DeFog: *Demystifying Fog System Interactions Using Container-based Benchmarking*

Software Development Report

Jonathan McChesney - 40126401

Contents

1. System Specification

DeFog currently integrates several fog applications for containerised fog application benchmarking:

- **YOLOv3**: Deep learning object classification tool using AlexNet weights.
- **PocketSphinx**: Continuous audio to text speech generation tool.
- **Aeneas**: Forced alignment tool (aligning text transcriptions to audio segments).
- **FogLAMP**: Internet of Things (IoT) Edge gateway application using Edge sensors.
- **iPokeMon**: Latency critical virtual reality (VR) GPS based online mobile game.

The system uses three pipelines to deploy applications and execute benchmarks: Cloud Only, Edge Only and Cloud-Edge combined as outlined in figure 1.

For each pipeline the data model follows a similar process. The Cloud-Edge pipeline has an additional step of transferring a pre-trained weight or model asset from the Cloud to the Edge as shown in figure 2.

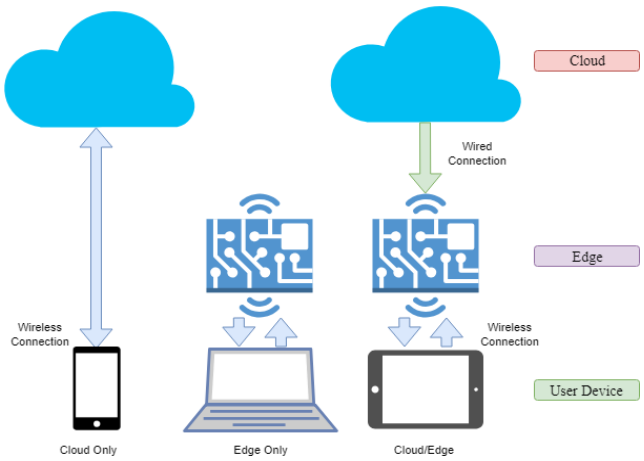

Figure 1: High Level DeFog Pipeline Overview

1.1 Data Model:

Input: Assets and data payloads:

- **YOLO input data**: Unlabelled Image set (.jpg and .png) to be classified. A pretrained AlexNet weights dataset is used as the Cloud model input for the Cloud-Edge pipeline.
- **PocketSphinx input data**: Audio files (.wav) to be converted into text. An en-us acoustic language model is used as the model input transferred from, the Cloud for the Cloud-Edge pipeline.
- **Aeneas input data**: A text (.txt) and audio (.wav) set to be segmented and aligned. Pre-set audio files are used for model input on the Cloud-Edge pipeline.
- **FogLAMP input data**: Text (.txt) files containing curl commands to simulate sensor interaction/data.
- **iPokemon client data**: JMeter (.jmx) payload to simulate player behaviour

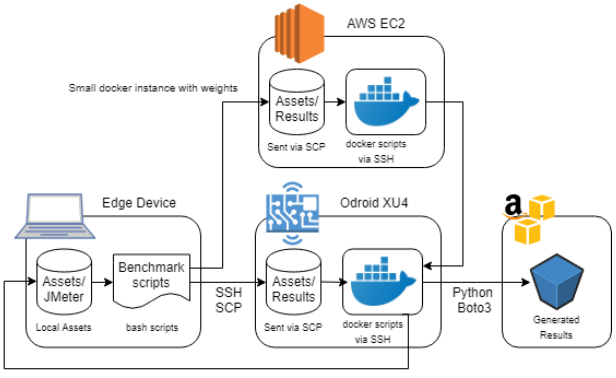

Figure 2: Cloud-Edge Pipeline Process

Compute Task: A computational task unique to each application is performed, such as *image classification, speech to text generation, forced alignment and simulated sensor interaction*.

Output Predicted Data: Application output data:

- **YOLO output data**: Classified and labelled image (predict.png).
- **PocketSphinx output data**: Text (result.txt) file containing the converted audio input.
- **Aeneas output data**: Mapping (map.smil) file containing the aligned audio to text segments.
- **FogLAMP output data**: '*Beautified'* text (foglampotput.txt) output from cURL command.
- **iPokemon server data**: Server response data metrics.

Output Metrics: Benchmark metric results

- **Communication metrics**: Time in flight, S3 transfer time, results transfer time, Cloud-Edge transfer time and communication latency.
- **Computation metrics**: Execution Time, return trip time, real time factor and compute cost.
- **Data Transfer metrics**: Bytes uploaded, bytes uploaded per second, bytes downloaded, and bytes downloaded per second.
- **Apache JMeter & Taurus Concurrency metrics**: Response latency times and response status metrics for concurrent users.
- **Platform metrics**: CPU Model, number of cores, CPU frequency, I/O Rate, network latency.

Additionally, DeFog provides functionality to run system benchmarks on the destination platform using third party system benchmark tools Unixbench and Sysbench.

1.2 Data Flow:

1. The user interacts with the DeFog command line tool on the edge device, e.g. a laptop, and selects the a series of benchmark options. DeFog then uses secure shell (SSH) to remotely access the destination service securely on the destination platform.
2. If DeFog has not been run on the destination platform before, then the directory architecture will be initialised.
3. Platform benchmarks are *optionally* run and then transferred back to DeFog on the Edge device (not available on the Cloud-Edge).

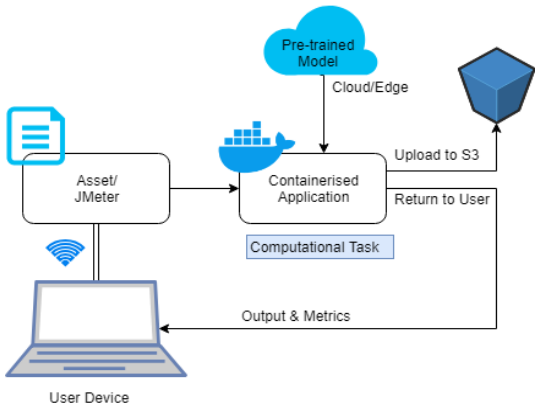

Figure 3: DeFog Benchmark Workflow

4. If the application docker image has not been built before, then the respective application docker folder from the DeFog repository will be fetched and built on the Cloud or Edge.

Yolo, PocketSphinx & FogLAMP:

5. An asset payload is transferred using SCP from the edge device to the Cloud or Edge.
6. The fog application is benchmarked for all assets within the asset folder. This captures the compute and communication metrics on the destination system. If running the Cloud-Edge pipeline, then the model input (e.g. weights file or acoustic model) is transferred to the Edge from the Cloud, before the computational task specific to each fog application is executed.
7. The resulting assets, e.g. the predicted image generated during the YOLO benchmark process, is uploaded to an AWS S3 instance and the catalog of captured metrics as well as the data asset payload are then transferred back to the local edge device running DeFog.
8. DeFog correlates the supplied metric data, calculates the remainder of the metrics and then outputs a comma separated file (.csv) and verbose results text file. These files detail the catalog of metrics generated. Results are available in the "*DeFog/results*" sub folder.

iPokeMon:

5. JMeter is invoked using the provided user input and .jmx file to simulate player behaviour.
6. Taurus is then invoked to generate further metrics using the input entered by the user to capture response success rates and latency metrics for concurrent users.
7. Results from JMeter and Taurus are then saved locally and returned to the user. These results are also available in the "*/DeFog/results*" sub folder.

1.3 Assumptions & System Constraints:

Regarding the Cloud-Edge pipeline running Yolo, PocketSphinx and Aeneas, it is assumed that model training etc, has occurred beforehand on the Cloud, due to the availability of superior hardware. This removes the compute intensive overhead of training weights and recomputing large training sets.

Regarding constraints, only a small number of asset workloads are used to benchmark the fog applications. The local network where experiments were performed was often slow and erratic. A constraint was implemented to prevent multiple images and containers of the same application build from being simultaneously built and run. This was to minimise memory use on the Edge to avoid overloading, this constraint was removed during the experiments. allow concurrent use.

The Odroid XU4 board and the Raspberry Pi 3 boards are used to simulate edge nodes. Edge nodes have a limited availability of hardware and reside closer to the user device, especially when compared to the geographically distant Cloud. DeFog makes use of only one Cloud systems within its benchmarking, Amazon Web Services. The EC2 instance created and used for the experiments was setup in Dublin, Ireland. The assumption for this decision is that a user of a system would ideally be connecting to a relatively close Cloud server in the best case.

1.4 Functional Requirements:

- Run system benchmarks (Network, CPU and I/O).
- Run application benchmark on the Cloud and the Edge.
- Run application benchmarks on the Edge using user input (amd pre-trained Cloud weights).
- Gather communication latency metrics.
- Gather computation latency metrics.
- Generate cost metrics for the Cloud and estimate the Edge.
- Generate byte transfer rate metrics (upstream and downstream).
- Integrate TPI system testing tools Unixbench and Sysbench.
- Integrate TPI testing tools Apache JMeter and Taurus to simulate user behaviour/load test.
- Allow for command line user interactivity and on-screen verbose feedback.
- Convert metrics to a comma separated file (.csv) and verbose text file (.txt).
- Display output metrics and application predictions to user through the command line.
- Autonomously build application docker images with verbose output.
- Run docker containers in detached mode with verbose output.
- Remove application docker containers and images when benchmark un finishes.
- Error handling and user feedback if a user enters incorrect or unexpected input.
- Allow multiple edge nodes to be selected for benchmarking (i.e. multiple boards).

1.5 Non-functional Requirements:

- **Performance**: The system design will allow for bound connection times, i.e. if an attempt to connect to the Edge or Cloud platform exceeds 30 seconds then the connection will be aborted, and a message will be displayed to the user. Verbose visual feedback will be displayed to the user for long running tasks such as building application containers.
- **Operational**: At least 95% of users will be able to set up and run benchmarks using DeFog while consulting the relevant documentation available on the Git repository.
- **Reliability**: The system will have less than 3% packet loss, if a command or connection fails then the user should be informed. The DeFog system should is not a live service, and so should allow use 24/7, 365 days a year, with an uptime rate of less than 95%, with periods of downtime for repository maintenance, e.g. 438 hours per year depending on the platforms.
- **Security**: OWASP security documentation will be consulted as well as AWS security documentation, to ensure standards are adhered to when leveraging different services, i.e. using IAM user credentials with necessary privileges, secure shell and secure copy protocol.

- **Maintainability**: The system will be extensible and easily maintainable; the mean time to repair and the mean time to refactor should be less than 6 hours. To accomplish this, coding standards such as SOLID design principles and OWASP security documentation were followed where appropriate throughout development.
- **Concurrency**: At least 3 distinct application containers can be benchmarked simultaneously, by running DeFog in separate terminals.
- **Scalability**: The design will allow for scaling up for larger user bases, this should be tested using JMeter load tests. The system leverages Cloud architecture and versatile edge nodes which also allow for scaling in the future, handling with different generation user devices.
- **Extensibility**: It should take a developer less than 5 hours to become fairly acquainted with the codebase due to informative code comments and intuitive structure. Refactoring, adding new applications and metrics should require less than 6 effort hours to implement.
- **Failure Handling**: Users will be altered 100% of the time if their input is invalid or if a connection cannot be established with the Cloud or Edge service within 30 seconds.
- **Availability**: The system is limited to the Cloud and Edge infrastructure's availability. The system should always be available (provided infrastructure availability), with minimal maintenance downtime allocated per year.
- **Interoperability/Integration**: The system will integrate various third party integration (TPI) tools such as Unixbench and Sysbench, and automate JMeter and Taurus. The extensible design will allow for a plethora of tools to be integrated in the future.

1.6 System Design & Component Implementation:

The DeFog system and component design is outlined in figure 4. The system makes use of several scripts such as the Defog scripts which accepts user input and runs the benchmarking process, Docker scripts which build and run applications, and helper scripts which extend DeFog functionality.

Docker Scripts: The docker component contains several scripts that pertain to building application images, running containers and executing the compute tasks. Each application folder has their own docker components. *A template component is also found in the DeFog/dockerScripts folder.*

- **Dockerfile**: Builds the necessary dependencies and pulls the modified applications using git. The *execute.sh* script is passed in as a volume and the root working directory is used. The Debian environment argument is set so that no dialog interaction is required.
- **build.sh**: Builds the application image using a verbose pseudo interactive terminal. An image name is set, this is used to create a container instance from the specific image build.

- **enter.sh**: Manually enter a container, volumes are attached to provide the container access to external resources. This is run in interactive mode and is removed upon exiting.
- **execute.sh**: Executes a compute task and generates relevant metrics. Volumes are attached to the container, and the pipeline ID and application ID are passed in as parameters. A Cloud model asset is transferred from the Cloud for Cloud-Edge Pipeline. The results are transferred to the S3 bucket using *s3Upload.py* and the metrics are sent to the user device.
- **remove.sh**: Stops and removes all docker containers and images from the platform.
- **run.sh and runedge.sh**: Runs the application container in non-interactive mode on the Edge or Cloud. Volumes are mounted and the *execute.sh* script is invoked with application ID and pipeline ID passed in as parameters. The name parameter corresponds to an image build.

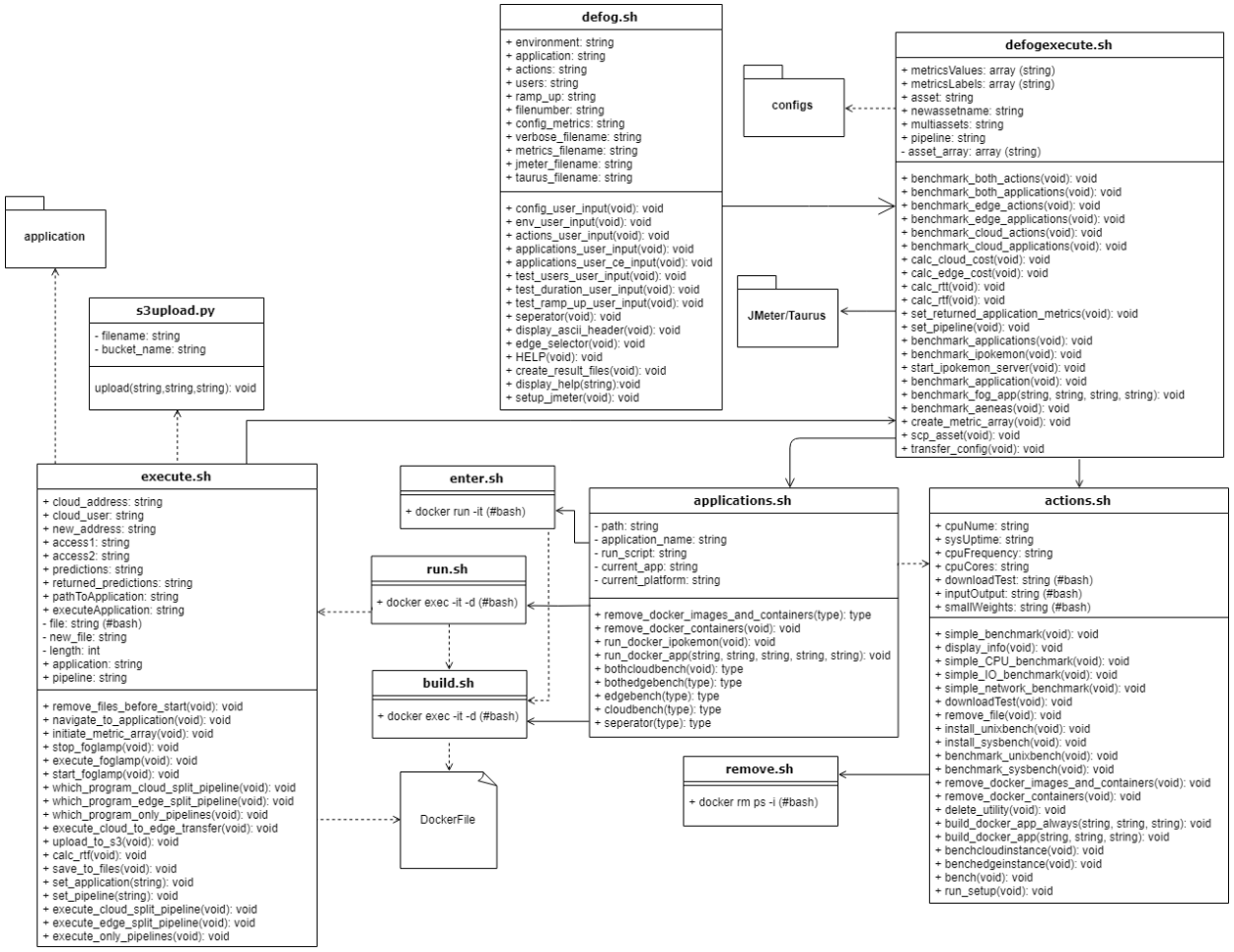

Figure 4: DeFog Class Diagram – DeFog System and Component Design

Utility Scripts: Found in the /*Scripts* folder. Template scripts found in the *Defog/utilityScripts folder.*

- **s3Upload.py & s3Download.py**: Transfers an asset to or from an S3 instance using Boto3. This accepts a string parameter which determines the input file and output file name.
- **sender.py & receiver.py**: Establishes a connection between a client and server (Cloud/Edge).
- **iPokemon.jmx**: Template jMeter file modified to accept user environment variables.

- **reporting.yaml**: Determines the Taurus bzt output results metric parameters and format.

Defog Scripts: Found in the */Defog* folder.

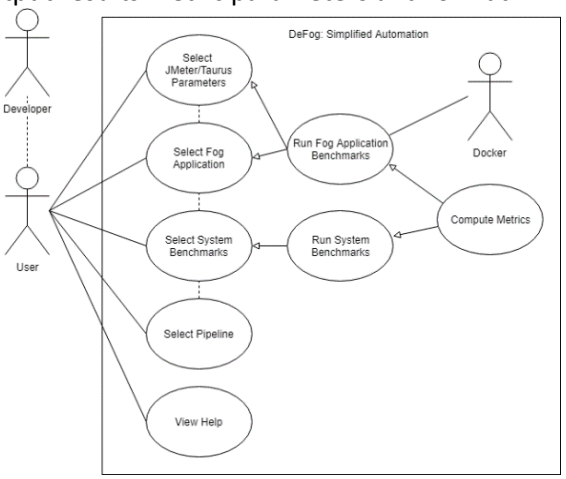

Figure 5: Generalised DeFog Benchmark UML Use Case

- **defog.sh**: Displays DeFog ascii header, help section, handles user input and error handling as shown in figure 5. The *config.sh* variables and *awskey.pem* location are sourced and passed to the *defogexecute.sh* script to run the DeFog pipelines on the Cloud and Edge.
- **defogexecute.sh**: Calculates various metrics, invokes the actions.sh and applications.sh scripts on the Cloud or Edge platform using Secure Shell, as shown in figure 6. Transfers the configuration files, assets and results data using Secure Copy Protocol and converts the metric data into appropriate file formats (.csv and .txt). Uses the sourced variables from parent script *defog.sh* to determine the application and system benchmarks and the environment variables for running JMeter and Taurus.
- **actions.sh**: Builds fog applications, runs system benchmarks, remove docker images and containers, as well as installing and executing TPI tools Sysbench and Unixbench. Secure Shell is used to invoke the docker *build.sh* script on the destination platform. The *action.sh* and *applications.sh* automated benchmarking process is also outlined in figure 6.
- **applications.sh:** Runs application containers using secure shell to invoke the *run.sh* or *runedge.sh* script by passing the application ID and pipeline ID as parameters. This script also updates the configuration folders permissions to prevent file mutation, i.e. *$ chmod 400*.

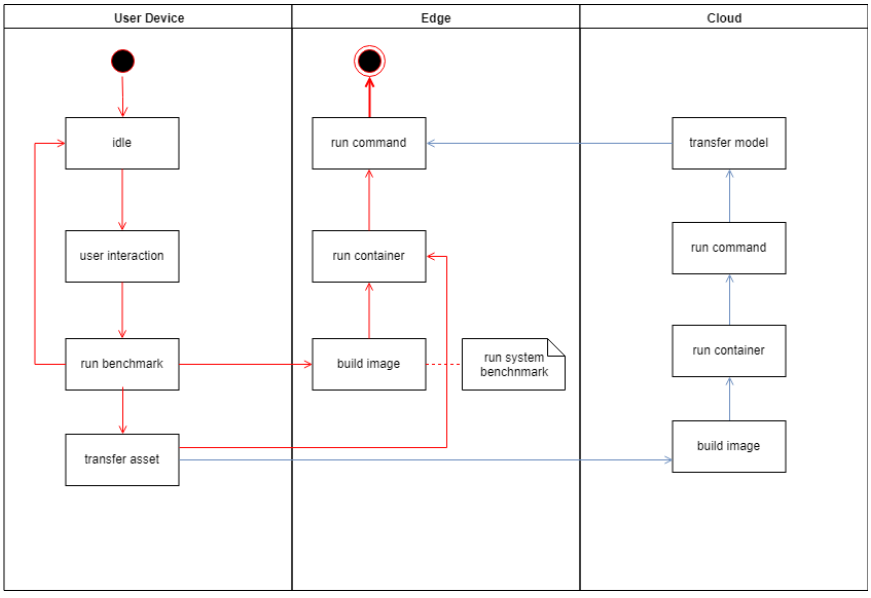

Figure 6: DeFog Benchmark Activity Diagram

In a typical workflow, for first time use the DeFog directories are created on the destination platform, sub trees are pulled from the main repository and application images are built using docker. The docker image is labelled, this is used to run a specific image for future runs. An asset is supplied to the Edge or Cloud by the user device and an application container is run in detached mode. The compute task is executed, and the container is removed upon completion. The Secure Shell connection ends and the metrics are calculated, parsed and saved on the user device.

2. Design

2.1 What is DeFog?

DeFog is a benchmarking tool that leverages docker to run system or containerised application benchmarks for fog applications deployed across the Edge and the Cloud. This allows developers of deep learning applications, IoT applications and online games to gain benefit from metrics relating to the relative performance of their fog applications leveraging the Edge. Developers can also make use of the TPI tools integrated within DeFog to capture additional comprehensive system benchmark metrics and generate synthetic workloads simulating user behaviour for concurrency load tests.

2.2 Dependencies & Pre-requisites:

Local Edge Device (Windows Laptop):

- **DeFog**: Pull or clone the latest version of DeFog from the main GitLab repository: *https://gitlab.eeecs.qub.ac.uk/40126401/csc4006-EdgeBenchmarking.git.* Ensure the line endings are set to Unix (LF) when pulled from GitLab, i.e. run *$ git config --global core.eol lf*.
- **Bash**: Ensure bash is installed and up to date on the user device.
- **bc**: Install the latest version of bc from: *https://www.gnu.org/software/bc/*.
- **JMeter & Taurus**: DeFog includes a template directory location for JMeter, use this or install JMeter (must have bin/jar file). Install Taurus from *http://gettaurus.org/docs/Installation/*, and update the $JAVA_HOME system variables to use a Java 8, i.e. *1.8.0_102 on the device*.
- **Configuration file**: Ensure the template configuration file provided in the DeFog folder is updated, and the config folder and file paths are updated at the top of *defog.sh script*.
- **Credentials**: Update or create the *.aws* folder, create a '*config'* file and populate it with the EC2 region. Add a '*credentials'* file to the*.aws folder* and populate this file with the IAM user_access, and secret_keys. Update the *.ssh* folders to use the generated rsa and rsa_pub ssh keys (using keygen). *For more detailed instructions see the Git README/help video.*

Cloud Platform (AWS):

- **IAM User**: Create an IAM user with the appropriate privileges.
- **EC2**: Create an EC2 instance, and open the available (TCP and UDP) ports.
- **S3**: Create an S3 bucket, and assign the name '*csc4006benchbucket*'.
- **Volumes**: Attach a volume with adequate space availability to the EC2 instance, i.e 16Gb.
- **Docker**: Install the latest version of Docker community edition.

Edge Platform (Odroid XU4 & Raspberry Pi 3):

- **Docker**: Install the latest version of Docker-ce, as well as bash, nano, time, grep & bc.
- **Credentials**: Update the */root/.aws* folder with the IAM users access/secret key credentials. Update the */root/.ssh* folder with an authorized_keys file, containing the generated public rsa ssh key from the user device, this will allow for password-less Secure Shell tunnelling.

2.3 How To Guide:

Run DeFog: *$ sh defog*

View Help and FAQ: *$ sh defog -?*

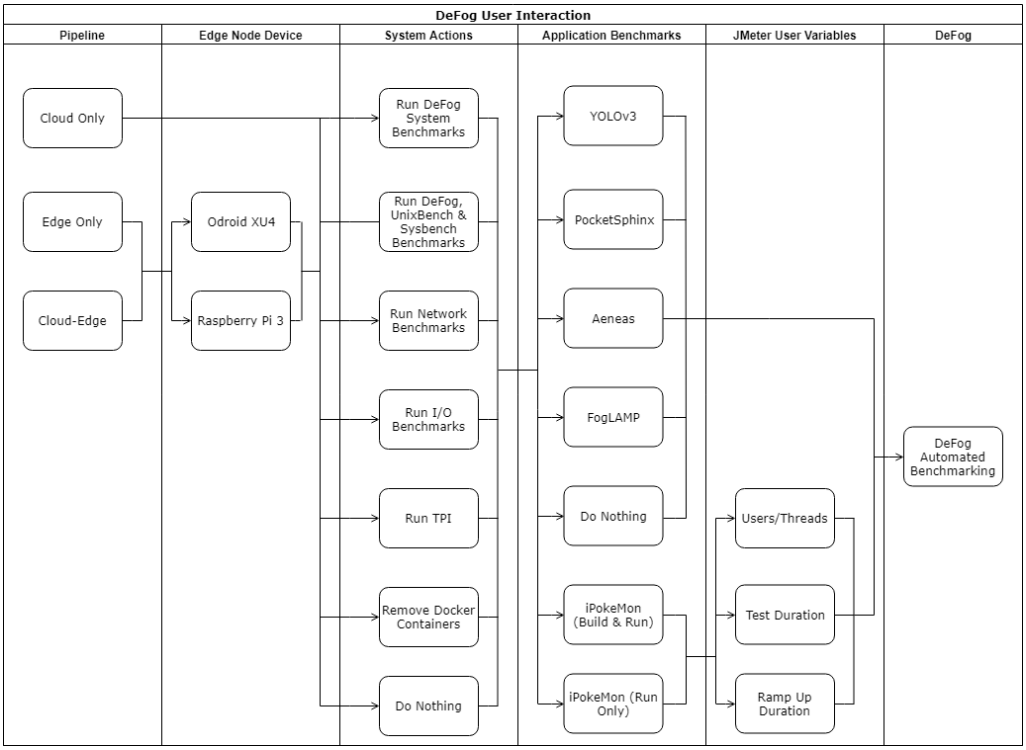

Figure 7: DeFog User Interactivity Flow, accessible by running $ sh run

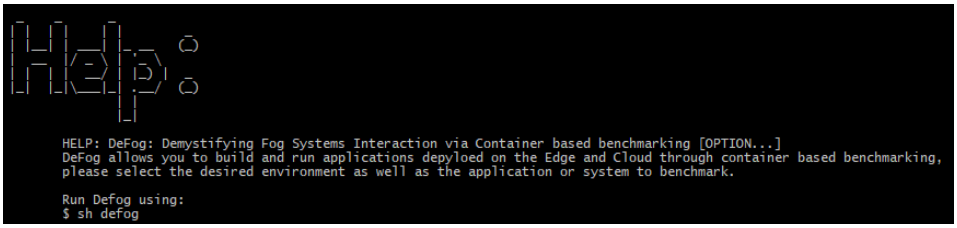

Figure 8: DeFog Help and FAQ Section, accessible by running $ sh defog -?

2.4 User Interface Design:

DeFog is an automated command line tool that accepts numeric user input before processing. Upon running Defog an ascii header is displayed to the user as shown in figure 9. This displays a short preamble to show the DeFog system is running. Alternatively, the user can view a help section intuitively via the command line, as shown in figure 8. A series of command line inputs are presented to the user, as detailed in figure 7. This incorporates error handling as discussed in section 2.6. The user interface was developed to ensure the system is intuitive, simple and easy to use for first time users. Upon finishing interacting with the system, the first of a series of autonomous benchmark runs begin. An example YOLOv3 Cloud-Edge use case is outlined in section 2.5.

2.5 Use Case:

The user starts DeFog as shown in figure 9, then selectes the Cloud-Edge pipeline as shown in figure 11. The Cloud Only action and platform benchmark options are also shown in figure 12. The compute task is performed in figure 14, and then the metrics are generated and returned in figure 13.

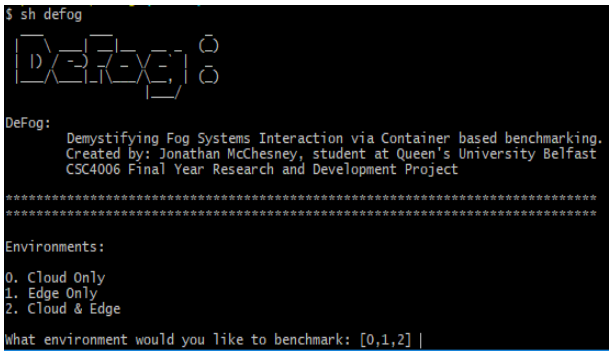

Figure 9: Preamble and Incorrect user input for application

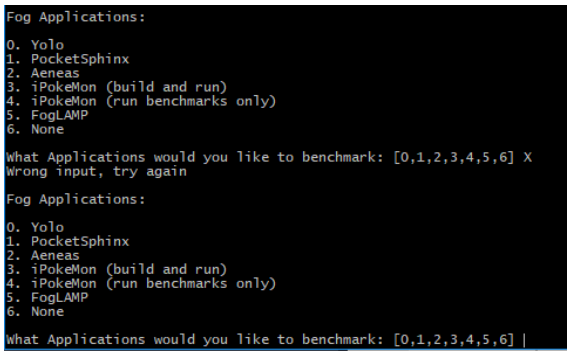

Figure 10: Incorrect user input for application selection

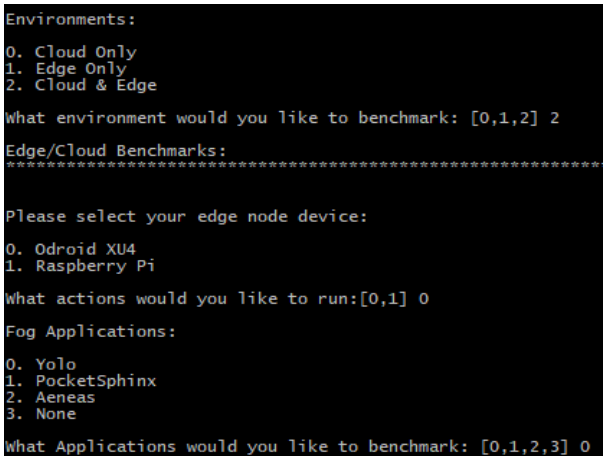

Figure 11: Cloud-Edge Pipeline Interaction

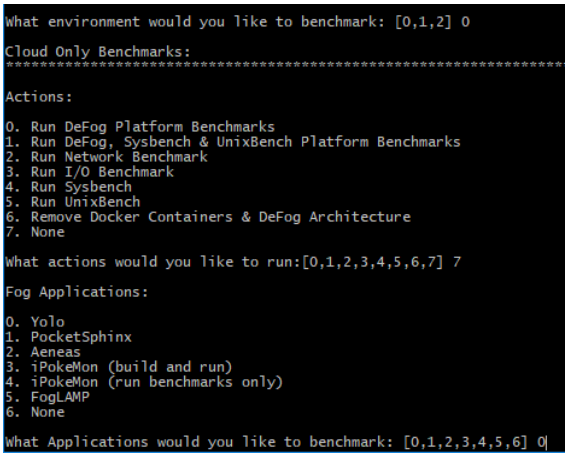

Figure 12: Alternative Cloud Only Pipeline Interaction

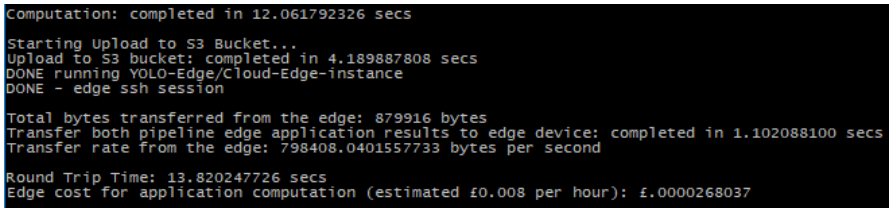

Figure 13: DeFog Benchmarking Compute Task Completion & Uploading Prediction Output to S3 Bucket

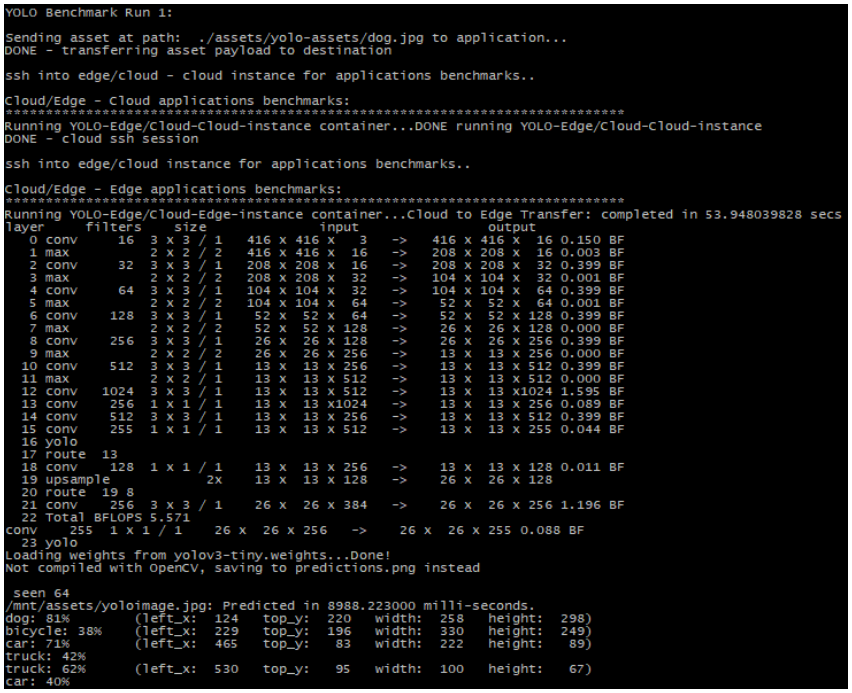

Figure 14: Yolo Compute Task – Transfer assets and weights, perform classification, report results and save prediction.

The remainder of the metric calculations, such as the computation cost on the Edge are performed by the *defog.sh* script on the user device. These results are then converted into the relevant result file types, e.g. csv and txt. Each application performs a unique compute task on the Cloud or Edge that generates data, such as: Yolo's classification, PocketSphinx's text to speech generation, Aeneas's forced alignment, FogLAMP's mocked sensor interactivity and iPokeMon's simulated user behaviour. As DeFog makes use of Shell scripting, any currently running command line processes can be cancelled using *ctrl-c*. DeFog automates the deploying, building and execution of several fog applications. This reduces the user interaction needed, and time spent building and running future application benchmarks, this is an important consideration for a fog benchmarking tool or method.

2.6 Error Handling:

The system handles incorrect or unexpected user input as shown in figure 10. When prompted to interact with the user interface, if the input is an out of range value or a non-numeric character, the error message "Wrong input, try again" is displayed and the selection is prompted to the user again. "Try except" blocks are also implemented in the *s3Download.py* component, using AWS *botocore*. If a connection to the Edge or Cloud cannot be established within 30 seconds, then the user will be immediately informed and the connection will be terminated, as shown in figure 15. If an issue is encountered when capturing a metric, then the current benchmark run will attempt to complete if the error is not fatal. A verbose message will be displayed to the user offering insight into the issue, and the metrics that were not generated will be displayed as *NA* within the results files.

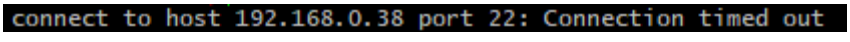

Figure 15: Error Handling – Edge node connection timeout user notification

3. Implementation

3.1 Dependencies, Languages, Packages & Tools Used:

Language	Description & Usage
Bash 4.4.19	The primary scripting language used throughout the system. Bash was chosen due its simplicity, readability, intuitiveness and potential to create powerful extensible code. Shell scripts are beneficial due the efficiency, portability and system interoperability offered, especially when used alongside a more elegant language such as Python. A shell scripts start-up time is 2.8 milliseconds compared to Python 2.7's start-up time of 11.1 milliseconds, this fast performance is important for benchmarking on the Edge, where the benchmarking process needs to be fast. This interoperability and flexibility will allow third party developers to quickly come up to speed with the codebase and allows for fast and efficient code deployment on a wide range of platforms and operating systems.
Python 2.7 & Python 3	The secondary scripting language used to create utility files to transfer and receive assets from an S3 bucket. Additionally, client/server scripts were created as an alternative to using the Secure Copy Protocol. While bash may be faster in terms of performance, the elegance of Python offers a greater range of functionality. By using Bash alongside Python faster performance is obtained from the bash shell scripts and a greater range of advanced features is available within the python components. This balance is used to great effect within the DeFog system.
C++	PocketSphinx was modified to store the decoded and converted output and save the text string to file. Modifications to 'continuous.c' were made using C++.
Java 8	This version of Java is required to run JMeter (jdk1.8.0_181), i.e. $JAVA_HOME.

Table 1: Languages Used During DeFog System Development

Dependency/Package/Library	Description & Usage
Git 2.17.1	Integrates git functionality, this is used to parse repository subtrees and pull the relevant applications and model assets.
Bc 1.07.1	An arbitrary precision numeric processing package used to perform complex computation, such as calculating the real time factor, computation cost and setting decimal places.
Python, python3	Python 2.7 and Python 3 language dependencies. Python3 is used to create all DeFog python scripts and leverages boto3 and Socket dependencies to transfer assets. Python is also a required dependency for Yolo detection and PocketSphinx.
python3-pip, python-pip	Package management tool for providing quick and easy installation of python dependencies, e.g. boto3 and chocolatey.
boto3 1.9.130	AWS SDK for python used to transfer files to an S3 bucket.
bashtest 0.0.7	Opensource bash assertion testing framework used to test configuration files and sourced parameters within DeFog.
ffmpeg 1.9.130	Catalog of software libraries for multimedia files, used within DeFog to calculate the length of an audio file, in seconds.
grep 3.3.0	Command line utility used by DeFog to search and parse text.

python-dev, python-dbus	Development dependencies required for deploying Aeneas and other fog applications, used to build python extensions.
time	Time utility dependency used to calculate the real, user and system (i.e. wall clock time) time for program execution.
autoconf 2.69.0, build-essential 8.3.1, libtool 2.7.0.2	Common dependencies for producing configure scripts, building programs and portable compiled libraries. Integrated within all Dockerfiles to compile each fog application.
inetutils-ping 2.1.9.4	Provides binaries for using cURL to ping FogLAMP services.
tee	Command to write to both the terminal (standard out) and file.
make, cmake, automake	Tools to automate and control the software build and compilation process. Used along with the modified Makefiles.
cURL 7.64.1	Opensource project providing CLI tools and libraries to transfer data using one of several supported protocols. Used within DeFog to simulate FogLAMP sensor API invocations.
sed, Jq 1.5.0	Command line tools to edit, parse and transform data, e.g. JSON, used in DeFog to format and 'beautify' curl output.
redis 3.0.0, redis-server 3.0.0	Open source in-memory data structure store used for cache data and message broking. Integrated on the Edge and Cloud platform services to work with the iPokemon server.
tcl 8.6.0	High-level, interpreted, dynamic programming language, used when compiling/integrating iPokeMon for scripted automation.
g++ 7.3.0, gcc 7.3.0	GNU Compiler Collection used to autonomously compile fog applications such as FogLAMP and PocketSphinx.
numpy 1.16, matplotlib 3.0.3	Packages used for scientific computing data manipulation, and data plotting for Python. Fog application dependency.
bison 3.0.0	GNU parser used in the compilation process for PocketSphinx.
swig 3.0.4	Simplified Wrapper and Interface Generator used to integrate Third Party Integation (TPI) software with PocketSphinx.
Chocolatey	Package manager for Windows, used to integrate with AWS.
libboost-system-dev, libsqlite3-dev, libboost-dev, libboost-thread-dev, libpq-dev, libpq-dev, libssl-dev, sqlite3	Fog application specific dependencies. These various packages and libraries relate to SQL database integration and library development dependencies. These are required for the compilation process and execution of the FogLAMP service.
uuid-dev, post-gresql, avahi-daemon, libpulse-dev, libz-dev, bluez 5.50	Further fog application specific dependencies and libraries. These are required for database and bluetooth integration as well as *'zero configuration networking implementation'*.

Table 2: Dependencies, Packages & Libraries Used During DeFog System Development

Tool/Service	Description & Usage
Apache JMeter 2.6	Used to simulate user behaviour as well as stress/load testing (JAVA 8).
Docker 18.06.1-ce	Used to build images and application containers. Provides isolated environments and consistent dependencies for all benchmarks.
Taurus bzt 7.8.1	Used to generate concurrency, latency and response metrics.
Excel	Used to format and present metric data (.csv) to the user.

AWS Management Terminal	Remotely control and maintain the EC2 and S3 instances. Allows for cost monitoring, mounting volumes and creating IAM users.
AWS-Command Line Interface	Remotely access AWS compute resources, installed on the edge nodes to allow seamless connection between the Cloud and Edge.
AWS Powershell for Windows	Remotely access AWS instances, installed on user device to connect to AWS using Secure Shell and execute script tasks.
Unixbench	Integrated within DeFog to provide additional platform metrics.
Sysbench	TPI within DeFog used to obtain further system test metrics.
AngryIPScanner	Determine the IP address of the edge nodes running on the network.
Putty	Used to create private and public (.pem) keys. Also allows for remotely interacting with the EC2 and edge node (alternatively use keygen).
Stress/Stress-ng	Command line tool to induce data and network stressor workloads.
Secure Copy Protocol (SCP)	A tool/protocol to securely transfer data between a local client and remote server, uses Secure Shell to ensure the authenticity and confidentially of the transferred data. SCP is used to transfer asset payloads and model data between the user device, Edge and Cloud.
Secure Shell (SSH)	Cryptographic network protocol for operating network remote services securely over a potentially unsecured network. This is used to leverage Cloud and Edge compute resources.
AWK	Command line utility to write statements that perform as processing programs. This is used to perform system benchmarks.

Table 3: Tools Used During DeFog System Development

3.2 Algorithms and components:

```
aws_cost_sec=$(bc <<< "scale=10;$awshrcost/$convert")
cost=$(bc <<< "$computetime*$ aws_cost_sec ")
```

Figure 16: Algorithm: Calculate Cloud Compute Cost

This algorithm (fig 16) calculates the cost of executing a compute task using the AWS pricing strategy ($0.08 per hour). Scale=10 is used to convert the output into 10 decimal places using the bc package.

```
$(ffprobe -v error -show_entries format=duration -of default=noprint_wrappers=1:nokey=1 $path)
rtf=$(bc <<< "scale=10;$computation/$length")
```

Figure 17: Algorithm: Calculate the Real Time Factor

This algorithm (fig 17) uses the ffmpeg package to determine the length of an audio file. Using the bc package and decimal scale of 10, this algorithm determines the real time factor of speech to text generation, where a scalar value of one implies the processing is done in real time.

```
$transfer=$(scp -v -o StrictHostKeyChecking=no -i $id $user@$address:$src $dest 2>&1 | grep "Transferred")
parsed_bytes=${ transfer//[!0-9\\ \\.]/}
```

Figure 18: Algorithm SCP data payload transfer parsing

This algorithm (fig 18) uses secure shell to determine the bytes up and bytes down for a payload transfer. Verbose mode is enabled, and an identity file is provided along with the automated credentials. Grep is then used to parse the returned object for "*Transferred*" data.

4. Function Declarations:

Function	Description (defogExecute.sh)

executebenchmarkhelp()	Invokes the HELP() function based on CLI user input.
benchmark_both_actions(), benchmark_edge_actions(), benchmark_cloud_actions()	Executes system and application container builds on either the Cloud or the Edge platform respectively for the Cloud-Edge Pipeline.
benchmark_both_applications(), benchmark_edge_applications(), benchmark_cloud_applications()	Invokes the relevant application pipeline. DeFog uses Secure Shell (SSH) to execute the application scripts on the Cloud and/or Edge by passing the application ID and pipeline ID. Bytes transferred per second and time metrics are also calculated. The metric data is updated and parsed into the results files, finally any temporary files are removed.
HELP()	When invoked displays a verbose and user-friendly guide on how to use DeFog, as well as a FAQ section.
calc_cloud_cost()	Determines the cost of running the application on the Cloud, using the AWS cost model, i.e $0.08 per hour.
calc_edge_cost()	This utility function estimates the cost of running the application on the Edge.
calc_rtt()	Calculates the latency times for the application benchmark. Metrics are calculated for the end to end latencies, return trip time, as well as communication and computation latency.
calc_rtf()	Calculates the real time factor (computation time/audio length) of speech to text detection (scalar value 1 = real time).
set_returned_application_ metrics()	Updates the application metric array with the returned values calculated during the Cloud and/or the Edge benchmarks.
benchmark_applications()	Outputs text to the verbose results (.txt) file detailing the current benchmark run, and then invokes the application specific benchmark function.
benchmark_ipokemon()	Invokes JMeter and Taurus to benchmark the iPokemon server autonomously, user variables are used as parameters based on user input sourced from the *defog.sh* script.
start_ipokemon_server()	Enters the iPokemon docker container to manually start the server. *Ctrl p and Ctrl q can be used to detach the container.*
benchmark_application()	Determines the pipeline and application to be benchmarked, invokes the connection and computes the relevant metrics.
benchmark_fog_app()	Accepts parameters specific to a fog application, iterates through the asset folder and transfers the necessary payload for each benchmark run. Invokes the *benchmark_application()* utility function to continue the benchmarking process.
benchmark_aeneas()	Sends two assets to the destination platform for the Cloud and Edge pipelines, transfers only the text file for Cloud-Edge.
create_metric_array()	Instantiate the metric catalogue array and header values.
scp_asset()	Uses Secure Copy Protocol (SCP) to securely transfer an asset payload to the destination pipeline service.
transfer_config()	Transfers the configuration file to the Edge or the Cloud depending on the pipeline ID passed in as a parameter.

seperator()	Prints a series of characters that form a separator line for user friendly verbose user output using *tee*.

Table 4: defogExecute.sh function declarations and implementation

Function	Description (defog.sh)
edgeaddress_selector()	Invoked for the Edge Only and Cloud-Edge combined pipelines. Determines the edge node address to use. A user friendly message is displayed to the user asserting the current options available for edge node selection: 0) Odroid XU4 or 1) Raspberry Pi 3.
actions_user_input()	Determines the actions and platform benchmarks to be performed. A user message is displayed informing the user of the choices available: 0) DeFog and TPI platform benchmarks, 1) Run network benchmark, 2) Run I/O benchmark, 3) Run Sysbench, 4) Run Unixbench, 5) Remove docker images, 6) Run DeFpg platform benchmarks 7) Continue. Error handling occurs on incorrect input.
env_user_input()	Determines the pipeline to deploy and run application benchmarks. A message is presented to the user detailing the available options: 0) Cloud Only 1) Edge Only 2) Cloud-Edge Combined.
applications_user_input()	Determines what application is to be benchmarked for the Cloud or Edge Only pipelines. The user is presented with a descriptive message displaying the applications available to be benchmarked: 0) Yolo 1) PocketSphinx 2) Aeneas 3) Build iPokeMon server and then run benchmarks 4) Run iPokeMon benchmarks without building 5) FogLAMP 6) Continue (No fog application benchmarks).
applications_cloud_edge_ user_input()	Determines what application is to be benchmarked for the Cloud-Edge Combined pipeline. The user is presented with a descriptive message displaying the applications available to be benchmarked for the selected pipeline: 0) Yolo 1) PocketSphinx 2) Aeneas 3) Continue (No fog application benchmarks are executed).
test_users_user_input()	Determines the number of concurrent users/threads to be run for JMeter and Taurus testing. The available options presented to user are as follows: 0) 1 user 1) 2 users 2) 5 users (default) 3) 10 users 4) 25 users 5) 50 users 6) 100 users 7) 250 users.
test_duration_user_input()	Determines the duration of a JMeter and Taurus simulated player behaviour test, in seconds. The user is presented with the following options: 0) 60 secs 1) 120 secs 2) 300 secs 3) 600 secs 4) 900 secs.
test_ramp_up_user_input()	Determines the ramp up duration of a JMeter and Taurus test, in seconds. 0) 0 secs 1) 10 secs 2) 30 secs 3) 60 secs 4) 120 secs.
display_ascii_header()	Outputs an ASCII header of the system name 'DeFog' as well as a short description of the DeFog System using *tee*.
setup_jmeter()	Executes the user input utility functions for initialising the JMeter (jmx file) user environment variables, e.g. number of threads, host.
create_result_files()	Determines the uniquely numbered result filenames to be saved for the current benchmark run.

Table 5: defog.sh function declarations and implementation

Function	Description (actions.sh)
simple_benchmark()	Runs a series of platform benchmarks: CPU benchmark, I/O benchmark and a network benchmark. The metrics and results from executing these benchmarks are outputted and saved to file by invoking *display*_info() function and using *tee*.
display_info()	Outputs to the console system benchmark results: run date, CPU model, number of cores, CPU frequency, system uptime, network download rate and I/O writing rate.
simple_CPU_benchmark()	Generates various CPU metrics: CPU model, system uptime, CPU frequency and the number of cores using *AWK* commands. The CPU is then benchmarked by creating a large file and capturing metrics on the time required to unzip the file using *$ tar*.
simple_IO_benchmark	Generates I/O metrics using *AWK* and command line utilities. A large weights/model file is created and tested. The I/O rate captured, outputted and then the file is removed.
simple_network_benchmark()	Benchmarks the download speed of a large weights/model file and records this captured network speed metric.
_downloadTest()	Invokes an extensive network download benchmark test. The download rate and total time taken are recorded.
download_benchmark()	Invokes the download test benchmark function and passes the weights URL/filename as a parameter.
downloads_benchmark()	Invokes the *download_benchmark()* function. Passes in a weight or model URL/filename as a parameter to download using *wget*.
instantiate_file()	Captures metrics on the time taken to create a large weights file.
remove_file()	Removes the large weights/model file from the Cloud or Edge.
install_unixbench()	Determines if Unixbench is already present, if not then the tool is downloaded using *wget* and unzipped.
install_sysbench()	Determines if Sysbench is installed, if not then use apt-get.
benchmark_unixbench()	Executes TPI Unixbench suite on the Cloud or Edge.
benchmark_sysbench()	Executes TPI Sysbench suite on the Cloud or Edge.
remove_docker_images_ and_containers()	Stops and removes all docker images and containers from the Cloud or Edge.
remove_docker_containers()	Removes all docker containers from the Cloud or Edge.
delete_utility()	Removes all DeFog folders and assets from the Cloud or Edge.
build_docker_app(), build_docker_app_always()	Determines if the application architecture exists on the platform before building. This accepts the build path, build name and application name as parameters. The application is pulled from GitLab (or GitHub) and the image is built using *build.sh*.
benchcloudinstance(), benchedgeinstance()	Determines the application to run on the Cloud or Edge for the Cloud-Edge pipeline. Then invokes the relevant build functions.
bench()	Determines the fog application to be benchmarked on the Cloud or Edge for the Cloud or Edge Only pipelines. System benchmarks and application builds are then invoked.
run_setup()	Creates the directory architecture on the destination platform.

Table 6: actions.sh function declarations and implementation

Function	Description (applications.sh)
run_docker_ipokemon()	Invokes the *enter.sh* script to manually start the iPokemon Cloud server, i.e. n*avigate to the Application/iPokeMon-CloudServer folder, input $. runCloud.sh. Use $ Ctrl P and Ctrl Q to detach the docker container.* The benchmark process continues after *exiting*.
run_docker_app()	Accepts the following string parameters: application path, application name, the name of the run script, application ID and the current pipeline ID, used to invoke the *run.sh* script.
bothcloudbench(), bothedgebench()	Invoked for the Cloud-Edge combined pipeline, this is invoked on the Cloud and Edge. User input determines the fog application ID to be passed as parameters to the *run_docker_app()* function.
edgebench(), cloudbench()	Invoked for the Edge Only and Cloud Only pipelines. Determines which fog application is to be run on the Edge or Cloud.

Table 7: applications.sh function declarations and implementation

Function	Description (execute.sh)
remove_files_before_start()	Ensures any pre-existing result files are removed.
navigate_to_application()	Navigates to the specific application folder on the destination platform service.
initiate_metric_array()	Initialises the local metric values array.
stop_foglamp()	Stops the FogLAMP root server.
execute_foglamp()	Executes a cURL command on the local server.
start_foglamp()	Starts the FogLAMP root server.
which_program_cloud_split_pipeline(), which_program_edge_split_pipeline(), which_program_only_pipelines()	Determines the local variable assignments determined by the application ID and pipeline ID parameters for the current compute task and benchmark run.
execute_cloud_to_edge_transfer()	Transfers the model asset from the Cloud to the Edge.
execute_program_only()	Executes an application specific computational task, this is run for the Cloud Only or Edge Only pipelines.
upload_to_s3()	Uploads the predicted data asset to an S3 bucket.
calc_rtf()	Calculates the real time factor of speech to text using *ffmpeg* and *grep* to determine the audio file length and *bc is used to* to compute the value (*10 decimal places*).
save_to_files()	Outputs the metric data to file.
set_application()	Determines the application from the application ID parameter passed by *run.sh* to be benchmarked.
set_pipeline()	Determines the pipeline to be run from the pipeline ID parameter passed by the *run.sh/runEdge.sh* script.
execute_cloud_split_pipeline(), execute_edge_split_pipeline(), execute_only_pipelines()	Invokes the necessary functions to remove files before execution, transfers the model data from the Cloud, captures metrics, executes a compute task, uploads data to an S3 bucket and outputs the metrics to file.

Table 8: execute.sh function declarations and implementation

5. Infrastructure:

Amazon Web Services is the Cloud platform leveraged by DeFog.

5.1 IAM user: AWS Identity and Access Management (IAM) is a web service that allows for secure remote access control over AWS resources, this controls who is authenticated and authorized to use Cloud resources. The IAM users created for DeFog follow the recommended best practices outlined by AWS and OWASP when assigned the relevant permissions by adding the necessary security groups. It is required to add the IAM users AWS_SECRET_ACCESS_KEY_ID and AWS_SECRET_KEY_ID to the edge device and edge nodes *.aws* credentials folder.

5.2 Credentials: Credentials are important for ensuring that DeFog has the correct privileges to perform the necessary benchmarking tasks on the Cloud and Edge. The aws .pem ssh access key is generated using putty and should replace the empty template file located in the */configs* folder. The config file needs updated to reflect this. The Git README and help video provides more information. For the user device, navigate to or create the .ssh folder, this is typically found in the user folder, e.g. C://Users/defoguser/.ssh. Generate the necessary ssh keys (*$ ssh-keygen -t rsa*) this will create the private and public rsa keys within the local .ssh folder. Remotely access the edge node and create a root .ssh folder, e.g. ~/.ssh. Create an *authorized*_keys file and append the public key (id_rsa_pub). Navigate to the .aws folder, this is normally found in the root directory, e.g. ~/.aws. Create a credentials files containing the IAM users aws_access_key_id and secret_aws_access_key_id on both the user device and Edge. A config file is also created to store the instance region.

5.3 Volumes: Volumes are attached to the EC2 instance to allocate memory capacity. DeFog makes use of a 32Gb volume mounted onto a EC2 instance located in Dublin, Ireland.

5.4 Elastic Compute Cloud: DeFog uses an EC2 t2.micro instance, in the eu-west-1c availability zone, located in Dublin, Ireland. This instance uses an ubuntu-bionic-18.04-amd64 image and security groups are created to open any necessary ports (UDP and TCP).

5.5 Simple Secure Storage: An S3 bucket "csc4006benchbucket" is setup in the EU-Ireland region.

5.6 Docker: DeFog makes considerable use of Docker images and containers for executing containerised fog application benchmarks. Docker-ce (18.06.1-ce) is installed on the platforms.

5.7 Redis: Redis is used during the compilation, building and execution of several fog applications such as iPokeMon. The redis-cli is used to manually interact with or stop with the redis-server.

6. Testing & Evaluation

6.1 Manual Acceptance Testing: Extensive manual acceptance testing was performed throughout

the development of the DeFog system and the underlying infrastructure. During an acceptance testing session, the system was run several times with the intent of trying to break the underlying mechanics. For example, attempting to run DeFog simultaneously on different terminals, using unexpected characters such as: 边缘 as user input and attempting to benchmark an application without any assets within the asset folder. These tests were repeated before an iteration was pushed to Git. A subset of these acceptance test use cases is present within the DeFog test folder.

6.2 Assertion Testing: Assertion tests were created using '*bashtest*' and '*grep*'. Testing followed the red green refactor test driven development technique, i.e. create a failing test as seen in figure 19, develop the minimal amount of code to pass the test, as shown in figure 20, and then refactor the implementation. These tests assert whether a certain criterion is met, if not the test will fail.

6.3 Infrastructure Testing: Frequent load and performance testing on the EC2 instance and edge nodes occurred throughout the development of DeFog using JMeter. This was to ensure correct execution when the system was put under stress. Wireshark was used to ensure a consistent throughput of work was maintained and packet loss was monitored. In addition to this the AWS-cli was used to test EC2 and S3 endpoint connection integration, this was important to ensuring the instances were available and set up correctly. AWS CloudWatch was used on the AWS management dashboard to send alerts if the instances went down or if the available space of the assigned volume fell below a threshold. Grafana was also used during development to monitor docker statistics.

6.4 Evaluation: The system was consistently evaluated using manual testing use cases to ensure the system was performing as intended. This evaluation was important to ensuring all functional requirements were met. For a requirement or user feature to be considered '*done done*' all acceptance criteria outlined was required to be met. In addition to this several use cases were created to handle potential misuse or unexpected use of the system, such as if an EC2 instance became unavailable during the benchmark process. The time taken to run components of the DeFog system as well as application benchmarks were recorded and if significant increases in this time were perceived after code changes were made, it implied potentially a more efficient solution or code refactor was needed. E.g. Cloud-Edge (Odroid) Yolo initially took 22.40321 seconds round trip time, this was reduced to 14.23878 by evaluating the code design and iterating on its weaknesses, such as removing inefficient waiting mechanisms and loops.

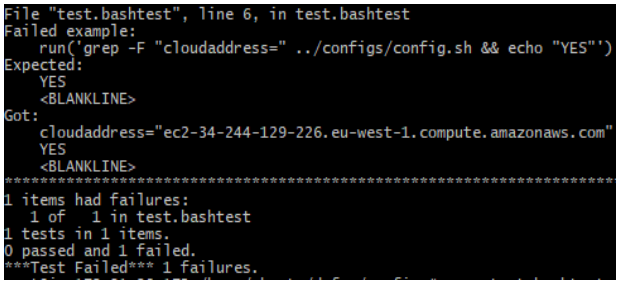

Figure 19: Failed config assertion test using bashtest

Figure 20: Passing config assertion test using bashtest