in Flink can be written in Java or Scala. The code submitted for execution goes through the *compiler* and a DAG is generated as a result. Subsequently the DAG is sent to the *optimizer* where modifications can take place in the way the DAG was submitted from the user e.g. the order of execution of the operations might change to optimize performance without affecting the correctness of the application results.

A comparative table from the survey on Big Data processing engines that were examined is presented in table 2.5. Our analysis is conducted on multiple criteria that we regard important such as data format, processing mode, data sources, languages supported, cluster manager, support for iterative algorithms, interactive mode, built-in machine learning capabilities and fault tolerance.

Hadoop, Flink and Storm use a key/value data format which is expressive enough for a multitude of use-cases. Spark uses key/value pairs as well but it represents it within an RDD. Spark also since version 2.0 support Datasets and DataFrames which strong typed RDDs for a more versatile data modelling capabilities. Samza uses messages which are represented as Java or Scala objects. In terms of processing mode the possible modes are batch and stream processing. Spark and Flink are the only ones that are supporting both of the modes. Hadoop supports only batch mode while Storm and Samza operate on stream mode only. In addition, since the code has to be executed on a distributed setting, deployment is critical. Spark users its own standalone cluster manager while it also support YARN and Mesos. Hadoop as of version 2.0 and Samza support YARN as well. Flink supports Mesos and Yarn while Storm uses its own deploy interface that is part of the its command-line tools. In the relevant table we also give an list of all the data formats that are natively supported by each framework. Also Spark is the only one that has a read-eval-print-loop tool to allow for interactive mode. This makes it particularly useful in testing simple cases or producing quickly small examples to test an idea. Moreover, frameworks such as Hadoop, Spark and Flink are bundled with built-in machine learning libraries whereas Storm and Samza do not include them by default. More specifically, to use Hadoop's machine learning features one ought to use a special version of Hadoop called Mahout. Spark's machine learning module is called SparkML and Flink's is called FlinkML. Finally, the frameworks presented are examined with regards to how they achieve fault tolerance. Samza, Storm and Fink use check-pointing while Spark has to re-compute the partitions of RDDs that have failed to compute again. Hadoop, since follows the same principle of re-computing HDFS partitions but will not so adverse consequences since Hadoop's Map Reduce implementation is comprised of only two steps and therefore having checkpoints would be of no benefit at all.

In table 2.5 we provide an summary of the Big Data analytics frameworks that we surveyed in section 2.6. In this summary we give comparison with respect to how each one

	Apache Hadoop	Apache Spark	Apache Storm	Apache Flink	Apache Samza
Data Format	Key/Value	RDD, DataFrames, Datasets	Key/Value	Key/Value	Message
Processing Mode	Batch	Batch, stream, iterative	Stream	Batch, stream	Stream
Data Sources	HDFS	HDFS, Kafka, Databases, Parquet, JSON, Avro files, Hive tables	Kestrel, RABBITMQ, Kafka, JMS, Kinesis	Kafka, RabbitMQ, Twitter, Kinesis	Kafka, Azure Eventhubs, inesis, HDFS
Languages supported	Java	Java, Scala, Python, R	Java, Closure Multilang protocol through Apache Thrift	Java, Scala	Java, Scala
Cluster Manager	YARN	Standalone, YARN, Mesos	Embedded	YARN, Mesos	YARN
Supports Iterative Algorithms	No	Yes	Yes	Yes	Yes
Interactive Mode	No	Yes	No	No	No
Built-in machine learning capabilities	Yes (Mahout)	Yes (Spark MLlib)	No	Yes (FlinkML)	No
Fault Tolerant	Depends on replication factor of HDFS	Re-compute partitions of RDDs	Checkpoints	Checkpoints	Local task state checkpoints

Table 2.5 Comparison of Big Data processing frameworks

of the frameworks addresses the challenges of distributed data processing. More specifically the comparison is performed on the following items.

1. **Data Format** - Data formats that the framework can operate on
2. **Processing Mode** - Whether the framework can operation on a batch, streaming or iterative mode
3. **Data Sources** - Types of data sources that they support out-of-the-box
4. **Languages supported** - Available APIs that the users can use to programmatically interact with them
5. **Cluster Manager** - Whether an additional component that is responsible for the dynamic allocation of resources on the cluster is available
6. **Supports Iterative Algorithms** - Whether iterative algorithms can be executed i.e. the ability to rerun algorithms where the output of the previous execution can feed into the next execution cycle. This is especially relevant for machine learning algorithms.
7. **Built-in machine learning** - Whether the framework offers out-of-the-box capabilities for the execution of machine learning algorithms, at least in their basic format with the ability for some basic parameterization.
8. **Fault Tolerant** - Whether there exist mechanisms that fend against failures be it software or hardware failures.

2.7 Big Data Workflow Definition Tools and Frameworks

In this section we will conduct a survey for open source Big Data workflow frameworks that can allow the definition of dynamically executed pipelines. The paradigm of pipelining multiple Big Data processing operations to describe more complex processes, is in alignment with the concept of Big Data processing where in multiple use-cases it is beneficial to break down the computations into multiple interconnected steps. Those steps can be arranged in a specific order and can get executed in-parallel or sequentially on a per case basis. Typically,

workflow engines provide some high level specification language and the necessary tools to enable the definition of the the pipelines as well as the definition of flow control structures such as *"if a step is successfully executed then proceed else if it fails to successfully get executed send an alert"*. These semantics provide a valuable instrument when it is required to set up workflows that require complex execution configurations where simply executing the individual steps in a sequence is not effective enough.

Apache Luigi - Luigi was originally crated by a company called Spotify and was opensourced in 2012 in Apache. Apache Luigi ¹⁵ is an execution framework for Big Data pipelines that is written as a Python package and can be installed as such with Python's package manager called pip. It offers the capacity for task-to-task dependencies and it uses a central scheduling component that is responsible for the coordination of the execution of the workflow. The scheduler exposes an HTTP REST API that makes the communication with Luigi's execution engine easy and intuitive.

The goal for Apache Luigi is not to build pipeline for short-lived jobs but to address the challenges that software engineers are presented with when building long-running jobs where failures are the norm and not the exception. In Luigi's terminology every step of the pipeline is a task and tasks can be arranged in a specific layout to achieve the desired objective. A Task represents a unit of work that can conceptually be grouped and thought as a single piece of independant work. The complexity of task is up the author of the pipeline regarding the granularity of tasks. Having said that, a task can be a call to a database, a computation on a single thread or a Big Data analytics service that is executed on distributed environment. Luigi comes with several common built-in types of tasks. More specifically, tasks can be queries to Hive tables, jobs in Hadoop, Spark, a Python program or a query to a database. Also, custom tasks are also possible to be implemented.

To facilitate the oversight of task execution, a Luigi server is available. The server exposes a web user interface that can help users search and filter tasks for inspection. The web interface is very useful for the supervision of tasks and the evaluation of the level of progress of execution of the pipeline. Task management is not possible through the interface i.e. start or stop tasks.

¹⁵<https://luigi.readthedocs.io/en/stable/index.html>

As a concept, Luigi is heavily influenced by make, which is the default utility of Unix for building large programs composed of smaller ones. The dependency graph of tasks is written purely in Python with the assistance of Luigi's programming interface. Every task exposes three methods; the first one is *requires()* where the dependencies with other tasks are described, the second one is *output()* where the location of the produced results should be placed and finally the third one is *run()* where logic that the tasks will execute is defined. Luigi's application programming interface also exposes callback functions where logic can be written when certain criteria have been met such as when a task has completed its work successfully. Finally, tasks can have initialization parameters that might be necessary to customize execution parameters or initialize the task of the state e.g. acquire a database connection.

Pinball - Pinball ¹⁶ is a Python package that was originally created by Pinterest as a workflow manager to support the execution of Extract Transform Load (ETL) pipelines and it is installed as a Python package. The definition of job workflows i.e. of the directed acyclic graph of jobs, is the result of the declarative of a Python dictionary of objects and snippets of Python code that make references to those objects and the job's logic is declared.

Pinball's architecture is comprised of 4 modules:

1. *Master* - The master is a facade that offers communication with a database where execution state of the workflow is maintained. By default MySQL is supported.
2. *Scheduler* - The scheduler performs calendar scheduling of executions
3. *Worker* - The workers execute the actual jobs. All workers need to connect to the master as a client application.
4. *Web server UI* - Web interface for inspection of jobs and how they make progress in terms of execution. The web UI collects all the execution metadata from the storage layer that the master uses as well.

An overview of Pinball's architecture is illustrated in figure 2.10.

In Pinball a workflow is a graph of jobs. The jobs are represented as the nodes in the graph whereas the dependencies between jobs are represented as the edges. Jobs can have zero or

¹⁶<https://github.com/pinterest/pinball>

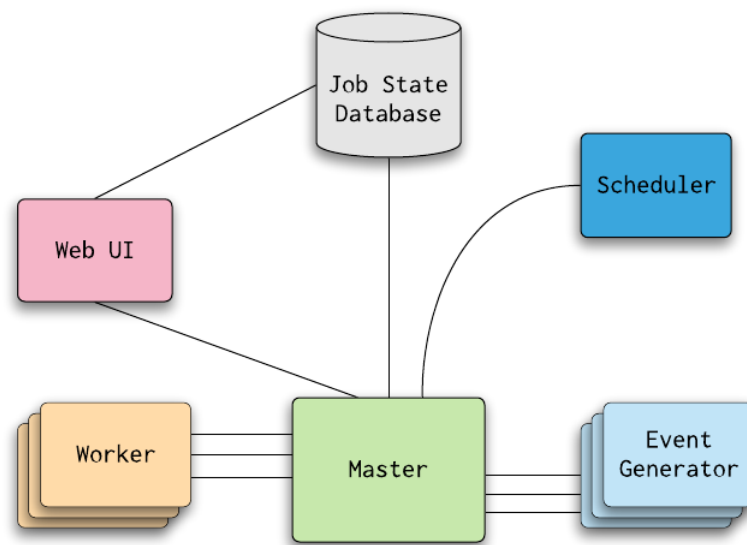


Fig. 2.10 Architecture of the Pinball workflow manger

more inputs and outputs. In the workflow only inputs needs to be defined explicitly. Outputs are defined by incoming dependencies in upstream jobs. For instance, consider jobs A and B where where job B is dependent on Job A. For A we need to explicitly define its inputs but we do not need to define its outputs. That is because, since we know that job B depends on A we know that job's A output is going to be job's B input.

Jobs in Pinball can exist in three predefined states namely *runnable*, *running* and *waiting*.

1. *Waiting* - Not all of its dependencies have completed successfully and the job is waiting until they do.
2. *Runnable* - All dependencies have completed successfully and the job is ready to get executed.
3. *Running* - The job is actively running right now.

A visualization of the job states in the form of a state diagram can be see in figure A.1.

Apache Airflow - Airflow has been designed and implemented from Airbnb as workflow scheduler. Like the frameworks mentioned above, Apache Airflow offers its API in Python and one can use Python to define tasks and dependencies as well. Airflow support the distributed execution of tasks across multiple worker nodes. It also has capabilities for

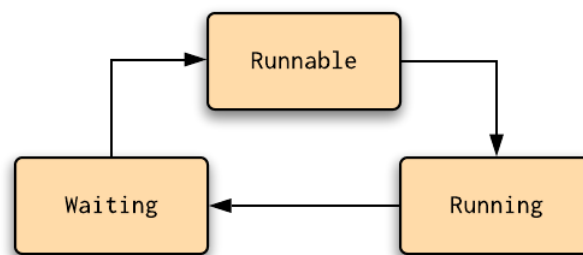


Fig. 2.11 State diagram for job statuses in Pinball

calendar job execution scheduling while it exposes through a web user interface a dashboard that visualizes the execution of the workflow. Also, Airflow's UI offers management capabilities over tasks, graphs of tasks or sub-graphs. Administrators can start a task, execute a sub-graph of tasks or even execute a task ignoring any input dependencies. This is particularly useful for testing purposes where the administrator needs to verify that a tasks can run successfully in isolation without having to execute the whole pipeline.

In Airflow state execution is stored in database and has support for multitude of database vendors such as MySQL and PostgreSQL by means of using SQLAlchemy. SQLAlchemy is an abstraction for interacting with relational databases and offers the ability to Airflow users to pick the vendor that they wish to use. Similar to Luigi, a directed acyclic graph is used to represent the workflow to be executed. The nodes are the smallest unit of work that is called a task. Since the graph is acyclic, Airflow will raise a red flag if a cycle is defined.

A core component in Airflow is an operator. Operators are executable components that are declared in the workflow and are hooks for executing logic. The framework comes with a set of embedded operators such as *BashOperator* to easily run a bash script, *HiveOperator* to execute a Hive query, *PythonOperator* to execute a Python program and many more. Custom operators can be implemented to satisfy specific user requirements is necessary.

Airflow operators are grouped into three categories based on the objective of the operator:

1. *Sensor* - Sensor operator is a type of operator that keeps running until a certain criteria is met. It probes the task that gets executed periodically to verify if the criteria provided are satisfied or not. The time interval for the probes and the timeout values for the successful task execution are parameterized and are provided to the operator. An

examples of such an operator is *MySQLOperator* that executes an SQL query on a MySQL database.

2. *Transfer* - Transfer operator is a type of operator that moves data from one system to another. An example of such operators are *S3ToHiveTransfer* that transfer data from Amazon's S3 distributed filesystem to a Hive table, *HiveToMySqlTransfer* that transfers data from a Hive table to a MySQL database and many more.

Typically operators in Airflow do not exchange information during execution. This is more of a desirable property and less of a limitation. If two operators need to exchange data then it is suggested that the operators get merged into one. If this is not possible Airflow supports the exchange of messages through XComs and abbreviation of cross-communication. CCom uses key/value pairs with timestamps to share state between tasks. XCom can use either a pull or a push model to receive or send message respectively.

Spring Cloud DataFlow - Spring Cloud DataFlow is part of the Spring ¹⁷ ecosystem and is suitable for building data pipelines. Spring Cloud DataFlow offers a tooling for piecing together Spring Boot ¹⁸ applications to build real-time data ingestion and data processing workflows. A more detailed account of the Spring Cloud DataFlow's features and capabilities can be found in section A.1.

A comparison between the workflow management frameworks that have been examined in this section can be summarised in table 2.6. Our analysis for the workflow managers that we surveyed is performed against the following properties:

1. ***Code/Domain Specific Language*** - Language to define the tasks
2. ***Web UI*** - Interface available to interact with the tasks and the workflow
3. ***State persistence*** - Ability for tasks to hold state
4. ***Task prioritization*** - Ability to set priority over what tasks will be executed
5. ***Configurable parallelism*** - Support for parallel execution of tasks in the workflow
6. ***Who starts workers*** - How are the workers that will execute the tasks triggered

¹⁷<https://spring.io/>

¹⁸<https://spring.io/projects/spring-boot>

	Luigi	Airflow	Pinball	Spring Cloud
Code/Domain Specific Language	Python	Python	Python	Java, DSL
Web UI	Yes(Basic)	Yes	Yes	Yes
State persistence	No	Yes	Yes	Yes
Task prioritization	Yes	Yes	No	No
Configurable parallelism	Yes	Yes	No	Yes
Who starts workers	User	Scheduler/User	User	User
Implemented in	Python	Python	Python	Java
Task communication	No	XCom	No	RabbitMQ
Can run multiple DAGs	No	Yes	Yes	Yes
Execution management	No	Yes	No	Yes
Code shipping	No	Pickle ¹⁹	Pickle	Maven ²⁰

Table 2.6 Comparison of workflow management frameworks

7. **Implemented in** - Programming language of implementation
8. **Task communication** - Ability of the tasks to communicate during task execution
9. **Can run multiple DAGs** - Ability to support parallelism of execution across different task workflows
10. **Execution management** - Capacity to perform management operations during task execution e.g. pause or resume a task
11. **Code shipping** - What is the mechanism that allows the submission of the task code to the workers for execution

2.8 Gap Analysis

In this section we will expand on the gaps that we have been able to identify when reviewing the literature in the context of the monitoring activity of security SLAs for Big Data services.

Ora *et al.* in [95] describe a solution that allows them to enforce data security and integrity by means of using RSA partial homomorphic encryption and MD5 cryptography. The data is partially encrypted and then uploaded in the cloud. Then an MD5 hash is calculated on the data to verify data integrity. However, in their work the authors refer to data that is at rest i.e. they do not enforce security during runtime but only when data is stored and not processing.

An interesting framework has been presented by Ba *et al.* in [23] where they describe a framework called jMonAtt that is able to monitor the integrity of JVM-based applications and provide attestation evidence with regards to how much users can trust that they haven't been tampered with. Their approach is appropriate for applications that are outsourced in the cloud and they propose a similar solution with the one that we propose. They use code instrumentation to collect runtime information where they evaluate the authenticity of the code that is executed and they compute a score that reflects the framework's confidence that they user's original code has not been modified. However, what they fail to address is the integrity of the data both when processed and when stored on the disk. Their focal point is the integrity of the code executed.

In the same vein, Bendahmane *et al.* in [27] present a novel system based on weighted t-first voting method for guaranteeing the integrity of MapReduce in public cloud computing environment. Their approach focuses identifying malicious worker nodes in an Apache Hadoop cluster that can change the original or intermediate data that is being processed and thus can compromise the integrity of the final outcome of the computation. However, the author's implementation is appropriate for MapReduce systems which is a significant limitation since graph processing or stream processing applications can be protected for the proposed solution. Also, their proposal does not handle attacks on the integrity of the data from external agents but only from malicious internal nodes. Attacks on the integrity of the data from external systems will go undetected.

¹⁹<https://docs.python.org/3/library/pickle.html>

²⁰<https://maven.apache.org>

An equally interesting approach has been demonstrated from Gao *et al.* in their work in [59]. In their proposal the authors illustrate how the use of a reputation based algorithm can be used to provide guarantees with regards to the integrity of the computations in a MapReduce application. At the beginning when the computation is initialised all the worker have a neutral score with respect to the level of trustworthiness. A custom scheduler that operates independently schedules the same tasks to be executed on multiple nodes and compares the results. According to the results that are produced, the framework constantly updates the scores of trustworthiness for every node and eventually nodes that have been compromised are located and eliminated. A key drawback of the system that is put forward is the fact that computation need to run more than once which raises significant considerations with regards to performance. Also, the authors do not explain how they compare the results between worker nodes that are compromised and worker nodes that have not been compromised. Comparing two datasets can be a laborious task. If all the data items of the dataset need to be compared one by one will exacerbate even more the overhead imposed on top of the duplicate execution of tasks.

A promising piece of work has been demonstrated by Shah *et al.* in their paper in [119]. The authors have been able to design, built and evaluate a mechanism for enforcing security for data-at-rest that is being processed with the Apache Spark framework. Their proposal is based on three pillars. The first one is to create a custom serialiser that will transparently encrypt the data every time it is persisted on the disk and decrypt it every time data is read from the disk. The second one is to create a customised RDD called SecureRDD that extends the basic interface of an RDD and implements its own *compute()* method that encrypts the data every time the RDD is stored on the disk and it decrypts its data every time it needs to run a computation. The third pillar of their approach is to modify Spark's native persistence mechanism and enforce encryption on every operation. The third approach that they present requires that Spark's central caching mechanism is modified. Despite the their interesting approach and the use of Apache Spark which is a generic Big Data processing tool and thus appropriate for many different types of Big Data applications, the authors do not address the issue of allowing the users to be explicit with regards to what security properties they would like to enforce on what parts of the Big Data pipeline. In all three proposal that they make the security policy of encrypting the data is applied across all the operations. Typically a more

nuanced approach is required to enable users to be explicit with regards to what security properties they need to enforce and on what aspects of the Big Data application. Also, the deployment of the components that perform the encryption need to be interweaved statically in Spark's source code. This must be done manually, it requires a fair understanding of Spark's code internals and cannot be performed dynamically.

In this section we presented several approaches that we believe that attempt to solve the same challenges as the one that we put forward in this thesis. From our analysis, the gaps in the literature that we view that our framework makes a contribution can be summarised in the following list:

1. Lack of a general monitoring framework for the continuous monitoring of security properties on a generic Big Data processing framework such as Apache Spark
2. Shortage of monitoring platforms that allow users to explicitly define what security properties they wish to enforce or monitor on what parts of the Big Data application
3. Lack of end-to-end monitoring solutions for Big Data pipelines that facilitate the automatic installation of the software components that will be responsible for the enforcement or the collection of monitoring data appropriate for the security property that the users require to enforce or monitor

2.9 Summary

In this chapter we presented relevant work that exists in the literature with regards to application and system monitoring. More specifically we survey the literature around security and privacy properties in particular with regards to Big Data pipelines. In our review we also look into service level agreement(SLA) monitoring and the existing set of metrics that have been used to measure the efficacy of the SLAs. Given that our solution is indented for the space of Big Data, we also give an account of the state of the art for Big Data processing frameworks and tools that can be used to define pipelines of Big Data services. Finally, this chapter is concluded with a gap analysis based on our survey in the space of application and system monitoring where we identify the areas where our thesis will have a contribution.

Chapter 3

Monitoring Framework for Big Data Security SLAs

3.1 Introduction

In this chapter we provide a detailed description of our proposal for the automation of the monitoring activity of security SLAs for Big Data pipelines. In section 3.2 we present an overview of the architecture of the framework and its constituent components. We also give an account of the individual modules that the proposed framework is comprised of and how they attain their objectives. More specifically, in section 3.2.1 we describe the process of service composition which is implemented as a pipeline of Big Data services, in section 3.2.2 we describe how the security requirements are specified, in section 3.2.3 we demonstrate how they are then translated into the right monitoring artefacts and finally in sections 3.2.4 and 3.2.5 we show how the appropriate event captors are installed and deployed in the cluster with the assistance of Apache Spark. Further down in the chapter in section 3.3, we give an account of the monitoring rules that are relevant for the run-time monitoring of three security properties that are pertinent to data availability, data privacy and data integrity respectively. In particular, with regards to data availability we monitor response time, with regards to data privacy we monitor the location (IP address) of execution of the operations of a Big Data service and finally with regards to data integrity we monitor the preservation of unchanged checksums of all the intermediate data in-between operations that is produced

during the execution of a Big Data service. Our analysis of the monitoring rules for each security property that we examine, is broken down into three segments; in the first segment we describe the Event Calculus rules and assumptions for the property that we examine, then in the second segment we describe the SLA template specification that are appropriate for each security property and finally in the last segment we give a detailed overview of the relevant event captor interceptor and delegator that will be used to realise the event capturing process.

3.2 Framework Architecture

In this section we will give an account of the overall architecture of the monitoring framework that we propose for the automatic generation, deployment and execution of the monitoring artefacts that will enable the runtime monitoring of Big Data analytics services. The main objective of the framework is enable the automatic generation of monitoring artefacts from a set of high-level security requirements for pipelines of Big Data services. In this thesis, we regard such pipelines as *composite services* i.e. services that are comprised of other simpler services that we call *atomic services*. Composite services are comprised of one or more atomic services that are executed on the basis of a workflow that is defined by the user. All the module communicate via HTTP RESTful APIs except for the event captors with the EVEREST monitor, where the default communication with RabbitMQ takes place via the Advanced Message Queuing Protocol (AMQP). AMQP is a binary messaging protocol that operates over TCP. The monitoring framework is comprised of 5 core software modules that operate in tandem to facilitate the monitoring activity. The modules are:

1. ***Spring Cloud DataFlow server*** - A tool for the definition of the composite service and the security properties that the user requires to monitor for the service.
2. ***SLA Manager*** - A web application for the management of the monitoring artefacts such as security properties to be monitored, what templates to be used for the generation of the concrete monitoring rules, what are the parameters for the templates and what are the monitoring results when the execution of the service completes.

3. **SLA Manager Integrator** - A standalone software module that is responsible for the automatic creation of an SLA monitoring project in the SLA Manager web application when a new composite service is created from a user. The integration allows users to add additional metadata with regards to the monitoring activity and also to view its results at real time as they happen.
4. **EVEREST Monitor** - Event reasoning engine that can reason about the events given that a set of monitoring rules have been provided in the form of EC-Assertions.
5. **Apache Spark Cluster** - Big Data processing engine where the execution of the Big Data composite service takes place.

An high level view of the architecture of the Big Data composite service monitoring framework can be viewed in figure 3.1

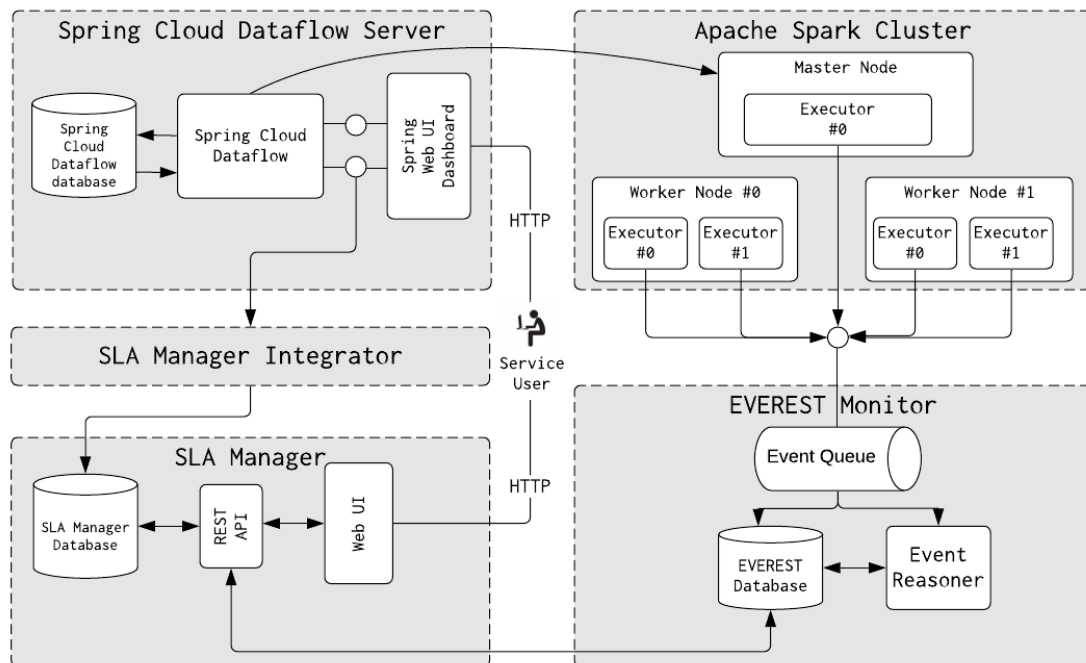


Fig. 3.1 Big Data Pipeline Monitoring Framework Architecture

Let's give a more detailed description of the series of actions that are required to enable the monitoring activity of composite services. Initially, users defines a composite service that will be executed according the users' requirements. A composite service is defined

as a pipeline of atomic services with the assistance of the Spring Cloud DataFlow Web UI. Subsequently, the atomic services of the composite service are associated with security properties that the users need to monitor at runtime. This association takes place in the *Spring Cloud Dataflow* Web UI as well and is provided by the users as a parameter of the service. The name of the relevant parameter is *securityProperty*. Possible values that can be passed to the *securityProperty* parameter are **availability**, **privacy** and **confidentiality** that represent groups of security properties that are available for monitoring from the system. This is critical because *Spring Cloud DataFlow* needs to be aware of security property that is required to be monitored for what atomic service so as to install the appropriate event captors when it sends the service for execution on the Big Data processing framework. As soon as the composite service has been declared and its atomic services have been associated with the relevant security properties, *SLA Manager Integrator* automatically creates a new SLA monitoring project and stores it in the repository of SLAs in the *SLA Manager*. An SLA monitoring project holds all the metadata regarding the monitoring artefacts that are appropriate for the composite service that it refers to. The newly created project has the same name with the workflow when it was defined in *Spring Cloud DataFlow* and the atomic services that the composite service is comprised of are presented to the users. For each atomic service, and based on the security property the has been previously associated with it, a list of predefined implementations of the property becomes available for selection in the format of EC-Assertion templates.

To provide some context with respect to how we use EC-Assertion templates to represent dynamically generated monitoring rules, we will give a rundown of the location of execution property template that is shown in A.2 in the appendix. In the template some boilerplate tags have been omitted for brevity since they are not related to the definition of the monitoring rules but are necessary for the correct parsing of the rules of the monitor. More specifically, from line 61 until line 73 a definition of the security property that the template refers to is defined. In our case it is data privacy. From line 76 until line 114 we provide the declaration of the interface to be monitored i.e. the function to be monitored. In our implementation we call this function **writerdd** and is the function that writes the results of each RDD across the various nodes that will produce them. Note that part of the parameter list of the function is the IP address where the operation gets executed. This piece of information is useful when

evaluating where operation are actually computed. A more comprehensive view of the rule for monitoring the location of operation execution can be seen in table 3.7. The initially assumption is defined from line 126 until line 149 as an event calculus fluent. Then, from line 154 until line 304 we define the rule that mandates that when an RDD is computed and the **writerdd** function is invoked then the IP address of the node that has performed the computation needs to be from a list of trusted IP addresses. This constraint is defined from line 244 until line 259 and is evaluated by invoking the previously defined fluent. The fluent, based on the predefined list of IP addresses that the user has provided, will evaluate to true or false and the rule will evaluate to true or false respectively. Templates might require input from the users to create concrete instances of the monitoring rules that need to in place to support the monitoring activity for each atomic service. To achieve this in the *SLA Manager*, users are prompted to define the parameters based on the type of the security property that has to be monitored. As soon as all the requirements have been collected from the framework, the monitoring rules are automatically generated from the templates in the format of EC-Assertion formulae and are sent to the *EVEREST monitor* . Now, the monitor is ready to accept events that relate to the execution of the atomic services and is able to reason about the events based on the users' security requirements. The emission of events requires that the composite service starts getting executed. The execution is triggered from the users when they need to initiate the process. At this point the composite service knows everything it need to know to deploy the event captors alongside the atomic services. When the service execution is completed and its results have been produced, users can view the monitoring results in the *SLA Manager*. The results are updated dynamically as the monitoring rules are unified as a result of the emission of the monitoring events. This dynamic presentation of the monitoring results is particularly useful in the case of composite services for stream processing types of services where, at least in principle, the service never completes but runs indefinitely. In such occasions users can inspect the monitoring results while the service is still executing.

To provide more context with regards to the use-cases that the system is able to cover, in figure 3.2 we present a use-case UML diagram. Note that each use-case is placed in the component responsible to implement and execute it. Also, note that the SLA Manager Integration is represented as an actor of the system. This is because the SLA Manager

Integrator initiates a set of automatic operations for the creating of an SLA monitoring project from a composite service and its security requirements.

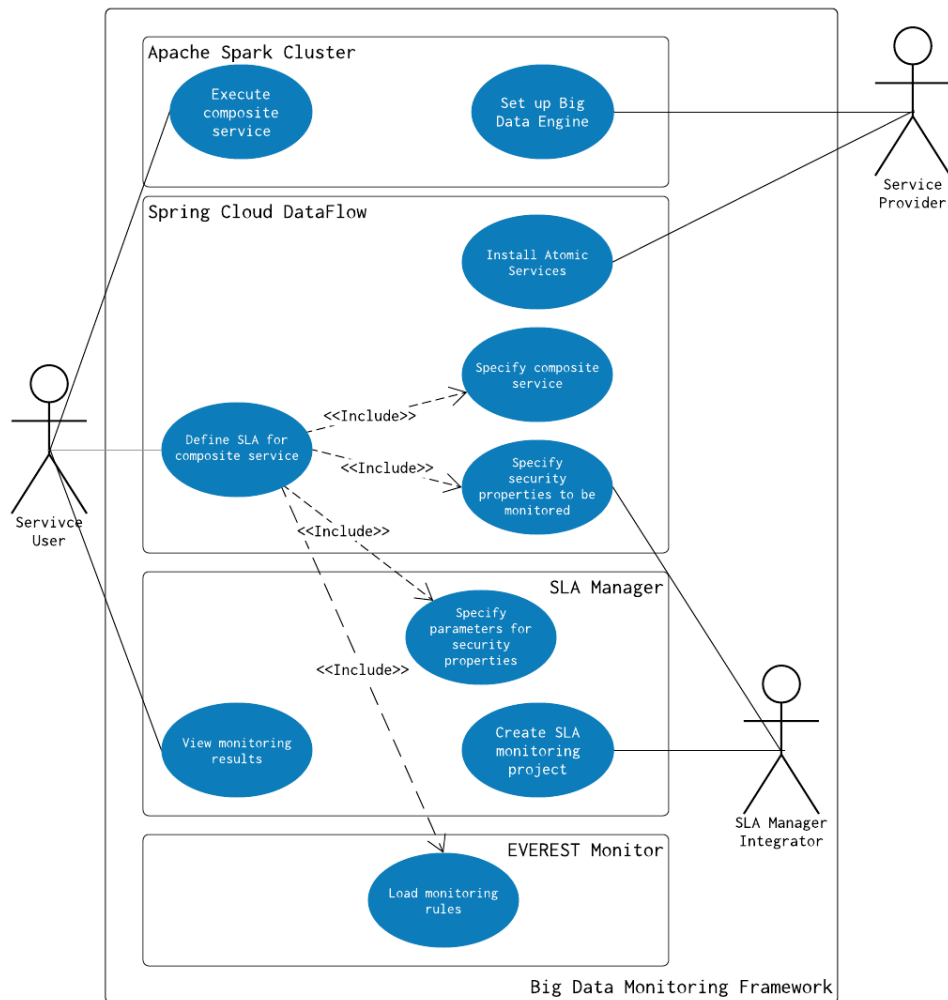


Fig. 3.2 Use Case UML diagram of the Big Data monitoring framework

To better describe the sequence of actions that are required for the runtime monitoring of security properties from our framework we will list all the steps in an ordered list of steps. The list is as follows:

1. Create a composite service through the *Spring Cloud DataFlow* web UI
2. (a) Add an atomic service
 - (b) Associate the atomic service with a security property

3. The *SLA Manager Interceptor* automatically creates and stores a new SLA monitoring project in the *SLA Manager*
4. In the *SLA Manager*, navigate to the newly created SLA and provide the template alongside its parameter values that the user chooses to use for the monitoring of each security property
5. The *SLA Manager* when the provision of all the necessary metadata with regards to what needs to be monitored, it automatically translates the user's high level security requirements into EC-Assertion formulae that it uploads on the EVEREST monitor
6. The user executes the Big Data pipeline i.e. the composite service through the *Spring Cloud DataFlow* server which in turn uses a Spring Boot application to deploy and execute the service on the Apache Spark cluster
7. Finally, through the *SLA Manager* can inspect the monitoring results for the SLA monitoring of interest

A formal view of the aforementioned steps that are required for the appropriate usage of the proposed framework can be viewed in figure 3.3.

3.2.1 Composite Service Definition

The big-data-as-a-service paradigm implies that a set of predefined Big Data applications are offered to users in the form of a software components that they can use in an out-of-the-box manner. The services are presented to the users as a list of pre-defined and pre-implemented *atomic services* that they can use to fulfil their own specific functional and non-functional requirements. To make this setup more useful, and to enable the implementation of more complex use-cases, our solution offers the ability to combine multiple atomic services to produce services of higher order i.e composite services. *Composite services* can incorporate additional semantics with respect to how the workflow should behave in case one or more atomic services fail to complete their execution. Failed atomic services can cause the whole computation to grind to a halt or allow it to continue and report the failure to the user. Also, atomic services can be combined to run sequentially or in parallel.

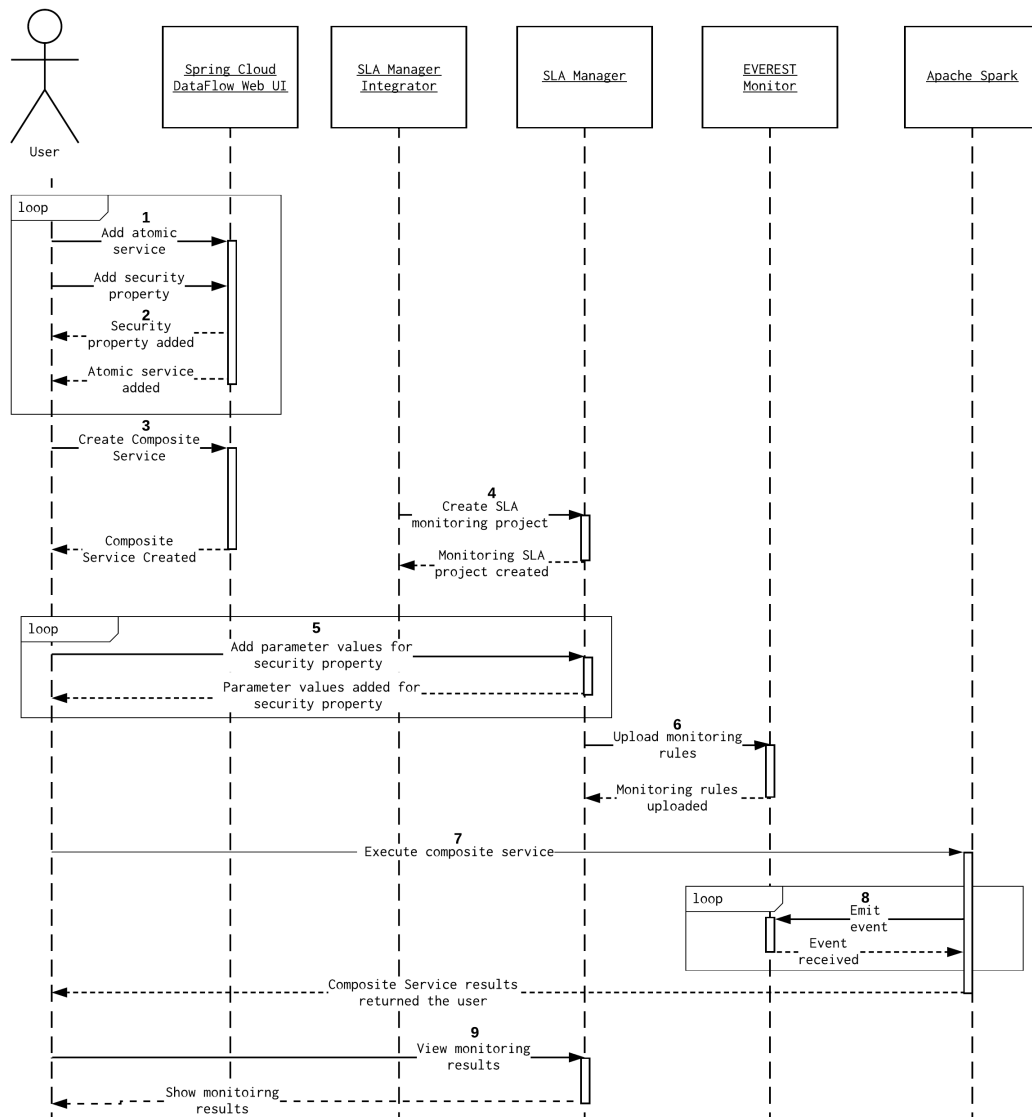


Fig. 3.3 Sequence diagram of the Big Data monitoring framework

The objective for the definition of the Big Data composite service workflow is to provide a declarative way of describing how the individual atomic service ought to be compiled to produce the desired result for the user. The definition strategy is by design declarative and not imperative, in an attempt to minimise the requirements for any particular knowledge from the user's end.

In our implementation, the declaration of the service pipeline is performed by mean of using *Spring Cloud Dataflow* (SCDF). SCDF offers a a set of software tools specifically

intended for the development of applications that are built for deployment in the cloud. These applications demonstrate traits such as loose coupling between software modules, capacity to take advantage of the elastic nature of the cloud and ability to run in a distributed fashion, to name a few.

SCDF comes pre-packaged with a domain specific language, both in Java and plain text, for the specification of composite services. The language enables the definition of the service workflows and makes its integration with other system intuitive and with minimal effort. The specification language is simple and declarative. It also provides rudimentary semantics for the declaration of flow control conditionals such as what action should be taken if an atomic services fails or if its status is changed to a custom user value. Finally, it supports the declaration of sequential and parallel executions of the individual atomic services.

Apart from the DSL, Spring Cloud DataFlow comes bundled with a built-in web UI where a drag-and-drop panel allows users to quickly and easily set up complex Big Data analytics pipelines.

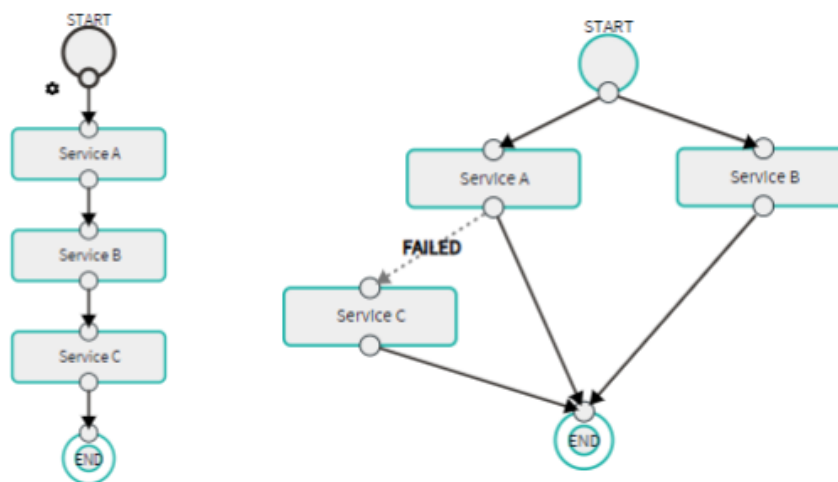


Fig. 3.4 Spring Cloud DataFlow pipelines

Figure 3.4 illustrates two examples of composite service pipelines. The view presented is the one offered by the web UI of Spring Cloud DataFlow. The pipeline on the left represents

a set of atomic services that run in sequence one after another. The pipeline on the right represents a composite service where **Service A** and *Service B* get execute in parallel.

3.2.2 Security Requirements Specification

The security requirement specification is a two step process. Firstly, during the workflow definition of the composite service, users need to specify what security properties need to be monitored on what atomic services. Secondly they need to specify the templates and parameter values that will be used from the monitoring framework to generate the monitoring rules and then evaluate the events against them. Defining the security properties in Spring Cloud DataFlow is a critical step because the pipeline needs to be aware of the security requirements to be able later on, when it is asked to execute the service, to associate the execution of each atomic service with the corresponding event captors. This enables the framework to capture only the minimum number of events that are absolutely mandatory to satisfy the monitoring activity of the security requirements defined. As soon as the composite service has been defined, via the SLA Manager web application UI, users can now open up the newly created SLA monitoring project and specify which security template is going to be used from the framework to generate concrete instances on the monitoring rules. Our goal for the specification of the security requirements is to be conducted by UI tools that are easy and intuitive for not technical users to use. This objective is attained first by using the web UI of *Spring Cloud DataFlow* when defining the workflow and second by using the *SLA Manager* web application where, through a set of dropdown lists and UI menu items, users can express complex security requirements for composite services without the need for a deep understanding of how they operate internally, what is required for their implementation and how the monitoring artefacts will have to be deployed to make the monitoring activity of the composite service possible.

3.2.3 Translation of Security Requirements into Monitoring artefacts

The translation of the security properties is a core functionality of the framework. In essence, it takes all the input from the users and automatically generates the EC-Assertion formulae that represent the monitoring rules that are required for the implementation of the

monitoring activity. The monitoring rules are the product of the security property templates that correspond to the security properties that the user requires to monitoring, populated with the parameter values that they have provided during the security requirement specification step as described in section 3.2.2 above. Monitoring rules are going to be represented as EC-Assertion formulae that can be used by the EVEREST monitor for the evaluation of the events. The generation of the monitoring rules is only possible when the security properties templates and their parameters have all been defined. Users, through the SLA Manager, have the ability to add the security at their own pace and when they have completed the process they will notify the system that the requirements specification process has been completed. This notification signifies the automatic generation of the monitoring rules in EC-Assertion formulae.

3.2.4 Installation of Monitoring Rules on the monitor

Once the monitoring rules have been generated, they need to be uploaded to the EVEREST monitor. The EVEREST monitor provides an API that allows the uploading of the rules by means of using a web service. The web service generates a unique identifier for the rule and uses the EC-Assertion expression that describes the monitoring rule to create internally in the monitoring all the necessary structures that will support the event evaluation activity. As soon as the monitoring rules are uploaded everything is in place to facilitate the monitoring of the composite service.

3.2.5 Definition and Installation of Event Captors on Apache Spark

The constituent components of event captors are comprised of two main components, namely the **interception** component i.e. what are the methods that need to be intercepted and the **delegation** component i.e. what should be executed when an intercepted method is invoked at runtime. Both conceptually and in terms of declaration, the interception component and the delegation component go hand-in-hand. Every delegation component needs to refer to an interceptor component. That is to say that for every event captor in the interceptor component one needs to first define which method will be intercepted and then, in the delegation component, give a description of what will be executed when

the intercepted method is invoked. The combination of initially intercepting and then delegating the intercepted code is key feature of all the event captors of the framework and is pattern that we have used across all three properties that we examined for the runtime code instrumentation of Spark's source code.

Apart from the collection of the events, the event captors are also responsible for the emission of the events to the monitor that will evaluate them. The process of event emission in our system has been designed with flexibility in mind. The event captor ought to be configurable with regards to the types of emission that the system should support. In our case we implemented two types of emitter namely a socket emitter where the events are sent over the network on an open socket and can be consumed from there and a RabbitMQ emitter where the events are sent to a RabbitMQ messaging system. Sending events at a socket has proven particularly useful during the development of the system for debugging purposes whereas sending events to a RabbitMQ server allows it to be consumed by the EVEREST monitor. From an implementation point of view, we have employed a rather common design pattern called a *Factory* pattern [125]. The factory pattern is a creational pattern that facilitates the creation of instances of classes that share the same behaviour with a systematic and consistent way. The only piece of information that is required from the factory class is the type of instance that it is required to instantiate. The factory method, based on the class type, can create a new instance and completely hide any additional parameters that might be required for the instantiation of the object. This pattern makes it very easy to add new emitter types only by means of implementing the methods of the base emitter type interface.

In our system for the implementation of the factory pattern we have been able to identify 3 basic operations that are required from any emitter:

1. Connect to the entity that the events will be sent
2. Sent an event in the form of a String
3. Close the connection and release any additional resources as soon as the event emission process has completed

The corresponding Java code snippet for the base emitter class can be seen in listing 3.1.

Listing 3.1 Base abstract class for description of the types of emitters

```
1 public abstract class Emitter {
2
3     public Properties properties = new Properties();
4
5     public abstract void connect();
6     public abstract void close();
7     public abstract void send(String event);
8 }
```

Note that in line 3 a global variable for properties is instantiated. This will enable the provision of a set of properties that will be passed on to the event emitter constructor to customise its operation. We need this feature to be available across all types of emitters and therefore it is appropriate to include it in the base emitter class.

The concrete implementation for the socket emitter is presented in listing 3.2.

Listing 3.2 SocketEmitter class implementation

```
1
2 public class SocketEmitter extends Emitter {
3
4     final static Logger logger = Logger.getLogger(SocketEmitter.class);
5
6     private String host;
7     private int port;
8     private Socket socket;
9     private BufferedWriter writer;
10
11     public SocketEmitter(Properties props){
12         this.properties = props;
13         this.host = properties.getProperty("host");
14         this.port = Integer.valueOf(properties.getProperty("port"));
15     }
16
17     @Override
18     public void connect() {
19         try {
20             this.socket = new Socket(host, port);
21             this.writer = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
22         } catch (IOException ioe) {
23             logger.info(ioe);
24         }
25     }
26
27     @Override
28     public void close() {
29         try{
30             if (socket != null)
31                 socket.close();
32             if (writer != null){
33                 writer.flush();
34                 writer.close();
35             }
36         } catch (IOException ioe){
37             logger.error(ioe);
38         }
39     }
40 }
```

```

41     @Override
42     public void send(String event) {
43         try {
44             writer.write(event);
45             writer.newLine();
46             writer.flush();
47         } catch (IOException ioe) {
48             logger.error(ioe);
49         }
50     }
51 }

```

In line 12 the properties variable is provided during construction. The properties that are appropriate for the emission of event on a socket are shown in line 13 and 14 where the host and the port number are defined respectively. In line 18 a socket is opened up on the host and port specified. Also a buffered writer is instantiated. In line 28 all the resources that were opened up when the emitter was connected are freed. Finally, in line 42 an event is written on the previously opened up buffered writer and the buffers is flushed to enforce the emission of the event.

The concrete implementation for the RabbitMQ emitter is presented in listing 3.3.

Listing 3.3 RabbitMQEmitter class implementation

```

1  public class RabbitMQEmitter extends Emitter {
2
3      private Connection connection;
4      private Channel channel;
5      private String channelName;
6      private String topic;
7
8      final static Logger logger = Logger.getLogger(RabbitMQEmitter.class);
9
10     RabbitMQEmitter(Properties props) {
11         this.properties = props;
12     }
13
14     @Override
15     public void connect() {
16         try {
17             ConnectionFactory factory = new ConnectionFactory();
18             factory.setHost(properties.getProperty("host"));
19             factory.setPort(Integer.valueOf(properties.getProperty("port")));
20             factory.setUsername(properties.getProperty("username"));
21             factory.setPassword(properties.getProperty("password"));
22             this.connection = factory.newConnection();
23             channel = this.connection.createChannel();
24             channel.exchangeDeclare(properties.getProperty("channel"), "direct");
25             channelName = properties.getProperty("channel");
26             topic = properties.getProperty("topic");
27         } catch (IOException ioe) {
28             logger.error(ioe);
29         } catch (TimeoutException te) {
30             logger.error(te);
31         }
32     }
33 }

```

```

34     @Override
35     public void close() {
36         try {
37             connection.close();
38         } catch (IOException ioe) {
39             logger.error(ioe);
40         }
41     }
42
43     @Override
44     public void send(String event) {
45         try {
46             if(!connection.isOpen())
47                 this.connect();
48             channel.basicPublish(channelName, topic, null, event.getBytes());
49         } catch (IOException ioe) {
50             logger.error(ioe);
51         }
52     }
53 }

```

From line 3 to line 6 a set of properties are defined that are mandatory for the establishment of communication between the event captors and the RabbitMQ messaging event bus that the EVEREST monitor is using. Also a username and password can be passed along if the communication between the event captors and the messaging engine is username and password protected. The communication of the event captor with the RabbitMQ server is done with the assistance of the RabbitMQ's standard client Java library ¹.

Finally the class that connects all the emitters and allows the instantiation of the different implementations of emitter is the factory class that is consistent with the factory pattern. The factory class is presented in listing 3.4.

Listing 3.4 EventEmitterFactory class implementation

```

1 public class EventEmitterFactory {
2
3     private EventEmitterFactory EventEmitterFactory(){
4         return new EventEmitterFactory();
5     }
6
7     public static Emitter getInstance(EmitterType type, Properties props){
8
9         Emitter emitter = null;
10
11         switch (type){
12             case RABBITMQ:
13                 emitter = new RabbitMQEmitter(props);
14                 break;
15             case SOCKET:
16                 emitter = new SocketEmitter(props);
17                 break;
18             case XMPP:
19                 break;

```

¹<https://www.rabbitmq.com/java-client.html>

```
20         default:
21             emitter = new RabbitMQEmitter(props);
22         }
23
24         return emitter;
25     }
```

Note in line 7 that the emitter factory needs to know the type of emitter type it needs to create. The types of emitters is defined as an enumeration that is presented in listing 3.5. Along with the emitter type a set of the right properties needs to be passed on to the emitter factory.

Listing 3.5 Emitter type enumeration class

```
1 public enum EmitterType {
2     SOCKET, RABBITMQ
3 }
```

The use of the factory pattern for the support of different emitter has two main benefits. Firstly, it abstracts away all the complex details that are required for the event captors to establish the communication i.e. connect, send events and close the connection with the entity that they will emit the events to. Secondly it keeps the code simple and tidy by means of taking advantage of the the separation of responsibilities principle where each class is responsible for one thing and one thing only. The factory class is responsible for the creation of the emitters and each emitter type class is responsible for the description of how the communication between the emitter and the receiving end of the events will be conducted. Also, by having the concrete implementation of emitters implement the *Emitter* interface, the requirements for the implementation of any new type of emitter is well-defined and bounded by the functionality included in their common interface.

The UML class diagram for the factory pattern of event captor emitter is illustrated in figure 3.5.

On the basis of the security properties that have been coupled with the individual atomic services, a series of events captors will be installed to collect the monitoring events. From an implementation point of view the event captors operate by means of instrumenting the underlying Big Data processing framework to collect and emit contextual run-time information to the monitor. The technology that we used for the code instrumentation is a built-in feature of the Java Virtual Machine (JVM) called Java agents ². A Java agent is a

²<https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html>

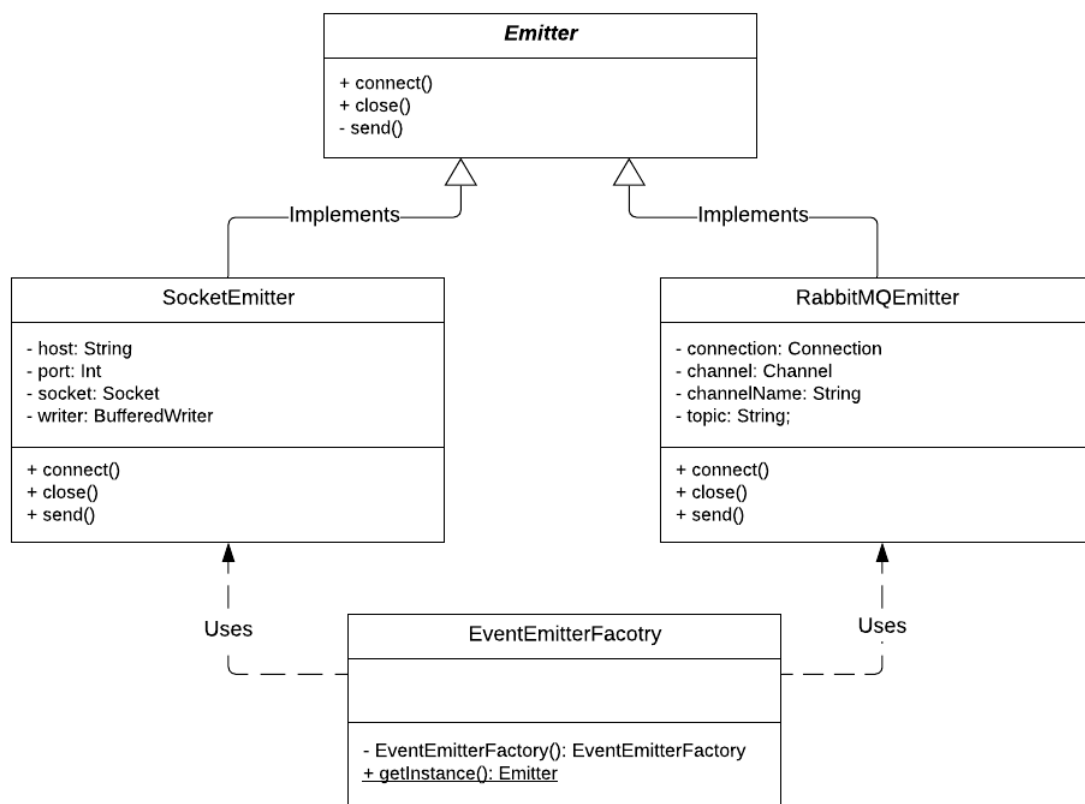


Fig. 3.5 UML class diagram of the factory pattern for the implementation of the different emitter types supported by the event captors

special type of Java class that must implement a special method called ***premain()***. Java agents make use of Java's standard instrumentation API and allows for the run-time modification of the code. When all the classes of a Java application get loaded during the instantiation from the JVM, by default it looks up for a ***premain()*** method within the Java agent. If one exists, it executes its body. The ***premain()*** method takes as a parameter a String and a reference to the instrumentation object. The reference to the instrumentation object provides a programmatic hook where the agents can inject the event capturing code.

From a programming point of view, Java's standard instrumentation API is low level. It gives back to the agent's creator a handler to the Java bytecode in the form of an array of bytes which is difficult for humans both to understand and manipulate. To bridge that gap we make use of the Byte Buddy [2] library that offers a high level API for manipulating Java bytecode. Byte Buddy exposes a builder API that allows the interception of native Java

code within methods, the creation of arbitrary classes and the run-time implementation of interfaces and abstract classes. Byte Buddy also provides a convenient tool for the authoring of Java agents by means of abstracting away the compiled bytecode with its corresponding Java code. As a consequence, coding the event captors with the assistance of Byte Buddy, feels more like coding in Java and less like coding in low level bytecode JVM instructions.

By definition, Java agents get executed before the main method gets executed. This feature enables the event captors to instrument Apache Spark's computational methods before they are executed from the Spark worker nodes. Java agents are applications like any other typical Java application except for two fundamental differences. Firstly they do not implement a *main()* method but a *premain()* method and secondly in the manifest file under the name MANIFEST.MF of the Java application of the agent a special entry name *Premain-Class* need to be set and point to the full package of the class where the *premain()* method is contained. To instruct the JVM to include the Java agent into Java's execution workflow, one has to pass the *-javaagent* argument when executing the code that needs to be instrumented. For instance, if the Java agent is named *agent.jar* that needs to run with a set of optional parameters named *parameters* and the application code to be instrumented is name *application.jar*, to run the application with the agent enabled, one should execute the following command:

```
java -jar -javaagent:/path/to/agent.jar[=parameters] application.jar
```

Since Apache Spark is a Big Data processing framework that enables the distributed execution of processing tasks across a computer cluster, it is critical to instrument the code that will be executed in such a distributed setting. This is a significant difference compared to other traditional runtime instrumentation solutions where the instrumentation of the application code takes place on a single physical engine. This implies that the instrumentation has to be carried out on every node that processes the data. From an execution point of view, when running an Apache Spark program, we use the *spark-submit* command. This command initiates the instantiation of multiple JVM instances across the cluster to enable the execution of the distributed tasks that will be scheduled for execution and subsequently executes the tasks at hand. As presented further up in this section, to include the Java agent into the execution of an application, we need to pass the *-javaagent* parameter as an argument to the

java command. However, when submitting an Apache Spark job, we do not have access to the actual java command that gets executed. Therefore, to intercept Apache Spark's code we need to make use of two special configuration properties; one for the driver node and one for the workers. That is because some of the event capturing needs to happen at the level of the worker nodes and some of it needs to happen at the level of the node where the driver gets executed. In Spark applications are submitted for execution through a standard utility named *spark-submit*³. The *spark-submit* command sets up the default Spark and invokes the *main()* method of the Spark application that is submitted for execution. A complete list of all the configuration parameters that can be set through the *spark-submit* command can be found in [7].

To add the Java agent at the Spark master node we need to add the following argument to the *spark-submit* command:

Listing 3.6 Configuration property to enable a Java agent at the driver node

```
1 --conf "spark.driver.extraJavaOptions=-javaagent:/path/to/agent.jar"
```

To add the Java agent at the Spark worker nodes we need to add the following argument to the *spark-submit* command:

Listing 3.7 Configuration property to enable a Java agent at a worker node

```
1 --conf "spark.executor.extraJavaOptions=-javaagent:/path/to/agent.jar"
```

These parameters will be passed to the *spark-submit* command when the composite service is going to be executed from the Spring Cloud DataFlow server on the Apache Spark cluster. Note that the Spring Cloud DataFlow is only responsible for triggering the execution of the composite service by submitting each atomic service to Spark and supervising the execution flow of the pipeline as it has been defined by the user. The engine that performs the actual processing is Apache Spark and that is why the Java agents that represent the event captors need to be passed as parameters at the Spark's executable code through the *spark-submit* parameters configuration API.

For the Java agents to operate we also need to provide a set of parameter with regards to what type of emitter the event captor will use as well as a set of other properties that the emitter that has been specified will require. For instance the socket emitter will require a host

³<https://spark.apache.org/docs/latest/submitting-applications.html>

and a port. A complete example for submitting a Spark job where the event captor for data integrity needs to be installed is shown in listing 3.8.

Listing 3.8 Spark-submit command example with data integrity event captors enabled

```

1 spark-submit \
2 --master spark://10.207.1.102:7077 \
3 --class package.main.MyClass \
4 --deploy-mode client \
5 --conf "spark.driver.extraJavaOptions=-javaagent:/path/to/agent/DataIntegrityEventCaptors.jar
6 =emitter=socket,host=10.207.1.103,port=10333" \
7 --conf "spark.executor.extraJavaOptions=-javaagent:/path/to/agent/DataIntegrityEventCaptors.jar
8 =emitter=socket,host=10.207.1.103,port=10333" \
9 /path/to/spark-application/application.jar

```

Note that after the complete path of the location of the data integrity event captor under the name *DataIntegrityEventCaptors.jar*, there exist an equals sign. After that a comma separate list of properties is defined accompanied by their respective values. This list of properties refers to the properties variable of the *Emitter* class shown in listing 3.1 and it is set during the submission of the job for execution. With that, the Java agent has all the necessary information to create the appropriate emitter and start emitting events the right events for the evaluation of data integrity.

3.3 Monitoring Rules

In this section we give an account of the monitoring rules that are being produced and that will be used for the evaluation of the events. Originally, users give a description of the monitoring capabilities that they require to enable for the Big Data services of a pipeline. The collection of all the security requirements constitute the service level objectives (SLOs) of the SLA. In our implementation, for each security property that we examine we give a detailed view of the Event Calculus rules and assumptions that describe what constitutes a violation, we give a description of the SLA template that corresponds to each property that will be used to produce the EC-Assertions used by the monitor and finally we provide an overview of the corresponding event captors that will support the emission of the events required to realise the monitoring activity each service level objective.

3.3.1 Monitoring Rules for Response Time

In the context of data availability, the metric that we are using to evaluate it, is response time. As shown in our literature review in section 2.2.1, one of the ways to gauge availability and give users a tangible measurement of data availability is response time. Response time in that context, is a measurement of how long it takes the system to make its results available. Gaining access at the results of a computation on time, is a critical aspect of a service request/response interaction and it can be the reason for poor user experience. Every atomic service of the pipeline is treated as an Apache Spark job and is the action that signifies the execution of the service's operations. Having said that, the measurement of response time is calculated as *"the time that has elapsed from the moment the job starts until the moment it has produced all its results"*. As explained in section A.2.1, actions trigger the execution of transformations and instruct Apache Spark to start computing realising the transformations that are part of the Spark program. In this context actions and jobs are used interchangeably and refer to the same concept.

Event Calculus

The Event Calculus rule needs to describe the fact that when a job starts to get executed it should be finished before a specific amount of time has elapsed. If the time that has elapsed is longer than the user-defined threshold then the rule should be violated to signify a longer response time than what the user requires. The relevant events that need to be emitted are a start job event and an end job event alongside with their timestamps. If the delta between the timestamp between the start and end events is longer than what the user has described, this constitutes a violation. The detailed description of the relevant events is Event Calculus notation is as follows:

<i>start</i> (<i>appId</i> , <i>appName</i> , <i>t</i>)
<i>appId</i> : Application id
<i>appName</i> : Application name
<i>t</i> : Timestamp when the start event occurred

Table 3.1 Events collected for monitoring response time - **start** job event

$end(appId, appName, t')$
$appId$: Application id
$appName$: Application name
t' : Timestamp when the end event occurred

Table 3.2 Events collected for monitoring response time - **end** job event

Rule 1
$\exists t_1 \geq 0, t_2 \geq 0,$ $Happens(start(appId, appName, t_1)) \rightarrow Happens(end(appId, appName, t_2)),$ $t_2 \in [t_1, t_1 + d]$

Table 3.3 Event Calculus rule for monitoring response time

For completeness the application's unique identifier that is assigned by Apache Spark, needs to be collected in the *appId* parameter as well as the application name that was given to the job when it was submitted for execution under the parameter *appName*. Both *appId* and *appName* are required because the monitor can potentially have to reason about multiple services that can get executed at the same time and a unique identifier needs to distinguish events that refer to different service executions. Even though the *appId* parameter would suffice from an implementation point of view to group events per service execution, we also collect the application name to provide a humanly readable label to help users interpret the monitoring results when they are presented to them in the web dashboard.

In table 3.3 below we present the Event Calculus rule in the appropriate notation to reflect the fact that as soon as a **start** event has been emitted, we expect an **end** event to be emitted within a predefined time d to avoid violating the service level objective. The value of d , the time that has elapsed, will be provided from the end-user and is going to be part of the SLA guarantee term. In our implementation we have taken into consideration the measurement unit to match the measurement unit of the timestamp of the events that have been emitted. This is important for the proper evaluation of the event against the rule.

Typically, to allow for a sufficient degree of granularity, event timestamps are measured in Unix milliseconds. An example of a start and end events is shown in table 3.4.

<i>start</i> (app-20181202162554-0401, AverageConsumptionCombineByKey, 1543865904)
<i>end</i> (app-20181202162554-0401, AverageConsumptionCombineByKey, 1543873201)

Table 3.4 Example of events for monitoring response time

In this particular example the response time i.e. the time that has elapsed from the moment the request has been placed until the moment the results of the computation have become available is $(1543865904 - 1543879201) = 13297ms$ or $13.297sec$. If, for example, the acceptable time provided by the users is set to 1 minute, the rule has been respected and no violation has occurred. A visual representation of how events can occur over time during the monitoring activity of response time is events, can be seen the in figure 3.6.

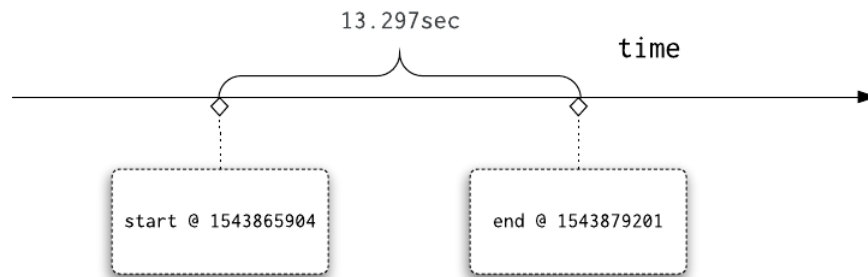


Fig. 3.6 Visual representation of events for monitoring response time

SLA Template specification

The SLA template for response time will produce the EC-Assertion rule that corresponds to the Event Calculus rule presented in table 3.3 that will later on will be loaded on Everest, our event reasoning engine, to support the monitoring of the SLA. For the monitoring of response time, there exist two aspects of the rule that need to be parameterized, namely the **response time** as an arithmetic value and the **unit of measurement**. In our definition of

the SLA template we support positive integer arithmetic values and for the time units we support seconds, minutes and hours respectively. In the SLA template for those two values two separate placeholders will be used and they will be replaced as soon as the security requirements have been provided by the end-users. Based on the time unit, the template will make a calculation and will produce the correct value for the acceptable time interval between the starting and ending time of a Spark job. An abbreviated version of the SLA template can be seen in listing 3.9.

Listing 3.9 Abbreviated version of the SLA template for response time

```

1  if( $timeUnits == "SECONDS" )
2      #set($result = $time * 1000)
3  elseif( $timeUnits == "MINUTES" )
4      #set($result = $time * 1000 * 60)
5  elseif( $timeUnits == "HOURS" )
6      #set($result = $time * 1000 * 60 * 24)
7  #end
8
9  Guaranteed forChecking="true" ID="availability" type="Future_Formula">
10 <quantification>
11     <quantifier>forall</quantifier>
12     <timeVariable>
13         <varName>t1</varName>
14         <varType>TimeVariable</varType>
15     </timeVariable>
16 </quantification>
17 <quantification>
18     <quantifier>existential</quantifier>
19     <timeVariable>
20         <varName>t2</varName>
21         <varType>TimeVariable</varType>
22     </timeVariable>
23 </quantification>
24 <precondition>
25     <atomicCondition conditionID="ac0">
26         <eventCondition unconstrained="true">
27             <event>
28                 <eventID forMatching="true" persistent="false">
29                     <varName>AVAIL1</varName>
30                 </eventID>
31                 <call>
32                     <interfaceId>BDASLA</interfaceId>
33                     <OperationId>1</OperationId>
34                     <operationName>start</operationName>
35                     <inputVariable forMatching="true" persistent="false">
36                         <varName>status1</varName>
37                         <varType>OpStatus</varType>
38                         <value>REQ-A</value>
39                     </inputVariable>
40                     <inputVariable forMatching="true" persistent="false">
41                         <varName>appId</varName>
42                         <varType>string</varType>
43                     </inputVariable>
44                     <inputVariable forMatching="true" persistent="false">
45                         <varName>appName</varName>
46                         <varType>string</varType>
47                     </inputVariable>

```

```

48         </call>
49         <tVar>
50             <timeVar>
51                 <varName>t1</varName>
52                 <varType>TimeVariable</varType>
53             </timeVar>
54         </tVar>
55         <fromTime>
56             <time>
57                 <varName>t1</varName>
58                 <varType>TimeVariable</varType>
59             </time>
60         </fromTime>
61         <toTime>
62             <time>
63                 <varName>t1</varName>
64                 <varType>TimeVariable</varType>
65             </time>
66         </toTime>
67     </event>
68 </eventCondition>
69 </atomicCondition>
70 </precondition>
71 <postcondition>
72     <atomicCondition conditionID="ac1">
73         <eventCondition unconstrained="false">
74             <event>
75                 <eventID forMatching="true" persistent="false">
76                     <varName>AVAL2</varName>
77                 </eventID>
78                 <reply>
79                     <interfaceId>BDASLA</interfaceId>
80                     <operationId>2</operationId>
81                     <operationName>end</operationName>
82                     <outputVariable forMatching="true" persistent="false">
83                         <varName>status2</varName>
84                         <varType>OpStatus</varType>
85                         <value>RES-B</value>
86                     </outputVariable>
87                     <outputVariable forMatching="true" persistent="false">
88                         <varName>appId</varName>
89                         <varType>string</varType>
90                     </outputVariable>
91                     <outputVariable forMatching="true" persistent="false">
92                         <varName>appName</varName>
93                         <varType>string</varType>
94                     </outputVariable>
95                 </reply>
96             </event>
97             <tVar>
98                 <timeVar>
99                     <varName>t2</varName>
100                     <varType>TimeVariable</varType>
101                 </timeVar>
102             </tVar>
103             <fromTime>
104                 <time>
105                     <varName>t1</varName>
106                     <varType>TimeVariable</varType>
107                 </time>
108             </fromTime>
109             <toTime>
110                 <time>
111                     <varName>t1</varName>
112                     <varType>TimeVariable</varType>

```

```

112         </time>
113         <Expression>
114             <plus>$result</plus>
115         </Expression>
116     </toTime>
117 </event>
118 </eventCondition>
119 </atomicCondition>
120 </postcondition>
121 </Guaranteed>

```

Let's examine each segment of the SLA template in greater details. From line 1 until line 7 an Apache Velocity snippet is responsible for the conversion of the time value provided by the user into the correct number of milliseconds based on the time unit of the events emitted. In the segment from line 9 until line 121, the guaranteed terms are defined. In the case of response time only one guarantee terms is required. At the beginning of the guarantee term specification, from line 10 until line 23, the qualifiers for the time variables are provided, namely variables t_1 and t_2 which in both occasions is an existential qualifier. Also note that the variable type for both variables is *TimeVariable*. Further down the guarantee term is broken down into a precondition segment, line 24 until line 70, and a postcondition segment, from line 71 until line 120. The precondition corresponds to the left part of the Event Calculus rule shown in table 3.3 and the postcondition corresponds to the right part of the rule after the right arrow i.e. the implication symbol. Both in precondition we define the input and output variables namely the *appId* and the *appName*. At runtime both variables will contain actual values and they will be used to unify the response time event calculus rule with events that belong in the same execution context. Finally, note how in the segment from line 96 until line 116 the value for t_2 is bound from values t_1 and $t_1 + d$. This instructs the monitor to check if the *end* event has occurred within a specific amount of time since the *start* event has been captured.

Event Captor Specification

The event captors for response time are tightly coupled with Apache Spark actions that commence the computation of a job. As explained in section A.2.1, all other operations in Apache Spark are lazily evaluated and are not executed until an action is invoked. Before we delve into the event captor implementation details, have to give an overview of the Apache Spark code that the event captor will intercept at runtime to support the emission of

the *start* and *end* events respectively. Apache Spark is written, for the most part, in Scala while some pieces of it are written in Java. Specifically, the main entry point function that executes an action is implemented in Scala, is called *runJob()* and is located in a class called *SparkContext*. The relevant code snippet for the *runJob()* function can be seen in listing 3.10.

Listing 3.10 Apache Spark Scala code for *runJob()* function

```
1
2 def runJob[T, U: ClassTag](rdd: RDD[T], func: (TaskContext, Iterator[T]) => U): Array[U] = {
3   runJob(rdd, func, 0 until rdd.partitions.length)
4 }
```

As it can be seen in the body of the *runJob()* function, function *func* is applied on all partitions. The *runJob()* is a base interface that all Apache Spark actions will invoke to start a computation. For instance, when the *collect()* action is invoked on an RDD, all the data items from the partitions of the RDD will be sent to the driver and will become available for further processing or presentation to the job submitter. By the same token, when the *foreach()* action is invoked, a custom user function will be applied on all the data items of the partitions of the RDD without however sending the data to the driver. In both cases, *collect()* and *foreach()* functions will internally invoke the *runJob()* base function with a set of different parameters. Please note that our decision to intercept the base function for the execution of Spark jobs is deliberate. By instrumenting the code of the base function we do not explicitly have to intercept every action separately. This simplifies the implementation of the event captor, helps us to avoid the introduction of unnecessary code and in our view is a more sensible approach. Also, our implementation is more complete because it addresses the interception of all possible actions that are supported by the Apache Spark API.

A list with all the actions that are supported by the response time event captor is demonstrated in figure 3.7.

Note that our implementation supports the standard RDD Spark API where actions are invoked on Spark's core abstraction, namely RDDs, and the streaming API as well. Similar to the core API, Spark's streaming API is built on RDDs as well and is implemented using a micro-batching approach i.e. at specific time intervals a short batch job gets executed to collect, process and combine with the old data the newly arrived data of a stream. This feature contributes to the generality of the event captor and can support the monitoring activity both for batch and stream processing types of Big Data analytics applications.

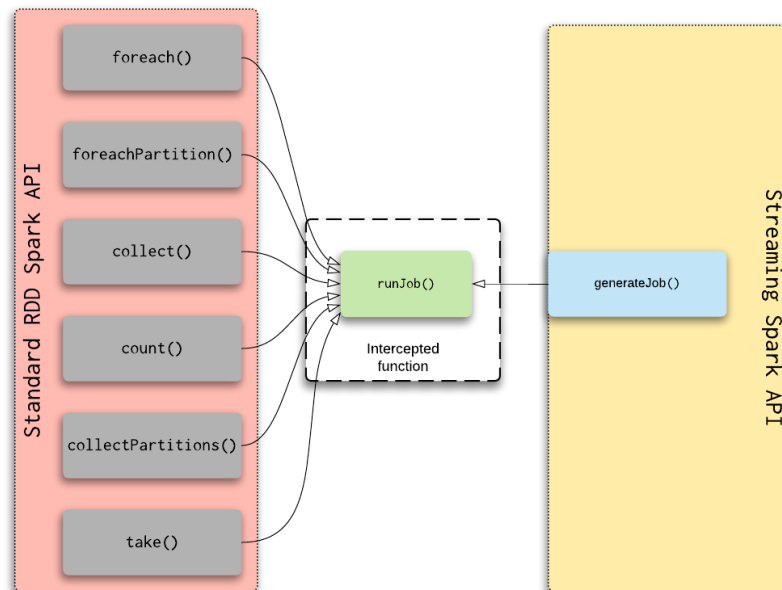


Fig. 3.7 List of actions supported by the event captor for response time

Interception Component

As explained above, the monitoring activity for response time requires the interception of the *runJob()* method which is declared in class *SparkContext*. Therefore, with the assistance of Byte Buddy as shown in section A.5.3, a *premain()* method has been implemented as part of the Java agent that will be loaded before Spark's code gets executed. In the body of the *premain()* method, with the assistance of Byte Buddy's *AgentBuilder* factory class, a new Java agent is created and installed. The relevant code snippet for the *premain()* method is shown in listing 3.11 below.

Listing 3.11 Interception component of event captor for response time

```

1 public static void premain(String configuration, Instrumentation instrumentation) {
2
3     new AgentBuilder.Default()
4         .type(type -> type.getName().equals("org.apache.spark.SparkContext"))
5         .transform((builder, typeDescription, classLoader, module) -> {
6             return builder
7                 .serialVersionUID(1L)
8                 .method(method -> (method.getName().equals("runJob") && method.getParameters().size() == 3))
9                 .intercept(MethodDelegation.withDefaultConfiguration()
10                     .withBinders(Morph.Binder.install(Morpher.class))
11                     .to(SparkContextRunJobInterceptor.class));
12         })
13         .installOn(instrumentation);
14 }

```

Note that in line 4 the full class name for *org.apache.spark.SparkContext* is defined whereas further down in line 8 the method called *runJob* that takes 3 arguments is defined as well. The assisiation of the interception component with the delegation component is done in line 11 and is implemented in a separate class called *SparkContextRunJobInterceptor*.

Delegation Component

As shown in the section 3.3.1 above, the the delegation method is implemented in the *SparkContextRunJobInterceptor* class. For consistency, we use the same name for the delegation method as the original name of the method that was intercepted. The relevant code snippet for the delegation method is presented in listing 3.12 below.

Listing 3.12 Delegation component of event captor for response time

```

1 public static class SparkContextRunJobInterceptor<T, U> {
2
3     @RuntimeType
4     public static Object[] runJob(@Argument(0) RDD rdd,
5         @Argument(1) Object f,
6         @Argument(2) Object classTag,
7         @Morph Morpher<Object[]> m)
8         throws JAXBException,
9             UnknownHostException,
10             DatatypeConfigurationException {
11
12         String appId = SparkEnv$.MODULE$.get().conf().get("spark.app.id");
13         String appName = SparkEnv$.MODULE$.get().conf().get("spark.app.name");
14
15         long operationId = generateRandomLong();
16
17         Emitter emitter = EventEmitterFactory.getInstance(type, properties);
18         emitter.connect();
19         emitter.send(createEventXML(operationId, OperationType.ACTION, "start", appId, appName, rdd, null));
20
21         Object[] result = m.invoke(new Object[]{rdd, f, classTag});
22
23         emitter.send(createEventXML(operationId, OperationType.ACTION, "end", appId, appName, rdd, null));
24
25         emitter.close();
26         return result;
27     }
28 }
```

In lines 12 and 13 the application id and name are respectively collected from the Spark environment, a shared key/value structure where common configuration values for the application execution are stored. Also, in line 17 an instance of an emitter object is created from the *EventEmitterFactory* factory class whose type depends on the configuration parameters passed at the event captor. In lines 19 and 23 *start* and *stop* events are emitted respectively. Note that a special method namely *createEventXML()* that formats the event in

a manner that it can be interpreted by the EVEREST monitor but is omitted here for the sake of space. Moreover, the original method i.e. *runJob()* is invoked in line 21 and its output is temporarily stored in a local variable under the name *result*. Finally, in line 25 the *close()* method is called to close any open channel of communication that has been used for the emission of the events whereas in line 26 the *result* variable that contains the output of the invocation of the original method is returned.

3.3.2 Monitoring Rules for Location of Execution

In the context of data privacy, the metric that we are using to evaluate it, is location of execution of the computations that make up the Big Data processing job. In our analysis of the literature in section 2.2.2, we list that the sharing data with Cloud providers that respect the privacy of its owner, is one of the pillars of data privacy. Execution of the computations of a Big Data analytics service, entails that computations will be distributed across multiple nodes in a cluster. It is very typical for Cloud storage and processing providers to use processing engines that are physically dispersed across the globe. This enables them to group the world into operational regions. Based on the location of the agent that makes the request, they can use infrastructure that is physically closer to that agent and thus minimize the latency of the response. Also, for fail-over capabilities, Cloud providers mirror their systems and have replicas of their infrastructure at different location around the world to take over if the infrastructure becomes unavailable in one location. This implies that computations of Big Data services that get executed on behalf of users in the Cloud, can be computed on computing engines that reside outside the geographical barriers that might be appropriate for the data owner leading to a compromise of data privacy.

An additional challenge is that the transfer of data can take place internally from the Cloud provider in an obscure manner and without the consent of the data owner. From a performance perspective this might facilitate an improved experience for the user however this can occur on the expense of a less stringent data privacy policy which might not be desirable for the user. A typical requirement might mandate that sensitive information, such as governmental data, should not be transferred for processing on a geographical region where the legislation regarding data ownership is not the one that the data owner requires. This is a case of particular interest especially with the enactment of the GDPR directive [46]

where consent from the users with regards how that data will be used and where it is going to be stored and processed is mandatory. In general description of the location of execution rule requires that *"all computations must take place on machines that the user trusts"*.

Event Calculus

The Event Calculus rule for the monitoring of location of execution needs to describe the fact the partitions of all the intermediate RDDs of Big Data processing computation should be performed on locations that are trusted by the user. Therefore the monitor should receive events that will correlate partitions, RDDs and IP addresses to signify that a partition with a specific id, for an RDD with a specific id has been processed on a machine with a specific IP address. Note that partitions have an identifier in each partition that is unique per RDD, and each RDD has a identifier that is unique across the whole computation. Therefore a reference to the combination of a partition id and an RDD id is unique across the computation as well. If a partition for an RDD is processed on a machine whose IP address is not trusted by the user, then a monitoring violation must be raised by the monitor.

The way the rule is defined is by allowing users to define a whitelist of IP addresses that they trust their data to be processed on. From an event calculus perspective this is described as an assumption. The monitoring rule evaluates this assumption for the events that are emitted where the partition id, the RDD id and the IP address are included and if it evaluates to true then the computation has occurred on a machine that is trusted by the user. Conversely, if it evaluates to false then the computations has taken place on a machine that is not in the whitelist and therefore the rule is not respected. Similar to the the other properties, when events are collected the application id *appId* and application name *appName* need to be collected as well to allow the monitor to distinguish between multiple executions of the same service. A description of the events in event calculus notation that the monitor will have to evaluate can be seen in table 3.5.

Example of events intended for the monitoring of the location of execution can be seen in table 3.6.

The event calculus assumption and rule that supports the monitoring activity for the location of execution is presented in table 3.7 below.

<i>compute</i> (<i>appId</i> , <i>appName</i> , <i>partId</i> , <i>rddId</i> , <i>IP</i> , <i>t</i>)
<i>appId</i> : Application id
<i>appName</i> : Application name
<i>partId</i> : Partition id
<i>rddId</i> : RDD id
<i>IP</i> : IP address of the node the computation is executed
<i>t</i> : Timestamp when the compute event occurred

Table 3.5 Events collected for monitoring the location of execution - **compute** event

<i>compute</i> (appId=app-20181202162554-0401, appName=LoadAndAnonymize, partId=4, rddId=2, IP=10.207.1.104, t=1543865904)
<i>compute</i> (appId=app-20181202162554-0401, appName=LoadAndAnonymize, partId=2, rddId=3, IP=10.207.1.105, t=1543867839)

Table 3.6 Example of events collected for monitoring the location of execution

Assumption 1
$\forall t \geq 0,$ Initially (<i>trustedIP</i> (<i>ip</i> ₁ , <i>ip</i> ₂ , <i>ip</i> ₃ , ...))
Rule 1
$\forall t \geq 0,$ Happens (<i>compute</i> (<i>appId</i> , <i>appName</i> , <i>partId</i> , <i>rddId</i> , <i>ip</i> , <i>t</i>)) \rightarrow HoldsAt (<i>TrustedIP</i> (<i>ip</i>), <i>t</i>)

Table 3.7 Event calculus assumption and rule for monitoring the location of execution of computations

In **Assumption 1** shown in table 3.7 above, a fluent under the name *trustedIP* is initialized with a list of IP addresses that represent the whitelist of IP addresses that the user will provide as part of the security requirement specification. The fluent will be evaluated to true when an IP address that is trusted is passes as an argument. If the IP passed to the fluent is not in the set of IPs defined by the user, the fluent will evaluate to false. In the same table in **Rule 1**, when a computation takes place at some point in time *t*, the *trustedIP* fluent is checked at that same point in time to verify that the computation has been conducted on a trusted node. A mathematical representation of the *trusyedIP* fluent can be seen in the piecewise function shown in 3.1

$$trustedIP(ip, t) = \begin{cases} \text{TRUE}, & \text{if } ip = ip_1 \text{ or } ip = ip_2 \text{ or } ip = ip_3 \dots \text{ and } t \geq 0 \\ \text{FALSE}, & \text{otherwise} \end{cases} \quad (3.1)$$

Note when the *trustedIP* fluent is initialized at time *t* = 0, a comma separated list of IP addresses is provided as input to the *Initially* event calculus formula.

A visual representation of how events can occur over time during the monitoring activity of the location of execution of computations, can be seen in figure 3.8.

For the sake of space only some examples of events are illustrated to convey the concept. More specifically, at *t*=0 the *trustedIP* holds true for two specific IP addresses namely 10.207.1.102 and 10.207.1.104. As time goes on, compute events take place. At time

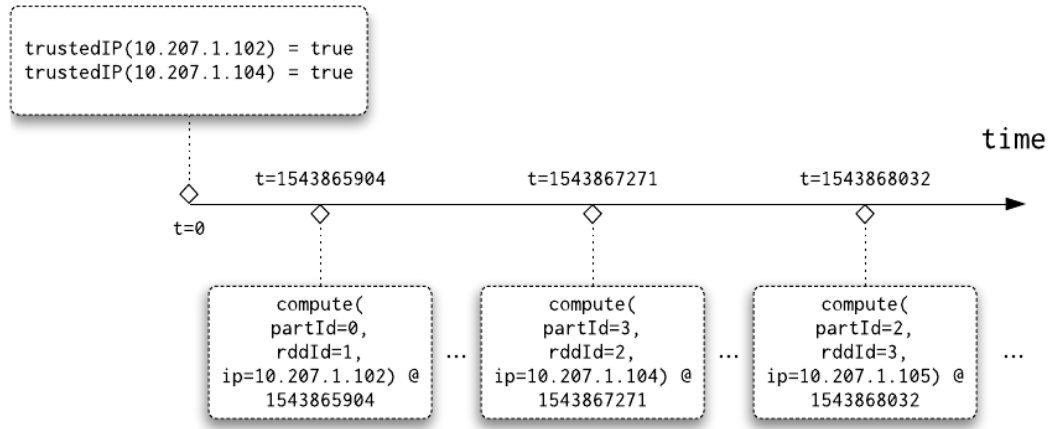


Fig. 3.8 Example of events that occur over time during the monitoring activity of the location of execution of computations

$t=1543865904$, partition with id 0, for RDD with id 1 is computed on the node with address 10.207.1.102. At time $t=1543867271$, partition with 3 for RDD with id 2 is computed on the node with IP address 10.207.1.104. At time $t=1543868032$ partition with id 2 for RDD with id 3 is computed on the node with IP address 10.207.1.105. Note that the rightmost event shown in figure 3.8 would cause a rule violation because the *trustedIP* fluent does not hold true for the IP value 10.207.1.105.

SLA Template specification

The SLA template specification for the location of execution will produce the EC-Assertion assumption and rule that corresponds to the event calculus assumption and rule presented in table 3.3.2. The EC-Assertion will be loaded on the Everest monitor to facilitate the monitoring activity of the relevant SLA. To produce a concrete SLA specification from the template, service users need to provide only one parameter under the name *trustedIPs* that represents the list of IP addresses that are acceptable for the computations to occur. Within the SLA template a relevant placeholder with the same name is used. The list of IP addresses will be provided as a comma separated list of strings. In the SLA template the list is broken down into the individual IPs of the list and added to the *trustedIP* fluent to initialize to true from the beginning of the computation i.e. when $t=0$. An abbreviated version of the

SLA template for the location of execution of the service's computations can be seen in the listing 3.13 below.

Listing 3.13 Abbreviated version of the SLA template for location of execution

```

1 <Guaranteed ID="trustedFluentID1" forChecking="false" type="future">
2   <quantification>
3     <quantifier>forall</quantifier>
4     <timeVariable>
5       <varName>t0</varName>
6       <varType>TimeVariable</varType>
7     </timeVariable>
8   </quantification>
9   <postcondition>
10    <atomicCondition conditionID="trustedFluentac1">
11      <stateCondition>
12        <initially>
13          <state name="trustedFluent">
14            <argument>
15              <variable forMatching="true" persistent="false">
16                <varName>trustedIP</varName>
17                <array>
18                  <type>stringArray</type>
19                  #set($count = 0)
20                  #foreach( $ip in $trustedIps )
21                    <value><indexValue>$count</indexValue>
22                      <cellValue>$ip</cellValue>
23                    </value>
24                  #set($count = $count + 1)
25                  #end
26                </array>
27              </variable>
28            </argument>
29          </state>
30          <timeVar>
31            <varName>t0</varName>
32            <varType>TimeVariable</varType>
33          </timeVar>
34        </initially>
35      </stateCondition>
36    </atomicCondition>
37  </postcondition>
38 </Guaranteed>
39 <Guaranteed forChecking="true" ID="PrivacyRule" type="Future_Formula">
40   <quantification>
41     <quantifier>forall</quantifier>
42     <timeVariable>
43       <varName>t1</varName>
44       <varType>TimeVariable</varType>
45     </timeVariable>
46   </quantification>
47   <precondition>
48     <atomicCondition conditionID="ac0">
49       <eventCondition unconstrained="true">
50         <event>
51           <eventID forMatching="true" persistent="false">
52             <varName>Privacy</varName>
53           </eventID>
54         <call>
55           <interfaceId>BDASLA</interfaceId>
56           <OperationId>1</OperationId>
57           <operationName>compute</operationName>
58           <inputVariable forMatching="true" persistent="false">

```

```

59         <varName>appId</varName>
60         <varType>string</varType>
61     </inputVariable>
62     <inputVariable forMatching="true" persistent="false">
63         <varName>appName</varName>
64         <varType>string</varType>
65     </inputVariable>
66     <inputVariable forMatching="true" persistent="false">
67         <varName>rddId</varName>
68         <varType>string</varType>
69     </inputVariable>
70     <inputVariable forMatching="true" persistent="false">
71         <varName>partId</varName>
72         <varType>string</varType>
73     </inputVariable>
74     <inputVariable forMatching="true" persistent="false">
75         <varName>ip</varName>
76         <varType>string</varType>
77     </inputVariable>
78 </call>
79 <tVar>
80     <timeVar>
81         <varName>t1</varName>
82         <varType>TimeVariable</varType>
83     </timeVar>
84 </tVar>
85 <fromTime>
86     <time>
87         <varName>t1</varName>
88         <varType>TimeVariable</varType>
89     </time>
90 </fromTime>
91 <toTime>
92     <time>
93         <varName>t1</varName>
94         <varType>TimeVariable</varType>
95     </time>
96 </toTime>
97 </event>
98 </eventCondition>
99 </atomicCondition>
100 </precondition>
101 <postcondition>
102     <atomicCondition conditionID="trustedAc1">
103         <stateCondition>
104             <holdsAt>
105                 <state name="trustedFluent">
106                     <argument>
107                         <variable forMatching="true" persistent="false">
108                             <varName>trustedIP</varName>
109                             <array>
110                                 <type>stringArray</type>
111                             </array>
112                         </variable>
113                     </argument>
114                 </state>
115             </holdsAt>
116             <timeVar>
117                 <varName>t1</varName>
118                 <varType>TimeVariable</varType>
119             </timeVar>
120         </stateCondition>
121     </atomicCondition>
122 </wrappedCondition>

```

```

123     <operator>and</operator>
124     <assertionCondition>
125         <atomicCondition conditionID="wc1">
126             <relationalCondition>
127                 <equal>
128                     <operand1>
129                         <operationCall>
130                             <name>exists</name>
131                             <partner>self</partner>
132                             <argument>
133                                 <variable forMatching="true" persistent="false">
134                                     <varName>trustedIP</varName>
135                                     <array>
136                                         <type>stringArray</type>
137                                     </array>
138                                 </variable>
139                             </argument>
140                             <argument>
141                                 <variable>
142                                     <varName>ip</varName>
143                                     <varType>string</varType>
144                                 </variable>
145                             </argument>
146                         </operationCall>
147                     </operand1>
148                     <operand2>
149                         <constant>
150                             <name>verified</name>
151                             <value>true</value>
152                         </constant>
153                     </operand2>
154                 </equal>
155             </relationalCondition>
156         </atomicCondition>
157     </assertionCondition>
158 </WrappedCondition>
159 </postcondition>
160 </Guaranteed>

```

Let's go through each segment of the SLA template and analyze it in greater detail. As it can be seen, the segment presented in listing 3.13 contains two guarantee terms to match the fact that two event calculus formulas are required for the monitoring activity for the location of execution of computation.

The definition of the assumption spans from line 1 until line 38. In this section, from line 2 until line 8, the *forall* existential qualifier is declared regarding the t_0 time variable of the assumption. This is because the fluent *trustedIP* that is included in the *Initially* event calculus formula, will be evaluated **for all** values of variable $t_0 \geq 0$ and for all the trusted IP addresses should evaluate to true. The assumption is defined from line 9 until line 37 as a postcondition. The name of the fluent is defined as *trustedIP* in line 16 and is of type

stringArray. The *stringArray* data type describes the fact the list of IP is passed to the SLA as parameter and is an array of strings. Also, from line 19 until line 25, a Velocity code snippet is responsible for the tokenization of the user's input in a list of strings. Note that the assumption for the *Initially* event calculus formula does not have a precondition. This is because, by definition according to event calculus' theory, *Initially* formulas do not require the occurrence of an event to hold true. They do hold true from the beginning of time i.e. when $t=0$.

The definition of the rule spans from line 39 until line 164 and is declared as a future formula. As in the assumption, the existential quantifier used for the t_1 variable of the rule is the *forall* quantifier. That is because the rule will be evaluated for all possible values of time variable $t_1 \geq 0$ that a compute event might occur. Finally, note that in the event that is defined in the precondition of the rule, the events that will be collected will be unified on the basis of the *appId*, *appName*, *partId*, *rddId* and *ip* variables.

Event Captor Specification

The event captor that will be used for the emission of the appropriate events for the location of execution of the computations is related to the *compute()* method of RDDs. In Apache Spark all the types of RDDs must implement certain methods while some others are optional. More specifically the RDDs are dependent of 5 core properties that are the following:

1. Parent RDDs - A list of dependencies of the RDD i.e. what RDDs are required to be computed for the current RDD to be materialized.
2. An array of partitions that comprise the current RDD
3. A compute function that when applied it computes the partitions of the RDD
4. A partitioner which is a function that describes how the keys of the data items of the RDD should be distributed in the the RDD's partitions (optional)
5. A list of preferred locations (locality metadata) for the partitions of the RDD (optional)

A key operation that is involved in the computation of the partitions of RDDs is listed in point 3 in the list above. The *compute()* method for each type of RDD describes in

detail how each partition for an RDD will be computed. In Spark's execution model the partitions of RDDs can be computed on different nodes, even for partitions are part of the same RDD. Therefore, the event captors in order to capture the location of execution of computations of partitions must intercept the *compute()* method for different types of RDDs. From an implementation point of view an RDD in Apache Spark is represented as an abstract class where the 5 functions mentioned above are not implemented or have a default basic implementation. An abbreviated version of the RDD class can be seen listing 3.14 below where the 5 methods are listed.

Listing 3.14 Apache Spark Scala code for the RDD class with basic set of methods

```
1 abstract class RDD[T: ClassTag](@transient private var _sc: SparkContext, @transient private var deps: Seq[Dependency[_]]) extends
2   Serializable with Logging {
3     def compute(split: Partition, context: TaskContext): Iterator[T]
4     protected def getPartitions: Array[Partition]
5     protected def getDependencies: Seq[Dependency[_]] = deps
6     protected def getPreferredLocations(split: Partition): Seq[String] = Nil
7     @transient val partitioner: Option[Partitioner] = None
8 }
```

Interception Component

All types of RDDs extend the abstract implementation of RDD class to override the default functionality or implement their own functions if it is appropriate for the type of RDD. Typically, all types of RDDs implement their own *compute()* method because otherwise there would not be a need for a new type of RDD. Spark uses the RDD abstraction to run application that span across multiple domains of the Big Data analytics such as batch processing, stream processing and iterative machine learning algorithms and our event captors ought to be able to capture the relevant events to support all types of computations from all the aforementioned domains. To attain that objective we intercept all the *compute()* methods for all the classes that extend the RDD class and therefore represent a separate type of RDD. The interception component can be seen in listing 3.15 below.

Listing 3.15 Interception component of event captor for response time

```
1 public static void premain(String configuration, Instrumentation instrumentation) {
2
3     Set<String> coreRDDs = new HashSet<>(Arrays
4         .asList("org.apache.spark.rdd.BlockRDD",
5             "org.apache.spark.rdd.CartesianRDD",
6             "org.apache.spark.rdd.CheckpointRDD",
7             "org.apache.spark.rdd.CoalescedRDD",
8             "org.apache.spark.rdd.CoGroupedRDD",
```

```

9          "org.apache.spark.rdd.EmptyRDD",
10         "org.apache.spark.rdd.HadoopRDD",
11         "org.apache.spark.rdd.HadoopRDD",
12         "org.apache.spark.rdd.JdbcRDD",
13         "org.apache.spark.rdd.MapPartitionsRDD",
14         "org.apache.spark.rdd.NewHadoopRDD",
15         "org.apache.spark.rdd.NewHadoopMapPartitionsWithSplitRDD",
16         "org.apache.spark.rdd.ParallelCollectionRDD",
17         "org.apache.spark.rdd.PartitionerAwareUnionRDD",
18         "org.apache.spark.rdd.PartitionPruningRDD",
19         "org.apache.spark.rdd.PartitionwiseSampledRDD",
20         "org.apache.spark.rdd.PipedRDD",
21         "org.apache.spark.rdd.MyCoolRDD",
22         "org.apache.spark.rdd.ShuffledRDD",
23         "org.apache.spark.rdd.SubtractedRDD",
24         "org.apache.spark.rdd.UnionRDD",
25         "org.apache.spark.rdd.ZippedPartitionsBaseRDD",
26         "org.apache.spark.rdd.ZippedWithIndexRDD"
27     ));
28
29     Set<String> graphxRDDs = new HashSet<>(Arrays
30         .asList("org.apache.spark.graphx.EdgeRDD",
31             "org.apache.spark.graphx.VertexRDD"
32         ));
33
34     Set<String> mlibRDDs = new HashSet<>(Arrays
35         .asList("org.apache.spark.mllib.rdd.RandomRDD",
36             "org.apache.spark.mllib.rdd.RandomVectorRDD"
37         ));
38
39     Set<String> sqlRDDs = new HashSet<>(Arrays
40         .asList("org.apache.spark.sql.execution.EmptyRDDWithPartitions",
41             "org.apache.spark.sql.execution.ShuffledRowRDD",
42             "org.apache.spark.sql.execution.datasources.FileScanRDD",
43             "org.apache.spark.sql.execution.datasources.jdbc.JDBCRRD",
44             "org.apache.spark.sql.execution.datasources.v2.DataSourceRDD",
45             "org.apache.spark.sql.execution.streaming.continuous.ContinuousDataSourceRDD",
46             "org.apache.spark.sql.execution.streaming.continuous.ContinuousWriteRDD",
47             "org.apache.spark.sql.execution.streaming.state.StateStoreRDD"
48         ));
49
50     Set<String> kafkaRDDs = new HashSet<>(Arrays
51         .asList("org.apache.spark.sql.kafka010.KafkaSourceRDD"
52         ));
53
54     Set<String> streamingKafkaRDDs = new HashSet<>(Arrays
55         .asList("org.apache.spark.streaming.kafka010.KafkaRDD"
56         ));
57
58     Set<String> streamingRDDs = new HashSet<>(Arrays
59         .asList("org.apache.spark.streaming.rdd.MapWithStateRDD"
60         ));
61
62     new AgentBuilder.Default()
63         .type(type -> coreRDDs.contains(type.getName())
64             || graphxRDDs.contains(type.getName())
65             || mlibRDDs.contains(type.getName())
66             || sqlRDDs.contains(type.getName())
67             || kafkaRDDs.contains(type.getName())
68             || streamingKafkaRDDs.contains(type.getName())
69             || streamingRDDs.contains(type.getName()))
70
71         .transform((builder, typeDescription, classLoader, module) -> {
72             return builder

```

```

73         .serialVersionUID(1L)
74         .method(method -> method.getName().equals("compute"))
75         .intercept(
76             MethodDelegation
77                 .withDefaultConfiguration()
78                 .withBinders(Morph.Binder.install(Morpher.class))
79                 .to(new RDDComputeInterceptor(type));
80         })
81         .installOn(instrumentation);

```

Note that in line 63 the names of the classes that are intercepted are the ones contained in the sets that are declared in lines 3, 29, 34, 39, 50, 54 and 58 that are subclasses of the RDD class. In those sets we have included all the RDD implementations included in the Apache Spark's API. This interception strategy will result in the interception of all the *compute()* methods for all types of RDDs and therefore the event captors will successfully emit the right events for any intermediate RDD that will be produced during a Big Data service execution. It is important to underline that this interception strategy is possible because the delegation method which will explain in the next section, is exactly the same for all intercepted *compute()* methods. If the delegation method was different and different types of events should be emitted then we would have to explicitly intercept every type of RDD and associate it with a separate delegation method for every RDD type.

Delegation Component

In listing 3.15 above in line 79 the intercepted *compute()* method for all RDDs is delegated to class called *RDDComputeInterceptor*. This class will contain a *compute()* method that will realize our objective of emitting the IP address of the node that performs the computation while at the same time it will execute the original *compute()* method for every type of RDD. The code snippet of the delegation class is shown in listing 3.16 below.

Listing 3.16 Delegation component of event captor for location of execution of computation

```

1 public class RDDComputeInterceptor {
2
3     private EmitterType type;
4     private final Properties properties;
5
6     public RDDComputeInterceptor(EmitterType type){
7         this.type = type;
8     }
9
10    @RuntimeType
11    public Iterator compute(
12        @Argument(0) Partition partition,
13        @Argument(1) TaskContext context,

```

```

14         @This RDD rdd,
15         @Morph Morpher<Iterator> morpher) throws UnknownHostException, JAXBException, DatatypeConfigurationException {
16
17         emitIp(rdd, partition, type);
18         return morpher.invoke(partition, context);
19     }
20 }

```

In listing 3.16 the variable *type* that is passed as a parameter in line 7, is the emitter type that allows the emission of events to multiple receiver types such as a socket, a queuing system or any other kind of synchronous or asynchronous communication channel. The *emitIp()* method in line 17 is the method that creates and sends an XML representation of the compute event with all the necessary metadata such as the application id, the application name, the partition id, the RDD id and the IP address of the host machine to the monitor in a format that the monitor can interpret it. The actual implementation of the *emitIp()* is omitted for the sake of space. Finally, note that the delegation method in line 18, after it has emitted the compute event to the monitor, it invokes the originally intercepted method which will be the corresponding *compute()* method for the RDD type that has been intercepted.

3.3.3 Monitoring Rules for Data Integrity During Service Execution

The monitoring of data integrity is implemented by means of verifying whether the intermediate data has or has not been modified in-between operations from an external agent. The only entity that can and should be able to modify the data during service execution, is the Big Data processing framework itself i.e. in our case Apache Spark. When a Big Data service gets executed in Spark, due to the fact that RDDs are immutable, multiple intermediate RDDs are produced as a result of the application of the service's operations. If RDDs are small enough to fit in memory they are cached and stored in it. If they do not fit in memory, part of the data is spilled on the disk. The intermediate storage of RDDs creates an attack surface and can expose the data to integrity violations which can be the result of malicious attacks such as RAM scrapping [110]. Also, the integrity of the data can be violated when the data is stored on the disk or when transferred over the network during transformations with wide dependencies. The unwanted modification of data does not necessarily imply an attack from a malicious entity. It can happen under many different circumstances and can be the result of a hardware or software failure as well. If the integrity of any intermediate data is compromised, the integrity of the result of the computation is also compromised. In

general, monitoring data integrity during service execution requires that *"the checksums of the partitions of all intermediate RDDs remain the same in-between all the types of operations, be it transformations or actions"*

To be able to make an assessment with regards to the preservation of the integrity of all the data that is produced as part of the computation, we compute two checksum values; one when a dataset is produced and one when it is consumed from its subsequent operations. As soon as the checksums are produced we compare them. If they are the same the integrity of the data has not been compromised. If they are not the same the implication is that some external entity other than the processing framework has modified the data in an undesirable way.

Since an RDD is the basic abstraction we will compute checksums on a per RDD basis. However, RDDs are comprised of partitions and therefore RDD partitions offer a logical piece of data that we can use to calculate the checksums. A checksum is a sequence of letters and digits that is produced as a result of a hashing algorithm that is applied on piece of data. The data could be a file or a sequence of data items of an in-memory data structure. Regardless of the size of the data, a checksum can be produced and with a high degree of confidence the generation of the same checksum for the data can guarantee that the integrity of the data has not been compromised.

There exist multiple hash function implementations that produce hash codes with different levels of probability for collisions i.e. different level of confidence that for two different datasets the same checksum will be produced. The most prominent implementations of hash functions are *MD5*, *SHA-1* and *SHA-256*. *MD5* produces a 128-bit or 16 byte hash value that in hexadecimal number is comprised of 32 digits. *MD5* is prone to collisions and therefore not the most appropriate solution for large datasets. *SHA-1* stands for Secure Hash Algorithms and has been designed and implemented by the National Security Agency (NSA). It produces larger hash values in size than *MD5* and are 160-bit or 20 byte long. When represented as a hexadecimal number *SHA-1* hash values are 40 digits long. *SHA-256* is a more sophisticated version of *SHA-1*, it has produced by NSA as well and produces hash values that are 256-bit or 28 byte long. In hexadecimal format they are 64 digits long. From all three algorithms the *SHA-256* is the one with the least likelihood for collisions. However,

due to its complexity, *SHA-256* is the hash function that imposes the greater overhead when computing the checksum of a dataset.

A high likelihood for collisions in hash values suggests that the evaluation of checksums would not suffice for the discovery of data integrity violations. If a piece of data is modified so that the checksums for both the original and the modified datasets are the same, it would be impossible to successfully track down that a modification has occurred on the data. Also, the modification of data will have to be such that it would alter it in a meaningful way. We argue that this probability is very low when a sufficiently collision-free algorithm is used. Therefore, examining the checksums of the intermediate data is a reasonable approach with regards to the evaluation of the preservation of data integrity during data processing.

As shown in Spark's architecture in section A.2.2, a Spark job is organized as a directed acyclic graph where the edges are the operations and the nodes are the intermediate data that is produced as a result of the operations applied on the data. In this setup, operations are strung together and the output of an operation becomes the input of the next operation until the final result is computed. Depending on the type of operation, different types of output will be produced. As explained, in Spark the operations available are transformations and actions. Transformations with narrow dependencies are compacted together and are executed in memory to increase performance and computational efficiency. On the other hand, transformation with wide dependencies need to take place in two stages; first the mappers have to prepare the data to be sent to the reducers and then the reducers have to read the mapped data over the network to apply the reduction function. The difference in implementation for the two types of transformations plays a significant role in the monitoring events that need to be captured in order to realize the monitoring activity for data integrity. We will examine each type of transformation separately. Finally, actions behave similarly to transformations with narrow dependencies except for the fact they trigger the materialization of RDDs that are declared prior to the action.

Monitoring transformations with narrow dependencies

In this subsection we provide the theoretical analysis for the monitoring of data integrity for transformations with narrow dependencies. Such transformations are in-memory operations

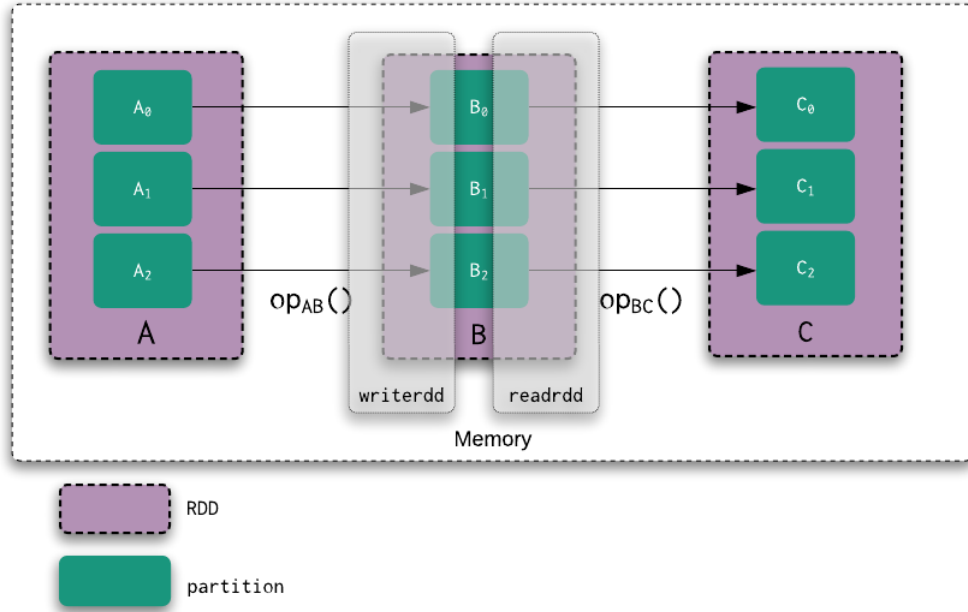


Fig. 3.9 Monitoring data integrity for transformations with narrow dependencies

and therefore violations in data integrity can occur if the heap space of the JVM is compromised. A visual portrayal of how such transformation occur can be seen in figure 3.9.

In figure 3.9 we depict 3 RDDs namely $A = \{A_0, A_1, A_2\}$, $B = \{B_0, B_1, B_2\}$ and $C = \{C_0, C_1, C_2\}$ with their respective partitions. As it can be seen, when $op_{AB}()$ is applied on the partitions of RDD A, the partitions of RDD B is written in memory. This denotes the occurrence of a *writerrdd* event. Subsequently, when $op_{AB}()$ has completed, $op_{BC}()$ is applied on the partitions of RDD B which implies that it takes its partitions as input. These concepts are presented in a formal manner in formulae 3.2 and 3.3.

$$op_{AB}(A_n) = B_n, \quad n = 0, 1, 2 \quad (3.2)$$

$$op_{BC}(B_n) = C_n, \quad n = 0, 1, 2 \quad (3.3)$$

As it can be seen from the formulae, the output of operation $op_{AB}()$, namely partitions $B_n, n = 0, 1, 2$, are provided as input to operation $op_{BC}()$. Having said that, the monitoring activity for data integrity would require the computation of checksums for each individual

<i>writerdd</i> (<i>appId</i> , <i>appName</i> , <i>partId</i> , <i>rddId</i> , <i>checksum</i> , <i>t</i>)
<i>appId</i> : Application id
<i>appName</i> : Application name
<i>partId</i> : Partition id
<i>rddId</i> : RDD id
<i>checksum</i> : The checksum hash value for the partition
<i>t</i> : Timestamp when the writerdd event occurred

Table 3.8 Writerdd events for monitoring data integrity for transformations with narrow dependencies

partition of RDD B so as to juxtapose it with the checksums of the partitions that $op_{BC}()$ will take as an input and check whether the hash values are the same or not. If they are the same, that implies that in-between operations $op_{AB}()$ and $op_{BC}()$ the data of the partitions of RDD B has not been altered by anyone outside the operations themselves. The collection and comparison of checksums for all partitions for all the intermediate RDDs that are the result of the application of transformations with narrow dependencies, is a process that will have to be applied exhaustively for all such transformations to guarantee that no data integrity violations have taken place. The types of events that will have to be captured to materialize the monitoring rule are *writerdd* events for the partitions i.e. when a partition for an RDD is computed and the *readrdd* events when a partition reads data from its parent RDD. Similar to the other monitoring rules, the application id and name will need to be included in the event to distinguish between multiple executions of the same service, the partition id, the RDD id and the actual checksum of the partition at hand. Note for the same partition first there will be a *writerdd* event when the RDD is computed and then, at a later point in time, a *readrdd* event for the same partition will occur when the RDD is fed an input to a subsequent operation. A description of the *writerdd* and *readrdd* events for the monitoring of transformations with narrow dependencies in event calculus notation can be viewed in tables 3.8 and 3.9.

Example of write and read events intended for the monitoring of data integrity during job execution can be seen in tables 3.10 and 3.11

<i>readrdd</i> (<i>appId</i> , <i>appName</i> , <i>partId</i> , <i>rddId</i> , <i>checksum</i> , <i>t</i>)
<i>appId</i> : Unique application id
<i>appName</i> : Unique application name
<i>partId</i> : Unique partition id across all the partitions of an RDD
<i>rddId</i> : Unique RDD id across all RDDs of a Spark job
<i>checksum</i> : The checksum hash value for the data of the partition
<i>t</i> : Timestamp when the read event occurred

Table 3.9 Read events for monitoring data integrity for transformations with narrow dependencies

<i>writerdd</i> (<i>appId</i> =app-20181202162554-0401, <i>appName</i> =LoadAndAnonymize, <i>partId</i> =4, <i>rddId</i> =2, <i>checksum</i> =A199B8D49D5688E2BA14FAB77CC34B14, <i>t</i> =1543865904)
<i>writerdd</i> (<i>appId</i> =app-20181202162554-0401, <i>appName</i> =LoadAndAnonymize, <i>partId</i> =2, <i>rddId</i> =3, <i>checksum</i> =CC21014E074E2BAE7D6ADBBEDF98521C, <i>t</i> =1543867839)

Table 3.10 Example of writerdd events collected for monitoring data integrity of transformations with narrow dependencies

<i>readrdd</i> (<i>appId</i> =app-20181202162554-0401, <i>appName</i> =LoadAndAnonymize, <i>partId</i> =4, <i>rddId</i> =2, <i>checksum</i> =C443B8D49D5688E2BA14FAA10AC34C66, <i>t</i> =1543868674)
<i>readrdd</i> (<i>appId</i> =app-20181202162554-0401, <i>appName</i> =LoadAndAnonymize, <i>partId</i> =2, <i>rddId</i> =3, <i>checksum</i> =A402014B938E2BAE7D6579BEDF985BBC, <i>t</i> =1543869951)

Table 3.11 Example of readrdd events collected for monitoring data integrity of transformations with narrow dependencies

Monitoring transformations with wide dependencies

In this subsection we provide the theoretical analysis for the monitoring of data integrity for transformations with wide dependencies. Such transformations require that the partitions of an RDD will be grouped based on the number of the reducers that will consume it and subsequently will be sent to the reducers over the network. In transformations with wide dependencies, based on its size, the grouped data is serialized in memory or on the

disk and then sent over the wire for the successful completion of the transformation. In principle, transformations with wide dependencies leave more room for malicious attacks compared to in-memory transformations. This is because of two reasons; firstly the data is serialized and persisted on the disk and secondly the data is sent over the network and is therefore susceptible to man-in-the-middle attacks as well. A visual depiction of a transformation with wide dependencies is shown in figure 3.10. Operation $op_{AB}()$ is applied on RDD $A = \{A_0, A_1, A_2\}$ and RDD $B = \{B_0, B_1\}$ is computed. Note that RDD A has 3 partitions while RDD B has only 2. This is because partition dependency in wide dependency transformations is not one-to-one but one-to-many. This means that the partitions of the parent RDD (RDD A) can and in most cases will be condensed into fewer partitions in the child RDD (RDD B).

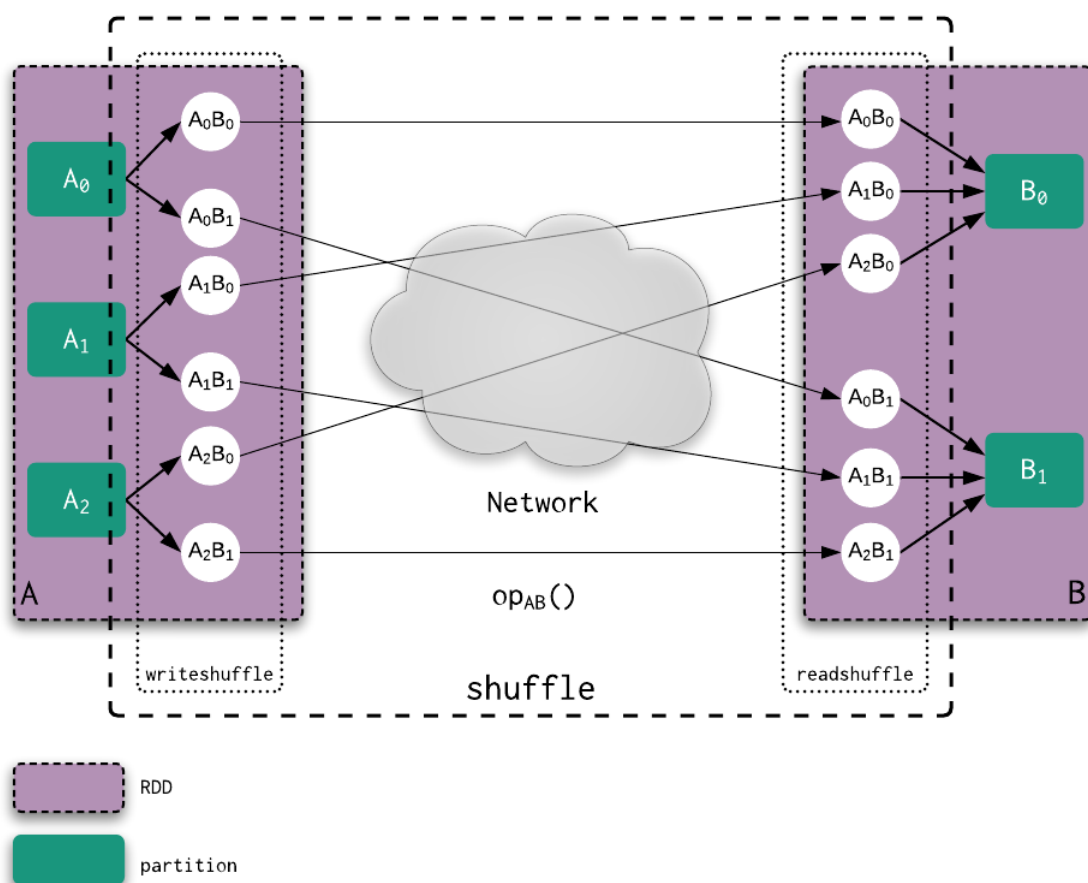


Fig. 3.10 Monitoring data integrity for transformations with wide dependencies

Similar to transformations with narrow dependencies, transformations with wide dependencies need to collect write and read events not for partitions but for the intermediate data sets that are generated during the map/reduce stage of the transformation. All transformations with wide dependencies require data to be shuffled. Every shuffle operation within a computation has a unique id to distinguish it from other shuffle operations during a Spark job. This id will have to be collected in order to differentiate between write and read events that refer to different shuffles within the same Spark job. Now, within a shuffle operation there exist multiple mappers and reducers, each one of them having a unique id for the shuffle operation that they refer to. Therefore, the combination of the shuffle id, the mapper id and the reducer id can only point to a specific intermediate dataset that will be written on the mapper's end and will then be read on the reducer's end. For instance, in figure 3.10, dataset A_1B_0 will be written in the context of a shuffle operation from the mapper with id A_1 for the reducer with id B_0 and the same dataset will be read from the reduced with id B_0 that has been sent from the mapper with id A_1 . The equality of checksums for those two datasets suggests that the integrity of the data has not been breached. If the checksums are not the same, we can be confident that some external agent has modified the data. A description of the events in event calculus notation for monitoring data integrity for transformations with wide dependencies is presented in tables 3.12 and 3.13.

<i>writeshuffle</i> (<i>appId</i> , <i>appName</i> , <i>shuffleId</i> , <i>mapId</i> , <i>reduceId</i> , <i>checksum</i> , <i>t</i>)
<i>appId</i> : Unique application id
<i>appName</i> : Unique application name
<i>shuffleId</i> : Unique shuffle id across all shuffles in a Spark job
<i>mapId</i> : Unique id of the mapper in the shuffle
<i>reduceId</i> : Unique id of the reducer in the shuffle
<i>checksum</i> : The checksum hash value for the data set that is written
<i>t</i> : Timestamp when the writerdd event occurred

Table 3.12 Write shuffle events for monitoring data integrity for transformations with wide dependencies

<i>readshuffle</i> (<i>appId</i> , <i>appName</i> , <i>shuffleId</i> , <i>mapId</i> , <i>reduceId</i> , <i>checksum</i> , <i>t</i>)
<i>appId</i> : Unique application id
<i>appName</i> : Unique application name
<i>shuffleId</i> : Unique shuffle id across all shuffles in a Spark job
<i>mapId</i> : Unique id of the mapper in the shuffle
<i>reduceId</i> : Unique id of the reducer in the shuffle
<i>checksum</i> : The checksum hash value for the data set that is read
<i>t</i> : Timestamp when the <i>writerdd</i> event occurred

Table 3.13 Read shuffle events for monitoring data integrity for transformations with wide dependencies

Monitoring Data Integrity for Actions

The monitoring activity for actions has similar characteristics with the monitoring activity of transformations with narrow dependencies. A key difference however, is that the result of actions are not RDDs but a data structure that is returned on the driver node. Having said that, since a new RDD is not created the collection of *writerdd* events is not appropriate. The only events that will have to be captured are *readrdd* events when actions read the partitions of the RDD that the action is applied on. In event calculus notation *readrdd* for actions are the same as the *readrdd* events for transformations with narrow dependencies illustrated in table 3.9. Also, examples of *readrdd* events for actions are presented in table 3.11.

Event Calculus

The Event Calculus rules and assumptions for the monitoring of data integrity during service execution need to reflect the fact that if the checksums of the intermediate data do not remain the same between operations a data, an integrity violation has occurred. Therefore, the events that will be collected for the evaluation of data integrity will have to correlate the checksums of partitions of RDDs with their respective checksums for the EVEREST monitor to be able to evaluate them. To realise the monitoring activity of runtime data integrity, we need to use different event calculus formulae that will correspond to the events for transformations

with wide dependencies shown in tables 3.12 and 3.13 and transformations with narrow dependencies and actions shown in table 3.8 and 3.9 respectively.

More specifically, the event calculus formulae for monitoring data integrity for transformations with narrow dependencies and actions are listed in table 3.14.

Assumption 1
$\forall t \geq 0,$ Initiates (<i>writerdd</i> (<i>appId</i> , <i>appName</i> , <i>partId</i> , <i>rddId</i> , <i>checksum</i>), writeRddFluent (<i>appId</i> , <i>appName</i> , <i>partId</i> , <i>rddId</i> , <i>checksum</i>), <i>t</i>)
Rule 1
$\forall t \geq 0,$ Happens (<i>readrdd</i> (<i>appId</i> , <i>appName</i> , <i>partId</i> , <i>rddId</i> , <i>checksum</i> , <i>t</i>)) \rightarrow HoldsAt (writeRddFluent (<i>appId</i> , <i>appName</i> , <i>partId</i> , <i>rddId</i> , <i>checksum</i> , <i>t</i>))

Table 3.14 Event calculus assumption and rule for monitoring the runtime data integrity for actions and transformations with **narrow dependencies**

In **Assumption 1** shown in tables 3.14 and 3.15, two fluents are initialised under the names *writeRddFluent* and *writeShuffleFluent* when *writerdd* and *writeshuffle* events occurs respectively. The fluents in the case of transformations with narrow dependencies reflect the fact that a partition for an RDD has been written whereas in the case of transformations with wide dependencies they reflect the fact that the map side of a shuffle operations has written its data. Note that in both cases the checksum of the data is calculated and included in the fluent as a parameter as well. This is necessary for the correct initialisation of the fluent that will allow it later on to compare the checksum value with the checksum of that data when it is being passed as input to a subsequent transformation. In the same tables, in **Rule 1** when a read operation is performed, be it an RDD partition read or a shuffle read, the previously instantiated fluents are evaluated. If the respective fluents are true it means that the data that is being read has previously been written by another operation and in fact the checksum is the same as the one that is being read and therefore the integrity of the data has been respected.

Assumption 1
$\forall t \geq 0,$ Initiate (<i>writeshuffle</i> (<i>appId</i> , <i>appName</i> , <i>shuffleId</i> , <i>mapId</i> , <i>reduceId</i> , <i>checksum</i>), <i>writeShuffleFluent</i> (<i>appId</i> , <i>appName</i> , <i>shuffleId</i> , <i>mapId</i> , <i>reduceId</i> , <i>checksum</i>), <i>t</i>)
Rule 1
$\forall t \geq 0,$ Happens (<i>readshuffle</i> (<i>appId</i> , <i>appName</i> , <i>shuffleId</i> , <i>mapId</i> , <i>reduceId</i> , <i>checksum</i> , <i>t</i>)) \rightarrow HoldsAt (<i>writeShuffleFluent</i> (<i>appId</i> , <i>appName</i> , <i>shuffleId</i> , <i>mapId</i> , <i>reduceId</i> , <i>checksum</i> , <i>t</i>))

Table 3.15 Event calculus assumption and rule for monitoring the runtime data integrity for transformations with **wide dependencies**

A visual representation of *writerdd* and *readrdd* events that occur over time during the monitoring activity of runtime data integrity for transformations with narrow dependencies and actions, can be seen in figure 3.11.

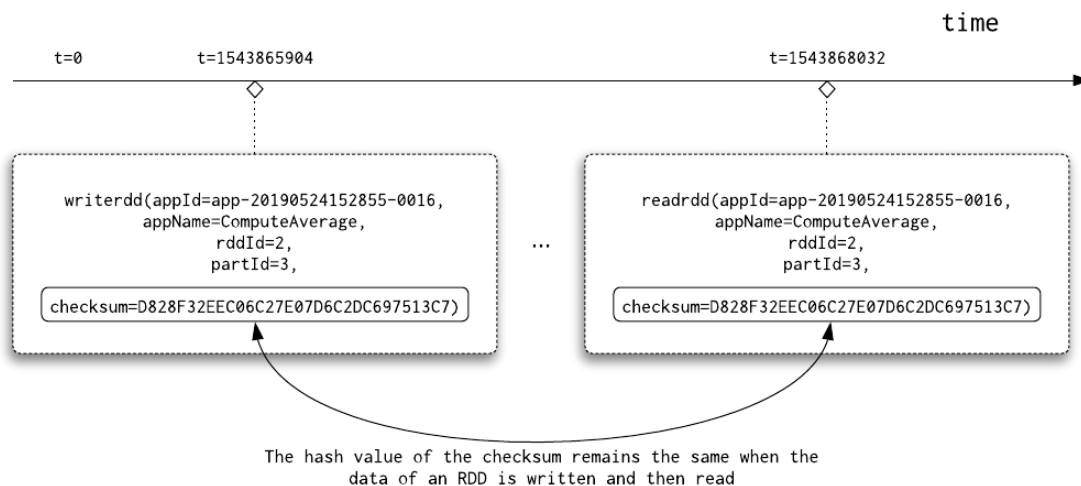


Fig. 3.11 Example of events that occur over time during the monitoring activity of data integrity for actions and transformations with **narrow dependencies**

In this particular example we use only two events to convey the concept. More specifically, at time $t=1543865904$ a *writeRDDFluent* fluent is set to true to represent the fact that for the partition with *partId* = 3 in the RDD with *rddId* = 2, the value of its checksum is

checksum = D828F32EEC06C27E07D6C2DC697513C7. As time goes on and the service keeps executing, a *readrdd* event takes place on behalf of an operation further down in the computation where the exact same partition is being read as an input. In particular, at time $t = 1543868032$ another transformation reads the same partition with *partId* = 3 in RDD with *rddId* = 2 and the corresponding *readrdd* event is captured. From an event calculus perspective to check if the integrity of the data has been preserved, when the *readrdd* event takes place we need to check if the corresponding fluent holds true for the given checksum. Note that in the example presented it does hold true.

By the same token, a visual representation of *writeshuffle* and *readshuffle* events that occur over time during the monitoring activity of runtime data integrity for transformations with wide dependencies, can be seen in figure 3.12.

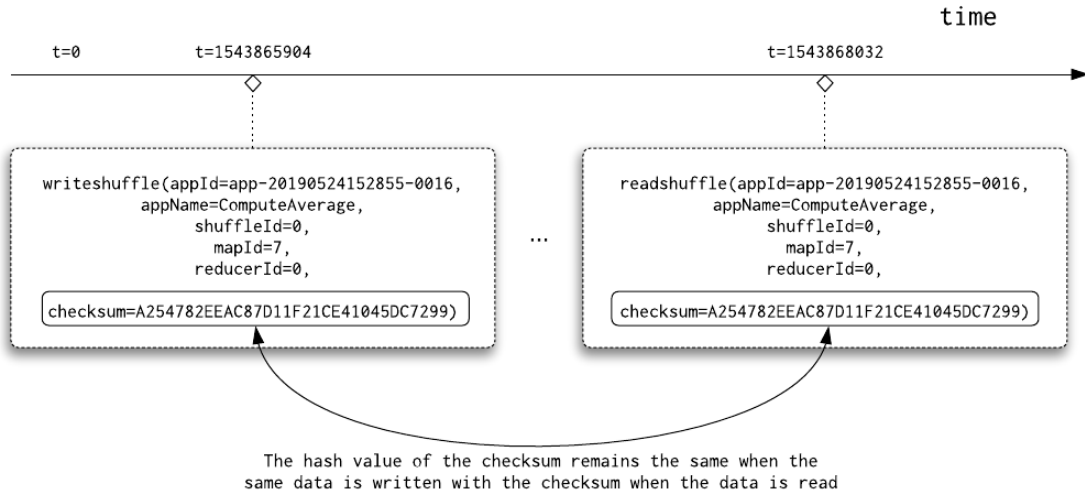


Fig. 3.12 Example of events that occur over time during the monitoring activity of data integrity for transformations with wide dependencies

In this example, in order to illustrate the basic idea we present only two correlated events. At time $t = 1543865904$ a fluent is set to true to represent the fact that for the shuffleId with *shuffleId* = 0, for the mapper with *mapId* = 7 that produces data that will be consumed by the reducer with *reducerId* = 0 the value of the checksum of that data is A254782EEAC87D11F21CE41045DC7299. Later, at time $t = 1543868032$, the grouped data from the mappers will be read within the context of the shuffle with *shuffleId* = 0 from

the reducer with *reduceId* = 0 that has been produced from the mapper with *mapId* = 7. This will enforce the collection of a *readshuffle* event. As it can be seen the checksum of the data that is being read is *checksum* = *A254782EEAC87D11F21CE41045DC7299*, which is exactly the same as the one stored in the fluent that was initiated earlier when the *writeShuffle* event took place.

SLA Template specification

The SLA template specification for the location of execution will produce the EC-Assertion assumption and rule that corresponds to the event calculus assumption and rule presented in tables 3.15 and 3.15. The EC-Assertion expressions will be loaded on the Everest monitor to support the monitoring activity of SLAs where data integrity monitoring is required. An abbreviated version of the SLA template for the location of execution of the service's computations can be seen in the listing 3.17.

Listing 3.17 Abbreviated version of the SLA template for the integrity of runtime data

```

1 <Guaranteed forChecking="false" ID="IntAssump" type="Future_Formula">
2   <quantification>
3     <quantifier>forall</quantifier>
4     <timeVariable>
5       <varName>t1</varName>
6       <varType>TimeVariable</varType>
7     </timeVariable>
8   </quantification>
9   <precondition>
10    <atomicCondition conditionID="asac1">
11      <eventCondition unconstrained="true">
12        <event>
13          <eventID forMatching="true" persistent="false">
14            <varName>Integrity</varName>
15          </eventID>
16        <reply>
17          <interfaceID>BDASLA</interfaceID>
18          <OperationID>1</OperationID>
19          <operationName>writerdd</operationName>
20          <outputVariable forMatching="true" persistent="false">
21            <varName>appId</varName>
22            <varType>string</varType>
23          </outputVariable>
24          <outputVariable forMatching="true" persistent="false">
25            <varName>appName</varName>
26            <varType>string</varType>
27          </outputVariable>
28          <outputVariable forMatching="true" persistent="false">
29            <varName>rddId</varName>
30            <varType>string</varType>
31          </outputVariable>
32          <outputVariable forMatching="true" persistent="false">
33            <varName>partId</varName>
34            <varType>string</varType>

```



```

99         <varType>TimeVariable</varType>
100     </timeVar>
101 </tVar>
102 <fromTime>
103     <time>
104         <varName>t1</varName>
105         <varType>TimeVariable</varType>
106     </time>
107 </fromTime>
108 <toTime>
109     <time>
110         <varName>t1</varName>
111         <varType>TimeVariable</varType>
112     </time>
113 </toTime>
114 </event>
115 <state name="writeRddFluent">
116     <argument>
117         <variable forMatching="true" persistent="false">
118             <varName>appId</varName>
119             <varType>string</varType>
120         </variable>
121     </argument>
122     <argument>
123         <variable forMatching="true" persistent="false">
124             <varName>appName</varName>
125             <varType>string</varType>
126         </variable>
127     </argument>
128     <argument>
129         <variable forMatching="true" persistent="false">
130             <varName>rddId</varName>
131             <varType>string</varType>
132         </variable>
133     </argument>
134     <argument>
135         <variable forMatching="true" persistent="false">
136             <varName>partId</varName>
137             <varType>string</varType>
138         </variable>
139     </argument>
140     <argument>
141         <variable forMatching="true" persistent="false">
142             <varName>checksum</varName>
143             <varType>string</varType>
144         </variable>
145     </argument>
146 </state>
147 <timeVar>
148     <varName>t1</varName>
149     <varType>TimeVariable</varType>
150 </timeVar>
151 </initiates>
152 </stateCondition>
153 </atomicCondition>
154 </postcondition>
155 </Guaranteed>
156 <Guaranteed forChecking="false" ID="ShuffleAssump" type="Future_Formula">
157     <quantification>
158         <quantifier>forall</quantifier>
159         <timeVariable>
160             <varName>t2</varName>
161             <varType>TimeVariable</varType>
162         </timeVariable>

```

```

163     </quantification>
164     <precondition>
165         <atomicCondition conditionID="asac1">
166             <eventCondition unconstrained="true">
167                 <event>
168                     <eventID forMatching="true" persistent="false">
169                         <varName>Integrity</varName>
170                     </eventID>
171                     <reply>
172                         <interfaceId>BDASLA</interfaceId>
173                         <OperationId>1</OperationId>
174                         <operationName>writeshuffle</operationName>
175                         <outputVariable forMatching="true" persistent="false">
176                             <varName>status1</varName>
177                             <varType>OpStatus</varType>
178                             <value>RES-B</value>
179                         </outputVariable>
180                         <outputVariable forMatching="true" persistent="false">
181                             <varName>sender1</varName>
182                             <varType>Entity</varType>
183                             <value></value>
184                         </outputVariable>
185                         <outputVariable forMatching="true" persistent="false">
186                             <varName>receiver1</varName>
187                             <varType>Entity</varType>
188                             <value></value>
189                         </outputVariable>
190                         <outputVariable forMatching="true" persistent="false">
191                             <varName>source1</varName>
192                             <varType>Entity</varType>
193                             <value></value>
194                         </outputVariable>
195                         <outputVariable forMatching="true" persistent="false">
196                             <varName>serviceId</varName>
197                             <varType>string</varType>
198                             <value></value>
199                         </outputVariable>
200                         <outputVariable forMatching="true" persistent="false">
201                             <varName>appId</varName>
202                             <varType>string</varType>
203                         </outputVariable>
204                         <outputVariable forMatching="true" persistent="false">
205                             <varName>appName</varName>
206                             <varType>string</varType>
207                         </outputVariable>
208                         <outputVariable forMatching="true" persistent="false">
209                             <varName>shuffleId</varName>
210                             <varType>string</varType>
211                         </outputVariable>
212                         <outputVariable forMatching="true" persistent="false">
213                             <varName>mapId</varName>
214                             <varType>string</varType>
215                         </outputVariable>
216                         <outputVariable forMatching="true" persistent="false">
217                             <varName>reduceId</varName>
218                             <varType>string</varType>
219                         </outputVariable>
220                         <outputVariable forMatching="true" persistent="false">
221                             <varName>checksum</varName>
222                             <varType>string</varType>
223                         </outputVariable>
224                     </reply>
225                 </tVar>
226                 <timeVar>

```

```

227         <varName>t2</varName>
228         <varType>TimeVariable</varType>
229     </timeVar>
230 </tVar>
231 <fromTime>
232     <time>
233         <varName>t2</varName>
234         <varType>TimeVariable</varType>
235     </time>
236 </fromTime>
237 <toTime>
238     <time>
239         <varName>t2</varName>
240         <varType>TimeVariable</varType>
241     </time>
242 </toTime>
243 </event>
244 </eventCondition>
245 </atomicCondition>
246 </precondition>
247 <postcondition>
248     <atomicCondition conditionID="asac2">
249         <stateCondition>
250             <initiates>
251                 <event>
252                     <eventID forMatching="true" persistent="false">
253                         <varName>Integrity</varName>
254                     </eventID>
255                     <reply>
256                         <interfaceId>BASLA</interfaceId>
257                         <OperationId>1</OperationId> <operationName>writeshuffle</operationName>
258                         <outputVariable forMatching="true" persistent="false">
259                             <varName>status1</varName>
260                             <varType>OpStatus</varType>
261                             <value>RES-B</value>
262                         </outputVariable>
263                         <outputVariable forMatching="true" persistent="false">
264                             <varName>sender1</varName>
265                             <varType>Entity</varType>
266                             <value></value>
267                         </outputVariable>
268                         <outputVariable forMatching="true" persistent="false">
269                             <varName>receiver1</varName>
270                             <varType>Entity</varType>
271                             <value></value>
272                         </outputVariable>
273                         <outputVariable forMatching="true" persistent="false">
274                             <varName>source1</varName>
275                             <varType>Entity</varType>
276                             <value></value>
277                         </outputVariable>
278                         <outputVariable forMatching="true" persistent="false">
279                             <varName>serviceId</varName>
280                             <varType>string</varType>
281                             <value></value>
282                         </outputVariable>
283                         <outputVariable forMatching="true" persistent="false">
284                             <varName>appId</varName>
285                             <varType>string</varType>
286                         </outputVariable>
287                         <outputVariable forMatching="true" persistent="false">
288                             <varName>appName</varName>
289                             <varType>string</varType>
290                         </outputVariable>

```



```

291         <outputVariable forMatching="true" persistent="false">
292             <varName>shuffleId</varName>
293             <varType>string</varType>
294         </outputVariable>
295         <outputVariable forMatching="true" persistent="false">
296             <varName>mapId</varName>
297             <varType>string</varType>
298         </outputVariable>
299         <outputVariable forMatching="true" persistent="false">
300             <varName>reduceId</varName>
301             <varType>string</varType>
302         </outputVariable>
303         <outputVariable forMatching="true" persistent="false">
304             <varName>checksum</varName>
305             <varType>string</varType>
306         </outputVariable>
307     </reply>
308     <tVar>
309         <timeVar>
310             <varName>t2</varName>
311             <varType>TimeVariable</varType>
312         </timeVar>
313     </tVar>
314     <fromTime>
315         <time>
316             <varName>t2</varName>
317             <varType>TimeVariable</varType>
318         </time>
319     </fromTime>
320     <toTime>
321         <time>
322             <varName>t2</varName>
323             <varType>TimeVariable</varType>
324         </time>
325     </toTime>
326 </event>
327 <state name="writeShuffleFluent">
328     <argument>
329         <variable forMatching="true" persistent="false">
330             <varName>appId</varName>
331             <varType>string</varType>
332         </variable>
333     </argument>
334     <argument>
335         <variable forMatching="true" persistent="false">
336             <varName>appName</varName>
337             <varType>string</varType>
338         </variable>
339     </argument>
340     <argument>
341         <variable forMatching="true" persistent="false">
342             <varName>shuffleId</varName>
343             <varType>string</varType>
344         </variable>
345     </argument>
346     <argument>
347         <variable forMatching="true" persistent="false">
348             <varName>mapId</varName>
349             <varType>string</varType>
350         </variable>
351     </argument>
352     <argument>
353         <variable forMatching="true" persistent="false">
354             <varName>reduceId</varName>

```

```

355         <varType>string</varType>
356     </variable>
357 </argument>
358 <argument>
359     <variable forMatching="true" persistent="false">
360         <varName>checksum</varName>
361         <varType>string</varType>
362     </variable>
363 </argument>
364 </state>
365 <timeVar>
366     <varName>t2</varName>
367     <varType>TimeVariable</varType>
368 </timeVar>
369 </initiates>
370 </stateCondition>
371 </atomicCondition>
372 </postcondition>
373 </Guaranteed>
374 <Guaranteed forChecking="true" ID="IntMonRule" type="Future_Formula">
375     <quantification>
376         <quantifier>forall</quantifier>
377         <timeVariable>
378             <varName>t1</varName>
379             <varType>TimeVariable</varType>
380         </timeVariable>
381     </quantification>
382     <precondition>
383         <atomicCondition conditionID="mrac1">
384             <eventCondition unconstrained="true">
385                 <event>
386                     <eventID forMatching="true" persistent="false">
387                         <varName>Integrity</varName>
388                     </eventID>
389                     <call>
390                         <interfaceId>BDASLA</interfaceId>
391                         <OperationId>1</OperationId>
392                         <operationName>readrdd</operationName>
393                         <inputVariable forMatching="true" persistent="false">
394                             <varName>status1</varName>
395                             <varType>OpStatus</varType>
396                             <value>REQ-B</value>
397                         </inputVariable>
398                         <inputVariable forMatching="true" persistent="false">
399                             <varName>sender1</varName>
400                             <varType>Entity</varType>
401                             <value></value>
402                         </inputVariable>
403                         <inputVariable forMatching="true" persistent="false">
404                             <varName>receiver1</varName>
405                             <varType>Entity</varType>
406                             <value></value>
407                         </inputVariable>
408                         <inputVariable forMatching="true" persistent="false">
409                             <varName>source1</varName>
410                             <varType>Entity</varType>
411                             <value></value>
412                         </inputVariable>
413                         <inputVariable forMatching="true" persistent="false">
414                             <varName>serviceId</varName>
415                             <varType>string</varType>
416                             <value></value>
417                         </inputVariable>
418                         <inputVariable forMatching="true" persistent="false">

```

```

419         <varName>appId</varName>
420         <varType>string</varType>
421     </inputVariable>
422     <inputVariable forMatching="true" persistent="false">
423         <varName>appName</varName>
424         <varType>string</varType>
425     </inputVariable>
426     <inputVariable forMatching="true" persistent="false">
427         <varName>rddId</varName>
428         <varType>string</varType>
429     </inputVariable>
430     <inputVariable forMatching="true" persistent="false">
431         <varName>partId</varName>
432         <varType>string</varType>
433     </inputVariable>
434     <inputVariable forMatching="true" persistent="false">
435         <varName>checksum</varName>
436         <varType>string</varType>
437     </inputVariable>
438 </call>
439 <tVar>
440     <timeVar>
441         <varName>t1</varName>
442         <varType>TimeVariable</varType>
443     </timeVar>
444 </tVar>
445 <fromTime>
446     <time>
447         <varName>t1</varName>
448         <varType>TimeVariable</varType>
449     </time>
450 </fromTime>
451 <toTime>
452     <time>
453         <varName>t1</varName>
454         <varType>TimeVariable</varType>
455     </time>
456 </toTime>
457 </event>
458 </eventCondition>
459 </atomicCondition>
460 </precondition>
461 <postcondition>
462     <atomicCondition conditionID="mrpc1">
463         <stateCondition>
464             <holdsAt>
465                 <state name="writeRddFluent">
466                     <argument>
467                         <variable forMatching="true" persistent="false">
468                             <varName>appId</varName>
469                             <varType>string</varType>
470                         </variable>
471                     </argument>
472                     <argument>
473                         <variable forMatching="true" persistent="false">
474                             <varName>appName</varName>
475                             <varType>string</varType>
476                         </variable>
477                     </argument>
478                     <argument>
479                         <variable forMatching="true" persistent="false">
480                             <varName>rddId</varName>
481                             <varType>string</varType>
482                         </variable>

```

```

483         </argument>
484     <argument>
485         <variable forMatching="true" persistent="false">
486             <varName>partId</varName>
487             <varType>string</varType>
488         </variable>
489     </argument>
490 <argument>
491     <variable forMatching="true" persistent="false">
492         <varName>checksum</varName>
493         <varType>string</varType>
494     </variable>
495 </argument>
496 </state>
497 <timeVar>
498     <varName>t1</varName>
499     <varType>TimeVariable</varType>
500 </timeVar>
501 </holdsAt>
502 </stateCondition>
503 </atomicCondition>
504 </postcondition>
505 </Guaranteed>
506 <Guaranteed forChecking="true" ID="shufMonRule" type="Future_Formula">
507     <quantification>
508         <quantifier>forall</quantifier>
509         <timeVariable>
510             <varName>t2</varName>
511             <varType>TimeVariable</varType>
512         </timeVariable>
513     </quantification>
514     <precondition>
515         <atomicCondition conditionID="mrac1">
516             <eventCondition unconstrained="true">
517                 <event>
518                     <eventID forMatching="true" persistent="false">
519                         <varName>Integrity</varName>
520                     </eventID>
521                 </call>
522                     <interfaceId>BDASLA</interfaceId>
523                     <OperationId>1</OperationId>
524                     <operationName>readshuffle</operationName>
525                     <inputVariable forMatching="true" persistent="false">
526                         <varName>status1</varName>
527                         <varType>OpStatus</varType>
528                         <value>REQ-B</value>
529                     </inputVariable>
530                     <inputVariable forMatching="true" persistent="false">
531                         <varName>sender1</varName>
532                         <varType>Entity</varType>
533                         <value></value>
534                     </inputVariable>
535                     <inputVariable forMatching="true" persistent="false">
536                         <varName>receiver1</varName>
537                         <varType>Entity</varType>
538                         <value></value>
539                     </inputVariable>
540                     <inputVariable forMatching="true" persistent="false">
541                         <varName>source1</varName>
542                         <varType>Entity</varType>
543                         <value></value>
544                     </inputVariable>
545                     <inputVariable forMatching="true" persistent="false">
546                         <varName>serviceId</varName>

```

```

547         <varType>string</varType>
548         <value></value>
549     </inputVariable>
550     <inputVariable forMatching="true" persistent="false">
551         <varName>appId</varName>
552         <varType>string</varType>
553     </inputVariable>
554     <inputVariable forMatching="true" persistent="false">
555         <varName>appName</varName>
556         <varType>string</varType>
557     </inputVariable>
558     <inputVariable forMatching="true" persistent="false">
559         <varName>shuffleId</varName>
560         <varType>string</varType>
561     </inputVariable>
562     <inputVariable forMatching="true" persistent="false">
563         <varName>mapId</varName>
564         <varType>string</varType>
565     </inputVariable>
566     <inputVariable forMatching="true" persistent="false">
567         <varName>reduceId</varName>
568         <varType>string</varType>
569     </inputVariable>
570     <inputVariable forMatching="true" persistent="false">
571         <varName>checksum</varName>
572         <varType>string</varType>
573     </inputVariable>
574 </call>
575 <tVar>
576     <timeVar>
577         <varName>t2</varName>
578         <varType>TimeVariable</varType>
579     </timeVar>
580 </tVar>
581 <fromTime>
582     <time>
583         <varName>t2</varName>
584         <varType>TimeVariable</varType>
585     </time>
586 </fromTime>
587 <toTime>
588     <time>
589         <varName>t2</varName>
590         <varType>TimeVariable</varType>
591     </time>
592 </toTime>
593 </event>
594 </eventCondition>
595 </atomicCondition>
596 </precondition>
597 <postcondition>
598     <atomicCondition conditionID="mrpc1">
599         <stateCondition>
600             <holdsAt>
601                 <state name="writeShuffleFluent">
602                     <argument>
603                         <variable forMatching="true" persistent="false">
604                             <varName>appId</varName>
605                             <varType>string</varType>
606                         </variable>
607                     </argument>
608                     <argument>
609                         <variable forMatching="true" persistent="false">
610                             <varName>appName</varName>

```

```

611         <varType>string</varType>
612     </variable>
613 </argument>
614 <argument>
615     <variable forMatching="true" persistent="false">
616         <varName>shuffleId</varName>
617         <varType>string</varType>
618     </variable>
619 </argument>
620 <argument>
621     <variable forMatching="true" persistent="false">
622         <varName>mapId</varName>
623         <varType>string</varType>
624     </variable>
625 </argument>
626 <argument>
627     <variable forMatching="true" persistent="false">
628         <varName>reduceId</varName>
629         <varType>string</varType>
630     </variable>
631 </argument>
632 <argument>
633     <variable forMatching="true" persistent="false">
634         <varName>checksum</varName>
635         <varType>string</varType>
636     </variable>
637 </argument>
638 </state>
639 <timeVar>
640     <varName>t2</varName>
641     <varType>TimeVariable</varType>
642 </timeVar>
643 </holdsAt>
644 </stateCondition>
645 </atomicCondition>
646 </postcondition>
647 </Guaranteed>

```

Let's examine each segment of the SLA template and analyse it in greater detail. The section presented in listing 3.17 contains four guarantee terms and more specifically two assumptions and two rules. The first two assumptions refer to the assumption formulae presented in tables 3.14 and 3.15 while the third and the fourth rule refer to the rule formulae illustrated in the same tables respectively. In particular, from line 1 until line 155 the initialisation of fluents under the name *writeRddFluent* are triggered by the occurrence of *writerdd* events when partitions of RDDs are computed. Similarly, from line 156 until line 373 the initialization of a fluents under the name *writeShuffleFluent* are triggered by the occurrence of *writeshuffle* events when intermediate datasets are produced during the map phase of a shuffle. Finally, the rules shown in tables 3.14 and 3.15 are declared in the SLA template from line 374 until line 505 and from line 506 until line 647 respectively. As it can be seen in the declaration of both rules, the precondition is the occurrence of

readrdd and *readshuffle* events that triggers the evaluation of the corresponding fluents namely *writeRddFluent* and *writeShuffleFluent*.

Event Captor Specification

The event captor for data integrity is the most complex event captor that we implemented in comparison to the other ones. This is because the monitoring of data integrity does not need to just collect metadata about the execution context of the transformations of a Spark job, such as how long it takes until the job completes or the IP address that partitions of RDDs are computed on. Instead, it requires the computation hash value on the actual data that is being processed. At this point, we regard that there are two important questions that we need to address. The first one *why burden the event captor and not the monitor with the responsibility of producing the checksums for the intermediate data produced during service execution?* The reason we took that decision is because, since the event captor instruments the actual code that operate on the data, it has immediate access on it and therefore no other interaction is required with any other external system in order to compute the hash values. If this was not the case and the hash values were to be computed from the monitor, all the data would have to be sent to monitor for it to be able to produce the checksums and then evaluate them. This would impose an unreasonable overhead on the network resources and the monitor itself. Also this would require that the monitor is scalable and can scale up to handle cases where the data increases in size. The second questions that we need to address is *why the level of granularity with which we produce checksums is per partition?* The argument for that choice is because partitions are a logical grouping of data within an RDD that are large enough to produce meaningful results but small enough to confine the discovery of violation within a narrow subset of the whole RDD. The other options would be to compute checksums per RDD i.e. higher degree of granularity or per tuple i.e. lower degree of granularity. In the first option, apart from the fact that we would not have any reference with regards to on what subset of the data did the violation happen, from an implementation stand point we would have to kill the parallelization of partition computation from Spark. That is because the event captor would have to go through all the tuples for all partitions of an RDD to produce the relevant checksum. Spark launches as many tasks as there are partitions in an RDD and that dictates the level of parallelism for

that transformation. If checksum were to be calculated per RDD all partitions would have to be processed sequentially in order to produce the correct checksum. In the case of the per tuple option, we are faced with the same problem of the over-utilization of the network resources for the transmission of checksums of every tuple. This would place an excessive computational load on the monitor that will have to be able to scale up as well to cope with the ever increasing event emission as the data grows. In conclusion, we make the case that partitions present us with a logical abstraction that is appropriate for the computation of checksums that are also in complete alignment with Spark's execution model. We argue that they offer a reasonable trade-off between event capturing overhead and a degree of granularity of the location of violation if data integrity is compromised.

A typical Spark job follows the ETL (Extract Transform Load) paradigm i.e. data is extracted from one or more sources, then one or more transformations are applied on the data to manipulate it in a meaningful way and eventually it is loaded on a persistence layer such as a database or a filesystem. In our implementation, in order to satisfy this requirement and to facilitate the monitoring activity of such jobs, the event captor for monitoring data integrity at runtime, it intercepts the computation of 3 types of RDDs, namely *HadoopRDD*⁴, *MapPartitionsRDD*⁵ and *ShuffledRDD*⁶. These 3 types of RDDs allows us to examine a plethora of ETL applications. This a pattern that is very common in batch processing. As mentioned, for all 3 types of RDDs we need to intercept the *compute()* method to enable the collection of checksums for each partition. All the types of RDDs extend the base RDD class and each one of them provides an implementation for the abstract *compute()* method that is declared in the RDD class. *HadoopRDDs* and *MapPartitionsRDDs* are RDD types that are computed as a result of the application of transformations with narrow dependencies such as *map()*⁷ and *filter()*⁸, whereas *ShuffledRDDs* are RDD types that are computed as a result

⁴<https://github.com/apache/spark/blob/2153b316bda119ede8c80ceda522027a6581031b/core/src/main/scala/org/apache/spark/rdd/HadoopRDD.scala>

⁵<https://github.com/apache/spark/blob/2153b316bda119ede8c80ceda522027a6581031b/core/src/main/scala/org/apache/spark/rdd/MapPartitionsRDD.scala>

⁶<https://github.com/apache/spark/blob/2153b316bda119ede8c80ceda522027a6581031b/core/src/main/scala/org/apache/spark/rdd/ShuffledRDD.scala>

⁷<https://github.com/apache/spark/blob/2153b316bda119ede8c80ceda522027a6581031b/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L373>

⁸<https://github.com/apache/spark/blob/2153b316bda119ede8c80ceda522027a6581031b/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L390>

of the application of transformations with wide dependencies such as *groupByKey()*⁹. The prototype of the abstract method that all RDDs need to implement is shown in listing 3.18.

Listing 3.18 Abstract method *compute()*¹⁰ in class RDD

```

1  /**
2   * :: DeveloperApi ::
3   * Implemented by subclasses to compute a given partition.
4   */
5   @DeveloperApi
6   def compute(split: Partition, context: TaskContext): Iterator[T]
```

Interception Component

As mentioned in the section above, our data integrity event captor will have to produce the right events for transformations with narrow dependencies, transformations with wide dependencies and actions. Now conceptually, transformations with narrow dependencies and actions are the same because no data shuffling is required. In those case, computing checksums on the partitions of the computed RDDs and sending the relevant events to the monitor would suffice. Conversely, in the case of transformations with wide dependencies checksums should be calculated on the grouped data on the map side of a data shuffle and on the mapped data that is read on the reduce side. We will examine those two cases separately.

As it can be seen in listing 3.18, the *compute()* method has a specific interface that all RDDs need to comply with. It takes as an input an iterator of data items and after applying a function on the data gives as an output an iteration on the data after the application of the function. This takes place in-memory and as explained, Apache Spark pipelines all those applications within a single stage to take advantage of parallel execution of the functions. Since we need to produce checksums when data is read i.e. *readrdd* events and when it is written i.e. *writerdd* events, we ought to process the data and then iterate it again to produce the checksums for each partition. This approach requires that the data is iterated twice; one for the actual processing and one for the generation of the checksums. From a performance standpoint, this is a poor solution that makes the computation twice as slow compared to running without the event captors activated. To address this challenge

⁹<https://github.com/apache/spark/blob/2153b316bda119ede8c80ceda522027a6581031b/core/src/main/scala/org/apache/spark/rdd/PairRDDFunctions.scala#L641>

¹⁰<https://github.com/apache/spark/blob/2153b316bda119ede8c80ceda522027a6581031b/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L120>

we have implemented a custom iterator that takes care of the calculation of checksums during the actual data processing without the need for a second iteration. The way we achieve this is by implementing the *AbstractIterator*¹¹ interface with a concrete class named *DataIntegrityMonitorableIterator* and overriding the abstract methods *hasNext()*¹² and *next()*¹³.

The code for the *DataIntegrityMonitorableIterator* class that is declared in the event captor is presented in listing 3.19.

Listing 3.19 Source code for *DataIntegrityMonitorableIterator* class

```

1 public class DataIntegrityMonitorableIterator<A> extends AbstractIterator<A> {
2
3     private Iterator<A> delegate;
4     private MessageDigest md;
5     private OperationType operation;
6     private Map<String, String> parameters;
7     private Properties properties;
8     private Emitter emitter;
9
10    public DataIntegrityMonitorableIterator(
11        Iterator<A> delegate,
12        EmitterType emitter,
13        Properties properties,
14        OperationType operation,
15        Map<String, String> parameters) {
16
17        this.delegate = delegate;
18        this.operation = operation;
19        this.parameters = parameters;
20        this.properties = properties;
21
22
23        try {
24            md = MessageDigest.getInstance(properties.getProperty("algorithm"));
25        } catch (NoSuchAlgorithmException nsae) {
26            logger.error(nsae);
27        }
28
29        this.emitter = EventEmitterFactory.getInstance(emitter, properties);
30        this.emitter.connect();
31    }
32
33    @Override
34    public boolean hasNext() {
35        if (!delegate.hasNext()) {
36            long operationId = MonitorUtilities.generateRandomLong();
37            parameters.put("checksum", DatatypeConverter.printHexBinary(md.digest()));
38            emitter.send(MonitorUtilities.createEvent(operationId, properties.getProperty("eventStyle"), operation, parameters));
39            emitter.close();
40            return false;
41        } else {
42            return true;
43        }
44    }
45

```

¹¹<https://www.scala-lang.org/api/2.12.3/scala/collection/AbstractIterator.html>

¹²<https://www.scala-lang.org/api/2.12.3/scala/collection/Iterator.html#hasNext:Boolean>

¹³[https://www.scala-lang.org/api/2.12.3/scala/collection/Iterator.html#next\(\):A](https://www.scala-lang.org/api/2.12.3/scala/collection/Iterator.html#next():A)

```
44     }
45
46     @Override
47     public A next() {
48         A item = delegate.next();
49         md.update(item.toString().getBytes());
50         return item;
51     }
52 }
```

Let us examine how the custom iterator that we have introduced actually operates. The custom iterator is a wrapper around the default iterator that Apache Spark uses and that is returned from the *compute()* method of RDDs. The default iterator is passed as an argument to the constructor of our custom iterator under the name *delegate* as it can be seen in line 11. In addition, in the constructor we pass as arguments in line 18 the *operation* which is the name of the event, in line 19 the parameters which the map with key/value pairs that represent the parameters of the event and finally in line 20 a list of properties that allows us to pass a set of different parameters as we see fit for every type of event. Also note in line 24 a special variable under the name *md* is initialized of type *MessageDigest*¹⁴. Message digests are secure one-way hash functions that take arbitrary-sized data and output a fixed-length hash value. In line 49 every time a data item is visited from the iterator the *md* variable is updated with the current value and the checksum is changed accordingly. Finally, in line 38, the initialized emitter is used to send to the monitor the calculated checksum. Note that the emitter sends the event only when the *hasNext()* returns false i.e. the iterator has been able to go through all the data items.

The *DataIntegrityMonitorableIterator* iterator acts as a basic interface for the emission of events for all transformations with narrow dependencies and actions alike. The dynamic definition of operations, parameters and properties enables the event captor to use it for emitting events both during the computation of *HadoopRDDs*, *MapPartitionsRDDs* and actions. A more thorough analysis on how each one of the uses the our custom monitoring iterator can be found in section 3.3.3 below.

Both *HadoopRDDs* and *MapPartitionsRDDs* are produced from the application of transformations with narrow dependencies. *HadoopRDDs* are RDDs that are used to load data from an HDFS filesystem and provides a set of convenient functions to pass Hadoop-related configuration properties when loading the data. Similarly, *MapPartitionsRDDs* are the RDDs

¹⁴<https://docs.oracle.com/javase/8/docs/api/java/security/MessageDigest.html>

that are used to transform the data by means of applying transformation function on every data item of an RDD. In both cases our event captor will intercept the Apache Spark source code that computes the two types of RDD. Finally, with regards to actions, the event captor will intercept the execution of the *runJob()*¹⁵ method declared in the SparkContext class.

The relevant code snippet for intercepting the computations of the *HadoopRDDs* with the assistance of Byte Buddy, is presented in listing 3.20.

Listing 3.20 Interception component of event captor for data integrity of HadoopRDDs

```

1 new AgentBuilder.Default()
2   .type(type -> type.getName().equals("org.apache.spark.rdd.HadoopRDD"))
3   .transform((builder, typeDescription, classLoader, module) -> {
4       return builder
5         .serialVersionUID(1L)
6         .method(method -> method.getName().equals("compute"))
7         .intercept(MethodDelegation
8           .withDefaultConfiguration()
9           .withBinders(Morph.Binder.install(Morpher.class))
10          .to(new HadoopRDDComputeDelegator(type, properties)));
11   }).installOn(instrumentation);

```

Note that in line 10 an interceptor is assigned with the execution of the *compute()* method for *HadoopRDDs*. This is presented in greater detail in section 3.3.3 below. Also, visual representation of interception of the *compute()* method of *HadoopRDD* is depicted in figure 3.13

On the left side of figure 3.13 the data is stored on a Hadoop filesystem and is broken down into multiple input splits. The *compute()* method will read that data and it will produce its result which will be preserved in memory as a HadoopRDD. When each *compute()* method has completed its tasks it will emit to the monitor a *writerdd* event. Note, that when the *compute()* method reads its input from the Hadoop filesystem a *readrdd* event is not captured. This is because, according to the event calculus formulae shown in 3.14, a *readrdd* event will not be unified with a *writerdd* since when the data was written in the Hadoop filesystem a checksum was not calculated. This is in alignment with our original commitment for monitoring data integrity during service execution. When data is stored on HDFS we regard it as being static and at rest. Therefore the event captor does not need to take it into consideration when monitoring the integrity of data at runtime.

¹⁵<https://github.com/apache/spark/blob/32461d474460044669ee938e61fda1aabb70ec2/core/src/main/scala/org/apache/spark/SparkContext.scala#L2046>

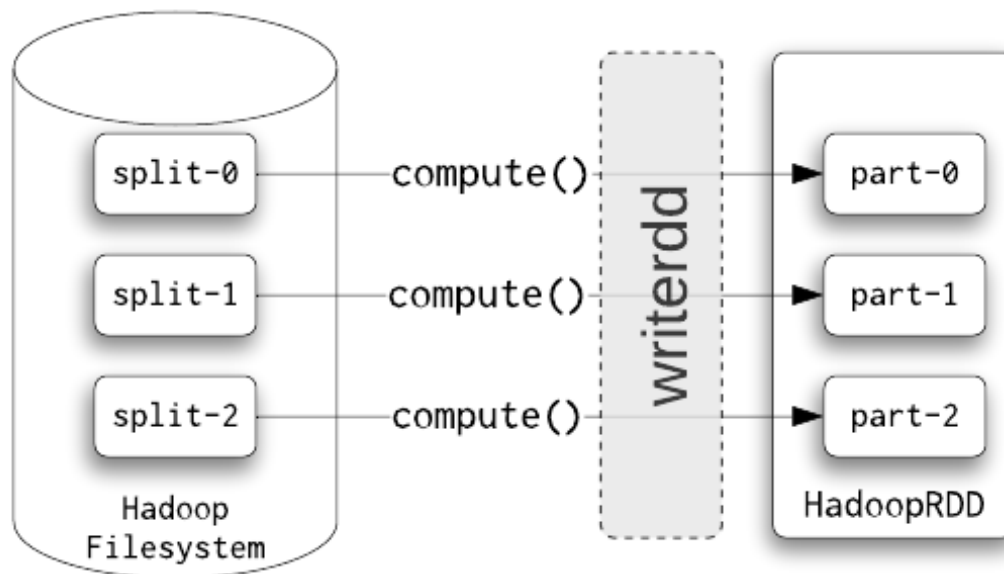


Fig. 3.13 Interception component for HadoopRDD

The relevant code snippet for intercepting the computations of the *MapPartitionsRDDs* with the assistance of Byte Buddy, is presented in listing 3.20.

Listing 3.21 Interception component of event captor for data integrity of MapPartitionsRDDs

```

1 new AgentBuilder.Default()
2   .type(type -> type.getName().equals("org.apache.spark.rdd.MapPartitionsRDD"))
3   .transform((builder, typeDescription, classLoader, module) -> {
4     return builder
5       .serialVersionUID(1L)
6       .method(method -> method.getName().equals("compute"))
7       .intercept(to(new MapPartitionsRDDComputeInterceptor(type, properties)));
8   }).installOn(instrumentation);

```

Note that in line 7 an interceptor is assigned with the execution of the *compute()* method for *MapPartitionsRDDs*. This is presented in greater detail in section 3.3.3 below. Also, a visual representation of interception of the *compute()* method of *MapPartitionsRDD* is depicted in figure 3.14. The data is read by the *compute()* method and a *readrdd* event is emitted whereas when the *compute()* method is completed and its outcome is produced a *writerdd* event is send to the monitor.

Now that we have provided the interceptors for transformations with narrow dependencies that will perform the computation of *HadoopRDDs* and *MapartitionsRDDs*, we will discuss

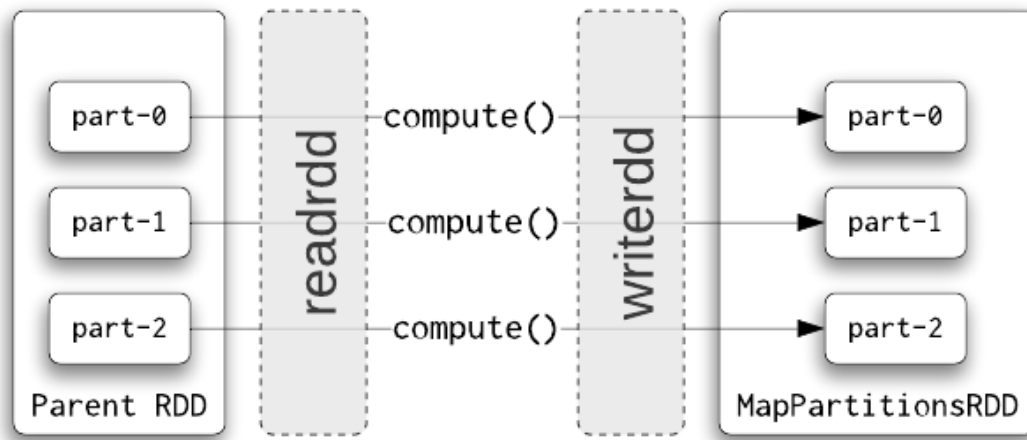


Fig. 3.14 Interception component for MapPartitionsRDD

the interception of the *runJob()* method that is responsible for the execution of actions. The relevant code snippet that intercepts the execution of *runJob()* declared in the *SparkContext* class can be seen in listing 3.22.

Listing 3.22 Interception component of event captor for data integrity of actions

```

1 new AgentBuilder.Default()
2   .type(type -> type.getName().equals("org.apache.spark.SparkContext"))
3   .transform((builder, typeDescription, classLoader, module) -> {
4     return builder
5       .serialVersionUID(1L)
6       .method(method -> (method.getName().equals("runJob") && method.getParameters().size() == 3))
7       .intercept(MethodDelegation
8         .withDefaultConfiguration()
9         .withBinders(Morph.Binder.install(Morpher.class))
10        .to(new SparkContextRunJobDelegator(type, properties)));
11  }).installOn(instrumentation);

```

Note that in listing 3.22, in line 10 we delegate the execution of the *runJob()* method to a custom implementation of the method. Having examined the event capturing process of events for transformations with narrow dependencies and actions, the only other type of transformations that are possible in a Spark job are transformations with wide dependencies. These types of transformations are more complex with regards to the generation of events because of the intermediate storage of data before the data shuffle.

A data shuffle is comprised of two stages. The first one is the map stage where the data is segmented in a number of groups that is equal to the number of the reducers that will

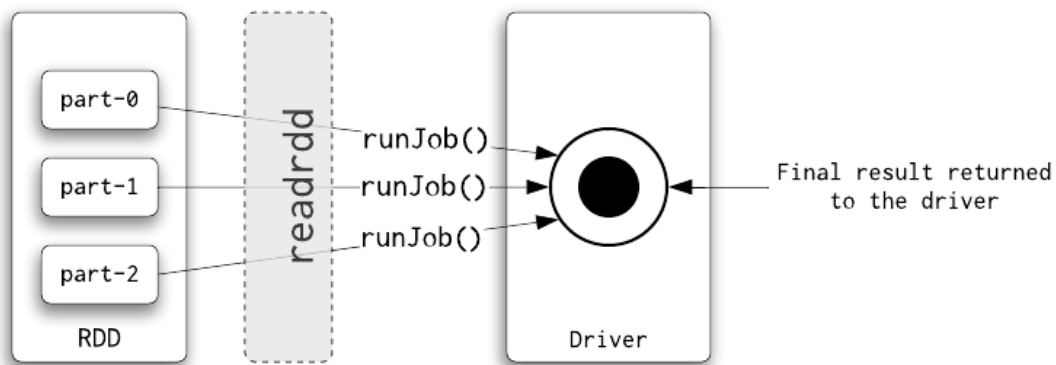


Fig. 3.15 Interception component for *runJob()* method in *SparkContext* class

apply the reduce function. The grouping is performed on the basis of some property of each individual data item e.g. the value of a key. Apache Spark has a default implementation for the grouping of the data that will be consumed by the reducers. To be able to calculate the checksums for the intermediate data we need to intercept the code that performs the grouping, and while the data is grouped to calculate the hash values for the checksums. The logic is exactly the same like the one we used when we built the custom monitorable iterator for transformations with narrow dependencies where we intentionally chose to avoid running the computation twice; one for the data processing and one for the event capturing.

From an implementation point of view, all data shuffles are conducted by means of implementing a base abstract class called *ShuffleWriter*. The default implementation is described in class *SortShuffleWriter*¹⁶ and it implements a sort-based shuffle. The sort shuffle writer writes the data items in separate files, one per reducer and eventually merges them into a single file combined with range values that define what segment of the file should be used by what reducer. The *SortShuffleWriter* writes the grouped data on the disk with the help of a method called *write()*¹⁷. In this method, with the assistance of a class called *ExternalSorter*, that data is written on separate partition files and the list of the lengths of the segments for each reducer partition is returned. This is done by means of invoking the *writePartitionedFile()*

¹⁶<https://github.com/apache/spark/blob/1b575ef5d1b8e3e672b2fca5c354d6678bd78bd1/core/src/main/scala/org/apache/spark/shuffle/sort/SortShuffleWriter.scala>

¹⁷<https://github.com/apache/spark/blob/1b575ef5d1b8e3e672b2fca5c354d6678bd78bd1/core/src/main/scala/org/apache/spark/shuffle/sort/SortShuffleWriter.scala#L51>

method in the *ExternalSorter* class where data is split and stored on files on the disk for each mapper. Therefore, the location where all the data tuples are processed and the most suitable place to calculate the checksums is when the *writePartitionedFile()* is invoked. The code snippet for the interception of the *writePartitionedFile()* method in the *SortShuffleWriter* class can be seen in listing 3.23.

Listing 3.23 Interception component of the event captor for data integrity for the *writePartitionedFile()* method in class *ExternalSorter*

```

1 new AgentBuilder.Default()
2   .type(type -> type.getName().equals("org.apache.spark.util.collection.ExternalSorter"))
3   .transform((builder, typeDescription, classLoader, module) -> {
4       return builder
5         .serialVersionUID(1L)
6         .method(method -> method.getName().equals("writePartitionedFile"))
7         .intercept(MethodDelegation.to(new ExternalSorterWritePartitionedFileDelegator(type, properties)));
8   }).installOn(instrumentation);

```

Note that in line 7 the instrumented *writePartitionedFile()* method will be delegated for execution to another method in a custom class named *ExternalSorterWritePartitionedFileDelegator* that we will further explain in section 3.3.3.

This takes care of the *writeshuffle* events from the side of the mappers. By the same token, the event captor has to emit *readshuffle* events when the reducer partitions arrive on the reducers. To achieve this we need to intercept the code that reads the partitions on the *compute()* method in the *ShuffledRDD* class. In this method, when Saprk is attempting to compute the *ShuffledRDD*, it reads the mapped partitions as streams of input data. More specifically, the data is read with the assistance of a helper structure called a shuffle manager. The shuffle manager exposes a set of utilities to enable the interaction of the reducers with the data that has been shuffled. More specifically, in the computation of the *ShuffledRDD* the *getReader()* method is invoked on the shuffle manager and subsequently on the read object returned by the *getReader()* method, the *read()* method is invoked. The result that is returned is an iterator where the data from all the mappers that are intended to be used by that reducer is returned to it as a iterator of iterators i.e. it represents a collection of iterators one for each mapper. This iterator is a custom type of iterator that is implemented internally in Apache Spark and is called a *ShuffleBlockFetcherIterator*¹⁸. The *ShuffleBlockFetcherIterator* represents an iterator of iterators that, when read by the reducers, is flattened by means of

¹⁸<https://github.com/apache/spark/blob/688b0c01fac0db80f6473181673a89f1ce1be65b/core/src/main/scala/org/apache/spark/storage/ShuffleBlockFetcherIterator.scala>

using the default implementation of the *flatMap()* method of the base scala *Iterator* class that the *ShuffleBlockFetcherIterator* class extends. At this point we will intercept the Apache Spark's source code to compute the checksums of the mapped data that each reducer will receive as input from the corresponding mappers. The code snippet for the interception of the *flatMap()* method in the *ShuffleBlockFetcherIterator* class can be seen in listing 3.24.

Listing 3.24 Interception component of event captor for data integrity when shuffled data is read from the reducers

```

1 new AgentBuilder.Default()
2   .type(type -> type.getName().equals("org.apache.spark.storage.ShuffleBlockFetcherIterator"))
3   .transform((builder, typeDescription, classLoader, module) -> {
4     return builder
5       .serialVersionUID(1L)
6       .method(method -> method.getName().equals("flatMap"))
7       .intercept(MethodDelegation
8         .withDefaultConfiguration()
9         .withBinders(Morph.Binder.install(Morpher.class))
10        .to(new ShuffleBlockFetcherIteratorFlatMapDelegator(type, properties)));
11   }).installOn(instrumentation);

```

Note that in line 10 the instrumented *flatMap()* method will be delegated for execution to another method in a custom class named *ShuffleBlockFetcherIteratorFlatMapDelegator* that we will further explain in section 3.3.3.

Finally, the last piece of code that needs to be intercepted is the Spark code that computes the *ShuffledRDD* and produces its output. The code snippet for the instrumentation of the *compute()* method of *ShuffledRDD* is presented in listing 3.25

Listing 3.25 Interception component of event captor for data integrity when shuffled data is written from the reducers

```

1 new AgentBuilder.Default()
2   .type(type -> type.getName().equals("org.apache.spark.rdd.ShuffledRDD"))
3   .transform((builder, typeDescription, classLoader, module) -> {
4     return builder
5       .serialVersionUID(1L)
6       .method(method -> method.getName().equals("compute"))
7       .intercept(MethodDelegation
8         .withDefaultConfiguration()
9         .withBinders(Morph.Binder.install(Morpher.class))
10        .to(new ShuffledRDDComputeDelegator(type, properties)));
11   }).installOn(instrumentation);

```

In line 10 the instrumented *compute()* method will be delegated for execution to another method in a custom class named *ShuffledRDDComputeDelegator* that we will further elaborate in section 3.3.3.

Delegation Component

In this section we will provide a detailed account of the delegation components for all the interceptors that we listed in section 3.3.3 above. Below we give a list with all the delegators that are responsible for the execution of the intercepted methods in order to collect the appropriate events that will support the monitoring process. A common property of all the delegator classes is that upon construction they use up two parameters to instantiate new delegator objects. The first one is the type of emitter that the delegation method will use to emit the events and the second one a set of properties that allows the invoker to parameterize the delegator. This pattern is universal across all delegators and offer a systematic way to customise the event captor when it is initially loaded.

The delegator for the *compute()* method of the HadoopRDD for the interception component in listing 3.20, can be seen in listing 3.26.

Listing 3.26 Delegation component for the execution of the *compute()* method of the HadoopRDD class

```

1 public class HadoopRDDComputeDelegator {
2
3     private final EmitterType type;
4     private final Properties properties;
5
6     public HadoopRDDComputeDelegator(EmitterType type, Properties properties){
7         this.type = type;
8         this.properties = properties;
9     }
10
11     @RuntimeType
12     public <K, V> Iterator<Tuple2<K, V>> compute(
13         @Argument(0) Partition theSplit,
14         @Argument(1) TaskContext context,
15         @This RDD<Tuple2<K, V>> rdd,
16         @Morph Morpher<InterruptibleIterator<Tuple2<K, V>>> morpher) throws InterruptedException {
17
18
19         String applicationId = SparkEnv$.MODULE$.get().conf().get("spark.app.id");
20         String applicationName = SparkEnv$.MODULE$.get().conf().get("spark.app.name");
21
22         Map<String,String> parameters = new LinkedHashMap<>();
23         parameters.put("appId", applicationId);
24         parameters.put("appName", applicationName);
25         parameters.put("rddId", String.valueOf(rdd.id()));
26         parameters.put("partId", String.valueOf(theSplit.index()));
27
28         return new InterruptibleIterator(context, new DataIntegrityMonitorableIterator<Tuple2<K, V>>(
29             morpher.invoke(theSplit, context),
30             type,
31             properties,
32             OperationType.WRITERDD,
33             parameters));
34     }

```

35 }

Note that in line 28 the `InterruptibleIterator`¹⁹ iterator that is returned, is wrapped around our custom `DataNtegrityMonitorableIterator` where *writerdd* events are emitted when the `HadoopRDD` produces its results when computed. The delegator for the *compute()* method of the `MapPartitionRDD` for the interception component shown in listing 3.21, is presented in listing 3.27.

Listing 3.27 Delegation component for the execution of the *compute()* method of the `MapPartitionsRDD` class

```

1 public class MapPartitionsRDDComputeDelegator {
2
3     private final EmitterType type;
4     private final Properties properties;
5
6     public MapPartitionsRDDComputeDelegator(EmitterType type, Properties properties){
7         this.type = type;
8         this.properties = properties;
9     }
10
11     @RuntimeType
12     public <T, U> Iterator<U> compute(
13         @Argument(0) Partition split,
14         @Argument(1) TaskContext context,
15         @This RDD<T> rdd, @FieldValue("f") Function3<TaskContext, Integer, Iterator<T>, Iterator<U>> f) throws IOException {
16
17         String applicationId = SparkEnv$.MODULE$.get().conf().get("spark.app.id");
18         String applicationName = SparkEnv$.MODULE$.get().conf().get("spark.app.name");
19
20         Map<String,String> readParams = new LinkedHashMap<>();
21         readParams.put("appId", applicationId);
22         readParams.put("appName", applicationName);
23         readParams.put("rddId", String.valueOf(rdd.firstParent(rdd.elementClassTag()).id()));
24         readParams.put("partId", String.valueOf(split.index()));
25
26         Iterator<T> input = new DataIntegrityMonitorableIterator<T>(
27             rdd.firstParent(rdd.elementClassTag()).iterator(split, context),
28             type,
29             properties,
30             OperationType.READRDD,
31             readParams);
32
33         Map<String,String> writeParams = new LinkedHashMap<>();
34         writeParams.put("appId", applicationId);
35         writeParams.put("appName", applicationName);
36         writeParams.put("rddId", String.valueOf(rdd.id()));
37         writeParams.put("partitionId", String.valueOf(split.index()));
38
39         return new DataIntegrityMonitorableIterator<U>(
40             f.apply(context, split.index(), input),
41             type,
42             properties,
43             OperationType.WRITERDD,
44             writeParams);

```

¹⁹<https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/InterruptibleIterator.scala>

```

45
46     }
47 }

```

Note that in line 26 the input of the *compute()* method is wrapped around our custom *DataIntegrityMonitorableIterator* and a *readrdd* event is emitted. The input represents the data of the first parent RDD and is the data that will be fed to the *compute()* method of the current RDD. Similarly, in line 39 the output of the *compute()* method is wrapped around our custom *DataIntegrityMonitorableIterator* and a *writerdd* is emitted. The delegator for the *runJob()* method of the SparkContext class for the interception component shown in listing 3.22, is presented in listing 3.28.

Listing 3.28 Delegation component for the execution of the *runJob()* method of the SparkContext class

```

1 public class SparkContextRunJobDelegator implements Serializable{
2
3     private final EmitterType type;
4     private final Properties properties;
5
6     public SparkContextRunJobDelegator(EmitterType type, Properties properties){
7         this.type = type;
8         this.properties = properties;
9     }
10
11     @RuntimeType
12     public Object runJob(
13         @Argument(0) RDD rdd,
14         @Argument(1) Function2 f,
15         @Argument(2) Object classTag,
16         @Morph Morpher<Object> morpher,
17         @This Object sc) throws IOException {
18
19         String applicationId = SparkEnv$.MODULE$.get().conf().get("spark.app.id");
20         String applicationName = SparkEnv$.MODULE$.get().conf().get("spark.app.name");
21
22         final class Func extends AbstractFunction2<TaskContext, Iterator, Object> implements Serializable {
23
24             @Override
25             public Object apply(TaskContext context, Iterator it) {
26
27                 Map<String,String> parameters = new LinkedHashMap<>();
28                 parameters.put("appId", applicationId);
29                 parameters.put("appName", applicationName);
30                 parameters.put("rddId", String.valueOf(rdd.id()));
31                 parameters.put("partitionId", String.valueOf(context.getPartitionId()));
32
33                 return f.apply(context, new DataIntegrityMonitorableIterator(it, type, properties, OperationType.READRDD, parameters));
34             }
35
36         }
37         return morpher.invoke(rdd, new Func(), classTag);
38     }
39 }

```

In listing 3.22, where the interception component for the `runJob()` is presented, the method takes as a parameter a function under the name `func` that is applied on each partition of the RDD. This function should be instrumented to emit the events for the data integrity monitoring activity since it is the location where the input data is being read from the action operation and therefore a `readrdd` event needs to be captured. The way to achieve this is shown in the declaration in the listing 3.28 from line 22 until line 36 where a custom function under the name `Func` is wrapped around the function `func` that is passed as an argument to the `runJob()`. Note that in line 37 the original method `runJob()` invoked but now the wrapper function `Func` with the event capturing capabilities is passed as an argument facilitating in that way the emission of the appropriate events.

Now that we examined the delegation components for transformations with narrow dependencies and actions, we will present the delegation components for the intercepted code that will produce the events for the transformations with wide dependencies. The delegator for the `writePartitionedFile()` method of the `ExternalSorter` class for the interception component shown in listing 3.23, is presented in listing 3.29.

Listing 3.29 Delegation component for the execution of the `writePartitionedFile()` method of the `ExternalSorter` class

```

1
2 public class ExternalSorterWritePartitionedFileDelegator {
3
4     private final EmitterType type;
5     private final Properties properties;
6
7     public ExternalSorterWritePartitionedFileDelegator(EmitterType type, Properties properties){
8         this.type = type;
9         this.properties = properties;
10    }
11
12    @RuntimeType
13    public <K, V, C> long[] writePartitionedFile(
14        @Argument(0) BlockId blockId,
15        @Argument(1) File outputFile,
16        @This Object sorter,
17        @FieldValue("blockManager") BlockManager blockManager,
18        @FieldValue("fileBufferSize") Integer fileBufferSize,
19        @FieldValue("org$apache$spark$util$collection$ExternalSorter$$context") TaskContext context,
20        @FieldValue("org$apache$spark$util$collection$ExternalSorter$$aggregator") Option<Aggregator<K, V, C>> aggregator,
21        @FieldValue("org$apache$spark$util$collection$ExternalSorter$$ordering") Option<Ordering<K>> ordering,
22        @FieldValue("org$apache$spark$util$collection$ExternalSorter$$keyComparator") Comparator<K> keyComparator,
23        @FieldValue("org$apache$spark$util$collection$ExternalSorter$$numPartitions") Integer numPartitions,
24        @FieldValue("org$apache$spark$util$collection$ExternalSorter$$serInstance") SerializerInstance serInstance,
25        @FieldValue("map") WritablePartitionedPairCollection<K, C> map,
26        @FieldValue("buffer") WritablePartitionedPairCollection<K, C> buffer) throws NoSuchElementException {
27
28        String applicationId = SparkEnv$.MODULE$.get().conf().get("spark.app.id");
29        String applicationName = SparkEnv$.MODULE$.get().conf().get("spark.app.name");

```

```

30 long[] lengths = new long[numPartitions];
31 DiskBlockObjectWriter writer =
32     blockManager
33         .getDiskWriter(
34             blockId,
35             outputFile,
36             serInstance,
37             fileBufferSize * 1024,
38             context.taskMetrics().shuffleWriteMetrics());
39
40
41 if(((ExternalSorter)sorter).numSpills() == 0){
42     WritablePartitionedPairCollection collection = aggregator.isDefined() ? map : buffer;
43
44     Iterator<Tuple2<Tuple2<Integer, K>, C>> sortedIterator =
45         collection.partitionedDestructiveSortedIterator(
46             (ordering.isDefined() aggregator.isDefined()) ?
47                 Some.apply(keyComparator) :
48                 Option.empty());
49
50     MessageDigest md = MessageDigest.getInstance("SHA-256");
51
52     while(sortedIterator.hasNext()){
53
54         Tuple2<Tuple2<Integer, K>, C> cur = sortedIterator.next();
55         int partitionId = cur._1()._1();
56
57         while (sortedIterator.hasNext() && cur._1()._1() == partitionId){
58             md.update(new Tuple2(cur._1()._2(), cur._2()).toString().getBytes());
59             writer.write(cur._1()._2(), cur._2());
60             cur = sortedIterator.next();
61         }
62
63         FileSegment segment = writer.commitAndGet();
64         lengths[partitionId] = segment.length();
65
66         Emitter emitter = EventEmitterFactory.getInstance(type, properties);
67         emitter.connect();
68
69         Map<String, String> parameters = new LinkedHashMap<>();
70         parameters.put("appId", applicationId);
71         parameters.put("appName", applicationName);
72         parameters.put("shuffleId", String.valueOf(((ShuffleBlockId)blockId).shuffleId()));
73         parameters.put("mapId", String.valueOf(((ShuffleBlockId)blockId).mapId()));
74         parameters.put("reduceId", String.valueOf(partitionId));
75         parameters.put("checksum", "external - " + DatatypeConverter.printHexBinary(md.digest()));
76
77         long operationId = MonitorUtilities.generateRandomLong();
78         emitter.send(MonitorUtilities.createEvent(operationId,
79             properties.getProperty("eventType"), OperationType.WRITESHUFFLE, parameters));
80         emitter.close();
81     }
82 }else {
83     Iterator<Tuple2<Integer, Iterator<Product2>>> partitionedIterator = ((ExternalSorter)sorter).partitionedIterator();
84
85     while(partitionedIterator.hasNext()){
86         Tuple2 tuple = partitionedIterator.next();
87
88         Integer id = (Integer) tuple._1();
89         Iterator<Product2> elements = (Iterator<Product2>)tuple._2();
90
91         if(elements.hasNext()){
92             while(elements.hasNext()){
93                 Product2 elem = elements.next();

```

```

94         writer.write(elem._1(), elem._2());
95     }
96
97     FileSegment segment = writer.commitAndGet();
98     lengths[id] = segment.length();
99 }
100 }
101 }
102
103 writer.close();
104 context.taskMetrics().incMemoryBytesSpilled(((ExternalSorter)sorter).memoryBytesSpilled());
105 context.taskMetrics().incDiskBytesSpilled(((ExternalSorter)sorter).diskBytesSpilled());
106 context.taskMetrics().incPeakExecutionMemory(((ExternalSorter)sorter).peakMemoryUsedBytes());
107
108 return lengths;
109 }
110 }

```

The implementation of the delegation componet shown in listing 3.29 is built on the image of the original `writePartitionedFile()` method of Apache Spark with the exception that it provides the capacity to produce the checksums for the reducer partitions. The delegation method, by means of using the Byte Buddy's `@FieldValue` annotation, gets access on the class variables of the `ExternalSorter` instance. This can be seen in lines from 17 until 26. In line 44, the sorted iterator that holds all the data tuples of the RDD sorted by the reducer that will consume it, is instantiated. Before the traversal of the iterator begins, in line 50 the message digest that will be used for the computation of the checksums of the partitions is instantiated. Then, in line 52 the traversal of the iterator starts. If the current reducer partition of the current tuple is the same as the partition of the previous one, the data tuples ought to be processed by the same reducer. This clause is shown in the while loop in line 57. In line 58 the bytes for each tuple that belongs to the same partition are fetched and the `update()` method is invoked on the message digest instance to produce the correct checksum when the traversal of the whole partition has completed. After that, in line 59, the tuple is stored on the disk by means of using an instance of the `DiskBlockObjectWriter`²⁰ class. Conversely, if the partition id is not the same as the partition id of the previous tuple i.e. the current tuple needs to be processed by the next in order reducer, the while loop in line 52 no longer holds true which means that all the tuples for the current partition have been visited and therefore it is time to calculate its checksum. Also, in line 66 an emitter is instantiated to support the emission of the relevant `writeshuffle` event to the monitor. The calculation of the checksum is conducted in line 75 by means invoking the `digest()` method on the message digest instance.

²⁰<https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/storage/DiskBlockObjectWriter.scala>

Finally, the event is sent to the monitor with the assistance of the *send()* method of the emitter in line 78.

The delegator for the *flatMap()* method of the *ExternalSorter* class for the interception component shown in listing 3.23, is presented in listing 3.29.

Listing 3.30 Delegation component for the execution of the *writePartitionedFile()* method of the *ShuffleBlockFetcherIterator* class

```

1 public class ShuffleBlockFetcherIteratorFlatMapDelegator {
2
3     private final EmitterType type;
4     private final Properties properties;
5
6     public ShuffleBlockFetcherIteratorFlatMapDelegator(EmitterType type, Properties properties){
7         this.properties = properties;
8         this.type = type;
9     }
10
11     @RuntimeType
12     public Iterator<Tuple2<BlockId, InputStream>> flatMap(
13         @Morph Morph<Iterator<Tuple2<BlockId, InputStream>>> morpher,
14         @Argument(0) Function1<Tuple2, Iterator> func) {
15
16         String applicationId = SparkEnv$.MODULE$.get().conf().get("spark.app.id");
17         String applicationName = SparkEnv$.MODULE$.get().conf().get("spark.app.name");
18
19         final class Func extends AbstractFunction1<Tuple2<BlockId, InputStream>, Iterator<Tuple2<BlockId, InputStream>>>
20         implements Serializable {
21             @Override
22             public Iterator<Tuple2<BlockId, InputStream>> apply(Tuple2<BlockId, InputStream> v1) {
23                 ShuffleBlockId blockId = (ShuffleBlockId) v1._1;
24
25                 Map<String, String> parameters = new LinkedHashMap<>();
26                 parameters.put("appId", applicationId);
27                 parameters.put("appName", applicationName);
28                 parameters.put("shuffleId", String.valueOf(blockId.shuffleId()));
29                 parameters.put("mapId", String.valueOf(blockId.mapId()));
30                 parameters.put("reduceId", String.valueOf(blockId.reduceId()));
31
32                 return new DataIntegrityMonitorableIterator(
33                     func.apply(v1), type, properties, OperationType.READSHUFFLE, parameters);
34             }
35         }
36         return morpher.invoke(new Func());
37     }
38 }

```

The *ShuffleBlockFetcherIterator* is an iterator that holds items of data type **Tuple2<BlockId, InputStream>** i.e. pairs of block ids and input streams. The block id is modelled as of type *BlockId*²¹ that identifies a particular block of data stored on the disk from the mappers. The open stream to the actual data tuples that the reducer will be consuming which corresponds to

²¹<https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/storage/BlockId.scala>

the specific block, is modelled as of type *InputStream*²². When the *flatMap()* will be invoked for each pair, it will produce a iterator with the data to be processed by the reducer that will be fetched from the corresponding input stream. To be able to compute the checksums on the data that had been previously been produced by the mappers, we need to wrap it around our *DataIntegrityMonitorableIterator* to enable the capturing of the right events. To achieve this, we create a custom function under the name *Func* that is declared in line 19. This function when invoked, it will invoke the original method *func* that it takes as an argument and will wrap its return value around our custom *DataIntegrityMonitorableIterator* iterator shown in line 32. The custom iterator is parameterised appropriately to allow it to capture the *readshuffle* events and produce the checksums the same way they were produced for transformations with narrow dependencies. In that way, we have embedded the event capturing capabilities on the iterator and no additional step need to be taken. Similar to the other delegation components for capturing events for the monitoring of data integrity, we avoid iterating over the data twice. Both data processing and event capturing takes place in one go with the help of the *DataIntegrityMonitorableIterator* iterator.

3.4 Summary

In this chapter we have provided a detailed overview of the architecture of the monitoring framework that we propose for the runtime monitoring of security properties for Big Data pipelines. The framework describes the separate steps that users need to take to enable the monitoring activity. More precisely, the monitoring framework will enable the users to do the following: a. define the pipeline in the form of a composite service that is composed of multiple atomic services, b. define their security monitoring requirements and provide all the necessary information that might be necessary to enable their monitoring, c. translate all the security requirements into low level monitoring artefacts that the monitoring infrastructure is able to understand and interact with, d. install the monitoring rules that are the result of the security requirements specification and finally e. deploy the event captors where necessary to facilitate the monitoring activity. For our analysis we were able to showcase three security properties that can be expressed in Event Calculus and therefore monitored by the proposed

²²<https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html>

framework. Concluding this chapter, we provided a detail analysis of the SLA template specification that correspond to the security properties that we examined.

Chapter 4

SLA Management Web Dashboard

4.1 Application Architecture Overview

In the SLA Manager web application we use the *Model View Controller* [79] paradigm. The *View* aspect of the application is implemented as a set of HTML web pages enriched with PHP¹ code to help with the invocation of the methods from the controllers. The *Controller* component of MVC is implemented as a set of HTTP methods in the form of a RESTful API in Java. Finally the *Model* of the MVC is represented as a collection of tables in a MySQL² database. To communication between the PHP web pages and the RESTful API is over HTTP and the communication between the API and the database is over TCP with the assistance of the Java Database Connectivity interface. A typical flow of execution is as follows: The user, through the web interface, triggers an action. By invoking a PHP script a method from the RESTful API is invoked which in turn makes a JDBC call to the database and either reads data from the model or updates it and gets back to the user. An overview of the architecture of the application can be seen in figure 4.1.

4.2 Application Repository

The application repository is implemented as MySQL database schema where a set of tables are used to model and store the data. In the database we persist all the metadata regarding

¹<https://www.php.net/manual/en/intro-what-is.php>

²<https://dev.mysql.com/doc/refman/5.7/en/>

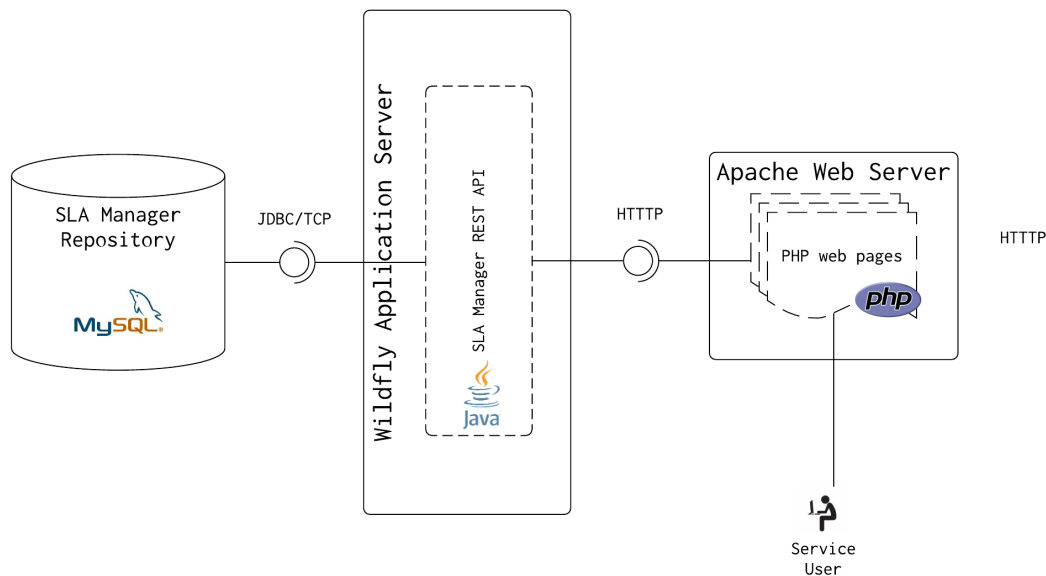


Fig. 4.1 SLA Manager web application architecture

the SLAs and their objective as well as all the parameters that are required for the generation of monitoring rules and the installation of the event captors that will realise the it. The tables that have been used are the following:

1. **users** - This is a table where all the user metadata is stored.
2. **projects**- This is a table where all the metadata for an SLA project is stored.
3. **compositeservices** - This is a table where the composite services for all SLA projects are stored. Composite services are associated with an SLA project.
4. **atomicservices** - This is a table where the atomic services for all SLA projects are stored. Atomic services are associated with the composite service that they are part of by means of a foreign key in the *compositeservices* table
5. **assets** - This is a table to store all the assets for an atomic service. In our implementation each atomic service has only one asset that operation itself. This is represented as a table under the name *operations*. This flexible design allows for the definition of more assets per atomic service such as the input of the atomic service or its output.

6. ***operations*** - This is a table where the operations of an atomic service are stored. Operations are associated with the atomic service that they refer to by means of a foreign key reference to the *atomiservices* table.
7. ***securityproperties*** - This is a table where all the available security properties are stored. Examples of properties that we included in the database are *Availability*, *Privacy* and *Integrity*.
8. ***slotemplates*** - This is a table where all the available templates for the security properties stored in the *securityproperties* table are defined. A security property can be associated with multiple slo templates which reflects the fact that a security properties can be measured by means of running different motoring rules. For instance, availability can be expressed as the total time of execution or the mean time to recover from a failure. Each one of those manifestations of availability will be stored as a separate SLO template and it is up to the user to decide which one will ber used.
9. ***sloparameters*** - This is a table where the parameters for each SLO template. A template can have multiple parameters. In this table, apart from the name of the parameter we also keep its data type. Our system supports a string, an enumeration and a list data type. This is a flexible design that makes the addition of new parameters for templates very straightforward. Also, the PHP web page that presents the parameters to the users uses the data types to build the appropriate UI elements dynamically. For the string data type it will present to the user an HTML textbox, for the enumeration it will present a dropdown menu whereas for the list data type it will present a specially designed component that allows the user to type in multiple values.
10. ***parametervvalues*** - This is a table where the actual values for the SLO paramters are stored as they have been keyed by the users when they associated a security property with an SLO template
11. ***assetsecuritypropertypairs*** - This is table that is used to correlate an asset with a security property
12. ***slos*** - This is a table that represents a service level objective for an asset of an atomic service. The slo keeps are reference by means of a foreign key to the *projects* table

to keep track of the project that each SLO belongs to. Eventually, the SLOs are the artifacts on which the system will produce the monitoring results.

For a comprehensive view of the database, in figure 4.2 we present the entity relationship diagram of the database schema that was used.

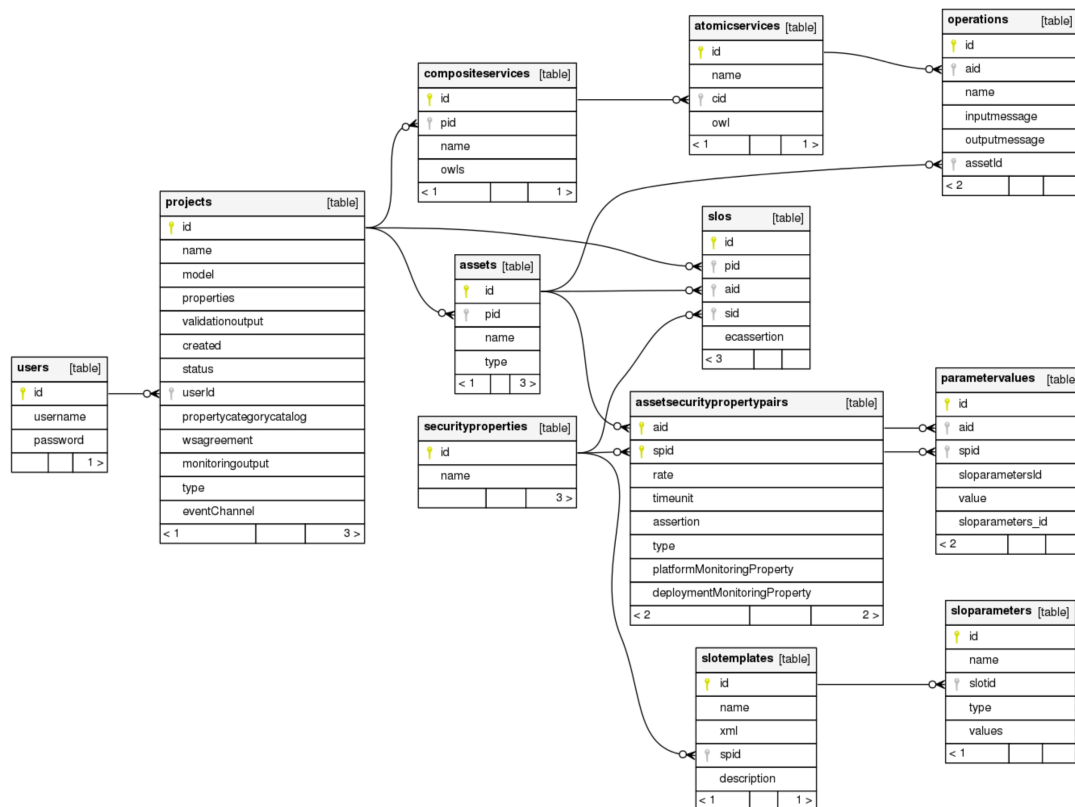


Fig. 4.2 SLA Manager database repository

4.3 Application REST API

The RESTful API that enables the communication between the PHP web application and the application repository comprises of two main controllers namely the *UserRESTController* and *ProjectRESTController*. For each workflow that the user defines with the help of Spring Cloud DataFlow through the *SLAManagerIntegrator* component, a new SLA project with

the same name as the workflow is created in the SLAManager web application repository. All the actions on the project are declared and implemented in the ***ProjectRESTController*** controller in the form of functions that can be invoked over HTTP as a set of RESTful API calls. Similarly, in the ***UserRESTController*** is the controller where all the user related actions are declared and implemented. A separate controller for the users has been created to enable the correlation of workflows with different users. Users can only see and interact with the SLA projects that they have created and own. Also, to enforce security, the invocation of all the RESTful methods require *HTTP Basic Authentication*³ i.e. a valid username and password are required for the invocation of each method of the API. The password for each user is stored in the application repository and is hashed using the BCrypt [104] algorithm. This is protecting the users' personal data while it significantly reduces the possibility for passwords being breached. Finally,

Below we give an overview of the operations that are available for every controller accompanied by a short description. In all the operations the data that is being exchanged is model in the JSON⁴ notation. The operations for the ***UserRESTController*** and ***ProjectRESTController*** can be seen in table 4.1.

Operations for <i>UserRESTController</i>	
URL	{host}:{port}/slamanager/rest/api/users/login
URL Parameters	None
Path Parameters	None
HTTP Body Parameters	None
HTTP Method	GET

³<https://datatracker.ietf.org/doc/rfc2617/>

⁴<https://www.json.org/>

Description	This operation allows the validation of the user by checking the credentials provided by the user with the credentials stored in the database. If the credentials that are provided is valid, then they are stored in the session of the PHP web application and they are used for the invocation of any subsequent REST methods. The login operation is the only operation that does not require basic authentication because by definition it is invoked when the user has not yet logged in.
URL	{host}:{port}/slamanager/rest/api/users
URL Parameters	<ol style="list-style-type: none"> 1. <i>username</i>: String - A string representation for the username of the new user 2. <i>password</i>: String - A string representation for the password of the new user
Path Parameters	None
HTTP Method	POST
HTTP Body Parameters	None
Description	Create a new user
URL	{host}:{port}/slamanager/rest/api/users/{id}
URL Parameters	None
Path Parameters	<ol style="list-style-type: none"> 1. <i>id</i>: Integer - A unique integer identifier for each user that is stored as a primary key in the <i>users</i> table

HTTP Body Parameters	None
HTTP Method	GET
Description	Get all the metadata for a user except for the password.
Operations for ProjectRestController	
URL	{host}:{port}/slamanager/rest/api/users/{id}/projects
URL Parameters	<ol style="list-style-type: none"> 1. <i>id</i>: Integer - Unique identifier of the users that owns the new project that will be created
Path Parameters	None
HTTP Body Parameters	<ol style="list-style-type: none"> 1. <i>name</i>: String - A string representation for the name of the new SLA project
HTTP Method	POST
Description	Create a new SLA project manually
URL	{host}:{port}/slamanager/rest/api/users/{uid}/projects/scdf
URL Parameters	None

HTTP Body Parameters	<ol style="list-style-type: none"> 1. <i>taskDefinition</i>: String - A string representation in JSON notation of tasks that are defined as a composite task from which the new SLA project is going to be created. For every task of the composite task a new atomic task is going to be created. Also, based on the security properties that were associated with the tasks when the workflow was created in Spring Cloud DataFlow, the appropriate service level objectives will be created.
HTTP Method	POST
Description	Create a new SLA project from a workflow of Big Data analytics service that has been defined in Spring Cloud DataFlow

Table 4.1 Operations of the SLA Manager RESTful API

4.4 Energy producer use-case

To demonstrate our system we will use a hypothetical SME (Small Medium Enterprise) that uses solar panels to produce energy and provide it to its subscribers. The company installs solar panels in the households that are subscribed to its services which are linked to a small gateway that collects information about how much energy the panels are producing and how much energy the appliances in the house are consuming. The households use a hybrid model for the energy provision which is a mix of electrical energy coming from the locally installed solar panel and the standard electricity energy provider. A critical service that the energy provider company requires to incorporate is the computation of the average consumption per household per appliance. This will reveal information with regards to how the appliances are being used energy-wise and the company could use this information to optimise the usage of the solar panels. Based on the computation of the average consumption of each appliance the solar energy provider company can decide how much energy should be provided from the

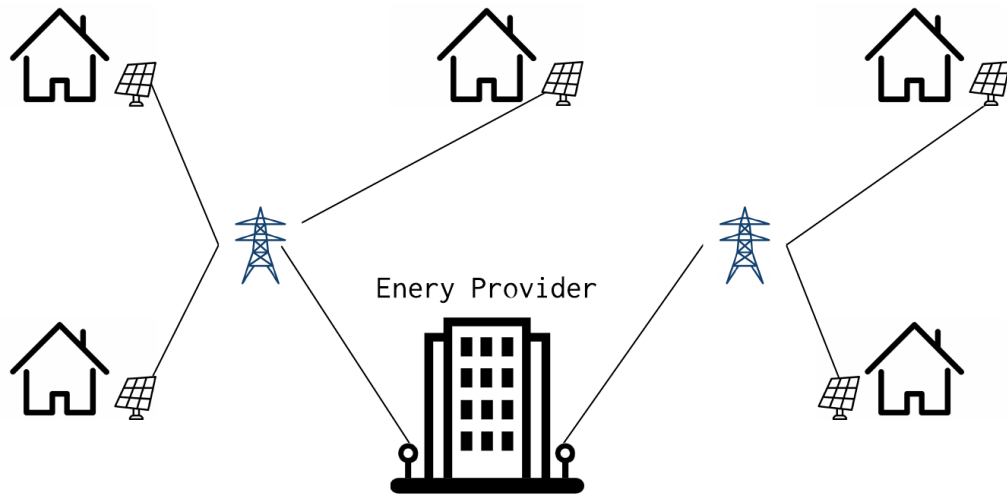


Fig. 4.3 Solar panel energy production use-case

batteries of the solar panels and how much from the ordinary network of the standard energy provider to optimise energy consumption for the household and to reduce the overall cost for the subscribers. A visual representation of the scenario described can be seen in figure 4.3

In our use-case we have implemented a simple example that is complex enough to demonstrate and address the fundamental challenges that we were faced with when we attempted to monitor certain non-functional properties for Big Data analytics service that we described. To implement the service, we rely on a set of hypotheses with respect to how the data is attained and what is the model that it complies with.

The first hypothesis is that at each household a gateway device is installed that sends the measurements for every appliance to a distributed file-system – in our case HDFS. Measurements are time stamped from the gateways that they are collected. When the service runs it does not consider real time data, but it only relies on measurements that has been stored in the file-system until the very moment that the execution of that services has commenced.

The second hypothesis is with regards to the data model i.e. how each measurement is represented. A measurement collected by a gateway is composed of a set of data-points that, depending on their position, they represent the energy consumed by a specific appliance. For example, each data item that is transmitted from the gateways contains all the energy consumption details for each appliance at a specific point in time.

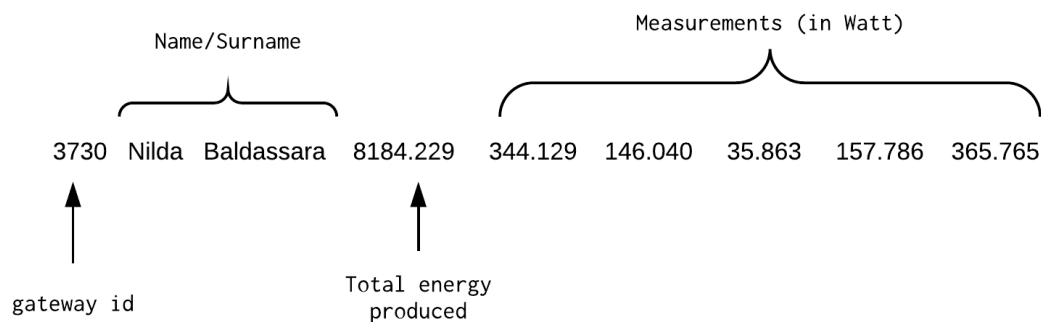


Fig. 4.4 Example of a measurement from a household

As it can be seen in figure 4.4, the id of the gateway and the name of the owner of the house is collected to be able to group the data. The scenario that we will examine to demonstrate the monitoring capabilities of the framework will require the execution of the separate Big Data analytics services that will represent the atomic services of the pipeline or composite service. The pipeline is comprised of the following services:

1. ***LoadAndAnonymizeService*** - Load the data from HDFS and apply a hash function to anonymise the name and surname of the owner of the household.
2. ***PrepareDataService*** - After the data has been anonymised in step 1, flatten the measurements per household per appliance to allow the computation of averages.
3. ***ComputeAverageService*** - After the data has been flattened in step 2, apply an appropriate aggregate function to compute the average consumption per appliance per household.

All the atomic services presented above will have to be executed in the order that they are presented in the list. Each one of the services will produce data that will be fed to the next service until the execution of the last service is completed. In our example we will monitor

the location of execution for the *LoadAndAnonymizeService*, a non-functional property that pertains to data privacy, because of the sensitive nature of the service which is to anonymise the data. We will also monitor response time for for *PrepareDataService* that relates to availability and data integrity for *ComputeAverageService*.

4.5 Screenshots for the energy provider use-case

Originally the service pipeline needs to be defined. This is done with the assistance of Spring Cloud DataFlow and the first screen the user can see is presented in figure 4.5. On the left menu by clicking on the *Apps* label the list of all the available applications that can be used are presented. Initially, no applications are available to enable the composition of service workflows.

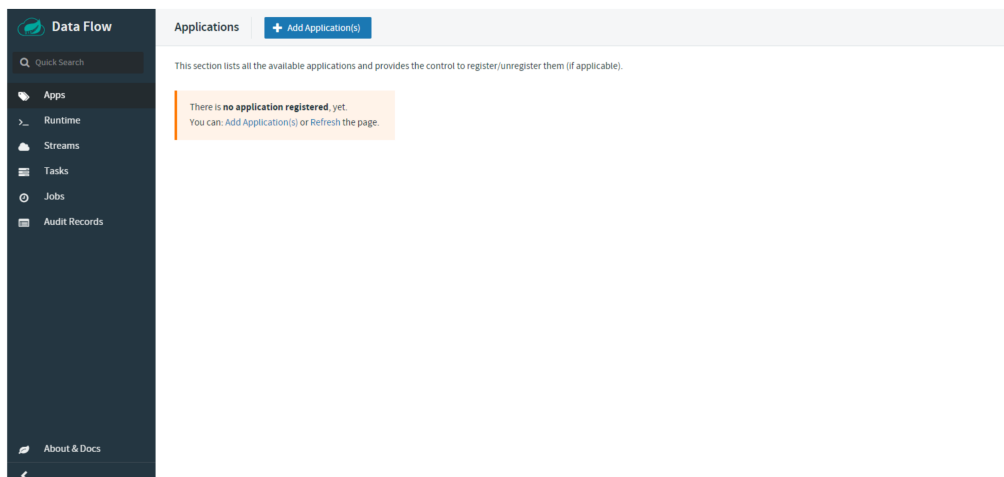
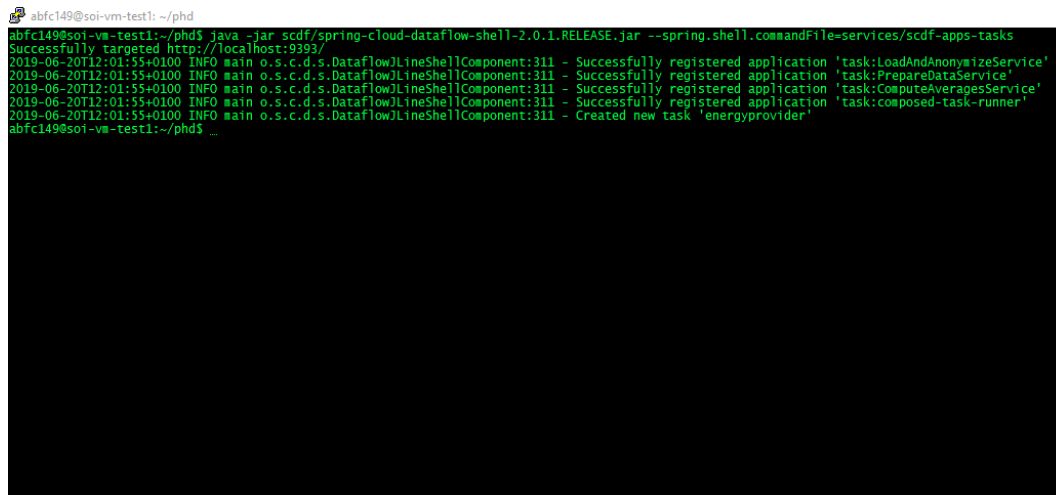


Fig. 4.5 Spring Cloud DataFlow UI - Empty list of available applications

To add the applications that will be used in the example use-case, we take advantage of command-line tool that is shipped with the standard version of the Spring Cloud DataFlow called the Spring Cloud DataFlow Shell. This utility can take as a parameter a configuration file with all the necessary parameters of the application that we need to load and make available in the list of applications in the Spring Cloud DataFlow server. The execution of the loading of the applications in the server is shown in figure 4.6. Note that apart from the three services described in section 4.4, another application named *composed-task-runner* is

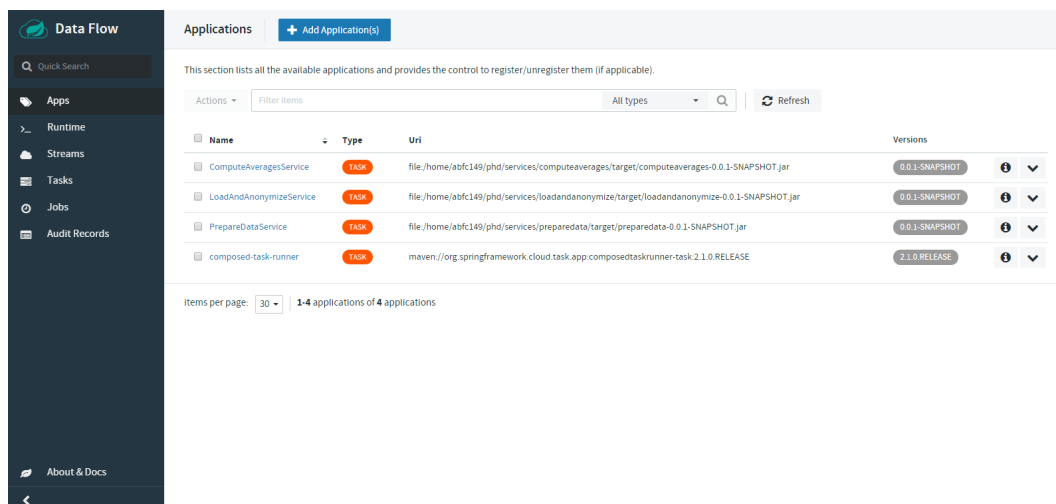
loaded as well. This is necessary to allow the composition of composite tasks that will later be executed from this application. The *composed-task-runner*⁵ is an open source application and has been built by the Spring Cloud DataFlow and is publicly available.



```
abfc149@soi-vm-test1: ~/phd
abfc149@soi-vm-test1:~/phd$ java -jar scdf/spring-cloud-dataflow-shell-2.0.1.RELEASE.jar --spring.shell.commandFile=services/scdf-apps-tasks
Successfully targeted http://localhost:9393/
2019-06-20T12:01:55+0100 INFO main o.s.c.d.s.DataFlowJLineShellComponent:311 - Successfully registered application 'task:LoadAndAnonymizeService'
2019-06-20T12:01:55+0100 INFO main o.s.c.d.s.DataFlowJLineShellComponent:311 - Successfully registered application 'task:PrepareDataService'
2019-06-20T12:01:55+0100 INFO main o.s.c.d.s.DataFlowJLineShellComponent:311 - Successfully registered application 'task:ComputeAveragesService'
2019-06-20T12:01:55+0100 INFO main o.s.c.d.s.DataFlowJLineShellComponent:311 - Successfully registered application 'task:composed-task-runner'
2019-06-20T12:01:55+0100 INFO main o.s.c.d.s.DataFlowJLineShellComponent:311 - Created new task 'energyprovider'
abfc149@soi-vm-test1:~/phd$
```

Fig. 4.6 Spring Cloud DataFlow UI - Load applications from the command line

As soon as the applications are loaded under the *Apps* section, all the available applications will be listed as shown in figure 4.7.



Name	Type	Uri	Versions
ComputeAveragesService	TASK	file:/home/abfc149/phd/services/computeaverages/target/computeaverages-0.0.1-SNAPSHOT.jar	0.0.1-SNAPSHOT ⓘ ▼
LoadAndAnonymizeService	TASK	file:/home/abfc149/phd/services/loadandanonymize/target/loadandanonymize-0.0.1-SNAPSHOT.jar	0.0.1-SNAPSHOT ⓘ ▼
PrepareDataService	TASK	file:/home/abfc149/phd/services/preparedata/target/preparedata-0.0.1-SNAPSHOT.jar	0.0.1-SNAPSHOT ⓘ ▼
composed-task-runner	TASK	maven://org.springframework.cloud.task.app.composedtaskrunner-task-2.1.0.RELEASE	2.1.0.RELEASE ⓘ ▼

Items per page: 30 1-4 applications of 4 applications

Fig. 4.7 Spring Cloud DataFlow UI - Populated list of available applications

Under the *Tasks* section as seen in figure 4.8, all the loaded applications are available and can be used for the composition of pipelines with them.

⁵<https://github.com/spring-cloud-task-app-starters/composed-task-runner>

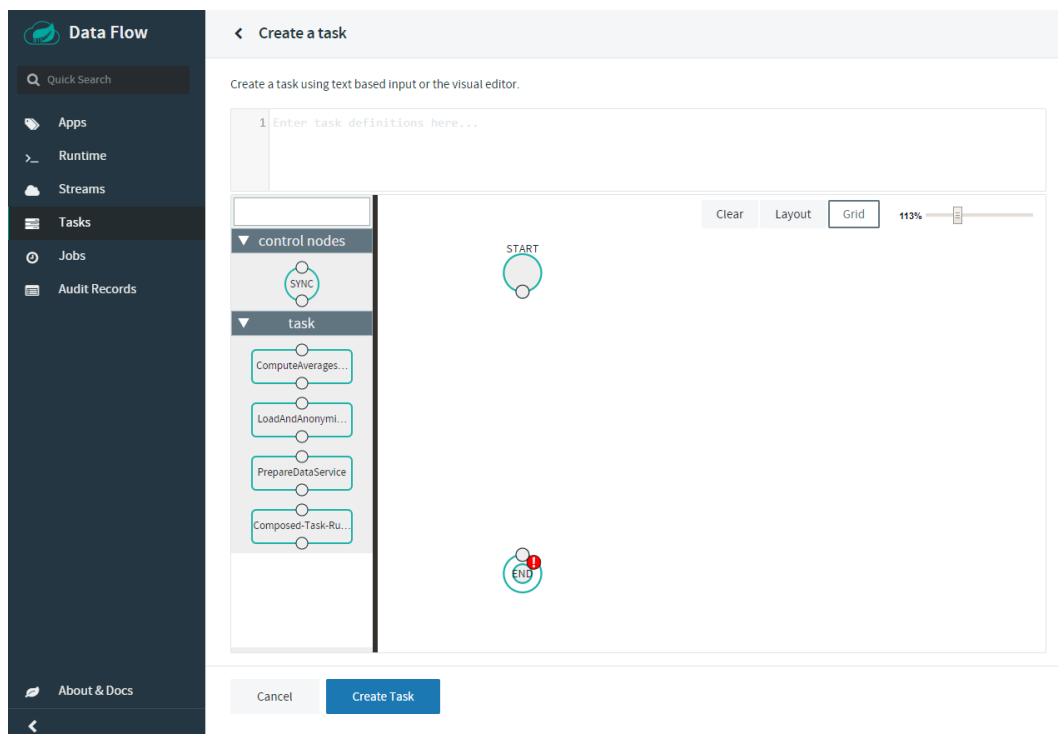


Fig. 4.8 Spring Cloud DataFlow UI - Create a new composite task from a drag-n-drop menu

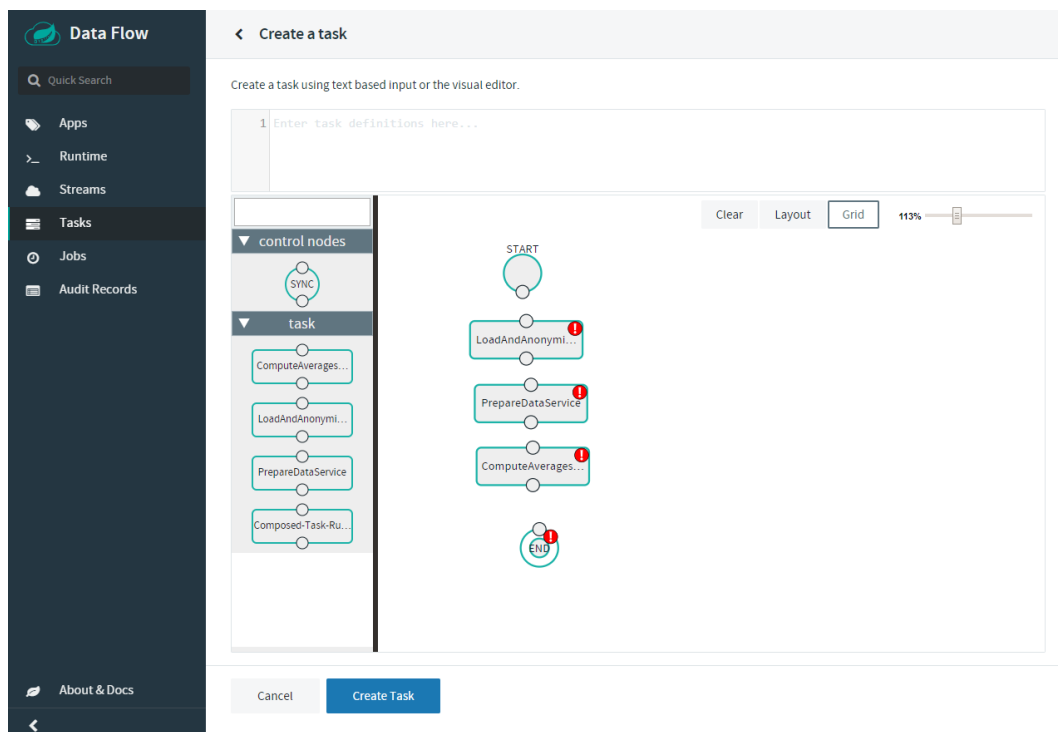


Fig. 4.9 Spring Cloud DataFlow UI - View of the composite task pipeline without the edges

The composition of the pipeline can be performed by means of dragging and dropping the applications into the drawing area on the right of the screen. This can be better viewed in figure 4.9 above.

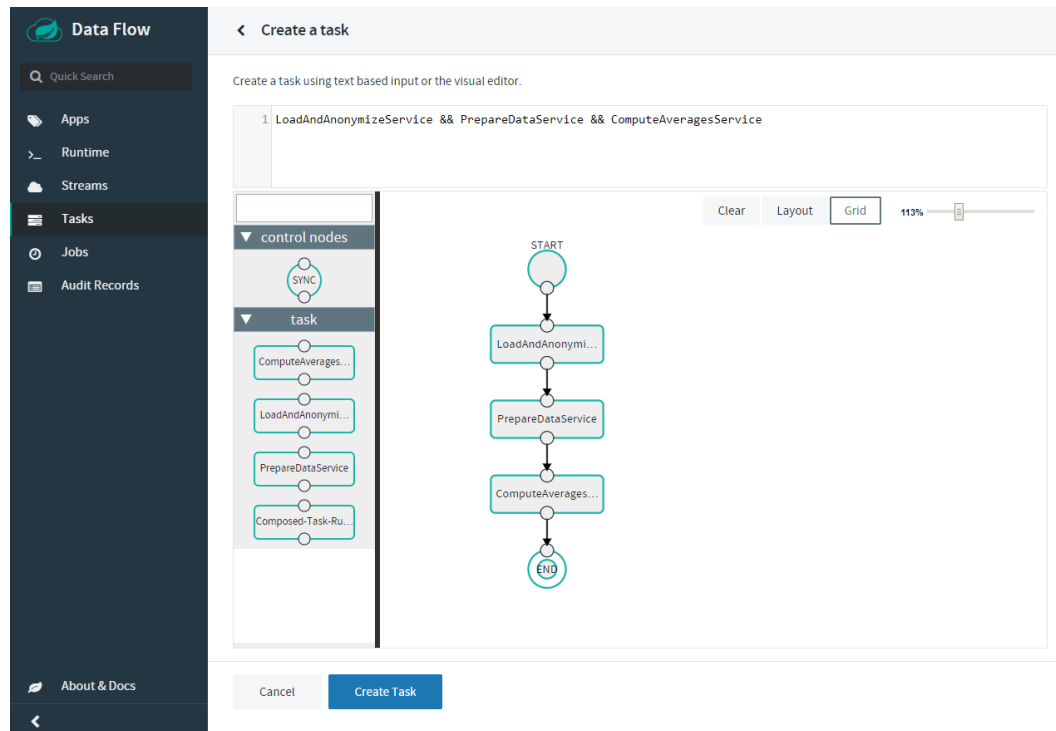


Fig. 4.10 Spring Cloud DataFlow UI - View of the composite task pipeline with the edges

Now that the applications that will be used in the pipeline have been defined, we need draw the arrows that will allow Spring Cloud DataFlow to generate the execution plan for the pipeline. In our case the applications will be executed in a sequential order. A view of the resulting pipeline can be seen in figure 4.10.

To associate each service with a security property that is required from the users to be monitored, we need to select the service and tap on the gear icon on the left of the box that the service label is contained. An example of this for the *LoadAndAnonymizeService* can be seen in figure 4.11.

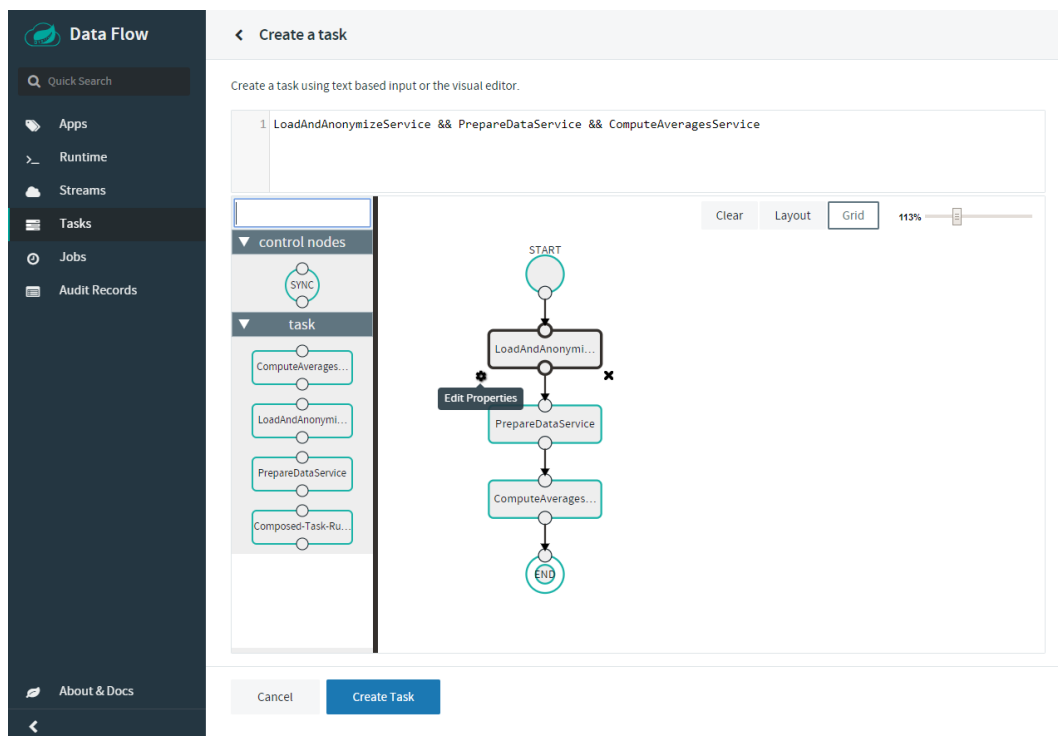


Fig. 4.11 Spring Cloud DataFlow UI - Edit the properties for the *LoadAndAnonymizeDataService*

A pop-up menu will prompt the user to type in the following: a label for each task which is useful for logging, the input and the output file names that will be used from the service to read its input and persist its processing results, and finally the security property that will be monitored for that particular service. The UI for the insertion of the service properties for each service can be viewed in figures 4.12, 4.13 and 4.14 respectively.

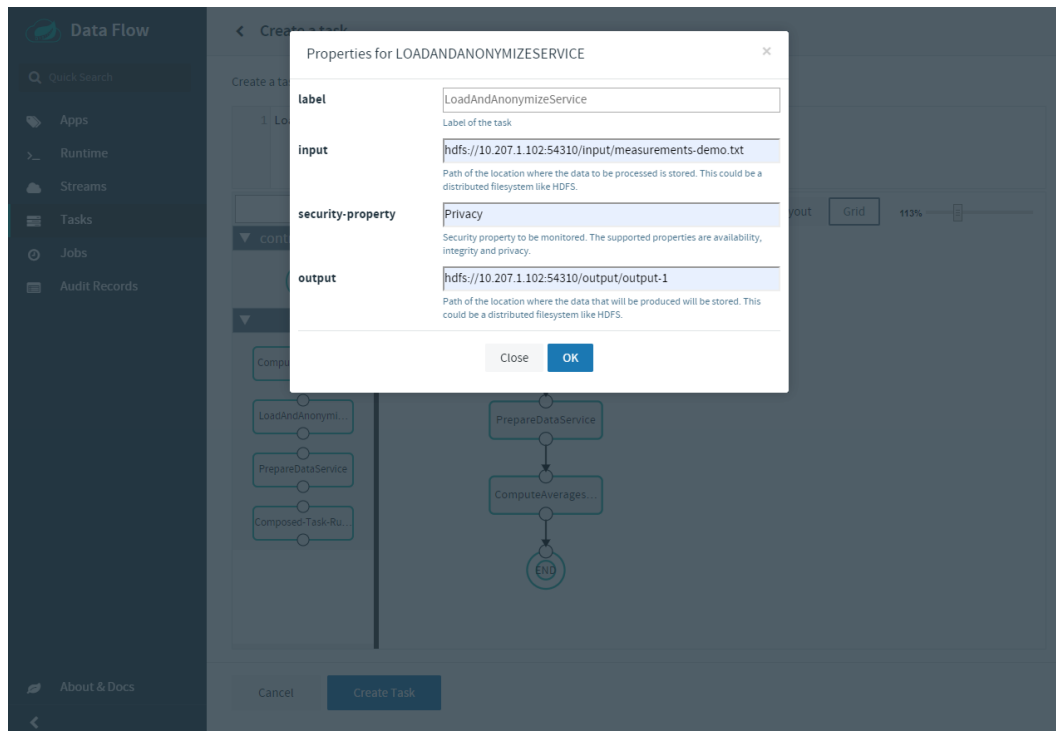


Fig. 4.12 Spring Cloud DataFlow UI - Properties for the *LoadAndAnonymizeDataService*

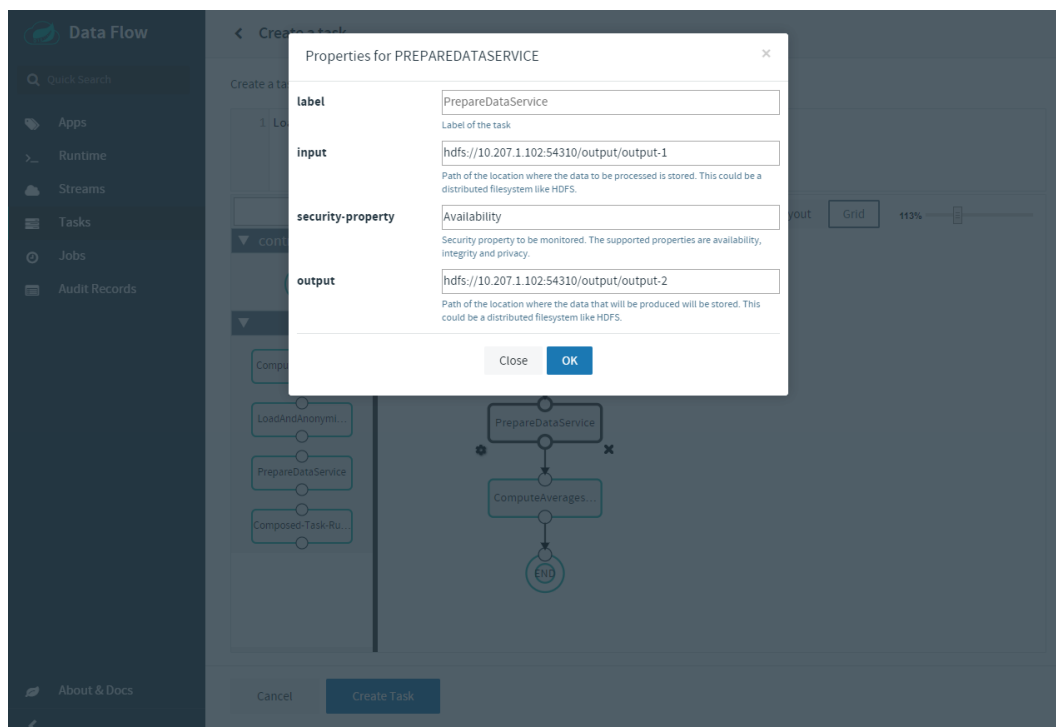


Fig. 4.13 Spring Cloud DataFlow UI - Properties for the *PrepareDataService*

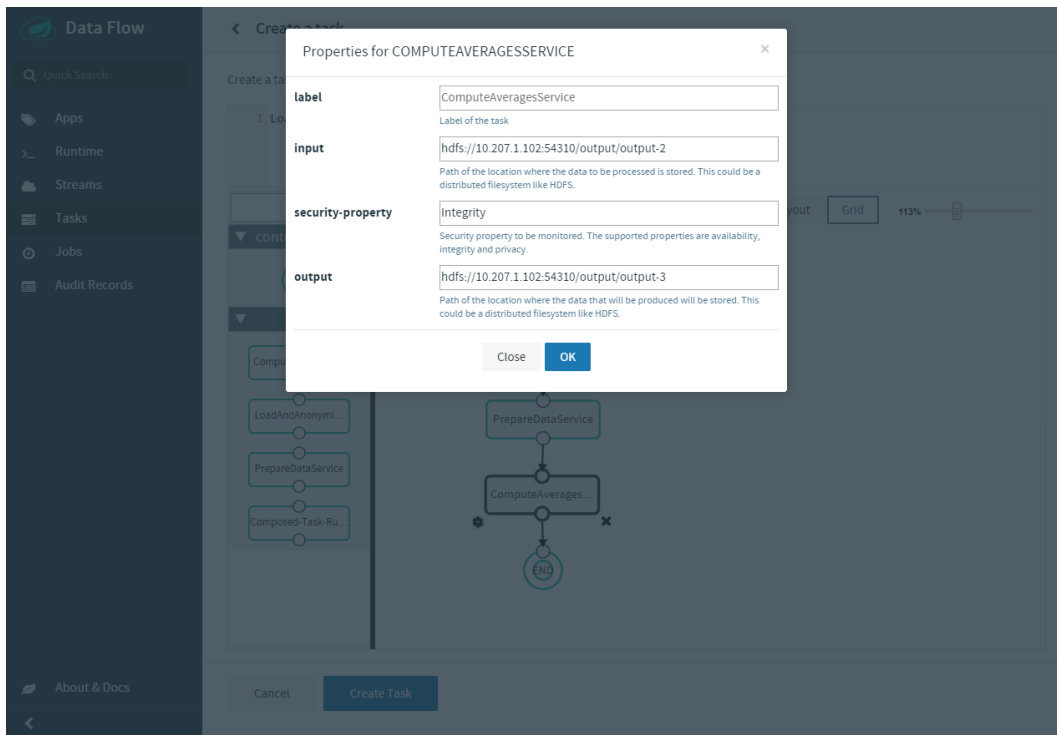


Fig. 4.14 Spring Cloud DataFlow UI - Properties for the *ComputeAverageService*

Finally, when all the properties have been typed-in, the user presses the *Create task* button and is prompted to type-in a name for the pipeline as shown in figure 4.15. This name is going to be used later in the SLA Manager applications to create a name for the SLA. The name of the SLA is exactly the same as the name of the pipeline with the suffix *-SLA* added at the end. E.g. in our case the name of the pipeline is **energyprovider** and therefore the name of the SLA is going to be **energyprovider-SLA**. Also note that before the pipeline is created and saved, a view of the pipeline is presented in the domain specific language that Spring Cloud DataFlow is using to describe composite tasks and is available for inspection from the user.

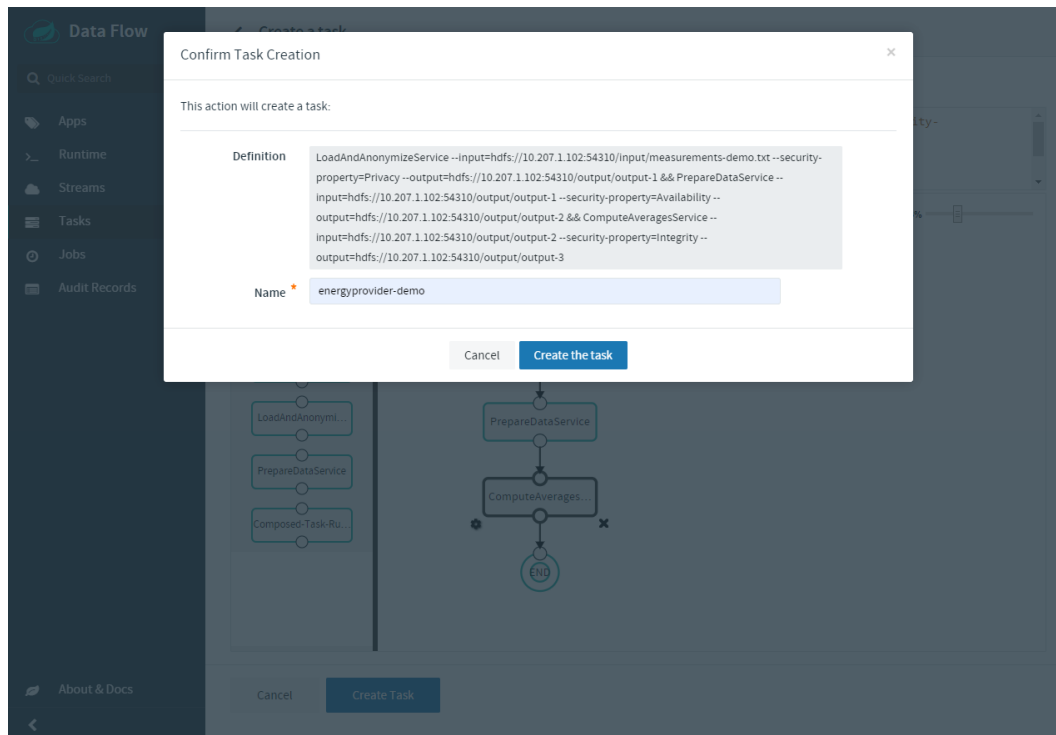


Fig. 4.15 Spring Cloud DataFlow UI - Type-in a name for the composite task

Now that the service pipeline has been created, the user will move on to the SLA Manager web application to complete the instantiation of the SLA and to view the monitoring results when the pipeline will get executed. Initially the users will have to use their credentials to login to the SLA Manager web application as illustrated in figure 4.16.

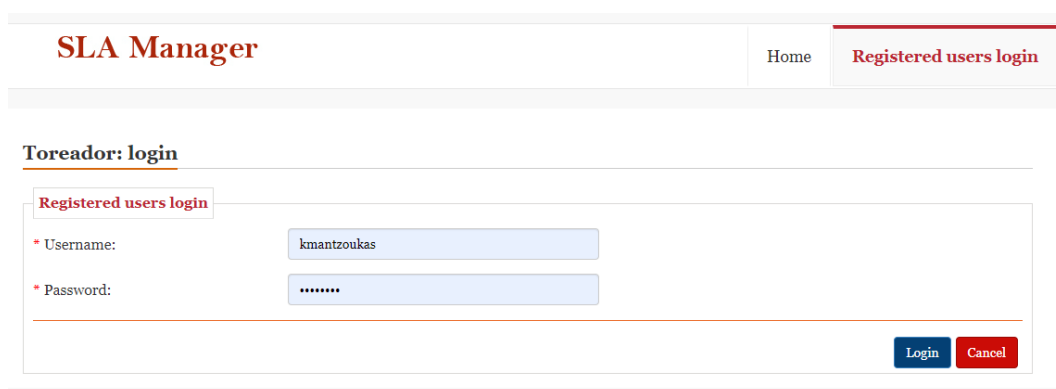


Fig. 4.16 SLA Manager - Login to the SLA Manager

Once users are logged in they can view the list of all the SLA projects that they own. This concept is illustrated in figure 4.17. Note that the *energyprovider-SLA* has been created

on behalf of the user automatically from the *SLAManagerIntegrator* module presented in the framework architecture layout shown in figure 3.2. The *SLAManagerIntegrator* will go through the pipeline that was originally created in Spring Cloud DataFlow and will figure out all the assets/atomic services that comprise it. It will then proceed to create and store those assets in the SLA Manager repository and associate them with the security properties that the users have selected when they created the pipeline. Each pair of an asset and a security property is a service level objective that our monitoring framework will have to monitor and evaluate when the pipeline gets executed.

The screenshot shows the SLA Manager web application. At the top, there is a header with the title "SLA Manager" in red. To the right of the title is a "Projects" button and a user profile "kmantzoukas: Logout". Below the header, there is a section titled "Existing SLA projects" with a table listing the projects. The table has three columns: "Name", "TimeStamp", and "Status". One project is listed: "energyprovider-SLA" with a timestamp of "2019-06-20 16:11:20" and a status of "CREATED". Below the table is a "Create new project" button.

Name	TimeStamp	Status
energyprovider-SLA	2019-06-20 16:11:20	CREATED

Fig. 4.17 SLA Manager - List of SLA projects of the user

When tapping on the SLA project shown in figure 4.17, the asset/security property pairs are presented to the user as shown in figure 4.18

The screenshot shows the "Monitoring results" section of the SLA Manager. At the top, there is a "Generate monitorable SLA" button. Below it, there is a table with three rows, each representing a service asset and its associated security property. The table has two columns: "Asset: service-operation-lightsource-PrepareDataService" and "Property: Availability". The status for each row is "Ecassertion".

Asset: service-operation-lightsource-PrepareDataService	Property: Availability	Ecassertion
Asset: service-operation-lightsource-ComputeAveragesService	Property: Integrity	Ecassertion
Asset: service-operation-lightsource-LoadAndAnonymizeService	Property: Privacy	Ecassertion

Fig. 4.18 SLA Manger - View of the list of the service assets and security property pairs

Pressing the **Generate monitorable SLA** button shown in figure 4.18 over the list of SLOs will take users to the screen shown in figure 4.19 where they need to specify how the security property that they selected will be monitored. This is done by choosing what template is going to be used to generate the monitoring rules that will be used during the monitoring process. This step is critical because it defines what monitoring rules will be loaded in the EVEREST monitor and what event captors will be installed across the cluster to facilitate the collection of the appropriate monitoring data.

Generate monitorable SLA: 1/2

Associate assets and templates

Create operations and associative actions.

Asset: service-operation-energyprovider-LoadAndAnonymizeService	Property: Privacy
* Template: Privacy - Location of execution	Template info Template xml
or Define property in EC assertion:	

Asset: service-operation-energyprovider-PrepareDataService	Property: Availability
* Template: Availability - Total time of exec	Template info Template xml
or Define property in EC assertion:	

Asset: service-operation-energyprovider-ComputeAveragesService	Property: Integrity
* Template: Integrity - Verification of check	Template info Template xml
or Define property in EC assertion:	

Next Cancel

Fig. 4.19 SLA Manager - View of the asset/property pairs

Hitting the *Next* button in the screen shown in figure 4.19 will take them to the screen illustrated in figure 4.20 where they will type-in the parameter values for the template that they have opted to use. Note that this UI is built dynamically by reading the data types of the parameters from the *sloparameters* table in the SLA Manager database. Hitting the *Generate* button will generate SLOs and will store it in the SLA Manager repository but most importantly will use the metadata of the SLOs to generate the monitoring rules and load them in the EVEREST monitor. Now that this step has been taken, the monitoring engine of EVEREST is ready to accept events and evaluate them against the newly created rules.

Now that the SLOs are created and the monitor is ready, users can go the the Spring Cloud DataFlow and execute the pipeline that can be viewed under the *Tasks*. The relevant

Generate monitorable SLA: 2/2

Define parameters

Define values for associated parameters.

For template: Availability - Total time of execution

* timeUnits:

MINUTES

* time:

2

For template: Integrity - Verification of checksums

No parameters

For template: Privacy - Location of execution

* trustedIps:

10.207.1.102

+

Previous

Generate

Cancel

Fig. 4.20 SLA Manager - Type-in the parameter values for the SLO templates

screen can be seen in figure 4.21. The pipeline can be executed by pressing the button of the down arrow all the way to the right of the task of interest and then selecting the *Launch task* option.

Data Flow

Quick Search

Apps

Runtime

Streams

Tasks

Jobs

Audit Records

About & Docs

Tasks

+ Create task(s)

Task 4 Executions

Actions

Filter items

Refresh

Name	Definitions	Status
energyprovider-demo	LoadAndAnonymizeService --output=hdfs://10.207.1.102:54310/output/output-1 --input=hdfs://10.207.1.102:54310/input/measu...	UNKNOWN
energyprovider-demo-ComputeAveragesService	ComputeAveragesService --output=hdfs://10.207.1.102:54310/output/output-3 --input=hdfs://10.207.1.102:54310/output/output...	UNKNOWN
energyprovider-demo-LoadAndAnonymizeService	LoadAndAnonymizeService --output=hdfs://10.207.1.102:54310/output/output-1 --input=hdfs://10.207.1.102:54310/input/measu...	UNKNOWN
energyprovider-demo-PrepareDataService	PrepareDataService --output=hdfs://10.207.1.102:54310/output/output-2 --input=hdfs://10.207.1.102:54310/output/output-1 ...	UNKNOWN

Items per page: 30

1-4 task definitions of 4 task definitions

Fig. 4.21 Spring Cloud DataFlow UI - Launch the composite task

When the execution commences events will be sent to the EVEREST monitor and will be evaluated. The users can inspect the monitoring results for the rules that were created if they navigate back to the SLA Manager web application.

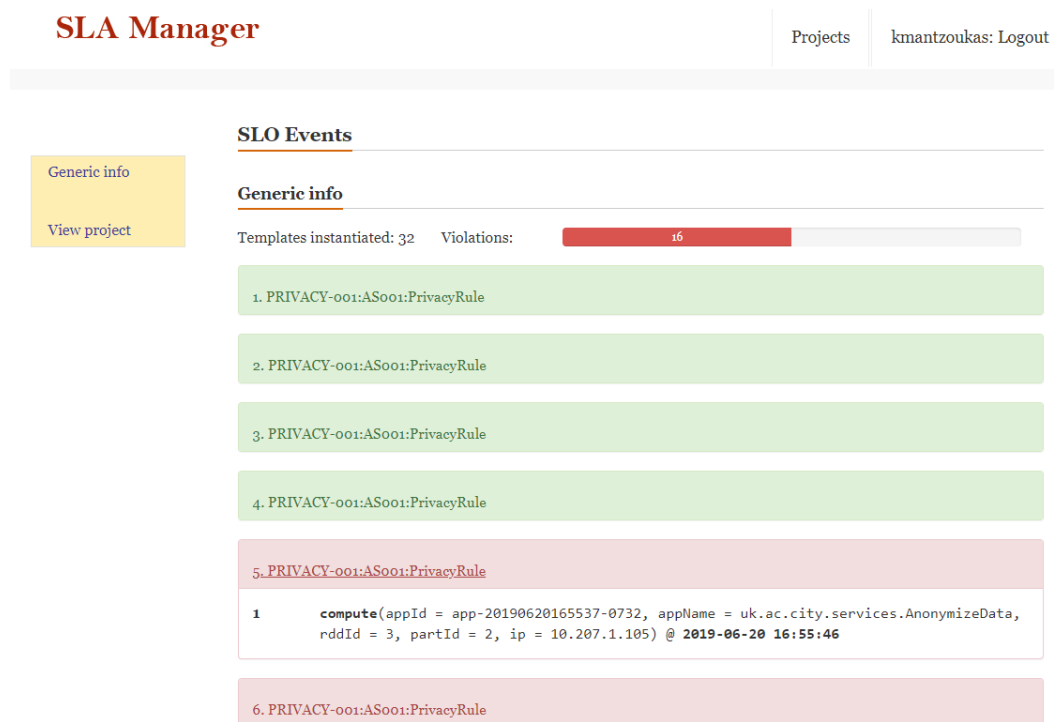


Fig. 4.22 SLA Manager - Inspect the monitoring results for the location of execution security property

Figure 4.22 shows a screen with the monitoring results for the location of execution security property where all the partitions for all the RDDs that have been computed on nodes with untrusted IP addresses are marked as red to indicate a violation. Similarly, partitions that are computed on trusted nodes are marked as green.

In figures 4.23 and 4.24 we illustrate what the monitoring results for the data integrity properties for the *ComputeAveragesService* will look like. The figures correspond to the two instances of the monitoring rules for data integrity as presented in table 3.14 and table 3.15 in section 3.3.3, where the event calculus formulas for monitoring data integrity is presented.

SLA Manager

Projects

kmantzoukas: Logout

Generic info

View project

SLO Events

Generic info

Templates instantiated: 60

Violations: 0

1. INTEGRITY-001:TI:IntMonRule

2. INTEGRITY-001:TI:IntMonRule

1 readrdd(appId = app-20190620170923-0734, appName = uk.ac.city.services.ComputeAverage, rddId = 0, partId = 0, checksum = AC11140D0D6F769BBB23EB2EDD3053E8) @ 2019-06-20 17:09:31

2 writerdd(appId = app-20190620170923-0734, appName = uk.ac.city.services.ComputeAverage, rddId = 0, partitionId = 0, checksum = AC11140D0D6F769BBB23EB2EDD3053E8) @ 2019-06-20 17:09:31

3. INTEGRITY-001:TI:IntMonRule

4. INTEGRITY-001:TI:IntMonRule

5. INTEGRITY-001:TI:IntMonRule

Fig. 4.23 SLA Manager - Inspect the monitoring results for the data integrity security property for transformations with narrow dependencies

16. INTEGRITY-001:TI:IntMonRule

17. INTEGRITY-001:TI:shufMonRule

18. INTEGRITY-001:TI:shufMonRule

1 readshuffle(appId = app-20190620170923-0734, appName = uk.ac.city.services.ComputeAverage, shuffleId = 0, mapId = 0, reduceId = 1, checksum = 9F408FBD5604E184B2CEAA8A45FC8A0E) @ 2019-06-20 17:09:35

2 writeshuffle(appId = app-20190620170923-0734, appName = uk.ac.city.services.ComputeAverage, shuffleId = 0, mapId = 0, reduceId = 1, checksum = 9F408FBD5604E184B2CEAA8A45FC8A0E) @ 2019-06-20 17:09:32

19. INTEGRITY-001:TI:shufMonRule

20. INTEGRITY-001:TI:shufMonRule

Fig. 4.24 SLA Manager - Inspect the monitoring results for the data integrity security property for transformations with wide dependencies

4.6 Summary

In this chapter we have presented the platform that has been built alongside the proposed monitoring framework to allow users to interact with the monitor. The platform is delivered as web application where the users specify the Big Data pipeline and associate its constituent components with the security properties that they wish to monitor. The platform also allows the consolidated view of the monitoring results in a concise way so the users can have a comprehensive view of the SLAs that are being monitored. The presentation of the platform is performed with the assistance of use-case from the domain of Internet of Things where three atomic services are put together to produce the Big Data pipeline that is monitored.

Chapter 5

Framework Evaluation

In this section we will evaluate the framework that was proposed in chapter chapter 3. In the first part of the this chapter at section 5.1, we give an overview of the environment where the execution of our experiments took place. This can help the reader gain a better understanding of the setup that our metrics were collected. Our evaluation of the framework is quantitative i.e. a series of metrics were collected to objectively assess the system's strengths and weaknesses as accurately and objectively as possible. More specifically in our analysis we examined the two main components of the event capturing process which are in section 5.2.1 the **deployment** of the event captors and in section 5.2.2 their **execution** alongside the services. Finally, at the end of this chapter in section 5.3, we discuss the findings of our qualitative analysis and make an attempt to explain them based on our experience during the deployment and execution of the monitoring activity. We also make some observations that are orthogonal to the results that were gathered.

5.1 Experimental setup

In this section we lay out the software and hardware components that were used to execute the Big Data services and then run the monitoring process. To facilitate the accurate collection of the metrics that are required for the framework evaluation and to be able to observe the framework's behaviour for different cluster sizes, we used an open source operating system virtualisation technology called Docker¹ and Docker Engine. Docker offer us the ability to

¹<https://www.docker.com/>

bundle and execute applications in a loosely isolated environment also known as a container. Containers enable the simultaneous execution of multiple containers on the same physical engine. In general, Containers use built-in features of the underlying operation system to virtualize its resources and therefore are lightweight, as opposed to virtualization that happens with the assistance of hypervisors. Containers use the host machine's kernel by isolating its resources such as CPU and memory addresses which makes it possible to run applications in a completely isolated manner.

For the purpose of our experiments, we have set up a docker configuration that allows us to spin up a Spark cluster for a different set of master and worker nodes. More specifically, we have evaluated our proposed thesis against different cluster setups that range from one worker to eight worker nodes. In all occasions, we also have available a master node that coordinates the service execution. Our host machine is available from a cloud provider, namely Google Cloud ² where a VM instance has been rented for the intent of running our experiments. The specification of the instance can be viewed in table 5.1. The operating system of the host machine is Debian with version 10.6 and the kernel version is 4.19.0-12-cloud-amd64 respectively.

Architecture:	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	8
On-line CPU(s) list:	0-7
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel

²<https://cloud.google.com/>

CPU family:	6
Model:	63
Stepping:	0
CPU MHz:	2300
BogoMIPS:	4600
Hypervisor vendor:	KVM
Virtualisation type:	full
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	46080K
NUMA node0 CPU(s):	0-7

Table 5.1 Hardware information for the Google VM instance host machine

To be able to more accurately assess the impact the event monitoring capabilities of our proposal on the deployment and execution of the underlying services, as part of our Docker configuration alongside Apache Spark, we have also installed the Hadoop file system, namely HDFS [120]. HDFS being a distributed file system allows for file replication and high availability through file partitioning and therefore is a more accurate representation of how Big Data services would run in a real setting.

Docker, by means of using a centralised repository, offers the capability to re-use existing open source docker configurations and pull docker images from its open access Docker Hub ³. For that purpose, we used the Big Data Europe Spark and Hadoop images that have been committed to Docker Hub as part of the work delivered for the Big Data Europe ⁴, an EU funded research project . In the docker hub we have been able to re-use docker images

³<https://hub.docker.com/>

⁴<https://www.big-data-europe.eu/>

for the master node ⁵ and the worker nodes ⁶. In this configuration, a Hadoop installation, where we store the data to be processed, is also included. A view of the docker compose file that describes the cluster setup, can be seen in A.5 in the appendix. In the docker compose file we provide a description of the configuration of the cluster where the master node and the workers are defined. In each occasion, we use this docker compose file and the docker compose utility to spin up workers as we need them. In addition, we provide all the necessary plumbing to inject the code that will intercept the service execution and will emit the relevant events required for the evaluation of the security properties to be monitored. The data to be processed is upload into the Hadoop cluster with a one-off operation and that is how the data become available for processing.

5.2 Quantitative Evaluation

In the quantitative evaluation we examine the overhead in terms of additional time that the event captors impose on the execution of the Big Data services. As explained, to use the event captors we need to take two steps; first we need to deploy the event captors i.e. the captors need to be sent to the appropriate nodes in a dynamic way and instrument the underlying Apache Spark code. Second, we need to allow the event captors to execute and collect the monitoring events and implement the monitoring activity. This two-step process mandates that we examine the influence of the event capturing process in each one of those steps. Initially, in we collect the execution time for each service that was described in section 4.4 in chapter 4 related to the energy producer company. Subsequently, we examine the delay that is imposed for the **deployment** of the event captors. Finally, we examine how each event captor is affecting the **execution** of the services for the security properties that were used in our use-case scenario.

To make our analysis more accurate, and to avoid producing results that could be outliers in terms of statistical significance, all the measurements that were collected have been executed 1000 times with the same input. In addition, we used different sizes of datasets and clusters with different numbers of workers. More specifically, we examined data sets with

⁵<https://hub.docker.com/r/bde2020/spark-master>

⁶<https://hub.docker.com/r/bde2020/spark-worker>

500K, 1M, 2M and 4M data points and Apache Spark cluster with one up to eight worker nodes respectively. This allows us to have a more accurate view of the system's performance and to be able to generate histograms of frequency for the measurements gathered. This approach solidifies the validity of our evaluation approach and helps us to draw a more accurate picture of the system when it operates under real-life circumstances and not in an isolated environment.

5.2.1 Event captor deployment overhead

The deployment of the event captors requires that the Java agents will be sent to the nodes that will process the data and will be loaded from each individual executor as a special type of Java application that will in turn, perform the instrumentation of the code. It is critical to highlight that when the Java agents are loaded from the JVM, the actual code to which the instrumented code is delegated does not get executed. In this phase the only thing that happens is that the underlying code is changed at run-time to facilitate the monitoring process but no code is yet executed. For all the event captors that were implemented which correspond to a specific property, we measured the deployment time on every node of the cluster for different data sets and different cluster sizes i.e. number of Spark workers.

Location of execution event captors

In this section we present our finding with regards to the overhead imposed to the monitoring activity when the monitoring of the location of execution is enabled. More specifically, we have plotted the deployment time that it takes for the data privacy event captors to be deployed over different cluster configurations where a different number of Spark workers are available. The experiments were conducted for 500K, 1M and 2M data points to facilitate the evaluation of the effect of the data set size on the overall deployment time. In more detail, in figure 5.1 and figure 5.2 we present the data for 500K data points, in figure 5.3 and figure 5.4 we present the data for 1M data points and finally in figure 5.5 and figure 5.6 we present the data for 2M data points. Note that, for completeness, we treat collect metrics separately for the master and the worker nodes.

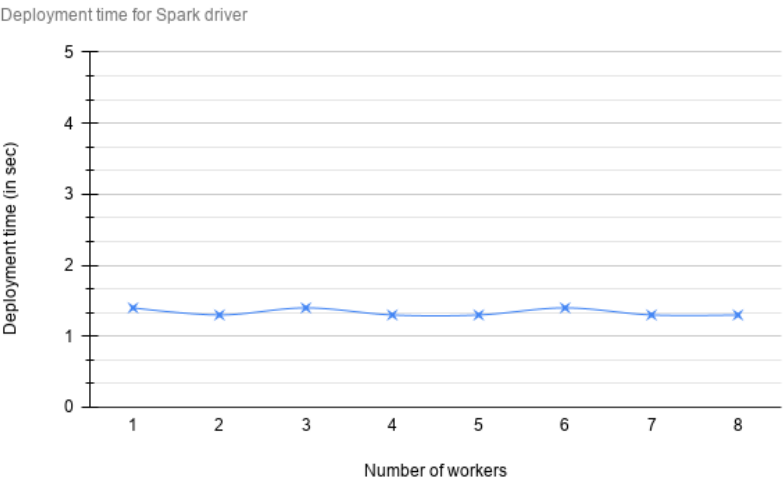


Fig. 5.1 Deployment time of data privacy event captor on the Spark **master** over the number of workers for **500K** data points

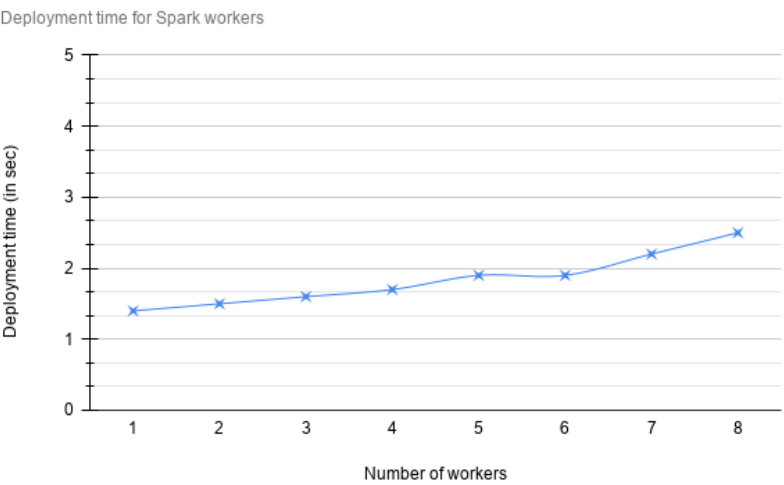


Fig. 5.2 Deployment time of data privacy event captor on the Spark **workers** over the number of workers for **500K** data points

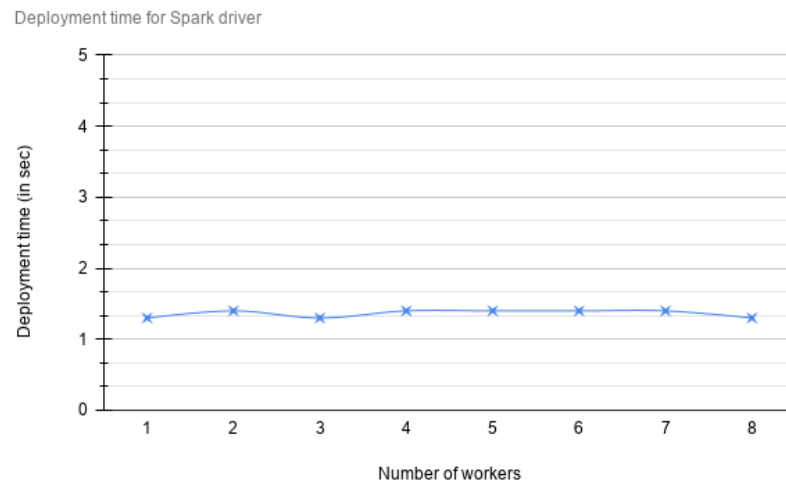


Fig. 5.3 Deployment time of data privacy event captor on the Spark **master** over the number of workers for **1M** data points

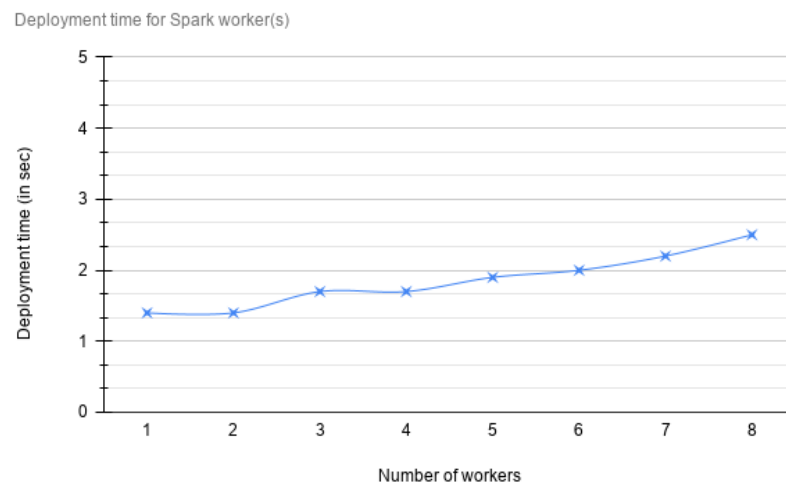


Fig. 5.4 Deployment time of data privacy event captor on the Spark **workers** over the number of workers for **1M** data points

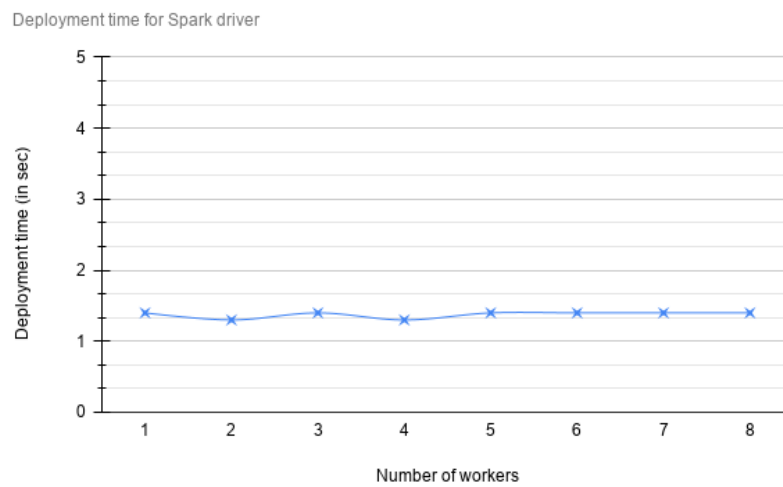


Fig. 5.5 Deployment time of data privacy event captor on the Spark **master** over the number of workers for **2M** data points

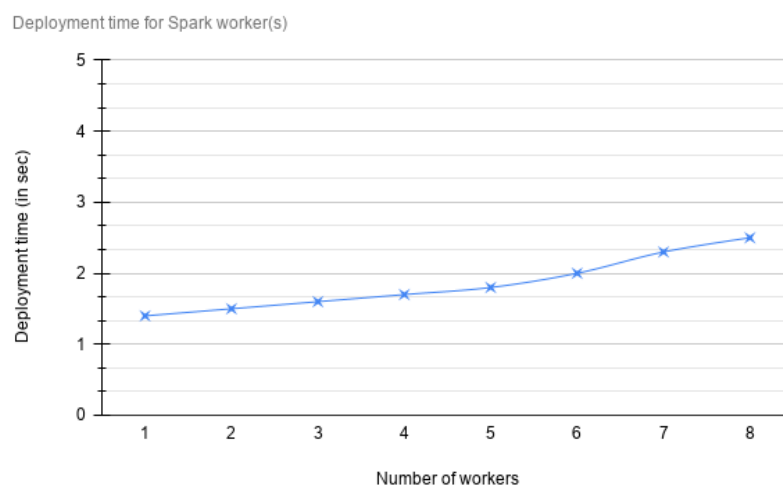


Fig. 5.6 Deployment time of data privacy event captor on the Spark **workers** over the number of workers for **2M** data points

In figures 5.7 and 5.8 below we combine all the graphs above to give an overview of the time that it takes for the data privacy event captors to be deployed on the master and worker nodes respectively. As it can be seen in the graphs, the deployment time of the event captors on the master node remains relatively the same. That is somewhat expected since there only one master node. Contrary to that, as the graphs suggest, as the number of worker increase there is an upwards trend in the deployment time due to the fact that more event captors

will have to be deployed and more coordination is required when a service gets executed. Another interesting observation is that for all the different data set sizes the data follow the same trend which implies that the size of the data do not affect the deployment time. This is expected since the deployment of the event captors is a totally separate operation that takes place before the data gets processed.

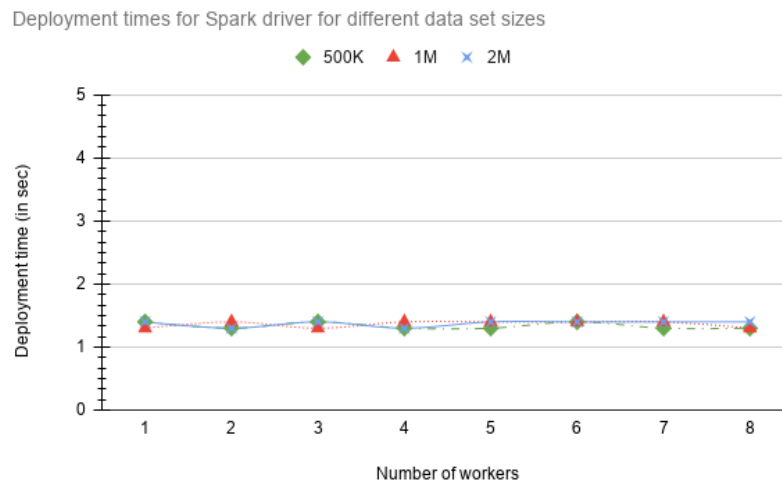


Fig. 5.7 Overlay graph for the deployment of the data privacy event captor on the Spark **master** for different data sets size

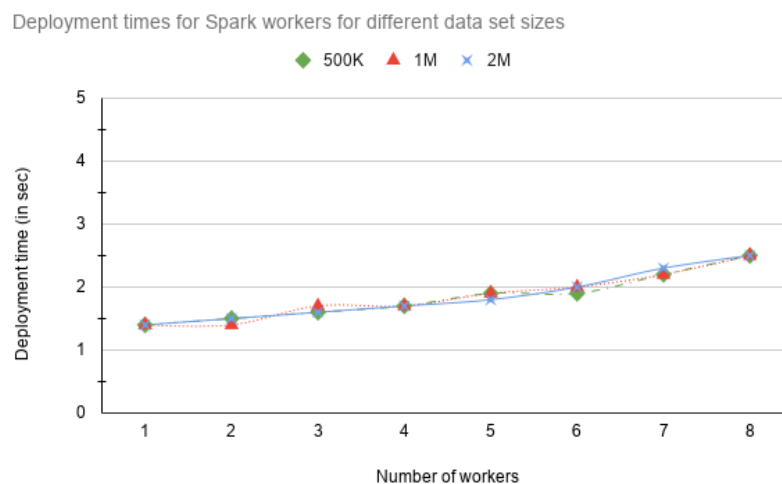


Fig. 5.8 Overlay graph for the deployment of the data privacy event captor on the Spark **workers** for different data sets size

Data integrity event captors

In this section we present our finding with regards to the overhead imposed to the monitoring activity when the monitoring of data integrity is enabled. More specifically, we have plotted the deployment time that it takes for the data integrity event captors to be deployed over different cluster configurations where a different number of Spark workers are available. The experiments were conducted for 500K, 1M and 2M data points to facilitate the evaluation of the effect of the data set size on the overall deployment time. In more detail, in figure 5.9 and figure 5.10 we present the data for 500K data points, in figure 5.11 and figure 5.12 we present the data for 1M data points and finally in figure 5.13 and figure 5.14 we present the data for 2M data points. Note that, for completeness, we treat collect metrics separately for the master and the worker nodes.

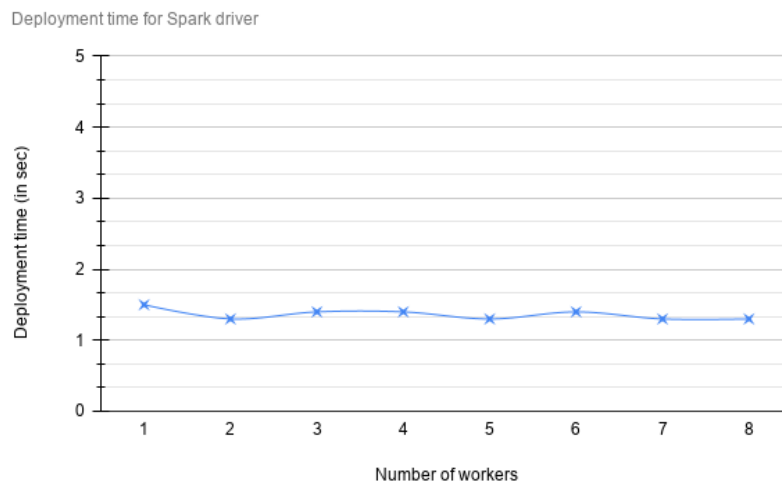


Fig. 5.9 Deployment time of data integrity event captor on the Spark master over the number of workers for **500K** data points

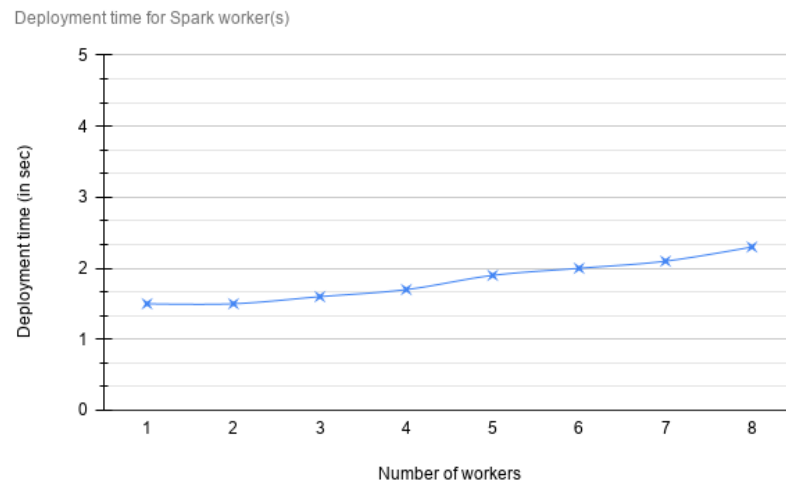


Fig. 5.10 Deployment time of data integrity event captor on the Spark workers over the number of workers for **500K** data points

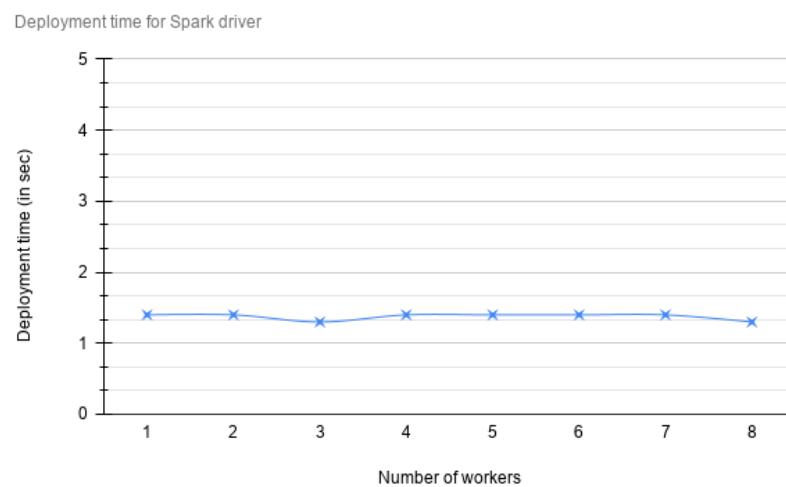


Fig. 5.11 Deployment time of data integrity event captor on the Spark master over the number of workers for **1M** data points

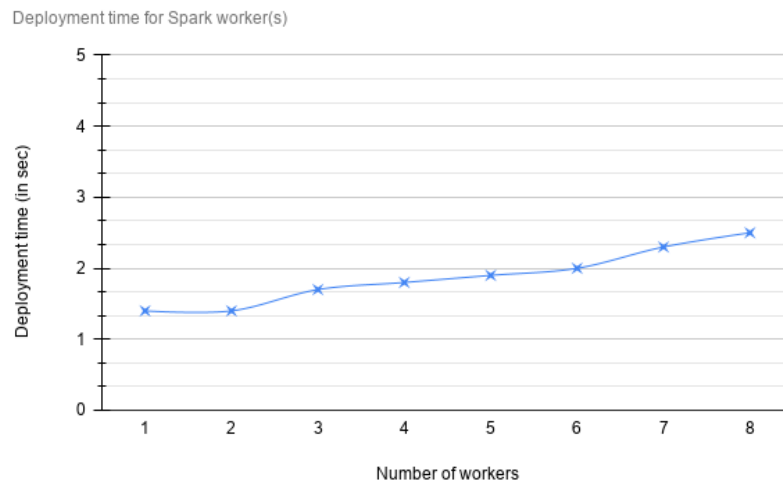


Fig. 5.12 Deployment time of data integrity event captor on the Spark workers over the number of workers for **1M** data points

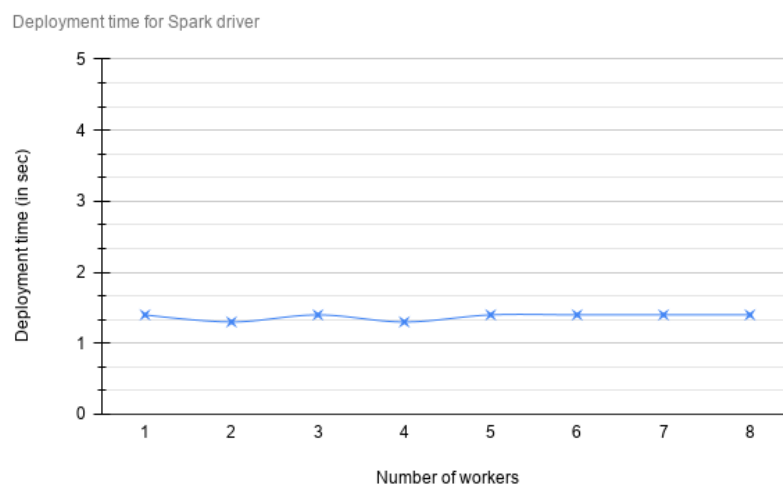


Fig. 5.13 Deployment time of data integrity event captor on the Spark master over the number of workers for **1M** data points

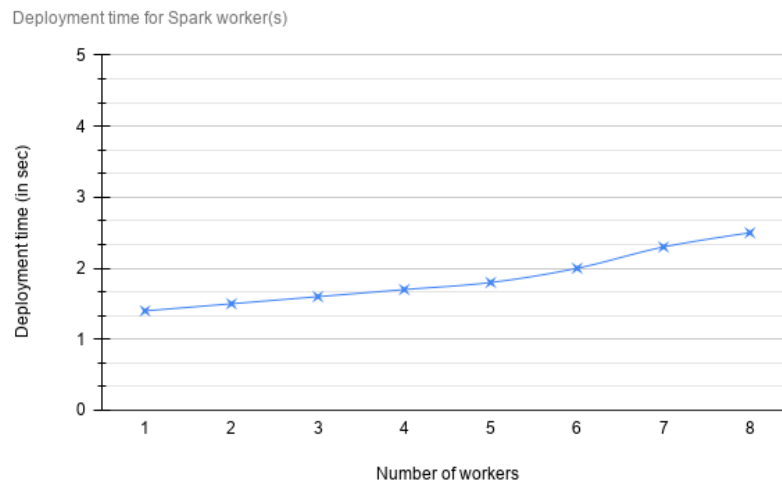


Fig. 5.14 Deployment time of data integrity event captor on the Spark master over the number of workers for **2M** data points

In figures 5.15 and 5.16 below we combine all the graphs above to give an overview of the time that it takes for the data integrity event captors to be deployed on the master and worker nodes respectively. As it can be seen in the graphs, the deployment time of the event captors on the master node remains relatively the same. That is somewhat expected since there is only one master node. Contrary to that, as the graphs suggest, as the number of worker increase there is an upwards trend in the deployment time due to the fact that more event captors will have to be deployed and more coordination is required when a service gets executed. Another interesting observation is that for all the different data set sizes the data follow the same trend which implies that the size of the data do not affect the deployment time. This is expected since the deployment of the event captors is a totally separate operation that takes place before the data gets processed.

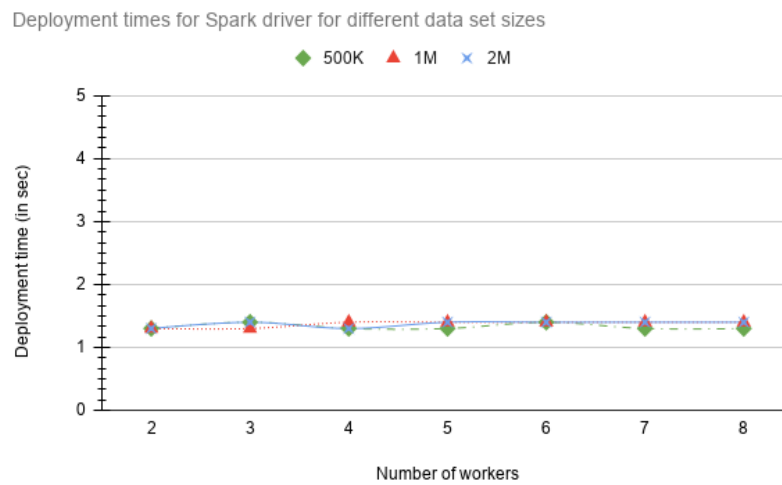


Fig. 5.15 Overlay graph for the deployment of the data integrity event captor on the Spark **workers** for different data sets sizes

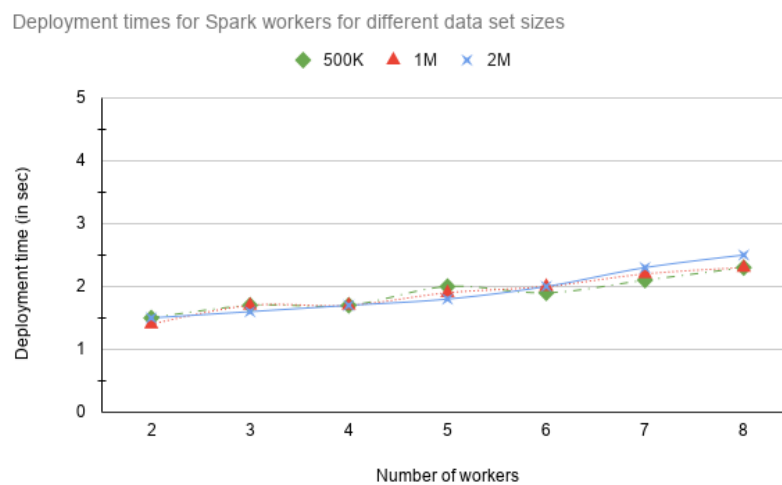


Fig. 5.16 Overlay graph for the deployment of the data integrity event captor on the Spark **workers** for different data sets sizes

Response time event captors

In this section we present our finding with regards to the overhead imposed to the monitoring activity when the monitoring of the location of execution is enabled. More specifically, we have plotted the deployment time that it takes for the data privacy event captors to be deployed over different cluster configurations where a different number of Spark workers are

available. The experiments were conducted for 500K, 1M and 2M data points to facilitate the evaluation of the effect of the data set size on the overall deployment time. In more detail, in figure 5.17 and figure 5.18 we present the data for 500K data points, in figure 5.19 and figure 5.20 we present the data for 1M data points and finally in figure 5.21 and figure 5.22 we present the data for 2M data points. Note that, for completeness, we treat collect metrics separately for the master and the worker nodes.

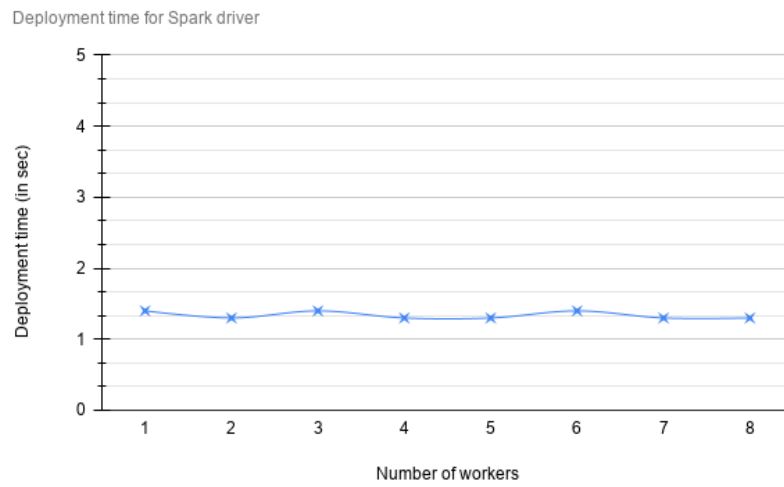


Fig. 5.17 Deployment time of data availability event captor on the Spark **master** over the number of workers for **500K** data points

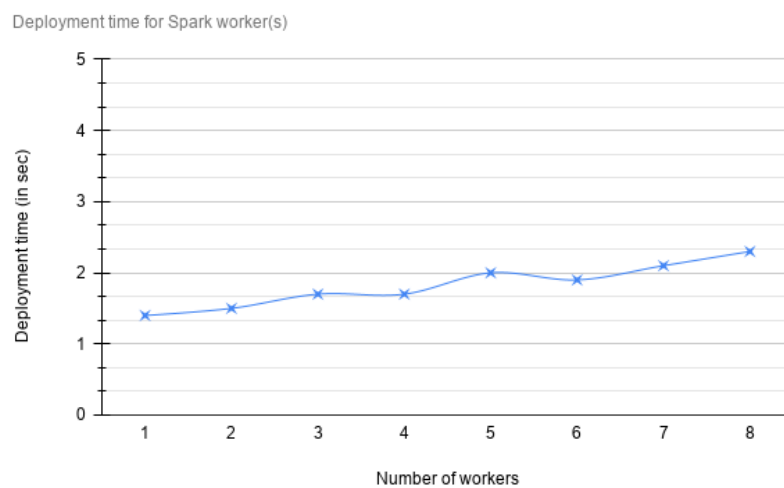


Fig. 5.18 Deployment time of data availability event captor on the Spark **workers** over the number of workers for **500K** data points

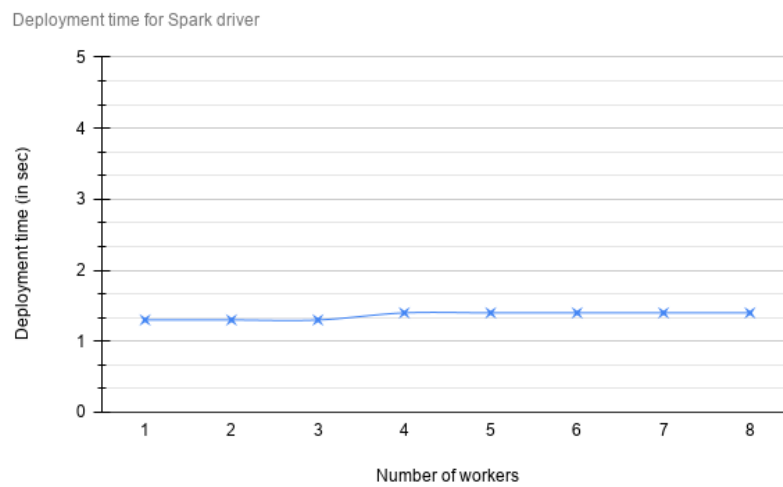


Fig. 5.19 Deployment time of data availability event captor on the Spark **master** over the number of workers for **1M** data points

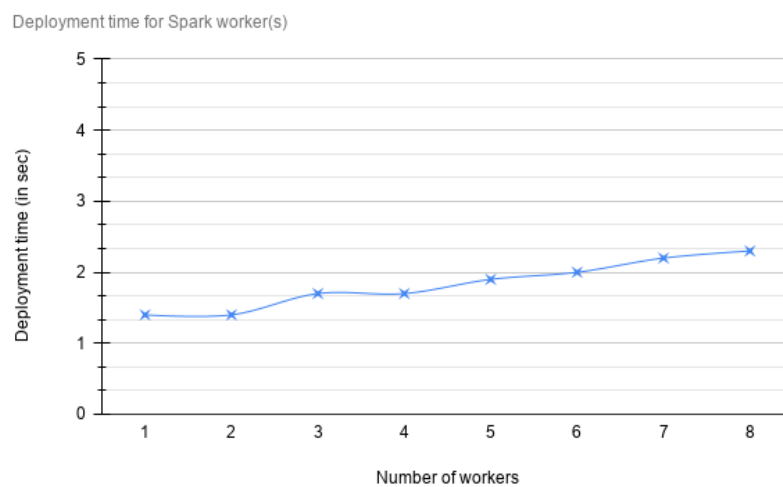


Fig. 5.20 Deployment time of data availability event captor on the Spark **workers** over the number of workers for **1M** data points

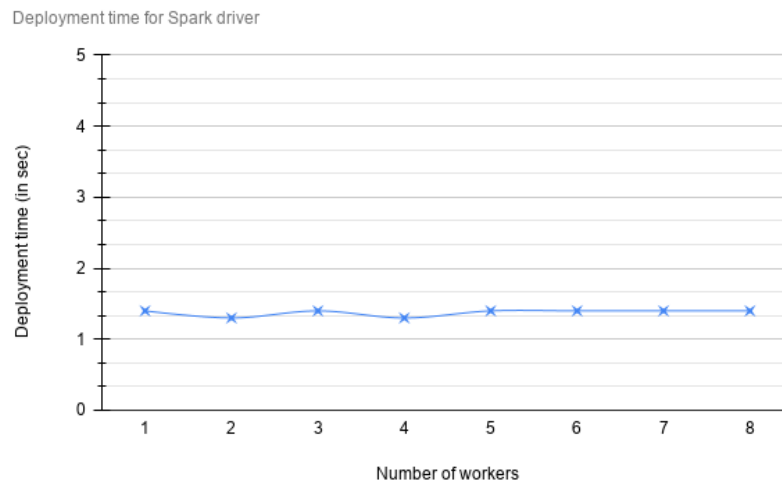


Fig. 5.21 Deployment time of data availability event captor on the Spark **master** over the number of workers for **2M** data points

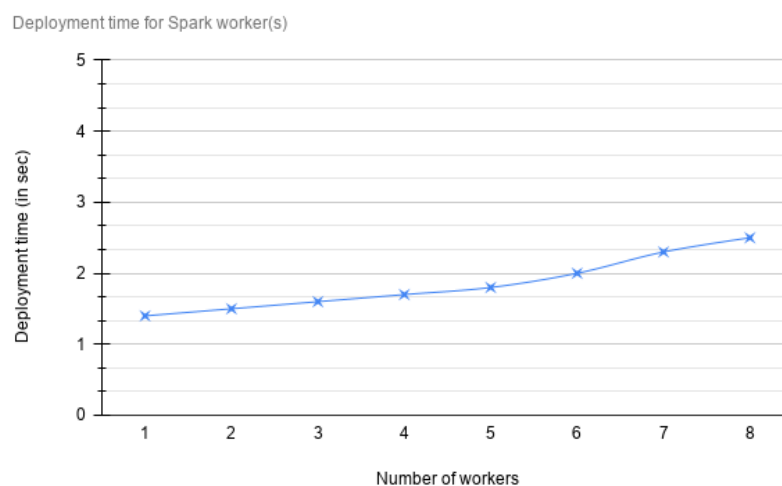


Fig. 5.22 Deployment time of data availability event captor on the Spark **workers** over the number of workers for **2M** data points

In figures 5.23 and 5.24 below we combine all the graphs above to give an overview of the time that it takes for the data availability event captors to be deployed on the master and worker nodes respectively. As it can be seen in the graphs, the deployment time of the event captors on the master node remains relatively the same. That is somewhat expected since there only one master node. Contrary to that, as the graphs suggest, as the number of worker increase there is an upwards trend in the deployment time due to the fact that more

event captors will have to be deployed and more coordination is required when a service gets executed. Another interesting observation is that for all the different data set sizes the data follow the same trend which implies that the size of the data do not affect the deployment time. This is expected since the deployment of the event captors is a totally separate operation that takes place before the data gets processed.

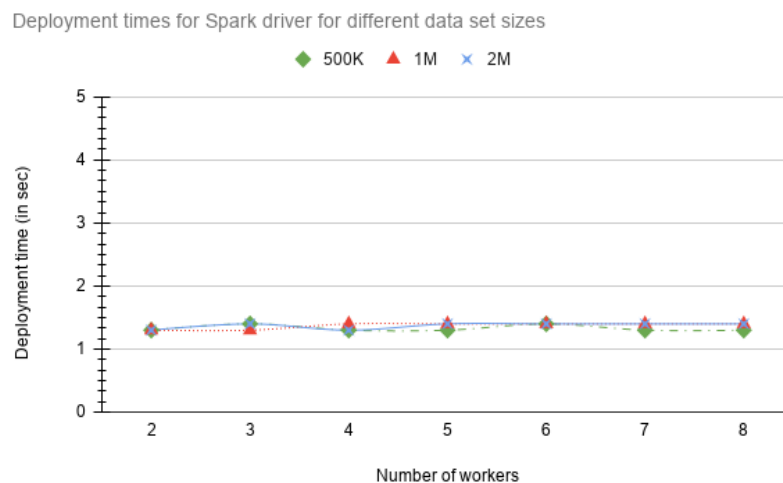


Fig. 5.23 Overlay graph for the deployment of the data availability event captor on the Spark **master** for different data sets sizes

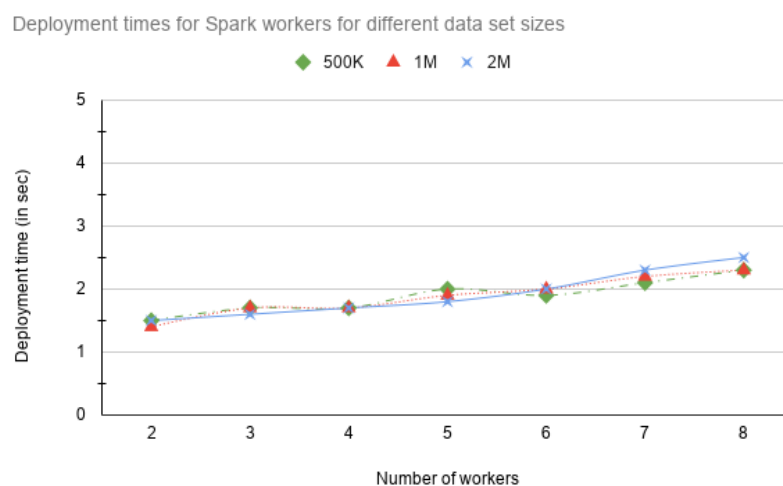


Fig. 5.24 Overlay graph for the deployment of the data availability event captor on the Spark **workers** for different data sets sizes

5.2.2 Event captor execution overhead

To be able to make a comparative analysis of the execution time for the services of our use-case, for each service we need to take a baseline with regards to execution times without the event capturing being enabled. Then, we need to take the execution times for every service with the event capturing enabled and compare them with the baseline. In all the experiments that we run to evaluate the overhead imposed as a result of the monitoring activity, we have run the services 1000 times and we averaged the values that we were able to collect. This help us to get a more realistic view of the system and eliminate possible one-off outliers.

Execution overhead for the data privacy event captors

In the case of the data privacy, the security property that is monitored is the location of execution of the service operations. In figures 5.25 all the way up to 5.32, we present the overlay graphs between the execution time with no monitoring and with monitoring enabled over different data set sizes, for multiple cluster configurations that contain from 1 up to 8 workers.

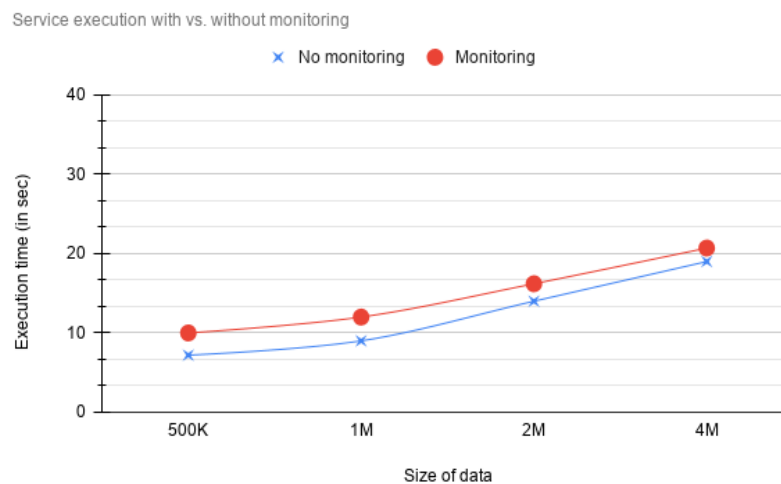


Fig. 5.25 Service execution time with and without **data privacy** monitoring on a cluster with **1 worker node** for multiple data set sizes

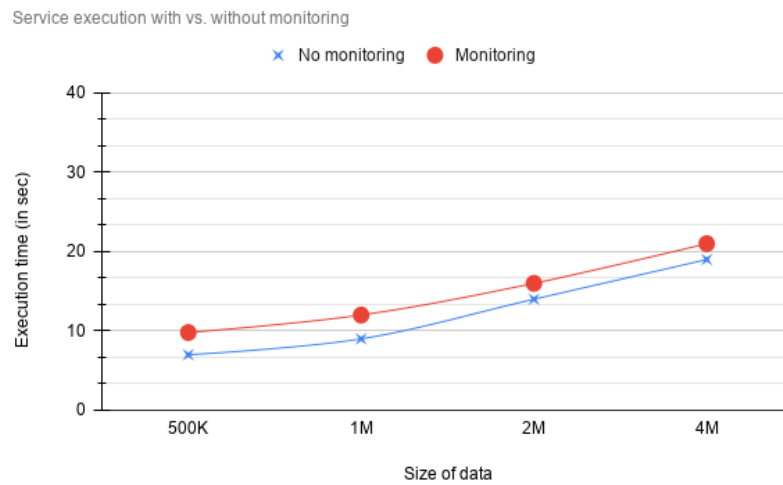


Fig. 5.26 Service execution time with and without monitoring **data privacy** on a cluster with **2 worker nodes** for multiple data set sizes

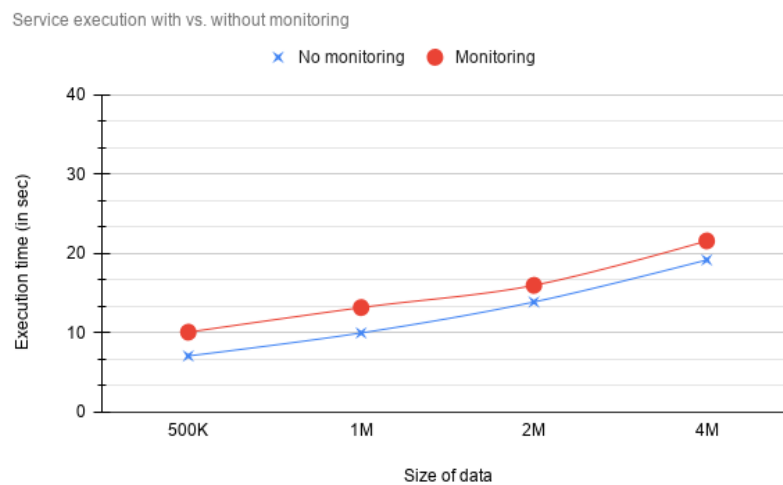


Fig. 5.27 Service execution time with and without **data privacy** monitoring on a cluster with **3 worker nodes** for multiple data set sizes

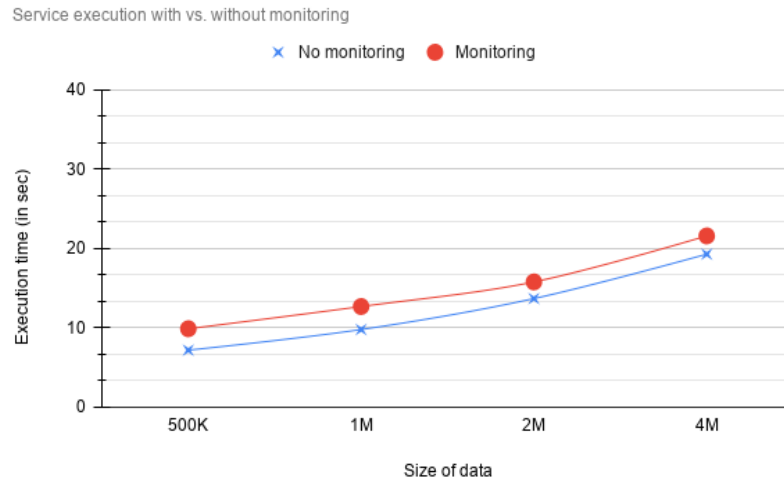


Fig. 5.28 Service execution time with and without **data privacy** monitoring on a cluster with **4 worker nodes** for multiple data set sizes

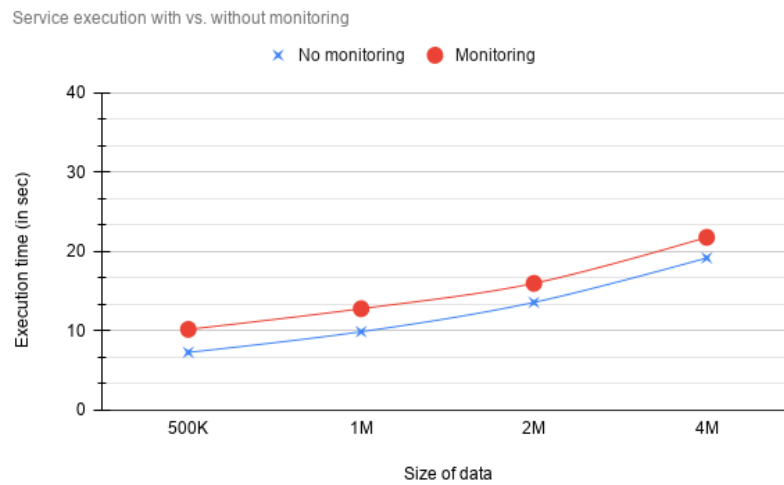


Fig. 5.29 Service execution time with and without **data privacy** monitoring on a cluster with **5 worker nodes** for multiple data set sizes

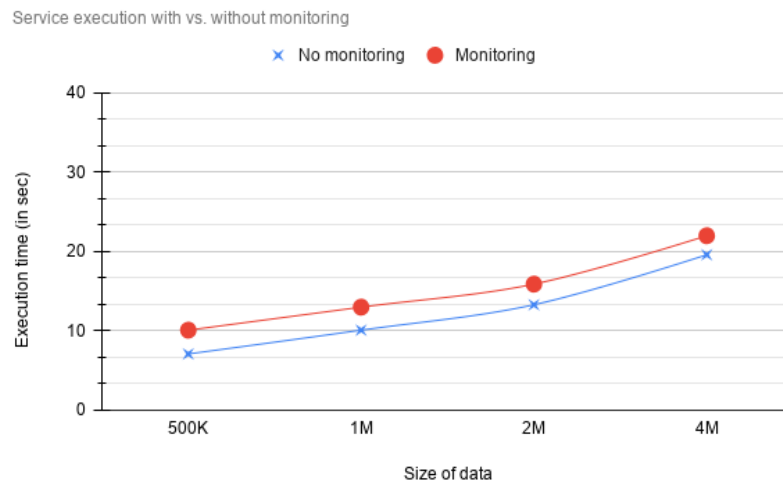


Fig. 5.30 Service execution time with and without **data privacy** monitoring on a cluster with **6 worker nodes** for multiple data set sizes

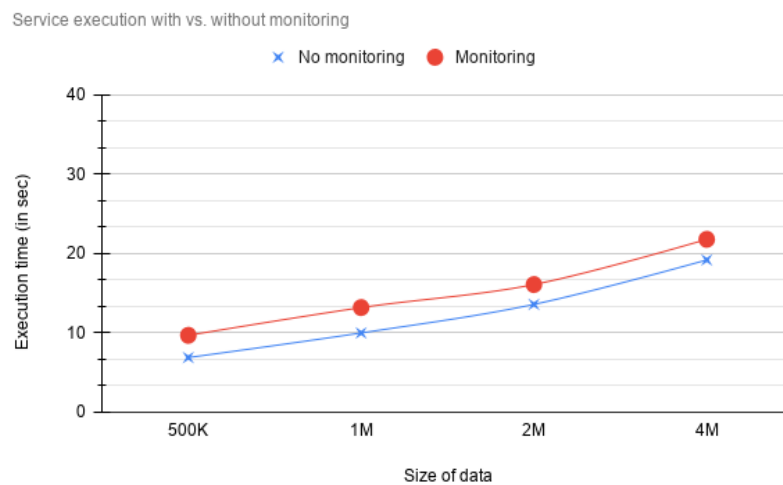


Fig. 5.31 Service execution time with and without **data privacy** monitoring on a cluster with **7 worker nodes** for multiple data set sizes

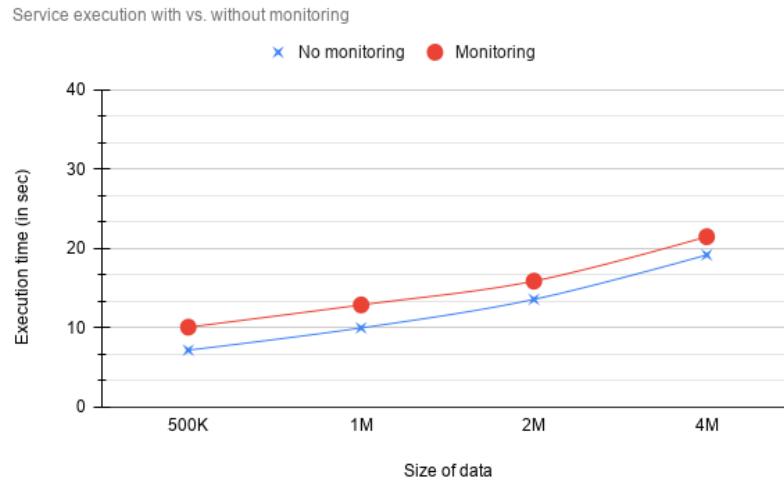


Fig. 5.32 Service execution time with and without **data privacy** monitoring on a cluster with **8 worker nodes** for multiple data set sizes

As it is shown in the graphs above, regardless of the cluster configuration, monitoring data privacy will impose a penalty on the overall service execution. We present the measurements that we have been able to collect for different data set sizes and different cluster configuration in table 5.2.

Data size	1	2	3	4	5	6	7	8	Overhead(%)
500K	38.89%	40.00%	42.25%	37.50%	39.73%	42.25%	40.58%	40.28%	40.18%
1M	33.33%	33.33%	32.00%	29.59%	29.29%	28.71%	32.00%	29.00%	30.91%
2M	15.71%	14.29%	15.11%	15.33%	17.65%	19.55%	18.38%	16.91%	16.62%
4M	8.95%	10.53%	12.50%	11.92%	13.54%	12.24%	13.54%	11.98%	11.90%

Table 5.2 Average overhead for monitoring data privacy for different data set sizes and number of worker nodes

The data suggests that the number of the worker nodes do not affect the overhead imposed for the proposed data set sizes as there are minor fluctuations when running the service for the same data set. However, the overall overhead expressed as a percentage of the total time it would the service to run without the data privacy monitoring enabled, seems to decrease as the data set size increases. Because of the nature of the data privacy event captors that emit events for each data partition that is being processed, the overhead that is imposed does

not have a linear relationship with to the size of the input data. In particular, as the service needs to handle more data, the time it take to process each partition is longer than the time it take the event captor to collect and emit the events related to those partitions respectively. That being said, more data to be processed means more partitions, and since the overhead is expressed in relation to the total time of execution without monitoring, the overhead goes down as the size of the input data set becomes larger. A visual representation of the data shown in the table above can be seen in figure 5.33.

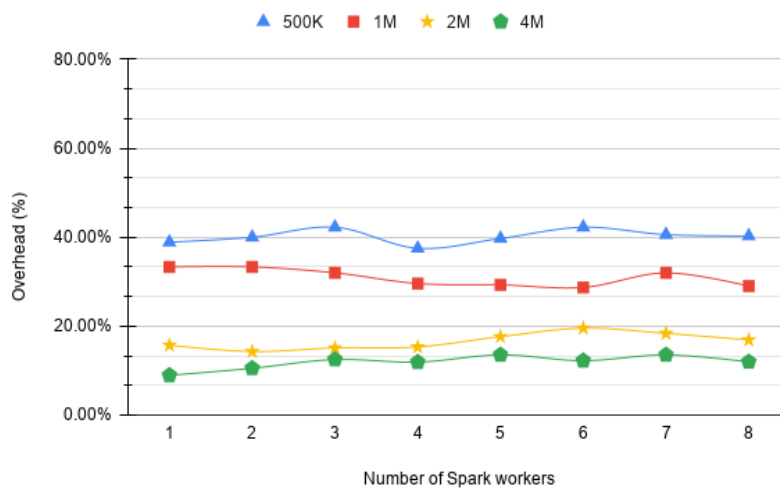


Fig. 5.33 Overlay graph for the service execution overhead of the data privacy event captor for different data sets on clusters with different number of workers

Overhead for the data availability event captors

In the case of the data availability, the security property that is monitored is the response time of the service operations. In figures 5.34 all the way up to 5.42, we present the overlay graphs between the execution time with no monitoring and with monitoring enabled over different data set sizes, for multiple cluster configurations that contain from 1 up to 8 workers.

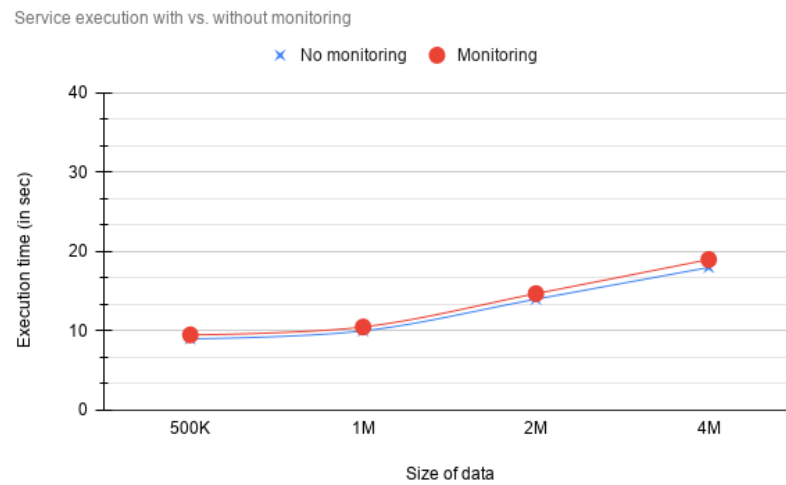


Fig. 5.34 Service execution time with and without data availability monitoring on a cluster with 1 worker node for multiple data set sizes

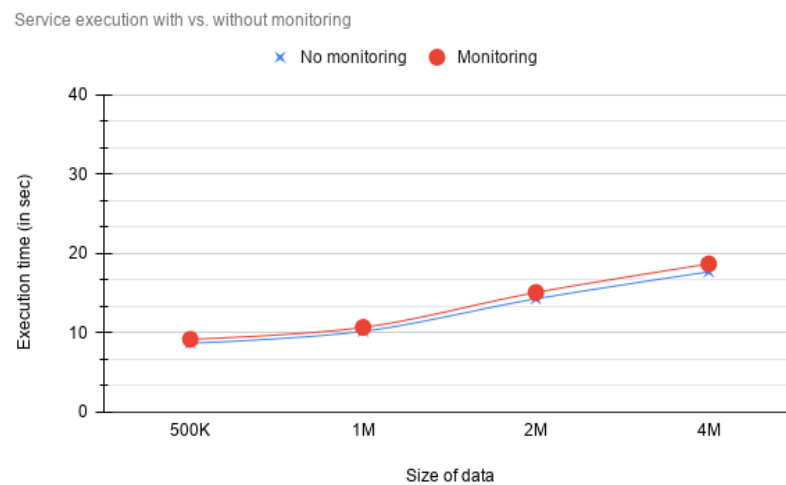


Fig. 5.35 Service execution time with and without data availability monitoring on a cluster with 2 worker nodes for multiple data set sizes

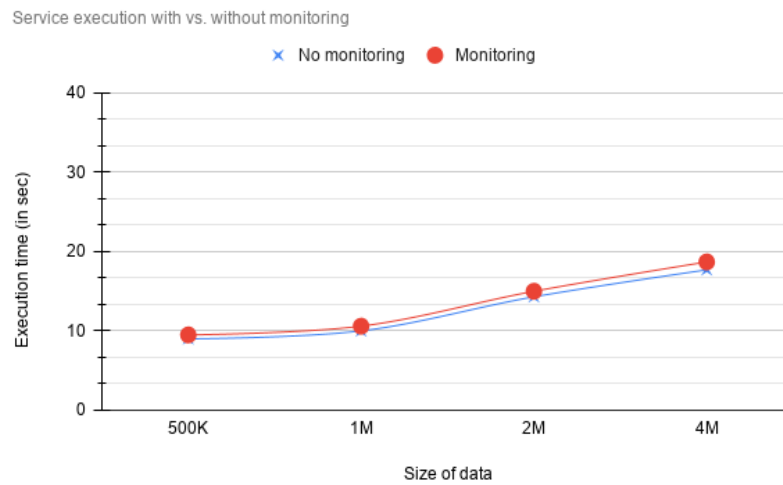


Fig. 5.36 Service execution time with and without data availability monitoring on a cluster with 3 worker nodes for multiple data set sizes

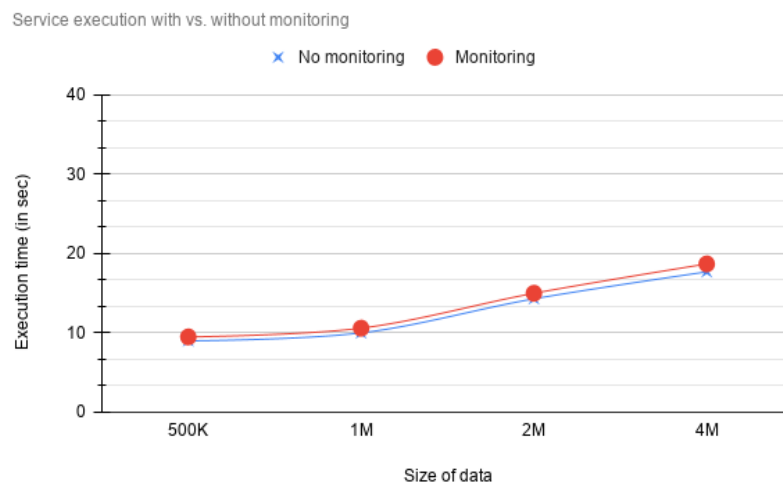


Fig. 5.37 Service execution time with and without data availability monitoring on a cluster with 3 worker nodes for multiple data set sizes

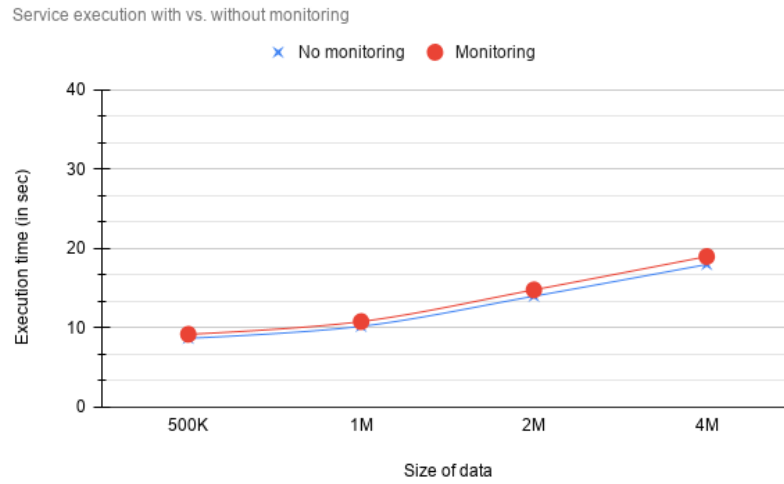


Fig. 5.38 Service execution time with and without data availability monitoring on a cluster with 4 worker nodes for multiple data set sizes

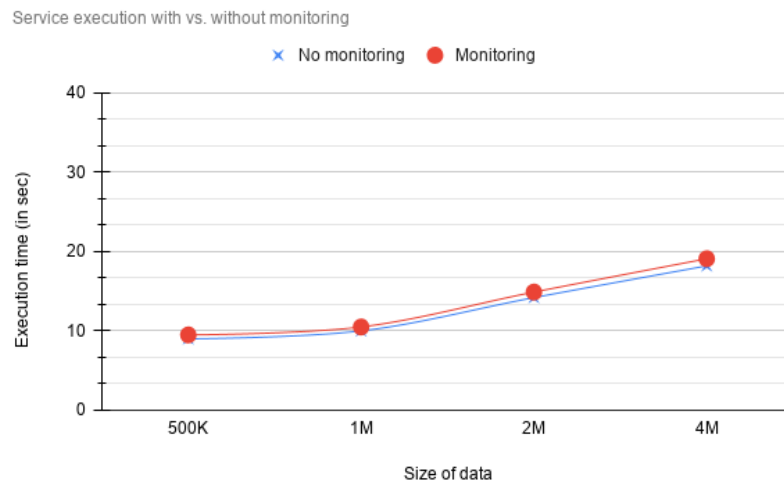


Fig. 5.39 Service execution time with and without data availability monitoring on a cluster with 5 worker nodes for multiple data set sizes

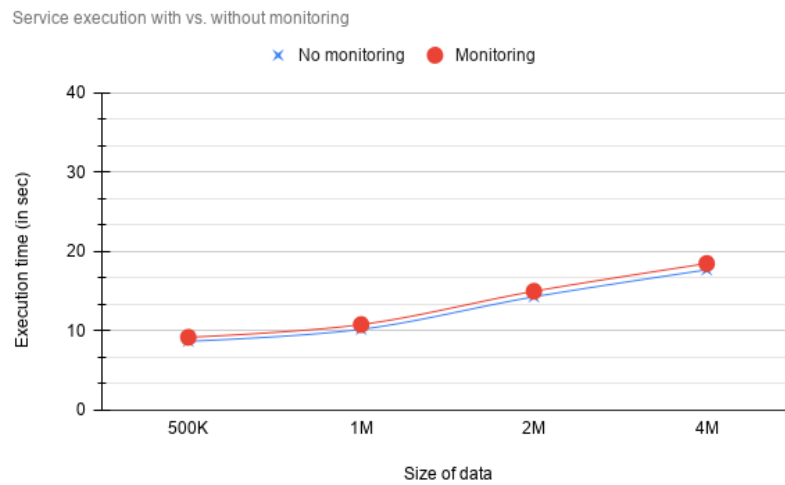


Fig. 5.40 Service execution time with and without data availability monitoring on a cluster with 6 worker nodes for multiple data set sizes

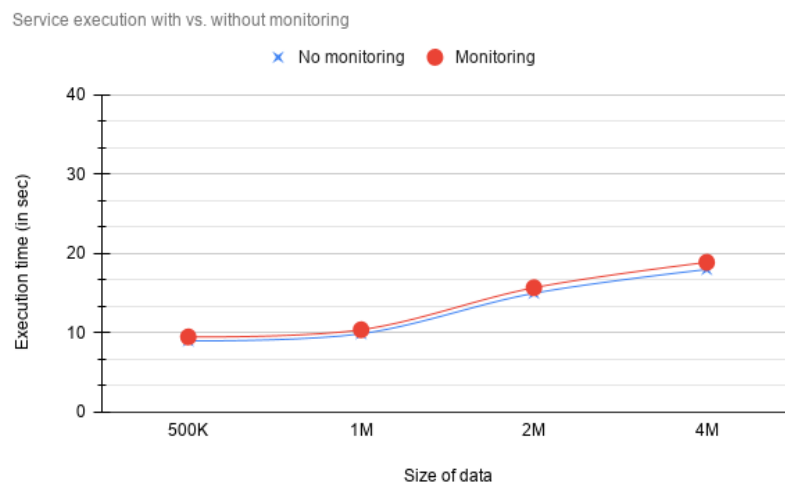


Fig. 5.41 Service execution time with and without data availability monitoring on a cluster with 7 worker nodes for multiple data set sizes

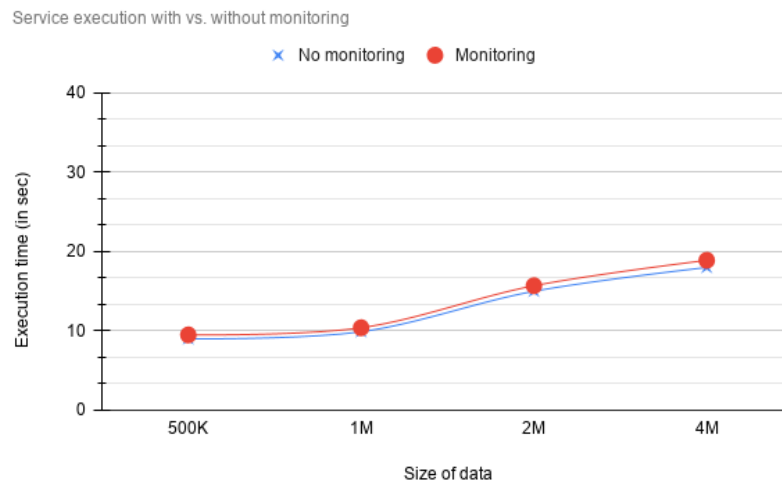


Fig. 5.42 Service execution time with and without data availability monitoring on a cluster with 8 worker nodes for multiple data set sizes

In the graphs above it is shown that monitoring data availability will impose a penalty on the overall service execution. We present the measurements that we have been able to collect for different data set sizes and different cluster configuration in table 5.3.

Data size	1	2	3	4	5	6	7	8	Overhead(%)
500K	5.56%	5.75%	5.56%	5.75%	5.56%	5.75%	5.56%	5.43%	5.61%
1M	5.00%	4.90%	6.00%	5.88%	5.00%	5.88%	5.05%	6.86%	5.57%
2M	5.00%	5.59%	4.90%	5.71%	4.93%	4.90%	4.67%	6.94%	5.33%
4M	5.56%	5.65%	5.65%	5.56%	4.95%	4.52%	5.00%	5.52%	5.30%

Table 5.3 Average overhead for monitoring data availability for different data set sizes and number of worker nodes

The data suggests that the number of the worker nodes do not affect the overhead imposed for the proposed data set sizes as there are minor fluctuations when running the service for the same data set. Similar to that, the size of the data does not seem to affect the imposed overhead as well which seems to remain the same across all metrics for all the data set sizes. This is explained by the fact that nor the number of workers neither the size of the input data are associated with what the event captor has to do in order to collect the necessary events.

This is expected since, the captor is only concerned with the collection of the start and end time of the service execution. The size of input data and the number of the workers should not influence the imposed overhead of the monitoring activity and this is in line with the measurements that have been able to collect. A visual representation of the data shown in the table above can be seen in figure 5.43.

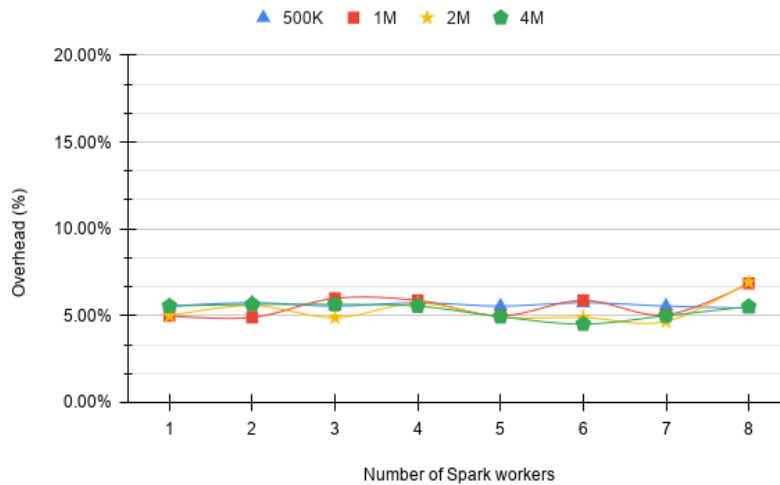


Fig. 5.43 Overlay graph for the service execution overhead of the data availability event captor for different data sets on clusters with different number of workers

Overhead for data integrity event captors

In the case of the data integrity, the security property that is monitored is the integrity of the data that the service produces during processing i.e intermediate results and the integrity of the data that the service will eventually produce when it has completed i.e. final result. A key characteristic of the data integrity monitoring activity is the requirement for the generation of the checksum values for the data of the partitions of the RDDs. The JVMs that we used in our cluster supports 3 hashing algorithms namely **MD5**, **SHA-1**, and **SHA-256**. Each one of the aforementioned algorithms produce checksums with hash values that contain different number of bits. More specifically, MD5 produces hashes with 128 bits, SHA-1 160 with bits and SHA-256 with 256 bits. The order that they are listed above is from the weakest to the strongest. A comprehensive analysis of the intricacies of the algorithms as well as a comparative comparison between them is surveyed in [83]. In their analysis, the authors, dismiss the use the MD5 algorithm as being prone to collision and not appropriate for online

systems whereas very recently Google in one of their technical blog posts [3] has been able to prove that the SHA-1 algorithm can produce collisions as well. In this thesis, apart from SHA-512, we do examine both MD5 and SHA-1 out of academic interest.

In this section we present the overlay graphs between the execution time with no monitoring and with monitoring enabled over different data set sizes, for multiple cluster configurations that contain from 1 up to 8 workers. In figures 5.44 all the way up to 5.52 we demonstrate the graphs when the MD5 algorithms is used, in figures 5.54 all the way up to 5.61 we demonstrate the graphs when the SHA-1 algorithms is used and finally In figures 5.63 all the way up to 5.70 we demonstrate the graphs when the SHA-256 algorithms is used.

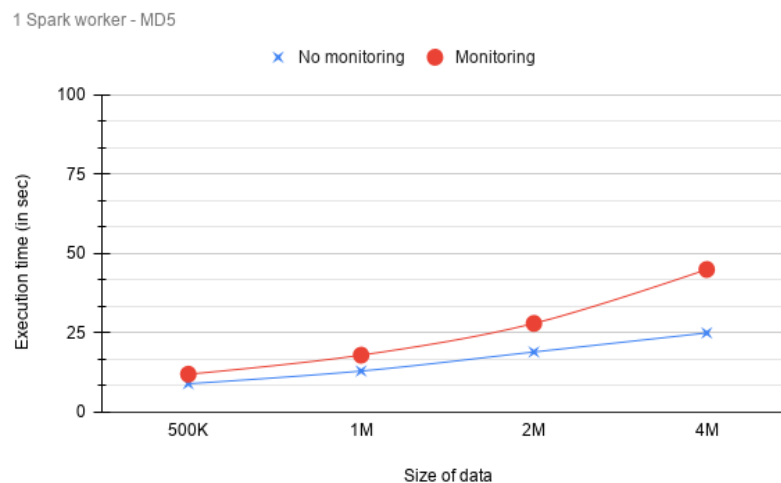


Fig. 5.44 Service execution time with and without data integrity monitoring using **MD5** on a cluster with **1 worker node** for multiple data set sizes

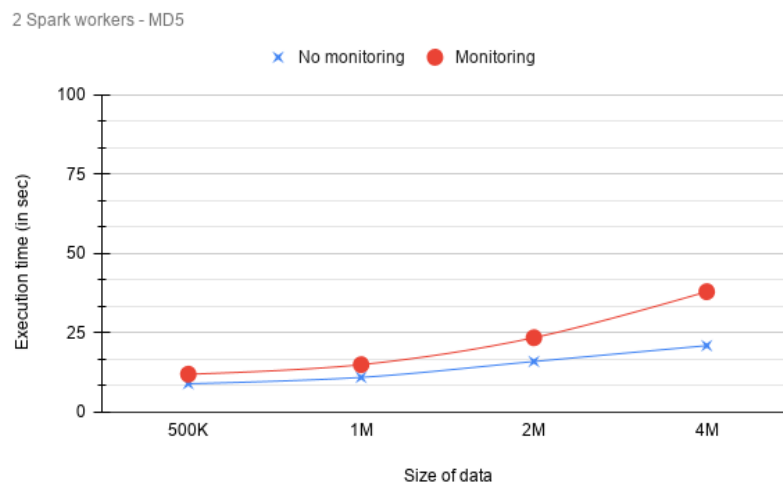


Fig. 5.45 Service execution time with and without data integrity monitoring using **MD5** on a cluster with **2 worker nodes** for multiple data set sizes

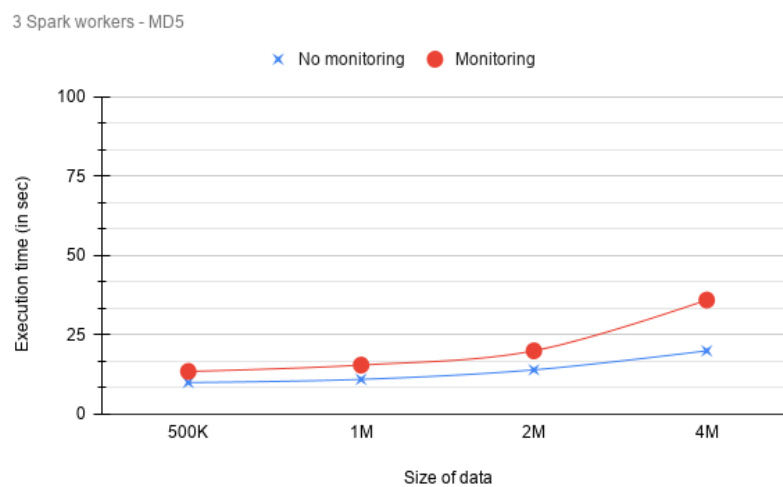


Fig. 5.46 Service execution time with and without data integrity monitoring using **MD5** on a cluster with **3 worker nodes** for multiple data set sizes

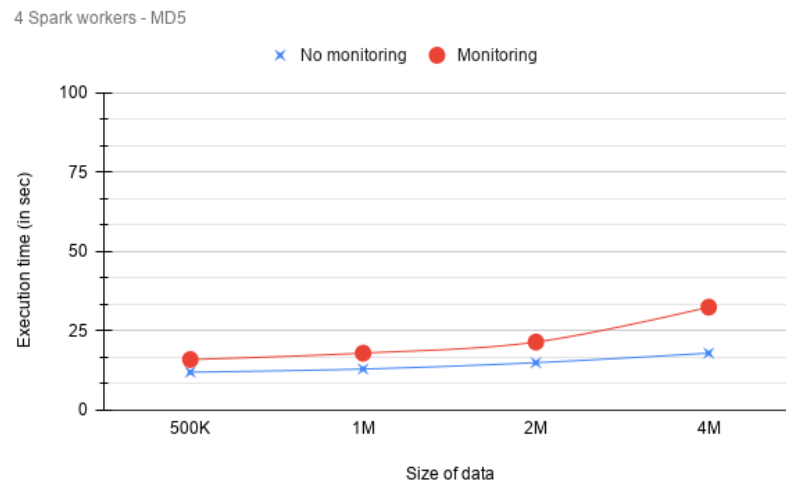


Fig. 5.47 Service execution time with and without data integrity monitoring using **MD5** on a cluster with **4 worker nodes** for multiple data set sizes

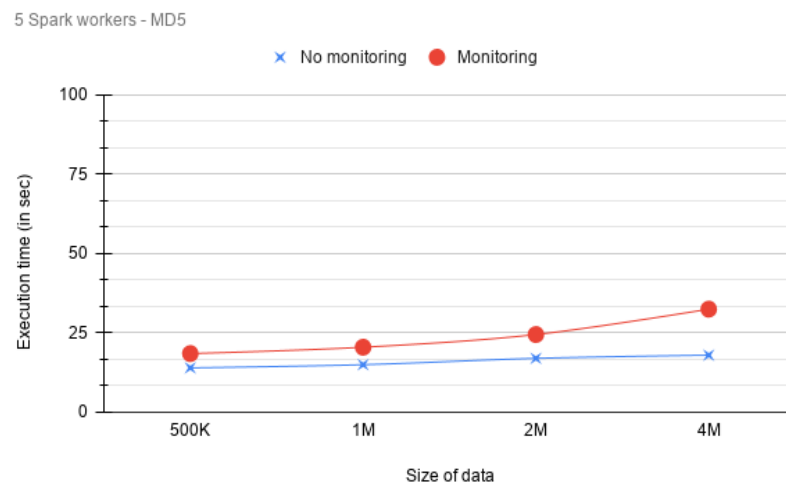


Fig. 5.48 Service execution time with and without data integrity monitoring using **MD5** on a cluster with **5 worker nodes** for multiple data set sizes

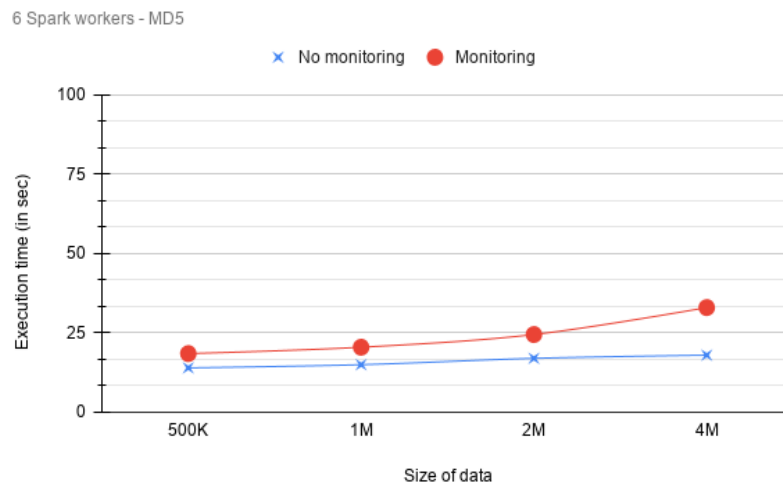


Fig. 5.49 Service execution time with and without data integrity monitoring using **MD5** on a cluster with **6 worker nodes** for multiple data set sizes

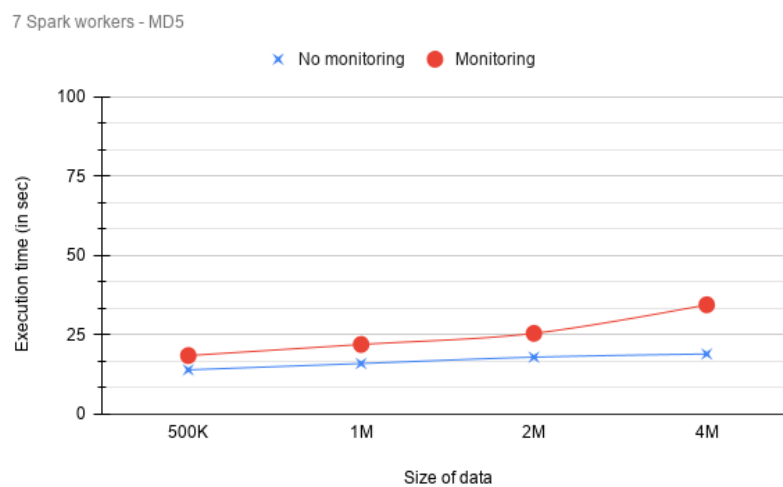


Fig. 5.50 Service execution time with and without data integrity monitoring using **MD5** on a cluster with **7 worker nodes** for multiple data set sizes

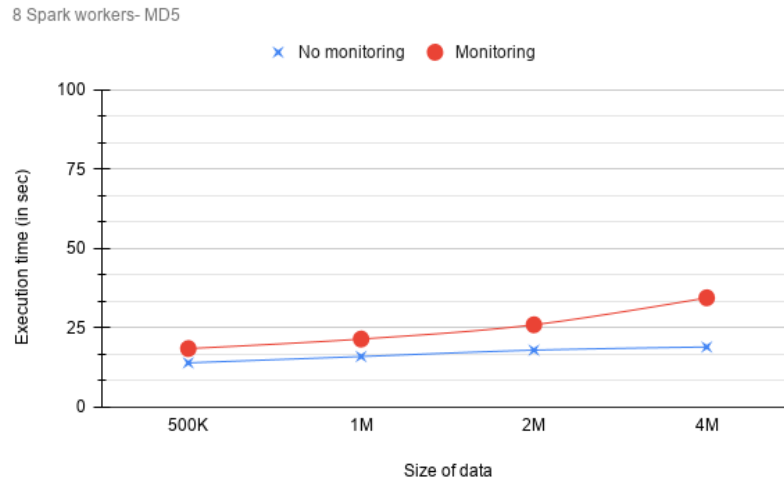


Fig. 5.51 Service execution time with and without data integrity monitoring using **MD5** on a cluster with **8 worker nodes** for multiple data set sizes

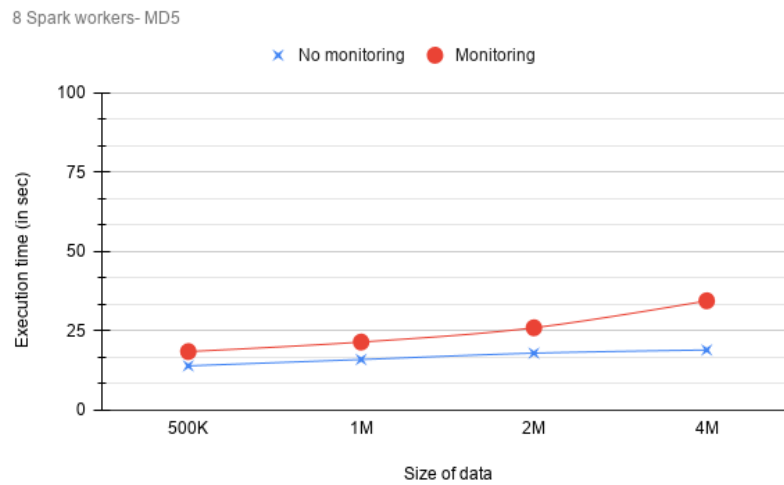


Fig. 5.52 Service execution time with and without data integrity monitoring using **MD5** on a cluster with **8 worker nodes** for multiple data set sizes

As it is shown in the graphs above, regardless of the cluster configuration, monitoring data privacy will impose a penalty on the overall service execution. We present the measurements that we have been able to collect for different data set sizes and different cluster configuration for **MD5** in table 5.4.

Data size	1	2	3	4	5	6	7	8	Overhead(%)
500K	33.33%	33.33%	35.00%	33.33%	32.14%	32.14%	32.14%	32.14%	32.95%
1M	38.46%	36.36%	40.91%	38.46%	36.67%	36.67%	37.50%	34.38%	37.43%
2M	47.37%	46.88%	42.86%	43.33%	44.12%	44.12%	41.67%	44.44%	44.35%
4M	80.00%	80.95%	80.00%	80.56%	80.56%	83.33%	81.58%	81.58%	81.07%

Table 5.4 Average overhead for monitoring data integrity using MD5 for different data set sizes and number of worker nodes

The data suggests that the number of the worker nodes do not significantly affect the overhead imposed as a result of the monitoring activity for data integrity. However, as expected, when the input data grows the overhead increases since more MD5 checksums need to be computed and emitted in the form of events in order for the monitor to reason about them. A visual representation of the data shown in the table above can be seen in figure 5.53.

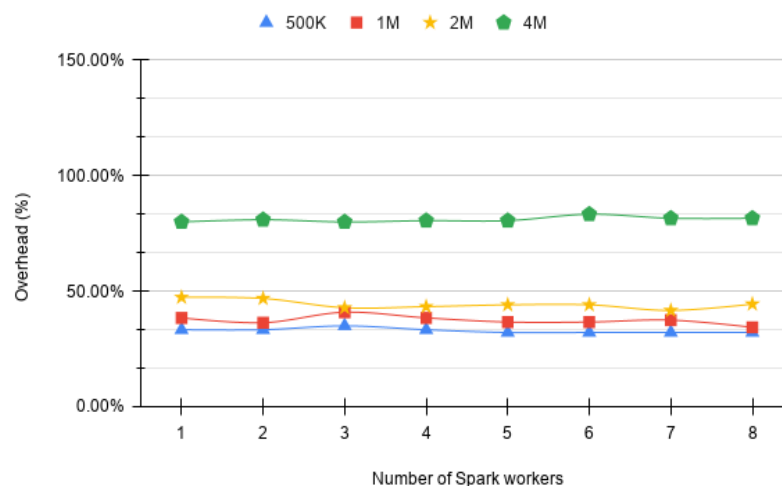


Fig. 5.53 Overlay graph of the overhead(%) over different number of workers for data integrity monitoring using MD5

Note from the figure 5.53 above that as the size of the data grows the total overhead imposed increases as well.

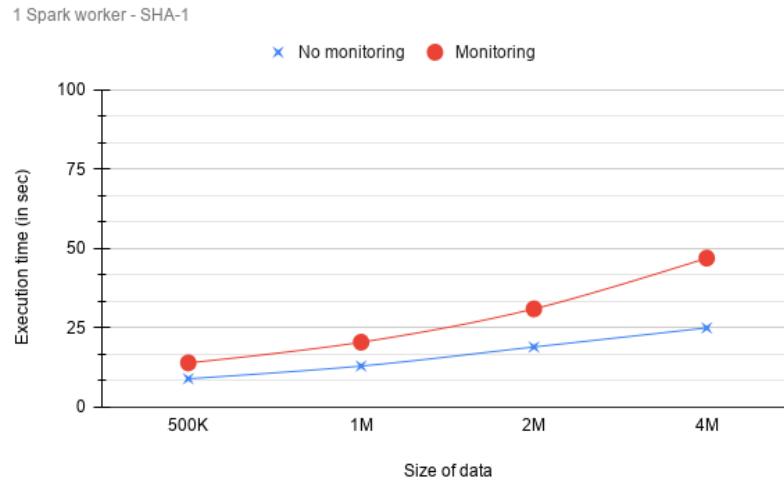


Fig. 5.54 Service execution time with and without data integrity monitoring using **SHA-1** on a cluster with **1 worker** node for multiple data set sizes

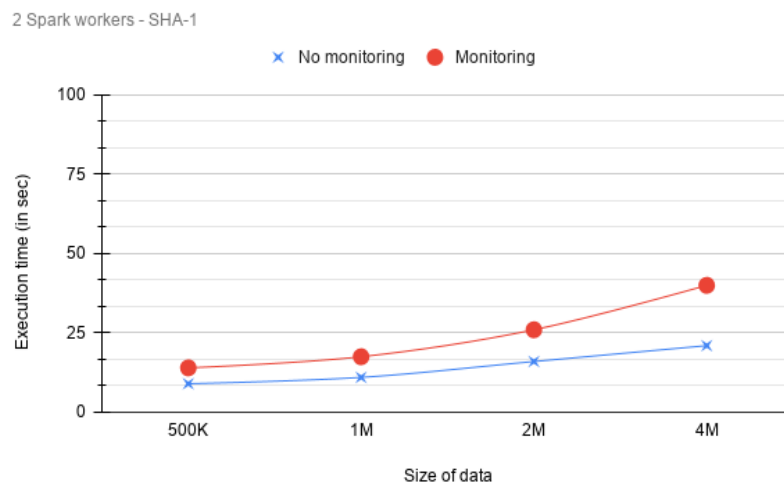


Fig. 5.55 Service execution time with and without data integrity monitoring using **SHA-1** on a cluster with **2 worker nodes** for multiple data set sizes

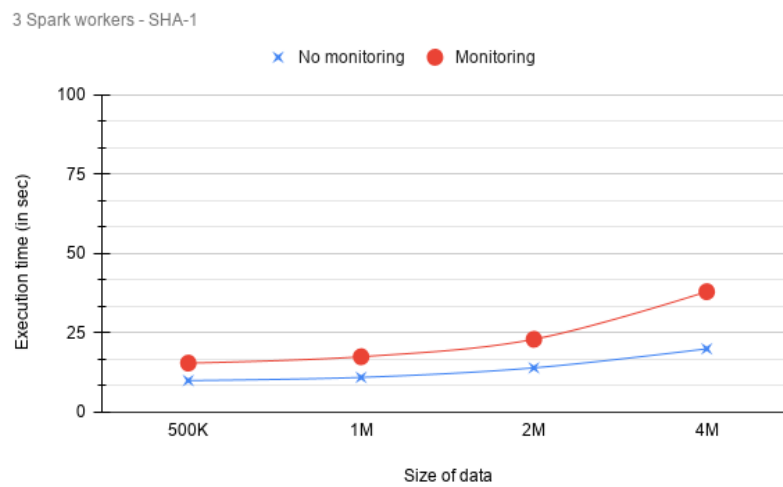


Fig. 5.56 Service execution time with and without data integrity monitoring using **SHA-1** on a cluster with **3 worker nodes** for multiple data set sizes

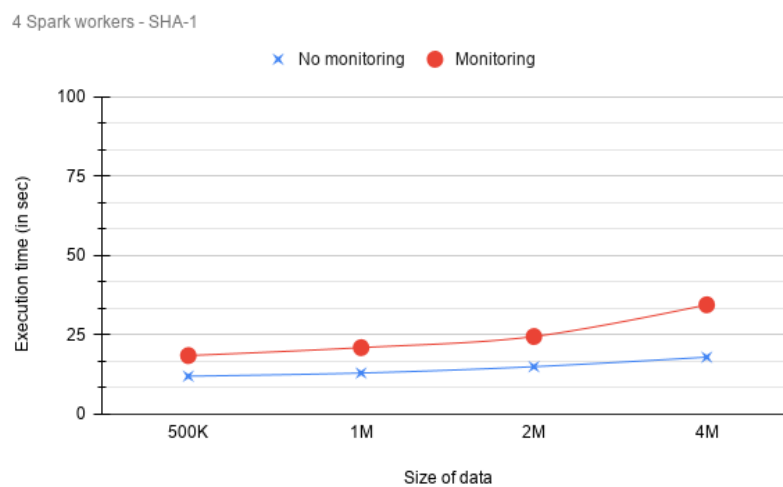


Fig. 5.57 Service execution time with and without data integrity monitoring using **SHA-1** on a cluster with **4 worker nodes** for multiple data set sizes

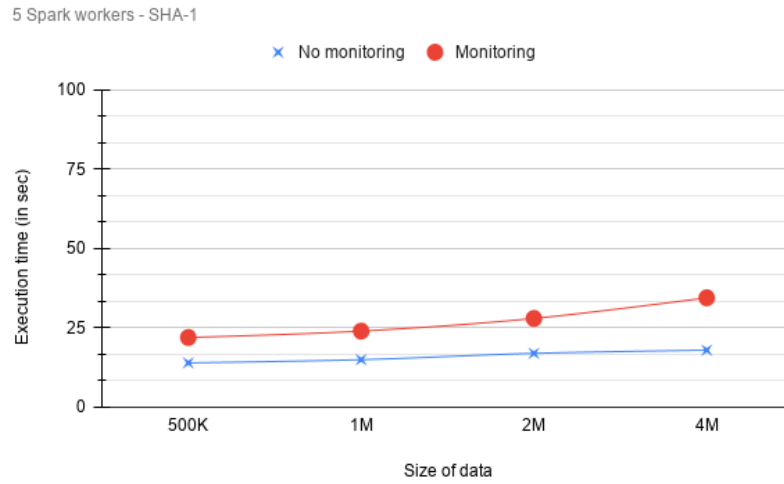


Fig. 5.58 Service execution time with and without data integrity monitoring using **SHA-1** on a cluster with **5 worker nodes** for multiple data set sizes

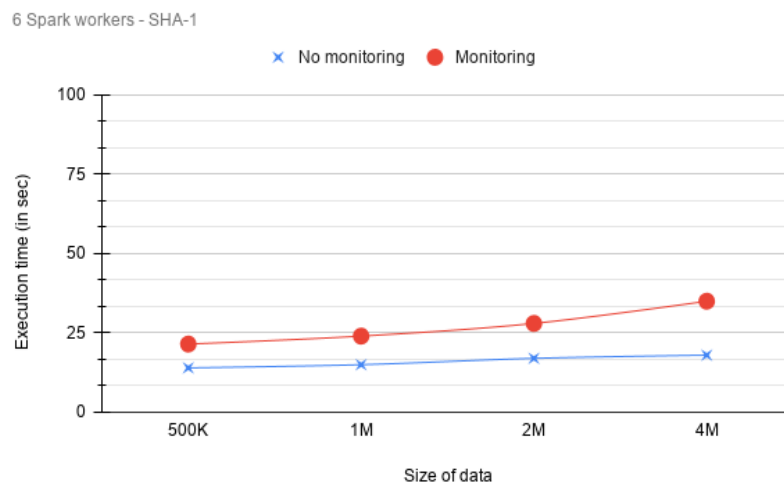


Fig. 5.59 Service execution time with and without data integrity monitoring using **SHA-1** on a cluster with **6 worker nodes** for multiple data set sizes

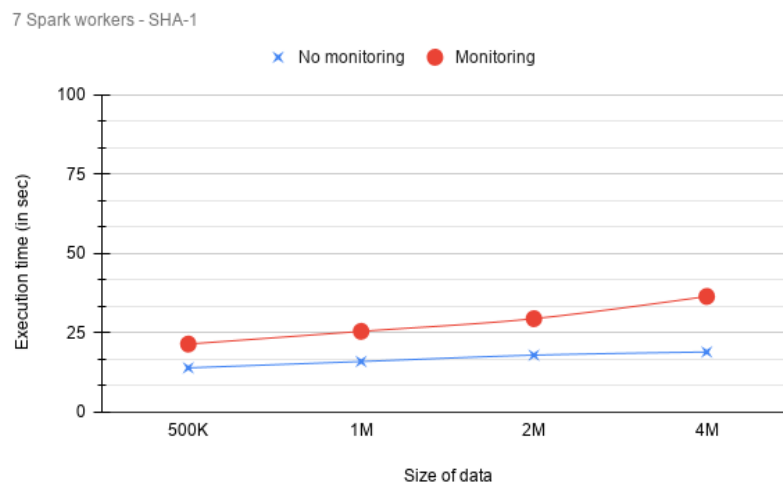


Fig. 5.60 Service execution time with and without data integrity monitoring using **SHA-1** on a cluster with **7 worker nodes** for multiple data set sizes

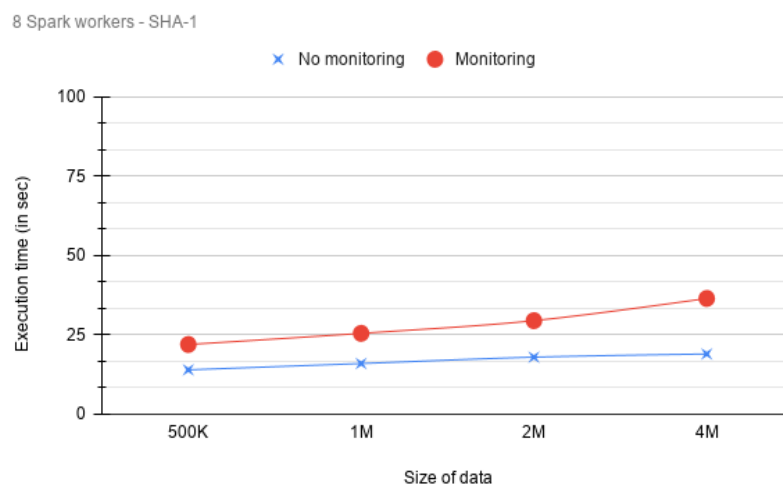


Fig. 5.61 Service execution time with and without data integrity monitoring using **SHA-1** on a cluster with **8 worker nodes** for multiple data set sizes

As it is shown in the graphs above, regardless of the cluster configuration, monitoring data privacy will impose a penalty on the overall service execution. We present the measurements that we have been able to collect for different data set sizes and different cluster configuration for **SHA-1** in table 5.5.

Data size	1	2	3	4	5	6	7	8	Overhead(%)
500K	55.56%	55.56%	55.00%	54.17%	57.14%	53.57%	53.57%	57.14%	55.21%
1M	57.69%	59.09%	59.09%	61.54%	60.00%	60.00%	59.38%	59.38%	59.52%
2M	63.16%	62.50%	64.29%	63.33%	64.71%	64.71%	63.89%	63.89%	63.81%
4M	88.00%	90.48%	90.00%	91.67%	91.67%	94.44%	92.11%	92.11%	91.31%

Table 5.5 Average overhead for monitoring data integrity using SHA-1 for different data set sizes and number of worker nodes

Similar to the data for MD5, the data suggests that the number of the worker nodes do not significantly affect the overhead imposed as a result of the monitoring activity for data integrity. However, as expected, when the input data grows the overhead increases since more SHA-1 checksums need to be computed and emitted in the form of events in order for the monitor to reason about them. A visual representation of the data shown in the table above can be seen in figure 5.62.

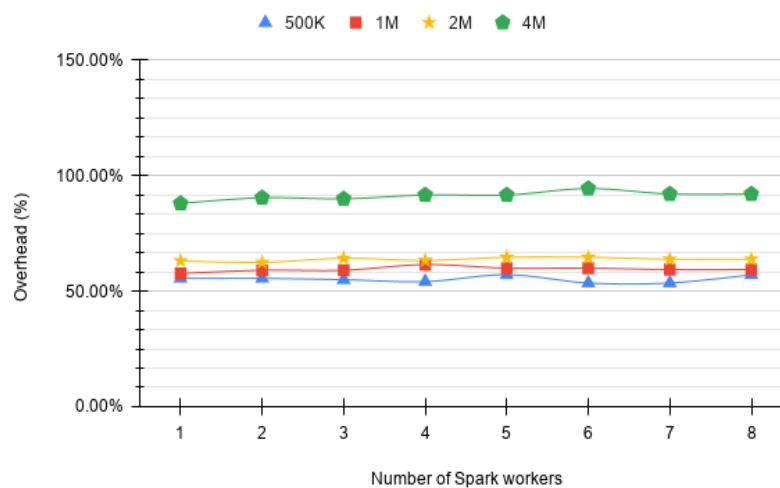


Fig. 5.62 Overlay graph of the overhead(%) over different number of workers for data integrity monitoring using SHA-1

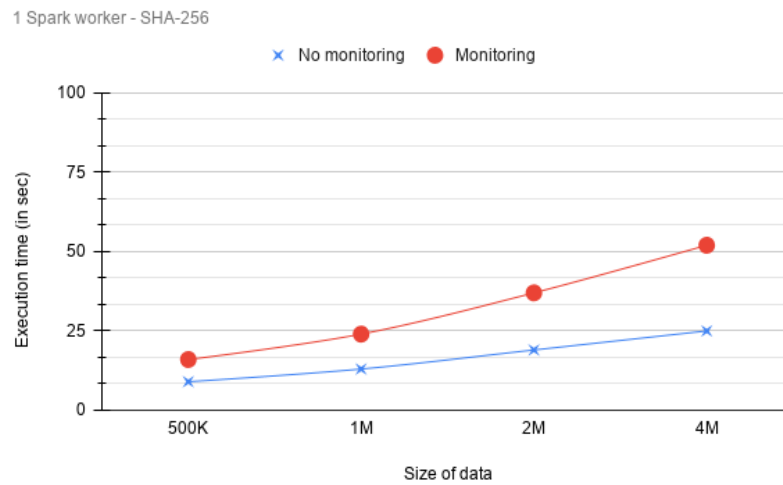


Fig. 5.63 Service execution time with and without data integrity monitoring using **SHA-256** on a cluster with **1 worker node** for multiple data set sizes

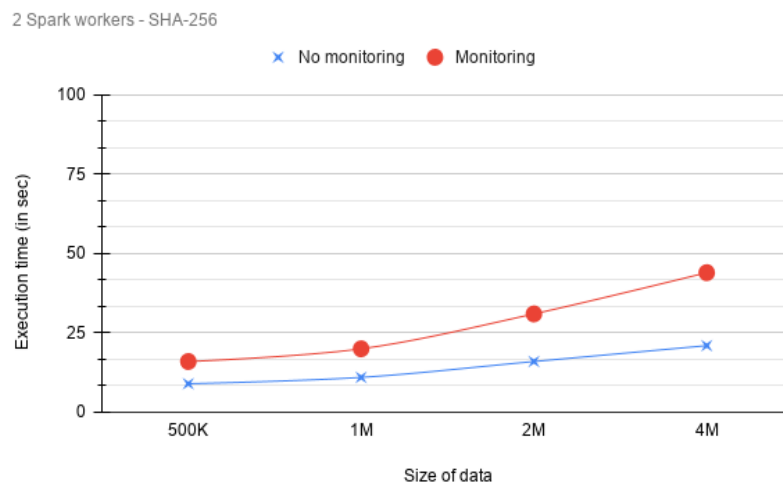


Fig. 5.64 Service execution time with and without data integrity monitoring using **SHA-256** on a cluster with **2 worker nodes** for multiple data set sizes

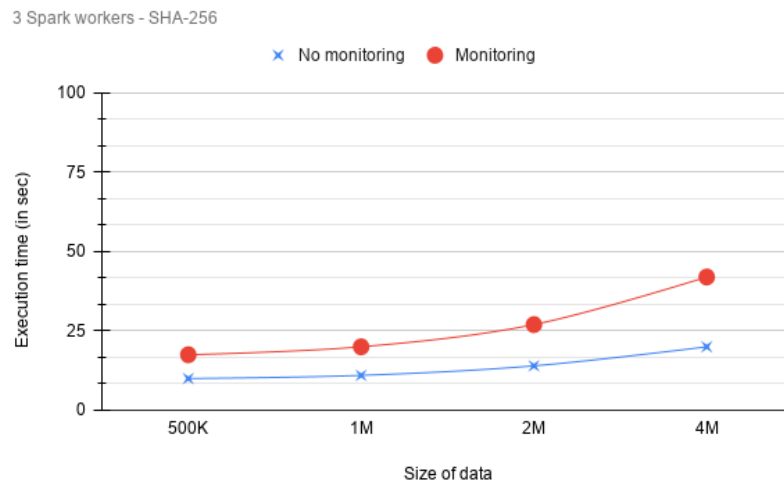


Fig. 5.65 Service execution time with and without data integrity monitoring using **SHA-256** on a cluster with **3 worker nodes** for multiple data set sizes

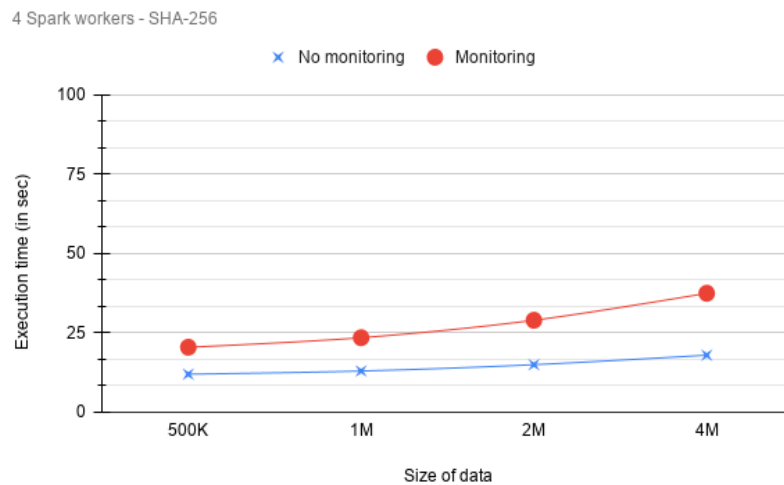


Fig. 5.66 Service execution time with and without data integrity monitoring using **SHA-256** on a cluster with **4 worker nodes** for multiple data set sizes

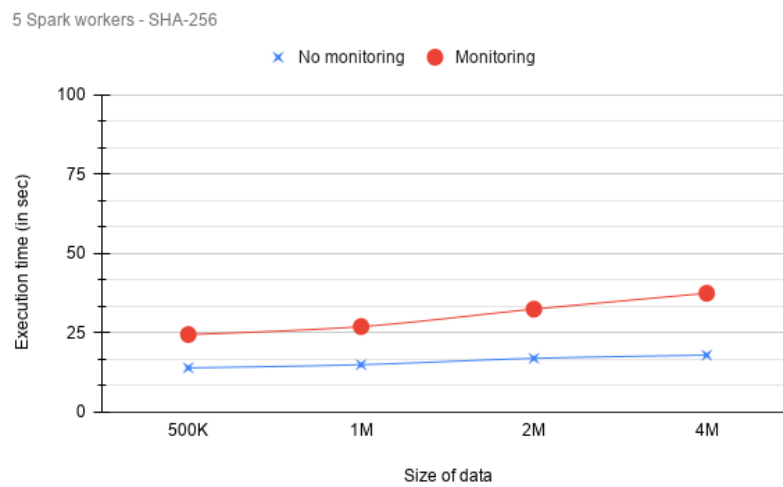


Fig. 5.67 Service execution time with and without data integrity monitoring using **SHA-256** on a cluster with **5 worker nodes** for multiple data set sizes

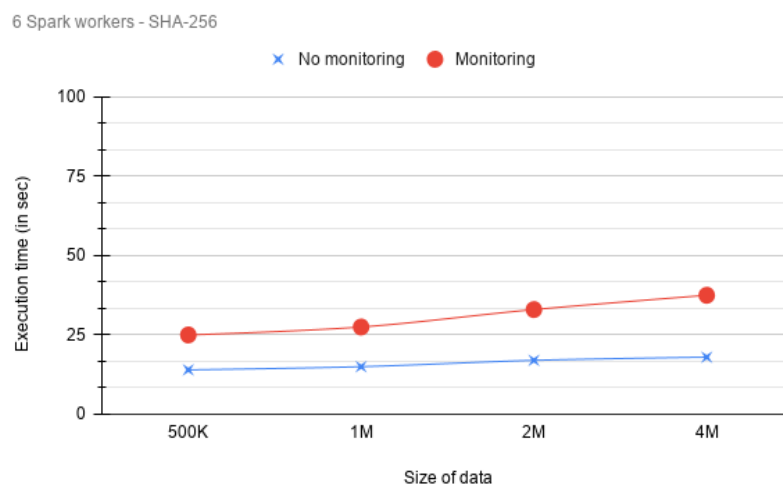


Fig. 5.68 Service execution time with and without data integrity monitoring using **SHA-256** on a cluster with **6 worker nodes** for multiple data set sizes

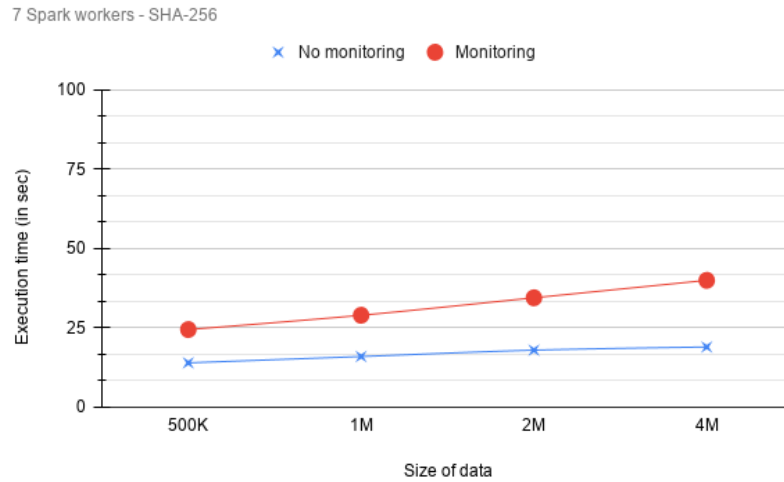


Fig. 5.69 Service execution time with and without data integrity monitoring using **SHA-256** on a cluster with **7 worker nodes** for multiple data set sizes

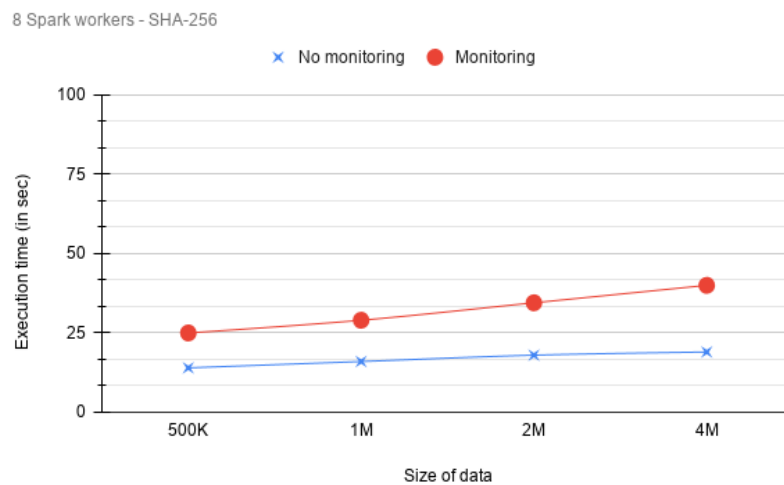


Fig. 5.70 Service execution time with and without data integrity monitoring using **SHA-256** on a cluster with **8 worker nodes** for multiple data set sizes

As it is shown in the graphs above, regardless of the cluster configuration, monitoring data privacy will impose a penalty on the overall service execution. We present the measurements that we have been able to collect for different data set sizes and different cluster configuration for **SHA-256** in table 5.6.

Data size	1	2	3	4	5	6	7	8	Overhead(%)
500K	77.78%	77.78%	75.00%	70.83%	75.00%	78.57%	75.00%	78.57%	76.07%
1M	84.62%	81.82%	81.82%	80.77%	80.00%	83.33%	81.25%	81.25%	81.86%
2M	94.74%	93.75%	92.86%	93.33%	91.18%	94.12%	91.67%	91.67%	92.91%
4M	108.00%	109.52%	110.00%	108.33%	108.33%	108.33%	110.53%	110.53%	109.20%

Table 5.6 Average overhead for monitoring data integrity using SHA-256 for different data set sizes and number of worker nodes

Similar to the data for MD5 and SHA-256, the data suggests that the number of the worker nodes do not significantly affect the overhead imposed as a result of the monitoring activity for data integrity. However, as expected, when the input data grows the overhead increases since more SHA-6 checksums need to be computed and emitted in the form of events in order for the monitor to reason about them. A visual representation of the data shown in the table above can be seen in figure ??.

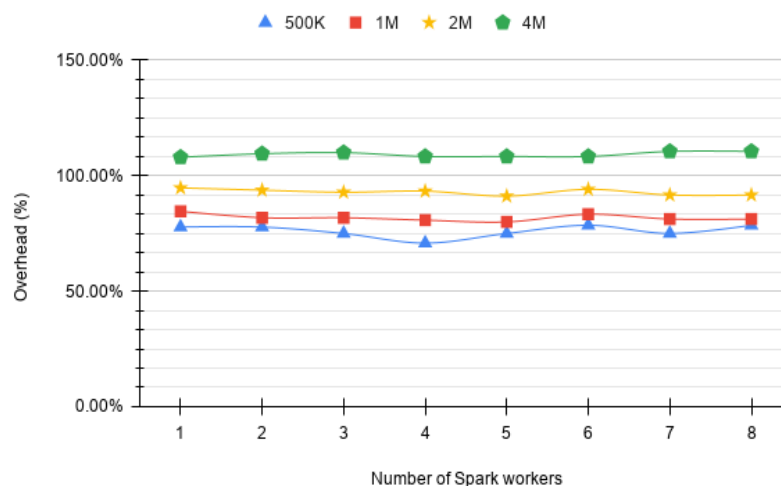


Fig. 5.71 Overlay graph of the overhead(%) over different number of workers for data integrity monitoring using SHA-256

In this section we have examined the overhead that is applied on the big data pipelines when data integrity is monitored. Our analysis was done across two variable namely the size of the data and the size of the cluster i.e. the number of workers available. However,

contrary to our analysis for data privacy and data availability, in the context of data integrity monitoring it is worth studying the effect of the hashing algorithm that is being used in relation to the overhead that this might cause on the service execution. For that reason we have plotted the average overhead for each individual hashing algorithm and we present it in figure 5.72.

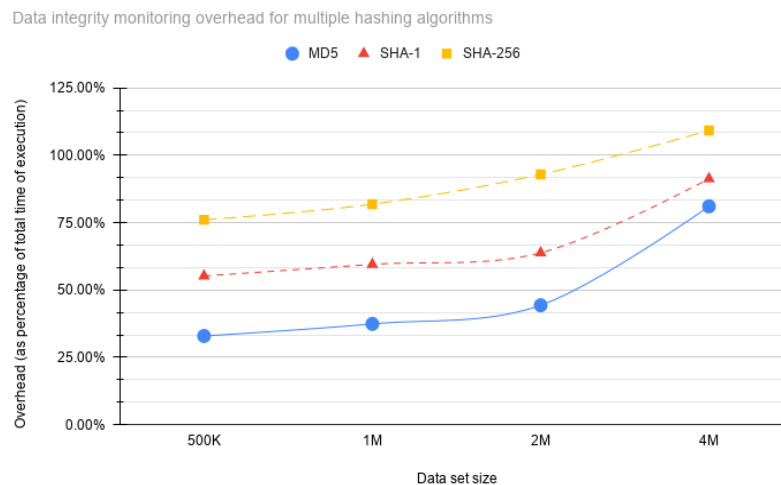


Fig. 5.72 Overlay graph of the overhead(%) over different number of workers for data integrity monitoring using MD5, SHA-1 and SHA-256

As it can be seen, MD5 inflicts the least average overhead, SHA-1 is the next closest with SHA-256 being the one that imposes the greatest overhead. This is somewhat expected and in agreement with what one would expect for two reasons. Firstly, MD5 is the least complex to produce in algorithmic terms with checksums of 128 bits, SHA-1 with checksums of 160 bits and SHA-256 with checksums of 256 bits. Secondly, due to the difference in the size of the checksums that they produce, more data need to be emitted over the network for the relevant events that are captured during execution which results in greater network latency and therefore it takes longer for the monitoring activity to complete. As the hashing algorithm that is being used becomes more complex and collision are less likely, an additional penalty is to be paid in terms of overhead. This is a case of allowing users to strike the perfect balance between efficiency and security by enabling them to choose the appropriate hashing algorithm that satisfies their requirements.

5.3 Evaluation Summary and Discussion

In this chapter we conducted a set of experiments to evaluate the efficacy of the framework that was proposed. More specifically, metrics were collected for the deployment of the event captors for the monitoring of data integrity, response time and location of execution of operations. Additional to that we also collected performance metrics for the execution time of the services with and without the event captors. Both the experiments for the deployment of the captors and the execution of the services were conducted 1000 times and averages were computed to evaluate the overhead that is imposed as a result of the monitoring activity in the average case.

For the deployment of the event captors the results that were collected are summarised in table 5.7. The deployment of the captors involves them being loaded by the corresponding Spark workers. All the event captors will have to be available on the worker nodes that they will operate on, to facilitate the capturing of the events. In our setup, this is achieved by having them available on a centralised location that the docker images will mount and therefore make them accessible to them.

Workers	Privacy	Availability	Integrity
1	1.4sec	1.4sec	1.4sec
2	1.46sec	1.46sec	1.46sec
3	1.63sec	1.66sec	1.66sec
4	1.7sec	1.7sec	1.7sec
5	1.86sec	1.9sec	1.9sec
6	1.96sec	1.96sec	1.96sec
7	2.23sec	2.2sec	2.2sec
8	2.5sec	2.36sec	2.36sec

Table 5.7 Summary table of the average deployment time for the event captors on clusters with different number of workers

As it is shown in table 5.7, the time it takes the event captors to get installed on the worker nodes, is similar, if not exactly the same, for all the event captors for the same cluster

configuration i.e. with the same number of workers. This holds true because the deployment of the event captors only describes what parts of the application will be intercepted and does not act on the data itself and therefore no additional overhead is imposed. The step of the event captor deployment is a one-off step and only happens when the Big Data analytics pipeline is submitted for execution. In addition, it can be seen that as the number of workers increase, it takes longer for all the event captors to be deployed. This is expected since the event captors will have to be deployed across the additional workers.

For the execution of the services the measurements that have been collected are summarised in table 5.8.

Data size	Availability	Privacy	MD5	SHA-1	SHA-256
500K	5.61%	40.18%	32.95%	55.21%	76.07%
1M	5.57%	30.91%	37.43%	59.52%	81.86%
2M	5.33%	16.62%	44.35%	63.81%	92.91%
4M	5.30%	11.90%	81.07%	91.31%	109.20%

Table 5.8 Summary table of the average overhead for availability, privacy and integrity monitoring for different data set sizes

In the table above, it can be observed that the overhead for data availability remains relatively similar regardless of the size of the data. This is justified by the fact that the way data availability is expressed i.e. response time, is not correlated with the size of the data or the number of the workers. As such it remains the same across different data set sizes and number of workers. In addition, the overhead for data privacy monitoring expressed as a percentage of the service execution without the monitoring capabilities being enabled, goes down as the data set size goes up. Finally, another important point that the data highlights is that in the case of the data integrity monitoring, the overhead that is imposed increases as we move to more complex hashing algorithms for the generation of the checksum for the intermediate produced data. More specifically, when the MD5 hashing algorithm is used the execution time of the service is affected the least whereas when the SHA-256 hashing algorithm is used, service execution times pay a higher price in terms of overall execution time. This is a direct result of the difference in the complexity of the algorithms used as

well as of the size of the checksums that are produced and need to be sent to the EVEREST monitor over the network. **MD5** produces the shortest checksums that are 128 bits long whereas **SHA-256** produces the longest checksums that are 256 bits long. The longer the length of the message digests that are produced, the larger the network overhead that will be imposed for the emission of the monitoring events. Also, we need to keep in mind that the size of the hashes has a cumulative effect with regards to the network overhead imposed during monitoring. That is because a hash value needs to be produced and emitted for every partition for every RDD that is involved in the execution of the service. In a real-life application this could easily get in the order of a few thousand hashes that need to be emitted and therefore an increase in the length of the checksums could have a significant effect on the overall service execution time.

A visual representation of the data presented in table 5.8 is plotted in figure 5.73.

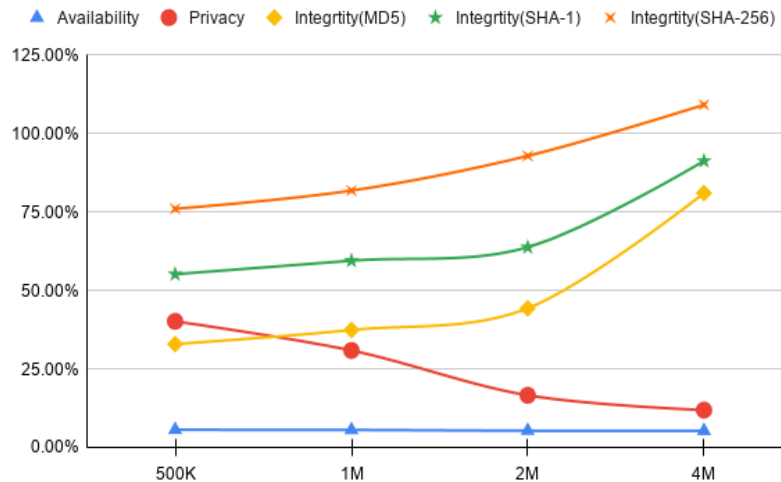


Fig. 5.73 Overlay graph of the average overhead(%) for all the security properties over different data set sizes

As it is depicted, data integrity monitoring is more expensive security property that we have been able to examine, with data integrity monitoring using SHA-256 as the hashing method, being the more resource intensive property to collect events for and monitor. On the flip side, data availability is the security property that imposed the least observable overhead regardless of the number of workers and data set size.

5.4 Summary

In this chapter we have presented the results of the evaluation of the monitoring framework that we have put forward in this thesis. Our evaluation has been conducted with regards to two different, but equally important, aspects of the monitoring activity i.e. the overhead imposed as a result of the automatic deployment of the event captors on the nodes of the cluster and the overhead imposed due to the event capturing activity itself. For a more comprehensive examination of the event capturing overhead, both in terms of deployment and service execution, we run our experiments for multiple data set size i.e. 500K, 1M, 2M and 4M data points on multiple cluster configuration i.e. with one up to 8 worker nodes. From our analysis, we were able to identify that the size of the data did not affect the overhead on the deployment of the captors which is something that we expected since the deployment is not concerned with the execution. However, as one would expect, an increase in the number of worker nodes caused an increase in the time it takes for all the event captors to be deployed. In addition, we were able to identify that number of worker nodes did not have a significant effect on the service execution time. The execution time overhead of the services remained relatively similar across different cluster configurations. Contrary to that, the size of the data seemed to affect the time it takes for the services to execute especially in the case of data integrity. The justification for that is that the monitoring of data integrity entails that each data point has to be processed as part of the computation but also a checksum for it needs to be produced. This suggests that more data points will lead to a slower service execution which was validated by the results that we were able to collection from our experiments.

Chapter 6

Conclusions and Future Work

6.1 Overview

In the final chapter of our thesis we will present an overview of the work that has been carried out for the design, development and evaluation of a security SLA monitoring framework for Big Data service pipelines. Moreover, we will provide an account of the contributions that our proposal makes in the state of the art in the domain of security SLA monitoring for Big Data services. Finally, we will highlight some limitations that we have been able to identify for our system and we will provide a list with future directions for the improvement and extension of the capabilities of the monitoring framework that was put forward in this thesis.

6.2 Summary of Research Work

In this thesis, and more specifically in chapter 3, we described a novel approach for addressing the challenges of runtime monitoring of security SLA for Big Data service pipelines. Our approach relies heavily on two pillars; the first one is the automation of the generation of the low-level artefacts that are necessary for the realisation of the monitoring activity from the end-user's high-level security requirements and the second one is the automation of the deployment process of the event captors that collect the monitoring data.

For the assessment of our framework we examined the runtime monitoring specification of three security property. The first one was the Big Data service response time that relates to

availability, the second one was the location of execution of the service operations that relates to data privacy and the third one was the integrity of the intermediate data that is produced during execution that relates to data integrity. All the rules for the monitoring activity for the properties that were explored were defined with the assistance of Event Calculus formulae. Subsequently the formulae were translated into EC-Assertion expressions that the EVEREST monitoring tool was able to interpret and reason about. The event captors were designed and developed in the form of Java agents that enabled the code instrumentation of the Big Data processing framework that performed the data processing, which in our case was Apache Spark. The evaluation of the monitoring events against the monitoring rules was performed by the EVEREST monitoring tool.

Finally, in chapter 4 we presented an integrated web platform that was used to facilitate the interaction of the end-users with the system in a comprehensive and straightforward manner with the assistance of a set of user interface components. The idea behind the development of this integrated platform that provides an single interface for the definition of the service level objectives that need to be monitored, is to provide to the end-users an SLA monitoring platform that is completely automated and requires the minimum amount of input from the user. Moreover, the SLA manager web application allowed us to completely streamline the definition and execution of the security SLA monitoring activity for technical and non-technical users alike.

6.3 Contributions

The contributions of the work presented in this thesis can be briefly summarised in the following 3 points:

1. Designed and developed a monitoring framework for the automatic translation of high-level security requirements into low-level monitorable artefacts that are then automatically monitored.
2. Designed and developed a monitoring framework where the event capturing process can handle changes both in terms of how the constituent services of the Big Data pipeline are arranged and in terms of the actual Big Data service code itself.

3. Designed and developed a set of event captors for the monitoring of security properties that relate to data availability, data privacy and data integrity, that are adaptive and elastic. Modifications in the Big Data service or the cluster that it gets executed does not entail any modification in the event captors.

6.4 Limitations

Within the context of our research, the proposed framework has been successfully designed and implemented. However, we have been able to identify a series of limitations that are presented in the following list:

1. The monitoring rules can only to be expressed in event calculus and subsequently in EC-Assertion formulae to support the evaluation of the events from the EVEREST monitor. This also implies that the events need to be emitted in an XML format that the monitor can understand and reason about.
2. The monitoring solution that we propose will not operate correctly on a compute cluster where the clocks of the nodes are not synchronised. This limitation is particularly pronounced in the case of nodes that are disparate with regards to the timezone that they are located.
3. The SLA manager that was developed for the definition of the service level objectives, can only be accessed via a web browser. No other interface is available for interacting with the system. Also, the end-users of the SLA management web application ought to have a basic understanding of the security properties that they need to monitor in order to express them through the application's user interface.

6.5 Future Work

We argue that the proposed framework is novel and that it makes a clear contribution in the current literature. However, our view is that an array of improvements and additions can be made to address a series of challenges that exist in the space of security SLA monitoring for

Big Data analytics that have not been addressed in this thesis. A list with future directions that can help to tackle some of those challenges are presented in the list below:

1. Create event captors for other Big Data processing frameworks apart from Apache Spark. This is particularly important with regards to our proposal because our framework has been designed to address pipelines of Big Data services and not just a single Big Data service. It is very typical for use-cases to require the specification of Big Data service pipelines that are composed of services that use different processing engines. Implementing event captors for multiple processing frameworks will allow the runtime security monitoring of pipelines composed of heterogeneous Big Data services that are executed on different platforms.
2. Combine multiple properties to support the monitoring activity of more than one properties for a single service. In our proposal we examined the monitoring of one security property per service. It would be highly desirable to allow the users to define multiple security properties to be monitored for the same service.
3. In the case of the data integrity, we monitor the preservation of data integrity for all the intermediate data that is produced during service execution but we do not monitor the integrity of the actual code that operates on the data. Since we are in the domain of distributed applications not only the data but also the executable code needs to be transmitted and executed across multiple nodes in the cluster and as such its integrity can be compromised as well. A more enterprise solution that deals with the runtime monitoring of data integrity as a whole in the context of Big Data service execution, should also involve the monitoring of the integrity of the code that gets executed across the nodes of the cluster.
4. Allow the users to define the level of granularity that they wish to monitor the integrity of the data. In our implementation we produce checksums for each partition and we argue that with this approach we make a reasonable trade-off. However, a more enterprise approach would be to enable users to take that decision. Users should be able to choose at what level the event captors should produce the hash values. This approach however will entail additional challenges such as the design of a monitoring engine that

would be able to reason about an increased number of events. For instance, producing checksums for every data tuple would incur a significant amount of computational stress on the EVEREST monitor which would have to be re-designed to support the increased load.

5. More security properties should be defined and monitored. Some interesting examples would be related to availability such as the mean time to repair (MTTR) and mean time to failure (MTTF). The aforementioned properties are particularly interesting given the fact that Big Data processing frameworks because of their distributed execution model are designed deliberately to address failures and have mechanisms to recover from them.
6. The communication channel between the event captors and the monitoring engine of EVEREST must be secure. Setting up the monitoring framework and taking all the necessary steps to automate the deployment and execution of the monitoring process without emitting the monitoring events in a secure manner, can severely undermine the credibility and integrity of the monitoring results. Modifying the monitoring events with man-in-the-middle type of attacks can allow security violation to go undetected or raise violations that haven't occurred.

References

- [1] Aws service-level-agreements. <https://aws.amazon.com/de/legal/service-level-agreements/>. Accessed: 1019-04-16.
- [2] Byte Buddy - runtime code generation for the Java virtual machine. URL <https://bytebuddy.net/>.
- [3] Google Online Security Blog: Announcing the first SHA1 collision, . URL <https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html>.
- [4] Google cloud platform service level agreements google cloud platform terms | google cloud. <https://cloud.google.com/terms/sla/>, . Accessed: 1019-04-16.
- [5] Sla for cloud services. https://azure.microsoft.com/en-us/support/legal/sla/cloud-services/v1_5/. Accessed: 1019-04-16.
- [6] Oecd guidelines on the protection of privacy and transborder flows of personal data - oecd. Accessed: 2019-03-06.
- [7] Configuration - Spark 2.4.3 Documentation. <https://spark.apache.org/docs/latest/configuration.html>.
- [8] Spring batch - reference documentation. <https://docs.spring.io/spring-batch/4.1.x/reference/pdf/spring-batch-reference.pdf>. Accessed: 2019-03-06.
- [9] Web Services Agreement Specification (WS-Agreement). In *Global Grid Forum GRAAP-WG*, volume 192, pages 1–80, 2004.
- [10] White paper: VMware high availability concepts, implementation, and best practices. Technical report, VMWare, 2007.
- [11] VMware Data Recovery. https://www.vmware.com/pdf/vdr_10_admin.pdf, 2009. [Online; accessed 30-April-2019].
- [12] White paper: Protecting mission-critical workloads with vmware fault tolerance. Technical report, VMWare, 2009.
- [13] CloudMonix AzureWatch Azure monitoring automation auto-scaling. <https://cloudmonix.com/aw/>, 2019. [Online; accessed 14-May-2019].
- [14] Amazon CloudWatch - Application and Infrastructure Monitoring. <https://aws.amazon.com/cloudwatch/>, 2019. [Online; accessed 19-May-2019].

- [15] LogicMonitor: SaaS-based Performance Monitoring Platform. <https://www.logicmonitor.com>, 2019. [Online; accessed 12-May-2019].
- [16] Web Performance Monitoring Tools - Monitis. <https://www.monitis.com>, 2019. [Online; accessed 14-May-2019].
- [17] G. Aceto, A. Botta, W. de Donato, and A. Pescapè. Cloud monitoring: Definitions, issues and future directions. In *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*, pages 63–67, Nov 2012.
- [18] European Union Agency and Information Security. *ENISA Threat Landscape 2013 Overview of current and emerging cyber-threats*. Number December. 2013. ISBN 9789279000775.
- [19] Charu C. Aggarwal and Philip S. Yu. A condensation approach to privacy preserving data mining. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *Advances in Database Technology - EDBT 2004*, pages 183–199, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24741-8.
- [20] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 530–533. ACM, 2011.
- [21] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. *SIGMOD Rec.*, 29(2):439–450, May 2000. ISSN 0163-5808.
- [22] Igor V. Anikin and Rinat M. Gazimov. Privacy preserving DBSCAN clustering algorithm for vertically partitioned data in distributed systems. *2017 International Siberian Conference on Control and Communications, SIBCON 2017 - Proceedings*, pages 1–4, 2017.
- [23] H. Ba, H. Zhou, S. Bai, J. Ren, Z. Wang, and L. Ci. jmonatt: Integrity monitoring and attestation of jvm-based applications in cloud computing. In *2017 4th International Conference on Information Science and Control Engineering (ICISCE)*, pages 419–423, July 2017.
- [24] Baker and Smith. Gridrm: an extensible resource monitoring system. In *2003 Proceedings IEEE International Conference on Cluster Computing*, pages 207–214, Dec 2003.
- [25] Bartosz Balis, Renata Slota, Jacek Kitowski, and Marian Bubak. On-line monitoring of service-level agreements in the grid. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7156 LNCS(PART 2):76–85, 2012. ISSN 03029743.
- [26] Carlos André Batista de Carvalho, Rossana Andrade, Miguel Franklin de Castro, Emanuel Coutinho, and Nazim Agoulmine. State of the art and challenges of security sla for cloud computing. *Computers & Electrical Engineering*, 59, 01 2017.

- [27] A. Bendahmane, M. Essaaïdi, A. El moussaoui, and A. Younes. A new mechanism to ensure integrity for mapreduce in cloud computing. In *2012 International Conference on Multimedia Computing and Systems*, pages 785–790, May 2012. doi: 10.1109/ICMCS.2012.6320295.
- [28] K. Bernsmed, M. G. Jaatun, P. H. Meland, and A. Undheim. Security slas for federated cloud services. In *2011 Sixth International Conference on Availability, Reliability and Security*, pages 202–209, Aug 2011.
- [29] Karin Bernsmed, Martin Jaatun, and Astrid Undheim. Security in service level agreements for cloud computing. pages 636–642, 01 2011.
- [30] Manuel Blum. Designing programs that check their work. *Journal of the ACM*, 42(1): 269–291, 1995.
- [31] Mike Boniface, Stephen C. Phillips, Alfonso Sanchez-Macian, and Mike Surridge. Dynamic Service Provisioning Using GRIA SLAs. In Elisabetta Di Nitto and Matei Ripeanu, editors, *Service-Oriented Computing - ICSOC 2007 Workshops*, pages 56–67, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [32] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In Yossi Azar and Thomas Erlebach, editors, *Algorithms – ESA 2006*, pages 684–695, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-38876-0.
- [33] Kevin D. Bowers, Ari Juels, and Alina Oprea. Hail: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09*, pages 187–198, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0.
- [34] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.
- [35] Pamela Carvallo, Ana R. Cavalli, and Wissam Mallouli. A platform for security monitoring of multi-cloud applications. In Alexander K. Petrenko and Andrei Voronkov, editors, *Perspectives of System Informatics*, pages 59–71, Cham, 2018. Springer International Publishing. ISBN 978-3-319-74313-4.
- [36] Roberto G. Cascella, Lorenzo Blasi, Yvon Jegou, Massimo Coppola, and Christine Morin. Contrail: Distributed application deployment under sla in federated heterogeneous clouds. In Alex Galis and Anastasius Gavras, editors, *The Future Internet*, pages 91–103, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38082-2.
- [37] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2): 4:1–4:26, June 2008. ISSN 0734-2071.
- [38] D. Chen and H. Zhao. Data security and privacy protection issues in cloud computing. In *2012 International Conference on Computer Science and Electronics Engineering*, volume 1, pages 647–651, March 2012.

- [39] Keke Chen and Ling Liu. Privacy preserving data classification with rotation perturbation. *Proceedings - IEEE International Conference on Data Mining, ICDM*, pages 589–592, 2005. ISSN 15504786.
- [40] Xu Chen and Qiming Huang. The data protection of mapreduce using homomorphic encryption. *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS*, pages 419–421, 2013. ISSN 23270586.
- [41] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. Sla decomposition: Translating service level objectives to system level thresholds. In *Fourth International Conference on Autonomic Computing (ICAC'07)*, pages 3–3, June 2007.
- [42] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security, CCSW '09*, pages 85–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-784-4.
- [43] Giuseppe Cicotti, Salvatore D'Antonio, Rosario Cristaldi, and Antonio Sergio. How to monitor QoS in Cloud Infrastructures: The QoSMONaaS approach. In Giancarlo Fortino, Costin Badica, Michele Malgeri, and Rainer Unland, editors, *Intelligent Distributed Computing VI*, pages 253–262, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-32524-3.
- [44] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. ISSN 0926227X.
- [45] Marco Comuzzi, Constantinos Kotsokalis, George Spanoudakis, and Ramin Yahyapour. Establishing and monitoring slas in complex service based systems. *2009 IEEE International Conference on Web Services, ICWS 2009*, pages 783–790, 2009.
- [46] Council of European Union. Council regulation (EU) no 679/2016, 2016. <https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1558202292114&uri=CELEX:32016R0679>.
- [47] Alfredo Cuzzocrea. Privacy and security of big data: Current challenges and future research perspectives. In *Proceedings of the First International Workshop on Privacy and Security of Big Data, PSBD '14*, pages 45–47, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-1583-8.
- [48] Maxwell Dayvson Da Silva and Hugo Lopes Tavares. *Redis Essentials*. Packt Publishing, 2015. ISBN 1784392456, 9781784392451.
- [49] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: WSLA-driven automated management. *IBM Systems Journal*, 43(1):136–158, 2010. ISSN 0018-8670.
- [50] S. A. De Chaves, R. B. Uriarte, and C. B. Westphall. Toward an architecture for monitoring private clouds. *IEEE Communications Magazine*, 49(12):130–137, December 2011. ISSN 0163-6804.

- [51] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.
- [52] Vincent C Emeakaroha, Ivona Brandic, Michael Maurer, and Schahram Dustdar. Low Level Metrics to High Level SLAs - LoM2HiS Framework : Bridging the Gap Between Monitored Metrics and SLA Parameters in Cloud Environments. pages 48–54, 2010.
- [53] Vincent C. Emeakaroha, Tiago C. Ferreto, Marco A.S. Netto, Ivona Brandic, and Cesar A.F. De Rose. CASViD: Application level monitoring for SLA violation detection in clouds. *Proceedings - International Computer Software and Applications Conference*, pages 499–508, 2012. ISSN 07303157.
- [54] Amitai Etzioni. *The limits of privacy*. BasicBooks, 2000.
- [55] Howard Foster and George Spanoudakis. Advanced service monitoring configurations with sla decomposition and selection. In *Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11*, pages 1582–1589, New York, NY, USA, 2011. ACM.
- [56] W. Fu and Q. Huang. Grideye: A service-oriented grid monitoring system with improved forecasting algorithm. In *2006 Fifth International Conference on Grid and Cooperative Computing Workshops*, pages 5–12, Oct 2006.
- [57] Y. Gahi, M. Guennoun, and H. T. Mouftah. Big data analytics: Security and privacy challenges. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 952–957, June 2016.
- [58] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.
- [59] Z. Gao, N. Desalvo, P. D. Khoa, S. H. Kim, L. Xu, W. W. Ro, R. M. Verma, and W. Shi. Integrity protection for big data processing with dynamic redundancy computation. In *2015 IEEE International Conference on Autonomic Computing*, pages 159–160, July 2015. doi: 10.1109/ICAC.2015.34.
- [60] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing, STOC '09*, pages 169–178, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-506-2. doi: 10.1145/1536414.1536440. URL <http://doi.acm.org/10.1145/1536414.1536440>.
- [61] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *Stoc*, volume 9, pages 169–178, 2009.
- [62] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.
- [63] A. J. Gonzalez and B. E. Helvik. System management to comply with sla availability guarantees in cloud computing. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 325–332, Dec 2012.

- [64] J. Gradecki and J. Cole. *Mastering Apache Velocity*. Java open source library. Wiley, 2003. ISBN 9780471457947.
- [65] Sam Guinea, Luciano Baresi, George Spanoudakis, and Olivier Nano. Comprehensive Monitoring of BPEL Processes. *IEEE Internet Computing*, 2011. ISSN 1089-7801.
- [66] Cheng Guo, Xinyu Tang, Yingmo Jie, and Bin Feng. Efficient method to verify the integrity of data with supporting dynamic data in cloud computing. *Science China Information Sciences*, 61(11):119101, Aug 2018. ISSN 1869-1919.
- [67] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.
- [68] G. Hogben and A. Pannetrat. Mutant apples: A critical examination of cloud sla availability definitions. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 379–386, Dec 2013.
- [69] Zaid Alaa Hussien, Hai Jin, Zaid Ameen Abduljabbar, Mohammed Abdulridha Hussain, Salah H. Abbdal, and Deqing Zou. Scheme for ensuring data security on cloud data storage in a semi-Trusted third party auditor. *Proceedings of 2015 4th International Conference on Computer Science and Network Technology, ICCSNT 2015*, 01(Iccsnt):1200–1203, 2016.
- [70] Nancy J. King and V.T. Raja. Protecting the privacy and security of sensitive customer data in the cloud. *Computer Law & Security Review*, 28:308–319, 06 2012. doi: 10.1016/j.clsr.2012.03.003.
- [71] Martin Gilje Jaatun, Karin Bernsmed, and Astrid Undheim. Security slas – an idea whose time has come? In Gerald Quirchmayr, Josef Basl, Ilsun You, Lida Xu, and Edgar Weippl, editors, *Multidisciplinary Research and Practice for Information Systems*, pages 123–130, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32498-7.
- [72] Surabhi Jain, Er Navneet Randhawa, and Deepali Kansal. Data Security in the Realm of Cloud Computing. *Ieeexplore.Ieee.Org*, pages 61–64, 2012.
- [73] C. Ji, Y. Li, W. Qiu, U. Awada, and K. Li. Big data processing in cloud computing environments. In *2012 12th International Symposium on Pervasive Systems, Algorithms and Networks*, pages 17–23, Dec 2012.
- [74] Ari Juels and Burton S. Kaliski, Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07*, pages 584–597, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2.
- [75] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qiam Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 42:29 – 42, 2003.

- [76] Fu Kevin, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. volume 4, San Diego, California, 2000.
- [77] Jay Kreps. Kafka : a distributed messaging system for log processing. 2011.
- [78] S. Lee, H. Park, and Y. Shin. Cloud computing availability: Multi-clouds for big data service. *Communications in Computer and Information Science*, 310 CCIS:799–806, 2012.
- [79] Avraham Leff and James T Rayfield. Web-application development using the model/view/controller design pattern. In *Proceedings fifth ieee international enterprise distributed object computing conference*, pages 118–127. IEEE, 2001.
- [80] Hongyu Liu, Leiting Chen, and Liyao Zeng. Cloud data integrity checking with deduplication for confidential data storage. In Sheng Wen, Wei Wu, and Aniello Castiglione, editors, *Cyberspace Safety and Security*, pages 460–467, Cham, 2017. Springer International Publishing. ISBN 978-3-319-69471-9.
- [81] D. Lorenzoli and G. Spanoudakis. Predicting software service availability: Towards a runtime monitoring approach. In *2011 IEEE International Conference on Web Services*, pages 736–737, July 2011. doi: 10.1109/ICWS.2011.77.
- [82] Björn Lundgren and Niklas Möller. Defining Information Security. *Science and Engineering Ethics*, pages 1–23, 2017. ISSN 14715546.
- [83] Ali Maetouq, Salwani Mohd, Noor Azurati, Nurazean Maarop, Nilam Nur, and Hafiza Abas. Comparison of hash function algorithms against attacks: A review. *International Journal of Advanced Computer Science and Applications*, 9, 01 2018. doi: 10.14569/IJACSA.2018.090813.
- [84] Khaled Mahbub and George Spanoudakis. Monitoring ws-agreements: An event calculus based approach. In *In Test and Analysis of Web Services*, pages 265–306. Springer Verlag, 2007.
- [85] Khaled Mahbub, George Spanoudakis, and Theocharis Tsigkritis. Translation of SLAs into Monitoring Specification. pages 636–642, 2011.
- [86] Dekker Marnix and Hogben Giles. Survey and analysis of security parameters in cloud SLAs across the European public sector. Technical report, 2011.
- [87] J. Mateo-Fornés, F. Solsona-Tehàs, J. Vilaplana-Mayoral, I. Teixidó-Torrelles, and J. Rius-Torrentó. Cart, a decision sla model for saas providers to keep qos regarding availability and performance. *IEEE Access*, 7:38195–38204, 2019. ISSN 2169-3536.
- [88] Daniel A. Menasce and Virgilio Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001. ISBN 0130659037.
- [89] Soumendra Mohanty, Madhu Jagadeesh, and Harsha Srivatsa. *Application Architectures for Big Data and Analytics*, pages 107–154. Apress, Berkeley, CA, 2013. ISBN 978-1-4302-4873-6.

- [90] Adina Mosincat and Walter Binder. Automated maintenance of service compositions with SLA violation detection and dynamic binding. *International Journal on Software Tools for Technology Transfer*, 13(2):167–179, 2011. ISSN 14332779.
- [91] Erik T. Mueller. Event calculus. In *Handbook of Knowledge Representation*, 2008.
- [92] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, CCSW '11*, pages 113–124, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1004-8.
- [93] Surya Nepal, John Zic, and Shiping Chen. WSLA+: Web service level agreement language for collaborations. In *Proceedings - 2008 IEEE International Conference on Services Computing, SCC 2008*, volume 2, pages 485–488, 2008. ISBN 9780769532837.
- [94] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringham, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proc. VLDB Endow.*, 10(12):1634–1645, August 2017. ISSN 2150-8097.
- [95] P. Ora and P. R. Pal. Data security and integrity in cloud computing based on rsa partial homomorphic and md5 cryptography. In *2015 International Conference on Computer, Communication and Control (IC4)*, pages 1–6, Sep. 2015.
- [96] Charles P. Pfleeger and Deborah M. Cooper. Security and privacy: Promising advances. *Software, IEEE*, 14:27 – 32, 10 1997.
- [97] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.
- [98] Scott Paquette, Paul T. Jaeger, and Susan C. Wilson. Identifying the security risks associated with governmental use of cloud computing. *Government Information Quarterly*, 27(3):245–253, 2010. ISSN 0740624X.
- [99] Adrian Paschke and M Bichler. Sla representation, management and enforcement. pages 158– 163, 01 2005. ISBN 0-7695-2274-2.
- [100] Adrian Paschke and Elisabeth Schnappinger-Gerull. A categorization scheme for sla metrics. pages 25–40, 01 2006.
- [101] Wayne Pauley. *An Empirical Study of Privacy Risk Assessment Methodologies in Cloud Computing Environments*. PhD thesis, 01 2013.
- [102] D. Playfair, A. Trehan, B. McLarnon, and D. S. Nikolopoulos. Big data availability: Selective partial checkpointing for in-memory database queries. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2785–2794, Dec 2016.
- [103] Sakshi Porwal, Srijith Krishnan Nair, and Theo Dimitrakos. Regulatory impact of data protection and privacy in the cloud. volume 358, pages 290–299, 06 2011. doi: 10.1007/978-3-642-22200-9_23.

- [104] Niels Provos and David Mazières. A future-adaptive password scheme. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '99*, pages 32–32, Berkeley, CA, USA, 1999. USENIX Association.
- [105] FIPS Pub. Standards for security categorization of federal information and information systems.
- [106] Deepak Puthal, Xindong Wu, Surya Nepal, Rajiv Ranjan, and Jinjun Chen. SEEN: A Selective Encryption Method to Ensure Confidentiality for Big Sensing Data Streams. *IEEE Transactions on Big Data*, pages 1–1, 2017.
- [107] M. Rak, S. Venticinque, T. M'hr, G. Echevarria, and G. Esnal. Cloud application monitoring: The mosaic approach. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 758–763, Nov 2011.
- [108] M. Rak, N. Suri, J. Luna, D. Petcu, V. Casola, and U. Villano. Security as a service using an sla-based approach via specs. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 2, pages 1–6, Dec 2013.
- [109] Seema Rawat and Ram Shankar. Predictive model for data availability in big data processing. *Ssrn*, pages 802–807, 2018.
- [110] Ricardo J. Rodríguez. Evolution and characterization of point-of-sale ram scraping malware. *Journal of Computer Virology and Hacking Techniques*, 13, 05 2016. doi: 10.1007/s11416-016-0280-4.
- [111] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping performance and dependability attributes of Web services. *Proceedings - ICWS 2006: 2006 IEEE International Conference on Web Services*, pages 205–212, 2006.
- [112] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce. pages 297–312, 07 2010.
- [113] A. Sahai, S. Graupner, V. Machiraju, and A. van Moorsel. Specifying and monitoring guarantees in commercial grids through sla. In *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings.*, pages 292–299, May 2003.
- [114] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Li Jie Jin, Fabio Casati, H P Laboratories, and Page Mill Road. Automated SLA Monitoring for Web Services. *Language*, pages 1–26, 2002.
- [115] Parnia Samimi and Ahmed Patel. Review of pricing models for grid & cloud computing. *2011 IEEE Symposium on Computers & Informatics*, pages 634–639, 2011.
- [116] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1-2): 460–471, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920902. URL <http://dx.doi.org/10.14778/1920841.1920902>.

- [117] N. Sfondrini, G. Motta, and L. You. Service level agreement (sla) in public cloud environments: A survey on the current enterprises adoption. In *2015 5th International Conference on Information Science and Technology (ICIST)*, pages 181–185, April 2015.
- [118] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, pages 90–107, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-89255-7.
- [119] Syed Yousaf Shah, Brent Paulovicks, and Petros Zerfos. Data-at-rest security for spark. *Proceedings - 2016 IEEE International Conference on Big Data, Big Data 2016*, pages 1464–1473, 2016.
- [120] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010. doi: 10.1109/MSST.2010.5496972.
- [121] Renata Słota, Darin Nikolow, Paweł Młoczek, and Jacek Kitowski. Semantic-based sla monitoring of storage resources. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 232–241. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31500-8.
- [122] A. P. Snow and G. R. Weckman. What are the chances an availability sla will be violated? In *Sixth International Conference on Networking (ICN'07)*, pages 35–35, April 2007.
- [123] George Spanoudakis, Christos Kloukinas, and Khaled Mahbub. *The runtime monitoring framework of SERENITY*, volume 45, pages 213–237. 03 2009. doi: 10.1007/978-0-387-88775-3_13.
- [124] R. Sravan Kumar and A. Saxena. Data integrity proofs in cloud storage. In *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*, pages 1–4, Jan 2011. doi: 10.1109/COMSNETS.2011.5716422.
- [125] Stephen A. Stelting and Olav Maassen-Van Leeuwen. *Applied Java Patterns*. Prentice Hall Professional Technical Reference, 2001. ISBN 0130935387.
- [126] Kurt Stuke. Vision, big data, and the allegory of the cave. *Open Journal of Business and Management*, 03:422–424, 01 2015. doi: 10.4236/ojbm.2015.34041.
- [127] Dawei Sun, Guiran Chang, Lina Sun, and Xingwei Wang. Surveying and analyzing security, privacy and trust issues in cloud computing environments. *Procedia Engineering*, 15:2852 – 2856, 2011. ISSN 1877-7058. CEIS 2011.
- [128] Dan Svantesson and Roger Clarke. Privacy and consumer risks in cloud computing. *Computer Law & Security Review*, 26:391–397, 07 2010.
- [129] H. Takabi, J. B. D. Joshi, and G. Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security Privacy*, 8(6):24–31, Nov 2010. ISSN 1540-7993.

- [130] Vassilka Tchifilionova. Security and privacy implications of cloud computing – lost in the cloud. In Jan Camenisch, Valentin Kisimov, and Maria Dubovitskaya, editors, *Open Research Problems in Network Security*, pages 149–158, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19228-9.
- [131] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2376-5.
- [132] H. Ulusoy, M. Kantarcioglu, and E. Pattuk. Trustmr: Computation integrity assurance system for mapreduce. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 441–450, Oct 2015.
- [133] Huseyin Ulusoy, Murat Kantarcioglu, Erman Pattuk, and Kevin Hamlen. Vigiles: Fine-grained access control for MapReduce systems. *Proceedings - 2014 IEEE International Congress on Big Data, BigData Congress 2014*, pages 40–47, 2014.
- [134] Huseyin Ulusoy, Pietro Colombo, Elena Ferrari, Murat Kantarcioglu, and Erman Pattuk. Guardmr: Fine-grained security policy enforcement for mapreduce systems. 04 2015.
- [135] Van der Wees Arthur, Catteddu Daniele, Luna Jesus, Edwards Mike, Schifano Nicholas, Scoca Lucia Maddalena, and Tagliabue Stefano. Cloud Service Level Agreement Standardisation Guidelines. Technical report, 2014.
- [136] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1.
- [137] Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya. Aneka: A software platform for .net-based cloud computing. *CoRR*, abs/0907.4622, 2009. URL <http://arxiv.org/abs/0907.4622>.
- [138] S. Venkatesan and Abhishek Vaish. Multi-agent based dynamic data integrity protection in cloud computing. In Vinu V. Das, Janahanlal Stephen, and Yogesh Chaba, editors, *Computer Networks and Information Technologies*, pages 76–82, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19542-6.
- [139] A. Viratanapanu, A. Kamil, A. Hamid, Y. Kawahara, and T. Asami. On demand fine grain resource monitoring system for server consolidation. In *2010 ITU-T Kaleidoscope: Beyond the Internet? - Innovations for Future Networks and Services*, pages 1–8, Dec 2010.
- [140] C. Wang, K. Ren, W. Lou, and J. Li. Toward publicly auditable secure cloud data storage services. *IEEE Network*, 24(4):19–24, July 2010.

- [141] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. *Proceedings - IEEE INFOCOM*, pages 1–9, 2010. ISSN 0743166X.
- [142] Yongzhi Wang and Jinpeng Wei. VIAF: Verification-based integrity assurance framework for MapReduce. *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011*, pages 300–307, 2011.
- [143] Yongzhi Wang, Jinpeng Wei, Mudhakar Srivatsa, Yucong Duan, and Wencai Du. IntegrityMR: Integrity assurance framework for big data analytics and management applications. *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013*, pages 33–40, 2013.
- [144] Hal Wasserman and Manuel Blum. Software reliability via runtime result-checking. *Journal of the ACM*, 44(6):826–849, 2002. ISSN 00045411.
- [145] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security, CCSW '09*, pages 91–96, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-784-4.
- [146] Wei Wei, Juan Du, Ting Yu, and Xiaohui Gu. SecureMR: A service integrity assurance framework for map reduce. *Proceedings - Annual Computer Security Applications Conference, ACSAC*, pages 73–82, 2009. ISSN 10639527.
- [147] Wei Wei, Ting Yu, and Rui Xue. IBigTable: Practical data integrity for bigtable in public cloud. pages 341–352, 02 2013. doi: 10.1145/2435349.2435399.
- [148] Branimir Wetzstein, Philipp Leitner, Florian Rosenberg, Ivona Brandic, Schahram Dustdar, and Frank Leymann. Monitoring and analyzing influential factors of business process performance. *Proceedings - 13th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2009*, pages 141–150, 2009.
- [149] Paul S Wooley. Identifying cloud computing security risks. 2011.
- [150] Z. Xiao and Y. Xiao. Security and privacy in cloud computing. *IEEE Communications Surveys Tutorials*, 15(2):843–859, Second 2013. ISSN 1553-877X.
- [151] Guangwei Xu, Chunlin Chen, Hongya Wang, Zhuping Zang, Mugen Pang, and Ping Jiang. Two-level verification of data integrity for data storage in cloud computing. In Gang Shen and Xiong Huang, editors, *Advanced Research on Electronic Commerce, Web Application, and Communication*, pages 439–445, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [152] Chuan Yao, Li Xu, Xinyi Huang, and Joseph K. Liu. A secure remote data integrity checking cloud storage system from threshold encryption. *J. Ambient Intelligence and Humanized Computing*, 5:857–865, 2014.
- [153] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. *Proceedings - IEEE INFOCOM*, pages 1–9, 2010. ISSN 0743166X.

- [154] Yong Yu, Man Ho Au, Yi Mu, Shaohua Tang, Jian Ren, Willy Susilo, and Liju Dong. Enhanced privacy of a remote data integrity-checking protocol for secure cloud storage. *International Journal of Information Security*, 14(4):307–318, Aug 2015. ISSN 1615-5270.
- [155] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, and Ankur Dave. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2, 2012. ISSN 00221112.
- [156] P. Zerfos, H. Yeo, B. D. Paulovicks, and V. Sheinin. Sdfs: Secure distributed file system for data-at-rest security for hadoop-as-a-service. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1262–1271, Oct 2015.
- [157] Kehuan Zhang, Xiaoyong Zhou, Yangyi Chen, XiaoFeng Wang, and Yaoping Ruan. Sedic: Privacy-aware data intensive computing on hybrid clouds. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11*, pages 515–526. ACM, 2011. ISBN 978-1-4503-0948-6.
- [158] Zibin Zheng, Jieming Zhu, and Michael R Lyu. Service-generated big data and big data-as-a-service: an overview. In *2013 IEEE international congress on Big Data*, pages 403–410. IEEE, 2013.
- [159] Dimitrios Zissis and Dimitrios Lekkas. Addressing cloud computing security issues. *Future Generation Computer Systems*, 28(3):583 – 592, 2012. ISSN 0167-739X.

Appendix A

Composed Task Runner for Spark Submit Command

Listing A.1 Apache Spark Client Task Configuration

```
1 package org.springframework.cloud.task.app.spark.client;
2
3 import org.apache.commons.io.FileUtils;
4 import org.apache.commons.logging.Log;
5 import org.apache.commons.logging.LogFactory;
6 import org.apache.velocity.Template;
7 import org.apache.velocity.VelocityContext;
8 import org.apache.velocity.app.VelocityEngine;
9 import org.apache.velocity.runtime.RuntimeConstants;
10 import org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader;
11 import org.springframework.beans.factory.annotation.Autowired;
12 import org.springframework.boot.CommandLineRunner;
13 import org.springframework.boot.context.properties.EnableConfigurationProperties;
14 import org.springframework.cloud.task.configuration.EnableTask;
15 import org.springframework.context.annotation.Bean;
16 import org.springframework.context.annotation.Configuration;
17
18 import java.io.File;
19 import java.io.StringWriter;
20 import java.util.Properties;
21
22 @EnableTask
23 @Configuration
24 @EnableConfigurationProperties(SparkClientTaskProperties.class)
25 public class SparkClientTaskConfiguration {
26
27     private final static String TEMPLATE_PATH = "spark-submit-template.vm";
28
29     @Bean
30     public CommandLineRunner commandLineRunner() {
31         return new SparkAppClientRunner();
32     }
33
34     private class SparkAppClientRunner implements CommandLineRunner {
35
36         private final Log logger = LogFactory.getLog(SparkAppClientRunner.class);
```

```

37
38     @Autowired
39     private SparkClientTaskProperties config;
40
41
42     @Override
43     public void run(String... args) throws Exception {
44
45         Properties properties = new Properties();
46         properties.setProperty(RuntimeConstants.INPUT_ENCODING, "UTF-8");
47         properties.setProperty(RuntimeConstants.OUTPUT_ENCODING, "UTF-8");
48         properties.setProperty(RuntimeConstants.RESOURCE_LOADER, "class");
49         properties.setProperty("class.resource.loader.class", ClasspathResourceLoader.class.getName());
50         VelocityEngine engine = new VelocityEngine(properties);
51
52         VelocityContext context = new VelocityContext();
53         Template template = engine.getTemplate(TEMPLATE_PATH);
54         context.put("config", config);
55
56         StringWriter writer = new StringWriter();
57         template.merge(context, writer);
58
59         File executable = File.createTempFile("toreador-spark-submit@", ".sh");
60
61         FileUtils.writeStringToFile(executable, writer.toString().replaceAll("\r\n", "\n"), "UTF-8");
62         Runtime.getRuntime().exec(new String[]{"chmod", "775", executable.getAbsolutePath().getAbsolutePath()});
63
64         final ProcessBuilder sparkSubmitCommand = new ProcessBuilder("sh", executable.getAbsolutePath().getAbsolutePath());
65         Process p = sparkSubmitCommand.start();
66
67         int submit = p.waitFor();
68     }
69 }
70 }

```

Listing A.2 Snippet of the EC-Assertion template for the location of the operation execution

```

1
2 <?xml version="1.0" encoding="UTF-8"?>
3 <model:CertificationModel
4     xmlns:sch="http://www.ascc.net/xml/schematron"
5     xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
6     xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
7     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
8     <CertificationModelID>toreador:cm:id:monitoring:00001</CertificationModelID>
9
10     <Signature>
11         <Name>City</Name>
12         <Role>CA</Role>
13     </Signature>
14     <ToC Id="toc-toreador">
15         <CloudLayer>ToreadorPlatform</CloudLayer>
16         <ConcreteToc></ConcreteToc>
17         <TocDescription>VIRTUALMACHINE</TocDescription>
18         <TocURI>http://xyz.com</TocURI>
19         <ToM>
20             <providesInterface>
21                 <ID>tor1</ID>
22                 <ProviderRef>Toreador</ProviderRef>
23                 <Endpoint>
24                     <ID>T01</ID>
25                     <Location>localhost</Location>
26                     <Protocol>SOAP</Protocol>

```

```

27         </Endpoint>
28     <Interface>
29         <InterfaceSpec>
30             <Name>toreadorSLA</Name>
31             <Operation>
32                 <interfaceId>toreadorSLA</interfaceId>
33                 <partition>id001</partition>
34                 <operationName>writerdd</operationName>
35                 <inputVariable forMatching="true" persistent="false">
36                     <varName>appId</varName>
37                     <varType>string</varType>
38                 </inputVariable>
39                 <inputVariable forMatching="true" persistent="false">
40                     <varName>appName</varName>
41                     <varType>string</varType>
42                 </inputVariable>
43                 <inputVariable forMatching="true" persistent="false">
44                     <varName>rdId</varName>
45                     <varType>string</varType>
46                 </inputVariable>
47                 <inputVariable forMatching="true" persistent="false">
48                     <varName>partId</varName>
49                     <varType>string</varType>
50                 </inputVariable>
51                 <inputVariable forMatching="true" persistent="false">
52                     <varName>ip</varName>
53                     <varType>string</varType>
54                 </inputVariable>
55             </Operation>
56         </InterfaceSpec>
57     </Interface>
58 </providesInterface>
59 </ToM>
60 </ToC>
61 <SecurityProperty SecurityPropertyId="TOREADOR-PRIVACY-001"
62     SecurityPropertyDefinition="BCR:privacy:toreador-privacy" Vocabulary="CSA"
63     ShortName="BCR:privacy">
64     <sProperty>
65         <propertyPerformance>
66             <propertyPerformanceRow>
67                 <propertyPerformanceCell name="verified">"true"
68             </propertyPerformanceCell>
69             </propertyPerformanceRow>
70         </propertyPerformance>
71         <propertyParameterList />
72     </sProperty>
73 </SecurityProperty>
74 <SecurityPropertyAssertions>
75     <Assertion ID="AS001">
76         <InterfaceDeclr>
77             <ID>001</ID>
78             <ProviderRef>city</ProviderRef>
79             <Endpoint>
80                 <ID>T01</ID>
81                 <Location>localhost</Location>
82                 <Protocol>SOAP</Protocol>
83             </Endpoint>
84             <Interface>
85                 <InterfaceSpec>
86                     <Name>toreadorSLA</Name>
87                     <Operation>
88                         <interfaceId>toreadorSLA</interfaceId>
89                         <partition>id001</partition>
90                         <operationName>writerdd</operationName>

```

```

91         <inputVariable forMatching="true" persistent="false">
92             <varName>appId</varName>
93             <varType>string</varType>
94         </inputVariable>
95         <inputVariable forMatching="true" persistent="false">
96             <varName>appName</varName>
97             <varType>string</varType>
98         </inputVariable>
99         <inputVariable forMatching="true" persistent="false">
100             <varName>rddId</varName>
101             <varType>string</varType>
102         </inputVariable>
103         <inputVariable forMatching="true" persistent="false">
104             <varName>partId</varName>
105             <varType>string</varType>
106         </inputVariable>
107         <inputVariable forMatching="true" persistent="false">
108             <varName>ip</varName>
109             <varType>string</varType>
110         </inputVariable>
111     </Operation>
112 </InterfaceSpec>
113 </Interface>
114 </InterfaceDeclr>
115 <Guaranteed ID="trustedFluentID1" forChecking="false" type="future">
116     <quantification>
117         <quantifier>forall</quantifier>
118         <timeVariable>
119             <varName>t0</varName>
120             <varType>TimeVariable</varType>
121         </timeVariable>
122     </quantification>
123     <postcondition>
124         <atomicCondition conditionID="trustedFluentac1">
125             <stateCondition>
126                 <initially>
127                     <state name="trustedFluent">
128                         <argument>
129                             <variable forMatching="true" persistent="false">
130                                 <varName>trustedIP</varName>
131                                 <array>
132                                     <type>stringArray</type>
133                                     #set($count = 0)
134                                     #foreach( $ip in $trustedIps )
135                                         <value>
136                                             <indexValue>$count</indexValue>
137                                             <cellValue>$ip</cellValue>
138                                         </value>
139                                         #set($count = $count + 1)
140                                         #end
141                                     </array>
142                                 </variable>
143                             </argument>
144                         </state>
145                     <timeVar>
146                         <varName>t0</varName>
147                         <varType>TimeVariable</varType>
148                     </timeVar>
149                 </initially>
150             </stateCondition>
151         </atomicCondition>
152     </postcondition>
153 </Guaranteed>
154 <Guaranteed forChecking="true" ID="PrivacyRule" type="Future_Formula">

```

```

155         <quantification>
156             <quantifier>forall</quantifier>
157             <timeVariable>
158                 <varName>t1</varName>
159                 <varType>TimeVariable</varType>
160             </timeVariable>
161         </quantification>
162     <precondition>
163         <atomicCondition conditionID="ac0">
164             <eventCondition unconstrained="true">
165                 <event>
166                     <eventID forMatching="true" persistent="false">
167                         <varName>ToreadorPrivacy</varName>
168                     </eventID>
169                     <call>
170                         <interfaceId>toreadorSLA</interfaceId>
171                         <OperationId>1</OperationId>
172                         <operationName>compute</operationName>
173                         <inputVariable forMatching="true" persistent="false">
174                             <varName>status1</varName>
175                             <varType>OpStatus</varType>
176                             <value></value>
177                         </inputVariable>
178                         <inputVariable forMatching="true" persistent="false">
179                             <varName>sender1</varName>
180                             <varType>Entity</varType>
181                             <value></value>
182                         </inputVariable>
183                         <inputVariable forMatching="true" persistent="false">
184                             <varName>receiver1</varName>
185                             <varType>Entity</varType>
186                             <value></value>
187                         </inputVariable>
188                         <inputVariable forMatching="true" persistent="false">
189                             <varName>source1</varName>
190                             <varType>Entity</varType>
191                             <value></value>
192                         </inputVariable>
193                         <inputVariable forMatching="true" persistent="false">
194                             <varName>serviceId</varName>
195                             <varType>string</varType>
196                             <value></value>
197                         </inputVariable>
198                         <inputVariable forMatching="true" persistent="false">
199                             <varName>appId</varName>
200                             <varType>string</varType>
201                         </inputVariable>
202                         <inputVariable forMatching="true" persistent="false">
203                             <varName>appName</varName>
204                             <varType>string</varType>
205                         </inputVariable>
206                         <inputVariable forMatching="true" persistent="false">
207                             <varName>rddId</varName>
208                             <varType>string</varType>
209                         </inputVariable>
210                         <inputVariable forMatching="true" persistent="false">
211                             <varName>partId</varName>
212                             <varType>string</varType>
213                         </inputVariable>
214                         <inputVariable forMatching="true" persistent="false">
215                             <varName>ip</varName>
216                             <varType>string</varType>
217                         </inputVariable>
218                     </call>

```

[illegible]


```

283                                     <varType>string</varType>
284                                     </variable>
285                                     </argument>
286                                     </operationCall>
287                                 </operand1>
288                                 <operand2>
289                                     <constant>
290                                         <name>verified</name>
291                                         <value>true</value>
292                                     </constant>
293                                 </operand2>
294                             </equal>
295                             <timeVar>
296                                 <varName>t1</varName>
297                                 <varType>TimeVariable</varType>
298                             </timeVar>
299                         </relationalCondition>
300                     </atomicCondition>
301                 </assertionCondition>
302             </WrappedCondition>
303         </postcondition>
304     </Guaranteed>
305 </Assertion>
306 </SecurityPropertyAssertions>
307 </model:CertificationModel>

```

Listing A.3 Apache Spark Client Task Properties

```

1  package org.springframework.cloud.task.app.spark.client;
2
3  import javax.validation.constraints.NotNull;
4  import org.springframework.beans.factory.annotation.Value;
5  import org.springframework.boot.context.properties.ConfigurationProperties;
6
7  @ConfigurationProperties("spark")
8  public class SparkClientTaskProperties {
9
10     private String master = "local";
11
12     @Value("${spring.application.name:sparkapp-task}")
13     private String appName;
14
15     private String appClass;
16
17     private String appJar;
18
19     private String[] appArgs = new String[]{};
20
21     private String resourceFiles;
22
23     private String resourceArchives;
24
25     private String executorMemory = "1024M";
26     /*
27     Security property to be monitored
28     */
29     private String securityProperty;
30
31     public String getMaster() {
32         return master;
33     }
34
35     public void setMaster(String master) {

```

```
36         this.master = master;
37     }
38
39     public String getAppName() {
40         return appName;
41     }
42
43     public void setAppName(String appName) {
44         this.appName = appName;
45     }
46
47     @NotNull
48     public String getAppClass() {
49         return appClass;
50     }
51
52     public void setAppClass(String appClass) {
53         this.appClass = appClass;
54     }
55
56     @NotNull
57     public String getAppJar() {
58         return appJar;
59     }
60
61     public void setAppJar(String appJar) {
62         this.appJar = appJar;
63     }
64
65     public String[] getAppArgs() {
66         return appArgs;
67     }
68
69     public void setAppArgs(String[] appArgs) {
70         this.appArgs = appArgs;
71     }
72
73     public String getResourceFiles() {
74         return resourceFiles;
75     }
76
77     public void setResourceFiles(String resourceFiles) {
78         this.resourceFiles = resourceFiles;
79     }
80
81     public String getResourceArchives() {
82         return resourceArchives;
83     }
84
85     public void setResourceArchives(String resourceArchives) {
86         this.resourceArchives = resourceArchives;
87     }
88
89     public String getExecutorMemory() {
90         return executorMemory;
91     }
92
93     public void setExecutorMemory(String executorMemory) {
94         this.executorMemory = executorMemory;
95     }
96
97     public String getSecurityProperty() {
98         return securityProperty;
99     }
```

```

100
101     public void setSecurityProperty(String securityProperty) {
102         this.securityProperty = securityProperty;
103     }
104 }

```

Listing A.4 Velocity template for the Apache Spark submit command

```

1 #set( $eventCaptor = "/home/abfc149/toreador-demo/captors/Data" + $config.getSecurityProperty() + "EverestEventCaptors.jar="
2 emitter=socket,host=10.207.1.103,port=10333,eventType=TEXT" )
3 #!/bin/bash
4 spark-submit \
5 --name $config.getAppname() \
6 --class $config.getAppClass() \
7 --master $config.getMaster() \
8 ## Load the appropriate Java agent on the driver
9 --conf "spark.driver.extraJavaOptions=-javaagent:$eventCaptor" \
10 ## Load the appropriate Java agent on every executor
11 --conf "spark.executor.extraJavaOptions=-javaagent:$eventCaptor" \
12 --deploy-mode client \
13 $config.getAppJar() \
14 #foreach($argument in $config.getAppArgs())#if($foreach.first)$argument#end#if(!$foreach.first) $argument#end#end

```

Listing A.5 Docker compose file for the Spark/Hadoop cluster

```

1 version: '2'
2 services:
3     namenode:
4         image: bde2020/hadoop-namenode:1.1.0-hadoop2.8-java8
5         container_name: namenode
6         volumes:
7             - ./data/namenode:/hadoop/dfs/name
8         environment:
9             - CLUSTER_NAME=test
10        env_file:
11            - ./hadoop.env
12        ports:
13            - 50070:50070
14        datanode:
15            image: bde2020/hadoop-datanode:1.1.0-hadoop2.8-java8
16            depends_on:
17                - namenode
18            volumes:
19                - ./data/datanode:/hadoop/dfs/data
20            env_file:
21                - ./hadoop.env
22            ports:
23                - 50075:50075
24        spark-master:
25            image: bde2020/spark-master:2.1.0-hadoop2.8-hive-java8
26            container_name: spark-master
27            ports:
28                - 8080:8080
29                - 7077:7077
30            env_file:
31                - ./hadoop.env
32            volumes:
33                - ../code/AnonymizeData/target/scala-2.11/anonymizedata_2.11-0.1.0-SNAPSHOT.jar:/data/bda/anonymizedata_2.11-0.1.0-SNAPSHOT.jar
34                - ../code/PrepareData/target/scala-2.11/preparedata_2.11-0.1.0-SNAPSHOT.jar:/data/bda/preparedata_2.11-0.1.0-SNAPSHOT.jar
35                - ../code/ComputeAverage/target/scala-2.11/computeaverage_2.11-0.1.0-SNAPSHOT.jar:/data/bda/computeaverage_2.11-0.1.0-SNAPSHOT.jar
36                - ../code/captors/DataPrivacyEverestEventCaptors/target/DataPrivacyEverestEventCaptors.jar:/data/captors/DataPrivacyEverestEventCaptors.jar
37                - ../code/captors/DataIntegrityEverestEventCaptors/target/DataIntegrityEverestEventCaptors.jar:/data/captors/DataIntegrityEverestEventCaptors.jar

```

```

38     - ../code/captors/DataAvailabilityEverestEventCaptors/target/DataAvailabilityEverestEventCaptors.jar:/data/captors/DataAvailabilityEverestEventCaptors.jar
39     - ../code/EventCaptor/target/EventCaptor.jar:/data/captors/EventCaptor.jar
40     spark-worker:
41       image: bde2020/spark-worker:2.1.0-hadoop2.8-hive-java8
42       depends_on:
43         - spark-master
44       environment:
45         - SPARK_MASTER=spark://spark-master:7077
46         - SPARK_WORKER_CORES=1
47         - SPARK_WORKER_MEMORY=2g
48       env_file:
49         - ./hadoop.env
50       volumes:
51         - ../code/AnonymizeData/target/scala-2.11/anonymizedata_2.11-0.1.0-SNAPSHOT.jar:/data/bda/anonymizedata_2.11-0.1.0-SNAPSHOT.jar
52         - ../code/PrepareData/target/scala-2.11/preparedata_2.11-0.1.0-SNAPSHOT.jar:/data/bda/preparedata_2.11-0.1.0-SNAPSHOT.jar
53         - ../code/ComputeAverage/target/scala-2.11/computeaverage_2.11-0.1.0-SNAPSHOT.jar:/data/bda/computeaverage_2.11-0.1.0-SNAPSHOT.jar
54         - ../code/captors/DataPrivacyEverestEventCaptors/target/DataPrivacyEverestEventCaptors.jar:/data/captors/DataPrivacyEverestEventCaptors.jar
55         - ../code/captors/DataIntegrityEverestEventCaptors/target/DataIntegrityEverestEventCaptors.jar:/data/captors/DataIntegrityEverestEventCaptors.jar
56         - ../code/captors/DataAvailabilityEverestEventCaptors/target/DataAvailabilityEverestEventCaptors.jar:/data/captors/DataAvailabilityEverestEventCaptors.jar

```

A.1 Spring Cloud Data Flow

A.1.1 Overview

Spring Cloud Data Flow is a programming model that aims at the development and deployment of cloud applications. It offers the ability to define, deploy and eventually execute composable micro-services on a variety of runtimes. Spring Cloud Data Flow, facilitates the creation and orchestration of data pipelines for a set of pre-defined use cases such as data ingestion, real time data processing and batch processing. On top of that, it offers the ability to create custom data processing modules to meet the requirements for custom use-cases that can not be implemented with the built-in processing components.

From an implementation perspective, streaming and batch processing modules are Spring Boot applications that are independent deployment units and can get executed on resource and container management systems such as Apache YARN [136], Apache Mesos [67], and Kubernetes [34]. In summary, Spring Cloud Data Flow offers to its users an assortment of programming models and best practices for the development of service-based distributed streaming and batch data workflows. Spring Cloud Data Flow comes bundled with a domain specific language, command line shell and a RESTful API to enable the definition of service pipelines. The diversity of communication with Spring Cloud Data Flow makes integration with other platforms effortless and intuitive. Users can choose to use the UI directly, invoke the shell command line client or the RESTful API.

Spring Cloud Data Flow uses HTTP as a transport for its management dashboard. It also supports HTTPS communication. This is an important feature that enables the secure connection of users on the platform and also protects the transmission of data by means of using digital certificates.

A.1.2 Application Types

A core domain module that Spring cloud Data Flow uses to describe composable microservices is applications. Applications can be of 4 different types namely *source*, *processor*, *sink* or *task* and can be compiled into linear pipelines that move data from a source to a sink, optionally with one or more processors types of applications in between. Modules that can not be modeled into as a source, processor or sink can be custom tasks which may be any process that does not run indefinitely e.g. an Apache Spark [155] batch job.

The default configuration comes with a set of pre-built applications that can be used out-of-the-box. However, custom applications can be added through the dashboard management user interface (UI) or the command line client. Spring Cloud Data Flow stores applications in an internal application registry which integrates seamlessly with maven repository managers or Java archive files that stored on a local or remote filesystem. Once applications are installed in the application catalogue, they can be used for the composition of service pipelines that in Spring Cloud Data Flow vernacular are called tasks and are not to be confused with application tasks described above.

Figure A.8 illustrates the addition of a task application in the application catalogue while figure A.9 demonstrates how a list of available applications is presented to the user. Finally, figure A.10 shows an example of a pipeline i.e. task that is composed from the applications available.

A.1.3 Workflow Specification Language

To enable the definition of service pipelines, Spring Cloud Data Flow propose the usage of a domain specific language (DSL). This specification is part of the framework itself. To implement this feature, a special type of application namely a *composed task runner* is used. The composite service pipeline is a graph that is described in a declarative manner by the

DSL. Each node in the graph represents an instance of a task that is associated with an application.

The composite task runner parses the graph DSL and instantiates the execution of the pipeline. As soon as the execution commences, it monitors the progress of execution for the individual tasks. Internally this is achieved by means of using a database table where the execution state of every task is stored during runtime. At regular intervals the task runner will poll the task execution status database table to inspect the state of the tasks and will act according to what has been described in the pipeline DSL definition. If a task is successfully completed the control of execution moves on to the next task until all the tasks of the graph are completed. If a task fails to complete, the task runner takes action on the basis of the sequence of tasks defined in the DSL.

The composed Task Runner is a Spring Batch [8] application that executes the task pipeline. Nodes in the graph represent a *Step*. A Step is an independent unit of work and represents a sequential stage of a batch job. Every step in the graph makes a RESTful call to the Spring Cloud Data Flow Server and instructs it to execute the individual tasks. Part of the DLS specification is the definition of *maxWaitTime* property that dictates for how long the task runner will wait until a task is completed before labelling it as failed. If *maxWaitTime* has elapsed, the composite task runner will throw an **TimeoutException**. Table A.1 shows the values for the **ExitStatus** for all possible task statuses.

Status description	Status value
<i>ExitMessage</i> is set and the task has not failed	ExitMessage
<i>ExitMessage</i> is not set and the task has completed successfully	0
<i>ExitMessage</i> is not set and the task has failed	1

Table A.1 Status values for Spring Cloud Data Flow tasks

Sequences

The Composed Task Runner is able to execute tasks in a sequential order. Tasks have a unique name when defined through the web UI or the RESTful API and therefore it can be used to refer to tasks when defining the pipeline. The **&&** symbol is an operator that can

be applied between two different or identical tasks to denote the sequential execution of the tasks. E.g. if we have tasks **TaskA**, **TaskB** and **TaskC** the sequential execution of the tasks would look like this:

```
(TaskA) && (TaskB) && (TaskC)
```

Bellow in Figure A.1 we present a visual representation of the pipeline:

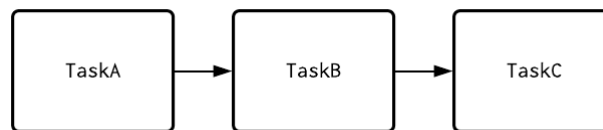


Fig. A.1 Pipeline of tasks executed in sequence

Parentheses are not necessary however they make a clear distinction between the tasks. If any of the tasks will return an ExitStatus of 'FAILED', all the subsequent tasks will not be launched. E.g. for the Composed Task Runner **(TaskA) && (TaskB) && (TaskC)**, if **TaskB** fails, task **TaskC** will not be executed.

In the case where the same tasks are launched multiple times the DSL specification would look like this:

```
(TaskA) && (TaskA) && (TaskA)
```

Bellow in Figure A.2 we present a visual representation of the pipeline where **TaskA** is launched repeatedly 3 times:



Fig. A.2 Pipeline of tasks executed in sequence

Transitions

When defining a Composed Task Runner it is possible to describe the control flow in case one or more task fail to execute successfully i.e. the ExitStatus for the task is 'FAILED'. By

default, as described above in section A.1.3, when tasks fail to complete successfully, the whole pipeline grinds to halt. With the assistance of transitions it is possible to allow the execution of the pipeline to processed even if a task fails. Transition achieve this by means of describing how the execution will continue if a specific task fails. Therefore, task execution is dependant on the ExitStatus of its previous task. En example of a pipeline with transitions can be seen below:

```
(TaskA) 'FAILED' -> (TaskB) 'COMPLETED' -> (TaskC)
```

In the case above **TaskA** will get executed initially and if it fails then **TaskB** will be launched. A visual representation of the workflow shown above can be seen in figure A.3.

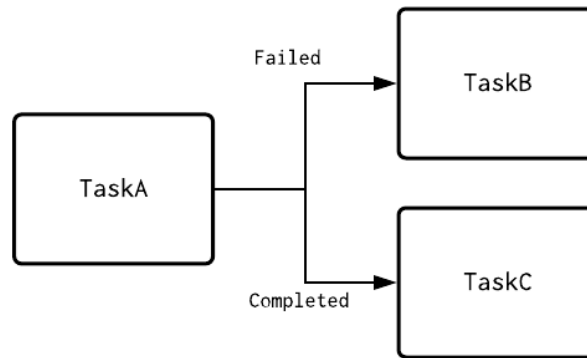


Fig. A.3 Pipeline of tasks with transitions for simple tasks

If TaskA completes successfully then **TaskC** will get executed. A transition can also proceed the sequential execution of a set of tasks. An example is presented below:

```
(TaskA) 'FAILED' -> (TaskB) && (TaskC) && (TaskD)
```

If **TaskA** fails then **TaskB** will get executed but tasks **TaskC** and **TaskD** will not get executed. However, if task **TaskA** completes successfully then **TaskC** and **TaskD** will get executed in a sequential manner. A visual representation of the this scenario is shown in figure A.4.

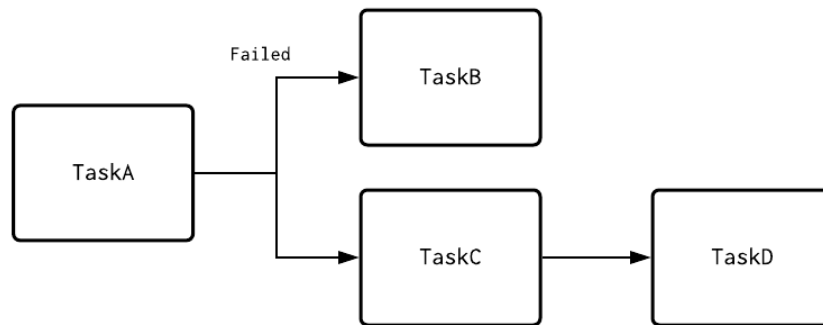


Fig. A.4 Pipeline of tasks with transitions before a sequence of tasks

Wildcards

The DSL pipeline specification language also supports the definition of wildcards to match more than one types of transitions. Wildcards can be a status replacement for any `ExitStatus` except for 'FAILED'. An example can be seen below:

```
TaskA 'FAILED' -> TaskB '*' -> TaskC
```

If **TaskA** gets executed and an **ExitStatus** other than FAILED is returned, Spring Cloud Data Flow will launch **TaskC**. Figure A.5 illustrates the pipeline above.

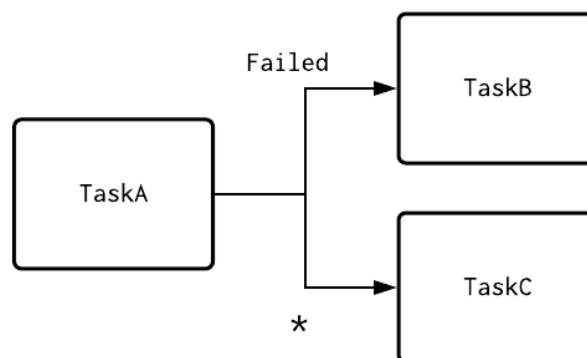


Fig. A.5 Pipeline of tasks with wildcards

Splits

Splits allow for the parallel launching of tasks. For instance:

```
<TaskA || TaskB || TaskC>
```

In the scenario shown above tasks **TaskA**, **TaskB** and **TaskC** will run in parallel due to the `||` operator. A visual representation of the parallel execution of tasks can be seen in figure A.6. Note that all tasks that get launched in parallel are enclosed in a dotted rectangle.

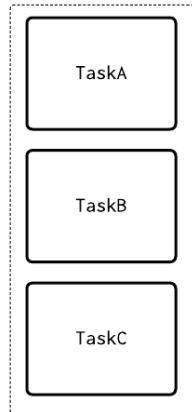


Fig. A.6 Pipeline of tasks launched in parallel

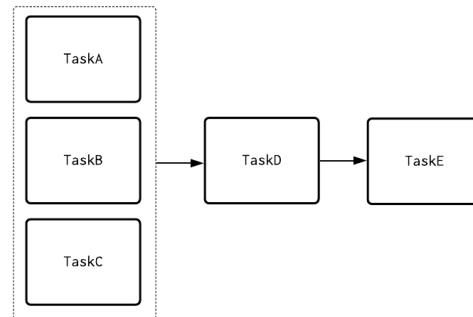


Fig. A.7 Pipeline of tasks launched in parallel that are connected to a sequence of tasks

If a split is part of a composite task then all individual tasks need to complete successfully before the computation moves on to the subsequent tasks. An example can be seen below:

```
<TaskA || TaskB || TaskC> && TaskD && TaskE
```

TaskA, **TaskB** and **TaskC** must complete successfully for tasks **TaskD** and **TaskE** to get launched in sequence. If any of the tasks **TaskA**, **TaskB** or **TaskC** fail, the execution flow will be interrupted with tasks **TaskD** and **TaskE** being ignored.

A.1.4 Application for the Execution of Apache Spark Jobs

For the purposes of this thesis, we implemented a custom Spring Cloud Data Flow application that is responsible for the execution of Apache Spark programs. Each Apache Spark program is associated with a separate task. Multiple such tasks can be organised into a pipeline of tasks with the help of the DSL described in section A.1.3.

The Spring Cloud Data Flow application is designed with the intention to be as generic as possible. To avoid building a separate application for each Spark application, the application is parameterised with the executable code that has to be submitted on an Apache Spark

< Add Application(s)

You can **import** or **register applications** your application(s).

- Register one or more applications** coordinates by entering a Name, Type and URI of the application.
- Bulk import application** coordinates from an HTTP URI location.
- Bulk import application** coordinates from a properties file.

Register one or more applications

Register one or more applications by entering a **Name**, **Type** and **App URI** of the application Jar. You can also provide an optional **metadata artifact URI**. The App URI & the Metadata Artifact URI are typically provided using the Maven coordinates of the Jar but can be a local file or a docker image URI.

Name *
Service A

Type *
Task

URI: *
file:///home/abfc149/spark-client/apps/spark-client-task/target/spark-client-task-1.3.1.BUILD-SNAPSHOT.jar
e.g. maven://io.spring.cloud.scdf-sample-app-jar:1.0.0.BUILD-SNAPSHOT

Metadata URI:
maven://io.spring.cloud.scdf-sample-app-jar:metadata:1.0.0 [OPTIONAL]

☐ Force, the applications will be imported and installed even if it already exists but only if not being used already.

Fig. A.8 Add a new Spring Cloud Data Flow application of type task

Applications

This section lists all the available applications and provides the control to register/unregister them (if applicable).

Actions All types

<input type="checkbox"/>	Name	Type	Uri	
<input type="checkbox"/>	Service A	TASK	file:///home/abfc149/spark-client/apps/spark-client-task/target/spark-client-task-1.3.1.BUILD-SNAPSHOT.jar	<input type="button" value="i"/> <input type="button" value="v"/>
<input type="checkbox"/>	Service B	TASK	file:///home/abfc149/spark-client/apps/spark-client-task/target/spark-client-task-1.3.1.BUILD-SNAPSHOT.jar	<input type="button" value="i"/> <input type="button" value="v"/>
<input type="checkbox"/>	Service C	TASK	file:///home/abfc149/spark-client/apps/spark-client-task/target/spark-client-task-1.3.1.BUILD-SNAPSHOT.jar	<input type="button" value="i"/> <input type="button" value="v"/>

Items per page: 30 | 1-3 applications of 3 applications

Fig. A.9 List of all the installed Spring Cloud Data Flow applications in the application registry

cluster along with the relevant execution parameters that users wants to include. One critical parameters that needs to be highlighted is the security properties that the user requires to be monitored for each individual task. All the required parameters are passed as execution parameters to the Spring Cloud Data Flow application. Internally, the application uses a velocity template to construct a bash script with the Apache Spark submit job command that

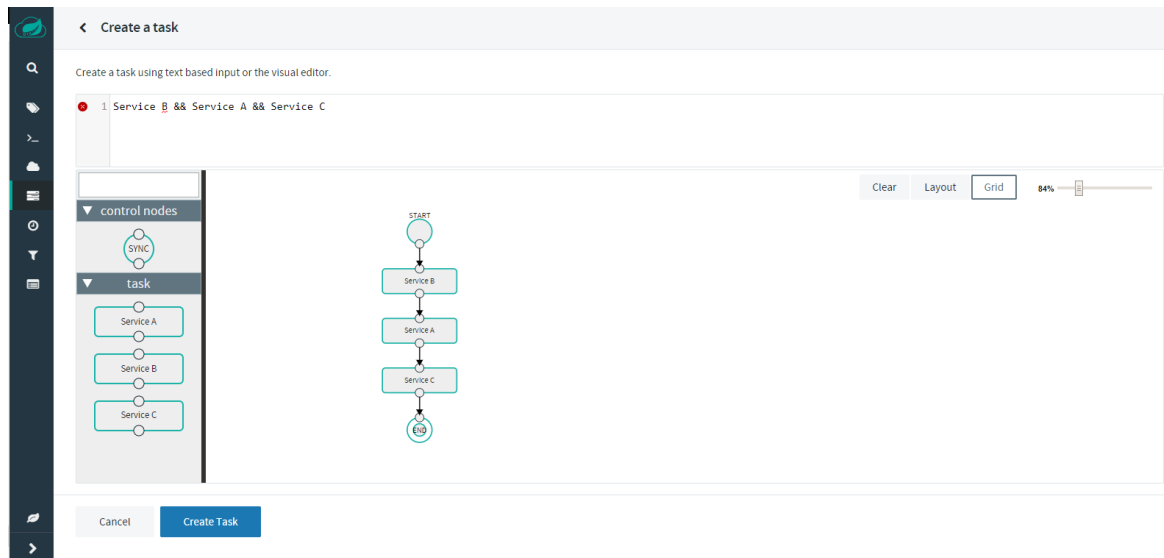


Fig. A.10 Example of a Spring Cloud Data Flow pipeline

can be seen in listing A.4 in Appendix A. The generated bash script is subsequently executed from a relevant task that is part of the pipeline. Spring Cloud Data Flow requires that the task is a Spring Boot application that is presented in listing A.1 in Appendix A. The spark submit command requires a series of parameters to submit a job successfully and can be configured to facilitate different deployment and execution parameters ¹. A list with all the possible parameters that were used can be seen in table A.2

From an implementation standpoint, the application parameters are stored in a Java object that is passed along to the velocity engine to get merged with the bash script template and eventually produce the Spark submit command. The relevant code of the Java class is shown in listing A.3 in Appendix A.

A.2 Apache Spark

Apache Spark is a fast and general-purpose Big Data processing framework. It offers a high-level APIs in Java, Scala, Python and R, and an optimised engine for the execution of directed acyclic graphs (DAGs). In addition, it makes available a rich set of high-level tools such as Spark SQL for SQL and structured data processing, MLlib for the generation of

¹<https://spark.apache.org/docs/latest/configuration.html>

Parameter name	Parameter value
<code>--master</code>	URL for the master node of the Apache Spark cluster. The URL could also represent a local Apache Spark cluster.
<code>--app-name</code>	Name of the Apache Spark application
<code>--app-class</code>	Java or Scala class that where the main() method exist
<code>--app-jar</code>	Location where the executable Java archive (Jar) is stored
<code>--app-args</code>	Arguments that are required from the application
<code>--resource-files</code>	A comma separated list of archieve files to be included in tha application submission
<code>--security-property</code>	A comma separated list of security properties that the user wishes to monitor. Based on the security properties defined all the relevant event captors will be installed at run-time to support the monitoring activity.

Table A.2 Parameters for the Apache Spark submit application registered in Spring Cloud Data Flow

machine learning models, GraphX for the processing of graphs and Spark Streaming for the real time processing on continuous streams of data.

A.2.1 Overview

To provide some context, we need to give an overview of the basic concepts that Apache Spark uses internally when operating on large datasets. In the remainder of this section we give a short description of a series of abstractions that are necessary to describe Apache Spark's architecture. We also provide examples from the API to make it easier for the reader to understand how Apache Spark utilises each of the concepts mentioned.

Resilient Distributed Dataset (RDD)

A resilient distributed dataset, hereafter refered in this thesis as RDD, is a collection of data objects that are distributed across multiple nodes in a cluster. There are two ways to create RDDs; the first one is by means of parallelising an existing collection and the second one is by loading the data from an external datasource such as a distributed file system, a distributed database or a stream of data. In most cases RDDs are loaded from distributed sources to leverage the parallel processing capabilities of the framework. A key feature of

RDDs is that they are immutable. When an operation is applied on an RDD, a new RDD is created as a result of the application of the operation. This is a pivotal feature that enables Apache Spark to recover in the case of failures by means of applying the same operations on the original dataset. Operations that are exposed in Apache Spark's API are created in the image of Scala's collection API. In that way, Apache Spark attempts to abstract away all the implementation details necessary for the parallel execution of operations from the engine and provide to its users a unified high-level API that feels like programming against collections of data that gets processed on a single node.

Partitions

RDDs are broken down into chunks of data called partitions. Each RDD can be comprised of multiple partitions that can be located on the same physical node or a different nodes in the cluster. The number of partitions of an RDD is a critical property of the RDD and play a significant role in the underlying level of task parallelism during data processing. The number of partitions of an RDD can be set programmatically or it can be left to the framework to decide how the data of the RDD is going to split. If the responsibility of defining the number of partitions is assigned to the framework then the data is sliced based on its size in an attempt to produce slices that are similar in size. This enforces the execution of tasks that, at least in theory, will take a similar amount of time to complete. Since RDDs are immutable every time an operation is applied on an RDD there is a parent RDD and a child RDD. The parent RDD represents the data set before the application of the operation and the child RDD represents the data set after the application of the operation. Based on the type of operation, the partitions of the parent and child RDDs can have two basic types of dependencies namely narrow dependencies and wide dependencies. A visual representation of an RDD and its partitions can be seen in figure A.11 whereas a visual representation of an operation applied on a parent RDD that results into a child RDD can be viewed in figure A.12.

Transformations with Narrow Dependencies

Transformations with narrow dependencies are operations that when applied on a parent RDD each partition of the RDD is going to be used at most by one partitions of the resulting child RDD. Typically, this includes transformations that can be applied on the data independently

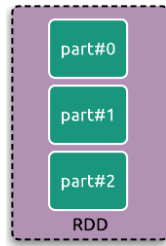


Fig. A.11 RDD with its partitions

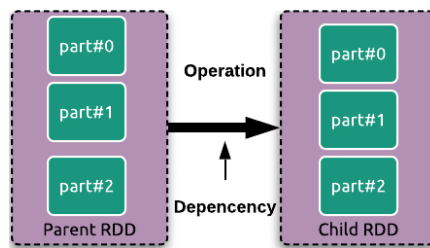
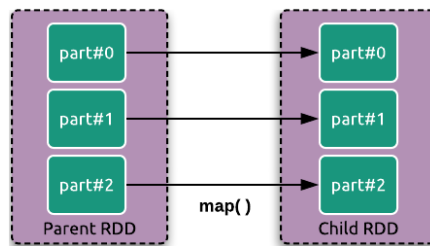


Fig. A.12 Parent and child RDD with applied operation and dependencies

and therefore can be executed in parallel. To optimize things further, Apache Spark can pipeline in-line transformations to improve performance. Examples of such transformations are *map()*, *filter()* and *flatMap()* where the partitions. A visual representation of the *map()* transformation which applies a user defined function to each data item of the parent RDD can be seen in figure A.13

Fig. A.13 *map()* operation - transformation with narrow dependencies

Note how the resulting child RDD has the same number of partitions with its parent RDD. This is a result of the fact that in operations where there is a narrow dependency between

partitions, there a one-to-one mapping between the partitions among the parent and child RDDs.

Listing A.6 Source code for `map()` operation in Apache Spark

```

1  /**
2   * Return a new RDD by applying a function to all elements of this RDD.
3   */
4  def map[U: ClassTag](f: T => U): RDD[U] = withScope {
5      val cleanF = sc.clean(f)
6      new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))
7  }

```

Also note how the returned value from the `map()` operation is a new RDD of type `MapPartitionsRDD`. Function f that is passed as an argument to the transformations is the user defined function that will be applied on all the data items of the RDD that the `map()` operation will be invoked upon.

Transformations with Wide Dependencies

Transformations with narrow dependencies are operations that when applied on a parent RDD each partition of the RDD can potentially be used by none, one or more partitions of the resulting child RDD. Compared to the transformations with narrow dependencies, transformations with wide dependencies are different in that data from multiple partitions of the parent RDD are required to produce a single partition of the child RDD. This is very important in terms of how transformations with wide dependencies are implemented in practice. They require data to be shuffled in an orderly manner to support the combination of data items from more than one partitions. Transformations with wide dependencies are expensive in terms of computational power and network resources and therefore they must be used wisely and only when necessary. Examples of such transformations are grouping operations such as `groupByKey()` and `combineByKey()` where data items with the same key can be located across multiple partitions. Grouping the data items in the reduce phase will require that data items with the same key will have to be shuffled to the same node to complete successfully the grouping operation. A visual representation of a transformation with wide dependencies is shown in figure A.14

Note that the partitions of the resulting RDD will fetch data items from all partitions of the parent RDD for the `groupByKey` operation to be complete i.e. include all the data items with same key from the source RDD. Also note that the resulting RDD is comprised of fewer

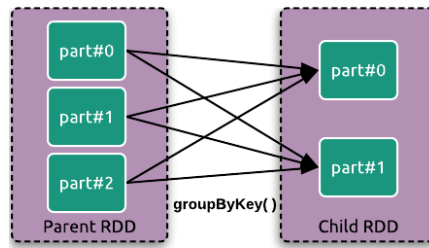


Fig. A.14 *groupByKey()* operation - transformation with wide dependencies

partitions compared to the original RDD. This is not mandatory but it is what happens in most cases. When the data items of the parent RDD are combined usually the number of the resulting data items will be fewer than the number of the parent RDD. This makes it possible to fit them into fewer partitions.

Actions

Actions are operations that produce a result that is either returned to the user or stored in a permanent storage location. As the name implies, actions trigger an actual computation to take place. Transformations, be it with narrow or wide dependencies, are lazy i.e. they are not executed until an action is invoked. When a transformation is defined Apache Spark keeps a reference with respect to what operation should be applied and on what RDD but it does not apply it. Actions are the types of operations that trigger the actual execution of the transformations and enable users to make computations on RDDs. Examples of action operations would be *collect()*, *count()* and *saveAsTextFile()* that return the data items of an RDD, the number of data items of an RDD and save the data of an RDD into a text file in tabular format respectively. A visual representation of an action can be seen in figure A.15

A.2.2 Framework Architecture

In this section we give an overview of all the components that are required to describe Apache Spark's architecture. Figure A.16 illustrates how the components are arranged when Apache Spark operates on big data.

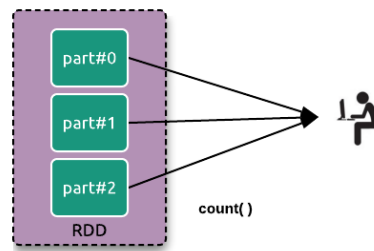


Fig. A.15 *count()* operation - return the number of items on an RDD to the user

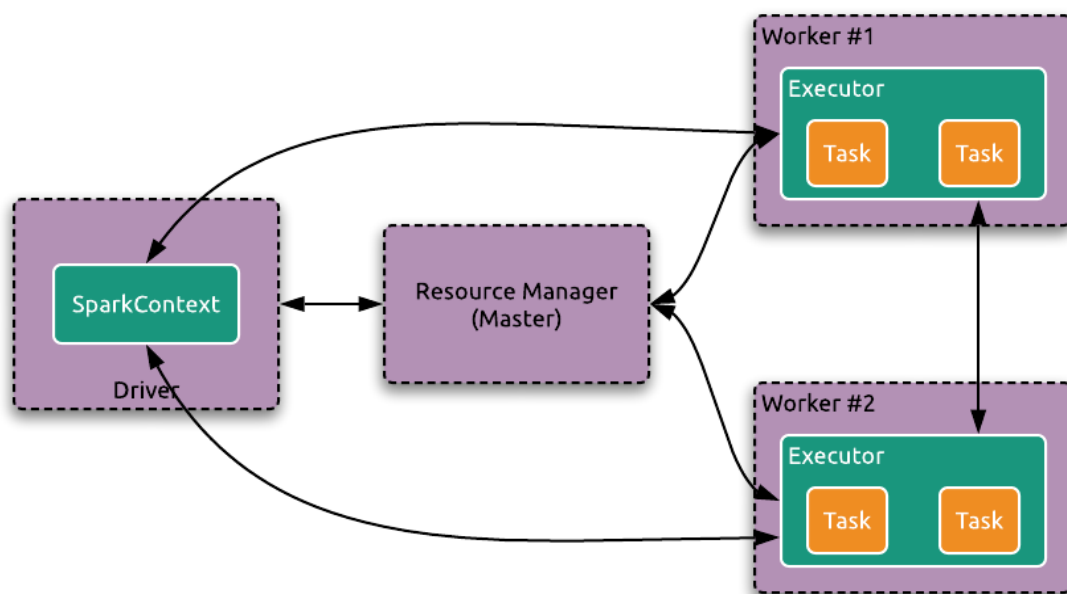


Fig. A.16 Apache Spark overall architecture

Spark Context

The *Spark Context* is the main entry point for all Apache Spark's functionality. It provides a series of useful operations that support the connection to a Spark cluster, and can be used to instantiate new RDDs, accumulators and broadcast variables on that cluster. Each JVM must be associated with one and only one Spark Context. If a new *Spark Context* needs to be instantiated then all the previous ones need to be stopped. In addition, the *Spark Context* is a way to customize the execution parameters of a Spark program.

Directed Acyclic Graph (DAG)

A lineage graph or a Directed Acyclic Graph is a graph that depicts the dependencies of RDDs i.e. what are the parent RDDs for each RDD and what operations need to be executed on each RDD for its child RDD to be produced. In this graph the vertices represent RDDs and the edges represent the operations applied on the RDDs. It is important to highlight that an RDD can be the product of more than one RDDs by means of applying an operation on multiple RDDs. The DAG represents a high level view of all the operations, both transformations and actions, but does not store any information with respect to the number of partitions, where the data should be stored or how many tasks will be instantiated. All that information is related to the execution plan that Apache Spark will employ to compute the DAG and will be taken care of as soon as the execution of the DAG commences from Apache Spark's execution engine. A visual representation of such a graph can be viewed in figure A.17.

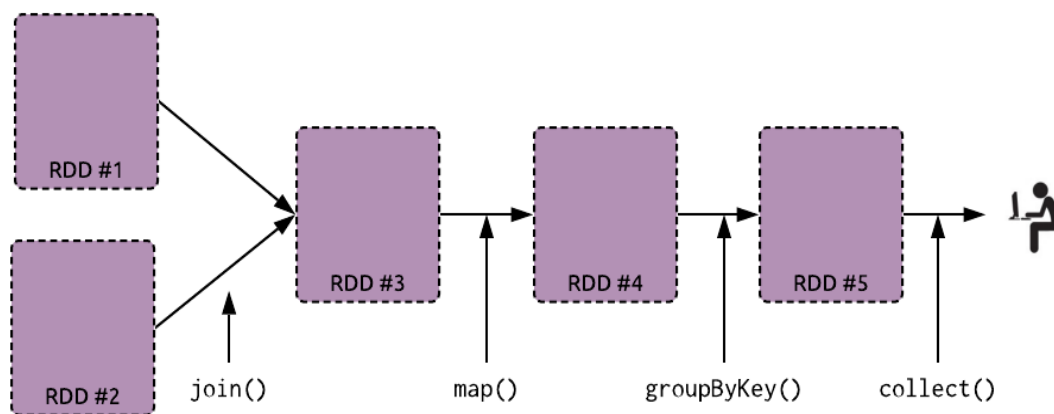


Fig. A.17 Example of a Directed Acyclic Graph (DAG)

Master

The master is a software component of the Apache Spark framework that acts as the resource manager of the underlying cluster. The master is Apache Spark's default resource manager assigned with the responsibility of allocating the necessary resources on the cluster to facilitate the execution of the DAG. The master will negotiate with the cluster's nodes for computational, network and storage resources based on the execution requirements specified

by the user. It will attempt to acquire all the necessary resources for the completion of the computation. If no upper limit is defined with respect to the amount of resources that the user will need to execute the Spark program, Apache Spark will greedily allocate as many resources as there are available.

Apart from the default resource manager that is shipped with Apache Spark, other open source resource managers can be used as well. Apache Spark supports out-of-the-box the usage of YARN [136] and Mesos [67] for the dynamic allocation of resources.

Driver

The driver is a software component of the Apache Spark framework that is responsible for the coordination of execution of the DAG. It is instantiated from the master and ensures that all operations complete successfully. The driver triggers the execution of the operations of the DAG, monitors their execution and guarantees that the operations will execute in the order that they have been defined. If any of the operations fail, the driver is responsible for attempting to re-submit them for execution until the maximum number of allowed re-executions has been reached.

Worker/Executor

The workers or executors are software components that run on the nodes of the cluster and are responsible for the actual execution of operations delineated in the DAG. From an implementation standpoint, they are implemented as JVMs that are dynamically instantiated with the assistance of the installed resource manager and within them Apache Spark can spawn tasks to facilitate the execution of transformations and actions. A key thing to note is that multiple executors can run on the same physical machine of the cluster. The number of executors is dynamic and can vary based on the size of the data that is being processed or on the parameters defined from the user when submitting a DAG of operations for execution to the master. The dynamic allocation and instantiation of the executors is a pivotal feature of Apache Spark's dynamic execution model and is fundamental for its ability to scale not only up but also across.

A.2.3 Execution Model

As soon as the user code is parsed and the DAG is created, Apache Spark will prepare all the underlying infrastructure by means of communicating with resource manager and will trigger the generation of an execution plan that is created based on the available resources. The execution plan involves the segmentation of the DAG into units of work that can get executed in parallel. Breaking down the user's code into separate chunks enable Apache Spark to achieve the parallelization of operation execution. More specifically, the submitted code is divided into *tasks*, *stages* and *jobs*. All three types of types of units of work are explained in greater detail in the order that they get executed below.

Tasks

A task is the smallest unit of work that operates on data. More specifically, a task operates on a partition of an RDD. Conceptually, when a transformation is applied on an RDD what we mean is that multiple tasks that are implementations of the transformation run in parallel on partitions of the RDD. Tasks that refer to transformations with narrow dependencies can be pipelined together and get executed in a single step. In Apache Spark's API there exist two types of tasks namely *ShuffleMapTask* and *ResultTask*. Tasks of type *ShuffleMapTask* get executed within a stage and produce data that is ready to be shuffled and passed along to the next stage. Conversely, tasks of type *ResultTask* get executed within a stage but are instead passed along to the driver and subsequently returned to the user.

Stages

A stage is a unit of execution that comprises of multiple tasks that each one of them can act upon the partitions of an RDD. All the tasks within a stage run in parallel. A Spark service can have multiple stages. The number of stages in a Spark service is defined by the number of shuffle dependencies. The boundaries of a stage are marked by wide transformations that require multiple partitions of an RDD to be combined.

Jobs

A job is the most top-level unit of work of an Apache Spark program. A job is called upon an RDD and signifies the execution of an action on the specific RDD. Jobs or actions trigger the computation of the RDD that they are called upon which in turn will result in the computation of all its previous RDDs. A job usually contains a set of stages that in turn contain a set of tasks.

Elaborating further on the DAG shown in figure A.17, we give an overview of the execution plan that is produced and can be viewed in figure A.18.

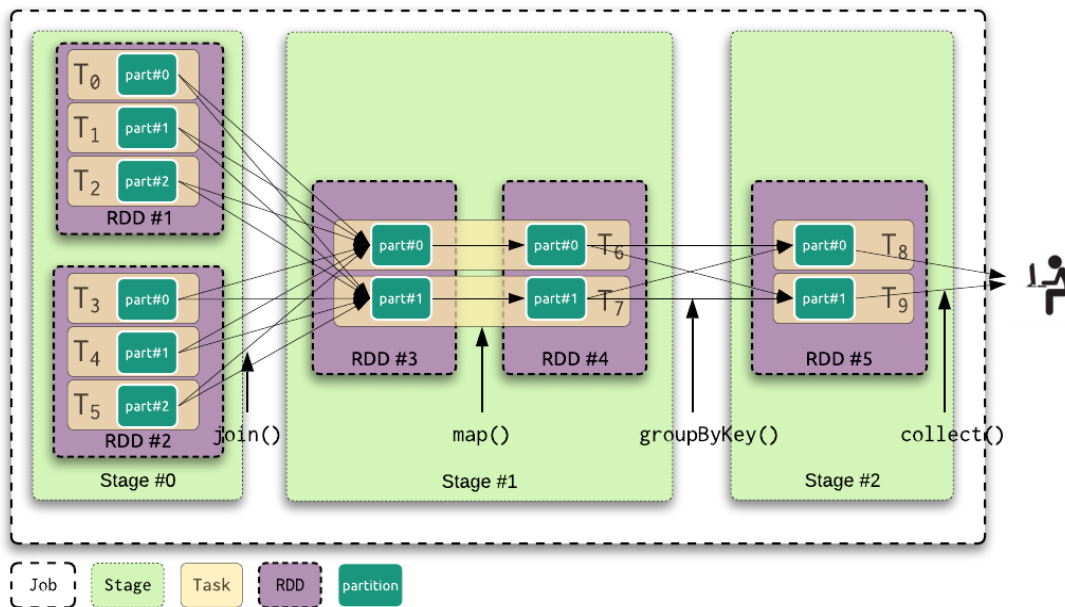


Fig. A.18 Example of a Directed Acyclic Graph (DAG) with jobs, stages, tasks, RDDs and partitions

Both `join()` and `groupByKey()` are operations with wide dependencies and therefore require data to be shuffled. As such, both operations denote the boundaries of a stage within which all tasks will have to complete before the computations moves on to the next stage. Stages are executed in the order that they are presented in the DAG. In the example presented in figure A.18 all tasks in stage 0 and 1 are of type *ShuffleMapTask* because `join()` and `groupByKey()` are transformations. Tasks in stage 2 are of type *ResultTask* because `collect()` is an action.

A.2.4 Deployment Model

Apache Spark programs can be deployed in two different deployment modes. The first one is in client mode and second one is in cluster mode. In client mode the driver application is instantiated on the machine where the *spark-submit* command is executed. This mode is interactive and feedback is sent back to the Spark code submitter from the command line. When applications are deployed in client mode it is critical for the driver to run as close the workers as possible to facilitate their efficient communication. Alternatively, in cluster mode the driver application is launched outside the cluster i.e. on a node that is part of the cluster and is decided from master to where the application is submitted. Application submission takes place by means of executing the *spark-submit* command which in turn makes use of Apache Spark's REST API for application submission. In cluster mode, by definition, the driver is launched within the cluster and therefore is close to the workers that will execute the tasks. To inspect the progress of execution for an application the user can visit Apache Spark's Web UI where runtime information can be seen both for the driver and workers.

A.3 EVEREST

EVEREST is an event reasoning toolkit that enables the evaluation of events against a set of event calculus [91] formulae. Internally it uses a reasoning engine where the unification of the event calculus formulae takes place, a database where the monitoring events and rule violations get persisted and finally a publish-subscribe event bus to collect the events from the event captors. As soon as the events become available from the event captors that have subscribed to the event bus, they become available to EVEREST as well and it can read them immediately. All the event calculus rules are translated into EC-Assertion expressions, EVEREST's event calculus specification language written in XML.

A.3.1 Event Calculus

Event calculus (EC) is a temporal first order logic that allows the association between events by means of applying temporal constraints. It enables the representation of the events and how they might influence the system or other events and provide a very powerful formalism

for the representation and reasoning of the behaviour of dynamic systems. EC formulae are expressed with the assistance of **events**, **fluents** and **time**. Events are action that can change the system's state. Fluents are variables that can be alter as time goes on. Finally time can be thought as a representation of concrete points in time. To enable the description of a system, EC supports the following predicates:

Additional to the predicates presented in table A.3, a series of axioms also are available for the evaluation of fluents. These axioms are the following:

1. **Axiom 1** - a fluent f is clipped i.e. it no longer holds true, if some event e happens at some point in time t that is in-between t_1 and t_2 and the fluent f is terminated at that point in time t
2. **Axiom 2** - a fluent f is declipped i.e. it is initialised and holds true at some point in time t that is in-between t_1 and t_2 , if an event e takes place at time t and the fluent f is initialised at that point in time t
3. **Axiom 3** - a fluent f holds true at some point in time t , if it was initially true and it wasn't clipped between time 0 and t
4. **Axiom 4** - a fluent f holds true at some point in time t_2 , if an event e has happened at some point in time between t_1 and t_2 , that initiated the fluent f at time point t_1 , and f has not clipped between time points t_1 and t_2

Predicate	Description
$Happens(e,t)$	An event e takes place at some point in time t
$Initiates(e,f,t)$	A fluent f holds true when an event e takes place at some point in time t
$Terminates(e,f,t)$	A fluent f does not hold true when an event e takes place at some point in time t
$Initially(f)$	A fluent f holds true from time $t = 0$
$HoldAt(f,t)$	A fluent f holds true at some point in time t

Table A.3 Event Calculus list of predicates

Axiom 1
$\exists e, t, Happens(e, t, R(t_1, t_2)) \wedge Terminates(e, f, t) \rightarrow Clipped(t_1, f, t_2)$
Axiom 2
$\exists e, t, Happens(e, t, R(t_1, t_2)) \wedge Initiates(e, f, t) \rightarrow Declipped(t_1, f, t_2)$
Axiom 3
$Initially(f) \wedge \neg Clipped(0, f, t) \rightarrow HoldsAt(f, t)$
Axiom 4
$\exists e, t_1, Happens(e, t_1, R(t_1, t_2)) \wedge Initiates(e, f, t_1) \wedge \neg Clipped(t_1, f, t_2) \rightarrow HoldsAt(f, t_2)$
Axiom 5
$\exists e, t_1, Happens(e, t_1, R(t_1, t_2)) \wedge Terminates(e, f, t_1) \wedge \neg Declipped(t_1, f, t_2) \rightarrow HoldsAt(f, t_2)$
Axiom 6
$HoldsAt(f, t_1) \wedge (t_1 < t_2) \wedge \neg Clipped(t_1, f, t_2) \rightarrow HoldsAt(f, t_2)$
Axiom 7
$\neg HoldsAt(f, t_1) \wedge (t_1 < t_2) \wedge \neg Declipped(t_1, f, t_2) \rightarrow \neg HoldsAt(f, t_2)$

Table A.4 Event Calculus axioms

5. **Axiom 5** - a fluent f does not hold at some point in time t_2 , if an event e happened at some point in time between t_1 and t_2 that terminated the fluent f and the fluent has not been declipped between the t_1 and t_2 time points
6. **Axiom 6** - a fluent f holds at some point in time t_2 , if it held true at time point t_1 , where t_1 was before t_2 and if fluent f was not clipped between between t_1 and t_2 .
7. **Axiom 7** - a fluent f does not hold at some point in time t_2 , if it did not held true at some point in time t_1 that is before t_2 and if the fluent f has not been declipped at any point in time between t_1 and t_2

A.3.2 Framework Architecture

Internally, EVEREST is comprised of three modules namely the **monitor manager**, the **monitor** and the **event collector**. The monitoring manager is the component responsible for initiating, coordinating and reporting the results of the monitoring process. As such, it

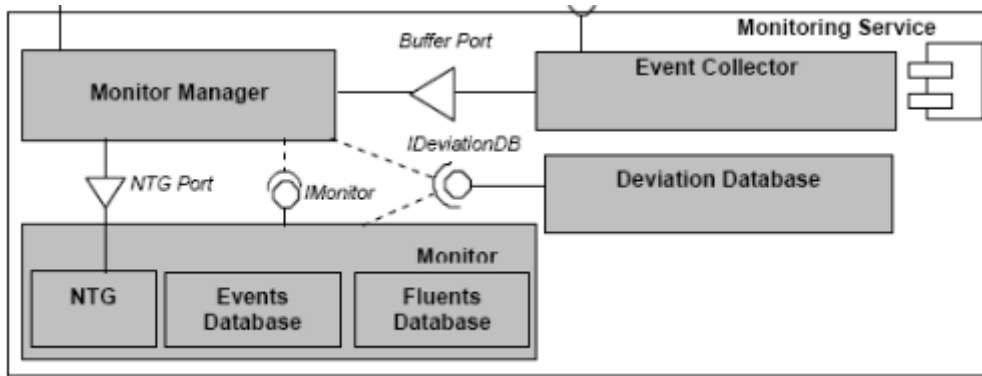


Fig. A.19 EVEREST framework architecture [84]

receives the monitoring rules from the SLA Manager web application and provides the API for obtaining monitoring results. The event collector is responsible for receiving events from the event captors and subsequently passes them to the monitoring manager. The monitoring manager forwards these events to the Native Type Generator (NTG) sub-component of the monitor, which translates the events from EC-Assertion formulae to internal Java objects. After receiving events from the manager, the monitor checks the events against the previously loaded rules. When a violation of a monitoring rule is detected, the monitor records it in a deviation database alongside the events that triggered the rule violation.

A visual representation of the framework's application can be seen in figure A.19

A.4 Apache Velocity

Apache Velocity [64] is a general purpose Java-based template engine. It is used from software engineers and IT practitioner to produce different formats of files by means of merging templates with programmable artefacts written in Java. It can be used to automate the generation of files in many different output formats. Instances of such files typically include HTML web pages, SQL script files or code snippets, configuration files and XML files. The framework supports basic control structures both for conditionals and iterations. It also supports more custom programming structures called macros where custom user logic can be formalised and used as Velocity components.

A.4.1 Overview

The Apache Velocity framework is comprised of two primary components namely *templates* and the *context*. Templates are the place where we define the general structure based on which the generated file will be created. One can think about templates as a blueprint where specific areas of the file have not been define. Specific sections of the template is filled in with placeholders that will get populated with actual values during the merging phase with the programmable artefacts. The context represents the place where all the information that will be interpolated in the templates. This configuration is flexible and provides a systematic way of populating sections of the template with dynamic content.

A.4.2 Velocity Template Language

Templates in Apache Velocity use a template language (VTL) to describe the parts of the template that will be interpolated with values from the context. The purpose of the Velocity Template Language (VTL) is to put forward an easy, simple and clean method for the injection of dynamic content in files that are produced from templates. VTL is simple in syntax but provides a set of powerful constructs for conditionals and loops. It uses references to incorporate dynamic content in a template, with variables being one such type. A variable in a template can cite a variable defined in the Java code of the context, or it can get its value from a VTL statement in the template itself. Tables A.5 and A.6 provide a comprehensive list of the programming structures that are part of the VTL specification. All the available commands have been grouped into two categories namely references and directives.

References

Variables
Create a variable and assign an actual value to it. Any reference to this variable within the template will return the value stored in it. E.g. <code>#set (\$variable="An actual value")</code>
Properties

Get reference to the fields of the Java POJO. This can be used to get the value of the property or assign to it a new value.

```
$obj.property
```

\$obj refers to an instance of a Java object that is passed along as a reference from the context to the template

Methods

Invoke a method in the Java POJO that is merged with the template. E.g.

```
$obj.method()
```

\$obj refers to an instance of a Java object that is passed along as a reference from the context to the template

Table A.5 Velocity template language references

Directives

Conditionals

#if, #elseif and #else directives provide a way to generate the content based on certain conditions. E.g.

```
#if($pojo.field == "value-1")
    <tag> $pojo.field </tag>
#elseif($pojo.field == "value-2")
    <tag> $pojo.field </tag>
#else
    <tag>No value</tag>
#end
```

Iterations

<p>Directive that enables the iteration over a collection of objects. E.g.</p> <pre><tag> #foreach(\$item in \$items) <value> \$item </value> #end </tag></pre>
Include
<p>Statically import files in the template. E.g.</p> <pre>#include("file.txt","image.png","index.html")</pre>
Parse
<p>Import a local template that uses VTL as well. The template is parsed and all the directives defined in it are available within the scope of the current template.</p> <pre>#parse (myTemplate.vm)</pre>
Evaluate
<p>Directive that allows the dynamic evaluation of a String literal or a reference to a String that contains VTL directives. The evaluate directive will treat this String as template and will evaluate it by replacing all its VTL elements with the appropriate values. Evaluate is a special case of parse where the template comes in the form of a String literal and not from the contents of a file. E.g.</p> <pre>#evaluate(\$variable)</pre>
Break
<p>Stops any further rendering of current execution scope. This is especially useful when running iterations there is to break out of the loop</p>

Stop
Stops any further rendering and execution of the template.
Macros
<p>Directive that allows for the definition of repeatable segments of the template to avoid repetition. Velocity macros can be parameterized to facilitate generalization. E.g.</p> <pre>#macro(macroName \$arg0 \$arg1) <tag> <value>\$arg0</value> <value>\$arg1</value> </tag> #end</pre> <p>The code above defines a macro under the name macroName that takes two arguments. To invoke the macro one should use it like so:</p> <pre>#macroName(4 8)</pre>

Table A.6 Velocity template language directives

A.4.3 Velocity Template Engine

The Apache Velocity template engine is implemented in Java and uses Java objects to make references to template variable and to create directives. Files are being generated by means of merging templates with Java code at runtime. The steps involved are the following:

1. Create a template engine
2. Load the template file which is a file with a *.vm* extension and associate it with the template engine
3. Create Java hash map and populate it with the variables that are referenced in the template

4. Merge the Java hash map with the template and produce the final result i.e. the file that emerges from the template and the relevant variables

Bellow in listings A.7 and A.8 follows code snippets for an example template and the relevant Java code. The example demonstrates how an Apache Velocity template can be used to dynamically produce a file.

Listing A.7 A simple example of a Velocity template

```
1 Hello $name!
```

Listing A.8 Example of Java code for the creation and usage of an Apache Velocity template engine

```
1 import java.io.StringWriter;
2 import org.apache.velocity.app.VelocityEngine;
3 import org.apache.velocity.Template;
4 import org.apache.velocity.VelocityContext;
5 public class VelocityExample {
6     public static void main( String[] args ) throws Exception {
7         /* Initialize the Velocity engine */
8         VelocityEngine engine = new VelocityEngine();
9         engine.init();
10        Template t = engine.getTemplate( "template.vm" );
11        VelocityContext context = new VelocityContext();
12        context.put("name", "World");
13        StringWriter writer = new StringWriter();
14        t.merge( context, writer );
15        System.out.println( writer.toString() );
16    }
```

In listing A.8 above in line 8 the template engine is created and in line 9 it gets initialized. Further down in line 10 the template file is loaded and associated with the template engine. After that, the context is created in line 11 and a variable is put in it in line 12. Then a `StringWriter` object is instantiated in line 13 and the template is merged with the template and the context to store the result in the writer in line 14, Finally, the result is printed in line 15. The final result would be the following:

Hello World!

A.5 Byte Buddy

A.5.1 Overview

Byte Buddy is a library that allows the generation and manipulation of Java classes during the class loading phase of a Java program. An important aspect of Byte Buddy's instrumentation model is that the class generation and manipulation is done without the assistance of the compiler i.e. it does not take place at compile time but at runtime. This is a powerful concept that can be very helpful when using the aspect oriented programming paradigm. An additional key feature of the library is that apart from code instrumentation it also be used to create runtime proxies by means of interface implementation. Byte Buddy is accompanied by a comprehensive API that can be easily combined with Java agents to offers its code instrumentation functionality.

Byte Buddy abstracts away the low level details of byte code that is produced after the compilation of Java code. Its intention is to provide programmatic hooks where the users will inject their code without having to fully understand the intricacies of the underlying JVM. It is written in Java 5 but but it can be used to intercept code for later versions as well. Byte Buddy has been designed with the intention to operate with minimum dependencies. It only depends on a Java byte code parser called ASM which has no further dependencies itself.

An important point to highlight is that Byte Buddy can be used for the instrumentation of code that will execute on a JVM. That implies that it can be used to intercept not only code for Java applications but also for any application that when compiled can run on a JVM. This is possible because Byte Buddy intercepts the code when loaded in the JVM just before it gets executed. JVM languages that fall into that category are Scala, Groovy and Ruby. In fact, for the purposes of our proof of concept implementation we instrument Apache Spark's source code that has been written in Scala.

Figure A.20 gives an overview of the code instrumentation process from Byte Buddy for the interpretation of programs that get executed on a JVM. The generated byte code is sent to the JVM and the instrumented code is executed from the JVM's interpreter and *Just-In-Time* (JIT) compiler.

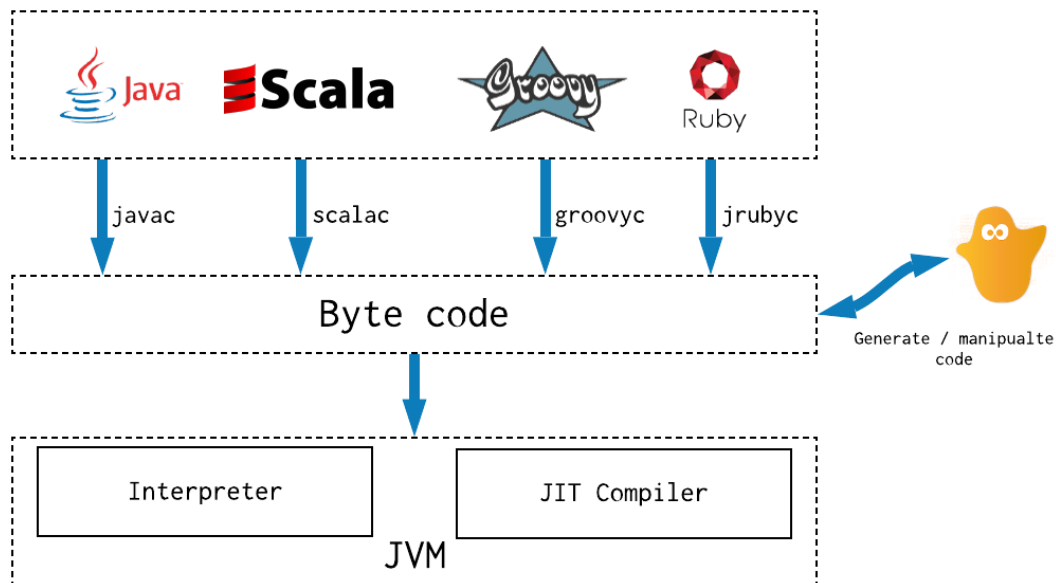


Fig. A.20 Overview of the code instrumentation process from Byte Buddy

A.5.2 Java's Instrumentation API

The ability to instrument Java code has been part of the standard Java API since Java 5. The instrumentation process is supported by Java agents that can be invoked during the execution of byte code on a JVM. By default the JVM allow the installation java agents are passed as options in the command that executes the program. In the case of Java programs one can use the `-javaagent` parameter and pass it as an argument the location where the Java agent is located. Java agents are Jar file that has a specific structure. They require that a manifest file is present where the agent class is defined and they also require that in that class a method called *premain()* is also available. The *premain()* method is invoked before the JVM loads the classes of a Java program and is the location where the logic for the code instrumentation needs to be placed.

Additional to that, Java agent installation can take place not only before the JVM has started executing the code but also after the execution has started. This can be achieved by means of dynamically attaching the Java agent to an application that is already executing on a JVM. The dynamic attachment of a Java agent on a JVM is implementation dependant and is not supported by all JVM implementations.

The command-line options to install a Java agent when running a Java program is:

```
-javaagent:/path/to/agent.jar[=options]
```

Note that the Java agent itself can take as an argument a set of options that can be useful for its internal operation. This is passes as an argument to the *java* command used for the execution of a Java program.

The manifest file of the Java agent has to conform to a certain structure in order for the JVM implementation to be able to apply it successfully during execution. Table A.7 that follows presents a list with all the possible manifest options.

Premain-Class
This option defines the complete name of the class where the <i>premain()</i> is declared and implemented. If a Premain-Class value is not specified the JVM will abort. The value of the Premain-Class parameter is a class name separated with dots following Java's packaging convention and not a path.
Agent-Class
This option specifies the class where the agent should look for the <i>agentmain()</i> which is responsible for the instrumentation of Java applications that have already been started. This feature is useful only in the cases where the JVM does support the dynamic attachment of agent to running JVMs. Similar to Premain-Class, the value of the Agent-Class parameter is a class name separated with dots and not a path.
Boot-Class-Path
This option specifies a space separated list of paths that the bootstrap class loader will have look for additional classes to load if a class fails to get loaded during normal the class loading phase. The paths defined in this option will be inspected one at a time and in the order that they have been provided. From a syntax point of view, paths follow the UNIX convention and if they start with a / then they represent an absolute path whereas if they do not start with a / they represent a relative path.
Can-Redefine-Classes
Define whether the agent can redefine classes. This is an optional value and possible values are true or false.

Can-Retransform-Classes
Define whether the agent can re-transform classes. This is an optional value, possible values are true or false and the default value is false.
Can-Set-Native-Method-Prefix
Define whether the agent can set native method prefix. This is an optional value and possible values are true or false. This is an optional value, possible values are true or false and the default value is false.

Table A.7 Options for manifest file of Java agents

A.5.3 Runtime code instrumentation and Code Generation in Byte Buddy

Create and install a Java agent

Byte Buddy uses the builder pattern [58] to support the creation and installation of a Java agent on the JVM's instrumentation implementation. More specifically it uses a class called *AgentBuilder* that instantiates a default Java agent. Listing A.9 shows a snippet that will create and install a default Java agent.

Listing A.9 Create and install a default Byte Buddy Java agent

```
1 new AgentBuilder.Default().installOn(instrumentation);
```

The code in the listing above creates and installs a Java agent without adding any logic with regards to code instrumentation.

Search for a class or set of classes

Byte Buddy uses element matchers to allow its users to query the pool loaded classes and find the ones that need to be instrumented. This is important because it makes possible the discovery of classes based on several criteria such as name, type, annotation or access modifier. Element matchers can be regarded as predicates and can be applied for matching multiple code artifacts such as types, methods, fields and annotations. Listing A.10 shows a code snippet for

Listing A.10 Use element machers to search for a class or a set of classes to instrument

```

1 new AgentBuilder
2 .Default()
3 .type(type ->type.getName().equals("name.of.class"))
4 .installOn(instrumentation);

```

Transform the method of a class

The transformation of a class method is a powerful concept and Byte Buddy provides to its users an array of tools to give access to all the methods runtime metadata. It supports the runtime access and modification of the method's input and output and it also offers the ability to override the method completely. Method overriding occurs in the form of method delegation where when the original method is invoked the its execution is delegated to the instrumented method. Listing A.11 presents a code snippet where a method is delegated to the method of another class and listing A.12 shows the class and method that will handle the delegation.

Listing A.11 Transform a class method by delegating its execution to another method

```

1 new AgentBuilder
2 .Default()
3 .type(type ->type.getName().equals("name.of.class"))
4 .transform((builder, typeDescription, classLoader, module) -> {
5     return builder.method(
6         method -> method.getName().equals("methodName")).intercept(
7             MethodDelegation.to(new DelegatedClass(type, properties)));
8     .installOn(instrumentation);

```

Listing A.12 Class that contains the delegation method

```

1 public class DelegatedClass {
2
3     @RuntimeType
4     public Object methodName(
5         @Argument(0) Object arg1,
6         @Argument(1) Object arg2,
7         @This Object this,
8         @Morph Morpher<Object> morpher) {
9         /*
10         Code of the delegation method
11         */
12     }
13 }

```

In listing A.11 in line 4 we define a transform method that takes as a parameter a lambda expression which represents a class transformer i.e. in what way the code of the original method should be modified. As it can seen in lines 6 and 7 in the same listing, the class transformation refers to the delegation of the invocation of the original method to a method

called *methodName()* that is defined within a class named *DelegatedClass*. Also note that a set of annotations have been used to decorate certain aspects of the delegation method. More specifically, the **@Argument(*n*)** annotation is a reference to the *n*-th argument of the intercepted method, the **@This** annotation is a reference to the instance of the object of the intercepted method and finally the **@Morph** annotation is a reference to the delegation method. This gives us the ability to invoke the intercepted method with its input modified on the basis of our code instrumentation requirements. This feature of the Byte Buddy library has been used heavily for the implementation of our proof of concept to support the monitoring activity and invoke the intercepted method with modified parameter values.