Reference Architecture for IoT Platforms towards Cloud Continuum based on Apache Kafka and Orchestration Methods

Zoltán Farkas[a] and Róbert Lovas[b]

Institute for Computer Science and Control (SZTAKI), Eötvös Loránd Research Network (ELKH),
Kende u. 13-17, Budapest, 1111, Hungary

Keywords: Cloud, IoT, Apache Kafka, Orchestration, Cloud Continuum, OpenStack, Azure, AWS.

Abstract: Apache Kafka is a widely used, distributed, open-source event streaming platform, which is available as a basic reference architecture for IoT use cases of the Autonomous Systems National Laboratory and other initiatives in Hungary, e.g. related to development of cyber-medical systems. This reference architecture offers a base for setting up a multi-node Kafka cluster on a Hungarian research infrastructure, ELKH Cloud. However, the capacity, accessibility or the availability of a given deployment using a single data center might not be sufficient. In this case Apache Kafka can be extended with additional nodes provisioned in the given cloud, but our solution also enables the expansion of the cluster by involving other cloud providers. In this paper we present our proposed approach for enhancing the existing basic reference architecture towards cloud continuum, i.e. allowing the supported IoT use cases to expand the resources of an already deployed Apache Kafka cluster with resources allocated even in third-party commercial cloud providers, such as Microsoft Azure and AWS leveraging on the functionalities of the Occopus cloud orchestrator.

1 INTRODUCTION

Cloud continuum reflects not only the rapidly growing penetration of cloud computing technologies but also the diversity of various use cases that requires (among others) agility to bring clouds to a wide range of users by expanding the cloud technologies towards edge, fog and cloud robotics. In Hungary, the use cases (Fényes et al., 2021) defined by the Autonomous Systems National Laboratory (ARNL, 2021) combined with the ELKH science cloud (ELKH, 2021) and other commercial providers might be considered as an illustrative example for such cloud continuum related efforts. Our institute, as the leader of this national lab and the ELKH Cloud research infrastructure, took the opportunity and has been gaining the expertise as well as the best practices in order to provide the first set of enhanced reference architectures accessible and seamlessly deployable by the national lab members.

The OpenStack-based ELKH Cloud offers a platform for enabling and accelerating AI-related, industrial IoT related and other current research activities. One of the reference architectures offered is Apache Kafka (Kafka, 2021b; Kreps et al., 2011), a widely used event streaming platform, which can be applied as the basis for many different applications, including the new developments of cyber-medical systems (Eigner et al., 2020).

The base platform for the Apache Kafka reference architecture is the ELKH Cloud (ELKH, 2021). In certain cases it may be possible to tune the performance of Kafka for the application (Le Noac'h et al., 2017), however it can happen that due to some reason the capacity allocated for the given project inside the ELKH Cloud is not enough to a larger Kafka cluster, or geographical distribution of the Kafka cluster is necessary (Kato et al., 2018).

In this paper we present our work to enable the deployment of the Kafka reference architecture onto cloud infrastructures beside the ELKH Cloud, as well. Our work leveraged on the Occopus orchestrator (Kovács and Kacsuk, 2018), which is capable of accessing APIs of multiple cloud providers (like OpenStack, OpenNebula (Kumar et al., 2014), AWS, Azure), thus allows the building of hybrid cloud infrastructures (Lovas et al., 2018; Kovács et al., 2018) in a seamless way. Occopus is also the integrated part of the MiCADO-Edge framework (Ullah et al., 2021) addressing the Cloud-to-Edge computing continuum.

[a] https://orcid.org/0000-0003-4459-9548
[b] https://orcid.org/0000-0001-9409-2855

205

The structure of the paper is as follows: first we give an overview on the available achievements related to the deployment of Kafka clusters in hybrid cloud environments. Afterwards, in order to introduce the existing reference architecture, we present both a virtual machine-based and a container-based version. Next, we compare the different possibilities to create a hybrid cloud variant of the reference architecture, and present two implementations, one using virtual machines in Azure, and one using Azure Container Instances (ACI). Afterwards, the startup times and the message throughput performance will be evaluated on the hybrid infrastructure applying the different technologies. Finally, we conclude our results, and highlight the possible future work.

All of the infrastructure descriptions, node definitions and different log files mentioned in this paper are available for download, examination and reproducibility in a Gitlab repository dedicated for this paper (Farkas, 2021).

2 RELATED WORK

As Kafka is a distributed platform, a number of solutions to make it work on multiple cloud providers already exist. In article (Waehner, 2021) authors describe the implementation a solution where existing legacy (non-cloud) application deployments can be modernized by involving a geographically distributed Kafka cluster to extend the infrastructure serving the application with resources coming from multiple cloud providers.

When considering fresh Kafka deployments, multiple proprietary solutions are available. For example, Canonical offers the deployment of a distributed Kafka cluster onto various cloud providers (Canonical, 2021). Confluent Cloud also offers the possibility to deploy a distributed Kafka cluster onto multiple cloud resources (Confluent, 2021)(du Preez, 2021).

Authors of (Torres et al., 2021) show an open-source framework for serving AI applications over cloud and edge clusters. The framework uses Kafka-ML (Martín et al., 2020), which allows the management of machine learning and AI pipelines through the data streams of Kafka. The authors evaluate the performance of the hybrid cloud-edge architecture in different scenarios, using on-premise, IoT and Google Cloud Platform (GCP) resources.

Author of (Sabharwal, 2016) shows the deployment of a Kafka clusters in a hybrid cloud environment, mixing on-premise and publicly available cloud resources. The article shows how Kafka MirrorMaker (Kafka, 2021a) can be used to replicate existing Kafka data onto a hybrid environment.

Although most of the related work shown here can be used as the basis for building hybrid Kafka clusters, the solution presented in this paper offers a user-friendly and convenient way to build and manage such a cluster. Its most important advantage is the encapsulation of the solution and best practices into a reference architecture description that might be the subject of an orchestrator tool - Occopus - for building or extending the hybrid Kafka cluster infrastructure in a highly automatized way.

3 APACHE KAFKA REFERENCE ARCHITECTURE FOR ELKH CLOUD USERS

Users of ELKH Cloud and supported researchers of the Autonomous Systems National Laboratory can access different sets of so called reference architectures (RAs). RAs can be used as a base to build an infrastructure in the cloud for supporting a given a computation. Examples for such RAs are the Docker Swarm cluster (Kovács et al., 2018), the Apache Spark RA with Hadoop (Lovas et al., 2018), a Kubernetes cluster, a Tensorflow-Keras-Jupyter stack or the Apache Kafka RA.

The different reference architectures can be built up in the ELKH Cloud with the help of a cloud orchestrator tool, like Terraform or Occopus. The advantage of using these tools is that the users do not need to work with the APIs of the underlying cloud providers; they simply have to specify the necessary credentials and instruct the orchestrator tool to build the infrastructure in the target cloud.

The Apache Kafka RA offered is implemented with the help of Occopus. Occopus, as a cloud orchestrator can interface with a number of cloud resources: AWS, Azure, OpenStack- or OpenNebula-based clouds. The connection with the different target clouds is implemented through different plugins, which provide different set of functionalities, like starting virtual machines with a contextualization script, or to start containers based on user-defined images.

Infrastructures created with Occopus consist of two parts: an infrastructure description and a set of node definition, all written in YAML format. The infrastructure description describes the nodes of the infrastructure, and also defines deployment dependencies between the nodes. On the other hand, node definitions describe the deployment information of the different nodes in the different target clouds. As

the ELKH Cloud is based on OpenStack, the Apache Kafka RA is implemented with the help of the `nova` plugin of Occopus.

The structure of the infrastructure created by the Apache Kafka RA can be seen in Figure 1. As it can be seen in the figure, the user is interacting with Occopus in order to provision the infrastructure.
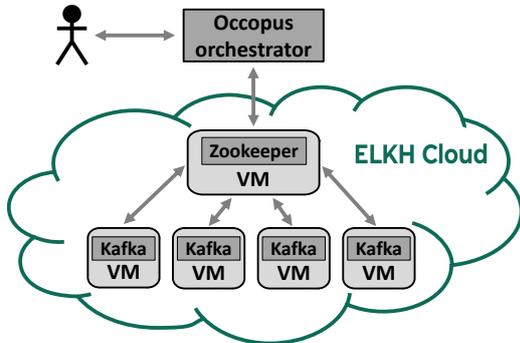

Figure 1: Apache Kafka RA in the ELKH Cloud.

The Apache Kafka RA consists of two types of nodes: a Zookeeper node, and a set of Kafka broker nodes. Zookeeper in this setup is used to keep track of the status of the Kafka cluster nodes and to keep track of Kafka topics and partitions. The infrastructure description of the Apache Kafka RA can be found in the Gitlab repository (Farkas, 2021), in the directory `kafka.vm`, called as `infra-kafka.yaml`). The description is self-explanatory, but details can be found in the Occopus documentation. There are two nodes called as `zookeeper` and `kafka`, where the `kafka` node must have at least 2, and at most 10 instances. The `kafka` nodes depend on the `zookeeper` node, thus Occopus will create the `zookeeper` node first. The respective node definitions are called `zookeeper_node` and `kafka_node`. We are not placing the node definitions and the related contextualization scripts in the paper, as they are quite long. In a nutshell, the node definitions describe the necessary properties to create an adequate VM for the nodes in the OpenStack-based ELKH Cloud, and also define the different health-checking methods (TCP ports 2181 and 9092 should accept connections for the `zookeeper` and `kafka` nodes, respectively). The contextualization scripts of the different nodes are cloudinit scripts, which deploy, configure, and start the given software components on the nodes. In case of the `kafka` nodes, the IP address of the `zookeeper` node is also used in the contextualization script, so that Kafka brokers know where to look for the Zookeeper service.

The logs for starting up the infrastructure, scaling the infrastructure, and destroying the infrastructure can be found in the log files `01_build_elkh.log`, `02_scale_elkh.log` and `04_destroy_infra.log` respectively, in the directory `kafka.vm/logs`.

The deployed Kafka cluster can be examined through different user interfaces for Kafka. We choose Kafka UI (UI, 2021), as it can be set up quickly through Docker, and is capable of giving a quick overview of the cluster. Figure 2 shows the scaled up cluster's overview. Once a producer is connected to the Kafka cluster, we can examine the messages produced, too.
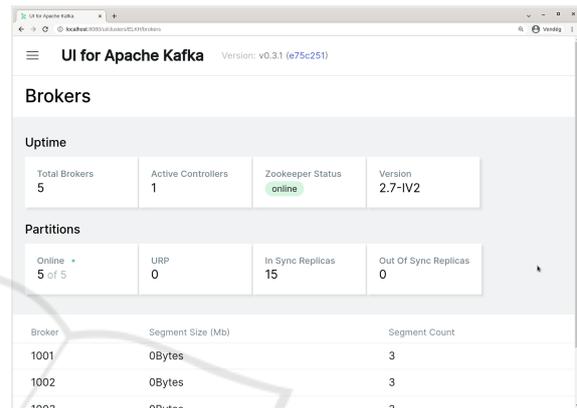

Figure 2: Kafka UI for the cluster in ELKH Cloud.

3.1 Containerized Version of the Reference Architecture

Although the deployment of Kafka services onto VM is portable thanks to the cloudinit script used, applying container images to run the services offers an even more portable way to bring up the architecture. Basically, any containerization environment (like Docker or Kubernetes) can use them. In case of the ELKH Cloud there is no native container support yet, so in order to bring up the containerized version of the Apache Kafka reference architecture, we have to start virtual machines on the ELKH Cloud, make sure Docker is deployed on them, and finally start up the proper container images on the VM hosts.

One problem related to the networking of Kafka can arise in this case: the advertised listener addresses may be unreachable from other containers. Thus, in this case we need to make sure that properly configured listeners are set up for the given Kafka service. Apache Kafka has the following configuration properties related to its listeners:

- `listeners`: enumerates the different listeners with a given name, for example `EXTERNAL://:9092`,
- `advertised.listeners`: enumerates the

listeners the given Kafka node will advertise towards it connected clients, for example `EXTERNAL://193.215.220.107:9092`,

- `inter.broker.listener.name`: the name of the listener to be used for inter-broker communication, for example `EXTERNAL`.

In case of the containerized deployment, if we do not define the virtual machine's IP address as the IP address of the listeners, then the brokers will propagate the IP address assigned to the Docker container, which is not accessible from the other VMs, as a consequence, the brokers will not be able to communicate with each other.

In order to overcome this issue, in the containerized deployment, for each Kafka service running inside the container, we are specifying only one listener (called as `EXTERNAL`), and set its advertised address to include the IP address of the VM running the container. This listener is also configured to be used as the inter-broker listener. Figure 3 shows this architecture.
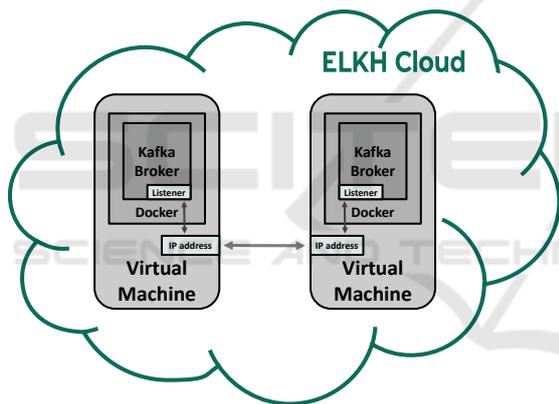

Figure 3: Listeners and IP addresses in the containerized architecture.

The containerized version of the reference architecture is available in the Gitlab repository (Farkas, 2021), in the `kafka.container` directory, contains both the infrastructure descriptions and the node definitions, and also includes logs files of starting up and destroying the infrastructures.

4 ENABLING HYBRID CLOUD OPERATION FOR THE REFERENCE ARCHITECTURE

As mentioned in the previous section, Occopus is using two descriptions for an infrastructure: the infrastructure description itself, and the node definitions for each node participating in the infrastructure. The infrastructure description is a resource-independent YAML file, whereas the node definitions are the YAML files which actually contain the cloud-specific provisioning information for the different node.

In Occopus, one node may have multiple node definitions, for example the Kafka node in the Apache Kafka RA can have one definition for the OpenStack-based ELKH Cloud, and one definition for Azure. In this case, when Occopus is about to bring up an instance of the Kafka node in the RA, then it will decide randomly which node definition to use for the given instance.

It follows from the above, that we have two options for creating the hybrid version of the Apache Kafka RA:

1. leave the infrastructure description unmodified, but add a node definition for the Kafka node to use another cloud resource,

2. extend the infrastructure description with an additional Kafka node, which also depends on the Zookeeper node, and has a new node definition for some other cloud resource.

Figure 4 shows the differences between two approaches. The top part of the figure shows the structures of the different infrastructures. The left hand case shows where we have only one Kafka node, but it has multiple node definitions: one for the ELKH Cloud, and one for Azure. The case shown on the right side shows that we have two different Kafka nodes in the infrastructure, one for the ELKH Cloud (with a node definition belonging to the ELKH Cloud), and one for Azure (with the relevant node definition attached).
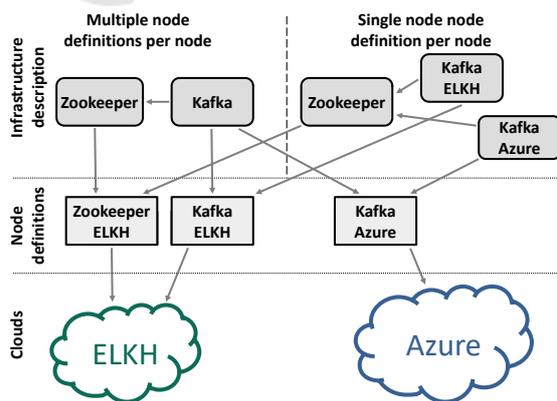

Figure 4: Approaches for creating hybrid infrastructures.

As it was mentioned earlier, in case of the unmodified infrastructure description extension method,

Occopus will decide randomly between node definitions when provisioning new node instances, thus we propose to use the updated infrastructure description method instead. In this case, the user can explicitly define how many Kafka nodes should be created in the ELKH Cloud, or in Azure, or in any other cloud provider for which a Kafka node definition exists.

The updated infrastructure description providing Kafka nodes in both the ELKH Cloud and Azure can be found in the Gitlab repository for the paper (Farkas, 2021), in the directory `kafka.vm`, the file is called `infra-kafka-hybrid.yaml`. This infrastructure description shows the structure of the infrastructure: one Zookeeper node is used, and there are two Kafka nodes, one (as earlier) for the ELKH Cloud, and one for Azure. Both ELKH Cloud and Azure variants should have at least 2, at most 10 instances when scaling the nodes.

If an ELKH Cloud user has already created an infrastructure based on the ELKH Cloud-only Apache Kafka reference architecture, then Occopus enables the extension of the available infrastructure with the new additional Azure-based Kafka nodes. The user simply has to instruct Occopus to build an infrastructure, based on the existing one, and using the new infrastructure description. Occopus will check if the nodes of the existing infrastructure are still alive (if not, will provision new ones), and will also create the Azure-based Kafka node instances.

The node definition part belonging to the updated, hybrid infrastructure depends on the target cloud used. In the following subsections we will examine how the extension can be implemented in Azure using virtual machines and in Azure using container instances. Occopus, beside others, has support for handling these cloud types.

4.1 Azure VM-based Kafka Nodes

Azure-based virtual machines in Occopus are handled by the `azure_vm` resource handler plugin. Through this plugin users can instantiate virtual machines in the different regions of Azure, cloudinit scripts are available for contextualization, just like in case of the OpenStack-based ELKH Cloud. The Occopus documentation provides detailed instructions on how to prepare a node definition for the `azure_vm` plugin, so we are only enumerating the necessary steps in a nutshell here:

- set authentication data to be used with Azure (using Azure service principal),
- set node properties, like region, virtual machine size, publisher,
- create contextualization script for the service.

As it was already mentioned, the Occopus documentation covers most cases. Also, as Azure provides cloudinit-based contextualization, very likely the contextualization script used for ELKH Cloud-based nodes can also be used for Azure.

However, a slight modification is necessary. As shown in Figure 3, the Kafka nodes are communicating with each other. As the ELKH Cloud-based instances, and the Azure-based instances are running on completely different networks, the different nodes in this hybrid scenario must have a public IP assigned, and this public IP address should be used as the advertised listener, and the listener used for inter-broker communication, too. Most cloud providers offer a way to query the public IP address associated to the virtual machine from inside the virtual machine, but this method differs for the clouds. Thus, the contextualization script of the Azure-based instances will differ from the ELKH Cloud-based ones in this manner: for querying the public IP address of the VM, the Azure Instance Metadata Service(IMDS, 2021) is used.

With the updated cloudinit script, we can start to extend the existing infrastructure in the ELKH Cloud with Kafka nodes coming from Azure, using the `azure_vm` plugin. The output of the Occopus command to extend the infrastructure can be found in the Gitlab repository's `kafka.vm` directory, in the file `logs/03_extend_hybrid.log`.

After the extension of the infrastructure, we will have one Zookeeper node running in the ELKH Cloud, 5 Kafka nodes running in the ELKH Cloud, and 2 additional Kafka nodes running in Azure.

At this point, we can start to attach new producers and consumers to the extended cluster. They can connect either to the ELKH Cloud-based or Azure-based nodes, but it is important to note, that according to the Kafka documentation new servers will not automatically be assigned any data partitions, so unless partitions are moved to them they will not be doing any work until new topics are created.

4.2 Azure Container-based Kafka Nodes

Containers offer a simple solution for running prepackaged applications in versatile environments. As it was mentioned earlier, ELKH Cloud does not provide a native service for running containers, but users are required to deploy some sort of container service (for example Docker Engine or Kubernetes) onto virtual machines, and run containers on top of that service.

Azure includes a service called Azure Container Instances (ACI), which enables Azure users to start

up containers without the need to allocate any hosting virtual machines beforehand. The functionality is straightforward and simplified: the user has to specify some basic container properties, like vCPU, RAM and GPU requirements, base image, networking requirements, and ACI will make sure that a container based on the specified image is started up, without the need to allocate VMs. This enables quicker service startups and easier migration when needed.

Occopus has support for exploiting the functionalities of ACI through the `azure_aci` plugin. In order to start container instances in Azure with Occopus, the user has to set the Azure credentials, and the basic properties for the container to be started. Features like specifying environment variables and the command to be run inside the container are also supported.

Migrating the existing reference architecture nodes to ACI is relatively easy, as the container image is already available. The only problem is that in order to set the advertised listening address for the service, we need to query the public IP address as described in the previous, VM-based deployment. In case of ACI, we do not have a service similar to the Azure Instance Metadata Service, so we cannot get the public IP address of the container from inside the container (for example, before the startup of the Kafka service). However, when allocating an ACI container, Occopus has the possibility to generate a base FQDN for the container's public IP address, through which other service can access it. And this functionality can be used to pass the container's pre-allocated public FQDN as an environment variable to the container, so the Kafka listener addresses can be configured when starting up the service. The dedicated environment variable is called `_OCCOPUS_ALLOCATED_FQDN`. The startup script of the existing Kafka container image has been updated to check if this environment variable is set, and if yes, then its value is used as the external advertised listener address for the Kafka service.

With these updates, ACI-based Kafka nodes can also participate in a hybrid cluster. The ACI-based hybrid infrastructure can be found in the Gitlab repository's `kafka.container` directory. The log file `logs/01_build_hybrid_aci.log` shows the output of the Occopus infrastructure build command for the `infra-kafka-hybrid-aci.yaml` infrastructure description. This contains an ELKH Cloud-based Zookeeper node, 2 ELKH Cloud-based Kafka nodes (the Kafka service is run inside Docker containers), and 2 Azure ACI-based Kafka nodes. Figure 5 shows the Kafka UI for the hybrid infrastructure, after 2,5 million short messages have been sent to a message topic with the replication factor of 4.
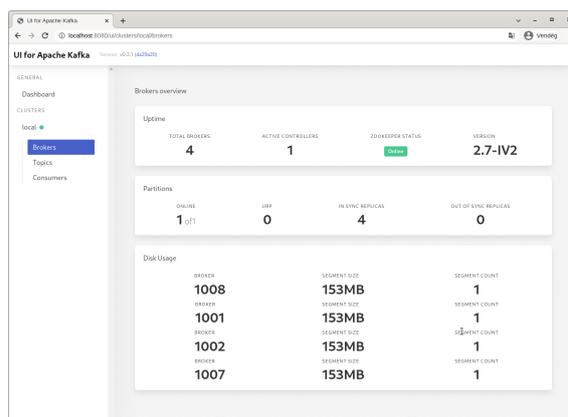

Figure 5: Kafka UI overview for hybrid cluster using containers.

5 PERFORMANCE EVALUATION

In this section of the paper we evaluate the performance of a hybrid Kafka cluster deployment, running Kafka brokers both in the ELKH Cloud and in AWS. AWS was chosen to show that the presented hybrid approach is not tied only to the ELKH Cloud and Azure, but can also be adapted to AWS as well.

A number of papers discuss the performance evaluation of Apache Kafka clusters.

Authors of (Wu et al., 2019) propose a queuing-based model to predict the performance of an Apache Kafka cluster. The metrics the authors are able to predict with very high accuracy based on their measurements includes message throughput (both for Kafka producers and consumers) and end-to-end latency (for real-time applications).

In paper (Hesse et al., 2020) authors present a monitoring framework for examining different performance metrics of Java Virtual Machine-based message broker systems, like Apache Kafka. Although the paper uses an evaluation in scenarios with only one Kafka node, but the different results, like the need to rely on multiple producers are applied in this paper as well.

We already referenced paper (Le Noac'h et al., 2017), in which authors overview how changing different parameters (like message size, batch size, replication properties for a topic) impacts different metrics of the Kafka cluster (like the number of messages accepted per seconds, or the message accepting latency). In our paper we do not have the possibility to evaluate the overall hybrid performance from all these aspects, but as we will show later, we are using a given configuration throughout the evaluation.

Beyond performance metrics, there are other aspects which can be examined while evaluating an

Apache Kafka cluster, like network fault tolerance (Raza et al., 2021). However, these are out of the scope of our paper, so we are not discussing them.

The Kafka cluster used for the evaluation was set up based on the following nodes: one Zookeeper node in the ELKH Cloud (2 vCPUs, 4 GB RAM, 100 GB SSD storage with 300 MB/s throughput), two Kafka nodes also in the ELKH Cloud (2 vCPUs, 4 GB RAM, 2 TB SSD storage with 300 MB/s throughput for each node), and two Kafka nodes in AWS (`c5ad.large` instance type - 2 vCPUs, 4 GB RAM, 400 GB gp3 volume attached for Kafka with 300 MB/s throughput for each node). Every node had a public IP address allocated, and the brokers were using their public IP as the advertised listener. The version of Kafka used was 3.0.0, without any special fine-tuning in the default configuration (except for the listener configuration to use the public IP address).

To monitor the different metrics of the Kafka cluster, we have set up a Grafana deployment with Prometheus as described in an article (Ahuja, 2022). The core of the monitoring solution is a JMX agent for Prometheus, which is able to query Kafka broker-specific metrics.

We have created the following topics for the evaluation (for each topic, the replication factor is 1, and the number of minimum in sync replicas is also 1):

- `ELKH`: this topic was set up with 2 partitions allocated to the two ELKH Cloud brokers,

- `AWS`: this topic was set up with 2 partitions allocated to the two AWS brokers,

- `Hybrid`: this topic was set up with 4 partitions allocated to the four available brokers (two from ELKH Cloud and two from AWS).

For the message throughput evaluation, we ran the stock `kafka-producer-perf-test.sh` script available in the Kafka distribution. The common configuration for the producers was to publish 500M messages with 100 random bytes each, not limiting the throughput from the producer side, have 64 MB of memory for buffering messages, and use a batch size of 64000 (pack at most so many messages into one request). The variable configuration for the producers was the topic to send the messages to, and the set of bootstrap Kafka brokers. During testing, we ran multiple producers in parallel, each producer had 2 vCPUS and 4 GB of RAM available, producers running in the ELKH Cloud were using the ELKH Cloud-based brokers as the bootstrap brokers (using their private IP addresses), whereas producers running in AWS were using the AWS brokers as the bootstrap brokers (using their public IP addresses).

The producers were started with the help of Occo-

pus, which has the feature to start independent infrastructure nodes in parallel. Each node of the infrastructure description were above-described producer nodes, either run on the ELKH Cloud or AWS. All the infrastructure descriptions, the measurements and figures are available in the Gitlab repository of the paper (Farkas, 2021), in the directory `performance`.

5.1 Evaluation 1: Maximum Throughput of the ELKH Cloud-based Brokers

In this scenario, we were interested in the maximum message throughput of the ELKH Cloud-based brokers. In order to measure this, we have started an increasing number of producers in parallel in the ELKH Cloud to produce messages to the `ELKH` topic. Table 1 summarizes the results.

Table 1: ELKH Cloud-based broker message throughput.

Number of producers	Max message throughput (messages/s)
6	2.861.774
12	3.540.749
18	3.458.745

The graphs for the monitored message throughput for the 6, 12 and 18 producer cases can be seen in Figures 6, 7 and 8, respectively.

Figure 6: 6 ELKH Cloud-based producers.

Figure 7: 12 ELKH Cloud-based producers.

We can state that the maximum throughput of the two ELKH Cloud-based brokers, when messages are produced by local producers is around 3,5 million messages/seconds. As both the brokers, topics and producers used were running on the ELKH Cloud, this is the stand-alone performance of the ELKH Cloud-based setup.

Figure 8: 18 ELKH Cloud-based producers.

5.2 Evaluation 2/a: Hybrid Setup, Per-data Center Topics

In this scenario, we have utilized the AWS-based brokers, as well. We were running multiple producers both in the ELKH Cloud and AWS, where ELKH Cloud-based producers were using the ELKH Cloud-based brokers as the bootstrap servers, and were sending messages to the ELKH topic, while the AWS-based producers were connecting to the AWS-based brokers, and were sending messages to the AWS topic. Thus, the cluster was set up in a hybrid manner, but the topic configuration allows us to stay inside a data center when producing messages. Later of course the topics can be mirrored to the other brokers, but now we were interested in the message throughput capacity of the setup. Table 2 summarizes the results.

Table 2: ELKH Cloud+AWS broker message throughput.

Producers	ELKH	AWS	Aggregated
6-6	2.753.677	2.691.826	5.431.932
12-12	3.446.712	3.056.977	6.477.289

As it can be seen, aggregated message throughput of the four brokers is close to the double of the message throughput of two brokers as show in Table 1. The message throughput of the different topics for the 6-6 and 12-12 producer cases is shown in Figures 9 and 10, respectively.
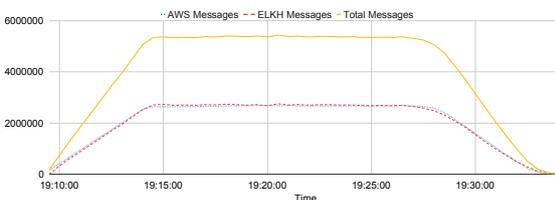

Figure 9: 6 ELKH Cloud- and 6 AWS-based producers.

5.3 Evaluation 2/b: Hybrid Setup, One Topic

This scenario is similar to the previous one, but instead of using dedicated topics in the different data centers, we have used one topic called Hybrid, which
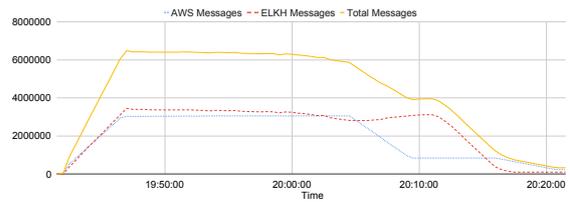
had four partitions allocated to the four brokers participating in the cluster. Our expectation was, that due to the communication requirements between the brokers, the message throughput will decrease in this case.

Similarly to the previous case, we have started 6-6 and 12-12 producers, all of the producers sending messages to topic Hybrid, the ELKH Cloud-based producers connecting to the ELKH Cloud-based brokers as the bootstrap servers, while the AWS-based producers connecting to the AWS-based brokers as bootstrap servers.

Table 3 shows our measurements, while Figures 11 and 12 show the message throughput graph as displayed by Grafana during the experiments.

Table 3: Hybrid topic throughput.

Number of producers	Max message throughput (messages/s)
6-6	899.013
12-12	1.964.962

Figure 11: 6 ELKH Cloud- and 6 AWS-based producers.

Figure 12: 12 ELKH Cloud- and 12 AWS-based producers.

5.4 Evaluation Summary

As it can be seen from the results of Evaluation 2/a and 2/b, that in case of a hybrid setup, the partitioning of topics has a notable impact on the message throughput. If we are about to increase the perfor-

mance of a Kafka cluster with new brokers from other data centers, then it is recommended to use new message topics partitioned only in the new data center, as in this case the overall message throughput is the summary of the local message throughput in the different data centers. If a topic is partitioned between different data centers, then we can expect performance degradation, which is very likely caused by the inter-broker communication.

6 SUMMARY AND FUTURE WORK

In this paper we presented our approach of enabling the hybrid cloud deployment of an Apache Kafka reference architecture, originally targeting the OpenStack-based ELKH science cloud. The presented work shows how the existing VM-based architecture can be moved onto a container-based solution, and how the hybrid operation is enabled in Azure using virtual machines or containers towards cloud continuum. We also performed a message throughput evaluation of the hybrid architecture, running on the ELKH Cloud and AWS resources.

The most important advantage of our approach is the encapsulation of the entire method and the related best practices into reference architecture descriptions that might be the subject of cloud orchestration in-line with the cloud continuum approach. For orchestration purposes the open-source Occopus tool has been chosen but the presented results could be applied for other higher level (Ullah et al., 2021) or third-party orchestration tools as well.

Regarding future work, the need of performing additional measurements of the hybrid infrastructures has been already identified. During the evaluation the focus was on the startup time of the infrastructures, and some basic message throughput characteristics, but from the applications' point of view additional metrics could also be considered, e.g. (i) how effectively consumers can process messaging sent to the different topics, or (ii) what is the performance of topic mirroring between different data centers in a hybrid setup. Following the results here, we aim at evaluating other performance metrics of different applications on hybrid Kafka clusters, running on top of VM- or container-based deployments, having connected the cluster nodes in different setups, such as connecting directly or through a Virtual Private Network connection. Further validation of results are to be performed with use case providers, involving experts from the field of autonomous and cyber-medical systems.

REFERENCES

Ahuja, M. (2022). Kafka monitoring using prometheus. https://www.metricfire.com/blog/kafka-monitoring-using-prometheus/. Accessed: 2022-01-10.

ARNL (2021). The national laboratory for autonomous systems website. https://autonom.nemzetilabor.hu/. Accessed: 2021-11-01.

Canonical (2021). Managed kafka on ubuntu. https://ubuntu.com/managed/apps/kafka. Accessed: 2021-12-01.

Confluent (2021). Confluent multi-cloud - the complete guide. https://www.confluent.io/learn/multi-cloud/. Accessed: 2021-12-01.

du Preez, D. (2021). Confluent aims to make it easier to set data in motion across hybrid and multi-cloud. https://diginomica.com/confluent-aims-make-it-easier-set-data-motion-across-hybrid-and-multi-cloud. Accessed: 2021-12-01.

Eigner, G., Nagy, M., and Kovács, L. (2020). Machine learning application development to predict blood glucose level based on real time patient data. In *2020 RIVF International Conference on Computing and Communication Technologies (RIVF)*, pages 1–6.

ELKH (2021). The elkh cloud website. https://science-cloud.hu/en. Accessed: 2021-11-01.

Farkas, Z. (2021). Gitlab repository for the paper. https://gitlab.com/zfarkas/iotbds-hybrid-kafka. Accessed: 2021-11-28.

Fényes, D., Németh, B., and Gáspár, P. (2021). A novel data-driven modeling and control design method for autonomous vehicles. *Energies*, 14(2).

213

Hesse, G., Matthies, C., and Uflacker, M. (2020). How fast can we insert? an empirical performance evaluation of apache kafka. *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*.

IMDS, A. (2021). Azure instance metadata service (linux). https://docs.microsoft.com/en-us/azure/virtual-machines/linux/instance-metadata-service?tabs=linux. Accessed: 2021-10-27.

Kafka (2021a). Mirroring data between clusters & geo-replication. https://kafka.apache.org/documentation/\#basic_ops_mirror_maker. Accessed: 2021-12-01.

Kafka, A. (2021b). Apache kafka. https://kafka.apache.org/. Accessed: 2021-11-01.

Kato, K., Takefusa, A., Nakada, H., and Oguchi, M. (2018). A study of a scalable distributed stream processing infrastructure using ray and apache kafka. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 5351–5353.

Kovács, J. and Kacsuk, P. (2018). Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures. *Journal of Grid Computing*, 16:1–19.

Kovács, J., Kacsuk, P., and Emődi, M. (2018). Deploying docker swarm cluster on hybrid clouds using occopus. *Advances in Engineering Software*, 125:136–145.

Kreps, J., Corp, L., Narkhede, N., Rao, J., and Corp, L. (2011). Kafka: a distributed messaging system for log processing. netdb'11.

Kumar, R., Adwani, L., Kumawat, S., and Jangir, S. (2014). Opennebula: Open source iaas cloud computing software platforms. In *National Conference on Computational and Mathematical Sciences (COMPUTATIA-IV)*.

Le Noac'h, P., Costan, A., and Bougé, L. (2017). A performance evaluation of apache kafka in support of big data streaming applications. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 4803–4806.

Lovas, R., Nagy, E., and Kovács, J. (2018). Cloud agnostic big data platform focusing on scalability and cost-efficiency. *Advances in Engineering Software*, 125.

Martín, C., Langendoerfer, P., Zarrin, P. S., Díaz, M., and Rubio, B. (2020). Kafka-ml: connecting the data stream with ml/ai frameworks.

Raza, M., Tahir, J., Doblander, C., Mayer, R., and Jacobsen, H.-A. (2021). Benchmarking apache kafka under network faults. In *Proceedings of the 22nd International Middleware Conference: Demos and Posters*, Middleware '21, page 5–7, New York, NY, USA. Association for Computing Machinery.

Sabharwal, N. (2016). Kafka mirroring in hybrid cloud. https://www.linkedin.com/pulse/kafka-mirroring-hybrid-cloud-introduction-neeraj-sabharwal/. Accessed: 2021-12-01.

Torres, D. R., Martín, C., Rubio, B., and Díaz, M. (2021). An open source framework based on kafka-ml for distributed dnn inference over the cloud-to-things continuum. *Journal of Systems Architecture*, 118:102214.

UI, K. (2021). Ui for apache kafka.

https://github.com/provectus/kafka-ui. Accessed: 2021-11-01.

Ullah, A., Dagdeviren, H., Ariyattu, R. C., DesLauriers, J., Kiss, T., and Bowden, J. (2021). Micado-edge: Towards an application-level orchestrator for the cloud-to-edge computing continuum. *Journal of Grid Computing*, 19(4):47.

Waehner, K. (2021). Bridge between edge / on premise and multi-cloud with apache kafka. https://kai-waehner.medium.com/bridge-between-edge-on-premise-and-multi-cloud-with-apache-kafka-e9196100fadf. Accessed: 2021-11-01.

Wu, H., Shang, Z., and Wolter, K. (2019). Performance prediction for the apache kafka messaging system. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 154–161.