Autonomous and Self-organized Artificial Intelligent Orchestrator for a Greener Industry 4.0

Deliverable

D3.2 NG-SDN & Distributed Intelligence Functions Toolkit

Actual submission date: 23/12/2023

Funded by the European Union

Project Number:	101070181		
Project Acronym:	TALON		
Project Title:	Autonomous and Self-organized Artificial Intelligent Orchestrator for a Greener Industry 4.0		
Start date:	October 1st, 2022	**Duration:**	36 months
D3.2 NG-SDN & Distributed Intelligence Functions Toolkit			
Work Package:	WP3		
Lead partner:	UBITECH (UBI)		
Author(s):	Petros Petrou (UBI); Ioannis Pastellas (UBI); Sophia Karagiorgou (UBI)		
Reviewers:	Foivos Psarommatis Giannakopoulos (UPV); Arcadio Garcia (EXOS)		
Due date:	31/12/2023		
Deliverable Type:	DEM - Demonstrator, pilot, prototype	**Dissemination Level:**	PU
Version number:	1.0		

Revision History

Version	Date	Author	Description
0.1	01/12/2023	UBI	ToC release
0.2	15/12/2023	UBI	Contributions in all sections
0.3	19/12/2023	UBI	1st draft ready – sent for internal review
0.4	20/12/2023	UPV, EXOS	Deliverable reviews with embedded comments
0.5	20/12/2023	UBI	Comments addressed; sent for quality review
0.6	22/12/2023	UBI	Addressed comments from quality review; sent to Coordinator
1.0	23/12/2023	ENG	Final coordinator review before submission

Table of Contents

List of Figures

List of Tables

Non è stata trovata alcuna voce dell'indice delle figure.

Definitions and acronyms

ACLs	*Access Control Lists*
API	*Application Programming Interface*
CA	*Consortium Agreement*
CNI	*Container Network Interface*
DAG	*Directed Acyclic Graph*
DoA	*Description of Action*
DSL	*Domain Specific Language*
DPDK	*Data Plane Development Kit*
eBPF	*extended Berkeley Packet Filter*
EC	*European Commission*
EU	*European Union*
E2C	*Edge-to-Cloud*
GA	*Grant Agreement*
JIT	*Just-In-Time*
KSM	*Kube-state-metrics*
LSTM	*Long Short-Term Memory*
ML	*Machine Learning*
NFV	*Network Functions Virtualisation*
NLAs	*Node Level Agents*
NN	*Neural Network*
OVN	*Open Virtual Network*
OVS	*Open vSwitch*
PPE	*Personal Protection Equipment*
PC	*Project Coordinator*
RMSE	*Root Mean Square Error*
QoS	*Quality of Service*
NG-SDN	*Next Generation Software Defined Network*
SLOs	*Service Level Objectives*
TC	*Technical Coordinator*
UCs	*Use Cases*
WP	*Work Package*

Disclaimer

Executive Summary

The goal of the deliverable is to report on the Distributed Intelligence Functions Toolkit bridging edge to cloud networking mechanisms by using Next Generation Software Defined Networks (NG-SDN). The term “Distributed Intelligence Functions” refers to the functionalities of the network orchestration framework that relate to a) in-line processing of data collected from the data plane, b) off-line processing of data collected from the data plane, c) traffic replication for further analysis, and d) network policies enforcing decisions at the control plane which are made in the context of distributed intelligence.

The data plane is constructed among the nodes of the cloud-native applications that are deployed from the orchestration engine. Besides, it relates with network policies for enforcement to materialise controls onto the control plane of the TALON framework. In this way, the distributed intelligence functions toolkit provides the only way to harvest data in the data plane translated onto policies / controls onto the control plane.

A distributed application is composed by a set of containerised pods that communicate with other pods or externally (authenticated) entities. A pod is the smallest deployable unit of computing that a developer may create and manage in Kubernetes. A pod componentises a module, a function, a script or any other self-sufficient application into a container that can be deployed and shipped independently into any distributed infrastructure. The network management is performed through the application of specific policies that can be activated and deactivated in real time. These policies dictate the ‘triggering conditions’ and the ‘proper enactment’ that has to be performed in case these conditions are satisfied. Both triggering conditions and actions may refer to an established data plane; that a policy may include some boolean expressions on top of the networking structures of Layer 3, 4 and 7 of the traditional OSI model [6] (i.e., pattern matching rules). On the other hand, enactment could refer to traffic-dropping, traffic-allowance, or traffic-tagging activities. The set of policies that are activated materialise the control plane of the TALON framework in the support of runtime adaptation and zero-touch orchestration of pods and containerised applications in the frame of distributed networking mechanisms.

In the context of TALON which both covers edge and cloud pods, policies are decomposed in conditions and actions. Conditions and actions per se are furtherly decomposed on Layer-3, Layer-4 and Layer-7 conditions and actions. These level-specific conditions and actions are concretely materialised in executable / binary artifacts that are created based on the hardware substrate that supports the deployment of the virtualised components. TALON will support general purpose network interfaces that support eBPF-based [7] acceleration and more niche that support programmability and acceleration. The translation process between the actions / policies and the binary artifacts is performed in the logically centralised part of the TALON orchestrator which supports the distributed intelligence functions considering the conditions of edge and cloud pods. The produced models are deployed through specific control-plane signalling by the Pod or Application-Level Agents.

1 Introduction

This section introduces the deliverable and explains its overall scope within the TALON projects and objectives. It also documents the positioning of the deliverable in the TALON project, namely the relation of the current deliverable with the other deliverables, tasks, and work-packages, mapping with the user requirements and technical specifications regarding the Next Generation Software Defined Network (NG-SDN) component and how the knowledge produced in the other deliverables and work-packages served as input to the current deliverable.

1.1 Scope and Objectives

The aim of this deliverable is to report on the early functionalities of the Next Generation Software Defined Network (NG-SDN) component (i.e., methods develop in the support of Distributed Intelligence, as well as methods, rules, and conditions for the Networking Mechanisms) of TALON. The Distributed Intelligence and NG-SDN aim to ensure the secure integration and functioning of a uniform network overlay across the edge and cloud nodes. We introduce the underlying (sub-) components, methods, and software modules in the support of NG-SDN which have been implemented in our tools and are used to build the Control Place.

We have thoroughly analysed existing network policy models and languages and compared them to the functional and non-functional requirements that need to be achieved in TALON in the frame of D2.1 and D3.1. The aim is to derive the correct definitions of the operational policies in the support of distributed intelligence between edge and cloud deployed pods, and within the overall edge-to-cloud (E2C) ecosystem. The expected outcome is to keep the whole process agnostic to the hardware and software specifications by utilising a layer of abstraction between a domain-specific language and the actual processing engine which is the TALON NG-SDN component.

1.2 Relation to other work packages, tasks, and deliverables

Task 3.2 uses valuable insights from D2.1 and D3.1 to design and develop the Distributed Intelligence and Network Policies starting from the functional requirements, ensuring it aligns with the use cases' objectives, success metrics and overall TALON architecture and specifications. The Deliverables D2.1 (defining the user requirements) and D3.1 (defining the TALON Conceptual Architecture and componentisation) are a crucial input for D3.2, because they identify supported Use Cases (UCs), determine the platform's architecture, requirements, stakeholders, and integration points between components, laying the foundation for the project's technical stages and helps partners understand the interplay between UC scenarios and the technical activities in T3.2.

Additionally, this deliverable will contribute to D3.6 (M33), which focuses on validating and demonstrating TALON's overall architecture. This process involves establishing communication among components and transforming the architecture into a functional and useful system.

1.3 Document Structure

The remainder of this document is organised as follows:

Section 1 – Introduction (i.e., this section) introduces the deliverable and explains its overall purpose.

Section 2 – presents the definition of the network policies and how the networking mechanisms enable the smooth interplay among the pods (coming both from the edge and cloud) and the interpretation of policies into actions.

Section 3 – presents the streams of processing and the policy translation mechanisms by considering the data which are travelling within the TALON ecosystem.

Section 4 – reports on the current development activities and the early prototype deployed in the TALON infrastructure in the support of NG-SDN programmability, and dynamic kernel policies enforcement for covering the required lifecycle for abundant observability, tracing, and networking.

Section 5 – Conclusion and future outlook section concludes the deliverable. It outlines the main findings of the deliverable that will guide the next steps, enhancements, future research, and technological efforts of the consortium.

2 Network Policy Management

2.1 NG-SDN and Distributed Intelligence Conceptual Architecture

The role of NG-SDN and Distributed Intelligence in TALON is to secure onboarding, and operation of a consistent overlay network among the edge and cloud nodes and the selection of a cluster-head which will cover the physical deployment of the TALON Use Cases and applications. This component will be responsible to generate and populate network metrics to feed AI Models. The results derived from the AI Model training will serve to meet specific Quality of Service (QoS) (e.g., high availability) and Service Level Objectives (SLOs) (e.g., delay <5ms). In D3.2, we report on the technical design, the configuration of the overlay network, the configuration and stream processing of the policies and first results about the analytics on top of the TALON NG-SDN infrastructure that we deployed.

The high-level conceptual architecture of the NG-SDN and Distributed Intelligence component is depicted in Figure 1. The core functionality of the component is to manage virtual overlay networks between edge and cloud. The input of the component is Network Rules that update routing or replacing a pod in the cluster on runtime by means of policies that are detailed in the following sections. The component feeds itself with new measurements to optimise the deployments based on the target QoS and the SLOs to be met.

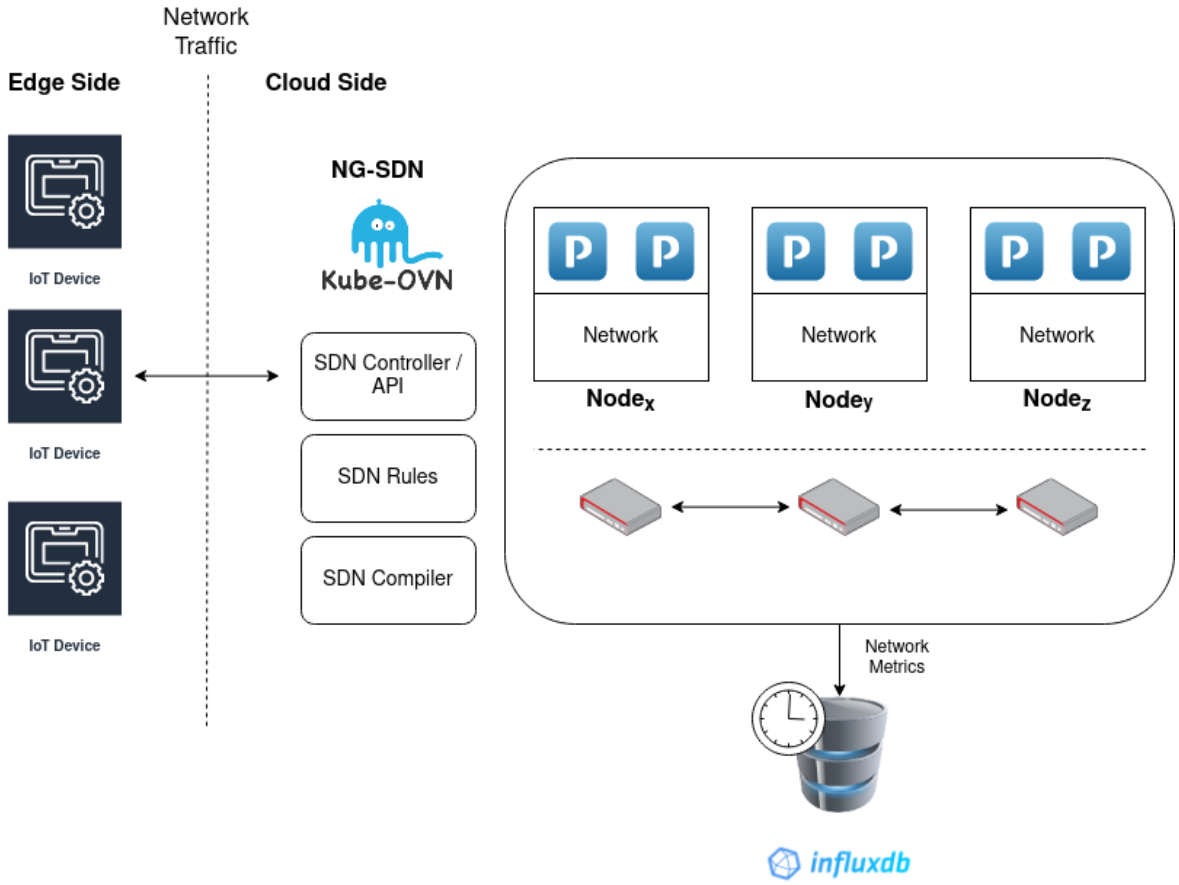

Figure 1. NG-SDN and Distributed Intelligence Conceptual Architecture.

2.2 Overview of Policies

A crucial facet of cloud-native applications management is the way network traffic is intercepted, processed, and handled. TALON will deliver a holistic SDN framework targeting cloud-native deployments in modern orchestration engines. Moreover, it will deliver a programmable control plane and data plane that will ensure that several sophisticated features can be provided seamlessly to the deployed edge-to-cloud applications.

In the edge-to-cloud ecosystem, the traffic that flows between nodes, edge devices, pods, and switches of the running application sockets constitutes the so-called data plane. Data plane refers to the functional layer responsible for the actual transmission of data packets across a network. Data plane is constructed through programmability of cyber-physical devices that span from virtual / physical switches and SDN controllers. It is worth mentioning / clarifying that the network policies can include pods, but the actual processing happens at the lower level of the infrastructure such as the Container Network Interface (CNI), nodes, switches etc. Control plane consists of all systemic components that manage and process these scripts.

TALON utilises network policies through the definition of arbitrary rules that constrain specific attributes such as network traffic, packet-flows, sessions network-aware placement. Policies are built based on an abstract Domain Specific Language (a.k.a. DSL) to create Layer-3, Layer-4, and Layer-7 conditions that will result in specific actions. These actions can be allowance, denial, forwarding / replication, or rerouting, placement and managing Quality of Service (QoS).

Node Level Agents (NLAs) will be developed and deployed in the edge-cloud infrastructure to act as a liaison between the central orchestrator and the individual nodes, overseeing tasks such as resource allocation, load balancing, and fault tolerance. By monitoring the health and status of each node, it contributes to the overall resilience, scalability, performance optimisation, and resource allocation of individual nodes within the distributed cloud network. The latter enables to support runtime adaptations to the monitored nodes.

2.3 Interpretation Flow

TALON network policy follows a specific execution process based on the available hardware resources. Each node (cloud or edge device) that is registered in the TALON control plane is able to accept network policies that are tailored to the profile of the hardware resources that are announced from the Node Level Agent (NLA).

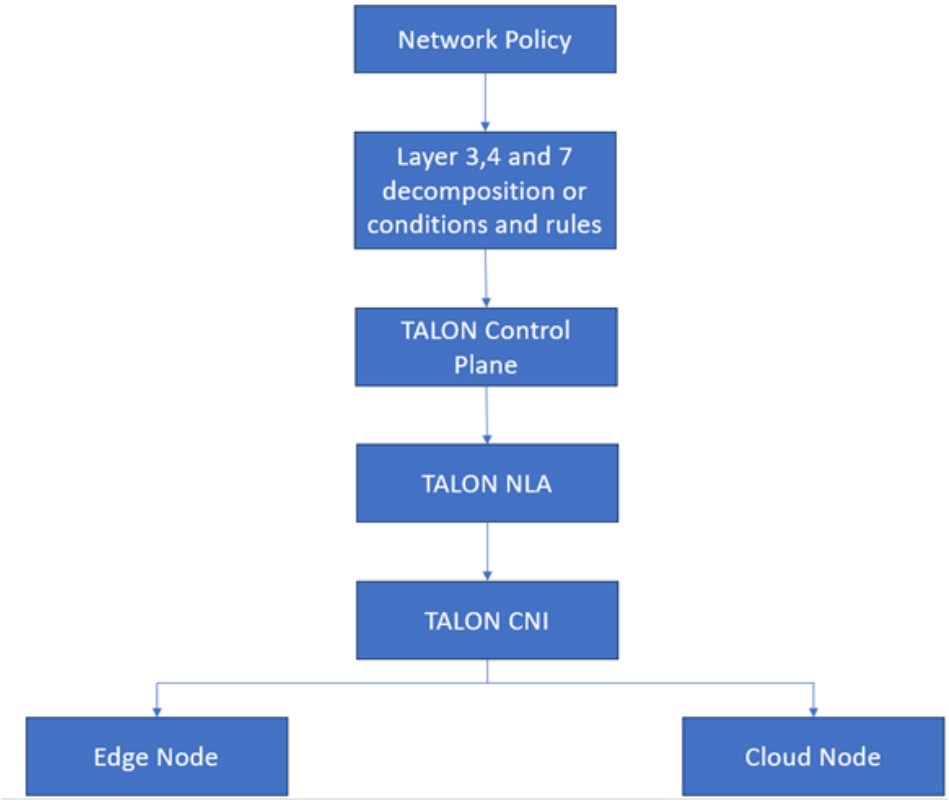

Figure 2. The TALON Control Plane.

As depicted in Figure 2, the translation process is the same for edge and cloud nodes. It should be noted that at the end of the interpretation process a proper instruction set is created for the control plane agent.

First and foremost, in TALON, policy conditions and actions are expressed in a common and abstract way. To this end, TALON policy will inherit and extend the Open Virtual Network (OVN), an open-source virtual networking project, to create and manage these virtual switches. These switches operate at the overlay level, enabling seamless connectivity between pods deployed across the edge-to-cloud infrastructure they are on the same logical network. The logical network is defined by a cloud / network administrator. This abstraction is essential for the flexibility and scalability of containerised applications within Kubernetes.

The virtual switches enable the enforcement of network policies, load balancing, network-based placement, and efficient routing of traffic between pods. The orchestration of these virtual switches by the TALON SDN Controller ensures that communication between pods remains secure, performant, and aligned with the specified networking configurations, contributing to the overall robustness of the Kubernetes networking environment.

The TALON's SDN Controller will adopt the P4 DSL formalism [1] (i.e., defining how network packets are processed independently of the underlying hardware and protocols) for layer 3, 4 and 7 expressivity. The interpretation flow will result in an updated YAML file that will be executed in the Kubernetes context to support runtime adaptations and at the same time will be as generic as possible.

3 Distributed Intelligence Network Functions and Management

The role of the Distributed Intelligence and SDN is to secure onboarding, and operation of a consistent network overlay among the edge and cloud nodes and the selection of a cluster-head which will represent an entire physical deployment. The purpose of the network overlay is to:

- Populate network metrics to feed an AI Model Training for Service Level Objectives (SLOs) adaptations through a Smart Policy Manager.
- Manage virtual ports, IP addresses and pods communication.
- Manage virtual overlay networks.
- Monitor, formulate and dimension the networking metrics of the federated learning methods and edge AI model training to reduce the information exchange overhead based on smart policies (e.g., energy- or mobility- aware).
- Take part in the selection process of a cluster-representative (cluster-head) which will be used to offload several computational tasks.

Semi-centralised techniques will be used for cluster-node selection. Each node is not statically configured with the address of the super-node. Instead prior to the joining protocol, the node is executing a cluster-head election protocol. The goal of such protocols is to logically split the network topology to several ‘islands’; each of which is represented by a cluster-head. This is performed using a combination of controlled flooding schemes initiated randomly by volunteering-nodes based on their capabilities (e.g., battery, memory, etc). Nodes within a specific proximity that receive these broadcasts and have no super-node, accept the cluster-head as their super-node. Hence, during joining and leaving, they require their ‘topological position’ from the elected node.

3.1 Automated Network Management and Policies

Network Functions Virtualisation (NFV) and Software Defined Networking (SDN) have changed the framework for the deployment of cloud-native services. On the one hand, vast amounts of data are produced every second by interconnected software and hardware services, and on the other hand data collection and data quality are taking a second role. Feature engineering algorithms should be applied in the modern SDN infrastructures. High-quality data ensure that SDN controllers can make precise adjustments to network configurations, optimise performance, and proactively respond to evolving conditions. Moreover, a well-crafted set of features not only enhances the accuracy of predictive models, but also facilitates intelligent decision-making within the dynamic network environment.

Software Defined Networking (SDN) and Machine Learning (ML) are two distinct but increasingly interconnected technologies that play a crucial role in the modern network management and optimisation. Machine Learning can play a pivotal role in automating network policies management and monitoring, by providing intelligent and adaptive solutions to the emerging challenges coming from edge-to-cloud applications. The applicability of automated decisions via network policies is made by analysing network traffic patterns. These patterns then enable to monitor metrics and thresholds and perform network optimisation, identify anomalies, perform predictive maintenance, and predict potential security threats in real-time. As an outcome, the interplay between network management mechanisms, machine learning and policies enable the availability, security, healthiness, and scalability of the edge-to-cloud ecosystem. This proactive approach enables the development of dynamic network policies that can autonomously adjust to evolving issues (e.g., load balancing when traffic bursts are detected), enhancing the overall resilience of the network infrastructure. Additionally, machine learning algorithms contribute to the creation of more sophisticated control mechanisms to fine-tune their network policies based on network / application / pod behaviour, infrastructure's environmental characteristics, and contextual information. The TALON SDN Controller will take ML

analysis results as input and then dynamically reconfigure the network to adapt to changing conditions or mitigate potential problems in the frame of runtime adaptations.

3.2 Policy Processing Stream

In a Kubernetes networking environment, policy processing typically involves defining and enforcing rules for communication between pods and nodes. These rules may specify which pods / nodes can communicate with each other, the protocols and ports allowed, load balancing and other network-related policies. In TALON, policy processing is the process of applying Network Policies, defined in the Control Plane, to the layer which is responsible for the actual forwarding of application / service / user data within the edge-to-cloud network, namely the Data Plane.

TALON SDN component will enforce network isolation, control traffic flow and network-aware placement within the Kubernetes cluster. TALON SDN component utilises the Open vSwitch (OVN) to implement its network fabric and leverages OVN's Access Control Lists (ACLs) to enforce Network Policy rules.

The policy processing mechanism in Kube-OVN [5] involves several steps:

1. **Policy Admission:** When a Network Policy resource is created or updated, Kubernetes compiles the policy to ensure it can be consumed by the Kubernetes Network Policy Application Programming Interface (API).
2. **Policy Translation:** The TALON SDN Controller translates the Network Policy resource into OVN ACL rules, which are the low-level constructs used to enforce traffic / network management.
3. **ACL Propagation:** The translated ACL rules are propagated to OVN data plane components, such as OVN switches and routers, where they are applied to the corresponding ports or logical switches associated with the pods or workloads.
4. **Traffic / Network Management:** When network flow or a packet enters the network, OVN data plane components review the applied ACL rules to determine, whether it should be managed in a certain way (i.e., allow or deny, rerouting, etc.).
5. **Policy Verification:** The TALON SDN Controller periodically verifies that the applied ACL rules accurately reflect the Network Policy resources and that the network is enforcing the intended policies.

4 Interpretation on top of eBPF-enabled E2C Native Networks

4.1 Utilisation of eBPF Data Planes

Kube-OVN is the core virtual networking framework of the TALON Data Plane that will be extended in the context of SDN and Distributed Intelligence component. As the default configuration, Kube-OVN employs the CNI chaining mode to augment its existing set of features.

Cilium [8] is an open source, cloud native solution for providing, securing, and observing network connectivity between workloads, fuelled by the extended Berkeley Packet Filter (eBPF) which allows sandboxed programs to run within the operating system and add additional capabilities to the operating system at runtime. The operating system then guarantees safety and execution efficiency, as if natively compiled with the aid of a Just-In-Time (JIT) compiler and verification engine. This has led to a wave of eBPF-based projects covering a wide array of use cases, including next-generation networking, observability, and security functionality.

By integrating Cilium, Kube-OVN users can have richer and more efficient policies. The only prerequisite is the Linux kernel (above version 4.19) for full eBPF capability support. Figure 3 depicts the Kube-OVN architecture after the integration with the Cilium.

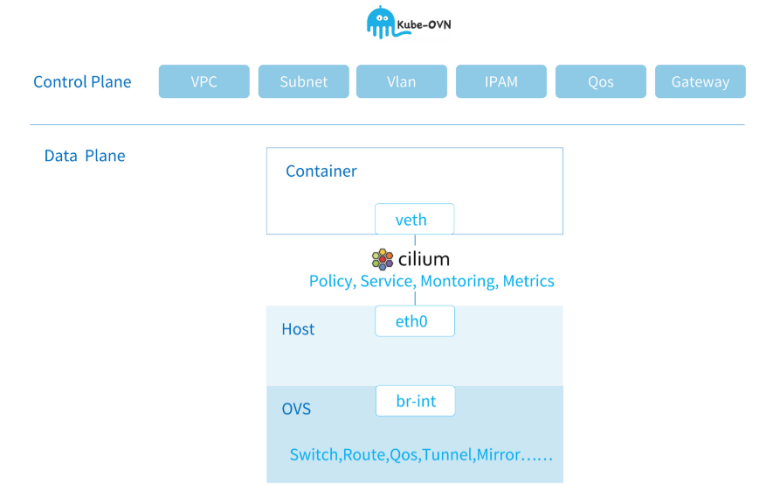

Figure 3. Integration of Cilium with Kube-OVN.

As already mentioned, the TALON's SDN Controller supports network policy capabilities to control the access of pods in the cluster by applying L3, L4 and L7 rules. Figure 4 and Figure 9 show a running example for L3 and L4 (protocol TCP) configuration rules that can be supported by the TALON SDN Controller and handle / block the traffic for the target pod.

```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "l3-rule"
  namespace: test
spec:
  endpointSelector:
    matchLabels:
      app: test
  ingress:
  - fromEndpoints:
    - matchLabels:
        app: dynamic
```

Figure 4. Cilium L3 Rule.

The test pod in the default namespace cannot access the destination pod, but the test pod to the destination pod in the test namespace is accessible. Figure 5 depicts the test results in the default namespace.

```
# kubectl exec -it dynamic-7d8d7874f5-9v5c4 -- bash
bash-5.0# ping -c 3 10.16.0.41
PING 10.16.0.41 (10.16.0.41): 56 data bytes

--- 10.16.0.41 ping statistics ---
3 packets transmitted, 0 packets received, 100% packet loss
```

Figure 5. Cilium L3 Rule – 100% packet loss.

We have carefully configured the namespaces in the TALON infrastructure to let pods interact successfully. Figure 6 shows the results in the test namespace.

```
# kubectl exec -it -n test dynamic-7d8d7874f5-6dsg6 -- bash
bash-5.0# ping -c 3 10.16.0.41
PING 10.16.0.41 (10.16.0.41): 56 data bytes
64 bytes from 10.16.0.41: seq=0 ttl=64 time=2.558 ms
64 bytes from 10.16.0.41: seq=1 ttl=64 time=0.223 ms
64 bytes from 10.16.0.41: seq=2 ttl=64 time=0.304 ms

--- 10.16.0.41 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.223/1.028/2.558 ms
```

Figure 6. Cilium L3 Rule – 0% packet loss.

If there is a network policy rule match, only the pod in the same namespace can access according to the rule, and the pod in the other namespace is denied access by default. To implement *cross-namespace* access, we specify the namespace information in the rule, as depicted in Figure 7.

```
# kubectl get cnp -n test  -o yaml l3-rule
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: l3-rule
  namespace: test
spec:
  endpointSelector:
    matchLabels:
      app: test
  ingress:
  - fromEndpoints:
    - matchLabels:
        app: dynamic
    - matchLabels:
        app: dynamic
        k8s:io.kubernetes.pod.namespace: default
```

Figure 7. Configure Cross-namespace L3 Rule Access.

The pod access in the default namespace, and the destination pod access is normal, as depicted in Figure 8.

```
# kubectl exec -it dynamic-7d8d7874f5-9v5c4 -n test -- bash
bash-5.0# ping -c 3 10.16.0.41
PING 10.16.0.41 (10.16.0.41): 56 data bytes
64 bytes from 10.16.0.41: seq=0 ttl=64 time=2.383 ms
64 bytes from 10.16.0.41: seq=1 ttl=64 time=0.115 ms
64 bytes from 10.16.0.41: seq=2 ttl=64 time=0.142 ms

--- 10.16.0.41 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.115/0.880/2.383 ms
```

Figure 8. Test L3 inter-pod Communication.

In a similar and explicit manner, we can configure access rules at the OSI [6] level 4 or 7 of the monitored application, as depicted in Figure 9.

```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "l4-rule"
  namespace: test
spec:
  endpointSelector:
    matchLabels:
      app: test
  ingress:
  - fromEndpoints:
    - matchLabels:
        app: dynamic
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
```

Figure 9. Cilium L4 Rule.

4.2 eBPF Data Plane Hooks

Data plane hooks play a crucial role in the field of networking, enabling the insertion of custom processing logic at different stages of data packet traversal within a network device. These hooks are essential for implementing functionalities such as packet filtering, traffic shaping, and security enforcement. Here, we explore the concept of data plane hooks, and how they contribute to the flexibility and extensibility of network architectures.

At its core, a data plane hook is a predefined point in the network data plane where external processing can be injected. These hooks act as gateways, allowing network engineers and developers to customise the behavior of the network device without altering its core functionality. By strategically placing hooks at specific stages of packet processing, such as ingress or egress points, one can exert fine-grained control over the flow of data through the network.

One common use of data plane hooks is in the context of packet filtering and inspection. By inserting hooks at the ingress point of a network device, administrators can implement access control policies, filtering out unwanted traffic based on predefined rules. This capability is fundamental for network security, enabling the identification and mitigation of potential threats before they traverse the network.

Traffic shaping and Quality of Service (QoS) enforcement are other areas where data plane hooks come into play. By placing hooks in the egress path of a network device, administrators can shape the outgoing traffic, prioritising certain types of data or limiting the bandwidth for specific applications. This level of control is essential for optimising network performance and ensuring a consistent user experience.

The advent of programmable data planes, such as those enabled by technologies like eBPF (Extended Berkeley Packet Filter), has significantly enhanced the capabilities of data plane hooks. These programmable hooks allow for the dynamic insertion of custom logic, giving network operators the flexibility to adapt their network behavior in response to changing requirements. This programmability is particularly valuable in dynamic and cloud-based environments, where the network's characteristics may evolve rapidly.

In conclusion, data plane hooks serve as essential points of extensibility and customisation within programmable network architectures. Their ability to allow the injection of custom logic at specific stages of packet processing empowers network administrators to tailor the behavior of their systems to meet specific needs, whether related to security, performance optimisation, or other requirements.

As networking technologies continue to evolve, the role of data plane hooks is likely to become even more central in creating agile, adaptive, and efficient network infrastructures.

The Data Plane Development Kit (DPDK) [9] offers a set of libraries and drivers that empower developers to accelerate packet processing workloads on a wide range of architectures. At its core, DPDK bypasses the traditional kernel networking stack, enabling direct access to network interfaces for optimised data movement. This approach enhances the efficiency and speed of data plane applications, making DPDK particularly well-suited for use cases such as cloud computing, and SDN. Kube-OVN with OVS-DPDK (Open vSwitch with Data Plane Development Kit) represents a powerful integration within the TALON's edge-to-cloud ecosystem.

OVS-DPDK is an extension of Open vSwitch that integrates with DPDK to accelerate packet processing in virtualised environments like Kubernetes. DPDK allows direct access to network interfaces, bypassing the traditional kernel networking stack and significantly increasing packets throughput and decreasing latency. Their combination enables efficient communication between containers and optimises the overall networking performance within an edge-to-cloud environment. The enhanced packet processing capabilities provided by OVS-DPDK contribute to the scalability and responsiveness of applications. Practically, this approach on data plane programmability adds zero overhead at the monitored pod / application, while it allows for processing and interpretation at the packet level in an accelerated manner.

4.3 TALON NG-SDN Programmability via Kube-OVN

Kube-OVN is a robust solution for Kubernetes networking plugin [16], providing robust and scalable networking solutions for edge-to-cloud ecosystem and cloud-native applications. It distinguishes itself from its counterparts (i.e., Kubernetes CNI mechanism like Calico, Flannel, etc.) by offering seamlessly integration of Open Virtual Network (OVN) thus offering a native overlay network that simplifies complex network configurations.

Kube-OVN's architecture, built on OVN, enhances its flexibility, efficiency, and ease of use, providing users with a comprehensive solution for managing networking policies, network elasticity, network-aware placement in Kubernetes environments. In essence, Kube-OVN acts as a connection between Kubernetes and OVN, merging established SDN with Cloud Native technologies. This implies that Kube-OVN not only enforces network standards within Kubernetes, such as CNI, Service, and Network Policy, but also introduces numerous SDN functionalities to the cloud-native environment. These functionalities encompass logical switches, logical routers, VPCs, gateways, QoS, ACLs, and traffic mirroring.

The technological architecture of the deployed NG-SDN infrastructure of TALON is depicted in Figure 10.

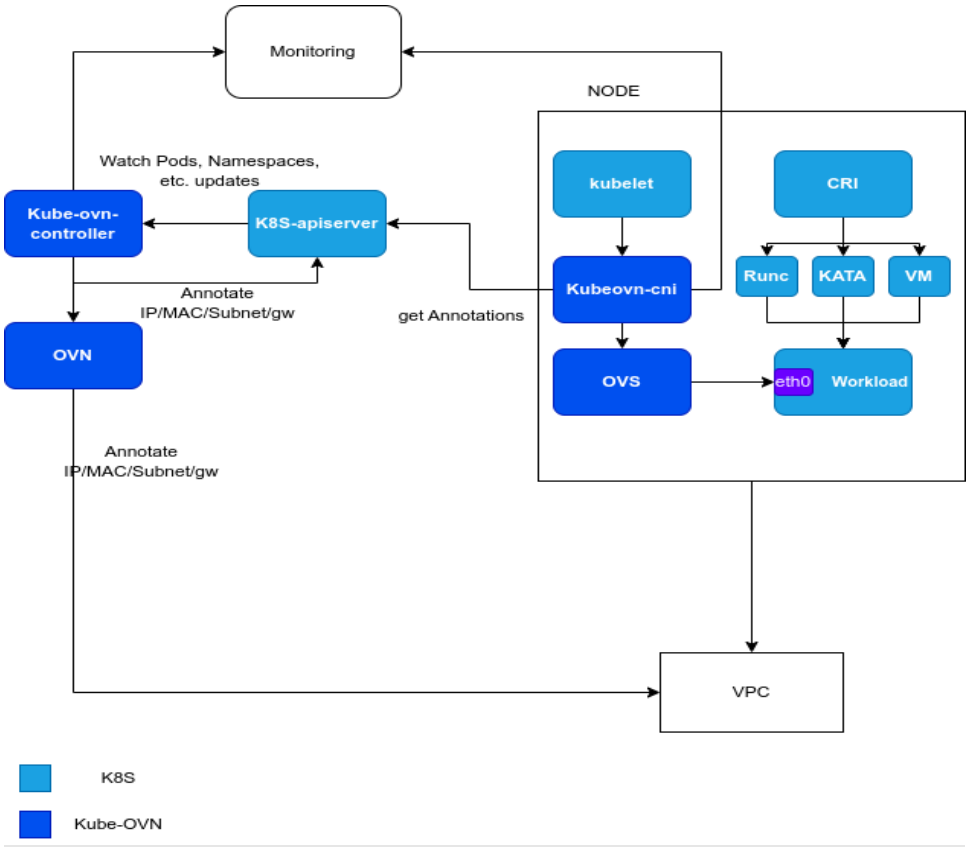

Figure 10. TALON NG-SDN Technological Architecture.

4.3.1 Dynamically Program the Kernel for Efficient Networking, and Observability

The TALON Kube-OVN core components that have been deployed and configured for inter-pod communication and monitoring, are as follows:

i. Ovn-central

- **ovn-nb:** Manages the storage of virtual network configurations and offers an API for virtual network administration. The primary interaction of kube-ovn-controller involves configuring the virtual network through ovn-nb.
- **ovn-sb:** Stores the logical flow table created from the logical network in ovn-nb, along with the current physical network state of each node.
- **ovn-northd:** Converts the virtual network information from ovn-nb into a logical flow table in ovn-sb.

ii. Ovn-ovs

- **ovs-ovn:** functions as a DaemonSet deployed on every node, featuring openvswitch, ovsdb, and ovn-controller operating within the pod. These elements serve as agents for ovn-central, facilitating the translation of logical flow tables into tangible network configurations.

iii. Kube-ovn-cni

This component, operating as a DaemonSet on each node, serves as a CNI interface implementation and manages the local Open vSwitch (OVS) for network configuration. The DaemonSet deploys the kube-ovn binary to each machine, facilitating interaction between

kubelet and kube-ovn-cni. This binary, located in the default /opt/cni/bin directory, communicates CNI requests to kube-ovn-cni.

kube-ovn-cni is responsible for configuring the specific network to handle various traffic operations. Its key tasks include:

a) Configuring ovn-controller and vswitchd.

b) Managing CNI Add/Del requests:
   - Creating or deleting veth pairs and binding or unbinding them to OVS ports.
   - Configuring OVS ports.
   - Updating host iptables, ipset, and route rules.
   - Dynamically updating network QoS.

iv. Kube-ovn-controller

- This component serves as the control plane for the entire Kube-OVN system by translating all Kubernetes resources into OVN resources. The kube-ovn-controller monitors events related to network functionality across various resources and updates the logical network within OVN based on changes in these resources. The primary resources monitored include:
  - Pod
  - Service
  - Endpoint
  - Node
  - NetworkPolicy
  - VPC
  - Subnet
  - VLAN
  - ProviderNetwork
- For instance, when a pod event occurs, the kube-ovn-controller listens for the creation event, assigns an address using the built-in in-memory IPAM function, and communicates with ovn-central to create logical ports, static routes, and potential ACL rules. Subsequently, kube-ovn-controller records details such as the assigned address, CIDR, gateway, route, etc., into the annotation of the pod. This annotation is then read by kube-ovn-cni, which utilises this information to configure the local network.

v. Monitoring

The monitoring lifecycle is activated through a set of daemon services, as depicted in Figure 11. We highlight here that we have successfully integrated all the metrics exposed by the node exporter of Prometheus [3] to an InfluxDB [4], which is a time-series database which ensures data persistency for a longer period of time. The deployed daemon services are as follows:

- **Kube-ovn-pinger:** This component is a DaemonSet running on each node to collect OVS status information, node network quality, network latency, etc.
- **Kube-ovn-monitor:** This component collects OVN status information and the monitoring metrics.
- **Node Exporter:** agent from Prometheus that exposes a wide variety of hardware- and kernel-related metrics, and thus gathers some Linux host metrics [3].
- **Kube-state-metrics:** Kube-state-metrics (KSM) is a straightforward service that monitors the Kubernetes API server and produces metrics regarding the status of objects (refer to examples in the Metrics section). Its primary emphasis is not on the well-being of individual

Kubernetes components but rather on the condition of diverse objects within, including deployments, nodes, and pods.

- **Kepler:** is responsible for exposing metrics related to energy consumption [2]. We have drawn inspiration from Cañete et. al. [15] to allow for energy aware E2C deployments within TALON.
- **InfluxDB:** InfluxDB, a time-series database is used for long-term persistence. The reason is that Prometheus only retains its data for 2 days. In the context of the TALON project, we monitor the application for a wider period to derive fruitful insights from the analytic services. The early analytic services are detailed in the following sections.

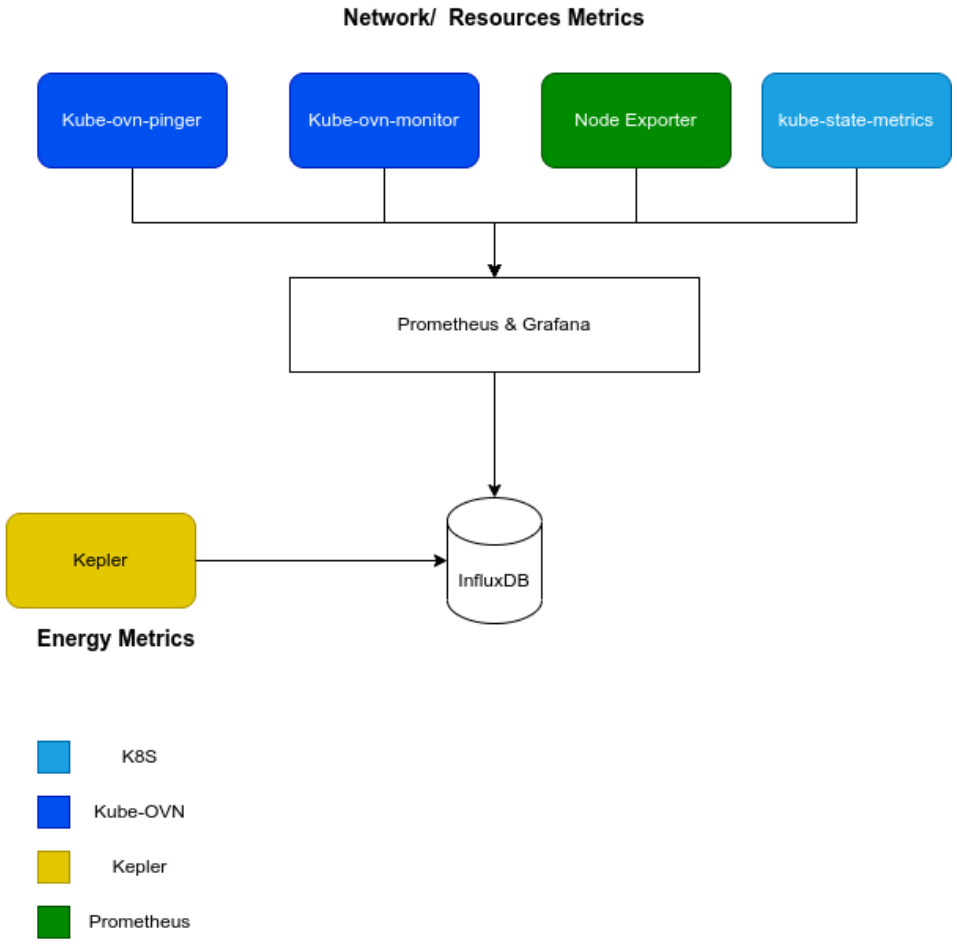

Figure 11. Monitoring Network and Other Metrics in TALON.

4.3.2 Example Monitoring Application

In the TALON context, the deployed applications are represented or conceptualised as Directed Acyclic Graphs (DAGs). Figure 12 depicts a custom application implemented as an indicative E2C example to collect the initial data and verify the functionalities of the frameworks that will be extended within the TALON's lifecycle. This example application is similar to the E2C TALON Use Cases and facilitate to progress the technical development of the NG-SDN functionalities and other components within the project.

We have deployed two producer applications that run on the edge and stream data to the RabbitMQ server [12] deployed on the cloud. Then a Spring boot [10] consumer retrieves the data from the message queue and stores them to a MongoDB server [11]. Moreover, the Spring boot application periodically receives HTTP requests and stores the results on a PostgreSQL server [13]. This can be considered as a typical application workflow that both handles online and offline data. In the rest of this document, we will focus on the online workflow of the application which is more challenging, and the AI algorithms can detect patterns in the generated time series.

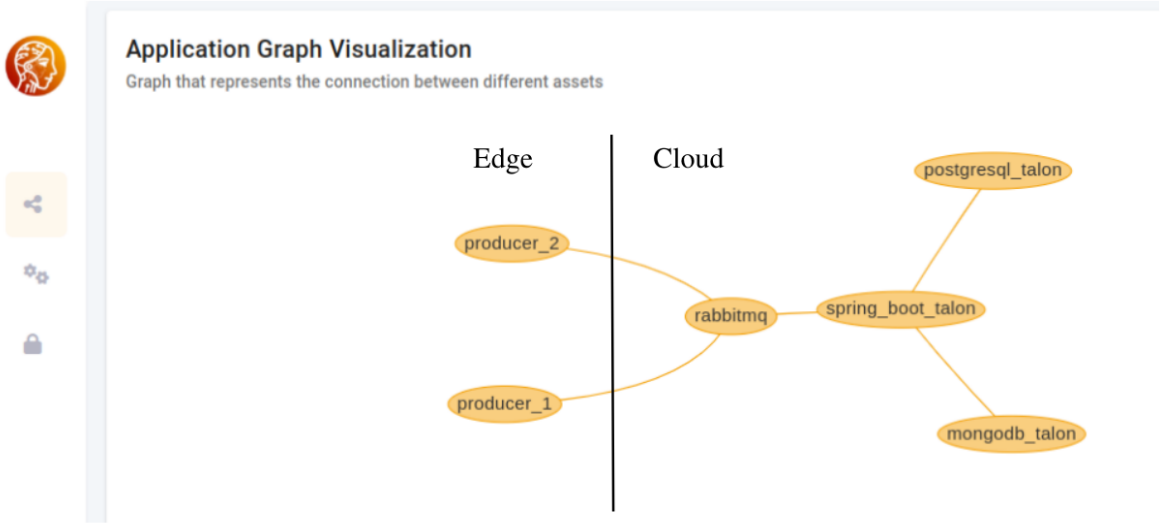

Figure 12. Example E2C Application in TALON.

The functionalities of the OVN are achieved in 2 ways. First, using the Kubernetes annotations and YAML files. The other way is by using the plugin **kubectl-ko,** which can commonly run some common operations using the CLI style.

An example of using YAML files can be seen in Figure 13, of creating a new subnet in the network. Also, in this YAML example the namespaces are declared where pods will be linked and created inside the given subnet. The commands show that a new switch is being added to the topology in relation to the new subnet. In this file, the configuration of the subnet is provided, including the CIDR block, gateway, excluded IPs, and the associated namespace(s) to realise the overlay network. This means that deploying a pod under one of these namespaces, the pod will automatically be assigned to the subnet.

```
cat <<EOF | kubectl create -f -
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: subnet1
spec:
  protocol: IPv4
  cidrBlock: 10.66.0.0/16
  excludeIps:
  - 10.66.0.1..10.66.0.10
  - 10.66.0.101..10.66.0.151
  gateway: 10.66.0.1
  gatewayType: distributed
  natOutgoing: true
  routeTable: ""
  namespaces:
  - ns1
  - ns2
EOF
```

Figure 13. Subnet Creation through YAML file.

Figure 14 depicts a logical topology of different subnets inside a single node. Due to the fact that over-commissioning of pods and services is being supported in the TALON cluster, a single node may host different subnets. The latter simulates the demand of real-world applications as compute resources are not requested simultaneously, so a single node may serve different subnets and pods.

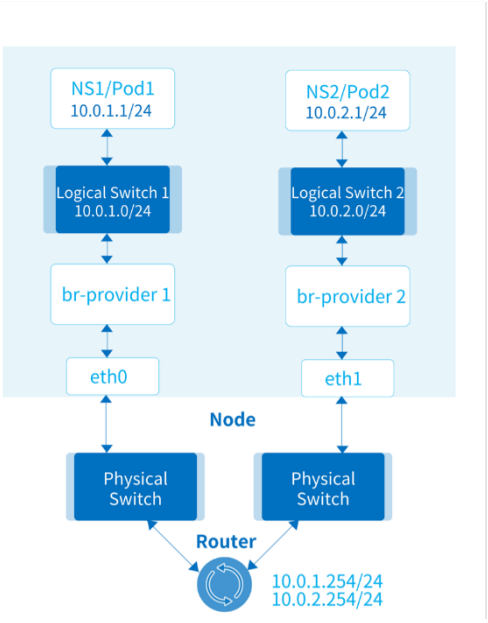

Figure 14. A Logical Topology of Different Subnets within the same Node.

Another operation on TALON is to manage QoS on pod-level, especially when combined with analytics that are detailed in the following sections. The TALON NG-SDN can provide a simple yet powerful way to intelligently manage QoS and the overall performance of the cluster utilising AI analytics. Again, this can be done with the help of YAML file (or annotations), as depicted in Figure 15. In this example, we configure ingress (receive) traffic and/or egress (transmitted) bandwidth of a pod in the “TALON” namespace using the deployed OVN capabilities. We have set to ingress traffic 1Mbits/s and to egress traffic 3Mbits/s, respectively.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: qos
  namespace: talon
  annotations:
    ovn.kubernetes.io/ingress_rate: "3"
    ovn.kubernetes.io/egress_rate: "1"
spec:
  containers:
  - name: qos
    image: docker.io/library/nginx:alpine
```

Figure 15. YAML File to configure Ingress and Egress Traffic.

4.3.3 *Analytics on top of TALON NG-SDN*

Using the TALON NG-SDN component, in this early prototype and demonstrator we trained two analytics categories. As the networking policies are linked with the cost of processing and therefore the CPU load, we performed one classification task for load class prediction; and one regression task for load value prediction in the next 60 seconds / 1 minute. These include both visual analytics of the raw metrics that are coming from the monitoring layer and analytics derived from the training of AI Models. Specifically, the analytics that are coming from the AI Models results are used to predict future values of specific objectives and metrics, such as CPU load usage, and classes of workloads.

The first part of the analytics is mostly referring to exploratory and visual analysis and are coming from the monitoring of the TALON-SDN node-level and pod-level behavior. Using a combination of Node Level Agents, we gather metrics that are being produced by the Kube-OVN, Kubernetes, and from the node-exporter of Prometheus [3]. In D3.2, we focus on network related metrics. In collaboration with T3.3, we also collect energy metrics of the cluster using Kepler [2] in the context of an energy-aware zero-touch orchestrator of TALON. Figure 16 illustrates network metrics collected for a given pod over the time. By using the example monitoring application, we observe that the bandwidth per pod, e.g. in this case MongoDB, and other network metrics can be collected and monitored by the Node Level Agents.

Figure 16. Grafana Dashboard displaying Network Metrics.

The second part of Analytics is generated by training AI Models that predict future trends and values of the collected metrics. More specifically, the first step of this module is to perform data analysis on the data resulting from the monitoring metrics. As described in the example monitoring application, to let the networking mechanisms take place and enforce policies, we have set up a Spring Boot Application with a collection of pods. This application has been modelled as a directed acyclic graph.

A visual exploration helps to explore the network metrics and identify their behaviour. Also, visual exploration helps us to determine a value threshold to be used as class discriminator. Figure 17, Figure 18, and Figure 19 illustrate the most important metrics that are coming from monitoring the example application. In this case, we filter the Spring Boot Pod, and we show the CPU Usage, RAM usage and Receive Bandwidth. Let us clarify here, that we also monitor all the network metrics of the TALON cluster, but we present some indicative examples.

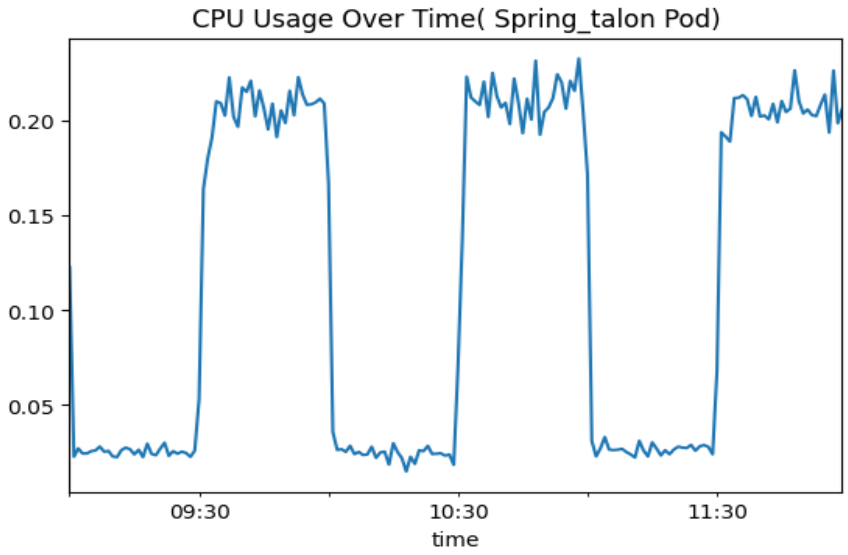

Figure 17. CPU Usage of Spring Boot Pod.

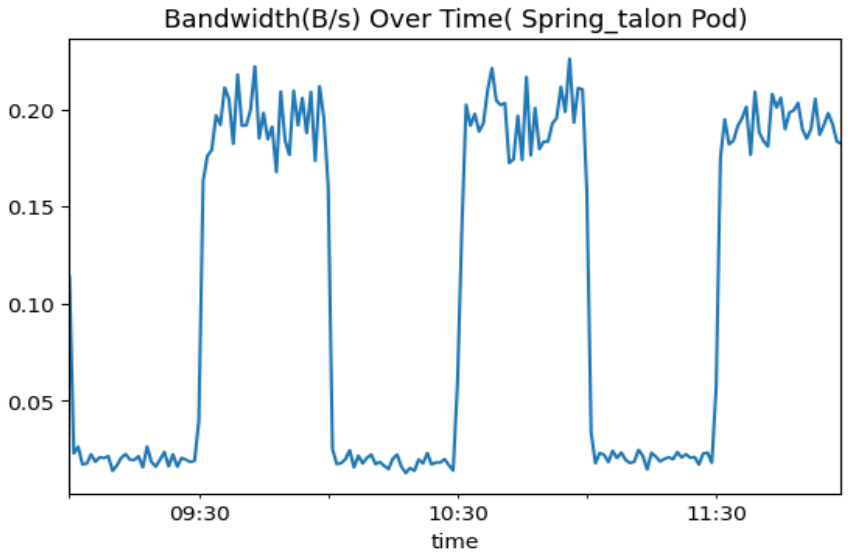

Figure 18. Bandwidth in Bytes of Spring Boot Pod.

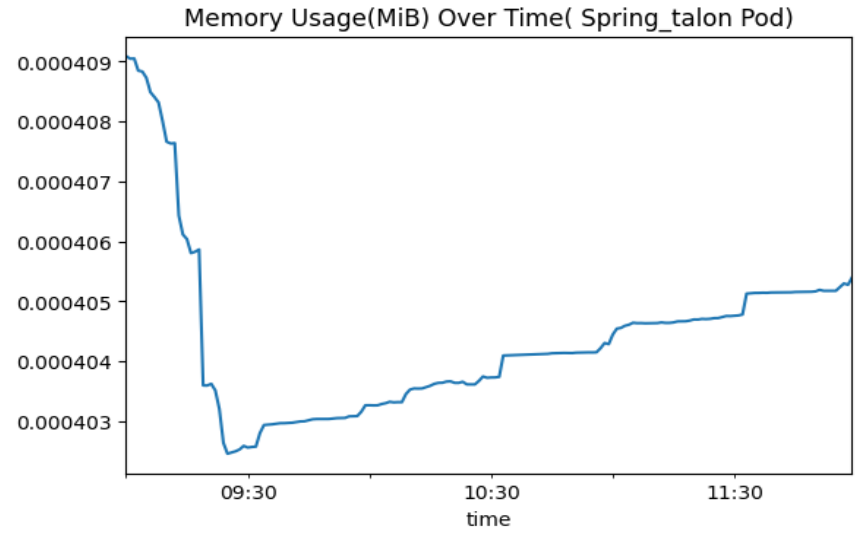

Figure 19. Memory (RAM) Usage of Spring Boot Pod in MiB (Mebibytes).

Next a correlation matrix is illustrated in Figure 20, to investigate the correlation between the network metrics. We have observed that when network traffic is increased in the cluster, this directly affects the CPU and RAM usage, as well as the bandwidth. The figure shows that there is a high and strong correlation of the CPU and RAM usage, and the receive bandwidth in the given pod. This is also evident from the above visualisation plots, where both CPU usage and receive bandwidth follow similar patterns. There is also a positive correlation (not that strong) with the memory usage. The results from correlation were expected, that is why these features are selected to model the CPU usage.

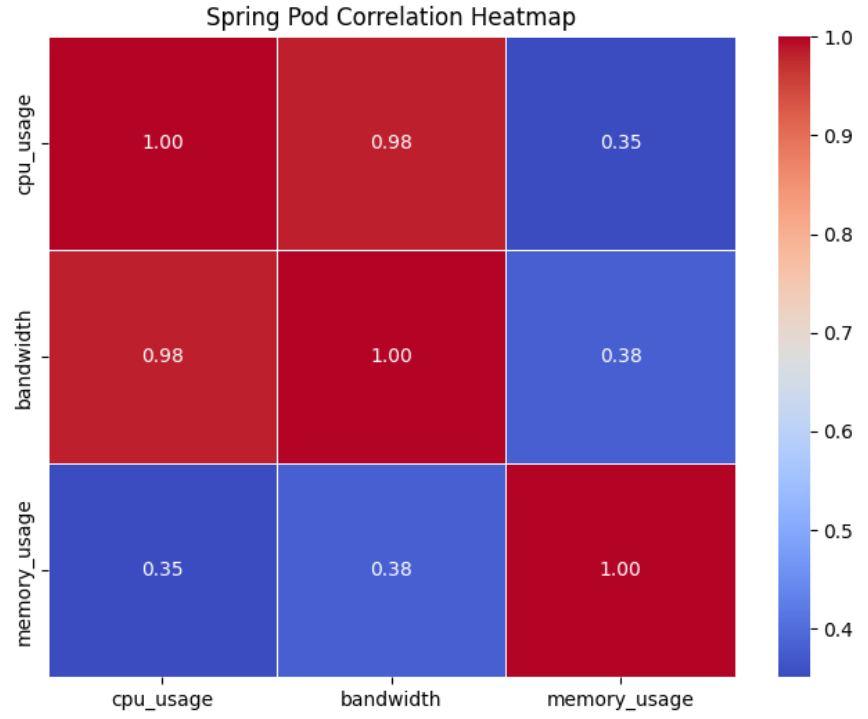

Figure 20. Correlation Matrix of 3 Variables.

This preliminary data analysis helps us capture the relationship between different metrics and use this information to create more accurate models.

4.3.3.1 Metric Prediction and Future Trends

This section describes the data analytic pipeline that we created to model and predict the future trend of the desirable metric (collected from the monitoring layer). By using the correlation measure, we performed feature importance and selected the most prominent features to train a Multivariate Time-Series model. After the feature selection, the data are processed into time windows of length k, indicating how many minutes the model will take in the past to compute its predictions. In the case of k=60 (i.e., 60 minutes or 1 hour), the model will be trained on sequences of past values of the features of size 60 and will be able to predict the future value of the next minute.

By performing this pre-processing step, we can then train a Neural Network (NN) with a Long Short-Term Memory (LSTM) architecture to capture time dependent patterns in the data and support strong prediction performance and robustness. Figure 21 illustrates the data analytic pipeline we set up and trained for the tasks of regression and classification.

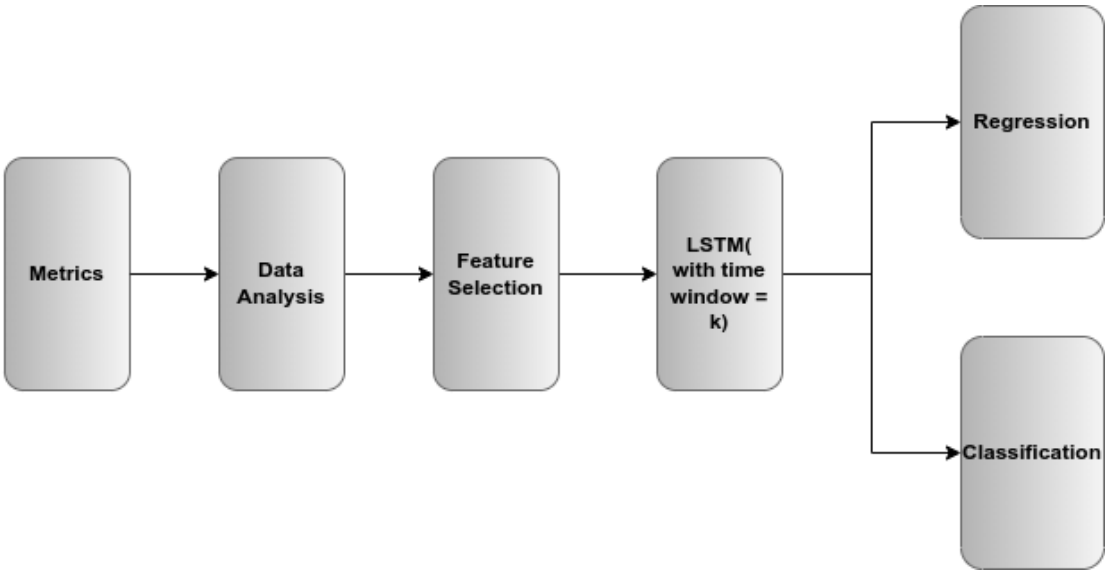

Figure 21. Analytic Pipeline for CPU Load.

4.3.3.2 CPU Load Prediction (Regression)

Using the above scheme of predicting future trends and values, we designed two models for predicting the next value of CPU usage (Regression) using the 3 features, as follows.

Input X:

- CPU Usage: Previous values of the CPU Usage (past hour).
- Memory Usage: Previous values of the Memory (RAM) Usage (past hour).
- Receive Bandwidth: Previous values of the of the bytes received rate (past hour).

Output y / Target variable:

- CPU Usage: value of the CPU Usage in the next minute.

Figure 22 displays the Neural Network architecture. It is a sequential model consisting of one LSTM layer (to capture time-dependent patterns of the data) followed by a linear layer of one neuron that is going to calculate the predicted value. The figure presents the summary of the NN architecture showing the type, output shapes and parameters of each layer in the model, as well as the model size.

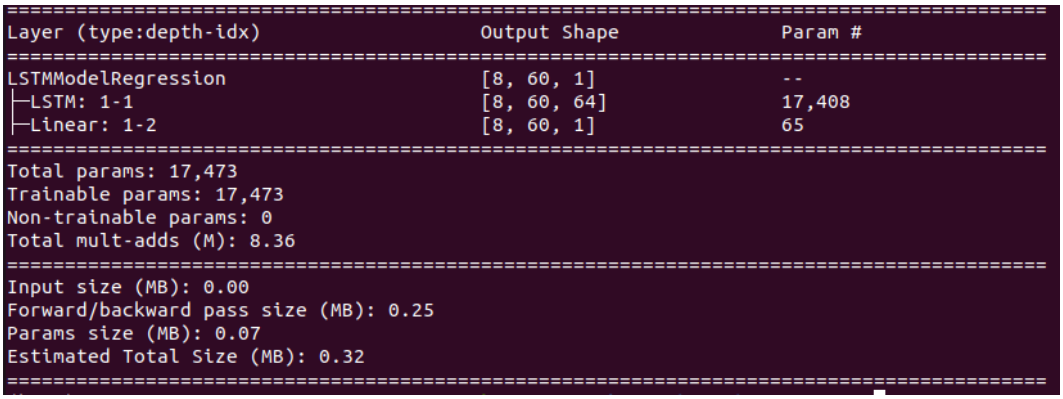

Figure 22. Neural Network Architecture for Regression Task.

Figure 23 shows the results of the prediction model. After 50 epochs, the AI model converges to low error, meaning that it can accurately predict future values of the target variable which is the CPU usage. The Root-Mean-Square-Error (RMSE) of the LSTM model is very low after 50 epochs. The RMSE formula is calculated, as follows:

$RMSE = \sqrt{(f-o)^2}$, where f = forecasts (expected values or unknown results), and o = observed values (known results).

The smaller the RMSE is, the better is the AI model performance. Note that both train and test RMSE are close, indicating the accuracy of the trained model over the training data.

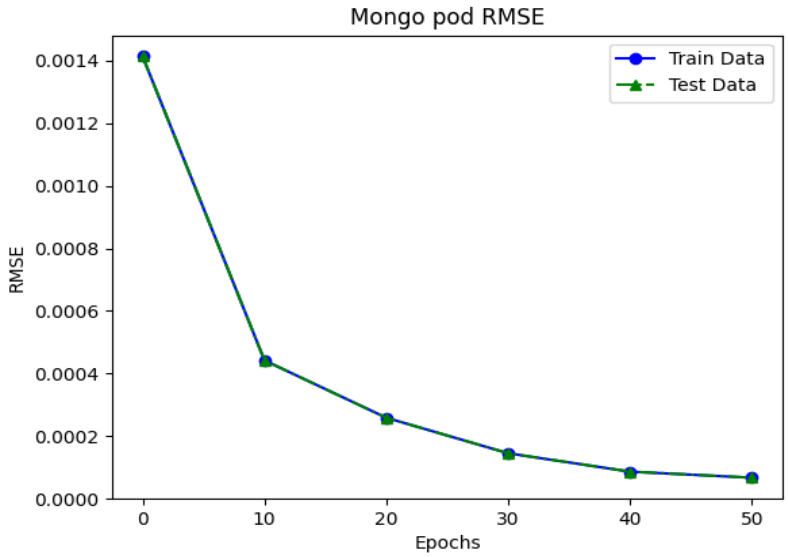

Figure 23. MongoDB Pod Prediction Accuracy using RMSE – Regression.

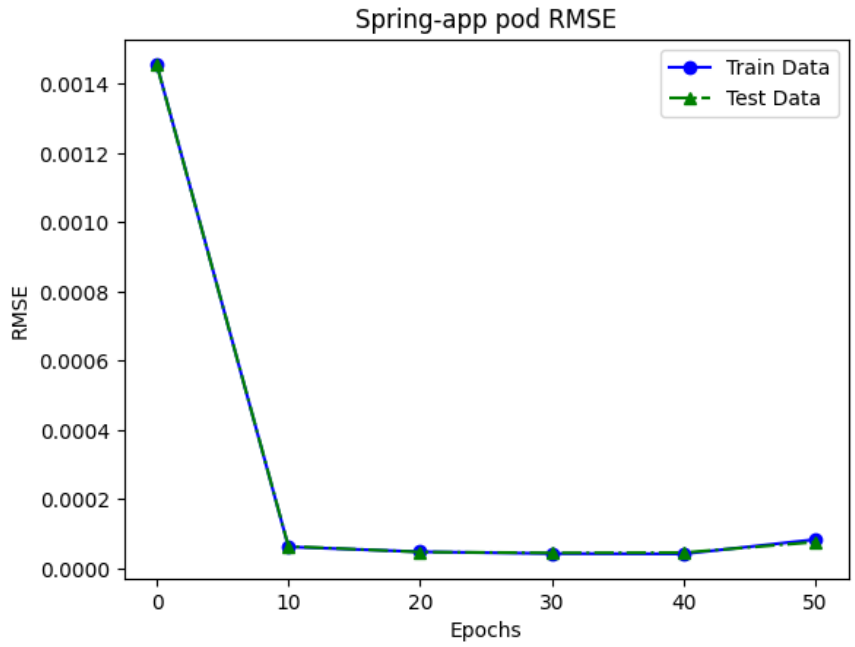

Figure 24. Spring Boot Pod Prediction Accuracy using RMSE – Regression.

In continuation to the Spring Boot Pod analysis, Figure 24 shows the convergence of CPU load using RMSE. The small RMSE indicates accurate model performance. The RMSE-epochs diagram depicts the performance evolution of a regression-based prediction model over successive training epochs. Since there are no labels, the only way to evaluate the performance is to use the target variable's actual values. So RMSE calculates the error between the raw values of model's prediction and the target value. As it can be seen, the y-axis represents the Root Mean Square Error, a metric that quantifies the model's prediction accuracy. Notably, the model has achieved commendable accuracy / performance even in the first epoch of training for both pods. However, fluctuations may be observed in the Spring Boot related diagram, and interpreting these fluctuations is crucial for optimising the AI

model. A stable descent in the RMSE signifies a robust and well-fitted model, while erratic changes might indicate overfitting or underfitting. Analysing the RMSE epochs diagram provides valuable insights into the model's learning process, aiding in the fine-tuning of parameters and the enhancement of the overall prediction performance.

4.3.3.3 CPU Load Prediction (Classification)

Also, using the same data analytic pipeline as depicted in Figure 21, we create a classification model to predict the future class of the load (i.e., binary classification: low or high load class). The class load may give us an indication to spawn new virtual machines when a high load is expected / predicted. The alteration needed from the previous regression model is to use sigmoid on the final linear layer and use a threshold of >0.5 to model the high load.

In addition to that, since there are raw and continuous values of the CPU usage, the class (low/ high load) should be defined to correctly perform the classification. The class is defined using the values of the metrics where there is spike (see spikes in Figure 17).

Input X:

- Memory Usage: Previous values of the Memory (RAM) Usage (past hour).
- Receive Bandwidth: Previous values of the of the bytes received rate (past hour).

Output y:

- CPU Usage: value (low/high load) of the CPU Usage in the next minute.

Figure 25 shows the architecture of the Neural Network. The figure depicts the summary of the architecture of the model (NN), displaying the type, output shapes and parameters of each layer in the model, as well as the model size. Eventually, it is the same architecture as the Regression Model described above. The difference is the forward pass of the network where the sigmoid function is used on the linear layer to predict a value between [0, 1]. A simple NN model was used, with a layer of LSTM neurons (to capture the time-dependent patterns), followed by a linear layer predicting the future class of the CPU load in this case.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
LSTMModelClassification                  [8, 60, 1]                --
├─LSTM: 1-1                              [8, 60, 64]               17,408
├─Linear: 1-2                            [8, 60, 1]                65
==========================================================================================
Total params: 17,473
Trainable params: 17,473
Non-trainable params: 0
Total mult-adds (M): 8.36
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.25
Params size (MB): 0.07
Estimated Total Size (MB): 0.32
==========================================================================================
```

Figure 25. Neural Network Architecture for Classification Task.

Similarly, Figure 26 and Figure 27 show the classification accuracy of CPU load (low/ high load) for the MongoDB pod and the Spring Boot pod of the TALON example application mentioned above. The figures show both training and testing accuracy. From the results, we can observe successful training as both training and testing accuracy are high and close around the 50 epochs.

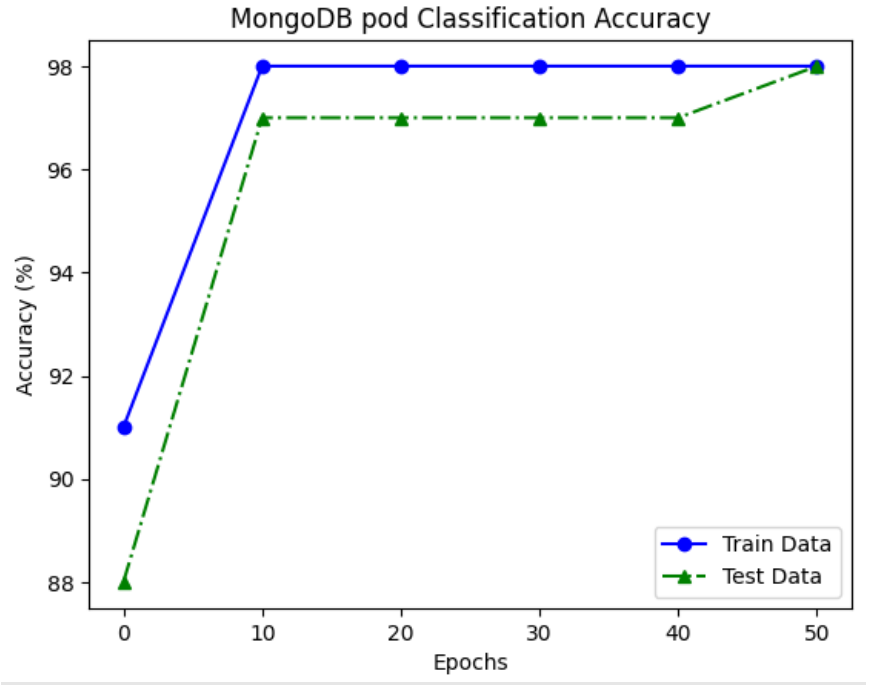

Figure 26. MongoDB Pod Load Class Prediction Accuracy – Classification.

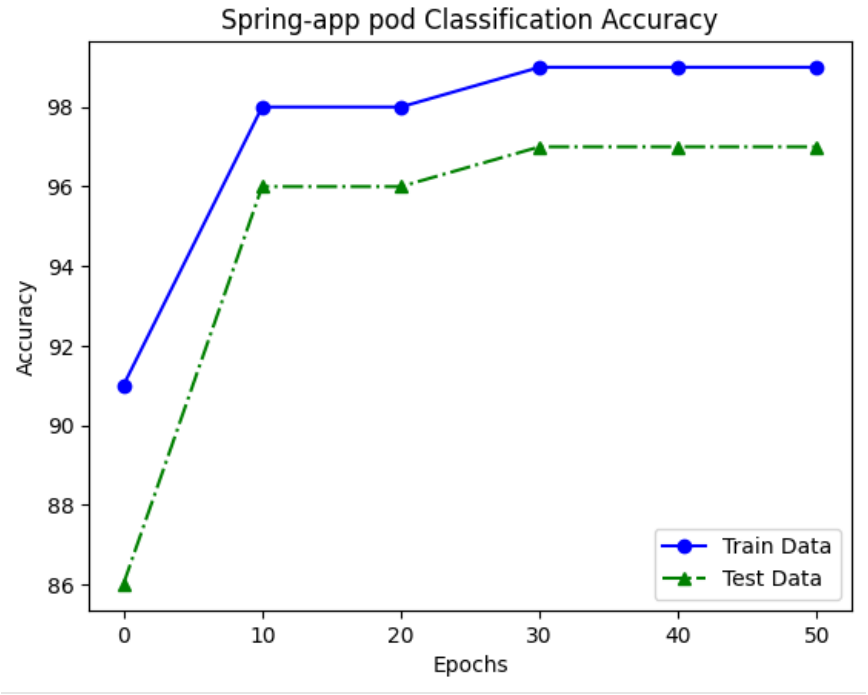

Figure 27. Spring Boot Pod Load Class Prediction Accuracy – Classification.

Accuracy is a fundamental metric used to assess the performance of an AI model in classification tasks. It represents the ratio of correctly predicted instances to the total number of instances in the dataset. The formula for accuracy is, as follows:

$$\left(\frac{True\ Positives\ +\ True\ Negatives}{True\ Positives\ +\ True\ Negatives\ +\ False\ Positives\ +\ False\ Negatives}\right)$$

Since the dataset is balanced, this metric can be used to assess the performance of the model. Also, from the figures we can draw that the model performs very well in each pod, as the validation accuracy reaches the values of 98% and above.

All the above models serve not only as analytics to the TALON cluster administrator, but also can be utilised for orchestration and enforcement of policies using the TALON-OVN infrastructure. For example, predictive analytics on the CPU load could trigger the activation or the release of computational resources (e.g., spawn more virtual machines with X CPU characteristics), or predictive analytics on the bandwidth could update the values of the QoS network management that the TALON-OVN provides.

4.3.4 Code Repository of the NG-SDN Early Prototype

From the beginning of the TALON project, we have set up a GitLab repository [14] with automated CI/CD pipelines1 to ease the development and deployment of the different components. UBITECH has made available to the entire project access to its proprietary GitLab and CI/CD templates for smooth deployment and integration. In the context of the NG-SDN and Distributed Intelligence functionalities, the code repository is as follows. We clarify here that user credentials within the

[1] https://about.gitlab.com/topics/ci-cd/

TALON Consortium are needed to access the source code or upon request. Figure 28 illustrates the structure of the NG-SDN code repository for the TALON project.

- Data Fetcher from InfluxDB: https://gitlab.ubitech.eu/pdml/talon/influx-fletcher
- AI Models & Predictive Analytics: https://gitlab.ubitech.eu/pdml/talon/ai-analytics
- Backend: https://gitlab.ubitech.eu/pdml/talon/talon-python
  https://gitlab.ubitech.eu/pdml/talon/talon-energy-flask

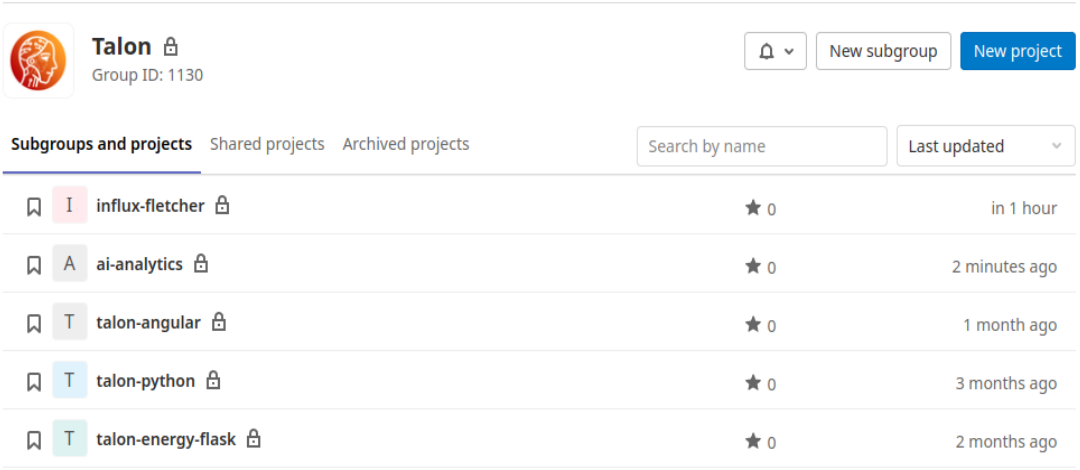

Figure 28. NG-SDN and Distributed Intelligence GitLab repository.

5 Conclusion and Future Outlook

The goal of the deliverable was to report on the first version of the TALON Network Management Services and more specifically on how network management policies are applied on top of the applications that are deployed using the TALON orchestration engine. Both the deployment and the analytic services reported in D3.2 serve the NG-SDN functionalities of TALON as well as capabilities for distributed intelligence of E2c applications.

Policies are applied to perform in-line processing, off-line processing, and traffic replication. As denoted, each policy contains 'triggering conditions' and the 'proper enactment' that have to be performed in case these conditions are satisfied. Practically, these policy elements are serialized as boolean expressions on top of the networking structures of Layer-3/4 and Layer- 7 of the traditional OSI model (i.e. pattern matching rules). The transition of these expressions from their definition to their realization to executable rules is undertaken by TALON.

During the interpretation process, policies are decomposed in conditions and actions which are further decomposed on Layer-3/4 and Layer-7 rules. These level-specific rules are materialized in executable format based on the advertised existing compute resources. The TALON control plane keeps track of the available resources and the state of deployment for each application. Applications expose endpoints and implement node to node communication. Traffic from both internet-facing services and node to node links realize the established data plane. The establishment of the data plane per se is performed by the TALON control plane. The TALON Network Management Services are interacting with the established data plane to optimally 'inject' rules that can dynamically change. The injection process is performed through communication with the TALON Node Level Agents monitoring the underlying infrastructure and pods.

In the future, we plan to enrich the network policies and trigger runtime adaptations by considering the results from the analytics and AI Models, as well as SLOs preconditions we need to fulfil along with QoS postconditions.

6 References

[1] Programming Protocol-independent Packet Processors (P4). Available at: https://p4.org/specs/

[2] Kepler. Available at: https://sustainable-computing.io/

[3] Prometheus. Available at: https://prometheus.io/docs/introduction/overview/

[4] InfluxDB. Available at: https://www.influxdata.com/home/

[5] Kube-OVN. Available at: https://kubeovn.github.io/docs/v1.12.x/en/

[6] OSI. Available at: https://en.wikipedia.org/wiki/OSI_model

[7] eBPF. Available at: https://ebpf.io/what-is-ebpf/

[8] Cillium. Available at: https://docs.cilium.io/en/stable/

[9] Data Plane Development Kit. Available at: https://www.dpdk.org/

[10] Spring Boot. Available at: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

[11] MongoDB. Available at: https://www.mongodb.com/docs/

[12] RabbitMQ. Available at: https://www.rabbitmq.com/documentation.html

[13] PostgreSQL. Available at: https://www.postgresql.org/docs/

[14] GitLab. Available at: https://gitlab.com/

[15] Cañete, Angel, et al. "Energy-Aware Placement of Network Functions in Edge-Based Infrastructures with Open-Source MANO and Kubernetes." International Conference on Service-Oriented Computing. Cham: Springer Nature Switzerland, 2022.

[16] Sekigawa, Shu, Chikara Sasaki, and Atsushi Tagami. "Toward a Cloud-Native Telecom Infrastructure: Analysis and Evaluations of Kubernetes Networking." 2022 IEEE Globecom Workshops (GC Wkshps). IEEE, 2022.

Funded by
the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No 101070181