Deliverable D3.3
Final Report on Platform Integration for the MonB5G AE/MS

Document Summary Information

Grant Agreement No	871780	**Acronym**	MonB5G
Full Title	Distributed Management of Network Slices in beyond 5G		
Start Date	01/11/2019	**Duration**	42 months
Project URL	https://www.monb5g.eu/		
Deliverable	D3.3 – Final Report on Platform Integration for the MonB5G AE/MS		
Work Package	WP3		
Contractual due date	M36	**Actual submission date**	
Nature	Report	**Dissemination Level**	Public
Lead Beneficiary	LMI		
Responsible Author	Ashima Chawla (LMI), Anne-Marie Bosneag (LMI)		
Contributions from	S. Barrachina (CTTC), L. Blanco (CTTC), E. Zeydan (CTTC), L. Vettori (CTTC), J. Mangues (CTTC), J. Serra (CTTC), Ashima Chawla (LMI), Anne-Marie Bosneag (LMI), Zhao Xu (NEC), Rafał Tępiński (ORA-PL), Robert Kołakowski (ORA-PL), Jerzy Jegier(ORA-PL), Sihem Cherrared (ORA-FR), Luis A. Garrido Platero (IQU)		

Revision history

Version	Issue Date	Complete(%)	Changes	Contributor(s)

Disclaimer

Copyright message

TABLE OF CONTENTS

List of Figures

List of Tables

List of Acronyms

Acronym	*Description*
3GPP	*Third Generation Partnership Project*
AE	Analytic Engine
AE-F	Analytic Engine Function
AE-S	Analytic Engine Sublayer
AI	Artificial Intelligence
AD	Anomaly Detection
CLA	Closed-loop Automation
CNF	Cloud Native function
COMS	Common Online Memory Store
DE	Decision Engine
DE-F	Decision Engine Function
DE-S	Decision Engine Sublayer
EEM	Embedded Element Manager
eMBB	Enhanced Mobile Broadband
eTOM	Enhanced Telecom Operations Map
ETSI	European Telecommunications Standards Institute
ECA	Event Condition Action
ENI	Experiential Networked Intelligence
FCAPS	Fault, Configuration, Accounting, Performance, Security
IDM	Infrastructure Domain Manager
IDMO	Inter-Domain Manager and Orchestrator
IDSM	Inter-Domain Slice Manager
ISM	In-Slice Management
ITU	International Telecommunication Union
KPI	Key Performance Indicator
LCM	Lifecycle Management
ML	Machine Learning

MANO	Management and Orchestration
MaaS	Management as a Service
MAN-F	Management Function
mMTC	Massive Machine Type Communications
MEO	MEC Orchestrator
MNO	Mobile Network Operator
MLaaS	MonB5G Layer as a Service
MS	Monitoring System
MS-F	Monitoring System Function
MS-S	Monitoring System Sublayer
MEC	Multi-access Edge Computing
NFVO	Network Function Virtualization Orchestrator
NSD	Network Service Descriptor
NSI	Network Slice Instance
NSO	Network Service Orchestrator
NSP	Network Service Provider
NSI	Network Slice Instance
NSMF	Network Slice Management Function
NSSMF	Network Slice Subnetwork Management Function
NST	Network Slice Template
NSSI	Network sub-Slice Instance
NGMN	Next Generation Mobile Networks
NFVI	NFV Infrastructure
OAI	Open Air Interface
ONAP	Open Network Automation Platform
OSM	Open-Source MANO
OSS	Operation System Support
PaaS	Platform as a Service
PoC	Proof of Concept
QoE	Quality of Experience

QoS	Quality of Service
RAN	Radio Access Network
SON	Self-Organizing Network
SLA	Service Level Agreement
SFL	Slice Functional Layer
SML	Slice Management Layer
SM	Slice Manager
TSDM	Time Series Database
uRLLC	Ultra-Reliable Low-Latency Communication
VIM	Virtual Infrastructure Manager
VNF	Virtual network Function
VNFM	Virtual network Function Manager
ZSM	Zero-touch network and Service Management

1 Executive summary

Beyond 5G systems require the design of a distributed, scalable and flexible slice management system. MonB5G provides such a management solution, which is based on distribution of management functions, intelligent AI-based methods for flexible analysis and automated decision making at different levels in the network. Apart from the architecture design, many integration and implementation choices must be carefully pondered to achieve the full potential of such a system.

This deliverable focuses on the design choices that MonB5G took for the implementation of the Monitoring System (MS) and its Analytics Engines (AEs), as well as choices regarding the flexible integration between these management components. The MS and AE components are based on cloud native techniques, being provided as containerized solutions that can be easily scaled and integrated into a management platform at different levels (slice, node, orchestration domain, etc.). The integration between these MS/AE services is also designed to allow either for fast real-time consumption of monitored data through a publish / subscribe mechanism, or for batch access to monitored data in a certain time interval.

Additionally, advanced AI mechanisms are built into the AE component to enable automatic behaviour learning for proactive optimization and fault management decisions. These mechanisms are based on Deep Neural Networks, including Graph Neural Networks, as well as Federated Learning. We show through our testing that these methods adapt well to changes in scale, while preserving their performance.

This deliverable also provides detailed description of the design of the MS containers, including instructions on how to easily integrate other management components (e.g., AE) with the MS containers to obtain access to the desired monitored data. This is a very important aspect in enabling further integration and use of our system. It also paves the way towards the implementation of the proofs-of-concept in this project.

Different AEs are included, based on various advanced AI methods as mentioned before and offering a different type of service, such as slice KPI predictions or anomaly detection. The tests focus on the scalability aspects and the ease of integration with the correct MS service, as well as additional features such as interpretability of results that can be further used by the decision engines in the MonB5G management platform. The experimental tests show that these AEs perform well at scale and are easy to integrate with the appropriate MS to provide the management functions required.

The key achievements covered by this deliverable are:

- Presentation of MS system, including design choices for ease of use and scalability, as well as detailed description on proper use and integration of the service at different levels in a MonB5G-like platform.
- Presentation of scalable AE system, including advanced AI techniques that have been tested at scale.
- Description of design choices for ease of integration of MS and AE components.

2 Introduction

2.1 Scope and Motivation

This deliverable presents the results accomplished in the WP3 of the MonB5G project with respect to the implementation and testing of the AE components, the implementation of the MS, as well as the integration of these components with the MS. This document relies on the results reported in the earlier deliverables in WP3, D3.1 and [1], where we explained the design and requirements for the AE and MS components, as well as the research results with respect to the implementation of the different AEs.

Additionally, we present here the details of the implementation of the MS and the different AE components and how these are integrated, following the requirements and the designed architecture. In particular, we show that the MS was designed for scalable collection of data and ease of integration with AE & DE components at all levels. The AE components were also designed, implemented and integrated with the goals of distribution and scale in mind, focusing on distributed AI techniques tailored to the decentralized and programmable management architecture described in [2] leading to significant reductions in communication overhead and time delays that can result from monitoring and analysis.

The way in which we implemented the MS and AE components, which are critical components of the distributed management architecture designed in MonB5G, is explained in detail, allowing the reader / user to understand how these components can be deployed and integrated into different 5G and beyond 5G platforms. This also paves the way for the implementation of the proofs of concept in WP6, which will be reported in the WP6 deliverables.

2.2 MonB5G Novelty and Contributions on MS/AE

The MonB5G approach to MS and AE enables the implementation of a decentralized data-driven and automated framework, which requires limited human interactions. Multiple features that leverage the operation of the MonB5G framework have been proposed, which facilitate the usage of AI-based methods and automation of orchestration and management processes in order to improve the performance, reliability and scalability of the system. The MonB5G framework incorporates **different control loops with different scopes, goals, and timescales** at the Global OSS/BSS level, Technological/Orchestration Domain level, Slice level and Node (VNF/PNF/CNF) level. Therefore, the MS/AE functionalities are distributed among multiple MonB5G system components (Figure 1). The details regarding the system internals and the scope of operation (including the MS/AE layer specifics) are described in MonB5G D2.4 [2].

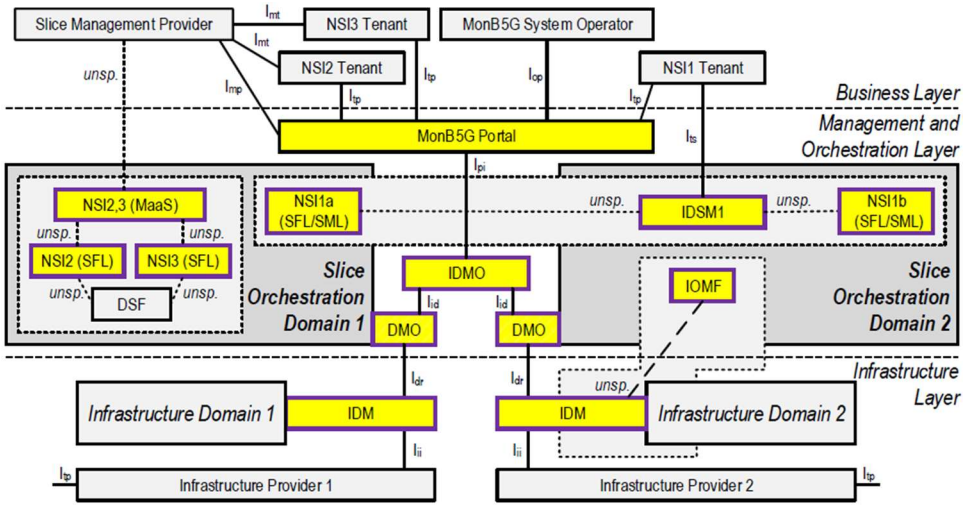

Figure 1: MonB5G architecture with marked MS (yellow fill) and AE (purple border) capabilities.

The ability of the MS to provide real-time data at different levels of the system is enabled by the following strategies:

- Distributed architecture of the MS, based on the microservice architecture and implemented as a cloud-native application in a Kubernetes cluster. This implementation takes advantage of the auto-scaling feature of Kubernetes, which mirrors our needs to scale up and down depending on the number of active slices in the system.
- Sampling loops implemented as a single Docker image, which is deployed only once and continuously receives data from the monitored element.
- Hierarchical architecture for the MS, where smaller sampling loops collect monitoring information from a local domain and can serve the local AE & DE, while higher level sampling loops do not monitor each element in the lower domains but get that data over multiple domains straight from the lower-level MS.
- Reduced footprint of the system through allowing the sharing of resources: e.g., the Kafka bus and database used for sharing and storing monitored data can also be used for sharing and storing AE & DE data (e.g., results of predictions, analyses, etc.), as well as using a single sampling function for sampling multiple telemetry data.

On the AE side, to ensure the scalability of the AE component and its suitable design for the MonB5G architecture, we have used the following strategies:

- Distribution of analytics over the different hierarchical domains in the MonB5G architecture, which ensures timely local analytics and usage of the local resources, while supporting scalability of the AE system.
- Deployment of AEs close to the collection points for the needed data reduces communication overhead and improves time to response.

- Use of advanced AI methods to optimize the distribution of analytics over the management entities, e.g., using methods such as Distributed Neural Networks and Federated Learning. These methods result in significant reduction in overhead and convergence time, while preserving the accuracy and precision.

Additionally, our choice of implementation for the MS/AE components, based on cloud native technologies, enables ease of integration, flexibility and scalability of the MonB5G management platform. Each management component is developed as an independent service, packaged as a container, while flexible communication between the containers is provided through a publish / subscribe mechanism. This design makes the management platform resilient, scalable and agile.

All these design choices and strategies will be explained in this deliverable, following the structure explained below.

2.3 Structure of the Deliverable

The main technical chapters of the deliverable are organized as the following table. Here we also map the committed tasks of the grant agreement (GA) with the outputs reported in this deliverable in order to further clarify and position the innovative contributions under the framework of the MonB5G project.

Table 1 Deliverable Structure and Mapping with Project Tasks

Chapter	Description	Task(s)	Duration
3	Scalable integration of MonB5G & AE	T3.4	
4	Advantages of our selection / development	T3.1-T3.4	
5	Cloud-native implementation of the MS	T3.1, T3.4	
6	Cloud-native implementation of the AE	T3.2, T3.3, T3.4	

3 Advantages of MonB5G MS/AE design & implementation choices

The MonB5G MS and AE aim to provide a scalable and zero-touch solution for monitoring and analysis of a massive number of network slices. To this end, we not only developed the AI technologies for distributed slice management (please see details in the deliverable [1]), but also creatively design the implementation and deployment of the AI-driven components in a distributed and scalable framework. The proposed cloud native implementation of the MonB5G MS and AE are fulfilled under the well-designed MonB5G architecture, such that the AI-enabled management operations are flexibly distributed at different levels of the management hierarchy and achieve significant complexity reduction and fast analysis of the network slices. In this section, we firstly give a brief introduction of the MonB5G architecture, which leads us to the current cloud-native implementation of the management components. Then, we analyse the advantages of our development to offer the interested users with a comprehensive vision of what they can benefit by employing the developed MonB5G MS and AE components.

3.1 MonB5G Architecture: Guidelines of MS/AE Implementation

The implementation of the MonB5G MS and AE is designed under the MonB5G architecture (proposed in the deliverable [2] of this project). To the best of our knowledge, it is the first attempt in the literature that addresses the scalability and robustness of network slicing management and orchestration by using a distributed and programmable management framework. Here is an overview of this architecture, which plays as guidelines of our cloud native implementation of MS/AE.

The MonB5G architecture aims to provide a zero-touch slice management and orchestration framework that can facilitate the deployment of a massive number of slices in different administrative and technological domains. We proposed a decentralized network management approach, which inherently increases flexibility, reliability, scalability, and security for all stakeholders of the ecosystem. The management operations are categorized as monitoring, analysis and decision, and the management hierarchy is composed of four levels, including node (VNF/PNF/CNF), slice, domain, and inter-domain. The MonB5G platform distributes the AI-driven management operations at multiple levels, and executes them locally, close to where the monitoring data generated, such that the communication overhead caused by monitoring can be largely reduced, and in the meanwhile, the analysis and decision operations can be done locally and efficiently without unexpected delay caused by data/results communication. Due to the distributed management hierarchy, the management functions can thus be implemented in a containerization way and are loosely coupled according to requirements of the slices' tenants. In general, the components of the MonB5G architecture are shown in Figure 2.

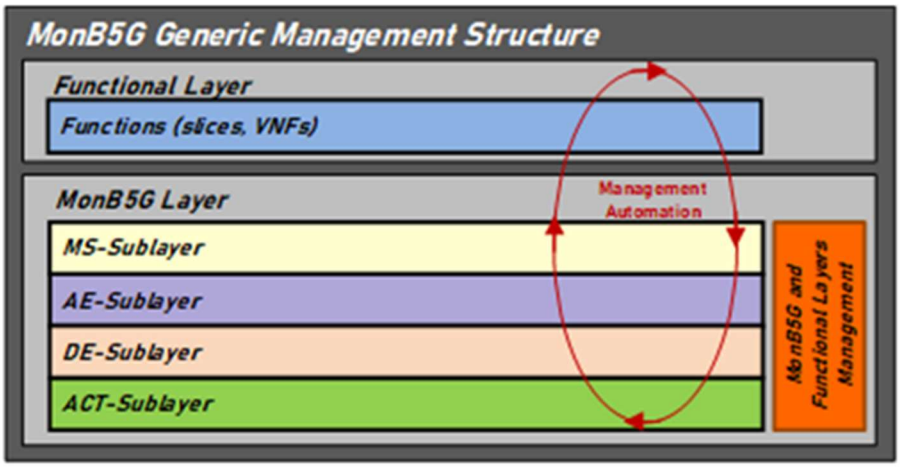

Figure 2: Generic view of MonB5G slice structure [D2.4]

One can find that the MonB5G management layer consists of four sublayers, which provide different operations described as Table 2: MonB5G management sublayers:

Table 2: MonB5G management sublayers

Sublayers	Tasks
MS-Sublayer	Monitoring subsystem aims to collect, aggregate, filter, and pre-process the monitoring data of the functional layer.
AE-Sublayer	Analysis engines are designed to analyze the performance of the functional layer for e.g., KPI prediction and anomaly detection, based on the collected data by MS sublayer.
DE-Sublayer	Decision engines are responsible for making reconfiguration decisions to optimize the performance of the functional layer according to the collected data of the MS sublayer and the analysis results of the AE sublayer.
Act-Sublayer	Actuators is to convert the DE decisions into multiple atomic reconfiguration-related operations that simplify the reconfiguration and reduces the traffic between DE and reconfigured node(s).

As this deliverable focuses on the implementation of MS/AE, we now illustrate more details of the MonB5G MS and AE Sublayers. For the other sublayers, please refer to the deliverable [2].

The generic structure of the **MS Sublayer** is shown in Figure 3: Monitoring System Sublayer [D2.4] [2], which consists of six internal components. These components are responsible for different tasks, illustrated in Table 3. Additionally, a publish-subscribe paradigm is implemented to expose the information to the entities (higher in the management hierarchy) via a dedicated message bus.

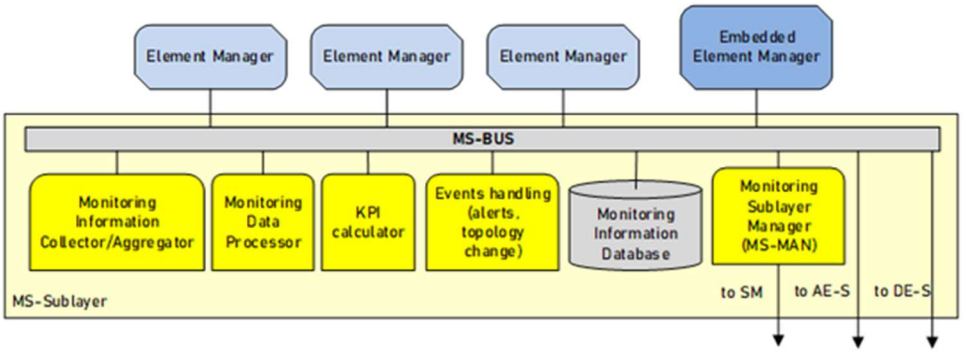

Figure 3: Monitoring System Sublayer [D2.4]

Table 3: Functions of internal components of Monitoring System sublayer

Internal components	Tasks
Monitoring Information Collector/ Aggregator	Collect data from the monitored functional layer.
Monitoring Information Processor	Process the raw monitoring data, e.g., filtering, cleaning.
KPI calculator	Calculate the predefined KPIs based on the raw monitoring data.
Monitoring Information Database	A (time series) database to store the raw and processed data.
Event Handling	Collect events based on predefined rules.
Monitoring Sublayer Manager	Enable remote configuration of the MS sublayer.

To monitor the running of a functional layer, the MS sublayer fulfils a set of features. It supports the basic operations, such as data collection, aggregation, filtering, and interpolation. In addition, the sublayer can also include the functions, such as computation of predefined KPIs. Moreover, the MS sublayer addresses both telemetry data (continuous time series) and event data (such as alarms, faults and topology changes). It enables the monitoring operations in different temporal granularity and varying degrees of data aggregation, depending on the optimization goal.

The generic structure of the **AE Sublayer** is shown in Figure 4, whose structure follows the same approach as that of MS sublayer. The AE sublayer includes a set of engines, each of which performs a singular analytic task. Examples of analytic operations include security threat detection, real-time fault/performance degradation detection, etc. Analytics results are stored in a separate database for usage of Decision Engine Sublayer. The entity of AE Sublayer Manager provides functions to remotely manage the set of engines. In addition, a message bus is embedded to obtain monitoring data from the MS sublayer and expose the analytics results to the Decision Engine sublayer.

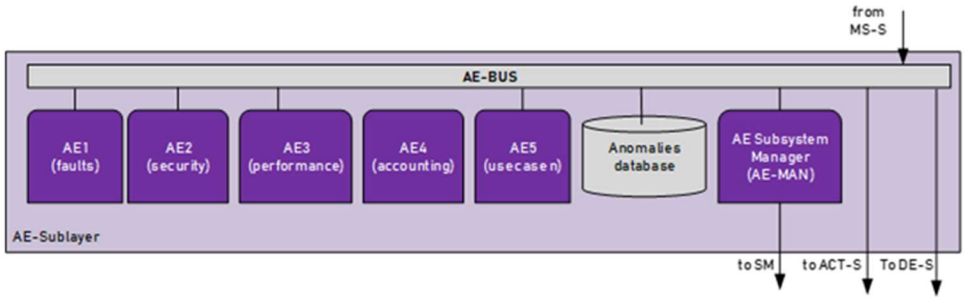

Figure 4: Analytic Engine Sublayer [D2.4]

Here we introduce the general architectures of the MonB5G MS/AE. Note that, the detailed internal structure, the characteristics of the processed data, as well as the scope of operations will depend on the deployment sites (the associated functional layers). In particular, the parts responsible for slice orchestration (i.e., IDMO, DMO) will mainly use the data and domain-level slice KPIs provided by the domain orchestrators (e.g., MANO), while at the slice level (IDSM, network slice) the scope of monitoring and optimization will be more focused on the operation of a slice itself. It should be emphasized that further changes may occur due to the type of domain in which the MS/AE components are deployed (e.g., RAN, Cloud). Nevertheless, the implementation architecture of both MS and AE follows the generic principles introduced here, so that they can be easily adapted or extended depending on the requirements derived from the specific deployment scenario (i.e., optimization goal, algorithms used, security requirements, etc.).

3.2 Interfacing MS & AE

The MS and AE are separate components, deployed as containerized services, that need an easy and flexible way to integrate and interact with each other. Figure 5 shows the high-level view of two integration interfaces between MSs and AEs. AEs with different functionalities, as shown in the upper half of the figure (i.e., AE-1 and AE-2) can communicate with different MSs belonging to different technological domains, or can also be part of different entities (e.g., DMO, IDMO, IDM, IDSM, or NSI) that monitor different segments of the infrastructure. Depending on the scenario, the request type and the requirements of the solution, two different ways of integrating AEs into MS have been followed within MonB5G:

1. **Through Kafka Cluster for real-time data processing.** In uRLLC network slice type scenarios, data should be collected and analyzed with low latency. In these scenarios, decisions should be made immediately based on the up-to-date data. In the MonB5G interface between MS and AEs defined earlier as (**I**ma) and for much faster feedback loops and higher integration, AEs can subscribe to the Kafka cluster (e.g., topic name) and fetch the relevant data generated by the sampling function into the Kafka cluster under a specific Kafka topic name.
2. **Through TSDB (or the COMS database) for batch data processing** Data is collected for large-scale monitoring and can be analyzed in offline mode. Using TSDB, which stores the historical data such as historical analytics insights, historical workflows, historical actions, etc., AEs can perform batch-based

data processing. In this way, the data received until a certain time period is collected and then processed by AEs as a batch. In batch-based data processing, data is collected at scheduled periods of time and organized into a transaction file that is stored until a sufficient amount of data has been collected (e.g., deep learning-based models require larger datasets compared to traditional ML approaches).

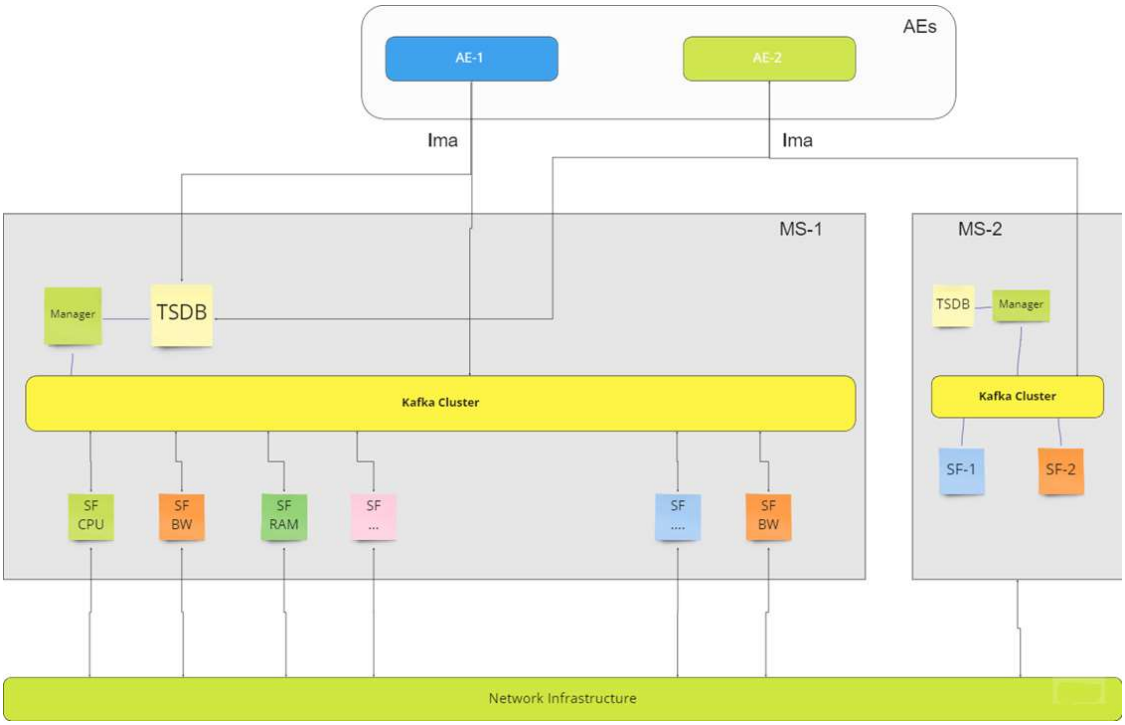

Figure 5: Two integration options of AEs with MS

In Figure 5, different AEs with different functions can consume data from different MSs. For example, in Figure 5, AE-1 consumes data from the TSDB of MS-1 for training purposes (to build a ML or DL model that can later be used for proactive actions on the network infrastructure) and also consumes data from the Kafka cluster for testing purposes. On the other hand, AE-2 consumes real-time data from MS-2, while it uses batch collected data from the TSDB of MS-1 to build its own analysis. This design ensures flexible and scalable integration, with MSs encompassing different scopes for the monitoring data and different AEs being served with the necessary data at the right level for each management task defined (Table 4).

Table 4: Different AE Solutions Mapping

AE Solution	**MS scope of input data**	**Output Consumers**
Anomaly Detection	Slice-level	DE layer
Stochastic FL	Slice-level	DE layer
Local KPI prediction	Slice-level	DE layer

Traffic Load Predictor	Slice Level	DE Layer

3.3 Cloud Native Implementation of MonB5G MS/AE and Advantages

Guided by the well-designed MonB5G architecture, it is natural to implement our MS/AE as cloud-native applications. Based on the Cloud Native Computing Foundation, cloud-native is officially defined as follows[1]

"Cloud-native technologies build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach."

In particular, each MonB5G MS/AE component is developed as an independent service, which is packaged as a container (such as a Docker container). Since everything (e.g., dependencies) is encapsulated into the container, the MS/AE component is well isolated from the underlying infrastructure, and thus is portable and easy to deploy in any environment that has the container runtime engine. The container orchestration (such as Kubernetes) manages the lifecycle of the containers, and provides the support to scale the containers in or out automatically according to the demand. The communication between the MS/AE components can be based on publish/subscribe messaging platform like Kafka. As shown in the Figure 6, the following tools are exploited to develop our cloud native MS/AE components:

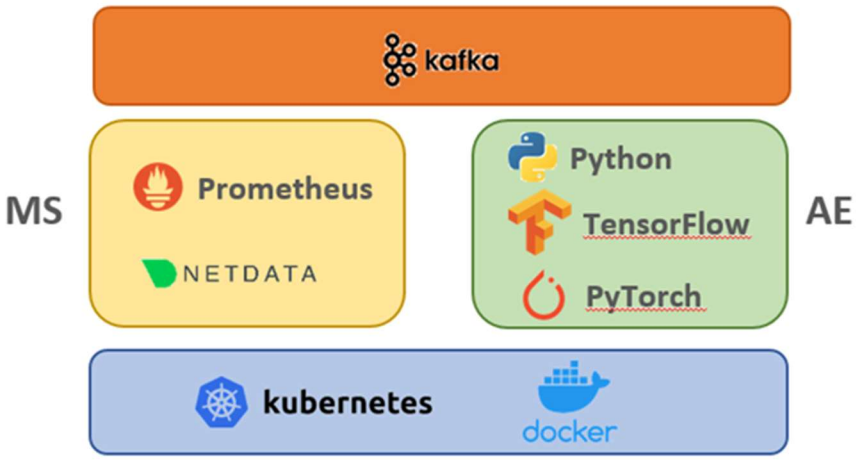

Figure 6: Tools for cloud-native implementation of MonB5G MS/AE

These cloud native techniques enable loosely coupled MS/AE, which makes our MonB5G management platform resilient, scalable and agile. In a nutshell, the selected implementation techniques offer the following five key advantages:

[1] https://github.com/cncf/foundation/blob/main/charter.md)

- Independence. By packaging each MS/AE components in separate containers, we can deploy them independently at different layers of the MonB5G network management hierarchy.
- Portability. Since the MS/AE containers are environment-agnostic and can operate in an isolated way, it is simple to deploy them according to the management demands of the slice customers, and move them to another infrastructure if needed.
- Scalability. The implemented MS/AE containers are supported by the container orchestration (such as Kubernetes), which automatically manages the lifecycle of these containers, and scales in and out based on the traffic load and computing requirements.
- Standards: The MS/AE's implementation and deployment are based on open source and standards-based technologies (such as Docker, Kubernetes), which enable interoperability and workload portability, as well as reduce vendor lock-in.
- Efficient development. Since we implement the MonB5G MS/AE as cloud native apps, the components are encapsulated with containers, and thus each functionality, developed by different partners, can be in different programming languages. Containerization removes the possible risk of confliction between languages, libraries and frameworks. In addition, the MonB5G MS/AE can be delivered via a DevOps pipeline that includes continuous integration and continuous delivery (CI/CD) tool chains, which automates the building, testing, and deployment of the created MS/AE components.

4 Scalable integration of intelligent mechanisms in MonB5G

MonB5G MS and AE enable distributed AI-driven monitoring and analytics for managing a large number of slices. The decentralized monitoring and analytics operations are distributed among the hierarchically structured management entities with a focus on reducing communication overhead and delay. Rather than a centralized solution, as it is considered by both ETSI NFV and 3GPP management architectures, MonB5G designs a decentralized Monitoring System (MS), Analytics Engine (AE) and Decision Engine (DE), which is deployed on the different entities taking part in the management process, i.e., MANO, OSS/BSS, NSMF, network slice owner, and the in-slice management plane, distributing the management functions among these entities, in order to ensure a scalable management system. To support automated and proactive decisions at the slice level, the AE components provide predictions of the slice KPIs. End-to-end KPIs at the slice level have been defined in line with standards specifications and include, for example upstream/downstream throughput for NSI, average end-to-end uplink/downlink delay, virtualized resource utilization per NSI, etc. Although KPI prediction has been implemented in older systems, there are some characteristics that we need to consider when implementing it in a 5G system at slice level. The number of slices, as well as the amount of data collected when automatic slice redeployment is enabled, are important factors that affect the effectiveness and efficiency of AE so a highly scalable solution must be developed.

In this chapter, we focus on the scalability of the MonB5G AE, working hand in hand with the design decisions discussed in the previous section. Apart from enabling scalability through flexible integration mechanisms of MSs and AEs, one important aspect to consider is how scalable the AI techniques used are. These AI methods must be able to deal with the large scale of the system, while preserving performance and accuracy. In this section, we start with a quick introduction on AI-driven Network Management, as an important component of our system, and then discuss the suitability of the AI methods we have chosen, in the context of the MonB5G architecture.

4.1 AI-driven Network Management

Future mobile networks are expected to support massive number of network slices for a variety of vertical applications that will leverage not only MNOs operation but also create business opportunities for the MNO customers, which will be able to dynamically request personalised services, tailored for the customers preferences in terms of performance, availability, resources and costs. The primary component of the service is a network slice i.e., a set of interconnected VNFs, where each VNF performs a set of specific functionalities that together compose that service. Hence, to meet individual service requirements a virtual subset of the physical resources of the network infrastructure are allocated to the slice from one or multiple network domains (depending on the service definition). To this end, the slice management systems should provide means for performing administration tasks in multiple domain types (RAN, Edge and Cloud). These capabilities, however, are not covered by the traditional network management systems. The anticipated Vast numbers of parallel and flexible slices, however, pose significant challenges in the context of network management and especially scalability of management operations.

For today, the majority of the existing network management systems (e.g. MANO) fail to provide the desired capabilities due to high centralization. Centralized network administration creates two major issues:

- Significant traffic overhead due to forwarding of monitoring information to one centralized system,

- Increased latency in performing FCAPS-related operations (reconfiguration, optimization, fault detection, healing etc.)

Network slicing further aggravates these problems, by increasing the volume of exchanged management-related data as well as stringent delay requirements for latency-critical services or specific domains (RAN, Edge). Moreover, centralization of the management system can also generate additional costs in terms of maintenance as introduction of a new slice requires development and dynamic deployment of system extensions to support it.

The high diversity of slices imposes two main requirements on the slice management systems:

- Collection of monitoring with little to no latency to reflect the current state of the components composing a slice,
- High degree of automation to enable online analysis and making decision to react to abrupt network changes or perform optimization tasks.

To satisfy these requirements, MonB5G introduces multiple intelligent closed-loops with different functional and time scope that use AI-driven methods for monitoring, analysis and decision making. The possible applications of AI in terms of monitoring can include selection of most significant KPIs in terms of slice SLA maintenance, creation and prediction of new KPIs that better reflect the component state, reduce telemetry data dimensionality, etc. Incorporation of intelligent monitoring with AI-driven analytics and decision making into the management framework, will enable carrier-grade massive scale, multi-domain network slices.

4.2 Major Achievements of MonB5G MS & AE Integration

To achieve MS and AE scalability, MonB5G considers a number of KPIs with regards to the slice acceptance ratio and SLA performance. To showcase these KPIs, we applied different techniques and solutions to face scalability such deep learning, federated learning, and graph representation techniques. The aim is to prove that the performance of the proposed Monb5G MS & AE solution remains high, not affected by the overhead induced by scalability.

1. ML/AI techniques for large scale network:

The use of ML methods such as deep learning allow learning in a highly heterogeneous and large data sets which meets the requirements of a highly dynamic and scalable network. In this essence, we applied Deep Reinforcement Learning (DRL) to increase the scalability with the automation of Network Slice Placement considering a multi-objective optimization approach to the problem. The optimization of resources allocation is a multi-objective optimization since it considers network resources (CPU, RAM, memory), bandwidth and the network load. We applied the Asynchronous Advantage Actor Critic (A3C) algorithm with two DNN layers that are trained in parallel: the Actor Network is used to generate the policy π_θ at each time step, and the Critic Network which generates an estimate for the state value function.

To test our solution, we applied a network load generator that generates slice requests arrivals considering three different network load scenarios: stationary, cycle-stationary, and non-stationary network load

scenario. The stationary network load scenario is a static load while the cycle-stationary and stationary loads vary in time. The cycle-stationary varies with a predictable periodic load while the non-stationary load in a non-predictable change as shown in Figure 7.

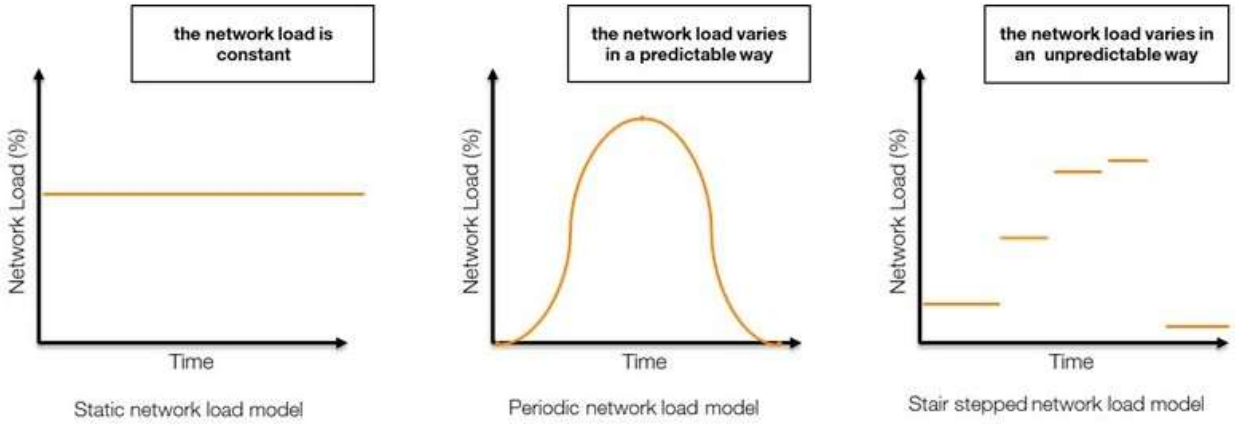

Figure 7: Different network load scenarios

The DRL learning was assisted with a Power of two choice (P2C) heuristic called (Ha-DRL). This solution was tested in the different network load scenarios. Figure 8 and Figure 9 showcase a comparison between the DRL, HA-DRL and the heuristic (HEU) approach in two different network loads (normal 50% and critical 80-90%). The β parameter is used to control how much heuristic influence the DRL policy. The learning was applied to a large physical substrate network (PSN) with around hundreds of nodes (Figure 3).

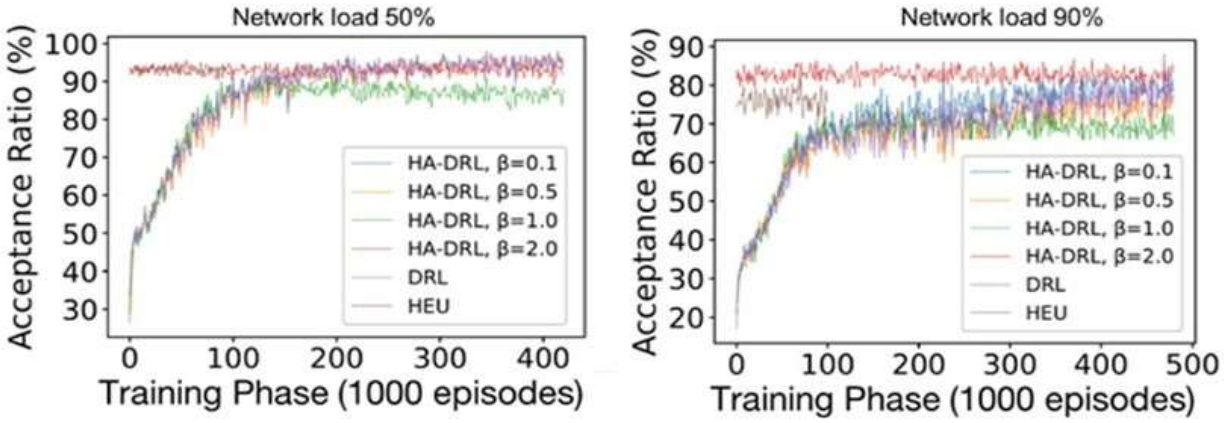

Figure 8: Evaluation Results of the DRL approach in stationary 50% and 90% network load

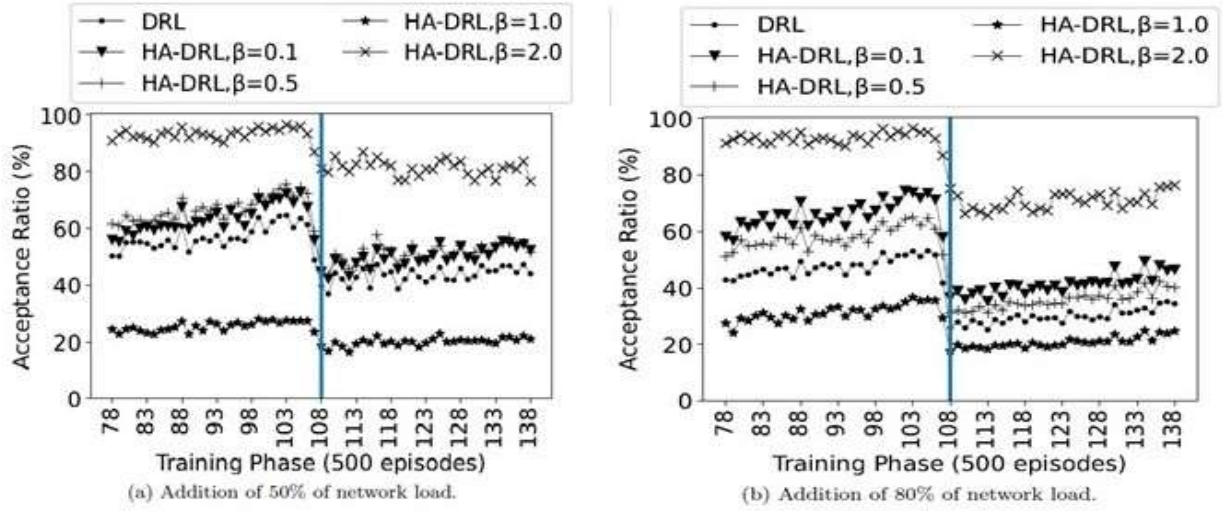

Figure 9: Evaluation Results of the DRL approach in non-stationary 50% and 80% network load

In the training of Figure 9, the blue line represents the change from normal network load 40% before the line to up to +80% network load after the blue line. We can observe in Figures Figure 8 and Figure 9, that with the reduced training time of 100 training phases only HA-DRL, with β = 2.0 has converged. This is due to the fact that the strong influence of the Heuristic Function helps the algorithm to become stable more quickly which reduces the training time. We notice that even in an excessive stationary network load (90%) the HA-DRL with heuristic β =2 can achieve high slice acceptances ratio (>90%) compared to other methods. The HA-DRL with heuristic β =2, was also effective in the case of +80% of disruption.

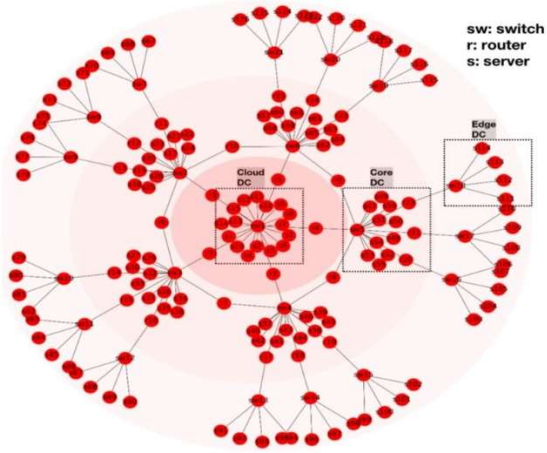

Figure 10: Multi-domain physical substrate network

2. Graph representation techniques:

Graph representation techniques such as Graph Neural Networks (GNNs) are a class of deep learning methods designed to perform inference on data described by graphs. The use of the graph provides a node-level, edge-level, and graph-level prediction tasks and it also includes the node neighbors features to the learning which increases the accuracy.

Moreover, to represent all the network features in case of large networks, we applied Graph Convolutional Network (GCN). GCN enables a scalable state information processing and variant sizes of slices which consist of dozens of nodes and links (neural networks usually require fixed-sized vector inputs).

In our AE solution illustrated in Figure 11, we use the GCN formulation proposed by to automatically extract advanced characteristics of the PSN. The characteristics produced by the GCN represent semantics of the PSN topology by encoding and accumulating characteristics of neighbor nodes in the PSN graph. The size of the neighborhood is defined by the order index parameter K. We consider in the following K = 3 and perform automatic extraction of 60 characteristics per PSN node. Both the network slice requests state and Network Load characteristics are separately transmitted to fully connected layers with 4 and 100 units, respectively.

The characteristics extracted by both layers and the GCN layer are combined into a single column vector of size 60|N| + 104 and passed through a full connection layer of actor critic network with |N| units.

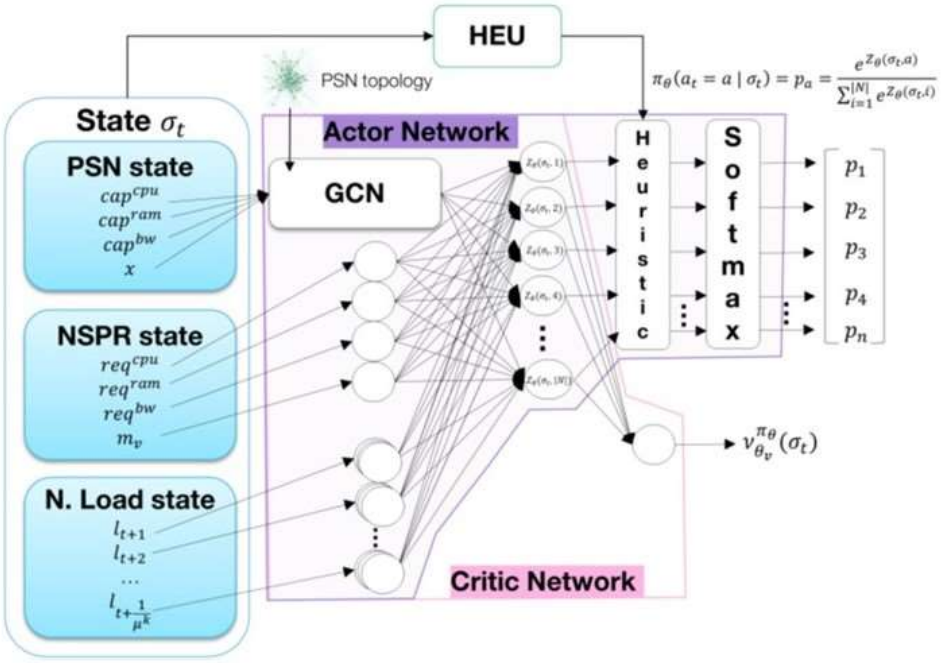

Figure 11: Application of GCN inputs into the actor-critic network

3. Policy-Driven Stochastic Federated Learning Scalable AE:

To deal with scalability and distributed zero-touch management in B5G/6G multi-domain systems, a Federated Learning algorithm was proposed in D3.2. This approach is able to provide a significant overhead reduction and convergence time gains compared to centralized AE, while fulfilling the statistical SLA constraints. In particular, the non-policy FL achieves more than x10 overhead reduction compared to centralized MANO [3].

In order to promote further scalability under a massive slicing framework a cloud native SLA-driven stochastic policy has been implemented. With this solution only a subset of the active AEs is selected in each of the FL round, based on their violation rate. This approach reduces the overhead compared to the non-policy FL algorithm by around 25% overhead reduction in comparison with the non-policy FL algorithm. Thus, providing more than 92% of overhead reduction in comparison with the centralized approach. Furthermore, the policy-FL algorithm exhibits a similar accuracy and violation rate than the non-policy FL algorithm and/or its centralized counterpart [4] (i.e., the fully centralized SLA-constrained deep learning algorithm, in which the whole data is collected from all distributed AEs).

4. Interpretable Anomaly Detection for AE:

Most Deep Neural Networks are a “black box” that cannot provide easily interpretable insights into the relationship between input and output. Particularly, when there is high dimensional data and multiple layers in the neural network, there is a need for a method (post-hoc, after training) that can be used to provide the interpretability of models on the datasets where the ground-truth of interpretation results is not available.

The scalable interpretability framework designed based on the probability score of the predicted slice KPI and SHAP values provides a deeper insight and support for causation analysis for anomalies in the network. This approach helps the analyst in the identification and feature explanation of anomalies providing faster root cause analysis, making the MonB5G architecture a significant step towards self-managed network slices and providing faster root cause analysis. Therefore, the solution provides better support for the network domain analysts with an interpretable and explainable Artificial Intelligence (AI) anomaly detection system.

5 Cloud-native implementation of the MS

5.1 Review of the MonB5G Monitoring System

The MonB5G monitoring system (MS) is designed to collect detailed information about the current status of network slices, which are often deployed in multiple domains. To provide up-to-date monitoring for online analysis of slice KPI and reconfiguration in response to unexpected network dynamics, scalability of MS is an essential aspect for managing a large number of slices. To achieve the goal of scalability, we employed several strategies in the design and implementation of MS.

First, MS is conceived from a *distributed architecture*, i.e., it is designed based on the microservice architecture and implemented as a cloud-native application in a Kubernetes cluster. Thus, the MS benefits from the Kubernetes orchestration/scheduling capabilities such as auto-scaling, which can be adjusted by rules in the Kubernetes deployment depending on the load. Remarkably, all components in the MS are implemented as containers, including the Sampling Functions (SFs), and deployed as pods in a cloud native manner. Besides, monitoring on multi-domain clusters spanning multiple nodes is realized through a messaging bus, implemented by Kafka stretched in all the nodes. This means that any message published by any producer on any node is available to all consumers on all nodes. All of these components communicate over the Kafka bus, enabling the linear scalability of the MS.

Second, to achieve the scalability goal, MS is designed based on a *hierarchical architecture* that allows to have multiple sub-monitoring systems which are collaborating in a master-slave pattern. So, the data monitored by the slave (or child) MS can be further collected by the master (or parent) MS as needed. With this hierarchical structure, monitoring tasks can be performed at lower levels, limiting data traffic, and reducing the reaction time. Monitoring information can be extracted directly from distributed MS entities and aggregated locally. So, monitoring systems in the lower part of the hierarchy are the low levels that directly monitor the VNF and PNF resources, while the MS in the upper part of the hierarchy is the higher-level MS that does not directly monitor the resources, but orders to lower level MSs to gather data from them. When using MS in a multi-domain network, a low-level MS can be instantiated for each technological domain, and the high-level MS would be a centralized MS that collects data from multiple domains. As shown in Figure 12, the telemetry data collected by these MSs is available through the “q” and “db” interfaces. The sampling functions in the central MS (the monitoring system in the upper part of the figure) can use these interfaces to collect the data from the other MS.

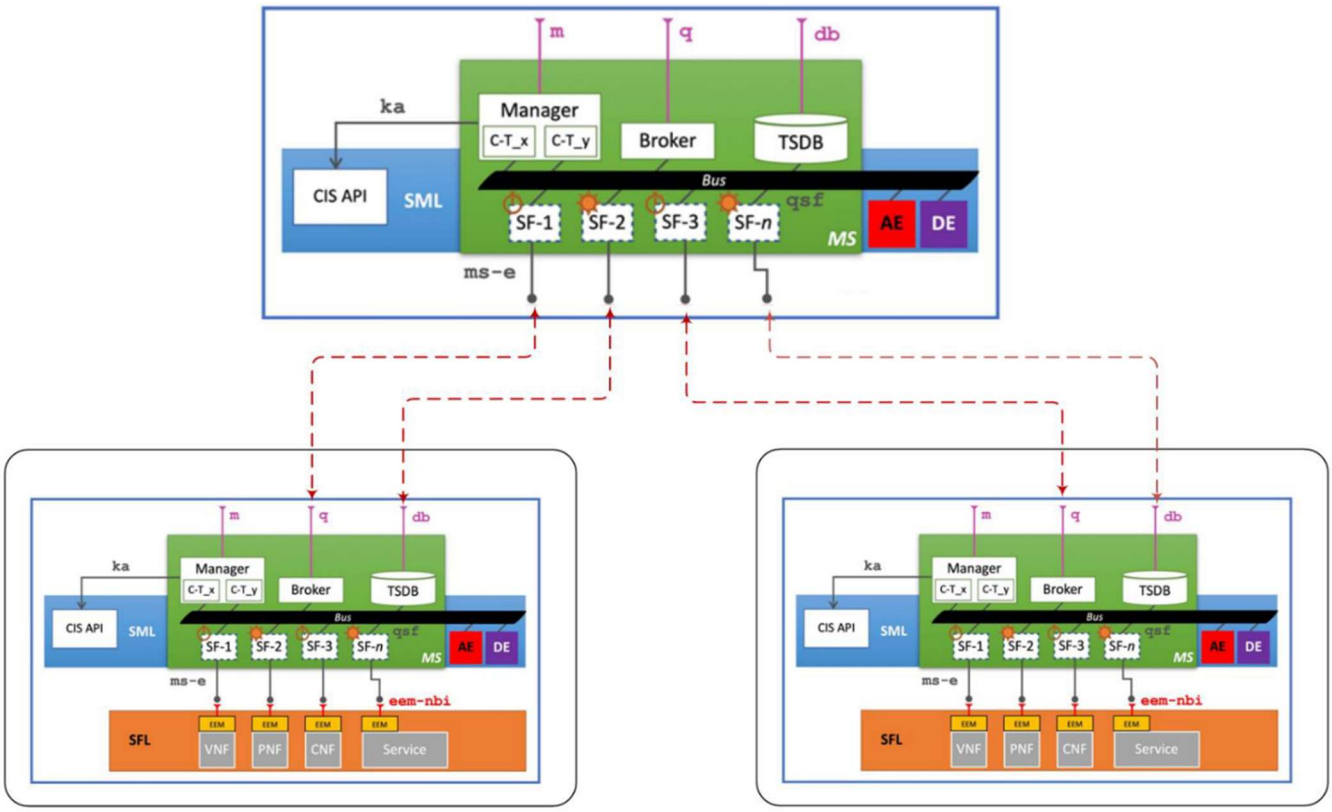

Figure 12: Hierarchal deployment of monitoring systems. [Extracted from D3.2]

Third, also related to the goal of scalability, is *to reduce the footprint* of the system. To reduce the resource footprint MS components are shared among other administrative components i.e., AE and DE. This can be partially seen in Figure 12, where the Streaming Bus also reaches AE and DE. Moreover, the TSDB is also accessible to AE and DE (via the "db" reference point) so, they can use it to store analytics or policies, respectively. Additionally, Sampling Functions can allow more than one TARGET for the same eem-nbi. This means that a single sampling function can sample multiple telemetry data if the corresponding EEM supports it. This feature can significantly reduce traffic between MS and EEMs and improve the scalability of the system. The final strategy to improve the monitoring scalability is the way sampling functions are implemented. Each sampling loop is implemented by a single Docker image. This way, the loop resides inside the Docker image and there is no need to frequently create and destroy the sampling function container. More specifically, the sampling function container is deployed only once and receives samples from the EEM on a regular basis.

5.2 MS instantiation, operation, and data workflow

The MS can be conceived as a cross-domain virtual layer hosted by a NFV IFA 029 compliant PaaS (i.e., Container Infrastructure System (CIS)). The decentralized MS distributes monitoring tasks across multiple levels of the management hierarchy (node, slice, domain, and inter-domain) in a programmable manner.

After being triggered and configured by the user or other components like the AE, the programmable MS entities connect the corresponding infrastructure and functional entities, e.g., VNFs, to gather the requested telemetry data with the desired granularity.

At a glance, the main task of MS is to receive the monitoring requests from the requesters, launch the corresponding sampling loops, and store the monitoring data in the database or streaming bus. This Section details how to operate the MonB5G MS, from its configuration and deployment to getting monitoring data from its broker and TSDB through interfaces "q" and "db", respectively.

5.2.1 DEPLOYING A MS INSTANCE

The distributed architecture of MonB5G MS is implemented on the CIS, realized through Kubernetes, across multiple technological domains. In this section, we describe details of the deploying phase.

5.2.1.1 KUBERNETES CLUSTER

The monitoring system is deployed as a Kubernetes cluster, i.e., the components of the system, including the "manager" and "sampling functions", are actually implemented as pods on the Kubernetes cluster. So, the first step to deploy a MS is to provide a Kubernetes cluster. By default, we consider two-node Kubernetes deployments (a master and a worker) for lightness and simplicity. However, for multi-domain scenarios, where there is the need to monitor data from different technological domains (e.g., RAN, edge, and cloud), a MS could also span multiple Kubernetes nodes. So, the number and location of the Kubernetes nodes actually depends on the intended MS.

Regardless of the number of targeted nodes, a Kubernetes cluster requires that all the nodes have a compatible container runtime (e.g., Containerd or Docker), and a series of tools like the kubelet or kubeadm. It is also important to remark that a node can be either a physical or a virtual machine. For flexibility, in the context of MonB5G trials, we rely on virtualized nodes realized through LXC virtual machines.

5.2.1.2 KAFKA BUS

Once the Kubernetes cluster is ready, we must endow it with a messaging bus. In particular, we chose Kafka, an open-source streaming distributed platform for the development of real-time event-driven applications that can run as a cluster that spans multiple servers. In other words, Kafka deploys a cluster of pods within the Kubernetes cluster.

For deploying the messaging bus in the MS, we use Strimzi[2], which simplifies the process of running Apache Kafka in a Kubernetes cluster. Strimzi provides container images and Operators for running Kafka on Kubernetes. The Operators provided with Strimzi are purpose-built with specialist operational knowledge to effectively manage Kafka. Operators are a method of packaging, deploying, and managing a Kubernetes application. Strimzi Operators extend Kubernetes functionality, automating common and complex tasks related to a Kafka deployment. By implementing knowledge of Kafka operations in code, Kafka administration tasks are simplified and require less manual intervention.

With Strimzi installed, we then create a Kafka cluster, and a topic within the cluster. Then, the Cluster Operator previously deployed when installing Strimzi watches for new Kafka resources. By default, we rely

[2] Strimzi website: https://strimzi.io/

on clusters with one Kafka broker container and one ZooKeeper container. As for the latter, ZooKeeper[3] is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. Figure 13 shows the complete data flow between the containerized elements of the MS through the Kafka cluster.

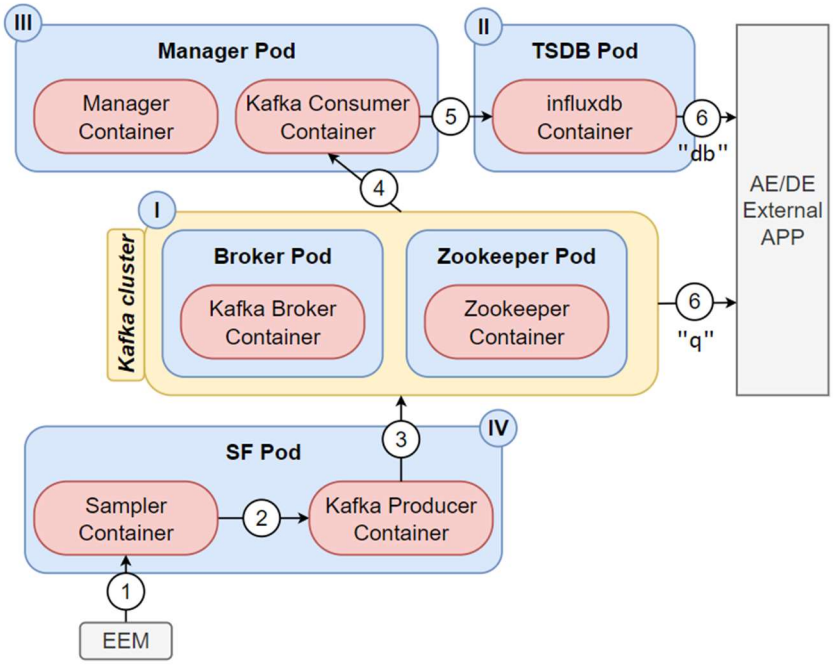

Figure 13: Data flow between the containerized elements of the MS through Kafka cluster. Roman numerals refer to the deploying order, while cardinal numbers refer to the data flow between components.

5.2.1.3 TSDB (COMS) WITH INFLUXDB

InfluxDB[4] is the open-source time series database we use in MonB5G scenarios. InfluxDB is written in the Go programming language for storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet of Things sensor data, and real-time analytics.

In order to deploy InfluxDB in the MS, we must configure a series of Kubernetes elements:

- **ConfigMap**: general configuration of InfluxDB like credentials.
- **StorageClass**: class storage with *retain* reclaim policy.
- **PersitentVolume**: a piece of storage in the Kubernetes cluster to be provisioned for the TSDB. A persistent volume is a volume plug-in that has a lifecycle independent of any individual pod that uses the persistent volume.

[3] ZooKeeper website: https://zookeeper.apache.org/

[4] InfluxDB website: https://www.influxdata.com/

- **PersistentVolumeClaim (PVC)**: a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources.
- **StatefulSet**: unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling. By default, we use only one replica of the influxdb:2.1.0 image for simplicity, but more resilient deployments are easily configurable.
- **Service**: ClusterIP service for internal clients to send requests to a stable internal IP address.
- **Ingress**: exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource. This way, external applications (e.g., AE/DE) can interact with the TSDB through the "db" interface.

5.2.1.4 MS MANAGER

The MS manager is responsible for lifecycle management: triggering, managing, and deleting a monitoring task. It also supports remote configuration of MS operations. There is only one pod of this type in the current implementation (it may be replicated in future versions). This pod consists of two containers, i.e., the "manager container" and the "Kafka consumer" container. The manager container implements the "m" interface and is responsible for LCM of the sampling functions, whereas the Kafka consumer container is responsible for copying the telemetry data into the TSDB.

The manager image for the container is based on a Python script that relies on the fastapi[5] library for providing the "m" interface, which provides a set of actions with regards to sampling functions such as register, configure, launch, stop, and deregister. On the other hand, the Kafka consumer image for container is also based on Python script, which this time provides a way for interconnecting to the InfluxDB acting as a TSDB. For that aim, the script uses influxdb_client[6] Python library. It is important to notice that consumers are subscribers to a specific Topic. Their only task at the Manager pod is to listen for metrics and then transfer them to the TSDB. Each Topic needs a Consumer at the Manager to transfer metrics to the TSDB.

5.2.1.5 SAMPLING FUNCTION

The final components in the MS are the SFs, which are in charge of requesting the data from the EEM and publishing it on the Kafka bus. More specifically, a SF is the agent that implements the Sampling Loop Operations, i.e., triggers an API to collect metrics at a given interval and then passes them to the streaming bus. SFs are also implemented via Docker images, which are realized as containers within the SF pod. The SF can have multiple flavors and should be customized depending on the data source to be monitored (or EEM). More details on how to implement samplers can be found in the next section.

Apart from the sampling container, the SF pod also needs a Kafka producer container, which receives the telemetry data from the sampler container via the virtual file system of the pod and publishes it on the Kafka bus. Like the Kafka consumer, in MonB5G MS we realize the Kafka producer via a Python script, which in this case essentially streams data to the Kafka bus via Python's kafka-python library[7].

[5] Fastapi website: https://fastapi.tiangolo.com/

[6] InfluxDB 2.0 python client website: https://influxdb-client.readthedocs.io/en/latest/

[7] kafka-python website: https://kafka-python.readthedocs.io/en/master/

The pods of a minimal MS instance deployed in a Kubernetes cluster is shown in Figure 14. In this case, netdatacpu is used as an example SF.

```
NAME                                  READY   STATUS    RESTARTS      AGE   IP
monb5g-entity-operator-5554f5d696-b5dlh   3/3     Running   3 (13d ago)   29d   192.168.144.22
monb5g-kafka-0                        1/1     Running   1 (13d ago)   29d   192.168.144.19
monb5g-zookeeper-0                    1/1     Running   1 (13d ago)   29d   192.168.144.21
ms-influxdb-0                         1/1     Running   1 (13d ago)   28d   192.168.144.26
ms-manager-7f6cfbcbd4-2jswz           2/2     Running   3 (13d ago)   28d   192.168.192.9
netdatacpu-665c84d6d4-ddpcg           2/2     Running   3 (13d ago)   28d   192.168.144.27
```

Figure 14: Pods of a minimal MS instance

5.2.2 *OPERATING A MS INSTANCE*

Once a MS instance is ready, the user or an external application like the AE or DE can interact with it via three northbound interfaces (NBI):

- **m**: sampling loop creation and management via an HTTP API.
- **q**: Kafka bootstrapping used to subscribe/publish or create topic.
- **db**: API endpoint for MS persistent storage. Consumers use this NBI to populate the TSDB.

5.2.2.1 BUILT-IN MS COMPONENTS

Before discussing the operations enabled in the MS, let us briefly review the components that are included by default (or off-the-shelf) in a MS instance. As shown in Figure 15, the MS provides by default the Kafka cluster, the manager, the TSDB, and the Kafka producer of the SF pod. The missing MS component, the sampler container should be implemented according to the data source to be monitored so to interact with the EEM. Likewise, external components like the sampling loop configuration file or the AE/DE are not in the scope of the MS.

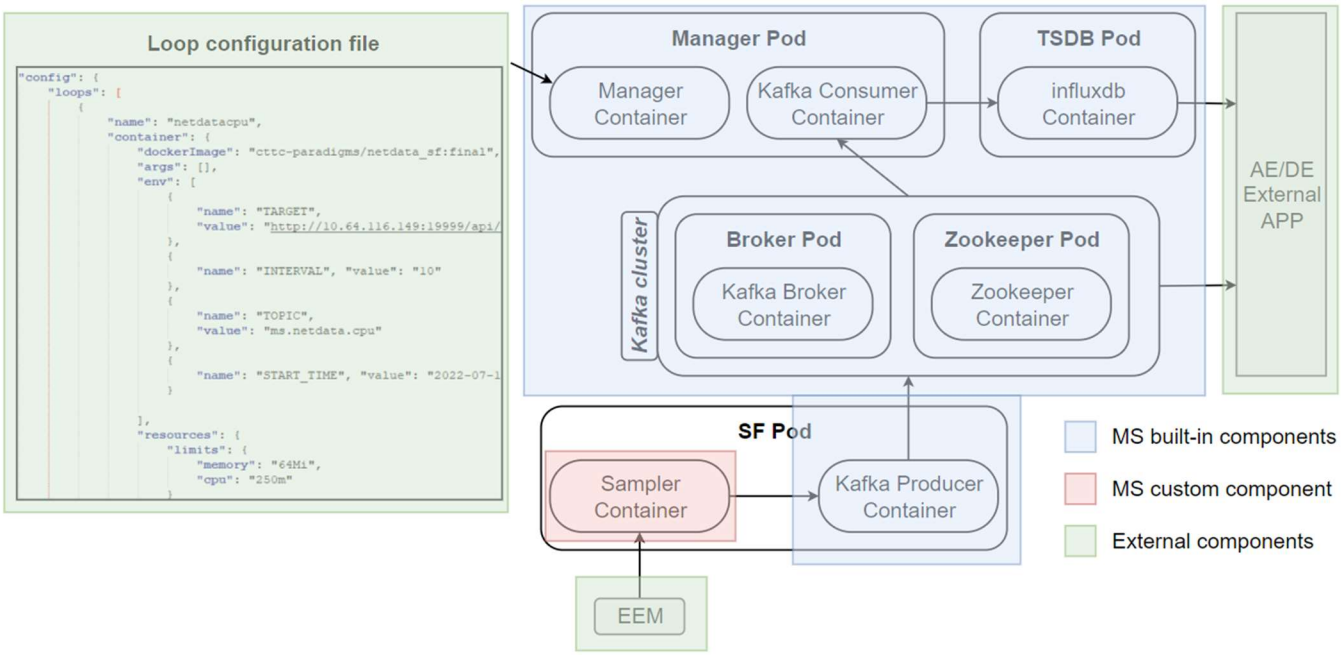

Figure 15: MS built-in components, MS custom components and external components

5.2.2.2 MS OPERATION WORKFLOW

The goal of this section is to describe how to operate the Monitoring System from the point of view of any interested user. As illustrated in Figure 16, there are 4 main steps, which we depict next.

First, the user creates a loop configuration file for setting up the sampling function. This configuration indicates different relevant parameters:

- Name of the SF
- Docker image of the built SF
- Resources assigned to the container, including *memory* and *CPU*.
- Environment parameters like the EEM *target* (normally a URL), the *interval* between measurements, the Kafka *topic*, and the monitoring *start time*[8].

Second, this loop configuration file is submitted (registered) to the MS manager through an API client application developed in Python. Then, the user can initiate (launch) the sampling loop also via the API client. Upon the reception of the *launch* instruction, the Manager deploys the sampling function as described in the loop configuration file.

[8] This parameter allows configuring scenarios where we might want to have the MS ready but start monitoring from a particular time instance.

Third, the deployed SF starts pulling metrics from the specified targets or EEMs, which can be of different flavors (e.g., CNF, VNF, or PNF). Notice that these SFs may gather information from different domains. In particular, in the example in Figure 16, we show three domains, each feeding a different number of SFs.

Finally, in the **fourth** step, external agents like the AE or DE can get the monitored directly from the Kafka Cluster (via the "m" interface), or from the TSDB (via the "db" interface).

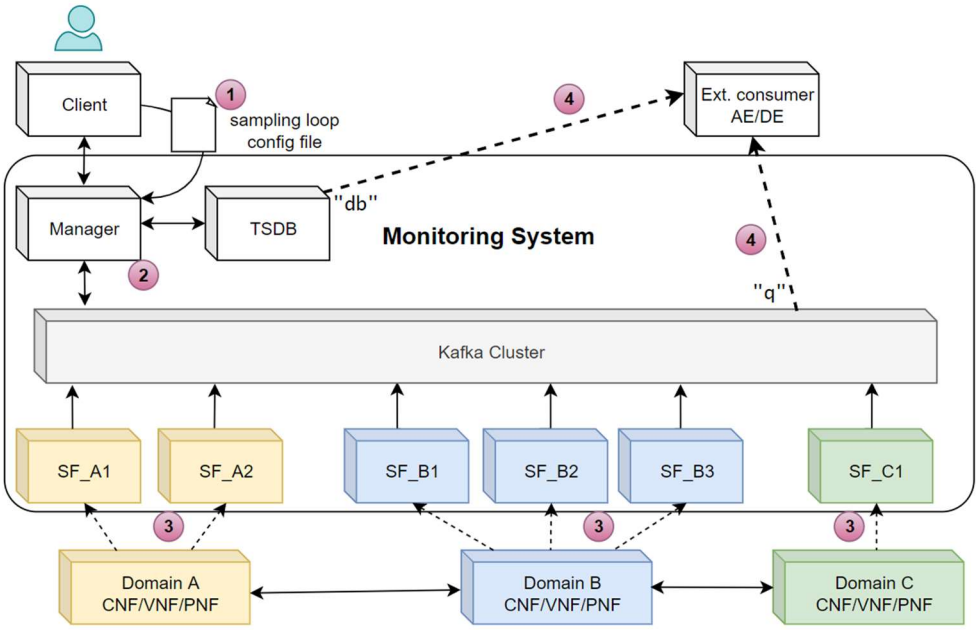

Figure 16: Monitoring System operation workflow

5.2.3 *USER DEVELOPMENTS*

As shown previously in Figure 15, there are some components which a MS instance does not build-in by default. In general, the MS user is expected to provide (and implement if needed) the external components like the EEMs (or data generation sources), and the sampler for the SF. Table 5 summarizes the possible cases arising from the availability of the EEM and SF.

Table 5: User development needs according to EEM and SF availability

Case	EEM	Sampler	Example	Expected user implementations
#1	**Available** *(by CTTC)*	**Available** *(by CTTC)*	EEM: Amarisoft API Sampler: Python API client	- Sample loop configuration file. - Kafka and/or TSDB clients.
#2	**Unavailable**	**Available**	EEM: Netdata server Sampler: Netdata client	- Sampler implementing communication protocol to EEM. - Docker image (Dockerize sampler).

				- Sample loop configuration file. - Kafka and/or TSDB clients.
#3	**Unavailable**	**Unavailable**	EEM: any data source Sampler: any sampler	- Implement EEM to generate data samples. - All steps in **Case #2**.

In Case #1, EEM and Sampler are provided. CTTC already provides Netdata collectors and Amarisoft Callbox/Simbox parameters. The user needs to provide the sample loop configuration file specifying start time of the sampling interval, end-point address of EEM, topic that the sample data should be published in, and the Kafka client / a TSDB client to read the data, process it, etc.

In Case #2, the sampler is not ready but there is EEM. In this case, the user needs to develop a sampler that implements the protocol to communicate the EEM, dockerize the sampler and provide the docker image, and create Sampling loop configuration file.

Finally, in Case #3, develop an EEM that can measure parameters, and all the steps in Case #2.

5.3 Use case examples

5.3.1 SINGLE-DOMAIN SLICE MONITORING: AN EXAMPLE ON ROUND-TRIP-TIME MONITORING

In this Section, we will cover how a MS instance is set up for measuring a simple parameter in a single-slice domain (Figure 17). Throughout this example, we cover the different steps required to the MS user so to arise clarity on the operation flow.

Assume a user wants to monitor the round-trip-time (RTT) between a source VNF and destination VFN (e.g., from a 5G UPF to a given virtual server in a MEC host). These kinds of measurements are really useful for monitoring not only the latency, but also the connectivity status.

We categorize this example as single-domain monitoring since we assume that both VNFs are within the same technological domain, or even the same physical machine (PNF); in this case, a MEC host deployed at the edge. So, the MS instance for in this particular can take a minimal flavor and can rely on just one SF.

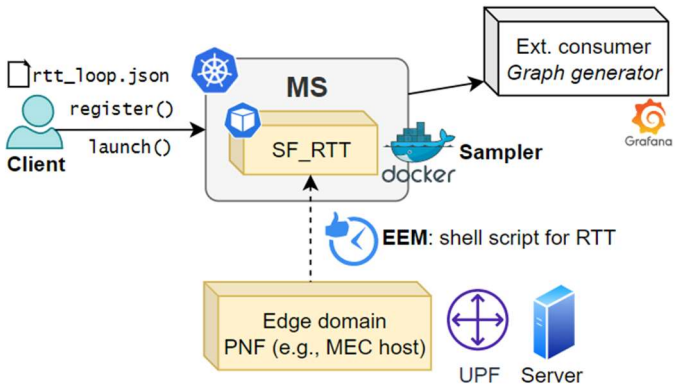

Figure 17: Single-domain MS example

We assume that there is not any EEM nor sampler available for monitoring this parameter. That is, we are in Case #3 of Table 5: User development needs according to EEM and SF availability. Therefore, the user needs to:

1. Develop an EEM to measure the RTT parameter.
2. Develop a sampler for pushing the RTT measurements to the Kakfa bus.
3. Containerize the sampler, so it can be deployed in the MS Kubernetes cluster.
4. Provide a loop configuration file for configuring general monitoring parameters like the interval between two consecutive measurements.
5. Deploy the sampling loop through the Kafka client by interacting with the Manager API.
6. *Consume the data*: this step is beyond the scope of the MS operation and is related to consumers like the AE or DE. For this example, we will consider a simple consumer that just plots the MS data.

First, we implement a bash script that computes the RTT from the UPF to the server based on the results provided by the ping utility. The EEM is a shell script that runs at the UPF and its code is as follows:

```
ping -c 10 <DST_VNF> | grep rtt | sed 's/rtt //' | sed 's/ = /\//' | sed 's/ ms//' | awk
'BEGIN{FS="/"} {for(i=1;i<=4;i++) printf("\"%s\": %s%s\n", $i, $(i+4), (i<4)?",":"")}'
```

This generates the following JSON output:

```
{"min": 12.367, "avg": 12.555, "max": 13.219, "mdev": 0.239}
```

Now, the parameter can be properly monitored, but the data is kept locally. Therefore, we also need to provide an API so the sampler can request this data. In this example, we rely on HTTP.

Second, we must develop the sampler to be run in the SF pod. The sampler will connect to the endpoint (EEM) later configured via the loop configuration file and take measurements also according to the loop configuration file. In order to make it suitable for the MS, the sampler must be a containerized image in order to be deployed as a container in the SF pod. For this, we rely on Docker. Notice that the created image should be available for the nodes in the k8s cluster, so it is a best practice to store the image in a remote registry like Docker Hub[9].

Next, the user must define the general behavior of the SF through the loop configuration file. As explained before, this file specifies parameters like the start time of the sampling process, endpoint address of EEM (the TARGET), the sampling interval, the sampler image, the Kafka topic, and the required resources to run the sampler container. The loop configuration file used in this example is shown in Figure 18: Example loop configuration file for RTT monitoringFigure 18.

[9] Docker Hub website: https://hub.docker.com/

```
"config": {
    "loops": [
        {
            "name": "rtt",
            "container": {
                "dockerImage": "cttc-paradigms/rtt_sf:final",
                "args": [],
                "env": [
        {
                    "name": "START_TIME",
                "value": "2022-09-16 11:16:19"
            },
                    {
                        "name": "TARGET",
                        "value": "http://10.1.20.35:80"
                    },
                    {
                        "name": "INTERVAL",
                        "value": "30"
                    },
                    {
                        "name": "TOPIC",
                        "value": "ms.rtt"
            }
                    ],
                    "resources": {
                        "limits": {
                            "memory": "64Mi",
                            "cpu": "250m"
                        }
```


Figure 18: Example loop configuration file for RTT monitoring

Then, the configuration file is submitted to the manager through the client. In particular, the user needs to register the loop, launch it, and (if needed) stop it.

Finally, the data is periodically gathered by the MS and exposed through the Kafka cluster and/or the TSDB. So, the consumer is ready to get such data. As explained before, the consumer can be of different types, normally an AE or DE that generates value from the data (e.g., decision making). Nonetheless, in this example, we showcase a consumer that simply plots the data without further processing. In particular, we rely on Grafana to plot the RTT parameter as shown in Figure 19. Notice that being able to appropriately represent the gathered data allows creating control dashboards of interest. For instance, we may trigger alarms whenever the RTT exceeds the 3 ms thresholds (shadowed in red in the Figure 19).

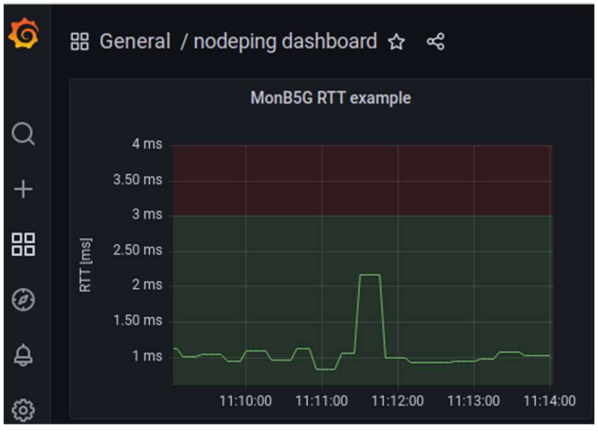

Figure 19: Grafana dashboard for the single-domain slice use case

5.3.2 *MULTI-DOMAIN SLICE MONITORING: MONITORING A MULTI-DOMAIN 5G NETWORK*

In the previous Section, we saw how a user can operate a single-domain slice. Nonetheless, the MS can also enable multi-domain slices where the data source (e.g., VNFs or PNFs) are spread across multiple technological domains. In this regard, we cover in this Section a use case for providing end-to-end monitoring in a 5G network. The setup is depicted in Figure 20. There are four domains: the UE, RAN, edge, and cloud.

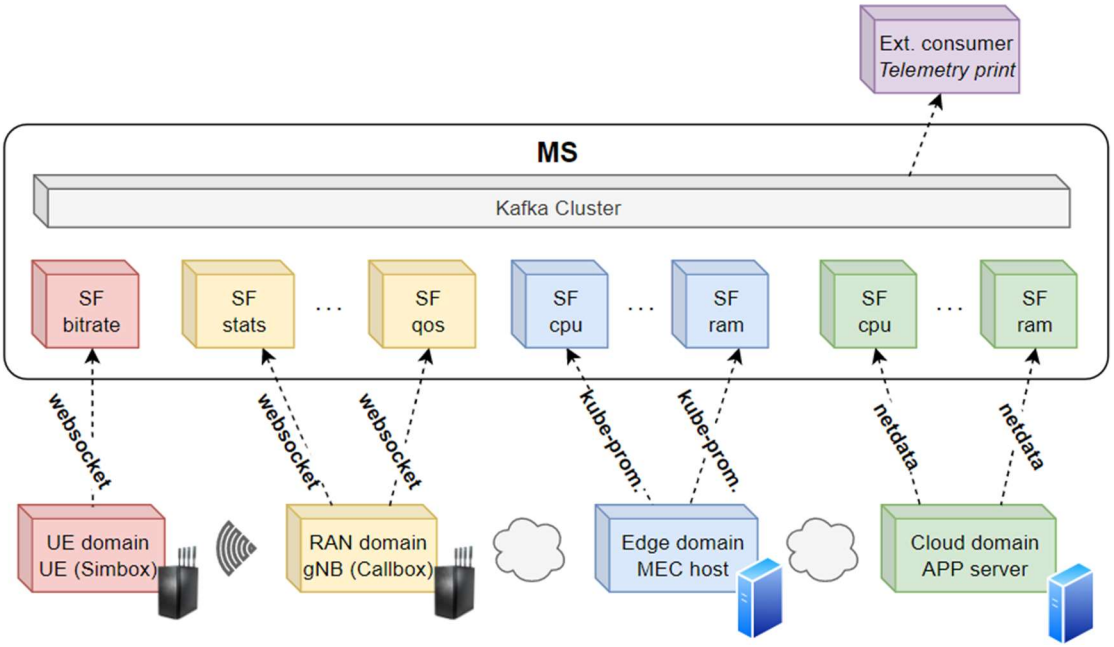

Figure 20: Multi-domain slice for 5G end-to-end monitoring

Notice that each domain has its corresponding sampling functions, which may monitor the same type of parameter (e.g., CPU and RAM for edge and cloud domains), but use different communication protocols and EEM tools (e.g., WebSocket for Amarisoft APIs, and kube-prometheus or NetData for the edge and cloud). In this example, we propose a variety of SFs to monitor different parameters, ranging from RAN-specific like the bitrate or QoS flows metrics, to general infrastructure metrics like RAM or CPU.

It is important to notice that by default the MS does NOT provide any compression or data filtering on top of the raw data provided by the EEMs. That is, the MS runs SFs (each realized as a pod in the K8s cluster) that periodically sample the metrics according to the loop configuration entered by the MS user. So, these SFs are *independent* containers, meaning that each will generate its own kind of message output periodically published on the bus. While in this way the MS completely fulfils with the envisioned requirements, it is up to the user (or, generally, external consumers) to handle the data in a proper way for their algorithms. For instance, a reinforcement learning algorithm usually expects an *observation* of the environment at a given

time. In such a case, it is up to the AE/DE to prepare data for converting raw measurements to observations/states.

Finally, Figure 21 and Figure 22 show examples of a partial monitoring sample (mixing different domains), and a specific example of the parameters provided by the RAN SF "stats", showcasing the richness and variety of these type of metrics.


```
1, 'nr_rrc_security_mode_command': 2, 'nr_rrc_dl_information_transfer': 3, 'nr_rrc_ue_capability_enquiry': 1, 'nr_rrc_reconfiguration
_complete': 3, 'nr_rrc_setup_complete': 2, 'nr_rrc_security_mode_complete': 2, 'nr_rrc_ul_information_transfer': 4, 'nr_rrc_ue_capabi
lity_information': 1, '5gs_nas_registration_accept': 1, '5gs_nas_registration_complete': 1, '5gs_nas_service_request': 1, '5gs_nas_se
rvice_accept': 1, '5gs_nas_configuration_update_command': 1, '5gs_nas_authentication_request': 1, '5gs_nas_authentication_response':
1, '5gs_nas_security_mode_command': 1, '5gs_nas_security_mode_complete': 1, '5gs_nas_ul_nas_transport': 1, '5gs_nas_dl_nas_transport'
: 1, '5gs_nas_pdu_session_establishment_accept': 1}, 'errors': {}}, 'cells': [{'path_loss': -8, 'dl_bwp_ids': [0], 'ul_bwp_ids': [0],
 'index': 0, 'pci': 500, 'rsrp': -52, 'rsrq': -10.7, 'snr': 33.3}]}, 'emm_state': 'registered', 'pdn_list': [{'apn': 'default.mnc001.m
cc001.gprs', 'ipv4': '192.168.2.2'}]}], 'time': 6451.857, 'target': 'ue', 'sample_time': 1657625983.586677, 'cnt': 425}, 'netdata.ram
': {'target': '10.64.116.149', 'sample_time': 1657625988.554225, 'cnt': 425, 'free': 5736.984, 'used': 326.9805, 'cached': 1224.9062}
, 'netdata.cpu': {'target': '10.64.116.149', 'sample_time': 1657625988.516096, 'cnt': 425, 'user': 0, 'system': 2.040816, 'iowait': 0
}, 'netdata.bw': {'target': '10.64.116.149', 'sample_time': 1657625989.498806, 'cnt': 425, 'received': 0, 'sent': 0}}
--------------------------------------------------
DL =  150110 , UL =  2420 , RAM =  5736.984
```


Figure 21: A monitoring sample of the multi-domain slice use case

Figure 22: Detail of the output generated by SF "stats"

6 Cloud-native implementation AE

The MonB5G Analytics Engine (AE) is designed to analyze the status of network slices using telemetry data collected by the MS. Its outputs, i.e., the analysis results, are then reported to DE as key indicator to learn from and infer actionable decisions to maintain and optimize the slice performance defined in SLAs. End-to-end KPIs at the slice level have been defined and include, for example upstream/downstream throughput for NSI, average end-to-end uplink/downlink delay, virtualized resource utilization per NSI, etc. The number of slices, as well as the amount of data collected when automatic slice redeployment is enabled, are important factors that affect the effectiveness and efficiency of AE so a scalable solution must be developed.

This chapter includes several prediction methods for AE, which has been deployed as a docker container and are described in the following sections. To achieve the goal of scalability, we employed several strategies in the design and implementation of the AE.

6.1 Integration of Different MonB5G AEs with MS

6.1.1 FEDERATED LEARNING APPROACH

The main goal is to deploy a cloud-native approach of FL agents to check the feasibility of our algorithm in the actual scenario and prove the scalability of our proposed policy. To emulate the cloud-native deployments, we use Docker compose tool. The reason behind choosing Docker-compose for deployments is that it usually runs on top of Kubernetes. Also, these kinds of implementations expect to be supported by Container Orchestration Engines (COE) offered Slices as Platform as a Service (PaaS), as specified in ETSI NFV [5] which is our future target to implement.

6.1.1.1 ARCHITECTURE

Figure 23 illustrates the cloud-native implementation of FL agents where FL Server (OSS server) along with one module who is responsible for overall orchestration is besides in one docker container. On the other hand, several AE's, in this case, clients simultaneously run by using Docker compose tool. Through REST API, the Server and clients can communicate with each other. FastAPI[10] as a REST API is used in our implementation because it is a modern, open-source, fast, and highly performant Python web framework used for building Web APIs with Python 3.6+ based on standard type hints.

[10] https://fastapi.tiangolo.com/

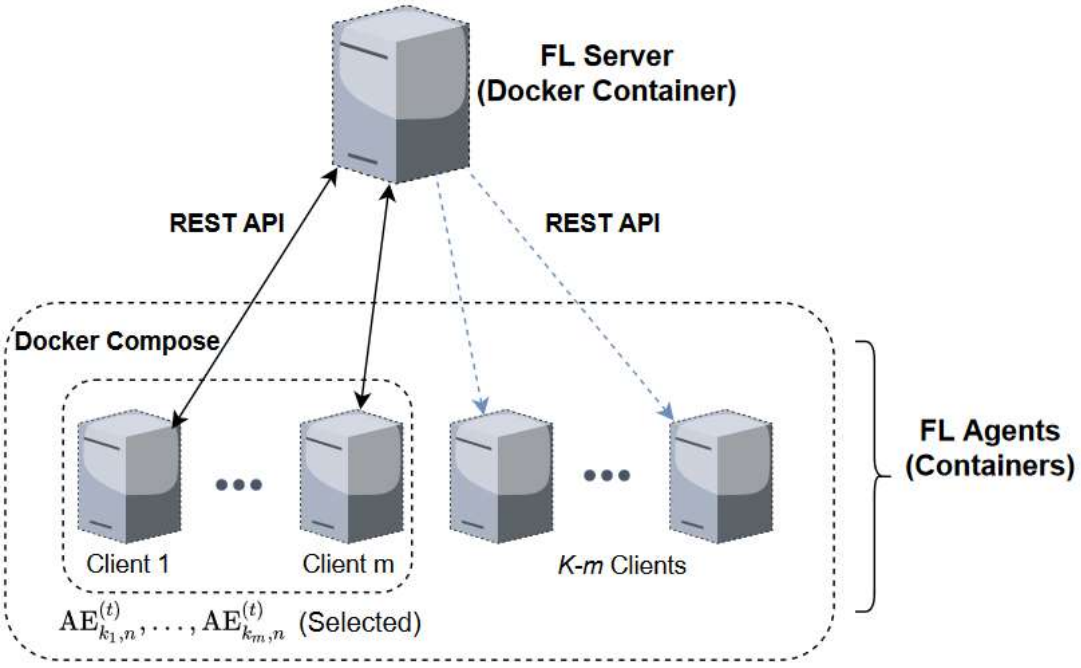

Figure 23: Cloud-native implementation of scalable FL agent selection

6.1.1.2 COMMUNICATION PROCESS

There are two main modules: the server module, which is responsible for conducting training among clients, and the other module, which is used by all the clients who will participate in the overall FL training process.

Communications are performed using the HTTP protocol through several REST interfaces among the server and client nodes.

Server Side (4 APIs):

The server container contains the main.py module that acts as a controller, the REST API that can communicate with other REST APIs, and the Server class responsible for all the logic.

It has the following set of basic REST operations:

- **POST/client:** Registering clients with the Server *(from Client to Server).*
- **GET/select clients:** Initiate policy for selecting clients and corresponding FL training *(from Admin to Server)*.
- **POST/SLA:** Clients send their SLA violation rate to the Server node *(from Client to Server).*
- **PUT/model-weights:** Clients send calculated model parameters to the Server node *(from Client to Server)*.

Client side (3 APIs):

The client container contains the app.py module responsible for REST API communications and the client module, where the Machine Learning models are implemented. It has the following set of basic REST operations:

- **PUT/SLA:** Server requests each of the clients to calculate their SLA violation rate *(from Server to Client)*.
- **POST/training:** Server requests the selected clients to start FL training *with new model weights (from Server to Client)*
- ***PUT/worker_model:*** *Update client initial model parameters. (from Server to Client)*.

6.1.1.3 WORKING PROCEDURE OF THE OVERALL NETWORK

For running the overall network correctly, the system's first requirement is that the server node is in running mode, and at least one client node is available for training. Following the mentioned way, the overall system will work.

- At first, all clients should know about the server node's IP address, and they register with the server node through the *POST/client* request with their own IP address.
- After registration, the server node sends requests to all registered clients to start the proposed clients selection policy through *POST/select-client* request.
- Then, all clients calculate and send their associated SLA violation rate value to the Server through *PUT/SLA* and *POST/SLA*.
- Next, the Server generates the probability distribution of all clients using *softmin* function and select clients for training by using *np.random.choice* functions.
- Finally, the Server send *POST/training* requests to the selected clients and start FL training.
- Later, model weights of each clients send to the Server through *PUT/model-weights*, and then the Server calculate the average of model weights and update the overall system and repeat the same procedure for upcoming FL rounds.

A sequence diagram of the above steps is summarized in Figure 24.

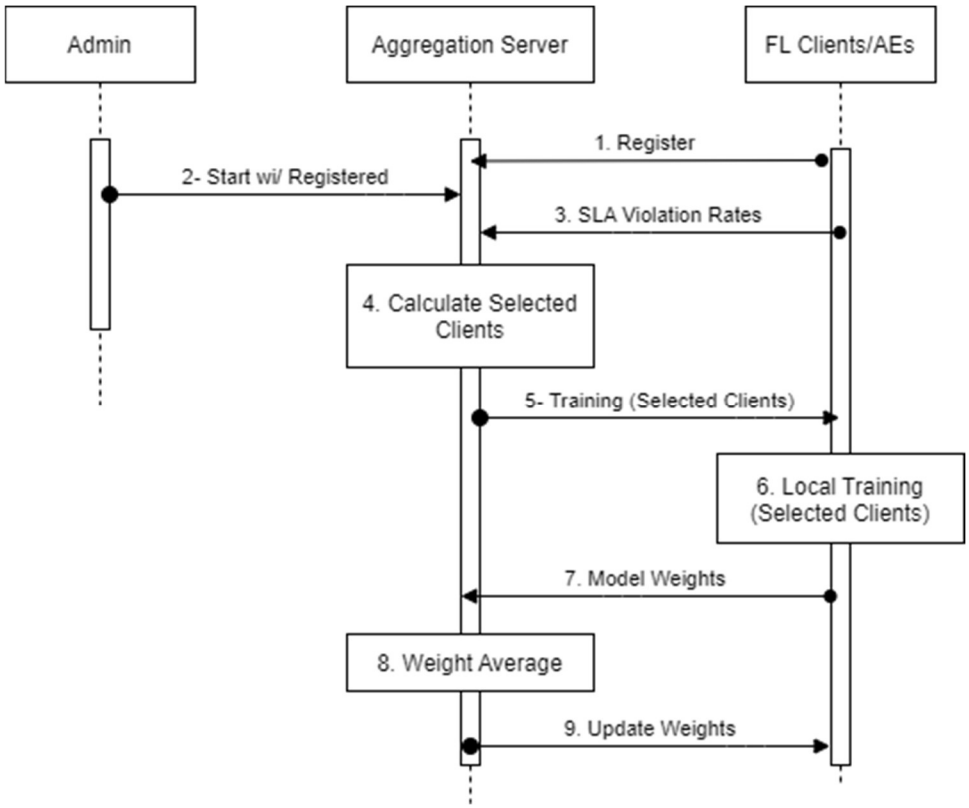

Figure 24: High level workflow of the cloud-native implementation of stochastic FL approach

6.1.1.4 VISUALIZATION OF CLOUD NATIVE IMPLEMENTATION OF SCALABLE FL AGENT SELECTION

For real-time visualization of changes in NMSE and SLA violation rates over the FL rounds during training for both policy-based and non-policy-based approaches, we used the ELK stack. As shown in Figure 25, Logstash[11] is used for the log transformer (as given in step-3), Elasticsearch[12] is used for the data indexer (as given in step-4) and Kibana[13] is used for the visualization (as given in step-5)) and Apache Kafka is used as message bus (as given in step-2) to received updates from the FL aggregation server (as given in step-1)

[11] https://www.elastic.co/logstash/

[12] https://www.elastic.co/

[13] https://www.elastic.co/kibana/

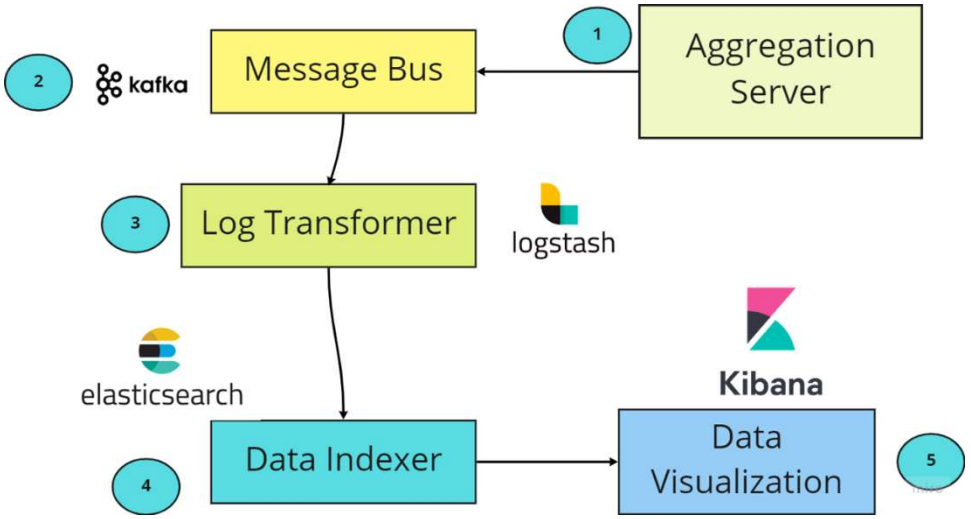

Figure 25: LK stack integration with FL Policy and Non-Policy Approaches for visualization purposes

Figure 26 - Figure 28 shows all metrics discussed above as well as the demo visualization dashboard for the eMBB slice under consideration. Figure 26 shows the number of clients as well as the selected number of clients for the considered non-policy-based (vanilla or classical FL) and policy-based FL approaches. Figure 27 shows the convergence time over the number of rounds. Figure 28 shows the NMSE values over the number of rounds. Finally, Figure 29 shows the SLA violation rate values over the number of rounds.

Figure 26: Kibana Dashboard showing the number of clients and selected number of clients for policy-based and non-policy-based FL during training phase

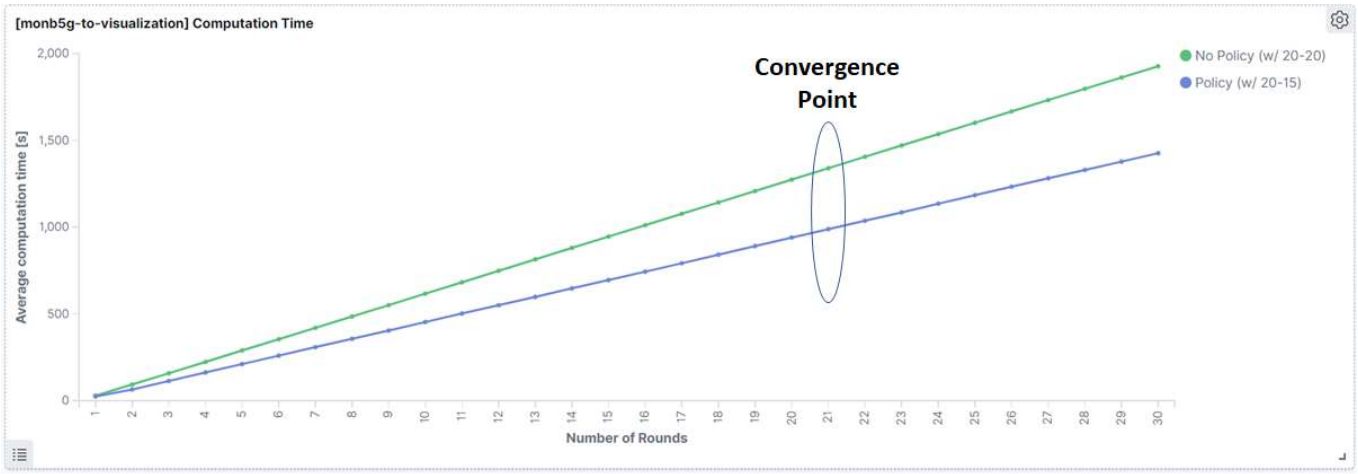

Figure 27: Convergence time versus FL rounds for non-policy and policy-based approaches during training phase

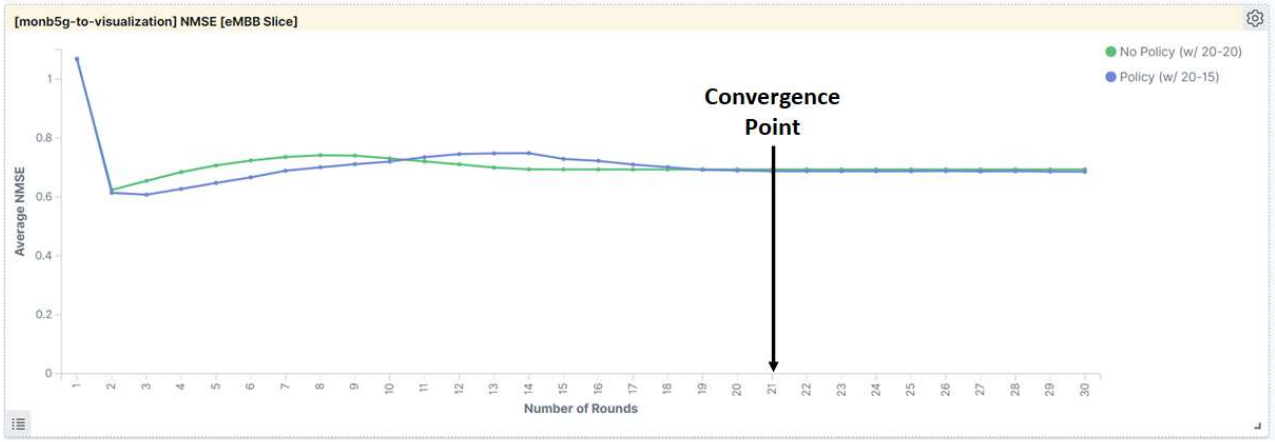

Figure 28: NMSE versus FL rounds for non-policy and policy-based approaches during training phase

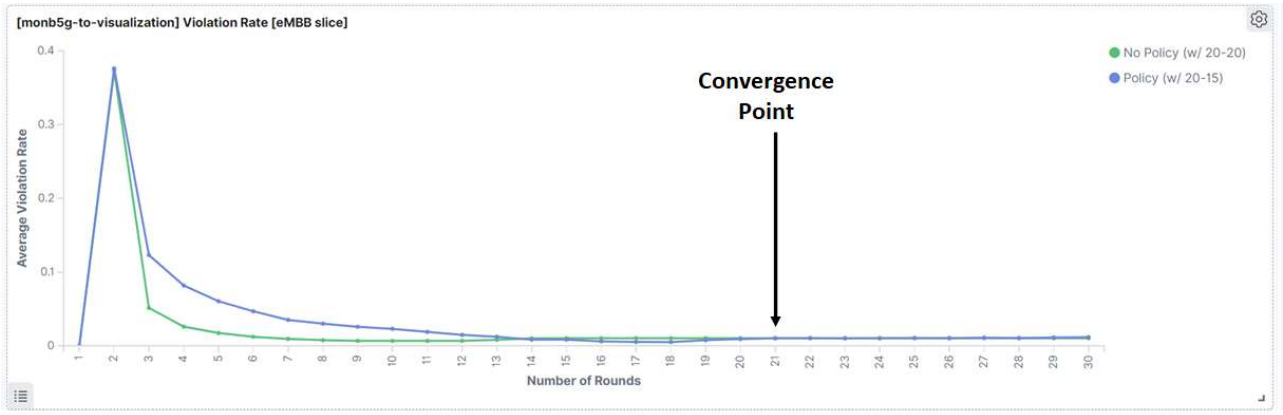

Figure 29: SLA Violation Rate versus FL rounds for non-policy and policy based FL approaches during training phase

Based the visualization and experimental platform described above, several calculations can be performed. First, the point of convergence occurs at approximately round 21 which corresponds to a violation rate of 0.01 (as shown in Figure 29) which is the upper bound of SLA violation rate threshold and and average NMSE (Normalized Mean Squared Error) value of 0.687 (as shown in Figure 28) and a convergence time of 988.193 seconds (as shown in Figure 27) for the considered eMBB network slice. In contrast, non-policy FL (or vanilla/traditional FL) takes about 1339.46 seconds to converge to the same values of NMSE and violation rates at approximately round 21. This makes the proposed stochastic policy-based FL approach converge around 158.46 seconds faster than classical FL approach.

6.1.1.5 OVERHEAD GAIN CALCULATION:

Table 6 shows the overhead induced by the baseline fully centralized SLA-constrained deep learning (CCL) [4] the non-policy StFL introduced in [3] and the policy-based strategy [8]. For the computation of the overhead, we have considered that both the datasets and update models are coded in 32 bits. In the uplink between the clients and the aggregation server, the approximate overhead can be calculated as

Uplink Overhead (bits) ≈ # FL Rounds x # Selected Clients x # Weights x # 32 bits x # features

This means that a communication round in the federated setup is equivalent to 100 epochs over a batch in the centralized one. Starting from the convergence point of StFL, more than 10 times overhead reduction is obtained in comparison with the centralized SLA-constrained algorithm.

Table 6. Overhead comparison between centralized solution and federated learning-based algorithms

Rounds	21	50	60	70	80
Overhead CCL (KB)	1875				
Overhead non-policy StFL (KB)	44.3	105.5	126.6	147.7	168.8
Overhead policy-based StFL (KB)	33.2	79.1	94.9	110.7	126.6

Considering from the above figures that convergence occurs at round 21 (SLA violation reaches 0.01 and Loss variation is low),

- Policy uplink overhead is ≈ 33.2 KB
- No-Policy FL uplink overhead is ≈ 44.3 KB

For this reason, the reduction in overhead compared to non-policy FL algorithm (or vanilla/traditional FL) is around 25%. Note that higher overhead gains are obtained in comparison to centralized learning approaches (in the case when all data are transmitted to centralized server) as shown in Table 6. Finally, demo video of this integration can also be found in MonB5G's official Youtube channel[14].

6.1.2 *LOCAL SLICE KPI PREDICTION*

To discover hidden associations among nodes, a graph learning layer computes the graph adjacency matrix, which is later used as an input to all graph convolution modules. The graph learning layer learns a graph adjacency matrix capable of capturing the hidden relationships among the time series data. By propagating the information through structures, graph neural networks allow each node in a graph to know the context of its neighbors. In our case, for the slice KPI defined as sliceLatency, we compute the correlation matrix of slice latency between different resource KPIs e.g. (CPU, RAM, Bandwidth, and Storage) which represents our adjacent matrix as time-varying features and the connections between them are the edges. These form the node features in the network.

The sequences for the input model as shown in Figure 30 are passed through the Recurrent Neural Network GRU layers, while the correlation matrices are processed by GCNs. Here the initial node features (time series data points) are provided as an input to the GCN and then, the node embeddings are computed by applying

[14] Online: https://www.youtube.com/watch?v=fHuwOFaxJIc, Available: October-2022

the series of convolutional modules. We first use the historical time series data as an input and the graph convolution network is used to capture topological structure of network to obtain the spatial feature. Second, the obtained time series with spatial features are input into the gated recurrent units where the units capture the temporal features. Finally, we get results through the fully connected layer to predict the slice KPIs.

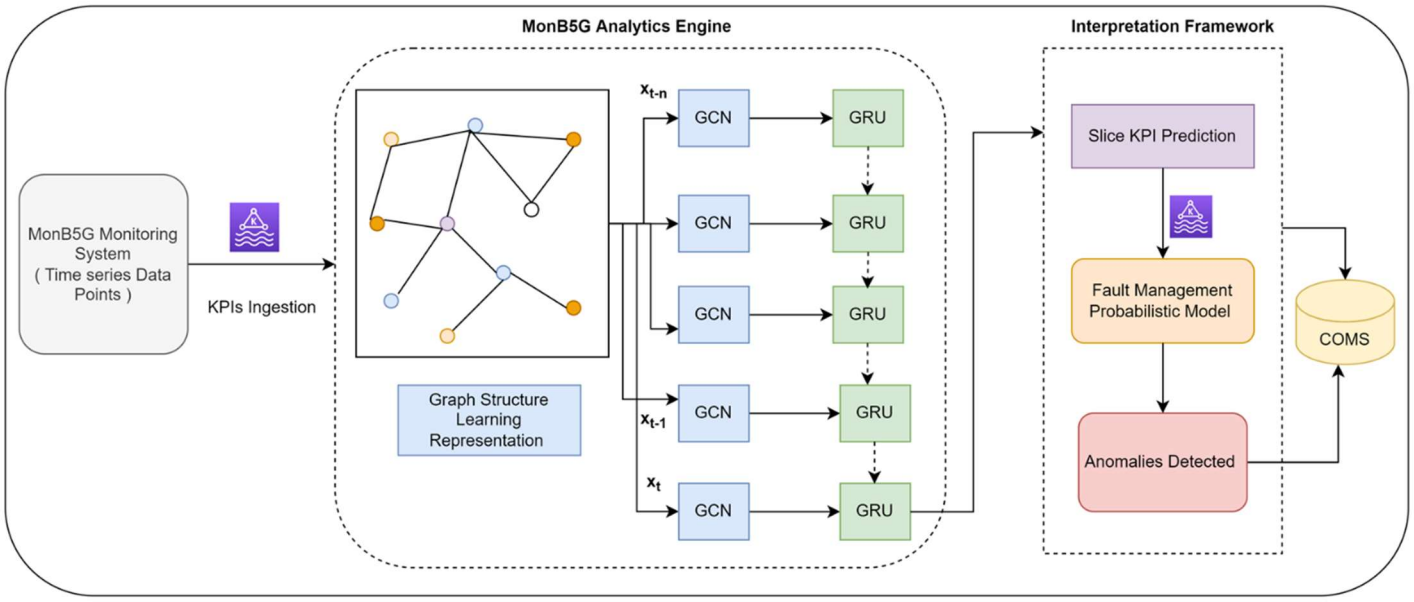

Figure 30: Interpretable B5G Analytics Architecture

6.1.2.1 EXPERIMENT AND EVALUATION

The initial experiments were performed on a computational platform which includes Intel® Core™ i7-8650U CPU @ 1.90GHz × 8, 25.8 GiB memory, Ubuntu 18.04.4 LTS operating system. The Keras python library 2.4.3 was used for running on top of a source build of Tensorflow 2.3.0.

The architecture makes use of the following hyperparameters: learning rate, batch size, training epoch, and the number of hidden layers. In the experiment, we set the learning rate to 3e-4, window size of 24, batch size to 32 and the training epoch to 100. We choose the number of hidden units from [64,100,250] and analyze the change of prediction accuracy. A set of 14,652 data points were generated by the simulator as an input data to train for referenced architecture, and further slice KPI predictions been used for anomaly detection for providing an interpretable anomaly explainer. The model optimizes the Mean Squared Error (MSE) loss using Adam optimizer [28], use early callback mechanism stopping with patience of 15 and relu activation function.

Figure 31 and Figure 32 represent the performance of the compared algorithms trained using multivariate KPIs to predict sliceLatency KPI in a multivariate network data environment. We show the loss curve

compared to different models, and observe that the model with Graph GCN generalizes well over the training data as compared to the neural network, CNN, GRU, and GRU with CNN models as shown in Table 7: MSE Comparison for Different Models. The key idea behind GNNs is to aggregate feature information from nodes' local neighbors via neural networks. Figure 33 shows the visual representation of the normalized predicted sliceLatency KPI.

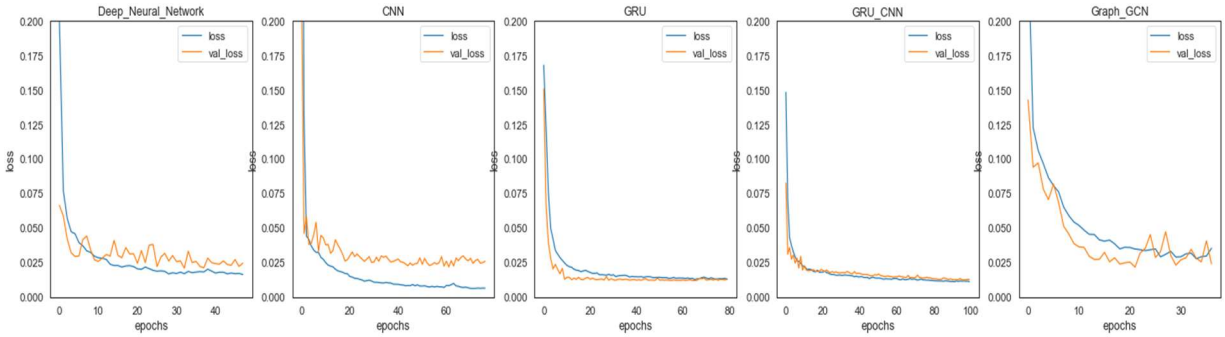

Figure 31: Performance Evaluation of Different Algorithms

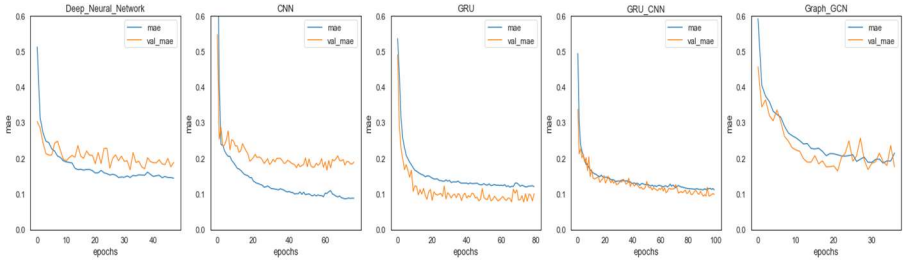

Figure 32: Model Evaluation based on MAE

Table 7: MSE Comparison for Different Models

Algorithm	MSE
Neural Network	0.030909
CNN	0.032368
GRU	0.036695
GRU- CNN	0.032652
Graph GCN with GRU	**0.022974**

Our results show that our AE architecture (joint learning framework based on GCN combined with GRU based architecture trained using network slice KPI data) helps to optimize the modeling of KPI data.

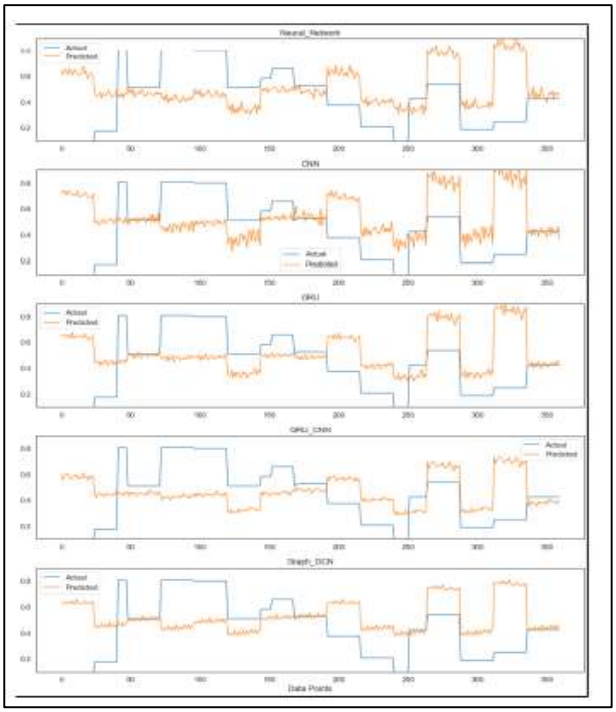

Figure 33: Slice KPI Prediction comparing Different Models

6.1.2.2 FAULT MANAGEMENT PROBABILISTIC MODEL

Fault Management has been a fundamental part of network management, included in the FCAPS operations (Fault, Configuration, Accounting, Performance and Security). It aims to detect and eliminate any malfunctions that have occurred in the monitored systems to prevent the degradation of the provided services. In 5G/B5G networks, because of the high degree of flexibility and change in the network, faults must be carefully considered within the particular context in which they appear [6].

In particular, deep learning algorithms are capable of processing and learning from large data sets efficiently. They can be applied to very large data sets that are generated in large operator's networks to find useful insights. Here, we leverage Recurrent Neural Networks and Hidden Markov Model [7] to estimate the probability of a sliceLatency KPI sequence using the probability distributions as shown in Figure 34.

A language model for sequences specifies a probability distribution for the next in a sequence given the set of previous sequences. The Gated Recurrent Unit Based Neural Network is trained here to produce this probability distribution using a training set of known normal sequences.

Note: $p(x_i \mid x_{1:\, i-1})$ is the probability of the integer xi occurring after the sequence $x_{1:\, i-1}$.

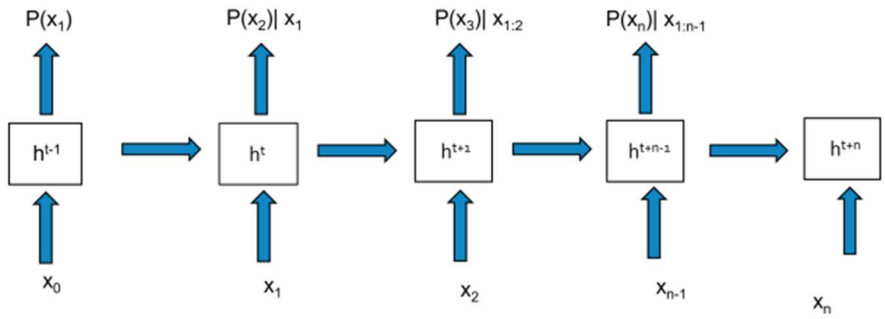

Figure 34: Probabilistic Classifier

Given a set of normal sequences, the anomaly detection algorithm evaluates the test instances as anomalous or normal. As the algorithm implemented is cost efficient (reduced training and testing times), and it consumes much less computational resources in comparison to the traditional deep neural network algorithms. The following steps specify the algorithm for anomaly detection and its evaluation on validation data.

1. **Training**

- The RNN based model is trained in minibatches on normal sliceLatency sequences in the dataset (no outliers).
- The model is trained in minibatches with the set of normal sliceLatency sequence (no anomalous sequences) dataset. The loss function is the categorical cross entropy function.

2. **Calculation of Sequence Probability**

- The input sequence is fed through the trained model. The output is a sequence of probability distributions using hidden states, which represent the probability distribution for the next integer in the sequence.

$$p(x) = \prod_{i=1}^{l} p(x_i | x_{1:i-1})$$

- A sequence probability value is calculated by essentially multiplying the probabilities of the next integer in the sequence occurring, across the entire length of the sequence.

3. **Identification of anomalies**
   - The negative log of the sequence probability is calculated for every sequence in the validation data.
   - If the negative log value for the sequence is greater than the threshold (in an unsupervised environment, the threshold is chosen as two standard deviations below/above the mean), the sequence is classified as anomalous (Positive), otherwise it is classified as normal (Negative).
   - The model architecture we designed, as shown in Figure 30, combines the AE predictions with the Network Fault Management to identify the faults (anomalies) to support root cause analysis. The local AE predictions used to detect the faults are based on an unsupervised learning probabilistic prediction approach trained using GRUs.

We detect the candidate anomalies in the network slice KPI data using an unsupervised technique where the labeled data is not normally available. We detect outliers in the test samples by calculating the robust Z-scores based on probability scores, defined as:

$$Z_{score} = \frac{0.6745 \times (x_i - \bar{x})}{MAD}$$

where 0.6475 is the 0.75th quartile of the standard normal distribution, xi the data points (MSE), x˜ the median and MAD denotes the median absolute deviation. A Z-score greater than the threshold value (5) is determined as an anomaly (anomalous KPI sequence).

Table 8: Anomaly Detection Models Evaluation

RMSE	RNN Units	Time taken to Train (sec)
4.296	LSTM with 200 units	433
4.320	GRU with 200 units	410
3.50	**GRU with 250 units + Regularizer**	**396**

Table 8 compares LSTM with 2 other models trained in a computationally efficiently way using regularization techniques (batch size = 32, Dropout= 20%, early callback mechanism technique). When a set of anomalous KPI sequence (sliceLatency KPI) is assigned to the trained probabilistic model, it leads to low probability score when compared to the normal KPI sequence.

6.1.2.3 INTERPRETABLE FRAMEWORK

Most Deep Neural Networks are a "**black box**" that cannot provide easily interpretable insights into the relationship between input and output. Particularly, when there is high dimensional data and multiple layers

in the neural network, there is a need for a method (post-hoc, after training) that can be used to provide the interpretability of models on the datasets where the ground-truth of interpretation results is not available.

We review local interpretation results generated by SHAP (normalized values) as shown in Figure 35. The Deep Explainer outputs features, SHAP values, and input values for anomalous sequences. To get a representation of the relationships between features for these anomalous KPI sequences, we obtained the most contributing feature(s) for sliceLatency KPI. Similar to this example, the SHAP values can help explain the impact of features on each anomalous entry detected by the proposed architecture. Figure 35 illustrates the order of importance of the features according to the impact (bandwidth has the highest) on the model output.

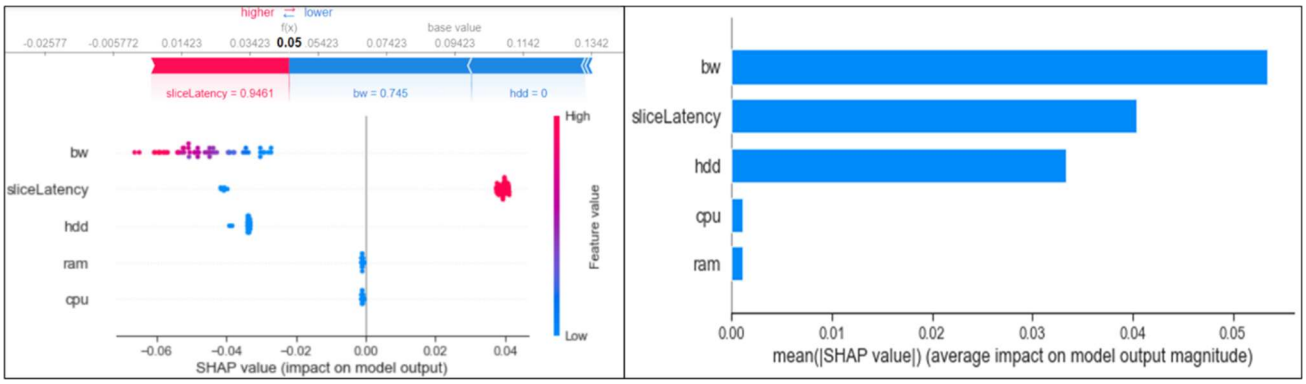

Figure 35: Framework depicting Feature Contributions

As mentioned in the MonB5G architecture, the AE and DE work together for optimizing network E2E average slice latency. The AE is deployed in a container as a Python script (Figure 36). The AE can be deployed quickly as a stateless container at any node of the network, eliminating the single point of failure. The AE receives a row of dataset at the predefined time interval of 60 seconds. The AE responds to the DE with a network state prediction of the next interval.

```
ubuntu@monb5g-lmi-ac-k8s-master:~/LMI$ sudo docker images
REPOSITORY                                          TAG                 IMAGE ID            CREATED             SIZE
ae                                                  latest              45893831b28c        7 days ago          3.27GB
<none>                                              <none>              11311340747d        7 days ago          3.27GB
<none>                                              <none>              b9e0b28f263b        7 days ago          3.27GB
<none>                                              <none>              2dc79e6f286c        7 days ago          3.17GB
<none>                                              <none>              33ed68c0a642        7 days ago          3.11GB
<none>                                              <none>              d729d38ba443        7 days ago          3.11GB
<none>                                              <none>              c7b10823ff05        7 days ago          1.46GB
hello                                               latest              a5f78de88c26        7 days ago          1.46GB
<none>                                              <none>              a9f7cf1de804        7 days ago          921MB
python                                              3                   e285995a3494        8 days ago          921MB
tensorflow/tensorflow                               latest              976c17ec6daa        2 weeks ago         1.46GB
nginx                                               1.19                f0b8a9a54136        16 months ago       133MB
k8s.gcr.io/kube-proxy                               v1.19.10            ecc0e51b4836        17 months ago       118MB
k8s.gcr.io/kube-apiserver                           v1.19.10            5b9e3415461d        17 months ago       119MB
k8s.gcr.io/kube-controller-manager                  v1.19.10            3e3b73af318b        17 months ago       111MB
k8s.gcr.io/kube-scheduler                           v1.19.10            98659ccbefae        17 months ago       46.5MB
quay.io/calico/node                                 v3.16.9             bf8c2ac5324d        18 months ago       167MB
quay.io/calico/cni                                  v3.16.9             c185726d452b        18 months ago       133MB
quay.io/calico/kube-controllers                     v3.16.9             5a4edae43ae8        18 months ago       52.9MB
k8s.gcr.io/dns/k8s-dns-node-cache                   1.17.1              21fc69048bd5        19 months ago       123MB
quay.io/coreos/etcd                                 v3.4.13             d1985d404385        2 years ago         83.8MB
amaksimov/python_data_science                       latest              da516e51ea2f        2 years ago         3.11GB
k8s.gcr.io/cpa/cluster-proportional-autoscaler-amd64   1.8.3            078b6f04135f        2 years ago         40.6MB
k8s.gcr.io/coredns                                  1.7.0               bfe3a36ebd25        2 years ago         45.2MB
k8s.gcr.io/pause                                    3.3                 0184c1613d92        2 years ago         683kB
k8s.gcr.io/pause                                    3.2                 80d28bedfe5d        2 years ago         683kB
ubuntu@monb5g-lmi-ac-k8s-master:~/LMI$ sudo docker run -v /home/ubuntu/LMI/mount:/mount ae
```

Figure 36: Cloud-native implementation of Analytics Engine as a Docker Container

6.1.3 *TIME-OF-DAY-AWARE SLICE ADMISSION CONTROL (TASAC) - RESOURCE CONSUMPTION PREDICTOR (RCP)*

TASAC is a DQN-based algorithm, which aims to derive optimal slice admission policy that maximizes the utilization of selected resources in time, considering the daily fluctuations of activity of network users. As an input, the TASAC algorithm requires the information about current usage of resources as well as prediction of future resources consumption in the time-period of concern (i.e., the operation time of the slice considered for deployment). To improve the learning process and reach optimal slice admission policy, which not only maximizes resources consumption but also minimizes possible SLA violations, it is essential to provide accurate and timely usage predictions. For this purpose, the RCP AE has been developed depicted in Figure 37.

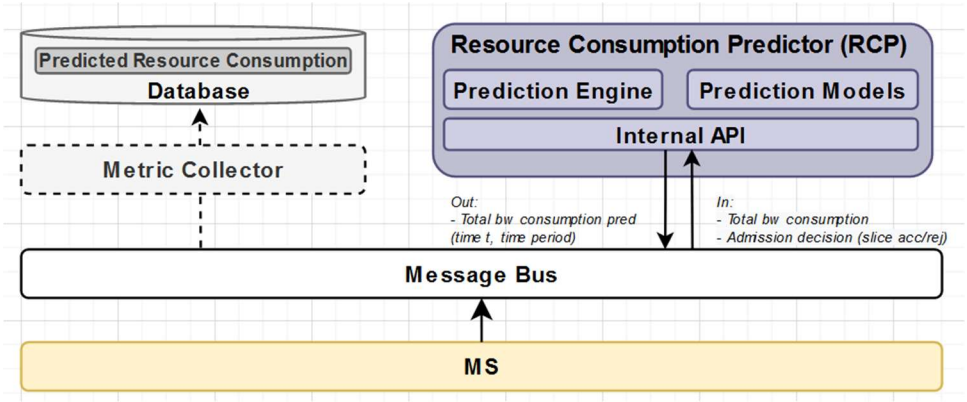

Figure 37: The overview of Resource Consumption Predictor module and interactions with Monitoring Service (MS)

The main objective of the RCP module is to provide means for prediction of consumption of any resources that can be required by other framework entities. For this purpose, the resources are defined and predicted in abstract units to better adjust to the specific slice or framework requirements (both the amounts as well as type of resource can be defined e.g. RAM, CPU, bandwidth). It enables prediction of the next occurring value, given the prediction interval (cf. Figure 38), as well as the series of values in the desired time-range. Current implementation enables selection of the Prediction Engine and specific Prediction Models to be exploited (analytic models derived from the activity in production network, offline-trained LSTM).

Figure 38: The prediction of resource consumption (aggregate bandwidth for all accepted slices)

The integration of RCP AE with other MonB5G layers (MS/AE/DE/ACT) is done via message bus and adaptation of message keys subscribed or published to the message broker. The RCP can also expose some of its functionalities via the Internal API e.g. to provide resource utilization prediction on other modules requests. The list of consumed and produced (marked as SUB and PUB respectively) by the RCP deployed in the testbed are listed in Table 9.

Table 9: Messages exchanged by RCP

Message Key	Type	I/O Data
ms.ms.total_bw	SUB	Current aggregate bandwidth consumption in the network (for all deployed slices

ae.rcp.api.prediction.req	SUB	Prediction requests containing the time range of prediction, unique request id
de.sac.new	SUB	The message generated in case of slice admission request acceptance (used by RCP to monitor the list of all accepted slices for the purpose of prediction)
ae.rcp.api.prediction.res	PUB	Prediction response containing the predicted values of the resource consumption in the requested time period together with the unique request id

To develop the RCP AE Python 3.10 has been used together with Docker as the containerization technology. The module functionalities are limited to predictions so as to fit the micro-service oriented MonB5G approach. Moreover, it provides means for configuration via environment variables (e.g., subscribed message keys) to improve re-usability and facilitate cloud-native deployments.

6.1.4 *ANOMALY DETECTION*

The main goal of the Anomaly Detection (AD) solution is quick detection of abnormal situations in the network to reduce reaction time to malfunctions. The proposed AD enabler uses LSTM autoencoder, as its primary building block, which has been trained offline using the production network data collected from the MS. The principles of operation are described in detail in deliverable D3.2 [1]. The generic implementation of AD enables the deployment to assist entities on multiple levels of hierarchy (e.g. DMO, IDMO) and covering multiple entities (deployment per slice, per slices group etc.). In this document however, the focus is laid on the integration of the AD with entities on the slice-level, which is considered as the most common deployment option. The integration of AD and interactions with MonB5G components are depicted in Figure 39. The considered scenario resolves around AD detecting anomalies in the slice bandwidth consumption trace.

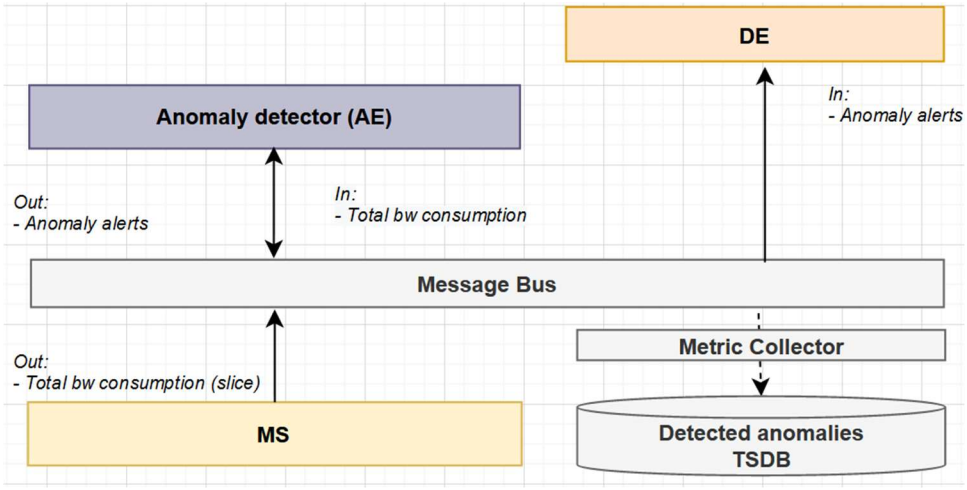

Figure 39: Anomaly Detection integration with MonB5G MS

Data exchange between MS, AD, and other components (i.e. various DE's or metric collector) is handled by a message bus. Integration of AD is performed by specific message keys to which AD subscribes, or under which it publishes the notifications regarding detected anomalies. Example keys used for the anomaly detection in a bandwidth consumption trace of a single slice (denoted as s1) are presented in Table 10.

Table 10: Example messages consumed and published by AD

Message Key	Type	I/O Data
ms.s1.bw	SUB	Current bandwidth consumption in slice
ad.s1.bw_detection_db	PUB	Publishing of detected anomalies to be stored in the database
ad.s1.bw_detection_de	PUB	Publishing of detected anomalies for other components like DE's

The Cloud-Native implementation of the AD solution has been developed in Python 3.10 and containerized using Docker. Container image has been successfully deployed in Kubernetes cluster and tested using the function mimicking slice behaviour.

6.1.5 ENHANCED TRAFFIC LOAD PREDICTION

This approach is formally called Enhanced Context-Aware Traffic Prediction (ECATP) and works as an AE in the framework of the MonB5G project. This is a supervised learning approach that provides a framework to extend a Deep Neural Network predictor beyond a purely time-series accuracy objective. This is done to make its predictions *aware* of the problem domain for which it is deployed. It is an enhancement of CATP, which was presented in the MidTerm Review of the MonB5G project, where it was successfully integrated with the Monitoring System (MS).

ECATP can be applied at any technological domain, and it can be parametrized well enough to adapt it to the respective problem domain. In its current implementation, ECATP works as a traffic predictor of different network slices in the RAN domain, generating prediction values for the traffic load of each slice coming from a Base Station (BS). Its context-aware predictions can then be used by a Decision Engine to realize orchestration decisions. Figure 40 shows an schematic of ECATP from an architectural point of view.

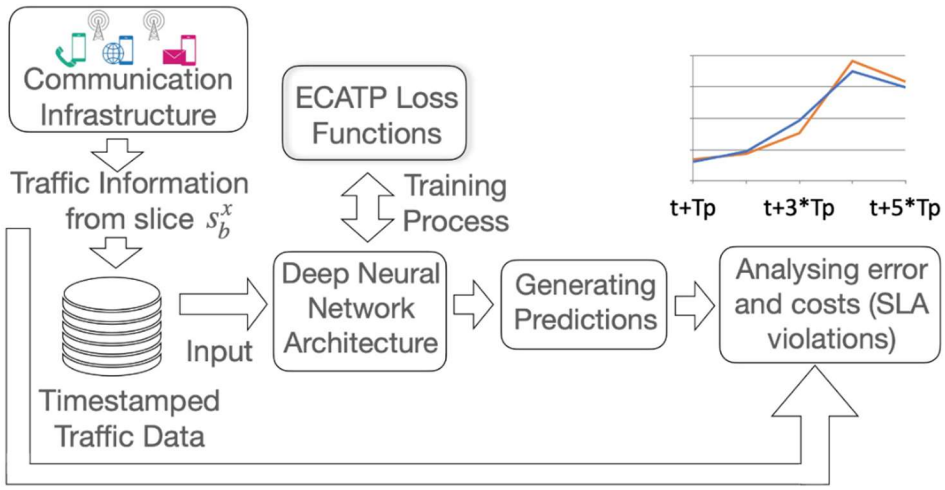

Figure 40: Architectural Diagram of ECATP

ECATP generates a DNN model that gets trained with historical time-series slice traffic data coming from a BS through the MS. It is the DB module of the MS where the historical data is stored and sampled in batches by ECATP. In addition, ECATP generates a prediction on the slice traffic load for the next time window using the current sample of slice traffic incoming from the BS->MS. Thus, ECATP requires the following inputs:

- Current sample of the slice traffic load of the respective slice
- Batches of historical time-series slice traffic from the DB module of the MS

The current sample of the slice traffic data is received through the Kafka Bus interface provided by the MS cluster, to which ECATP connects in a different container. It is important to note that even though the MS and ECATP do not have to run in the same physical server, it is recommended that ECATP and MS are deployed in the same cloud facility since the whole point of ECATP is to perform its predictions on site. In this way, data does not have to be sent upstream the network, consuming unnecessary resources.

Given that ECATP and MS are running in relative proximity to each other, it is possible for ECATP to connect to the Kafka Cluster already deployed by the MS. A publisher is configured in the MS that pushes the current samples of the traffic load for each slice, which the ECATP then consumes by subscribing to the respective publisher. ECATP will perform this prediction as fast as possible with the latest sample that arrived from the publisher.

In order to train ECATP, a batch of samples of time-series slice traffic data is sampled from the TSDB module of the MS. This batch size can be of variable of size, depending on the deep neural network model used by ECATP and its optimal parameters for training. The current size is 32 samples. The process of sampling these batches from the TSDB database occurs when ECATP launches its training procedure. This procedure is executed periodically, and it occurs in a way that it does not block the inference process (i.e. prediction generation) of the current samples.

Prior to the training procedure, ECATP issues requests for batches of sample data from the TSDB module, and the latter will return the data to the ECATP through the TSDB I/F. Then, the neural network model is copied,

and the training is done over this copy using the batch traffic data from the TSDB module. Once the training is complete, the new trained model will replace the initial one. When ECATP reads the next sample coming through the Kafka Bus I/F, then it will perform the prediction with the newly trained model.

Figure 41 shows the integration of ECATP with the MS in its current state. The "Analytics Engine Container" has the implementation of ECATP as shown in Figure 40, in addition to the Kafka Bus Interface (I/F) and the TSDB I/F. It is through the Kafka Bus I/F that ECATP subscribes to the respective publishers of the Sampling Functions for the gNB (which can be physical or emulated, as shown). Once ECATP generates the results of its predictions, it forwards the result to an Orchestrator.

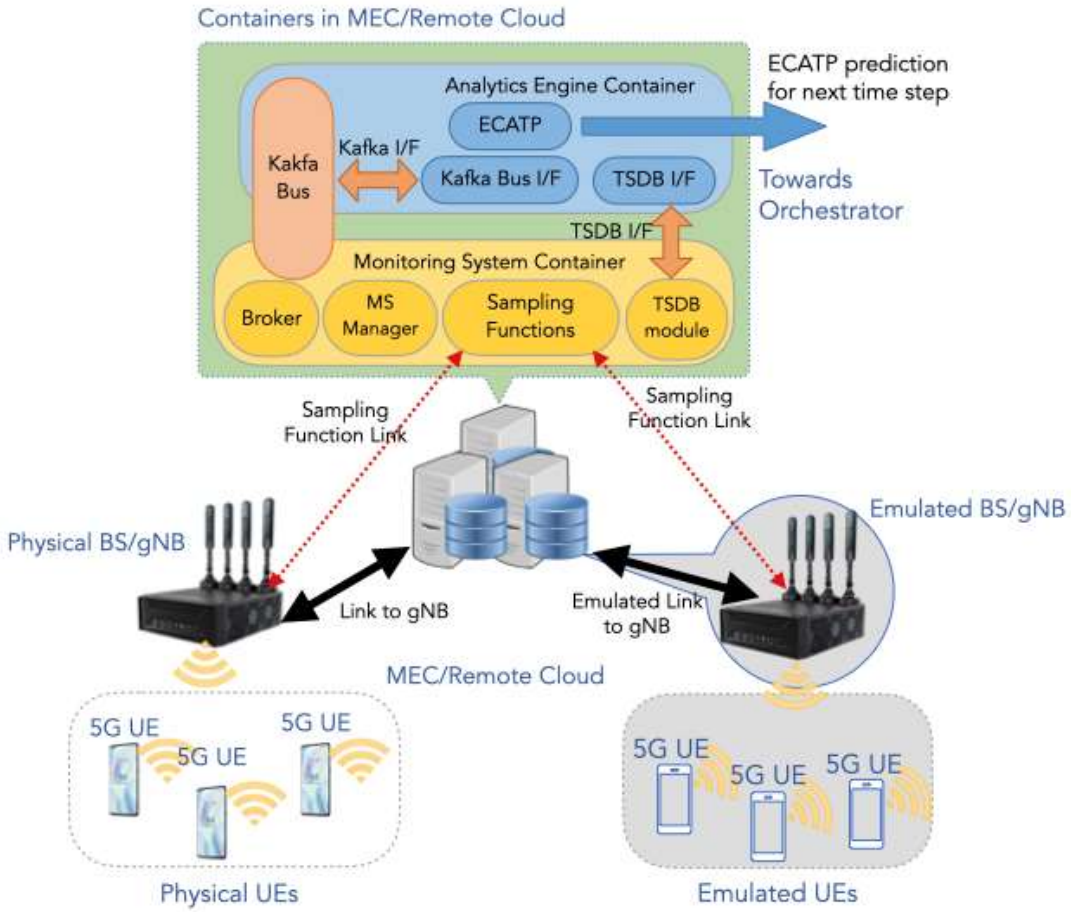

Figure 41: Diagram illustrating the Integration of ECATP with the MS

ECATP has been fully developed and implemented in Python 3.8 and Tensorflow 2.1, and runs in an LXD container in a cloud infrastructure in which the MS is also present. This cloud infrastructure is in proximity to a gNB (as shown in Figure 41) from which it gets its traffic information. The gNB can be physical or emulated,

depending on its ability to support network slicing and able to propagate traffic. In any case, the MS and ECATP are agnostic to this fact, as long as the proper sampling functions are configured in the MS and the stream of traffic information from the network slices is maintained.

7 Conclusions

In this deliverable, we have presented the design and implementation choices that we have taken for the MS and AE components of the MonB5G management platform. Additionally, we have presented a flexible and scalable way of integrating these components into a Beyond 5G system, while ensuring intelligent, scalable management mechanisms for a large-scale slice-based communication system.

Our choices include using cloud native techniques, providing the MS and AE components as containerized solutions that can be easily integrated at different levels in the system, and can also be orchestrated in a scalable way. The intelligent mechanisms included in the AE component are based on advanced AI (e.g., Deep Neural Networks, Graph based Deep Learning and Federated Learning), which have shown in our tests to perform well at scale in our tests.

Our design, integration and implementation choices are used as basis for the proof-of-concept use cases that will be described in the WP6 deliverables.

8 References

[1] D3.2, MonB5G, *Deliverable D3.2: Final Report on AI-driven Techniques for the MonB5G AE/MS, April,* 2022.

[2] D2.4, MonB5G, *Deliverable D2.4: Final release of MonB5G architecture (including security), Oct,* 2021.

[3] H. Chergui, L. Blanco and C. Verikoukis, "Statistical Federated Learning for Beyond 5G SLA-Constrained RAN Slicing," *IEEE Transactions on Wireless Communications,* vol. 21, no. 3, pp. 2066-2076, March 2022.

[4] H. Chergui and C. Verikoukis, "Offline SLA-Constrained Deep Learning for 5G Networks Reliable and Dynamic End-to-End Slicing," *IEEE Journal on Selected Areas in Communications,* vol. 38, no. 2, pp. 350-360, Feb 2020.

[5] "ETSI GR NFV-IFA 029: Network Functions Virtualisation (NFV) Release 3;Architecture; Report on the Enhancements of the NFV architecture towards"Cloud-native" and "PaaS"".

[6] A.-M. Bosneag and S. H. M. Wang, "Fighting Fire with Fire: Survey of Strategies for Counteracting The Complexity of Future Networks Management," in *19th International ICIN Conference - Innovations in Clouds, Internet and Networks*, Paris, 2016.

[7] A. Chawla, B. Lee, S. Fallon and P. Jacob, "Host Based Intrusion Detection System with Combined CNN/RNN Model," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2019.

[8] S. Roy, H. Chergui, L. Sanabria-Russo and C. Verikoukis, "A Cloud Native SLA-Driven Stochastic Federated Learning Policy for 6G Zero-Touch Network Slicing," *ICC 2022 - IEEE International Conference on Communications (ICC)*, 2022, pp. 4269-4274.