MASTER

Scalable architecture and platform for data analytics in wearable health solutions

Ekimov, V.

Award date: 2016

Link to publication
 Scalable Architecture and Platform for Data Analytics in Wearable Health Solutions

Victor Ekimov

imec

TU/e Technische Universiteit Eindhoven University of Technology
Scalable Architecture and Platform for Data Analytics in Wearable Health Solutions

Master's Thesis in Business Information Systems

System Architecture and Networking Group
Department of Mathematics and Computer Science
Security and Embedded Networked Systems W&I Section
Eindhoven University of Technology

Victor Ekimov

7th August 2016
Author
Victor Ekimov

Title
Scalable Architecture and Platform for Data Analytics in Wearable Health Solutions

MSc presentation
15th August 2016

Graduation Committee
Prof.dr. J.J. (Johan) Lukkien   Eindhoven University of Technology
Prof.dr. C.M. (Claudia) Chituc   Eindhoven University of Technology
Prof.dr. Yannis Velegrakis   University of Trento
MSc. Bishal Lamichhane   IMEC, Eindhoven
Dr. Pierluigi Casale   Philips Lighting Research, Eindhoven
Abstract

Rapid development of the wearable sensor technology has enabled a large number of healthcare and lifestyle applications. These applications use the continually monitored bodily physiological signals to enable estimation of activity, energy expenditure, stress, mood, etc. Most of these applications are data-driven. The data collected from different individuals are used to train and develop models which relate the input physiological signals to the target parameter of the application. The robustness of a data-driven application grows only with the growth of the data used to train the underlying model. This requires the associated data analytics to shift from the computation on a single machine to distributed computation, to enable a quicker hypothesis testing, development and validation. We observe the big data analytics becoming a necessity for data analytics in wearable health solutions.

Developing a large scale data processing application requires a diverse list of competences on algorithm design, application integration and execution in cluster environment. All these skillsets are not necessarily acquired by every researcher, most of them only specialized in algorithm design. Researchers with no or little background in cloud computing still need a tool to ease the application development process and support for quick testing, validation and deployment of their hypothesis at big data scale. There is a need for a data platform which would automate the application development process and enable a transparent management of developed application on the computational cluster. Such a platform needs to be highly configurable to match to the need of different application and development team.

In this thesis, we propose a data platform, providing an end to end solution for application development process related to data analytics in wearable health solutions domain at big data scale. Applications are built to be run on the computational cluster, allowing large scale distributed computation. The platform allows for the development and management of the application lifecycle easily and transparently. The proposed platform is designed to be highly configurable to support different computational frameworks and execution environment. We demonstrate the use of the platform and its configurability using the application on detection of moments of stress using physiological signals (ECG and accelerometer) measured with wearable sensors.
Preface

I would like to express my sincere gratitude to my tutor, Bishal Lamichhane, for constant support and the ability to explain complex ideas in a simple way. Additional thanks go to Pierluigi Casale for pulling out my application for an internship and making a hiring decision on it. I would also like to thank professor Johan Lukkien for being my supervisor and managing my graduation procedure. Also, I want to thank Claudia Chituc for valuable feedback on the report. I want to thank IMEC for building a Holst Center at High Tech Campus and providing me with an AWS account to play around.

At last, I would like to thank the owner of a bike shop near my house for making a 20 EUR discount, converting my 45 min walking experience into 15 min biking experience.

Victor Ekimov

Eindhoven, The Netherlands
7th August 2016
Chapter 1

Introduction

We have been witnessing the rapid proliferation and use of wearable sensors in the recent years. Wearables enable continuous monitoring of different physiological signals from the human body. This has led to the development of different data-driven healthcare and lifestyle applications. The nature of research has also begun to change. There is a shift from studies where only a few users are monitored for few minutes in a controlled environment towards studies involving free-living monitoring of hundreds of users for multiple days. The latter type of study is an absolute prerequisite for robust data-driven applications using wearable sensors. Thus, more and more researchers are finding themselves working in the big data realm. As such, data management and computation become a non-trivial task of the application development process.

In this work we propose, develop and validate a data platform to support the cloud-based workflow for data-driven research using wearable sensors. The developed data platform allows the applications to be developed, deployed and executed in a transparent and automated way. Such capabilities of the data platform enable the researchers to solely focus on the application logic. Data platform provides an end to end solution for data storage (inputs and outputs of the application) and computation in a scalable way. The applicability of the data platform meets the needs of a wide range of researchers. A researcher well-versed in cloud computing can benefit from the workflow automation. Meanwhile, researchers without the knowledge of cloud computing can work unhindered on big data problems as only the application logic needs to be plugged into the data platform and everything else is managed by the platform. The data platform is developed to be configurable and flexible to support different computing frameworks, machine types and data sources. This serves for the platform to be a single tool to be used across different research teams. Each team is free to instantiate and configure the platform, according to the their application needs. The configurability also helps the team to benchmark their application for optimal configuration and trade-off analysis.

We validate the developed data platform using the application for moments of stress detection. Stress detection from physiological sensors is an active re-
search domain in the wearable health solutions team at IMEC where this work has been conducted. The application uses the data gathered from physiological sensors worn by multiple participants. These data are stored in the persistent storage. From that storage, data platform can query the data as an input for the execution of the application. Main research components for stress detection involve feature extraction from the physiological sensors. These components in the stress detection pipeline are implemented and benchmarked for different configurations of the data platform in our work. One purpose of the benchmark is to validate the configurability property required for the data platform. The second purpose is to provide the insights on optimal data platform configuration to meet the need of computation in the stress detection application pipeline.

In this chapter, we begin with the context described in Section 1.1. Next, we define the goals of our project in Section 1.2. Contributions from this work are summarized in Section 1.3. Finally, we conclude this chapter with an outline of the thesis in Section 1.4.

1.1 Context

Stress detection using physiological signals is an active research topic. An ongoing study at IMEC involves monitoring of hundreds of participants using wearable sensors. In this research, sensors are worn by participants for a period of 5 to 7 days. Data are recorded every millisecond and stored in HDF5\(^1\) file in the form of data frames, queryable by Pandas\(^2\). From each participant, about 1 GB of data is obtained per day of the measurement. The measured signals are ECG\(^3\) and body movements measured by an accelerometer. The main research goal of the study is to detect moments of stress by analyzing the raw signal from the wearable sensors. Computation is performed on the cluster, which is a collection of VMs (Virtual Machines) in the cloud.

The workflow for data collection and analysis is depicted in Figure 1.1. Wearable sensors sample ECG and accelerometer data of the participant (step 1). Then, the data is streamed through Kinesis\(^4\) (step 2) into S3\(^5\) (step 3). This is a breaking point, at which data collection is finalized and data analysis can start. Input data are read from S3 into a client application (step 4). Researchers configure and launch the cluster to support the computation in their application, followed by client application deployment to EMR\(^6\) (step 5). The results of the computation are collected back to the client application (step 6). Finally, results are stored to S3 (step 7).

\(^1\)HDF5 [23] is a data model, library, and file format for managing different datatypes
\(^2\)Pandas [30] is a Python [34] library providing data structures and data analysis tools
\(^3\)Electrocardiogram (ECG) is the signal from the electrical functions of the heart
\(^4\)Amazon Kinesis [5] is a platform for real-time streaming data in the AWS [8] cloud
\(^5\)Amazon Simple Storage Service [7] is an object cloud storage with web service interface
\(^6\)Amazon Elastic MapReduce [3] is a managed Hadoop [13] framework
The current workflow is a time consuming process, involving cluster maintenance and support for interactions between the cluster and a client application. Furthermore, different researchers use different frameworks, either due to the legacy reasons or the scope of their work. It is desirable to develop a single project solution to support the different needs and workflows. Both the algorithm design as well as large scale data collection is of interest here. Therefore, a direction is sought in which data can be collected in a standardized fashion and also algorithms can be deployed easily on top of the data. This thesis proposal intends to achieve the automation of application development, deployment, execution and analysis in order to reduce maintenance complexity of the cluster and increase application development speed.

1.2 Goals

The main goals of this project are:

1. Design a data platform which would automate the application development process for researchers. In particular, it must manage the lifecycle of the client application and the cluster to support computation, including the following: launch the cluster, deploy and execute client application, collect metrics of executing application and terminate the cluster.
2. As a validation of the data platform, implement a client application for stress research to be run upon the data platform using different computation frameworks. The application must compute ECG and accelerometer feature extraction workloads over the data aggregated by the hour. Data are read from HDF5 files, containing ECG and accelerometer recordings.

3. Benchmark client application for stress research in order to determine the best possible cluster configuration for different workloads across the following dimensions: framework (Hadoop \(^7\) or Spark \(^8\)), machine type (memory, compute, storage optimized or general purpose) and datasets size (small or large files). These workloads are: ECG all features, mean heart rate, statistical features and accelerometer features including min, max, mean, median features among the others. This benchmark helps to identify the optimal configuration for the given computational workload and validate the configurability/flexibility aspect of the data platform.

With these goals on the data platform proposal, the following design questions arise:

**DQ1** How to automate application development process for researchers?

**DQ2** What type of a data platform can support multiple computational frameworks inside of one client application?

**DQ3** How to enable transparent management of client application execution on the cluster for computation?

**DQ4** What are the constraints to be followed by the client application in order to be integrated into the data platform?

The data platform proposal raises the following configuration questions:

**CQ1** How to enable a data platform to benchmark the client applications? Such configurable platform should support different framework, computing environment, etc.

**CQ2** What are the performance metrics of the client applications for given workloads across the dimensions of different configuration options?

**CQ3** Which configuration type is best in the particular case of the stress application?

### 1.3 Contributions

The contributions of this thesis are as follows:

\(^7\)Apache Hadoop [13] is an open-source framework for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware

\(^8\)Apache Spark [40] is a fast and general engine for large-scale data processing
• Give an architectural description and implementation of a data platform. The data platform would enable the development of applications based on different frameworks (Hadoop and Spark) while making the execution of the developed application transparent to the user. The entire life-cycle of a computation cluster to support the application execution is managed by the platform so that user is free to focus on application logic. The platform couples a scalable solution for data storage and computation to provide a complete, scalable support for a wide range of applications (Chapter 2).

• Present a client application implementation for the validation of the data platform. The client is developed in both frameworks (Hadoop and Spark) and application execution guidelines are provided. This provides a template for the users of the data platform for integrating their applications onto the platform (Chapter 3).

• Provide an evaluation of the experimental results on moments of stress detection application built using the data platform. Selected components from the pipeline of the algorithm on the moments of stress detection is executed on the platform for various configurations. This demonstrates the platform’s support for different configurations and provides the insights about optimal configuration for this particular application. Ease of use of the automated solution is compared as opposed to the manual approach (Chapter 5).

1.4 Outline

This chapter serves as an introduction to the project, describing the context, goals and contributions.

Chapter 2 gives an architectural description of the data platform. We describe some of the essential components (Cluster Compute and Cluster Deploy) and how they interact together through the client application.

Chapter 3 outlines the client application implementation for stress research as a validation of the data platform described in Chapter 2. In particular, we explain how to integrate the client application with two data platform components (Cluster Compute and Cluster Deploy) and provide development guidelines.

Chapter 4 presents related work in the fields of data platform and benchmark of cloud applications. We discuss the findings in relation to our proposal.

Chapter 5 provides the experimental results for stress research application built using the data platform. We define the experimental setup for benchmark and present the results of client application execution developed in Chapter 3. We compare execution time in dimensions of workloads, frameworks, dataset size and machine instance types.

Chapter 6 concludes with discussion of achieved results and propositions for future work.
Chapter 2

Data Platform Architecture

In this chapter, we provide the details about the architecture and implementation of the developed data platform. The data platform is realized and delivered in the form of repository containing source files embodied in several related subsystems. These subsystems are named as Cluster Series when combined together. Cluster Series gives a foundation for researchers to build and execute their applications. It is composed of two major components which are necessary to enable data platform functionality: Cluster Compute and Cluster Deploy. These components serve different purposes. While Cluster Compute is used to build applications in different frameworks, Cluster Deploy is used to execute the developed application. The focus of this chapter is to answer design questions, stated in Section 1.4. Cluster Series answers the main design question \textbf{DQ1} about automating the application development process. Cluster Compute answers the subsequential \textbf{DQ2} question about supporting multiple computation frameworks. Cluster Deploy answers the \textbf{DQ3} question about execution of the developed application. Both Cluster Compute and Cluster Deploy answer \textbf{DQ4} question about defining the contract which client application has to follow in order to fit into the data platform.

In this chapter, we begin with the introduction of the stakeholders and their requirements for the data platform in Section 2.1. Then, we illustrate the fitness of the data platform into the application development process at research organizations in Section 2.2. In Section 2.3, the interactions of the user with the system are presented. We conclude the chapter with a logical view of the Cluster Series, its core components and major third-party libraries used in Section 2.4. This chapter contains only general introduction to the main outline of the data platform. The detailed development guidelines and APIs are presented in Chapter 3. The related work is discussed in Chapter 4.
2.1 Requirements

The target users of the data platform are researchers at IMEC. However, the proposed solution is a generic one and is not bound to one organization. Any researcher performing data analytics can benefit from the use of data platform. Thus, it is important to realize the functionalities needed for researchers in general. We conducted a face-to-face interview with researchers at IMEC. During the requirements gathering step we uncovered the following desires in regard to a data platform solution:

1. **Application versatility:** Researchers wish to write an application, in both Hadoop and Spark framework and be able to execute either of them on the cluster without redeploying. Existing algorithms need to be reusable and compliant with the proposed data platform solution.

2. **Environment adaptability:** Researchers wish to wire up custom dependencies on per client application basis. Common dependencies must be already available on the cluster in order to reduce the complexity of client applications. Applications can be dependent on publicly available (PIP) and/or private in-house (BitBucket) modules.

3. **Application testability:** Researchers wish to test the application locally in order to ensure its correctness and reduce the costs of public cloud usage (AWS). Environments must be easily switchable from local testing to production deployment on the cluster.

4. **Configuration:** Researchers wish to control the settings of the data platform easily through configuration files. The client application must be configurable via additional parameters. Dependencies of client application must be specified via configuration files.

5. **Deployment automation:** Researchers wish to manage the application lifecycle on the computation cluster in a transparent way. Remote communication (SFTP and SSH) needs to be abstracted and be simple to use.

6. **Computation automation:** Researchers wish to automate the deployment of the client application to the EMR cluster, including installation of the dependencies and management of the execution of the Spark and Hadoop applications.

7. **Tool for configuration:** Researchers need a tool that allows to quickly set up different cluster configuration for their computation and compare the performance, e.g. the performance on use of Hadoop and Spark framework.

---

1 PIP [33] is a package management system for software packages written in Python
2 BitBucket [15] is a web-based hosting service for Mercurial or Git version control systems
3 SSH File Transfer Protocol [32] a network protocol for secure file transfer over secure shell
These requirements are to be realized in the data platform. Next, we need to understand the fitness of the data platform in the current application development process, in the specific case of IMEC.

2.2 Application Development Process

We use stress research as an example for identification of the current application development process. First, we outline the end to end workflow of how stress research was performed:

1. **Ideation:** Researchers create the hypothesis about the relation between body signals and actual moments of stress of participants. The research question is derived from the hypothesis which needs to be experimentally validated. Timeline and resources needed are identified at this stage.

2. **Preparation:** Researchers identify the test subject group. Participants are recruited from this group. Researchers also prepare the sensors and configure them for streaming data into persistent storage. Other infrastructure is set up, such as S3 for data storage and Kinesis for data streaming.

3. **Experiment:** Researchers have initiated the experiment and monitor progress. At this stage, this step helps in the correct operation of the sensors in the field by observing the streaming data and periodically generated automated reports about the data being collected in the persistent storage.

4. **Development:** Researchers develop the computational application for data analysis.

5. **Computation:** Researchers perform batch computations over the data collected from the experiments upon their completion.

6. **Evaluation:** Researchers analyze the results and reason about the correctness of the initial hypothesis. Improvements for the future research experiments are also delivered at this stage.

Data platform focuses on the development and computation steps of the stress research workflow. Thus, the developed application has a contract of taking raw data as an input and producing results of the computations as an output. On the input side, it assumes that the data are already collected and reside on S3. On the output side, it produces results which is stored back in the S3 for further analysis. No other additional input/output sources are involved, unless the client application logic states otherwise.

Next, we will be looking into the details of the computation process to identify the steps that the data platform needs to automate in stress application:
1. **Algorithm development:** Researchers implement a scientific algorithm for computation over the data. Most of the algorithm development focuses on extraction of features from the measured data signal. Features are divided into different workloads based upon the time required for the computation of these features. Some features are computationally complex and require a very long computation time, while others are simple and computed quickly on a given dataset.

2. **Development environment setup:** Researchers set up a development environment with required software, such as Anaconda \(^4\). Moreover, researchers setup AWS credentials for interaction with AWS services.

3. **Application integration:** Researchers create a new Spark application for execution on the cluster. Necessary algorithms are integrated into the Spark application. Due to legacy reasons, certain teams have been using Hadoop for all their workloads.

4. **Application testing:** Researchers launch Spark in local environment for application testing over small subset of the data from S3. Hadoop applications are only testable via StarCluster based on EC2. If the test is successful, the process moves to the next step.

5. **Production application deployment:** Researchers run EC2 copy-dir script for application deployment on the cluster. Deployment requires manual SSH and SFTP to every node in case of StarCluster. The additional dependencies for running the client application are captured in a custom AMI \(^5\).

6. **Production Application execution:** Researchers execute Spark applications via EC2 script. Hadoop applications are executed via Hadoop streaming \(^6\).

7. **Production environment cleanup:** Researchers shutdown the cluster upon successful completion of the job.

The application development process steps are outlined in the Figure 2.1. Both development and computational steps are the main focus of the data platform. The development step is covered by the Cluster Compute project. The computation step is covered by the Cluster Deploy project.

---

4Anaconda \([9]\) is a Python distribution including packages for data science

5An Amazon Machine Image (AMI) \([6]\) provides the information required to launch a virtual server in the cloud

6Hadoop streaming \([21]\) allows to run MapReduce jobs with any executable or script
Figure 2.1: Application development process steps.
From observing and analyzing these steps, we gathered several findings:

- Spark and Hadoop applications require different development and execution procedures.
- The number of small steps involved in deployment and execution is time consuming and error prone.
- Local testing involves charges for EC2 and S3 usages.
- Introducing new dependency into a client application requires the creation of a brand new AMI.

Data platform will cover the current process starting from application development up until the production environmental cleanup. Data platform is materialized in the form of Cluster Series capable of answering DQ1 by introducing two related subsystems, Cluster Compute and Cluster Deploy, which enable automation of the application development process. Both Cluster Compute and Cluster Deploy answer DQ4 by defining the client application structure for integration into the data platform. The researcher will perform application development and testing steps with Cluster Compute. Next, researchers will perform production application deployment, execution and environment cleanup steps with Cluster Deploy.

### 2.3 Functionality

The data platform needs to enable both frameworks (Hadoop and Spark) to be available for the application development. Moreover, the data platform needs to support the execution of the developed application in the computational cluster. Researchers achieve such functionality via Command Line Interface (CLI). The CLI is available in the Cluster Deploy project. The CLI commands reflect the features required by the researchers to for execution. The parameters of the CLI commands are needed in accordance with the configurability property of the data platform. The CLI commands are described further in Chapter 3. Interactions involve EMR, EC2 and S3 as the main components, where the EMR is composed of multiple EC2. Thus, the aimed features include the ability to develop applications in different frameworks and execute the developed application via a simple CLI.

Figure 2.2 shows the intended way of interactions between the researcher and the computational cluster. This interaction highlights the Cluster Deploy functionality. The workflow is generalized and does not depend on the client application. The execution of different client applications is done in the same fashion through the run command. The run command takes additional parameter which points to the location of the deployed client application on the EMR. The deployment is done through the deploy command. For deployment of the client application, additional parameter is used. This parameter is the location
of the deployment ready package on the researcher’s machine. Researcher issues commands through the Cluster Deploy to the EMR. Interaction starts with the \textit{launch} command sent to the EMR. This command will cause the EMR to launch and bootstrap the EC2 machines according to the configuration specified by the researcher. At this point, EC2 machines are started and contain the most common dependencies required by any scientific applications on data analytics. Then, researcher issues the \textit{deploy} command in order to distribute the application across the EC2 machines in EMR cluster and install the additional dependencies, specific for that application. This step prepares the application to be executed. After the deployment is complete, researcher issues the \textit{run} command. Internally, run command translates into execution using the framework specified by the researcher. The application requests the input data from S3, performs the computation and stores the result of the computation back to S3. Lastly, researcher issues \textit{terminate} command to shutdown the computational cluster and release the EC2 instances.

Figure 2.2: Sequence diagram of a researcher’s interactions with the data platform.

The researcher can execute \textit{launch}, \textit{deploy}, \textit{run} and \textit{terminate} commands. This commands are designed for the data platform. The number of commands is more than one in order to make the data platform more flexible. If researcher had only one command, this would involve starting the cluster, deploying, computation and termination all at once. Such design does not allow long lasting clusters capable of computing more than one application. We design the lyfe-
cycle within 4 commands instead of having a monolithic architecture of only one command. The launch command takes no parameters. Instead, it uses the configuration file to realize the number of machines in the cluster, their instance type, bid price, security parameters, region and availability zone to run. The deploy command takes one parameter. This parameter is the location of the deployment ready package of the client application on the local machine of the researcher. The run command takes one argument and a list of parameters. The argument is a program name of the client application. This refers to the directory name of the client application deployed on the EMR. The list of parameters consist of options differentiating between frameworks, environments, persistence storage, additional arguments for the client application, etc. These parameters are outlined in Section 3.2.1.

Figure 2.3: Deployment diagram of the application in the computational cluster.

Figure 2.3 illustrates the deployment structure of the application in the computational cluster. The researcher has CLI installed on his client device. Researcher wants to perform the functionality illustrated on the Figure 2.2. Each functionality is mapped to a corresponding command. Researcher issues commands to the EMR cluster through this CLI. EMR cluster embodies EC2 machines with the driver program and the client application. Both of them
needs to be deployed first and do not exist by default in the EMR. The client application can be a Hadoop or Spark compliant code. Application execution is scheduled from the driver program, located on the master EC2 node. EC2 worker nodes contain the client application. Input data are residing in the S3 and are accessible from EMR. The commands are given as a part of Cluster Deploy. The commands are executed from the researcher’s machine. The execution can be done on any operating system if it supports Python. The Python execution environment is necessary to run the commands from Cluster Deploy. The execution is done in the following way: researcher execute python file via "python emr.py" command and then issue the commands (launch, deploy, run, terminate). More detailed description of the commands is given in Section 3.2.

2.4 Software Design

Cluster Series is a set of related subsystems, together composing the data platform. Main subsystems include Cluster Compute and Cluster Deploy. Cluster Compute provides a foundation for developing client applications. Cluster Deploy provides a tool for managing the execution of the client application. In this section we discuss an overview of the Cluster Series and interactions among its subsystems.

Cluster Series heavily relies on open source projects. Cluster Compute uses Luigi \(^7\) for satisfying the requirements on unification of Spark and Hadoop applications and easy switching between local testing environment and production deployment on the cluster. Cluster Deploy uses Boto3 \(^8\) and Paramiko \(^9\) to satisfy the requirements on communication with AWS and transparent application execution on the cluster.

Overview of Cluster Series internals is depicted in Figure 2.4. The workflow for every new application follows the bottom up approach. First, researcher installs Cluster Compute subsystem via PIP \(^10\) in order to access base classes necessary for client application development (indicated by the arrow "use"). Second, researchers integrate algorithm into client application based on Hadoop and/or Spark (indicated by arrow "develop"). This is done by integrating the client application into Luigi framework discussed further in Section 2.4.1. The unit of Luigi framework is a task. The task is a class consisting of three methods: required, run and output. The required specifies the input data for the task. The output specifies the result data of the task. The run specifies the computational function performed over input data in order to produce the output. The researcher’s main activity is to fill the run method. The application development effort is the same as with manual approach. For Hadoop version, the development is slightly simplified due to Luigi Hadoop streaming wrapper. The porting of existing applications is done via transferring the Spark application.

---

\(^7\)Luigi [27] is a Python module for building complex pipelines of batch jobs
\(^8\)Boto3 [16] is a Python AWS SDK with object-oriented and low-level APIs
\(^9\)Paramiko [31] is a Python implementation of the SSHv2 protocol for both client and server
\(^10\)PIP [33] is a package management system used to install Python software packages
into the run method. Hadoop applications are transferred into two methods: mapper and reducer. Both methods represent the general run method of the Luigi task. The mapper and reducer methods come from the open source contribution to the Luigi framework. This contribution extends the general Luigi task to the MapReduce paradigm by replacing the run method with two methods for map and reduce functions. The actual gain of using the data platform is the automation of the client application execution on the computational cluster supported by the Cluster Deploy. After the application is developed, researchers execute the client application on the cluster via Command Line Interface (CLI) of Cluster Deploy. Cluster Compute and Cluster Deploy does not interact with each other directly. Instead, they use client application (indicated by yellow box) as a man in the middle. Cluster Compute can build a so called deployment ready package out of the client application. Such a package will be accepted by Cluster Deploy for execution.

Figure 2.4: Overview of the Cluster Series and development process steps.

Such loosely coupled architecture enables separation of concerns. For instance, researchers can be split into two groups: developers and production managers. Developers will be granted only a Cluster Compute subsystem to build and test client applications without any possibility of deploying to a production environment. This gives a safe sandbox to play with no costs involved. Production managers will be granted only Cluster Deploy subsystem upon which the client applications are executed. This reduces the knowledge needed for developing a client application and puts more emphasis on management part. Moreover, Cluster Compute and Cluster Deploy become independently replaceable. For instance, current AWS based Cluster Deploy can be substituted with the one based on Microsoft Azure.

Cluster Compute and Cluster Deploy interact with the client application if and only if it follows a few structural restrictions. Next, we examine each
subsystem in details with respect to the structural constraints to be followed by the client application.

2.4.1 Cluster Compute

Cluster Compute is a foundation for any client application in Cluster Series. It supplies run, build and configuration functionality required for deployment, testing and execution. Moreover, it provides abstract classes to be extended for Hadoop and/or Spark client applications. Cluster Compute answers DQ2 by unifying Hadoop and Spark under the same client application. It does so by using the Luigi process workflow framework discussed in the next section.

Interactions between client application and Cluster Compute are loosely coupled in order to maintain backwards compatibility when introducing new features. The researchers have to make the client application compliant with Cluster Compute. There are two types of distinct interactions: client application management (run, build and configure) and development (Hadoop and/or Spark).

Figure 2.5 illustrates the internal structure of Cluster Compute and its contract with a potential client application. Typically, the client is required to contain dummy "run.py" and "build.py" files which will internally call respective functions from "project.py". Their only purpose is to act as an executable for testing the client application and building a deployment ready package. Abstractions for both frameworks are provided in "hadoop.py" and "spark.py". Client application extends abstract classes contained in these files and uses "storage.py" for definition of the input/output locations. Configuration of the client application is done via "project.ini" file. There, researchers can define additional parameters, dependencies, AWS settings and paths to Spark/Hadoop classes to execute.

The deployment ready package is generated from the client application by the "build.py" file. The "build.py" contain one line, which internally calls the build method of "project.py". This method executes the EGG building of the client application. The EGG is a distribution package for Python, similar to JAR file in Java. The script clean up the existing build and executes the standard command for building a new EGG: "python setup.py bdist_egg". The result of the "build.py" execution is the folder with project name under the target folder of root directory. This folder is refereed to as deployment ready package. It is called so, because this folder is ready to be deployed to the EMR cluster via "deploy" command of the Cluster Deploy project. The execution is done via "run.py" file, which internally calls the run method of "project.py". This method first reads the configuration file, then looks at the properties provided when calling the "run.py" in order to determine the framework needed (Spark or Hadoop). Then, run method performs localized import of either SparkTask (from "spark.py") or HadoopTask (from "hadoop.py"). Both classes extend from one Task class from "task.py" and have one method in common, called "run". The "run" method maps the parameters of the run command into the Luigi framework. Moreover, run method accept
a path to executable class, specified in the configuration file. This path points to the Luigi task, where the computation should start, for instance: "wordcount.hadoop.wc_hadoop.WordCountHadoop". This path is later evaluated as Luigi task in the "_run" method of concrete SparkTask and HadoopTask. This allows for Luigi to take the task from given the location and execute it.

The "spark.py" contains the SparkLuigi class. The purpose of it is to be a parent class for the client Luigi task in "client_spark.py". SparkLuigi provides a mapping from run method parameters to Luigi fields, which can be accessed in the client code. The researcher extends the SparkLuigi class by redefining three methods of the Luigi task: required, output and run. For required and output methods, researcher can use the corresponding methods from the "storage.py" file. For instance, "storage.py" has Storage class with two static methods: input and output. Therefore, researcher uses Storage.input in place of required method of Luigi task and Storage.output in place of output method. These methods take parameters, such as target, bucket and input_file for Storage.input and target, bucket, output_dir and flag for Storage.output. Target refers to the target option in the "project.ini" file (explained further in the Chapter 3). The bucket refers to the location on S3. This option is omitted if local target is used. The input_file and output_dir are relative locations to the files. If target is set to s3, then files are searched in s3 bucket. If the target is set to local, then files are searched in the local disk from the root directory where the program was run. The "hadoop.py" contains the HadoopLuigi and PandasHadoopLuigi classes. The HadoopLuigi is the class to be extended if no dependencies are needed for the client application. The PandasHadoopLuigi should be extended when the computation is done with pandas or numpy dependencies of Python Anaconda distribution package. The difference between the two is that PandasHadoopLuigi contains more dependencies already in place, in comparison to the vanilla HadoopLuigi.

The environmental variables are specified in "environmental.py". This file contains three enum classes, one for framework selection (spark or hadoop), another one for storage (local or s3) and one more for execution environment (cluster or local). These enums are used throughout the Cluster Compute code within the if statements when switching between different configuration options. The "parameters.py" is a class for reading the parameters passed to the run method. The parameters could be in the form of "-t" or "-d" etc. (more details in Chapter 3). The "parameters.py" takes these options from the execution command (for instance: "python run.py -t s3 -d local"), reads and converts them into fields of SparkTask or HadoopTask depending on the framework used. These fields are then passed to Luigi task in SparkLuigi and HadoopLuigi classes.

Design illustrated on Figure 2.5 is the solution for DQ2 about supporting both Hadoop and Spark applications with a simple way of switching between the two. It provides a base class for Hadoop application using the Luigi framework (discussed further below in detail), which is a wrapper for Hadoop streaming. Researchers extend this class to create their own client application. There is also a base class for Spark with similar functionality. It also allows for extension by the client application, but does not depend on the Luigi framework. This is done
because Spark applications are easy to build and integrate. Hadoop streaming, however, requires a significant amount of extra work for reading/writing from/to standard input/output. A number of wrappers exist to simplify the development of Hadoop streaming applications. Luigi framework provides both a wrapper and a process workflow, which we find important in iterative algorithms. Through this process workflow it becomes possible to have two unified processes, one for Hadoop and another for Spark.

Figure 2.5: Cluster Compute internals and interactions with the client application. Arrows from one file to another means that one file "uses" another.

Additionally, Cluster Compute design provides a set of utilities for managing the build and run processes. Build process involves the creation of the deployment ready package from the client application. Run process involves the execution of the client application for testing in the local environment and subsequently on the computational cluster. The design allows for easy switching between the environments through the parameter of the run command or the configuration file. All environmental dependencies, such as persistence storage (s3 or local), execution environment (computational cluster or local) and framework (spark or hadoop) are grouped in "environment.py" file in the form of enums. These can be switched either through the configuration file or through the additional parameters of the run command.

First, researchers provide the desired configuration in "project.ini". The "project.ini" is a part of the designed data platform, which is a configuration file in the form of a key value mapping. There are 12 option, such as: program name; path to Spark and Hadoop executable Luigi tasks; target, framework and deployment options (valid selections are provided in "environment.py" file in respective enum classes); paths to input and output; s3 bucket to use for
input/output; list of PIP and BitBucket packages that the client application depend upon and the additional arguments. Researcher can provide his own defined arguments to the client application through this additional arguments. For instance, if researcher want to introduce parameter named "foo" with value "777", he gives an argument in the following form: "-a foo=777". This automatically creates a field in the main executable Luigi task named foo and assigns the value equal to 777. This action is performed by the "task.py" file while parsing the parameters. The detailed description of the configuration is provided in Section 3.2.1. They integrate algorithm into Hadoop (for instance, "client_hadoop.py") or Spark (for instance, "client_spark.py"). Third, researchers test the client application via "run.py". The running of the specific framework is done in isolation. Thus, clients never have to worry about collisions between Hadoop and Spark in the cases where they are included in the same application. Lastly, researchers build a deployment ready package via "build.py". At this point, researchers can use the Cluster Deploy to execute the deployment ready package.

**Luigi Process Workflow**

Cluster Compute uses Luigi for the unification of Hadoop and Spark based client applications. Luigi enables the execution of the applications in both frameworks through a single API. Moreover, Luigi provides a wrapper for Hadoop streaming to simplify the application development.

Luigi framework’s primary purpose is to provide a process workflow. That means Luigi can build a dependency graph of one task dependent on the others and run the entire topology. Luigi automatically finds a task with no dependencies, executes it first and then searches for other tasks which were dependent on the executed ones. This way Luigi can evaluate all the tasks in a pre-defined order. This can potentially be used for iterative Hadoop jobs, since there is a clear dependency between the tasks. However, Luigi at the core is a process framework with no vision of distributed computing.

A unit of work in Luigi is a task. It has three parts: input source, output source and computational function. As soon as the input source is available, Luigi realises that it can execute the computational function. The result of the computation is written to the output. Usually in a pipeline of tasks, the output of one task is an input for the other task. So when the output is ready from one task, it triggers the execution of the subsequent connected task. The task with no input or always available input is taken as a first task to execute. If the output of the task already contains some value (non empty), that means that the task has been executed.

Open source contribution extended original Luigi framework to support Hadoop streaming. It creates a wrapper by redefining the computational part in the Luigi process workflow. The rest of semantics in the Luigi process workflow remains the same for input and output source. The contribution includes two modes for Hadoop streaming wrapper: simulated and production. Simulated mode is used for testing on the local machine. Production mode uses Hadoop
streaming jar and can be executed on the computational cluster.

This combination differentiates Luigi from other alternatives. For instance, Airflow from Airbnb is also a process workflow framework, but it lacks essential Hadoop integration. Thus, we achieve two benefits with Luigi: simplified Hadoop streaming wrapper for Hadoop applications and process workflow structure for iterative algorithms. Moreover, process workflow can mask the particular implementation. Since both Hadoop contribution and general Luigi tasks are alike, it is possible for us to create two versions for Hadoop and Spark. This way, we extend Luigi classes and create base classes for Spark and Hadoop. Spark version remains exactly the same, since computational function embodies the typical Spark application. Hadoop version overrides the methods for setting up common dependencies used by all client applications. This is done to reduce the routine code in the client application and also reduce the effort of researchers in creating new applications with the same settings. The implied generic client application is the stress application described further in Chapter 3. We have chosen this application because it highlights the most common computation done by the researchers. Therefore, other applications will share the dependencies, the way of execution and development.

Figure 2.6: Luigi based integration of Hadoop and Spark in Cluster Compute.

Differences in Hadoop and Spark are handled by Luigi structure, as depicted on Figure 2.6. Luigi provides a set of abstract classes to simplify hadoop streaming process (indicated by yellow boxes). The top level class is the Task, which is common to all the other tasks. Hadoop streaming contribution defines BaseHadoopJobTask and JobTask classes, where the computational method of general Luigi Task is overridden in the form of two methods: mapper and reducer. Cluster Compute provides additional classes (indicated by blue boxes) to fill the
gap for Spark and simplify the integration of Hadoop applications. *HadoopLuigi*
class contains general purpose abstraction for Hadoop based client applications. *PandasHadoopLuigi* class extends *HadoopLuigi* with scientific packages already enabled for computations based on Pandas. The necessary methods to manage the dependencies include "_extra_modules()" and "_jobconfigs()". Extra modules refer to additional python imports, such as "io", "boto3" or researcher specific "imec" packages. Job config refer to the given hadoop streaming parameters, such as "mapreduce.job.maps=3" or "mapreduce.job.reduces=3". Additional "hadoop_streaming_jar" field needs to point to the location of the hadoop streaming jar on the cluster. By default we use the standard location of the hadoop streaming jar on the EMR cluster.

In Luigi world, each class is a task in the pipeline execution. Single task has input, output and computational function. Thus, Luigi reads input, performs the computation and writes to the output. This approach fits very well with iterative Hadoop jobs. Cluster Compute simplifies the Luigi hadoop streaming by providing two functions for supplying additional parameters and dependent modules. Most importantly, Luigi unifies both frameworks and makes them executable in the same fashion.

### 2.4.2 Cluster Deploy

Cluster Deploy is a convenient wrapper around EMR and S3. It provides seamless integration of the Cluster Compute based client application and the EMR cluster. Cluster Deploy takes care of wiring up required dependencies, bootstrapping nodes in the cluster and transparent management of client application lifecycle via Command Line Interface (CLI). Cluster Deploy requires a deployment ready package, which can be obtained by building the client application with Cluster Compute. Cluster Deploy answers **DQ3** by allowing the execution of the developed application on EMR. It provides a simple way of issuing commands to EMR in order to manage the application lifecycle.

Figure 2.7 illustrates the inner structure of Cluster Deploy, its contract with a potential client application and points of interactions for researchers. Dummy "run.py" file is used for the execution of the client application in the same way as describe in the Cluster Compute. The only difference is in the execution environment parameter being set to cluster. Dependencies are read from PIP and BitBucket options of the "project.ini" and auto-wired on per client application basis. These are the only two files required by Cluster Deploy. Researchers configure the cluster properties via "cluster.ini" file. Researchers interact with Cluster Deploy via "emr.py" and "s3.py" files through CLI.

Design, illustrated in Figure 2.7 is the solution for **DQ3** about enabling the client application lifecycle management on the computational cluster. The structure reflects the functionality required to be supported for the researchers shown as a sequence diagram in Figure 2.2. Researchers can configure the computational cluster via configuration file. They can issue commands via Python interpreter as illustrated in Figure 2.2. They are able to operate with the input/output data through "s3.py" and manage the cluster through "emr.py".
The design uses the development environment of the researchers’ machine to read AWS credentials for connecting to EMR. The design enables the researcher to interact with the system via CLI and manage the client application on the computational cluster, including the deployment and execution.

Abstraction for the management of the client application lifecycle is defined in "distribution.py". It provides template methods to be overridden by the concrete implementations. Thus, "distribution.py" has no knowledge on how to launch or terminate the cluster. The "emr.py" has that knowledge and is responsible for launching and terminating the particular cluster. Currently, Cluster Deploy has a concrete implementation for EMR only. Theoretically, it is also possible to add Cloudera or manual Hadoop distribution implementations.

Figure 2.7: Cluster Deploy internals and interactions with the client application. Arrows from one file to another means that one file "uses" another.

In order to ease the integration of the new classes into CLI for researchers, we decide to use the inspection feature of the Python programming language. The aim here is to simplify the development process and make it more generic for the integration of the future classes into the CLI. Using the inspection allows
to avoid a multiple level of if statements used for parsing the input of CLI and mapping it to the actual function call. Multiple if statements are error prone, create unnecessary lengthy broilerin code and are difficult to maintain. Inspection allows to avoid this burden and explicitly map the CLI input to the function call.

The researcher start by configuring the options in "cluster.ini" file. These options include the security groups, region, availability zones, machine instance types, their number and bid price as shown on the snippet below.

```ini
[AWS]
Region = us-east-1
AvailabilityZone = us-east-1a
SecurityGroupId = sg-12ab34cd
SecurityGroupName = example
KeyFilename = C:\Users\user\example.pem

[S3]
Bucket = example-bucket

[EC2-amazon]
Username = ec2-user
AMI = ami-5fb8c835
BidPrice = 0.004
InstanceType = t1.micro
InstanceCount = 1

[EC2-ubuntu]
Username = ubuntu
AMI = ami-b2e3c6d8
BidPrice = 0.03
InstanceType = m1.medium
InstanceCount = 1

[EMR]
Name = Cluster Master
Username = hadoop
ReleaseLabel = emr-4.7.0
Ec2SubnetId = subnet-122345a1
LogUri = s3://example-bucket/logs/
MetricsUri = s3://example-bucket/metrics/
MetricsTimerMin = 15
ServiceRole = example_emr_role
JobFlowRole = example_emr_ec2_role

[EMR-Master]
BidPrice = 1.0
InstanceType = m4.4xlarge
InstanceCount = 1

[EMR-Core-Slave]
```
The "emr.py" and "s3.py" files represent a set of methods that can be called as command lines. Loading a .py file gives a set of commands that are implemented in Python and internally call functions in other files. Certain functions are expected from these files and represent functionalities the researcher wants to perform.

The launch interactions between the researcher and Cluster Deploy is shown in Figure 2.8.

Figure 2.8: Researcher interaction with Cluster Deploy via CLI (launch).

Then researcher starts the interaction with "emr.py" file of the Cluster Deploy. This is the file, thorough which all the commands are issued. The CLI can be started using the Python interpreter as follows: "python emr.py". This opens an interactive CLI of the Cluster Deploy capable of accepting the commands form the sequence diagram in Figure 2.2. First, researcher starts the cluster by issuing the launch command through the "emr.py". This file is using "input.py" for parsing the commands from the researcher and translating them into function calls. Another file used is the "cluster.ini" which contains all the information needed to start the cluster. We store AWS credentials as
environmental variables. They are read by the "credentials.py" and distributed to the "emr.py" in order to launch the cluster. The "emr.py" file contains one class called EMR which is a concrete implementation of the abstract class Distribution from "distribution.py". The reason for the abstraction is the potential extension of the data platform to support solution other than EMR. The possibilities include Cloudera and other Hadoop distributions. The responsibilities are split into general workflow and concrete work with the cluster. The general workflow includes bootstrapping the machines, installing dependencies, deploying and running the client application. These features are embodied in "distribution.py" file. The knowledge on how to launch and terminate the cluster is implemented in the "emr.py" file. There EMR class is using Boto3 AWS SDK for communication with AWS EMR. The "deployable.py" file contains a high level Deployable abstract class, which is a parent class for Distribution class from "distribution.py" and S3 class from "s3.py". The only purpose of the Deployable class is to provide a logger, so that all the commands from researcher and the output to these commands from the EMR will be displayed in the CLI.

The Distribution class in "distribution.py" can establish an SSH connection using Paramiko wrapper from "sshclient.py" given that EMR class has launched the cluster. The file contains a SSHClient class which internally uses Paramiko for establishing a remote connection to the master node of the EMR. Thus, through that class all the researchers' commands are being send to the cluster for execution. The class is instantiated provided a username, hostname and path to the security key file. The class has two main methods: execute and upload. Execute accepts a valid command, which is a general Linux shell command, such as "ls", "pwd" or "ps ax". The method returns two variables: standard output of the executed command and error output. Both variables are being logged in the CLI for the researcher to read. Upload method of SSHClient does recursive uploading of the directory from researchers' machine to the remote master node. Upload takes two parameters, the location of the folder on the local machine and the desired path to upload to on the remote machine. This feature is used when deploying the client application. After establishing the SSH connection to a newly launched cluster, Distribution class uses Boto3 library to query the IP addresses of the nodes in the cluster. These IP addresses will be used by SSHClient when distributing the client application across the nodes. The launch cycle finishes by using the execute method of SSHClient to install the common dependencies, such as boto3, git, luigi, etc. on every node of the cluster.

The deploy interactions between the researcher and Cluster Deploy is shown in Figure 2.8.
The researcher issues the `deploy` command through the CLI of Cluster Deploy. The command typed by the researcher is interpreted by the "emr.py" and corresponds to the function call to the parent Distribution class from "distribution.py". First, the script examines the parameter of the deploy command, which is a path to the local deployment ready package. The script parses the parameter and extracts the program name from it, which is the root folder under the target directory. Then, the script removes the existing program from every node in the cluster via SSHClient execute command and uploads new program via SSHClient upload command. After that, the deploy method of the Distribution class scans through the "project.ini" file of newly deployed application and parses the PIP and BitBucket dependencies. The resolved dependencies are installed on every node via SSHClient execute command. This way the Cluster Deploy installs the additional dependencies required by the specific client application.

The run interactions between the researcher and Cluster Deploy is shown in Figure 2.8.
The researcher issues the `run` command through the CLI to invoke the run method of the Distribution class. The run takes one parameter and a list of arguments. The parameter is the name of the program to execute. It must match the specified program option in the "project.ini" configuration file. The program name is also the name of the directory on the master node of EMR where the client application is located. The list of arguments will be discussed in Section 3.2.1 and represent the selection of using s3 or local persistence storage, hadoop or spark frameworks, input and output locations. The run method of Distribution class parses the list of arguments and determines which framework to run. Based on that, either hadoop streaming jar or spark-submit commands are executed via execute method of SSHClient. All the logs produced by the execution of these commands is captured by Paramiko and displayed to the researcher in the CLI.

The terminate interactions between the researcher and Cluster Deploy is shown in Figure 2.8.

Figure 2.11: Researcher interaction with Cluster Deploy via CLI (terminate).

The researcher issues the `terminate` command through the CLI to terminate the cluster. This invokes the terminate method of the EMR class in "emr.py". This internally invokes the boto3 method of emr object called `terminate_job_flows`. This method takes one parameter, which is the job flow id, and it refers to the id of the EMR cluster we obtained when launching it.

The researcher can also reestablish the connection to an already launched cluster by issuing the `connect` command to the CLI. The command takes three parameters: username, hostname and cluster id (in boto3 refereed as job flow id). This information is obtained at the launch time. If the researcher knows these parameters, he can establish the connection to existing cluster through the connect method of the Distribution class in "distribution.py". It creates the state for new SSHClient object.

**Boto3**

Boto3 is an officially supported SDK by Amazon for communication with AWS in Python programming language. Boto3 solves the problem of deploying the client application to the computational cluster and managing the input/output
data. Boto3 provides both low level APIs based on objects and high level APIs based on the client. In low level version, each resource is represented by the object, such as S3, EC2, EMR etc. For instance, S3 can be obtained using the following command: boto3.resource('s3'). In the high level version, the developer is given less freedom in terms of functionality, because the APIs are targeted to the specific need. The client for S3 is obtained via the high level API as follows: boto3.client('s3').

Boto3 provides the required solution for managing the Amazon Web Services. This SDK has been chosen in order to match the client application development language, which is Python. Cluster Deploy uses Boto3 AWS SDK for interaction with EMR and S3 given valid AWS credentials. S3 wrapper is used as a helper tool to upload, download, remove or list large files with pagination of the results. EMR wrapper is used to manage the cluster and client application lifecycles. AWS credentials are read from environment variables of the researcher’s machine. These credentials will be distributed across the cluster to enable interactions with S3 and EMR.

We define "s3.py" file which uses the S3 wrapper of Boto3. We use "download" and "upload" functions of the Boto3 in order to either download the output files or upload an input file. Both functions use S3Transfer wrapper which automatically switches to multipart uploading/downloading in case the file size exceeds 5 GB. We use it in Cluster Deploy as a helper tool for researchers to upload a single input file or download the output directory. Downloading is done using the prefix. This means that every path on S3 which matches the prefix will be downloaded from S3 to the local machine of the researcher.

We define "emr.py" file which uses the EMR wrapper of Boto3. We use "run_job_flow" function to launch the EMR cluster with pre-defined configurations. We let the entire infrastructure run on the spot instances in order to reduce the price. This function also takes the configurations from "cluster.ini" defined by the researcher. We also use "terminate_job_flows" of the "emr" object in Boto3 to terminate the EMR cluster.

**Paramiko**

Paramiko allows issuing the commands from the researcher’s machine to the computational cluster. The usual way for researcher to do so is to open an SSH to each of the nodes in the cluster. This requires knowing the IP address of each of them. Nodes scale in numbers, therefore a tool for iterating over all the IPs is needed. In case of EMR, it is possible to schedule "steps" for execution through Boto3. For instance, to schedule a Spark job on EMR, the client application first needs to be uploaded to S3, then the "step" needs to kick off the execution. Under the hood, EMR will download and distribute the client application from S3 to every node in the cluster. This involves many small steps and unexpected usage of S3. Conceptually, S3 as an object storage should be used to store the user data, not used as an inter-mediator between the researcher and EMR for passing shell bootstrap scripts or Spark applications.

Thus, researchers have an option between manual SSH to every node the
ad-hoc step notion of the EMR for distribution of the client application and its execution. This issue can be solved by Paramiko framework. Paramiko is capable of establishing the SSH remote connection to another machine programmatically. We create "sshclient.py" file to wrap the essential Paramiko commands for that purpose. Paramiko is a low level tool. There exists a number of higher level abstractions built on top of the Paramiko. However, we decided to keep the original low level Paramiko in order to reduce dependencies and only use a few essential functions. Namely, we use "connect" functionality given the username, hostname (obtained from Boto3 "describe_cluster" function) and key filename, which is a security key necessary for connection. The hostname is the master public IP address.

Moreover, Paramiko allows for two more actions: issuing commands for execution over SSH and uploading files through SFTP. Issuing commands is done via "exec_command" function. We also achieve interactiveness of the CLI by reading back the output of the issued command. Available streams contain regular output as well as error messages. All these are logged for researcher into CLI. Paramiko has SFTP wrapper for upload feature to the remote machine. Pure SFTP object of Paramiko allows only the upload of a single file to the remote machine. We build on top of this feature to allow recursive uploading of the directory. We use it for to upload the client application to every node in the cluster.

Paramiko allows the researcher to interact with the cluster. It allows to integrate the establishment of the SSH connection directly into the Cluster Deploy, reducing the manual steps needed from the researcher. This way the Cluster Deploy becomes self-contained and does not require any further extra steps to be performed outside of the data platform.

One of the important requirement of the data platform is the transparency of issuing commands from the researcher’s machine to the cluster. Researchers must be able to deploy and execute the client application with minimal effort. Such functionality is achieved through the usage of Paramiko. The tool is capable of establishing a remote connection to the cluster programmatically over SSH and SFTP protocols. This removes the need of manually copying the files and assessing terminal windows for SSH connection. Paramiko can also read back the result of submitted command in the form of output or error messages. Cluster Deploy enhances low level SFTP single file copying to support recursive uploading of the folder structure. This transfer of deployment ready package to the cluster. Cluster Deploy uses Paramiko in "sshclient.py".

2.5 Concluding Remarks

We have established the requirements for the data platform. Among the others, researchers need a unified way of executing application of different frameworks and transparent computational cluster management via a simple CLI. The first requirement is handled by the Cluster Compute, while the later is handled by the Cluster Deploy. We have discovered the application development process
and decided to focus on the development and computational steps. These steps include the integration of the developed application, testing, production deployment and execution on the computational cluster. In order to fulfill these steps, we outlined the functionality required in the data platform and presented a deployment of the application components on the computational cluster. Internal components of Cluster Compute and Cluster Deploy with related third-party frameworks show the technical capabilities to perform the application development process. With this architecture, we develop an application to find the moments of the stress on top of the data platform in order to validate the compliance with the given requirements.
Chapter 3

Stress Application Implementation

In this chapter, we describe the implementation and execution of the application on moments of stress detection using the data platform. The application involves batch processing over the collected ECG and accelerometer recordings of the participants to extract the relevant features. Depending upon the computational complexity of the features and the corresponding signals from which these features are derived, different workloads are defined. The purpose of stress application is to validate the data platform for support of data analytics computations (Chapter 2). Moreover, we use stress application to also benchmark for different configurations of the data platform (Chapter 5).

In this chapter, we begin with the design outline of the stress application in Section 3.1. In Section 3.2, we provide detailed guidelines of the client application development process with respect to the data platform.

3.1 Design

In a typical MapReduce scenario, computations are performed in reduce function over the data received from map function. However, recordings from the stress research are contained in HDF5 files residing on S3 with one file per participant. This data storage scheme has been chosen based upon the constraints which are outside the scope of our work. Each file has a set of ECG and accelerometer recordings split into days. Actual recordings exist in the form of data frames, queryable by Pandas. The computation procedure will take data of one hour of each day from these data frames and produce one value as a result. The challenge is in getting all these files on the machine in the cluster to perform computations.

A computation cluster consists of master and slave nodes. The client application is submitted to the master node. Then, the computations are scheduled to be executed on the slave nodes. One option is to let the master node down-
load all the files, read the data frames, slice them per hour and distribute these hourly data to slave nodes for computation. Slaves will perform the computation and return the result back to the master node. Final results are stored to S3. This approach forces the slaves to wait until the master downloads and slices the data frames. This creates a network bottleneck for master node and increases idle time for the slaves.

Another option is to create an input file with the paths to the HDF5 file on S3. Master downloads the input file and distributes these paths across the slaves. This increases the downloading and scheduling speeds of the master due to the small size of the input file. Slaves receive a path to HDF5 file, download it, slice data frames by the hour, compute workload over the slice and store the final result in S3 independently. This way network load becomes distributed across all the slaves in the cluster. This design is not a part of the data platform. This is the client application meant to be integrated into the data platform to provide a use case for it. The data platform consist of the Cluster Compute and Cluster Deploy. The stress application is integrated into the data platform in the form of a client application. It is built based on Cluster Compute and executed with Cluster Deploy. The client application is a program which can be deployed to the master node and slave nodes. It can also be executed from the master node. The client application is deployed to the master node via Cluster Deploy. The versions deployed to master and slave nodes differentiate in the way of the execution. The package stays the same for master and slaves, however the execution is performed from the master. All the required dependencies are installed on both master and slaves.

We decided to go with the second option of the master node distributing only the input file with the path to the data due to its simplicity. The process is shown in Figure 3.1. The original data were collected from the participants of the stress research with the wearable sensors. The data were streamed through Kinesis to the S3. We have generated an additional file, which contains the paths to all the raw data files. Input file with file paths is read from S3 into the client application (step 1). Then, master node distributes the paths to HDF5 files to the slave nodes (step 2). Each slave individually downloads and processes HDF5 files from S3 (step 3). The results of the computations are collected back in the client application (step 4) followed by the uploading of the resulting data to S3 (step 5). This data is the result of the computation of the selected workloads over the raw data collected from the participants. Client application execution metrics are periodically pushed to S3 (step 6). These metrics will be used to benchmark and for subsequent analysis, if required.

For benchmark purposes, we run algorithms over a reduced subset of participants of the stress research. We also generate smaller files out of the original data in order to introduce variability in dataset dimension. Thus, we split one HDF5 file of a single participant into several files. Each of them contains only one day of recordings. Such splitting of data into smaller files helps us to understand the impact of dataset size on computational time and other execution metrics. The data platform is capable of taking input data from different location, therefore it is easy to switch from the small subset for testing to real
production with all the data.

Figure 3.1: Stress application design and client development process steps.

3.2 Integration

In this section, we give detailed guidelines on how existing algorithms and workloads can be integrated into the data platform, with the specific context of the discussed stress application. We present the integration of existing workloads in the stress application using the Cluster Compute in Section 3.2.1. We discuss the execution of the deployment ready package on the cluster using the Cluster Deploy in Section 3.2.2.

3.2.1 Development

The necessary client application structure includes two dummy "run.py" and "build.py" files for testing and building the deployment ready package. Stress application properties are defined in "project.ini" residing at the same directory level as "run.py". The configuration file is necessary to provide information such as which framework to run, where are the input/output data located, where can the executable Luigi task be found in the client application and what dependencies does the application have. These properties can be overridden at the launch time using the following options:
1. **Program** (-p, --program) [string: directory name]: name of the application. Used as a root directory name for deployment ready package. Must be unique and should not contain spaces or special characters.

2. **Spark** (-s, --spark) [string: classpath]: location of the Spark executable class in the form of 
   "{package}.{sub_package}.{classname}".

3. **Hadoop** (-h, --hadoop) [string: classpath]: location of the Hadoop executable class in the form of 
   "{package}.{sub_package}.{classname}".

4. **Target** (-t, --target) [Enum: local, S3]: location of input/output is either local or S3. If "s3" is chosen, then "Bucket" option has to be set.

5. **Framework** (-f, --framework) [Enum: spark, hadoop]: the chosen framework must come with either "Spark" or "Hadoop" properties set.

6. **Deployment** (-d, --deployment) [Enum: local, cluster]: chosen environment is either local for testing or cluster for production.

7. **Input** (-i, --input) [string: filepath]: relative location of the input file. Could be local or S3 depending on the "Target" property.

8. **Output** (-o, --output) [string: dirpath]: relative location of the output directory. Could be local or S3 depending on the "Target" property.

9. **Bucket** (-b, --bucket) [string: bucket name]: bucket name on S3 for reading/writing input/output. Only used if "Target" option is set to "s3".

10. **Args** (-a, --args) [string: key/value pair]: pipe separated key/value map of arguments in the form of: a=1|b="two"|c=3.

11. **Pip** [string: pip packages]: comma separated list of public PIPs to be installed on the cluster in the form of: boto3,luigi.

12. **BitBucket** [string: ssh git repo]: comma separated list of BitBucket projects to be installed on the cluster in the form of: git@bitbucket.org:vekimov/cluster-compute.git,git@bitbucket.org:vekimov/cluster-deploy.git.

An example of the "project.ini" file look as follows:

```ini
[Project]
Program = pandas
Spark = cluster_pandas.spark.pandas_spark.PandasSpark
Hadoop = cluster_pandas.hadoop.pandas_hadoop.PandasHadoop
Target = local
Framework = spark
Deployment = local
Input = pandas/data.txt
Output = pandas/output
Bucket = victor-ekimov-bucket
Args = feature=accelerometer_all_features
Pip =
BitBucket = git@bitbucket.org:blamichhane/pyimec-personal.git
```
The stress application fits into the data platform as depicted on the Figure 3.2. The client application represents two independent Luigi tasks and contains both Hadoop and Spark version. Spark version has run method for performing the computation. Hadoop version has two methods: mapper and reducer. Both of them are coming from Luigi contribution to the hadoop streaming integration. The workflow goes from left to right. First, the raw input data of ECG and Accelerometer recordings from S3 is passed to the required method of the Luigi task in the client application (1). Then, the input data are passed through the computational function (run for Spark or mapper/reducer for Hadoop). The computation involves the ECG workloads (such as mean heart rate computation) and accelerometer workloads. The result of that computation goes into the output method of the Luigi task. This method writes the result of the computation over the raw data in S3 (2).

Figure 3.2: Stress application integration into the data platform.

The Hadoop version implies the creation of a Luigi task by extending from `PandasHadoopLuigi` class from Cluster Compute. Hadoop version is contained in "pandas_hadoop.py" and follows these steps:

1. Extend `PandasHadoopLuigi` class of Cluster Compute. The base class allows for the integration of a custom algorithms. The base class already contains the dependencies needed by the data platform. This simplifies the client application development by outsourcing the common code into the parent class.

2. Point "Hadoop" option from "project.ini" to the derived class. This is required for the execution of the client application. The Cluster Compute uses "project.py" to find an executable class. If the "framework"
parameter is set up to "hadoop", then the Cluster Compute will look at "Hadoop" option from "project.ini" and use that string to locate the executable class. Because Python allows to import and execute the class from the string, we resolve the path to the class and execute the "run" method. Regardless of the Spark or Hadoop versions, both have the "run" method. Therefore we know that both versions are executed in the same fashion through the same method.

3. Add dependency modules to `extra_modules()`. These are extra modules related to the client application. For example, it could be the internal libraries used in the computational function. It could also be other modules in the same client application, but from a different package. Researcher add the dependency by importing the package with "import" keyword inside of the "extra_modules()". Then, the function must return an array of such imports. For instance, if researcher wants to import Python os module along with internal imec package from the custom project, then researcher imports both: "import os, import imec" which is followed by "return [os, imec]".

4. Add MapReduce parameters to `jobconfs()`. This function returns an array of the Hadoop streaming parameters, similar to those when executing the regular application with Hadoop streaming jar. For instance, researcher might want to specify the number of mappers and reducers to be 3. This will result into the following code: "return [‘mapreduce.job.maps=3’, ‘mapreduce.job.reduces=3’]".

5. Read from the input file in `required()`. Since we extend the Luigi based class, we receive three functions: input, output and computational, as discussed in Section 2.4.1. Here, the requires function refers to the input data for further processing. This is done via helper function from "storage.py" file. For instance, reading the input can be done by constructing a new object as follows: "Storage.input(target=self.target, bucket=self.bucket, input_file=self.input_file)". All the parameters are taken either from the configuration file or from the parameters provided to the run method. This code assigns the target, which could refer to either s3 or local and interprets this as the place from where the input file is to be read. The bucket parameter is used only when the target is set to s3. Lastly, input_file refers to the relative path to the input file. It is relative path, because it depends on the target option.

6. Fill in `mapper() / reducer()` using yield to emit values. The input for these functions is read from the input function mentioned above. This follows the standard flow of the MapReduce jobs. Map function accepts a key and emits a set of key value pairs. Reduce function accepts key and the associated values with that key. Reduce function emits a key and some aggregated value. These methods are used by the researchers to perform computation in the MapReduce fashion.
7. Write to output directory in `output()`. This refers to the output part of the Luigi task. This is the place where the computational part (map and reduce functions) write the result. The usage example is: "Storage.output(target=self.target, bucket=self.bucket, output_dir=self.output_dir, flag=True)". Here, target and bucket serve the same purpose as for the input target and bucket discussed in the previous point. However, output_dir refers to the location of the output folder, where reduce function will write the result. The output is written to the directory, therefore the output_dir must refer to a directory. The flag is used as an indicator of the outcome of the computational function. That means, if the Luigi task has competed successfully, then an empty "_SUCCESS" file will be created in the output directory. If the Luigi task finished unsuccessfully, then an empty "_FAILED" file will be placed in the output directory. If the flag option in the output_dir method is set to false, then no files are created.

The example of the Hadoop version is as follows:

```python
#!/usr/bin/env python
import os
import luigi
from compute.hadoop import PandasHadoopLuigi
from compute.storage import Storage

from cluster_pandas.shared.shared import Shared

class PandasHadoop(PandasHadoopLuigi):
    feature = luigi.Parameter()

    def _extra_modules(self):
        import os
        import warnings
        import itertools
        import imec
        import cluster_pandas.shared.shared
        return [os, warnings, itertools, imec, cluster_pandas.shared.shared]

    def _jobconfs(self):
        return ['mapreduce.job.maps=3',
                'mapreduce.job.reduces=3']

    def requires(self):
        return Storage.input(target=self.target, bucket=self.bucket, input_file=self.input_file)
```
def output(self):
    return Storage.output(target=self.target, bucket=self.bucket,
                         output_dir=self.output_dir, flag=True)

def mapper(self, key):
    yield key.strip(), os.path.join('pandas', 'temp', key.strip())

def reducer(self, key, filename):
    yield key, Shared.process(deployment=self.deployment,
                               storage_dir=self.storage_dir, bucket=self.bucket,
                               key=key, filename=''.join(filename),
                               feature=self.feature)

Spark version is contained in "pandas_spark.py" and follows these steps:

1. Extend SparkLuigi class of Cluster Compute. This parent class does not contain any extra pre-defined properties and is simply used for achieving a unification between the Hadoop and Spark versions of the client application.

2. Point "Spark" option from "project.ini" to the derived class. This functionality is similar to the Hadoop version we discussed above. The only difference being that the application version is Spark and not Hadoop.

3. Read from the input file in required(). The input path can be obtained using the "this.input().path" code inside of the run method. This input file will be read in the run method and data inside of the input file can be used for computation.

4. Fill in run(). This is similar to the Luigi task computational step we discussed for Hadoop version. However here it is not split into map and reduce functions. This means that existing Spark application can be used as is into the run method.

5. Write to the output directory in output(). The output path can be obtained using the "this.output().path" code inside of the run method. The run method writes the result of the computation into the output directory.

The example of the Spark version is as follows:

```python
#!/usr/bin/env python
import os
import luigi
from compute.spark import SparkLuigi
from compute.storage import Storage
from pyspark import SparkContext, SparkConf
from cluster_pandas.shared.shared import Shared
```
class PandasSpark(SparkLuigi):
    
    feature = luigi.Parameter()

    def requires(self):
        return Storage.input(target=self.target, bucket=self.bucket, input_file=self.input_file)

    def output(self):
        return Storage.output(target=self.target, bucket=self.bucket, output_dir=self.output_dir, flag=True)

    def run(self):
        executors = 3
        conf = SparkConf().
            setAppName('{!s} (bucket={!s}, input={!s}, output={!s})'.
                format(self.program, self.bucket, self.input_file, self.output_dir)).
            set('spark.executor.instances', str(executors))
            sc = SparkContext(conf=conf)

            sc.parallelize(map(lambda key: (key.strip(), os.path.join('pandas', 'temp', key.strip()))),
                sc.textFile(self.input().path).collect()).
            flatMap(lambda (key, filename):
                Shared.process(deployment=self.deployment,
                    storage_dir=self.storage_dir,
                    bucket=self.bucket, key=key, filename=filename,
                    feature=self.feature)).
            coalesce(executors).
            saveAsTextFile(self.output().path)

        sc.stop()

First, researcher needs to get the Cluster Compute project on the local machine. This is needed for the development of the client application. If this step is skipped, the client application will not be able to resolve the base classes for Hadoop and Spark versions, which are necessary for the development. Make Cluster Compute available in the development environment:

$ python ..\cluster-compute\setup.py develop

Before launching the EMR cluster and performing real computations, researchers might want to test the client application locally. This is done to prevent the most common error from happening in the production environment on the cluster. Some of the examples of such errors could be copy-pasted values,
misplaced assignments or even compiling errors. The testing is possible through the "-d" option, which refers to the deployment environment. It could be either cluster or local. For testing we use local. The cluster variant will be used to run applications on the cluster. To test the Spark version locally with custom input file and output directory:

```
$ python src\run.py -t local -f spark -d local -i README.md -o output
```

Similar to the Spark version, researchers can test the Hadoop. This is done via "-f" option which refers to framework. Internally, both base classes for Spark and Hadoop extend from one parent. This makes it possible to run them through the "run" method. Testing is possible with both local files or the files from s3. This is achieved because the environment is decoupled from the location of the input/output files. To test the Hadoop version locally with custom input file and S3 bucket:

```
$ python src\run.py -t S3 -f hadoop -d local -i README.md -b luigi-bucket
```

After the local testing, the deployment ready package needs to be built. We need the package because we want to transfer it to the cluster and distribute across the nodes. The package is a directory containing the client application. The build process is orchestrated from "project.py" file where we built two versions, for Spark and Hadoop. Hadoop version requires a folder with python files, while Spark version needs an EGG package. An EGG is an archive of the python project files. They are generated in the process of building an EGG. Thus by simply building an EGG we can get files for both of the versions. To build the deployment ready package:

```
$ python src\build.py
```

The build script will produce a deployment ready package under the target directory resembling the structure, as shown on Figure 3.3.
With deployment ready package generated, stress application is ready to be executed on the cluster using Cluster Deploy.

### 3.2.2 Execution

Researchers can interact with S3 and EMR via "S3.py" and "emr.py" of Cluster Deploy respectively. Both scripts provide Command Line Interface (CLI) for the researchers to interact with.

#### Preparation and Setup

Researchers need to configure AWS credentials for interactions with EMR and S3. Moreover, they need to have an access to the private BitBucket repository with Cluster Compute. Cluster Compute will be installed on the cluster during the bootstrapping stage. Security policies are summarized as follows:

1. Export AWS credentials as environmental variables ("aws_access_key_id" and "aws_secret_access_key").

2. Add SSH public key to the access list of researchers’ BitBucket account: [https://bitbucket.org/account/user/{username}/ssh-keys](https://bitbucket.org/account/user/{username}/ssh-keys). Public key can be located as "id_rsa.pub" in "cluster/bootstrap.sh" of Cluster Deploy.

   Optionally, the researcher may choose to regenerate the ssh key pair. In this case, both private and public keys in "bootstrap.sh" need to be replaced with the output from the following command:
First, researcher needs to have an input file. Usually, when running the application on the cluster, the input files are located on S3. Having them as local would require distribution of the file across all the nodes. If the file resides on s3, it can be downloaded from one location by the client application. Cluster Deploy comes with the tool for uploading/downloading files from/to s3 via Boto3 Python AWS SDK. If the file is not yet on S3, researcher can use this tool to upload one. This tool is available from "s3.py" file. We start S3 CLI by launching "S3.py" file:

```bash
$ python src\S3.py
```

Now researcher is ready to execute commands, available from "s3.py" file. Currently, there are only two options: download and upload. Uploading takes two parameters. First parameter is the path to the local file on the researchers machine. Second parameter is the path to the local file on s3 persistence storage. Essentially the upload command copies the file from the location of the first parameter on the local machine to the location of the second parameter on s3. We upload files to S3 through "upload" command with [LocalFile, S3Path] parameters:

```bash
$ upload ..\cluster-pandas\pandas\data.txt input.txt
```

Eventually, the computation will put the result into the output directory on s3. Researchers might want to download this directory in order to examine the results. Downloading is also done through "s3.py" file, which uses Boto3 multi-part downloading. Multi-part is chosen because the output can be very large. Thus download will become more efficient. Download method takes one parameter, which is the prefix on S3. This is because S3 is fundamentally a flat storage lacking the notion of directories. Instead, S3 has a logical notion of folders to simplify the organization of the files. Thus, by providing the prefix, download method matches all the files which start with it and downloads them. Upon the successful completion of the application, we can observe the results by downloading the output directory through "download" command with [S3Path] parameters:

```bash
$ download output
```

With the preparation of the security policies and the transfer of the input file on S3, we can proceed with stress application execution.
Application Lifecycle Management

First, researcher use the CLI provided by the Cluster Deploy project in order to manage EMR. Interactions with EMR are facilitated by Boto3 and Paramiko Python frameworks. Boto3 manages the AWS integration, while Paramiko enables remote SSH to the master node in the computational cluster. We start EMR CLI by launching "emr.py" file:

```
$ python src\emr.py
```

In order to execute the client application on the computational cluster, we first need to launch the cluster. This is done by the Boto3 framework with the properties specified by the researcher in "cluster.ini" configuration file. The cluster runs on spot instances. Researcher can configure the number, instance type and bid price of these instances in configuration file. The launch command is blocking. It takes about 10 to 15 minutes to launch and bootstrap the cluster. The control of the CLI will be returned back to the researcher once the cluster is up. We launch the EMR cluster through "launch" command:

```
$ launch
```

CLI uses Paramiko for remote SSH and execution of the commands remotely. Because Paramiko acts as a mean of transferring command, we can send any arbitrary command to the master node. This is done through the "do" command of "emr.py" file in Cluster Deploy project. For instance, to execute the "pwd" command on the master node of the cluster, researcher needs to type "do pwd" in the CLI. Thus, all the commands are pre-pended by the "do" keyword. This helps distinguish general functions of the CLI from bash commands to be run remotely on the master node. Optionally, we can issue arbitrary commands though "do" command:

```
$ do ps ax
```

If researcher closes the terminal window or did not launch the cluster himself, he does not have the internal state for the CLI to work. Connect method of Distribution class in "distribution.py" requires to know three things: username, hostname and cluster id. If researcher launches the cluster, these properties are automatically associated with the internal state of the CLI. The state can be reconstructed by providing the same three parameters of the existing cluster which is currently running. In case of lost connection or the closing of terminal window, we can reestablish the SSH connection to a launched cluster through "connect" command with [ClusterId, Username, PublicDNS] parameters:

```
$ connect j-1234567890123 hadoop ec2-12-34-567-89.compute-0.amazonaws.com
```
Now that the cluster is running and the researcher has a connection established to it, he can work on the client application. This can be done in two ways: submit command or deploy and run commands. Submit command is simply a facade for the deploy and run. Internally, it first deploys the client application to the cluster, then runs it. This is done for simplification, so that the researchers can execute the client application in one go. Submit takes two parameters. First parameter is the path to the deployment ready package on the local machine of the researcher. It will be copied to the cluster. Second parameter is the list of arguments. This list is similar to the run command with parameters. Next, we need to submit a deployment ready package to the cluster through "submit" command with [LocalBuildDir, Args] parameters:

```
$ submit ..\cluster-pandas\target\pandas -f spark -i input.txt -o output
```

Submit results in both the deployment and execution of the client application. Optionally, researchers can split submit functionality into deployment and execution.

For instance, researchers might first upload the deployment ready package to the cluster. This will be done by the SFTP wrapper of the Paramiko framework with our modifications. This modifications extend single file uploading via SFTP to recursive uploading of the folder. As a result, the package will be distributed across all the nodes, enabling the execution feature. Deploy takes only one parameter, which is the first parameter of the submit command. We deploy the package through "deploy" command with [LocalBuildDir] parameter:

```
$ deploy ..\cluster-pandas\target\pandas
```

After the package has been distributed with deploy command, researchers can run it. Run is done in the same fashion as for local testing, except that the "-d" deployment parameter is set to "cluster" and not "local". Run takes only one parameter, which is the second parameter of the submit command. We run the stress application through "run" command with [ProjectName, Args] parameters:

```
$ run pandas -f hadoop -i pandas/input.txt -o pandas/output
```

After the client application has been completed, there might be no need of the cluster if not further applications are to be deployed. Researchers might want to shutdown the cluster to avoid paying for idle computation time. The cluster is terminated through "terminate" command:

```
$ terminate
```
3.3 Concluding Remarks

We have presented the integration of the stress application into the data platform. We reasoned about the design of the application and integrated existing computational algorithms into the data platform. We gave the explanation for each of the parameters of the run command in order to execute the application. We provided the steps necessary for creating both Hadoop and Spark version. Then, we showed the process of building the deployment ready package and its deployment to the computational cluster. We have also given the examples of the commands for each of the functionalities required from Section 2.3. The algorithm is written in MapReduce paradigm for Hadoop and as a Spark application for Spark. With the developed data platform and the associated workflow, it takes the same amount of effort from the development point of view, but it considerably simplifies the execution of the application on the cluster. Re-configuration does require re-computation because configuration always precedes the computation. Next, we discuss related work in data platform development and compare them to our solution.
Chapter 4

Related Work

There are some existing projects and works which relate to application development and execution in a distributed cluster computing framework. Apache Beam (Section 4.1) provides support for multiple frameworks to be integrated into one application by using the Beam programming model, which acts as a wrapper. StarCluster (Section 4.2) provides lifecycle management of the application in the cluster based on EC2. MRJob (Section 4.3) provides lifecycle management of Python application on EMR or Hadoop cluster. In the next sections, we discuss these related projects in further details. As we will see, none of the existing frameworks provide an end-to-end solution for both application lifecycle managements on the cluster and application development support on different frameworks.

4.1 Apache Beam

Apache Beam\(^2\) is focused solely on building an abstraction for different distributed computation frameworks. It does not manage cluster lifecycle nor does it have any types of interactions with AWS\(^3\). Instead, the project provides so-called Beam programming model to handle the framework abstraction. This model allows developers to choose a Beam Software Development Kit (SDK) or Domain Specific Language (DSL) to develop their application. This SDK or DSL provides an abstraction on top of the supported frameworks (Hadoop, Spark and Flink\(^4\)). Developers create applications in the Beam model using Beam pipelines which abstracts operations from concrete frameworks. Then, the Beam model internally translates Beam pipeline into specific function calls for a chosen framework.

\(^1\)Amazon Elastic Cloud Compute [2] is a web service that provides resizable compute capacity in the cloud

\(^2\)Apache Beam [10] is a unified programming model used to create a data processing pipeline

\(^3\)Amazon Web Services [41] is a suite of cloud computing services

\(^4\)Apache Flink [12] is a streaming dataflow engine for distributed computations over data streams
Beam name itself comes from "Batching" and "strEAMing". The name hints that the framework is capable of being an abstraction for both batching and streaming applications. Beam pipelines are used for the unification of batching and streaming applications. Beam pipelines are executed and translated into a specific runtime environment by internal runners. Such runners are claimed to be easily extensible for future frameworks. As of May 2016 [11], Beam supports runtimes for Spark, Flink and Google Cloud Dataflow ⁵. Hadoop is not currently supported in Beam. In terms of programming languages, Java ⁶ is already supported while the support for Python and Scala ⁷ are still under development.

In contrast with Beam, our proposal supports only batching applications for Hadoop and Spark frameworks in Python. Language selection comes from Python’s extensive usage for data analysis activities. In the data platform, we develop the foundation for batching applications with support for Spark and Hadoop. Flink has been left for the future work since it is fundamentally a streaming framework. Beam provides an abstraction level for frameworks through Beam pipeline. In contrast with Beam, our platform gives developers the ability to write applications directly in the chosen framework. The motivation is to reduce the time to learn yet another programming model. Moreover, it enables easy integration of existing applications. Finally, our proposal manages the cluster lifecycle along with the development. The Beam does not intend to provide cluster management capabilities.

4.2 StarCluster

StarCluster ⁸ is the project from MIT designed to facilitate the cluster lifecycle management in the AWS cloud. StarCluster does not support application development process. Instead, it acts as a simplified and free of charge version of the EMR. The framework is extensible via plugins written in Python and can be customized via configuration files. Additionally, StarCluster gives password-less SSH ⁹ connection to every node and basic Command Line Interface (CLI) for issuing commands. StarCluster comes with Elastic Load Balancer, which is similar to resizing feature of EMR. The difference is that StarCluster can auto resize cluster based on the job queue while EMR needs to explicitly know the exact number of machines to add or remove.

Currently, StarCluster supports EC2 based clusters with either internal storage or EBS ¹⁰. StarCluster can be composed of spot instances in order to reduce prices for the acquired computation power [18]. Both StarCluster and our data

---

⁵Google Cloud Dataflow [19] is a unified programming model for developing and executing a wide range of data processing patterns ⁶Java [25] is a platform for development and deployment of portable applications ⁷Scala [35] is a statically typed programming language with full functional support ⁸StarCluster [37] is an open source cluster-computing toolkit for Amazon EC2 [2] ⁹Secure Shell [36] is a cryptographic network protocol for operating network services securely over an unsecured network ¹⁰Amazon Elastic Block Store (EBS) [1] provides persistent block level storage volumes for use with EC2 instances
platform come with management tools for S3. StarCluster launches cluster with Ubuntu distributions of Linux and has a set of AMIs \(^{11}\) configured for scientific computations. This is not easily achieved in pure EMR scenario due to the limitations of choice in installed applications. StarCluster is a bit more flexible with customizing instance types which can be done per node basis as opposed to per instance group on EMR. Finally, StarCluster has an interesting feature of supporting the restarting of the entire cluster. The same functionality can be achieved in EMR through cloning of the cluster.

In contrast with StarCluster, our proposal is backed up by EMR. This imposes additional fees for the managed cluster. In return, it provides a reliable solution maintained by the vendor and simple API for managing the cluster lifecycle. We provide a customized setup with dependencies installed for performing scientific computations. This solution avoids the limitations of AMIs range selection in EMR. Our data platform bundles cluster lifecycle management with automation of the application development process. StarCluster does not intend to provide application development support.

### 4.3 MRJob

MRJob \(^{12}\) is a Yelp! project attempting to combine the functionality of Apache Beam and StarCluster. It provides a platform to run Hadoop jobs locally, on a manual Hadoop distribution or EMR. Recently, MRJob has been extended to support Google Dataproc \(^{20}\) execution environment. However, MRJob lacks support for Spark or any other Hadoop based projects. The programming language supported for application development is Python.

Programming model consists of a single Python class executed from the main method. A single class approach is the bottleneck of MRJob. There is a way to embed such class into custom programs via custom runners. However, it does not appear as an elegant solution with respect to software engineering practices. An MRJob based application can only be launched and executed by MRJob. While doing this, MRJob also launches the cluster for the application. This imposes a high level of dependency in the development process and constraints in the execution process.

MRJob applications can be configured for execution in local environments with input/output files from local storage, on manual Hadoop distribution (for instance from Cloudera \(^{17}\), Hortonworks \(^{24}\) or MapR \(^{28}\)) with files from HDFS \(^{13}\), or on EMR with files streamed from S3. It is not possible to mix these properties. For example, it is not possible to use EMR with HDFS. This brings additional constraints.

In contrast with MRJob, our proposal allows the usage of both Spark and Hadoop. The developed data platform also relaxes the programming model constraint of having only one class policy. Storage settings are decoupled from the

---

\(^{11}\)Amazon Machine Images (AMI) \(^{6}\) provides the information to launch a virtual server

\(^{12}\)MRJob \(^{29}\) allows to create MapReduce jobs and run them on several platforms

\(^{13}\)Hadoop Distributed File System \(^{39}\) is used for storing large volumes of data
runtime environment enabling the mixture of local, HDFS and S3 input/output files with local and EMR clusters. During our initial tests, MRJob proved to be inflexible for application development and too constrained in the cluster lifecycle management.

4.4 Concluding Remarks

We have discussed three projects related to the data platform development. We conclude that StarCluster handles only the computational step of the application development process from Section 2.2. Apache Beam handles only the development step. MRJob is the strongest competitor to our data platform. However, it enforces programming constraints, limiting the developers’ ability to flexibly integrate the application. Cluster Series provides more capabilities to the researchers and allows for flexible configurability of the computational cluster. Moreover, Cluster Series enables the development of both Hadoop and Spark versions, while MRJob is only a Hadoop based platform. Next, we proceed with running the experiments on the computational cluster with stress application as an example.
Chapter 5

Evaluation

In this chapter, we present the evaluation of the developed data platform for the stress application set in Chapter 1. The data platform is realized by combination of different tools in order to optimize the time it takes to develop and execute application on the computational cluster and reduce the amount of errors associated with the process. It provides a scalable computation and storage solution besides the end to end solution for application development and deployment. The evaluation and validation of the data platform are done with an example application of detecting the moments of stress based upon measured physiological signals from the wearable sensors. In the context of this application, first the evaluation of the workflow according to the developed platform is presented in Section 5.1. This is presented in comparison with the existing workflows. The evaluation of the time taken for all the steps in executing an application is made and presented. The configurability of the data platform is validated with the benchmark of performance of different configurations for various workloads of the target application. The evaluation method and results are presented in Section 5.2. The evaluation results are also used to give guidelines for the optimal configuration of the data platform in context of the workloads of the application on moments of stress detection based upon the physiological signals.

5.1 Automated Application Development

We compare the automated way of setting up the cluster and executing client application using our proposed data platform solution to the manual setup and execution. First, we compare the time it takes to setup the cluster for computation using manual way versus the automated way using our data platform. Second, we quantify the errors involved in the setup process. The data platform provides a combination of different tools in order to avoid the need to switch between different interfaces and thus reduce the possibility of error in the process. Using our data platform for the application development, researcher always interacts with only one tool and follows the same process workflow instead of
switching between various tools and interfaces.

The development effort of the application on top of the proposed data platform remains roughly the same as with the manual approach. Some additional complexity in development comes from the need to learn Luigi framework and write the algorithms in the form of Luigi tasks. This adds minor complexity in comparison to the MRJob solution. However an additional ease of use in the data platform for Hadoop based applications comes from the Hadoop streaming contribution within the Luigi framework. The enables easy representation and porting of both the Spark and Hadoop based applications into the computational part of the Luigi task. Thus, the development step of the application development process is slightly simplified for the researchers at a price of the need to learn Luigi process workflow.

The manual approach to application development and deployment involves the following actions:

1. **Prepare the environment**: Open the browser and login with AWS credentials.

2. **Setup the configuration**: Go to EMR page in AWS console and select "Create cluster" button. Manually browse and select for various configuration options like: spot instances or on demand instances, choices of hadoop distribution (Amazon, MapR), available software to install on every node with different versions.

3. **Launch the cluster**: Open the browser and go to the EMR page. After the configuration setup press the "Launch the cluster" button. The configuration needs to be entered every time for the new cluster.

4. **Bootstrap the machine**: Prepare a shell script for the download and installation of the dependencies, such as Anaconda Python distribution, Boto3, Pandas and mumpy packages. Open the terminal window, provide the username, hostname and security key and connect to the master node. Open the browser, go to the EMR page and select the "Hardware" tab. The lists the three instance group. Click each of them to get the list of machines and their IP addresses. Open an SSH to each of the machines and execute the prepared shell script that downloads and installs the required dependencies.

5. **Deploy the application**: Take the existing Spark or Hadoop application and transfer it to every machine in the cluster with a SFTP client. Every transfer requires opening the connection, moving the application to remote machine and closing the connection.

6. **Execute the application**: Connect to the master node in the cluster over SSH. Run spark-submit for Spark application or hadoop-streaming for Hadoop applications. Spark can be executed from the root directory. Hadoop streaming requires referencing to the jar file, mapper and reducer executable files and packaging all the dependencies into a tar archive.
7. **Terminate the cluster**: Once all the applications have executed, the cluster can be terminated. Open the browser and go to the EMR page. Select the existing cluster and click the "Terminate button".

The automated workflow enabled by the data platform involves the following actions:

1. **Prepare the environment**: Export AWS credentials as environmental variables.

2. **Setup the configuration**: Edit "cluster.ini", the configuration file. The editing is done for 5 to 10 parameters, where researcher selects the EMR version, master and slave machine instance types and bid price, region for EC2 machines, security groups and security key file for SSH.

3. **Launch the cluster**: Start "emr.py" in python interpreter and issue `launch` command with no parameters. Any existing configuration file can be re-used, e.g. the configuration file edited for commonly used cluster configuration. The `launch` command will spawn an instance of the cluster from the configuration.

4. **Bootstrap the machine**: Machines are bootstrapped automatically by the Cluster Deploy. No action from the researcher is required. The Paramiko module of Cluster Deploy automatically goes through the IP addresses obtained from "describe-cluster" method of Boto3 in order to SSH and install the common dependencies.

5. **Deploy the application**: In the "emr.py" interpreter issue the `deploy` command with one parameter which is the location of the deployment ready package on the researcher’s machine. The Paramiko module of Cluster Deploy hides the SFTP transfer of the client application. Additional dependencies for this client application are installed automatically. The client application is internally distributed across all the machines.

6. **Execute the application**: Start the "emr.py" in the interpreter and issue the `run` command with two parameters. First parameter is the name of the client application deployed on the cluster. The second parameter is the list of arguments, discussed in Section 3.2.1. The Luigi framework within the Cluster Compute can automatically resolve which framework to run based on the provided arguments.

7. **Terminate the cluster**: In the "emr.py" interpreter issue the `terminate` command with no parameters. The Cluster Deploy knows the EMR cluster that it is associated with, therefore no cluster selection is required.

The gain from usage of the data platform comes in the streamlining of the application deployment on the cluster. The manual way involves many small steps and switching between environments resulting into human errors. We
summarize the time taken and the type of the manual and automated computation step of the application development process in Figure 5.1. We compare the ease of use of the application deployment in a manual approach with vanilla EMR setup against the automated approach with the developed data platform. The "start action" refers to the amount of time taken by the researcher to perform the action. The actual time for EMR to launch, execute, bootstrap, etc. remains the same for both automated and manual approach.

Table 5.1: Comparison of the manual and automated application execution on the cluster.

<table>
<thead>
<tr>
<th>Action</th>
<th>Manual (min)</th>
<th>Manual (type)</th>
<th>Automated (min)</th>
<th>Automated (type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepare</td>
<td>1 browser</td>
<td>1 browser</td>
<td>manual</td>
<td></td>
</tr>
<tr>
<td>Setup</td>
<td>5 browser</td>
<td>2 &quot;cluster.ini&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Launch</td>
<td>2 browser</td>
<td>1 browser</td>
<td>command</td>
<td></td>
</tr>
<tr>
<td>Bootstrap</td>
<td>30 (proportional to the number of machines)</td>
<td>SSH (regardless of the number of machines)</td>
<td>automated</td>
<td></td>
</tr>
<tr>
<td>Deploy</td>
<td>15 (proportional to the number of machines)</td>
<td>SSH, SFTP (regardless of the number of machines)</td>
<td>command</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>5 SSH</td>
<td>1 SSH</td>
<td>command</td>
<td></td>
</tr>
<tr>
<td>Terminate</td>
<td>2 browser</td>
<td>1 browser</td>
<td>command</td>
<td></td>
</tr>
</tbody>
</table>

Manual approach to application deployment involves opening the browser, going to the EMR administration panel, selecting the configuration from the list of options and interacting with the launch button. Vanilla EMR is not enough for the researchers’ needs. Additional software needs to be installed. EMR does not provide a way to create the cluster from a custom AMI. Therefore, researchers need to manually open the terminal window to every node in the cluster and install the software through the PIP interface. This procedure does not scale with large clusters. Errors might occur if the researcher forgets to install the dependencies on one of the nodes. We found that in about 1 out of 20 times, the manual approach to application deployment involves errors due to multiple terminal windows being open and issuing commands in the wrong window based upon the application deployment logs obtained from the researchers. Automated approach to application deployment using the developed data platform prevents this issue because it provides a single point of interaction with the
The developed data platform enables the automation of the application deployment process and interactions with the cluster. Applications can be developed in the same fashion regardless of the framework used (Spark or Hadoop). Applications can be deployed to the cluster in a transparent way by auto-wiring all the dependencies required. Transparency is achieved by the Paramiko framework and means that users require only one tool for establishing an SSH connection to the computational cluster. It is transparent because Paramiko does remote connection on behalf of the user and hides the specific details. Researchers do not need an extra tool, everything is done inside of the same CLI. Auto-wiring is the process of resolving the dependencies. In the manual cluster setup, every dependency needs to be wired by the researcher in a manual way by installing the specific software on every machine. With auto-wiring, researcher simply specifies a publicly available PIP package in the configuration file of the client application. Private BitBucket projects can also be specified in the same configuration file. The difference is that PIPs are used publicly by any developer who wishes to use it, while BitBucket can contain a company sensitive projects which should only be internally available. Both can be auto-wired by the data platform. This is done by the Cluster Deploy project. It scrolls through the list of package names in PIP property of the client configuration file and installs the package in the machine. Similarly, Cluster Deploy scrolls through the GIT links to the BitBucket repositories and installs them through GIT clone and PIP install commands. In case of using the BitBucket, the SSH key needs to be put into the web console for the data platform to gain access to the repository (as discussed in Section 3.2.2). The dependencies on PIP or BitBucket packages are required because the client application uses certain classes from within the package. If the package is not imported then the execution of such application will result into a runtime error. Naturally, all new applications are build either on top of or with support of existing applications and libraries. This leads to a necessary feature of having these dependencies installed on the machine in the cluster.

Additionally, one can issue commands to the cluster simply via Command Line Interface (CLI). All these were the requirements for the validation of the data platform. We perform the empirical comparison between the manual application development and execution on the cluster versus the automated workflow enabled by the data platform. This analysis gives the quantitative insight into the ease-of-use of the developed platform and the obtained gains.

For researchers who develop only Spark applications, the cluster is launched with the spark EC2 script. Other alternatives are to use StarCluster or MRJob, both of which are based on EC2 instances. These variations bring complexity and wide dependency. The exchanges and interactions between the project teams using different workflow is hindered. Data platform allows to manage the cluster with one simple CLI and a few commands to support the lifecycle. This feature reduces the time and complexity of managing the cluster and allows the unification into a single workflow for different needs.
Currently, applications on different framework like Spark and Hadoop are developed as a separate standalone program. This creates complexity in managing the execution environment for each of the frameworks. Data platform allows to standardize the execution environment which can handle both of the frameworks in a unified fashion. The launching of the cluster is done with one configuration file and one command with no parameters. On average, it takes about 10 min to launch an EMR cluster.

Currently, dependencies for applications are captured in AMI and instantiated on each EC2 machine. Changes in dependencies force recreation of the AMI. This is a slow process and cannot be done over the running cluster. Data platform allows to easily manage dependencies without the use of custom AMI and auto-wire them at run time. On average, it takes about 2 min to bootstrap data platform with all dependencies required. Researchers specify the dependencies in a configuration file by simply typing a name of the desired PIP library or SSH path to Git on BitBucket.

Currently, Spark copy-dir tool or manual copying is widely used for application deployment. Application dependencies are managed separately. This introduces multiple steps in deployment procedure. Data platform simplifies deployment both in time and complexity. Deployment of the application takes under 1 minute in case of no or few dependencies involved. Usually this is the case, since most of the common scientific packages are already pre-installed during the bootstrap time.

Hadoop applications have a bit complex development procedure due to low level MapReduce operations. This happens because researchers often use Python applications executed through Hadoop streaming. In such case, data is passed between map and reduce steps via standard system input output streams. This limits the ability to debug applications and introduces overhead for processing the streams. Existing wrappers around Hadoop streaming for Python either lack documentation or enforce overly strict Application Programming Interface (API) [22]. Data platform solves this issue by providing a simple API for developing Hadoop applications.

Data platform allows easy porting of existing applications for Spark as is into the data platform. At the same time, Hadoop applications will be required to be slightly rewritten, the cost well justified for the simplification for the workflow.

Currently, commonly the applications are executed in the cluster via standard Spark or Hadoop streaming interfaces. For Spark, the switch between test and production environment is required. For Hadoop streaming, multiple additional parameters is needed, including all the dependencies packed into the archive and mapper/reduce script files. Data platform unifies the execution of applications for both of the frameworks via a parameter switch.

---

1 MapReduce [38] is a programming model and an associated implementation for processing and generating large data sets

2 Python [34] is a high-level, general-purpose, interpreted, dynamic programming language

3 Hadoop streaming [21] allows to run MapReduce jobs with any executable or script
5.2 Data Platform Configurations

Here we answer the CQ3 about what configuration is the best for the particular case of the stress application. There are many types of scientific applications supported by the data platform. However, we decided to focus on one of them, just as an example of the client application.

Here we outline the experimental methods and results obtained for the benchmarking of the selected workloads of the target application for different configurations of the data platform. In the stress application, the feature extractions from the measured ECG and accelerometer signals are one of the computational workload. For given workloads, we benchmark the data platform configuration across different dimensions for the performance comparison. All possible configurations need to be embodied in one application for benchmarking convenience. Thus, both Spark and Hadoop versions need to coexist together. This way, one can easily switch between frameworks without redeploying the application. The ability to cope with different configuration variations on per application basis is considered a strong need for a unified application development process. Such automation is achieved by the capability of the data platform to easily switch between different configurations.

Further in the section, we define the dimensions across which the client application is benchmarked in Section 5.2.1. The procedure for running the experiments is described in Section 5.2.2. Detailed configuration of the experimental environment is provided in Section 5.2.3. In Section 5.2.4 we present the benchmark results.

5.2.1 Dimensions for Benchmark

The stress application research to find the moments of stress based upon the physiological signal is taken as the use case for the validation of the configurability property of the proposed data platform. A single configuration is a valid data point in the available dimensions for the benchmark. We consider the following four dimensions: machines, datasets, frameworks and workloads. Machines are EC2 instance types forming an EMR based cluster. Datasets reflect the size of HDF5 file storing the recorded ECG and accelerometer data. Frameworks represent the large scale data processing framework to run in a distributed computing environment. Workloads in our application are different sets of features from the signal modalities to be computed over collected data. Single workload consists of multiple features and represent different components in the application.

The possible values across different dimensions for the benchmark is presented in Table 5.2. For every option on machine type, there are 16 experiments possible based upon the combination of options from other dimensions. An experiment is defined as a single run of a workload in a given configuration in the cluster. Given 4 different machine types we get 64 experiments in total. Each experiment is repeated 10 times for further analysis on variability of the obtained results and possible statistical tests about the results. The performance
of a particular configuration is analyzed by the performance metrics of running time which can also be extended to include CPU usage, memory usage, network usage and costs.

<table>
<thead>
<tr>
<th>Machines</th>
<th>Datasets</th>
<th>Frameworks</th>
<th>Workloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>m4.4xlarge (general)</td>
<td>Large files</td>
<td>Hadoop</td>
<td>ECG (all features)</td>
</tr>
<tr>
<td>c4.4xlarge (compute)</td>
<td>Small files</td>
<td>Spark</td>
<td>ECG (mean heart rate)</td>
</tr>
<tr>
<td>r3.4xlarge (memory)</td>
<td></td>
<td></td>
<td>ECG (statistical features)</td>
</tr>
<tr>
<td>i2.4xlarge (storage)</td>
<td></td>
<td></td>
<td>Accelerometer (all features)</td>
</tr>
</tbody>
</table>

We decided to use the latest generation of instance types and have configured Virtual Private Cloud (VPC) to meet the EMR requirements. We benchmark the following instance types: general purpose (M type), compute optimized (C type), memory optimized (R type) and storage optimized (I type). We decided to exclude the GPU-optimized instance, as the benefits of such machine is unrelated to our workloads. All machines in the cluster are of the same instance type and capacity. All instance types have the same virtualization type, namely hardware virtual machine (HVM).

5.2.2 Experimental Environment

Experiments with different configurations are run on EMR with data sourced from S3 for input/output. In EMR terminology, machine instance types are grouped into master, core and task slave groups. The master group consists of only one machine and is used to orchestrate the computation on slaves. Core slaves are used for computation and have HDFS storage. Task slaves are only used for computations with no HDFS storage attached.

For the experimental setup we chose to have 1 master node and 3 core group slaves. The reasoning behind the use of core slaves over task slaves is the potential ability to extend persistent storage dimension to support HDFS. Another reason for this choice is the cluster constraint of having a non-empty core instance group. Having split the machines in the core and task groups would have added unnecessary variety between the slaves.

EC2 machines are geographically distributed in different regions. Several availability zones exist and are interconnected by the low latency link within one region. We run in the availability zone (AZ) 1a of us-east region. All communication is done within one zone and no cross-region calls are taking place.

S3 does not have a notion of AZ. However, they are geographically distributed in regions as well. Our data on S3 are placed in the same region as EC2 machines.

Both of Hadoop and Spark implementations run on YARN \(^4\) in client mode.

\(^4\)YARN (Yet Another Resource Negotiator) [42] is a cluster management technology
thus making driver program to operate from the master node (YARN cluster mode is not yet available for Python).

The benchmarking study is done on a subset of the representative data. Data from 10 participants are taken, where each participant being monitored for 5-7 days with measurements taken every millisecond result in a total of 10 files each of 5-7 GB of time-ordered data. Since the dimension of analysis across dataset size, in large dataset we have 1 file per 1 participant and in small dataset we generate 65 smaller size files, each containing only 1 day for 1 participant.

Cluster configurations are based on EMR-4.7.0 version of June 2016 [4]. Additional applications, installed along with Hadoop 2.7.2 distribution are Spark 1.6.1, Mahout 0.11.1 and Ganglia 3.7.2. Few settings were applied on top of providing configuration. For the purpose of Hadoop jobs being able to acquire all the resources, we set "mapreduce.task.timeout" to "0" and "yarn.nodemanager.pmem-check-enabled" to "false". This change default 30 min limit for the execution of the Hadoop job and disables physical memory check for container. We also set "yarn. scheduler.capacity.resource-calculator" to "org.apache.hadoop.yarn.util.resource.DominantResourceCalculator" in order to distribute the workload evenly across the slave nodes.

5.2.3 Experimental Procedure

The following experimental procedure is executed for every workload:

1. Launch the cluster
2. Deploy the stress application
3. Wait for 10 min
4. Run the experiment
5. Wait for 5 min
6. Repeat 2 previous steps 10 times
7. Terminate the cluster

Waiting procedure is necessary to ensure that metrics are not affected by bootstrapping actions or previous executions. Computing all the features from the ECG i.e. ECG (all features) is expected to be the most expensive workload while the accelerometer features are expected to be cheaper in terms of the execution time.

5.2.4 Benchmark Results

As discussed earlier, the results on the benchmark for the execution of the given application components in the data platform serves different purposes.

First, it validates the configurability property of the data platform. Thus, different teams can configure the data platform to match their own needs. The
configuration is done through the configuration file or via parameters of the run. Configuration options have been discussed in Section 3.2.1. Researchers can tune the configuration by running the application over a smaller subset for testing purposes before running over a real data in production.

Table 5.3: Amazon EC2 Linux/UNIX instance types in US East (N. Virginia). Actual on Friday the 13th of May 2016 [18]

<table>
<thead>
<tr>
<th>Model</th>
<th>Category</th>
<th>vCPU</th>
<th>Memory (GiB)</th>
<th>Storage (GB)</th>
<th>Usage ($/hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2.nano</td>
<td>General</td>
<td>1</td>
<td>0.5</td>
<td>EBS-only</td>
<td>0.0065</td>
</tr>
<tr>
<td>t2.large</td>
<td>General</td>
<td>2</td>
<td>8</td>
<td>EBS-only</td>
<td>0.104</td>
</tr>
<tr>
<td>m4.large</td>
<td>General</td>
<td>2</td>
<td>8</td>
<td>EBS-only</td>
<td>0.12</td>
</tr>
<tr>
<td>m4.4xlarge</td>
<td>General</td>
<td>16</td>
<td>64</td>
<td>EBS-only</td>
<td>0.958</td>
</tr>
<tr>
<td>c4.large</td>
<td>Compute</td>
<td>2</td>
<td>3.75</td>
<td>EBS-only</td>
<td>0.105</td>
</tr>
<tr>
<td>c4.4xlarge</td>
<td>Compute</td>
<td>16</td>
<td>30</td>
<td>EBS-only</td>
<td>0.838</td>
</tr>
<tr>
<td>c4.8xlarge</td>
<td>Compute</td>
<td>36</td>
<td>60</td>
<td>EBS-only</td>
<td>1.675</td>
</tr>
<tr>
<td>g2.2xlarge</td>
<td>GPU</td>
<td>8</td>
<td>15</td>
<td>60 SSD</td>
<td>0.65</td>
</tr>
<tr>
<td>g2.8xlarge</td>
<td>GPU</td>
<td>32</td>
<td>60</td>
<td>2 x 120 SSD</td>
<td>2.6</td>
</tr>
<tr>
<td>r3.large</td>
<td>Memory</td>
<td>2</td>
<td>15</td>
<td>1 x 32 SSD</td>
<td>0.166</td>
</tr>
<tr>
<td>r3.4xlarge</td>
<td>Memory</td>
<td>16</td>
<td>122</td>
<td>1 x 32 SSD</td>
<td>1.330</td>
</tr>
<tr>
<td>r3.8xlarge</td>
<td>Memory</td>
<td>32</td>
<td>244</td>
<td>2 x 32 SSD</td>
<td>2.66</td>
</tr>
<tr>
<td>i2.4xlarge</td>
<td>Storage</td>
<td>16</td>
<td>122</td>
<td>4 x 800 SSD</td>
<td>3.410</td>
</tr>
<tr>
<td>i2.8xlarge</td>
<td>Storage</td>
<td>32</td>
<td>244</td>
<td>8 x 800 SSD</td>
<td>6.82</td>
</tr>
<tr>
<td>d2.xlarge</td>
<td>Storage</td>
<td>4</td>
<td>30.5</td>
<td>3 x 2000 HDD</td>
<td>0.69</td>
</tr>
<tr>
<td>d2.8xlarge</td>
<td>Storage</td>
<td>36</td>
<td>244</td>
<td>24 x 2000 HDD</td>
<td>5.52</td>
</tr>
</tbody>
</table>

Second, the obtained benchmark results provide a relevant result to reason about the optimal configuration to be used in the case of the computational workload of the stress application research. We reason about the performance through total application execution and costs. Benchmark machine instance type was m4, c4, r3 and i2 of 4xlarge capacity. As of the 22nd of July 2016, prices for a single EC2 machine instance per hour of c4 type is $0.838, m4 $0.958, r3 $1.33 and i2 $3.41 [18]. The full list of prices is shown in the Table 5.3. Thus, compute optimized is the cheapest to use (c4) while storage optimized is significantly more expensive. EMR poses additional charges for managing the cluster. These charges are equal regardless of the instance type used.

In all combinations of configurations r3 (memory optimized) and i2 (storage optimized) machine instance types showed the worst performance metrics in running time regardless of the framework or datasets sizes used. They are also the most expensive ones among the machine instance types. The best performance was achieved by c4 (compute optimized) followed by m4 (general purpose). These are the cheapest ones at the same time.

Performance metrics for large scale data processing frameworks differ based on machine instance types and computational workloads. Hadoop version for computing the workloads of ECG all features and ECG statistical features run
Figure 5.1: Results of the experiments (Hadoop vs Spark).

(a) Hadoop vs Spark, Data, Accelerometer all features

(b) Hadoop vs Spark, Data, Ecg mean heart rate

(c) Hadoop vs Spark, Data, Ecg all features

(d) Hadoop vs Spark, Data, Ecg statistical features

Figure 5.2: Results of the experiments (Data vs Datasplit, Hadoop).

(a) Data vs Datasplit, Hadoop, Accelerometer all features

(b) Data vs Datasplit, Hadoop, Ecg mean heart rate

(c) Data vs Datasplit, Hadoop, Ecg all features

(d) Data vs Datasplit, Hadoop, Ecg statistical features
Figure 5.3: Results of the experiments (Data vs Datasplit, Spark).

(a) Data vs Datasplit, Spark, Accelerometer all features
(b) Data vs Datasplit, Spark, Ecg mean heart rate
(c) Data vs Datasplit, Spark, Ecg all features
(d) Data vs Datasplit, Spark, Ecg statistical features

on c4 experienced failures due to the insufficient memory capacity. This failure is highlighted in Figure 5.1 (c)(d) with running time of c4 reaching the running time of Spark on r3 and i2. The running time significantly exceeds the running time for any other instance type. Client application fails as YARN kills the application beyond the memory limits container. Spark version of the same configuration was executed successfully. Thus, it is advised to choose Spark when running c4 machine instance type. Hadoop can be used with m4, r3 or i2 instance type without any fatal consequences.

Client application performance depends on the type of the workload used. Computation of accelerometer all features is most efficiently done in Spark regardless of the machine instance type used (Figure 5.1). All ECG related computations give different performances depending on the machine instance type used. For example, Hadoop runs faster on i3 and i2 while Spark runs faster m4 and c4 (Figure 5.2 and Figure 5.3).

Using different size of input datasets does not have a large impact on the Spark performance. Only a slight benefit of using larger files has been reported across all the machine instance types (Figure 5.3). On the other hand, Hadoop tends to favor smaller files over the larger ones regardless of the workload (Figure 5.2). The only exception is the r3 machine instance type for accelerometer all features workload where larger files took slightly less time to compute (Figure 5.2 (a)).
5.3 Concluding Remarks

We have validated the configurability of the data platform and reasoned about the execution time of the stress application. The comparison can be done quickly and easily with the data platform. We have experimentally found the best configuration for our particular application. We have found that c4 and m4 instance types perform the best for stress application workloads for ECG and accelerometer, while i2 and r3 perform the worst. Cost wise, c4 is also the cheapest option. Hadoop version was failing on c4 instance, while Spark ran successfully. Dataset size showed little variation in Spark. Hadoop, however, performs faster with smaller size files rather than larger ones. On average, Spark version of the application runs faster than Hadoop. Thus, the best possible configuration would be to use Spark on c4 machine instance type with the application working with full data.

We have answered DQ1 on how to automate the application development process for researchers with data platform (Chapter 2). The sub question DQ2 about unifying Hadoop and Spark frameworks has been answered by Cluster Compute subsystem of the data platform (Section 2.4.1). The DQ3 about client execution on the cluster is answered by the Cluster Deploy of the data platform (Section 2.4.2). The DQ4 about the constraints for the client application have been introduced from both Cluster Deploy and Cluster Compute sides (Section 2.4). The configurability of the data platform for the purpose of benchmark (CQ1) is evaluated in Section 5.2.1. The performance metrics (CQ2) are given in Section 5.2.4. The best configuration for the particular stress application (CQ3) has been found in Section 5.2.4.
Chapter 6

Conclusions

Various data-driven lifestyle and healthcare applications are made possible with the ubiquity of unobtrusive and comfortable wearable sensors measuring different body physiological parameters. However, data processing of such applications comes with the complexity of developing computational algorithms and managing the infrastructure to run the applications upon. We provide a platform for the development of data analytics applications in the wearable health solutions domain. We developed the data platform for computation support in developing the application using wearable sensors. The platform supports various storage solutions, computational frameworks and execution environment. This allows the platform to support the needs of different application and development teams.

6.1 Future Work

Although, actual work is already being adopted by researchers at IMEC, we suggest the following improvements:

1. The current implementation of the data platform focuses on batch processing only. However, volume is not the only property of big data applications. Velocity must be taken into consideration due to the increasing usage of stream processing algorithms. Such approach can be used for real-time prediction and analysis. Apache Storm \(^1\) and Apache Flink \(^2\) should be considered in the next version of the data platform by extending the Cluster Compute. One option to integrate streaming could be through Lambda Architecture \(^3\) framework. It allows for both batching and streaming applications to co-exist in one framework.

---

\(^1\)Apache Storm [14] is an open source distributed real-time computation system  
\(^2\)Apache Flink [12] is a platform for distributed stream and batch data processing  
\(^3\)Lambda Architecture [26] is a processing architecture for a wide range of workloads and use cases with low-latency connections
2. Cluster Deploy needs to be fault tolerant. It should detect faults during the application execution and cluster launch. Typical errors are: spot instance requests are not fulfilled, availability zone constrains, no machines available, etc. Additionally, there needs to be a way to monitor failed applications and report the reason for failure. Ideally, a meaningful notification should be provided to the researcher via CLI of Cluster Deploy subsystem.

3. Cluster Deploy takes the benefit of the elasticity feature of EMR. Current implementation allows for a simple cluster resizing through task instance groups. New instances are bootstrapped according to EMR version and configuration. However, they are unaware of the application context. New machines can accept the applications scheduled by YARN, but do not have the capabilities to successfully execute them due to the lack of dependencies. After bootstrapping, new instances need to be set up with existing client applications, dependency libraries and Python Anaconda distribution in the same way as it is done during the launch step.

4. Benchmarking should be extended to the different solutions for persistent storage. In our work we provided support for s3 only. We propose to include HDFS as an input/output data storage. This will help to study if it is better to attach EBS with data to machine instance, instead of streaming data directly from S3. Typically, HDFS is considered to be suitable for more frequent data access, in comparison to S3. Some use cases propose to store final input/output data on S3, while using HDFS for intermediate results. There is a room to explore both non-iterative (only HDFS) and iterative (combination of HDFS and S3) options.
Bibliography


[41] Sajee Mathew. Overview of Amazon Web Services, December 2015.