National College of Ireland

Serverless Computing: Containerizing FaaS with Docker

MSc Research Project
Cloud Computing

Taofeek-Femi Abdullahi
Student ID: 19104219

School of Computing
National College of Ireland

Supervisor: Manuel Tova-Izquierdo

Student Name:	Taofeek-Femi Abdullahi
Student ID:	19104219
Programme:	Cloud Computing
Year:	2019
Module:	MSc Research Project
Supervisor:	Manuel Tova-Izquierdo
Submission Due Date:	17/08/2020
Project Title:	Serverless Computing: Containerizing FaaS with Docker
Word Count:	5722
Page Count:	20

Signature:	
Date:	16th August 2020

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Serverless Computing: Containerizing FaaS with Docker

Taofeek-Femi Abdullahi
19104219

Abstract

Serverless computing enables cloud providers to abstract the process of spinning up servers to run programs. Developers can simply write their backend logic as pieces of code that get triggered by events such as a change in database, object storage, or request to an API endpoint. In a real world scenario, most applications require external libraries and packages in order to produce a consistent result across multiple deployed environments. This is however one of the limitations of serverless computing as some providers only provide a limited amount of packages at runtime. Developers will need to ensure that these libraries are installed in their serverless environment before deploying their code. This paper introduces ADEPS - Application Dependency Packager for Serverless. With the help of Docker, we were able to package serverless actions, with all the required dependencies, as a Docker image which is then deployed to IBM OpenWhisk. We tested our solution by deploying Spleeter, an open-source python library that decouples audio tracks. We also deployed a graph plotting function that leverages matplotlib. Our results showed that ADEPS can facilitate a more efficient way of deploying customized serverless functions to IBM OpenWhisk.

Keywords - Serverless Computing, Cloud Computing, Docker, IBM OpenWhisk

1 Introduction

The rise of serverless computing does not seem to be slowing down anytime soon. This doesn't come as a shock seeing as how serverless computing promises to eliminate the need to ever spin up, configure or manage a server. Server management has come a long way but the concept of servers is still the same. In the beginning writing an application meant that you had to deploy that application to a live server that had been configured with a certain amount of memory and disk space. You also needed to set up a network topology to ensure that the outside world can communicate with your application. Soon enough, virtualization was discovered. This meant that you no longer had to waste resources as you can provision your surplus resources to a whole new operating system. Productivity seemed to improve until server management started to take over core operations and therefore we had to move all our resources to the cloud.
The cloud is basically a cluster of servers in a data center. The current cloud computing landscape adopts a "data center" approach where dedicated servers are leveraged to provide services (Mengistu et al.; 2017). Cloud providers rent servers and resources to end users for a price. The ease and reliability of cloud computing is the reason why application development has become so popular. Most Cloud Service Providers(CSPs)

promise a 99.9% uptime of servers which is commendable seeing as how that might not be achievable on-premise. CSPs also provide resources such as Load Balancers, Elastic Container Services, to help with scalability of applications.
If applications need servers to run, where then does the term Serverless come from? "Serverless Computing" is a misnomer. The term "serverless" does not actually mean there are no servers. It was a term coined by the developers of this technology to define the abstraction of servers from developers. In a typical serverless scenario, the servers that run the application code of choice are deployed and run in containers following an event triggering the invocation of that function. Think about serverless computing as zero management of servers.
Containerization is a form of virtualization that happens in the operating system. The guest operating system is packaged and ran as an application in the host operating system. Containerization is the key element of serverless since applications run in containers can easily be spun up or shut down. With the ability to spin up containers that are loaded with application dependencies at runtime, serverless computing promises automatic scalability of application logic. There are several serverless offerings. AWS leads the pack with AWS Lambda being the most popular serverless offering. Others include Apache Openwhisk, Microsoft Azure Functions, Google Cloud Functions e.t.c.

1.1 Motivation

Application logic that handle tasks that scale quickly in a short time, such as, food ordering at peak hours, viewers logged in to a virtual conference, are typical use cases for serverless computing. CPU intensive tasks such as image manipulation are also examples of where serverless computing is applicable.

This process of outsourcing application logic as mentioned above, prevents physical resources from being clogged up which can lead to a slow response time or in worse scenarios, a crash of the application. Developers can simply deploy their application logic to a serverless environment which gets triggered when an event occurs. Serverless platforms provide API endpoints that returns a json response on invocation. However, a limitation which is often found on serverless platforms is that most providers only provide a limited amount of dependencies at runtime. In some cases, the dependencies required for the execution of the application logic is not provided at runtime.

We therefore lead this research paper with our research question. **"How can application development with serverless computing leverage third party dependencies by using Docker containers on IBM Openwhisk?"**

1.2 Contribution

This research paper contributes the following:

- A comprehensive literature on cloud computing, serverless computing and Docker containers.
- A critical review of related work.
- ADEPS - Application Dependency Packager for Serverless. A tool that deploys a Docker image as a serverless function on Apache Openwhisk.

The rest of this paper is structured as follows: Section 2 is where we take a look at background and related work in this area. In Section 3, we look at the methodology employed in this research. Section 4 details the design decisions taken in this work. We discuss our implementation in Section 5, evaluate our work in Section 6 and this paper comes to a conclusion in Section 7.

2 Background and Related Work

2.1 Virtualization Vs Containerization

2.1.1 Virtualization

Virtualization enables software applications to behave as if they were their own operating system (Barrett and Kipper; 2010). This virtual operating system which is commonly known as the "Guest OS" acts as if it was an independent system with its own binaries and libraries. The guest OS is usually provisioned a virtual memory, virtual hard disk and even a virtual memory from the host operating system. Virtualization, being the pre-cursor to cloud computing, helps to solve the problem of a surplus of resources on physical machines. A windows system can easily provision resources for 2 or more Linux hosts depending on how much resources it has. Virtualization also makes it possible to back up the entire state of a system which can be part of a larger disaster management workflow. You can simply have copies of your machine in its entire state and in case of a crash, a backup can easily be restored.
Virtualization was made possible by an advancement in x86 servers and it works by placing the virtualization layer underneath the OS which then acts as a VMM (Virtual Machine Monitor) to manage and control the OS (Barrett and Kipper; 2010)

2.1.2 Containerization (Docker)

Virtualization is the technology that paved way for containerization. The need to containerize applications emerges as an application scales and there becomes a need to set up different environments such as development, testing, debugging, production e.t.c. It becomes an issue when an application is not consistent across multiple environments simply because the application that depend on a particular version of a dependency is deployed to an environment with the wrong version of that dependency and thus causing the application to crash or produce inconsistent results.
The tediousness that comes with replicating an application runtime environment across numerous setups is why Docker has become so popular.
Docker provides developers with the capabilities to run applications in isolated runtime environments called containers [1]. These containers or Docker containers are lightweight operating systems configured with the necessary libraries and bins required to run that application. The lightweight nature of Docker containers (which is due to the absence of an hypervisor since they are run directly within the host machines kernel) is what makes it possible to deploy numerous containers to a host machine provided there are enough resources on the host machine to cater to all the containers deployed.
By packaging up an application in a Docker container, consistency of every deployed copy of that containerized application is assured.

[1]Docker Overview: `https://docs.docker.com/get-started/overview/`

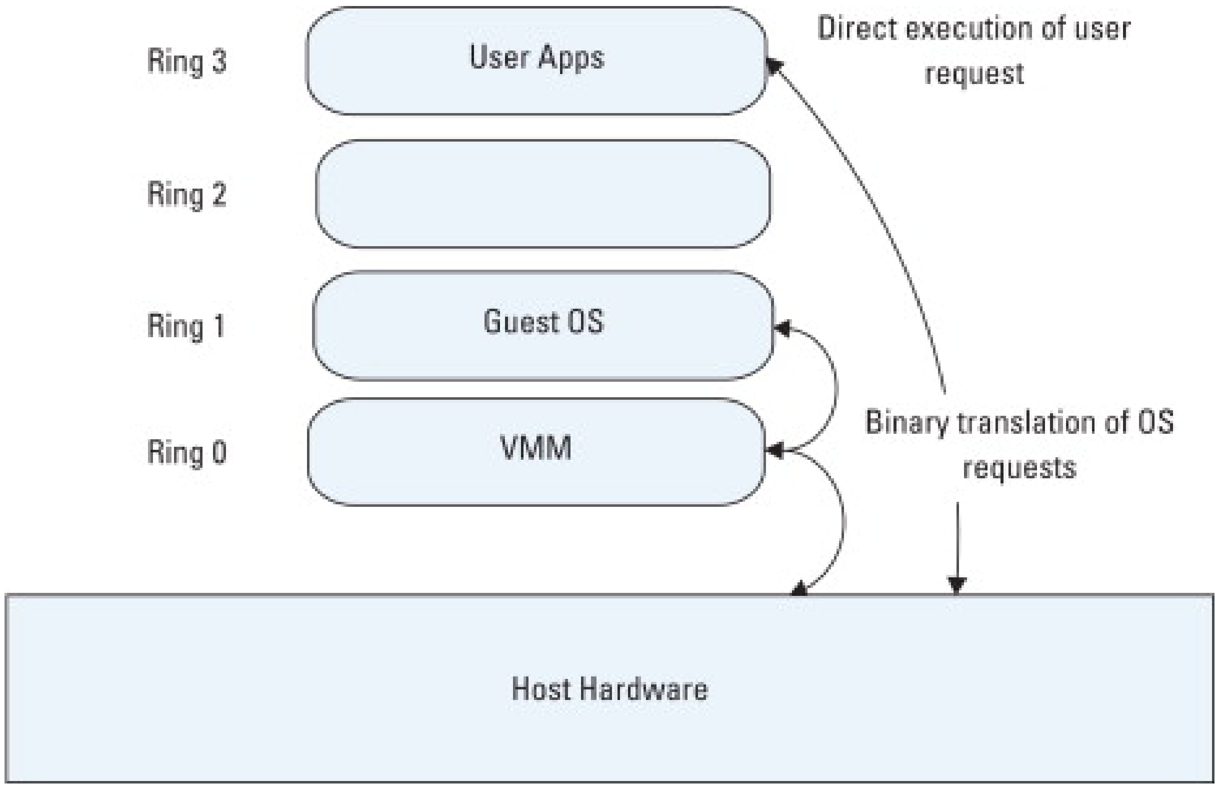

Ring 3
User Apps
Direct execution of user request
Ring 2
Ring 1
Guest OS
Ring 0
VMM
Binary translation of OS requests
Host Hardware

Figure 1: Full Virtualization (Barrett and Kipper; 2010)

Docker extends LXC [2] with kernel-level API which runs processes in isolation together with the application-level API. To thoroughly isolate the contained applications view from the host OS, Docker employs the use of linux namespaces. Application view is also isolated from user IDs, network, file systems and process trees (Bernstein; 2014).

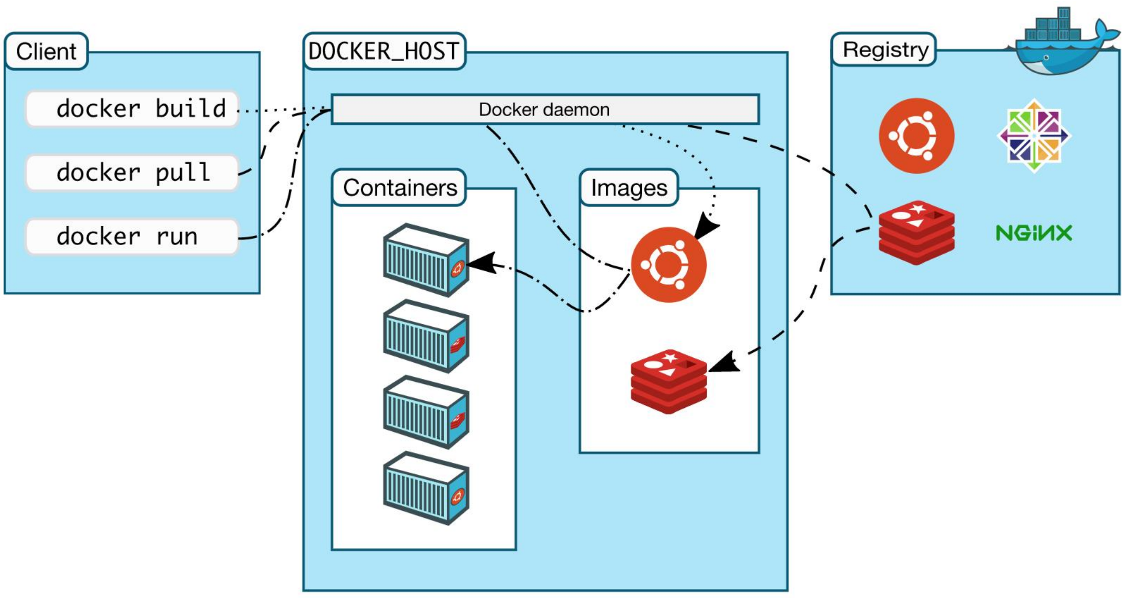


Figure 2: A representation of the Docker architecture

[2]LXC Introduction: `https://linuxcontainers.org/lxc/introduction/`

Gomes et al. (2018) introduced uDocker which is a tool that enables the execution of Linux containers in user mode. Docker simplifies the management of Linux containers and also, provides an efficient container repository such as Docker Hub. The layered file system enables a new image to be created from an existing one by reusing or sharing its layers. This process of sharing layers can help minimize downloads and save storage space. (Gomes et al.; 2018)

In SCAR, Perez et al. (2018) addressed the issue of the restriction imposed on serverless functions by their providers. They introduced a framework to create highly parallel event-driven functions with customized runtimes. They leveraged on top of AWS Lambda. They tested their solution on an image processing use case and reported that SCAR provided much more efficient deployment of high throughput computing workloads to AWS. They went further by implementing a cache within the Docker container loaded into the serverless container for execution. This technique allowed for reduced latency in subsequent requests in concurrent function execution.

To conclude their work, Perez et al. (2018) suggested the adaptation of their tool to other serverless offerings. They also highlighted that to reduce boot up time of containers, through the periodic invocations of serverless functions, the SCAR framework can reduce the latency issue faced in serverless.

2.2 Serverless Optimized Containers

In a bid to understand the inner workings of containers, Oakes et al. (2018) proposed bind mounting for the population of a directory when allocation the file system for a container. This comes as an alternative to the use of union file systems. Bind mounting is the process of making the same directory visible at various locations which becomes a more efficient solution as opposed to populating directories by physically copying files which can be a slow process. They were able to conclude that at scale, bind-mounting is twice as fast as the system adopted in Docker. Bind mounting into a container involves binding the users unprivileged ID as opposed to the ID in the users namespace to the host (Priedhorsky and Randles; 2017). This method makes it a safe process.

Their proposed solution SOCK was implemented by replacing Docker as the primary container for OpenLamda. There are 2 major operations associated with SOCK; (1) Effective sandbox initialization that enables all workers to achieve high throughput. (2) Reduced latency in library importation by python handlers. Both operations will have to be considered in order to make serverless more ideal for applications.

The distinct Zygote provisioning strategy adopted by SOCK displays the following characteristics; Determining the set of pre-imported packages based on their usage and at runtime. Maintaining multiple zygotes which can scale to a large package set of pre-imported packages. It also handles the complete integration of provisioning with containers and finally, securing processes from malicious packages they did not import.

2.3 Limitations in Serverless

For applications with a compute period longer than 15 minutes, data-rich applications or distributed computing Hellerstein et al. (2018) proposed that serverless technology is not ideal for such workloads. They further added that serverless computing is best suitable for computations that are embarassingly parallel. Such computations have been described in ExCamera (Fouladi et al.; 2017) which is a program that attains low-latency video encoding by separating the video encoding process into 2 parts. Concurrent invocation in serverless can be leveraged by applications that can be divided into embarassingly parallel smaller tasks (Lee et al.; 2018). Hellerstein et al. (2018) further reported that the reason why serverless computing is considered "less" is as a result of their limited lifetimes. Running functions in warm containers can be a solution to this but often leads to a waste of resources as functions are invoked periodically even when they are not used. In addition, state management in serverless computing is another glaring limitation of this paradigm since subsequent invocation of that function might not run in the same container. A feasible solution is to implement an external storage to store state and reading that state into the function execution. This however can introduce latency into the system thereby causing function execution to be delayed more than necessary.

To achieve high throughput, fine-grained interleaving of various functions will have to be leveraged by serverless applications. This can thus result in under performance of some components within the application. (Shahrad et al.; 2019).

2.3.1 Understanding Cold Start in Serverless

Discussing the cold start issue in serverless, Baldini et al. (2017) reported that it is critical to develop techniques to minimize this issue while still scaling down to zero. They further identified it as a system level challenge and pointed out the inability to use declarative approaches to control deployment as a devOps problem. To address this issue, Pérez et al. (2019) used an high-level declarative language to outline the deployment of their OSCAR framework which made deployment reproducible across cloud runtimes.

3 factors influencing cold start latency was highlighted by Jonas et al. (2019) These factors are outlined as following (1). Time it takes to invoke a serverless function, (2). Time it takes to load required modules and libaries, and (3). Time it takes to load application-specific initialization in the users code. All these factors influence the cold start latency in serverless.

van Eyk et al. (2019) pointed out that HTTP requests, in the form of an event, reaches a function router which leads to the deployment of a function instance. A cold start is occured if there are no function instances to handle that invocation because a new function instance will have to be created. They suggested having a function instance already available will help mitigate the cold start issue. However, Xu et al. (2019), presented a contrasting conclusion by pointing out that this strategy leads to a waste of resources.

As an application scales, cold start latency can be worsened. As discussed in Mahajan et al. (2019), each new function execution will incur this cold start latency. This is especially more noticeable in concurrent requests where the initialization of multiple function containers is required to handle multiple function invocations.

The adoption of serverless technology for a wider range of application will be delayed by by the cold start issue. (Xu et al.; 2019). They proposed strategies to mitigate cold start latencies. These strategies are called; Adaptive Warm-Up (AWU) strategy and an Adaptive Container Pool Scaling (ACPS).

Using time series prediction, AWU can predict, to a certain degree of accuracy, the time functions will be invoked. AWU goes further to warm up containers in advance to handle the incoming invocations. ACPS manages resources to ensure efficient usage and minimal wastage. ACPS also provides support to AWU. Future work in container provisioning using time series prediction could discuss variable timeframes for functions with unorthodox invocation history.

2.3.2 Improving Latency in Serverless

Container start up latency issues were tackled using a novel approach in the work of Akkus et al. (2018). Their proposed system, SAND, attempts to lessen function interaction latencies. Showing similarities is the work of Du et al. (2020) where they introduced Catalyzer, which restores a function instance from a checkpoint, eradicating the initialization process of a function instance.
(Akkus et al.; 2018), pointed out that each serverless function executed on a different container instance. They further highlighted the lack of interaction between between functions as that there is no way for a function to be aware of the origin of the trigger event which results in functions going through the full end-to-end path of function execution thereby increasing latency.
Their sandbox mechanism was able to achieve 2 levels of isolation: 1) isolation between applications, 2) isolation between functions used in the same application.
In their discussion, they reported that a large percentage of container startup latency is connected to the installation of libraries during container initialization. Lazy loading of libraries to reduce latency as described in Harter et al. (2016) was suggested.
Evaluating their solution, startup time until function execution was measured. Sandboxing setup that included Docker and processes developed in C, Go, Python, Java, NodeJS. Their results showed that starting an application or process in a warm container is much faster than invoking a new container. This result has been highlighted as future work in Perez et al. (2018).

The works described above tells us that the serverless computing paradigm is still yet to be fully explored as solutions to some of the problems addressed above has not come to light yet. It also justifies the direction we have taken in this research work. The contrast between usability of cloud platforms is one of the reasons why developers working on IBM OpenWhisk might not be eager to switch to a different provider like AWS Lambda. There is also the issue of vendor lock-in which deters developers from easily migrating their workloads between cloud provider. For this reason, it is critical to also develop tools for developers on other platforms so they can take full advantage of the potential of serverless computing.

3 Methodology

This research paper details an Application Dependency Packager for Serverless (ADEPS) tool. The goal of this tool is to deploy a serverless function written in Python to IBM OpenWhisk.
Running custom containers on serverless runtimes as described in this work has been seen in SCAR (Perez et al.; 2018) where they packaged Docker containers as custom runtime for serverless functions on AWS Lambda. The work detailed in this paper follows a similar methodology for IBM OpenWhisk.

ADEPS packages a serverless function loaded with all the necessary dependencies required to run as a Dockerfile and deploys to IBM Openwhisk Cloud Functions through the IBM Cloud CLI. To enable this functionality, ADEPS is built as a CLI tool which can be run as a command in a Linux operating system. This tool is built with developers

in mind which what motivated the decision to build ADEPS as a CLI tool as opposed to a Graphical User Interface.
The motivation behind this research is to develop a tool that made deploying a serverless function as easy as writing a command such as: *adeps command [<args >]*

3.1 Tools and Technologies

The following tools and technologies were utilized in carrying out this research.

- IBM OpenWhisk (IBM Cloud)
- IBM Cloud CLI
- Docker
- Python
- Linux

IBM OpenWhisk
IBM OpenWhisk is IBMs cloud offering. It is built on top of Apache OpenWhisk which is an open source self hosted serverless enviroment. IBM OpenWhisk enables us to execute functions/actions in response to triggers. The diagram below represents the OpenWhisk execution sequence from client request to function execution.

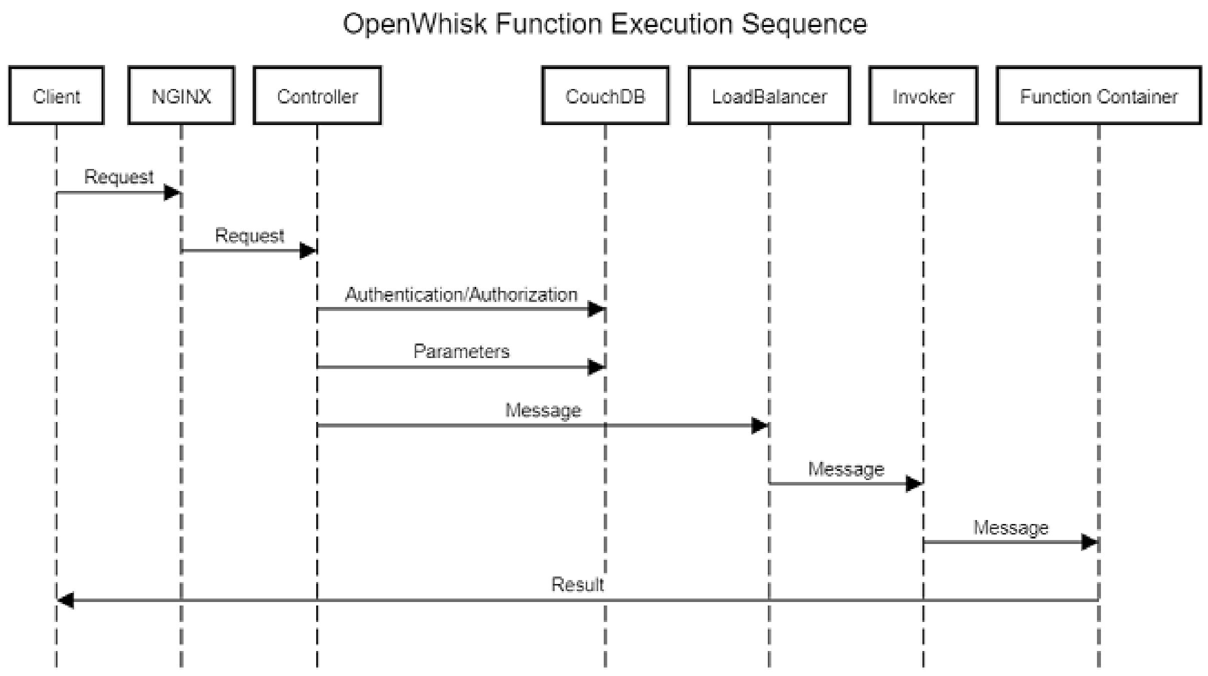


Figure 3: OpenWhisk Function Execution Sequence

OpenWhisk supports languages such as NodeJs, Python, Go, Java, Scala, Ruby.

IBM Cloud CLI
IBM Cloud CLI provides a REST API command line interface that allows us to send HTTP requests and interact with our services on the IBM cloud.

Docker
Docker creates execution environments in the form of containers for us to execute our functions. Docker is basically OS level virtualization that allows us to load our runtime environment and all the necessary packages needed for the function execution into one container. Serverless computing is built on the foundation of containers and OS virtualization. This type of virtualization is what makes it possible to provision and deprovision runtime environments at will.

Python
ADEPS is written in Python. The choice of this programming language is due to the ease of providing the functionalities required in this project. The main python library used in this work is *argparse*. Argparse [3] is a python module that parses arguments obtained from command-line input. The goal of argparse is to help create user friendly command line tools. Other python libraries used include docker, os, sys, subprocess, getpass.

4 Design Specification

In this section, we detail the design decisions that were taken during the development of this tool.

4.1 ADEPS Client

ADEPS relies on DockerHub as a repository where it pulls the docker image of choice before deployment to OpenWhisk. For this reason, the program provides a user input prompt to allow the user enter the correct authorization credentials.

To deploy a serverless function with ADEPS, a user has to provide some key arguments that are required for the function to successfully carry out the deployment. This program leverages "argparse" to parse arguments that are supplied at the command line prompt. An "adeps deploy" command should be suffixed with the following arguments.

- -d / –dockerhub : User should indicate true if they will like to build their Dockerfile and push to a remote repository. If the user already has a docker image in the Dockerhub repository, the user should indicate false.

- -r / –reponame : User should provide the name of their Dockerhub repository. This argument is still required even if that repository is currently empty. However, a repository must be created before attempting to deploy the serverless function.

- -t / –tagname : A tagname is required for every docker image pushed to a Dockerhub repository. This argument should also be provided when using the deploy command.

- -f / –functionfile : OpenWhisk requires a function file. This is the file that contains the code that will be executed upon invocation of the function. This function must be named "main" as any other name will lead to the function not being invoked. ADEPS requires the functionfile to exist within the same folder the deploy command is called from. In addition to the functionfile, The Dockerfile should also be located in the same folder if the user chooses to build and push a new docker image to the Dockerhub repository.

[3] Argparse `https://docs.python.org/3/library/argparse.html`

- -a / –actionname : Every serverless function deployed to OpenWhisk must have a name. This argument must be provided when using the deploy command.

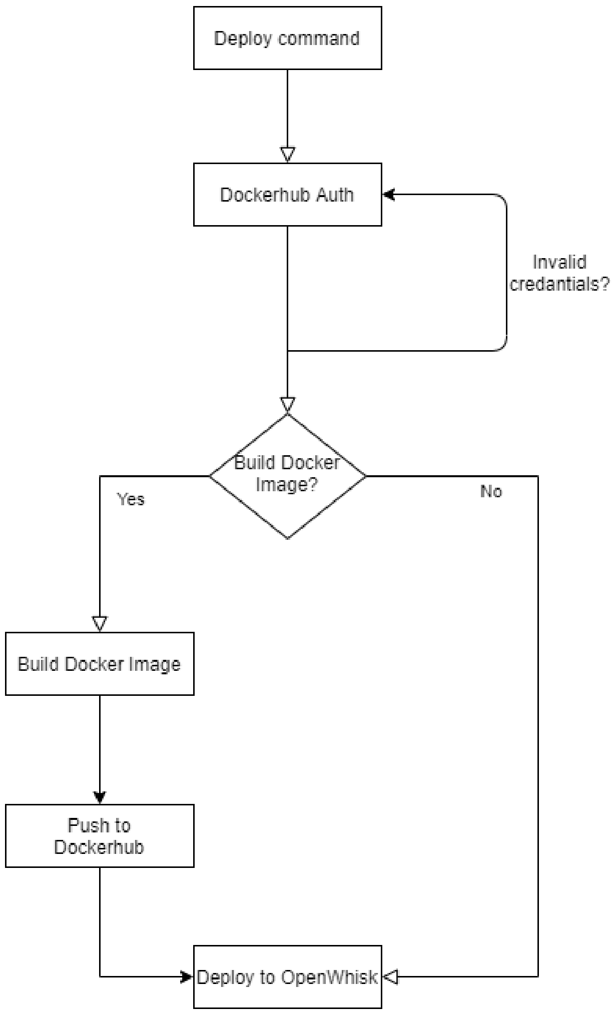


Figure 4: ADEPS deploy flowchart

4.2 Deploying Serverless Function

ADEPS works hand in hand with the IBM Cloud CLI. A user is required to have the CLI installed and configured and also authorized. IBM Openwhisk requires a user to specify a "SPACE" and "DEV" as part of the resources in the users control. This namespacing allows a user to have different resources allocated to different namespaces.
Upon successful image build, ADEPS makes a call to OpenWhisk and deploys the serverless function. The user will see the new serverless function in their list of functions ready for invocation.

Algorithm 1 ADEPS pseudo code to deploy functions

```
Result: Deploy serverless function to IBM OpenWhisk
input: args
while user_logged_in == false do
    login_to_dockerhub()
end
if dockerhub == true then
    build_image()
    push_image_to_repo()
end
deploy_to_openwhisk()
```

5 Implementation

In this section, we discuss how ADEPS was implemented as part of the workflow for deploying a serverless function.
ADEPS was written and developed in the Python programming language and makes use of python modules such as os, docker, argparse, sys, subprocess. Interacting with the operating system shell is a key functionality of this application and that functionality is made possible using the os module.

5.1 Implementing with Docker SDK

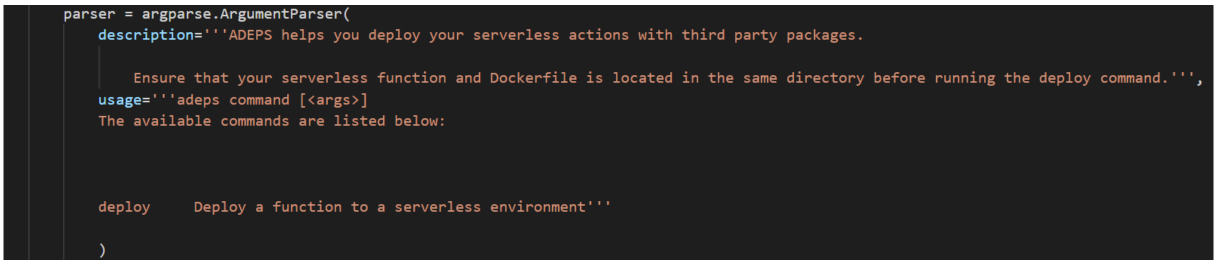
The Docker SDK handled all the operations involving Docker. These operations include: user authentication, image build and image push to repository.

A user is prompted to enter their Dockerhub credentials for authentication. Invalid credentials will cause the prompt to show again until the valid credentials are provided.

Once the user has provided their valid credentials, an image is built from the Dockerfile in the same directory where the deploy command is run. In the process of implementing this program, we discovered that the Dockerfile required for a function deployed with ADEPS has to contain the IBM Cloud image as the Linux runtime. This might be due to the configurations that are necessary to integrate that function with the IBM infrastructure. For a Python deployment, the Dockerfile should contain an image similar to this "ibmfunctions/action-python-v3.7".
Finally, a successful image build in turn leads to that image being pushed to the Dockerhub repository provided by the user.

5.2 Implementing with Argparse

Argparse as mentioned in earlier sections is a Python module that provides command line interface capabilities to a python application. This module parses arguments supplied in the terminal into values that can be used within the application. Argparse has an ArgumentParser() object that accepts "description" and "usage" as parameters. The description parameter holds the value for the description of the CLI tool. This is where the developer provides a brief description of the tool. The usage parameter on the other hands denotes how the tool is to be used and what commands are available.



```
parser = argparse.ArgumentParser(
    description='''ADEPS helps you deploy your serverless actions with third party packages.

        Ensure that your serverless function and Dockerfile is located in the same directory before running the deploy command.''',
    usage='''adeps command [<args>]
    The available commands are listed below:



    deploy     Deploy a function to a serverless environment'''

    )
```

Figure 5: Argparse Argument Parser

5.2.1 The Deploy Command

The deploy command is a sub command of "adeps" made possible with the argparse module. To implement this functionality, the constructor of the ADEPS class initiates the first level of this multi level command structure. Within the constructor, The description and usage of the ADEPS tool is added. In our work, within the usage value, we provided the available lists of commands as seen in figure 5. We consider this the first level of the command structure. In the second level, the user will have to provide the arguments required by the "deploy" command. These commands are detailed in section 4. To implement this, we first check if the command provided is valid, if not, we throw an error to the user. On input of the valid command, we execute that command function within our class, in our case, the function will be deploy(self). Thus, all arguments entered after the deploy command will be parsed by the argparse instance within the deploy function. For each argument that will be parsed, argparse allows us to add a "required=True" option to make this a required argument.

6 Evaluation

In this section, we discuss the experiments that were used to evaluate ADEPS. ADEPS provides a command line client for a user which allows the user to deploy a serverless action, packaged with all necessary dependencies, to OpenWhisk. Our experiments used different applications that required unique packages that were not exclusive to the OpenWhisk runtime.

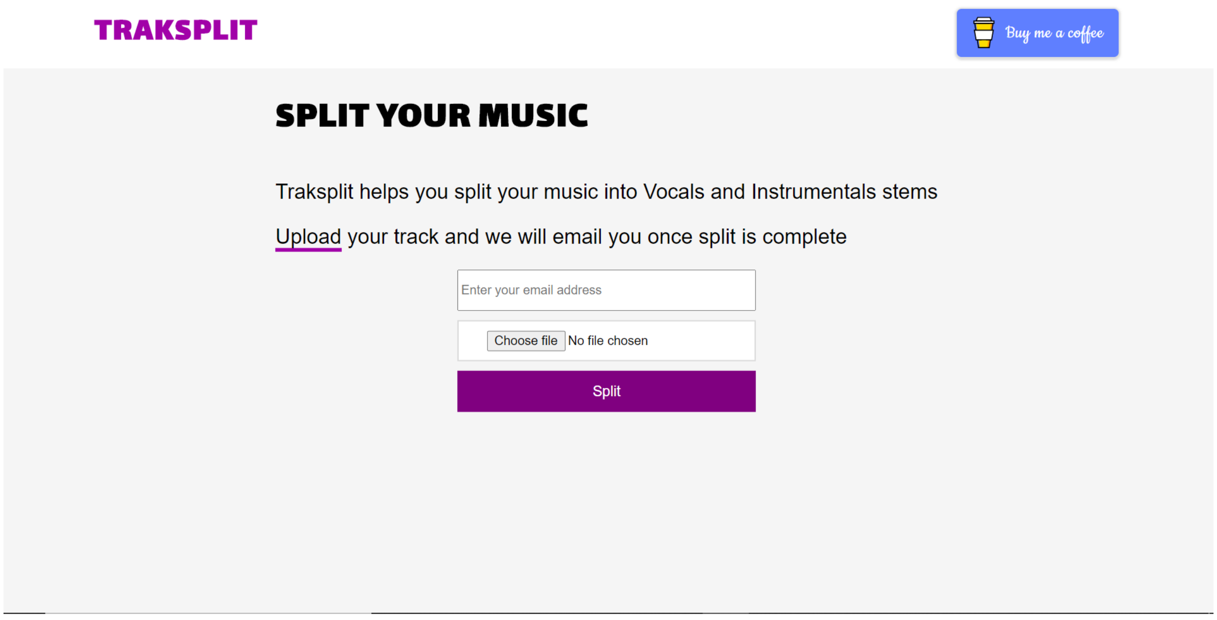
6.1 Case Study 1

6.1.1 TrakSplit

Traksplit is an application built with NodeJS that allows a user split an .mp3 song file into multiple parts. This application leverages Spleeter (Hennequin et al.; 2020), an open source seperation library built in Python and Tensorflow and comes with pretrained models to generate a variety of seperations. Spleeter involves high performance computing in order to accurately analyze and split the source audio file. The workflow of Traksplit can be described below:

- User uploads file through NodeJs app.
- File is stored on an IBM Cloud storage bucket.
- Upload automatically triggers invocation of serverless function.

- File name stored in parameters is extracted and transformed.
- File is downloaded from bucket to local machine.
- Spleeter splits file on local machine into 2 new audio stems.
- New stems are uploaded to a Dropbox storage.
- File download links is extracted from metadata of files.
- New files are sent to user by email.



Traksplit helps you split your music into Vocals and Instrumentals stems

Upload your track and we will email you once split is complete

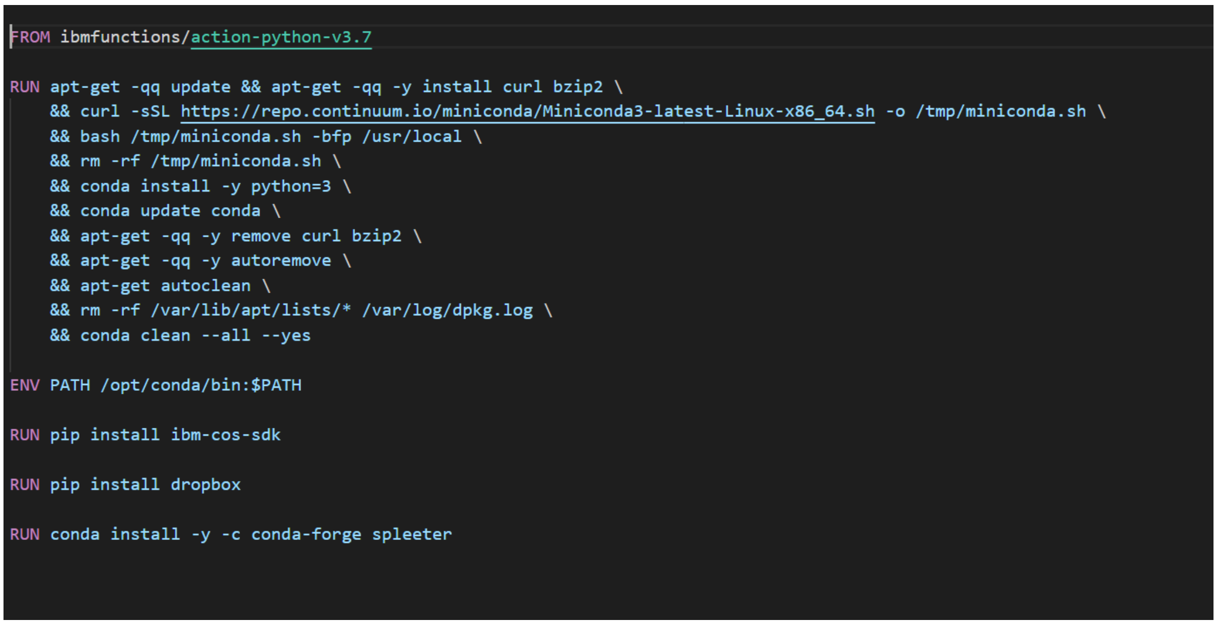
Figure 6: Traksplit Interface

Traksplits function logic is written in python and heavily depends on being able to use Spleeter as part of this workflow. For this reason, ADEPS was used to package Spleeter as parts of the libraries contained in a Dockerfile and was deployed to IBM OpenWhisk.

Traksplit depends on Spleeter which in turn depends on Conda. Spleeter is installed with Conda alongside numerous libraries such as tensorflow, ffmpeg, conda-forge etc. These libraries need to be available at runtime which is why they have to be packaged as a Docker container so that when that container is replicated, the same runtime is guaranteed.

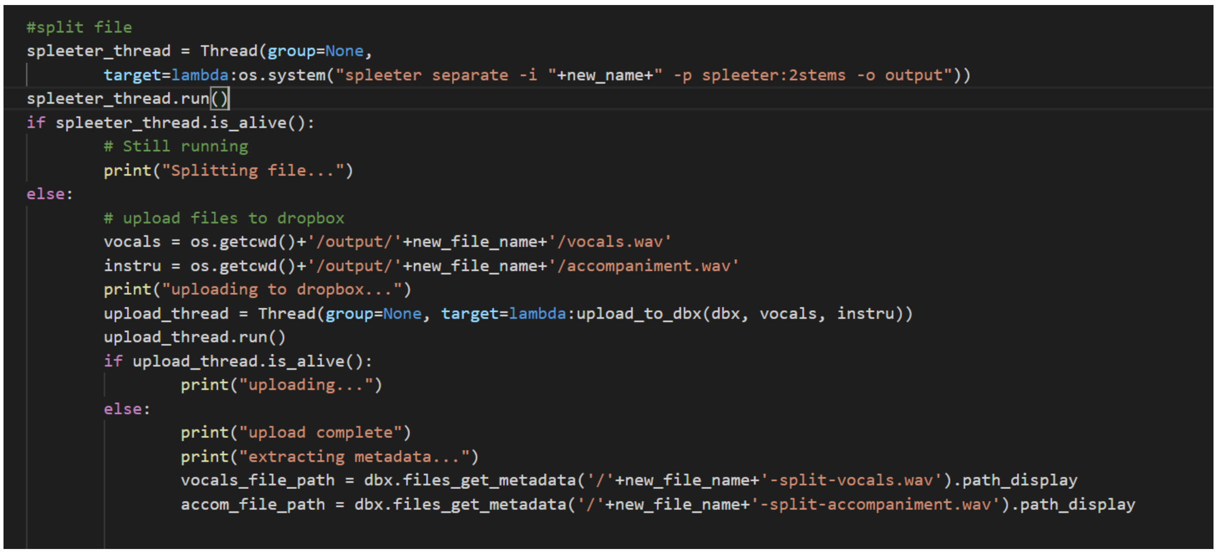
Spleeter commands are executable commands that need to interact with the shell of our operating system. For this reason, we used the python module "os" to make calls to our shell and by using a thread, we are able to make sure that the processes are executing asynchronously. We tested our local action with test data using a virtual machine of specification:

- Operating System: Ubuntu (64 Bit)
- Memory: 4830MB
- Video Memory: 16MB



```
FROM ibmfunctions/action-python-v3.7

RUN apt-get -qq update && apt-get -qq -y install curl bzip2 \
    && curl -sSL https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh -o /tmp/miniconda.sh \
    && bash /tmp/miniconda.sh -bfp /usr/local \
    && rm -rf /tmp/miniconda.sh \
    && conda install -y python=3 \
    && conda update conda \
    && apt-get -qq -y remove curl bzip2 \
    && apt-get -qq -y autoremove \
    && apt-get autoclean \
    && rm -rf /var/lib/apt/lists/* /var/log/dpkg.log \
    && conda clean --all --yes

ENV PATH /opt/conda/bin:$PATH

RUN pip install ibm-cos-sdk

RUN pip install dropbox

RUN conda install -y -c conda-forge spleeter
```

Figure 7: Dockerfile for Traksplit fuction



```
#split file
spleeter_thread = Thread(group=None,
        target=lambda:os.system("spleeter separate -i "+new_name+" -p spleeter:2stems -o output"))
spleeter_thread.run()
if spleeter_thread.is_alive():
        # Still running
        print("Splitting file...")
else:
        # upload files to dropbox
        vocals = os.getcwd()+'/output/'+new_file_name+'/vocals.wav'
        instru = os.getcwd()+'/output/'+new_file_name+'/accompaniment.wav'
        print("uploading to dropbox...")
        upload_thread = Thread(group=None, target=lambda:upload_to_dbx(dbx, vocals, instru))
        upload_thread.run()
        if upload_thread.is_alive():
                print("uploading...")
        else:
                print("upload complete")
                print("extracting metadata...")
                vocals_file_path = dbx.files_get_metadata('/'+new_file_name+'-split-vocals.wav').path_display
                accom_file_path = dbx.files_get_metadata('/'+new_file_name+'-split-accompaniment.wav').path_display
```

Figure 8: Algorithm to split tracks, upload stems to Dropbox and extract metadata

- Storage: 50GB

Our result showed some inconsistencies as we got accurate results on some tests and Spleeter failed to split the file on some other tests. The reason for this failure can be attributed to the system running out of memory and crashing. The intensive computing nature of Spleeter, since it has to run an analysis on a source audio file based on some pretrained models, requires large memory specifications. Surely, upon deployment of this action to IBM Cloud functions, we were unable to produce the same results we got locally due to our application running out of memory. At the time of this writing, the maximum memory that can be allotted to a cloud function on IBM OpenWhisk is 2048MB. The inabilty to produce results in IBM Cloud can also be due to the container timing out before Spleeter is done computing the results. IBM Cloud functions at the time of this writing only provides a maximum of 600s runtime.

In evaluating ADEPS using TrakSplit as a case study, we can confirm that ADEPS is indeed suitable for a function deployment to IBM OpenWhisk. However, it doesn't provide any resource management functionalities and is completely unaware of resource deficiencies of the infrastructure it deploys a serverless function to.

6.2 Case Study 2

6.3 Graph Plot with Matplotlib

To further evaluate ADEPS, we built a simple application that plots a simple chart based on some values provided. The aim of building this application is to make use of the "matplotlib" Python module.
The matplotlib library is not available on the IBM OpenWhisk Python runtime and thus it made a good use case to evaluate our tool. Our graph plot application accepts values for the X axis and values for the Y axis from a user and the following operations are performed in our serverless action.

- Extract the X and Y axis parameters.
- Plot the graph using matplotlib
- Extract the image and save to a local directory.
- Compress the saved image using Tinify and save the optimized version.
- Upload compressed image to a Cloudinary image repository.

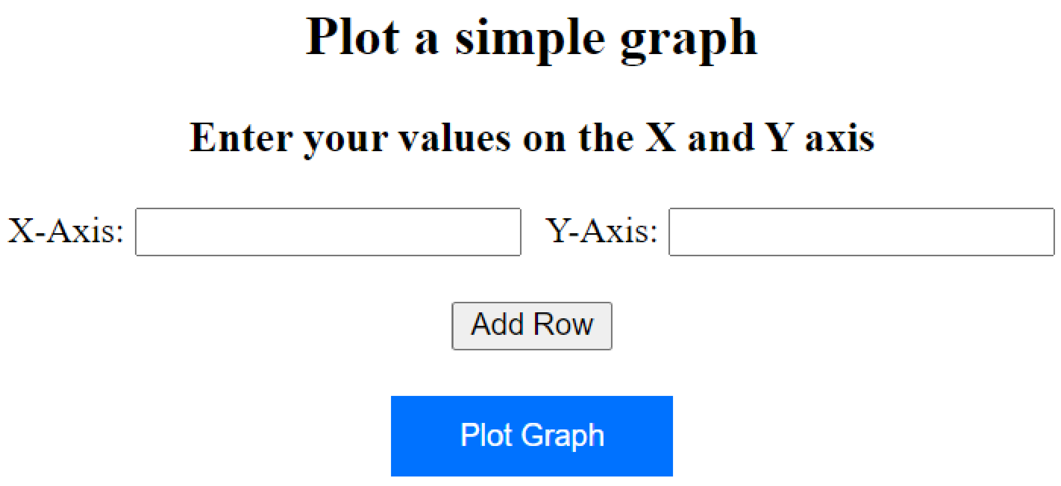
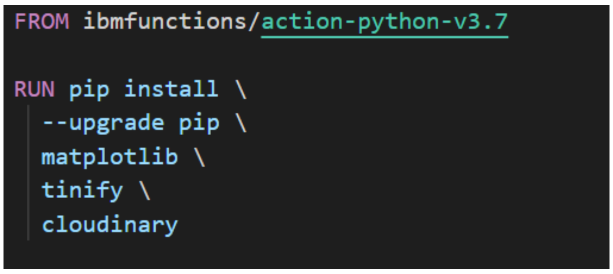


Figure 9: Graph plotting application interface

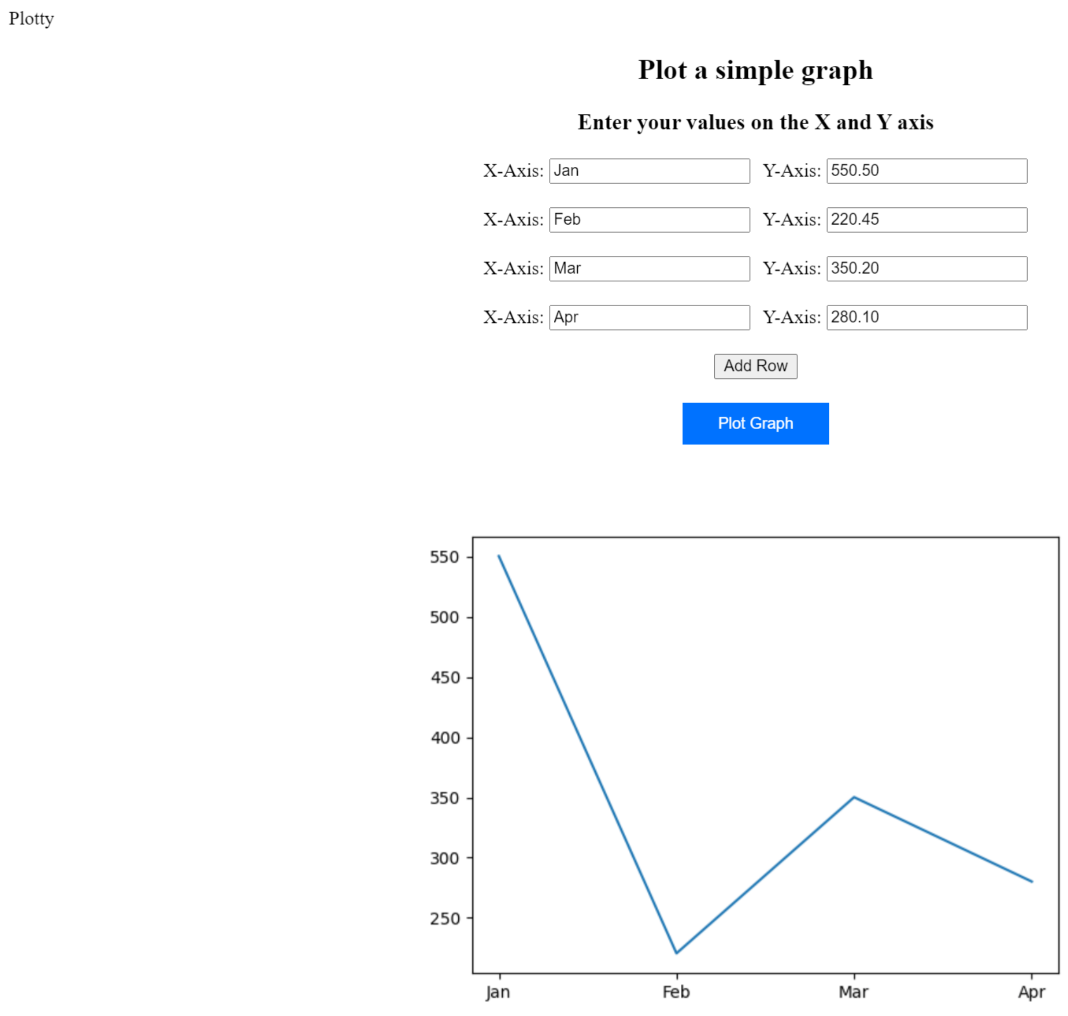
This case study also gave us the opportunity to use Tinify and Cloudinary, 2 additional Python modules that are not included in the OpenWhisk runtime. Tinify is an application the compresses images over an API and returns an optimized version of the image. Cloudinary on the other hand is an image repository where images can be saved externally to and also modified remotely.



```
FROM ibmfunctions/action-python-v3.7

RUN pip install \
  --upgrade pip \
  matplotlib \
  tinify \
  cloudinary
```

Figure 10: Graph plotting application Dockerfile

ADEPS was able to successfully package the Dockerfile as shown in Figure 10 and deploy the serverless function to IBM OpenWhisk. The function produced the same results at runtime and didn't show any inconsistencies.



Plotty

Plot a simple graph

Enter your values on the X and Y axis

X-Axis: Jan Y-Axis: 550.50

X-Axis: Feb Y-Axis: 220.45

X-Axis: Mar Y-Axis: 350.20

X-Axis: Apr Y-Axis: 280.10

Figure 11: Graph plotting application results

Our final application was able to collect input from a user, send them as a POST request to the endpoint of our serverless function and also pull the correct image from

the cloudinary repository to display to the user. ADEPS was able to reduce the total deploy time for this function by upto 40%
Using the graph plotting application, we were able to plot a graph of the invocation time of 10 invocations. The results were obtained on the console of IBM Cloud Functions.

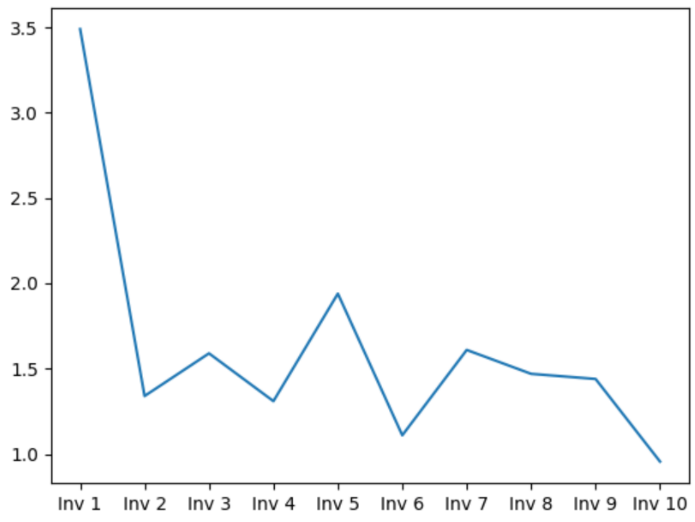


Figure 12: Latency profile of graph plotting application

6.4 Discussion

From Fig 12, we can see the latency profile of 10 subsequent invocation of our graph plotting serverless function. The first invocation showed a startup time of 3.5s and maximum invocation time after that is 1.9s. This obvious difference in startup times can be attributed to the cold start latency issue as described in earlier sections. What this further tells us is that IBM OpenWhisk keeps a warm container running for future invocations after the first invocation. This method of mitigating cold start issue has also been discussed in earlier section.

In the process of evaluating of this tool, we came across some limitations. Unfortunately some of the limitations are associated with the entire serverless paradigm and serverless cloud providers. Functions that required high start up time simply couldn't work even with their dependencies properly packaged. The reason for this is due to the limited runtime of each serverless function on OpenWhisk. As mentioned earlier, the maximum runtime for an OpenWhisk function is 600s(10m). In addition, the maximum memory space that can be allocated to a serverless function on OpenWhisk is 2048 and so for applications - for example model training with machine learning - require a large amount of memory and simply might produce consistent results with serverless computing.
ADEPS didn't show any inconsistencies when deploying resource efficient functions that execute within the 600s timeframe. Testing functions locally is always a good idea because serverless functions can be sometimes hard to debug once they are in their runtime environments. If the function works as supposed to on a localhost, same behavior should be expected in a serverless runtime environment.
Serverless gives us the opportunity to really decouple large architectures into microservices

that are independent of each other. The future of serverless computing is bright and that future should involve better resource allocation as more resource intensive workflow migrate to serverless deployments.

7 Conclusion and Future Work

In this research paper, we introduced ADEPS - a tool to deploy serverless functions with dependencies to a serverless runtime environment. With this tool, we were able to take advantage of the full potential that serverless computing offers. Writing backend logic and deploying them to an environment that's easily scalable is transforming the way we write and think about our applications. Function invocation as a response to an event trigger are ideal for short bursty workloads, automation workflows and even IOT and machine learning workflows.
The purpose of this research was to propose a solution to the restriction of packages on the IBM OpenWhisk runtime. As highlighted on our research questions, it was imperative to develop a solution that leverages Docker containers as serverless runtime environments. This work succeeds in packaging applications with compute time less than 10 minutes and RAM specifactions under 2GB. However, for resource intensive workloads, ADEPS still manages to package and deploy the function as required but the function might exhibit some inconsistencies at runtime. ADEPS fails to accomodate functions developed in other programming language. As at the time of this writing, ADEPS only packages functions written in Python. Functions written in other languages might show some inconsistencies. Integrating a deploy pipeline for programs written in other programming languages can be highlighted as future work in this field. Future researchers can also implement resource management functionalities that proposes efficient ways to deploy resource intensive workflows.

References

Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P. and Hilt, V. (2018). SAND: Towards high-performance serverless computing, *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, pp. 923–935.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A. and Suter, P. (2017). *Serverless Computing: Current Trends and Open Problems*, pp. 1–20.

Barrett, D. and Kipper, G. (2010). 1 - how virtualization happens, *in* D. Barrett and G. Kipper (eds), *Virtualization and Forensics*, Syngress, Boston, pp. 3 – 24.

Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes, *IEEE Cloud Computing* **1**(3): 81–84.

Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q. and Chen, H. (2020). Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting, *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, Association for Computing Machinery, New York, NY, USA, p. 467–481.

Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G. and Winstein, K. (2017). Encoding, fast and slow: Low-latency video processing using thousands of tiny threads, *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, USENIX Association, Boston, MA, pp. 363–376.

Gomes, J., Bagnaschi, E., Campos, I., David, M., Alves, L., Martins, J., Pina, J., López-García, A. and Orviz, P. (2018). Enabling rootless linux containers in multi-user environments: The udocker tool, *Computer Physics Communications* **232**: 84 – 97.

Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. (2016). Slacker: Fast distribution with lazy docker containers, *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, USENIX Association, USA, p. 181–195.

Hellerstein, J. M., Faleiro, J. M., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A. and Wu, C. (2018). Serverless computing: One step forward, two steps back., *CoRR* **abs/1812.03651**.

Hennequin, R., Khlif, A., Voituret, F. and Moussallam, M. (2020). Spleeter: a fast and efficient music source separation tool with pre-trained models, *Journal of Open Source Software* **5**(50): 2154. Deezer Research.

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Menezes Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J., Popa, R. A., Stoica, I. and Patterson, D. A. (2019). Cloud programming simplified: A berkeley view on serverless computing, *Technical Report UCB/EECS-2019-3*, EECS Department, University of California, Berkeley.

Lee, H., Satyam, K. and Fox, G. (2018). Evaluation of production serverless computing environments, *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 442–450.

Mahajan, K., Mahajan, S., Misra, V. and Rubenstein, D. (2019). Exploiting content similarity to address cold start in container deployments, *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '19, Association for Computing Machinery, New York, NY, USA, p. 37–39.

Mengistu, T., Alahmadi, A., Albuali, A., Alsenani, Y. and Che, D. (2017). A "no data center" solution to cloud computing, *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pp. 714–717.

Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. (2018). Sock: Rapid task provisioning with serverless-optimized containers, *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, USENIX Association, USA, p. 57–69.

Perez, A., Moltó, G., Caballer, M. and Calatrava, A. (2018). Serverless computing for container-based architectures, *Future Generation Computer Systems* **83**: 50 – 59.

Priedhorsky, R. and Randles, T. (2017). Charliecloud: Unprivileged containers for user-defined software stacks in hpc, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, Association for Computing Machinery, New York, NY, USA.

Pérez, A., Risco, S., Naranjo, D. M., Caballer, M. and Moltó, G. (2019). On-premises serverless computing for event-driven data processing applications, *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 414–421.

Shahrad, M., Balkind, J. and Wentzlaff, D. (2019). Architectural implications of function-as-a-service computing, *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, Association for Computing Machinery, New York, NY, USA, p. 1063–1075.

van Eyk, E., Grohmann, J., Eismann, S., Bauer, A., Versluis, L., Toader, L., Schmitt, N., Herbst, N., Abad, C. L. and Iosup, A. (2019). The spec-rg reference architecture for faas: From microservices and containers to serverless platforms, *IEEE Internet Computing* **23**(6): 7–18.

Xu, Z., Zhang, H., Geng, X., Wu, Q. and Ma, H. (2019). Adaptive function launching acceleration in serverless computing platforms, *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 9–16.