Benchmarking the performance of standard Spark Python examples (Native vs. SCONE)

Manik Khurana

Technische Universität Dresden and Scontain GmbH

October 12, 2022

Abstract

The aim of the project is to determine the time difference and the performance overhead introduced by implementing Confidential Computing methodologies via the SCONE framework. Confidential Computing is a cloud computing technology that can isolate data within a protected CPU while it is processed in a physical space called the enclave which can only be accessed by specially authorized programming code that has privileged access to it. The SCONE framework makes it easier to implement this. SCONE protects the application code from the OS and the hypervisor by isolating each service individually using the Intel Software Guard Extensions (SGX). Spark was chosen as it is very powerful and it executes much faster by caching data in memory across multiple memory operations. The examples chosen for the benchmarks are most suited to common workflows, and are computationally intensive, to begin with. The programs selected work without taking any additional input and the performance gap observed is consistent among them. These programs are run in three environments - Native, Minikube Kubernetes, and Kubernetes with SCONE. The results obtained show that there is a significant performance overhead imposed when the concepts of Confidential Computing like Local Attestation (LAS) and Configuration and Attestation Service (CAS) are introduced. A comparison of these benchmarks shows us that the programs when to run on the native system take 14.8 seconds on average, while on Minikube Kubernetes it is 47.5 seconds (i.e. 2.3x more vs.Native) and when using the Sconified image it is 2:26 minutes (i.e. 2.1x more vs. Minikube). However, this does not mean they are exactly 2.3x or 2.1x slower. A lot of factors justify this time overhead, like the time taken by SGX to create a separate enclave for each program, the network latency between the distributed architecture of Azure, the multiple encryption-decryption cycles, etc. More on this can be found in the later sections.

Contents

List of Figures

1 Introduction

1.1 What was the internship about?

The internship was about learning confidential Computing and benchmarking the Python examples of Apache Spark in a Native environment vs in a Confidential environment using the SCONE framework. To learn more about it a few examples of Apache Spark in Python were chosen to work with. The whole project was divided into 3 phases to ease the learning process.
In the **first phase**, we focused on learning the Apache Spark terminologies and setting up the machine (Linux-Ubuntu). The machine needed at least 8GB RAM and 2 Processors. Initially, we

tried running it on a Ubuntu Virtual Machine hosted on Windows via Oracle VM Virtual Box. It had its own limitations so we shifted to the Dual-Boot Ubuntu-Windows solution. This worked fine and the results seemed satisfactory. The programs were first tested individually. From a pool of 40+ programs available, these 17 were selected to be worked upon. These programs did not require any additional input. For programs that needed any additional input, (e.g. wordcount.py) there is a provision to pass the input file in line 20 of Script - phase 1. Since we had only one such program, we hard-coded the name in the py script.
In the **second phase**, we run the same examples from phase 1, this time in a Minikube Kubernetes environment. For this Minikube Kubernetes was set up and installed. The system requirements were checked and ours was sufficient for running these. As in phase 1, each program was tried one by one. There were a lot of configuration changes that were done to make the first program run successfully. As per the instructions given on [6], the master, deploy-mode, name, class, and conf tags were played around with. It was a long, grueling session of hits and misses after which we landed on the exact configuration that runs the first Pi program. After this, each program was tested one by one and then a script was coded to make the process simpler and easy to understand. This is the Script - phase 2. It starts with initializing the Minikube Kubernetes so we have a K8s-server address. Then the serviceaccount and clusterrolebinding were set up for spark using kubectl. Then each program will run one by one and the duration will be captured and sent to the file specified. This is the Duration results - Script 2.
In the **third phase**, we run these examples using SCONE on Kubernetes. A sconified image was provided by the company Scontain. We were also provided with access to some private repositories, unreleased packages, and Readme documents that help complete the project. Over the course of 4 months, a lot of approaches were considered and then dropped as the project was still under development at the time of writing this report. Nevertheless, the whole process was made simple by coding the entire working solution as a bash script that once run, will deliver the results. The Script - phase 3 (Main) is the one that runs mainly and calculates the time duration taken by each program and then sends it to the file Duration results - Script 3. This script refers to 2 scripts - the before and after. The Script - phase 3 (Before) sets up the las and sgx-dev-plugin as pods using helm charts. It then runs the provided script to build the spark-sconified image which is then pushed to the docker-hub. It also sets up the environmental variables and creates policies for running the programs. One of the most important steps done by this script is when it attests to the CAS - Configuration and Attestation service. The program can only run if the CAS is 'trustworthy.' The Script - phase 3 (After) creates the required policies and contains the main spark-submit command.

The examples selected are:

1. AFT Survival Regression
2. ALS
3. Bucketed Random Projection LSH
4. Bucketizer Example
5. CHI Square Test Example
6. Chisq Selector Example
7. correlation Example
8. Feature Hasher Example
9. Pi
10. Pipeline Example
11. Polynomial Expansion Example

12. Quantile Discretizer Example
13. Rformula Example
14. Stopwords Remover Example
15. String Indexer Example
16. Univariate Feature Selector Example
17. Word2vec Example

1.2 What was the motivation for your internship task?

The motivation to take up this internship project was to get into researching the topic of Confidential Computing. To learn more about it by implementation, a study case was found where benchmarking the selected Python examples in a Native environment and in a SCONE-based environment would help understand the basics of it. This also gives us the basis for improvement. New versions could use these benchmarks as is to optimize the process and reduce the time complexity.

2 Project Overview

2.1 What system have you worked on?

The system used was my own laptop with Ubuntu OS in the dual boot to Windows OS. The tools used include Apache Spark, Kubernetes implementation using Minikube, and SCONE. In phase 3, however, there was a need for more resources for which an Ubuntu Virtual Machine (hosted on Azure with 60GB RAM and 8 processors) was provided. Along with that, for real-world use cases and testing a Kubernetes cluster (hosted on Azure) was also taken into account. Using the combination of these two resources, we were able to get the results of Phase 3.

2.1.1 Installation of Apache Spark and Minikube Kubernetes

To download and install Apache Spark [7] on your system, you need to make sure the requirements are satisfied:

- Java
- Python 3 (As we need to run PySpark examples)
- Scala
- Git

To make things easier, running the complete steps will help you install all the basic requirements of Spark and Minikube.

Installation of Spark

```
## Step 1: Install Python3, Java and git
sudo apt install curl mlocate default-jdk scala git -y
# To see if it's all installed correctly, run:
java -version; javac -version; scala -version; git --version

## Step 2: Download Spark
# To make sure, you always run the correct command, check out the official Apache
    Spark website and get the wget command form there.
wget https://dlcdn.apache.org/spark/spark-3.2.1/spark-3.2.1-bin-hadoop3.2.tgz
# Now, extract the saved archive using tar:
tar xvf spark-3.2.1-bin-hadoop3.2.tgz
```

```
# Then, move the Spark folder created after extraction to the /opt/ directory.
sudo mv spark-3.2.1-bin-hadoop3.2/ /opt/spark

## Step 3: Setting Spark Environment
vim ~/.bashrc
# Now, to this file add:
export SPARK_HOME=/opt/spark
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
# Activate the changes made.
source ~/.bashrc
```

Installation of Minikube

```
## Step 1: Update System
sudo apt-get update
sudo apt-get install apt-transport-https
sudo apt-get upgrade

## Step 2:  Download and install kubectl
# We need kubectl which is a command line tool used to deploy and manage
    applications on Kubernetes:
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https
    ://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/
    amd64/kubectl
# Make the kubectl binary executable.
chmod +x ./kubectl
# Move the binary in to your PATH:
sudo mv ./kubectl /usr/local/bin/kubectl
# Check version:
kubectl version -o json  --client

## Step 3:  Download and install Minikube
# We need to download the minikube library. We also need to put the binary under /
    usr/local/bin directory since it is inside the Path variable.
wget https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
chmod +x minikube-linux-amd64
sudo mv minikube-linux-amd64 /usr/local/bin/minikube
# Confirm the version installed.
minikube version
```

2.2 What was the problem you had to solve?

Introducing the techniques to make any functional system totally secure and confidential also introduces performance overhead to the system. If these performance overheads are known before the planning stage of any project, it would be easier to estimate the final result and whether one should make it confidential or not. And if yes, then what are the repercussions?

2.3 What was the task and the goal of your project?

The task for the project required us to select the appropriate Python examples from the Apache Spark GitHub [8] repository and run them first on the Native system, then on a local setup Minikube Kubernetes, and then again in the Confidential Environment using the SCONE Framework.

3 Experiences

3.1 What experiences have you gathered during your internship?

The experience of learning to work with tools like Apache Spark and Kubernetes has been an enriching one. Using Minikube is now within reach for me and its coordination with the other apps is something I know very well now. One of the most important skills I've enhanced by

working on this Internship project was Patience and Resilience. The project was finished and we never gave up on it.

3.2 What new concepts have you learned about?

3.2.1 Intel SGX

Intel Software Guard Extensions allow applications to ensure confidentiality and integrity even if the OS, and hypervisor of the BIOS are compromised/attacked. Intel SGX also protects them against attackers who have physical access to the system, if and only if the CPU package is not breached [9]. The SGX provides trusted execution environments to the applications, known as Enclaves. The Enclave Page Cache (EPC) is a region of the protected memory and it holds the Enclave data and code.

3.2.2 Confidential Computing

As per the definition available on Intel's website [9]: Confidential computing is an initiative that focuses on helping to secure the data in use. The efforts that are put into it can enable the encrypted data to be processed in memory. This helps lower the risk of exposing it to the rest of the system which in turn reduces the potential for sensitive data to be exposed. This is done while providing a higher degree of control and transparency for the users of the system. In multi-tenant cloud environments, the sensitive data is meant to be kept isolated from other privileged portions of the system stack. Intel SGX plays a large role in making this capability a reality. As computing moves to span multiple environments from on-premises to public cloud to edge—organizations need protection controls that help safeguard sensitive IP and workload data wherever the data resides.

3.2.3 Containers

Containers provide a lightweight mechanism for isolating an application's environment. For a given application, we can specify the system configuration and libraries we want to be installed without worrying about creating conflicts with other applications that might be running on the same physical machine. We encapsulate each application as a container image that can be executed reliably on any machine with Docker installed thereby providing us the portability to enable smooth transitions from development to deployment. This also means that we can have multiple workloads running simultaneously without any hassle and thus achieve higher resource (memory and CPU) utilization - ultimately lowering all kinds of costs involved. Containers also come in line with the same design principle as the microservices architecture. They're principally made for only one and single purpose and are hence immutable by design [2]. A good difference between Virtual Machines, Containers to physical machines can be seen in 1.

3.2.4 Kubernetes

As good as containers are, they do not provide a solution for **fault tolerance**. In most real-world workloads the containers need to communicate via a networking channel with each other, something not provided by them. Kubernetes is a container orchestration platform. Orchestration as in the director of the orchestra holds the *vision* for a musical performance and *communicates* with the musicians in order to *coordinate* their individual instrumental contributions to achieving this overall vision. As the architect of a system, our job is simply to *compose the music* (specify the containers to be run) and then hand over control to the *orchestra director* (container orchestration platform) to achieve that vision. Kubernetes has objects used to make sure we have our desired state of the system and they can be defined using either YAML or JSON files. The files which do that in action are called *manifests* [2].
The object **Pod** is the fundamental building block in Kubernetes. It has one or more tightly

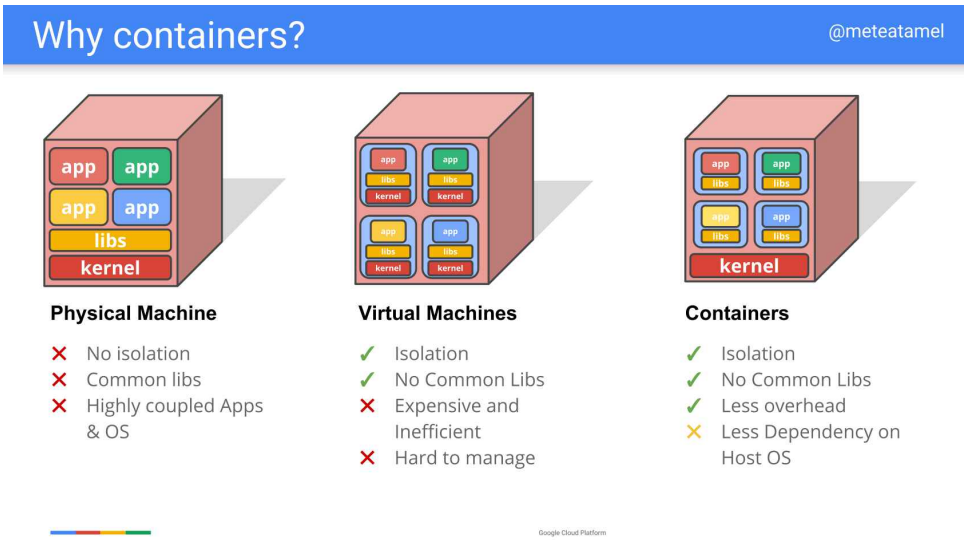

Figure 1: Why should we use containers [1]?

related containers, a shared networking layer of its own, and shared filesystem volumes.

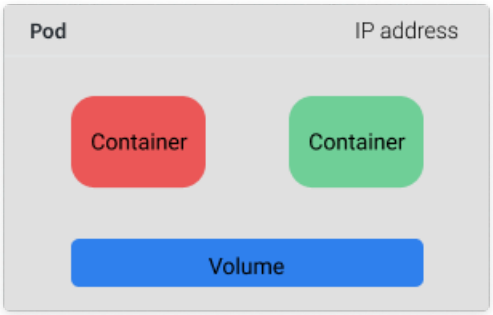

Figure 2: A single Pod in Kubernetes [2].

The object **Deployment** encompasses a collection of Pods which are defined by a particular template and a replica count n. Deployments also allow us to specify how we would like to roll out updates when we have new versions of our container image.

There also exists the **Kubernetes control plane** which oversees how workloads are executed, monitored, and maintained in the cluster. The 2 classes of machines on the cluster are (i) **Master node** which contains most of the components that make up the control plane. (ii) **Worker node** which actually runs the application workloads [10].

3.3 What technologies have you learned about/worked with?

3.3.1 Apache Spark

Apache Spark is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters as defined on [11]. It is a unified analytics engine for large-scale data processing. It provides high-level APIs in Java, Scala, Python, and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level

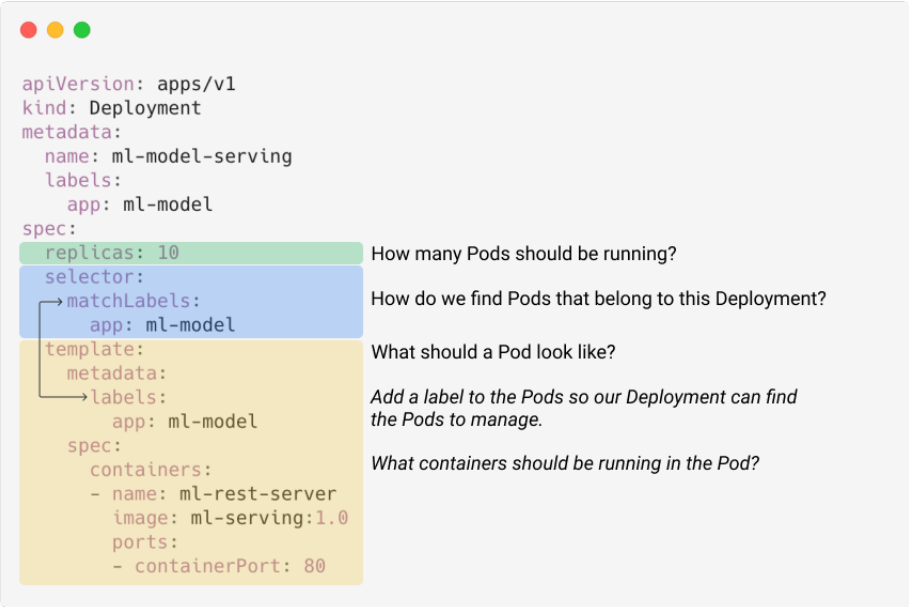

Figure 3: A Deployment Object in Kubernetes [2].

tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Structured Streaming for incremental computation and stream processing.

3.3.2 Minikube (Kubernetes)

Minikube is a local Kubernetes. It makes the development and testing of applications easier and is a great help to learn how the actual Kubernetes functions. For Minikube, all we need is a Docker container.
Minimum requirements:

- 2 CPUs or more.
- 2GB of free memory.
- 20GB of free disk space.
- A stable Internet connection
- Container manager: Docker

Some examples of commands for Minikube and kubectl:

```
minikube start : to start minikube.
minikube stop : to stop minikube.
minikube delete : to delete everything related to minikube (this does not delete
    the installation).
kubectl cluster-info :  to check cluster status
kubectl config view :  to view Config
kubectl get nodes :  to get nodes
kubectl get pods :  to get pods
kubectl delete pod --all :  to delete all pods
#Minikube also offers a Dashboard:
minikube dashboard
```

3.3.3 SCONE - LAS and CAS

SCONE stands for Secure Container Environment [9]. It is a secure container mechanism for Docker that uses the SGX (Intel Software Guard Extensions) trusted execution support of the Intel CPUs to protect container processes from outside attacks. The design choice of SCONE benefits the system in a way that (i) It has a small Trusted Computing Base (TCB). (ii) a low-performance overhead. The objective of SCONE is to protect the **Confidentiality, integrity** and **consistency** of an application's **data, secrets** and **code** everywhere - **in use** (i.e. in the main memory), **in transit** (i.e. on the network) and **at rest** (i.e. on disk) during the entire lifetime of the application.

SCONE and Confidential Computing The first aspect of Confidential Computing is that applications can be protected against privileged software like the OS and the hypervisor. To ensure that the data and code of a native application are protected enough, there is a need to make sure that all software and hardware components are sufficiently protected. The SCONE framework isolates each of the services individually thereby reducing the attack's surface and the effort to certify that an application is properly protected. The isolation also enables secure outsourcing of the management of components like the OS and Kubernetes to a cloud service provider. The SCONE framework also decouples the integrity and confidentiality of data and code from all other software components and from the entity that manages the software and hardware stack [12]. It is well known to everyone that many vulnerabilities are caused by using out-of-date software or hardware or firmware components. The second aspect of Confidential Computing is that one can attest all components that are needed to ensure the confidentiality and integrity of the application. These components include the CPU, its firmware and the application code, and its data itself. The attestation ensures that these components are up-to-date and no vulnerabilities are known for these components at the time. In other words, one can establish trust in all components that are required to execute the application. In SCONE, this attestation is done transparently at the start of each program. SCONE supports the typical Docker workflow to create Docker images that contain the Python engine as well as the Python program. SCONE supports the encryption of Python programs to ensure both the confidentiality as well as the integrity of the programs. A typical workflow might look like the one in figure SCONE Workflow of a Python code [3].

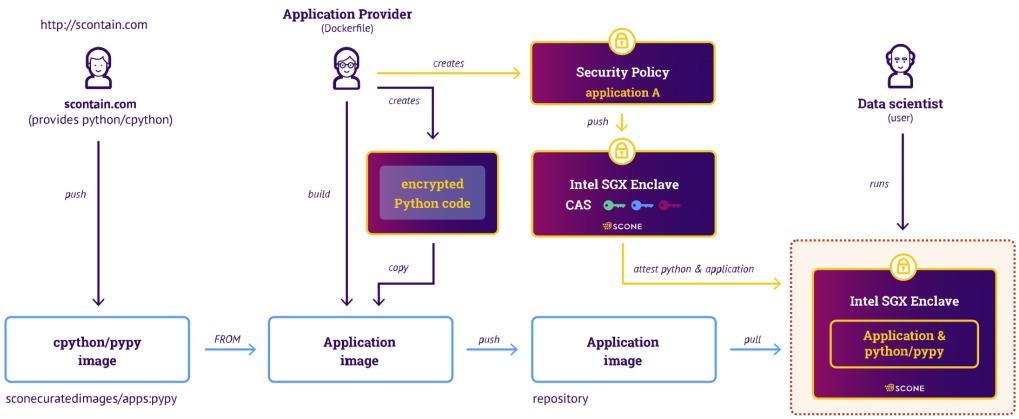

Figure 4: SCONE Workflow of a Python code [3]

Local Attestation (LAS)
The LAS instance is needed to perform Local Attestation which is creating a *quote* that can be later verified by CAS. The LAS can be installed using helm [13]. In this project, LAS is installed in Phase 3. It is required to run the programs in a confidential manner.

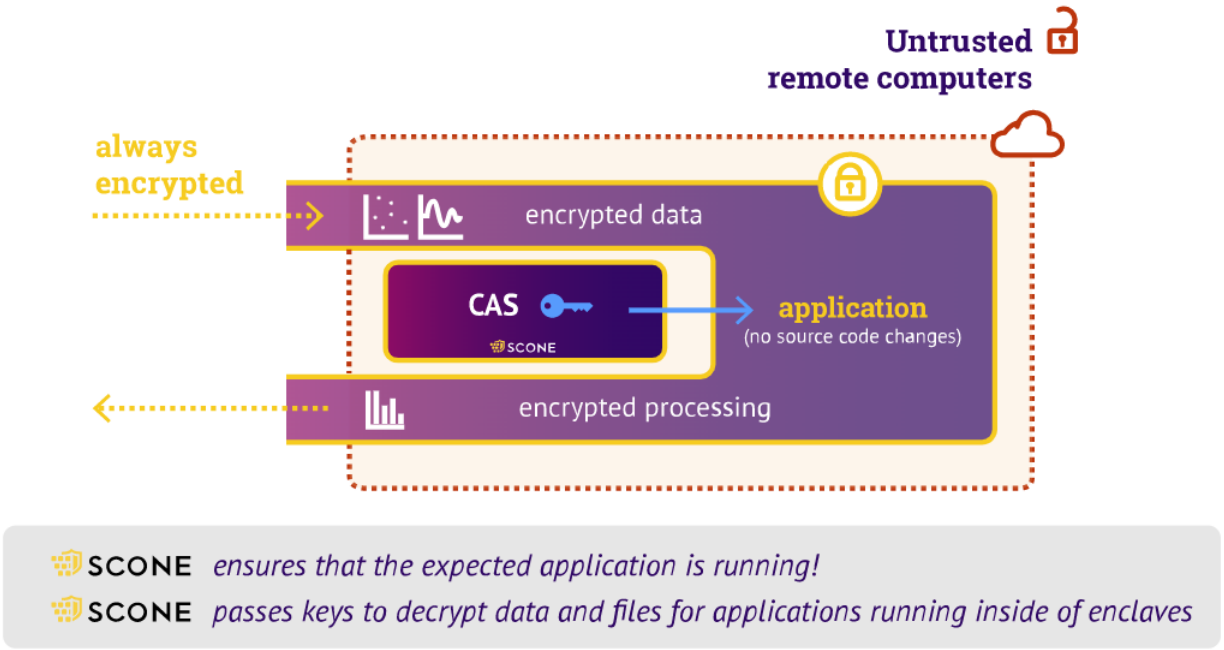

Figure 5: SCONE in action [4].

Configuration and Attestation Service (CAS)
The SCONE CAS manages the secrets of an application from inside an enclave. The application is in control of the configuration data and only selected services can get access to the keys, encrypted data, and configuration data. These services are given this access only after successful attestation and verification. More about Key Generation, Policy Control, Secure Configuration, Access control, and more can be read on [5]. The way we used CAS in this project is in Phase 3. The Script - phase 3 (Before) has commands that initialize variables for CAS and then attest it before the namespaces and policies are created.

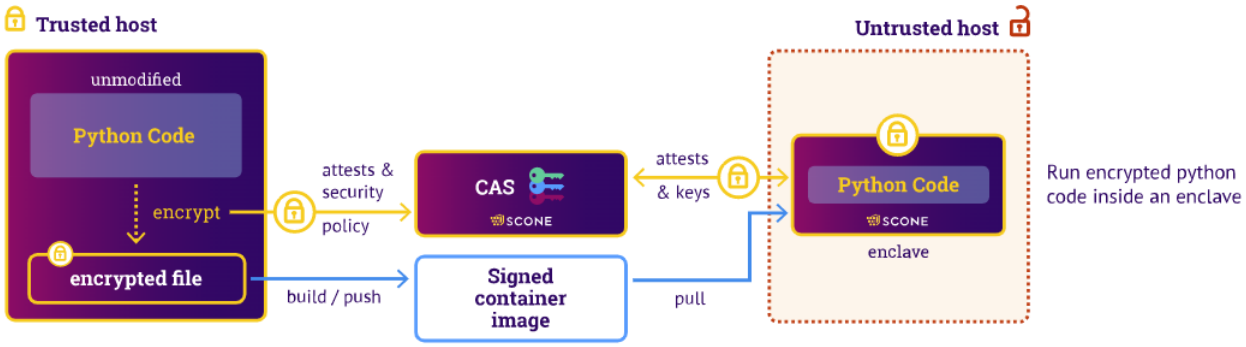

Figure 6: Encryption of Python programs and input in CAS SCONE [5].

4 Internship Tasks

4.1 Which approach have you followed to solve the problem?

The initial dilemma was which programs to select for the time benchmarks. Upon discussion, it was noted that we can use the Python Machine Learning (ML) examples provided by Apache Spark in their GitHub repository [8]. The approach deployed can be easily explained in 3 phases. These three phases began by applying brute force techniques to first get the system running perfectly. Then, to optimize the solution obtained, the whole operation was automated and thus, can now be run with a single click/command.

4.1.1 Phase 1: Running Benchmarks on Native System

In order to be efficient and make lives easier, the automated python script reads the files from the folder and executes them one by one on the system. The results include the minimum, average and maximum time of execution for each of the 17 Python examples.

- **Script explained:** The Python Script - phase 1 starts by importing the required files/libraries. The **os** library is required to run the programs and return the results back to the script. The **statistics** library is used to calculate maximum, minimum, and average values where the value is the time taken by the program to execute. The **time** library is required to calculate the duration/time of execution taken by each individual program. The constant variable **RUN** is the number of times the program needs to be run. Here, we also include an example - wordcount.py as a special but useful case where we had to figure out how to run a program if it needs any kind of input.

- **System specifications:** The base machine for running benchmarks in the Native environment was an old HP Laptop with a Windows-Ubuntu dual boot. The laptop has an Intel i5-6200U Processor from the year 2016. The script was coded on VS-Code and run on Terminal and the programs when they were measured ran on a fresh copy of Ubuntu OS 22.04 LTS (released in April 2022). The system is equipped with fully usable 16 GB of DDR4 RAM, a 256GB SSD, a 1TB HDD and a 4GB NVIDIA 940M Graphics card. These specifications were enough to benchmark the spark python examples as mentioned on [11]. At this stage, the limitations found along the way include (but are not limited to) choosing the right programs for the benchmark process, setting up different machines, hits-and-trials while running the script, capturing the times in the way it was understandable, finding the mean, min and max values of time taken to execute each program, etc.

4.1.2 Phase 2: Running Benchmarks on Minikube

In this phase, we run the aforementioned examples in a Minikube environment. To get results, another bash script was coded to make things efficient. Using the Script - phase 2 we can get the execution times of these programs. The main difference between the approach in phase1 and phase2 was the commands used to run these examples. In phase1 the examples are run as normal Python codes and in the second one, they are sent to the Minikube Kubernetes using the spark-submit script one by one.

- **Script explained:** This script starts the Minikube, specifies the environment, and navigates to the spark directory. It specifies the environment variable for the Kubernetes Server after creating the serviceaccount and custerrolebinding. These programs are then run and the time duration of each program is then sent to the Duration results - Script 2 file.

- **System specifications:** The same machine as phase 1 was used here with a slight modification. Minikube Kubernetes set up a local docker container which took about 10GB of the available RAM. It was made sure that the Ubuntu machine did not have any existing containers which were running or deployed. At this stage, the limitations found along the

way include (but are not limited to) testing the same on different machines - local and on the Azure cloud, the spark-submit parameters took too much of a hit-and-trials to decide, the system crashed multiple times thereby delaying the process, multiple VMs were tested to check where the spark-submit went wrong, the script was coded in bash instead of python to ensure more robust performance and correct capture of time duration for each program, etc.

4.1.3 Phase 3: Running Benchmarks using SCONE

In this phase, we run all these examples using the sconified image provided to me by the company Scontain using the Script - phase 3 (Main). The implementation is well documented on the Scontain homepage [4]. With access to the repository that lists the steps to run the spark predefined examples, the execution was made possible. To also automate the whole of this phase, we made a script that can help someone new to understand the work and start with the implementation. The script refers to 2 other scripts inside it. The excerpts and how the whole process works is explained in detail at [14].

- **Getting Access:** An important step before running these steps is to get access from Scontain GmbH. To find out more about it in detail, contact Scontain at [4] or follow the steps listed at [15]. One needs to create an access token to access the Scontain Gitlab page and there one needs to create a personal access token in order to get push access to the repository which is needed to run the script perfectly.

- **Script - Main Explained:** All the commands that are *program* specific are written in the Script - phase 3 (Main). In this, we run all the commands to run each program one by one and then send the time duration to a separate file Duration results - Script 3.

- **Script - Before Explained:** The Script - phase 3 (Before) has the commands that need to be run before specifying the name of the program that needs to be run. This has the commands to set up LAS, the image build script, the image push process, and the creation of policies.

- **Script - After Explained:** The Script - phase 3 (After) has the commands that are to run after exporting the name of the program as an environment variable as shown in lines no. 9, 17, and so on, in Script - phase 3 (Before). This script has the commands to generate a SCONE policy and calls the spark-submit script to point to the specified Azure Kubernetes cluster.

- **System specifications:** For this, a Kubernetes cluster hosted on Microsoft Azure was deployed and sent requests from a Virtual Machine (VM), also set up on Microsoft Azure. The cluster had 25GB RAM and 60GB available hard disk space. The VM had fully usable 25GB RAM and 60GB Hard disk space. The huge RAM on the VM was, however, underutilized as the main computation was done on the cluster. The machine performed fine during the computations after the Intel SGX [9] support was enabled. It was only after this, the script was able to execute without any errors. The Azure cluster was picked as it supported SGX. The reason for these high computational requirements can be better understood when we see how SCONE actually works. The need for that huge a system is that all the components, in order to protect them, are run inside the enclave - the Java Virtual Machine (JVM) the Python interpreter, the Spark context, etc. The whole concept is described in great detail in the SCONE paper [9].

  To support all of these a minimum of 32GB RAM is recommended [14] where we can also find more details that helped us choose the right machine.

Sr. No.	Program Name	Local System(s.)	K8s Minikube(s.)	SCONE(s.)
1	aft survival regression	16.055	54	149
2	als	11.909	40	202
3	bucketed random projection lsh example	17.893	56	279
4	bucketizer example	17.118	42	122
5	chi square test example	14.489	51	130
6	chisq selector example	15.801	50	141
7	correlation example	16.210	50	164
8	feature hasher example	13.577	46	118
9	pi	9.105	36	113
10	pipeline example	20.526	56	147
11	polynomial expansion example	13.540	46	121
12	quantile discretizer example	12.108	44	128
13	rformula example	14.838	49	134
14	stopwords remover example	13.913	42	122
15	string indexer example	13.622	47	132
16	univariate feature selector example	14.782	50	144
17	word2vec example	16.527	50	147

Table 1: The Benchmarks of Native, Minikube Kubernetes, and SCONE runtimes in seconds.

4.2 What tasks have you fulfilled to solve the problem?

The initial task to select the programs required a lot of local hits and trials. After that, these 17 programs were finalized for further benchmarking. The 3 phases mentioned in the previous section give a detailed overview of the approach to solving the problem at hand. Scripts have been made to automate the whole process. The time duration for each program, in each environment then obtained is shown in the next section.

5 Conclusion

5.1 What is the final outcome of your internship?

The final outcome includes but is not limited to Benchmarks for the time taken in:

1. Native environment (PC).
2. Kubernetes environment (Minikube).
3. Confidential SCONE environment in Kubernetes.

These can be summarised in the table 1 and shown in the graph 7. The results are as expected as on the native system, the programs take 14.8 seconds on average, while on Minikube Kubernetes 47.5 seconds, and when using the Sconified image it is 2:26 minutes. This means that on average the python programs chosen take 2.3x more time to run on the Minikube Kubernetes set up on the same system. These numbers go significantly up when run using SCONE as the time duration is 9x more than the time taken on the native system. And the time taken with SCONE on Kubernetes is approximately 2x more than the time taken on minikube Kubernetes.

5.2 How do you assess the solution in relation to the initial goals?

Our results were obtained as we had decided in the initial goals. There were a lot of obstacles with respect to the computing environment and the commands that needed to be run. The installation and setup looked easy on the respective websites but took more time to understand and

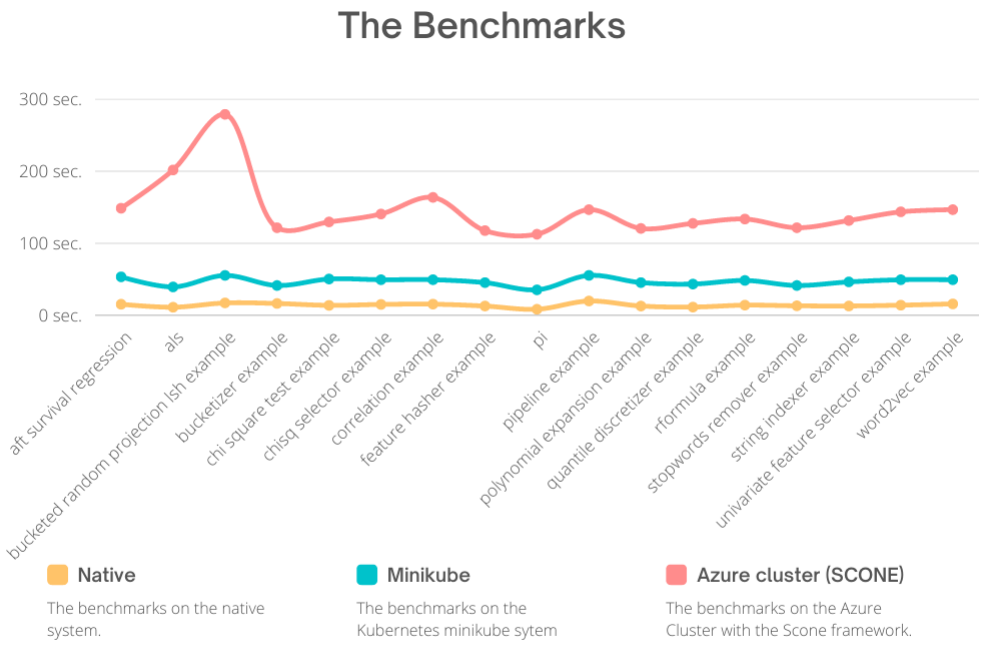

Figure 7: The Benchmarks Graph showing the difference in time taken by the programs in Native (Yellow), Minikube Kubernetes (Blue), and SCONE (Red) Environments. On the X-axis are the programs and on the Y-axis is the time taken by each of these programs.

implement. The steps when condensed too much, miss out on important factors a developer can go wrong with. We figured these things out on our own by hits and misses. The results that were obtained by the end of the project were the ones we aimed for. As mentioned above the performance overhead was as expected. This 9x time difference between the native system and the Kubernetes system with SCONE is because the base systems are very different, and the second time value is dependent on too many factors, as discussed above. The difference between the time duration values of 2.1x between minikube environment vs SCONE is because the programs need the SGX Enclave to be created before running the program. So, each of these programs needs around 1 minute and 50 seconds to create this enclave. Once this enclave is created, the programs take a similar time in general. This can also be observed in the Line Chart 7.

This also means that for all the big data processing programs that have more than 1.5B data values, the program will take approximately an extra 1:50 minutes to create the enclave, attest the system, and start the processing. This time duration, however, is fixed. The examples chosen might be the quickest to compile for all workloads. Most of the spark workloads that work on Big Data queries take over 10 hours to give the final result. As the enclave creation and SCONE overhead (in terms of time duration) were only 1:50 minutes, it might seem a large contributor when the total duration is just 3 minutes, but since it is fixed, for a program that takes more than 10 hours, it is negligible. The programs we selected are not processing any Big Data and the real difference can be seen in the ones which do so.

Future Prospects: We were able to get the differences in time of execution for each program in this internship project. One of the future prospects that we see in this project would be to find out what took more time, the SCONE or the distributed architecture of Azure, or the communication of spark instances via the encrypted communication channel. The reason for the performance degradation (and by how much) is something to research that we leave as an open topic.

References

[1] "Containers." https://www.infoq.com/presentations/microservices-docker-kubernetes-cluster/. (document), 1

[2] "Introduction to kubernetes." https://www.jeremyjordan.me/kubernetes/. (document), 3.2.3, 3.2.4, 2, 3

[3] "Scone workflows - python codes.." https://sconedocs.github.io/Python/. (document), 3.3.3, 4

[4] "Scone homepage.." https://sconedocs.github.io/. (document), 5, 4.1.3

[5] "Scone configuration and attestation service (cas)." https://sconedocs.github.io/CASOverview/. (document), 3.3.3, 6

[6] "Apache spark kubernetes page." https://spark.apache.org/docs/latest/running-on-kubernetes.html. 1.1

[7] "Apache spark installation for ubuntu." https://computingforgeeks.com/how-to-install-apache-spark-on-ubuntu-debian/. 2.1.1

[8] "Apache spark github." https://github.com/apache/spark. 2.3, 4.1

[9] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "Scone: Secure linux containers with intel sgx," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (USA), p. 689–703, USENIX Association, 2016. 3.2.1, 3.2.2, 3.3.3, 4.1.3

[10] “Kubernetes overview.” https://kubernetes.io/. 3.2.4

[11] “Apache spark main page.” https://spark.apache.org/. 3.3.1, 4.1.1

[12] “Scone workflows..” https://sconedocs.github.io/workflows/. 3.3.3

[13] “Scone las installation..” https://sconedocs.github.io/helm_las/. 3.3.3

[14] “Scone pyspark..” https://sconedocs.github.io/azure/scone-pyspark/. 4.1.3

[15] “Scontain access token..” https://sconedocs.github.io/registry/#create-an-access-token. 4.1.3