VU VRIJE UNIVERSITEIT AMSTERDAM

Deploy and Benchmark an Edge Resource Manager in the cloud and locally

Authors: Tim Roos, Paul Püttbach, David Püroja, Marijn Vollaard, Ghasem Gholi
Emails: t.roos@student.vu.nl, p.puttbach@student.vu.nl, d.a.puroja@student.vu.nl, m.j.vollaard@student.vu.nl, g.mohammadgholi@student.vu.nl
VUnetIDs: 2649061, 2622550, 2518803, 2726662, 2553232
Supervisor: M.S. Jansen
Course Instructors: A. Iosup, S. Talluri, J. Donkervliet, A. Trivedi, T. Hegeman & M.S. Jansen

December 21, 2021

Contents

1 Abstract

Cloud computing is becoming more and more important in day to day life. This means that scaling of cloud computing is necessary to keep up with the daily demand. In this paper, we try to get an insight into the performance of such a Google cloud-based environment and compare this to a local environment. We achieve this by deploying and benchmarking an edge resource manager that is running on an image classifying application. With it, we will measure the application execution time, deployment overhead and internet delay of the environments and compare these. Furthermore, we aim to have more insight into the performance of Google cloud and learn more about cloud-based applications as a whole.

2 Introduction

Cloud computing offers on-demand computer system resources such as compute instances, such as storage of data with scaling options, without the need for active management of those resources. These resources can be theoretically ran in any data centre in the world. Another emerging form of computing is edge computing, where computation and the data are done close to the source of the data. The devices are commonly heterogeneous and with limited resources. A way of deploying applications to these resources is by using containerized applications. Containers bundle the application into one single software package with all the libraries and dependencies which is then run isolated from its environment. The first complete implementation of containerization was LXCC in 2008.Later different versions like Warden and LMCTFY emerged. In 2013, Docker managed to compete with prevailing systems by offering an ecosystem for container management. [2]
Although Docker automated a large part of the deployment, it did not do so fully, nor did it fully automate the scaling or management. To alleviate this issue, Google started developing Kubernetes which can be used in concert with Docker to achieve containerization and management of a system that is made of containerized applications. Kubernetes was largely developed with cloud computing in mind. KubeEdge aims to bring the functionalities of Kubernetes to the Edge. This project will explore KubeEdge, which is a resource system enabling containerized applications to host in edge devices. [5]

2.1 Research Questions

To properly investigate how to *"Deploy and Benchmark an Edge Resource Manager in the cloud and locally"* it is important to explain the deployment in the cloud and locally, and evaluate the performance of these deployments. To do this, the following sub-questions must be answered :
RQ1. How do we automate the deployment of a KubeEdge environment locally and in the cloud?
RQ2. How do we benchmark the application execution time in the edge environment for both cloud and local and what are their results?
RQ3. How do we benchmark the deployment overhead in the cloud and locally and what are their results?
RQ4. How do we benchmark the network delay in the edge environment for both cloud and local and what are their results.

2.2 Contributions

This paper makes the following contributions:
C1. We provide a short explanation of a typical distributed application.
C2. We provide source code to a local deployment and a cloud deployment with a description, both open source.[1]
C3. We provide benchmarks of the locally and cloud-deployed environment.

[1]Code and a copy of this report can be found at **https://github.com/GLaDAP/edge-computing-benchmark**

3 Background

For this paper, we extend an already existing KubeEdge deployment. The goal is to design, implement, and automate a benchmark for the KubeEdge application. The benchmark must at least contain the following metrics:

- Application execution time
- Deployment overhead
- Network delay

This section provides some of the required knowledge to understand this paper. It first covers what applications are used to benchmark the system. Then we try to explain what environments they are run in.

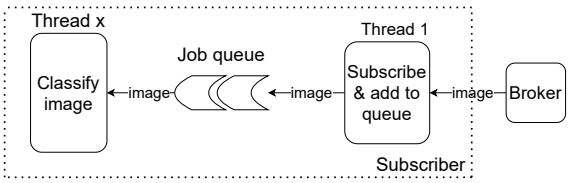

Figure 1: Subscriber overview.

3.1 Applications

The application we benchmark is an image identifier. It consists of a publisher, a subscriber, and a message broker that implements MQTT. The publisher and subscriber are called clients. The publisher and subscriber can be described as 2 different nodes, where a node can have the role of subscribing or publishing. A simple sequence sketch of this application can be seen in Figure 2. The publisher connects with a broker and sends the broker images marked with a certain topic. The broker receives this data and makes it available on that topic. It then pushes the data to all subscribers that subscribed to this topic. A subscriber will therefore eventually receive new data whenever that data becomes available at the broker for any of the topics the subscriber is subscribed to. Once this image is received, one thread from the subscriber used explicitly for receiving new images puts it in a queue where a different thread takes that image from the queue and performs image classification on it. This process is shown in Figure 1.

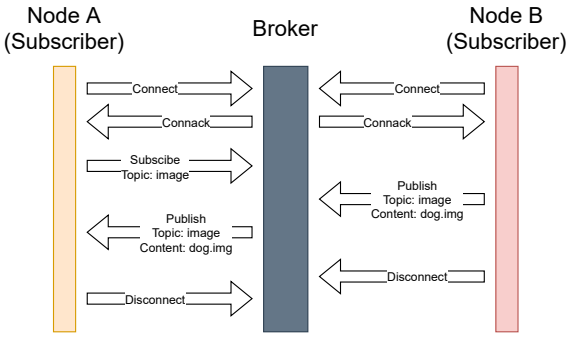

Figure 2: MQTT sequence overview.

The combined application is a bit simpler and skips the middleman of the broker. It iterates over a directory containing images and classifies them. In total there are 15 images within this directory.

3.2 Setup local environment

To set up our local environment, we use Vagrant with Virtualbox to create the virtual machines. Vagrant is a tool for building and managing virtual machines in a single workflow.[4] The virtual machines are then provisioned using Ansible playbooks to run the applications. Ansible automates the provisioning. First, the virtual machine for the cloudcore is set up, which consists of creating a Virtualbox VM and then executing the Ansible playbooks, which installs Kubernetes and KubeEdge cloudcore. A join token is then created and stored in a shared folder which is accessible by the EdgeCore nodes. After initialization of the n-number of edgecore nodes, the join token is copied into the VM from the shared folder when created. By executing the KubeEdge join command, the edgecore nodes are controlled by the cloudcore node. The cloudcore and Edgecore nodes initialization and development stack is visualized in figure 3.

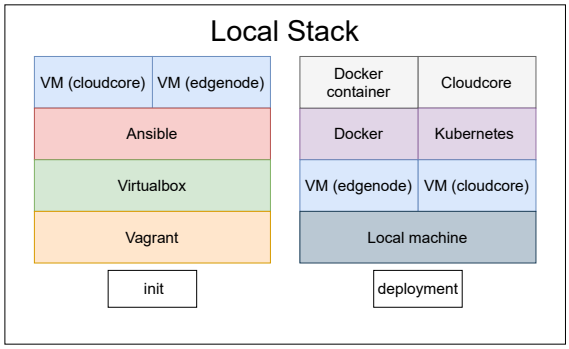

Figure 3: Local stack

The local test environment has the following specifications:

- **CPU:** AMD Ryzen 7 5800X
- **GPU** Gigabyte GeForce GTX 1070 G1 Gaming 8GB
- **Memory** Corsair platinum 4x4GB 2133MHz C15
- **Storage** Samsung SSD 950 PRO 512GB

3.3 Setup Cloud environment

For the cloud environment, Google Cloud is used as cloud provider where the virtual machines (compute instances), storage and network resources are deployed. Deployment of these resources is done using Terraform, an infrastructure-as-code software to manage and deploy cloud resources. The virtual machines are then provisioned using Ansible playbooks. In Google Cloud, a compute network together with a firewall is created where the virtual machines can communicate with each other. To provide the virtual machines with the Ansible playbooks, the playbooks are uploaded to a cloud storage, Google Cloud Storage Bucket, which is accessible by the virtual machines. At creation of the virtual machines, a shell script is executed by Terraform to copy the files from the storage bucket to the VM and to execute the Ansible playbooks. The VM for the cloudcore uploads the join-token to the storage bucket when the node is fully initialized. The Edgecore virtual machines waits until the token is present in the storage bucket, after which it executes the join command to join the cloudcore. The cloudcore and edgecore node initialization and development stack for the cloud setup is shown in Figure 4.

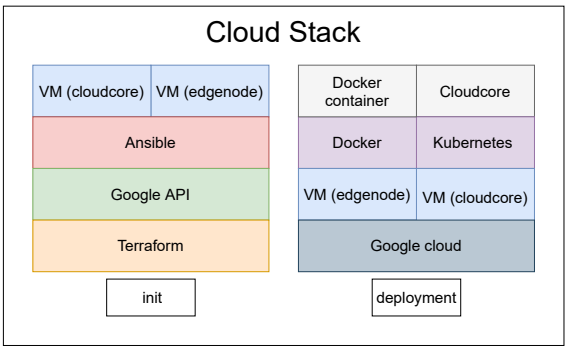

Figure 4: Cloud stack

In the cloud setup, three different Google compute instances of the General purpose family are used for the edgecore nodes. The cloudcore was fixed to one type, also from the general purpose machine family. The used machine types are:

- **Cloudcore:** `e2-standard-2` (2 vCPU, 8GB RAM)
- **Edgecore-0:** `e2-small` (2 vCPU with 50% CPU time, 2GB RAM, shared core, 60 seconds burst-time)
- **Edgecore-1:** `e2-medium` (2 vCPU with 100% CPU time, 4GB RAM, shared core, 120 seconds burst-time)
- **Edgecore-2:** `e2-standard-2` (2 vCPU, 8GB RAM)

The shared core instances are instances for which the compute resources are not fully available (indicated with the percentage of the vCPU time resources available). The `e2-small` has theoretically half the speed of the `e2-medium`, both still have 2 vCPUs. These instances also work with CPU-bursting. [2] This makes it possible to fully utilize the CPU for a period of time if the CPU time is available on that particular instance. CPU bursting also has constraints on how often it can take place and depends on the utilization of the CPU by the process.

[2] **https://cloud.google.com/compute/docs/general-purpose-machines#cpu-bursting**

4 Design experiments

There are three benchmarks relevant for the distributed environment as identified in 2.1. In this section, each of them will be described as what that benchmark means practically how we measure it and its implications.

4.1 Application execution time

4.1.1 Description

Marilyn Wolf states in her book that the execution time of a program is computed as the sum of the execution times of the statements.[6]
For our benchmark, we use the combined Docker image. The image opens an image and classifies it. It does so 15 times for different images. We repeat this process 4 times. Then, we measure the total time it takes to process each image and show the distribution of the time it takes for each image. We assume the logging and time-stamping necessary for bench-marking is negligible overhead. The implication of this benchmark is if we increase the resources and the application time shrinks proportionally to the resources, the application allows for good scalability.

4.1.2 Reference

As a reference, we run the combined image directly on the PC instead of a virtual machine. This is most likely to scale well and can show us how well the combined image scales with multiple threads. We use the reference because it shows us the best possible execution time of the application itself. Using a virtual machine to run the docker image might show different behaviour than expected.

4.1.3 Local

For running the benchmark locally, we set up the framework, and after the edgecore is connected to the cloudcore, we give it the combined image to run in the virtual machine. We provision the VM 1, 2 and 4 CPUs in VirtualBox to test with 1,2 and 4 Threads.

4.1.4 Google cloud

As mentioned in 3.2, Google Cloud provides different machine families[3]. Using a different amount of resources could influence the execution time of the application. The `E2` general-purpose machine type has been chosen due to the fact it is available in every region, zone and on custom VMs. This is not the case for other machine types like n2d that is stated to work well for containerization. The machine resources of these are actually the same but differ in burst time. The `e2-small` instance should last 60 seconds of 2 vCPUs and drops to 0.5 vCPUs, `e2-medium` 120 seconds of 2 vCPUs and drops to 1 vCPU and `e2-standard-2` stays at 2 vCPU. We run the benchmark twice in parallel to get close to 100% of CPU usage. Concluding, we want to benchmark the time it takes to finish publishing 15 images, then receive and classify these images in an edge node using 3 different family machines in google cloud. For each of these, we can also use 1,2 and 4 threads for the combined image.

4.2 Deployment overhead

"Overhead Matters: A Model for Virtual Resource Management" implies that the deployment overhead is the time it takes a server to process a request for a virtual workspace.[3] Taken the deployment stacks from figure 3 and figure 4 into account, the deployment overhead is the time difference between the Google Cloud and local machine receiving our request to initialize the environment, and the moment all edgecore nodes are ready to start running the Python applications required to start the benchmark. The resource performance likely accounts for some bottlenecks in the deployment overhead. Therefore, like the application execution time, we will benchmark on the 3 different resources specified in the cloud previous section. Since the deployment during development showed to be larger than 2 minutes, we would expect drop-offs in speed during deployment, and different amounts of deployment times on each machine. This is with the assumption that the measuring itself has

[3] **https://cloud.google.com/compute/docs/machine-types**

no implications on the performance.

4.3 Network delay

In a structural analysis of network delay, it is stated that most research in link delay has been from end-to-end measurements with a per-packet delay.[1] With the knowledge from how the application works, it can be defined in two different ways. The end-to-end can be seen from a publisher & subscriber perspective. This includes the delay of transporting data from the publisher to the MQTT broker, actions within the MQTT broker and transporting the data from the MQTT broker to the subscriber. For testing, we test with 1,2,4,8 publishers at the same time and a subscriber with 8 threads. With the multiple publishers, we can try and reach the limit of the throughput of this network. Alternatively, we can view it as a publisher/broker & and broker/subscriber connection, where the time the broker needs is not part of the connection. This makes the testing simpler because it does not require the consideration of possible delay factors within the broker. Nonetheless, we will use the first description, since for the function of the application this is the end-to-end packet delay and partially because the data will remain within an Edgecore node. The measurement is then the average time it takes for an image to reach a subscriber from a publisher with one publisher and one subscriber connected to a broker. If the network delay is low in the cloud environment, that means the cloud environment did well placing the application physically in the data centre. Unlike the previous 2 benchmarks, where we only require the timestamps to be synced within one system, we assume that the timestamps are synced with the host and the virtual machine. While virtual machines are known to be unreliable with syncing their time with their host, some packages try and improve this problem. However, it remains a problem and the results should be taken with a grain of salt. Google provides solutions for time syncing, but again, here we assume that the publisher is within their cloud provider.

4.4 Local setup

The publisher runs in a docker container and publishes on a broker running in a VM that sends that image to a subscriber that is also in the container as shown in figure 5.

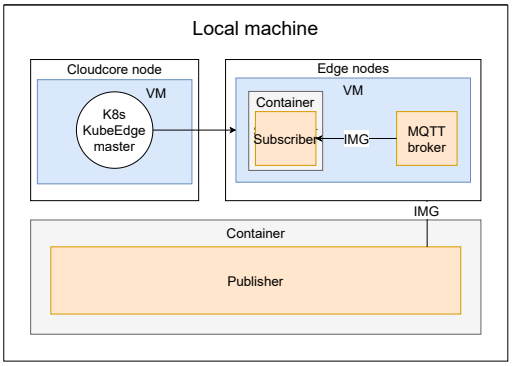

Figure 5: Local network setup

4.5 Cloud setup

The publisher runs in a docker container at home and publishes on a broker running in a VM (in Google cloud) that sends that image to a subscriber that is also in the container as shown in figure 6.

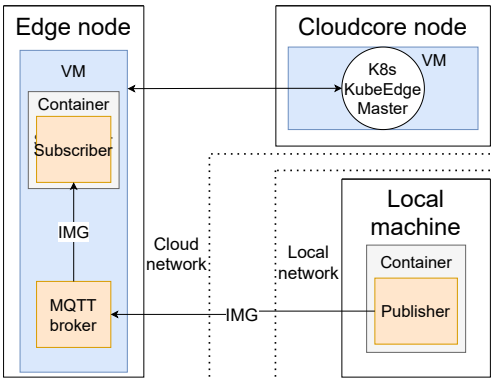

Figure 6: Cloud network setup

5 Experiment results

In this section, we will discuss the results of our experiments. We analyse the application execution time, deployment overhead and network delay as described earlier.

5.1 Application execution time

We measure the execution time in three different environments for an image classifying application. We measure the reference execution time directly on the PC, we measure an applications execution time locally on VMs and we measure the applications execution time on Google cloud. For each environment, we run the application with three different amounts of threads: 1, 2 and 4.

5.1.1 Reference native execution time

The reference execution time is shown in figure 7.

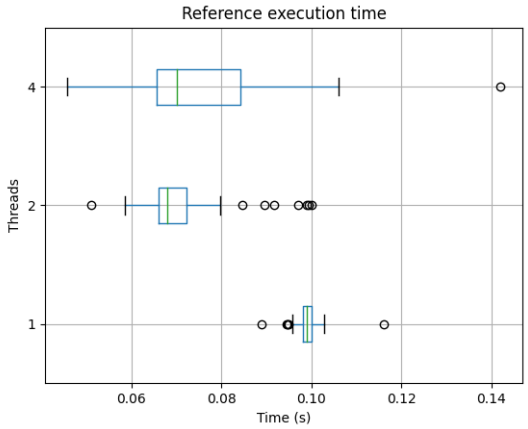

Figure 7: A boxplot for the execution time of an image classifying application for 1, 2 and 4 threads for the reference environment

First, we notice that the mean execution time for 1 thread is around 0.1 seconds and it took a total of 45 seconds to finish 450 images. This decreases to around 0.07 seconds for 2 threads for which took about 31.5 seconds to finish 450 images, which is in the line of expectation. However, the application has roughly the same mean execution time for 4 threads compared to 2 threads. We think this is due to the communication overhead and blocking of the threads. Also, we notice with the increase in threads, the distribution of execution time of each run is bigger. We feel this is due to the same reasons. The execution time becomes more variable because the arrival of threads, blocking of threads and communication between threads introduce a higher variability.

5.1.2 Local execution time

The local execution time is shown in figure 8.

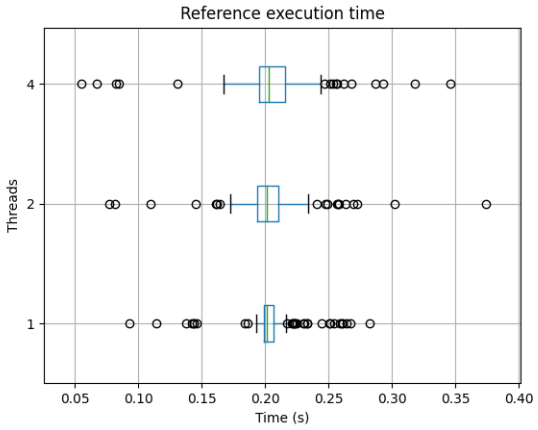

Figure 8: A boxplot for the execution time of an image classifying application for 1, 2 and 4 threads for the local environment

We immediately can see that the execution time for each amount of threads stays the same. The total time it took was 31.2 seconds to finish 150 images. We feel that this is due to communication and synchronization between the different threads. We can also see that the execution time is twice as high compared to the mean execution time of the reference implementation for 1 thread. We feel that this might be due to the network delay between VMs. But we will have to discuss this further in section 5.3.

5.1.3 Cloud execution time

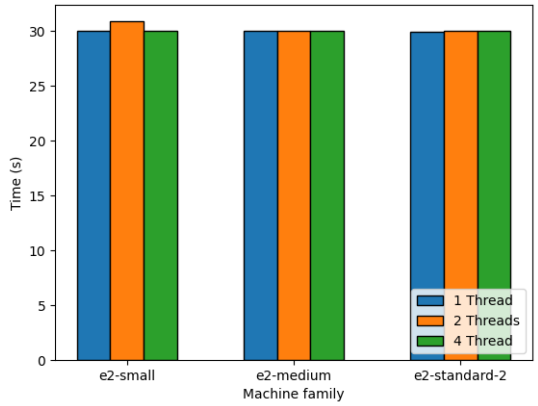

Figure 9: Measured total execution time with 1,2 and 4 threads and the different suggested family machines

We see here that the `e2-small`, `e2-medium` and `e2-standard-2` have the same performance. This is to be expected according to the google documentation since they get the same resources within a certain time period.

Google Cloud `e2-small` instance

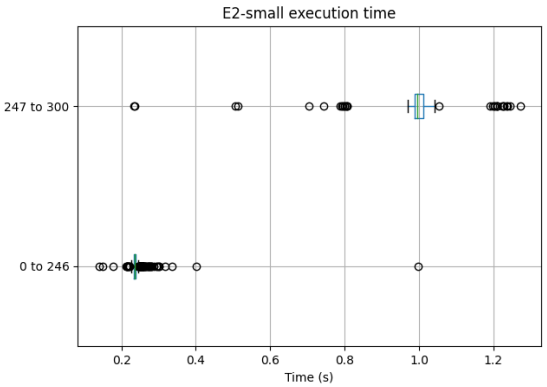

Figure 10: Difference

According to Google's documentation on their machine families, the e2-small gets a burst of 60 seconds with 2 vCPUs and drops to 0.5 vCPUs. We have used a frequency of 10 in the combined benchmark. And after 57 seconds (247*2 images processed) it dropped from 0.12 seconds to 0.5 seconds to process an image. This is half of the average on figure 10 because the image assumes they are in sequence and not in parallel. This confirms the documentation of what Google provides with their performance and our expectations.

Google Cloud `e2-medium` instance

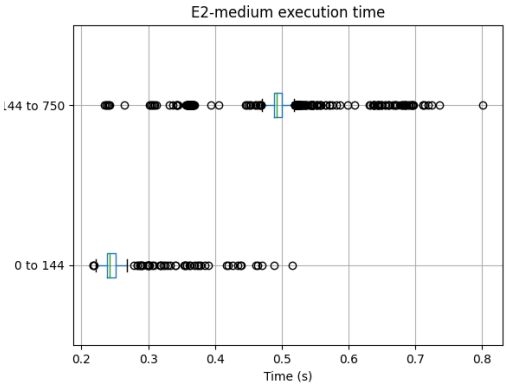

Figure 11: Difference

According to Google's documentation on their machine families, the e2-medium gets a burst of 120 seconds with 2 vCPUs and drops to 1 vCPUs. We have used a frequency of 20 in the combined benchmark. And after 35 seconds (247*2 images processed) it dropped from 0.12 seconds to 0.25 seconds to process an image. This is half of the average on Figure 11 because the image assumes they are in sequence and not in parallel. This is not what Google has provided in their documentation and does not confirm our expectations since the performance was halved (expected) but already after 35 seconds instead of the documented 120 seconds. This could be because previous benchmarks have a negative effect on the burst time of 2 vCPUs. But it is a comparative steep drop-off in provided burst time only having run that benchmark once on

that VM and 2 on different ones.

5.2 Deployment overhead

In this section, we discuss the deployment overhead we measure for both the local instance as well as the cloud instance.

5.2.1 Local deployment overhead

The measured deployment overhead in our local environment is shown in figure 13. For each node, we measure the startup time by keeping a log of the activity of each node. Edge nodes start up one after the other, so to get the total deployment overhead for one network, the deployment overhead of each node should be added up to get the total deployment overhead. Some initialization is done in parallel for each edge node, therefore we divide this time over the number of edge nodes and add this time to each individual edge node. This way we can compare the different networks with different amounts of edge nodes easily.

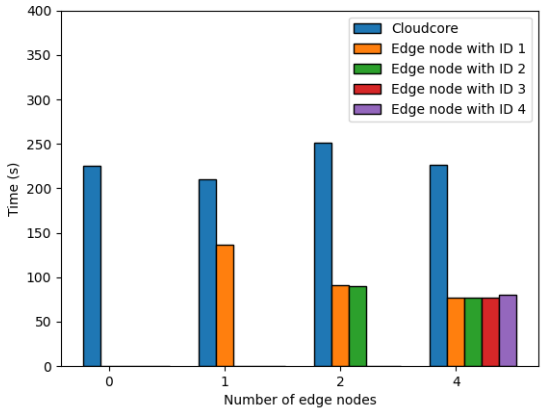

Figure 12: Measured deployment overhead on local environment for the cloudcore without edge nodes, with 1 edge node, edge nodes and 4 edge nodes

We measure the startup time for each node in the network in four different networks: A cloudcore standalone, a cloudcore with 1 edge node, a cloudcore with 2 edge nodes and a cloudcore with 4 edge nodes. In the results we can see that the startup time per node decreases when the number of edge nodes increases. This makes sense since the parallel initialization will be relatively less time consuming for more edge nodes. However, do notice that the total added deployment overhead for a network increases with the number of edge nodes. We show this in Figure 13. This is, of course, due to the fact that the edge nodes are partially initialized sequentially.

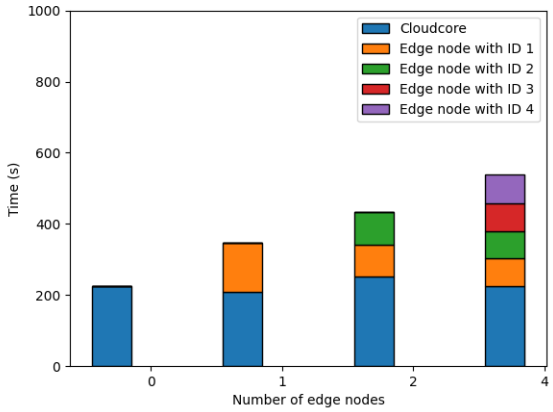

Figure 13: Measured total deployment overhead on local environment for the cloudcore without edge nodes, with 1 edge node, edge nodes and 4 edge nodes

5.3 Network delay

In this section, we try to measure the network delay between 1, 2, 4 and 8 publishers and a subscriber. Both locally and on the cloud, we will run tests and display the measured internet delay and measured internet speed.

5.3.1 Local

The measured local network delay per image is shown in Figure 14. For each image, we measure the delay for 1, 2, 4 and 8 publishers.

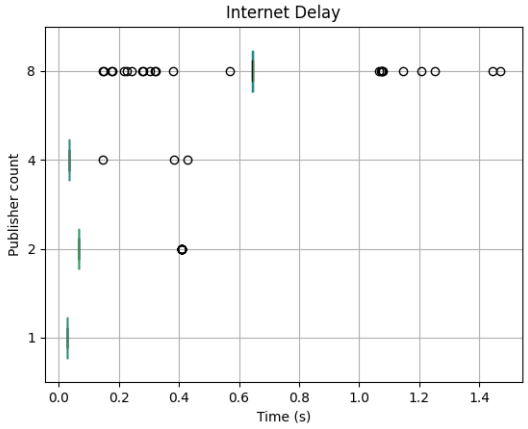

Figure 14: Measured delay of each image with 1, 2, 4 and 8 publishers

We can see that, overall, the internet delay goes up with the number of publishers. We feel that this is due to the publishers needing to share the bandwidth. With more publishers, each publisher can use less throughput. This results in the delay increase for each individual publisher, even though more or the same bandwidth is used in total.

We also measured the internet speed of each image with 1, 2, 4 and 8 publishers.

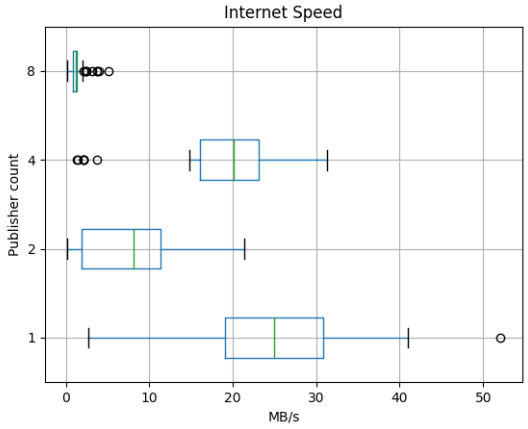

Figure 15: Measured internet speed of each image with 1, 2, 4 and 8 publishers

In Table 1 we recognize two different columns: The average throughput per image during the time that the image is being transferred and the total average throughput over the whole length of the application for all traffic. The former is also shown in Figure 15. We see that when the amount of publishers increases, the average total throughput goes up. However, we also see that the average internet speed per image goes down when the amount of publishers increases. This is expected since the available bandwidth is shared over the images for bigger amounts of publishers. Furthermore, we expected the total average throughput to be higher with more publishers since we expect more publishers to result in less downtime in the system.

	Average per IMG	Total average
1 pub	24.02 MB/s	0.66 MB/s
2 pub	6.35 MB/s	1.39 MB/s
4 pub	18.11 MB/s	2.55 MB/s
8 pub	1.17 MB/s	3.75 MB/s

Table 1: Table of average internet speed with and without downtime

6 Discussion

In this section the results are discussed and the ease of use for the deployments tool used.

6.1 Setup local environment

For the local environment, we opted for Vagrant in combination with Virtualbox. Vagrant simplifies the deployment of virtual machines and in combination with Virtualbox it can be ran on MacOS, Windows and Linux. Vagrant also takes care of initialization of the network, the virtual machines and deployment of the operating system in the VM and has support to execute Ansible playbooks by automatically connecting to the virtual machine. This method also provides the means of creating the same KubeEdge cluster on different systems. The ease of use of Vagrant was a decisive factor in using it over alternatives as libvirt, which would require more effort in understanding the library and usage.

6.2 Setup cloud environment

The cloud part of this project was based on code provided in a GitHub repository [4] containing an automated setup to deploy a KubeEdge cluster to Google Cloud. Additionally, as of current date (December 2021) Google Cloud offers a free trial with $ 300 credit which gives a free opportunity for the team to try out a cloud provider without the stress of the unknown cloud costs. There are however limitations in the free trial: there can only be 4 resources with an external IP and up to 8 compute instances can be created. However, due to the limited knowledge of networking in Google Cloud and the time limitation of executing this project, we could not create an efficient way (e.g. virtual switch) to stretch out the total of compute instances.
For the deployment, Terraform was used. While the Terraform code was from the provided GitHub repository, knowledge about Terraform was present within the team to be able to interpret the code. This was however not of advantage since the Google Cloud resources and setup was new to all team members. The provisioning of the virtual machines in Google Cloud was done using an initialization script coupled with the compute instance, which started the Ansible playbooks. Most of the time creating a working KubeEdge cloud instance went to learning the Google Cloud instances as well as the Google Cloud SDK commands to make debugging of the resources possible.

6.3 Application execution time

Since the combined image only reads and classifies the images, we felt this was a better image to use, since it excludes the reliance of the network delay when using a publisher and subscriber. And thus the results are easier to clean for data analysis.
We used a reference for the application execution time where we ran the combined Docker image since we did not know what the impact of the usage of a VM would have on adding new threads. Using this would also show a more likely limit of scaling within the application while using multiple threads.
We only ran our benchmark in one edgenode, and not multiple at the same time, because it is supposed to be an edge environment. This means that there would not be multiple edgenode instances on one machine. Thus ultimately, one working VM should not influence another on a different machine.
Of course, the total time of the reference was quicker than the local or cloud environment, since it does not have the added layer of abstraction of a VM. It also showed that within the docker image, the application scaled well up until the usage of 2 threads. The VM's locally and in the cloud showed a similar level of performance but surprisingly did not scale with more threads because performance stayed similar with different amounts of threads. Since we did not get any errors during execution with multiple threads, we assume it was able to use them, but some documentation online shows that VM providers have issues with the support of hyperthreading and thus might be the reason for not scaling.
For the cloud set up, we wanted to test the E2 machine family, because it was the most logical family type. Others like the N2 are relatively more expensive and are stated to not be for this purpose, nor are they available in every region. Because of different testing of E2 types we wanted to test the specifications of the bursting of 2vCPU's with it.

6.4 Deployment overhead

The deployment overhead is done by logging in the VMs in our local environment. An uncertainty is that logging can be delayed for any VM. This means that this section may not portray a completely accurate representation of the actual deployment overhead. However, we do feel that this delay is insignificant.
The deployment overhead does not allow for a lot of variation on how it is set up, since that depends on your implementation and might only differ with different hardware. Thus for a local

[4] **https://github.com/itselavia/kubeedge-cluster-gcp**

environment, there was not much to change. For the cloud we would have liked to use different versions of the E2 family machine type, but we were not able to benchmark this due to time limitations.

6.5 Network delay

As stated in our results, network delay was a difficult thing to measure, since this would require syncing virtual machines to the host, or virtual machines to sync with a remote host. Although we felt we should try to at least perform the benchmarks, there is a high chance the actual results deviate from our measured results. With more time, we would have liked to spend more time thinking about a better testing environment that would not rely on time syncing between machines.
For our local environment, we decided to run the edgenode and the publisher on the same machine, the broker and subscriber in their virtual machine. This allowed for quicker testing and most likely somewhat better results because it is easier with Linux packages to try and sync the time between the VM and the host than syncing the time between the VM and a different machine altogether. Although this is not a real situation for an edge deployment, it fitted better within our time budget.
And since we did not have the time for benchmarking this version in the cloud, we felt we could just do the more realistic design for it.

7 Conclusion

In conclusion, we have automated the deployment of a Kubeedge environment locally and in the Google cloud. We also have benchmarked the execution time, deployment overhead and network delay for an image classifying application on a local and cloud environment in order to get more insight into Kubeedge and the Google cloud environment. Although we were not able to properly describe and process our benchmarks in the cloud, we have performed them all successfully. This does mean that we are unable to compare the local and cloud environment we set up. However, we have run all benchmarks on the local environment. We can see that the application does not scale based on the results we have gathered for the application execution time. Furthermore, we notice that the deployment overhead also increases with the number of edge nodes. With this application, the throughput of image transfers does go up with more threads as well. This indicates good scalability. Further research is necessary in order to get a better analysis of the environments we have presented.

Appendix

References

[1] A. Abdelkefi and Y. Jiang. "A Structural Analysis of Network Delay". In: *2011 Ninth Annual Communication Networks and Services Research Conference* (2012). DOI: https://ieeexplore.ieee.org/abstract/document/5771190.

[2] Stuart Anderson. *A Brief History of Containers: From the 1970s Till Now*. URL: https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016. (accessed: 2021.12.20).

[3] K. Keahey B. Sotomayor and I. Foster. "Overhead Matters: A Model for Virtual Resource Management," in: *First International Workshop on Virtualization Technology in Distributed Computing* (2006). DOI: https://ieeexplore.ieee.org/abstract/document/4299350.

[4] HashiCorp. *"introduction to Vagrant*. URL: https://www.vagrantup.com/intro.

[5] Kevin Wang and Fei Xu. Oct. 2017. URL: https://kubeedge.io/en/.

[6] Marilyn Wolf. "High-Performance Embedded Computing (Second Edition)". In: MK, 2014. Chap. 3.4.

A Time spend on project

Time group	Time spend
Total time	285H
Think time	30H
Dev time	150H
XP time	15H
Analysis time	10H
Write time	50H
Wasted time	30H