Cluster Computing (2021) 24:3085–3099
https://doi.org/10.1007/s10586-021-03325-0

Migration of CMSWEB cluster at CERN to Kubernetes: a comprehensive study

Muhammad Imran[1,2] · Valentin Kuznetsov[3] · Katarzyna Maria Dziedziniewicz-Wojcik[2] · Andreas Pfeiffer[2] · Panos Paparrigopoulos[2] · Spyridon Trigazis[2] · Tommaso Tedeschi[4,5] · Diego Ciangottini[5]

Received: 22 January 2021 / Revised: 28 April 2021 / Accepted: 31 May 2021 / Published online: 9 June 2021

Abstract
The Compact Muon Solenoid (CMS) experiment heavily relies on the CMSWEB cluster to host critical services for its operational needs. The cluster is deployed on virtual machines (VMs) from the CERN OpenStack cloud and is manually maintained by operators and developers. The release cycle is composed of several steps, from building RPMs to their deployment, validation, and integration tests. To enhance the sustainability of the CMSWEB cluster, CMS decided to migrate its cluster to a containerized solution based on Docker and orchestrated with Kubernetes (K8s). This allows us to significantly speed up the release upgrade cycle, follow the end-to-end deployment procedure, and reduce operational cost. In this paper, we give an overview of the CMSWEB VM cluster and the issues we discovered during this migration. We discuss the architecture and the implementation strategy in the CMSWEB Kubernetes cluster. Even though Kubernetes provides horizontal pod autoscaling based on CPUs and memory, in this paper, we provide details of horizontal pod autoscaling based on the custom metrics of CMSWEB services. We also discuss automated deployment procedure based on the best practices of continuous integration/continuous deployment (CI/CD) workflows. We present performance analysis between Kubernetes and VM based CMSWEB deployments. Finally, we describe various issues found during the implementation in Kubernetes and report on lessons learned during the migration process.

Keywords Cluster computing · Kubernetes · Container · CMSWEB · LHC · CMS · Docker

1 Introduction

The CMS [1] is a general-purpose detector at the Large Hadron Collider (LHC) at CERN, Geneva, Switzerland. It has a broad physics program ranging from studying the Standard Model (including the Higgs boson) to searching for extra dimensions and particles that could make up dark matter. The CMS experiment is one of the largest international scientific collaborations in history, involving 5000 particle physicists, engineers, technicians, students, and support staff from 200 institutes in 50 countries [2].

The CMS experiment runs hundreds of thousands of jobs on its distributed computing system to simulate, reconstruct, and analyse the data taken during collision runs. A dedicated cluster ("CMSWEB") is used to host essential CMS central services that handle the CMS data management, data discovery, and various data bookkeeping tasks. The cluster is based on virtual machines (VMs) deployed at the CERN OpenStack cloud infrastructure. Each service is managed by its own development team. Due to the complexity of the heterogeneous environment and different schedules of development teams, only monthly release cycles can be afforded. Each upgrade cycle includes the build of RPMs from source code (including all dependency chains), the cross-validation of all software components, and the validation of the correct interactions of all services. This typically requires a lot of communication between development teams and operators, as well as coordination of various efforts in a coherent manner. By policy, production releases are only deployed once a month, so developers may have to wait for up to four weeks before the new version of their services is deployed into production.

Extended author information available on the last page of the article

To enhance the sustainability of CMSWEB, CMS decided to migrate its infrastructure to a containerized solution based on Docker [3] orchestrated with Kubernetes (K8s) [4]. With this approach, we can accomplish end-to-end deployment procedures for our services and allow developers to quickly deploy new versions of their services. This significantly reduces the release upgrade cycle, unifies deployment procedures, and reduces operational cost.

Recently, we have performed the migration of the CMSWEB cluster to the Kubernetes [5]. In this paper, we give an overview of the CMSWEB services and VM cluster in Sect. 2. In Sect. 3, we present the proposed architecture of CMSWEB in Kubernetes and its implementation strategy. Section 4 presents the proposed mechanism of horizontal pod auto-scaling based on the custom metrics of CMSWEB services. In Sect. 5, we discuss automatic service deployment based on CI/CD workflows. Section 6 presents the performance analysis of current and proposed architecture. We discuss our plans for the future in Sect. 7. Finally, we conclude in Sect. 8.

2 Architecture of CMSWEB

The CMSWEB cluster comprises dozens of services maintained by CMSWEB operators. The deployment framework (mostly written in bash scripts) provides the ability to deploy individual services in a consistent manner. Each service deployment relies on *deploy* and *manage* scripts, while the entire deployment procedure is abstracted to deal with many services. The CMSWEB cluster allows:

- independent development and evolution of underlying services,
- simplified integration and regression testing when rolling out new service versions,
- building external services that integrate information from several sources in a clean manner.

Each service in CMSWEB has its own development group that upgrades the relevant service in the monthly release cycle. The CMSWEB development life cycle includes all the hosted services that are actively developed. The CMSWEB team is not responsible for maintaining individual services but only provides the web infrastructure and manages service deployment.

The architecture of the CMSWEB VM cluster is shown in Fig. 1. It has two layers of services, i.e., frontend and backend services. The frontend service is based on Apache which performs authentication using X509 certificates and redirects requests to the requested backend service. The individual backend services perform their relevant tasks. The frontends follow redirect rules to forward the requests

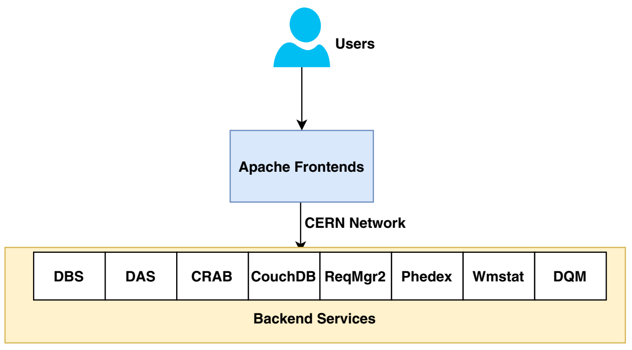


Fig. 1 The CMSWEB cluster consists of two layers, i.e. the frontend and backend services. The Frontend service runs an Apache server that performs certificates-based authentication and redirects requests to backend services

to the relevant VM node running the backend service. The backend services only accept requests coming from the frontend service and perform authorization based on provided CMS HTTP headers (these headers are constructed by CMSWEB frontends). We provide the short description of all CMSWEB services below.

2.1 CMSWEB services

The current list of CMS services deployed in CMSWEB cluster is given below. Please note, that over lifetime of the CMSWEB cluster certain services may come and go, and here we present only list of currently available services.

- *ACDCServer* A database that contains documents for failed jobs in production.
- *CouchDB* A database that is used to store various documents related to workflow management core services [6].
- *CRAB3* An analysis submission utility that is used to submit CMSSW jobs to distributed computing resources [7].
- *Crabcache* A service used to cache the job files that are submitted by users for 1 month [8].
- *DAS* A CMS service that unifies a variety of CMS data-services into a common layer used by CMS physicists and production tools to look up the CMS metadata [9].
- *DBS* A CMS service providing event data catalog for Monte Carlo and recorded data of the CMS experiment [10].
- *DBSMigration* A CMS service used to migrate data between DBS instances. [10].
- *DQMGUI* A web based user interface for browsing CMS Data Quality Monitoring Information [11].
- *Exitcodes* A web based service returning a list of known CMS exit codes from various sub-systems.
- *Exporters* A service that exposes various metrics related to different services for monitoring purposes.

- *Frontend* The Apache server used in CMSWEB for authentication and authorization [12].
- *MongoDB* A backend database for the DAS service [13].
- *PhEDEx Web Services* A web service that provides monitoring for PhEDEx CMS Data transfers [14].
- *Request Manager 2* A CMS service to create, store and manage central production workflow descriptions and their configuration [15].
- *Request Manager 2 MS* A CMS service containing a set of small services responsible for some features of the overall CMS Workload Management system [15].
- *T0_WMStats* A service that monitors production requests on Tier-0 CMS management system.
- *T0wmadatasvc* A REST service used by Tier-0 CMS management system.
- *WorkQueue* A service to manage Request Manager workflows [15].
- *WMStats* A service to monitor central production requests [16].

2.2 CMSWEB environments

There are currently three environments available for CMSWEB services: development, pre-production, and production environments. The *development* environment, which is used for *testing*, has only a single VM server that hosts all CMSWEB frontend and backend services, such that both can co-exists on a single node. Both CMSWEB operators and developers can use this environment to deploy their services in a single VM. The *pre-production* deployment relies on 2 VMs for frontend and 2 VMs for backend services.[1] Pre-production nodes are used to deploy monthly releases and perform cross-validation and integration tests among all CMS services. The CMSWEB operator deploys all services in the pre-production environment and developers test their services after each iteration of the release. After this cycle is finished, the final stable release is deployed to the production nodes. The *production* environment has 4 frontend VMs and 17 VMs for backend services.[2] The specification of CMSWEB nodes is available in Sect. 2.3. Further detail about these environments and services hosted on these VMs is available at [17].

[1] The 2×2 environment for pre-production was sufficient enough to deploy and run all CMSWEB services and perform integration tests.

[2] The 4×17 combination for the production environment is selected based on the requirement of services to manage production workload.

2.3 Specifications of servers in VM clusters

The specification of VM servers in both preprod and production VM clusters is shown in Table 1. This table shows the cluster nodes, the total number of nodes, RAM, CPU and Disk of each node.

3 Deployment of CMSWEB cluster to Kubernetes

While the existing VM based deployment procedure serves the deployment goals quite well, it requires a lot of interactions between CMSWEB operators and developers. Because of this load, upgrades of the CMSWEB production nodes could only be done once a month. To reduce this latency and release control of individual services to developers, it was decided to migrate the CMSWEB cluster to Kubernetes infrastructure. For that, we evaluated the existing Kubernetes infrastructure at CERN which is based on the OpenStack platform.

We consider a two cluster model for migration of the CMSWEB cluster to Kubernetes as shown in Fig. 2. It has the two components:

- frontend cluster (cluster A on a diagram),
- backend cluster (cluster B on a diagram).

The frontend cluster contains the CMSWEB Apache frontend behind the Nginx ingress controller (server). The backend cluster contains all CMSWEB backend services behind its ingress controller. The frontend cluster ingress controller provides TLS pass-through capabilities to pass the client's requests (with certificates) to the Apache frontend. The Apache frontend performs CMSWEB authentication and redirects the request to the backend cluster. On the backend cluster, the ingress controller has basic redirect rules to the appropriate services and only allows requests from the frontend cluster.

The reason for selecting two clusters is related to the structure of Kubernetes, i.e., the mapping of services (ingress = > services). Using Apache as an authentication layer requires keeping redirect rules from Apache to other nodes. As the Kubernetes host network does not allow for replicas, the 2 cluster architecture was needed. Plus, it allows independent maintenance of individual clusters.

To host services in Kubernetes, we need container images of all services, which are then deployed in Kubernetes. The Docker images are created using RPMs of the current VM cluster and are kept in a central repository that is available at [18]. The directory structure of this repository is shown in Fig. 3, where each service has its own directory which contains 4 files: *Dockerfile, install.sh, run.sh* and *monitor.sh*. The *Dockerfile* contains the

Table 1 Specifications of servers in VM clusters

Cluster	# Nodes	RAM	CPU	Disk
Preprod Frontend	2	14.6	8	80
Preprod Backend	1	29.3	8	80
	1	14.6	8	80
Production Frontend	4	14.6	8	80
Production Backend	10	29.3	8	80
	5	58.6	16	160
	2	58.6	32	160

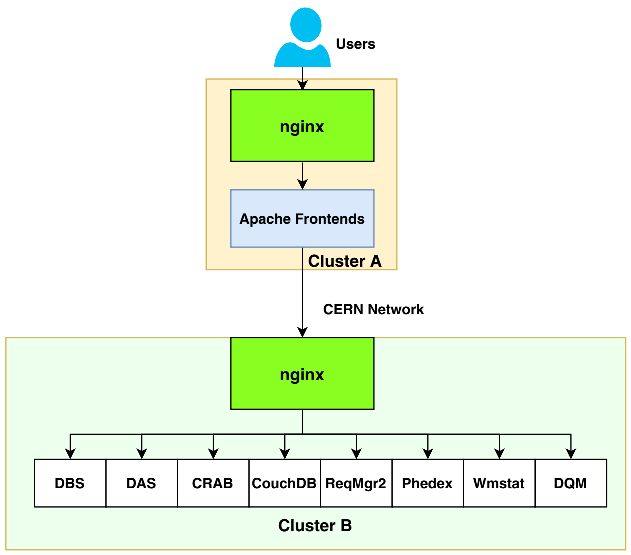


Fig. 2 The CMSWEB Kubernetes architecture has two clusters: the frontend cluster (Cluster A) and the backend cluster (Cluster B). The frontend cluster contains CMSWEB Apache frontend server behind Nginx ingress controller (server). The backend cluster contains all CMSWEB back-end services behind its own Nginx ingress controller. The ingress controller only provides routing functionality and does not perform TLS termination

specification of building a service-specific image from the base cmssw/cmsweb image. The *install.sh* file contains instructions to install the RPMs for each service in the CMSWEB cluster. The *run.sh* script instructs how to run services and the *monitor.sh* script controls how to run monitoring tools. Even though our images were quite large, this infrastructure allowed us to keep consistent builds between VM and Kubernetes clusters as well as to ensure a common OS layer including all dependency chains for all pods running within the Kubernetes cluster.

Similarly to the *docker* area, there is also a *kubernetes* area that contains service manifest files used in the Kubernetes deployment procedure. As shown in Fig. 4 the central *cmsweb* directory in the repository contains sub-directories for all CMSWEB namespaces [19]. Each namespace directory contains service manifest files for

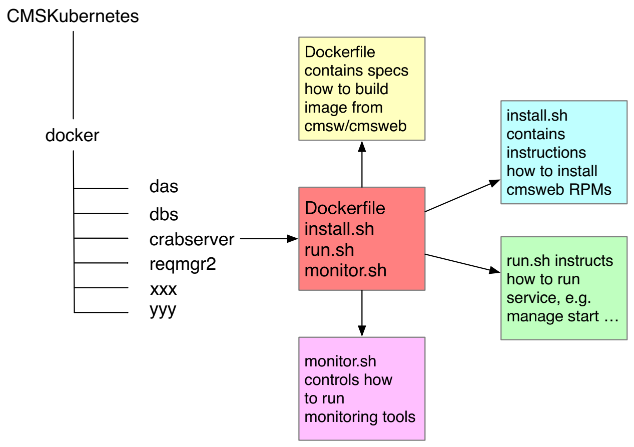

CMSKubernetes
docker
das
dbs
crabserver
reqmgr2
xxx
yyy
Dockerfile install.sh run.sh monitor.sh
Dockerfile contains specs how to build image from cmsw/cmsweb
install.sh contains instructions how to install cmsweb RPMs
run.sh instructs how to run service, e.g. manage start ...
monitor.sh controls how to run monitoring tools

Fig. 3 The CMSWEB Docker repository is structured based on the services, i.e. each service has 4 files located in their own directory: the *Dockerfile, install.sh, run.sh* and *monitor.sh* scripts that are responsible for installation, service management, and monitoring, respectively

each service, the service manifest file is represented in *YAML* data-format and contains the deployment instructions of the service container in the Kubernetes cluster.

Figure 5 shows the deployment cycle of individual services in the Kubernetes cluster. Starting from the users updating a spec file in the *cmsdist* repository [20], new RPMs are then generated. Next, new Docker images are built based on the new RPMs, and these images are uploaded to the Docker Hub repository for CMSSW [21]. Finally, the service manifest files are adjusted to deploy those images in the Kubernetes cluster. This cycle is repeated for all new releases and CMSWEB upgrades. We want to emphasize that existing redundancy, e.g. creation of RPMs followed by image builds, is only required during the migration phase from the VM cluster to Kubernetes and present only due to our legacy deployment procedure. Currently, several services already completely skip the RPM-based procedure and fully automate their deployment via the new CI/CD procedure discussed in Sect. 5.

3.1 Namespaces and services

Kubernetes supports namespaces to allow service separation within a cluster. In the Kubernetes cluster, we combined services into namespaces according to the developers' group. For example, we have 5 services for DBS and we use *dbs* namespace for all DBS services. Similarly in DMWM, we have several services that belong to a single group and we use *dmwm* namespace for that. In short, we use various namespaces that have various services under their umbrella. These namespaces are *confdb, couchdb, crab, das, dbs, dmwm, dqm, phedex*, and *tzero*. However, in the frontend cluster, we only use the *default*

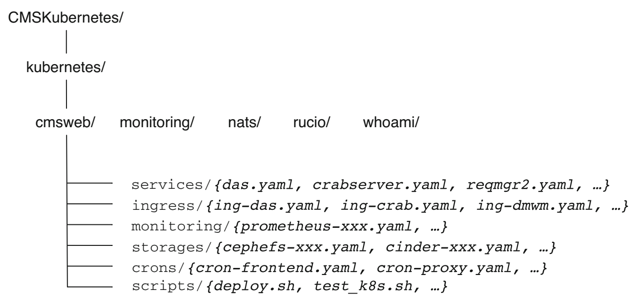

CMSKubernetes/
kubernetes/
cmsweb/
monitoring/
nats/
rucio/
whoami/
services/{das.yaml, crabserver.yaml, reqmgr2.yaml, …}
ingress/{ing-das.yaml, ing-crab.yaml, ing-dmwm.yaml, …}
monitoring/{prometheus-xxx.yaml, …}
storages/{cephefs-xxx.yaml, cinder-xxx.yaml, …}
crons/{cron-frontend.yaml, cron-proxy.yaml, …}
scripts/{deploy.sh, test_k8s.sh, …}

Fig. 4 The CMSWEB Kubernetes repository contains sub-directories for all CMSWEB namespaces. In each namespace directory, there are service manifest files for each service that specify the deployment instructions of the container image in the Kubernetes cluster

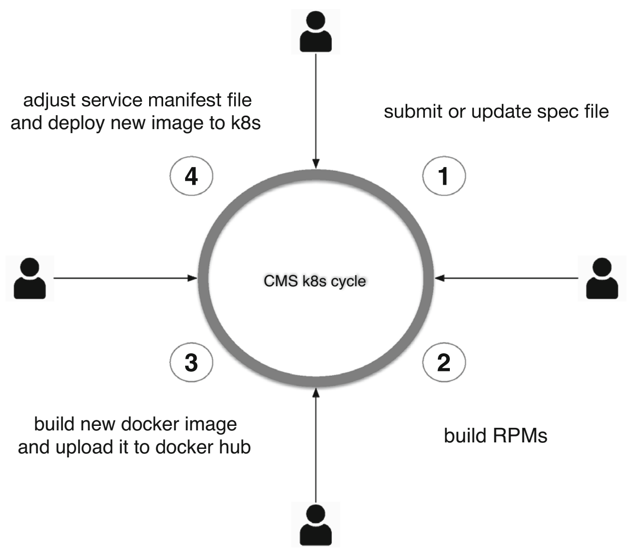

adjust service manifest file and deploy new image to k8s
submit or update spec file
4
1
CMS k8s cycle
3
2
build new docker image and upload it to docker hub
build RPMs

Fig. 5 The CMSWEB Kubernetes cycle representing the mechanism of how services are deployed in CMSWEB Kubernetes cluster

namespace for frontend service, which is available by default in every Kubernetes cluster.

3.2 Summary of resource usage

Table 2 shows the resources which are currently being used by the three environments of CMSWEB VM clusters and the new Kubernetes preprod/prod clusters. These resources are given in terms of the total number of VMs, total RAM (GB), total CPUs, and the total size of storage volumes (TB) attached to the nodes. The new clusters were created according to the service requirement of the VM production cluster. The resources were determined via the monitoring profile of each service and its average usage within the VM cluster. Autoscalers in the proposed Kubernetes cluster dynamically allocate service resources and provide efficient utilization of resources. In the K8s cluster, there are some services/daemonsets which are provided by CERN IT by default with every cluster. The examples of these are Nginx-ingress-controller, metrics-server, csi-cephfsplugin-provisioner, calico-node and many others in the kube-system namespace. All these services need some resources in the form of CPUs/memory. Therefore, additional resources are required for them in K8s cluster which was not the case in the VM cluster.

3.3 Resource allocation of CMSWEB services

The resource allocation for CMSWEB services in the Kubernetes production cluster is shown in Fig. 6. The pie-charts show the total number and the percentage of the replicas of the services, the maximum CPUs allocated to various services, and the maximum memory allocated to various services. For instance, currently, DBS pods are the most resource-hungry services.

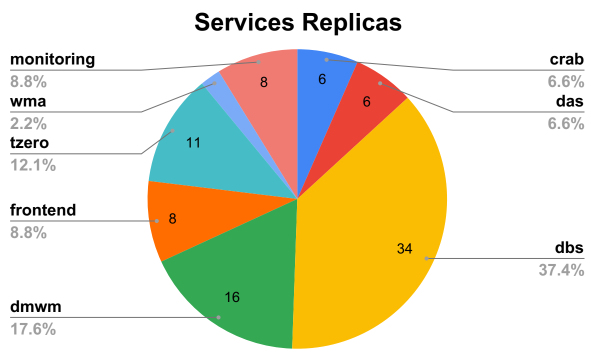
We selected the values for resources of CMSWEB services based on the actual usage of the resources in the VM production cluster. The monitoring infrastructure at CERN IT provides these statistics [22]. For example, the CMS frontend service [23] received on average 2000 requests/s. Therefore, we used 4 frontend nodes in the VM production cluster evenly spreading the load and working in a regime of 500 requests/s per node. Similarly, the same number of replicas were selected for the Kubernetes frontend service. Likewise, the minimum and maximum values for CPUs and RAM for a given service are based on the utilization of this service as obtained in the VM cluster.

If the resource requirement is greater than the maximum value assigned to it, then the Kubernetes infrastructure will kill the container and start a new container.

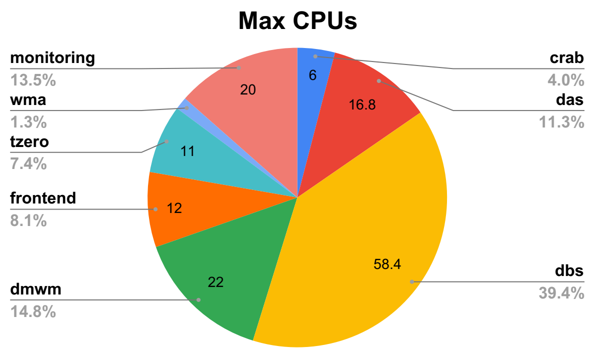
We consider Cinder and Ceph File System (CephFS) storage in the CMSWEB Kubernetes cluster. Cinder is block storage for OpenStack. It virtualizes the management of block storage devices and provides end-users with a self-

Table 2 Summary of resource usage for various CMSWEB cluster environments

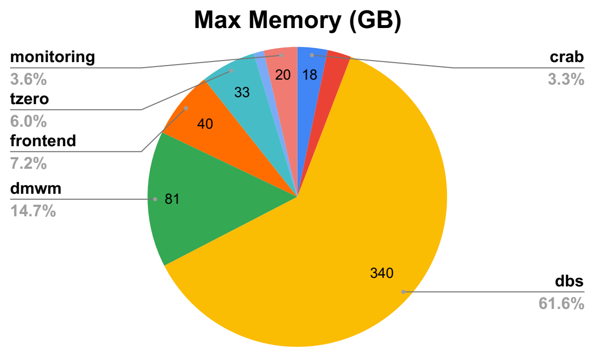
Cluster environments	Total nodes	Total RAM (GB)	Total CPUs	Total volume (TB)
VM Dev	1	7.3	4	0.15
VM Preprod	4	73.1	32	2.5
VM Production	21	794.4	256	16
Kubernetes Preprod	8	240	128	2
Kubernetes Prod	18	540	288	2



(a) The total number of replicas of various services.t



(b) The total maximum CPUs allocated to various services.



(c) The total maximum memory (GB) allocated to various services.

Fig. 6 These plots show the number and percentage of replicas, maximum CPUs, and maximum memory of CMSWEB services in their namespaces. The DBS services consume a major portion of the resources

service API to request and consume those resources without requiring any knowledge of where their storage or type. It can be attached or detached to or from a VM without losing the persistence of data [24]. The CephFS is a POSIX-compliant file system built on top of Ceph's distributed object store, Reliable Autonomic Distributed Object Store (RADOS). CephFS endeavors to provide a state-of-the-art, multi-use, highly available, and performant file store for a variety of applications, including traditional use-cases like shared home directories, High-Performance Computing (HPC) scratch space, and distributed workflow shared storage [25]. We used CephFS for logs and cache data, and these are cross mounted in the Kubernetes pods and a VM.[3] The users can access these logs in real-time directly from the VM. We consider Cinder storage for storing data for services that require a lot of disk space, e.g. database files or caching areas. This space is available to us in the form of Cinder storage which can be mounted to a single Kubernetes node.

3.4 Cluster monitoring

We organize cluster monitoring using three open-source solutions: the Prometheus [26] to scrape the metrics from our services, the Filebeat daemons to read local log files and push their content to Logstash service, and the Logstash service to preprocess logs and push their content to the CERN MONIT infrastructure. The monitoring architecture is presented in Fig. 7, and it is part of a more complex CMS Monitoring infrastructure [27]. The Prometheus servers are integrated into each (frontend and backend) CMSWEB cluster. The collected metrics are accessible via the Prometheus data-source in the CERN MONIT infrastructure, and various metrics are visualized as different dashboards. The service logs are pushed into Elasticsearch instance and later used to build dashboards of various users' access patterns, e.g. for monitoring top-10 IPs, APIs, queries, etc. This infrastructure requires minimal effort from the operator and was proven to be reliable for our needs.

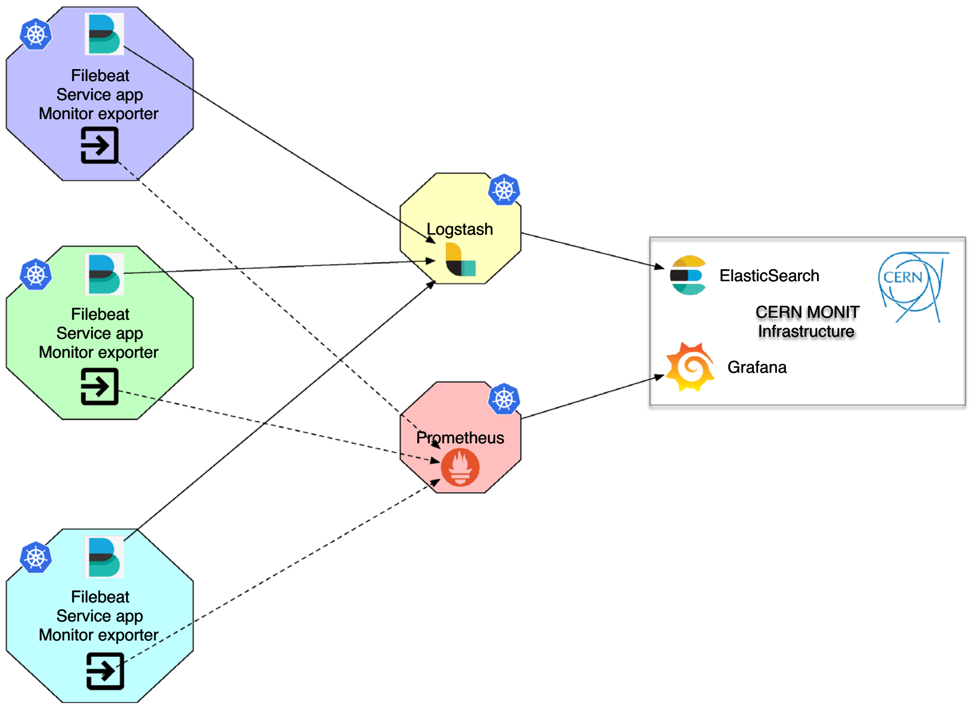
4 Horizontal pod autoscaling via Prometheus

As of today, Kubernetes natively exports only RAM or CPU-based metrics. In order to scale applications on the basis of custom metrics, the usage of third-party components becomes necessary: in fact, Kubernetes' Horizontal Pod Autoscaler (HPA) [28, 29] makes it possible to scale deployments according to any metrics as long as they are exposed through the Custom Metrics API (or the External Metrics API [28]). The full workflow is depicted in Fig. 8.

First, a Prometheus exporter that exposes metrics from single pods is needed. A Prometheus server will then regularly collect all metrics from various exporters in the form of time series. To make these values retrievable by HPA, a

[3] Each service in Kubernetes is represented by a pod. Within a pod, it writes log files to CephFS storage. Finally, we mount this area outside of K8s to allow unified access to all service logs.

Fig. 7 The CMSWEB Kubernetes monitoring architecture is based on the Prometheus server for collecting live service metrics and Logstash for offloading service logs to the CERN MONIT infrastructure



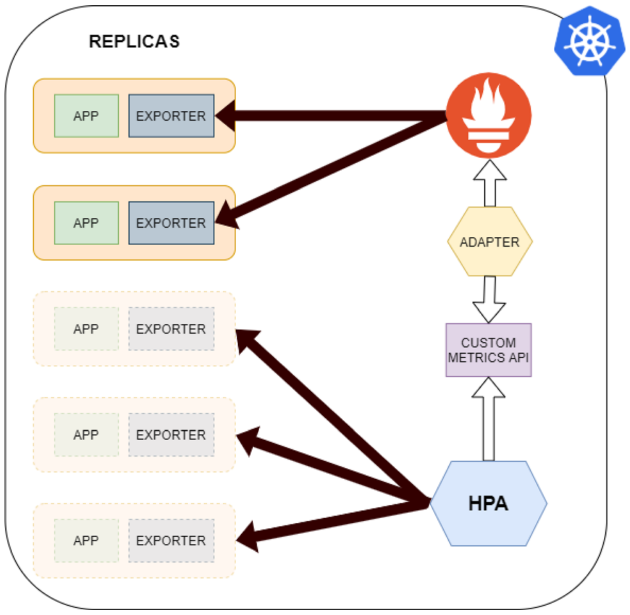


Fig. 8 Horizontal pod autoscaling based on custom metrics collected by Prometheus

Prometheus Adapter [30] application selects certain time series from the Prometheus Server and exposes them through Custom Metrics API. Finally, HPA resources are used to determine which deployments need to be scaled and what metric values should be used to scale them accordingly. A detailed explanation of each one of the steps above will be discussed in the next section.

4.1 Workflow details

The first step is to deploy an application (generally only a single replica is sufficient) along with its specific Prometheus exporter. Exporters are application-specific servers that export internal metrics from the application of interest, converting them to a predefined format. These are then served at the `/metrics` path. General rules and exposition formats can be found at [31, 32] while a non-comprehensive list of existing Prometheus exporters can be found at [33, 34]. The monitoring task is then performed by the Prometheus server collecting all the metrics from various exporters in the form of time series, regularly making HTTP calls to each `/metrics` target which may be statically configured or dynamically discovered. Then, the Prometheus Adapter [30] application is deployed. It has a key role, acting as the link between Prometheus and Kubernetes. In fact, it queries the Prometheus server looking for certain time series that are manipulated and exposed through the Custom Metrics API. The Adapter makes use of the following set of rules:

– `Discovery`, which tells the Adapter how to find metrics for this rule.
– `Association`, which tells the Adapter how to associate Kubernetes resources and metrics.
– `Naming`, which tells the Adapter how to expose metrics in the custom metrics API.
– `Querying`, which tells the Adapter how to query Prometheus server.

Finally, one can turn on autoscaling based on arbitrary metrics by deploying an HPA (making use of the autoscaling/v2beta2 API version). The HPA is basically a control loop that periodically queries a metrics API. In this case, it searches for a custom metric when regularly querying the Custom Metrics API. When the queried metric values are found above thresholds (set by the user), HPA scales up the targeted deployment creating additional replicas (up to a maximum) until the metric values go below the previously mentioned threshold. Then a cool-down policy is used to decide when to scale down after the scaling up is over. Maximum and minimum number of replicas are also set by the user, and parameters regarding scale-down or scale-up policies can be specified.

4.2 Benchmark tests

The full workflow has been tested with different applications (deployments) and thus different exporters [35]. A benchmark test was performed to explore scaling-up times as a function of container image size. This gives us a hint about how image size affects scaling times. This is important since the effectiveness of autoscaling depends on how much time it takes to increase the number of replicas. In systems where the load (and hence the metric value) rapidly increases, having a slow scaling could be critical. We used three Docker images that contained the same service (an httpgo server and a process exporter) and only a different underlying OS stack:

– `ttedesch/httpgo_and_exporter-small (19.17 MB)`,
– `ttedesch/httpgo_and_exporter-medium (325.96 MB)`,
– `ttedesch/httpgo_and_exporter-big (2.73 GB)`.

For this benchmark test, we used a Kubernetes cluster with 1 master (2 CPUs, 4 GB) and 8 slaves (4 CPUs, 8 GB) deployed at the ReCaS-Bari data center [36]. We used anti-affinity rules and a clean-up mechanism to be sure that each container was deployed in a different node in which the image was not already present: in this way, the autoscaling time we measured (i.e. the time it takes to Kubernetes to scale from 1 to 8 replicas of the httpgo deployment after HPA is triggered) included both pulling and deploying times. Results averaged across 5 autoscaling experiments are shown in Fig. 9.

As foreseen, the bigger the image size, the more time it takes to scale up the deployment, since downloading and creating the corresponding pod in each node require much heavier actions. In an environment characterized by very short load peaks, autoscaling can be effective only if the process of creation of replicas is fast enough. Hence the size of the image could play an important role in scenarios where the speed of the response to heavy load is critical.

5 CI/CD workflows

To automate service deployment, we rely on best practices of CI/CD workflows. Originally, we applied the following procedure to all services. All our manifest files resides in the CMS Kubernetes/docker area [19]. As was discussed in Sect. 3 each service docker area contains a corresponding Dockerfile, and services of auxiliary scripts. All images are built centrally, pushed to the Docker hub repository, and then used in K8s deployment. Even though this procedure works perfectly it still requires manual steps, like tagging code in the repository, building the corresponding image, and pushing it into the Docker hub repository. Recently, we automated all of these steps via a GitHub Action [37] workflow. Each workflow manifest file contains the series of steps associated with a given package. For example, it performs compilation, testing, and building the service executable as well as the service image. Then, the image can be uploaded to Docker hub, and, optionally, the HTTP POST request can be placed to the imagebot service [38] (deployed within our Kubernetes cluster) to upgrade the service image within specific cluster namespace.[4] Currently, we apply this procedure only in the testbed cluster to allow developers to quickly test their service with every git tag push action. For the production cluster we still rely on the manual upgrade procedure. This is because in the production cluster, we deploy service only when it is ready after being tested in the testbed cluster after various iterations of service updates using CI/CD.

6 Performance analysis

We used the *hey* tool [39] to evaluate the performance of the VM based clusters (both testbed and production) and the new testbed clusters in Kubernetes. The initial Kubernetes cluster used version 1.15 and showed a severe network degradation caused by a faulty network driver (“flannel”). This issue was fixed in Kubernetes version 1.17, which uses the calico network driver. In the following comparison, we label the clusters using Kubernetes version 1.15 as “old cluster”, and the ones using version 1.17 as “new cluster”.

To test the performance, three scenarios were considered: in the first scenario, a configuration with $n = 10$ and

[4] Each request is authenticated via a limited lifetime token, and provides all necessary meta-data information about the new deployment such as image tag, corresponding namespace, etc.

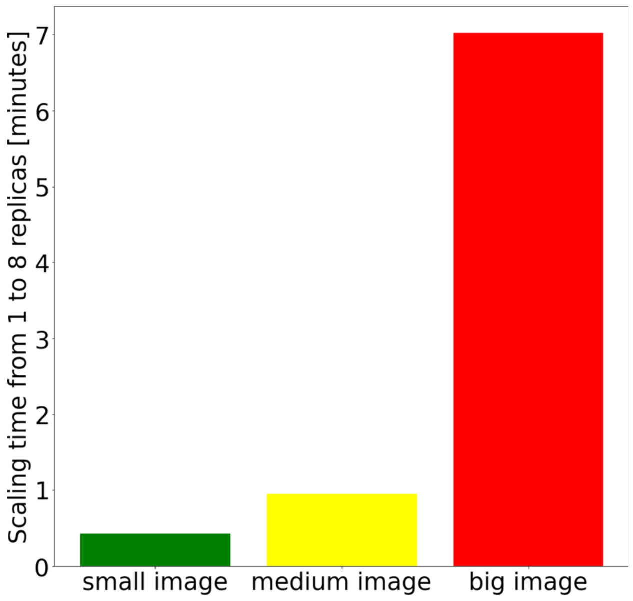


Fig. 9 Average autoscaling time for small (`19.17 MB`), medium (`325.96 MB`) and big (`2.73 GB`) image from 1 to 8 replicas

$c = 5$ was chosen, where n is the number of requests to run and c is the number of workers to run concurrently. Initially, this small load was sufficient to investigate the network degradation issue. In latter scenarios, a more realistic benchmark with $n = 1000$ requests and $c = \{100, 200\}$ concurrent clients was used, which represents a typical load in our daily operations.

The performance results of this first scenario are shown in Fig. 10: various services of CMSWEB are shown on the x-axis while the resulting requests/s are shown on the y-axis. A comparative analysis of the VM production cluster, VM testbed cluster, old Kubernetes cluster, and new Kubernetes cluster was performed. The old and new Kubernetes clusters were studied with different numbers of replicas of the frontend (FE) i.e. (4, 6, 8) to see the impact of replicas on the overall performance. It can be seen that the new Kubernetes cluster performs much better as compared to the production, testbed, and old Kubernetes cluster. The main purpose of this scenario is to demonstrate the network problem in the "old k8s cluster". This small load was sufficient to investigate the network degradation issue, and variations in number of FE replicas have no effect in this case. The increase in number of FE replicas matters when the load is very high and this is demonstrated in the second/third scenario as in Fig. 11a, b.

For the second scenario, only the frontends at a very heavy load were considered. We used n and c i.e. $n = 1000$, and $c = \{100, 200\}$ options for the *hey* tool; the results are shown in Fig. 11. We performed benchmark tests with and without reusing TCP connections, see Fig. 11a and b, respectively. As can be seen from these plots that the new Kubernetes cluster outperforms the VM cluster benchmarks when we increase the number of FE replicas. Based on these results we can find the optimal configuration of frontends to be used in the production cluster. It can also be noted that with an increase in the number of concurrent clients, the Kubernetes setup performs better than the VM one as the Kubernetes efficiently utilizes application resources and performs load balancing. The impact of reusing and not using existing TCP connections can also be seen here. By using existing TCP connections, we get better performance as compared to the scenario when the existing TCP connections are not used. This is because of the additional latency that is incurred to establish a TCP connection when the cache is not used.

During stress testing, we noticed a few services with poor performance results. To investigate this issue, we explored various parameters of Nginx ingress controller settings. The Nginx uses two types of connections, i.e. connections with the clients and connections with the upstream server. The Nginx multiplexes requests from various clients to the upstream server. However, since we use Apache in addition to Nginx, which handles the connections with servers as well, the Nginx upstream connections caused performance degradation. This issue was resolved after disabling the connections to the upstream server and use Nginx as the proxy only. The relevant parameter for this is upstream-keepalive-connections (UKC) in the Nginx setting. This issue was observed when we run stress tests in K8s cluster using Nginx UKC! = 0 and UKC = 0 values and compared these results with the production VM cluster, see Fig. 12. In this test, we used the same number of K8s instances and VM production instances for various services. We found that by disabling Nginx upstream connections with UKC= 0, the performance issue was resolved and the K8s cluster outperformed the VM one in this case. Due to old architecture some of the CMSWEB services do not scale well when they are deployed in the K8s infrastructure. However, service developers are working on this issue and they are improving their services. For example, there are undergoing migration from Python2 to Python3, re-writing some services from Python in Go language to improve their scalability, etc. As we run services in the K8s infrastructure now, the newer versions of the services are expected to perform well in new infrastructure.

6.1 Issues faced during migration

During this migration process various issues were encountered, which are categorized into two categories:

- Infrastructure issues,
- Service issues.

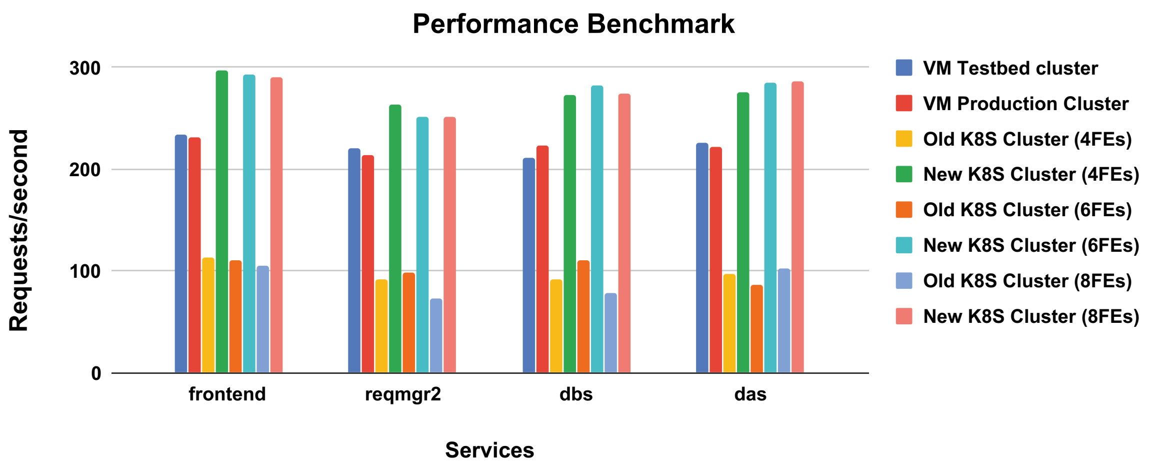
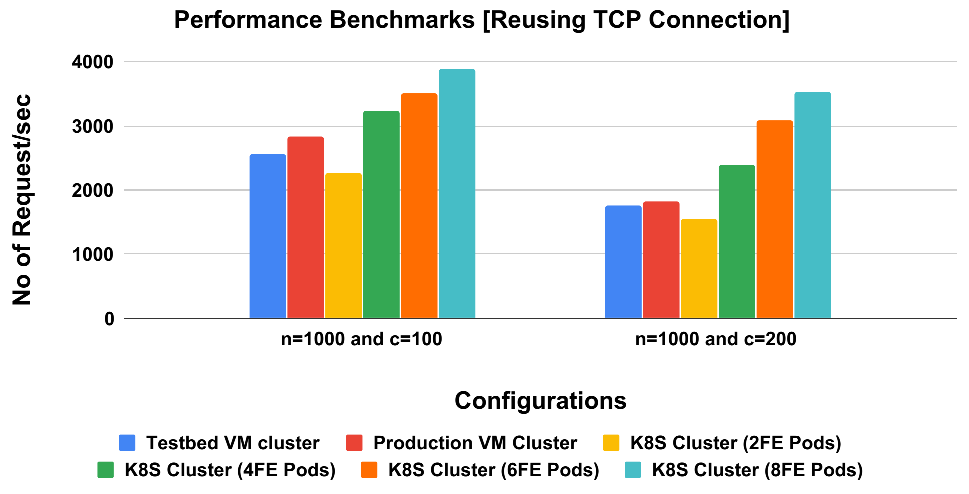
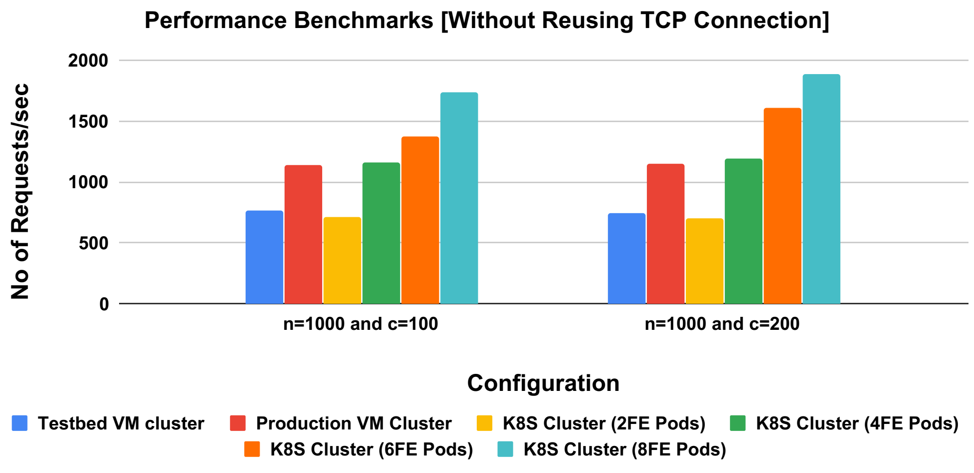


Fig. 10 The performance benchmark results in the form of requests/s (y-axis) of some of the commonly used CMSWEB services (x-axis) for various clusters (i.e. VM production cluster, VM testbed cluster, old Kubernetes cluster, and new Kubernetes cluster using a different number of replicas). A total of 100 tests were performed for each configuration and results show average values from these tests

Fig. 11 The performance benchmark results as a function of requests/s (y-axis) of various frontend service configurations. Plot (**a**) shows performance numbers using existing TCP connections, while plot (**b**) shows numbers without keep-alive TCP connections. A total of 100 tests were performed for each configuration and results show average values from these tests



(**a**) Performance Benchmark using existing TCP connections in cache



(**b**) Performance Benchmark without using existing TCP connections in cache

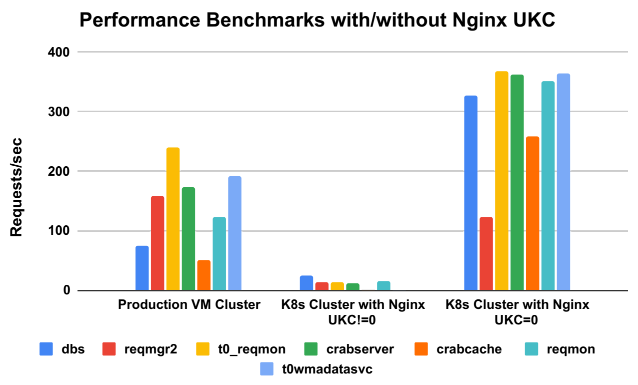


Fig. 12 The performance benchmark results in the form of requests/s (y-axis) of some commonly used CMSWEB services for VM and K8s clusters (x-axis) using Nginx UKC! = 0 and UKC = 0 setting). A total of 100 tests were performed for each configuration and results show average values from these tests

6.1.1 Infrastructure issues

The following issues related to Infrastructure were found:

- *Network degradation* A major issue of network degradation was identified by us and other groups. The issue was found to be caused by the network drivers in Kubernetes 1.15 version. The issue was resolved in a new Kubernetes cluster using version 1.17.
- *Cluster creation issues* With the new Kubernetes version 1.17, cluster creation failed because of timeout. This issue was related to storage volumes on servers that were in different availability zones and had higher latencies, causing timeouts. Availability zones are an end-user visible logical abstraction for partitioning a cloud without knowing the physical infrastructure.[5] This issue was related to infrastructure use and was subsequently fixed by experts in CERN IT.
- *Ceph mount issues* A problem with CephFS mounts showed up after migrating clusters to the new Kubernetes version. This issue was caused by a version of the cloud client which is provided for the management and creation of clusters in the network, which was not compatible with the new version of Kubernetes. Also, this issue was fixed by the experts in CERN IT by upgrading the cloud client to a compatible version.
- *Permission mount issues* Permission issues of the mount point in /etc/grid-security in the new Kubernetes version were found to be related to the security context of an unconfigured pod. A security context defines privilege and access control settings for a Pod or Container. This was fixed after adjusting the manifest files by configuring the security context according to the new Kubernetes version with the help of CERN IT.
- *Nginx-ingress controller Issues* A problem with Nginx-ingress controller was encountered when we performed stress tests on the K8s cluster. It was related to the low value of file descriptors in the Nginx-ingress controller. During stress tests, many requests failed because of that. This was fixed and updated in the new configuration of Nginx-ingress controller provided by CERN IT. Another issue was related to the upstream connections in the Nginx servers. The Nginx uses two types of connections i.e. connections with the clients and connections with the upstream server. The Nginx multiplexes requests from various clients to the upstream server. However, since we use Apache in addition to the Nginx which handles the connections with servers as well, the Nginx upstream connections caused the performance degradation. This issue was resolved after disabling the connections to the upstream server and use Nginx as the proxy only. This setting was applied by setting upstream-keepalive-connections parameter in the Nginx setting to 0.
- *DNS caching issue* During stress tests, we noticed a large number of queries for the backend cluster from the frontend cluster are sent to the central DNS server. This was because of the low value of caching (30 s) in the coredns settings of the cluster provided by the CERN IT. This issue was resolved by increasing caching value to 900 s.
- *CephFS plugin issue* Occasionally, we noticed some CephFS plugin crashed for some reason and did not restore automatically by the Kubernetes. A CMSWEB operator had to manually drain the node on which the CephFS plugin crashed and then uncordon the node after restart fixed that issue. This issue was fixed in the new Kubernetes version 1.19. However, CERN IT also provided the fix to apply to the previous versions.

6.1.2 Service issues

We observed following issues that were related to the CMS services.

- *CouchDB issue* We noticed that CouchDB crashed in the Kubernetes cluster. Handling things like databases imposes specific requirements, like consistency and persistence across distributed service. That makes it challenging to run a database in a distributed environment. The best option is to use a VM which is also called the full-ops option, where the operators take full responsibility for building the database, scaling it, managing reliability, setting up backups, and more. While this is usually a lot of work, it has the advantage

[5] CERN has 4 availability zones which are cern-geneva-a, cern-geneva-b, cern-geneva-c, and nova located at different data-centers.

that all the features and database flavors are at our disposal [40]. It was therefore decided to keep this service in the VM cluster and all CouchDB requests are redirected to the VM cluster.

- *PhEDEx issue* The PhEDEx service is one of the legacy application we are required to support during the transition to Kubernetes. As it is an Apache+Perl based application with its own security authentication module, it was decided to not spend time on porting it to the new infrastructure and keep it in the dedicated VMs.
- *DBS issue* The DBS service requires individual accounts in each cluster. Initially, in the Kubernetes cluster, we used the same accounts as the ones of the VM-based clusters, which caused issues with our workflows in the production VM cluster. In order to avoid potential overwrite of data in production DBS DB instances, we were asked by DBS team to use separate accounts of DBS for the Kubernetes cluster.
- *DBS load balancing issue* During stress tests of the DBS service only, we noticed that most of the requests are forwarded to a single pod and are not uniformly distributed across the cluster. After investigation of the load balancing techniques/settings of the Kubernetes cluster, it was confirmed that the issue itself is in the DBS application and not in K8s infrastructure.

6.2 Lessons learned

The existing VM-based deployment procedure requires a lot of interactions between the CMSWEB operator and developers. In addition, a lot of manual interventions from the CMSWEB operator for service deployment and to maintain the clusters is required. Using the new Kubernetes infrastructure greatly reduces the efforts and workload on the CMSWEB operator. The new Kubernetes infrastructure automates the procedure of service deployment as the developers are able to deploy their services directly in the Kubernetes cluster without needing input from the CMSWEB operator. Furthermore, developers will not have to wait for a month before their services are put into production.

The auto-scaling feature of Kubernetes scales up cluster resources as soon as they are required and scales them back down once they are not needed any more. The current VM cluster lacks this feature; every service is deployed on the particular VM nodes and the resources are assigned on the VM level instead of the service level. When the services are overloaded, they often become unresponsive, then the CMSWEB operator manually interferes and restarts individual services. The auto-scaling feature of Kubernetes, however, automatically manages the resources based on the workload. This greatly simplifies the manual tasks of the CMSWEB operator and also enhances the overall availability of the CMSWEB services. However, there was also a standard limitation of the Kubernetes auto-scaler: it performs auto-scaling based on CPU and RAM usage, while we need auto-scaling based on service-specific matrices. We implemented this feature as discussed in Sect. 4, and can now scale down/up services based on the custom metrics of CMSWEB services.

Manifest files in Kubernetes require verification: a small typo and wrong indentation leads to failures. Therefore, the manifest files need to be carefully written with proper indentation, and ideally verified automatically in a CI process.

7 Future work

After the migration of CMSWEB cluster to Kubernetes infrastructure, we plan to work on the following items:

- *Single cluster model* we plan to replace the present two cluster model with a single cluster model that will run frontend apache service as a daemonset. This model will not use Nginx, which causes an additional layer of complexity and will need fewer resources as compared to the current two cluster model.
- *Service-mesh deployment* the service-mesh provides plenty of benefits to Kubernetes, including traffic encryption within the cluster, traffic routing between different releases, canary deployment, and rolling release cycles. We would like to bring this functionality to our infrastructure either via the Istio or the Gloo middlewares.

8 Conclusions

In CMS at CERN, we have performed the migration of the CMSWEB cluster from the VM cluster to the Kubernetes cluster. In this paper, we give an overview of the CMSWEB VM cluster and the issues we faced during this transition. We discuss the new architecture of the CMSWEB cluster and its implementation strategy in the Kubernetes. Kubernetes perform horizontal pod autoscaling based on CPUs and memory. However, in this paper, we propose horizontal pod autoscaling based on the custom metrics of CMSWEB services and evaluated its performance on the testbed. We implemented service deployment automation based on the best practices of CI/CD workflows. We have done a performance analysis of the CMSWEB cluster in Kubernetes and also performed a comparison with VM based CMSWEB cluster. The results show better performance of the new cluster in Kubernetes as compared to the VM cluster. We describe various issues found during the implementation in Kubernetes and report on lessons learned during the migration process. We also give insight into the future direction of this work.

The new cluster of CMSWEB in Kubernetes enhances sustainability and reduces the operational cost of CMSWEB. With the containerized approach, developers will not have to wait long for the operators to deploy their services, they can deploy new versions of their services in a few seconds. This allows CMS to significantly reduce the release upgrade cycle, follow the end-to-end deployment procedures, and reduce operational cost. The migration to K8s infrastructure has also saved hardware resources (CPU, memory, and disk) by decommissioning them from VMs.

Funding Open Access funding provided by CERN.

Data availability Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

References

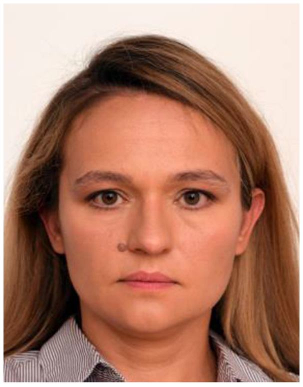
1. Collaboration, C.M.S.: The CMS experiment at the CERN LHC. JINST **3**, S08004 (2008)
2. CMS experiment at CERN. https://home.cern/science/experiments/cms. Accessed 13 April 2020
3. Rad, B.B., Bhatti, H.J., Ahmadi, M.: An introduction to Docker and analysis of its performance. Int. J. Comput. Sci. Netw. Secur. **17**(3), 228 (2017)
4. Luksa, M.: Kubernetes in Action. Manning Publications, Shelter Island (2018)
5. Imran, M., Kuznetsov, V., Marcella, L., Maria Dziedziniewicz-Wojcik, K., Pfeiffer, A., Paparrigopoulos, P.: Migration of CMSWEB cluster at CERN to Kubernetes. In: PoS ICHEP2020, p 911 (2021). https://doi.org/10.22323/1.390.0911
6. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide: Time to Relax. O'Reilly Media, Inc., Newton (2010)
7. Mascheroni, M., Balcas, J., Belforte, S., Bockelman, B., Hernández, J., Ciangottini, D., Konstantinov, P., Silva, J., Ali, M., Melo, A.: CMS distributed data analysis with CRAB3. J. Phys. Conf. Ser. **664**, 062038 (2015)
8. Cinquilli, M., Spiga, D., Grandi, C., Hernandez, J.M., Konstantinov, P., Mascheroni, M., Riahi, H., Vaandering, E.: CRAB3: establishing a new generation of services for distributed analysis at CMS. J. Phys. Conf. Ser. **396**, 032026 (2012)
9. Giffels, M., Guo, Y., Kuznetsov, V., Magini, N., Wildish, T.: The CMS data management system. J. Phys. Conf. Ser. **513**, 042052 (2014)
10. Afaq, A., Dolgert, A., Guo, Y., Jones, C., Kosyakov, S., Kuznetsov, V., Lueking, L., Riley, D., Sekhri, V.: The CMS dataset bookkeeping service. J. Phys. Conf. Ser. **119**, 072001 (2008)
11. De Guio, F.: The CMS data quality monitoring software: experience and future prospects. J. Phys. Conf. Ser. **513**, 032024 (2014)
12. Laurie, B., Laurie, P.: Apache: The Definitive Guide. O'Reilly Media, Inc., Newton (2003)
13. I. MongoDB, Mongodb, URL https://www. mongodb. com/. Cited on (2014) **9** (2014)
14. Rehn, J., Barrass, T., Bonacorsi, D., Hernandez, J., Semeniouk, I., Tuura, L., Wu, Y.: Computing in High Energy and Nuclear Physics (CHEP), vol. 2006 (Citeseer, 2006)
15. Boudoul, G., Franzoni, G., Norkus, A., Pol, A., Srimanobhas, P., Vlimant, J.: Monte Carlo production management at CMS. J. Phys. Conf. Ser. **664**, 072018 (2015)
16. Spinoso, V., Missiato, M.: A flexible monitoring infrastructure for the simulation requests. J. Phys. Conf. Ser. **513**, 032092 (2014)
17. CMSWEB environment at a glance. https://cms-http-group.web.cern.ch/cms-http-group/activity.html. Accessed 4 April 2020
18. CMS repository for Docker. https://github.com/dmwm/CMSKubernetes/tree/master/docker. Accessed 4 April 2020
19. CMSWEB Kubernetes repository. https://github.com/dmwm/CMSKubernetes/tree/master/kubernetes. Accessed 4 April 2020
20. CMSDIST repository. https://github.com/cms-sw/cmsdist. Accessed 4 April 2020
21. Docker Hub repository for CMSSW. https://hub.docker.com/u/cmssw. Accessed 5 May 2020
22. Aimar, A., Corman, A.A., Andrade, P., Fernandez, J.D., Bear, B.G., Karavakis, E., Kulikowski, D.M., Magnoni, L.: MONIT: monitoring the CERN data centres and the WLCG infrastructure. EPJ Web Conf. **214**, 08031 (2019)
23. Monitoring of CMSWEB frontend nodes. https://bit.ly/2BIKvJf. Accessed 6 Sep 2020
24. Cinder storage in OpenStack. https://wiki.openstack.org/wiki/Cinder. Accessed 14 April 2020
25. Mascetti, L., Rios, M.A., Bocchi, E., Vicente, J.C., Cheong, B.C.K., Castro, D., Collet, J., Contescu, C., Labrador, H.G., Iven, J.: CERN disk storage services: report from last data taking, evolution and future outlook towards Exabyte-scale storage. EPJ Web Conf. **245**, 04038 (2020)
26. Turnbull, J., et al.: Monitoring with Prometheus. Turnbull Press, Brooklyn (2018)
27. Ariza-Porras, C., Kuznetsov, V., Legger, F.: The CMS monitoring infrastructure and applications. Comput. Softw. Big Sci. **5**, 5 (2021). https://doi.org/10.1007/s41781-020-00051-x
28. Horizontal pod autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale. Accessed 17 Sep 2020
29. Horizontal pod autoscaler walkthrough. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough. Accessed 17 Sep 2020
30. Prometheus adapter. https://github.com/DirectXMan12/k8s-prometheus-adapter. Accessed 17 Sep 2020
31. General rules for writing Prometheus exporters. https://prometheus.io/docs/instrumenting/writing_exporters/. Accessed 7 Dec 2020
32. Prometheus exporter exposition formats. https://prometheus.io/docs/instrumenting/exposition_formats/. Accessed 7 Dec 2020
33. Prometheus exporters. https://prometheus.io/docs/instrumenting/exporters. Accessed 17 Sep 2020

34. Prometheus exporters assigned ports. https://github.com/prometheus/prometheus/wiki/Default-port-allocations. Accessed 17 Sep 2020
35. HPA via Prometheus walkthrough. https://github.com/Cloud-PG/prometheus-hpa. Accessed 17 Sep 2020
36. ReCaS data center. https://www.recas-bari.it/index.php/en/. Accessed 5 Dec 2020
37. Kinsman, T., Wessel, M., Gerosa, M.A., Treude, C.: How Do Software Developers Use GitHub Actions to Automate Their Workflows? arXiv preprint (2021). arXiv:2103.12224
38. Imagebot. https://github.com/vkuznet/imagebot. Accessed 7 Dec 2020
39. Hey tool. https://github.com/rakyll/hey. Accessed 4 April 2020
40. To run or not to run a database on Kubernetes: what to consider. https://bit.ly/37eB9k9. Accessed 27 April 2021

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

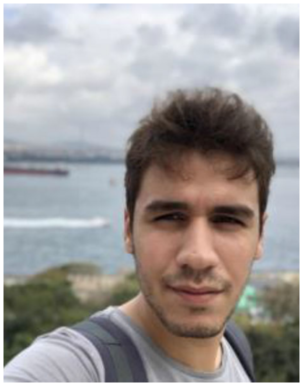
Muhammad Imran received a Ph.D. Degree in Electronic Engineering from Dublin City University, Ireland, in 2017. He is currently working in CMS Offline Computing Group at CERN, Geneva, Switzerland since October 2019. In addition, he holds a permanent position as a Senior Scientific Officer in National Centre for Physics, Pakistan since July 2008. His research interests include cloud computing, cluster computing, big data, data science, software engineering, SDN, and optical networks.

Valentin Kuznetsov received a Ph.D. Degree in Physics from the Joint Institute of Nuclear Research (JINR). Dubna, Russia in 1999. He is currently employed by Cornell University, Ithaca, NY, USA and working in CMS Offline Computing Group at CERN, Geneva, Switzerland. His research interests include data discovery, data management, security, Machine Learning, Big Data, Analytics, and various aspects of Data Science.



Katarzyna Maria Dziedziniewicz-Wojcik received a M.Sc. Degree in Computer Science from Warsaw University of Technology, Poland, in 2008. She has held a position in the Information Technology Department at CERN, Geneva, Switzerland since 2008. She is currently working in the Database Group. Her research interests include databases, big data, distributed systems, and cloud computing.

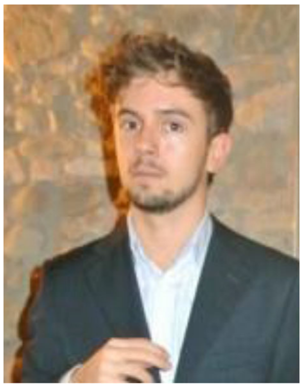
Andreas Pfeiffer got his Ph.D. in Physics from the University of Heidelberg, Heidelberg, Germany in 1988. He has been employed by CERN, Geneva, Switzerland since 1999, and is currently working in the CMS experiment on computing and web-based collaborative services. His research interests cover web service architecture and development, microservices, data management, data analysis, and security/privacy.



Panos Paparrigopoulos is a Computing Engineer at CERN who studied Informatics and Telecommunications at the University of Athens. He is a Member of the CERN WLCG Team and he is currently working on the CMS Web Services Group, CRIC, the Computing Resource Information Catalog, and the Operational Intelligence Initiative. His research interests cover web services development and architecture, machine learning, and data science.

Spyros Trigazis is a Computing Engineer and a Member of the CERN Cloud Infrastructure Team which provides computing resources to the High Energy Physics community. He has been contributing to open-source projects like Fedora, Kubernetes, and OpenStack.

Tommaso Tedeschi received his Master's and Bachelor's Degrees in Physics from the University of Perugia (Italy) in 2019 and 2017, respectively. Currently, he is a Ph.D. Student in Physics at the University of Perugia and collaborates with local CMS experiment computing and analysis groups. His research interests include High Energy Physics data analysis, Machine Learning, Big Data management, cloud, and distributed computing, and other Data Science-related fields.



Diego Ciangottini (m) got his Ph.D. in Physics (2015) in Perugia. He is working as INFN Computing Researcher at Perugia to develop innovative workflow and data management solutions for large scale science and to investigate the impact of a distributed cache layer in a future WLCG data lake. He is also involved in the design of automatic and on-demand deployments involving computing and cache storage resources for different scientific communities at INFN.

Authors and Affiliations

Muhammad Imran[1,2] · **Valentin Kuznetsov**[3] · **Katarzyna Maria Dziedziniewicz-Wojcik**[2] · **Andreas Pfeiffer**[2] · **Panos Paparrigopoulos**[2] · **Spyridon Trigazis**[2] · **Tommaso Tedeschi**[4,5] · **Diego Ciangottini**[5]

✉ Muhammad Imran
muhammad.imran@cern.ch

Valentin Kuznetsov
vkuznet@protonmail.com

Katarzyna Maria Dziedziniewicz-Wojcik
katarzyna.maria.dziedziniewicz@cern.ch

Andreas Pfeiffer
andreas.pfeiffer@cern.ch

Panos Paparrigopoulos
panos.paparrigopoulos@cern.ch

Spyridon Trigazis
spyridon.trigazis@cern.ch

Tommaso Tedeschi
tommaso.tedeschi@pg.infn.it

Diego Ciangottini
diego.ciangottini@pg.infn.it

1 National Centre for Physics, Islamabad, Pakistan
2 CERN, Geneva, Switzerland
3 Cornell University, New York, USA
4 Università degli Studi di Perugia, Perugia, Italy
5 INFN - Sezione di Perugia, Perugia, Italy