



Operations Manual

# National Infrastructure for Secure Federated Learning

---

Author: **Chris Davey**

Organisation:

**Queensland  
Foundation Ltd**

**Cyber**

**Infrastructure**

01/07/2025

Version 1.0

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION****Revision History**

<b>Date</b>	<b>Comment</b>	<b>Author</b>	<b>Version</b>
dd-mm-yyyy	comment	Author Name	1.0

# Contents

1	Project Description.....	5
1.1	Overview.....	5
1.1.1	Goals.....	5
1.1.2	Scope.....	5
2	System Model.....	6
2.1	Network Deployment Topology.....	10
3	Deployment Model.....	14
3.1	Environments.....	14
3.1.1	Test Environment.....	14
3.1.2	User Acceptance Test (UAT) Environment.....	15
3.2	Deployed Software Components.....	15
3.2.1	Recommended Operating System.....	15
3.2.2	Federated Learning Server.....	15
3.2.3	Federated Learning Client.....	18
3.2.4	Virtual Private Network Components.....	20
3.2.5	Monitoring and Operational Support Components.....	21
3.3	Network.....	22
3.3.1	Network Protocols/Ports.....	23
3.4	Security.....	26
4	Procedures.....	28
4.1	Provisioning.....	28
4.1.1	Directory Structure.....	28
4.1.2	Flower.Server.....	29
4.1.3	Flower.Client.....	58
4.2	Deployment Validation.....	66
4.3	Data Management Procedures.....	66
4.3.1	Database Backup Procedures.....	66
4.3.2	Database Restore Procedures.....	67
5	Monitoring.....	67
5.1	Grafana and components installation.....	67
5.1.1	GPU Monitoring Components.....	68
5.2	Configuration.....	70
5.2.1	Configuration Customisation – Grafana.....	71
5.2.2	Configuration Customisation – Prometheus node exporter.....	71

## DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION

5.2.3	Configuration Customisation – Prometheus Server.....	72
5.3	Alerting.....	73
5.3.1	Common alerting metrics – CPU Load.....	74
5.3.2	Common alerting metrics – Memory.....	74
5.3.3	Common alerting metrics – Disk volume space.....	74
5.3.4	Common alerting metrics – Disk volume space.....	74
5.3.5	GPU alerting metrics – Temperature.....	75
5.3.6	GPU alerting metrics – Memory Utilisation.....	75
5.4	Logging Details.....	76
5.4.1	Systemd journal.....	77
5.4.2	Logrotate.....	77
5.5	Diagnostics.....	78
5.5.1	Grafana Dashboards.....	78
5.5.2	Systemd status.....	79
5.5.3	Journalctl Logs.....	80
5.5.4	File based logs.....	80
5.5.5	Flwr command line.....	80
5.6	References.....	81
5.7	Appendix 1. Supernode Docker Build Files.....	81
5.7.1	Docker build file for GPU enabled cuda image.....	81
5.7.2	Docker build file for CPU only.....	82

# 1 Project Description

## 1.1 Overview

The National Infrastructure for Federated Learning project is described in the QCIF document “Flow Federated Learning Framework for use in the NINA Project” [1]. In short, the purpose of the Federated Learning (FL) project is to provide infrastructure to support the secure access to medical data to train machine learning models by transmitting only model parameters instead of data. It enables external medical organisations to support trusted research partnerships by allowing machine learning models to be trained on data within a secure partition. Researchers do not have direct access to the medical data sets but instead are able to develop their model architecture based on representative and/or simulated data sets. After they have developed their chosen modelling approach, they can organise their training scripts to support the execution within the flower FL framework. The flower FL framework consists of a central “aggregation” server and multiple training client “nodes”. The central aggregation server (a superlink) is located within the Nectar infrastructure. Whereas client “nodes” (known as a supernode) is located on the client premises. A secure trust exists between the Nectar hosted superlink and the client hosted supernode. The end research submits their model training script to the superlink inside of the Nectar cluster. The superlink distributes the model training scripts to registered supernodes, executes the training process with access to the client premise data sets and aggregates the model training weights. The model weights are aggregated at the central superlink server and the final modelling result is saved on the superlink server within the nectar cluster. It is important to note that at no time during the FL training process is the on-premise data set transferred between the two network locations, all access to data resides within the on-premise supernode. Only the network parameters are transferred, over a secured and trusted network connection.

### 1.1.1 Goals

The project goals aim to support secure and trusted access to sensitive medical data without leaking the data records to the researcher or transmitting data outside of the on-premise network partition. This enables institutions to participate in impactful and critical medical research projects without exposing their own sensitive data to potential leakage or unauthorised use.

### 1.1.2 Scope

The purpose of this document is to provide an overview of the system model and deployed system components. The document will describe the concrete operational details of software deployment, operational mechanisms for security as well as support for business continuity through monitoring, and management of the deployed system. This document will describe manual provisioning steps. These steps are recorded with the intention to support development of automation pipelines where practical. It is also intended to capture key operational support procedures.

## 2 System Model

The high-level requirements [1] document multiple deployment scenarios. This document will describe the scenario “On-premise deployment for federated learning” as a deployment method since the deployment requirements can be applied to other scenarios. This scenario is a hybrid cloud

## DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION

deployment where the aggregation server (the superlink) is hosted in the Nectar cloud network. The training nodes (the supernodes) are hosted in the on-premise servers and interconnectivity between the Nectar cloud is supported over virtual private network (VPN) enabling the host organization to participate within a dedicated network partition for the federated learning.

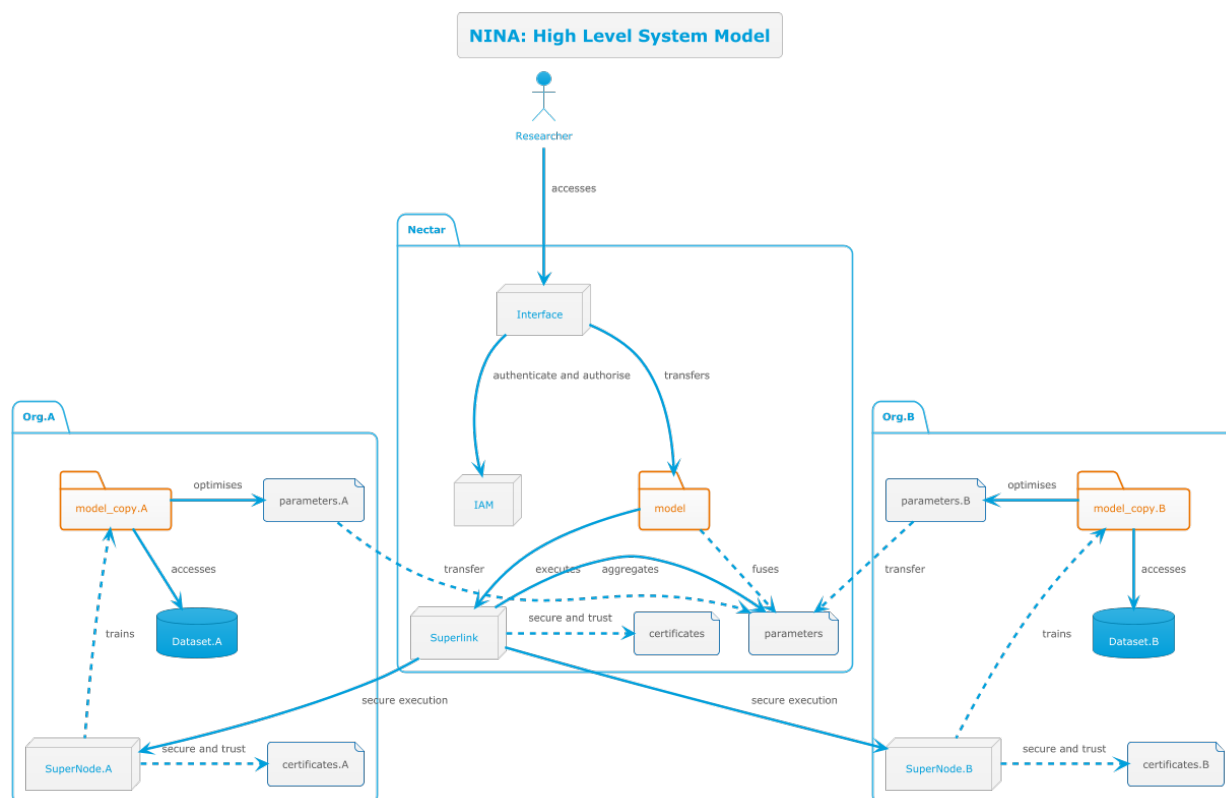


Figure 1 A high level overview of the main system objects deployed to support federated learning.

Figure 1 depicts the categories of users, organisations, software components and assets that are deployed in the system to support FL.

Table 1 Description of system entities within the federated learning framework.

System Entity	Description	Role
Researcher	A researcher is a trusted user who has access to the interface layer of the system. The researcher can access this layer over a secure https connection or alternately over ssh. The researcher accesses the system for the purpose of modelling, analysis and discovery.	Modeler
Nectar	Nectar research cloud infrastructure supported by QCIF and ARDC. This infrastructure hosts the coordinating software components.  The Nectar cluster does not have access to the sensitive data.	Infrastructure

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Interface	<p>The interface to the system is the only point through which the researcher can transfer their project files, and submit training jobs to the appropriate superlink node.</p> <p>It is possible to access the system either through a JupyterHub service or alternately through a terminal access via SSH. Since terminal access is less secure, JupyterHub can be sandboxed with limited privileges, access via this latter method may be preferable.</p>	External Access Point
IAM	<p>Identity and Access Management (IAM) Services. These services are hosted in the Nectar infrastructure and provide a secure mechanism of authentication and authorization for registered researchers to access the system.</p> <p>It is also possible to partition access to the system using realms for multiple tenancies.</p>	Authentication and Authorization
model	<p>The “model” represents a specific project structure for use with the FL framework. This consists of the following units:</p> <ul style="list-style-type: none"> <li>- Server script: run at the superlink during the aggregation process.</li> <li>- Client script: performs training on the supernode on-premise and defines data partitioning schemes.</li> <li>- Tasks: these provide extension functions for auditing training or saving checkpoints of models.</li> </ul> <p>Depending on the complexity of the method or whether horizontal or vertical partitioning is used, the client and server may both define “model” architectures, auditing and checkpoint.</p> <p>The set of scripts must follow the prescribed patterns defined by the flower FL framework.</p>	Model and Training Definition
Superlink	<p>The superlink component acts as the coordinator during the training process. It distributes “model” scripts, coordinates supernodes and aggregates results.</p> <p>The superlink does not have access to</p>	Coordination, Aggregation

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

	sensitive data.	
certificates	Transport layer security between the superlink and participating supernodes is enabled through provisioning of certificates. Mutual trusts between superlink and supernode are also established by configuring the public keys of participating supernodes in the superlink process.	Service Authentication and Network Security.
parameters	The parameters represent the model parameters that are transmitted over the secure connection between the superlink and supernode. The parameters at the superlink result from an aggregation strategy in horizontal FL or alternately may result from model fusion in vertical FL.	Model Assets
Org.A, Org.B	Org.A and Org.B represent organisational boundaries as well as separate network segments. Connectivity between the Nectar and organization networks is supported over VPN to form a hybrid cloud network.	Organisational Boundary
SuperNode.A, SuperNode.B	The supernode is deployed inside of the on-premise network segment and connected to the nectar network over a secure VPN. Each supernode is responsible for execution of training in the on-premise network segment. Supernodes are authorised to connect to the superlink using SSH certificates.	Execution
parameters.A, parameters.B	The parameters within the on-premise network are extracted from the model and securely transmitted to the supernode where they are combined using an aggregation strategy. Different instances of the client training script executing on different networks may consume different populations of the training data, therefore parameters.A will differ from parameters.B.	Model Assets
model_copy.A, model_copy.B	To train the model a copy of the training script assets are distributed by the superlink to the participating supernodes. Each supernode has a copy of these assets which are then executed during	Model Assets



**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

	the training procedure.	
Dataset.A, Dataset.B	<p>Each organisation curates and manages their own data sets. To support interoperability participating organisations may adopt a common standard such as the Observational Medical Outcomes Partnership (OMOP) common data model (CDM). However this will also require the the training client scripts must also know how to interpret this source data. The data structure and attributes are decided and socialised prior to researchers prior to use of the FL system. The research develops scripts to consume the agreed data structure and may test their model locally using synthetic data or non-sensitive data in the same structure.</p> <p>On-premise datasets are secured due to the network partitioning and firewall rules and can only be accessed by the client script when executing on the supernode. Datasets are not transmitted over the network and do not leave the organisational boundary.</p>	Secure Data
certificates.A, certificates.B	Supernodes are registered with the Superlink using SSL certificates to establish a secure mutual trust.	Service Authentication and Network Security.

## 2.1 Network Deployment Topology

The network deployment approach is modeled around separate network partitions within the Nectar cluster and participating organisation. There are two primary network segments in the FL cluster connected over a Virtual Private Network (VPN) between the trusted participating organisations to create a hybrid cloud network between Nectar and the participating organisations. A high-level description of the hybrid network topology is shown in Figure 2. The pattern supports one or more participating organisations (Org.N) which may be linked via a virtual network partition over VPN and connection to the FL cluster. Independent projects/contracts may be supported through isolation between network partitions.

Note that this same topology is also operable within a cloud only environment contained entirely within the Nectar cloud. When deployed in this manner the server Nectar partition is logically separated from the Org.N network partition using separate network segments and does not require a VPN as it is possible to route traffic between dedicated network partitions within the cloud environment. Each partition maintains isolation and traffic between segments is filtered by firewall rules.

## DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION

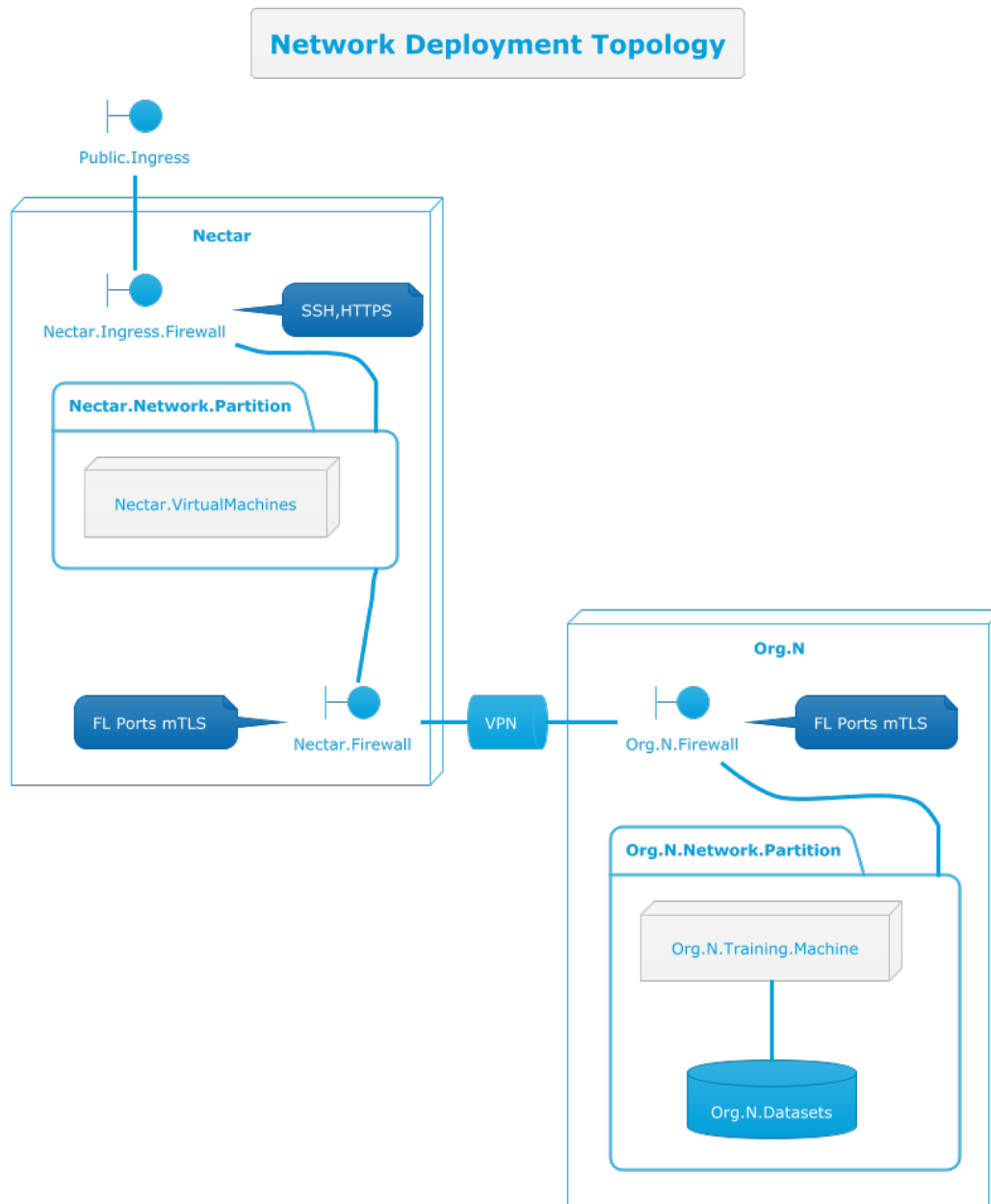


Figure 2 High level view of network topology forming a virtual cloud between dedicated Nectar network segments and participating Organisation network segments.

Table 2 enumerates each of the elements shown in Figure 2.

Table 2 Description of elements shown in the network topology diagram. Organisational responsibility is also indicated.

Object	Description	Responsibility
Public.Ingress	The system is accessible over the internet using either SSH or HTTPS. There are two types of roles which make use of the public ingress.	QCIF

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

	<p>Researchers – access the system to carry out modelling in the secure environment.</p> <p>Operations Team – access the system for support of business continuity.</p>	
Nectar	The cloud infrastructure is provided by the Nectar ARDC cloud environment.	QCIF
Nectar.Ingress.Firewall	Traffic is filtered between the public ingress and internal network partition allowing only supported protocols of HTTPS and SSH from the public facing side of the network.	QCIF
Nectar.Network.Partition	The nectar network partition represents the network segment dedicated to the FL infrastructure.	QCIF
Nectar.VirtualMachines	The nectar virtual machines contain the user interface, FL server and operational software components. Each component may be deployed on a dedicated VM or optionally on a single large VM.	QCIF
Nectar.Firewall	<p>The nectar firewall component filters traffic from the Nectar network to the participating organisations. Only those ports required for the FL communication are allowed, and traffic is encrypted over mTLS (provisioned at the software component). Network protocols related only VPN connection will be permitted.</p> <p>The nectar firewall is defined within the Nectar cloud security groups.</p>	QCIF
VPN	The VPN represents the virtual private network between organization vpn client and nectar cloud vpn server. The VPN server will be hosted inside of the Nectar Network partition.	QCIF

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

	When the system is deployed entirely within a cloud environment, the VPN component is not required. Network partitions can be logically separated by network address translation (NAT) and firewall rules.	
Org.N	Org.N represents the separate boundary managed by partner organisations.	Org
Org.N.Firewall	The organisation firewall is provisioned to allow only the network traffic needed for FL components and VPN connection. The organisation firewall is managed by the organization ICT operations team.	Org
Org.N.Network.Partition	The organisation network partition represents a logical network segment that is dedicated to connecting to the hybrid network to participate in the FL business case. It may be configured as an external facing DMZ from the perspective of the organisation.	Org
Org.N.Training.Machine	The organization training machine is a dedicated server(s) that has a hardware profile suitable for executing deep learning training. It participates as a “supernode” within the FL cluster and hosts the client training process. It enables the machine learning process to access sensitive data during the training procedure.	Org
Org.N.Datasets	The data assets that the organisation provides to support the FL learning procedure. These assets are not directly accessible outside of the organization network partition and are read only during the FL training procedure from the training	Org

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

	machine.	
--	----------	--

## 3 Deployment Model

### 3.1 Environments

There are several key environments supported by this project. Each environment is to be identified and recorded in this document.

#### 3.1.1 Test Environment

This environment exists within the Nectar cluster and provides two separate network segments for the purpose of testing.

##### 3.1.1.1 Environment Details (TBD)

Details are TBD

Asset	Details	Notes
Server network segment		
Jupyter Hub VM		
Flower Superlink VM		
VPN Server VM		
Org network segment		
Org Supernode VM		

#### 3.1.2 User Acceptance Test (UAT) Environment.

This environment exists within the Nectar cluster and provides an environment for end user testing and training with external organisations.

##### 3.1.2.1 Environment Details (TBD)

Details are TBD

Asset	Details	Notes
Server network segment		
Jupyter Hub VM		
Flower Superlink VM		
VPN Server VM		
Org network segment		
Org Supernode VM		

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

--	--	--

## 3.2 Deployed Software Components

Software components are grouped into several functions.

- Federated Learning Server
- Federated Learning Client
- Virtual Private Network Components
- Monitoring and Operational Support Components

### 3.2.1 Recommended Operating System

The recommended operating system is Ubuntu 24.04 LTS.

This document will describe installation procedures for this operating system.

### 3.2.2 Federated Learning Server

The FL server (Figure 3) will consist of the components listed in Table 3. The FL Server environment will need GPU hardware acceleration to support vertical mode FL.

## DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION

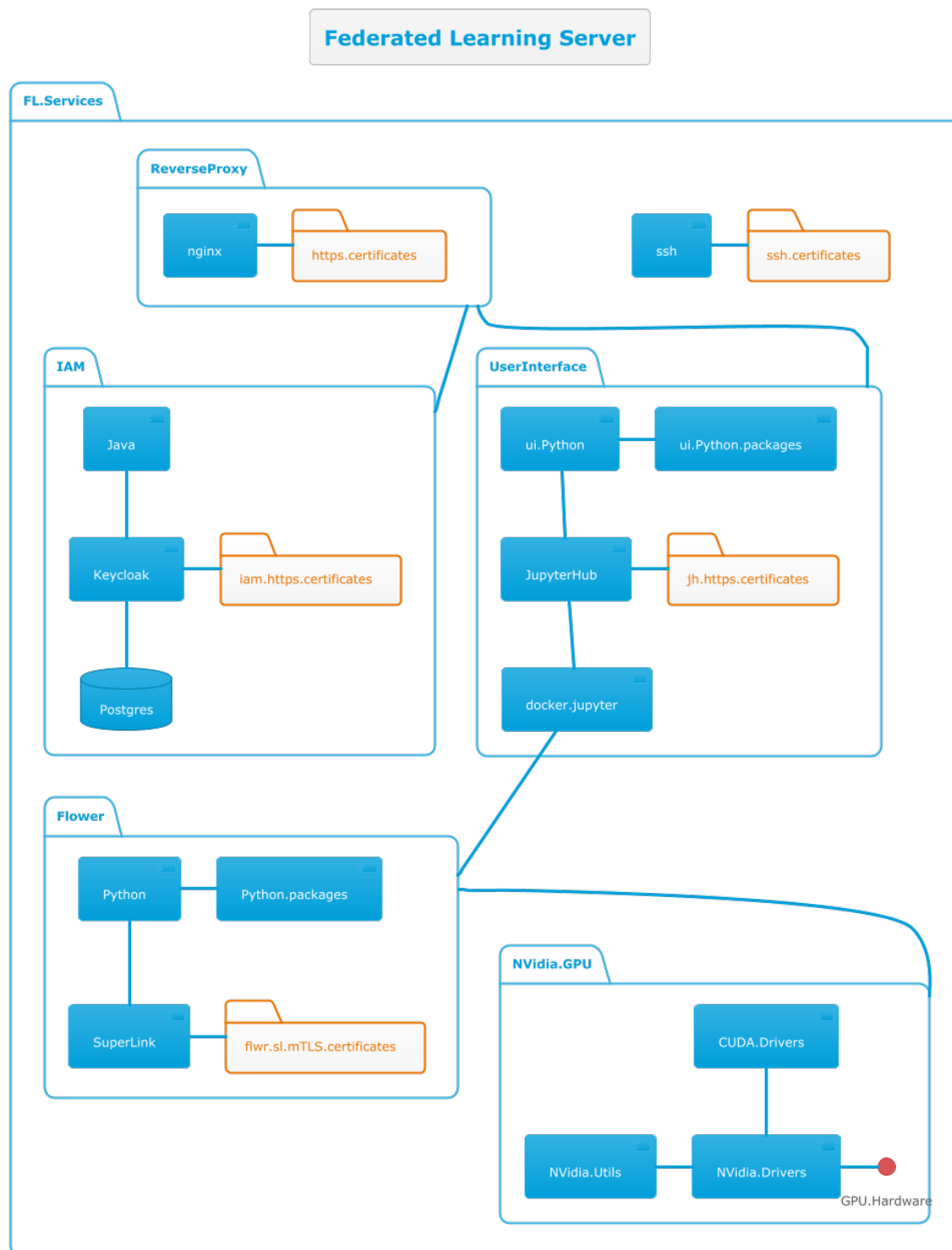


Figure 3 The key application components delivering FL functionality that are managed within the FL server environment.

Table 3 List of components deployed in the FL Server environment.

Package	Component	Description
Global	ssh	The sshd service supporting

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

		terminal access to the platform.
Global	Ssh.certificates	SSH certificates and associated management processes.
ReverseProxy	nginx	The reverse http proxy is implemented by the nginx server and associated configuration. Nginx server will act as the end-point in-front of the public network.
ReverseProxy	https.certificates	The https certificates for the reverse proxy need to be provisioned to support the public domain. Management process around these certificates will be required.
IAM	Java	The LTS JVM runtime of the openjdk 21.0.7 LTS
IAM	Keycloak	The keycloak server providing OIDC support for the platform.
IAM	Iam.https.certificates	HTTPS certificates for the internal keycloak hostname within the network segment and associated management process.
IAM	Postgres	The Postgres (17) database server hosting the keycloak application database. Only accessible from the keycloak host.
UserInterface	ui.Python	The python 3.12 virtual environment for the user interface layer.
UserInterface	Ui.Python.packages	Python dependencies for the ui layer.
UserInterface	JupyterHub	JupyterHub provisioned and configured within the UI layer.
UserInterface	Jh.https.certificates	HTTPS certificates and management processes for the jupyterhub UI instance.
UserInterface	Docker.jupyter	Docker configuration, startup hooks and images to support non-privileged jupyter instances for non-system user accounts within the UI layer.



**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Flower	Python	The python 3.12 virtual environment for the FL server layer.
Flower	Python.packages	Python dependencies for the FL server layer.
Flower	SuperLink	The configured superlink node within the FL layer.
Flower	Flwr.sl.mTLS.certificates	mTLS certificates and management process for secure network transport and public keys for authentication of trusted organization clients.
Nvidia.GPU	Nvidia.Drivers	Nvidia drivers (575) include kernel modules for the target environment.
Nvidia.GPU	Nvidia.Utills	Nvidia utils (575) include monitoring tools for the nvidia hardware.
Nvidia.GPU	CUDA.Drivers	Cuda 12 drivers provide the interface and JIT compilation between deep learning libraries and underlying Nvidia drivers.

**3.2.3 Federated Learning Client**

The Federated Learning Client will require GPU hardware acceleration to perform training in both horizontal and vertical learning modes.

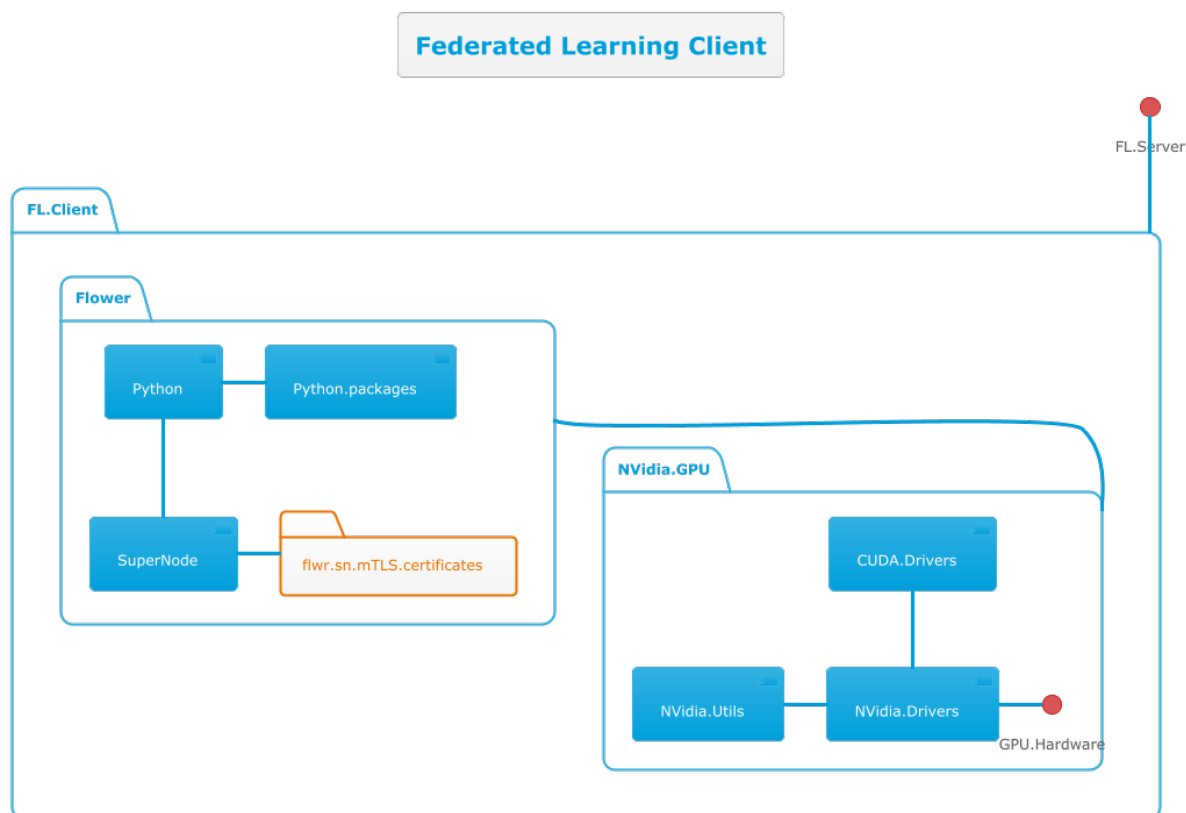
**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Figure 4 The software components supporting FL application functionality in the client environment.

Table 4 FL Client software components.

Package	Component	Description
Flower	Python	Python 3.12 virtual environment for the FL learning system .
Flower	Python.packages	Python package dependencies supporting the flower and deep learning libraries.
Flower	SuperNode	The supernode component configured for interconnection with the superlink.
Flower	Flwr.sn.mTLS.certificates	TLS and authentication certificates for mutual trust with the Superlink and associated management processes.
Nvidia.GPU	Nvidia.Drivers	Nvidia drivers (575) include kernel modules for the target environment.
Nvidia.GPU	Nvidia.Utils	Nvidia utils (575) include monitoring tools for the nvidia

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

		hardware.
Nvidia.GPU	CUDA.Drivers	Cuda 12 drivers provide the interface and JIT compilation between deep learning libraries and underlying Nvidia drivers.

### 3.2.4 Virtual Private Network Components

The virtual private network is provided by the openvpn software package. The openvpn server is configured with a local certificate authority which is used to issue certificates for each client connection. The client certificates must be provisioned to the VPN client host and the client is configured to connect automatically. The VPN enables multiple geographically separate clients to connect to the Nectar cloud environment but does not permit interconnection between separate client organisations. Instead, each client is isolated and communications are limited to VPN and superlink/supernode communications. Figure 5 indicates the vpn server and client and dependency on provisioning for ssl certificates.

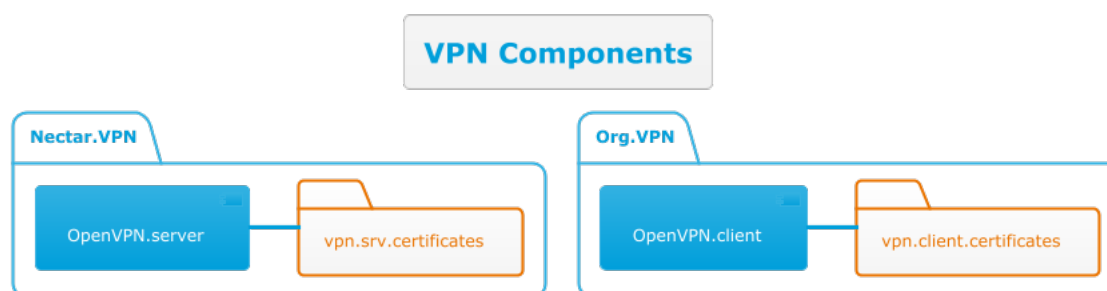


Figure 5 The components provisioned to support VPN between Nectar and organisational networks.

## DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION

### 3.2.5 Monitoring and Operational Support Components

Operational Monitoring and Alerting is necessary for all deployed production systems. The purpose of operational monitoring alerting is to enable the operational support team to proactively manage the platform. The key functions of the operations software components are to provide:

- Operations monitoring/observability
  - o status of system and components and resource use.
- Alerting – notifications for operational support.
- System Management eg: Log rotation.

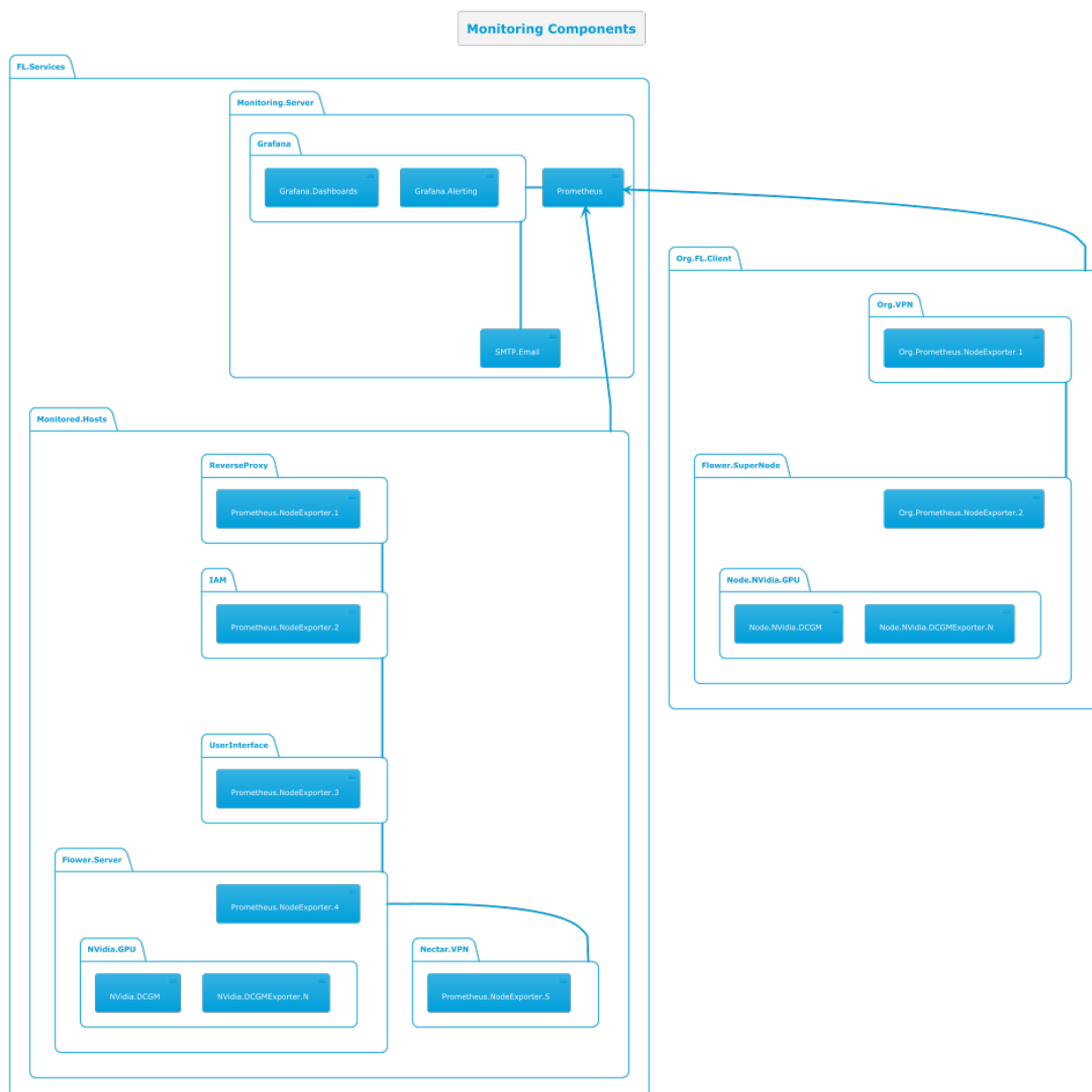


Figure 6 Overview of monitoring components deployed in system. Monitoring and alerting is implemented through the Grafana, Prometheus and Prometheus Node Exporter components.

Figure 6 provides a high-level diagram of the monitoring and alerting system components. The functionality is provided by the open source Grafana and Prometheus systems. These systems are composed of:

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

- The storage layer
  - o Implemented by the Prometheus time series storage.
- The instrumentation layer (for data collection)
  - o Implemented by the Prometheus Node Exporter and Nvidia DCGM Exporter components installed on each host in the system.
- The Alerting and Display Layer
  - o implemented by the Grafana application which consumes data from the Prometheus time series store and enables the operations team to configure rules for alerts which can be issued via Email SMTP or to other tools such as Slack. It also enables the team to define dashboards which can be used to monitor system metrics that can assist diagnostics of the runtime status of the system.

As shown in Figure 6 a Monitoring Server is deployed within the Nectar cloud and hosts the Grafana instance and Prometheus storage. Access to the monitoring system can be controlled via IAM login, and the monitoring system is configured to issue alerts via an email SMTP server.

Each host that is to be monitored is required to have a Prometheus Node Exporter service installed which collects host level metrics for resource use and service status. This includes those client organisation hosts that are connected via the VPN gateway. The Prometheus server is configured to discover node exporters available in the network. For those hosts which provide GPU hardware for deep learning the Nvidia Data Center GPU Management software must be installed and configured and the Nvidia dcgm-exporter is deployed to enable the retrieval of runtime GPU resource metrics. This applies both within the server network as well as for those clients connected over VPN.

### 3.3 Network

The network is separated into four segments indicated in Figure 7. The first is the segment is the client organisation network. This is a geographically separate network partition which is interconnected to the Nectar cluster over VPN. The second network partition is the public facing network of the Nectar cluster. This network partition enables access to the publicly exposed ports for the vpn connection and the https ports of the nginx reverse proxy server. The third network partition is the internal network which hosts the supporting services including the superlink. The fourth network partition is the network segment allocated to the vpn connections. This segment can connect to the internal network through ip forwarding via the iptables configuration of the vpn server.

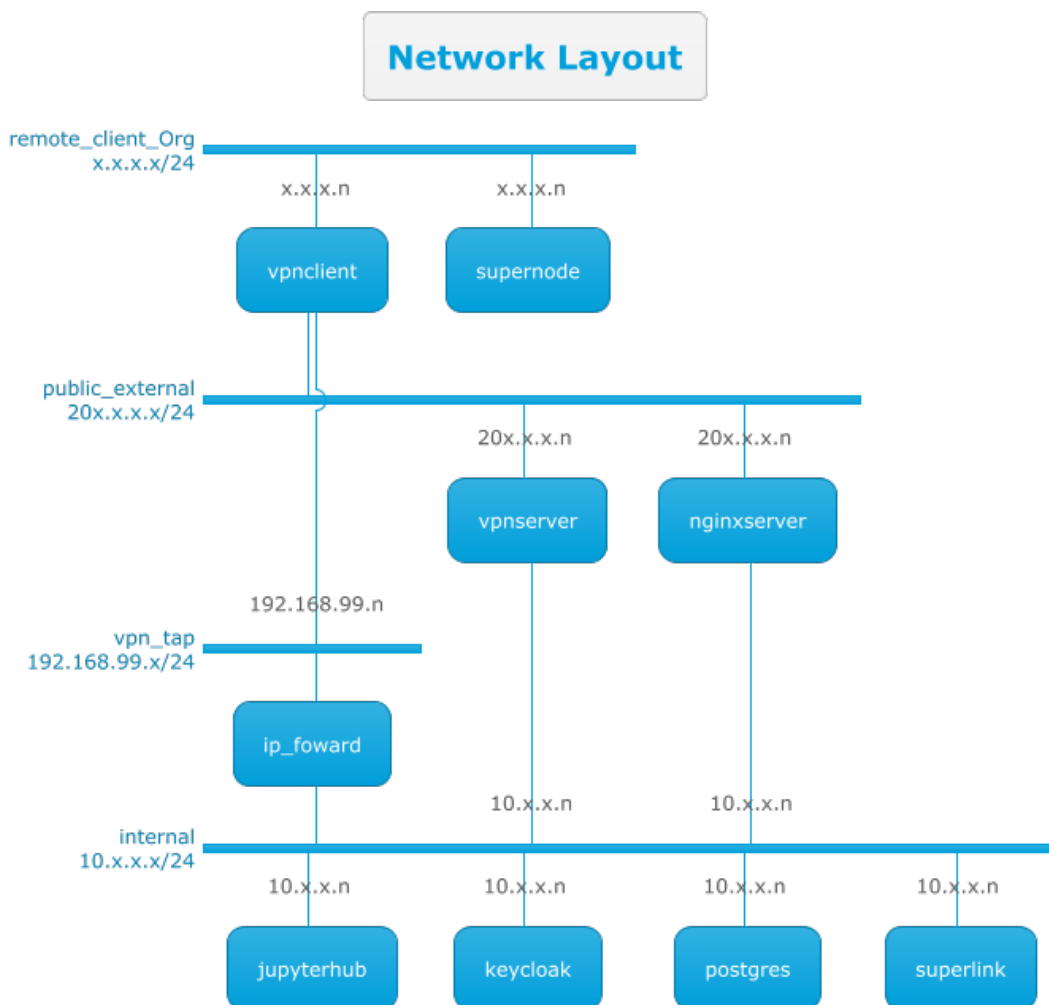
**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Figure 7 High level model of network topology. Client organisations are physically separated, the client network is connected over VPN to the Nectar cloud server environment enabling the client supernode to connect with the server superlink. The Nectar environment contains a public facing external network segment, an internal network segment and leverages ip forwarding in a vpn network segment for connected clients. Only those servers necessary to be exposed on the public interface are available to the external network segment.

### 3.3.1 Network Protocols/Ports

List of network protocols and ports required for system operation. Those ports indicated on the public network are to be permitted through the public firewall. Those ports on the Internal network are required for operations within the internal network segments including the hybrid network segments formed by connection of client organisations over VPN.

Package	Network	Component	TCP Ports	UDP Ports	Protocol
Global	Public	SSH	22		ssh
ReverseProxy	Public	Nginx	443		https

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

UserInterface	Internal	JupyterHub	443		https
IAM	Internal	Keycloak	8443		https
IAM	Internal	Postgres	5432		postgres
Flower	Internal	SuperLink	9091		Flwr ServerAppIO API
Flower	Internal	SuperLink	9092		Flwr Fleet API
Flower	Internal	SuperLink	9093		Flwr Deployment Engine
Flower	Internal	SuperNode	9094 – 9099		Flwr gRPC ClientAppIO (may also extend to other port ranges if configured)
Nectar.VPN	Public	OpenVPN	443	1194	TCP 443 is used when traversing firewalls that don't permit UDP.  UDP 1194 is the default transit.
Monitoring	Internal	Prometheus Node Exporter	9900		Configured port for node exporter for interconnect between hosts and over client organisation VPN.
Monitoring	Internal	Grafana	3000		Dashboard web interface for internal network.
Monitoring	Internal	Prometheus	9090, 9091, 9093		Prometheus time series storage. System should be installed on a separate server to Superlink as ports conflict.

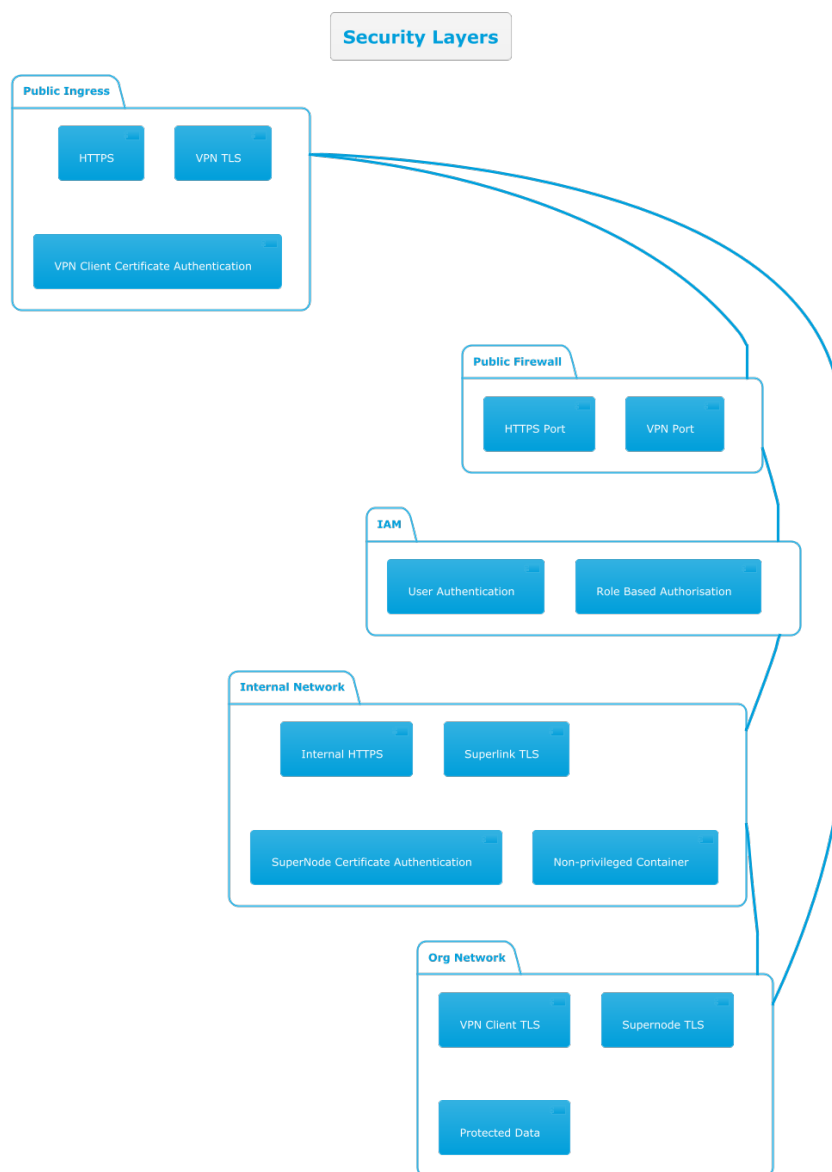
**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**



**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

### 3.4 Security

The system security approach follows the principal of least trust. The security is implemented in several layers shown in Figure 8.



*Figure 8 Security is implemented in multiple layers. Network segments are separated by firewalls and SSL certificates are applied for certificate-based authentication between client components. Transport layer security is also applied to data in transit at the transport level for web applications and both at the application protocol level and transport level for superlink to supernode communications. Organisation data does not leave the organisation network boundary and is accessed only during training on the supernode inside the organisation network partition. Model parameters are transmitted over the secure link during the training process.*

Network traffic for Public Ingress is limited by the Public Firewall to only those protocols (https, SSL, VPN) which are permitted. On the HTTPS based browser access the IAM layer is applied to authenticate and authorise users with role-based access to be able to run applications via the Jupyterhub UI. Services on the Internal Network support HTTPS encryption and communication between superlink and registered supernodes is encrypted with TLS as well as requiring mutual certificates for authentication of authorised client organisations. All traffic between the client

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

organisations Org Network is encrypted over the VPN connection and the client connection is authorised using mutual trust certificates issued by the Internal Network CA. When the researcher accesses the system through JupyterHub they do not have direct access to the host shell environment but instead access a non-privileged docker container. The researcher therefore has limited permissions when executing application code on the system. In addition the Superlink and Supernode services are also executing as non-privileged accounts without direct shell access. The data that is provided in the client organization Org Network, is protected firstly because it is not transmitted over the network, and secondly because access to the data (other than client administration access) can only occur when executing an training procedure within the Supernode.

## 4 Procedures

### 4.1 Provisioning

#### 4.1.1 Directory Structure

The cluster will follow a configuration and component directory convention. The following directory structure will apply across the cluster.

*Table 5 Configuration directories.*

Node Type	Path	Description
All	/etc/ssl/certs	Location of trusted CA keys
All	/etc/ssl/private	Conventional location for private ssl keys. This is also used in the case of the signing CA server.
Flower Superlink and Supernode	/etc/flwr	The root configuration directory for the Flower FL learning system.
Flower Superlink	/etc/flwr/superlink	Base configuration path for the superlink node.
Flower Superlink	/etc/flwr/superlink/certs	Location of certificates and keys used by the superlink node.
Flower Supernode	/etc/flwr/supernode	Base configuration path for the supernode.
Flower Supernode	/etc/flwr/supernode/certs	Location of certificates and keys used by the supernode.
Server Jupyterhub	/etc/jupyterhub/	Base configuration path for jupyterhub.
Server Jupyterhub	/etc/jupyterhub/config/	Configuration directory storing jupyter_config.py file for the environment.
Server Reverse Proxy	/etc/nginx	Base path to nginx configuration.
Server Reverse Proxy	/etc/nginx/conf.d/	Include directory for virtual hosts configured for reverse proxy.
Server and Client VPN	/etc/openvpn/	The base openvpn directory for configurations.
Superlink and SuperNode	/opt/python/flower_env	Python virtual environment for

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

		FL framework.
JupyterHub	/opt/python/jupyterhub	Python virtual environment for jupyterhub.

*Table 6 Component Directories*

Node Type	Path	Description
SuperLink and SuperNode	/opt/python/flower_env	Location of trusted CA keys
Jupyterhub	/opt/python/jupyter_env	Conventional location for private ssl keys. This is also used in the case of the signing CA server.
SuperLink	/opt/flwr/superlink	Superlink systemd and runtime files.
SuperNode	/opt/flwr/supernode	Supernode systemd and runtime files.
Jupyterhub	/opt/jupyterhub	Jupyterhub systemd and runtime files.
Keycloak IAM	/opt/keycloak	Keycloak systemd and runtime files.
Certificate Authority	/usr/share/easy-rsa	The easyrsa tools for management of PKI.

**4.1.2 Flower.Server**

This section describes the manual deployment steps for the server on virtual machine instance in the Nectar cloud. Each component may be deployed to a dedicated virtual machine or run on a single large virtual machine image. The intention of this section is to describe all manual deployment steps. These may be incorporated into an automated process at a later stage (such as via combination of Nectar orchestration, Ansible or Jenkins).

**4.1.2.1 Local Service Users**

A set of local service users will be created with no shell access. They will all be members of the system group “fl\_system”.

As this is a local system group it needs to be executed on all servers, including the supernode.

```
sudo groupadd fl_system
# add the current admin user to the fl_system group
sudo usermod -a -G fl_system $USER
# add service users
sudo useradd -r -s /bin/false keycloak
sudo useradd -r -s /bin/false superlink
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
sudo useradd -r -s /bin/false supernode
sudo useradd -r -s /bin/false jupyterhub
# add the service users to the fl_system group.
sudo usermod -a -G fl_system keycloak
sudo usermod -a -G fl_system superlink
sudo usermod -a -G fl_system supernode
sudo usermod -a -G fl_system jupyterhub
```

**4.1.2.2 Local Certificate Authority (CA)**

The local certificate authority is used to generate SSL certificates for mutual Transport Layer Security authentication between organisations SuperNodes and SuperLinks. It can also be used to generate internal HTTPS certificates that are not exposed to the external network.

The local CA is used with those services which are not publicly accessible. Since certbot requires a publicly accessible domain to generate certificates. Instead the local CA can be used to manage certificates for the internal network segments.

The local certificate can be used to sign locally generated certificates that are used inside the server network segment and that of the collaborating organisations. It is not a publicly facing certificate authority.

The simplest mechanism to create a local CA is to use the easyrsa set of utilities (<https://easy-rsa.readthedocs.io/en/latest/>).

```
sudo apt install easy-rsa
```

This installs the easyrsa tools to **/usr/share/easy-rsa**. To use these tools we first need to configure an environment variables file so that it is customised for your organisation.

```
cd /usr/share/easy-rsa
sudo cp vars.example vars
# open the file for editing.
sudo vi vars
```

Customise the environment variables for your organisation.

```
set_var EASYRSA_PKI "$PWD/pki"

set_var EASYRSA_REQ_COUNTRY "AU"
set_var EASYRSA_REQ_PROVINCE "Queensland"
set_var EASYRSA_REQ_CITY "Brisbane"
set_var EASYRSA_REQ_ORG "QCIF"
set_var EASYRSA_REQ_EMAIL "my.email@qcif.edu.au"
set_var EASYRSA_REQ_OU "DAS"

set_var EASYRSA_KEY_SIZE 2048
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Then save the file and initialise and configure the CA. Note the process will generate a ca.key will ask for a password (save the password in a password manager for reuse).

```
sudo ./easysrsa init-pki  
sudo ./easysrsa build-ca
```

The following components will require internal certificates issued by the local CA.

- openvpn server
- superlink server
- keycloak server
- jupyterhub server

The procedure for generating each of the certificates is performed on the local CA server and follows the pattern:

```
sudo cd /usr/share/easy-rsa  
sudo ./easysrsa --nopass build-server-full servername
```

Where the servername is the hostname of the server requiring the certificate. The resulting certificate and keys are contained in the paths:

- /usr/share/easy-rsa/pki/issued/servername.crt
- /usr/share/easy-rsa/pki/private/servername.key

The certificate and key along with the /usr/share/easy-rsa/pki/ca.crt will need to be distributed to the hosts forming the part of the system. Additionally supernode hosts will require a copy of the ca.crt certificate.

To trust the signing authority, the **/usr/share/easy-rsa/pki/ca.crt** file needs to be distributed to all participating hosts and a pem certificate will need to be generated on each host.

This can be achieved by using the ca-certificates tool on each host.

```
sudo apt install -y ca-certificates  
sudo cp ca.crt /usr/local/share/ca-certificates  
sudo update-ca-certificates
```

To update the ca.crt each time it expires, the process can be repeated (this can be automated using the above tools).

The ca.crt file will be referenced on both the superlink server and the supernode instances. These will be described in the sections associated with each step.

#### 4.1.2.3 VPN Deployment

The Virtual Private Network capability is provided by the openvpn software package.

The security for the server is realised by ssl certificates. For this purpose the VPN host server depends on the local signing authority to issue server and client certificates. Client certificates are used to provide authentication for connection of vpn clients. These certificates can be generated in the same

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

manner described for flower server and clients. Once configured VPN server and client will be managed as systemd services.

#### 4.1.2.3.1 Management of keys via easyrsa

The procedure for preparing the openvpn server is to define a path where the certificates will be stored for use by the openvpn server. The installation of the easyrsa package contains a template directory structure and variables file.

When generating server keys the EASYRSA\_REQ\_EMAIL is defined, however, will be set to empty when generating client keys.

```
sudo ./easyrsa --nopass build-server-full openvpnserver
sudo ./easyrsa gen-dh
sudo openvpn --genkey secret pki/private/ta.key
```

The following keys will be used by the openvpn server. If the openvpn server is on a host other than the local CA, these certificates will need to be distributed to the openvpn server.

- /usr/share/easy-rsa/pki/ca.crt
- /usr/share/easy-rsa/pki/dh.pem
- /usr/share/easy-rsa/pki/issued/openvpnserver.crt
- /usr/share/easy-rsa/pki/private/openvpnserver.key
- /usr/share/easy-rsa/pki/private/ta.key

Each client system that connects to the vpn server must have its own client certificate. The certificate contains the client identity, and the vpn server will assign ip addresses based on that client identity.

To generate a client certificate first open the vars file and set the EASYRSA\_REQ\_EMAIL key to empty, then use the following command.

```
sudo ./easyrsa --nopass build-client myclientname
```

This will generate the client certificate and key in the following paths:

- /usr/share/easy-rsa/pki/issued/myclientname.crt
- /usr/share/easy-rsa/pki/private/myclientname.key

To be able to connect from each client machine the following keys need to be copied for each machine (myclientname will be different for each client machine):

- /etc/openvpn/clientcerts/ca.crt
- /etc/openvpn/clientcerts/issued/myclientname.crt
- /etc/openvpn/clientcerts/private/myclientname.key

#### 4.1.2.4 Openvpn server configuration.

The openvpn server will be configured in a non-bridged mode and iptables will be used to route traffic from the openvpn tap network device to the internal network. In a multiple tenancy deployment, where there is more than one superlink, a separate openvpn server should be deployed per tenant, to maintain separation between tenancies.

In this example the following is assumed.

- There are two network interfaces configured for two separate networks on the openvpn server eth0 and eth1.
- One network is a publicly facing IP address accessible on the internet and filtered by a firewall configuration.

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

- The second network is an internal network address. This will be the bind address for internal services and will be reachable by the client only after connection to the VPN.

The server will create a separate VPN tap interface with a private subnet 192.168.99.0/24 network.

The server configuration file is created in /etc/openvpn/server/server.conf

```
port 1194
dev tap

server 192.168.99.0 255.255.255.0

tls-auth /usr/share/easy-rsa/pki/private/ta.key 0
ca /usr/share/easy-rsa/pki/ca.crt
cert /usr/share/easy-rsa/pki/issued/openvpnserver.crt
key /usr/share/easy-rsa/pki/private/openvpnserver.key
dh /usr/share/easy-rsa/pki/dh.pem

persist-key
persist-tun
keepalive 10 60

# eg: 10.255.135.45 internal network
push "route 10.255.0.0 255.255.0.0"

user nobody
group nogroup # use "group nogroup" on some distros

daemon
log-append /var/log/openvpn/openvpn.log
```

The configuration will use udp for the vpn connection (alternate methods can use tcp if udp is not permitted by routers).

The configuration can be tested using the command:



**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
sudo openvpn --config /etc/openvpn/server/server.conf
```

Checking the server logs will confirm if the server started, as well as checking the netstat output and grepping for port 1194.

**4.1.2.4.1 Iptables IP Masquerading.**

The sysctl ip\_forwarding needs to be enabled on the openvpn server by editing the /etc/sysctl.conf file and uncommenting the ip\_forward configuration.

```
net.ipv4.ip_forward=1
```

Network forwarding is then configured between the source ip address range for the subnet 192.168.99.0/24 and the target internal ethernet device (in this case eth1). Be sure to check which network device should be configured for forwarding.

```
sudo iptables -t nat -A POSTROUTING -s 192.168.99.0/24 -o eth1  
-j MASQUERADE
```

Additional commands for iptables such as listing and deleting rules are shown below:

```
# list commands on the nat configuration.  
sudo iptables --list-rules -t nat  
# list numbered commands on the nat configuration.  
sudo iptables -L -t nat --line-numbers  
# delete a rule by its line number  
sudo iptables -t nat -D POSTROUTING 2
```

**4.1.2.4.2 Configure the openvpn server as a systemd service.**

As the openvpn server is configured and is able to start successfully, it can be configured to run as a systemd service and controlled with systemctl.

```
sudo systemctl -f enable openvpn-server@server.service
```

This will enable the openvpn with configuration in /etc/openvpn/server/server.conf to start automatically at boot time. Other useful systemctl commands are listed below:

```
sudo systemctl start openvpn-server@server.service  
sudo systemctl status openvpn-server@server.service  
sudo systemctl stop openvpn-server@server.service
```

Reboot the server and check the openvpn logs, netstat and systemctl status after reboot to confirm the openvpn service has started automatically.

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION****4.1.2.4.3 Nectar security groups.**

On the openvpn server ensure there is a security group configured to permit udp port 1194 on the public IP of the vm.

Displaying 3 items

<input type="checkbox"/>	Direction	Ether Type	IP Protocol	Port Range	Remote IP Prefix	Remote Security Group	Description	Actions
<input type="checkbox"/>	Egress	IPv4	Any	Any	0.0.0.0/0	-	-	Delete Rule
<input type="checkbox"/>	Egress	IPv6	Any	Any	::/0	-	-	Delete Rule
<input type="checkbox"/>	Ingress	IPv4	UDP	1194	0.0.0.0/0	-	OpenVPN UDP	Delete Rule

Displaying 3 items

Figure 9 example security group configuration for openvpn server permitting udp port 1194.

**4.1.2.5 Install Nginx**

On the nginx server.

```
sudo apt install nginx
```

**4.1.2.6 Configure Public Domain HTTPS certificate**

As the Nginx server will be the public endpoint it will require a https certificate generated for a publicly available domain name. This can be applied using encryptbot and LetsEncrypt with the **public domain** of the service.

When generating the certificate ensure to allow wildcard subdomains to support multiple subdomains within the primary https domain. Other internal services will be configured with private certificates signed by the trusted local-ca and used internally.

**4.1.2.7 Nginx certificate and reverse proxy configuration.**

On the nginx host in the configuration for each subdomain that needs to be supported.

For example a reverse proxy configuration file located in /etc/nginx/conf.d/reverse.proxy.conf may contain multiple subdomain-based server blocks.

TODO: need to test putting internal CA signed ssl https behind nginx frontend with public signed ssl certs. The actual environment with public domain will be required to test this. Otherwise might need to simply use http internally.

```
server {
    listen 443 ssl;
    ssl_certificate /etc/nginx/certs/server.crt;
    ssl_certificate_key /etc/nginx/certs/server.key;
    server_name keycloak.hostname.org.au;
    access_log /var/log/nginx/keycloak.access.log;
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
        proxy_set_header Host $host;
        proxy_redirect off;
        server_name_in_redirect off;
        location / {
            proxy_pass
https://keycloak.local:8443/;
            sub_filter "keycloak.local:8443"
"keycloak.hostname.com";
            sub_filter_once off;
        }
    }
server {
    listen 443 ssl;
    ssl_certificate /etc/nginx/certs/server.crt;
    ssl_certificate_key /etc/nginx/certs/server.key;
    server_name flhub.hostname.org.au;
    access_log /var/log/nginx/flhub.access.log;
    proxy_set_header Host $host;
    proxy_redirect off;
    server_name_in_redirect off;
    location / {
        proxy_pass
https://jupyterhub.local:8080/;
        sub_filter "jupyterhub.local:8080"
"flhub.hostname.com";
        sub_filter_once off;
    }
}
```

#### 4.1.2.8 Postgresql server deployment.

Postgresql is a dependency of the keycloak server. As the keycloak server is deployed within the Nectar environment, postgres should be deployed as a “Database” using the Nectar environment tools, rather than hosting on a VM. The postgres database should permit connections on the internal network only. Firewall rules should permit only interconnection between the keycloak server and postgres server, and interconnection only over vpn for the purpose of management.

The keycloak server will require that a database, schema and user account exist on the postgres server to enable it to install the dependent database schema.

Login to the db server and use the postgres user account to create the required database objects (alternately use pgadmin).

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
sudo -u postgres psql -p 5432
```

**Create keycloak DB owner account.**

```
CREATE ROLE keycloak WITH  
    LOGIN  
    NOSUPERUSER  
    INHERIT  
    CREATEDB  
    CREATEROLE  
WITH PASSWORD 'mypassword';
```

**Create database keycloak.**

```
DROP DATABASE IF EXISTS keycloakdb_server;  
  
CREATE DATABASE keycloakdb_server  
    WITH  
    OWNER = keycloak  
    ENCODING = 'UTF8'  
    LC_COLLATE = 'en_AU.UTF-8'  
    LC_CTYPE = 'en_AU.UTF-8'  
    LOCALE_PROVIDER = 'libc'  
    TABLESPACE = pg_default  
    CONNECTION LIMIT = -1  
    IS_TEMPLATE = False;
```

**Exit the psql console and reconnect to the database and create the schema keycloak.**

```
sudo -u postgres psql -p 5432 -d keycloakdb_server
```

Create the keycloak schema.

```
CREATE SCHEMA IF NOT EXISTS keycloak  
    AUTHORIZATION keycloak;
```

**4.1.2.9 Keycloak server deployment.**

Keycloak server depends on the JVM runtime.

```
sudo apt install openjdk-21-jre openjdk-21-jdk
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

The most recent stable keycloak server should be downloaded from the keycloak.org website.

<https://www.keycloak.org/getting-started/getting-started-zip>

Keycloak should be extracted into the path (assuming version 26.3.0) /opt/keycloak/keycloak-23.6.0

**The keycloak user will be part of the group fl\_system**

```
sudo mkdir /opt/keycloak
# if fl_system is not created yet
sudo groupadd fl_system
#set group ownership of the /opt/keycloak directory.
sudo chgrp -R fl_system /opt/keycloak
sudo chmod -R g+w /opt/keycloak
# if keycloak user has not been created
sudo useradd -r -s /bin/false keycloak
sudo usermod -a -G fl_system keycloak
```

**Copy and extract keycloak to the target path.**

```
wget https://github.com/keycloak/keycloak/releases/download/26.3.0/keycloak-26.3.0.zip
sudo -u keycloak cp keycloak-26.3.0.zip /opt/keycloak
cd /opt/keycloak
sudo -u keycloak unzip keycloak-26.3.0.zip
```

**Open the keycloak configuration for editing.**

```
cd keycloak-26.3.0
vi conf/keycloak.conf
```

Configure the database connection.

```
# The database vendor.
db=postgres

# The username of the database user.
db-username=keycloak

# The password of the database user.
db-password=myspassword

# The full database JDBC URL. If not provided, a default URL is
set based on the selected database vendor.
db-url=jdbc:postgresql://dbhostname:5432/keycloakdb_server?
currentSchema=keycloak
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

The second stage is to create the secure keys for the keycloak host and once this is done verify that keycloak starts and that the host is reachable through the nginx reverse proxy at the keycloak subdomain.

On the local CA server generate the certificates for the keycloak server (see 4.1.2.2). The resulting certificate and keys will need to be distributed to the keycloak server and converted into pem format if they are in crt and key files.

```
openssl x509 -in server.crt -out server.pem  
openssl rsa -in server.key -text > serverkey.pem
```

Then the keycloak server configuration needs to be updated to enable the https interface using the keys in /opt/keycloak/keycloak-26.3.0/conf/certs/.

```
# The file path to a server certificate or certificate chain in  
# PEM format.  
https-certificate-file=${kc.home.dir}/conf/certs/server.pem  
  
# The file path to a private key in PEM format.  
https-certificate-key-file=${kc.home.dir}/conf/certs/  
serverkey.pem  
  
# The proxy address forwarding mode if the server is behind a  
# reverse proxy.  
proxy=reencrypt
```

The default https port for the keycloak server is port 8443. When running the keycloak server the command in the bin directory is **kc.sh**. However refer to the systemd section (4.1.2.17.1) on how to configure the server under systemd so that it is automatically started and managed by systemctl.

#### 4.1.2.10 Keycloak server realm and client configuration.

The keycloak server will require a realm configured for OIDC for the researcher to access the system (Figure 10). Research users will be registered in the realm allocated for the FL project. Authentication is then delegated by the jupyterhub and superlink components to the keycloak server to obtain access and refresh tokens.

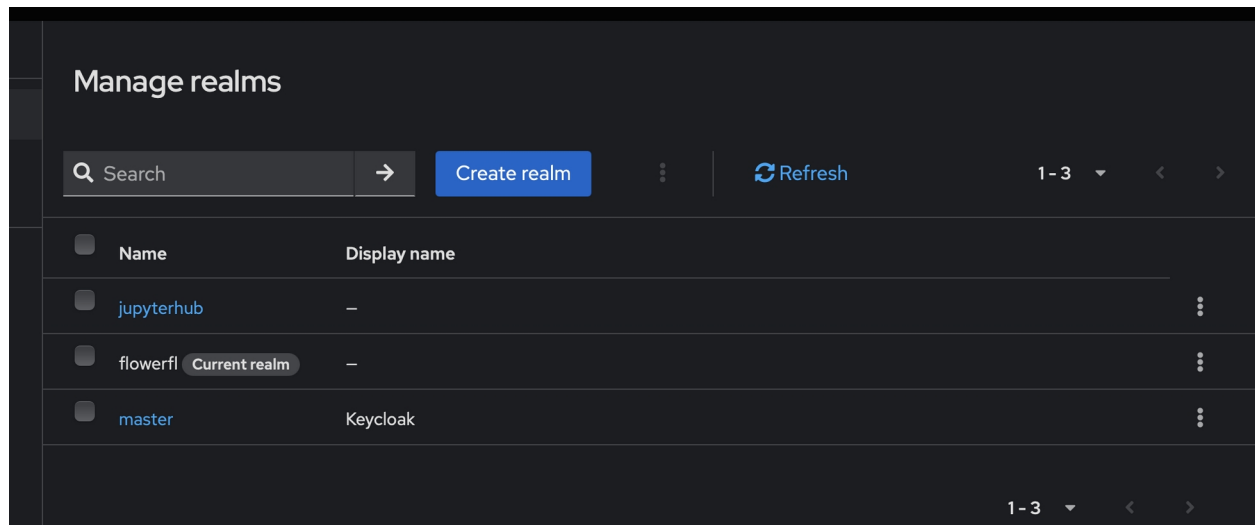
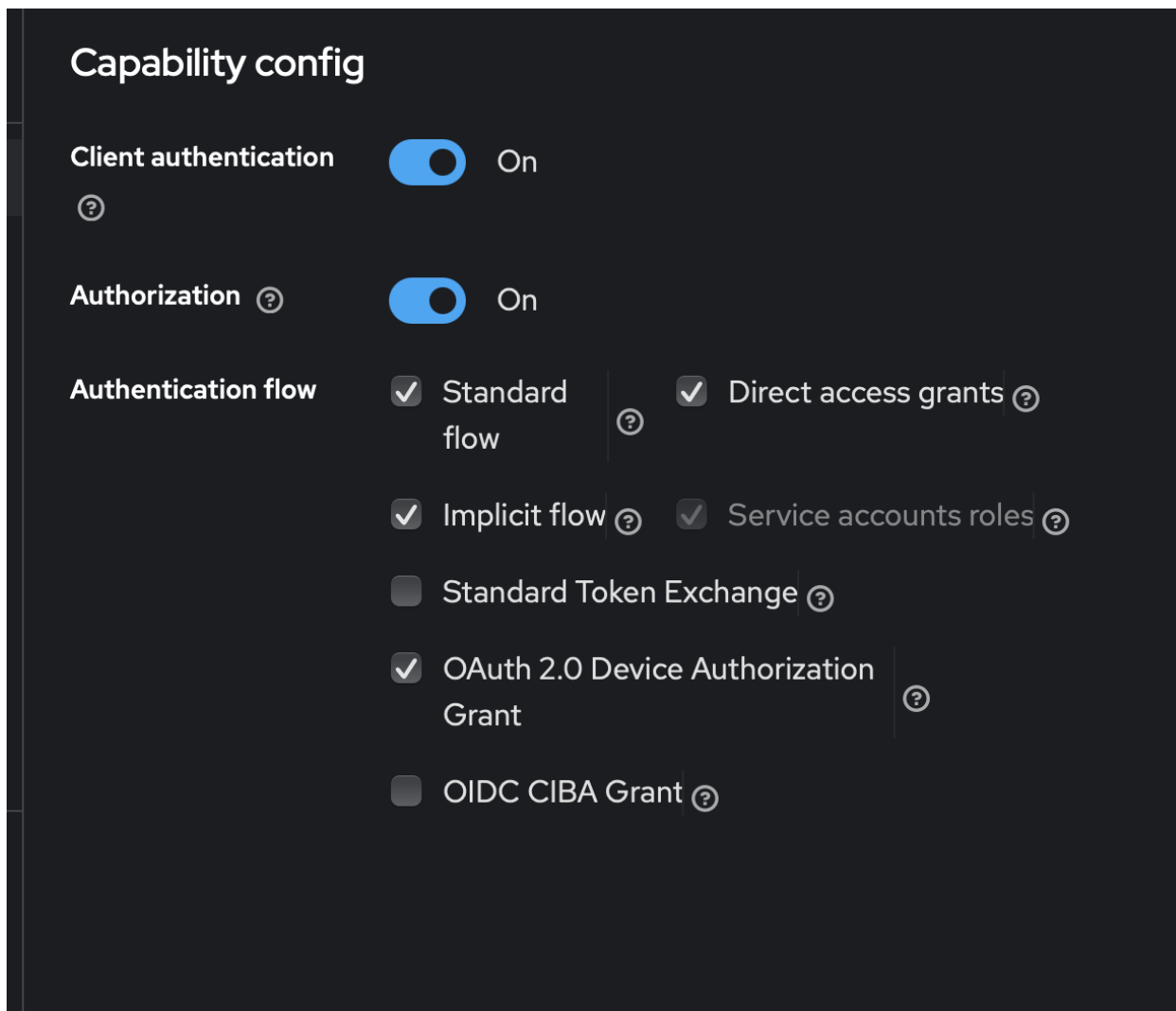
**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Figure 10 An example realm configuration for the flowerfl domain.

A client (eg flower\_auth\_client) needs to be configured for the realm to support programmatic access from other software components. The client will need to be configured with capabilities to support standard OAuth flows as well as OAuth 2.0 device authentication (Figure 11). The client id and associated client secret will be used to configure the jupyterhub and superlink to permit OIDC authentication.

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

### Capability config

**Client authentication** ☒ On

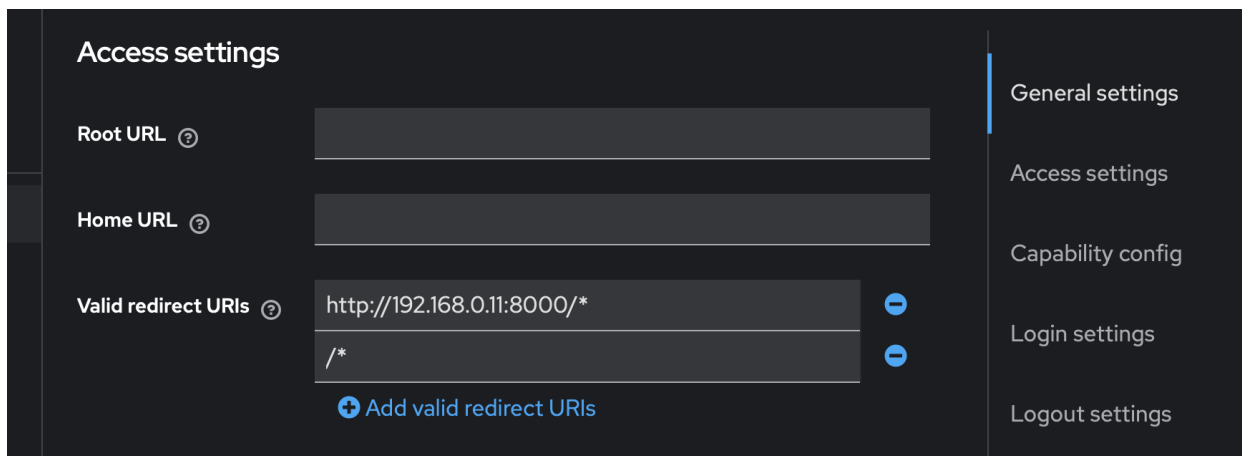
**Authorization** ☒ On

**Authentication flow**

- ☒ Standard flow
- ☒ Direct access grants
- ☒ Implicit flow
- ☒ Service accounts roles
- ☒ Standard Token Exchange
- ☒ OAuth 2.0 Device Authorization Grant
- ☐ OIDC CIBA Grant

Figure 11 Example selection of capabilities for client configuration.

The client configuration should also include the valid external domain URLs to permit redirection during the OAuth procedure (Figure 12).



### Access settings

**Root URL**

**Home URL**

**Valid redirect URIs**

- http://192.168.0.11:8000/\*
- /\*

[+ Add valid redirect URIs](#)

- General settings
- Access settings
- Capability config
- Login settings
- Logout settings

Figure 12 An example of the redirect uri configuration for a client access setting. This shows a local network configuration. In production a valid sub domain name (such as [https://flhub.mydomain.org.au/\\*](https://flhub.mydomain.org.au/*)) will be required to support redirection from the perspective of a client browser.



**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Additional realm roles and groups are to be defined to enable authorisation at the application level post the authentication procedure.

The following realm roles are recommended.

- jupyter\_admin
  - o A user with this role can administrate the jupyterhub.
- jupyter\_user
  - o A user with this role can access the jupyterhub notebooks.
- superlink\_user
  - o A user with this role can submit tasks to the superlink.

The groups are used to assign multiple users to inherit the roles allocated to the group. Three groups are recommended.

- jupyter\_admins
  - o Users in this group may administrate the jupyterhub, they may access jupyter notebooks and submit jobs to the superlink via jupyter and ssh.
  - o Associated roles: superlink\_user, jupyter\_admin
- jupyter\_users
  - o Users in this group may access jupyter notebooks and submit jobs to the superlink via jupyter and ssh.
  - o Associated roles: superlink\_user, jupyter\_user

#### 4.1.2.11 Python Virtual Environments

There are two virtual environments required for the platform.

Python will be installed on the servers hosting the jupyterhub, superlink and client supernode.

System accounts will be used to run these processes, and these system accounts will be the member of the group fl\_system.

The python virtual environments will be located at:

- /opt/python/jupyter\_env
  - o The python environment that will be configured to run jupyterhub.
- /opt/python/flower\_env
  - o The python environments configured for running superlink on the server.

Create the fl\_system group for setting up the python virtual environment.

```
# on all servers
sudo groupadd fl_system
```

Create the python folder.

```
# on all servers
sudo mkdir /opt/python
sudo chgrp -R fl_system /opt/python
sudo chmod -R g+w /opt/python
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Create the system user accounts on the servers running these components.

```
# on the keycloak server
sudo useradd -r -s /bin/false keycloak
sudo usermod -a -G fl_system keycloak
# on the jupyterhub server
sudo useradd -r -s /bin/false jupyterhub
sudo usermod -a -G fl_system jupyterhub
# on the superlink server
sudo useradd -r -s /bin/false superlink
sudo usermod -a -G fl_system superlink
# on the supernode clients
sudo useradd -r -s /bin/false supernode
sudo usermod -a -G fl_system supernode

# add the local operations user to the fl_system group
sudo usermod -a -G fl_system $USER
```

Create the python virtual envs.

```
# On the superlink server and supernode clients
python -m venv /opt/python/flower_env
# on the jupyterhub server
python -m venv /opt/python/jupyter_env
```

**4.1.2.12 Docker installation.**

Installing docker on ubuntu 24.04 uses the apt repositories.

Docker is installed on the jupyterhub server.

```
sudo apt install docker.io
sudo addgrp docker
sudo usermod -aG docker $USER
sudo usermod -aG docker jupyterhub
```

Login and logout of the shell to get the updated membership for the ops user.

Test the install can run docker from the intended user account (without sudo access).

```
docker run hello-world
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION****4.1.2.13 Jupyterhub deployment.**

Jupyterhub is deployed within the jupyter\_env python virtual environment.

```
source /opt/python/jupyter_env/bin/activate
pip install jupyterhub oauthenticator dockerspawner
sudo apt install npm -y
mkdir /opt/jupyterhub
cd /opt/jupyterhub
sudo npm install -g configurable-http-proxy
jupyterhub --generate-config
```

Note the node configurable-http-proxy is installed in /usr/local/lib/node\_modules.

The base directory “/etc/jupyterhub” will be used to store scripts and working folders required to execute the jupyterhub instance. After the configuration file “**jupyter\_config.py**” is generated it can be customised to support OIDC against keycloak. Once customised and tested it can be stored in /etc/jupyterhub/config/jupyterhub\_config.py.

```
# OAuth configuration
c.JupyterHub.authenticator_class = "generic-oauth"
#
#c.GenericOAuthenticator.allow_existing_users = True
# OAuth2 application info
# -----
c.GenericOAuthenticator.client_id = "flower_auth_client"
c.GenericOAuthenticator.client_secret = "mysecret-..."
# Identity provider info
# -----
c.GenericOAuthenticator.authorize_url = "https://
keycloak.domainname.org.au/realms/jupyterhub/protocol/openid-
connect/auth"
c.GenericOAuthenticator.token_url = "https://
keycloak.internal:8080/realms/jupyterhub/protocol/openid-
connect/token"
c.GenericOAuthenticator.userdata_url =
"https://keycloak.internal:8080/realms/jupyterhub/protocol/open
id-connect/userinfo"

# What we request about the user
# -----
# scope represents requested information about the user, and
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
since we configure
# this against an OIDC based identity provider, we should
request "openid" at
# least.
#
# In this example we include "email" and "groups" as well, and
then declare that
# we should set the username based on the "email" key in the
response, and read
# group membership from the "groups" key in the response.
#
c.GenericOAuthenticator.manage_groups = True
c.GenericOAuthenticator.scope = ["openid", "email", "groups"]
c.GenericOAuthenticator.username_claim = "preferred_username"
c.GenericOAuthenticator.auth_state_groups_key =
"oauth_user.groups"

# Authorization
# -----
#c.GenericOAuthenticator.allowed_users = {"user1@example.com"}
c.GenericOAuthenticator.allowed_groups = {"jupyter_users"}
#c.GenericOAuthenticator.admin_users = {"user2@example.com"}
c.GenericOAuthenticator.admin_groups = {"jupyter_admin"}

## The base URL of the entire application.
# #           Add this to the beginning of all JupyterHub URLs.
#           Use base_url to run JupyterHub within an existing
website.
# Default: '/'
# c.JupyterHub.base_url = '/'

## The public facing URL of the whole JupyterHub application.
# #           This is the address on which the proxy will bind.
#           Sets protocol, ip, base_url
# Default: 'http://:8000'
c.JupyterHub.bind_url = 'https://jupyterhub.internal:8000'
```

The external domain name is used for the configuration which is the target url the user is redirected to authenticate.

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
c.GenericOAuthenticator.authorize_url = "https://  
keycloak.domainname.org.au/realms/jupyterhub/protocol/openid-  
connect/auth"
```

Internal urls are used for the keycloak api.

```
c.GenericOAuthenticator.token_url = "https://  
keycloak.internal:8080/realms/jupyterhub/protocol/openid-  
connect/token"  
c.GenericOAuthenticator.userdata_url =  
"https://keycloak.internal:8080/realms/jupyterhub/protocol/open  
id-connect/userinfo"
```

The jupyterhub also binds to the internal host address as the external domain is terminated at the nginx reverse proxy.

TODO: configure internal ssl certificates

#### 4.1.2.14 Docker jupyter processes.

To prevent the requirement for system user accounts, the DockerSpawner method is applied to spawn instances of the docker container for jupyter.

```
# If set to 0, no limit is enforced.  
# Default: 100  configure based on hardware capabilities.  
c.JupyterHub.concurrent_spawn_limit = 20  
  
c.JupyterHub.spawner_class = 'dockerspawner.DockerSpawner'  
# notebook options  
c.DockerSpawner.image = "quay.io/jupyter/datascience-notebook"  
  
# list of allowed images.  
def allowed_images(self):  
    return ["quay.io/jupyter/datascience-notebook",  
            "quay.io/jupyter/tensorflow-notebook",  
            "quay.io/jupyter/pytorch-notebook"]  
c.DockerSpawner.allowed_images = allowed_images  
  
# User containers will access hub by container name on the  
# Docker network  
c.JupyterHub.hub_connect_ip = 'NNN.NNN.NNN.NNN'
```

The parameter "allowed\_images" specifies a set of images that the researcher can choose from after authenticating.

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Each of the docker images should be pulled prior to end use, this prevents timeouts occurring when launching the container eg:

```
docker image pull quay.io/jupyter/datascience-notebook
docker image pull quay.io/jupyter/tensorflow-notebook
docker image pull quay.io/jupyter/pytorch-notebook
```

**4.1.2.15 Jupyter docker mount points.**

The container can be configured without any persistence enabled. However to enable users to save their notebooks, it is useful to define a bind mount point.

This is achieved with the following configuration (in the jupyterhub\_config.py file).

```
import os
notebook_dir = os.environ.get('DOCKER_NOTEBOOK_DIR') or
'/home/jovyan/work'
c.DockerSpawner.notebook_dir = notebook_dir

# Mount a directory on the host to the notebook user's notebook
directory in the container
c.DockerSpawner.mounts = [
    {'source':
'/home/jupyterhub/workspace/qcif/jupyter/temp/{username}',
'target': notebook_dir, 'type': 'bind'}
]

def pre_spawn(spawner):
    print('Running pre-spawn')
    print(spawner)
    username = spawner.user.name
    ns = spawner.template_namespace()
    print(f'Escaped Username: {spawner.escaped_name}')
    mounts = spawner.mounts
    print(spawner.mounts)
    for mount_path in mounts:
        src_path = mount_path['source']
        src_path = src_path.replace("{username}",
spawner.escaped_name)
        if not os.path.exists(src_path):
            print(f'Create {src_path}')
            os.makedirs(src_path, exist_ok=True)
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
c.DockerSpawner.pre_spawn_hook = pre_spawn
```

The “pre\_spawn\_hook” function ensures the target path on the host exists prior to starting the container.

#### 4.1.2.16 Flower Superlink Deployment

The following section describes deployment of the superlink on the host server.

```
sh ~/python_envs/flower_env/bin/activate
pip install flwr flwr-datasets[vision] tensorflow==2.17
torch==2.2
```

Note tensorflow 2.17 and torch 2.2 depend on cuda 12.x and cudnn 8.9.x. later versions of torch may not be compatible with tensorflow 2.17.

When invoking commands for the superlink and supernode the flower\_env must be activated.

##### 4.1.2.16.1 Superlink Certificate generation.

On the local CA server responsible for generating certificates, it is possible to generate a signing request with custom certificate.conf file to provide additional aliases and ip addresses for the superlink server.

The server node generates a signing request and this is then the csr is sent to the local CA server which imports and signs the request. The resulting crt is sent back to the superlink server.

The configuration file for the local signing request must include all additional domain names for the internal network that will be used to identify the superlink servers. This is defined in a **certificate.conf** file.

```
[req]
default_bits = 4096
prompt = no
default_md = sha256
req_extensions = req_ext
distinguished_name = dn

[dn]
C = AU
ST = QLD
O = Flower
CN = localhost

[req_ext]
subjectAltName = @alt_names

[alt_names]
DNS.1 = hostname
DNS.2 = hostname.internal
DNS.3 = hostname2.internal
IP.1 = ::1
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
IP.2 = 127.0.0.1
IP.3 = 10.x.x.x
```

The signing request for the superlink server first generates a key and then generates the signing request csr file (assume \$CERT\_DIR is a directory the superlink will reference such as /etc/flwr/superlink/certs).

```
# Generate the root certificate authority key and certificate
based on key
openssl genrsa -out $CERT_DIR/server.key 4096

# Create a signing CSR
openssl req \
  -new \
  -key $CERT_DIR/server.key \
  -out $CERT_DIR/server.csr \
  -config certificate.conf
```

The signing request is signed by the CA and produces the server crt file. Only the csr file is sent to the local CA server. The key file remains on the superlink server.

```
# Generate a certificate for the server
sudo cp server.csr /usr/share/easy-rsa/temp
sudo cd /usr/share/easy-rsa
sudo ./easy-rsa import-req temp/server.csr superlinkhostname
sudo ./easy-rsa sign-req server superlinkhostname
```

The output of this command will produce a server key in the path:

- /usr/share/easy-rsa/pki/issued/server.crt

The resulting crt file can be copied back to the superlink server and converted to pem format:

```
openssl x509 -in server.crt -out server.pem
```

The resulting server.key and server.pem should be stored in a system path such as:  
/etc/flwr/superlink/certs

**4.1.2.16.2 Mutual Trust Certificates for Supernodes.**

Each supernode must have a public and private key pair for mutual trust authentication. The public keys of trusted supernodes are registered in csv file at the superlink.

Mutual trust certificates do not need signing by the local CA.



**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

The simple method for generating each required keypair per supernode is to use ssh-keygen.

```
ssh-keygen -t ecdsa -b 384 -N "" -f "${KEY_DIR}/client_credentials_$i" -C ""
```

The private and public key for the supernode should be stored in:

/etc/flwr/supernode/certs/client\_credentials\_n

/etc/flwr/supernode/certs/client\_credentials\_n.pub

The public key for each supernode needs to be sent to the superlink server and registered in a csv file. Using the pattern where the public key is indexed from 1 to n each key can be added to a superlink csv file containing trusted clients using a script similar to:

```
printf "%s" "$(cat "${KEY_DIR}/client_credentials_1.pub" | sed 's/.$//')>" > $KEY_DIR/client_public_keys.csv
for ((i=2; i<=${1:-2}; i++))
do
    printf ",%s" "$(sed 's/.$//' < "${KEY_DIR}/client_credentials_$i.pub")" >>
    $KEY_DIR/client_public_keys.csv
done
printf "\n" >> $KEY_DIR/client_public_keys.csv
```

The flower authentication example provides scripts for server and client key generation. These scripts should be modified to suit the target environment. Alternately, the above commands can be included in an automated deployment process.

**4.1.2.16.3 Command line arguments for the superlink.**

The superlink node may be executed with the generated certificates and command line arguments:

```
#!/bin/bash
export TF_FORCE_GPU_ALLOW_GROWTH=true
mkdir -p temp
flower-superlink \
    --ssl-ca-certfile /etc/ssl/certs/local-ca.crt \
    --ssl-certfile /etc/flwr/superlink/certs/server.pem \
    --ssl-keyfile /etc/flwr/superlink/certs/server.key \
    --auth-list-public-keys
/etc/flwr/superlink/trusts/client_public_keys.csv \
    --database temp/superlink_state.db
```

This document will also detail how to configure the the superlink and supernode for execution as a daemon using systemd.

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION****4.1.2.16.4 Custom configuration for OIDC.**

An additional layer of security may be added to the superlink node. This consists in the deployment of a QCIF customisation for the superlink. It is not required to be deployed on the supernodes.

The QCIF customisation (package TBD) will be deployed as a git repository and when executing the superlink the PYTHONPATH variable must be modified to include the base directory of the QCIF modules.

```
export PYTHONPATH=/usr/local/qcif/superlink/src:$PYTHONPATH
```

The custom script includes a modified wrapper for the superlink. It accepts configuration for OIDC authentication in keycloak. This method of authentication relies on the “device authentication flow” being enabled for the realm (shown in Figure 11). To enable this feature a custom authentication configuration is required. It is recommended to store in the path /etc/flwr/superlink/oidc/auth\_config.yaml.

```
authentication:
  auth_type: keycloak_authenticator
  settings:
    keycloak_url:
https://keycloak.hostname/realms/flowerfl/protocol/openid-
connect/
    keycloak_realm: flowerfl
    keycloak_client_id: flower_oidc
    keycloak_client_secret: mysecret
    keycloak_verify_ssl: false
    keycloak_mfa_code: false

authorization:
  authz_type: keycloak_authorizer
  settings:
    required_roles: ["fed-learn-user"]
```

Any roles named in the file need to be created and assigned to the appropriate groups to allow access to the superlink node for authenticated users.

The custom superlink script can then be invoked as:

```
#!/bin/bash
export TF_FORCE_GPU_ALLOW_GROWTH=true
export PYTHONPATH=/usr/local/qcif/superlink/src:$PYTHONPATH
python =/usr/local/qcif/superlink/custom_superlink.py \
  --ssl-ca-certfile /etc/ssl/certs/ca.crt \
  --ssl-certfile /etc/flwr/superlink/certs/server.pem \
  --ssl-keyfile /etc/flwr/superlink/certs/server.key \
  --auth-list-public-keys
/etc/flwr/superlink/trusts/client_public_keys.csv \
  --user_auth_config
/etc/flwr/superlink/oidc/auth_config.yaml \
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
--disable-oidc-tls-cert-verification
```

This will execute with TLS, mTLS and the additional OIDC authentication required for the end user.

The end user will not be able to use the superlink without authentication. Their project toml file must include a section to indicate to expect authentication at the flwr command client when running inside of jupyter. They will need also need a copy of the ca.crt included in their project.

```
[tool.flwr.federations.my-deployment]
address = "superlink.hostname:9093"
#insecure = true
root-certificates="./local-ca/ca.crt"
# experimenting with user authentication
enable-user-auth = true
auth-type = "oidc"
```

**4.1.2.17 Systemd configuration.**

The systemd configuration allows the services keycloak, superlink, supernode and jupyterhub to be managed under the systemctl command and to be configured to start automatically at boot time (after network initialization). The appropriate systemd configuration need only be applied on those hosts where the corresponding services are executed (for example on the supernode, only the supernode service need be configured). Services will have custom wrapper bash scripts associated with them as well as a service unit definition.

Some services will require a custom etc directory structure.

**4.1.2.17.1 Keycloak systemd configuration.**

The systemd unit definition and wrapper shell script are located /opt/keycloak.

```
/opt/keycloak
├── keycloak-26.3.0
├── keycloak.service
└── run_keycloak.sh
```

The systemd unit can be installed in the following manner.

```
sudo cp /opt/keycloak/keycloak.service /etc/systemd/system/
sudo systemctl enable keycloak.service
sudo systemctl start keycloak.service
sudo systemctl status keycloak.service
```

The stdout of the service can be traced using journalctl.

```
sudo journalctl -xeu keycloak.service
```

The “keycloak.service” unit has the following definition.

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
[Unit]
Description=Keycloak Server
After=network.target

[Install]
Alias=keycloak.service
WantedBy=multi-user.target

[Service]
User=keycloak
Group=fl_system
Type=simple
WorkingDirectory=/opt/keycloak/
ExecStart=bash run_keycloak.sh
Restart=always
RestartSec=5
```

The bash shell script “run\_keycloak.sh” is invoked by the systemd unit and has the following listing, the keycloak version depends on which version of keycloak is installed, so should be updated accordingly.

```
#!/bin/bash

export KEYCLOAK_VERSION="keycloak-26.3.0"
cd /opt/keycloak/$KEYCLOAK_VERSION

sh bin/kc.sh start --optimized
```

#### **4.1.2.17.2 Jupyterhub systemd configuration.**

As described in the section on installing jupyterhub, the jupyterhub has an /etc/jupyterhub directory with the following structure.

```
etc
├── jupyterhub
│   ├── config
│   │   └── jupyterhub_config.py
│   └── jupyterhub.conf
```

The jupyterhub.conf file contains the environment variable defining which jupyterhub\_config.py to load. It can be customised per environment.

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
# the config file to load.  
jupyter_config_file=jupyterhub_config.py
```

The /opt/jupyterhub directory has the following structure.

```
opt  
├── jupyterhub  
│   ├── jupyterhub.service  
│   ├── run_jupyterhub.sh  
│   └── temp
```

The service can be installed as follows:

```
sudo cp /opt/jupyterhub/jupyterhub.service /etc/systemd/system/  
  
sudo systemctl enable jupyterhub.service  
sudo systemctl start jupyterhub.service  
sudo systemctl status jupyterhub.service
```

The service unit “jupyterhub.service” has the following listing.

```
[Unit]  
Description=Jupyterhub Server  
After=network.target  
  
[Install]  
Alias=jupyterhub.service  
WantedBy=multi-user.target  
  
[Service]  
User=jupyterhub  
Group=fl_system  
Type=simple  
WorkingDirectory=/opt/jupyterhub/  
ExecStart=bash run_jupyterhub.sh  
Restart=always  
RestartSec=5
```

And calls the wrapper bash script “run\_jupyterhub.sh”, listed below.

```
#!/bin/bash
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
export PYTHONENV=/opt/python/jupyter_env/
source $PYTHONENV/bin/activate

cd /opt/jupyterhub

export TF_FORCE_GPU_ALLOW_GROWTH=true
export PYTHONPATH=/opt/jupyterhub/
export CONF_DIR=/etc/jupyterhub/

# load the server config variables.
source $CONF_DIR/jupyterhub.conf

#
jupyterhub -f $CONF_DIR/config/$jupyter_config_file
```

#### **4.1.2.17.3 Superlink systemd configuration.**

The superlink configuration directory /etc/superlink has the following structure.

```
etc
├── flwr
│   ├── superlink
│   │   ├── certs
│   │   │   ├── ca.crt
│   │   │   ├── client_public_keys.csv
│   │   │   ├── server.key
│   │   │   └── server.pem
│   │   └── superlink.conf
```

The directory contains an environment variable file “superlink.conf” which parameterizes the wrapper shell script, listed below:

```
# the superlink configuration file.
# the server certificate keyserver_certificate_key=server.key
server_certificate_pub=server.pem

# the CA certificate keyca_certificate_key=ca.key
ca_certificate=ca.crt
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
# allowed clients public keys csv.  
client_allowed_csv=client_public_keys.csv
```

This file defines the ssl certificates and allowed client public keys files relative to the /etc/flwr/superlink/certs directory.

The structure of the /opt/flwr/superlink directory contains the service unit definition and wrapper shell script shown below:

```
opt  
├── flwr  
│   ├── superlink  
│   │   ├── run_superlink.sh  
│   │   └── superlink.service
```

The service unit can be installed on the host system as follows:

```
# enable the superlink on the superlink server  
sudo cp /opt/flwr/superlink/superlink.service  
/etc/systemd/system/  
  
sudo systemctl enable superlink.service  
sudo systemctl start superlink.service  
sudo systemctl status superlink.service
```

The “superlink.service” unit definition is listed below:

```
[Unit]  
Description=Superlink Server  
After=network.target  
  
[Install]  
Alias=superlink.service  
WantedBy=multi-user.target  
  
[Service]  
User=superlink  
Group=fl_system  
Type=simple  
WorkingDirectory=/opt/flwr/superlink/
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
ExecStart=bash run_superlink.sh
Restart=always
RestartSec=5
```

And the wrapper shell script “run\_superlink.sh” sources the environment variable file and has the following structure:

```
#!/bin/bash

export PYTHONENV=/opt/python/flower_env/
source $PYTHONENV/bin/activate

cd /opt/flwr/superlink

export TF_FORCE_GPU_ALLOW_GROWTH=true
export PYTHONPATH=/opt/flwr/superlink/
export CONF_DIR=/etc/flwr/superlink/

# load the server config variables.
source $CONF_DIR/superlink.conf
# make sure the state directory is created
mkdir -p $PYTHONPATH/temp
flower-superlink \
  --ssl-ca-certfile $CONF_DIR/certs/$ca_certificate \
  --ssl-certfile $CONF_DIR/certs/$server_certificate_pub \
  --ssl-keyfile $CONF_DIR/certs/$server_certificate_key \
  --auth-list-public-keys $CONF_DIR/certs/$client_allowed_csv \
  --database $PYTHONPATH/temp/superlink_state.db
```

When running a multi-tenant deployment, each separate superlink node should be deployed on its own dedicated virtual machine with a separate openvpn server for the tenancy.

Note the superlink\_state.db file may become large over time. It is provided to persist logs of execution cycles by various clients. During operation it may need to be deleted and recreated depending on file size. This may be indicated through monitoring the available space of the filesystem for the superlink server.

### 4.1.3 Flower.Client

The flower client hosts the supernode and the associated VPN in site-to-site configuration. As this is the client side of the deployment, the end customer will most likely be responsible for the



**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

deployment. As such, it is intended that an automated method of deployment be defined, to install the base files and associated templates.

This section describes the manual installation procedure on ubuntu, and the associate structure. The intention is to provide information that can later be drawn on to develop an automated installation procedure for the flower supernode client. On ubuntu, this is likely to be a custom “deb” package.

#### 4.1.3.1 Local Service Users

A set of local service users will be created with no shell access. They will all be members of the system group “fl\_system”.

As this is a local system group it needs to be executed on all servers, including the supernode.

```
sudo groupadd fl_system
# add the current admin user to the fl_system group
sudo usermod -a -G fl_system $USER
# add service users
sudo useradd -r -s /bin/false supernode
# add the service users to the fl_system group.
sudo usermod -a -G fl_system supernode
```

#### 4.1.3.2 Client VPN Deployment

The client vpn will be deployed on the remote geographically separated host. The client vpn will connect to the openvpn server at the public ip address over udp on port 1194. It will use the client certificates generated on the openvpn server for authentication.

Create a directory for the client certificates on the client host /etc/openvpn/clientcerts.

Copy the ca.crt and issued client certificates to the client host.

```
clientcerts
├── ca.crt
├── issued
│   ├── client1.crt
├── private
│   ├── client.key
│   └── ta.key
```

In the path /etc/openvpn/client/client.conf save the client configuration.

```
client
proto udp
# the public IP of the openvpn server for example:
remote 203.101.229.46
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
port 1194

dev tap
nobind

remote-cert-tls server
tls-auth /etc/openvpn/clientcerts/private/ta.key 1
ca /etc/openvpn/clientcerts/ca.crt
cert /etc/openvpn/clientcerts/issued/client1.crt
key /etc/openvpn/clientcerts/private/client1.key
```

The client vpn connection can be tested by running the command:

```
sudo openvpn --config client/client.conf
```

The client connection should be established and this can be confirmed in the output of ifconfig. The internal network on the openvpn server can be reached using the ping command.

**4.1.3.2.1 Configure openvpn client to start at boot.**

The client vpn connection should be configured to start at boot using systemd like the server using the command:

```
sudo systemctl -f enable openvpn-client@client.service
```

Additional relevant systemctl commands are:

```
sudo systemctl status openvpn-client@client.service
sudo systemctl stop openvpn-client@client.service
sudo systemctl start openvpn-client@client.service
```

Reboot the client server and verify the client vpn connection starts at boot and that the internal network of the remote server is accessible.

At this point network traffic is encrypted over the vpn and the system is ready to install a supernode to connect to the superlink server.

**4.1.3.3 Flower SuperNode**

The flower supernode python environment needs to be installed in the same manner as the superlink environment (see 4.1.2.16).

The flower supernode connects to the superlink with mutual trust TLS certificates (created in 4.1.2.16.2). A private and public certificate must be created for each supernode forming part of the FL cluster. These are to be distributed to the supernode and stored in the paths:

/etc/flwr/supernode/certs/client\_credentials\_n

## DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION

/etc/flwr/supernode/certs/client\_credentials\_n.pub

The supernode command can then be invoked in the similar manner as to the superlink however the name of the superlink should be supplied, the number of partitions in the cluster (total number of supernodes) the partition number (starting from 0) of the supernode and the port on which the supernode will listen. If running more than one supernode each supernode must have a separate set of certificates (where the public key has been registered at the superlink) and must listen on a unique port.

In the example script below the supernode connects to superlink.hostname:9092, listens on port 9094 and represents the first partition (partition 0) out of 4 partitions.

Each supernode must participate as distinct partitions and the total number of partitions need to be specified in each cluster.

```
#!/bin/bash
export TF_FORCE_GPU_ALLOW_GROWTH=true

flower-supernode \
  --root-certificates /etc/flwr/supernode/certs/ca.crt \
  --superlink superlink.hostname:9092 \
  --clientappio-api-address 0.0.0.0:9094 \
  --node-config="partition-id=0 num-partitions=4" \
  --auth-supernode-private-key
/etc/flwr/supernode/certs/client_credentials_1 \
  --auth-supernode-public-key
/etc/flwr/supernode/certs/client_credentials_1.pub
```

### 4.1.3.3.1 Supernode systemd configuration

The supernode is installed as a systemd unit on the client system in a manner similar to the superlink. However it has more configuration options, and must have a unique partition number for each member of the FL fleet.

The configuration directory of the supernode is located in /etc/flwr/supernode and has the following layout.

```
etc
├── flwr
│   └── supernode
│       ├── certs
│       │   ├── ca.crt
│       │   ├── client_credentials_1
│       │   └── client_credentials_1.pub
│       └── supernode.conf
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

The `supernode.conf` file contains the environment variables which customise the shell wrapper script. These must be edited to suit the target environment. The purpose of each parameter is indicated with a comment.

```
# supernode configuration file.configuration
# the superlink server address to connect to.
superlink_server=superlink.ipaddress
superlink_port=9092
# the local ip address defaults to 0.0.0.0
local_address=0.0.0.0

# the local port for the clientapi io of the flower supernode.
local_port=9094

# the partition this system represents. must be unique in the
cluster.

local_partition=0

# The total number of partitions the system is participating
in.
num_partitions=2

# the client credentials certificates

client_credentials_key=client_credentials_1
client_credentials_pub=client_credentials_1.pub
ca_certificate=ca.crt
```

The **superlink\_server** and **superlink\_port** refer to the target superlink node at the internal address accessible over the VPN.

The **local\_port** is the local port on which the supernode will listen to communicate back to the superlink. If more than one supernode is configured on the same host this port must be unique.

The **local\_partition** must be unique over the entire cluster of supernodes, this must be identified by the FL system administration team when communicating with member organisations participating in the same cluster. When executing multiple tenancies the **local\_partition** is unique at the tenancy level.

The total **num\_partitions** identifies the total number of partitions in the cluster. This corresponds to the number of supernodes in the entire cluster for a given tenancy.

The additional parameters specify the ssl certificates described in the configuration.

The supernode can be enabled as a service unit in the following manner:

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
sudo cp /opt/flwr/supernode/supernode.service  
/etc/systemd/system/
```

```
sudo systemctl enable supernode.service  
sudo systemctl start supernode.service  
sudo systemctl status supernode.service
```

Note if running more than one supernode, a unique service file and service alias needs to be defined for each supernode (eg: supernode2.service). Different configuration files should also be defined and a separate wrapper script based on the original can also be defined.

The service unit definition “supernode.service” has the following listing:

```
[Unit]  
Description=Supernode Client  
After=network.target  
  
[Install]  
Alias=supernode.service  
WantedBy=multi-user.target  
  
[Service]  
User=supernode  
Group=fl_system  
Type=simple  
WorkingDirectory=/opt/flwr/supernode/  
ExecStart=bash run_supernode.sh  
Restart=always  
RestartSec=5
```

The service unit invokes the shell wrapper script “run\_supernode.sh”:

```
#!/bin/bash  
  
export PYTHONENV=/opt/python/flower_env/  
source $PYTHONENV/bin/activate  
  
cd /opt/flwr/supernode  
  
export TF_FORCE_GPU_ALLOW_GROWTH=true  
export PYTHONPATH=/opt/flwr/supernode/
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
export CONF_DIR=/etc/flwr/supernode/

# load the configuration variables into the environment.
source $CONF_DIR/supernode.conf

echo Connect to superlink $superlink_server Port:
superlink_port
echo Local address: $local_address Local port: $local_port
echo Partition Id: $local_partition Num-partitions:
$num_partitions

flower-supernode \
  --root-certificates $CONF_DIR/certs/$ca_certificate \
  --superlink $superlink_server:$superlink_port \
  --clientappio-api-address $local_address:$local_port \
  --node-config="partition-id=$local_partition num-
partitions=$num_partitions" \
  --auth-supernode-private-key
$CONF_DIR/certs/$client_credentials_key \
  --auth-supernode-public-key
$CONF_DIR/certs/$client_credentials_pub
```

**4.1.3.3.2 Supernode using docker and podman containers**

Deployment on the organisation system is achieved with docker or podman container images. The advantage of using containers is that the dependencies can be packaged within the container image and the participating organisation only requires installation of the docker runtime and has the choice to support runtime execution on CPU only or with GPU pass through.

Both versions of the containers have been configured with two customizable volumes.

- **/etc/flwr/supernode**
  - o The path inside the container for configuration.
- **/media/data**
  - o The path inside the container pointing to the host data directory.

The configuration of the supernode is the same as when deploying under systemd. The configuration file and certificates must be customized to suit the target environment (4.1.3.3.1).

The container is invoked using the appropriate run shell script which defines the custom paths to the **mount** paths for **src** (host path) and **dst** (container path).

When executing each command the **--mount** command line option defines this mapping. For example, the following declares a mapping between a local directory path containing the supernode configuration assets (src) and the path inside the container (dst).

```
- -mount
type=bind,src=$LOCAL_DIR/etc/flwr/supernode,dst=/etc/flwr/super
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
node
```

Similarly the following indicates a mapping between a local data directory (src) and the path inside the container (dst).

```
--mount type=bind,src=$LOCAL_DIR/data,dst=/media/data
```

**4.1.3.3.2.1 Deployment on docker or podman with CPU only containers.**

Execution of the cpu only container via shell script using docker is demonstrated in the listing below.

```
#!/bin/bash

export LOCAL_DIR=`pwd`
docker run --mount
type=bind,src=$LOCAL_DIR/etc/flwr/supernode,dst=/etc/flwr/super
node \
--mount type=bind,src=$LOCAL_DIR/data,dst=/media/data \
supernode_cpu:v1.0
```

Podman varies only slightly in that the hostname for the configured superlink is added so that it can be resolved using the container network gateway.

```
#!/bin/bash

export LOCAL_DIR=`pwd`
podman run --mount
type=bind,src=$LOCAL_DIR/etc/flwr/supernode,dst=/etc/flwr/super
node \
--mount type=bind,src=$LOCAL_DIR/data,dst=/media/data \
--add-host=mysuperlink.hostname:host-gateway \
localhost/supernode_cpu:v1.0
```

The **add-host** option should be customized so that the example “superlink.hostname” is replaced with the actual IP or hostname of the superlink server in production. The superlink server is reachable when the VPN is connected to the Nectar cloud.

**4.1.3.3.2.2 Deployment on Linux host using docker, podman with GPU pass through.**

In order to enable Nvidia GPU pass through for linux, in addition to installation of Nvidia drivers the nvidia container toolkit must also be installed on the linux host. Instructions to achieve this are described at the link: <https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html>.

The container image includes cuda runtime dependencies. In docker it is executed as follows:

```
#!/bin/bash

export LOCAL_DIR=`pwd`
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
docker run --mount
type=bind,src=$LOCAL_DIR/etc/flwr/supernode,dst=/etc/flwr/super
node \
--mount type=bind,src=$LOCAL_DIR/data,dst=/media/data \
--runtime=nvidia --gpus '"device=1"' \
supernode_cuda:v1.0
```

Note that the “device” should be customised to suit the GPU environment or optionally set to “all”.

#### 4.1.3.3.2.3 Deployment on Windows Host with docker, podman with WSL2 and GPU pass through.

TODO: need to test deployment on a windows host and test WSL2 GPU passthru.

## 4.2 Deployment Validation

Describe how to validate the system is deployed correctly. Ideally step by step.

TODO: clone a QCIF test repository. Update configuration for the superlink node. Run test.

## 4.3 Data Management Procedures

The primary objects that require backup are:

- Configuration files
- Certificates
- Databases

Configuration files and certificates should be backed up to a representative directory tree and synchronised with a git repository. This can be done at the time a change is made to the system. Each time a change is pushed to git it should be tagged with an appropriate tag associated with the change.

Database backup and restore are described in the following sections.

### 4.3.1 Database Backup Procedures

The following databases are deployed on the system.

System Partition	Database	Schema	Description
Server	keycloakdb_server	keycloak	PostgresDB instance for the keycloak IAM server.
Server	grafanadb_server	grafana	PostgresDB instance for the Grafana monitoring server.

A backup of the postgresdb can be performed using pg\_dump from a connected system which has postgres installed (security group dependent) or on the same database host. In the following



**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

example the pgdump command is used to make a binary format backup of the database named: "database\_name", where the user will be prompted for their password after the connection is made.

```
pg_dump -Fc -d database_name -h hostname -p 5432 -U username > database_yyyyMMdd.pgdump
```

A database backup should be made after the initial release and on month schedule and archived depending on storage availability.

Alternate methods of backup are provided by backing up the instance within the nectar cloud infrastructure and depends on storage allocation.

### 4.3.2 Database Restore Procedures

A database can be restored from a binary dump file using the pg\_restore command. In the following example the database "database\_name" is restored from a backup pgdump file.

```
pg_restore -h hostname -p 5432 -U username -c -d database_name database_yyyyMMdd.pgdump
```

A database restore will lose data depending on the length of time between backups.

## 5 Monitoring

Monitoring is implemented using Grafana to provide visibility as well as alerting, Prometheus for data storage and Prometheus Node Exporter processes on each of the hosts in the cluster to export metrics for each node. Grafana enables the operations team to configure alarming and monitoring functionality in a flexible manner. This section will describe the key alarms which will alert operations to system failure modes and require action. Additional diagnostics are provided in the Grafana dashboard, log files and host system utilities.

### 5.1 Grafana and components installation.

Grafana, Prometheus and Prometheus node export installation processes are well documented. These components can be installed via apt repositories for ubuntu. For windows clients the Prometheus-node-export package should be installed independently (or if using ubuntu on WSL2 may be installed via apt).

The Grafana and Prometheus time series database are to be installed on the Nectar cloud network segment on a server dedicated for operational monitoring and alarming. The Prometheus-node-exporter is to be installed on those servers which will be monitored within the infrastructure.

Details for installation of Grafana can be found at:

<https://grafana.com/docs/grafana/latest/setup-grafana/installation/debian/>

Details for installation of Prometheus can be found at:

<https://prometheus.io/download/#prometheus>

It can be installed via the command "apt install prometheus".

For those hosts requiring monitoring, the node-exporter can be installed via "apt install prometheus-node-exporter".

#### 5.1.1 GPU Monitoring Components

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Components to monitor the GPU are more complex. To achieve this the system requires an installation of the Nvidia Data Center GPU Management components and a manual installation of the dcfgm-exporter tool (installation and further details here: <https://developer.nvidia.com/dcfgm> ).

On Ubuntu the Nvidia DCGM service can be installed by using the same repository as for cuda. This is documented here:

<https://docs.nvidia.com/cuda/cuda-installation-guide-linux/#prepare-ubuntu>

The main package can be installed through apt.

```
sudo apt install datacenter-gpu-manager
```

Once installed it can be enabled using systemctl.

```
sudo systemctl enable nvidia-dcgm
sudo systemctl start nvidia-dcgm
```

The dcfgm-exporter needs to be compatible with the libdcgm version installed via the datacenter-gpu-manager package (version 3.3.9 at the time of writing). Check the version in the output of apt search datacenter-gpu-manager:

```
sudo apt search dcgm

Sorting... Done
Full Text Search... Done
datacenter-gpu-manager/unknown,now 1:3.3.9 amd64 [installed]
  NVIDIA® Datacenter GPU Management Tools
```

The tag of the release for dcfgm-exporter that is compatible with 3.3.9 is "3.3.9-3.6.1" and can be obtained from the git repository.

```
mkdir nvidia-dcgm
cd nvidia-dcgm
git clone https://github.com/NVIDIA/dcfgm-exporter.git
# get version compatible with libdcgm-3.3.9
git pull origin 3.3.9-3.6.1
git checkout 3.3.9-3.6.1
cd dcfgm-exporter
make binary
sudo make install
```

Note that building requires gcc and go lang to be installed. Once built and installed the dcfgm-exporter can be tested (it requires root access for the libdcgm to access the gpu).

```
sudo dcfgm-exporter --address=":9400"
```

After testing, exit the process. To install a systemd unit can be defined in the file "dcgm-exporter.service" as follows:

```
[Unit]
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
Description=DCGM Exporter
After=network.target

[Install]
Alias=dcgm-exporter.service
WantedBy=multi-user.target
After=nvidia-gpu-reset.target
Wants=nvidia-gpu-reset.target
```

```
[Service]
User=root
PrivateTmp=false
Type=simple
ExecStart=dcgm-exporter --address=":9400"
Restart=always
RestartSec=5
```

And can be installed as per the systemd service unit.

```
sudo cp dcgm-exporter.service /usr/lib/systemd/system/
sudo systemctl enable dcgm-exporter.service
sudo systemctl start dcgm-exporter.service
sudo systemctl status dcgm-exporter.service
```

This will cause the dcgm-exporter to be enabled at boot and to listen on port 9400.

To include it in the Prometheus data collection it can be added as another job section on the Prometheus host in **/etc/Prometheus/prometheus.yml** (refer to 5.2.3).

Each node hosting a dgcm-exporter needs to be added.

```
- job_name: gpu_nodes

  scrape_interval: 30s
  scrape_timeout: 5s

  static_configs:
    - targets: ['host1:9400', 'host2:9400']
```

Restart the Prometheus service to start collecting gpu metrics.

Grafana dashboards for GPU metrics can be obtained from the github project page <https://github.com/NVIDIA/dcgm-exporter>.

The dashboard includes key metrics of interest include the temperature and memory use of GPU.

## DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION

## 5.2 Configuration

Configuration should be customized to support the target environment. In particular, configuration of Grafana should include https certificate for the web host, and OAuth authentication against keycloak. Since postgres is deployed in service of keycloak, a separate database instance can be created to enable Grafana to store its own data, this can be prepared in a manner similar to the keycloak procedure.

The configuration procedure should follow the following steps.

1. Prepare the postgres account role, Grafana database and schema for use by Grafana.
2. Install and configure Grafana host.
3. Collect information about each host to be monitored and deploy and configure the Prometheus-node-exporter components.
4. Install and configure Prometheus to collect metrics for each of the monitored hosts.
5. Use Grafana to assign Prometheus data sources, define alerting and dashboards.

For full reference to configuration of Grafana, Prometheus and Prometheus-Node-Exporters refer to the following links:

Grafana Configuration - <https://grafana.com/docs/grafana/latest/setup-grafana/configure-grafana/>

Prometheus

Configuration

-

<https://prometheus.io/docs/prometheus/latest/configuration/configuration/>

Prometheus Node Exporter Configuration - <https://prometheus.io/docs/guides/node-exporter/>

### 5.2.1 Configuration Customisation – Grafana

Configuration customisations for Grafana includes the postgres database connection.

The configuration file for Grafana is located in:

```
/etc/grafana/grafana.ini
```

After a postgres Grafana role, database and schema have been created (see keycloak installation for an example). The configuration file can be updated to allow connection to the postgres database in the following way:

```
# Either "mysql", "postgres" or "sqlite3", it's your choice
type = postgres
# Use either URL or the previous fields to configure the
database
url =
postgres://grafana:mypassword@127.0.0.1:5432/grafanadb_server?
options=-csearch_path=grafana
```

The above configuration uses the postgres url to include the role username and password, the database server and the schema defined in the options parameter (named grafana).

Additional configurations for Grafana include OAuth configured in the Generic OAuth section of the grafana.ini file (<https://grafana.com/docs/grafana/latest/setup-grafana/configure-security/configure-authentication/keycloak/> ).

Email configuration for alerting is also necessary (<https://grafana.com/docs/grafana/latest/alerting/configure-notifications/manage-contact-points/integrations/configure-email/> ).

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Optional features for alerting also include configuration for Slack (<https://grafana.com/docs/grafana/latest/alerting/configure-notifications/manage-contact-points/integrations/configure-slack/>).

Once configured, systemctl can be used to enable and start the Grafana server at boot time.

### 5.2.2 Configuration Customisation – Prometheus node exporter

The default port for the node exporter is 9100. Depending on the target environment, the port may be changed. Custom command line arguments can be overridden by modifying the file:

```
/etc/default/prometheus-node-exporter
```

For example to change the default port:

```
ARGS="--web.listen-address=:9900"
```

The service is managed with systemd.

```
sudo systemctl stop prometheus-node-exporter
sudo systemctl start prometheus-node-exporter
sudo systemctl status prometheus-node-exporter
```

Restart to apply the custom arguments.

### 5.2.3 Configuration Customisation – Prometheus Server

On the server hosting the monitoring server install Prometheus. The configuration is located in:

```
/etc/prometheus/prometheus.yml
```

Each node exporter in the network that is monitored needs to be included in the jobs configuration. For example:

```
scrape_configs:

# The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.

- job_name: 'prometheus'

# Override the global default and scrape targets from this job every 5 seconds.
  scrape_interval: 30s
  scrape_timeout: 5s

# metrics_path defaults to '/metrics'

# scheme defaults to 'http'.
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
static_configs:

  - targets: ['localhost:9090']

- job_name: node
  # If prometheus-node-exporter is installed, grab stats about the local
  # machine by default.
  static_configs:
    - targets: ['localhost:9900', 'server1:9900', 'server2:9900']
```

Storage arguments can also be customized depending on the available disk space to retain the collected metrics. On very small servers it may require only 1d and a small amount of space for storage. For operational purposes it is useful to have sometime to review metrics in the dashboard however this is dependent on available resources. The custom startup parameters are found in the file:

```
/etc/default/prometheus
```

An example of modifying storage parameters for a small system are shown below:

```
ARGS="--storage.tsdb.retention.time=1d --storage.tsdb.retention.size=128MB"
```

For more information on storage parameters see:  
<https://prometheus.io/docs/prometheus/latest/storage/>

Prometheus is also managed under systemd.

```
sudo systemctl enable prometheus.service
sudo systemctl stop prometheus.service
sudo systemctl start prometheus.service
sudo systemctl status prometheus.service
```

## 5.3 Alerting

Alerting is necessary to proactively manage the infrastructure. It is useful to start with a common set of alerts and add additional detail as operations learns more about supporting the platform.

Standard alerting should be applied to:

- Disk free space
- Persistent High CPU
- Persistent High RAM
- Services crashing.

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

It is also useful to add log / pattern matching tool which can match on specific patterns indicating:

- Out of memory exceptions in logs
- Other key error messages indicating service failure.

Grafana supports simple “alert rule” capabilities which can send a notification by email or to other channels such as slack. The following basic metrics or similar should be used for common operational alerting. Additional resources on alerting with Grafana are available here: <https://grafana.com/tutorials/alerting-get-started/>.

### 5.3.1 Common alerting metrics – CPU Load

CPU Load is estimated for all cpus combined (this article is insightful regarding load averages <https://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html>). A simple method of alerting over the 15 minute average load can be configured in Grafana using the query:

```
node_load15/(count without(cpu, mode)
(node_cpu_seconds_total{mode="idle"}))
```

The divisor is a query which extracts the number of cpus in the system. The load average is therefore the ratio between 0 and 1.

The threshold for the alert can be set at > 0.8 (80% load average) for at least 30 minutes for medium severity and > 0.95 for at least 30 minutes for high severity.

### 5.3.2 Common alerting metrics – Memory

The ratio between free memory and total memory available in the system is expressed in the Grafana query below:

```
node_memory_MemFree_bytes / node_memory_MemAvailable_bytes
```

This is a ratio between 0 and 1. Values closer to 1 mean a high amount of free memory, values close to 0 mean a low amount of memory.

The threshold can be set to < 0.1 for 30 minutes for medium and < 0.05 for 30 minutes for high severities (depending on the physical system resources).

### 5.3.3 Common alerting metrics – Disk volume space.

Alerting on available disk volume metrics is very useful, especially since some logging may not be configured to be cleaned up appropriately.

Since there may be many mount points in the system related to containers, it is possible to filter only for the most important devices or mount points in the query. The following Grafana query gives the ratio between free bytes and total size bytes for the mount point “/”.

```
node_filesystem_avail_bytes{mountpoint="/" } /
node_filesystem_size_bytes{mountpoint="/" }
```

It is useful to add this alert for each priority mount point in the system.

The ratio is a value between 0 and 1. Like the memory ratio, it is close to 1 if there is a lot of free space, otherwise close to 0 if there is very little free space.

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Multiple levels of thresholds can be configured for  $< 0.3$ ,  $< 0.2$ ,  $< 0.1$  for low, medium and high severity alerts.

**5.3.4 Common alerting metrics – Disk volume space.**

If a service has failed, this may be a critical alert for key services. hence monitoring the running status can be a useful indicator. But other cases may be useful if the is also restarting ('activating') repeatedly which could flag an issue in the service. For each key service it may be useful to raise and alert if it is in a failed state (value is 1 if the service is failed) for a period such as 30 minutes. The threshold should be set if  $> 0$ .

An example Grafana query for the nginx.service running in systemd is shown below:

```
node_systemd_unit_state{name="nginx.service", state=~"failed|inactive", instance="localhost:9900"}
```

Note the instance includes the host where nginx is deployed.

In addition, if there is a sum for "activating" that is greater than 1 over the last 30 minutes, this can indicate that the service may be restarting repeatedly which indicates a failed state.

```
sum(node_systemd_unit_state{name="nginx.service", state="activating", instance="localhost:9900"})
```

A separate alert rule should be configured for each of the key service units installed in the system. This includes the following system.

- docker.service
- nginx.service
- jupyterhub.service
- keycloak.service
- openvpn-server@server.service
- openvpn-client@client.service
- postgresql.service
- ssh.service
- superlink.service
- supernode.service

**5.3.5 GPU alerting metrics – Temperature.**

GPU alerting is important for systems performing deep learning or machine learning on GPU hardware. It is not uncommon for GPU to run at reasonably high temperature, such as 70 degrees Celsius. However, running a very high temperatures for an extended duration of time above accepted thresholds such as 85 degrees Celsius may be an indication of hardware failure. The dgcgm exporter produces metrics to support monitoring the GPU. Temperature can be alerted on after prolonged duration of 30 minutes or 1 hour depending on expected workload.

```
DCGM_FI_DEV_GPU_TEMP{gpu=~"(0|1)"}
```

Threshold may be set at  $\geq 85$  degrees Celsius.

Note the GPU expression may be altered where there are more than two available GPUs.

**5.3.6 GPU alerting metrics – Memory Utilisation.**



## DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION

In general, a high GPU workload will maintain a high memory percentage utilisation. This will of course limit the ability for GPU enabled jobs running on the cluster as libraries such as tensorflow and pytorch will need to claim memory during the runtime. Monitoring the GPU memory use will allow identification of which servers are processing heavy workloads and will allow operations to proactively respond if jobs are unable to execute. This may mean deliberately limiting the workload due to limited hardware, or alternately provisioning CPU only supernode/superlink clusters in parallel to the GPU enabled clusters to allow deprioritised training processes.

Memory metrics exported by DCGM are:

```
DCGM_FI_DEV_MEM_COPY_UTIL{gpu=~"(0|1)"}
```

A threshold above 90% for 1 hour will allow detection of long running tasks. This alert may simply be a low priority threshold since deep learning training processes will require high memory use for long durations. More than 24 hours will not be unusual.

Note the GPU expression may be altered where there are more than two available GPUs.

## 5.4 Logging Details

There are three types of log management approaches.

1. Systemd – journal logging.
  - a. Most systemd service units will have logging managed under the journal system.
2. Logs managed by logrotate.
  - a. Most packages installed by apt will be managed under the logrotate daemon if not managed by systemd journal.
3. Logs managed by application specific configuration.
  - a. Custom applications will typically require customized log configuration for the associated framework such as python logging, log4j, logback, log4net and serilog. These custom frameworks typically support limits for log file size, and file rotation.

The list of components and associated log management method are shown in Table 7.

*Table 7 List of components and log management procedure.*

System Partition	Component	Log Management	Path
Server	NGinx	Logrotate.d	/var/log/nginx
Server	Jupyterhub	Systemd journal	/var/log/journal
Server	Keycloak	Systemd journal	/var/log/journal
Server	Postgres	Logrotate.d	/var/log/postgres
Server	SuperLink	Systemd journal	/var/log/journal
Server	OpenVPN Server	Systemd journal	/var/log/journal
Server	Grafana	Logrotate.d	/var/log/grafana
Server	Prometheus	Logrotate.d	/var/log/prometheus
Server,Client	Prometheus Node Exporter	Logrotate.d	/var/log/prometheus-node-exporter
Client	Open VPN Client	Systemd journal (on ubuntu)	/var/log/journal
Client	SuperNode	Systemd journal	/var/log/journal

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

		(on ubuntu)	
--	--	-------------	--

**5.4.1 Systemd journal.**

The default maximum size of the journal on ubuntu is 4Gb but can be modified in the configuration file `/etc/systemd/journald.conf` for example reducing the maximum size to 1Gb:

```
# /etc/system/journald.conf
[Journal]
SystemMaxUse=1G
SystemKeepFree=100G
```

Details of configuration are found at:

<https://www.freedesktop.org/software/systemd/man/latest/journald.conf.html>

The journald process is managed under systemd for example:

```
sudo systemctl restart systemd-journald.service
sudo systemctl status systemd-journald.service
```

Viewing service unit logs via the journal uses the command “journalctl”, for example tailing the log for nginx.service unit:

```
sudo journalctl -u nginx.service -f
```

Or reviewing nginx logs for the last two hours:

```
sudo journalctl -u nginx.service --since "2 hours ago"
```

The “journalctl” can also be used to reduce the journal size:

```
sudo journalctl --vacuum-size=200M
```

For additional usage see also: <https://www.loggly.com/ultimate-guide/using-journalctl/>

**5.4.2 Logrotate**

Logrotate is configured by adding configuration specifications in the path `/etc/logrotate.d/`.

Each file (or wildcard file match) can be declared in a separate configuration file, for example the `/etc/logrotate.d/prometheus-node-exporter` configuration below:

```
/var/log/prometheus/prometheus-node-exporter.log {
    weekly
    rotate 10
    copytruncate
    compress
    delaycompress
    notifempty
    missingok
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```
}
```

The frequency of rotation and the number of backups depend on storage space available and requirement of operations to look back in history to support an application.

For example, to add a file to rotation, create a new record for that service for example an application called "myapp" which logs to "/var/log/myapp/myapp\*.log" and rotates daily the following configuration may be used:

```
/var/log/myapp/myapp*.log {  
    daily  
    rotate 5  
    copytruncate  
    compress  
    delaycompress  
    notifempty  
    missingok  
}
```

The logrotate is managed by a systemd timer instance on ubuntu which by default runs daily.

Systemd timers can be customised or added using the systemctl functions. Further details on systemd timers are available at: <https://opensource.com/article/20/7/systemd-timers>.

Additional notes on logrotate are available at:

<https://www.digitalocean.com/community/tutorials/how-to-manage-logfiles-with-logrotate-on-ubuntu-20-04>.

## 5.5 Diagnostics

Several methods of diagnostics are available.

- Grafana
- Systemd status
- Logging
- Flwr command line

### 5.5.1 Grafana Dashboards.

It is recommended to setup a dashboard that displays the status of configured alerts. Figure 13 illustrates a simple configuration for system alerts. Grafana also provides multiple features for system monitoring and the display of collected metrics such as illustrated in Figure 14. As the operations team become more familiar with the system, they will add dashboards for specific use cases. However, a default set of dashboards can be built initially to monitor alerts and the metrics associated with alerting. These can be used to visually diagnose the status of alerts and the associated system metrics.

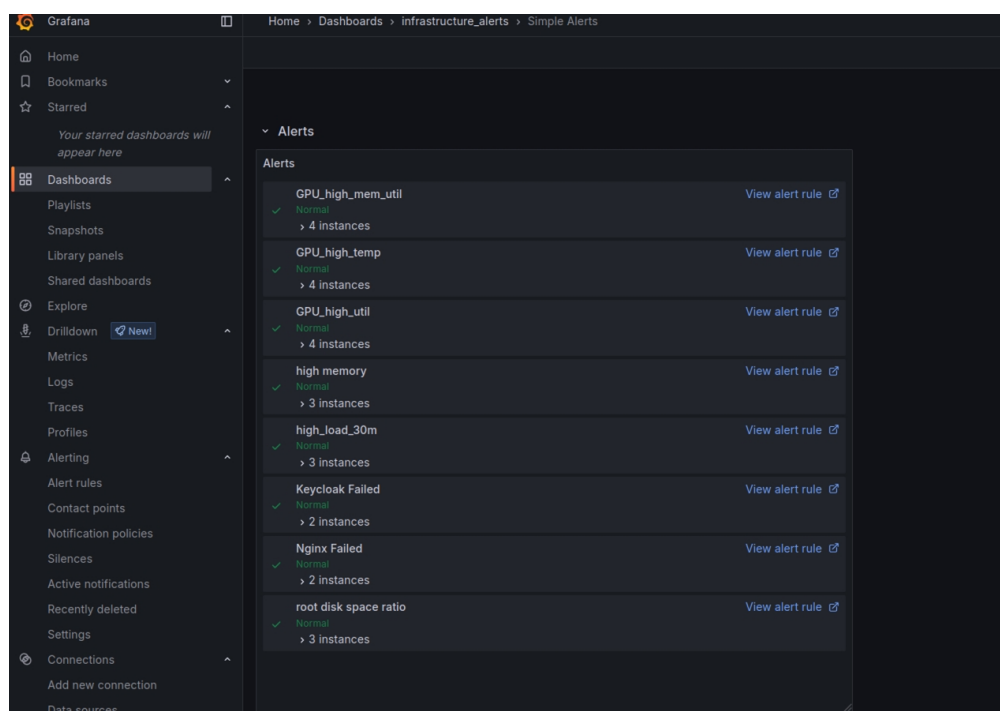
**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

Figure 13 Example alert dashboard for default system alerts.

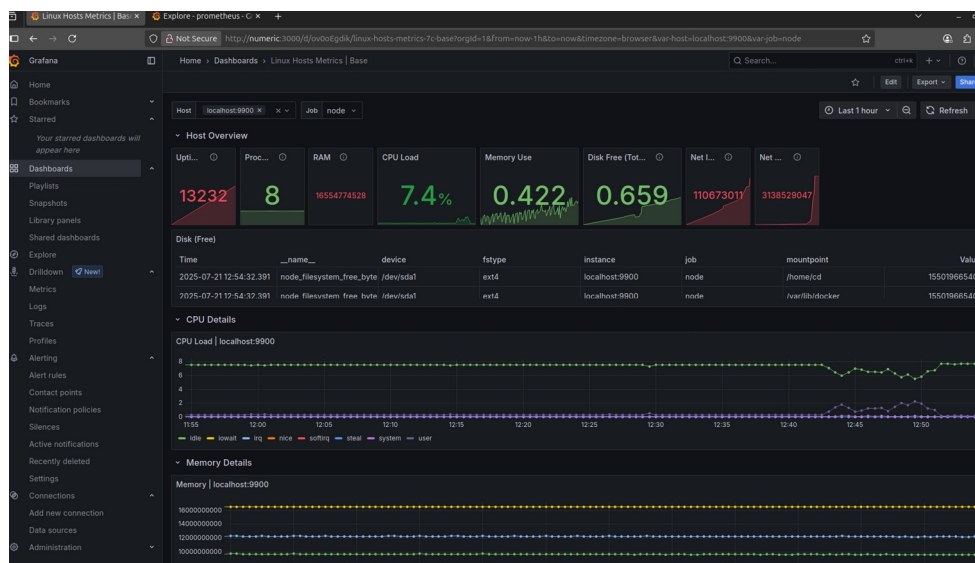


Figure 14 An example Grafana dashboard monitoring metrics collected from linux hosts.

## 5.5.2 Systemd status

The status of systemd units can be obtained via an SSH session to the target system. The service unit must be known to obtain status. The list of available service units can be displayed:

```
sudo systemctl list-units --type=service --state=loaded
```

The key service units in the system are:

- docker.service
- nginx.service
- jupyterhub.service

## DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION

- keycloak.service
- openvpn-server@server.service
- openvpn-client@client.service
- postgresql.service
- ssh.service
- superlink.service
- supernode.service

The runtime details for a service unit can be obtained with the status command, as for nginx in the following example:

```
sudo systemctl status nginx.service
```

### 5.5.3 Journalctl Logs

For those service units logging outputs to journal the log can be accessed through the journalctl command. As in the following examples for nginx service.

```
sudo journalctl -u nginx.service -f
sudo journalctl -u nginx.service --since "2 hours ago"
```

The same command can be piped to tools such as grep.

Components using journald logging are listed in 5.4.

### 5.5.4 File based logs.

File based logs can be accessed directly from the file system, followed using the tail command and searched via the grep command.

Components using file based logging are listed in 5.4.

### 5.5.5 Flwr command line.

The flwr command line enables the end user to obtain logs for jobs submitted to a superlink. The requirement is that the project toml file be present and declare the associated superlink where the job has been submitted.

Details of work done by the superlink and supernode are by default accessible via journalctl when installed as a systemd service unit. However the end user will not have access to this level of the system. Instead they can obtain status and details of their project using the build in commands such as:

```
flwr ls . superlink-deployment
```

Where superlink-deployment is the project section naming the appropriate superlink. The output of the ls command will provide a list of running jobs.

```

Loading project configuration...
Success
■ Listing all runs...

```

Run ID	FAB	Status	Elapsed	Created At	Running At	Finished At
11693208818826240040	cd/flower-tf-quickstart (v1.0.0)	running	00:00:05	2025-07-21 03:57:18Z	2025-07-21 03:57:21Z	N/A

Figure 15 Example output of the flwr list command (ls).

## DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION

To view the logs of the running task the RUN ID is supplied to the log command. Such as in the following example:

```
flwr log 11693208818826240040 . superlink-deployment
```

These commands are available for previous deployments when the superlink has been started with a state database using the options  
"--database=path/filename.db".

This allows an operations team member to access the list of previous runs for a given superlink node. Note that if the state database file grows too large the file may need to be removed so that it is recreated and unused logs are purged.

Note that the end user should leverage tools such as tensorboard or logging metrics to CSV files for access to historical training performance values such as loss, accuracy or other metrics. This can be achieved in the custom server application code.

## 5.6 References

- [1] P. Marendy, "National Infrastructure for Federated Learning in Digital Health (NINA) Flower Federated Learning Framework," Queensland Cyber Infrastructure Foundation Ltd, QLD, Requirements 1, Mar. 2025.

## 5.7 Appendix 1. Supernode Docker Build Files

The supernode build files assume the systemd configuration is available in an adjacent file system as these are copied into the build area during the procedure (refer to section 4.1.3.3.1).

### 5.7.1 Docker build file for GPU enabled cuda image.

The following docker file assumes a GPU pass through deployment and uses the nvidia-cuda base container. The image it produces is quite large due to the cuda dependencies.

```
FROM nvidia/cuda:12.9.1-cudnn-devel-ubuntu24.04 as build_supernode
SHELL ["/bin/bash", "-c"]
RUN groupadd fl_system
RUN useradd -r -s /bin/false supernode
RUN usermod -a -G fl_system supernode
WORKDIR /home/supernode
RUN chgrp -R fl_system /home/supernode
RUN chmod -R g+w /home/supernode
WORKDIR /opt/python/
RUN chgrp -R fl_system /opt/python
RUN chmod -R g+w /opt/python
RUN apt update
RUN apt upgrade -y
RUN apt install python-is-python3 python3-pip python3.12-venv -y
RUN which python
RUN python -m venv flower_venv
WORKDIR /opt/python/flower_venv
ENV PATH="/opt/python/flower_venv:$PATH"
RUN . bin/activate
RUN /opt/python/flower_venv/bin/pip --timeout=60 install flwr flwr-
```

**DRAFT – IN CONFIDENCE – NOT FOR DISTRIBUTION**

```

datasets[vision]
RUN /opt/python/flower_venv/bin/pip --timeout=60 install tensorflow==2.17
RUN /opt/python/flower_venv/bin/pip --timeout=60 install tensorflow-
probability==0.24.0
RUN /opt/python/flower_venv/bin/pip --timeout=60 install torch==2.2
RUN /opt/python/flower_venv/bin/pip --timeout=60 install torchvision==0.17
RUN /opt/python/flower_venv/bin/pip --timeout=60 install torchaudio==2.2
RUN /opt/python/flower_venv/bin/pip --timeout=60 install pandas scipy
scikit-learn seaborn matplotlib
#
FROM build_supernode as supernode_container
# Volume to hold configuration for the supernode
VOLUME ["/etc/flwr/supernode"]
# Volume for data directory accessible within scripts running in the
container.
VOLUME ["/media/data"]
WORKDIR /opt/flwr/supernode
COPY ./systemd_configs/opt/flwr/supernode/run_supernode.sh
/opt/flwr/supernode/
# run supernode as the supernode user
USER supernode
# the entrypoint is the script run supernode.
# in docker we don't use systemd inside the container.
ENTRYPOINT ["bash", "run_supernode.sh"]

```

The image can be built using a script similar to (assuming systemd config tree is two directories above the working folder in this example):

```

#!/bin/bash

cp -r ../../systemd_configs ./
docker build -t supernode_cuda:v1.0 .
rm -rf systemd_configs

```

**5.7.2 Docker build file for CPU only.**

The following docker file assumes the supernode will run in a CPU only environment and uses the ubuntu base image. It results in a smaller docker image size.

```

FROM ubuntu:noble as build_supernode_cpu
SHELL ["/bin/bash", "-c"]
RUN groupadd fl_system
RUN useradd -r -s /bin/false supernode
RUN usermod -a -G fl_system supernode
WORKDIR /home/supernode
RUN chgrp -R fl_system /home/supernode
RUN chmod -R g+w /home/supernode
WORKDIR /opt/python/
RUN chgrp -R fl_system /opt/python
RUN chmod -R g+w /opt/python
RUN apt update
RUN apt upgrade -y
RUN apt install python-is-python3 python3-pip python3.12-venv -y
RUN which python

```

**DRAFT - IN CONFIDENCE - NOT FOR DISTRIBUTION**

```
RUN python -m venv flower_venv
WORKDIR /opt/python/flower_venv
ENV PATH="/opt/python/flower_venv:$PATH"
RUN . bin/activate
RUN /opt/python/flower_venv/bin/pip --timeout=60 install flwr flwr-
datasets[vision]
RUN /opt/python/flower_venv/bin/pip --timeout=60 install tensorflow==2.17
RUN /opt/python/flower_venv/bin/pip --timeout=60 install tensorflow-
probability==0.24.0
RUN /opt/python/flower_venv/bin/pip --timeout=60 install torch==2.2.1 --
index-url https://download.pytorch.org/whl/cpu
RUN /opt/python/flower_venv/bin/pip --timeout=60 install torchvision==0.17.1
--index-url https://download.pytorch.org/whl/cpu
RUN /opt/python/flower_venv/bin/pip --timeout=60 install torchaudio==2.2.1
--index-url https://download.pytorch.org/whl/cpu
RUN /opt/python/flower_venv/bin/pip --timeout=60 install pandas scipy
scikit-learn seaborn matplotlib
#
FROM build_supernode_cpu as supernode_container_cpu
# Volume to hold configuration for the supernode
VOLUME ["/etc/flwr/supernode"]
# Volume for data directory accessible within scripts running in the
container.
VOLUME ["/media/data"]
WORKDIR /opt/flwr/supernode
COPY ./systemd_configs/opt/flwr/supernode/run_supernode.sh
/opt/flwr/supernode/
# run supernode as the supernode user
USER supernode
# the entrypoint is the script run supernode.
# in docker we don't use systemd inside the container.
ENTRYPOINT ["bash", "run_supernode.sh"]
```

The image can be built using a script similar to (assuming systemd config tree is two directories above the working folder in this example):

```
#!/bin/bash

cp -r ../../systemd_configs ./
docker build -t supernode_cpu:v1.0 .
rm -rf systemd_configs
```