



Medical Imaging Processing Architecture on ATMOSPHERE Federated Platform

Ignacio Blanquer^{1,2} ^a, Ángel Alberich-Bayarri^{2,3} ^b, Fabio García-Castro³, George Teodoro⁴, André Meirelles⁴, Bruno Nascimento⁵, Wagner Meira Jr.⁵ and Antonio L. P. Ribeiro⁵

¹*Instituto de Instrumentación para Imagen Molecular, Universitat Politècnica de València, Valencia, Spain*

²*GIBI230, Biomedical Imaging Research Group-IIS La Fe, Valencia, Spain*

³*QUIBIM, Valencia, Spain*

⁴*Universidade Nacional de Brasília, Brasília, Brazil*

⁵*Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais, Brazil*

Keywords: Cloud Federation, Privacy Preservation, Medical Imaging.

Abstract: This paper describes the development of applications in the frame of the ATMOSPHERE platform. ATMOSPHERE provides means for developing container-based applications over a federated cloud offering measuring the trustworthiness of the applications. In this paper we show the design of a transcontinental application in the frame of medical imaging that keeps the data at one end and uses the processing capabilities of the resources available at the other end. The applications are described using TOSCA blueprints and the federation of IaaS resources is performed by the Fogbow middleware. Privacy guarantees are provided by means of SCONE and intensive computing resources are integrated through the use of GPUs directly mounted on the containers.

1 INTRODUCTION

Cloud federated infrastructures are normally used to distribute requests to balance workloads. Cloud bursting and metascheduling in cloud orchestration enables multiple providers to deal with resource demand peaks that cannot be fulfilled locally. Moreover, Cloud federation also opens the door to other requirements such as complementing cloud offerings with additional resource configurations or dealing with restrictions based on geographic boundaries.


ATMOSPHERE exemplifies one of this usages with the support of elastic container-based applications on top of a federated IaaS and using secure storage and a federated network. In the specific case of ATMOSPHERE, applications are a combination of several levels of services involving the actual medical imaging applications and the execution environment. On the top, application developers create medical-image processing applications for model creation, data analysis, and production. The applications also include different dependencies in software com-


ponents with different restrictions and requirements. Applications may include functions for the provisioning of metrics to the trustworthiness system and the adaptation of the application.

1.1 Description of the Use Case

The use case focuses on the early diagnosis of the Rheumatic Heart Disease (RHD). The RHD is a disease that can be easily treated in its early stages and it remains the main cause of heart valve disease in the developing world. However, it may produce severe damage to the heart if it remains untreated, including severe sequelae and death. In addition, early detection is currently limited, since no single test accurately provides early diagnosis of RHD.

The model building application is a conventional model-driven image analysis algorithm that allows the extraction of features that are then fed into a classifier by means of deep learning techniques based on Convolutional Neural Networks (CNNs). This model is used to build a classifier should serve as a first screening to differentiate between normal and abnormal (screen-positive: definite or borderline RHD)

^a  <https://orcid.org/0000-0003-1692-8922>

^b  <https://orcid.org/0000-0002-5932-2392>

echo-cardio studies.

The training is based on a large database includes 4615 studies that contain several echocardiogram videos and demographic data, classified into three categories (Definite RHD, Borderline and Normal). The first step is to differentiate the views (e.g. Parasternal long axis, right ventricular inflow, basal short axis, short axis at mid or mitral level, etc.) for their appropriate classification.

2 SOFTWARE ARCHITECTURE

2.1 Application Types

We differentiate between two types of applications: On the one hand, we consider Pre-processing applications, which build the classifying models or extract relevant features that require complex processing and are used by users knowledgeable in computing. These applications are typically computationally-intensive and require multiple executions over the same data to fine tune the working parameters. On the other hand, production applications are used to classify images, using the trained models from the previous application and do not require (individually) intensive computing, although they may be accessed by multiple users simultaneously or applied to multiple input data concurrently.

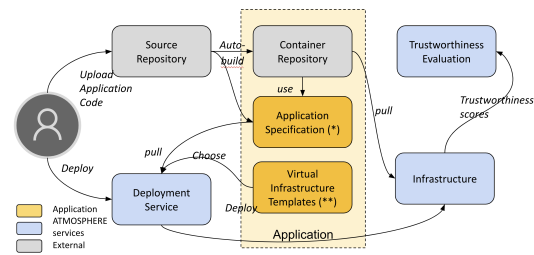
An application in the context of ATMOSPHERE includes mainly three components:

- The final application, which comprises the interface layer and the processing back-end.
- The application dependencies, coded in Dockerfiles that the automatic system will use to build the container images.
- The virtual infrastructure blueprint, which includes third-party components for managing the resources and the execution back-end.

2.2 Application Development

Figure 1 shows a detailed diagram of the interactions at the level of the development and deployment, which are closely coupled. In this figure, we differentiate with a colour code the services that are included in the ATMOSPHERE platform. Grey boxes relate to external services.

The development cycle (top part of figure 1) involves three main interactions: First, the management of the source code of the application, which involves



(*) https://github.com/eubr-atmosphere/d42_applications
 (**) <https://github.com/grycap/ec3>

Figure 1: Deployment use cases from the Application developer perspective.

writing the code and the tests. Second, the specification of the software dependencies, by defining containers and/or Ansible roles that install the dependencies required. Third, the customization of the application with the trustworthiness properties to satisfy the requirements of a specific execution. The data for these three main blocks may be stored and managed in the same repository. The deployment cycle (bottom part of figure 1) involves two steps:

- Building of the container where the dependencies are embedded. This process will be automatic, once the proper processing pipelines are defined.
- Deploying the Virtual Infrastructure to run it.

3 IMPLEMENTATION OF THE ARCHITECTURE

3.1 Application Back-end

The application is based on an Elastic Kubernetes (kub, 2019) cluster, which will be deployed along with the infrastructure:

- A Kubernetes cluster, to manage the containers and services.
- A CLUES (de Alfonso et al., 2013) elasticity manager, linked to Kubernetes and the Deployment Service, which will request the deployment of additional nodes as new resources are requested due to the deployment of new pods.
- A storage service, based on a shared volume, exposed to the pods within the deployment. Eventually, a database may be instantiated.

3.2 Application Topology

From the Application Developer point of view, two components are required:

- A set of Docker containers with the processing code and the dependencies. The containers are build up on top of the base images provided by ATMOSPHERE, as provided in Docker Hub EU-BraATMOSPHERE organization¹. The catalog include images supporting Machine Learning and image processing software.
- A Jupyter Notebook instance with the notebooks with the code to implement the processing actions and its visualization. This Notebook will run in a container deployed on Kubernetes, with the libraries needed to run Kubernetes jobs and to access the data from a shared volume.

3.3 Data Access Layer

ATMOSPHERE will provide two types of secure access. As the interfaces will be standard (POSIX and ODBC/JDBC), this will not affect application development, which could be implemented using standard packages and finally substituting them by the ATMOSPHERE Data Management components. ATMOSPHERE will also provide insecure file systems, and local volumes to store data that do not need special protection. By insecure we do not mean that they are not access protected, but that they are not executed in an encrypted environment.

Figure 2 shows the mapping of the storage components into the ATMOSPHERE services. Data that does not need special protection, such as totally anonymised data, public data or intermediate data may fall in this category. The application will include an NFS component that may use a Kubernetes volume for persistent storage. Data will be accessible only through the internal network, where all the applications are deployed.

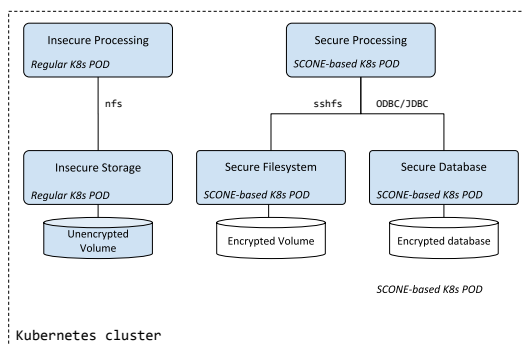


Figure 2: Architecture of the storage solutions for the applications in ATMOSPHERE.

The second scenario is much more complex and re-

¹<https://cloud.docker.com/u/eubraatmosphere/repository/list>

flects the added value of ATMOSPHERE. Sensitive data must be stored encrypted and access keys will only be provided in a secure environment. Moreover, processing will only be possible in a protected container running on an enclave. This way we can guarantee that the system administrator, or anyone that has granted access to the physical or virtual resources is not able to access the data. It is important to state that all the communication takes place in the private overlay network, using secure protocols. This model is valid for both the filesystem and the databases. Access control will be available at volume level. Different tenants will have separate deployments. Even if there are different applications in the same federated network, access control will guarantee that only the users authorised for accessing a volume will be accepted. Despite there will be no fine-grain file-level authorization, the model fits quite well the requirements and expectancy of the users.

3.4 Deployment

The application is implemented as a set of services delivered inside containers deployed by Kubernetes. The first part is to deploy a Kubernetes cluster on top of the federated offering.

For this purpose, we make use of Elastic Compute Clusters in the Cloud (EC3) (Calatrava et al., 2016)(Caballer et al., 2015). EC3 provides the capability of deploying self-managed clusters in different cloud backends, including Fogbow (Brasileiro et al., 2016). EC3 self-manages the Kubernetes cluster by adding or reducing working nodes to the cluster depending on the workload. The cluster is expanded along a federated network and resource matching is performed by the affinity of the pods and deployment to the storage and the resource dependencies.

3.5 Implementation

The application leverages the federation model to implement an scenario where data should not be stored outside of the boundaries of a specific datacentre, while processing capabilities are located in other datacentre. For this purpose, we identify two scenarios depending on the privacy requirements of the data to be processed.

Privacy-sensitive data must be securely processed. The data will be stored encrypted in a persistent storage and it will be accessed through inside a secure container (using SCONE (Arnautov et al., 2016) and VALLUM (Arnautov et al., 2018)). Despite the data is stored only in the authorized site, the processing containers can run on any site, accessing the data through

a secure connection.

Not privacy-sensitive data has less restrictions. They can be accessed remotely and transferred to resources out of the secure boundaries. The result of the processing (classifiers, error estimations) can be safely stored in a persistent storage in any site, and can be accessed only with the application credentials.

Figure 3 shows the architecture of the application.

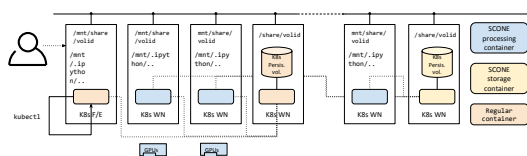


Figure 3: Architecture of the applications in ATMOSPHERE.

The distributed execution environment is provided by an iPython Parallel cluster, the interface is provided by a Jupyter notebook and the storage is managed through a ssh server and sshfs. For this purpose, ATMOSPHERE has built the following containers:

- Front-end container. It includes Python3, Jupyter notebook, sshfs client, Tensorflow and Kubernetes client tool. It mounts the Kubernetes credentials to enable managing the deployment, starts the ipcontroller service and mounts via sshfs the remote filesystem.
- Processing worker. It includes Python3, sshfs client and Tensorflow, and runs the ipengine service that binds to the ipcontroller service, and mounts via sshfs the remote filesystem. The containers access the GPUs mounting the device from the VM, which is connected to the Hardware via PCI passthrough.
- File Server. It is basically a ssh server that mounts an encrypted filesystem and obtains the credentials using VALLUM.

3.6 Parallel Execution

The parallel application uses the ipyparallel module to run the Tensorflow application on all the nodes. A client can connect to the cluster to run a callable function in the cluster using the configuration of the cluster, which is available in the shared directory. The execution goes through three phases: The setup of the execution cluster, the preparation of the iPython cluster and the execution of the distributed classifier.

The cluster is defined by means of a Kubernetes application available in a GitHub repository².

²https://github.com/eubr-atmosphere/d42_applications/tree/master/integrated

This application uses a set of Docker containers that automate the process of setting up the cluster. The base containers is available in Docker Hub³ under the organization eubraatmosphere, the name eubraatmosphere_autobuild and the tag ubl6_ssfsf11-client, which run the command ipengine using a shared directory that stores the ipcontroller-engine.json file needed for the setup of the cluster.

The setup of the cluster for the execution of the distributed tensorflow(Tensorflow, 2019) is performed through the definition of a cluster context that defines three types of actors: master, parameter server and worker. Master process manages the execution, Workers are the nodes that perform the most computation intensive work and Parameter Servers are resources that store the variables needed by the workers, such as the weights variables needed for the networks. Figure 4 shows the commands required for building up context information to run the applications.

Once the cluster is setup and the configuration is obtained, the ipyparallel module enables running the same python function over several nodes in the cluster. This eases the management and monitoring of the execution of a distributed tensorflow application, as it provides a centralised console to follow up the status of the processing. With the combination of Jupyter notebook, the user can even retrieve the model in an object and use it for error evaluation and prediction on an interactive console. Figure 5 show the code needed for the execution.

The last step is the use of GPU accelerators for the computation intensive training. For this purpose, GPUs connected through PCI Pass through to the Virtual Machines. Containers mount the device directly from the Virtual Machine, and the matching between resources and containers is performed through the Kubernetes affinity.

4 CONCLUSIONS

The paper shows the architecture of a container-based medical application that leverages the use of federated infrastructures to gather computing-intensive resources with secure storage. The application is designed with the components provided by the ATMOSPHERE platform.

The applications benefit from ATMOSPHERE in several senses:

- Provide a simplified scenario to train and build the models, capable of managing parallel computing

³<https://hub.docker.com/>

```

In [1]: import ipyparallel as ip
c=ip.Client('/mnt/.ipython/profile_default/security/ipcontroller-client.json')
c.ids

Out[1]: [0, 1, 2, 3]

In [101]: dview = c[:]
dview

Out[101]: <DirectView [9, 11, 13, ...]>

In [3]: def ip():
import socket
return socket.gethostbyname(socket.gethostname())

In [5]: ips=c[:].apply_sync(ip)
ips

Out[5]: ['10.243.1.108', '10.243.1.109', '10.243.2.130', '10.243.2.129']

In [9]: NUMGPUS=0
DATAPATH="/mnt/share/vol003/cifar-10-data"
MODELPATH="/mnt/share/vol003/model_dir/"
TRAINSTEPS=100
EXECPATH="/mnt/share/vol003/DistributedTensorFlowCodeForIM/resnet"

NOTEBOOK=ip()
PSLIST=ips[0]
WNLIST0=ips[1]
WNLIST1=ips[2]
WNLIST2=ips[3]

In [10]: NOTEBOOK

Out[10]: '10.243.1.110'

In [11]: clusterspec={"ps": [ PSLIST+":8000" ],"worker": [WNLIST0+":8000", WNLIST1 + ":8000", WNLIST2 +
clusterspec

Out[11]: {'ps': ['10.243.1.108:8000'],
'worker': ['10.243.1.109:8000', '10.243.2.130:8000', '10.243.2.129:8000']}

In [12]: import tensorflow as tf
import keras

cluster = tf.train.ClusterSpec(clusterspec)
cluster
server = tf.train.Server(cluster, job_name="ps", task_index=0)

/opt/conda/lib/python3.6/site-packages/h5py/_init_.py:36: FutureWarning: Conversion of the
second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will
be treated as `np.float64 == np.dtype(float).type`.
from .conv import register_converters as _register_converters
Using TensorFlow backend.

In [72]: def distribkeras(myclusterspec,myjob_name,mytask_index):

```

Figure 4: Preparation of the cluster.

```

In [73]: c1=c[9].apply_async(distribkeras, clusterspec, "ps", 0)
c2=c[10].apply_async(distribkeras, clusterspec, "worker", 0)
c3=c[11].apply_async(distribkeras, clusterspec, "worker", 1)
c4=c[12].apply_async(distribkeras, clusterspec, "worker", 2)

In [19]: def funtest(a):
import socket
ip = socket.gethostbyname(socket.gethostname())
return ip+a

In [95]: c[:].shutdown()

In [69]: myjob_name="worker"
mytask_index=1
logdir="/mnt/share/vol002/train_logs/"+myjob_name+str(mytask_index)
logdir

Out[69]: '/mnt/share/vol002/train_logs/worker1'

```

Figure 5: Execution of Distributed Keras through the cluster.

resources. Model training requires intensive computing resources to be provisioned. Data scientists should not be charged with the burden of setting up this environment, and despite that there are several solutions in public clouds, there is no solution as flexible as ATMOSPHERE is.

- Provide a seamless transition from model building to production. ATMOSPHERE provides a development scenario that supports building models and publishing them as web services which can be called safely for obtaining a diagnosis on an image.
- To be able to work seamlessly in an intercontinental federated infrastructure, without having to specify geographical boundaries and trusting in the cloud services to select them according to restrictions in data or performance.
- To securely access and process data with the guarantees that neither the data owner can have access to the processing code nor the application developer can retrieve the data out of the system.

Clinical measures on the outcomes of RHD cases, before and after the application of these approaches, will allow an assessment of the results and compare them with the expected benefits.

ACKNOWLEDGEMENTS

The work in this article has been co-funded by project ATMOSPHERE, funded jointly by the European Commission under the Cooperation Programme, Horizon 2020 grant agreement No 777154 and the Brazilian Ministério de Ciência, Tecnologia e Inovação (MCTI), number 51119.

The authors also want to acknowledge the research grant from the regional government of the Comunitat Valenciana (Spain), co-funded by the European Union ERDF funds (European Regional Development Fund) of the Comunitat Valenciana 2014-2020, with reference IDIFEDER/2018/032 (High-Performance Algorithms for the Modelling, Simulation and early Detection of diseases in Personalized Medicine).

REFERENCES

- (2019). Kubernetes web site. <https://kubernetes.io>. Accessed: 29-12-2018.
- Arnautov, S., Brito, A., Felber, P., Fetzer, C., Gregor, F., Krahn, R., Ozga, W., Martin, A., Schiavoni, V., Silva, F., Tenorio, M., and Thümmel, N. (2018).

Pubsub-sgx: Exploiting trusted execution environments for privacy-preserving publish/subscribe systems. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 123–132.

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O’Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. (2016). SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA. USENIX Association.

Brasileiro, F., Vivas, J. L., d. Silva, G. F., Lezzi, D., Diaz, C., Badia, R. M., Caballer, M., and Blanquer, I. (2016). Flexible federation of cloud providers: The eubrazil cloud connect approach. In *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 165–170.

Caballer, M., Blanquer, I., Moltó, G., and de Alfonso, C. (2015). Dynamic Management of Virtual Infrastructures. *Journal of Grid Computing*, 13(1):53–70.

Calatrava, A., Romero, E., Moltó, G., Caballer, M., and Alonso, J. M. (2016). Self-managed cost-efficient virtual elastic clusters on hybrid Cloud infrastructures. *Future Generation Computer Systems*, 61:13–25.

de Alfonso, C., Caballer, M., Alvarruiz, F., and Hernández, V. (2013). An energy management system for cluster infrastructures. *Computers & Electrical Engineering*, 39(8):2579 – 2590.

Tensorflow (2019). Distributed tensorflow. <https://github.com/tensorflow/examples/blob/master/community/en/docs/deploy/distributed.md>. Accessed: 28-2-2019.