Final Demo Report

Group 9: *Akash Singh (axs210026), Aditi Chakravarthi (axc200021), Jiazheng Liu (jxl210006), Payton N Harmon (pnh170000), Thor Allan Rydahl (tar180000)*

Introduction

The link to our code: https://github.com/WalmartDeli/CloudComputingClass

This project aims to explore container technologies used in the cloud, and to study the edge cloud paradigm for IoT. It does this by imitating an edge cloud through creating, deploying, and managing user defined workflows using containerization technology. By combining containerization with our cloud design we were able to successfully execute custom workflows that integrate multiple containers across multiple machines to emulate a cloud edge system.

Approach

VMM Installation

- Installed KVM and QEMU as the hypervisor
- Installed VMM and associated packages, libvirt and virt-manager
- Downloaded CentOS7 as VM image
- Used virt-manager to create VMs with the downloaded image
- Installed 4 VMs with 7GB RAM and 100 GB disk space each on our physical machine (csa-6343-13.utdallas.edu)
- Create the virtual disk for the 4 VMs in the “qemu” format

Docker Engine Installation

To install Docker Engine, we used the official documentation from Docker’s website.
https://docs.docker.com/engine/install/centos/#install-using-the-repository

1. Install Docker Engine, container, and Docker Compose

```
$ sudo yum install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

2. Start Docker

```
$ sudo systemctl start docker
```

3. Run Docker Engine

```
$ sudo docker run hello-world
```

Docker Swarm Installation

1. Install docker on the 2 UVMs and 4 PVMs
2. Create a swarm on the UVM1 (it is the manager node)
command: sudo docker swarm init --token <token> 10.176.67.108:2377
3. Add nodes to swarm (run command on the worker node, e.g: UVM2)

command: sudo docker swarm join --token <token> 10.176.67.111:2377
4. View the information of nodes and status of the swarm
command: sudo docker info
command: sudo docker node ls

Comments on the installations

Following the steps in the documentation, the installations of docker engines and docker swarm are not hard after the internet connection problems are solved.

Comments about docker swarm

We installed the docker swarm, but we aren't using it in our project.

Overview of System Design

We will go into great detail below, but the system is designed using a three tier architecture. A client connects to the workflow manager and specifies the workflow they would like to run. The workflow manager talks to each machine in the cloud through the router on those machines where it will send deploy commands to each router based off of its load balancing algorithm when creating a service. The router will handle any communication between services between machines as specified by the routing table in the workflow manager.

Workflows

Workflow for NLP pipeline and summarization.

Containers and docker image:

- preprocessing-nlp: Data preprocessing (360.12 MB)
  https://hub.docker.com/r/aditichak/preprocessor-nlp
- CountVectorizer: transform into vectors for training and testing (415.28 MB)
  https://hub.docker.com/r/aditichak/training
- SVC: train a SVM model (415.28 MB)
  https://hub.docker.com/r/aditichak/svc
- NaiveBayes: train a NaiveBayes model (415.25 MB)
  https://hub.docker.com/r/aditichak/naivebayes
- LR: train a LogisticRegression model (415.25 MB)
  https://hub.docker.com/r/aditichak/lr
- model test: test the trained model (421.43 MB)
  https://hub.docker.com/r/aditichak/modeltest
- summary: summary and statistics on the training data (353.19 MB)
  https://hub.docker.com/r/aditichak/summary
- graph: create data structure for a word cloud to send to the data sink (360.12 MB)
  https://hub.docker.com/r/aditichak/graphs

Workflows: 4 workflows consist of above containers

- Workflow1: preprocessing-nlp -> CountVectorizer -> SVC -> model test -> data sink
- Workflow2: preprocessing-nlp -> summary -> graph -> data sink
- Workflow3: preprocessing-nlp -> CountVectorizer -> LR -> model test -> data sink
- Workflow4: preprocessing-nlp -> CountVectorizer -> NaiveBayes -> model test -> data sink

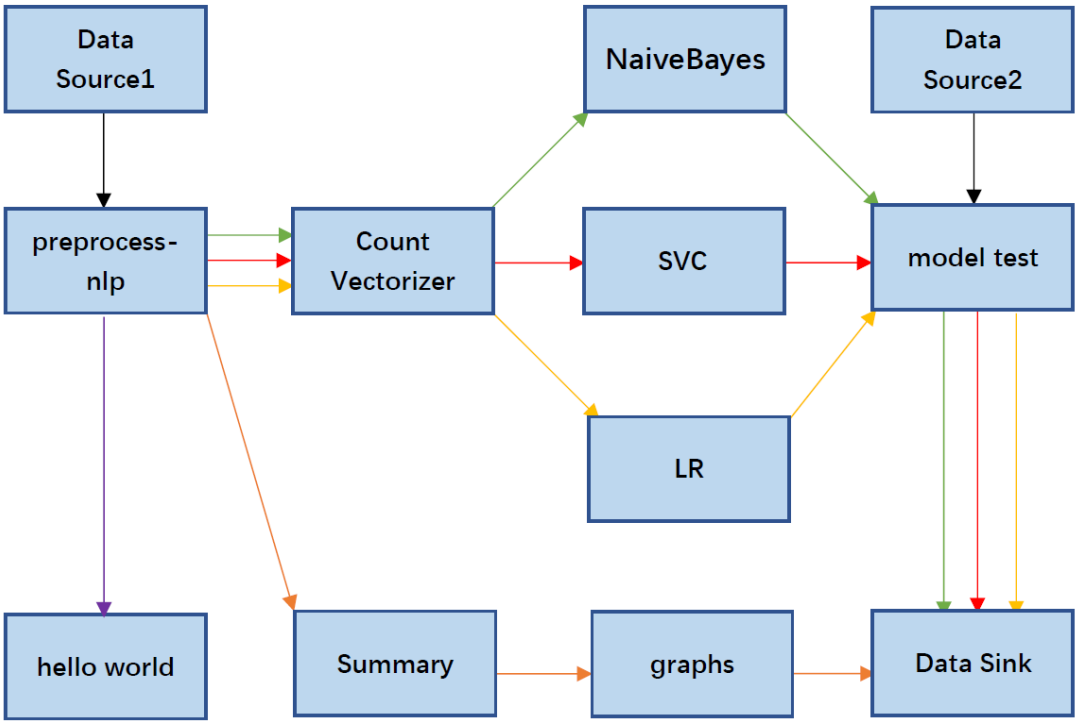


Figure 1

- Workflow5: preprocessing-nlp -> CountVectorizer -> {SVC, LR, NaiveBayes} -> { model test, model test, model test} -> data sink

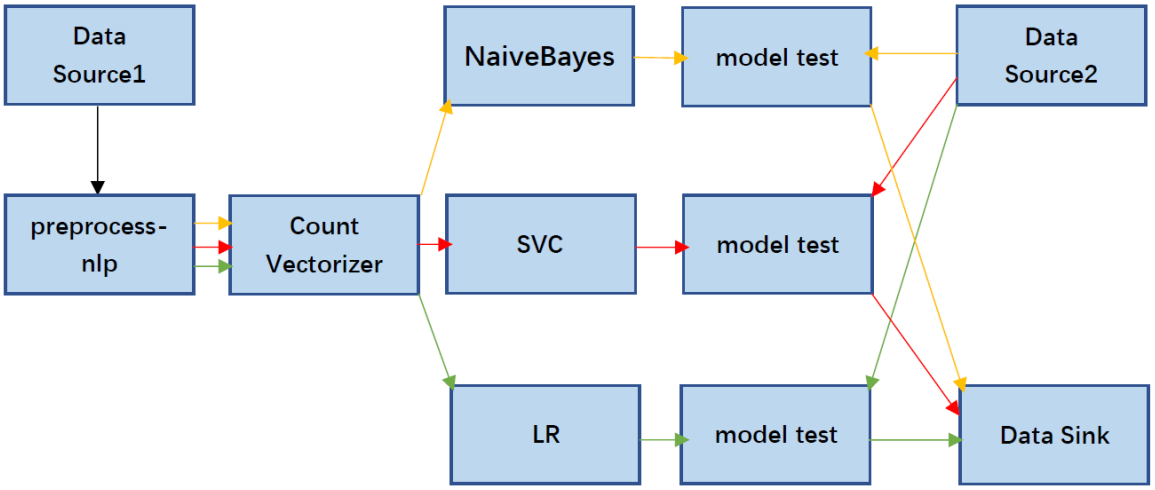


Figure 2

- Workflow6: preprocessing-nlp -> hello_world (for demo testing purposes)

Sample Workflow Results (Workflows 1, 3, 4):

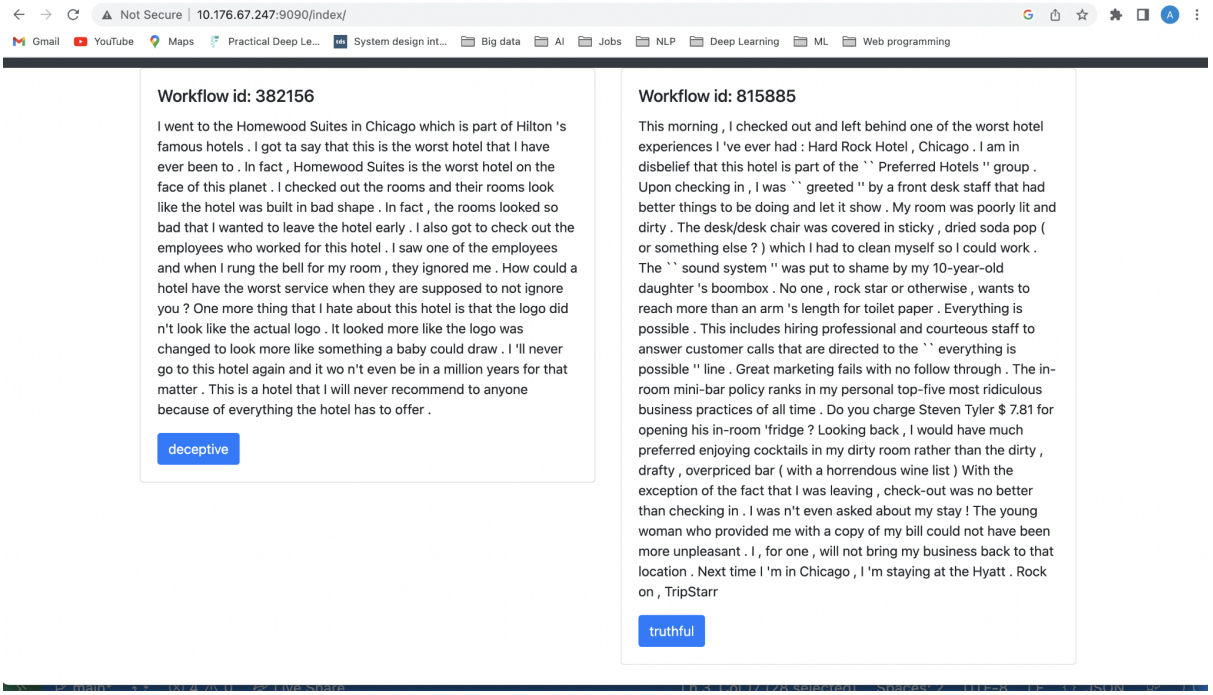


Figure 3

Word Cloud for Data Summary (Workflow 2):

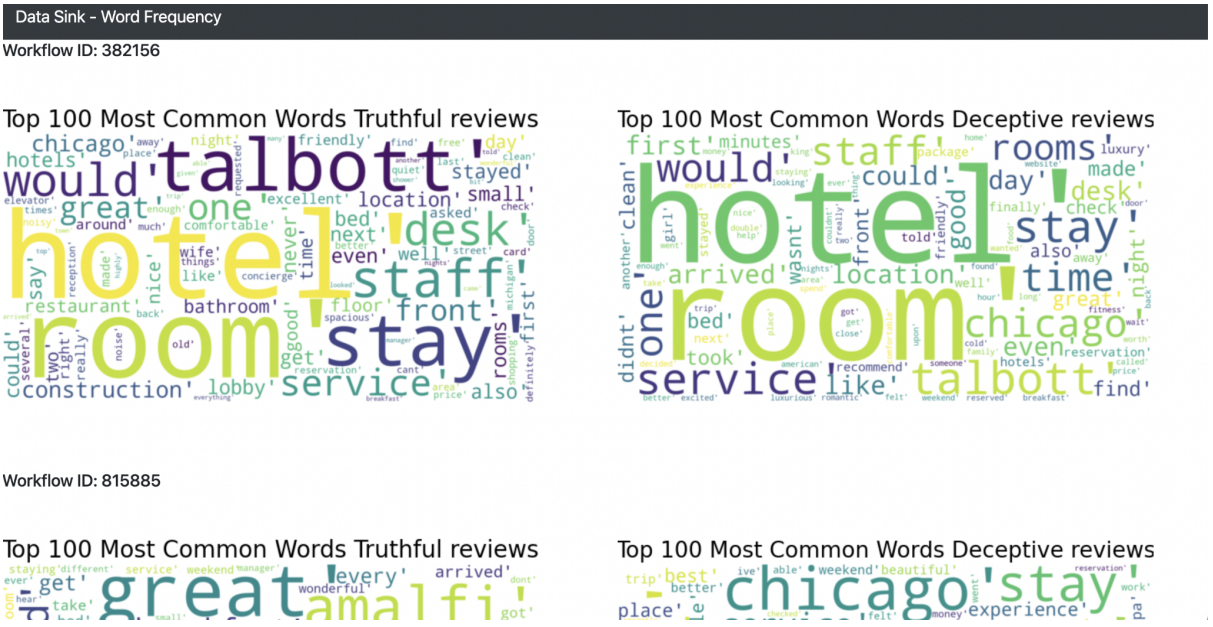
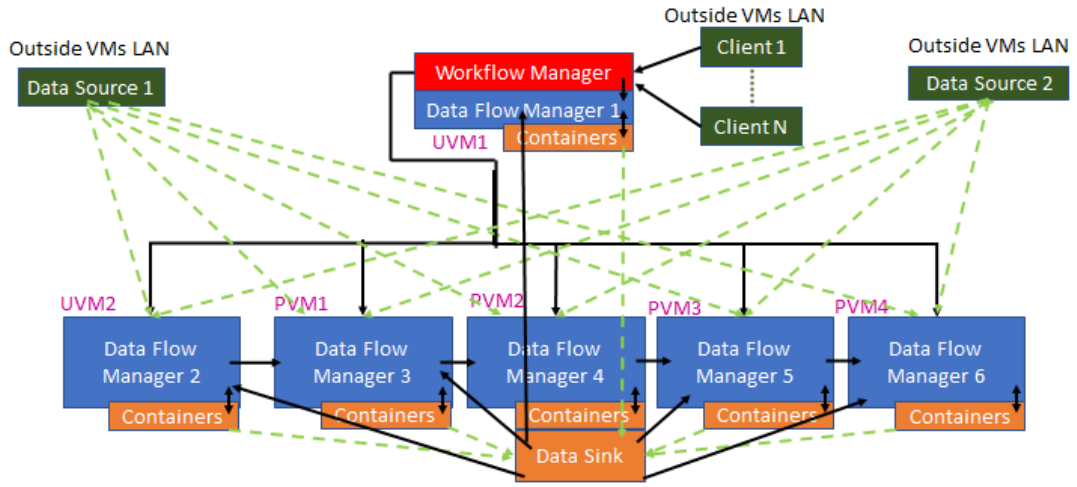


Figure 4

Design of Workflow Management Framework



- DataSource1 pass data (training data) and DataSource2 pass data (testing data) to all the data flow manager, and the data flow managers decides if the data is required by any of container deployed on same VM.
- The "modeltest" container (always the last container in our workflow) will post data to the data sink, which would be accessible by client
- All the communication between components is through Rest API

Figure 5

Data Source: Outside VMs LAN

- Pass data (training data or testing data) to all data flow managers, and the data flow manager checks if it has an active container which requires this data then passes the data to the container.

Data Sink: PVM2

- Flask app
- Receive the final result from the last container from all workflows (modeltest/graphs) in either labeled data or as a word cloud which shows word frequency in summarized data.
- All results are associated with a Workflow ID
- Dynamically displays new results from workflow
- Accessible by client
- Two different types of workflows can be accessed by the /index and /cloud pages

Client: Outside VMs LAN

- Gets to enter the json file name and the content in the json file should look like below JSON i.e. it should mention the components (image name on docker hub) we want run in our workflow, and mention the adjacency (how these component index should pass data to form a pipeline):

```
{
    "components": [
      {"image": "aditichak/preprocessor-nlp"},
```

```
      {"image": "aditichak/training"},
      {"image": "aditichak/svc"},
      {"image": "aditichak/modeltest"}
    ],
    "adjacency": [
      [1],
      [2],
      [3],
      []
    ]
}
```

- The client once sends a request the workflow manager will receive it and provide an acknowledgment with WorkflowID in it.
- Using this WorkflowID, the client can access the data sink to check for the results of its request.

Workflow Manager: UVM1

- Listens for client requests.
- Network config file stores available machine ID(format M##)/IP address pairs to deploy containers on.
- Also, keep track of ports assigned to each IP in the system.
- Receive the workflow as a parameter, and based on round robin share the containers information and port to be deployed by data flow managers in their VMs.
- Also creates a routing table based on which the data flow managers need to pass data in the pipeline.
- API Commands

Routers (Data Flow Manager): All the VMs

- These will listen to the WFM requests to deploy the required containers in each VM.
- Decide if the data from data sources is required by any container deployed on the same VM
- They pass the data received to them to the respective container based on the routing table and also to the next dataflow manager if the container has completed its job and needs to send data to the next container.

Containers (Services):

- Use RestAPI to communicate
- Once data is received, containers will process the data.
- Once processed, containers will send processed data, workflow id and port number to the local router, which will route data to the next-hop container..
- The last container of a workflow sends the result to data sink
- The data generators send training to the first container and test data to the last container.

Design of Each Service

- Use RestAPI to communicate
- For every service, it has 3 subcomponents: receiving data, processing data and sending data.
- Processing data: use functions we import from libs and define to process data, just like other python program
- Receiving data: We use method in python package 'Flask' to keep listening and receive data

```
from flask import Flask, request, json
app = Flask(__name__)
@app.route("/datasink",methods=["POST"])
def func_name():
        data=json.loads(request.json["DATA"])

if __name__ == "__main__":
        app.run(host='0.0.0.0', port=8080)
```

- Sending data: We use a method in python package 'requests' to send data (The data is always sent to the local router. The address of local router is given by sys.argv)

```
import requests
address = sys.argv[2]
requests.post(address, json=dictionary)
```

- Choose model: Different training models are in different services, one can join different services into his workflow to train and test different models.

Design of Data Generators

Output

- Data generator1: training data
- Data generator2: testing data
- Data for hotel reviews is present at: https://myleott.com/op-spam.html

Data generation

- Data generator1: Choose randomly from 20 subsets of the training data set of 1600 rows, each subset has 80 rows.
- Data generator2: Choose randomly from 116 subsets of the testing data set of 116 comment text, each subset has 1 comment text.
- For each data generator, the user manually chooses when it sends data. (By press the enter key)

Problems encountered and how they are resolved

Setting Up VMs:

We tried to install VMs with Virtual-Manager but we weren't able to setup a VM with working inter-connected network. We also tried using Cockpit Browser, VMWare but the results were same. Later-on we figured out that the problem was with the network settings. We were trying to automatically assign IPs to VMs using DNS, but later-on we figured out that it was the problem. And this was resolved by manually assigning the IPs to the VMs.

The design of the dataflow

At the beginning, we decided to set up an overlay network for all the containers. Every container in the network will have a unique IP address and port so that they can communicate with each other. And the workflow information is available for every container in it. After processing the data, the service sends the data to the next container according to the workflow information. However, it is not a good design for a cloud computing project. Dealing with workflow information makes every service more complex. And the dataflow is messy. So we designed a dataflow manager for our project. There is a router on each node. After processing data, a container sends the data to the router which is on the same node with the container. According to the information about workflow, the router will pass the data to another router which will send the data to the container waiting for the data.

Docker Hub pull rate limits

We ran into issues with Docker Hub rate limits with the error message "Error response from daemon: toomanyrequests: You have reached your pull rate limit. You may increase the limit by authenticating and upgrading: https://www.docker.com/increase-rate-limit". As per the Docker website, the free version of Docker Hub has 100 and 200 container image pull requests per six hours. We definitely exceeded that when repeatedly modifying and pulling all eight images in a single session. Unfortunately this problem was only resolved by waiting the specified six hours before trying again.

DNS Service Down

We were using the hostname such as csa-6343-103.utdallas.edu, csa-6343-93.utdallas.edu in our code, but the network was failing when we were connecting to UVMs. The reason was DNS Service was down, which we figured out after some time and moved to IPs for sending requests.

Rebuild Network Settings

Lost network settings multiple times because the machine was restarted by someone, which made us set the bridge network every time which also disallowed working remotely.

Experiments and Results

1. Image Transmission Time

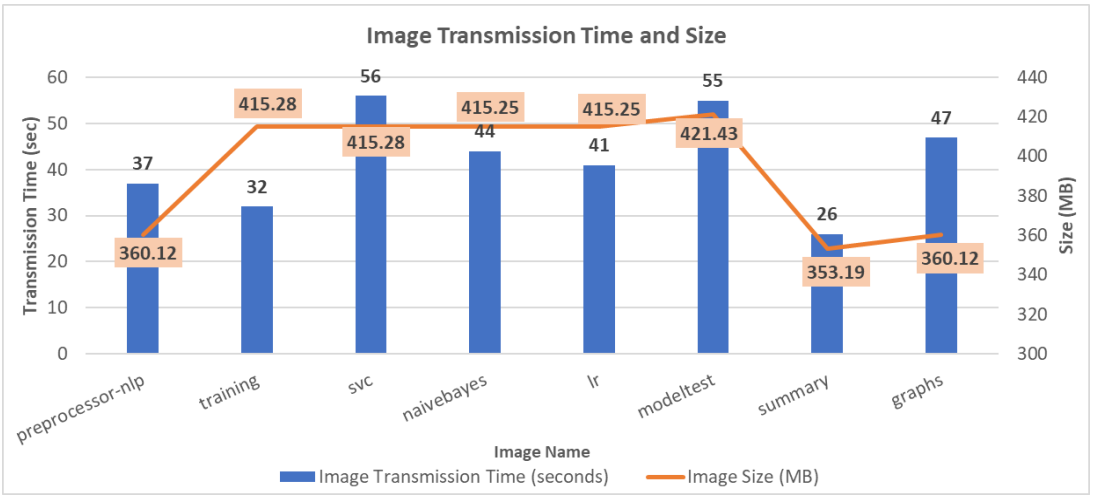


Figure 6

2. Image Start-up Time

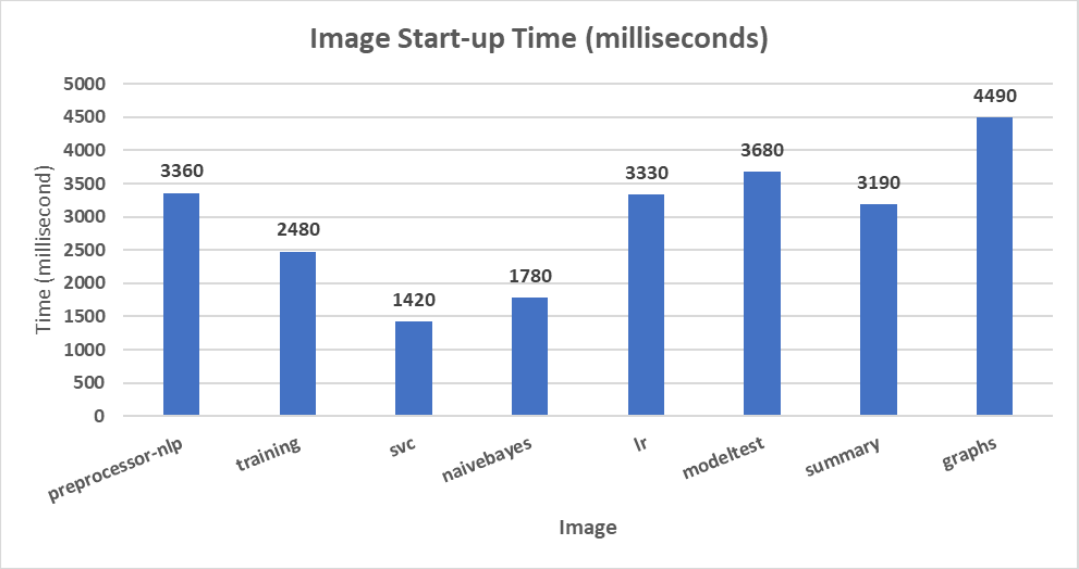


Figure 7

3. Execution Time

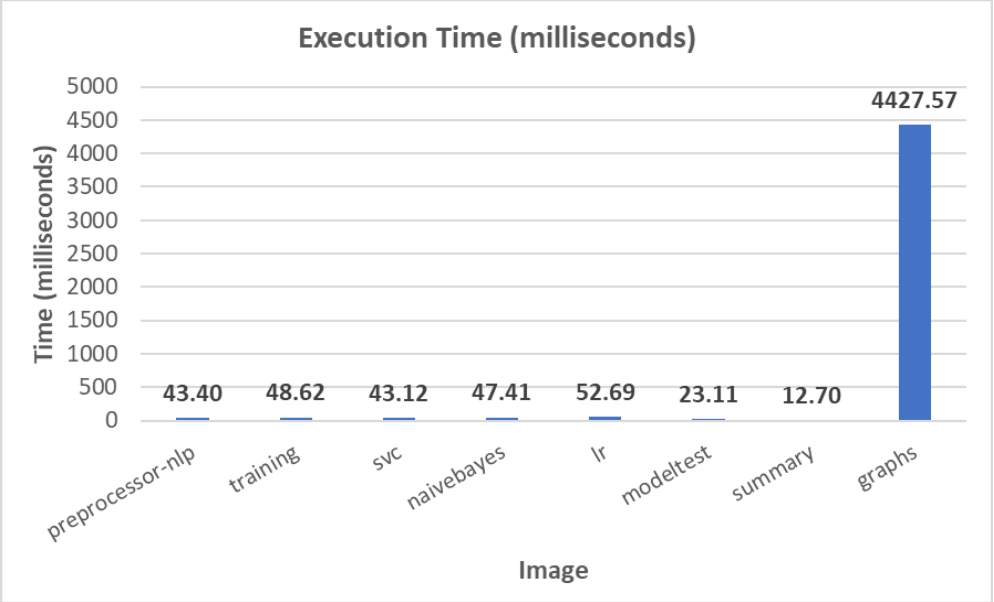


Figure 8

4. Communication Time

Source container	Target container	Communication Cost (milliseconds)
preprocessor-nlp	training	9.28
training	svc, naivebayes, lr	23.72
svc, naivebayes, lr	modeltest	42.00
summary	graphs	10.41

Figure 9

Achievement

- Our system is successfully synchronizing workflow components to generate output.
- Our system can be run with all the designed workflows and using all the VMs.
- Our system is capable of handling multiple concurrent workflow requests.
- Our client from external LAN is able to access the data sink to view results.

Exploration

Container placement and persistency:

The current design for persistence is to make all new containers persistent and reuse them for later workflows. This works by checking to see if the image requested by the workflow is already deployed on one of the servers, and if it is it will route the workflow to traffic its data through the existing container. With more time and testing, a smart algorithm can be used to keep track of common

images used across multiple workflows and ensure they get placed on servers according to load. One of the challenges with the current way the system is built is that containers don't gracefully quit once their workflow is complete. Therefore, whenever a new workflow spins up using the same containers as another workflow, completely new containers are created while the old ones persist unused. The current persistence plan gets around this issue by reusing the older containers while a solution can be written to gracefully terminate containers.

Container placement algorithm:

As mentioned earlier in this report, the base implementation of our project uses a round-robin deployment schema to start new containers. We extended our project to work with two additional heuristic algorithms for container placement: best-fit and worst-fit. These algorithms are based on the CPU utilization of the machine. Available memory and running average CPU utilization are reported to the WFM by the routers on request at the time of deploying a new workflow, at which point the WFM decides which machine to deploy each container to using the selected heuristic. Worst-fit spreads the load more evenly due to prioritizing deploying new services to VMs with the smallest load.

- Worst-fit was the best for performance due to the fact that our container persistence design requires containers to grow, so it is better to have free space available than it is to use 100% of available CPU time.
- Best-fit tries to fill the most loaded VM first, which means that VM's router is overworked and the other VMs are underutilized.

Deploy Time: Deploy workflow 5 and measure the time cost by deployment

Workflow 5	Worst Fit	Round Robin	Best Fit
Without Persistency	10s	10s	11s
With Partial Persistency	10s	8s	7s
With Full Persistency	5s	<1s	5s

- Without persistency: no workflow was persistent.
- With Persistency (Partial): some of the workflows were persistent.
- With Persistency (Full): every workflow was persistent.

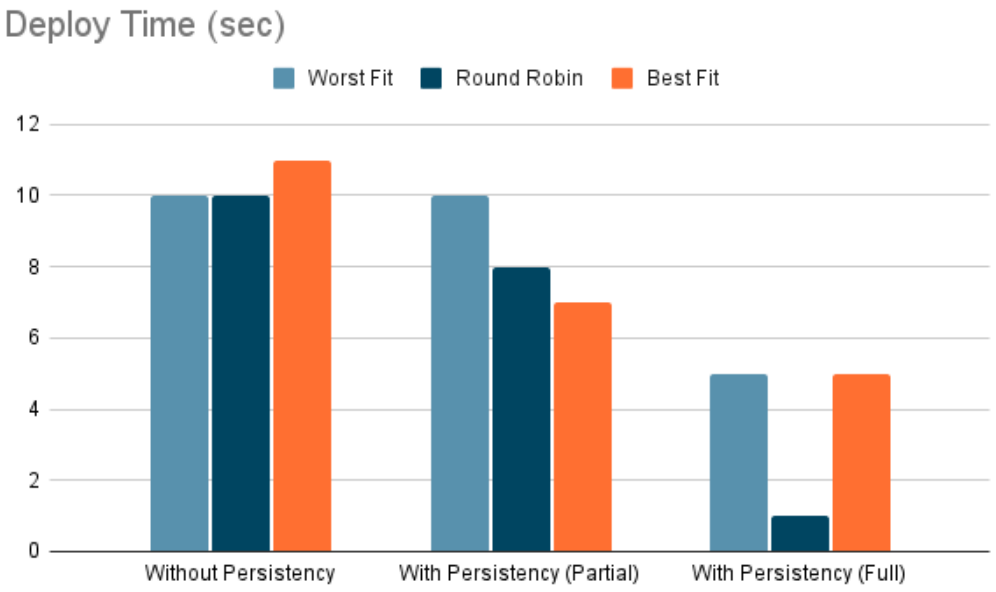


Figure 10

Process Time:

75 requests made	Worst Fit	Round Robin	Best Fit
With Persistency	<1s	<1s	13s
Without Persistency	<1s	1s	Failed

- '75 requests made' means that deploy 25 copies of workflow 5, measure the time between data generator2 send testing data and the datasink receives all the 75 results from the workflows.
- Failed: System took a long time to deploy all 25 copies of workflow5 and would not process data once deployment was complete.

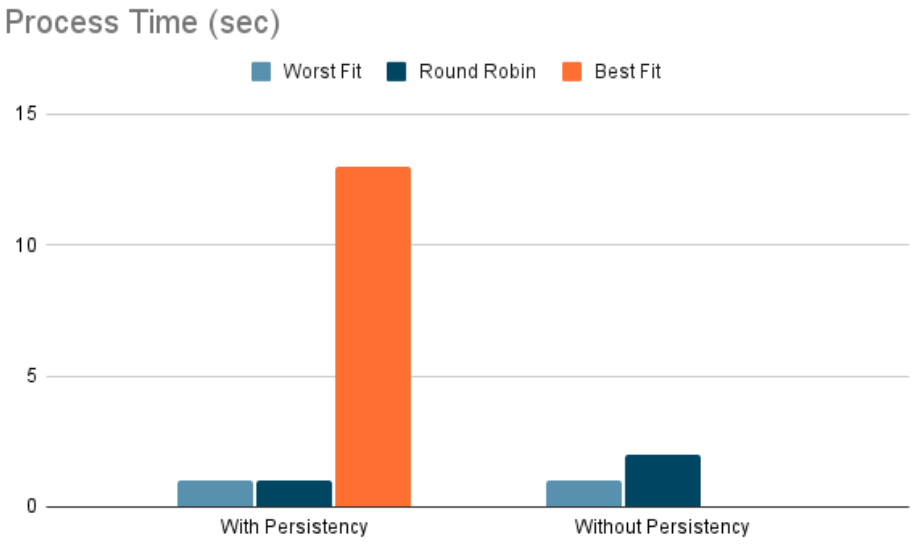


Figure 11

Workflow Processing Time with different algorithms

The graphs below illustrate the time it took to complete all five types of workflows with different algorithms and with different levels of persistency.

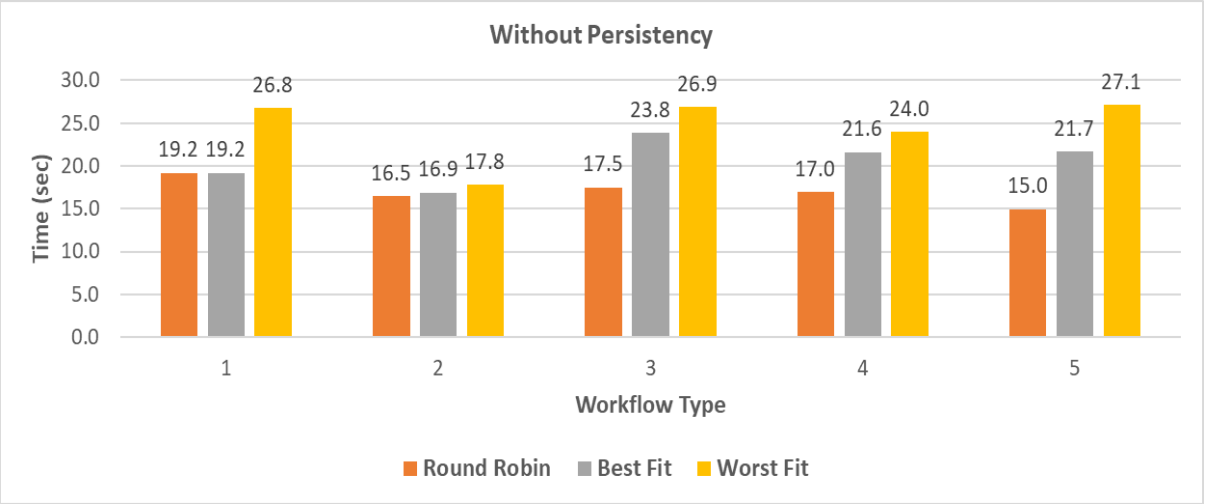


Figure 12

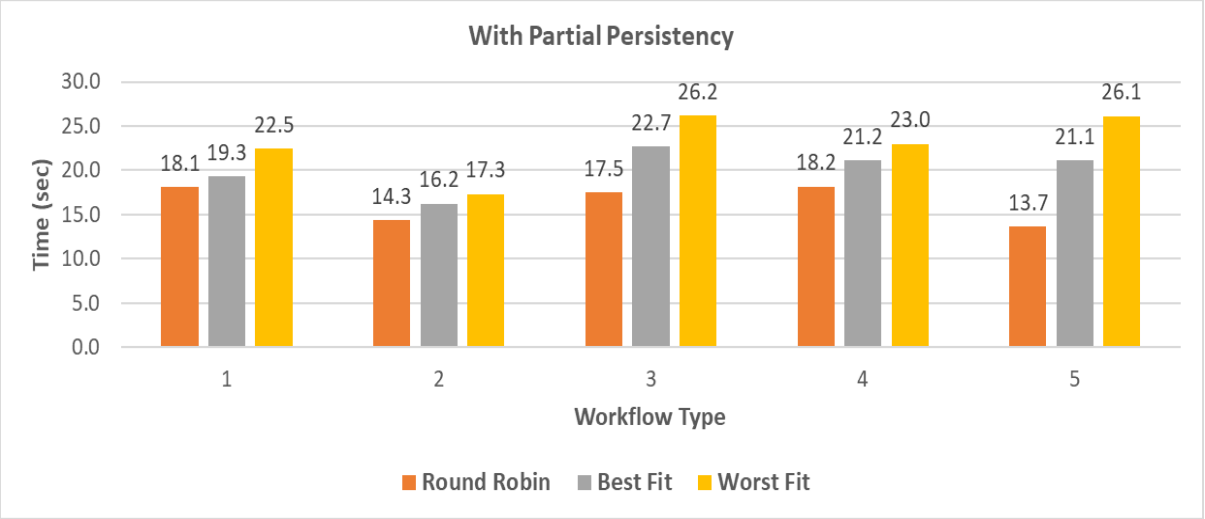


Figure 13

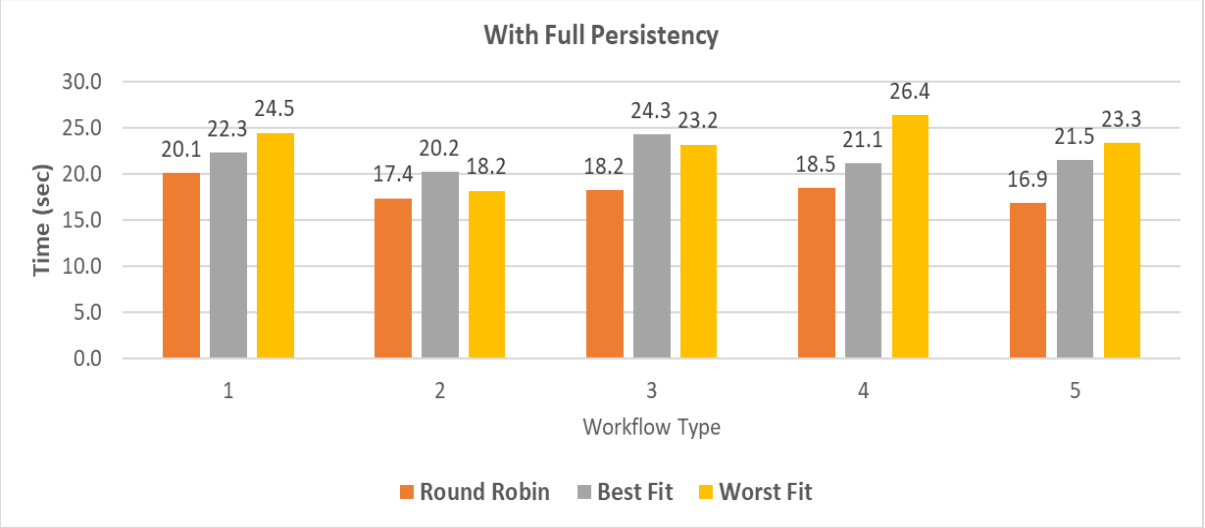


Figure 14

From the graphs we can see that Round Robin, even compared to Best Fit, performs quite well among all workflow types and the different algorithms and level of persistency.

Operation Manual

Start the workflow management framework

1. Start router.py in all the six virtual machines ("10.176.67.108", "10.176.67.111", "10.176.67.248", "10.176.67.247", "10.176.67.246", "10.176.67.245") with command python3 workflow_router.py.
2. Start the workflow_manager.py on UVM1 (“10.176.67.108”) with command python3 workflow_manager.py.
3. Start the datasink on PVM2 (“10.176.67.247”) using command “sudo docker run -p 9090:9090 aditichak/datasink”.

4. Start the data generators on local machine (external LAN) by running data_generator1.py and data_generator2.py
5. Issue a client request by running the command python3 cloud_client.py on a local machine (external LAN), and enter the name of the json file (i.e. the workflow user wants to run). The json file should look like below JSON. (detailed introduction of the json file is in the Design of Workflow Manager Framework part)

```
{
    "components": [
      {"image": "aditichak/preprocessor-nlp"},
      {"image": "aditichak/training"},
      {"image": "aditichak/svc"},
      {"image": "aditichak/modeltest"}
    ],
    "adjacency": [
      [1],
      [2],
      [3],
      []
    ]
}
```

6. To access the result for the issued workflow, access the below link on web browser:
    a. http://10.176.67.247:9090/index (NLP)
    b. http://10.176.67.247:9090/cloud (Data summarization displayed as a word cloud graphic).
7. The result shown on the above link will be like the content in Figure 3 and Figure 4.

Workload Distribution

Summary:

- VMs installation, network connection, docker swarm installation
- Design and code for Workflow Manager and Dataflow Manager
- Design different workflows
- Coding for each service in the workflows and create docker images
- Create data generators and data sink for the workflows
- Algorithm for container persistence and container deployment
- Test and comparison for different workflows and algorithm

Detailed List

Akash: Installed VMM, VMs, and necessary dependencies on VMs and Physical Machines. Connecting VMs via bridge network, rest communication between client and workflow manager. Examined system capabilities to design the initial workflows and system design. Built data generators and integrated in with the existing workflow router and workflow manager code. Helped team in deploying different system components and workflows. Experimented with different workflows.

Aditi: Created and deployed all images in the NLP and summarization pipeline to Docker Hub. Created the data sink as a web app, added containerization and overall styling. Created workflows as per specifications. Worked on integrating containers with data flow router and manager. Worked on end-to-end system integration, testing, and debugging. Collected experimental data for images.

Jiazheng: Deploy containers and set up an overlay network among all the containers so that they can communicate with each other directly. Work on a service for data sink to receive the result. Work on a data source for sending testing data. Work on developing more workflows for further experiment. Use docker swarm to distribute and balance the workload of the designed workflow and compare the performance with our project. Write the final report.

Payton: Wrote part of the client. Designed the workflow specification language. Wrote parts of the workflow deployment function including the round robin assignment algorithm. Wrote persistence into the router and manager scripts. Tested processing time with persistence. Tested and debugged overall system.

Thor: Designed and wrote workflow manager and workflow router framework. Wrote worst-fit and best-fit deployment options. Performed end-to-end debugging and testing for workflow deployment and persistence, as well as dataflow.