New Software Architecture for Monitoring Mobile Applications

Mobile Data Collection and Indexing for Mobile Application Monitoring

Mourlin Fabrice
Algorithmic, Complexity and Logic Laboratory
UPEC University
Cretéil, France
e-mail: fabrice.mourlin@u-pec.fr

Djiken Guy Lahlou
Applied Computer Science Laboratory
Douala University
Douala, Cameroon
e-mail: gdjiken@fs-univ-douala.cm

Laurent Nel
Leuville Objects
Paris-Saclay University
Paris, France
e-mail: laurent.nel@universite-paris-saclay.fr

Abstract—**The monitoring activity remains an activity that disturbs the system under control. We all try to minimize these disturbances in order to observe a behavior as close as possible to reality. In IT, this requires the implementation of a specific software architecture. Our use case concerns the monitoring of embedded applications on mobile devices for which the collected data sometimes contain errors that we want to explain. To this end, we seek to trace the important events of our calculations in order to qualify the anomalies in our processing. We have implemented a monitoring layer within mobile applications in order to perform intelligent monitoring on a set of mobile devices. We have defined a Big Data workflow to collect, index and store log data for submission to an artificial intelligence (AI) model. A crucial aspect of this collection relies on the use of partitioned topics and thus a better distribution of the data. With the increase of the data flow to be processed, the performance remains insufficient and we have opted for a persistence layer adapted to our data processing. We detect behavioral anomalies through the analysis of software logs deployed on embedded devices. Based on the patterns recognized in the logs, our AI model provides us with a sequence of system operations. These operations are then scheduled to redeploy a service, change a driver, perform a library update, etc. In the end, we build management reports every week for the maintenance team. These documents help track maintenance activities. They provide a record of important events such as equipment downtime or removal of obsolete services.**

Keywords— Big Data; indexing; log analysis; distributed application; AI model; storage efficiency; anomaly detection, explanatory report.

I. Introduction

Software monitoring remains without a doubt an active area of research. The diversity of situations is great because software supports very different deployment constraints. From the application installed on an application server, to the component running on a micro controller, the situations are very different. There is no single platform to monitor them. Therefore, everyone tries to specify his or her typical use case and cover a relevant sub-domain. The monitoring of mobile applications remains complex because it relies on operating systems with specific management rules. A crucial principle is to consider software administration as a facet of any mobile application. Thus, any installed application exposes resources that are used to evaluate its state. This one can be judged abnormal or not and actions are then implemented.

In order to have this information, developers use logging application programming interfaces (APIs) to transmit all the behavioral data. To make the information usable, the log messages usually have a format that facilitates information extraction. Each message usually has a priority level or severity hierarchy and a timestamp associated with an origin. The application administrator manages the level of expressiveness per module in order not to suffer from an excess of information.

The applications of log messages are numerous; they communicate an internal state to the users, but they can offer more like the reconstruction of a state. Database servers like Postgresql have log files containing the history of activity, from database creation to application connection triggers. Embedded applications have the same need. An Android application has access to a logging API, whether it is written in Java, Kotlin or C++. An Android logger corresponds to a variable in memory. It can be stored in a file or sent to a socket.

Log messages play a key role in the life cycle of a project. From the unit, integration, system and functional testing phases, logs are used to highlight processing steps. During development, they provide a view of the application's progress in terms of network, security, activity, etc. In the case of an embedded application, this data cannot be displayed because the device does not necessarily have a screen and it is useful to redirect the information into a persistence unit (memory card, memory stick, etc.). During the debugging phase, these same messages have a tag that allows them to be filtered, or even to specify a different level of severity from one packet to another to configure the feedback. In this study, the effort is focused on the analysis of the log messages used. For this purpose, we use topics on which each device publishes its messages. On the one hand, this allows partitioning the messages into categories; on the other hand, the processing of the published messages is easily distributed [1].

The difficulties linked to the analysis of logs are firstly linked to the volume of messages received. Indeed, this volume grows rapidly with the number of sources. Thus, in the case of monitoring embedded applications, when the number of devices increases, it is no longer possible in human terms to analyze the logs with serenity. The automation of this process is necessary.

A second difficulty is the flow of these messages, which depends on the use of the monitored applications. When the number of users increases, it is then necessary to set up a sampling of information. The third difficulty is that it is not uncommon for each embedded application to have its own log format, even if it is generated by the same API. In this case, it is necessary to consider a standardization of the formats during the collection in order to be able to extract information from different sources and to put them in a causal relationship. After processing, these log messages must also be stored in a persistence system capable of supporting variations in volume, velocity and variable format (3V of Big Data). This persistence system then becomes the underlying information center of the indexing module.

The set of properties related to the processing of log messages naturally leads us to consider this work as a Big Data workflow applied on a temporal scale. Indeed, it is essential to react to detected problems before they get worse. In this paper, we present the results of our work, which started more than two years ago with the Big Data prototyping of an anomaly prediction solution for Android applications. These mobile applications are used to take pictures of biology experiments and visualize them. Users can annotate each photo and reorganize the photos into a document related to a lab experiment. Users export their final document from the mobile device to a web server accessible from the Wi-Fi network at the experiment site.

The rest of this paper is structured as follows. Section II describes the works close to our domain. Section III provides a precise description of our use case. Section IV addresses the software architecture of our distributed platform and more precisely the partitioning of log data. Section V goes into finer details on our streaming approach, which includes an indexing step and a new storage layer oriented distributed document. Section VI focuses on our results, the impact on the maintenance task and the generation of explanatory report. The acknowledgement and conclusions close the article.

II. Related Works

Predictive log analysis is a widely studied topic. Part of the work focuses on enhancing the information itself. A second part concerns the use of this data to react, alert, and more generally automates a process.

A. Log analysis methods

Adam Oliner et al. describe, through several use cases, the information useful to report during execution for software monitoring [2]. They stress, among other things, the importance of adopting a consistent format throughout an application. They make the analogy between the follow-up of manufacturing on one meeting on a production line and the follow-up of the software activity, which is the subject of this work.

T. Yen et al. describe how to leverage distributed application logs for the detection of suspicious activity on corporate networks [3]. Their work highlights the use of the beehive tool for extracting information and producing easily exploitable messages. Analysis against a signature database is then possible.

S. He et al. present six methods for log analysis of distributed systems: three of them are supervised and three others are unsupervised. The authors make a comparative evaluation of these methods on a significant volume of log messages. They emphasize the strengths of software monitoring task automation [4] but the authors do not take into account the model storage regarding the properties of their data.

In more constrained fields such as real time, log analysis systems must be able to detect an anomaly in a limited time. B. Debnath et al. present LogLens that automates the process of anomaly detection from logs with minimal target system knowledge [5]. LogLens presents an extensible index process based on new metrics (term frequency and boost factors). The use of temporal constraint also intervenes in the recognition of behavioral pattern. Therefore, abnormal events are defined as visible in a time window while other events are not. This allows semi-automatic real-time device monitoring.

B. Log analysis and machine learning

The development of machine learning has greatly impacted the use of logs. Depending on the work, studies lead to the detection of anomalies or the discovery of software attacks.

Q. Cao et al. presented a work on web server log analysis for intrusion detection and server load reduction. The use of two-level AI model allowed them to increase the efficiency of their detection system. In this approach, the use of decision trees structures the log data [6].

W. Li considers that logs are a complement to the software-testing phase. Since the time allocated to testing is insufficient, he presents a failure diagnosis strategy based on the use of an AI model [7]. He provides a comparative study between several models.

There is a large body of work on network log analysis for various protocols including HTTP [8] or data-centric protocols such as Named Domain Networking (NDN) [9]. In all cases, the strategy is based on formatted messages where part of the information is filtered and then submitted to a model for prediction. Once again, the nature of the persistence system is not highlighted because the constraints of volume and data rate are not important.

C. Text indexing and storage

Textual data is widely used in Big Data, especially in linguistic analysis. It is mostly unstructured data, not referenced in a database. These data are therefore not interpretable by machines. S. Melink and S. Raghavan have built a distributed full text-indexing algorithm. They propose a storage scheme using an embedded database system of the H2 type [10]. Their results are promising on data from web browsing.

S. Melink and S. Raghavan have defined a novel pipelining technique for structuring the core index-building system. It substantially reduces the index construction time but the data and the index are stored in the same persistence layer. They provide a storage scheme for creating and managing inverted files using an embedded database system [11]. They present performance results obtained during experiments on a distributed web indexing testbed where we see that the data structure has no impact on the type of database used.

M. A. Qader and S. Cheng gather a very interesting comparative study of indexing techniques in the world of NoSQL databases. They allow a fast writing flow and fast searches on the primary key. Some of these persistence systems have added support for secondary indexes. These new indexes are useful for queries on non-primary attributes. Each NoSQL database usually supports a secondary index type. Their conclusion shows that there is no single system, which supports all secondary index types [12]. The authors highlight two classes of indexes: embedded indexes that belong to the storage system and autonomous indexes that are data structures distinct from the stored data. Their results show that none of these indexing techniques dominates the others. On the other hand, embedded indexes provide higher write throughput and are more memory efficient, while standalone secondary indexes provide faster response times when querying. In the end, the optimal choice of secondary index depends on the workload of the application, which is the case when analyzing log messages.

D. *Reporting of artificial intelligence prediction model*

In order to obtain a set of guidelines for the use of predictive machine learning models, it is essential to build regular reports on the quality of predictions. In the context of clinical experiments, W. Luo et al. published a rulebook for AI model development [13].

P. Henderson et al. present a systematic reporting of the energy and carbon footprints of machine learning. The authors' goal is to adapt an efficient reinforcement learning strategy and explain the reinforcement learning events [14]. Events from the environment are associated with their evaluation and recorded. The report traces the life cycle of the AI model.

L.M. Stevens et al. present a recommendation for transparent and structured reporting of Machine Learning (ML) analysis results specifically directed at clinical researchers [15]. Their goal is to convince many clinicians and researchers who are not yet familiar with evaluating and interpreting ML analyses. The model provides evaluation measures that offer a means of comparison between models and underlying strategies.

D.P. Dos Santos et al. take a similar approach to the analysis of radiological images. Their quality is uneven and it becomes difficult to provide a reproducible analysis approach. It then becomes essential to build reports to explain the state of the AI model that led to certain predictions. The authors explain how to structure to help build a post analysis explanation [16].

The use of AI models relies on data from persistence systems. The use of Big Data processing aims to bring data from a data lake into a data lab. This data lab usually consists of a NoSQL database and events such as insertion and update have a strategic role on the life cycle of the associated model. S. Afonin, et al. describe an automatic report generation system based on the database activity. They use a zero-code solution where the underlying software is either Jasper Report [17]. The interest is the provision of a report in a format adapted to the use (Web, pdf, etc.).

III. USE CASE DESCRIPTION

A. *Context Description*

In biology trainings, many experiments are done where students are asked to prepare, perform and follow up. In this context, mobile devices are provided to take pictures, record sounds, or even use the device's sensors to collect data. To save different documents in the memory of the mobile device, a software suite is installed. It allows the authentication of the user, the dating of each collected information and the transfer at the end of the experiment to a server for validation.

During an experiment, all the devices are connected to the laboratory Wi-Fi network. This connection authorizes data exchange with the laboratory server, which will receive all the students' data at the end of the experiment for validation by the supervisor. This network connection is also used to send log messages to monitor the activity of each mobile device. This concerns the capture of information: taking a picture or recording a single comment or a short video. This type of recording is not often used during an experiment because several students are monitoring the same experiment and this leads to noise pollution for the other participants.

The analysis of an experiment by a group of twenty students takes place over a period of one-day maximum in the same experimentation room. This means that the connection is made with the same access point for all devices. Even if the batteries are initially charged, it is possible at any time to have a recharging point in the room.

Laboratory observation sessions can be short in the early grades, such as showing the release of gas bubbles by an aquatic plant. Students construct a document to highlight the conditions of this phenomenon and then make a video to support their comments. Then, they observe the role of light and measure its value with the light sensor of the Android tablet. A second video will show the release of gas bubbles by an aquatic plant in the presence of light. In the absence of light, the students make a comparison with pictures.

In the lab room, a group of students follows an experiment with one tablet per student. Each tablet allows a student to take pictures or videos in order to build his observations of an experiment. The student first saves them locally on the tablet. A typical scenario consists of one Wi-Fi access point per room, a set of mobile devices and a remote storage server for document backups at the end of the session. This scenario is to be multiplied by the number of groups, possibly in parallel in different lab rooms. Two properties are thus highlighted: on the one hand, a local authentication phase on the mobile device, on the other hand a centralized storage server (see Figure 1). In addition, the lab room has a laptop for the teacher and a shared printer. The teacher thus has access to the documents that have been saved on the storage server. Furthermore, he can observe the tablets connected to the lab network and record the addresses of the tablets participating in the study.

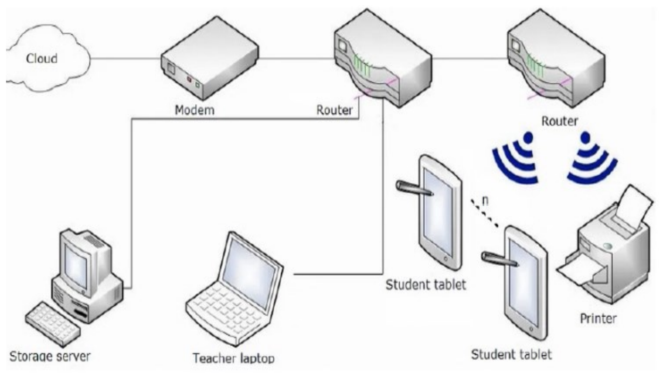


Figure 1. Network diagram of a laboratory room.

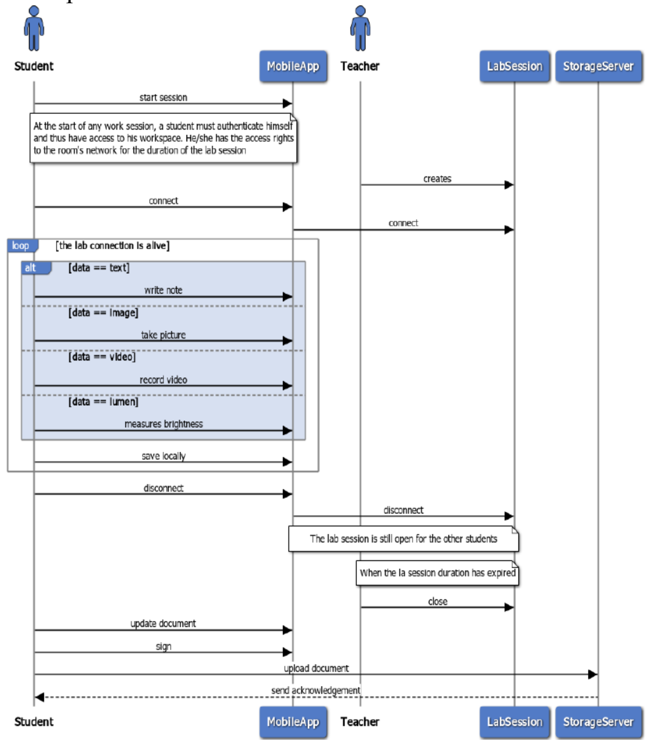
The first router provides a Wi-Fi network to the devices in the lab room. The second router provides access to the Big Data workflow, which starts with a set of message queues available to mobile devices. Without this bridge between the lab room and the data center, we would not have a mobile data responsive architecture.

B. *Scenario description*

In order to describe a nominal scenario more precisely, let us take the case of a student from the beginning of a lab session to the submission of a document at the end of the session.

Figure 2 describes the general flow of the scenario in the Unified Modeling Language (UML) notation; it concerns an observation session (Lab Session). The biology teacher manages this session. Each student manages their own documents locally on their local device. Thus, the student takes notes, photos, videos and measurements via the available sensors. For example, infrared radiation allows the detection of the closest objects. This provides appropriate distance measurements during experiments. When his work is finished or the teacher has closed the session, a student prepares his final document, signs it and deposits it on the storage server.

The storage server receives the work of students for each experiment. The teachers will consult the works to make their post experimentation evaluation.


At the start of any work session, a student must authenticate himself and thus have access to his workspace. He/she has the access rights to the room's network for the duration of the lab session

The lab session is still open for the other students

When the la session duration has expired

Figure 2. Nominal scenario during an experiment in a lab room.

We do not address in this work the management of student-provided materials throughout the academic year. We focus on the aspects related to an experimentation and the follow-up of this activity by processing the associated log messages.

IV. Software Architecture

If the software architecture of the business part is very simple, it is only the entry point of the information collection, which gathers the log data on the storage server. The analysis of these logs is more complex because it takes into account additional constraints: the arrival of log data continuously, the need to impose a data schema to index the information, refine the search for information and the detection of anomalies.

A. *Client application*

In order to collect information from the activity of the actors in the scenario in Figure 2, the log system of the devices is enabled by the students and the teacher. Our goal is to collect and cross-reference information from the various sources. Thus, it is essential to monitor the events related to the management of the laboratory sessions. In addition, any event related to an information capture or modification is useful.

1) *Mobile application*

The *MobileApp* instance in Figure 2 is an Android component installed on each tablet. The set of class is written in Java using the log API specific to this system. In the business part, we have defined a message format in order to easily extract the information. The creation of the log messages occurs by using the *android.util.Log* class, which allows not only to prefix with a semantic tab, but also to add a severity level. Thus, from a set of messages, a regular expression filters the relevant results to focus on the essential data.

In addition to the business events, in this effort, we have traced the memory events provided by the garbage collector, the transmissions and receptions of information from the http network. Moreover, we used the Android Mobility Management API to define usage profiles such as Student profile. It allows business apps and data to be stored in a separate, self-contained space within a device. The teacher has full management control over the applications, data, and Student profile settings on the device, but has no visibility or access to the device's personal profile. This strong separation allows teachers to control *MobileApp* data and security without compromising student privacy if they are using applications other than those intended for the biology course.

We have developed a Device Policy Controller (DPC), which logs network activity. Network activity logging helps us detect and track the spread of malware on tablets. Our DPC uses network logging APIs to report the Transmission Control Protocol (TCP) connections and Domain Name System (DNS) lookups from system network calls.

To further process the logs on our Big Data cluster, we have configured DNS deny lists to detect and alert for suspicious behavior. We have enabled Android network logging to record every event from the *MobileApp* application. It uses the system's network libraries. Network logging records two types of events: DNS lookups and network connections. The logs capture every DNS query that resolves a host name to an IP address. Other supporting DNS queries, such as name server discovery are not logged. The Network Activity Logging API presents each DNS lookup as a DnsEvent instance. Network logging also records an event for each connection attempt that is part of a system network request. The logs capture successful and failed TCP connections, but User Datagram Protocol (UDP) transfers are not recorded. The Network Activity Logging API presents each connection as a ConnectEvent instance. This entire network log configuration is complex, but grouped in a specific concrete class named DevicePolicyManager. The configuration is taken into account asynchronously and it is important to validate it before distributing the tablets to students at each software update.

2) Mobile component

The component deployed on the teacher's laptop is a traditional Java component (version 11) also configured with a message format and a log level. This provides a trace of important events that take place on this workstation. Log analysis is the fastest way to detect what went wrong, which makes logging in Java essential to ensure the performance and health of our distributed application. The goal is to minimize and reduce any downtime, to reduce the mean time to repair.

We used the slf4j library because it represents a simple and highly configurable API. In particular, we have configured the directory where the log messages are saved as well as the expression to generate the file names with the date. The size of the messages is voluntarily limited, so that the subsequent collection is always done within a reasonable time. In addition, the stack trace is provided for any anomaly. Finally, the structure of all logging events follows a pattern consistent with the *MobileApp* component. We have added a log converter to hide some information such as student IDs. It is important that sensitive information is not traced because this data is then transmitted to our Big Data cluster for analysis.

B. Server application

The server application part is deployed on the storage server. This component, also written in Java (version 11), contains the implementation of Web service allowing on the one hand to receive the documents of the students but also to acknowledge the receptions. This part is developed with the Spring Boot library. We use intensively the Spring configuration for the logs, it is indeed a simple way to define a different log level from one package to another but also for the persistence aspects. The database is Postgresql version 10. This database server is used for the persistence of data resulting from the work of students and teachers. As in the previous section, the location of the log files for our server component or for the Postgresql server is imposed. As an example, we record the trace of any http request received by our Web services. The headers are kept as well as the response headers. The version of the http protocol used is http/1.1. In the same way as for the Laptop component, we have imposed a log message format.

C. Big Data architecture

This section focuses on describing our Big Data workflow from collection to building our AI model. We wanted to automate our approach as much as possible because any human intervention leads to blockages or even loss of information.

In this section, we have made technical choices leading to the use of open source software. First, we use the Apache Kafka message queue server. Its role is to receive log messages coming from mobile devices by classifying them by topic. These topics are distributed and Apache Kafka acts as a mediator between two worlds: the mobile device and the Big Data workflow. Secondly, we use the Apache Flume server, which allows defining routes between two or more software. Thus, data can flow from software A to software B. In some aspects, it plays the role of an ETL (Extract, Transform, Load). Third, we use Apache Spark to develop and run our Big Data programs. This library helps build in-memory SQL tables that are then stored in a database such as MongoDB which is a document-oriented NoSQL database. Its strong point is to allow the use of simple foreign keys. Finally, Apache Spark contains a sequencer that manages the execution of the built programs. It plays the role of a Big Data engine. The fourth tool is the Apache SolR server, which plays the role of index creator of the data from the previous SQL tables. This means that we define our data index calculation algorithm with the SolR library. At runtime, Apache SolR only stores the indexes while the data is stored in the MongoDB database. The searches are thus faster. The fifth and last tool is Jasper Report from Tibco. It is a tool for building and automatically generating reports. It plays the role of an information distributor for end users.

1) Data collection

This part deals with the collection of log files in order to send them to a Kafka queue. These Kafka files are the entry points of our Big Data cluster. Because there are 3 different types of components, our best choice was to build an event-based collection based on scenario actions. For the *MobileApp* part, the logs are recorded locally on the device. The sending of the information to a Kafka topic is done when the student sends his final document to the storage server. This approach reduces the number of accesses to the Kafka topic server. Thus, the access point of the lab room has been used to send an http request with an attachment part (the document). This sending is also present in the logs so that the next time only a request is sent, not the same data but only the new ones.

The same approach is used for the laptop component. We use an event-based approach. When the lab session is closed, the logs on the laptop are sent to a Kafka file of the same topic. The message volume is lower, but the information is essential when associating with the logs of the mobile devices.

For the storage server, a repetitive task was our best bid because this central point does not reveal any particular interaction but a continuous flow of data. A cron table was used to collect logs from the Postgresql server and the server component to a Kafka file of the concerned topic. Data are automatically routed to the Kafka server where the topics are managed. All log message traffic can be observed via dedicated Kafka system scripts or by using existing JMX components (Java Message Extension).

2) Big Data analysis

A Big Data cluster that is built from Hortonworks Data Platform (HDP) 3.0.1 virtual machines is used to perform log analysis. This solution offers the advantage of deploying software from the Hadoop ecosystem while remaining open to other installations. Moreover, the Ambari console allows a simple configuration of servers such as Kafka for topics or Flume for routes. Our software architecture for Big Data is based on two software routes from Kafka topics to the persistence system. In a first version presented at the AllData 2022 conference, our persistence layer for log messages was based on the HBase server. Indeed, it is installed by default in the HDP virtual machine and offers a data mapping on top of HDFS Hadoop Distributed File System). This type of column family oriented database has natural advantages for data parallelization. HBase is designed to work with key/value data and random read and write access patterns. Its Java library is easy to use but the types of data accepted are poor or very poor. It is penalizing because our data, essentially textual, also have integer and real fields following our analysis and indexing. The consequences are a cost in time increasing with the volume of processed data. While the times were acceptable for the first full-scale tests, when the number of course sessions increased, we observed very long processing times. Thanks to the monitoring of the VMs, we were able to understand the origin of the waits: access to the region servers, encoding and decoding phases of large numbers of data, overly complex queries, etc.

Figure 3 shows the layer components of our project. Deployed on the lab room platforms, we developed Kafka producers for the peripherals, the teaching laptop and the storage server. All these producers issue log messages in a Kafka topic that is partitioned on the server. This improves the access time to the information.

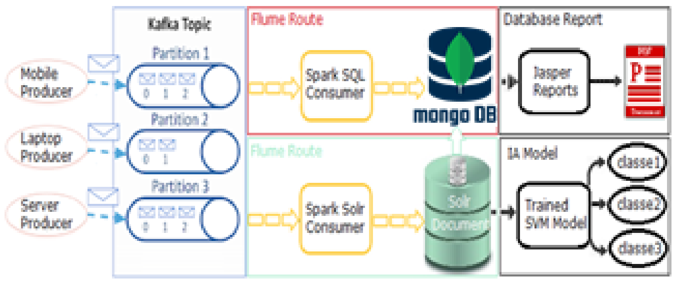


Figure 3. Big Data Software architecture.

The topic partition is the unit of parallelism in Kafka. On both the producer and the broker side, writes to different partitions are done fully in parallel. At the output of the Kafka topics, two Flume routes have been defined within this experiment, each managed by an agent. A first route (red on Figure 3) consumes the messages in order to transform them for some residual format differences and store them in a document-oriented database, MongoDB, installed on the cluster. A second route (green on Figure 3) consumes the messages to index them according to a Solr data schema. Each persistence system has its own role: MongoDB keeps the structured log data and Solr keeps the indexes on these data to enrich the searches. We consider MongoDB and Solr as two data sources accessible from Spark components. The Spark SQL API is easily used to write to MongoDB collections on a Hadoop cluster. In contrast, our Spark to Solr consumer does not have such an easily accessible API and we used Solr Cloud REST services for our updates.

The data indexed by Solr enables our system to classify the messages in order to carry out maintenance operations on the various materials. A relevant option here was a linear classifier with margin calculation. In fact, in several evaluations of AI models, it is established that in the category of linear classifiers, the Support Vector Machine (SVM) are those that obtain the best results. Another advantage of SVMs, and one that is important to note, is that they are very efficient when there is little training data: while other algorithms would fail to generalize correctly, SVMs are observed to be much more efficient. However, when there is too much data, the SVM tends to decrease in performance.

In order to understand the MongoDB events and their distribution on the cluster, we have defined a report template to generate a pdf report. It summarizes the activities by collection, their events, in particular the use of shards. The use of a template guarantees the scalability of these reports according to the evolutions of the consumer SQL Spark. We added a page header with a table name and the current edition date and a page footer with the page number. The column header band is printed at the beginning of each detail column with the column names in a tabular report. This means the part name of a log message.

3) Log Data storage

A first Spark consumer (named "Spark SQL consumer") has an essential task to recognize and process the contents of the file and load them into an SQL table in memory, perform filter operations and put them in a common format. Then, the route continues with a backup of these data in MongoDB collections. The role of this Flume route is to store structured information in a document-oriented database (the red route in Figure 3). In this effort, we experimented keeping software routes with Flume for event routing and defined Kafka topics to ensure decorrelation between components. This makes it possible to simplify the management of components, among other things for software updates. In addition, the Kafka API allows more controls on the management of messages associated with a topic; for example time management. We have added rules to ensure that a received message is processed within an hour (from a configuration file). In that case, the system raises an alert and the data saved in the local file system.

A Flume agent is an independent daemon process, which manages the red route. The Flume agent ingests the streaming data from the Kafka topic source to the Spark SQL sink. The channel between the source and the sink is a temporary storage. It receives the events from the Kafka source and buffers them until they are consumed by Spark sinks. It acts as a bridge between the source and the sinks. We have added a Flume interceptor to decide what sort of data should pass through to the channel. It plays first a filter role in case of unsuitable data from the Kafka source and inserts the time in nanosecond into the event header. If the event already contains a timestamp, it will be overwritten with the current time.

In a previous prototype of this project, we noticed the performance limits of a storage solution based on HBase. Although it is distributed on the cluster, the distribution of data in column families was not well adapted to our data types and our queries when searching for information for the AI model construction. HBase remains ideal for very large-scale use cases but with a simple format. This is not the case with our formatted log messages. HBase offers very fast searches if we are looking for information on a particular key, but MongoDB provides a much richer model that allows us to follow the evolution of information during its life cycle. MongoDB's data model is relational and allows us multi-document ACID transactions, and its query language is rich.

The HBase persistence system was uninstalled from the reference VM to install the MongoDB suite with monitoring and query software. Therefore, MongoDB becomes a part of the Ambari stack. We monitor and manage this service remotely via REST API. MongoDB offers a wide choice of cluster types to create a specific cluster. Each type represents feature limitations and space limitations of the specific cluster. We chose a 10 GB space for our new prototype. Then, we created accounts for the projects that use the collections in the database, with an associated authentication method. We have customized the access privileges to the database. MongoDB schema design works a lot differently than relational schema design because there is no rule and no obvious process. Two different applications that use the same exact data have different schemas if the applications are used distinct manners.

We have created our collections by using JSON schema (JavaScript Object Notation). If this definition comes from the structure of the log messages, we have added additional fields to track the processing (date of analysis, date of inclusion in model, etc.). We have favoured the integration of data in the same collection to reduce joins. We only use tables of a size fixed by the schema [Table I]. Finally, our data schema depends essentially on the access to our data during the construction of the AI model. We used references to data only to avoid duplication of data and avoid data cycles. Log entries are written as a sequence of key-value pairs, where each key indicates a log message field type, such as “origin” or "severity", and each corresponding value records the associated logging information for that field type.

TABLE I. STRUCTURE OF THE MAIN COLLECTION

Field name	Type	Description
ts	DateTime	Timestamp of the log message in ISO-8601 format.
severity	String	Short severity code of the log message.
id	Integer	Unique identifier for the log statement.
context	String	Name of the thread that caused the log statement.
message	String	Log output message passed from the server or driver.
attributes	Object	Optional: one or more key-value pairs for additional log attributes (limit to 10)
tags	Array	Strings representing any tags applicable to the log statement (limit to 10 strings)
origin	String	Network description of the message source.
creation	DateTime	Timestamp of the log message insertion event in MogoDB
consumption	DateTime	Timestamp of the consumption event of the log message into the AI model
report	DateTime	Timestamp of the consumption event of the log message into the generated report

Our Spark SQL consumer uses the Spark SQL module to store data in a MongoDB database whose schema is structured in collections of documents. The labels of these key/value pair are involved in the data schema of the second Spark consumer. Our MongoDB cluster is used in data replication mode. This means that replication is done on a group of MongoDB servers that hold copies of our data. This is a critical property for deployment as it ensures high availability and redundancy, in case of outages and maintenance periods.

4) *Log Data indexing*

In parallel, another route has the role of indexing the data from the logs (green route in Figure 3). From the same Kafka source, a second Spark consumer (named "Spark Solr consumer") takes care of data indexing while respecting the Solr schema. The index is updated for the query steps and then the use of a model for the prediction of maintenance tasks. Solr Cloud is the indexing and search engine. It is completely open and allows us to personalize text analyses. It allows a close link with MongoDB, database so the schemas used by both tools are designed in a closely related way. On our Big Data cluster, the Solr installation is also distributed. In that context, we have four shards with a replication rate equals to three. This allows us to distribute operations by reducing blockages due to frequent indexing. We have configured, not only the schema, but also the data handlers (schema.xml and solrconfig.xml files).

Our schema defines the structure of the documents that are indexed into Solr. This means the set of fields that they contain, and define the datatype of those fields. It configures also how field types are processed during indexing and querying. This allows us to introduce our own parsing strategy via class programming. Having evolved our persistence layer in order to have a richer data model, it seems natural to choose a data schema compatible between MongoDB on the one hand and Apache Solr on the other hand. The data processing being separate, we had to adapt ourselves in order to make them match as well as possible. For example, the Datetime type of MongoDB plays the same role as the DatePointField type of Solr but its interpretation is not identical.

The Spark Solr consumer uses the Spring Data and SolrJ library to index the data read from the Kafka topic. It splits the data next to the Solr schema where the description of each type includes a "docValue" attribute, which is the link to the MongoDB identifier. For each Solr type, our configuration provides a given analyzer. We have developed some of the analyzers in order to keep richer data than simple raw data from log files. Finally, the semantic additions that we add in our analysis are essential for the evaluation of Solr query. Likewise, we store the calculated metrics in MongoDB main collection as an attribute for control. SolrCloud is deployed on the cluster through the same Zookeeper agents. Thus, the index persistence system is also replicated. We therefore separate the concepts of backup and search via two distinct components. This reduces the blockages related to frequent updates of our Mongo database [18].

At the beginning of our Solr design, we have built our schema based on our data types. Some of them were already defined, but some others are new. In addition, we have implemented new data classes for the new field types. For example, we used RankFieldType as a type of some fields in our schema. It allows us to manage enumeration values from the log message. Then, it becomes a sub class of FieldType in our Solr plugin. We have redesigned Solr filters so that they can be used in our previous setups. Our objective was to standardize the values present in the logs coming from different servers. Indeed, the messages provide information of the form <attribute, value> where the values certainly have units. However, the logs do not always provide the same units for the same attribute calculation. The analysis phase is the place to impose a measurement system in order to be able to compare the results later. The development pattern proposed by SolrJ is simple because it proposes abstract classes like TokenFilter and TokenFilterFactory then to build inherited classes. Then we have to build a plugin for Solr and drop it in the technical directory agreed in the installation of the tool [19].

5) *Model factory*

In Artificial Intelligence, Support Vector Machine (SVM) models are a set of supervised learning techniques designed to solve discrimination and regression problems. SVMs have been applied to a large number of fields (bioinformatics, information research, computer vision, finance, etc.) [20]. SVM models are classifiers, which are based on two key ideas, which allow to deal with nonlinear discrimination problems, and to reformulate the ranking problem as a quadratic optimization problem. In our project, SVMs can be used to decide to which class of problem a recognized sample belongs. The weight of these classes if linked to the Solr metrics on these names. This amounts to predicting the value of a variable, which corresponds to an anomaly.

All filtered log entries are potentially useful input data if it is possible that there are correlations between informational messages, warnings, and errors. Sometimes the correlation is strong and therefore critical to maximizing the learning rate. We have built a specific component based on Spark MLlib. It supports binary classification with linear SVM. Its linear SVMs algorithm outputs an SVM model [21]. We applied prior processing to the data from our Mongo collections before building our decision modelling. These processes are grouped together in a pipeline, which leads to the creation of the SVM model with the configuration of its hyper-parameters such as weightCol. Part of the configuration of these parameters comes from metrics calculated by our indexing engine (Figure 2). Once created and tested, the model goes into action to participate in the prediction of incidents. We use a new version of the SVM model builder based on distributed data augmented. This comes from an article written Nguyen, Le and Phung [22].

6) *Report generation*

Jasper Report library allows us to build weekly graphical reports on indexing activity. MongoDB events are collected for displaying. The goal is to correlate the volumes of data saved in the database with the updates of the AI model. We would like to refine this report template in order to have metrics to decide on the model update. Currently, only MongoDB movements are represented graphically. Based on an MongoDB handler, we handle the change events at runtime and send data beans to the Jasper Report Server.

Jasper Report has its own query language in JSON format. It allows you to specify the data to be extracted from MongoDB. The connector converts this request into appropriate API calls and uses the MongoDB Java connector to query our MongoDB cluster. In particular, it is possible to perform aggregation in the form of map-reduce, but more efficient than a simple pipeline. The map-reduce key specifies a map-reduce operation whose result will be used for the current query. The collection name specifies the target collection. This optimized extraction gives us better performance when building the AI model.

V. Data Streaming Part

A. *Data streaming*

Our component called Spark SQL Consumer contains a Kafka receiver class, which runs an executor as a long-running task. Each receiver is responsible for exactly one input discretized stream (called DStream). In the context of the first Flume route, this stream connects the Spark streaming to the external Kafka data source for reading input log data.

As an example in Table II, we provide an example of a log messages from a Kafka topic called “RedTopic”:

TABLE II. Message from Kafka topic

```
{
  "ts": "2020-10-02T18:10:22Z",
  "severity": "INFO",
  "id": 192674,
  "context": "kafka.server.KafkaServer",
  "message": "1000ms metadata for topic =
RedTopic    partition    =    part00002    not
propagated to all brokers",
  "attributes": {
        "process-id": 4122,
        "event-type": "ReadEvent"
  },
  "tags": [
       "Startup"
  ]
}
```

Structured Streaming, allows us to write streaming queries in the same way as batch queries. Spark streaming uses micro-batch processing, which means that data is delivered in batches to executors. If the executor idle time is less than the time required to process the batch, the executor is constantly added and then removed. If the executor idle time is longer than the batch time, the executor is never deleted. Therefore, we have disabled dynamic allocation by defining when running streaming applications. A Kafka partition is only be consumed by one executor, one executor consumes multiple Kafka partitions. This is consistent with Spark Streaming.

B. *Filtered log strategy*

Because the log data rate is high, our component reads from Kafka in parallel. Kafka stores the data logs in topics, with each topic consisting of a configurable number of partitions. The number of partitions of a topic is an important key for performance considerations as this number is an upper bound on the consumer parallelism. If a topic has N partitions, then our component can only consume this topic with a maximum of N threads in parallel. In our experiment, the Kafka partition number is set to four.

Since log data are collected from a variety of sources, data sets often use different naming conventions for similar informational elements. The Spark SQL Consumer component aims to apply name conventions and a common structure. The ability to correlate the data from different sources is a crucial aspect of log analysis. Using normalization to assign the same terminology to similar aspects can help reduce confusion and error during analysis [21]. This case occurs when log messages contain values with different units or distinct scales. The log files are grouped under topics. We apply transformations depending on the topic the data come from. The filtered logs are cleaned and reorganized and then are ready for an export into a MongoDB instance.

In the next step, the Spark SQL Consumer component inserts the cleaned log data into memory data frames backed to a schema. We have defined a mapping between MongoDB and Spark tables, called Table Catalog. There are two main difficulties of this catalog.

a) The identifier definition implies the creation of a specific index generator in our component.

b) The mapping between table column in Spark and the document in MongoDB needs a component for dynamic data frame creation with Spark SQL.

The MongoDB sink exploits the parallelism on the set of MongoDB nodes. A MongoDB replication is enforced by providing data redundancy and high availability over more than one MongoDB server. In addition to fault tolerance, replica sets also provide extra read operations capacity for the MongoDB cluster, directing clients to the additional servers, subsequently reducing the overall load of the cluster. A MongoDB cluster has a primary node and a set of secondary nodes in order to be considered a replica set. One primary node exists, and its role is to receive all write operations. All changes on the primary node's data sets are recorded in a special capped collection called the operation log (oplog). The role of the secondary nodes is to then replicate the primary node's operation log and make sure that the data sets reflect exactly the primary's data sets.

The driver Spark generates tasks per data set. The tasks are sent to the preferred executors collocated with a Mongo server, and are performed in parallel in the executors to achieve better data locality and concurrency. By the end of an exportation, a timed window a log data are stored into MongoDB collections.

C. *Index construction and query*

The strategy of the Spark Solr Consumer component deals with the ingestion of the log data into Apache Solr for search and query. The pipeline is built with Apache Spark and Apache Spark Solr connector. Spark framework is used for distributed in memory compute, transform and ingest to build the end-to-end pipeline.

The Apache MongoDB role is the log storage and the Apache Solr role is the log indexing. Both are configured in cloud mode Multiple Solr servers are easily scaled up by increasing server nodes. The Apache Solr collection, which plays the role of a SQL table, is configured with shards. The definition of shard is based on the number of partitions and the replicas rate for fault tolerance ability. The Spark executors run a task, which transforms and enriches each log message (format detection). Then, the Solr client takes the control and send a REST request to Solr Cloud Engine. Finally, depending on the Solr leader, a shard is updated.

We use also Solr Cloud as a data source Spark when we create our ML model. We send requests from Spark ML classes and read results from Solr (with the use of Solr Resilient Distributed Dataset (SolrRDD class). The pre statement of the requests is different from the analysis of the log document. Their configuration follows another analysis process. With Spark SQL, we expose the results as SQL tables in the Spark session. These data frames are the base of our ML model construction. The metrics called Term Factor (TF) and Inverse Document Frequency (IDF) are key features for the ML model. We have also used boost factor for customizing the weight of part of the log message. The data in the database is updated with the addition of attributes to the documents that have just been read. Finally, the use of this data for the AI model is recorded in order to distinguish it from new information to come.

VI. RESULTS AND TASK MAINTENANCE

We have several kinds of results. A part is about our architecture and the capacity to treat log messages over time. Another part is about the classification of log messages. The concepts behind SVM algorithm are relatively simple. The classifier separates data points using a hyperplane with the largest amount of margin. In our working context, the margin between log patterns is a suitable discriminant.

A. *Data features*

For our tests, we used previously saved log files from a month of application server and database server operations. We were interested in the performance of the two Spark consumers: For Spark SQL Consumer, the volume of data to analyze is 102.9 M rows in HBase. To exploit this data, we used a cluster of eight nodes on which we deployed Spark and MongoDB. The duration of the tests varies between 38 minutes and 3 hours and 51 minutes.

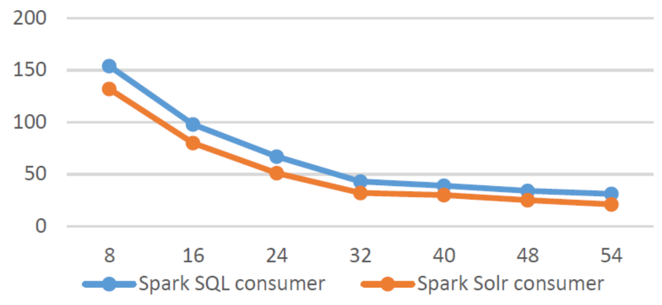


Figure 4. Spark consumer runtime versus number of partitions.

For Spark Solr Consumer, the volume of data indexed is 100.5M rows indexed in about an hour. The number of documents indexed per second is 35k. We only installed Solr on four nodes with four shards and a replication rate of three. We have seen improved results by increasing the number of Spark partitions (RangePartitioner). At runtime for our data set based on a unique log format, the cost of Spark SQL consumer decreases when the partitioning of the dataset increases, as illustrated in Figure 4. The X-axis represents the partition number and the Y-axis represents the time consuming. We have to oversize the partitions and the gains are much less interesting.

SVM offers very high accuracy compared to other classifiers such as logistic regression, and trees. There are several modes of assessment. The first is technical; it is obtained thanks to the framework used for the development (Spark MLlib). The second is more empirical because it relates to the use of this model and the anomaly detection rate on a known dataset. The analytical expression of the features precision, recall of retrieved log messages that are relevant to the find: Precision (1) is the fraction of retrieved log messages that are relevant to the find:

$$precision = \frac{|\{relevant\ \mathrm{log}\ messages\} \cap \{retrieved\ log\ messages\}|}{|\{retrieved\ \mathrm{log}\ messages\}|} \qquad (1)$$

Recall (2) is the fraction of log messages that are relevant to the query that are successfully retrieved:

$$recall = \frac{|\{relevant\ \mathrm{log}\ messages\} \cap \{retrieved\ log\ messages\}|}{|\{relevant\ \mathrm{log}\ messages\}|} \qquad (2)$$

$$F_\beta = (1+\beta^2) * \frac{precision*recall}{(\beta^2*precision)+recall} \quad (3)$$

In Table III, we have four classes and for each class we compute three metrics: true positive (tp), false positive (fp) and false negative (fn). For instance, for the third class, we note these numbers tp3, fp3 and fn3. From these values, we compute precision by label, recall by label and F-score by label.

TABLE III. SVM MODEL MEASURES

Class number	Metrics		
	Precision by label	*Recall by label*	*F1 score by label*
0.0000	0.8158	0.8901	0.8966
1.0000	0.9110	0.9810	0.9910
2.0000	0.8545	0.7857	0.8515
3.0000	0.8524	0.7589	0.8331

Our prediction models are similar to a multiclass classification. We have several possible anomaly classes or labels, and the concept of label-based metrics is useful in our case. Precision is the measure of accuracy on all labels. This is the number of times a class of anomaly has been correctly predicted (true positives) normalized by the number of data points. Label precision takes into account only one class and measures the number of times a specific label has been predicted correctly normalized by the number of times that label appears in the output. The last observations are:

- Weighted precision = 0.9017
- Weighted recall = 0.9318
- Weighted F1 score = 0.9817
- Weighted false positive rate = 0.04009

Our results for four classes are within acceptable ranges of values for the use of the model to be accepted.

The test empirical phase on the SVM model was not extensive enough to be conclusive. However, our results suggest that increasing the number of log patterns deteriorates the performance. In addition, we defined a finite set of log patterns for a targeted anomaly detection approach.

B. *Reporting*

1) *From storage activities*

MongoDB is not well designed for analytics purpose, we have set up a SQL data warehouse, and we have used Apache Camel to load data into a H2 warehouse. Apache Camel is a simple ELT (Extract Load and Transform) data from a data source into a SQL sink [23]. Thanks to the events performed on our collections, we can visualize the traffic variations on our collections according to the sampling duration we have chosen (Figure 5). We also monitor AI model updates following log message collections. These results are first displayed in the weekly report. We also count events related to the use of damaged replicas as well as all incidents during our data retrievals. In the end, we currently get a set of tables summarizing the activity on the data in our persistence layer. As a first observable result, we see that the number of unavailability is much lower than in our previous HBase-based prototype.

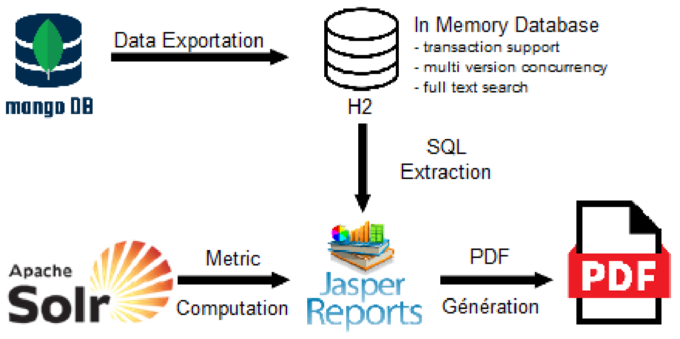


Figure 5. JSR and two data sources

2) *From indexing activities*

We have created a custom data source to connect to Apache Solr, therefore we are able to retrieve data and provide them back in following the JRDataSource interface of Jasper Report. With this access point, we have extracted metrics about the document cache and Query result cache. Both give an overview of the Solr activities and is meaningful for the analysts. We have deployed the CData JDBC Driver on Jasper Reports to provide MongoDB data access from reports. We have found that running the underlying query and getting the data to our report takes the most time. When we generate many pages per report, there is overhead to send that to the browser.

For the reporting phase, we have developed two report templates based on the use of a JDBC adapter. With system requests, we collect data about the last events (Create, Update, and Delete). From these H2 view, we have designed the report templates with cross tables. For the storage phase, we compute and display the number of Update events per timed window or grouped over a period. We periodically updated the data across report runs. We export the PDF files to the output repository where a web server manages them.

VII. CONCLUSION AND FUTURE WORK

We have presented our approach on log analysis and maintenance task prediction. We showed how an index engine is crucial for a suitable query engine. We have developed specific plugin for customizing the field types of our documents, but also for filtering the information from the log message. Because indexing and storage are the two sides of our study, we have separated the storage into a Hadoop database. We have stressed the key role of our Spark components, one per data source. The partition management is the key concept for improving the performance of the Spark SQL component. The data storage into data frames during the micro batches is particularly suitable for the management of flows originating from Kafka files. We observed that our approach supported a large volume of logs.

From the filtered logs, we presented the construction of our SVM model based on work from the Center for Pattern Recognition and Data Analytics, Deakin University, (Australia). We were thus able to classify the recognized log patterns into classes of anomalies. This means that we can identify the associated maintenance operations. Finally, to measure the impact of our distributed analysis system, we wanted to build automatically reports based on templates and highlight indexing and storage activity. Our study also shows the limits that we want to push back, such as the management of log patterns. The use of an AI model is not the guarantee of an optimal result. We want to make more.

A first perspective will be to improve the indexing process based on a custom schema. We think that the use of DisMax query parser could be more suitable in log requests where messages are simple structured sentences. The similarity detection makes DisMax the appropriate query parser for short structured messages. The log format has a deep impact on the Solr schema definition and about the anomaly detection. We are going to evolve our approach. In the future, we want to extract dynamically the log format instead of the use of a static definition. We think also about malicious messages, which can perturb the indexing process and introduce bad request in our prediction step. The challenge needs to manage a set of malicious patterns and the quarantine of some message sequences.

A second perspective on the performance comparison obtained with the MongoDB cluster with replication and a sharded MongoDB cluster or horizontal scaling. In that context, data are distributed across many MongoDB servers. The main purpose of sharded MongoDB is to scale reads and writes along multiple shards and then to reduce the communication time.

REFERENCES

[1] F. Mourlin, G. L. Djiken and N. Laurent, “Big Data for Monitoring Mobile Applications,” The 8th International Conference on Big Data, Small Data, Linked Data and Open Data, ALLDATA, IARIA. April 2022.

[2] A. Oliner, A. Ganapathi, and W. Xu, “Advances and challenges in log analysis,” Communications of the ACM, 2012, 2nd ed., vol. 55, pp. 55-61.

[3] T. F. Yen et al., “Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks,” In Proceedings of the 29th Annual Computer Security Applications Conference, pp. 199-208, December 2013.

[4] S. He et al., “Experience report: System log analysis for anomaly detection,” In 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), pp. 207-218, October 2016.

[5] B. Debnath et al., “Loglens: A real-time log analysis system,” In 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 1052-1062, July 2018.

[6] Q. Cao, Y. Qiao, and Z. Lyu, “Machine learning to detect anomalies in web log analysis,” In 2017 3rd IEEE International Conference on Computer and Communications (ICCC), pp. 519-523, December 2017.

[7] W. Li, “Automatic log analysis using machine learning: awesome automatic log analysis version 2.0.,” 3 edition, November 2013.

[8] A. Juvonen, T. Sipola, and T. Hämäläinen, “Online anomaly detection using dimensionality reduction techniques for HTTP log analysis,” Computer Networks 91, pp. 46-56, November 2015.

[9] J. Dongo, C. Mahmoudi, and F. Mourlin, “Ndn log analysis using big data techniques: Nfd performance assessment,” In 2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService), pp. 169-175, March 2018.

[10] S. Melink et al., “Building a distributed full-text index for the web,” ACM Transactions on Information Systems (TOIS), 2001, vol.19, n°3, pp. 217-241.

[11] J. He, H. Yan, and T. Suel, “Compact full-text indexing of versioned document collections,” In Proceedings of the 18th ACM conference on Information and knowledge management, pp. 415-424, November 2009.

[12] Q. M. Abdul, S. Cheng, and V. Hristidis, “A comparative study of secondary indexing techniques in LSM-based NoSQL databases,” In Proceedings of the 2018 International Conference on Management of Data, pp. 551-566, May 2018.

[13] W. Luo et al., “Guidelines for developing and reporting machine learning predictive models in biomedical research: a multidisciplinary view,” Journal of medical Internet research, 12 ed., vol. 18, 2016.

[14] P. Henderson et al., “Towards the systematic reporting of the energy and carbon footprints of machine learning,” Journal of Machine Learning Research, 2020, 248 ed., vol. 21, pp. 1-43.

[15] L. M. Stevens et al., “Recommendations for reporting machine learning analyses in clinical research. Circulation: Cardiovascular Quality and Outcomes,” 2020, 10 ed., vol. 13.

[16] D. P. Dos Santos and B. Baeßler, “Big data, artificial intelligence, and structured reporting,” European radiology experimental, 2018, 1st ed., vol. 2, pp. 1-5.

[17] S. Afonin, A. Kozitsyn, and I. Astapov, “SQLReports yet another relational database reporting system,” In 2014 9th International Conference on Software Engineering and Applications (ICSOFT-EA) IEEE, pp. 529-534, August 2014.

[18] K. Koitzsch, “Advanced Search Techniques with Hadoop, Lucene, and Solr,” Pro Hadoop Data Analytics, Apress, Berkeley, CA, 2017, pp. 91-136.

[19] J. Kumar, “Apache Solr search patterns,” Packt Publishing Ltd, 2015.

[20] M. F. Ghalwash, D. Ramljak, and Z. Obradović, “Early classification of multivariate time series using a hybrid HMM/SVM model,” 2012 IEEE International Conference on Bioinformatics and Biomedicine, IEEE, pp. 1-6, 2012.

[21] M. Assefi, E. Behravesh, G. Liu, and A. P. Tafti, “Big data machine learning using apache Spark MLlib,” 2017 IEEE International Conference on Big Data (Big Data), 2017, pp. 3492-3498.

[22] T. D. Nguyen, V. Nguyen, T. Le, and D. Phung, “Distributed data augmented support vector machine on Spark,” 23rd International Conference on Pattern Recognition (ICPR), IEEE, 2016.

[23] F. Gosewehr et al., “Apache camel based implementation of an industrial middleware solution,” In 2018 IEEE Industrial Cyber-Physical Systems (ICPS), IEEE, pp. 523-528, May 2018.