GROUP 4: CS 6343 Cloud Computing – Term Project

Yen, Ming-Hsuan; Tiwari, Ayu; Gadekar, Rutuja Vijaykumar; Baddam, Maharshi Reddy

Supervisor: Prof. I-Ling Yen

Abstract:

We live in the age of smart connected devices. The proliferation of devices is generating a massive amount of data every day. The end user devices generate data and transfer it to the public cloud or private cloud environment or on-premises data centers. But many real life IoT (Internet of Things) application needs faster results with a requirement of highly available network with minimal latency time to process large amounts of data in real time, which is not possible on a traditional IT infrastructure where data storage, data processing, and other data analytical operations leads to latency and bandwidth issues. To overcome these limitations, Edge Computing paradigm is a viable choice which provides the distributed computing capabilities with intelligence on edge nodes to reduce latency time and increase data availability. In this project we will explore the lightweight virtualization technologies on the edge node cluster, such as containers. We will demonstrate the Docker implementation and deploy various persistence and reusable services to support IoT devices data processing on the edge cluster such as data Ingestion and data processing with machine learning applications. Later we will define performance statistics to suggest which docker service gives the best results in optimal time.

Introduction: Cloud computing provides on-demand computing resources, network and infrastructure resources, Services and Application platforms over the Internet. Edge computing is an enhancement on the top of existing cloud infrastructure to facilitate local edge node-based distributed computing environment, hence data can be stored and processed, and analyzed locally to deliver faster results. The combination of these two technologies provides a better performance on IoT applications.

Few key advantages of Edge Computing- Speed and Latency, Security, Cost Savings, Greater Reliability, Scalability.

Architecture: High level Edge Node cluster workflow architecture Figure 1.

- IoT client devices will access edge clusters by authenticating themselves over the Internet gateway.

- Edge Cluster will contain multiple edge nodes: One Manager Node (Master Node) and two worker Nodes (Slave Nodes) to facilitate the distributed computing capability.

- Edge cluster environment will run multiple lightweight virtualization processes called "Docker Container Services/images". These containers will be managed by Docker Daemon process.

- Each node in the cluster will run a “Docker Swarm” to activate the orchestration and deployment of container-based services/images for application processing and data pipelines.

- Manager Daemon process will expose the end point to the external IoT client devices and handle requests to launch the container(s).

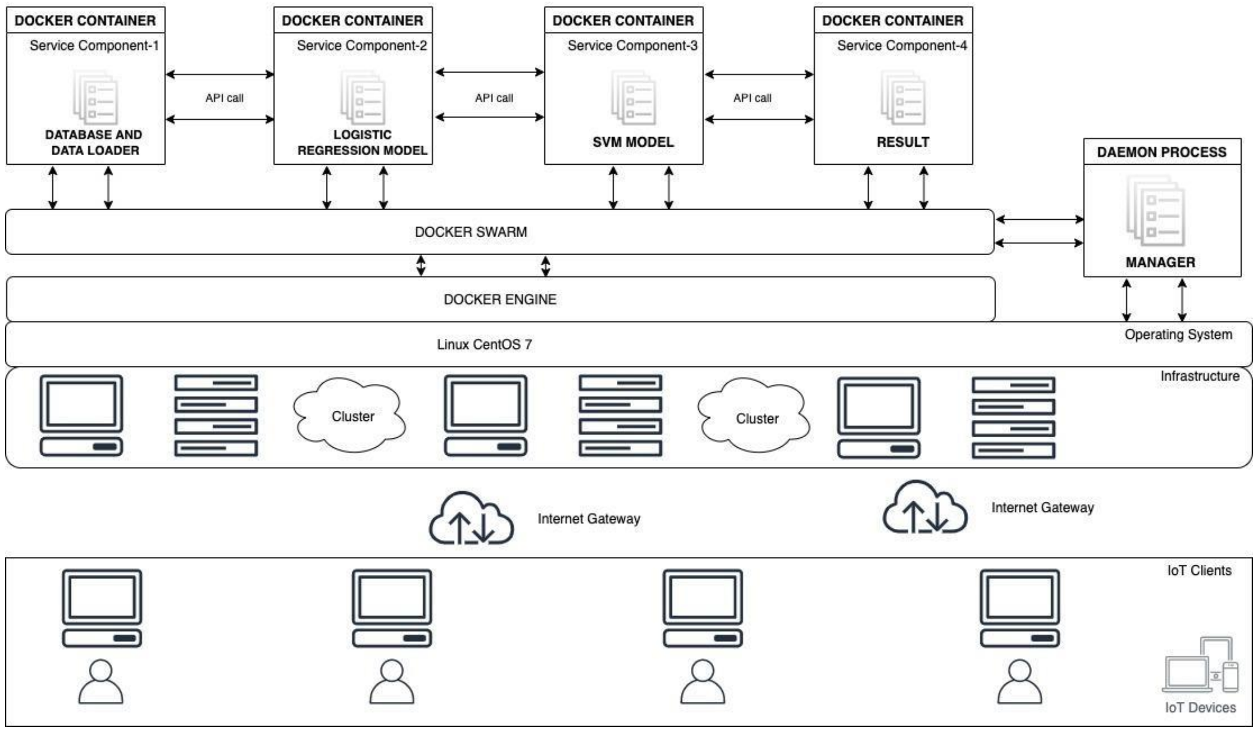


Figure 1: Edge Node Cluster based Simulation using Docker Virtualization

Docker:

Docker Engine and Swarm Installation:

We followed the instructions on the Docker Homepage for swarm installation. The websites are mentioned in the References section. We have documented the steps executed for swarm installation on Github. Please find the documentation at https://github.com/Conradyen/docker-swarm-setup

Docker Swarm:

Docker Swarm is more lightweight when compared to other alternatives like Kubernetes. It can deploy a container much faster and thus enable fast reaction time.Swarm consists of a DNS element that helps in distributing incoming requests to a service name. The user can specify the ports for a service which help in balancing the load or the service can be assigned automatically to any node. The load balancing is internal Swarm and also we can expose ports for external load balancing services. Swarm offers high availability since the services can be replicated among the nodes.

USE CASES:

In this project we have implemented two use cases problems to support Edge Node and IoT based solution:

USE CASE I: Employee

Global Data centers Employees attendance tracking system and human resource capacity planning based on prediction:

Component 1 - Database and Data Loading Service: All employees need to authenticate themselves before entering into the office premises. Companies can implement either an employee card scan/swipe method to authenticate employee identity or a biometrics-based authentication system. In either case an IoT device authentication system will generate the data log. Data will be generated periodically on the client machines and IoT devices will continuously transfer data to the Edge Computing cluster for storing and processing purposes. Each company's employee data will contain the following mandatory features:

Employee Feature set (E) ={employee_id, department_type, gender, race, day_of_week, checkin_datetime, duration}

Data will be stored in Cassandra Database, which is designed to support the Multi-Tenancy model to give complete data isolation to each company. Below diagram shows the logical representation of client data isolation in Cassandra DB:

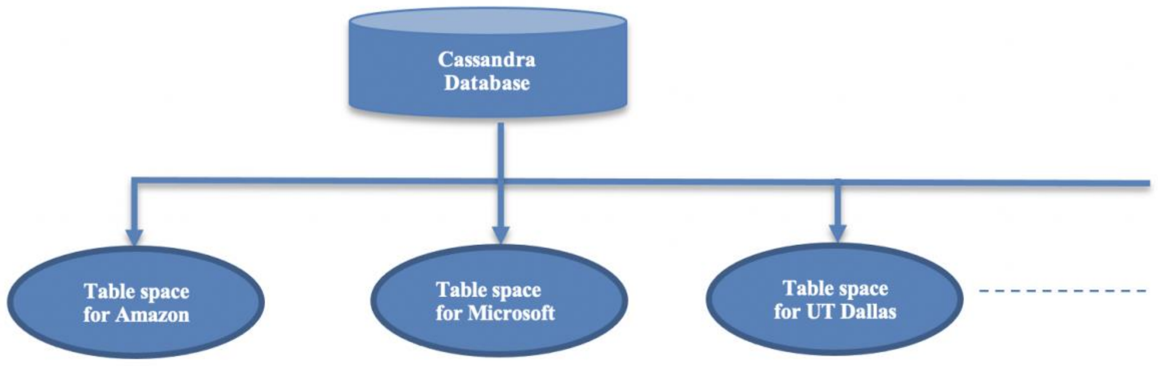


Figure 2: Database and Data isolation for each client

Component 2 – Logistic Regression Model Service: Logistic Regression Model will train the model for each company/client during the workflow launch time (one-time training for each workflow type), after that periodically accepts the data from the IoT devices to do the prediction for employees exit time from the Data Centre to determine the duration that each employee at onsite. Below diagram shows the logical representation of LR Model service:

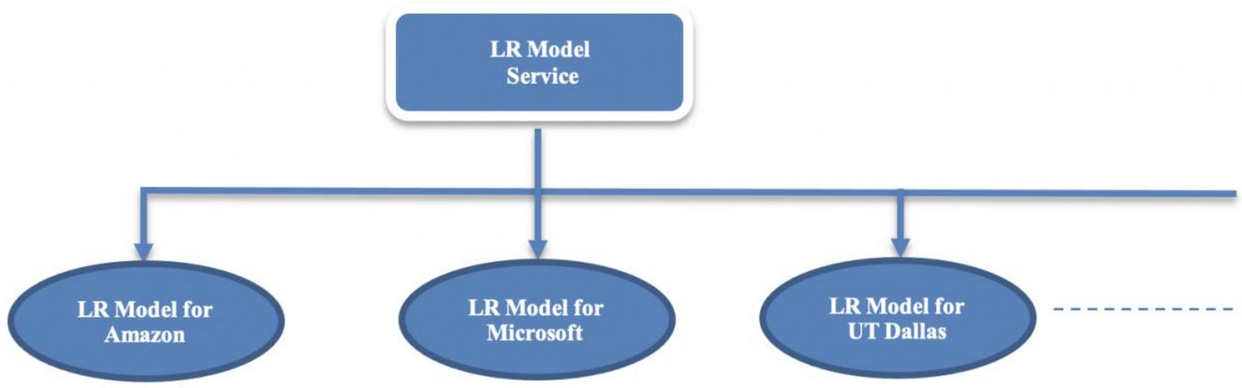


Figure 3: LR Model container will maintain the Training Model and Prediction logic for each client.

<u>Component 3 – SVM Model Service:</u> Support Vector Machine (SVM) Model will train the model for each company/client during the workflow launch time (one-time training for each workflow type), after that periodically accepts the data from the IoT devices to do the prediction of total number of employees should be available onsite at the given time for the capacity planning. Below diagram shows the logical representation of SVM Model service:

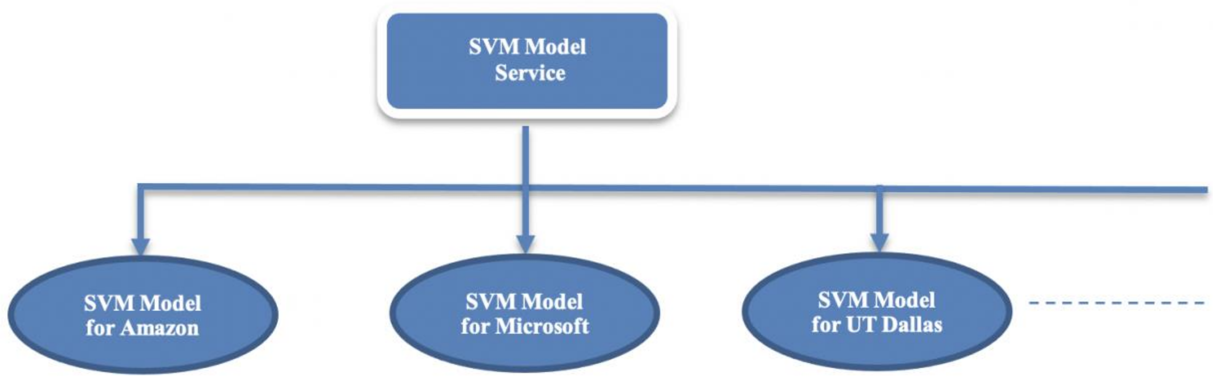


Figure 4: SVM Model container will maintain the Training Model and Prediction logic for each client.

<u>Component 4 – Result:</u> Each company/client will have one dedicated result interface container, which will facilitate the results. Result data transfer from Edge cluster container to the configured client interface.

USE CASE II: Hospital

Goal is to predict the hospital stay duration of a patient and predict the number of patients at a given point of time which can be further used to identify the number of beds required at the hospital

<u>Component 1 - Database and Data Loading Service</u>: When any Patient admission is detected, hospital staff will do the registration and it is assumed that the identity of patients will be replaced by some dummy IDs to protect the patient privacy and maintain the compliance guidelines. IoT devices will generate data periodically and data will continuously be transferred to the Edge Computing cluster for storing and processing purposes. Hospital/Medical institution data should contain the following mandatory features:

Hospital Feature set (E) ={hospital_admission_id, hospital_expire_flag, insurance, duration, number_in_icu, amount, rate, total_items, value, dilution_value, abnormal_count, item_distinct_abnormal, checkin_datetime, day_of_week}

Data will be stored in Cassandra Database, which is designed to support the Multi-Tenancy model to give data isolation to each company, this process similar to the employee workflow as described in figure 2.

Component 2 – Logistic Regression Model Service: Logistic Regression Model will train the model for each hospital organization/client during the workflow launch time (one-time training for each workflow type), after that periodically accepts the data from the IoT devices to do the prediction for the number of days the patient will stay in the hospital.
LR Model service will maintain the train Model and prediction logic for each hospital client, process is similar to the employee workflow as described in figure 3.

Component 3 – SVM Model Service: Support Vector Machine (SVM) Model will train the model for each company/client during the workflow launch time (one-time training for each workflow type), after that periodically accepts the data from the IoT devices to do the prediction for the number of patient in the hospital at the given time for the capacity planning.
SVM Model service will maintain the train Model and prediction logic for each hospital client, this process is similar to the employee workflow as described in figure 4.

Implementation:

Service components and Precedence Order:

To prevent human errors (e.g., typo or wrong spellings) we have assigned the unique "**ServiceID**" number to each service component. Instead of typing the full-service name, clients can select the one digit ServiceID number to define their workflow.
We have also proposed the order of precedence for each service to avoid potential errors/failures because of the wrong workflow order. Following are the definitions:

HIGH- The high precedence service should be the first service component in the workflow order.

MEDIUM- The medium precedence service should be second or third in workflow request. For example, Logistic regression and SVM can be selected in any order.

Table 1: Service component Codes and precedence Order

ServiceID	Service Name	Precedence
1	Database + Data Loader	HIGH
2	Logistic Regression Model	MEDIUM
3	SVM Model	MEDIUM

NOTE: If a user will select the MEDIUM precedence service components as the first component in their workflow request then that service will fail, and workflow request will be rejected. For example, Machine learning Models cannot do the prediction if there is no train Model is available for that client, so Data Loader service gets the higher precedence in service precedence order.

Design of Database and Data Generator:

Cassandra : Cassandra keyspace will maintain a separate table for each client/company, shared key space is used for both use cases (employee and hospital), but tables will be unique to each company/workflow request. Cassandra is using following key space parameters:-

Class Strategy = Simple_Strategy
Data Center = 1
Replication Factor = 2 (Two cassandra containers will be launched to simulate two node replication)

Docker deployment: “docker stack deploy” command is used to deploy the Cassandra data centre with 2 node replication on the docker swarm.

Employee Tables:

Keyspace name : **ccproj_db**
Table name : **employee_<company_name>** **(**here company name will be given by the client in request)

Table Schema:

S.No.	Column Name	Description	Value Type	Constraints
1.	uu_id	unique UUID code	UUID	Primary Key
2.	emp_id	Employee ID	Numeric	
3.	dept_type	Department Type	String	
4.	gender	Gendre [Male or Female]	String	
5.	race	Race code	String	
6.	day_of_week	Day of the week	String	

7.	checkin_datetime	check in Date and time	String	
8.	duration	shift duration hours	String	

Data Generator Logic for Employee:

Employee data requests consist of the following parameters and each data value generation logic is given below.

emp_id : Unique employee ID for each company
dept_type : Randomly selected from the list of department [1, 2, 3, 4, 5, 6]
gender : Randomly selected from the list of department ["male", "female"]
race: Randomly selected from the list of race codes [1, 2, 3, 4, 5, 6]
day_of_week : Randomly selected days from list ["MON", "TUE", "WED", "THU", "FRI"]
checkin_datetime: Randomly selected date and time
duration: Randomly select work hours duration from 1 to 8 hours.

Hospital Tables:

Keyspace name : **ccproj_db**
Table name : **hospital_<company_name>   (**here company name will be given by the client in request)

Table Schema:

S.No.	**Column Name**	**Description**	**Value Type**	**Constraints**
1.	uu_id	unique UUID code	UUID	Primary Key
2.	hadm_id	Patient Admission ID	Numeric	
3.	hospital_expire_flag	Identify if the patient expired in hospital	Numeric	
4.	insurance	Type of Insurance	Numeric	
5.	duration	Shift duration hours	String	
6.	num_in_icu	Number of days in hospital in ICU	Numeric	
7.	amount	Amount of drug administered	Numeric	
8.	rate	Rate at which drug is	Numeric	

		administered		
9.	total_items	Total unique item drug substances administered	Numeric	
10.	value	Measure of each item/drug substance	Numeric	
11.	dilution_value	Measure of antibiotic sensitivity on a patient	Numeric	
12.	abnormal_count	Abnormal count of item levels	Numeric	
13.	item_distinct_abnormal	Total number of distinct items measured as abnormal	Numeric	
14.	checkin_datetime	check in Date and time	String	
15.	day_of_week	Day of the week	String	

Data Generator Logic for Hospital:

Hospital data requests consist of the following parameters and each data value generation logic is given below.

hadm_id: Unique hospital patient admission ID for each company
hospital_expire_flag : Randomly assign code [1, 2]
insurance: Randomly selected from [1,2,3,4,5]
duration: Randomly select work hours duration from 1 to 24.
num_in_icu: Random number between 1 and 40
amount, rate,,total_items, value, dilution_value, abnormal_count, item_distinct_abnormal : all these columns are random floating point number generated uniformly between 0 and 1
checkin_datetime: Randomly selected date and time
day_of_week : Randomly selected days from list ["MON", "TUE", "WED", "THU", "FRI", "SAT", "SUN",]

Workflow and Dataflow specifications:

The proposed architecture (figure 1) is flexible and supports custom dataflow requests from multiple clients. This system also has a feature to define any workflow with two possible options:

Reuse Services (Shared Model)– This is a cost saving option and avail clients to use the shared service models. In the shared service model all service containers are shared with other clients with full data isolation. Clients will be sharing services components, but each client will get a separate logical data flow and one company will not see or use the data of another company.

No-reuse Services (No Share Model)-This is a costly option and requires a higher resource demand. In the No-share service model every workflow request will get its own dedicated service containers with full virtualization and full data isolation. No service containers will be shared.

Figure 5 and 6 explain the concept of Shared Model design vs No-Share Model design:

Client-1 has data flow request [Data Loader (ServiceID#1) -> LR Regression (ServiceID#2) -> SVM Model (ServiceID#3)]
Client-2 has data flow request [Data Loader (ServiceID#1) -> LR Regression (ServiceID#2) -> SVM Model (ServiceID#3)]
Client-3 has data flow request [Data Loader (ServiceID#1) -> SVM Model (ServiceID#3)]

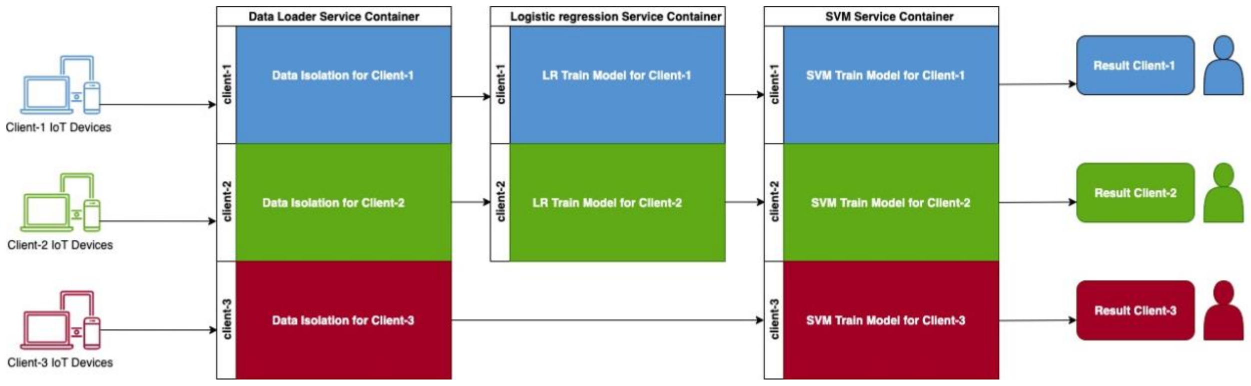

Data Loader Service Container
Logistic regression Service Container
SVM Service Container
Client-1 IoT Devices
Client-2 IoT Devices
Client-3 IoT Devices
client-1
client-2
client-3
Data Isolation for Client-1
Data Isolation for Client-2
Data Isolation for Client-3
LR Train Model for Client-1
LR Train Model for Client-2
SVM Train Model for Client-1
SVM Train Model for Client-2
SVM Train Model for Client-3
Result Client-1
Result Client-2
Result Client-3

Figure 5: Reuse Services (Shared Model) – Each client isolation is highlighted in different colors in shared service containers

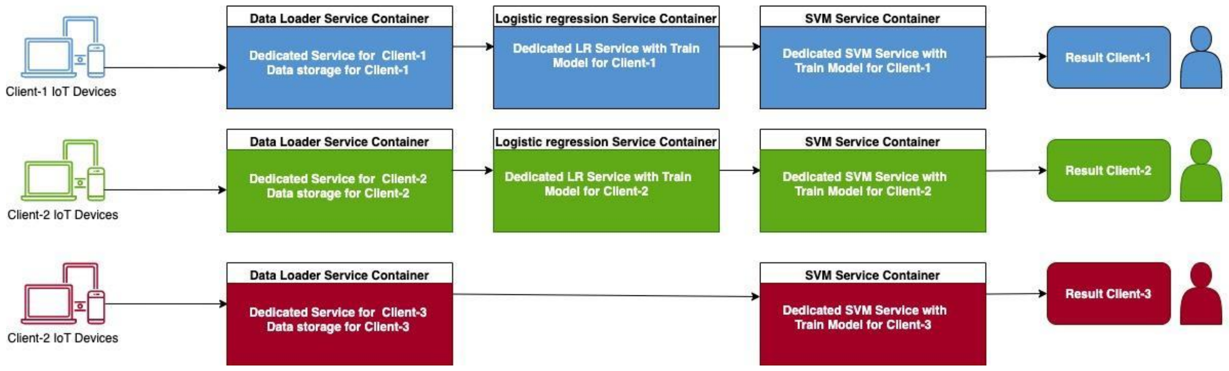

Client-1 IoT Devices
Client-2 IoT Devices
Client-2 IoT Devices
Data Loader Service Container
Dedicated Service for Client-1 Data storage for Client-1
Dedicated Service for Client-2 Data storage for Client-2
Dedicated Service for Client-3 Data storage for Client-3
Logistic regression Service Container
Dedicated LR Service with Train Model for Client-1
Dedicated LR Service with Train Model for Client-2
SVM Service Container
Dedicated SVM Service with Train Model for Client-1
Dedicated SVM Service with Train Model for Client-2
Dedicated SVM Service with Train Model for Client-3
Result Client-1
Result Client-2
Result Client-3

Figure 6: No-reuse Service (No Share Model) - Each client will get the full virtualization highlighted in different colors. Separate containers will be launched for each service.

Figure 5 and 6 show that the Edge Nodes Cluster load to serve 3 clients workflow requests, if services are launched in shared mode then Docker Engine needs to manage only 3 containers, on the other hand if services are launched with No-share mode then Docker Engine needs to manage 8 containers to fulfill the same 3 client request.

IoT clients have flexibility to design their own custom dataflow order based on business needs and configure the workflow by selecting service components as described Table 1.

WorkFlows:

The client will design their own workflow with the company name along with the workflow specification as described in figure 7:

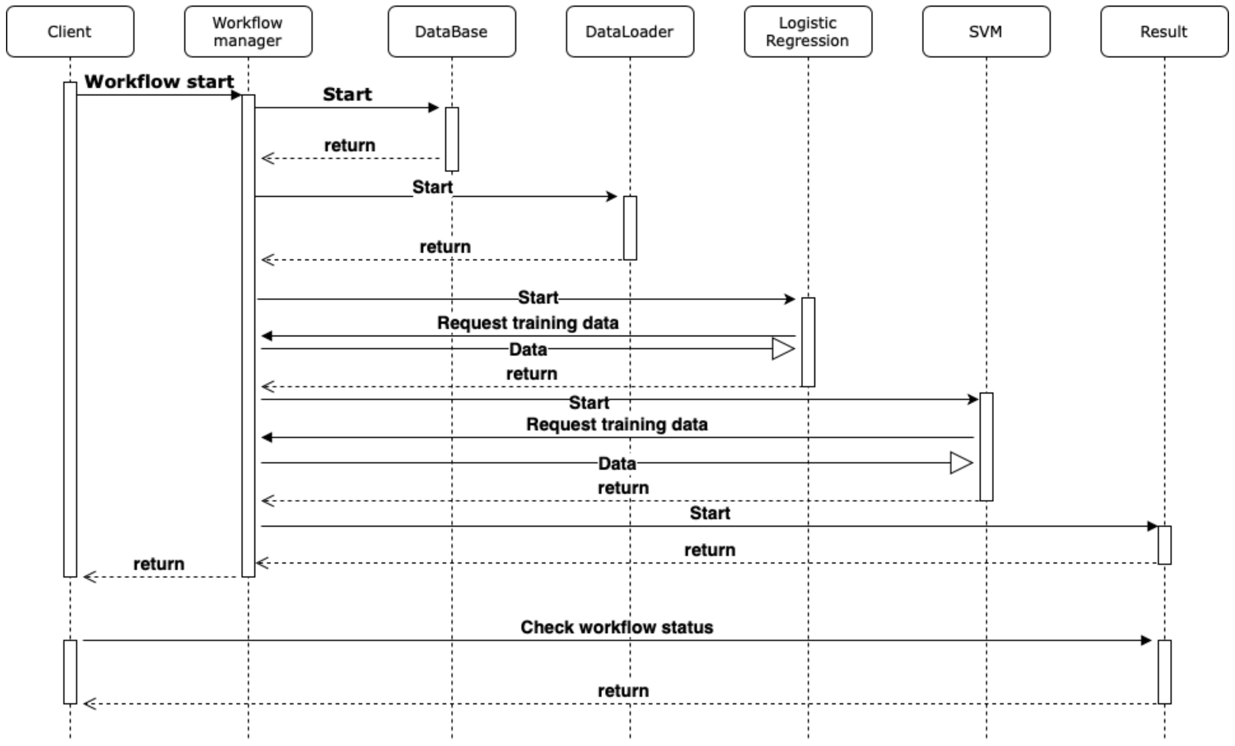


Figure 7: IoT client will request the workflow

Workflow Deployment Request originated from the client machine:

```
{
"client_name": "amazon" ,
 "workflow": "employee",
“workflow_specification”:[["1"],["2"],["3"]],
}
```

Here, client_name: Company Name

workflow: workflow use case i.e., “Employee” or “Hospital”

workflow_specification: ServiceID code assigned to below services
1 : DataLoader component
2 : Logistic Regression component
3 : SVM component

The Manager Daemon process on the server will be the workflow Manager and responsible to handle the client request to launch the workflows components as specified in the workflow_specification of the client request.

When the manager will launch the data loader service, **the data Loader** component will generate the new client table and dataset for the respective client and add it to the cassandra keyspace. Note this dataset will be used for the ML models training. Then the manager launches the **Logistic Regression** and **SVM** models sequentially (as specified in the client request), both components will read the training data from cassandra database and perform the training for the respective clients.

The Manager will also capture the container service address on the Docker Swarm by capturing the IP address of the components and store it into the Hash memory.

Once the deployment request is done (all the components are up and running), the workflow manager will make an Rest API call to each service component to update the service addresses (IP address list) of the launched workflow. Manager will send the below details (Rest API request) to all the components and each Component response ACK to the manager after storing these details in their **hashmap in memory**, as described in figure 8.

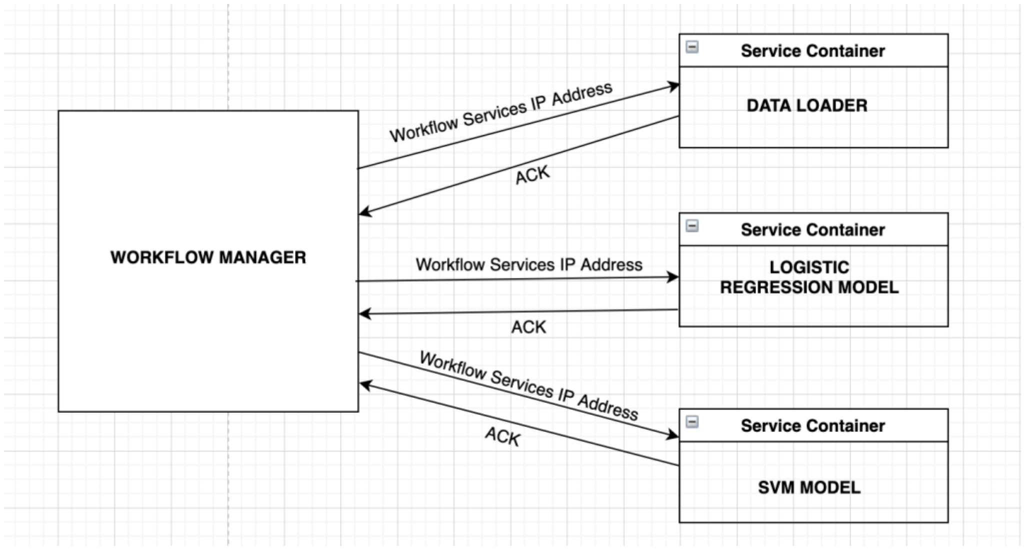

WORKFLOW MANAGER
Workflow Services IP Address
ACK
Workflow Services IP Address
ACK
Workflow Services IP Address
ACK
Service Container
DATA LOADER
Service Container
LOGISTIC REGRESSION MODEL
Service Container
SVM MODEL

Figure 8: Workflow Manager sends IP addresses update calls to each service component

Workflow Manager and all service components will use the Hashmap to store the IP addresses of the services in below format :

Hashmap key format:-

HashMap <key=(workflow, client_name), value=(Service_IP_addresses)>

Example - HashMap update request from Manager to all service components

```
{
"client_name": "amazon" ,
"workflow": "employee",
“workflow_specification”:[["1"],["2"],["3"]],
"ips" : {
        “1” : "10.0.12.31:5000",
        “2” : "10.0.12.32:5001",
        “3” : "10.0.12.31:5002",
        "4":"10.0.12.33:5003"
        }
}
```

Now that all the containers are launched for the respective **client_name** and **workflow**, data sources from IoT client devices will start sending prediction requests to the manager.
Manager will figure out who should the request be sent to by reading details from the hashmap for the respective **client_name** and **workflow**

NOTE: All IoT clients data prediction requests are routed via Manager because Manager access point is exposed to the external world for communication.

Data Flows:

Figure 9 shows the end to end data flow sequence diagram from client device to the final result.

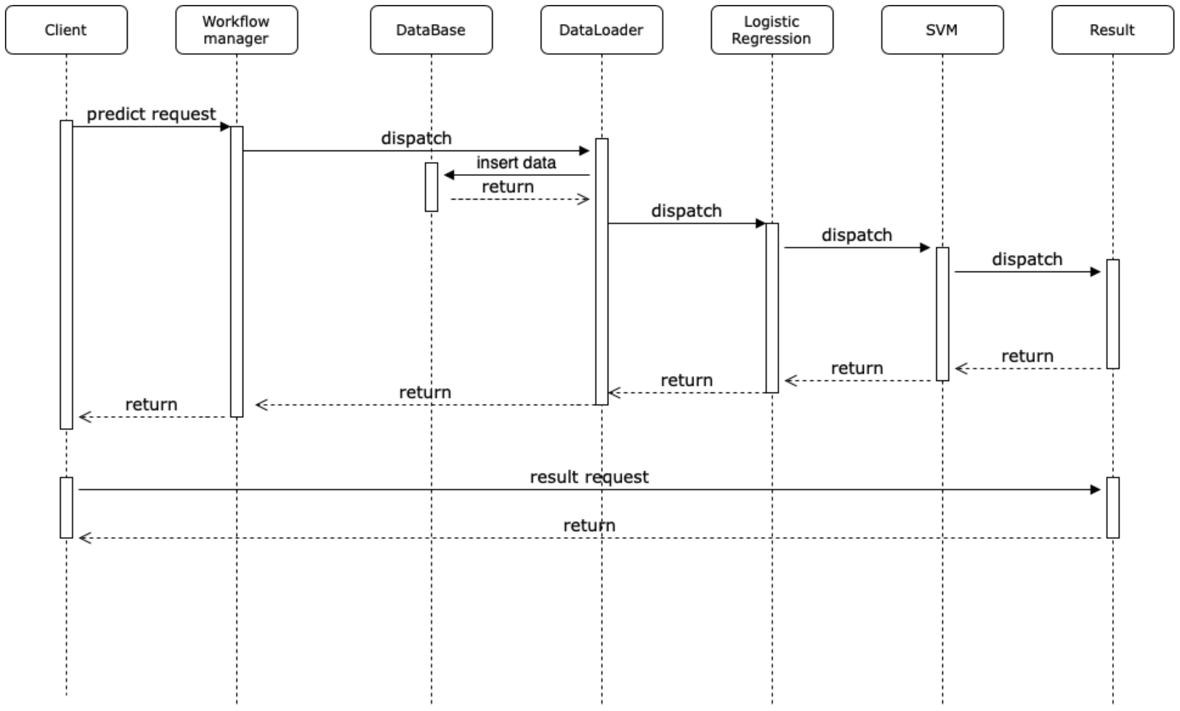
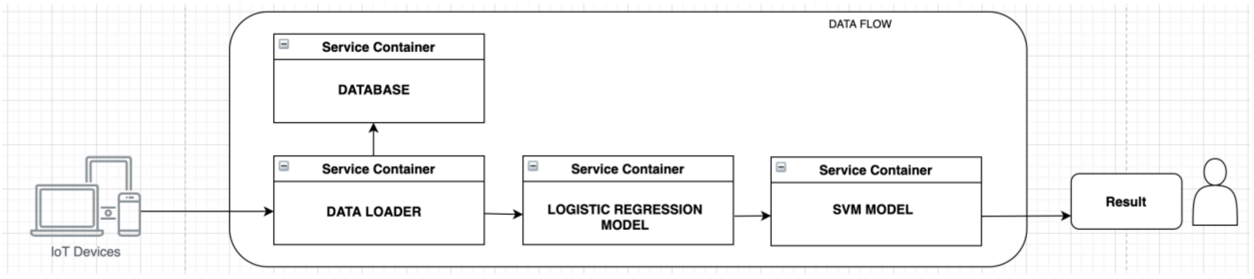


Figure 9: IoT client will supply data with the prediction request

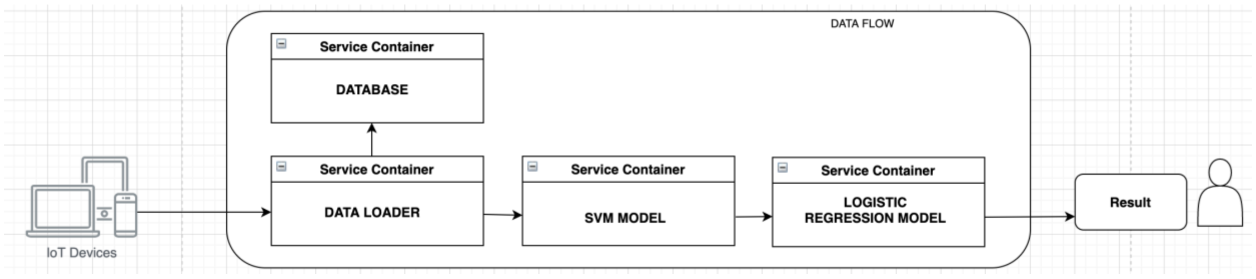
Below are few examples of data flow supported by the existing function:

Data Flow Scenario 1:



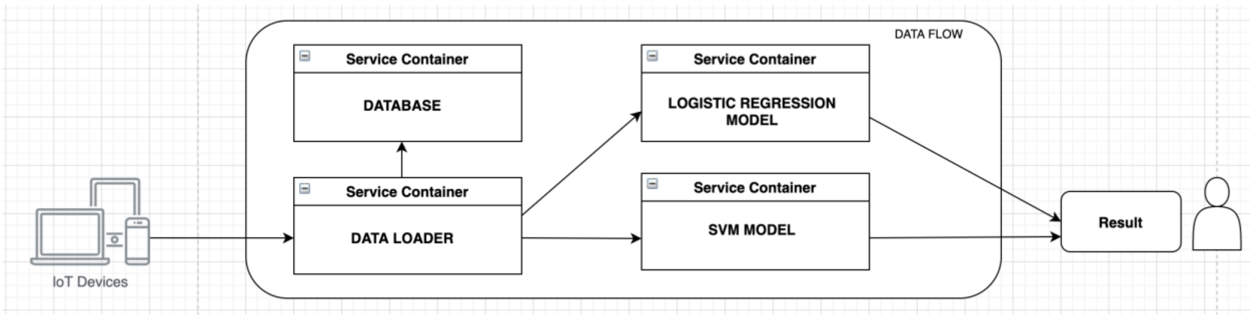
Data Loader (ServiceID#1) -> LR Regression (ServiceID#2) -> SVM Model (ServiceID#3)

Data Flow Scenario 2:



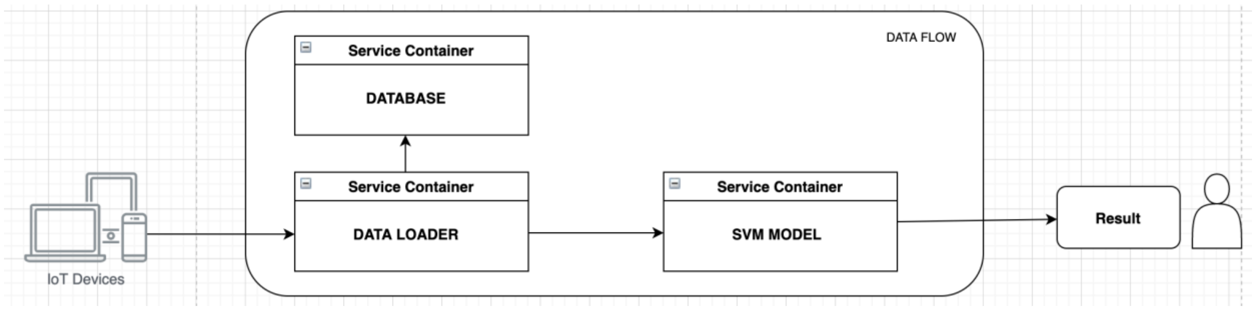
Data Loader (ServiceID#1) -> SVM Model (ServiceID#3) -> LR Regression (ServiceID#2)

Data Flow Scenario 3:



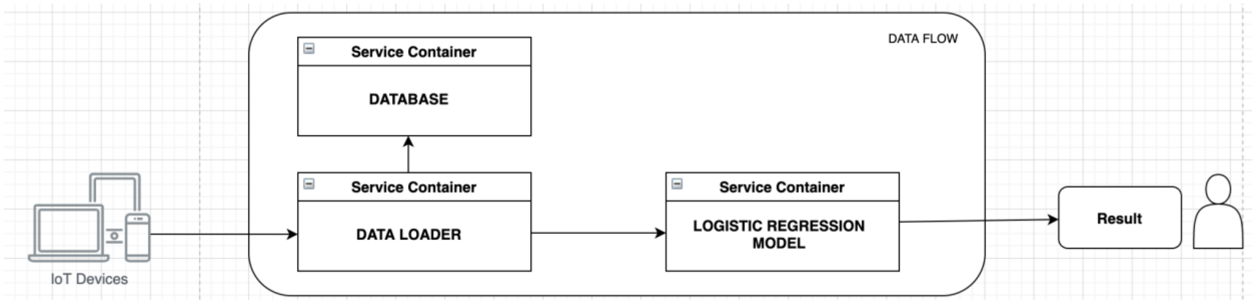
Data Loader (ServiceID#1) -> { LR Regression (ServiceID#2) , SVM Model (ServiceID#3) }

Data Flow Scenario 4:



Data Loader (ServiceID#1) -> SVM Model (ServiceID#3)

Data Flow Scenario 5:



Data Loader (ServiceID#1) -> LR Regression (ServiceID#2)

Data Prediction request Samples:

1. Prediction request from IoT Client to Workflow Manager:

```
{
"client_name": "amazon" ,
"workflow": "employee",
“data” : {“index”:1,"emp_id":"1","dept_type":"1", "gender" : "male", "race": "2",
          "day_of_week":"MON", "checkin_datetime" : "8:00", "time":"14:00"}
}
```

2. Prediction request from Workflow manager to component:

```
{
"client_name": "amazon",
"workflow": "employee"
“data” :  {"emp_id":"1","dept_type":"1", "gender" : "male", "race": "2", "day_of_week":"MON",
           "checkin_datetime" : "8:00", "time":"14:00"}
}
```

3. Prediction request from one service component to another service component:

```
{
"client_name": "amazon",
"workflow": "employee"
“data” : {"emp_id":"1","dept_type":"1", "gender" : "male", "race": "2", "day_of_week":"MON",
            "checkin_datetime" : "8:00", "time":"14:00"}
"analytics": [{"start_time":123,"end_time":130, “prediction_result”: 100}]
}
```

Here, each component will append its performance statistics to the analytics field before passing down to the next component along with the original request data in the data flow. Also each component will perform its assigned job, for example, Data Loader will add new data coming from IoT devices to the client respective tables in the cassandra, Logistic Regression and SVM models will perform predictions and forward results to the result component and result component will consolidate and send output to the client.

4. Prediction request from component to result:

```
{
"client_name": "amazon",
"workflow": "employee"
“data” :  {"emp_id":"1","dept_type":"1", "gender" : "male", "race": "2", "day_of_week":"MON",
            "checkin_datetime" : "8:00", "time":"14:00"}
"analytics": [{"start_time":123,"end_time":130, “record_count”: 2001},
               {"start_time":133,"end_time":135,”prediction_LR”:”17:00”},
              {"start_time":136,"end_time":138,”prediction_SVM:”266”}]
}
```

Performance and Load Evaluation:

Note: In all the experiments, we have hosted Cassandra database and Prometheus on the manager node and hence the CPU, memory and network usage of the manager node is higher.

Control params: Number of workflows, Number of prediction requests, Training data size

Experiment 1: Edge Cluster workload comparison between Service Reuse vs No-Reuse mode deployment

As explained figure 5 and 6, we extended our experiment to run first 20 client workflow deployment requests which include all the service components with No-Reuse mode (10 workflows) and with Reuse mode (10 workflows), separately: -

Clients	Data Loader	LR Model	SVM Model	Total # of containers deployed on the server with No-Reuse mode	Total # of containers deployed on the server with Reuse mode
1	✓	✓	✓	3	3

2	☑	☑	☑	6	3
3	☑	☑	☑	9	3
4	☑	☑	☑	12	3
5	☑	☑	☑	15	3
6	☑	☑	☑	18	3

Table 2: Table shows top 6 clients deployment requests with all service components in No-Reuse mode and Reuse mode

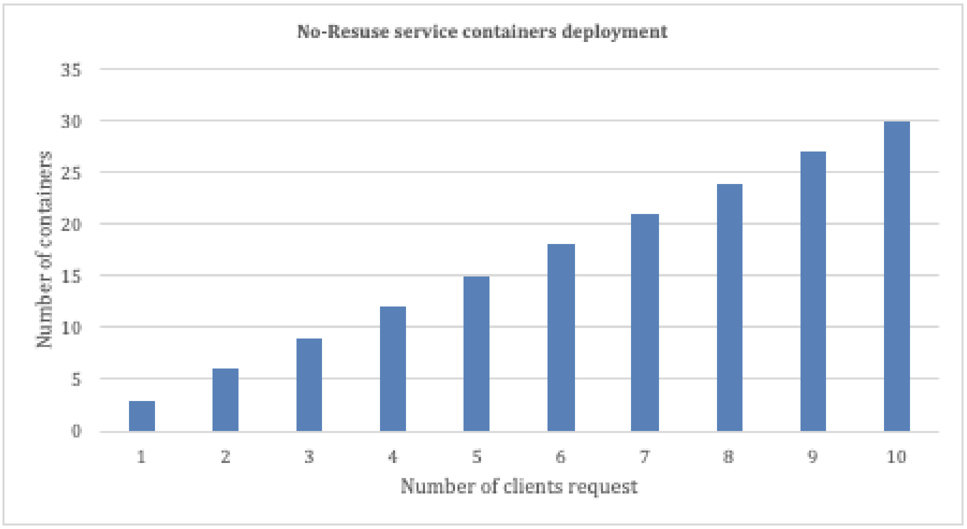


Figure 10: Edge cluster load with No-Reuse mode deployment of 10 workflows

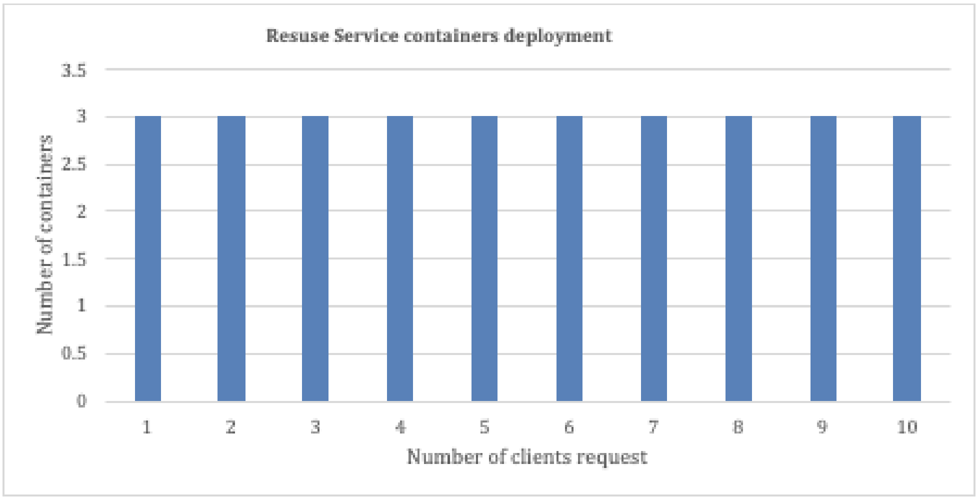
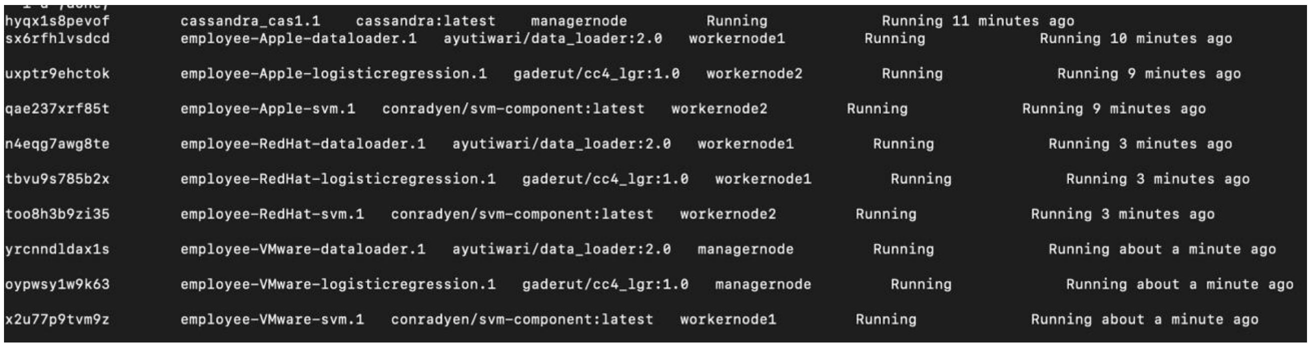


Figure 11: Edge cluster load with Reuse mode deployment of 10 workflows

Observations clearly state that, number of containers will increase in multiple of Number of services in the workflow request. But incase of reuse, the number of containers will stay constant.
<u>Docker swarm workload distribution</u>

Number of containers on each node after deployment for 3 clients[No-reuse]



```
hyqx1s8pevof       cassandra_cas1.1    cassandra:latest    managernode        Running          Running 11 minutes ago
sx6rfhlvsdcd       employee-Apple-dataloader.1   ayutiwari/data_loader:2.0   workernode1        Running             Running 10 minutes ago

uxptr9ehctok       employee-Apple-logisticregression.1   gaderut/cc4_lgr:1.0   workernode2        Running             Running 9 minutes ago

qae237xrf85t       employee-Apple-svm.1   conradyen/svm-component:latest   workernode2        Running             Running 9 minutes ago

n4eqg7awg8te       employee-RedHat-dataloader.1   ayutiwari/data_loader:2.0   workernode1        Running             Running 3 minutes ago

tbvu9s785b2x       employee-RedHat-logisticregression.1   gaderut/cc4_lgr:1.0   workernode1        Running             Running 3 minutes ago

too8h3b9zi35       employee-RedHat-svm.1   conradyen/svm-component:latest   workernode2        Running             Running 3 minutes ago

yrcnndldax1s       employee-VMware-dataloader.1   ayutiwari/data_loader:2.0   managernode        Running             Running about a minute ago

oypwsy1w9k63       employee-VMware-logisticregression.1   gaderut/cc4_lgr:1.0   managernode        Running             Running about a minute ago

x2u77p9tvm9z       employee-VMware-svm.1   conradyen/svm-component:latest   workernode1        Running             Running about a minute ago
```

<u>Observation</u>: Swarm launched 3 containers on manager node, 4 containers on workernode1 and 3 containers on workernode2. We have used docker command to get the container loads from the all nodes for each service:

```
“sudo docker service ls --format {{.ID}}  | while read line ; do sudo docker
service ps $line -f desired-state=Running | sed '1 d';done;”
```

<u>**Experiment 2: Container services deployment time:**</u>

By comparing figure 12 and figure 13 large reduction in deployment time when applying reuse container methods to deploy new workflows. The second and third workflow reduces from 85 second to 24 seconds , more than 50% reduction, when applying the reuse method. The main reason the first workflow starts slower is due to the cassandra database which takes around 90 seconds to start.Thus launching a separate database for each workflow is not a practical design.

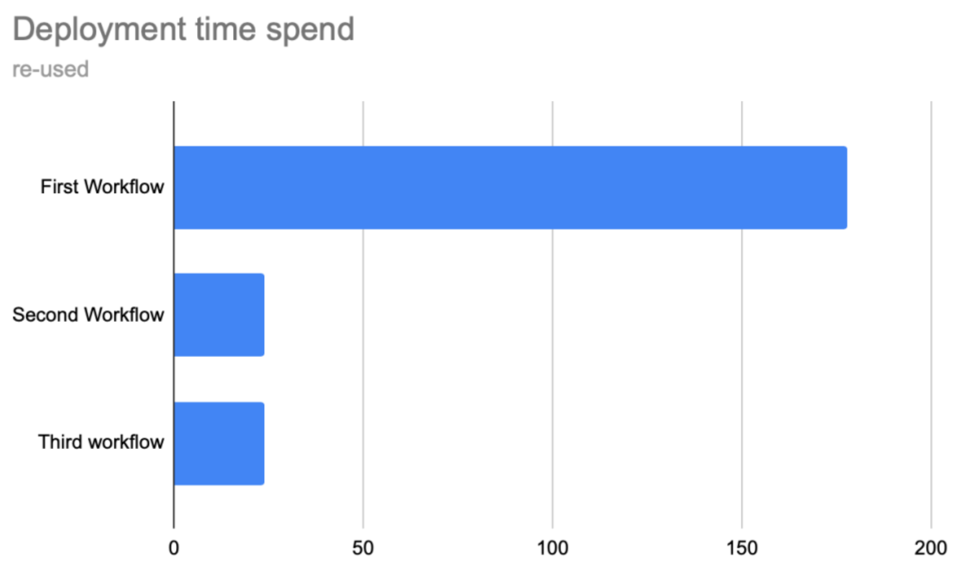


Figure 12. Deployment time on reused deployment method.

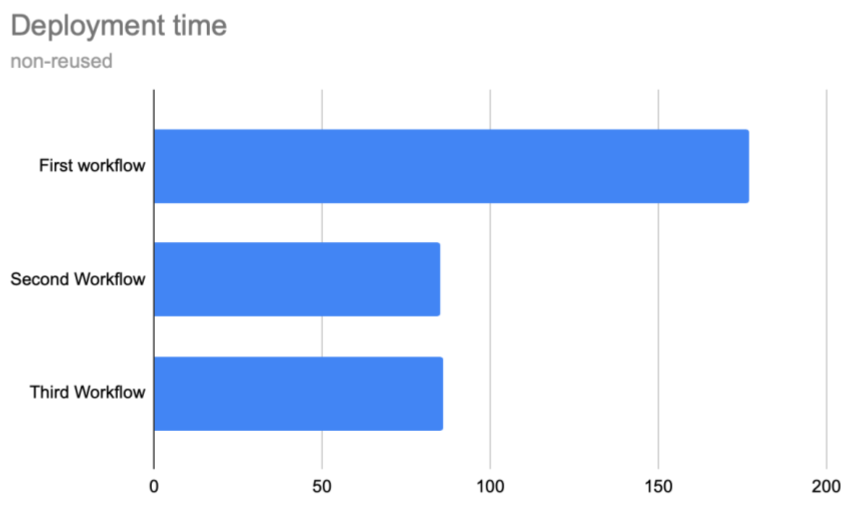


Figure 13. Deployment time on non-reused deployment method.

Table 3: Docker service container deployment time

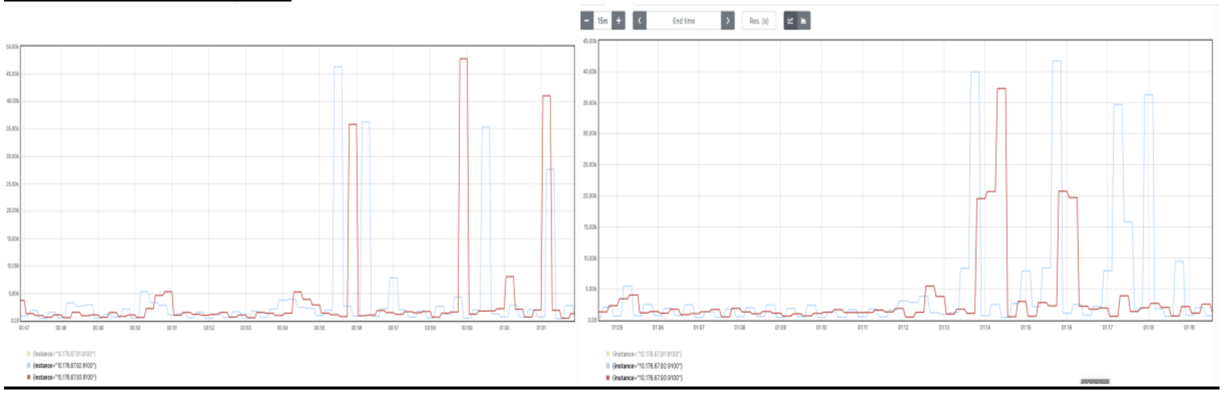
Service Component ID	**Service Component Name**	**Docker Hub Image size**	**Time taken for the component to created and start (seconds)**
1	Data Loader	336 MB	5.7
2	Logistic Regression Model	392 MB	10.29

3	SVM Model	349MB	9.3

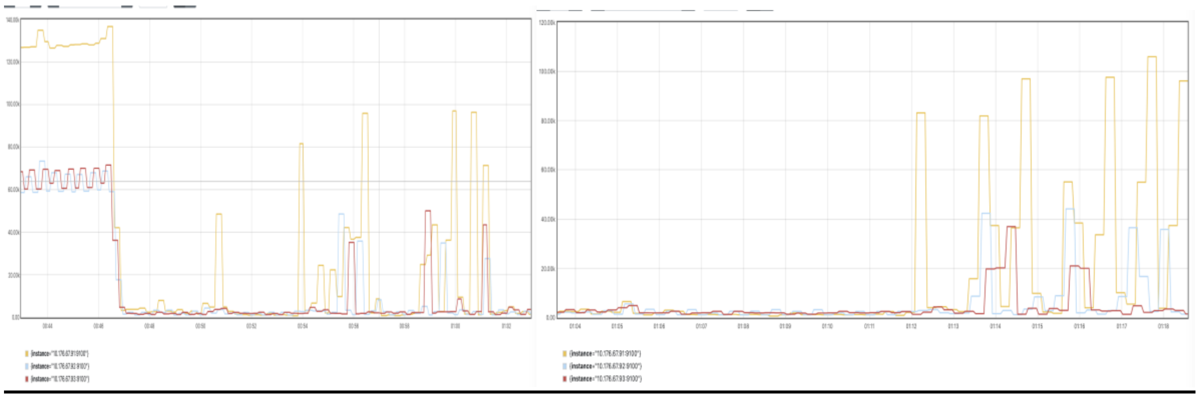
We used docker command to get the service image deployment creation time and start time from each service container inspect logs:

"sudo docker container inspect <service_container_name/Id>"

Network receive bytes: ***reuse vs noreuse***



Network transmit bytes: ***reuse vs noreuse***



Observation: From the above graph above it is easy to see the noreuse mode has more network bandwidth usage since we are making more requests on creating docker service. At the container start time workflow manager sends required data to the container this also accounts for the increase in network loading.

Experiment 3: Service Response times:

Service average response time is captured separately for two types of deployment methods–

- **Service Response time with Reuse mode.**
- **Service Response time with No-Reuse mode.**

In both cases client IoT devices send data periodically with ML model prediction requests. Data size will vary based on the number of requests initiated by the client device.

In this experiment we have considered 3 clients sharing all the services, and generating data and prediction calls periodically in parallel from the IoT devices. We have simulated this experiment with Employee use case workflow considering clients are opted-in for all the services:

a) Service Response time with reuse mode

In this experiment we have considered 3 clients using their workflows in reuse (shared service) mode, all client will share service containers:

Client name	**Number of requests**	**Data Loader Average Response time** (captured in the docker container) **(milli seconds)**	**LR Model Average Response time** (captured in the docker container) **(milli seconds)**	**SVM Average Response Time** (captured in the docker container) **(milli seconds)**	**Average Total time taken to finish the client request.** (captured on the client terminal) **(milli seconds)**
Client 1	100	43.67	2.59	0.46	46.72
Client 2	100	54.00	2.74	0.54	57.29
Client 3	100	39.64	2.81	0.433	42.88

b) Service Response time with No-Reuse mode

In this experiment we have considered 3 new clients using their workflows in No-Reuse (No shared service) mode, each client will have it's own dedicated service containers:

NOTE: Here we shutdown previous experiment (Service Response time with Reuse mode) containers before starting this experiment.

Client name	**Number of requests**	**Data Loader Average Response time** (captured in the docker container) **(milli seconds)**	**LR Model Average Response time** (captured in the docker container) **(milli seconds)**	**SVM Average Response Time** (captured in the docker container) **(milli seconds)**	**Average Total time taken to finish the client request.** (captured on the client terminal) **(milli seconds)**
Client 1	100	35.23	2.87	0.44	38.56
Client 2	100	50.381	4.166	0.521	55.068
Client 3	100	32.6408	2.9339	0.4776	36.052

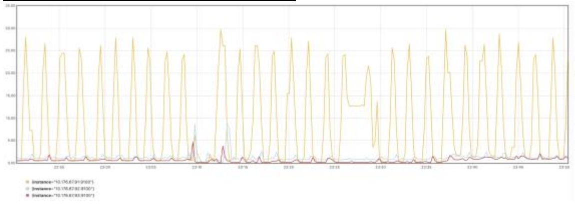
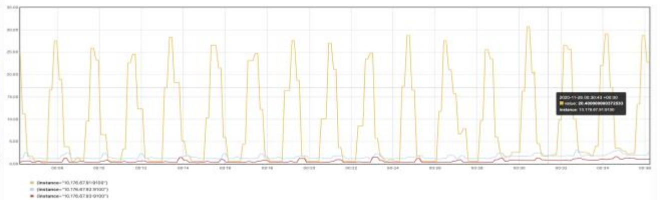
Observations: From the above experiment we have noticed that shared service mode response time is greater than the no-share service mode response time. In reuse service mode all containers are shared and each shared service

container is handling many requests from multiple clients, if service is busy processing one request then the other request will be buffered in a wait queue and as soon as resources are available requests get processed in the receiving order.
Same case with no-reuse containers but those containers get less traffic (less calls) because they are serving only one client and each client has their own set of service containers.

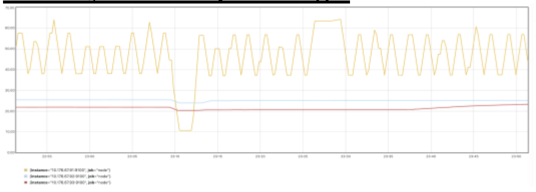
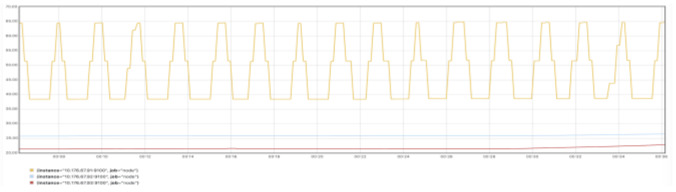
<u>Experiment 4: Node performance</u>

In this experiment we have measured CPU utilization, Memory utilization and Network bandwidth utilization parameters with heavy load on the cluster.
Launched reused (shared service mode) for the 3 clients with 1000 prediction requests per minute. Similarly, no-reuse (no-shared services) scenario for 3 clients with 1000 prediction requests per minute.

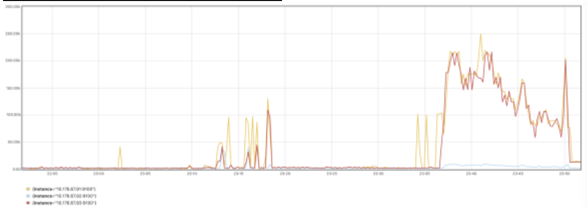
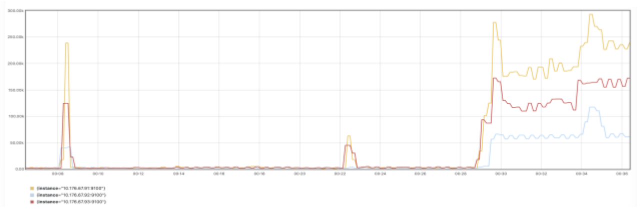
<u>CPU usage percentage:</u> *reuse vs noreuse*

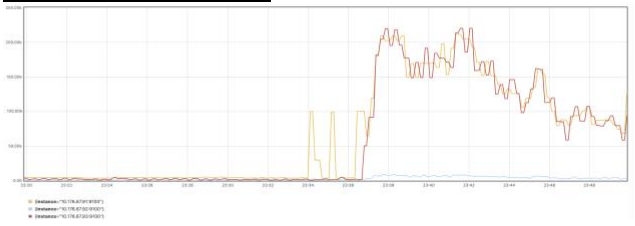
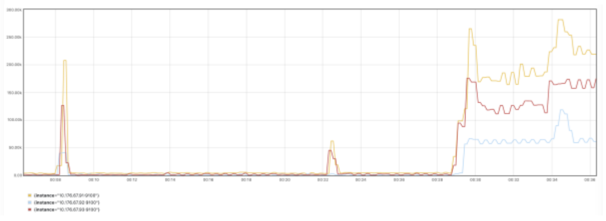
<u>Memory available percentage:</u> *reuse vs noreuse*

<u>Network transmit bytes:</u> *reuse vs noreuse*

<u>Network received bytes</u>: *reuse vs noreuse*

<u>Observations</u>:
When certain number of workflows are deployed (in reuse mode & no reuse mode), prediction requests are triggered and then measuring the Node CPU, memory and network usage

The manager node has the most number of containers and Cassandra runs on the manager node hence the CPU utilization is high on the manager node. For memory usage in reuse mode due to more data storing in memory higher memory usage can be observed in the reuse case. on the other hand, in the non-reuse case has lower memory usage.
Network transmitted bytes and received bytes are almost the same in both the cases because for every prediction request there has to be a network call made, irrespective of the container getting reused or not. The memory utilization is more in the case of reuse, as we can see the worker nodes also utilize memory whereas in the case of no-reuse worker nodes do not utilize much memory.

Docker image size optimisation:

Docker image size issue

One of the performance issues of the system is the size of the docker container. A normal build docker image can be more than 1 GB when the size of the actual app is only several MB. That is a lot of overhead especially on the system we are in. There are four different docker images which means that in the worst case scenario will have to transfer 5 to 6 GB of data alone on docker images without accounting data transfer for actual computing usage.

Improvement

One technique is to do docker layer build. Instead of package every library and package in every docker images layer build can build small images for just libraries which can be re-used when launching containers. For example, python Machine learning libraries are in both Logistic regression and support vector machine components. When using docker layer build the layers containing these libraries can be shared by these two components. As seen in Table 1. There are two different docker layers being built one is builder another one is base. Builder is only for installing libraries than the base layer can copy these libraries. The builder layer can be re-used in later image builds which results in faster image building time due to no need to install libraries in every build.

To further reduce the size of docker images there are some pre-built smaller size base images available on docker hub. As in Table 1 line one python:3.7-slim image. These smaller base images reduce the pre installed libraries and use lightweight OS like alpine. Smaller based images are suitable for the use case since there are not many system libraries called in application also can install the needed libraries using the previous mation docker layer build.

Table 4. Example docker file of docker layer build and layer caching

```
FROM python:3.7-slim as base                           # Pull small size base image
FROM base as builder                                   # Create libraries install layer
RUN mkdir /install
WORKDIR /install
COPY requirement.txt /requirement.txt
RUN pip install --prefix=/install -r /requirement.txt  # Install libraries
FROM base                                              # Start from base layer
COPY --from=builder /install /usr/local                # Copy installed libraries
COPY . /app                                            # Copy actual application
WORKDIR /app
CMD [ "python3", "model.py" ]                          # Start command
```

Image size improvement result

Take the SVM component as an example.As in Table 2, The size of the image without docker layer caching is 1.25 GB. After applying above techniques the size of the image reduced to 349 MB which results in a 75% reduction in docker image size. In some special cases the application can be compiled to a text file more specifically, the frontend application built using react can be compiled from .jsx to HTML file. can further reduce the image size. The frontend and load balancer image size reduced from 1GB with direct docker image build to 17MB,shown in Table 2, by compiling the .jsx to HTML combined with the docker layer caching.

Table 5. Size comparison on before and after docker layer caching.

Component	Image size without docker layer caching	Image size with docker layer caching
Data loader	933 MB	336 MB
Logistic regression	1.23GB	392 MB
SVM	1.28GB	349MB
Frontend+Load balancer	1.09GB	17MB

Conclusion

In conclusion, reuse and no reuse case has similar cold start time but the noreuse case had a higher overhead for the second and above workflow. On the response time perspective noreuse case has smaller response time due to the container only processing requests from one workflow. Both reuse and no-reuse methods have similar loading on hardware and network utilization. The reuse case has slightly higher memory usage because the requesting routing data is being stored in memory.

APPENDIX

Project Code Repository: https://github.com/gaderut/CloudComputing-docker

- Load balancer and Frontend : https://github.com/Conradyen/workflow-generator

Each folder corresponds to the code for each component. For example, the SVM folder has the code for the SVM container.

Technology Stack:

- Operating system:
    - CentOS
    - Alpine Linux
- Frontend:
    - ReactJS
- Database:
    - Cassandra
- Web Framework:
    - Python
    - Flask
    - Node.js
    - Express
- Load Balancing and web server :
    - Nginx
    - Ngrok
- Machine Learning Framework:
    - Scikit-Learn
- Deployment:
    - Docker
    - Docker Swarm
- Tools to capture performance metrics
    - Prometheus
    - Node exporter

Operational Manual:

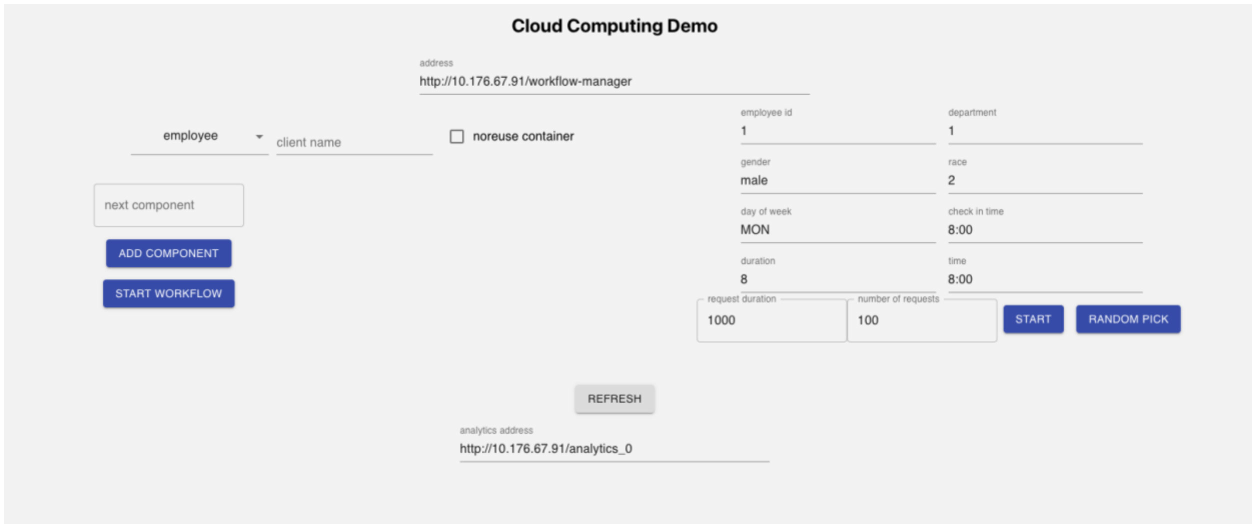
Installation:

Make sure the following libraries are installed:

- Python3
- Flask
- Flask_cors
- Docker

1. Login into the server cluster4-1.utdallas.edu with the generic account
2. Install Docker Swarm and add all the nodes to the swarm.

3. Open linux terminal, Pull the code repository from ‘https://github.com/gaderut/CloudComputing-docker’ and run cd manager/
4. Then run sudo python3 app.py (Workflow manager is started)
5. Then Run : sudo docker run -d -p 80:80 conradyen/demotool (UI is started) .
6. Go to http://10.176.67.91/ to issue workflow request



address: http://10.176.67.91/workflow-manager

employee | client name | noreuse container

employee id: 1	department: 1
gender: male	race: 2
day of week: MON	check in time: 8:00
duration: 8	time: 8:00
request duration: 1000	number of requests: 100

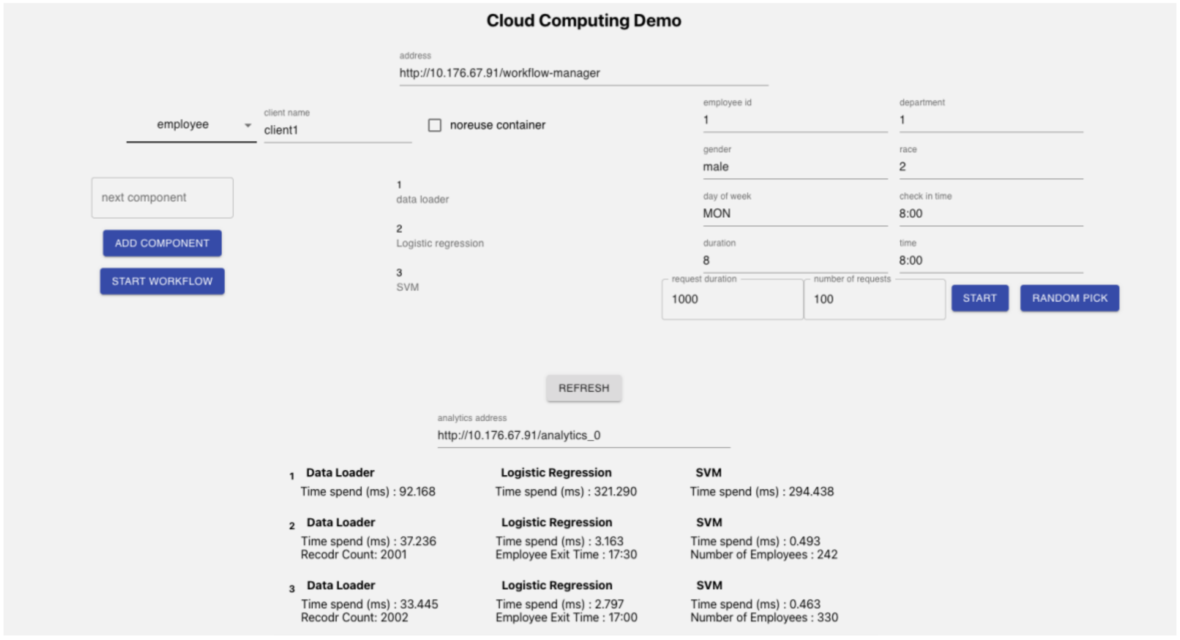
analytics address: http://10.176.67.91/analytics_0

6.1 Select the type of workflow(employee or hospital) from the dropdown and mention the name of the client in the input field besides it.

6.2 In the next component field mention the workflow specification using the “Add Component” button and Click the “Start Workflow” button to start the workflow.

6.3 Check the box again “No reuse container” to avoid reusing the containers.

7. To start the data generator the client can mention the frequency and number of requests and click the “Start” button.



address: http://10.176.67.91/workflow-manager

employee | client name: client1 | noreuse container

1 data loader
2 Logistic regression
3 SVM

employee id: 1	department: 1
gender: male	race: 2
day of week: MON	check in time: 8:00
duration: 8	time: 8:00
request duration: 1000	number of requests: 100

analytics address: http://10.176.67.91/analytics_0

	Data Loader	Logistic Regression	SVM
1	Time spend (ms) : 92.168	Time spend (ms) : 321.290	Time spend (ms) : 294.438
2	Time spend (ms) : 37.236 Recodr Count: 2001	Time spend (ms) : 3.163 Employee Exit Time : 17:30	Time spend (ms) : 0.493 Number of Employees : 242
3	Time spend (ms) : 33.445 Recodr Count: 2002	Time spend (ms) : 2.797 Employee Exit Time : 17:00	Time spend (ms) : 0.463 Number of Employees : 330

8. Click on the "Refresh" button to see the Results. The user can see the processing time for each component. The Logistic Regression results show the predicted exit time for an employee and the SVM results show the predicted number of employees in the office at the given point of time.

References:

1. https://docs.docker.com/engine/swarm/swarm-tutorial/
2. https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/
3. https://docs.docker.com/engine/swarm/swarm-tutorial/add-nodes/
4. https://alibaba-cloud.medium.com/how-to-install-and-configure-docker-swarm-mode-on-centos-7-c0b32f0fbc82