Exploring the self-service model to visualize the results of the ATLAS Machine Learning analysis jobs in BigPanDA with Openshift OKD3

Ioan-Mihail Stan[1,*], *Siarhei* Padolski[2,**], and *Christopher Jon* Lee[3,***] on behalf of ATLAS Software and Computing

[1]University Politehnica Bucharest
[2]Brookhaven National Laboratory (BNL)
[3]Stony Brook University

Abstract. A large scientific computing infrastructure must offer versatility to host any kind of experiment that can lead to innovative ideas. The ATLAS experiment offers wide access possibilities to perform intelligent algorithms and analyze the massive amount of data produced in the Large Hadron Collider at CERN. The BigPanDA monitoring is a component of the PanDA (Production ANd Distributed Analysis) system, and its main role is to monitor the entire lifecycle of a job/task running in the ATLAS Distributed Computing infrastructure. Because many scientific experiments now rely upon Machine Learning algorithms, the BigPanDA community desires to expand the platform's capabilities and fill the gap between Machine Learning processing and data visualization. In this regard, BigPanDA partially adopts the cloud-native paradigm and entrusts the data presentation to MLFlow services running on Openshift OKD. Thus, BigPanDA interacts with the OKD API and instructs the containers orchestrator how to locate and expose the results of the Machine Learning analysis. The proposed architecture also introduces various DevOps-specific patterns, including continuous integration for MLFlow middleware configuration and continuous deployment pipelines that implement rolling upgrades. The Machine Learning data visualization services operate on demand and run for a limited time, thus optimizing the resource consumption.

ATL-SOFT-PROC-2021-016
26/06/2021

1 Introduction

Cloud computing is perhaps one of the most important game changers in the IT business industry in the last 10 years. With the advent of container engines and complementary cloud orchestrators, the entire industry moved forward to a new era - an era of portability. Therefore, the software can now be bundled together with dependencies and runtime and run on various heterogeneous systems. In addition, aspects like scalability, security or high availability can be outsourced to a cloud orchestrator and fully provided by this software solution.

Unquestionably, Kubernetes remains a relevant technology when talking about modern cloud computing. It is a highly customizable orchestration solution that can address a wide

*e-mail: ioan.stan@upb.ro
**e-mail: siarhei.padolski@cern.ch
***e-mail: chris.lee@cern.ch

This work was supported by the U.S. Department of Energy, Office of Science, High Energy Physics contract No. DE-SC0012704 and by the U.S. National Science Foundation.

range of production scenarios and configurations[1, 2]. It is mainly used to manage containers via Pods and it also facilitates the development of native cloud applications. Led by the Cloud Native Computing Foundation, the Kubernetes project is used as a foundation for multiple commercial and community-driven orchestrators. An important Kuberentes flavor available on the market, is OKD, an open-source version of the Openshift Container Platform, currently maintained by the Redhat community. OKD combines the versatility of Kubernetes with some production-graded automation mechanisms and security admission controllers, to provide a fully clusterized solution, ready for production activities. In terms of DevOps methodologies[3], the OKD supports the construction of continuous integration, delivery or deployment pipelines, aiming to support the entire application development lifecycle. Similar to Kubernetes, the OKD exposes a complex API, offering the possibility to interact and integrate with 3rd party solutions.

In the context of ATLAS Distributed Computing[4], Openshift OKD[5] can serve as an extension to BigPanDA[6]. BigPanDA is a complex monitoring tool that provides information about distributed analysis processing[7, 8]. Developed as a pilot project, the current case study focuses on demonstrating that Openshift OKD can take over part of the workload of BigPanDA. The proposed solution will give the BigPanDA service the possibility to delegate the governance of the Machine Learning[9, 10] data visualization microservices to a Kuberentes solution. Thus, if a tenant would like to see the results of a machine learning activity previously processed by the dedicated infrastructure of the ATLAS Distributed Computing[11], a corresponding catalog service can be accessed, on request, from the BigPanDA interface. BigPanDA will further locate the results, and it will spin up a new MLFlow[12] service container in the OKD Openshift cluster to display the required data. MLFlow is an open source platform that aims to support the entire Machine learning life cycle. It has several features very useful for scientists working with Machine Learning data. Among these features, we can find tracking capabilities for experiments, packaging mechanisms for projects, models management and versioning. For our particular purpose, we selected MLFlow to view experimental data in an intuitive format. Each MLFlow instance will be exposed to the outside world, and it will be accessible from the browser. The web service will be discoverable for a limited time, after which the BigPanDA core controller will delete the MLFLow instance and, consequently, will automatically recover the associated computational resources. Openshift OKD implements multi-tenancy, therefore, multiple MLFlow instances can run in parallel, serving multiple stakeholders.

The aim of this pilot project is to probe whether Openshift can supplement BigPanDA on a daily basis for activities that are not the main purpose of the platform. Furthermore, by taking this approach into account, some doors can be opened for a transition from the current architectural model to a cloud native or hybrid model, able to run in any public or private cloud. The goal is to expand the platform's capability with loosely-coupled features, managed by 3rd party cloud orchestrators. The logic of governance will also be outsourced, taking advantage of the orchestration routines and all the abstractions provided by Kubernetes and Openshift OKD.

In the current paper, we will formally discuss aspects related to the implementation of a new capability of the BigPanDA platform which consists in visualizing the results of Machine Learning jobs in an on-demand system. In Section 2, we will present the concept architecture of our hybrid solution, including the architectural decisions based on the constraints encountered. In Section 3, we will reiterate through the functional objects and present the way we approached each particular aspect from a technical perspective. In addition, we will present some aspects that can be optimized in a future release. In Section 4, we will run a quick functional test suite in order to validate end-to-end usability. In Section

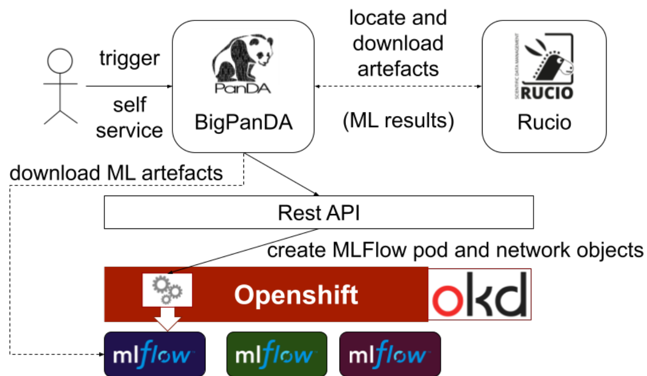

Figure 1: Concept architecture of a self-service system for data visualization services resulting from the processing of machine learning jobs/tasks

5, and the last section, we will conclude the development and integration process and provide the results of our tests.

2 Architecture

In the current case study, the authors will focus on defining the interaction between BigPanDA and Openshift OKD. The context of such an interaction is related to the need to view Machine Learning analysis data in a friendly format. At the same time, a management model detached from the core ATLAS orchestration is being tested. As can be observed in the concept architecture and interaction flow (Figure 1), a tenant, which has previously submitted a Machine Learning job or task to the ATLAS processing infrastructure, can also request a visualization service to display the results. For each demand, the BigPanda controller triggers the creation of a web service in Openshift OKD. As part of this routine, BigPanda's role is to locate the results and download[6] them from the ATLAS distributed infrastructure, via Rucio[13]. Once data is stored and indexed within BigPanDA, the controller calls the Openshift OKD API and triggers the creation of an MLFlow web service along with all the Openshift (Kubernetes) communication and configuration objects. The MLFlow pod downloads the Machine Learning artefacts from BigPanDA and stores them locally, in a temporary/volatile location.

Following such an approach, the BigPanDA solution outsources data visualization management to an external cloud platform, optimized for this type of interaction. The principle on which we based our architectural model is "segregation of duty". Therefore, instead of having one solution that fits all, various auxiliary routines can be detached from the main solution and executed through specialized platforms. Openshift OKD is a viable platform for various scenarios, especially when it involves creating on-demand services, multi-tenancy and intelligent container management. To take advantage of OpenShift and also to increase the portability of our solution, we deliver our MLFlow instances in containers and pods. Following this cloud native model, our solution is extremely portable and can be easily adjusted to run on most public and private clouds.

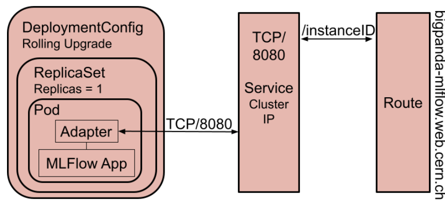

Figure 2: Openshift objects and communication model

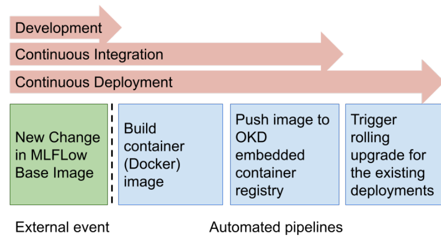

Figure 3: DevOps practices

Openshift OKD is a Kubernetes implementation, therefore, it inherits the same base architecture as the one implemented by the core project. A service that needs to be exposed to the outside world will most often be accessible through the ingress-managed load balancer. Therefore, if a client would want to connect to a service running inside an Openshift OKD cluster, the client will be able to use a specific domain name to reach that service. Furthermore, an ingress controller can also manipulate internal routes to forward traffic toward various services based on subdomains or paths selectors. As we will further discuss in the next section, BigPanDA will use the routing mechanism (ingress controller) to enable multi-tenancy by creating unique fan-out definitions (Figure 2) and communication primitives for each MLFlow instance, through the OKD API.

In addition, we observed that both OKD 3.11 and MLFlow v1.9 (versions selected for our proof of concept) do not support target rewriting. This concept means that if an HTTP request path does not correspond to an existing resource or application endpoint, the request will end up generating a resource-not-found error. Since we will be using one DNS domain for all MLFlow instances running in parallel, we will identify each instance by a random string embedded within the resource path. Some web servers or web applications support remapping a non-existing endpoint to the root path. However, both out-of-the-box OKD 3.11 with HAProxy Ingress Controller and MLFlow v1.9 do not have support for such configuration. To work around this issue, we adopted a multi-container design pattern for pods - the adapter pattern[14]. An Adapter object is located in front of the main MLFlow application service, and it handles the requests coming from the load balancer. Each HTTP request will be translated and sent to the MLFlow service through the localhost interface (Figure 2).

Finally, and also part of the architectural vision, we decided to create our own base container. It includes the MLFlow middleware and the assembly scripts and is built using the native build capabilities available in Openshift OKD. Therefore, we separated the preparation of the base MLFlow image from the actual application deployment, respecting the DevOps methodology and practices (Figure 3). All the configuration items are currently stored separately from the BigPanDA code base, and everytime a change is detected, a trigger executes the Continuous Deployment pipeline.

3 Implementation

In order to spin up a data visualization process for Machine Learning analysis tasks in Kubernetes, a series of steps has to be performed in order to make this service available for consumption. Each configurable item in Kubernetes and Openshift can be generated through

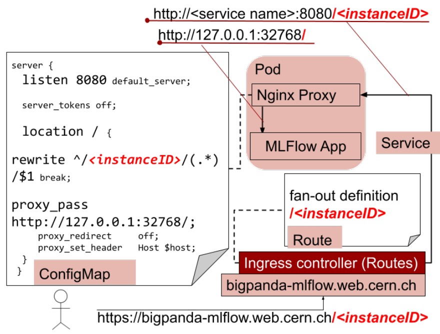

Figure 4: Adapter entity configuration and communication flow

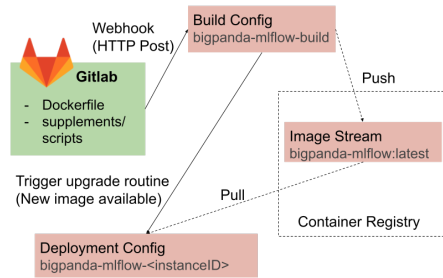

Figure 5: The interaction flow to external events

native or custom object definitions, processed by specific internal controllers and manipulated from outside via Kubernetes/Openshift API. In our case study, BigPanda needs to create four such objects, two defining the web-service that includes the MLFlow application and Machine Learning artefacts and other two abstracting the network configuration (Figure 2).

The access to data visualization service is provided through Routes (ingress definitions). When a new MLFlow service is requested from BigPanDA, the platform generates a seven byte unique id per request. BigPanDA uses this string to create a mapping between the workload id of a job/task submitted in the ATLAS processing infrastructure and the corresponding Openshift request. As this string uniquely identifies a MLFlow service instance, it is also used to compose the unique URI for external access. Therefore, the string is concatenated as an endpoint (path) to a shared, pre-provisioned DNS domain (fan-out definition) and configured internally to reach the corresponding MLFlow instance. This approach facilitates the multi-tenancy character of the ensemble, as each MLFlow service will be obfuscated and reachable by a specific, randomized URI. However, once a request passes the ingress controller and it is forwarded to the MLFlow web service, the HTTP request keeps the original path (unique id) in the HTTP request header. When an HTTP request reaches the MLFlow service, the middleware tries to find, in the application structure, an endpoint corresponding to the path selected. Since there is no such resource, MLFLOW will return an HTTP 404 error - resource not found.

Because the purpose of having a random endpoint in the URI is to select the appropriate MLFlow instance, the need to have an intermediate HTTP proxy arises. Therefore, if the application itself is not able to deliver resources from custom endpoints, an HTTP request adapter must handle the original HTTP call and alter the original content. Therefore, the adaptor must remove the random path string and replace it with the root (/) path. In the vanilla Kubernetes implementations, the most common ingress management solution adopted is the NGINX ingress controller, which, by design, supports target rewriting via annotations. However, Openshift OKD 3.11 comes with an out-of-the-box HAproxy Ingress Controller for managing routes which does not have support for these particular use cases. Also the MLFLow version 1.9 which has been selected for the current Proof-of-Concept, is also unable to handle custom endpoints for serving data. In this regard, as described in the previous section, we adopted an adaptive pod design pattern. Therefore, instead of having only one container exposing the Machine Learning job results, our solution proposes a two container model, one acting as a proxy (adapter), filtering all requests and staying in front of the MLFlow service and the other, the web service itself, responsible for data presentation.

In programming, this approach is called the adapter model/pattern. As shown in Figure 4, the adapter entity works as a reverse proxy, altering the HTTP header of the initial request and replacing the random path sting with the root path. Since a Pod, in Kubernetes terminology, shares the same network namespace among the containers, the communication between the MLFlow service and the Proxy-Pass NGINX server is handled by the localhost. In more extensive scenarios, an adapter service can define more complex models, acting on the entire spectrum of input data. In our particular case, the main role is to remove any identification information from the request and to expose the MLflow capabilities to outside on behalf of the webservice.

Machine learning analysis data must be located within the MLFlow container. For this initial iteration of our pilot project, the data is downloaded from BigPanda and temporarily stored inside the container. When a new Pod is created, the initial step performed by the main container is to connect to the BigPanDA and to obtain the specific machine learning results. The source of data is encoded in an environment variable and set in the pod definition at the time of creation. At this point in time, the BigPanDA openshift controller sends a request to the Openshift OKD API in order to create the application service and all the communication primitives. Since the underlying pod volumes are volatile, data downloaded from BigPanDA will be conserved as long as the web service is up and running. In case of a failure, the pod may be recreated, and the data may be lost during this process. Therefore, a new container will have to re-download all the artefacts. This approach respects the principle of idempotency, essential when developing cloud applications for Kubernetes. Thus, one has to ensure that consecutive executions of the same initialization routine will not affect the results. A more optimal solution would be to use init containers and maybe also to mount directly the distributed ATLAS storage system[15] in the container. By implementing the latter feature, our solution will avoid duplicating the date and it may contribute to the sustained effort to ensure a smaller disk footprint in the organization. For the former idea, the approach will help in cleaning the architectural model by detaching the initialization function from the actual execution.

Opting for the OKD Openshift could also provide the opportunity to adhere to a safer and more restrictive operating model. OKD enforces several security policies over the container context, including non root execution and UID randomization. Therefore, in the context of an on-premise cluster implementation, a vulnerability in one of the components running inside a container will not expose the entire hosting system. Moreover, for our particular toolkit, all the artefacts selected were imported from official certified sources, which are constantly updated and actively maintained. An essential process to ensure a quick reaction in the event of a major security problem is the use of the OKD automation pipelines (Figure 3 and Figure 5). In addition to the existing Kubernetes primitives, OKD Openshift brings several other solution-specific objects to support the creation of an application development lifecycle pipeline. Thus, we benefit from the existence of such a feature to manage the construction and configuration of our own MLFlow middleware container image, using OKD's native BuildConfig routine. Moreover, once the middleware image is reconstructed and stored in the embedded registry (artefacts repo), the build routine notifies the DeploymentConfig controller about the changes. This process will further trigger the upgrade routine, and it will redeploy all the existing running MLFlow instances with the new middleware version. The update strategy selected for our pilot is the rolling-upgrade model. This strategy significantly reduces the downtime by running two application instances at the same time - one with the new version and the other the current one. Once the new application instance is initialized and ready to serve content, the network will automatically configure the routing service to point to the new instance.

All the middleware configuration items and build artefacts are stored in an external GitLab repository. Once a PUSH event is detected, Gitlab notifies Openshift via a predefined webhook. When the endpoint is reached, Openshift OKD starts a new build process, which imports the latest version of the artefacts and assembles them according to the instructions. During this process a new container image is built. Once this process is successfully completed, the rolling-upgrade routine is also triggered. Therefore, all the existing MLFlow instances, running on the cluster, are automatically updated one by one (Figure 5). As can be seen in Figure 3, the entire pipeline is developed in accordance with DevOps methodologies and practices and it is fully generated with native OKD Openshift routines.

The last component developed for the integration between BigPanDA and Openshift is a service deletion routine (garbage collector). The adopted model proposes a retention period of 24 hours for both services and data. After this period, BigPanda will clear all resources associated with a previously submitted request. In order to partially detach this routine from the BigPanDA controller, one way would be to use privileged Kubernetes cron jobs. For this particular approach, we have developed a Python procedure that interacts with the OKD API and scans for DeploymentConfig objects within the project namespace. This procedure retrieves the creation timestamp from the objects attributes (deploymentconfig.metadata.creationTimestamp) and calculates the number of hours elapsed since initialization. If the 24-hour deadline has passed, the python routine will also collect the instance ID, encoded in the labels section at the time of creation, and delete all associated objects (all four Kuberentes/Openshift objects identified by the same instance id). Kuberentes uses user-defined labels (key value pairs) to identify and manipulate objects. For our solution, we decided to attach the MLFLow service identification request string in the label section of each object created in OKD. Thus, when the garbage collector identifies an old instance running within the cluster, by just retrieving the identity string, our Python procedure should be able to identify all the objects corresponding to an old request (Fig. 02) and delete them. The Kubernetes cron job can be set to run once per hour and therefore, to continuously monitor the cluster.

3.1 Opportunities for improvement and future work

For a second release, we have already identified several optimization development tracks aiming to improve both the provisioning process and architectural model. A way to better segregate the responsibility for certain processes is to work with micro services, small routines that accomplish only one atomic activity[10]. Currently, the main application container processes two important tasks at bootstrap time: download Machine Learning analysis data from BigPanda and start the MLFlow web service. In order to separate responsibilities and to improve the overall experience, especially regarding recovery policies applied by Kubernetes, the data retrieval task can be delegated to an initialization container entity. An init container is an application container that runs during the initialization stage and must run to completion before starting the main application containers (MLFlow and Adapter). Once such a segregation model is applied, if a health check probe fails for example (eg: liveness probe detects an anomaly), Kubernetes will only try to restart the application service and not the entire initialization process. The artefacts can be shared through emptydir volumes and attached to both sequential containers. Furthermore, in order to keep the idempotency and to follow the best practices, BigPanDA will also have to compute and provide the checksum of the Machine Learning artefacts. This will allow the init process to compare what version of data is available upstream and what is already downloaded locally.

As previously mentioned, another tangential scenario could be the configuration of Kuberentes pod-specific health checks (liveness, readiness and startup) for each MLFlow

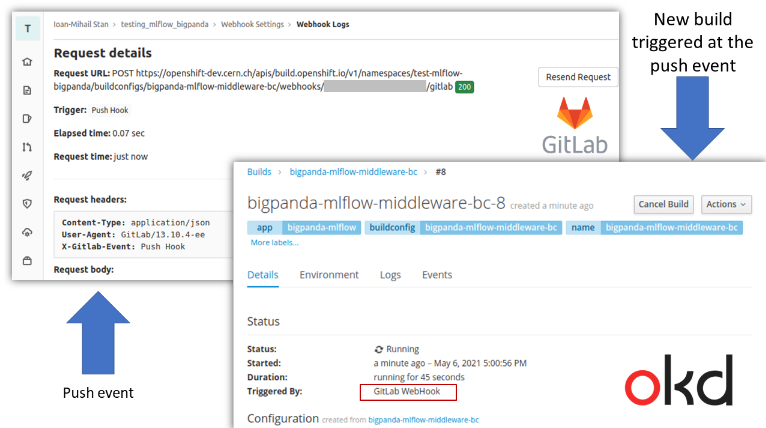

Figure 6: The interaction between the source code management platform and the cloud orchestrator at middleware configuration change events

instance. Although MLFlow service doesn't have a special / health endpoint to be queried, one can use the tracking services (mlflow. set_tracking_uri ()) to check the consistency of the web service.

Another possible improvement would be to combine all OKD object definitions into a single jumbo object (template), applied through a single REST API call from BigPanda to OKD Openshift. However, in a modern approach, the previous capability can be further enhanced through operators, a design pattern and development framework that extends the existing Kubernetes / OKD API and can delegate the provisioning management of MLFlow instances to a custom developed controller. Thus, instead of sending one or more HTTP calls describing four native Kubernetes/Openshift objects, we will need to create a single custom object description that instructs our custom controller how to provision a new ATLAS MLflow instance.

In addition, the migration to OKD 4 could also improve the current architectural model, as since its official launch, this flavor provides an enhanced HA proxy, capable of handling target rewrites. Therefore, we can reduce the footprint of the main application pod by removing the adapter function.

4 Functional testing (end-to-end testing)

As mentioned in the previous sections, we are using GitLab to store all the configuration items for middleware (MLFlow). GitLab supports event monitoring and notification and it has native integrability with 3rd party solutions through several interfaces including webhooks. Therefore, when a PUSH event occurs, GitLab runs the webhook routine and it sends an HTTP POST message to the Openshift OKD, announcing that a change has been detected. As can be seen in the Figure 6, this event triggers a build process that concludes by pushing a new container image to the internal registry. Following the DevOps methodology, this chain of events defines a continuous integration pipeline.

During the build process, the entire image configuration repository is cloned. The build pipeline uses the Docker Build strategy, a method that searches for a Dockerfile in the repository and creates a docker container image based on the instructions written there. In the Figure 7, it can be seen that the previous statement is correct. Once the build pipeline is

Figure 7: Middleware container image build routine triggered automatically when configuration changes are detected

Figure 8: Applying the rolling upgrade strategy on active middleware instances when new service image becomes available

triggered, the entire repository is cloned with the last commit, the Dockerfile is located within the root folder and the build process runs to completion.

The link between a build and a deployment can be provided by triggers, an Openshift OKD native mechanism that monitors a build process and notifies the upgrade routine about the existence of a new base container image version. Thus, we have used a trigger to constantly check the base image fingerprint (MLFlow docker image) and to initiate a rolling upgrade process when the base image has changed. Following this strategy, a new pod with the new version is created and executed in parallel with the current one. Once the initialization stage is concluded and all the health checks are passed, the networking is switched to the new pod, and the old one is killed. In Figure 8, this process can be seen in action, as we captured the moment of transition between pods.

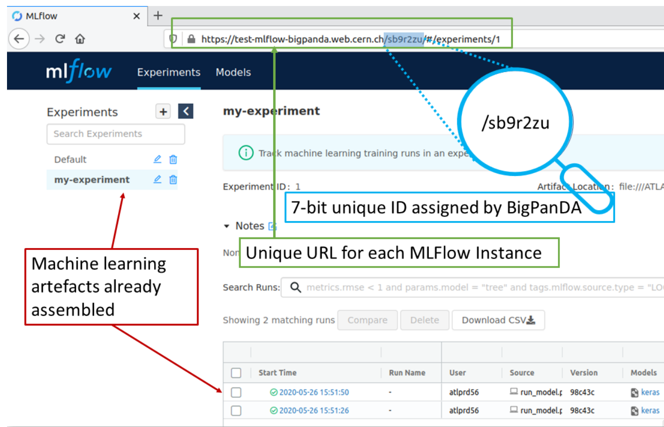

Figure 9: Data visualization service exposed as a web application and uniquely identified by the request ID

Each MLFLow instance will have a unique ID generated by BigPanDA and configured in the Openshift object definitions. This random string is applied as a fan-out definition in the ingress router and also as part of the proxy-pass config in the Adapter component. As can be seen in the Figure 9, the fan-out definition is properly configured and the randomized URL is pointing to the corresponding MLFLow instance. Also the service is accessible from the browser, meaning that the service has been properly exposed to the outside world.

5 Conclusion

In the current paper we have developed a new feature for BigPanDA that offers the ability to visualize the machine learning results, produced in the ATLAS Distributed Computing infrastructure. A tenant (PhD student, scientific groups etc.) can request such a service directly from BigPanDA, and the workload will be further delegated to an Openshift OKD cluster via REST API Calls. Openshift will spin-up a MLFlow pod per request, download the Machine Learning artefacts from BigPanDA, expose the web service to the outside world and maintain high availability through the native healing mechanisms. Following this delegation model, we demonstrated that BigPanDA can easily adopt a cloud native approach and also that it can function as a catalog of scientific services, in the context of the ATLAS Distributed Computing at CERN. Moreover, we also followed several DevOps methodologies and practices to facilitate the build of the base MLFlow container from scratch. Therefore, we implemented continuous integration and continuous deployment pipelines using the native mechanisms of Openshift OKD. These pipelines are triggered automatically each time a PUSH event occurs in the external configuration items repository. Furthermore, as a good practice, the deployment strategy is using the rolling-upgrade model, a method that minimizes downtime inevitable for an upgrading process. Finally, we also implemented a garbage collector, a Python procedure that identifies old service instances and deletes them if they passed the 24 hours expiration time.

The results obtained during the testing phase certifies the end-to-end functionality of the integrated solution. For new iterations, we have already identified a few other paths of development and optimization and here we include: the use of initialization containers, a migration to the operators pattern, use of health probes and the delegation of the routing

process to another ingress controller solution. These changes can also bring significant improvements to the solution architecture since they apply a better segregation model and also eliminate the need of having auxiliary components like the adapter object.

References

[1] A.M. Beltre, P. Saha, M. Govindaraju, A. Younge, R.E. Grant, *Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms*, in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)* (IEEE, 2019), pp. 11–20

[2] M. Orzechowski, B. Balis, K. Pawlik, M. Pawlik, M. Malawski, *Transparent deployment of scientific workflows across clouds-kubernetes approach*, in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* (IEEE, 2018), pp. 9–10

[3] B. Kiyana, T. Vardanega, *DevOps meets dynamic orchestration*, in *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment* (Springer, 2018), pp. 142–154

[4] *ATLAS Distributed Computing*, https://twiki.cern.ch/twiki/bin/viewauth/AtlasComputing/AtlasDistributedComputing (2021), accessed: 2021-01-28

[5] *OKD - The Community Distribution of Kubernetes that powers Red Hat OpenShift*, https://www.okd.io/ (2021), accessed: 2021-06-17

[6] T. Korchuganova, A. Alekseev, S. Padolski, T. Wenaus, A. Klimentov, *The ATLAS BigPanDA Monitoring System Architecture* (CEUR Workshop Proceedings, 2018)

[7] A. Alekseev, A. Klimentov, T. Korchuganova, S. Padolski, T. Wenaus et al., *ATLAS BigPanDA monitoring*, in *Journal of Physics: Conference Series* (IOP Publishing, 2018), Vol. 1085, p. 032043

[8] S. Padolski, T. Korchuganova, T. Wenaus, M. Grigorieva, A. Alexeev, M. Titov, A. Klimentov, *Data visualization and representation in ATLAS BigPanDA monitoring* (Scientific Visualization, 2018), Vol. 10, pp. 69–76

[9] I. Vukotic, R. Gardner, D. Barberis, F. Legger, *ATLAS Analytics and Machine Learning Platforms* (CHEP, 2018)

[10] A. Radovic, M. Williams, D. Rousseau, M. Kagan, D. Bonacorsi, A. Himmel, A. Aurisano, K. Terao, T. Wongjirad, *Machine learning at the energy and intensity frontiers of particle physics* (Nature Publishing Group, 2018), Vol. 560, pp. 41–48

[11] S. Campana et al., *ATLAS Distributed Computing in LHC Run2*, in *Journal of Physics: Conference Series* (IOP Publishing, 2015), Vol. 664, p. 032004

[12] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S.A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe et al., *Accelerating the Machine Learning Lifecycle with MLflow.* (IEEE Data Eng. Bull., 2018), Vol. 41, pp. 39–45

[13] C. Serfon, M. Barisits, T. Beermann, V. Garonne, L. Goossens, M. Lassnig, A. Nairz, R. Vigne et al., *Rucio, the next-generation Data Management system in ATLAS*, Elsevier (ScienceDirect, 2016), Vol. 273, pp. 969–975

[14] B. Burns, D. Oppenheimer, *Design patterns for container-based distributed systems*, in *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016)

[15] F. Berghaus, K. Casteels, A. Di Girolamo, C. Driemel, M. Ebert, F. Furano, F. Galindo, M. Lassnig, C. Leavett-Brown, M. Paterson et al., *Federating distributed storage for clouds in ATLAS*, in *Journal of Physics: Conference Series* (IOP Publishing, 2018), Vol. 1085, p. 032027