The evolution of the CMS monitoring infrastructure

Christian Ariza-Porras[1], *Valentin* Kuznetsov[2], *Federica* Legger[3,*], *Rahul* Indra[4], *Nikodemas* Tuckus[5], and *Ceyhun* Uzunoglu[6] on behalf of the CMS Collaboration

[1]Universidad de los Andes, Colombia
[2]Cornell University, Ithaca NY, 14850 USA
[3]Istituto Nazionale di Fisica Nucleare, via Pietro Giuria 1, 10125 Torino, Italy
[4]Indian Institute of Engineering Science and Technology, Shibpur, India
[5]Imperial College London, London, SW7 2AZ, UK
[6]CERN, Geneva, Switzerland

Abstract. The CMS experiment at the CERN LHC (Large Hadron Collider) relies on a distributed computing infrastructure to process the multi-petabyte datasets where the collision and simulated data are stored. A scalable and reliable monitoring system is required to ensure efficient operation of the distributed computing services, and to provide a comprehensive set of measurements of the system performances. In this paper we present the full stack of CMS monitoring applications, partly based on the MONIT infrastructure, a suite of monitoring services provided by the CERN IT department. These are complemented by a set of applications developed over the last few years by CMS, leveraging open-source technologies that are industry-standards in the IT world, such as Kubernetes and Prometheus. We discuss how this choice helped the adoption of common monitoring solutions within the experiment, and increased the level of automation in the operation and deployment of our services.

1 Introduction

A tiered distributed computing infrastructure is used to store and process data collected and produced by the CMS experiment [1] at CERN. Access to the distributed computing infrastructure, authentication, workload management, and data management are handled by a suite of central services and components. CMS compute nodes are provisioned through GlideinWMS [2] and are available as execution slots in a Vanilla Universe HTCondor pool [3]. Specific tools handle job submission. WMAgent is used for central data processing and Monte Carlo production jobs, and CRAB for user jobs [4]. The data management system includes several components. PhEDEx [5] used to be the data transfer and location and has been replaced by Rucio [6]. DBS [7] is the Data Bookkeeping Service, a metadata catalog. DAS [8], the Data Aggregation Service, is designed to aggregate views and provide them to users and services. Data from these services are available to CMS collaborators through a web suite of applications known as CMSWEB. A detailed description of the computing model of the LHC experiments can be found in [9].

The monitoring system for such a complex computing infrastructure must fulfill a diverse and comprehensive set of requirements:

*e-mail: federica.legger@cern.ch

- The status of all computing systems must be monitored in real time using predefined views.
- Detailed information to debug issues must be easily accessed by the operation teams. To pinpoint the root cause of a specific problem, correlating information from different subsystems is often crucial.
- An efficient alert system to collect and sort metrics from several sources, raise alerts in case specific conditions are met, and route the alerts to the right team must be in place.
- Relevant metrics to monitor the evolution of the performances of the systems over time must be available and allow for in-depth analyses of the main system parameters (such as data access patterns, wall-time consumption of computing resources, memory, CPU and storage usage).

A successful monitoring system should be easy to use, develop, maintain, and extend; meet the experiment requirements for monitoring data storage; provide easy access to the data (both programmatically and from a graphical interface), favour the adoption of common solutions and minimise custom developments.

2 The CMS monitoring infrastructure

The main components of the CMS monitoring infrastructure [10] are the monitoring services offered by the MONIT team [11] of the CERN IT department, and dedicated CMS monitoring applications and services which are deployed on Kubernetes (k8s) clusters [12].

The MONIT ecosystem is extensively used by CMS to inject and store data from various CMS computing subsystems, such as HTCondor job monitoring data, CMSWEB user activities, metrics about analysis and Monte Carlo (MC) production workflows coming from the WMAgent and CRAB job submission tools. The MONIT architecture is based on ElasticSearch [13], HDFS [14], and InfluxDB [15] for data storage. Kibana [16] and Grafana [17] are used for data visualization and access. Data on HDFS can be accessed through Apache Spark [18] workflows and the SWAN [19] service at CERN. Data is injected through ActiveMQ [20] messages or an HTTP endpoint (the latter option only for data providers inside the CERN network boundaries). On average, CMS producers send more than 3.5 million messages per hour to the ActiveMQ brokers, with rates close to 7.5 KHz. The total size of CMS data stored in ES is more than 30 TB, with a daily index average of around 30 GB. In HDFS we collected 25 TB of compressed data[1] over the last five years, with a daily average of around 300 GB.

The dedicated CMS monitoring infrastructure provides monitoring of individual CMS services and nodes via Prometheus [21] with VictoriaMetrics (VM) [22] as a backend (see Sect. 2.1 and Sect. 2.1.1). Alerts based on Prometheus metrics are handled by AlertManager (AM) [23]. The Prometheus service currently covers more than one hundred computing nodes with 125 exporters providing more than three thousand measurements and almost one hundred different alert records and alert rules. In VM we store around 500 billion data points with a data retention policy of thirty days. The CMS monitoring k8s infrastructure allows us to deploy and scale services with minimal operational and maintenance effort.

2.1 The CMS monitoring Kubernetes infrastructure

The maintenance of the CMS monitoring infrastructure represents certain challenges, such as service deployment, version control, and resource utilization. The CMS monitoring services were gradually migrated to a Kubernetes infrastructure. We currently run and maintain four individual clusters: the main CMS monitoring cluster, two High-Availability (HA) clusters for

[1]The majority of data is stored in JSON format which allows to achieve a 90% compression level.

critical services, and a cluster for NATS (Neural Autonomic Transport System) services [24]. This setup allows us to independently manage the k8s resources for upgrades, re-configuration, hardware allocations, and to apply different security and access policies to each cluster.

The most critical components (Prometheus, VM, and AM services) of the CMS monitoring infrastructure are deployed in High-Availability (HA) mode as a protection against outages, and to improve service availability. We carefully evaluated, implemented and tested the architecture depicted in Fig. 1.

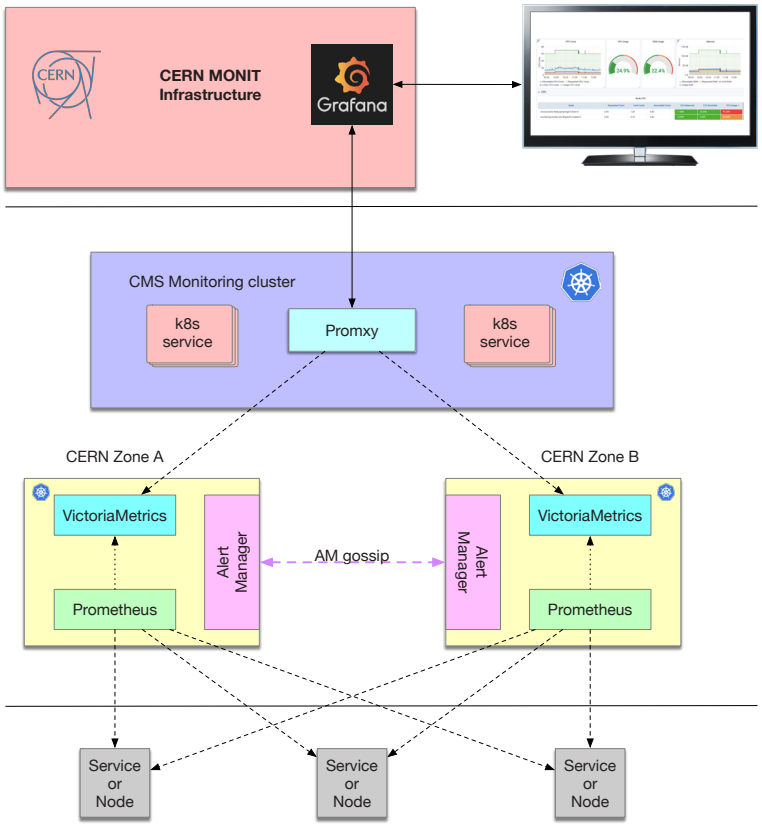

Figure 1. The High Availability mode of the CMS Monitoring infrastructure consists of three k8s clusters, the main CMS monitoring cluster and two independent HA clusters. The main cluster runs a Promxy proxy server, which is used to access the services on the two HA ones. Each HA cluster runs Prometheus, AlertManager and VictoriaMetrics services. The Prometheus service scrapes metrics from CMS services and nodes, and it is configured to access both AM services in the HA clusters. The AM can exchange information through a gossip-based mechanism if necessary.

The HA architecture consists of three k8s clusters allocated in different zones in the CERN network. The main cluster runs non-critical services as well as a Prometheus Promxy [25] proxy server. The Promxy server is used to access two independent HA k8s clusters which provide Prometheus and VM services. The Prometheus server scrapes metrics from a set of CMS services and stores them in the VM backend. The Promxy server guarantees that if an outage happens in one of the HA cluster it can repopulate metrics from the other one. The alerts are managed by individual AM services in each HA cluster. Similarly, AM is setup in cluster mode such that the information across the HA clusters can be populated through gossip

protocol. Such infrastructure provides us with fault-tolerance and it is easy to maintain via the k8s deployment procedure.

The main CMS monitoring cluster fits into two nodes with 16 CPU cores and 30 GB RAM, and hosts 48 services:

- The **Promxy** service to access VM on the HA clusters,
- A set of Apache **Sqoop** [26] jobs used to create snapshots in HDFS of heavily populated CMS databases, such as DBS, PhEDEx, Rucio,
- the **NATS subscribers** which listen to various real-time messaging channels from the NATS server, as described in Sect. 2.1.1,
- the **HTTP exporters** to monitor the status of our services,
- The **Intelligent Alert system** to process alerts from external services such as the CERN SSB (Site Status Board) [27] and GGUS (Global Grid User Support) ticketing system [28], and generate annotations for Grafana dashboards, as described in Sect. 4.1,
- the **Rumble** service to provide query access to HDFS data as discussed in Sect. 4.3.

The HA clusters are much smaller, each HA cluster has one node with 8 CPU cores and 16 GB of RAM.

2.1.1 NATS

NATS (Neural Autonomic Transport System) is a simple, secure, and high performance open-source messaging system for cloud-native applications. It provides a real-time messaging infrastructure to monitor the status of several processes, such as CMS production workflows and campaigns. The NATS service is deployed on a dedicated k8s cluster. It is accessible to all CMS collaborators and services through token-based authentication even outside of the CERN firewall. The NATS service works as a proxy between CMS data providers such as CMSSW (the CMS software framework), DBS, and WMAgent, and data subscribers located either on the client infrastructure or within the CMS monitoring cluster. In the latter case, we run a set of dedicated NATS subscribers which consume data from the NATS server and store them into the VM backend. Metrics are visualised in dedicated Grafana dashboards.

3 The CMS monitoring applications

To satisfy all the requirements for a comprehensive monitoring system listed in Sect. 1, we complement the monitoring services provided by the MONIT infrastructure with additional applications, leveraging the open-source technologies provided by the CMS monitoring infrastructure. The development of common tools to handle data access, processing, and visualization allows us to consolidate the resources needed to operate, maintain, and develop the infrastructure itself and the monitoring applications. Knowledge and development efforts can be shared among several CMS groups.

3.1 The CMS spider

The HTCondor infrastructure is central to distributed data processing for CMS [29]. The CMS spider [30] is a Python application that periodically queries the HTCondor infrastructure for job information and feeds it into the MONIT infrastructure. Every twelve minutes, the CMS spider queries the *schedds* - the HTCondor component responsible of managing the job queues - for a snapshot of the current queued and running jobs, and for a list of completed or

removed jobs since the last query from the HTCondor history. Data are sent to MONIT through ActiveMQ, and stored in HDFS and ES. The available metrics are used to visualise historical information about HTCondor jobs and tasks in Grafana, for both analysis and production workflows.

The CMS spider is currently being migrated to the CMS k8s infrastructure. We use Celery [31] for asynchronous execution of Python tasks, with Redis [32] as message broker and Flower [33] for monitoring of task statuses, as shown in Fig. 2. The tasks are kept in Redis queues and distributed across the workers. The CMS spider has six Celery workers by default which are replica pods. They are responsible for running three asynchronous tasks with separate queues:

- query the schedds: queries the schedds every twelve minutes, retrieves HTCondor job properties and triggers the execution of the "processing documents" task. This task also sets checkpoints - date and time of last run - after each execution;
- processing documents: converts the HTCondor job properties to JSON [34] data format and triggers the execution of the "post to ES" task;
- post to ES: sends the JSON documents containing the HTCondor job properties to MONIT through AMQ.

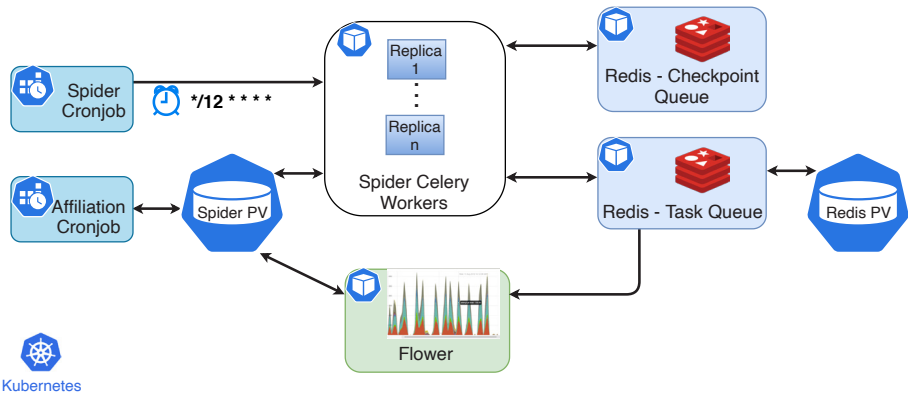

Figure 2. The CMS Spider deployment in Kubernetes.

All tasks must be executed in less than twelve minutes, i.e. the time between two sets of queries. This time constraint mainly comes from the latency of querying the HTCondor schedds, which in the current deployment may take up to several minutes. The CMS spider will be deployed in its own k8s cluster, which may allow us to increase the frequency at which we poll the schedds.

3.2 CMS Spark workflows

Workflows based on Apache Spark are used to access and process data on HDFS. Examples of such workflows include the production of popularity datasets by combining data from DBS, PhEDEx in the past and now Rucio, as well as other sources like EOS [35] or XRootD [36] data access logs. A common framework, CMSSpark [37], was developed by CMS to hide the complexity of Spark submission jobs and to abstracts access to common CMS datasets on

HDFS. CMSSpark is extensively used to produce aggregated plots which require processing metrics over a large period of time (several months or years).

3.3 Command Line tools

Command Line Interface (CLI) tools play an important role in any infrastructure. In CMS, we developed a set of CLI tools to address common use cases:

- the **monit** tool to access data sources in the CERN MONIT infrastructure via a Grafana proxy;
- the **alert** tool to manage alerts from all CMS data sources. The user queries are provided via ES JSON or InfluxQL [38] formats;
- the **annotationManager** to handle annotations on any CMS Grafana dashboard based on its tag;
- a set of tools (**ggus_parser**, **ggus_alerting**, **ssb_parser** and **ssb_alerting**) to parse content from the GGUS and SSB ticketing systems, and generate and manage related alerts.

All tools are implemented in Go to simplify the deployments across various hardware platforms and are publicly available on CVMFS [39] in the */cvmfs/cms.cern.ch/cmsmonit* area.

4 Current developments and R&D

In the near future LHC experiments will be required to cope with higher data rates due to the upgrade of the LHC to high luminosity (the HL-LHC). This will pose challenges to all computing systems, including the monitoring infrastructure. It is paramount to engage and support R&D activities to be able to sustain the experiment needs as data rates increase. The activities described below are developed as part of the Operational Intelligence [41] effort, which aims to reduce the human cost of WLCG operations by pursuing automation of repetitive tasks.

4.1 Intelligent Alert System

Alerts are critical to promptly address system failures. The sheer volume and variety of alerts produced by the large number of monitored services must be efficiently handled. We developed an intelligent layer to detect, analyze and predict abnormal system behaviors. Figure 3 shows the overall architecture of the Intelligent Alert system. It is based on a pipeline for AM where we automated the mundane process of alert bookkeeping. This allows operation teams to focus more on finding solutions for the source of alerts rather than searching, filtering and collecting the alerts. The pipeline was designed to be as generic as possible. It consists of the following steps:

- fetching the existing alerts and information from data sources external to MONIT,
- pre-processing the alerts to extract the relevant information,
- spotting anomalies in alerts (if any),
- annotating corresponding Grafana dashboards with relevant alerts,
- silencing false alerts,
- deleting resolved alerts,
- feed back the updated information into AM.

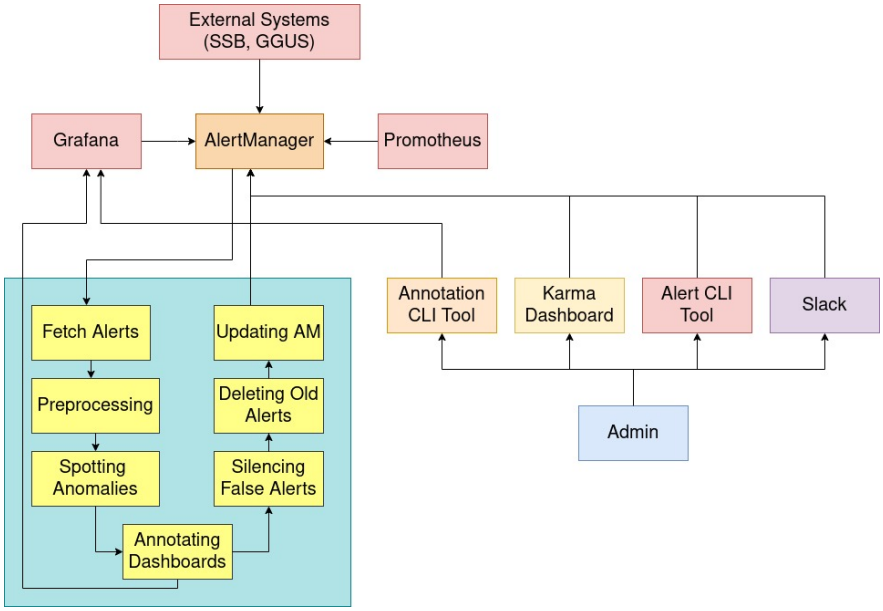

Figure 3. The intelligent alert system architecture. The steps of the pipeline discussed in Sect. 4.1 are shown in the diagram.

Two additional systems are integrated in the pipeline: the SSB, which provides information about ongoing interventions and outages in the CERN computing infrastructure, and the GGUS system which keeps track of user tickets related to WLCG (Worldwide LHC Computing Grid) sites. The information from these systems is fed into AM, and is used to add annotations of relevant alerts to existing dashboards based on specific tags. Such additions provide useful insights about when outages happen and how they affect the productivity reported by various systems in our dashboards. Alerts are currently selected and filtered based on specific lists of keywords. We plan to further develop alert selection and silencing employing machine learning (ML) techniques such as anomaly detection and clustering.

4.2 FTS log analysis

The CERN File Transfer System (FTS) is one of the most critical services for CMS distributed computing [40]. FTS is a low level protocol used to transfer data among different WLCG sites. FTS sustains a data transfer rate of 20-40 GB/s, and it transfers daily a few millions files. If a transfer fails, an error message is generated and stored in HDFS. Failed transfers are of the order of a few hundred thousand per day. Understanding and possibly fixing the cause of failed transfers is part of the duties of the experiment operation teams. Due to the large number of failed transfers, not all can be addressed. We developed a pipeline to discover failure patterns from the analysis of FTS error logs (see Fig. 4). Error messages are read in from HDFS, cleaned from meaningless parts (file paths, host names), and grouped in clusters based on the similarity of their text using the Levenshtein distance [42]. A second approach based on ML techniques such as word2vec [43] is currently under development [44]. The message patterns of each cluster are stored in ES and HDFS, together with source and destination host names

of the failing transfers. The biggest clusters and their relationship with the host names with largest numbers of failing transfers is presented in a dedicated Grafana dashboard. The clusters can be used by the operation teams to quickly identify anomalies in user activities, tackle site issues related to the backlog of data transfers, and in the future to implement automatic recovery procedures for the most common error types.

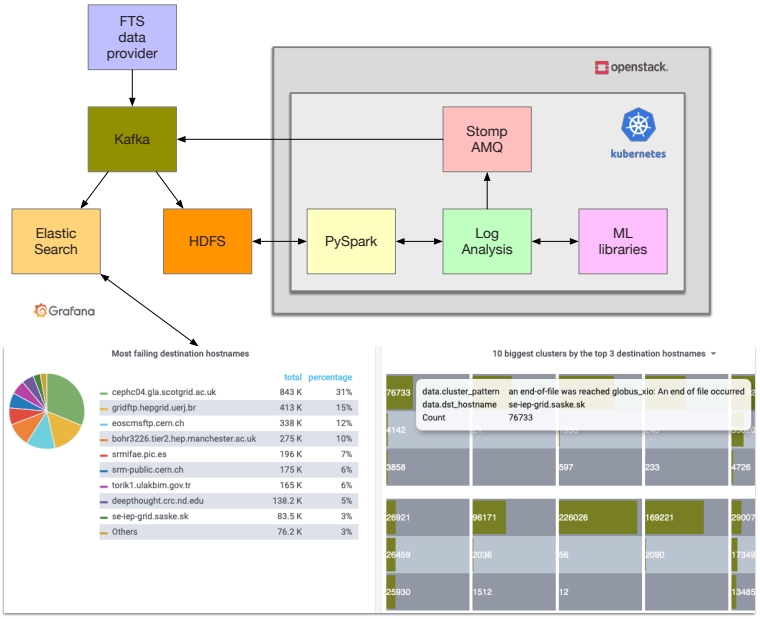

Figure 4. The FTS log analysis workflow. FTS error messages are read in from HDFS, pre-processed and grouped by a clustering algorithm based on their similarity. The results are represented as JSON records, fed back into the MONIT infrastructure, and displayed in a dedicated Grafana dashboard.

4.3 Rumble

HDFS storage plays an important role in the CMS monitoring infrastructure, since it holds large amount of historical data that cannot be stored on ES or InfluxDB due to performance reasons. Running queries on HDFS data requires domain expertise in Hadoop ecosystem components - mostly Spark APIs in Java, Python, or Scala - with a steep learning curve. Rumble [45] is an open-source querying engine which provides easy SQL-like query access to HDFS data. Rumble high-level JSONiq query language [46] maps common FLWOR (For, Let, Where, Order by, Return) expressions to Spark DataFrames and SQL. An example query can be found in Listing 1.

We deployed Rumble as a service within the CMS k8s monitoring cluster. The Docker image of Rumble consists of Spark and Hadoop packages, Rumble JAR (Java ARchive file format), and scripts which run Rumble in server mode. In the Kubernetes perspective, the deployment can be defined as sending a Rumble JAR with a spark-submit job [47] to the CERN Spark cluster Analytix from a Kubernetes pod, which requires full-fledged Spark and Kubernetes configurations. The Rumble server accepts HTTP requests through a dedicated endpoint in the CMS monitoring cluster, translates and executes queries in the Analytix cluster, and returns the query results back to the client. In order to send JSONiq queries to Rumble

```
for $doc in json-file(
"/project/monitoring/archive/wmarchive/raw/metric/2021/01/01/*"
)
where $doc."data"."meta_data"."jobstate" eq "failed"
group by $task := $doc."data"."task"
where count($doc) ge 1
return {
  "task name": $task,
  "count": count($doc)
}
```

Listing 1: An example of JSONiq that returns the total number of failed jobs and task names of tasks with at least one failed job.

server, Go and Python clients [48] are implemented. It is also possible to submit JSONiq queries from a Jupyter Notebook.

5 Summary

We presented an overview of the current CMS monitoring infrastructure and future developments. The choice of open-source technologies enabled us to build scalable and maintainable applications. The development of a common monitoring infrastructure for CMS brought several benefits: the consolidation of the resources needed to operate, maintain, and develop the infrastructure itself and the monitoring applications, the portability of monitoring solutions when using common data formats for metrics, and common visualization tools. We regularly provide training and built a community of users of our monitoring services, so that developments and knowledge can be shared among different groups. The deployment of the monitoring applications and services in a Kubernetes infrastructure provided better scalability and reduction in operational costs. We have a solid monitoring infrastructure and sustained R&D program that allows us to cope with current and future challenges at the HL-LHC, and actively participate to the WLCG Operational Intelligence effort.

References

[1] CMS Collaboration, JINST 3 S08004 (2008)
[2] I. Sfiligoi et al., proceedings of the WRI World Congress on Computer Science and Information Engineering, Vol. 2, 2428-432 (2009)
[3] D. Thain, T. Tannenbaum, and M. Livny, Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4, 323-356 (2005)
[4] T. Ivanov et al, EPJ Web Conf, 03006 (2019)
[5] Rehn, et al. Proc. CHEP06, Computing in High Energy Physics, Mumbai, India, (2006).
[6] Barisits, M., Beermann, T., Berghaus, F. et al. Comput Softw Big Sci (2019) 3: 11
[7] V. Kuznetsov et al. J. Phys.: Conf. Ser. 219 042043 (2010)
[8] M. Giffels, Y. Guo, V. Kuznetsov, N. Magini and T. Wildish, J. Phys.: Conf. Ser., Vol. 513, Issue 4 (2014)
[9] I. Bird et al., CERN-LHCC-2014-014, LCG-TDR-002 (2014)
[10] C. Ariza-Porras, V. Kuznetsov, F. Legger, Comput Softw Big Sci (2021) 5:5

[11] A. Aimar, et al., J. Phys.: Conf. Ser. 898 (2017) 092033
[12] *Kubernetes*, https://kubernetes.io/ (2021), accessed: 2021-02-18
[13] *Elasticsearch*, http://elastic.co (2021), accessed: 2021-02-18
[14] *Apache Hadoop*, http://hadoop.apache.org (2021), accessed: 2021-02-18
[15] *InfluxDB*, https://www.influxdata.com/time-series-platform/influxdb/ (2021), accessed: 2021-02-18
[16] *Kibana*, https://www.elastic.co/products/kibana (2021), accessed: 2021-02-18
[17] *Grafana*, http://grafana.org (2021), accessed: 2021-02-18
[18] *Apache Spark*, http://spark.apache.org (2021), accessed: 2021-02-18
[19] D. Piparo, et al., Future Gener. Comput. Syst. 78 (2018) 1071-1078
[20] *Apache ActiveMQ*, http://activemq.apache.org (2021), accessed: 2021-02-18
[21] *Prometheus*, https://prometheus.io/ (2021), accessed: 2021-02-18
[22] *VictoriaMetrics*, https://victoriametrics.com/ (2021), accessed: 2021-02-18
[23] *Prometheus AlertManager* https://prometheus.io/docs/alerting/alertmanager/ (2021), accessed: 2021-02-18
[24] *NATS* https://nats.io/ (2021), accessed: 2021-02-18
[25] *Prometheus proxy*, https://github.com/jacksontj/promxy (2021), accessed: 2021-02-18
[26] *Apache Sqoop*, https://sqoop.apache.org/ (2021), accessed: 2021-02-18
[27] J. Andreeva, et al., J. of Grid Comp. 8 323-339 (2010)
[28] T. Antoni, W. Buhler, H. Dres, G. Grein and M. Roth, J. Phys.: Conf. Ser. 119, 052002 (2008)
[29] J Balcas et al, J. Phys.: Conf. Ser. 898 052031
[30] *CMS Spider repository*, https://github.com/dmwm/CMSKubernetes/tree/master/kubernetes/spider (2021), accessed: 2021-02-18
[31] *Celery*, https://docs.celeryproject.org/ (2021), accessed: 2021-02-18
[32] *Redis, in-memory data structure store*, https://github.com/redis/redis (2021), accessed: 2021-02-18
[33] *Flower*, https://flower.readthedocs.io/en/latest/ (2021), accessed: 2021-02-18
[34] *JSON (JavaScript Object Notation)*, https://www.json.org (2021), accessed: 2021-02-18
[35] A. J. Peters, et al., J. Phys.: Conf. Ser. (2015) 664 042042
[36] *XRootD project page* http://www.xrootd.org/ (2021), accessed: 2021-02-18
[37] *CMSSpark framework*, https://github.com/dmwm/CMSSpark (2021), accessed: 2021-02-18
[38] *Influx Query Language*, https://docs.influxdata.com/influxdb/v1.8/query_language/ (2021), accessed: 2021-02-18
[39] P. Buncic, et al., J. Phys. Conf. Ser., 219 042003 (2010)
[40] E. Karavakis et al., EPJ Web Conf. 245 04016 (2020)
[41] A. Di Girolamo et al., EPJ Web Conf. 245, 03017 (2020)
[42] *Levenshtein distance* https://www.wikiwand.com/en/Levenshtein_distance (2021), accessed: 2021-02-18
[43] *Word2vec algorithm* https://www.wikiwand.com/en/Word2vec (2021), accessed: 2021-02-18
[44] A. Di Girolamo et al., Operational Intelligence, these proceedings
[45] *Rumble*, https://rumble.readthedocs.io/en/latest/ (2021), accessed: 2021-02-18
[46] *JSONiq*, https://www.jsoniq.org/ (2021), accessed: 2021-02-18
[47] *Spark Documentation*, https://spark.apache.org/docs/latest/submitting-applications.html (2021), accessed: 2021-02-18

[48] *Rumble query examples*, https://github.com/dmwm/CMSMonitoring/tree/master/doc/Rumble (2021), accessed: 2021-02-18