



A Holistic, Innovative Framework for the Design,
Development and Orchestration of 5G-ready
Applications and Network Services over Sliced
Programmable Infrastructure

DELIVERABLE D3.2

INTELLIGENT ORCHESTRATION MECHANISMS

Due Date of Delivery:	M24 <i>Mx</i> (31/05/2019 <i>dd/mm/yyyy</i>)
Actual Date of Delivery:	20/08/2019 <i>dd/mm/yyyy</i>
Date of Revision Delivery:	02/10/2020 <i>dd/mm/yyyy</i>
Workpackage:	WP3 – Intelligent Orchestration Mechanisms
Type of the Deliverable:	OTHER
Dissemination level:	PU
Editors:	UBITECH, UPRC, INC, SUITE5, ININ, ATOS, NCSRD, INTRA
Version:	2.0

Co-funded by
the Horizon 2020
Framework Programme
of the European Union



Call:

H2020-ICT-2016-2

Type of Action:

IA

Project Acronym:

MATILDA

Project ID:

761898

Duration:

38 months

Start Date:

01/06/2017

Project Coordinator:

Name:

Franco Davoli

Phone:

+39 010 353 2732

Fax:

+39 010 353 2154

e-mail:

franco.davoli@cni.it

Technical Coordinator:

Name:

Panagiotis Gouvas

Phone:

+30 216 5000 503

Fax:

+30 216 5000 599

e-mail:

pgouvas@ubitech.eu

List of Authors	
ATOS	ATOS SPAIN SA
Aurora Ramos, Fernando Díaz and Javier Melián	
INC	INCELLIGENT IDIOTIKI KEFALAIOUCHIKI ETAIREIA
Panagiotis Demestichas, Kostas Tsagkaris, Nikos Stasinopoulos, Athina Ropodi, Aristotelis Margaritis, Dimitris Cardaris, Marinos Galiatsatos, Vasilios Bourdouvalis, Nikos Gavriil, Konstantinos Bitsakos	
ININ	INTERNET INSTITUTE, COMMUNICATIONS SOLUTIONS AND CONSULTING LTD
Luka Koršič, Dušan Mulac, Jaka Cijan, Janez Sterle	
INTRA	INTRASOFT INTERNATIONAL SA
Kostas Thivaivos, Marios Logothetis	
NCSRD	NATIONAL CENTER FOR SCIENTIFIC RESEARCH “DEMOKRITOS”
Themistoklis Anagnostopoulos, Akis Kourtis	
S5	SUITE5 DATA INTELLIGENCE SOLUTIONS LIMITED
Lefteris Lampathakis, Katerina Zerva	
UBITECH	GIOUMPI TEK MELETI SCHEDIASMOS YLOPOIISI KAI POLISI ERGON PLIROFORIKIS ETAIREIA PERIORISMENIS EFTHYNIS
Panagiotis Gouvas, Anastasios Zafeiropoulos, Eleni Fotopoulou, Thanos Xirofotos	
UPRC	UNIVERSITY OF PIRAEUS RESEARCH CENTRE
Chrysostomos Symvoulidis, Emmanouil Alexakis, Eftychia Vorila, Dimitris Drakoulis	

Disclaimer

The information, documentation and figures available in this deliverable are written by the MATILDA Consortium partners under EC co-financing (project H2020-ICT-761898) and do not necessarily reflect the view of the European Commission.

The information in this document is provided “as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The reader uses the information at his/her sole risk and liability.

Copyright

Copyright © 2020 the MATILDA Consortium. All rights reserved.

The MATILDA Consortium consists of:

CONSORZIO NAZIONALE INTERUNIVERSITARIO PER LE TELECOMUNICAZIONI (CNIT)

ATOS SPAIN SA (ATOS)

ERICSSON TELECOMUNICAZIONI (ERICSSON)

INTRASOFT INTERNATIONAL SA (INTRA)

COSMOTE KINITES TILEPIKOINONIES AE (COSM)

ORANGE ROMANIA SA (ORO)

EXXPERTSYSTEMS GMBH (EXXPERT)

*GIOUMPI TEK MELETI SCHEDIASMOΣ YLOPOIISI KAI POLISI ERGON PLIROFORIKIS
ETAI REIA PERIORISMENIS EFTHYNIS (UBITECH)*

INTERNET INSTITUTE, COMMUNICATIONS SOLUTIONS AND CONSULTING LTD (ININ)

INCELLIGENT IDIOTIKI KEFALAIOUCHIKI ETAIREIA (INC)

NATIONAL CENTER FOR SCIENTIFIC RESEARCH “DEMOKRITOS” (NCSR)

UNIVERSITY OF BRISTOL (UNIVBRIS)

AALTO-KORKEAKOULUSAATIO (AALTO)

UNIVERSITY OF PIRAEUS RESEARCH CENTER (UPRC)

ITALTEL SPA (ITL)

BIBA - BREMER INSTITUT FUER PRODUKTION UND LOGISTIK GMBH (BIBA)

SUITE5 DATA INTELLIGENCE SOLUTIONS LIMITED (S5)

This document may not be copied, reproduced or modified in whole or in part for any purpose without written permission from the MATILDA Consortium. In addition to such written permission to copy, reproduce or modify this document in whole or part, an acknowledgement of the authors of the document and all applicable portions of the copyright notice must be clearly referenced.

Table of Contents

DISCLAIMER.....	3
COPYRIGHT.....	3
TABLE OF CONTENTS	4
TABLE OF ACRONYMS	5
1 EXECUTIVE SUMMARY.....	6
2 INTRODUCTION	7
2.1 SCOPE OF THE DOCUMENT	7
2.2 MATILDA END-TO-END STORY.....	8
3 DEPLOYMENT AND EXECUTION MANAGER	12
3.1 MAIN COMPONENTS.....	14
3.2 INTERFACES	15
3.3 BASELINE TECHNOLOGIES	16
4 MATILDA AGENT AND SERVICE DISCOVERY MECHANISMS	17
4.1 MAIN COMPONENTS.....	19
4.2 INTERFACES	21
4.3 BASELINE TECHNOLOGIES	22
5 MONITORING MECHANISMS.....	23
5.1 MATILDA OVERALL MONITORING SOLUTION	23
5.1.1 Application component monitoring.....	25
5.1.2 Network Slice monitoring	28
5.1.3 NFVO/OSM metric collection.....	28
5.1.4 qMON NFV-based monitoring	31
5.2 BASELINE TECHNOLOGIES	37
6 DATA FUSION, REAL-TIME PROFILING AND ANALYTICS TOOLKIT	39
6.1 MAIN COMPONENTS.....	43
6.1.1 Data Fusion.....	43
6.1.2 Real-time profiling and Analytics module.....	46
6.2 INTERFACES	47
6.3 BASELINE TECHNOLOGIES	48
7 POLICIES ENFORCEMENT AND CEP MECHANISMS	49
7.1 MAIN COMPONENTS.....	51
7.1.1 Policies Enforcement mechanism.....	51
7.1.2 Complex Event Processing mechanism	52
7.2 BASELINE TECHNOLOGIES	54
8 NORTHBOUND APIS FOR COMMUNICATION SERVICE PROVIDERS.....	55
8.1 MAIN COMPONENTS.....	55
8.2 INTERFACES	56
8.3 BASELINE TECHNOLOGIES	59
9 CONCLUSIONS	60
REFERENCES.....	61

Table of Acronyms

Acronym	Definition
5G PPP	5G Public Private Partnership
API	Application Programming Interface
BSS	Business Support System
CSM	Computing Slice Manager
DoA	Description of Action
NFVO	Network Function Virtualization Orchestrator
IE	Inference Engine
JSON	JavaScript Object Notation
OSM	Open Source MANO
OSS	Operational Support System
PCE	Path Computation Engine
PM	Production Memory
SDS	Service Discovery Server
UE	User Equipment
UI	User Interface
VAO	Vertical Application Orchestrator
VSC	Version Control System
VIM	Virtual Infrastructure Manager
VNF	Virtual Network Function
WIM	Wide-area Infrastructure Manager
WM	Working Memory

1 Executive Summary

The scope of MATILDA is to deliver a holistic framework for the design, development and orchestration of 5G-ready applications and network services. In this context, a **5G-ready application** is represented by an application graph, which consists of chainable application components. Moreover, a 5G-ready application includes in its description a set of network-oriented requirements that can be utilized towards the production of **slice requirements**. Such requirements are interpreted in appropriate **slice intents** to the MATILDA-enabled telco provider. The slice intent can be then translated by a telco provider towards the provision of an application-aware network slice that can **optimally fulfil** the application needs.

This document covers the design and development of a set of **Vertical Application Orchestration (VAO)** mechanisms, in order to realize the proper **placement and orchestration of 5G-ready applications** over the created application-aware network slices. The orchestration business logic of VAO **should not** be confused with the business logic of the telco provider's resources orchestration (VNFs, etc.) which is **coordinated by the OSS**.

The main components constituting the MATILDA VAO are: **(i) the deployment and execution manager** that supports the production of optimal deployment plans as well as manages the overall execution of the application, **(ii) a set of data monitoring mechanisms** which collect feeds from network and application-level metrics, **(iii) a data fusion, real-time profiling and analytics toolkit**, that produces advanced insights through machine learning mechanisms and provide real-time profiling of the deployed components, application graphs and VNFs, **(iv) service discovery mechanisms for supporting registration and consumption of application-oriented services** following a service mesh approach, **(v) a context awareness engine** in order to provide inference over the acquired data and support runtime policies enforcement, and **(vi) mechanisms supporting interaction among the VAO and the OSS**.

2 Introduction

2.1 Scope of the Document

This deliverable aims at describing the design and implementation of the final release of the MATILDA intelligent orchestration mechanisms. The mechanisms comprise the Vertical Application Orchestration (hereinafter VAO) layer of the MATILDA architecture. According to the MATILDA reference architecture [MATILDA – D1.1] a vertical application will make use of **two distinct** orchestration loops which will be responsible for the entire lifecycle of the application per se. These loops include the VAO loop undertaken by WP3 and the intra-OSS loop undertaken by WP4. VAO is responsible to orchestrate the cloud-native application per se in programmable resources that are provided by the telco-provider while the OSS is responsible to provide the aforementioned resources along the prerequisite end-to-end connectivity. Such connectivity is achieved through programmability of the telco resources.

The term “final release” is partially precise since VAO and OSS are continuously evolving in order to include additional features that even super-exceed the initial (contractual) scope of their existence. This is part of the “moving-target” problem that MATILDA had to face since technology and standardization is evolving. However, the term “final” is precise as far as the communication (i.e. signaling) between the VAO and the OSS is concerned, as the former can currently express location-specific constraints to the latter. Furthermore, given that this version of the deliverable regards a revised version of the document that is delivered close to the end of the lifetime of the project, the provided specifications can be considered as final in terms of evolution of the VAO and the OSS within the MATILDA 5G PPP project lifetime. The main changes in the revised version regard the change in the presentation mode of the deliverable from a diff-based methodology (compared to the initial version of the description of the VAO mechanisms in D3.1) to a full-description-based methodology where the implemented mechanisms in the VAO components are presented. A detailed specification of the components comprising the VAO mechanisms is presented, including the interfaces between them, as well as the interactions with external entities. The specification is accompanied with additional information regarding the technologies used for the development of the mechanisms.

As already mentioned, this document focuses on the description of the final design and implementation of the intelligent orchestration mechanisms of the VAO that reside at the Applications’ Orchestration Layer. In more detail, the document is structured as follows: **Chapter 3** is dedicated to the description of the deployment and execution manager mechanisms. The Deployment and Execution Manager has been **designed in such a way that it abstracts the elasticity functionality and extracts the elasticity “business logic” outside the core orchestration loop**. In this way, a community of elastic-by-design frameworks can be onboarded onto the repository.

Chapter 4 provides the description of the MATILDA Agent along the service discovery mechanisms. The MATILDA Agent has **extended functionality in order to be able to handle security policies** making use of state-of-the-art Linux kernel filters.

Chapter 5 describes the final release of the monitoring mechanisms. The orchestrator contained within the Monitoring Mechanisms possesses the capability to report

measurements regarding its internal actions (e.g. VM spawning). In this way the ability to elaborate on the elasticity efficiency of the vertical applications is provided.

Chapter 6 provides details for the real-time profiling and analytics toolkit. The Profiling and Analytics Toolkit has the ability to “virtually” tap on the aggregated monitoring metrics and facilitate a set of analytics pipeline. Through this pipeline many analytics insights regarding the running apps can be generated.

Chapter 7 describes the **policies enforcement approaches** and the complex event processing mechanism. The characteristics of Policies enforcement and CEP Mechanisms **ensure that the component per se is horizontally scalable** and thus a large amount of rules can run simultaneously for many vertical applications.

Chapter 8 details the specification and implementation of the **APIs created for the communication with the OSS**. The Southbound API which interconnects the OSS and the VAO is capable of providing location-specific services such as the activation of a slice on a specific territory.

Finally, **Chapter 9** summarizes the results and concludes the deliverable.

Each chapter is divided into three major sections providing information about each mechanism of the intelligent orchestration suite of MATILDA. A small introduction to the chapter is followed by the first “**Main components**” sub-section which provides a short presentation of the key components of each mechanism, while in the “**Interfaces**” sub-section the overall (e.g. external) interfaces of the prototype are described. Finally, the “**Baseline technologies**” sub-section discusses the baseline technologies, programming languages and tools that have been used for the implementation of the corresponding mechanisms.

2.2 MATILDA End-to-End story

The advent of 5G networks is predicted as a leading factor in the fourth industrial revolution impacting multiple vertical sectors and changing the way services are developed. Nevertheless, the necessary integration between the digital systems that enable those services and the network layer remains undefined and represents a big challenge.

MATILDA aims to fill this gap, providing the tools to foster and speed up the extension/evolution of the cloud paradigm into the 5G ecosystem, intrinsically bridging the vertical application and the network service domains. MATILDA comes up with a novel and holistic approach for tackling the overall lifecycle of applications’ design, development, deployment and orchestration in a 5G environment.

A set of novel concepts are introduced, including the design and development of 5G-ready applications -based on cloud-native/microservice development principles- the separation of concerns among the orchestration of the developed applications and the required network services that support them, as well as the specification and management of network slices that are application-aware and can lead to optimal application execution.

MATILDA follows a top-down approach where application design and development leads to the instantiation of application aware-network slices, over which vertical industries’ applications can be optimally served. Different stakeholders are engaged in this process, with clear separation of concerns among them.

Still, MATILDA's main target is the Telecom Provider, including Virtual Network Operators (VNOs), Network as a Service (NaaS) companies and small players with innovative, telecom-centric, business models. MATILDA aims to help them to satisfy their vertical customers' need, by creating a fruitful environment where network-intensive services can be easily prototyped and quickly deployed into production.

The framework allows software developers to create applications following a simple and conventional microservices-based approach where each component can be independently orchestratable. Based on the conceptualization of metamodels (application component and graph metamodels), they can formally declare information and requirements -in the form of descriptor- that can be exploited during the deployment and operation over programmable infrastructure.

Such information and requirements may regard capabilities, envisaged functionalities and soft or hard constraints that have to be fulfilled and may be associated with an application component or virtual link interconnecting two components within an application graph. The produced application is considered as 5G-ready application.

MATILDA also encourages new business and wide collaboration by providing a Marketplace where not only the created applications and components can be published but also Virtual Network Functions (VNFs) and Network Services (NS) (in the form of enhanced descriptors).

Service Providers are able to adopt the developed 5G-ready applications (published to the Marketplace or created internally) and specify policies and configuration options for their optimal deployment and operation over programmable infrastructure. Based on the provided application descriptor, service providers are able to design operational policies and formulate a slice intent. These operational policies describe how the application components should adapt their execution mode in runtime. On the other hand, the slice intent includes a set of constraints that have to be fulfilled during the placement of the application and a set of envisaged network functionalities that have to be provided. This information is used by the Vertical Application Orchestrator to request the creation of an appropriate application-aware network slice from the Telecommunication Infrastructure Provider.

While the instantiation and management of the application-aware network slice (including the set of network functions) is realised by the Network and Computing Slice Deployment Platform (managed by the telecommunications infrastructure provider), the deployment and runtime management of an application is realised by the vertical application orchestrator (managed by the service provider), following a service-mesh-oriented approach.

This recently introduced approach is adopted as a software management layer for controlling and monitoring internal traffic in microservices-based applications. It consists of a data and a control plane. The data plane consists of a set of intelligent proxies deployed alongside the application software components supporting the provision of support/backing services (e.g. service discovery, load balancing, health checking, telemetry). The control plane manages the set of intelligent proxies based on distributed management techniques and provides policy and configuration guidance for all the running support/backing services. Policies definition for the activation and management of the set of required support/backing services is realised based on a policies editor, while policies enforcement is realised based on a rules-based management system. Advanced monitoring and analysis techniques are also applied for extracting insights that can be proven useful for service providers.

In order to instantiate and manage the application-aware network slice during the overall lifecycle of the 5G-ready application, Telecommunication Infrastructure Providers rely on the concept of network slice to fulfil the vertical application needs. A network slice is a logical infrastructure partitioning allocated resources and optimized topology with appropriate isolation, to serve a particular purpose of an application graph.

The Network and Computing Slice Deployment Platform includes an OSS/BSS system, a NFVO and a resources manager for managing the set of deployed WIMs and VIMs. Based on the interpretation of the provided slice intent, the required network management mechanisms are activated and dynamically managed.

The Telecommunication Infrastructure Provider is responsible to realise the instantiation of the slice over the programmable infrastructure. The reserved resources for this slice combine both network and compute resources. A Telecommunication Infrastructure Provider may deliver all these resources based on his own infrastructure or come into an agreement with a Cloud Infrastructure Provider and acquire access to additional compute resources (e.g. in the edge of the network).

These actions are realized in an agnostic way to application service providers. However, through a set of open APIs, requests for adaptation of the slice configuration may be provided by the Vertical Applications Orchestrator to the Network and Computing Slice Deployment Platform.

The materialization of the network slice requires the instantiation of network services (NSs) that are composed of virtual network functions (VNFs) chains. These NSs and VNFs can be imported into the telecommunications infrastructure provider's catalogue from the MATILDA marketplace.

A summary of the described overall lifecycle of an application created with the MATILDA framework is represented in Figure 1 below, highlighting the interaction among the different stakeholders and the usage of the metamodels.

The MATILDA reference architecture is divided in three distinct layers: namely, the 5G-ready Applications Layer, the Applications' Orchestration Layer and the Network and Computing Slice Management Layer. Separation of concerns per layer is a basic principle adhered towards the design of the overall architecture. The Applications Layer is oriented to software developers, the 5G-ready Application Orchestration Layer is oriented to service providers and the 5G Infrastructure Slicing and Management Layer is oriented to telecommunications infrastructure providers.

The 5G-ready Applications Layer takes into account the design and development of 5G-ready applications per industry vertical, along with the specification of the associated networking requirements. The associated networking requirements per vertical industry are tightly bound together with their respective 5G-ready applications' graph, which defines the business functions, as well as the service qualities of the individual application.

The Applications' Orchestration Layer supports the dynamic on-the-fly deployment and adaptation of the 5G-ready applications to its service requirements, by using a set of optimisation schemes and intelligent algorithms to provide the needed resources across the available multi-site programmable infrastructure.

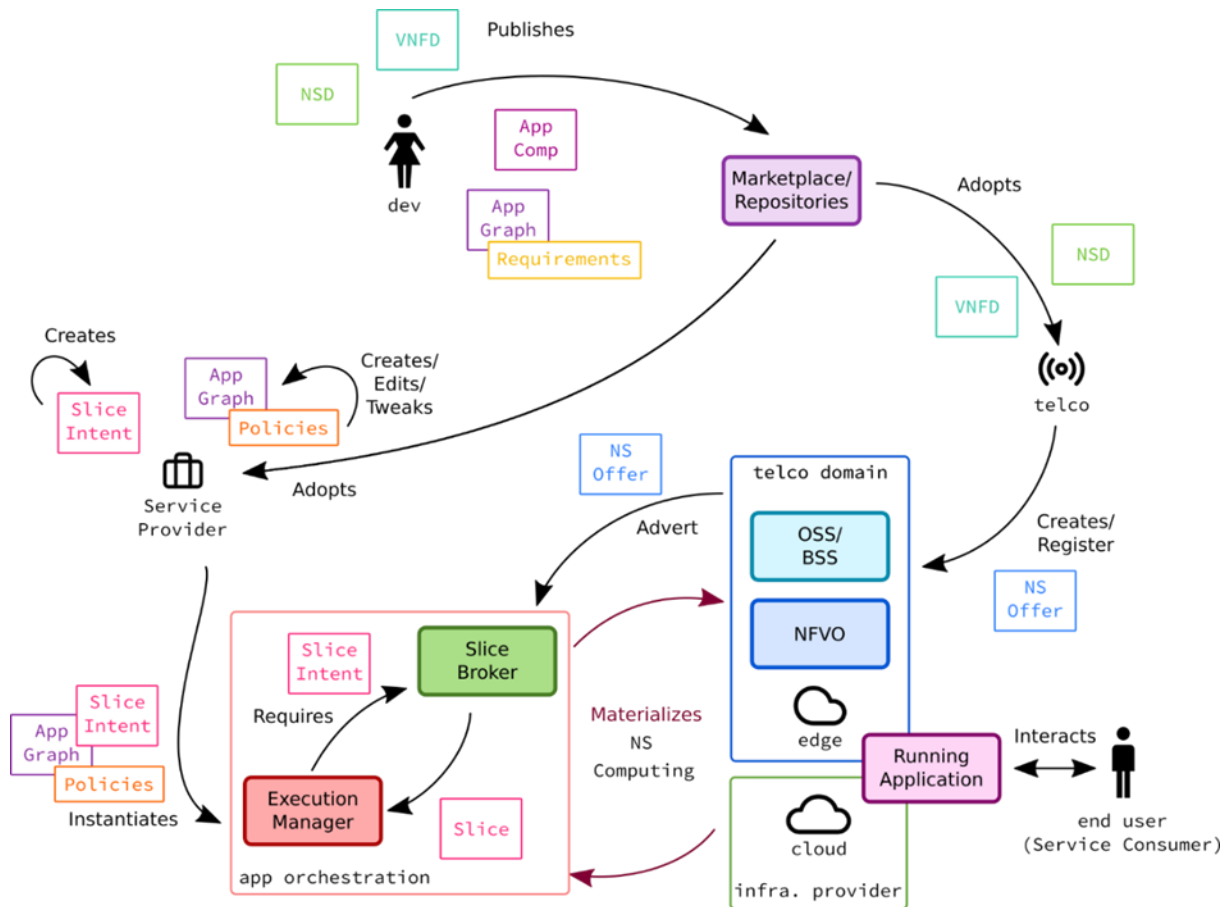


Figure 1: MATILDA workflow highlighting the different stakeholders and metamodels

The Programmable 5G Infrastructure Slicing and Management Layer is responsible for setting up and managing the 5G-ready application deployment and operation over an application-aware network slice. Network slice instantiation and management, network services and mechanisms activation and orchestration as well as monitoring streams management are realized. Such actions are triggered based on requests provided by the Applications' Orchestration Layer through the specification of Open APIs.

3 Deployment and Execution Manager

The operational goal of the Deployment and Execution Manager is to facilitate the initial deployment, the execution and the deprovision of the vertical components. Part of the execution management business logic is the handling of the **elasticity business logic**. Elasticity is the trait of a system to self-expand or shrink based on the undertaken load. Elasticity is “**technically interpreted**” in a **completely different way** based on the nature of the vertical component. For example, a **stateless http component** can scale under two assumptions: **a)** there is a mechanism to spawn/destroy stateless workers based on the demand and **b)** there is a ‘central’ component that can split and redirect the traffic to the various workers (a.k.a. balancer). In a storage component the ‘high-level’ assumptions are the same but the mechanism-insights are completely different. There is the assumption that a mechanism will spawn/destroy storage elements **but the balancing logic is completely different**.

The ability to support, inherently, scalability of **http-load-balanced components** was addressed as horizontal scalability feature. However, the question that was raised was “**how other types of elastic components will be supported?**”. Architecturally-wise there is no easy answer to that, in the sense that each component/framework introduces its own business logic during scale-in and scale-out.

The primary problem is that if the elasticity business logic is embedded in the core-business logic of the VAO, each new elastic framework that will be supported will require the roll-out of a new VAO release. That is definitely not an option for a commercial release. Instead, the most elegant solution was to define an abstract API of an elasticity controller and alter the core orchestrator logic in order to dynamically interact with an instance of this controller (see Figure 2). In other words, the VAO’s core-orchestrator, besides the deployment and the checking of the application and component status, has also to manage the elasticity controller.

The core-orchestrator must **initialize the metadata for each elasticity-controller** on the deployment phase and **update them in the scale in and out phases**. For that, so, both the backend and the core-orchestrator have some extra steps to do for the elastic components, some of which are strictly coupled with the elasticity framework and some other with the framework itself.

Another crucial question is “**how does a developer implement an elasticity controller in a MATILDA-certified way?**” To do so, we followed a Service-Provider-Interface adapter architecture¹ (a.k.a. SPI) and ensured the **elasticity logic was decoupled from both the VAO’s backend** as also the core-orchestrator, and thus introduced the elasticity-framework module-template. As we need also to let the developer implement their elasticity-adapters without the need to know the VAO’s architecture, we **created some services and utilities that act as middleware between the adapter and the backend/core-orchestrator**. So, having those services, that will be not explained in this deliverable, the developer needs to implement two adapters, one that will be used from the backend and one that will be used from the core-orchestrator.

¹ <https://docs.oracle.com/javase/tutorial/sound/SPI-intro.html>

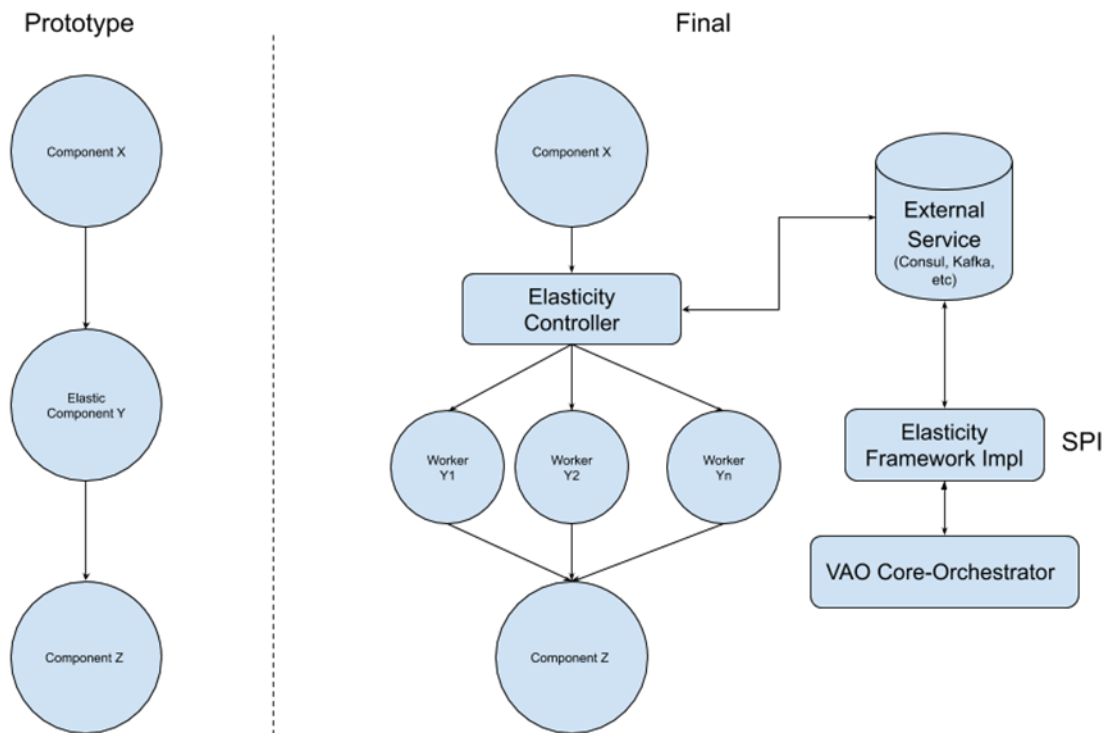


Figure 2: Elasticity Controller interaction with core orchestrator

The adapter that is consumed by the backend requires to implement the following methods:

- ***addComponentToDB()***, which is responsible to add the elasticity controller to the database. We are offering some services that are communicating with the database.
- ***getElasticityType()***, which is responsible to give the backend the parameterization that the controller may need.
- ***addElasticityController()***, which is responsible to add the controller to the graph. For this method we offer some utilities to the developer that modify the application instance graph before the slice request.
- ***scaleOut()***, which is responsible to return the workers that need to be deployed.
- ***scaleIn()***, which is responsible to delete the undeployed workers from the database.

The adapter that is consumed by the core-orchestrator requires to implement the following methods:

- ***initControllerMetadata()***, which is responsible to initialize the controller metadata, if there are any.
- ***getHealthyWorkers()***, which returns all the healthy workers of the controller, either by using the service that communicates with the platform's discovery service or by a custom way.
- ***updateElasticityController()***, which is responsible to pass the appropriate changes to the elasticity controller.

- ***scaleOut()***, which is responsible to calculate how many workers to add, request them by the backend using the respective service and return the response to the orchestrator.
- ***findWorkersForRemoval()***, that calculates how many workers can be removed during scale in phase, and returns all the workers that are qualified for removal.
- ***removeWorkersFromController()***, which is responsible to communicate with the elasticity controller and removes the qualified workers before the deletion of the VMs.
- ***postScaleInControllerMetadataClean()***, that is responsible to update the controller metadata and to do whatever else is needed after the deletion of the VMs.

We have used this extensibility mechanism to create elasticity controllers for two frameworks, namely JPPF² and Spark³.

3.1 Main components

A core component of the orchestrator is the Deployment and Execution Manager, which is the component that is responsible to **materialize a placement plan of a vertical application**. As already analysed in the architectural deliverable, each vertical application consists of multiple components that formulate an application graph. The application graph and the components adhere to a specific metamodel. A vertical application provider is introducing some constraints at the component/application level, which are submitted to the MATILDA-enabled telco provider through the Northbound API that is described in Section 8. The telco provider is interpreting the constraints to a constraint-satisfaction problem. Upon the identification of a solution the telco provider is creating a **slice** which facilitates the requirements of the provider.

The slice per se is sent back to the orchestrator. The slice contains placement instructions, i.e. where each component should be placed. The deployment manager is responsible to trigger the VIMs that are included in the slice response. Beyond spawning the VMs the Deployment and Execution manager is responsible to monitor the proper instantiation of the vertical components within the VM. This is practically performed by a component that is addressed as MATILDA Agent and is loaded in each VM that is spawned within the telco provider. The Agent is responsible to report on the success boot sequence of the vertical components and even react on managed exceptions (e.g., VM is not available at the moment, component is loaded but health check is failing).

Therefore, the main components of the “Deployment and Execution Manager” are the following:

- **Slice Interpreter:** The interpreter is responsible to parse the response of the telco provider and infer the proper VIMs that should be used in order for deployment to take place.
- **VIM client:** This component enables the execution of the VIM-related commands that are provided to the MATILDA-enhanced OSS of the telco provider.
- **MATILDA Agent:** The agent is responsible to monitor the boot sequence of the vertical application. Moreover, it is responsible to activate the monitoring probe of the VM.

² <https://www.jppf.org/>

³ <https://spark.apache.org/>

- **Service Discovery Server (SDS):** The server keeps track of the operational state of all vertical applications and their components. It acts as a cache memory; thus, keeping track of the state and the configuration of each VM.
- **Execution Manager Control loop:** Each orchestration engine spawns one ‘infinite’ control loop per deployed application. This control loop is responsible for tracing the execution of all components and react on possible errors.
- **Elasticity Controller:** It is responsible to handle the scale-in/out operations as analysed above

The components mentioned above interact using the interfaces that are presented below.

3.2 Interfaces

Figure 3 depicts the basic interfaces among the components of the Deployment and Execution Manager.

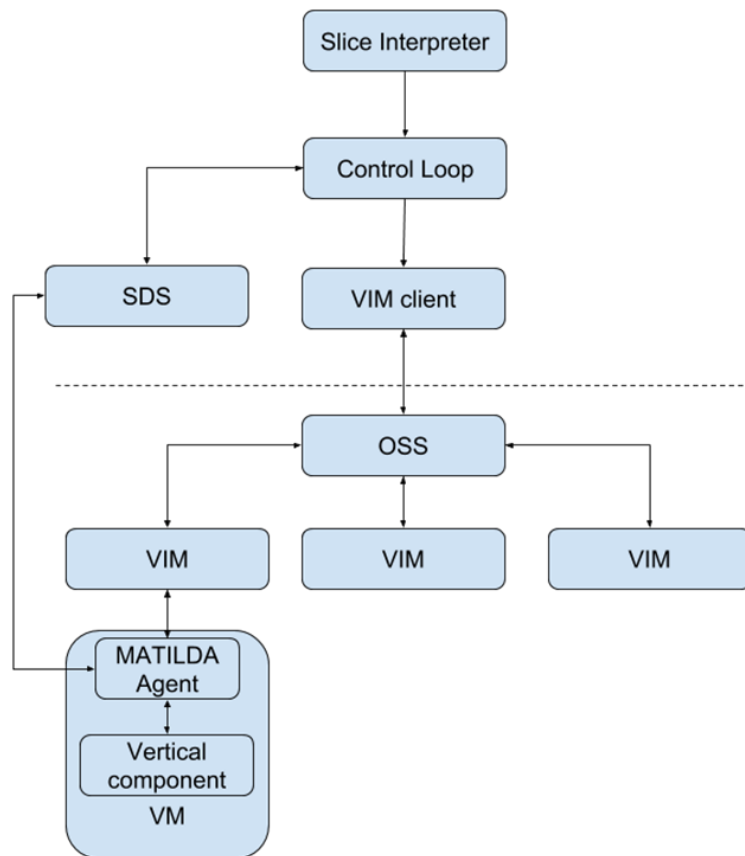


Figure 3: Components of Deployment and Execution Manager

As it is depicted, the Slice Interpreter is interacting with the Control Loop in order to trigger the initial deployment of the vertical components that comprise the vertical application. The Control Loop will trigger a **parallel** deployment of the vertical components. The term parallel refers to the **VM spawning process** that will host the vertical components and **not the component booting process**. The components must be booted in such an order that each component should **guarantee** that its **direct dependencies** are available. i.e. booted and

operational. This business logic is implemented with the MATILDA Agent and is implemented using an interaction with the SDS. This process will be analysed in the next section.

The Control Loop component is interacting with the VIM client in order to trigger the VM spawning commands. The VIM client is able to interact with **multiple VIMs** that are proxied by the (MATILDA-enabled) OSS layer. The OSS is proxying the VIM command and the actual VM is spawned. During the VM spawning a specific **initialization script**, which is passed as an argument to the VIM, is executed in order to perform the initial configuration of the VM.

Initial configuration is a multi-step process according to which some executable prerequisites are downloaded. These include **a)** an SDS client, which is Consul in our case; **b)** the monitoring probe; **c)** a container execution engine and **d)** the MATILDA Agent. The SDS client is required in order to interact with the SDS server that is installed in the vertical orchestrator. The monitoring probe will be used by the monitoring and analytics components that are analysed below and the container execution engine is required in order to abstract the way the component is bundled. Finally, the Agent is responsible for the lifecycle management of the component per se, i.e. boot/pause/destroy.

3.3 Baseline technologies

The Deployment and Execution Manager is built using the Spring Framework [Spring]. The reason for that is that Spring offers a tool suite for production-grade development of microservice-based solutions. As such, many aspects of development, such as development of RESTful interfaces, implementation of high concurrent business logic, uniform reporting and analytics during operation, etc., are considered granted. On the other hand, the MATILDA Agent is developed using pure Java 8 Compact Profile [Compact-Profile] in order to make use of the interoperability aspects of Java without sacrificing the minimum footprint that an Agent should have. Finally, the web interface that triggers the deployment of a graph is developed using React JS [React-JS], as also depicted in Figure 4.

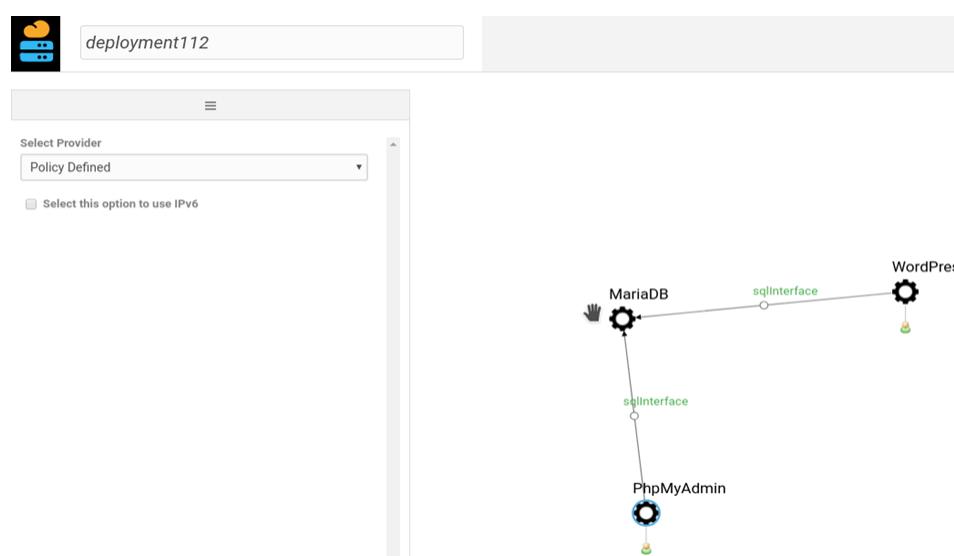


Figure 4: Triggering a deployment through MATILDA web interface

4 MATILDA Agent and Service Discovery Mechanisms

The MATILDA Agent's main duty is to handle the signalling (at layer 7) between the orchestrator and the core platform. Moreover, it contains functionality in order to **support efficient security policy enforcement at the component level**. Two crucial questions that are raised are: a) why at the component level? and b) what do we mean with the term efficient?

One of the traditional issues in security is the policy enforcement of **perimeter security rules**. This practically relates to the problem of translating a specific allowance/dropping policy to **tangible packet filtering business logic**. Such business logic can be provided by the **telco provider** (through proper programmability of the OSS layer) **or** through the **sidecar** of the vertical components. Which is the proper layer depends on the circumstances and the nature of cyber threats. MATILDA provides the ability to deploy security VNFs on the telco side in order to cope with packet filtering/dropping logic but also allows for the sidecar to perform such dropping also.

A generic **kernel-oriented method is used** which is totally IaaS-agnostic and highly efficient. As depicted, MATILDA offers a **Perimeter Security Policy Editor** which is used to provide allowance/dropping packet policies based on an abstract format. These rules are translated in specific scripts that are directly **executable by the kernel of the operating system** that is hosting the docker container of the component. This executable is formatted as an **extended-BPF (a.k.a. eBPF)** [eBPF] script that allows extremely efficient filtering of incoming packets.

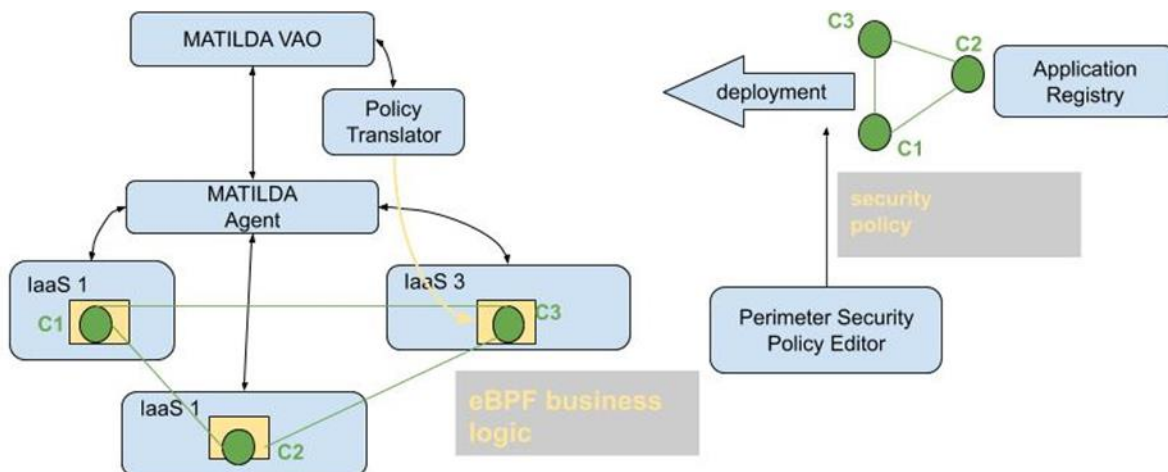


Figure 5: Perimeter security business logic

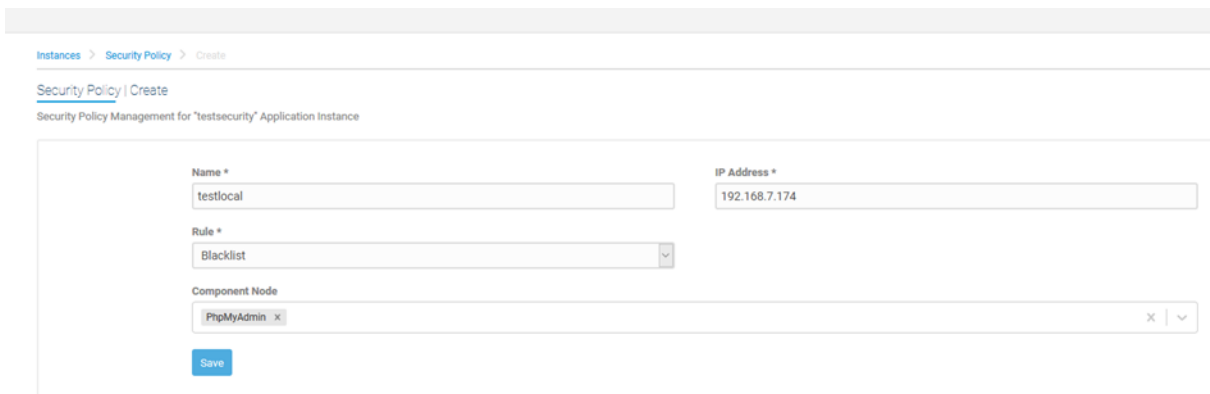
The need for fast **network packet inspection and monitoring** was obvious in early versions of UNIX with networking support. In order to gain speed and **avoid unnecessary copying with packet contents between kernel and userspace**, the notion of a **kernel packet filter agent was utilised** [eBPF]. The solution adopted by Linux is referred as Berkeley Packet Filter (BPF). This agent allows a userspace program to attach a filter program

onto a socket and limit certain dataflows coming through the socket in a fast and effective way. Linux BPF originally provided as set of instructions that could be used to program a filter: this is nowadays referred to as **classic BPF (cBPF)**.

Later a new, more flexible, and richer set was introduced, which is referred to as **extended BPF (eBPF)**. While originally designed for network packet filtering, nowadays Linux BPF is used in many other areas, including network security. eBPF allows users to drop, reflect or redirect packets **before they have a socket buffer metadata structure added to the packet**. This leads to a performance improvement of about 4-5 times.

MATILDA allows a) the creation of the eBPF script through a plain editor; b) the Loading of the script into the kernel and creating necessary eBPF-maps and c) the attachment of the loaded program to a system.

The MATILDA UI offers the aforementioned **Perimeter Security Policy Editor** which is used to provide allowance/dropping packet policies based on eBPF (see Figure 6). As already mentioned, these rules are translated in specific scripts that are directly executable by the kernel of the operating system that is hosting the docker container of the component. This executable is formatted in eBPF script and it is extremely efficient.



The screenshot shows the 'Security Policy | Create' form in the MATILDA UI. The breadcrumb navigation at the top reads 'Instances > Security Policy > Create'. Below this, the page title is 'Security Policy | Create' and the subtitle is 'Security Policy Management for "testsecurity" Application Instance'. The form contains the following fields: 'Name *' with the value 'testlocal', 'IP Address *' with the value '192.168.7.174', 'Rule *' with a dropdown menu showing 'Blacklist', and 'Component Node' with a tag 'PhpMyAdmin' and a close button 'x'. A 'Save' button is located at the bottom left of the form.

Figure 6: Adding a packet filtering rule

This approach also provides great flexibility, as application-specific firewall rules can be set by each application component individually (see Figure 7) and without requiring root access. These changes are performed instantly through the MATILDA UI.

Configure "MariaDB" Component
ID: vvMJJ456NS

Select SSH Key

adminKey

Select Provider

UBIDELL

Select Region *

gr-athens

Prevention Rules

☒ Activate IPS

Figure 7: Enabling Security Agent extensions for a component through the UI

The result of applying such a rule of a component is instantly realised, and in this case the blacklisting of an IP address is performed in a very efficient way that is suitable for both cloud and edge resources (see Figure 8).

Instances > testsecurity > Security Policy

Security Policy

Security Policy Management for testsecurity Application Instance

Name

Search by Name

Filter Reset

Name	Status	IP Address
policy1	APPLIED	213.249.38.66
testlocal	APPLIED	192.168.7.174

```

PS C:\Users\glede> ping 212.101.173.4
Pinging 212.101.173.4 with 32 bytes of data:
Reply from 212.101.173.4: bytes=32 time<1ms TTL=62
Reply from 212.101.173.4: bytes=32 time<1ms TTL=62
Reply from 212.101.173.4: bytes=32 time<1ms TTL=62
Reply from 212.101.173.4: bytes=32 time<1ms TTL=62
Ping statistics for 212.101.173.4:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
PS C:\Users\glede> ping 212.101.173.4
Pinging 212.101.173.4 with 32 bytes of data:
Reply from 212.101.173.4: bytes=32 time<1ms TTL=62
Reply from 212.101.173.4: bytes=32 time<1ms TTL=62
Reply from 212.101.173.4: bytes=32 time<1ms TTL=62
Reply from 212.101.173.4: bytes=32 time<1ms TTL=62
Ping statistics for 212.101.173.4:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 1ms, Average = 0ms
PS C:\Users\glede> ping 212.101.173.4
Pinging 212.101.173.4 with 32 bytes of data:
Request timed out.
        
```

Figure 8: Packet filtering activation example

4.1 Main components

As already mentioned, when a vertical application is deployed to a MATILDA enabled provider each component is associated with a VM and each VM is spawned simultaneously. The reason for that 'parallel' spawning is that the VM booting time is a significant portion of the total time that is required for a vertical component to be operational. Upon VM spawning there are 7 discrete steps that have to be executed in order for the vertical component to be operational.

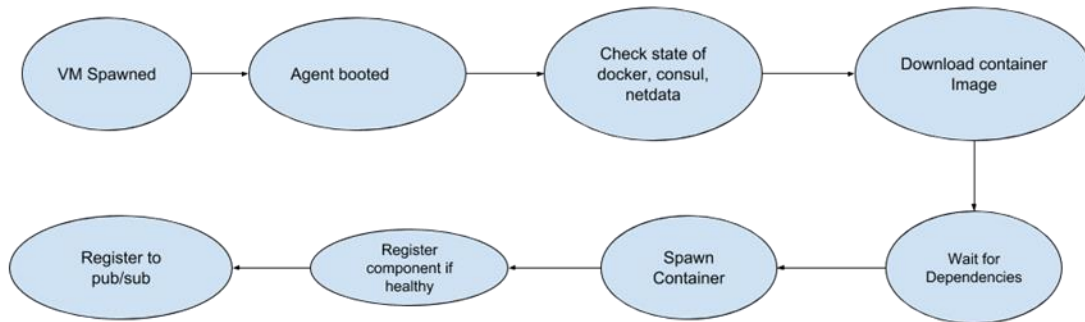


Figure 9: Lifecycle of MATILDA Agent

These steps are graphically depicted in Figure 9 and will be explained in detail:

Step 1 – Agent booted: During this step the small-footprint Agent is loaded. That verifies that the VM boot process has completed successfully and the initialization script (init.d) that was passed as an argument from the Deployment Manager was valid.

Step 2 – Check executable prerequisites: During this step the Agent tries to resolve if three prerequisite executables are already installed in the VM. These are a) the monitoring probe (i.e. Netdata), b) the Container Engine (i.e. docker-engine) and c) the Service Discovery Client (i.e. Consul). If these prerequisites are not met, the Agent terminates abnormally. If they are met, the component is registering to the SDS server.

Step 3 – Fetch Image of Vertical Component: During this step the actual transfer of the executable of the vertical component is performed. In order to cope with the problem of vendor lock-in format of the executable the container format has been chosen.

Step 4 – Block until dependencies are resolved: During this step the Agent is trying to identify what is the operational state of the vertical components that are direct dependencies to the component that is bound to the Agent. To do so, the Agent is not contacting other Agents directly because this would be inefficient and problematic. Instead, it queries the SDS server to fetch the latest state of each direct dependency. If the operational state of all dependencies is not satisfied, then the Agent blocks.

Step 5 – Spawn container of Vertical Component: When all dependencies are operational, the Agent is triggering the execution of the pulled container.

Step 6 – Register component to SDS when health-check passes: Upon triggering of the execution, the Agent is polling the service in order to infer whether or not the booted service is actually running. If this 'health-check' is successful, then the Agent notifies the SDS that the component is up and running.

Step 7 – Register to a pub/sub queue: During this step the Agent is polling a pub/sub system for specific commands that may be issued by the Deployment and Execution Manager. The commands that are rather crucial are the **perimeter security commands that are implemented using the BPF technology**.

Figure 10 provides indicative agent states during one deployment in a telco provider.

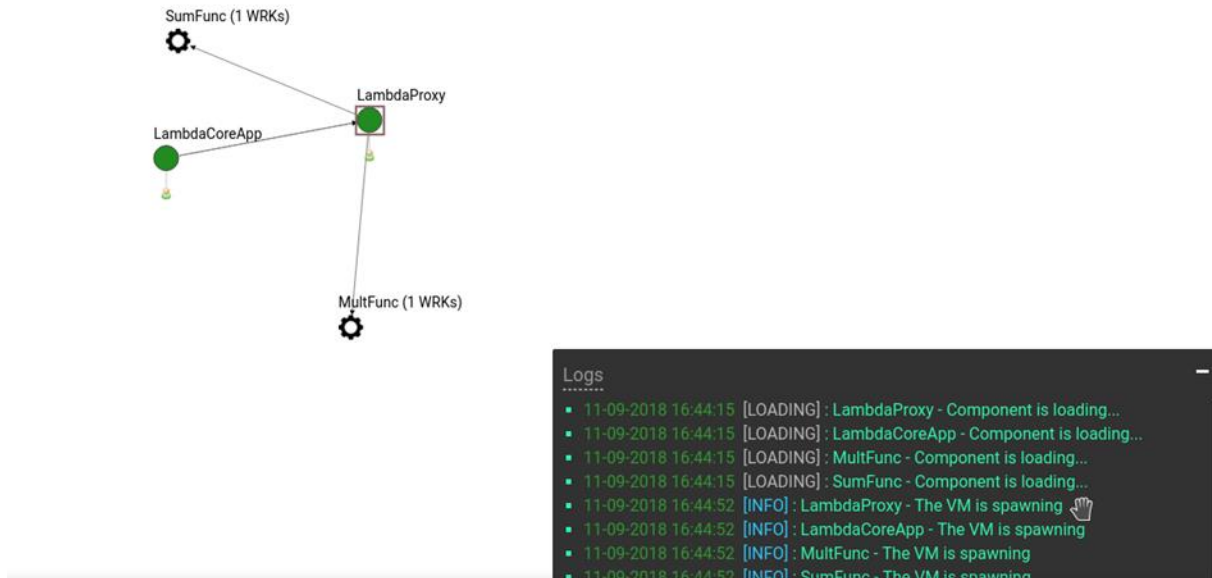


Figure 10: Agent steps

As inferred from the steps above, the Service Discovery Server is a rather crucial component of the architecture, since it acts as a Key-Value store which is accessible by all Agents that are booted. In this store, all aspects regarding Agent arguments, vertical component dependencies and docker image location are provided. Figure 11 illustrates indicative data that are persisted in the SDS during one deployment.

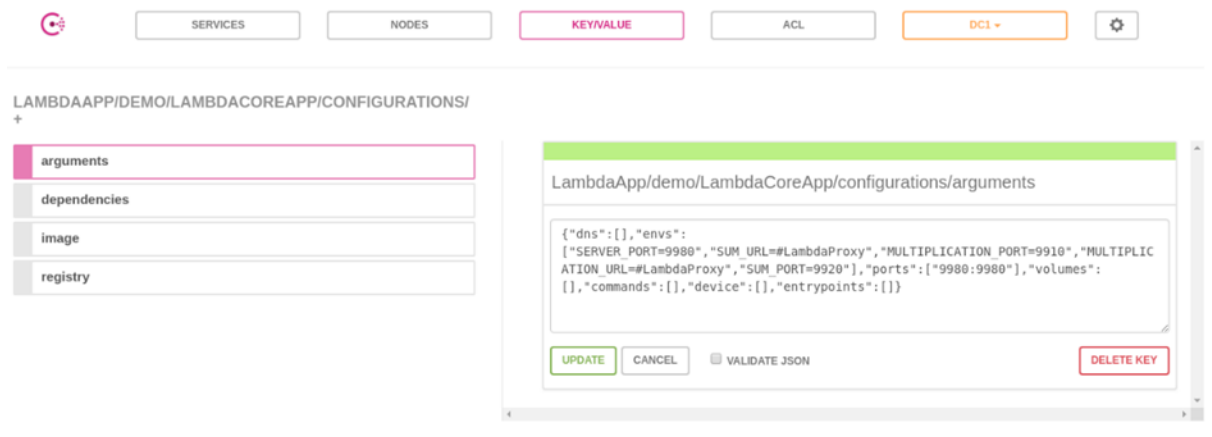


Figure 11: Agent information stored in the SDS

4.2 Interfaces

The interfaces of the Agent and the SDS are already described in Section 3 and depicted in Figure 3. It should be noted that the Agent is architected in such a way that no synchronous API call is exposed to it. The reason for that is that most of the times Agents will be operating behind NAT. Hence all communications that have to be performed are undertaken by the SDS.

4.3 *Baseline technologies*

The technology base of the SDS is Consul. Consul is a distributed service mesh to connect, secure, and configure services across any runtime platform and public or private cloud. The selected SDS provides a full featured control plane with service discovery, configuration, and segmentation functionality. Each of these features can be used individually as needed, or they can be used together to build a full-service mesh. The key features of Consul are:

- **Service Discovery:** Clients of Consul can register a service, such as API or MySQL, and other clients can use Consul to discover providers of a given service. Using either DNS or HTTP, applications can easily find the services they depend upon.
- **Health Checking:** Consul clients can provide any number of health checks, either associated with a given service ("is the webserver returning 200 OK"), or with the local node ("is memory utilization below 90%"). This information can be used by an operator to monitor cluster health, and it is used by the service discovery components to route traffic away from unhealthy hosts.
- **KV Store:** Applications can make use of Consul's hierarchical key/value store for any number of purposes, including dynamic configuration, feature flagging, coordination, leader election, and more. The simple HTTP API makes it easy to use.
- **Secure Service Communication:** Consul can generate and distribute TLS certificates for services to establish mutual TLS connections. Intentions can be used to define which services are allowed to communicate. Service segmentation can be easily managed with intentions that can be changed in real time instead of using complex network topologies and static firewall rules.
- **Multi Datacentre:** Consul supports multiple datacentres out of the box. This means users of Consul do not have to worry about building additional layers of abstraction to grow to multiple regions.

5 Monitoring Mechanisms

One of the major challenges of the MATILDA VAO was the **unification of the monitoring streams** that are generated from the operation of the various components/layers. These metrics are categorized in the following groups:

a) Infrastructure-benchmarking: Such metrics quantify the **quality of the provided IaaS** resources (from the telco-provider) during the slice creation. They refer to CPU speed, amount of memory, storage speed (IOs per second), etc. These measurements are performed by the **VAO Agent prior to the deployment** of a vertical component.

b) Probable-Vertical Component Runtime metrics: Such metrics quantify the several execution parameters that can be **measured passively, i.e. through a probe**. Such probes are installed and parameterized by the VAO Agent.

c) Vertical Component Runtime exportable metrics: Such metrics quantify the several execution parameters that are **exposed by the Vertical component per se**. The export process must follow guidelines. To do so, exporter libraries for Java and Python have been developed in order for developers to be able to follow the norms.

d) Communication Service Provider Metrics: These are metrics that are measured **within the administrative zone of the OSS** and they are performed by specific VNFs that are dynamically deployed.

As is easily inferred, these metrics refer to both VAO and OSS. As such, the **unification process is logical and not physical**. This is imperative for political reasons (i.e., non-technical). After extensive discussions within the consortium regarding this ‘unification issue’, it turned out that telco providers are extremely reluctant to expose specific types of OSS-monitoring streams outside of their administrative boundaries. This is the reason why two Metric aggregation platforms have been setup (Prometheus) in order to contain the full set of measurements.

5.1 MATILDA Overall monitoring solution

The MATILDA monitoring solution addresses multi-site network infrastructure deployments and performs metrics acquisition from a variety of domains. Specifically, the resources to be monitored fall in one of the following domains:

- NFV Infrastructure (NFVI) resources that comprise of physical and virtual compute, network and storage resources
- SDN-enabled elements, including physical and virtual resources
- Physical devices that do not belong to the previous categories, such as non-SDN compliant network routers and switches for which we want to capture monitoring information
- Linux containers deployed to run application components that form the 5G-ready vertical applications.

The monitoring system is responsible for the management of the metrics captured from the various infrastructure components, the management of alerts and events based on these metrics, and the visualization of the available data.

Furthermore, the monitoring mechanisms can operate in passive or active manner. Passive monitoring in MATILDA refers to the capture of service and network metrics locally at the application or VNF component level. Example of such metrics are CPU utilization, RAM usage, etc. On the other hand, the active monitoring provides QoS/QoE measurements based on the injected traffic by the monitoring application itself. The simplest of such monitoring tools in MATILDA would be ICMP (Internet Control Message Protocol) PING request/reply mechanism that enables measuring the RTT (round-trip time) between application components or application component and UE.

In Figure 12, the general architecture regarding application component monitoring is presented. The MATILDA monitoring solution is based on the Prometheus monitoring tool [Prometheus] which will collect and aggregate the monitoring data from all monitoring systems (i.e. active/passive KPIs at the application component and VNF KPIs from the OSM).

The logical entities forming MATILDA Prometheus-based architecture are as follows:

- Data collection:
 - Prometheus Server with Collector engine as the core module of Prometheus framework that is responsible for polling the measured data from monitoring targets.
 - Prometheus Push Gateway module that will allow the MATILDA framework to push the data from monitored components when the components cannot be polled directly from Prometheus (e.g., the component is behind the FW/NAT).
 - Prometheus Alerts module that support triggering and sending out alerts via mail or various other services (e.g. Slack, Telegram, etc.) based on the predefined application or network KPI threshold.
- Data sources:
 - Netdata [Netdata] agent on each application component that provides **passive** monitoring KPIs at the VM/Container level. Currently, the plan is to integrate the Netdata agent in the base VM image hosting the application component.
 - qMON agent [qMON] on each application component which enables **active** monitoring mechanisms such as ping and download/upload bandwidth capacity measurements. Currently, the qMON agent can operate at the application component level as a separate Docker container.

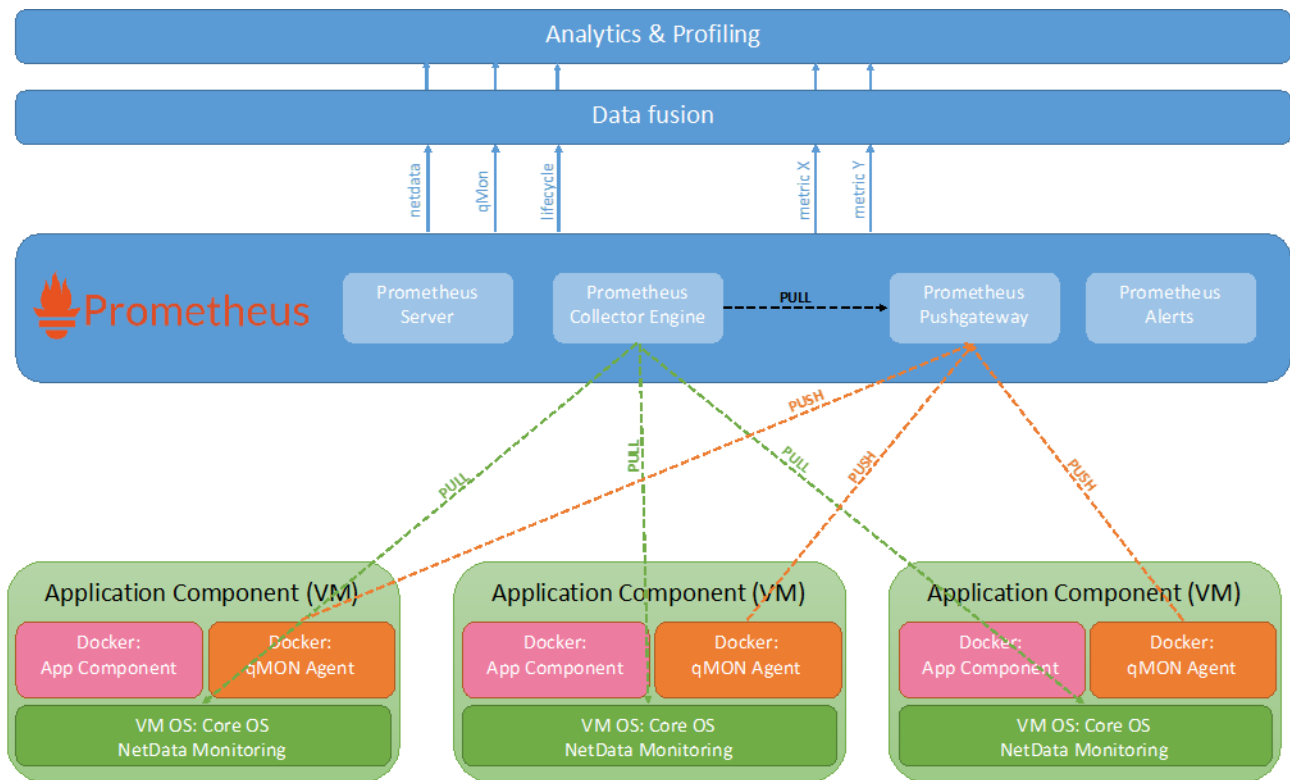


Figure 12: Prometheus-based MATILDA monitoring architecture supporting active and passive measurement scenarios

5.1.1 Application component monitoring

A crucial task when defining the Monitoring and Analytics architecture is the identification of metrics that need to be collected from the project's infrastructure, including the NFVI, the physical infrastructure devices apart from the NFVI and the deployed NSs, VNFs and containers. The table below summarizes a list of such metrics, which are "generic" and are targeted to serve measurements tailored to the key drivers of MATILDA. This list is meant to be continuously updated throughout the project in order to align with the technical capabilities and requirements of the components under development and the use cases which are implemented.

Table 1: List of metrics to be collected through MATILDA monitoring solution

Metric	Unit	Category	Method/data source
CPU utilization	percent %	Host node and VNF/Container generic metrics	Passive/netdata
RAM allocated	MB	Host node and VNF/Container generic metrics	Passive/netdata
RAM available	MB	Host node and VNF/Container generic metrics	Passive/netdata

Metric	Unit	Category	Method/data source
Network interface in/out bitrate	Mbps	Host node and VNF/Container generic metrics	Passive/netdata
Network interface in/out packet rate	pps	Host node and VNF/Container generic metrics	Passive/netdata
Disk read/write rate	MB/s	Host node and VNF/Container generic metrics	Passive/netdata
Port in/out bit rate	Mbps	Network generic metrics	Passive/netdata
Port in/out packet rate	pps	Network generic metrics	Passive/netdata
Port in/out packet drops rate	pps	Network generic metrics	Passive/netdata
VM provisioning latency	Msec	Service quality metrics (speed)	Passive/
Packet round-trip time (RTT)	ms	Service quality metrics (speed)	Active/qMON
Packet delay variation (jitter)	ms	Service quality metrics (speed)	Active/qMON
Download throughput	kbps	Service quality metrics (capacity)	Active/qMON
Upload throughput	kbps	Service quality metrics (capacity)	Active/qMON
Packet loss rate	pps	Service quality metrics (accuracy)	Active/qMON

Application component monitoring - passive

The Netdata agent can expose these metrics via local HTTP/HTTPS server on each application component, so the data can be polled on the predefined time intervals. The example KPI exposed on the application component is shown below:

- CPU Utilization

```
netdata_services_cpu_percent_average{chart="services.cpu", family="cpu", dimension="apache2", instance="netdata-collector"} 0.0000000 1535625324000
```

Application component monitoring - active

On the other hand, the qMON agent typically operates as a monitoring client that uses simple push mechanism to send monitoring data to the Prometheus. In the following paragraphs, the sample forms of measured metrics are shown as they are exposed through Prometheus Push Gateway.

- Packet round-trip time (RTT)

```
inin_ping_client_lat{aliasHash="57b34",client_ip="-1",client_ipv4_for_geoloc="-1",client_version="5.0.51-iptv-dev",collector_host="qoe-volta",geohash="u25j6r0",gps_client_loc_hash="true",hash="57b34bad5040038c1edf12ab694cb306f1f1b556",ip_version="4",os_name="Ubuntu",os_version="14.04",seq_no="0",status="Success",target_ip="8.8.4.4",test_type="ping_test"} 45.96152
```

- Download/upload throughput

```
inin_iperf_duration{aliasHash="a0adb",client_ip="10.0.151.111",client_ipv4_for_geoloc="10.0.151.111",client_version="5.0.51-docker",collector_host="qoe-volta",direction="DL",geohash="7zzzzzzzzzz",gps_client_loc_hash="false",hash="a0adb5fc2c39485bf21fe2f3f121408d19a824f",ip_version="4",iperf_clone="true",os_name="Ubuntu",os_version="16.04",status="Success",target_ip="10.0.151.122",test_type="iperf_test"} 15.0002
```

- WEB MOS

```
inin_web_web1_mos{aliasHash="smevo",client_ip="-1",client_ipv4_for_geoloc="-1",client_version="4.3.1lk",collector_host="kette",gps_client_loc_hash="false",hash="smevosdrpi",ip_version="4",os_name="debian",os_version="8.0",target_url="www.google.com",test_type="web_test",web1_status="Success"} 4.86
```

The qMON-based MATILDA architecture will allow two types of active measurements:

- Using a reference measurement server that supports all types of measurements and represents the entity outside the scope of MATILDA framework (Figure 13). A special case of such architecture would put the UE in place of the reference server, and in this way end-to-end network service KPIs can be measured (e.g., SLA Monitoring for the PPDR Use Case).
- Between application components in single or multi-IaaS environment that will enable real-time network KPIs that can be used to trigger rules regarding network service (Figure 14).

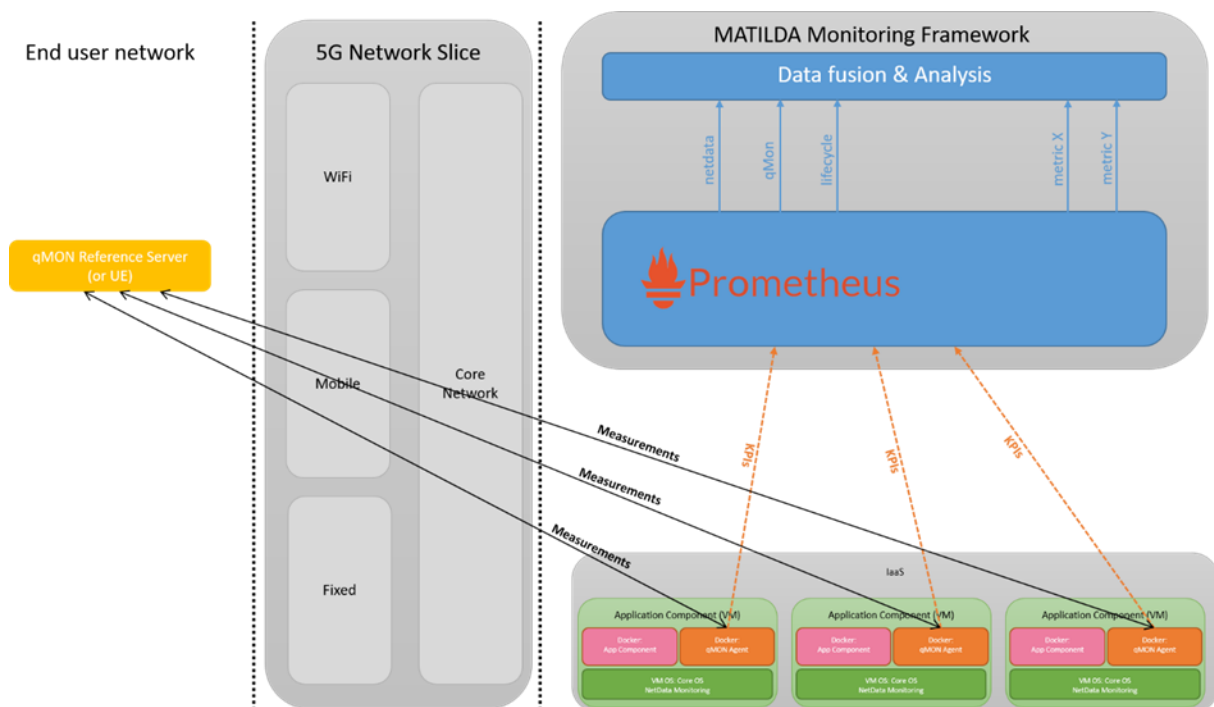


Figure 13: MATILDA Active network monitoring with reference server or UE

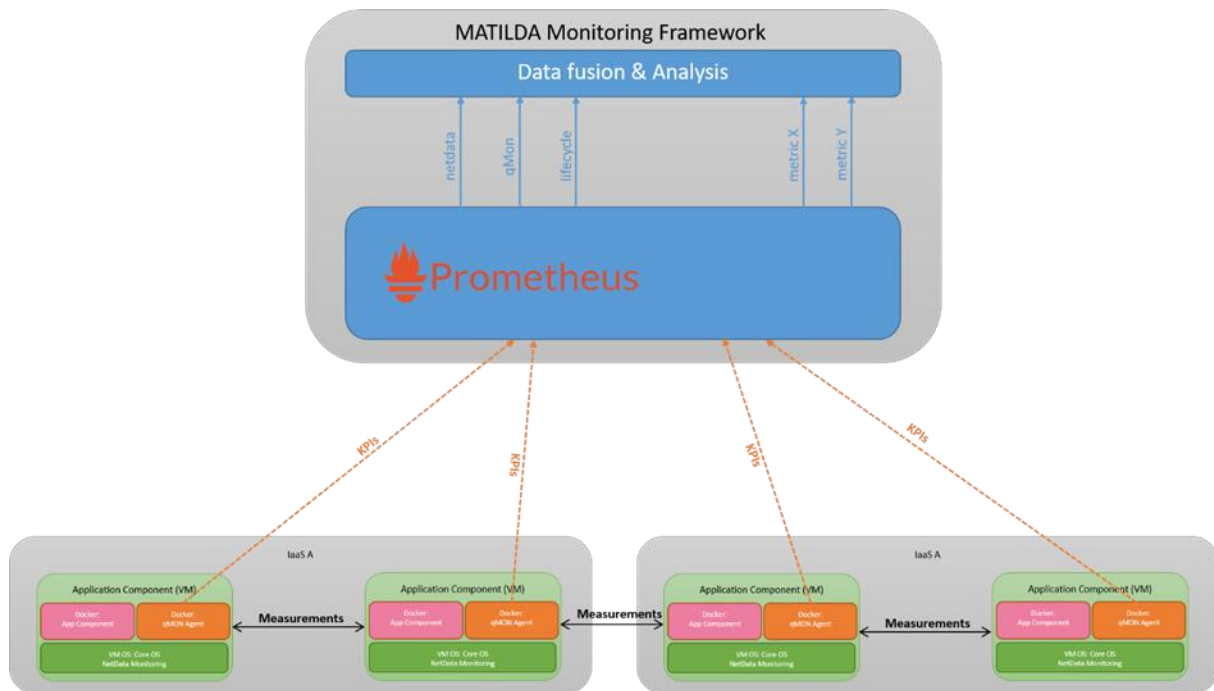


Figure 14: MATILDA Active network monitoring between application components

5.1.2 Network Slice monitoring

Network slice monitoring will provide metrics based on two operations: NFVO/OSM metric collection for monitoring of mainly compute resources consumed by VNFs and qMON NFV-based metric collection for monitoring network services and/or end-to-end services.

5.1.3 NFVO/OSM metric collection

The hardware resources consumed by each of the VNF instances should be monitored in order to measure its performance. For this matter, the NFVO (OSM) features a Kafka bus for asynchronous communications, performance/fault management features and an NBI exposing a unified REST API [OSM]. This architecture is shown in Figure 15.

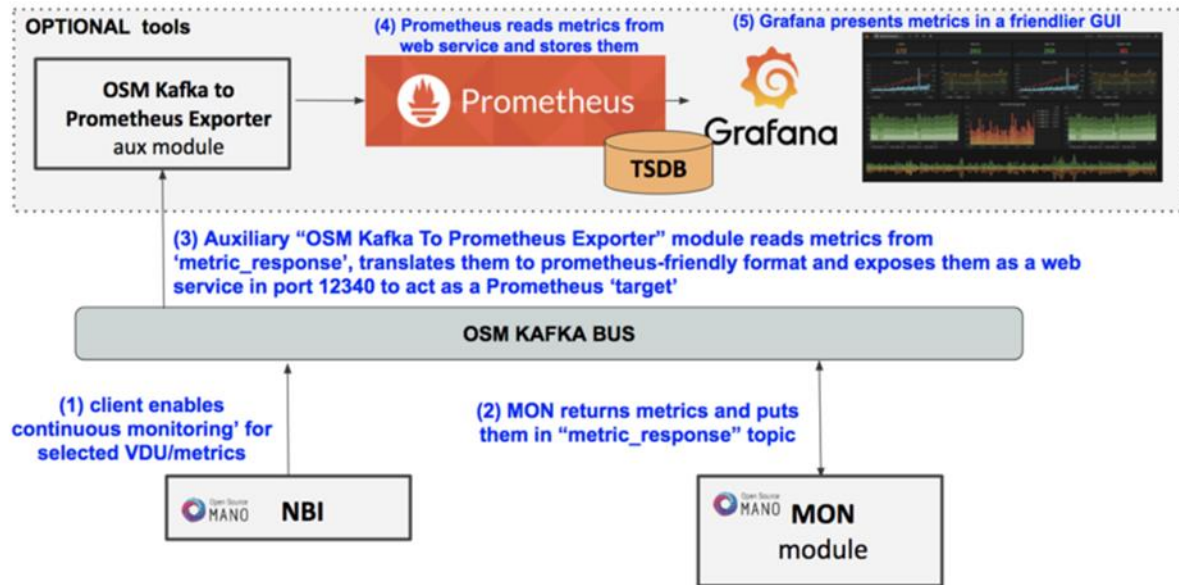


Figure 15: Basic OSM monitoring architecture

Once deployed the VNFs in the VIMs throughout the NFVI (utilizing Openstack with Gnocchi service to support these capabilities), the metrics that are going to be exposed at the Kafka bus must be exported per VNF and VDU (VM). The supported metrics are described in Table 2.

Table 2: Supported metrics by the OSM Kafka Bus

Metric	Unit	Category
cpu_utilization	percent %	Host node and VNF/Container generic metrics
average_memory_utilization	MB	Host node and VNF/Container generic metrics
disk_read_ops	ops	Host node and VNF/Container generic metrics
disk_write_ops	ops	Host node and VNF/Container generic metrics
disk_read_bytes	bps	Network generic metrics
disk_write_bytes	pps	Network generic metrics
packets_dropped	pps	Network generic metrics
packets_received	pps	Service quality metrics (speed)
packets_sent	pps	Service quality metrics (speed)

The architecture in Figure 16 provides information on how OSM can be monitored. Given the need to receive monitoring information for the NSs and the slices, the "OSM Performance Management" is under extension to use the MATILDA Prometheus. Monitoring information regarding the VNFs is not relevant for the MATILDA OSS, as it is not aware of how the Telco

Provider instantiated the slice. Therefore, a new module is under development to aggregate the VNF monitoring information into NS and NS monitoring into slice monitoring that is indeed more useful for calculating the overall 5G-ready application monitoring.

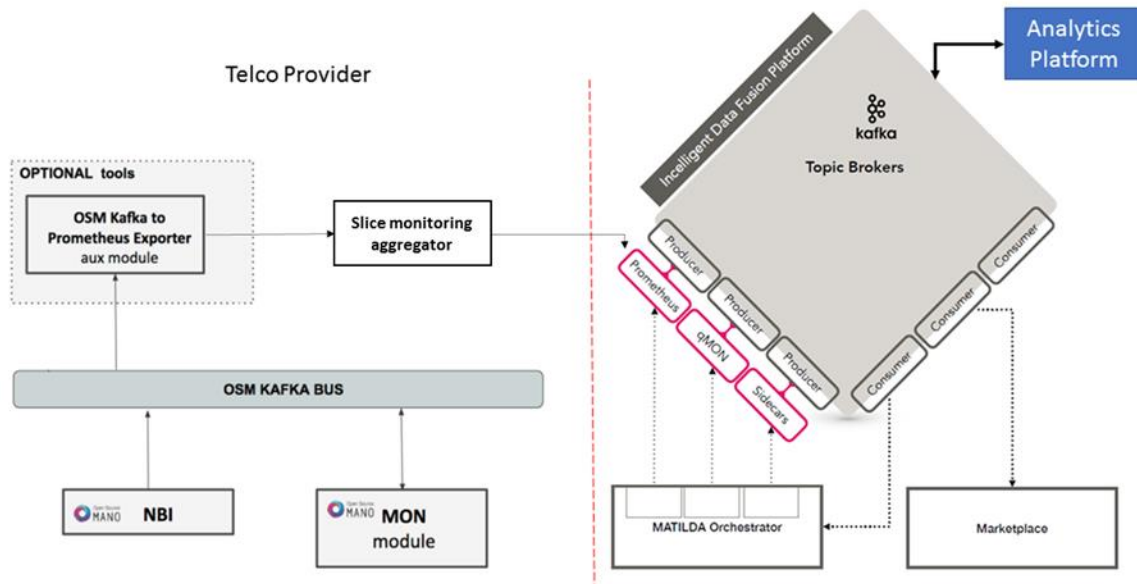


Figure 16: Integrated monitoring environment

Main components

As depicted in Figure 17, the components comprising the Monitoring platform are the Alert Engine, the Notification Engine, the Persistence Manager, the Storage Database, the Aggregation Manager and the Visualisation component. The Analytics module as cited in the figure above, Figure 16, consumes and analyses information from the monitoring platform to identify trends and meaningful patterns. Additional information is provided in the next section of this report.

The Alerting engine is responsible for the management of alerts (creation, deletion). For the management of alerts, the Alert Engine should communicate with the Orchestration layer components. The Notification Engine should allow the different orchestration-level components to receive notification messages of events by offering the necessary interfaces for subscribers to receive events of interest. For generating notifications on events, the Alert Engine is evaluating the metrics received against the created alerts and upon an event occurrence, the Notification Engine distributes notifications to the subscribers of the specific event.

The Aggregation Manager is responsible for collecting metrics from different sources and aggregating them over a specific timeframe exploiting several statistical functions or other filtering options. Data gathered from the Aggregation Manager will be sent to the Persistence Manager to store selected collected metrics and alerts at the Storage Database for preserving historical data required for off-line analysis and future reference. Finally, the Visualization component will offer a dashboard for the visualization of captured metrics and created alerts.

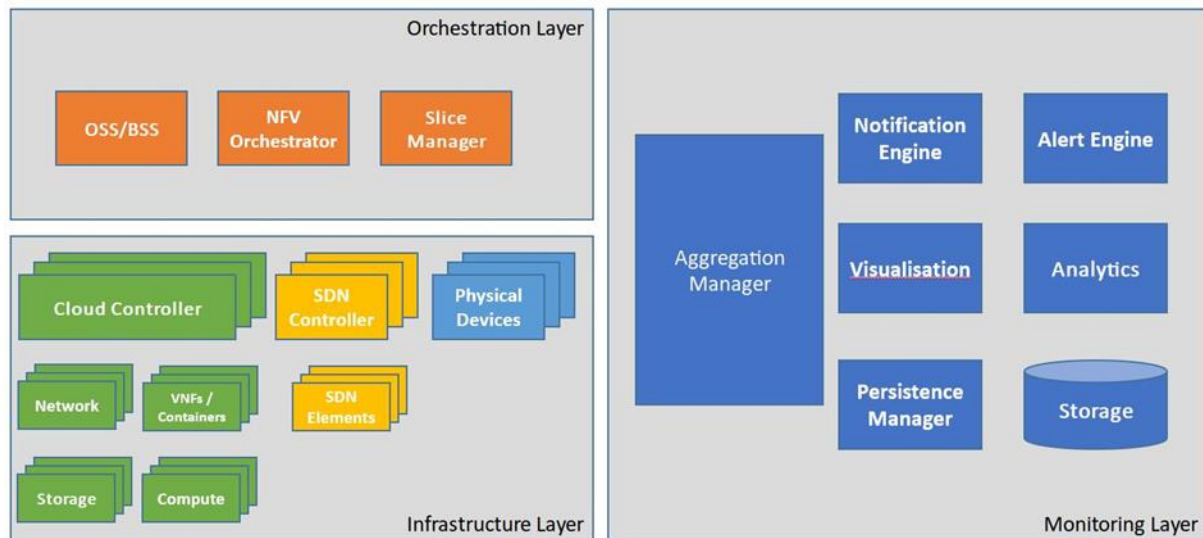


Figure 17: Monitoring solution high-level architecture

5.1.4 qMON NFV-based monitoring

Additional to NFVO/OSM metric collection the qMON NFV-based monitoring will provide network KPIs for the network service monitoring on the end-to-end principle. The qMON NFV-based solution will extend the overall MATILDA active and passive monitoring presented in chapter 5.2 with the capabilities for active network service monitoring as shown in the figure below.

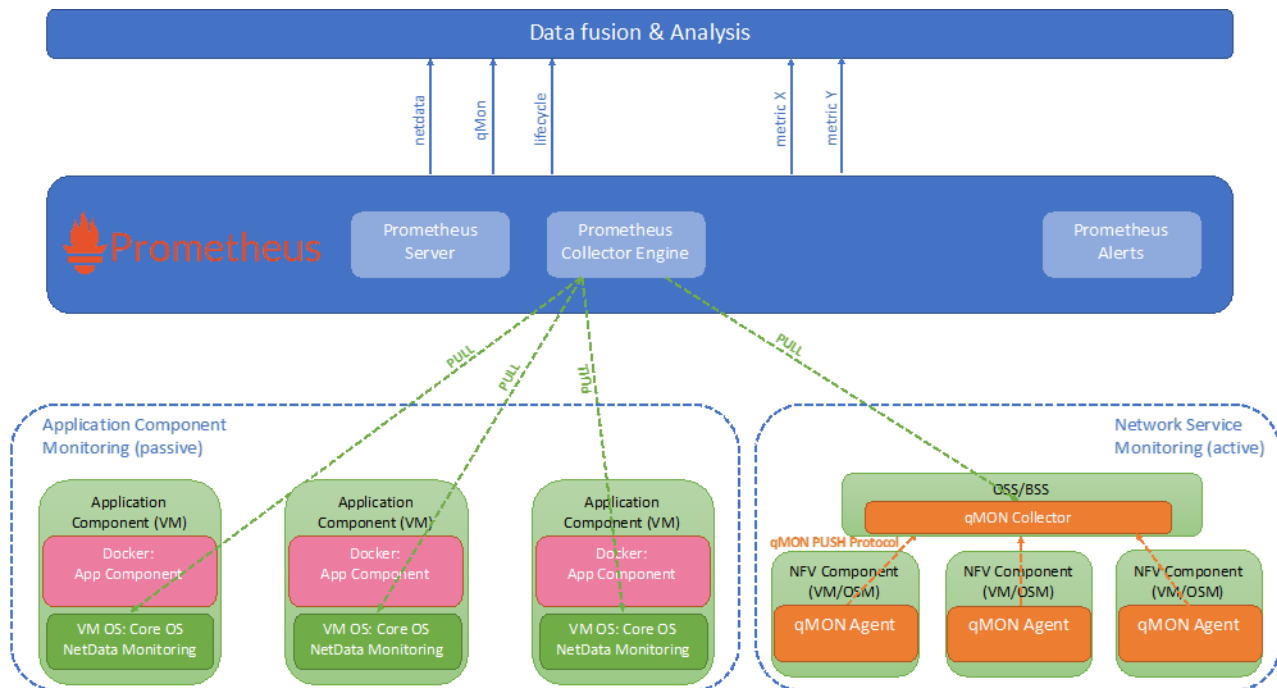


Figure 18: MATILDA monitoring architecture with integrated qMON NFV-based solution

Main Components

The qMON NfV-based solution consists of the following modules:

- qMON VNF Agent,
- qMON VNF Server,
- qMON Collector.

The qMON VNF Agent and qMON VNF Server are OSM-compliant VNFs which are orchestrated through the OSS on-demand when the request for active monitoring comes from MATILDA application orchestrator through the slice intent mechanism and is then forwarded to the OSS as presented in the following figure.

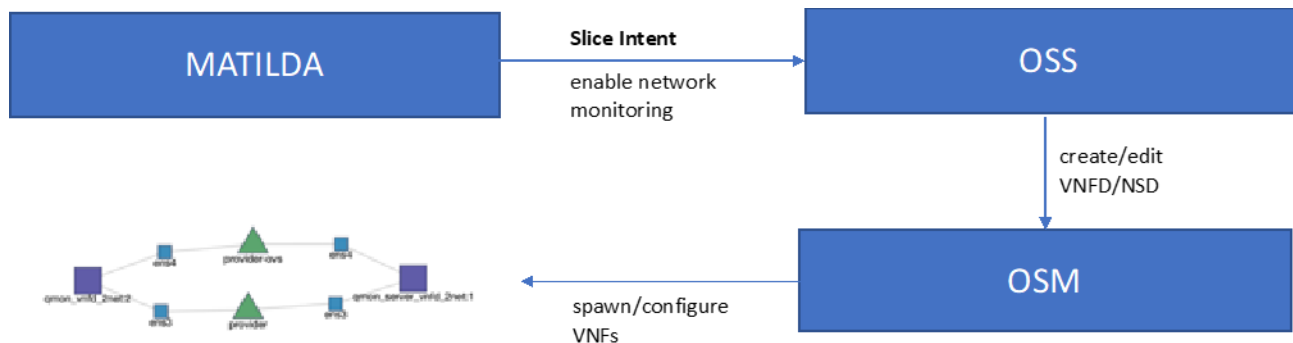


Figure 19: MATILDA monitoring deployment steps

The qMON VNF Agent represents the monitoring client while the qMON VNF Server provides the measurement endpoint (server) for monitoring clients. Single qMON VNF Server can provide measurement endpoint to multiple qMON VNF Agents in parallel.

The qMON Collector acts as a centralized qMON results collection endpoint to which the qMON VNF Agents upload measurements results via proprietary qMON protocol. At the same time, it acts as the actual data source (Prometheus target) for MATILDA-integrated Prometheus. qMON Collector is not orchestrated through MATILDA or OSS/OSM but is rather treated as a requirement for the infrastructure provider that supports network service monitoring.

Such an architecture allows various monitoring scenarios, e.g. including monitoring links between different IaaS hosts (Figure 20) or end-to-end network service monitoring (Figure 21).

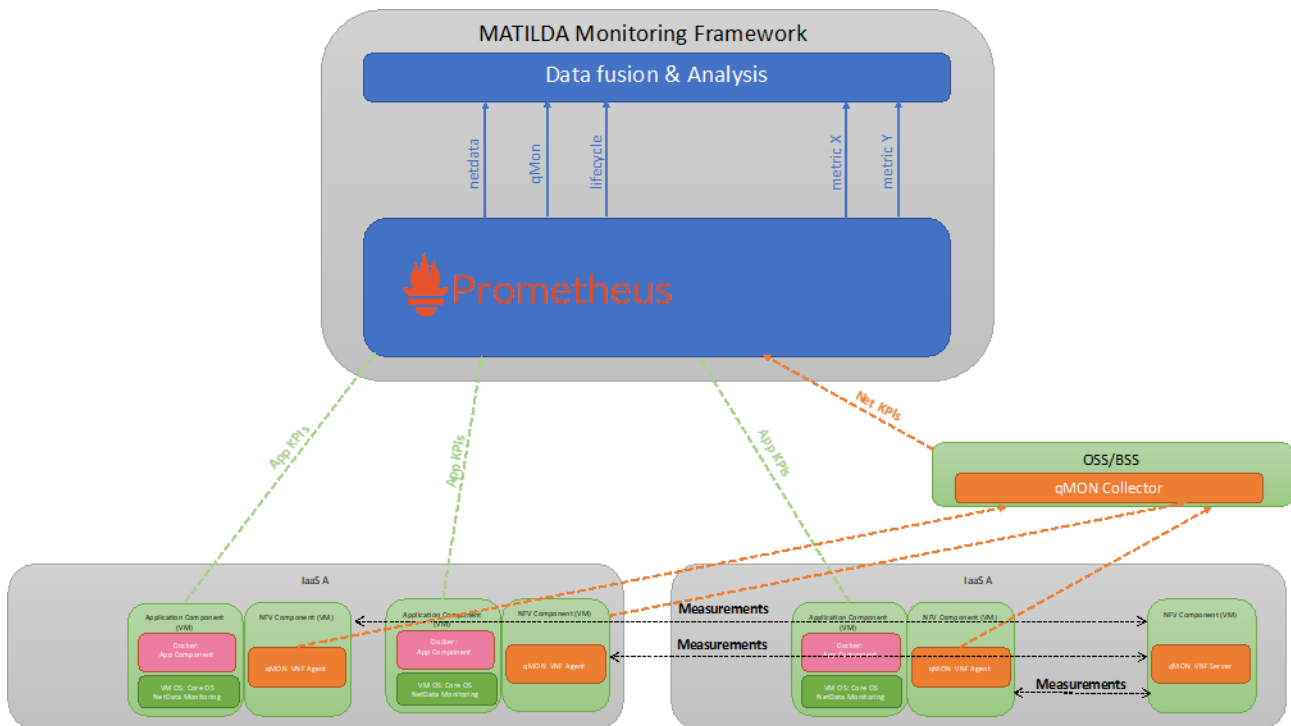


Figure 20: MATILDA qMON NFV-based active monitoring links between hosts or between IaaS

As shown in the figure, such a deployment architecture allows measurements of different network segments, e.g. between network core (i.e. EPC) and access (i.e. ENB) where qMON VNF Server would be placed in IaaS along with ENB and the qMON VNF Server in IaaS where the EPC network components are deployed.

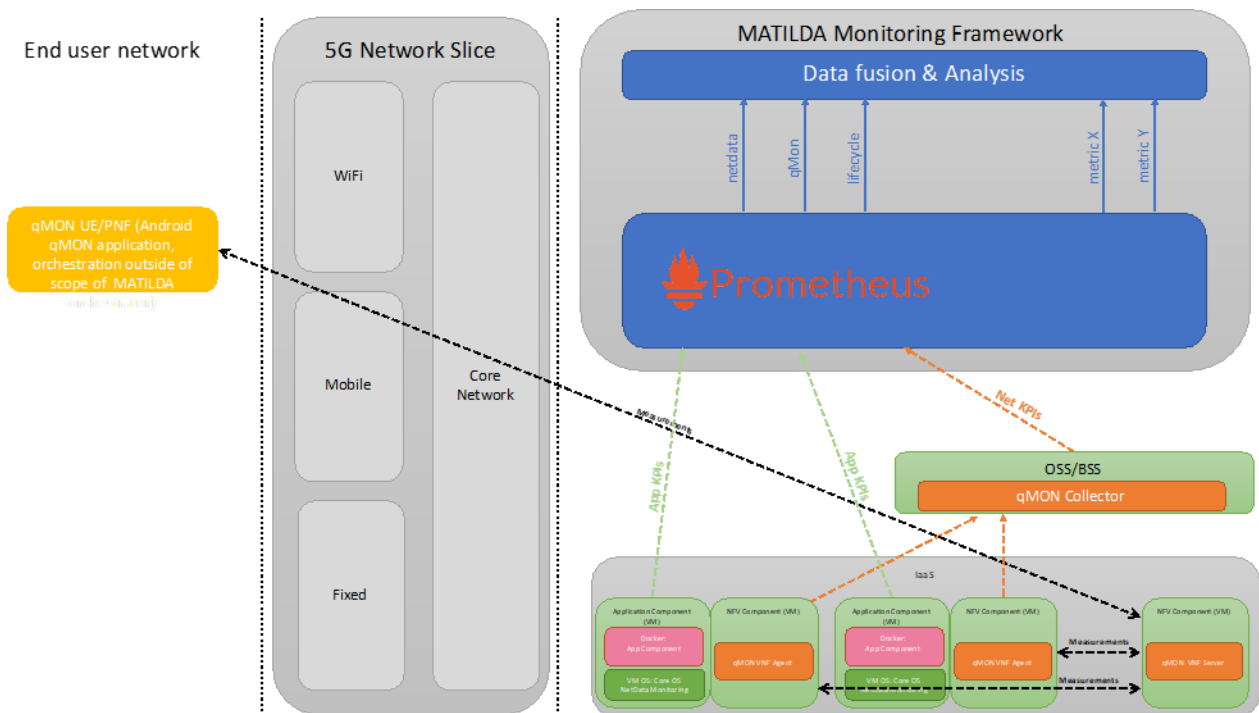


Figure 21: MATILDA qMON NFV-based active end-to-end network service monitoring

The figure above shows how the qMON NFV-based monitoring solution can be used to provide end-to-end service monitoring. Similar as in previous case, the qMON VNF Agents and qMON VNF Server are deployed on-demand through slice intent mechanism requesting network service monitoring from the OSS/OSM. Additionally, the qMON UE⁴ (i.e. Android qMON application) acts as a PNF (qMON PNF Agent) that can measure end-to-end network service KPIs between UE and qMON VNF Server placed somewhere in the core IaaS. This way, the general end-to-end network service KPIs can also be made available through qMON Collector where the results are collected from qMON VNF Agents and qMON UE. This scenario provides a mechanism to support SLA monitoring that is required by the PPDR use case.

Additionally, as qMON UE provides also physical network layer KPIs (e.g. radio) and since the qMON NFV-based solution should be treated as part of the infrastructure provider/telco, detailed radio KPIs (e.g. RSRP, RSRQ, EARFCN, APN etc.) are available to infrastructure provider/telco along with general end-to-end network service KPIs. The scenario is presented in the following figure.

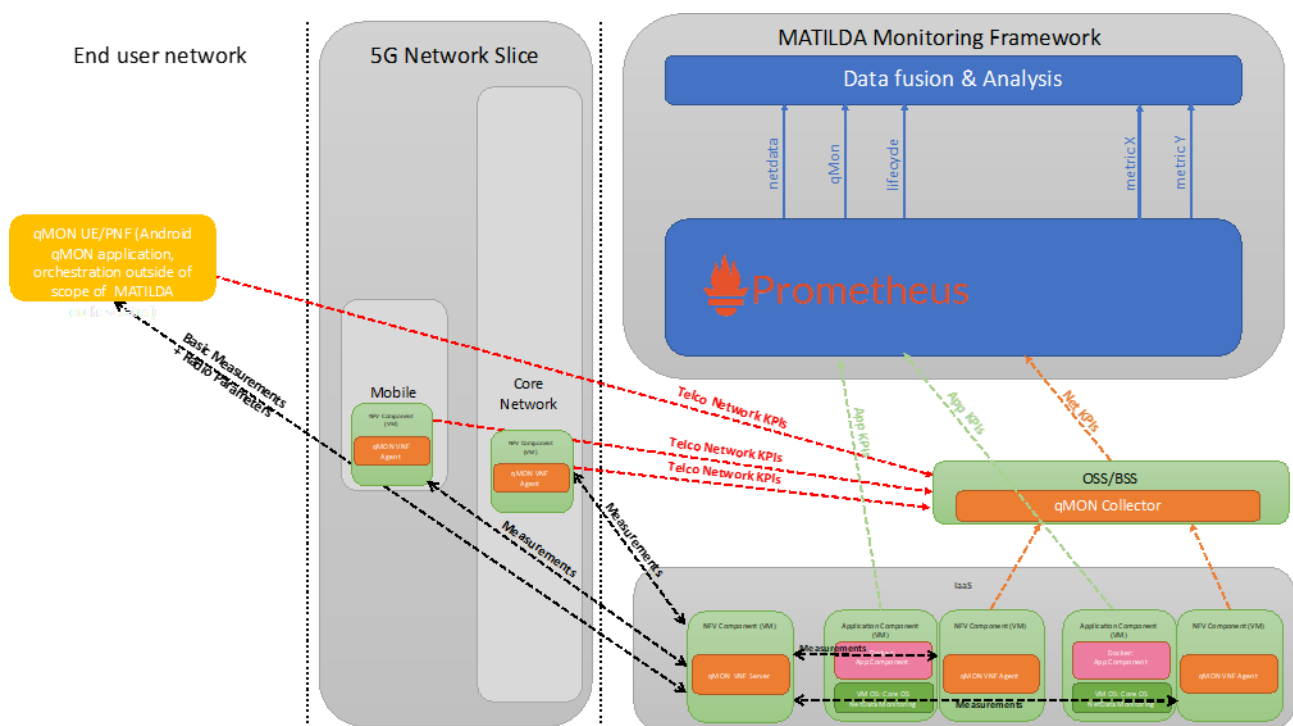


Figure 22: MATILDA qMON NFV-based active end-to-end network service monitoring with detailed telco KPIs

The telco provider is not necessarily willing to expose all the detailed radio KPIs. The collected data is controlled by the telco but can be made available also to the MATILDA-integrated Prometheus since the data is still collected by the qMON Collector.

⁴ qMON UE is not part of MATILDA orchestration.

Network service KPIs

The general network service KPIs collected by qMON NFV-based solution are basically the same as presented in the overall MATILDA monitoring solution with the means to ensure the MATILDA vertical application orchestrator (VAO) can use monitoring data in both cases, i.e. application component monitoring and network service monitoring. Available KPIs are not limited to this list but can be extended to provide advanced scenarios for different use cases.

Table 3: MATILDA qMON NFV-based active monitoring KPIs

Metric	Unit	Category	Method/data source
Packet round trip time (RTT)	ms	Network service quality metrics (speed)	Active/qMON VNF
Packet delay variation (jitter)	ms	Network service quality metrics (speed)	Active/ qMON VNF
Download throughput	kbps	Network service quality metrics (capacity)	Active/ qMON VNF
Upload throughput	kbps	Network service quality metrics (capacity)	Active/ qMON VNF
Packet loss rate	pps	Network service quality metrics (accuracy)	Active/ qMON VNF

The following example for “Packet RTT” metric is presented in Prometheus data format to show how the qMON collected data is to be used by the MATILDA VAO.

```
inin_ping_rtt_ms{aliasHash="cade0",client_ip="10.0.151.102",client_ipv4_for_geoloc="",client_version="4.2.9lk",collector_host="qoe-volta",gps_client_loc_hash="false",hash="cade069207747d2f016e5eef3f253b7083f8ddb",ip_version="4",os_name="Ubuntu",os_version="14.04",seq_no="0",status="Success",target_ip="193.2.1.66",test_type="ping_test"} 2.3
```

Important parameters:

- **client_ip**: qMON Client VNF IP (e.g. Openstack data plane IP of the qMON Agent VNF);
- **target_ip**: configured end point IP to which RTT is measured (e.g. qMON Server VNF on the same Openstack data plane allowing communication between qMON Agent VNF and qMON Server VNF);
- **test_type**: “packet delay” is identified as “ping_test” test type, will be different for other test types, e.g. DL/UL;
- **hash**: unique ID for the qMON Client VNF to be able to register to and get the configuration from the qMON Management Server;
- **value**: round-trip time (RTT) between the client and server (e.g. qMON Client VNF and qMON Server VNF).

Infrastructure provider/telco network KPIs

As already mentioned above, the qMON NFV-based solution can provide additional physical layer KPIs measured by the qMON UE. Typically, the infrastructure provider/telco will use this data for internal monitoring architecture and quality assurance scenarios, however the presented qMON NFV-based monitoring architecture allows this data also being exposed to the MATILDA monitoring platform if needed. The metrics collected are presented in the following table. The list is not limited to these KPIs and can be extended to support various use case requirements.

Table 4: MATILDA qMON UE collection of radio KPIs

Metric	Unit	Category	Method/data source
RSRP	dBm	Radio network quality metrics (signal strength)	Active/qMON UE
RSRQ	dB	Radio network quality metrics (signal strength)	Active/ qMON UE
SINR	dB	Radio network quality metric (signal strength)	Active/ qMON UE
EARFCN	LTE frequency band number	Radio network generic metric	Active/ qMON UE
Transmit Power (Tx Power)	pps	Radio network quality metric (signal strength)	Active/ qMON UE
LTE Channel Bandwidth	MHz	Radio network generic metric	Active/ qMON UE
APN	APN name	Radio network generic metric	Active/ qMON UE
Carrier Aggregation State	on/off	Radio network generic metric	Active/ qMON UE

Implementation requirements

The infrastructure provider/telco should support the following requirements:

- OSS should support provisioning NFV-based network service monitoring through the OSM when the request for network service monitoring comes from MATILDA VAO.
- qMON Collector node treated as part of OSS/telco infrastructure (it must be present if the network service monitoring is supported).
- MATILDA Prometheus pulls the data from qMON Collector node which acts as a standard Prometheus target.
- Communication on port 443 must be allowed between qMON VNF Agent/UE and qMON Collector node.
- Communication to the internet must be allowed from qMON VNF Agent/UE to ensure that they can register to and get configuration from qMON Management Server⁵.

⁵ qMON Management Server is the integral part of the qMON Monitoring Solution. It runs in ININ's cloud and is not part of the MATILDA orchestration.

5.2 Baseline technologies

VNF monitoring

Monitoring of the VNFs can be done in two ways. One would be getting the needed metrics from the host system on which the VNFs will run or even better from the Virtual Infrastructure Manager (i.e. OpenStack in the case of MATILDA). Monitoring is inherently supported in Openstack by several integrated components such as Ceilometer, Gnocchi, Aodh projects. Respectively, the aforementioned components provide means for collecting utilization data of the physical and virtual resources, persistent storage of historical data in multi-tenant time-series database, alarming and action triggering based on pre-defined policies. Alternatively, the Monasca project is a multi-tenant, highly scalable, performant, fault-tolerant monitoring-as-a-service solution, which uses a REST API for high-speed metrics processing and querying and has a streaming alarm engine and notification engine. Integration of such a system and Prometheus platform would then be necessary. Openstack and the integrated monitoring components listed above are being used for VNF monitoring.

Container monitoring

A Linux container is a technology that allows a group of processes to be isolated from the host system that they run on. Containers behave like virtual machines. To the outside world, they can look like their own complete system. However, and unlike a virtual machine, rather than creating a whole virtual operating system, containers replicate only the individual components they need in order to operate. The key difference between containers and VMs is that while the hypervisor abstracts an entire device, containers just abstract the operating system kernel. The underlying technologies that make this functionality possible are mainly namespaces and Cgroups.

Specifically, Docker creates a set of namespaces for each container. These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace. The namespaces used are for process isolation, managing network interfaces, IPC resources, filesystem mount points, kernel and version identifiers.

Docker also makes use of kernel control groups for resource allocation and isolation. A Cgroup limits an application to a specific set of resources. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints. Cgroups used by Docker Engine deal with allocation and management of memory, huge pages, CPU, block I/O, network traffic control, device access.

It becomes evident that there is a wealth of performance and utilization metrics that can be collected from containers. The MATILDA framework uses open source software to integrate Docker container metrics with the Prometheus platform.

Continuous network performance monitoring

As presented, MATILDA monitoring framework can be extended with the qMON agent integrated as a VNF providing network-level KPIs in the same manner as at the application component level. Furthermore, if the radio network KPIs (i.e. RSSP, RSSQ, RSSI, SINR, etc.) are to be monitored, the qMON agent in the role of PNF can be implemented. Gathered data can be used by the communications infrastructure provider for continuous network service and slice monitoring or, if supported, even by the slice orchestrator to get the real-time network KPIs

(e.g. round-trip time, download/upload capacity) of a network segment, network slice or end-to-end network service.

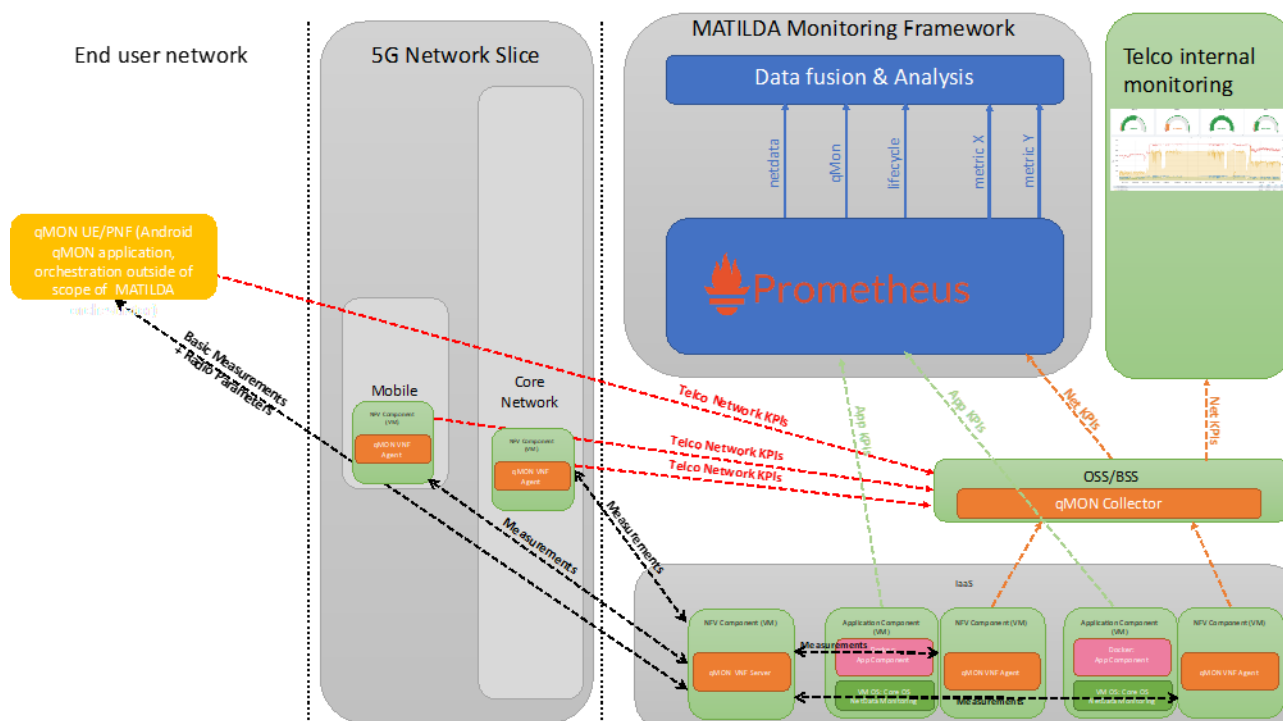


Figure 23: Continuous network performance monitoring

MATILDA Monitoring Solution

The implementation is based on the Prometheus white-box monitoring platform, which is an open-source system monitoring and alerting toolkit. Prometheus uses a time-series database based on LevelDB, an alert management system, to facilitate the evaluation of alert conditions, with a series of system metrics exporters acting as agents for the monitored systems. A key aspect in the data analysis is the Prometheus specific query language, PromQL, which facilitates high performance time-series data aggregation. Prometheus supports both push and pull based data collection, strongly encouraging pull strategy using agents. Agents are being deployed at the monitored subject or at the host it resides in. The agents expose the metrics as APIs, which then can be requested by the Prometheus server at a fixed time period.

6 Data Fusion, Real-time Profiling and Analytics Toolkit

The analytics Toolkit's design and implementation focuses on supporting the following:

- The ease of integration of analysis processes/scripts by data scientists independently of the programming language used.
- The ease of selection of monitoring metrics (resource usage, orchestration, application component specific metrics) and the fetching of the required time-series data from the Monitoring Engine in order to realise analysis over them.
- The production of analysis results in the form of URLs that can be easily viewed and compared by the interested parties (e.g. data scientists, network administrators)
- The design and implementation of a set of APIs for supporting the registration and execution of analysis processes.

The Toolkit contains a set of analysis processes/scripts including:

Correlation Analysis: identify strong correlations, relations and trends among infrastructure-oriented and application component-specific metrics, leading to insights that can be used for runtime policy definition and proactive decision making by the various orchestration mechanisms. Two types of diagrams are produced: a correlogram in the form of a table as well as a Chord diagram providing the most significant correlations per metric.

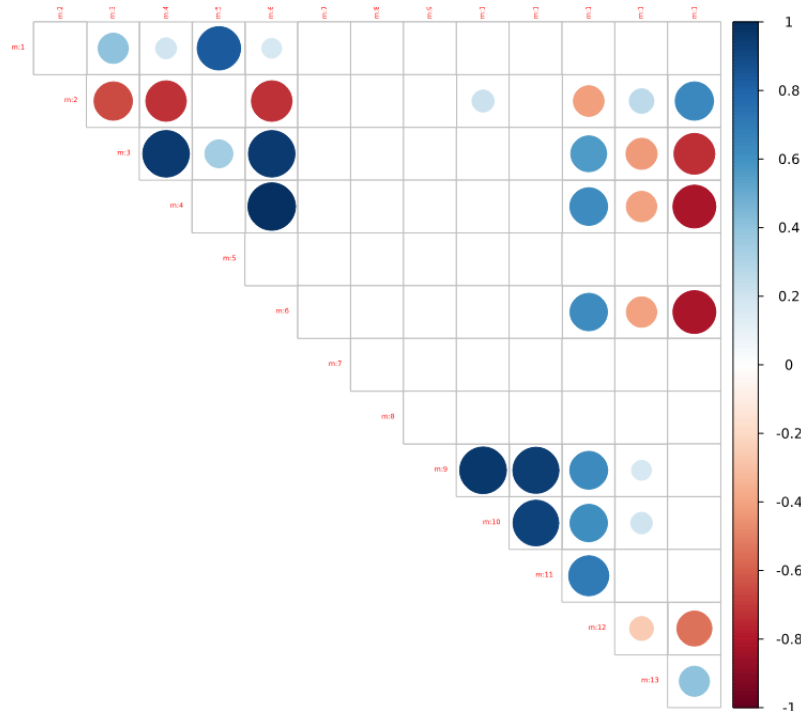


Figure 24: Indicative screenshot (correlogram) from a correlation analysis

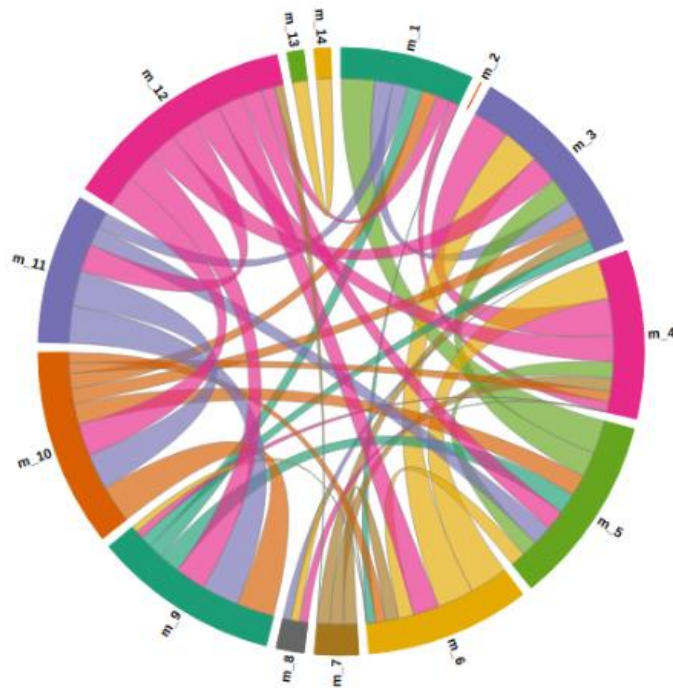


Figure 25: Indicative screenshot (chord diagram) from a correlation analysis

Time Series Decomposition and Forecasting: identify trends and provide accurate forecasting models, forecast resource demanding periods and scale proactively the deployed functions to optimally serve the workload.

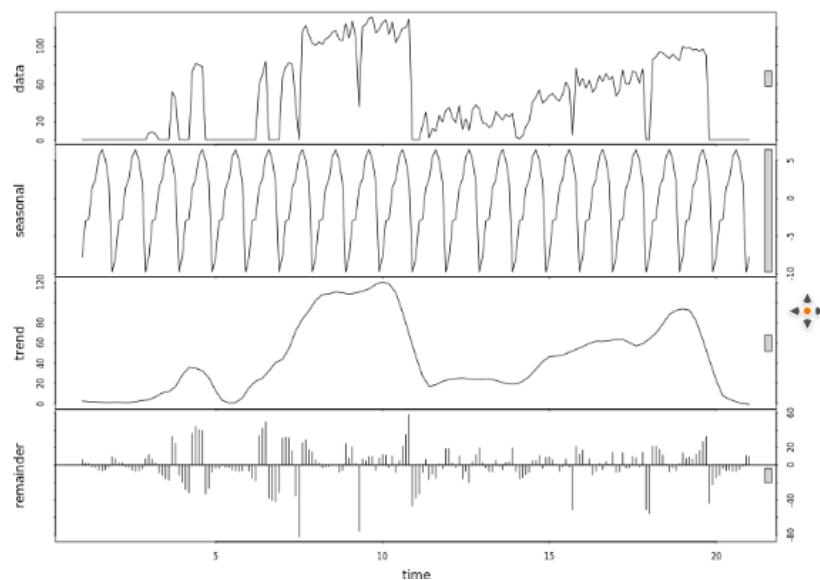


Figure 26: Indicative screenshot from a time series decomposition analysis

Resource Efficiency Analysis:

identify resource consumption trends and capacity limits, used for planning accordingly optimal reservation of resources.

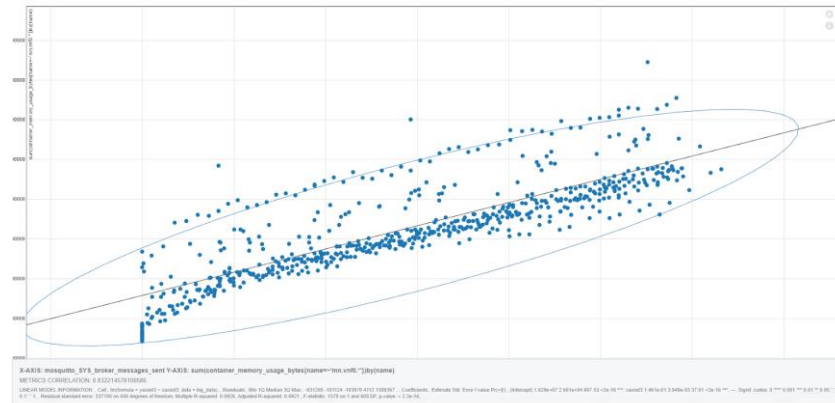


Figure 27: Indicative screenshot from a linear regression analysis

Elasticity Efficiency Analysis:

identify the performance of scaling operations, along with the impact of scaling actions in the service output efficiency. Elasticity efficiency may be expressed as a pair of discrete metrics (Application Capacity Change as output and Capacity Change Lead Time as input). Application Capacity Change is the incremental capacity change related to a scaling action. Capacity Change Lead Time is the time required for a capacity change. Both metrics are going to be depicted in relevant visualisations.

Time-series data for orchestration metrics are collected, including data for the triggering and realization of elasticity actions. Such data lead to the extraction of elasticity efficiency profiles. Future work includes the potential for providing automated graphs, such as the one shown in the following figure.

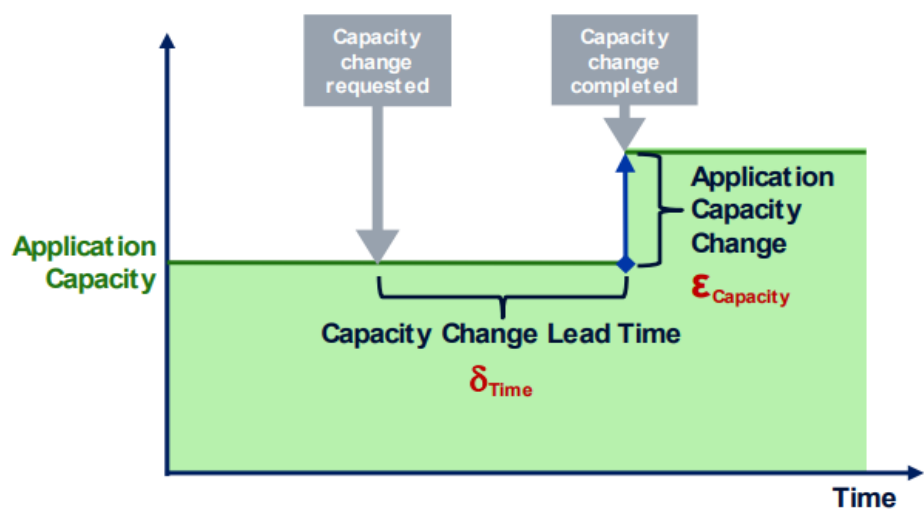


Figure 7-3 Dynamic Operational Efficiency of Cloud-Based Applications

Figure 28: Indicative graph for an elasticity efficiency analysis [NFVEfficiency]

Clustering:

identify clusters based on time series data from multiple metrics, leading to identification of groups of metrics with similar behaviour. Upon the clustering analysis, identification of the boundaries of elasticity rules' triggering based on the component operation is also realised.

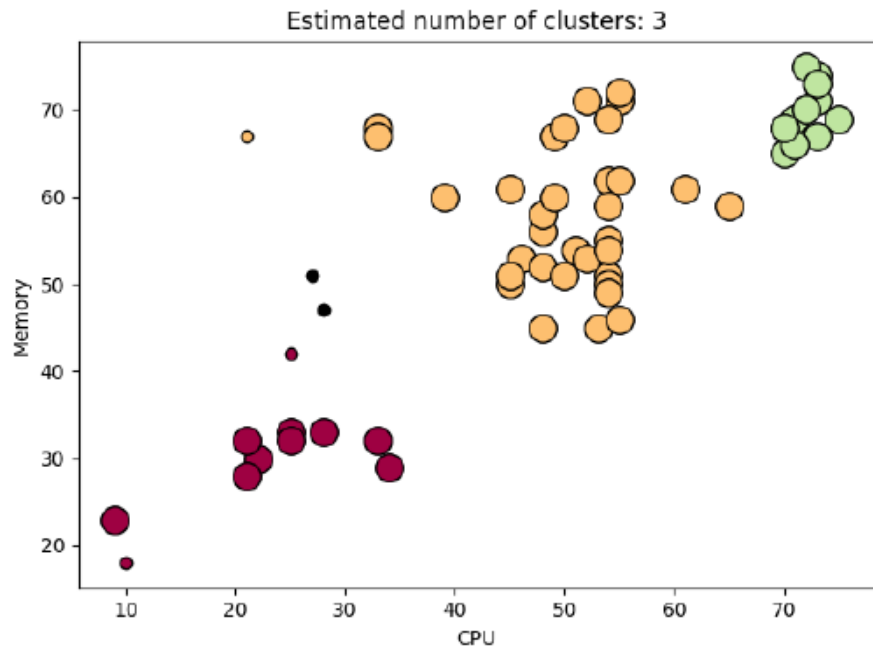


Figure 29: Indicative screenshot from a clustering analysis

Filter healthy metrics:

check the quality of the collected time series data and provide indication about the percentage of the qualitative time-series data (e.g., no many empty values)

The outcome of this analysis is a short text description with the percentage of the monitoring metrics that provide qualitative time-series data.

The overall architectural approach of the analytics toolkit is depicted in Figure 30. Access to the supported algorithms (analysis scripts) is provided through APIs provided by a developed Proxy. The Proxy is based on the OpenCPU framework in case of R analysis scripts or the Flask framework in case of Python analysis scripts. Following, analysis templates can be designed and introduced in the Orchestrator Dashboard for supporting specific analysis processes. Based on the templates, analysis processes can be initiated, where configuration parameters and start and end time for the analysis data are provided. The related time series data is fetched by the monitoring engine and led as input in the analysis process. The analysis is then executed and the analysis results are made available in the form of reports in the dashboard. It should be also noted that interconnection with workload generators is supported over the deployed application graphs, enabling the stress testing of the deployed graphs and the collection of valuable monitoring data for the analysis processes.

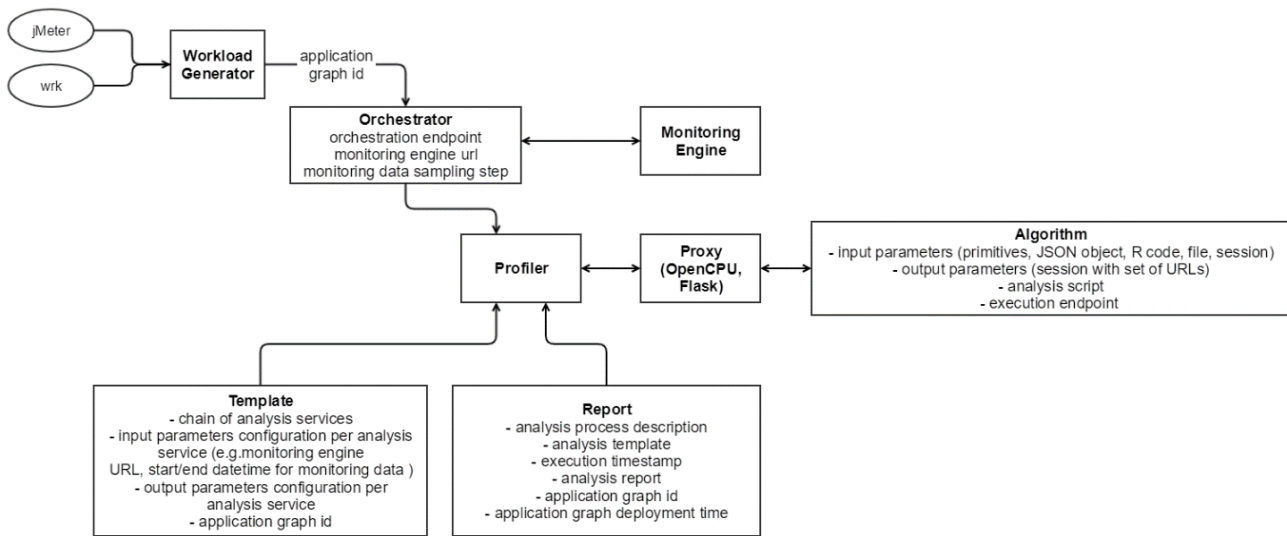


Figure 30: Analytics toolkit architectural approach

6.1 Main components

6.1.1 Data Fusion

The success of a Vertical Slice deployment of a service mesh application graph very much depends on the Quality of Service (QoS) it offers to the end user with regard to the aspects of end to end delay and general user experience. Furthermore, the monitoring, descriptive and predictive analysis of the functional and non-functional aspects of the Application (for instance, non-buggy business logic and resource load allocation) are useful to the Service (Application) Provider and to the Telco (Infrastructure) provider together. The direct benefits may span from the reduction of Capital and Operational expenses to the development of further profitable functionalities tailored to the user needs.

Publish – Subscribe Data Streaming Paradigm

We introduce the Publish – Subscribe pattern (shortly, pub-sub) here for message-based data systems. The general data flow, as illustrated in Figure 32, starts with a producer publishing a message to a broker. The messages are often sent to a topic, which can be thought of as a logical grouping for messages. Next, the message is sent to all the consumers subscribing to that topic. It may not be obvious initially, but often a producer publishing a message doesn't mean that it needs to subscribe to a topic. Nor is it required that a subscriber produce a message. Apache Kafka [Kafka] is the underlying technology for such a streaming system and is being used for the Data Fusion module.

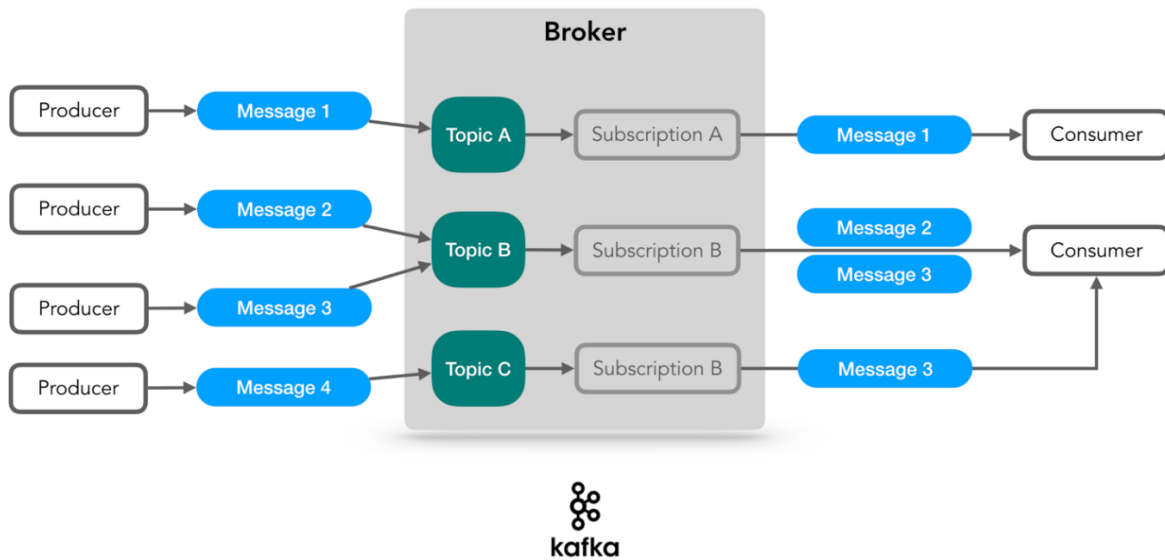


Figure 31: The Publish Subscribe Paradigm for Streaming Data

To support and enhance this kind of advanced insights, the Service Provider and the Telco provider must tap on to as many data streams as possible. Three types of raw and composite KPIs streams are identified in the context of a service mesh positioned on top of a 5G network infrastructure:

- Application (business logic) data streamed by all application components that comprise the entire Service. In the MATILDA framework such data can be relayed to the Orchestrator plane. For instance, how many users are connected to each application microservice or the duration of their visits can be either stored on a specialized persistence microservice (e.g. an SQL/NoSQL Database) or, rather, published to a publish/ subscribe data stream (i.e. a Kafka topic).
- Microservices data that contain resource usage or relevant data. Examples of such information are: the number of concurrent queries to a database, the CPU load or memory usage on a specific container, I/O read/write statistics, etc. Again, this information can be directly reported through a subscription to a topic with the “heartbeat” of each application container or, alternatively, through a monitoring mechanism (such as Prometheus) which republishes the same data or aggregates of them.
- Compute and network resource usage data than can be also published directly by the underlying OpenStack infrastructure’s Kafka reporting bus, thus reducing data traffic within the MATILDA deployed infrastructure.

Furthermore, topology (application and network graph) metadata are available, which enables the deployment of MATILDA application graphs based on the optimization step of selecting the most appropriate resources. Furthermore, any alterations made to the graph are transferred to a newer version of the same graph. Examples of this include scaling components through commissioning or decommissioning resources by the Telco provider.

The Data Fusion component is responsible for the delivery of accurate, complete, and dependable information from the data streams discussed above. The fusion of data does not

necessarily involve some kind of summarizing aggregations of, for instance, the average resource usage, but can also allow for higher data quality of all data sources by interpolation, imputation or reasoning over data. A beneficial by-product of this data fusion process can be that the network traffic generated by monitoring streams can be intelligently reduced and thus analytics can be performed on smaller sized data with obvious speed and cost implications.

For the Data Fusion mechanism to exist pub-sub brokers are employed to monitor topics.

Such topics are scanned and filtered for useful information which is in turn channelled via Kafka producers to software modules that perform –among others- fuzzy and deep learning-powered data fusion and from there persisted to their appropriate storage. Fused data are available to other MATILDA components for further real-time or batch analysis: The Analytics Engine & Profiling Engine, the Monitoring and Visualization mechanism, the MATILDA Orchestrator’s rule-based reasoning engine and the Application Logging mechanisms.

These functionalities are architecturally described in Figure 32.

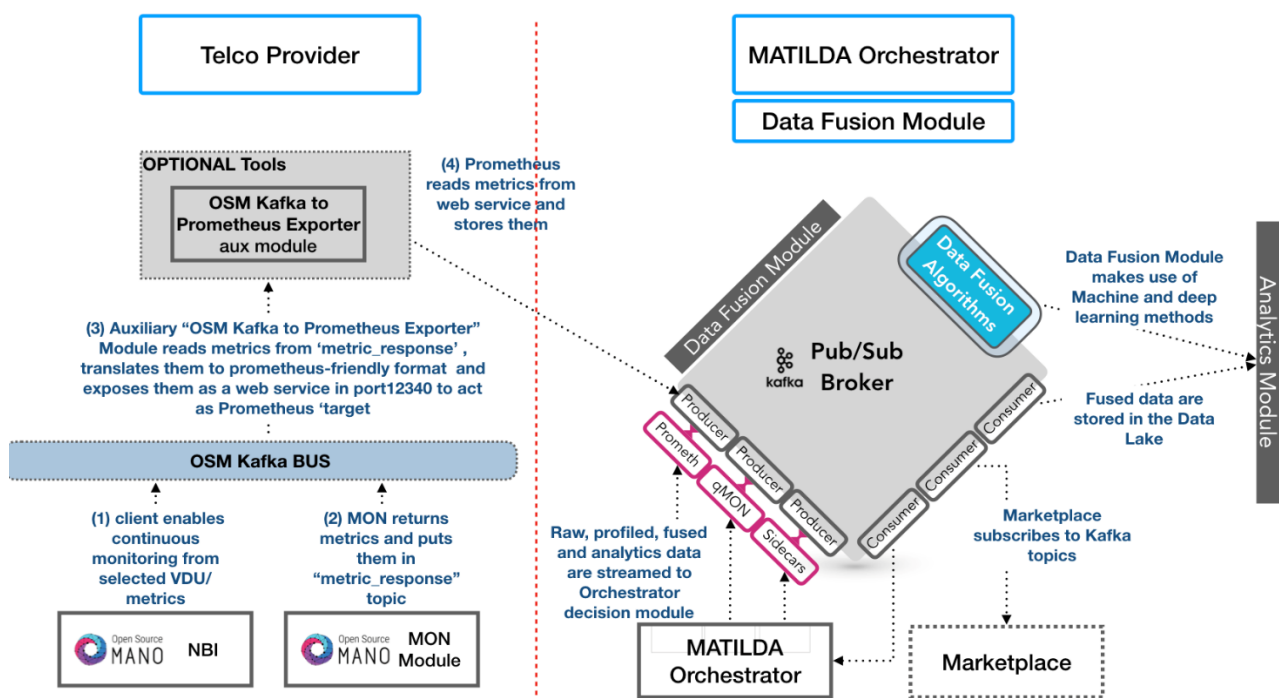


Figure 32: Data Fusion Module and functionalities on multiple Data Streams

The set of functionalities envisioned above within the Data Fusion module must be translated into software components. The main components that comprise the Data Fusion module by the MATILDA partners are listed below (in Table 5). These include: the pub-sub software components that are tasked with handling streaming data (producers, consumers, brokers), adapters to other software modules of the MATILDA Orchestrator architecture (Prometheus monitoring, Orchestrator Engine, Marketplace, Analytics Engine) and the Data Fusion algorithms software component.

Table 5: List of Data Fusion module SW components

SW Component Identifier	Description	Partners
Data Fusion	Implementation of Data Fusion algorithms on multiple data streams	INC
OSM Kafka Bus adapter	Adapter/Producer on OSM NBI/MON for the Data Fusion Module	ATOS, INC
Producer for qMON	Kafka Producer for qMON metrics collection/monitoring service	INC
Producer/Adapter Prometheus	Kafka Producer from Prometheus to Data Fusion and Analytics Module	INC, UPRC
Producer from ServiceMesh sidecar	Kafka Producer from the service mesh sidecar (e.g. Envoy, or Linkerd) to Data Fusion and Analytics Module	INC
Generic Fusion Consumer	Kafka Generic Consumer for other MATILDA Orchestrator Modules (e.g. Policy Manager including validation aspects, Profiling)	INC, UBI
Internal Analytics Consumer	Internal Kafka Consumer for the Analytics Module	INC

6.1.2 Real-time profiling and Analytics module

A set of profiling mechanisms are designed and partially implemented, focusing on supporting various profiling aspects in application component and application graph level. Such mechanisms may be applied in benchmarking scenarios, as well as over data collected in an operational environment. Profiling concerns the examination of the behaviour of the application characteristics (e.g., resources usage efficiency, elasticity profiles, graph-based dependencies, potential bottleneck points) under various conditions, leading to its characterization with regards to a set of performance aspects.

In case of benchmarking, a set of scenarios are executed, each one of which considers a specific input dataset and a set of rules towards obtaining the measurement results. Benchmark definitions often refer to the concept of a System Under Test (SUT) that is a collection of components necessary to run the benchmark scenario. The idea of a SUT is to define a complete application architecture containing one or more components of interest, which in our case are the components of the MATILDA architecture including the vertical application orchestration mechanisms and the 5G programmable infrastructure management mechanisms. A set of tools are provided, creating and feeding the SUT with the required workload for the realisation of the test. Such tools include -among others- MoonGen [MoonGen], a fully scriptable high-speed packet generator, wrk [wrk] HTTP benchmarking tool.

In case of operational environments, deployment of 5G-ready applications is realised over the provided infrastructure leading to the activation of a series of monitoring mechanisms

and the collection of application and infrastructure-oriented metrics. The collected data is made available in a time series format through Prometheus. Following, analysis may take place based on analysis scripts based mainly on the R programming language.

The set of testing activities includes load testing (understand the behaviour of the application under a specific expected load), stress testing (testing beyond normal operational capacity, often to a breaking point, in order to observe the results), portability testing (examine the deployment and operation of the applications across various platforms and operating systems), reliability testing (exercising an application so that failures are discovered and removed before the system is deployed). The outcome of these series of tests is going to lead to profiling reports (Figure 33) with regard to resources usage (CPU, Memory, Storage intensiveness) and elasticity efficiency of the software (time and capacity for supporting scaling operations), identification of capacity limits and breaking points upon stressing the software processes, reliability reports taking into account time series data with identified problems and failures, insights with regard to set of deployment platforms supported, as well as reports regarding behavioural aspects of the software in terms of capacity for recovery to failures and responsiveness and efficiency of the developed self-configurability mechanisms.

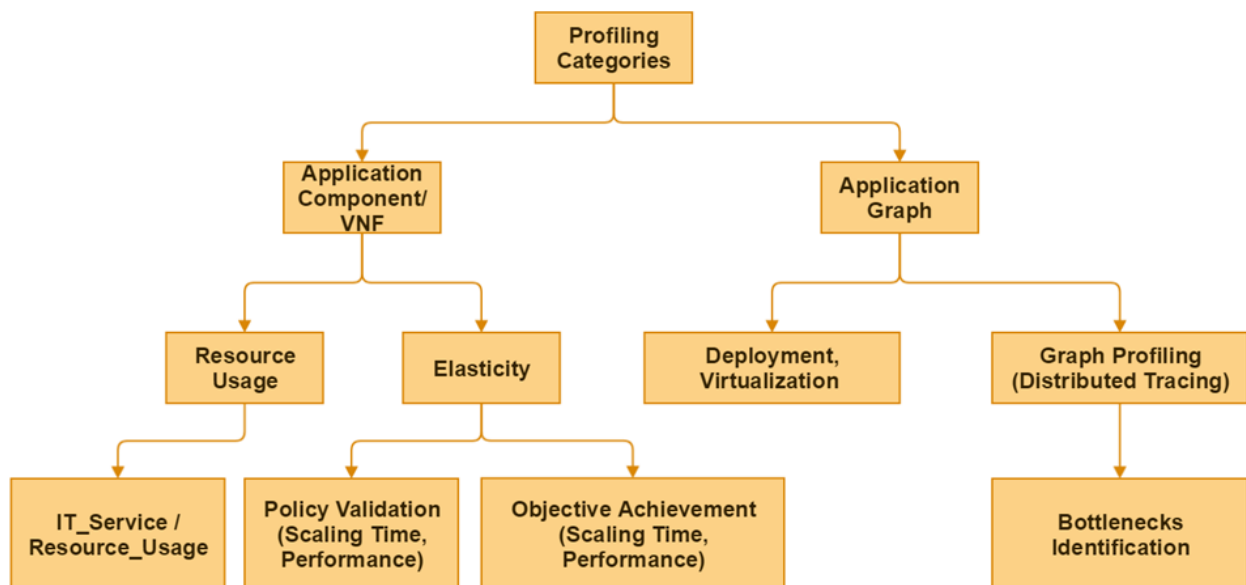


Figure 33: Profiling Categorization

6.2 Interfaces

A set of interfaces have been specified and implemented for supporting the set of data fusion, real time analysis and profiling mechanisms, including the following:

- Interface for submitting queries to Prometheus monitoring engine: used by the various services that require access to monitoring data.
- Interface for registering monitoring alerts to Prometheus monitoring engine: used for registering expressions, as well as used by the various data fusion, analysis and profiling mechanisms.

- Interface for registering an analysis script that can be used for realising an analysis, upon proper configuration.
- Interface for executing an analysis process: used for triggering the execution of an analysis script and fetching back the results.
- Interface for providing a workload to an application graph instance: used for profiling purposes.

6.3 *Baseline technologies*

The baseline technologies used for the development of the data fusion, real-time profiling and analytics toolkit include:

- OpenCPU framework for scientific computing
- Flask framework
- Apache Spark for Big Data Processing
- Apache SparkML for Machine Learning tasks
- Apache Kafka for streaming data pipeline
- Apache HDFS, Parquet for Distributed Persistence (Data Lake)
- Tensorflow, for Deep Learning tasks
- Incelligent Analytics Platform
- R statistics package
- Prometheus for monitoring
- Moongen, wrk for workload preparation

7 Policies Enforcement and CEP Mechanisms

The policies enforcement mechanisms include a set of modifications in the policy manager for improving performance and scalability aspects. Specifically, in order to overcome potential computational problems, a distributed implementation of the policy manager has been realised, in addition to the monolithic approach and thus horizontal scalability of the policy manager is supported, as shown in Figure 34.

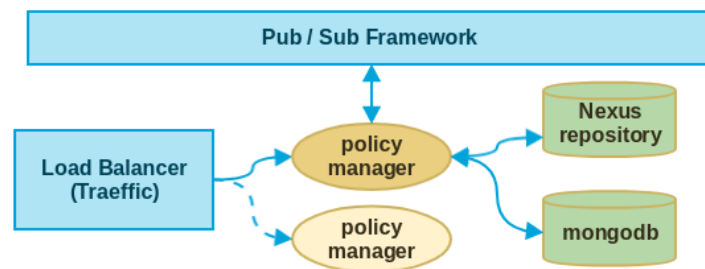


Figure 34: Horizontally Scalable Policy Manager Design

The consistent hashing technique is being used so as to obtain the optimal creation of the application-based policies and the optimal consumption of the monitoring messages delivered via the pub/sub framework (Figure 35). In more detail, thanks to the use of the consistent hashing technique, the messages targeted to the same application graph are always routed at the same queue, leading to the minimum set of exchangeable messages between the broker and the policy manager. This results in the relief of any possible computational problems, in case of a large number of operational policies that consume constantly messages from the pub/sub framework and generate elasticity actions.

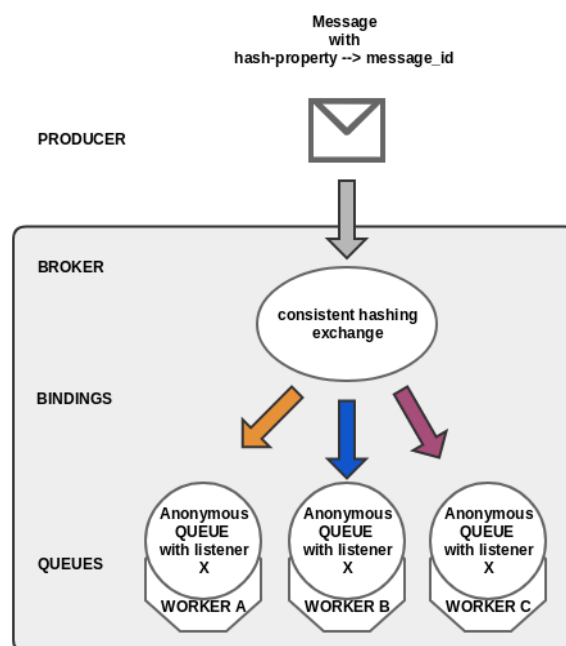


Figure 35: Consistent Hashing Technique for Scalable Policy Manager

In order to further optimize the policy manager in terms of resiliency, a remote maven repository (NEXUS) has been introduced in order to support the quick enforcement of all necessary policies in case of an independent failure of a policy manager worker (see Figure 36). In case of a worker failure, policy rules are simply downloaded as package from the Nexus server without any recreation via the descriptors. This boosts the time of recovery in case of failure that is so important in cases where immediate actions have to be taken. Also, thanks to the Nexus server, the worker that updates the policy does not have to be the same as the one that enforces the policy, which simplifies the overall solution adaptation. The use of Nexus also leads to smaller size of workers, since it hosts all the drools-based policies and makes them available on demand at the workers. Last but not least, the distributed approach permits the update of rules on the fly. The policies can be updated at operation time without affecting the working memory. This leads to quick updates of the new policy rules and zero loss of the monitoring metrics information that has been gathered.

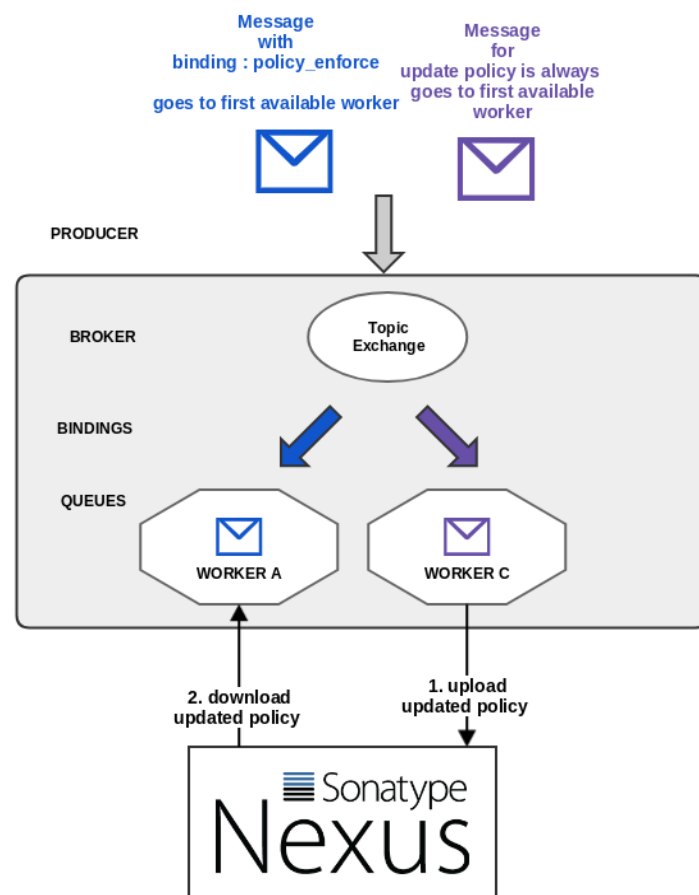


Figure 36: Nexus Repository for hosting Policy Descriptors

7.1 Main components

7.1.1 Policies Enforcement mechanism

The Policies Manager in MATILDA provides policies enforcement over the deployed application graphs following a continuous match-resolve-act approach. Specifically, the match phase regards the mapping of the set of applied rules that are satisfied based on the alerts coming from the monitoring infrastructure. The resolve phase regards the process of conflict resolution for different rules that may be valid and triggered at the same time. Thus, the resolve phase aims at resolution among these rules taking into account the pre-defined salience of each rule. The act phase regards the provision of a set of suggested actions by the policy manager to the orchestration components, the Deployment Manager and the Execution Manager of the MATILDA orchestrator, responsible for application graphs placement and management, respectively. Policies enforcement is realized through a rule-based framework that attempts to derive execution instructions based on the current set of data and the active rules; rules associated with the deployed application graphs at each point of time. Specifically, we have adopted Drools rules-based management system [Drools], an open-source solution that supports the implementation of runtime policies enforcement mechanisms.

Specifically, the Policy manager (following a Drools approach) consists of (i) the working memory (WM); facts based on the provided data, (ii) the production memory (PM); set of defined rules, and (iii) an inference engine (IE) that supports reasoning and conflict resolution over the provided set of facts and rules, as well as triggering of the appropriate actions. Data is fed to the WM through the monitoring mechanisms that is responsible to collect data based on a set of active monitoring probes. The PM is also fed by policies associated with the deployed application graphs, as provided through the Policies Editor - the editor made available to service providers for policies definition.

Data monitoring and management processes are supported through a set of passive monitoring probes by the Prometheus monitoring engine. Collection and consumption of information is based on the configuration of a Publish/Subscribe framework -namely the Kafka framework-, where set of components, resource usage and application graph metrics are provided based on application graph-oriented topics. Policy manager dynamically handles and converts the collected data to WM facts. Such facts can then be matched with already defined rules on the active policies. Definition of rules per policy is supported through the Policy Editor in a per application graph basis, based on the concepts represented in the MATILDA metamodel. An application graph may be associated with a set of policies; however, only one can be active during its deployment and execution time. Each policy consists of a set of rules. Each rule consists of the conditions part - denoting a set of conditions to be met- and the actions part -denoting actions upon the fulfilment of the conditions. The defined policies are translated to a set of rules that become part of the Policy Manager. Expressions may regard custom metrics of an application graph or a component/microservice. Detailed description of the policies metamodel and the supported types of conditions and actions is provided at D1.5 “Deployment and Runtime Policy Metamodel” [MATILDA-D1.5].

Each rule has attached a specific salience that is used as a priority indicator during conflict resolution by the IE. A time window can be optionally specified per rule for the validation of the successful enforcement of the proposed actions. When attaching a specific runtime policy

to an instantiated application graph, the specified set of policy rules are deployed to the policy manager PM, while the WM agent is constantly feeding the WM with new facts.

The high-level interaction between the Policy Manager and the Monitoring mechanisms is depicted in Figure 37. Upon the instantiation of a 5G-ready application graph and the enforcement of a policy, a set of monitoring metrics are collected and processed according to a set of expressions defined in the policies. The metrics and the corresponding expressions may regard application-component-specific metrics, application-graph-specific metrics or resources usage metrics. Processing of the collected data is realised within the Monitoring mechanisms (Prometheus), leading to the triggering of alerts that leads to the publishing to the Message Broker (Kafka) at a monitoring topic. Such alerts are consumed by the Policy Manager (Drools based implementation) for realising inference over the defined set of rules. The suggested actions from the inference results are then published to the Message Broker at a relevant topic in order to be consumed by orchestration mechanisms.

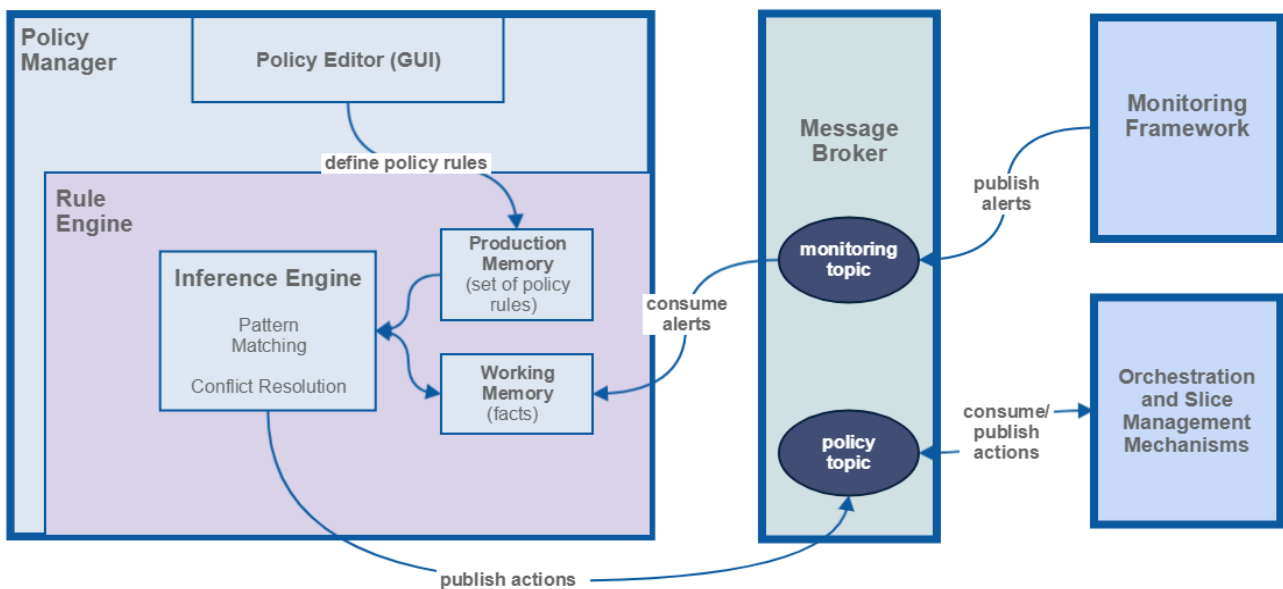


Figure 37: Policy Manager and Monitoring Mechanisms Interaction

7.1.2 Complex Event Processing mechanism

The Complex Event Processing mechanism of the MATILDA Intelligent Orchestrator suggests a dynamic engine that, enriched with Machine Learning techniques can be fully adaptive to its environment [Symvoulidis-2019]. The high-level architecture was described in the deliverable regarding the first release [MATILDA-D3.1]. In this deliverable a detailed description of the mechanism will be given. As presented in Figure 38, the engine is divided in two major components, namely the *Complex Event Processing engine* and the *Threshold identifier*, which are going to be described in more detail below.

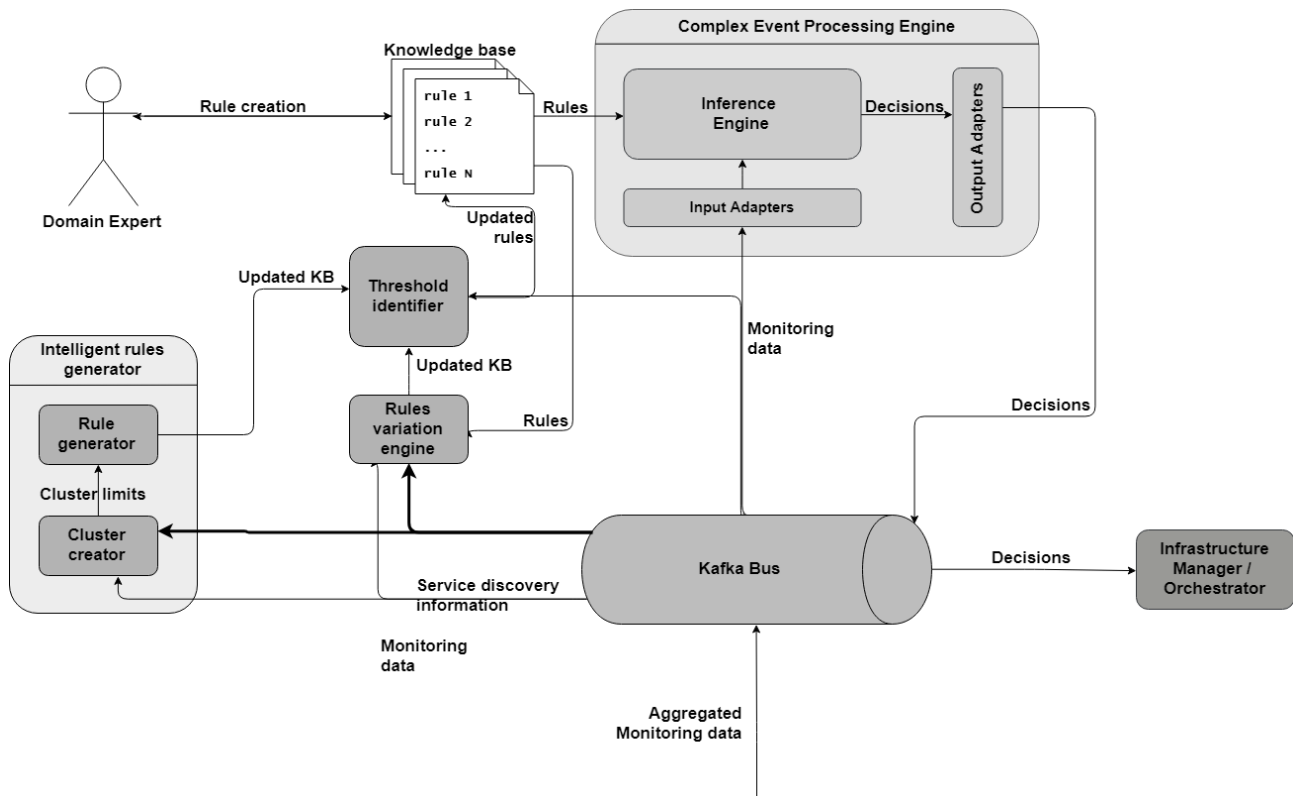


Figure 38: Dynamic Complex Event Processing mechanism – Overall architecture

The Complex Event Processing engine regards a Drools Fusion engine, whose job is to trigger the rules from the Knowledge Base when a condition is met. These rules are created in the first place by a Domain expert or a Service provider.

The Complex Event Processing engine, as is, regards a typical engine without any level of intelligence. For this reason, the Threshold identifier service is developed, whose purpose is to identify in real-time the behaviour of the deployed services and update the Knowledge Base (i.e. the rules) in order for the CEP engine to act accordingly. Under the hood, the Threshold identifier regards an Incremental Learning algorithm specifically developed for the purposes of the MATILDA project. In more detail, an Incremental DBSCAN reference implementation is developed that identifies which is the normal behaviour of the service based on the monitoring metrics.

The process of the identification of the behaviour of the deployed services starts with the collection of a dataset of historical monitoring data that are fed to a batch DBSCAN algorithm in order to create the first clusters that represent the usage of the service.

At runtime the monitoring data are provided to the Incremental DBSCAN algorithm that, based on the outcomes of the batch algorithm, decides whether it should be added in an already existing cluster, considered as an outlier, or to create a new cluster with other outliers. In any case the outcomes are taken under consideration the next time a new data is inserted, something that we could not achieve using only the batch implementation of the DBSCAN.

As already mentioned, the clusters represent the behaviour of the deployed service. We use that information to make the CEP engine more adaptable to its environment, using a simple, yet effective methodology. The cluster with the most elements constituting it represents the normal behaviour of the service. We take the limits of the cluster and based on them we adjust the rules in the Knowledge Base. This leads to a more context aware solution that can identify the changes of its environment faster and adapt without the need of external assistance.

The use of Incremental DBSCAN over the batch DBSCAN approach is preferred in order to avoid the time-consuming process of re-training of the algorithm. In addition, using the incremental approach of DBSCAN the newly incoming data is taken under consideration instantly and the decisions made are up-to-date.

7.2 Baseline technologies

This chapter describes the baseline technologies used for the implementation of the MATILDA Complex Event Processing and Policies Enforcement engines. As far as the Complex Event Processing mechanism is concerned, Drools Fusion [Drools] is used as the CEP solution. Drools Fusion is an open-source tool developed by JBoss. More regarding Drools Fusion on the official documentation [Drools-Fusion]. The language used for developing the Complex Event Processing engine is Java [Java].

Drools Fusion is the module created by Redhat as an extension to Drools, responsible for adding Complex Event Processing capabilities into the platform. Drools is a business rule management system that allows fast and reliable evaluation of business rules. Drools supports both forward- and backward-chaining. Fusion is a module among others (OptaPlanner, WorkBench, etc.) that comprise the overall Drools platform, making it a complete Business Rule Management framework. Drools has been written and is available in Java.

Drools Fusion has many features that make it a great solution for problems that need to be solved through Complex Event Processing techniques. Events in Drools Fusion are distinguished as a record of a state change and have a few distinguishing characteristics; the events are **immutable** meaning that they cannot be changed after they are detected and there are strong **temporal constraints** correlating the events.

Sliding windows are also supported in Drools Fusion, using the timestamp that is linked to each event for the creation the windows. There are two ways a sliding window can be created. The first regards the match of events that occur in the last X time units, called *Sliding Time Window*, and the second matches the events based on the times they occurred, called *Sliding Length Window*.

Drools Fusion handles events as a special type of facts. Facts are an instance of an application object represented as a Java Object. These facts are declared as events giving them the aforementioned characteristics. After the declaration of events is complete, a set of rules are created that are match with the events. These rules are in the form of “*WHEN condition THEN action*”.

8 Northbound APIs for Communication Service Providers

From a Vertical Orchestrator point of view, a MATILDA-enabled 5G provider is a combined Telecommunications and Infrastructure Service provider that is able to process Slice Intents and generates network Slices that are required in order for an application graph instance to be operational.

The characteristics of this layer are “imposed” by the final OSS layer which is analysed in Deliverable D4.2. For the sake of completeness, it should be mentioned that the VAO has the ability to define (or restrict) the edge locations that the vertical application is expected to scale. **Such restriction constitutes another distinct slice-request trait** that is included in the implementation.

8.1 Main components

Northbound APIs are provided by an Operations Support System (OSS). OSS is a web application, which offers a user interface along with a RESTful API that is used by the Orchestrator. The list of relevant components is cited in the following table:

Table 6: List of Northbound APIs components

SW Component Identifier	Description	Partners
Operations Support System	Implementation Northbound APIs	INTRA, UBI

The overall architecture of the Northbound APIs is depicted in Figure 39:

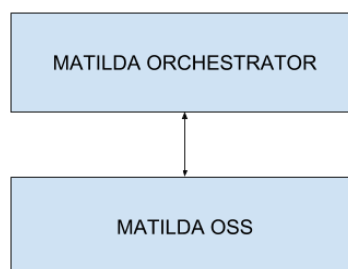


Figure 39: MATILDA Northbound APIs architecture

Regarding the MATILDA OSS, the following figure (Figure 40) cites the interface of the OSS depicting a graph and the corresponding components and their information (e.g., identifiers).

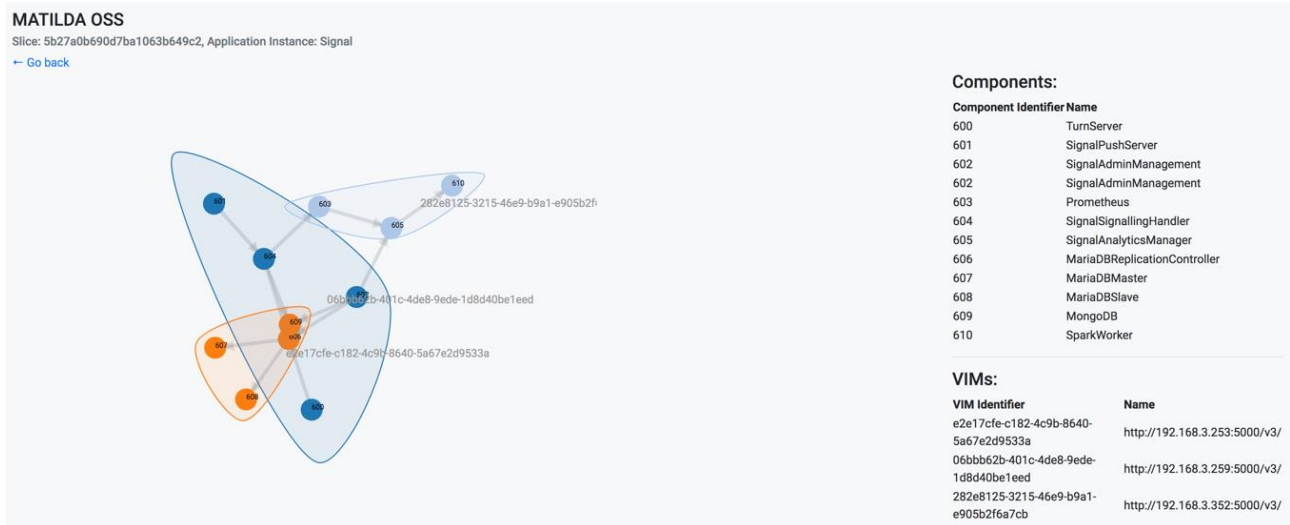


Figure 40: MATILDA OSS user interface

8.2 Interfaces

Two interfaces have been specified and partially implemented for supporting the Northbound APIs:

- Interface for accepting a Slice Intent from the Orchestrator. Asks the telco provider to materialize a Slice given a specific Application Graph Instance and a set of constraints. The response pattern of this interface is asynchronous.
- Interface for informing the Orchestrator if a Slice can be materialized or not in order to start the deployment of the specific Application Graph Instance. The response pattern of this interface is synchronous.

Indicative serialized argument and an expected response are presented in Table 7 and Table 8 respectively.

Table 7: Indicative Slice Intent serialized in JSON Format

```
{
  "applicationInstanceID": "580",
  "name": "OSSScenario",
  "callbackURL": "http://localhost:8080/api/v1/callback/slice/580",
  "authenticationDetails": {
    "clientToken": "!telcoprovider!",
    "clientKey": "telcoprovider"
  },
  "componentNodeInstances": [
    {
      "componentNodeInstanceID": "581",
      "componentNodeInstanceName": "TestCaseMariaDB"
    },
    {
      "componentNodeInstanceID": "587",
      "componentNodeInstanceName": "TestCasePhpMyAdmin"
    }
  ],
  "constraints": [
    {
      "constraintID": "591",
      "interfaceInstanceID": "590",
      "qi": "10",
      "radioServiceType": "1",

```

```
    "resourceType": "DELAY_CRITICAL_GBR",
    "allocationRetentionPriorityProfile": 1,
    "minimumGuaranteedBandwidth": 120.0,
    "maximumRequiredBandwidth": 200.0,
    "constraintUnit": "kbps",
    "category": "ACCESS",
    "type": "HARD"
  }, {
    "constraintID": "592",
    "graphLinkNodeID": "544",
    "constraintMetric": "DELAY",
    "constraintUnit": "ms",
    "constraintValue": "100.0",
    "category": "GRAPH_LINK",
    "type": "HARD"
  }, {
    "constraintID": "593",
    "componentNodeInstanceID": "587",
    "constraintMetric": "MIN_V_CPU",
    "constraintUnit": "amount",
    "constraintValue": "4.0",
    "category": "COMPONENT_HOSTING",
    "type": "HARD"
  }, {
    "constraintID": "594",
    "componentNodeInstanceID": "587",
    "constraintMetric": "MIN_RAM",
    "constraintUnit": "gb",
    "constraintValue": "16.0",
    "category": "COMPONENT_HOSTING",
    "type": "HARD"
  }, {
    "constraintID": "595",
    "componentNodeInstanceID": "587",
    "constraintMetric": "MIN_STORAGE",
    "constraintUnit": "gb",
    "constraintValue": "10.0",
    "category": "COMPONENT_HOSTING",
    "type": "HARD"
  }, {
    "constraintID": "596",
    "componentNodeInstanceID": "581",
    "constraintMetric": "MIN_V_CPU",
    "constraintUnit": "amount",
    "constraintValue": "4.0",
    "category": "COMPONENT_HOSTING",
    "type": "HARD"
  }, {
    "constraintID": "597",
    "componentNodeInstanceID": "581",
    "constraintMetric": "MIN_RAM",
    "constraintUnit": "gb",
    "constraintValue": "10.0",
    "category": "COMPONENT_HOSTING",
    "type": "HARD"
  }, {
    "constraintID": "598",
    "componentNodeInstanceID": "581",
    "constraintMetric": "MIN_STORAGE",
    "constraintUnit": "gb",
    "constraintValue": "16.0",
    "category": "COMPONENT_HOSTING",
```

```

    "type": "HARD"
  }],
  "graphLinkNodes": [{
    "graphLinkNodeID": "544",
    "fromComponentNodeInstanceID": "587",
    "toComponentNodeInstanceID": "581",
    "type": "CORE"
  }],
  "dateCreated": "Jun 13, 2018 12:51:38 PM"
}

```

Table 8: Indicative Slice serialized in JSON Format

```

{
  "applicationInstanceID": "580",
  "vimDescriptors": [{
    "vimID": "a4ab0bf9-188f-40da-8624-2f4a879f2257",
    "domain": "default",
    "project": "maestro",
    "username": "maestro",
    "password": "!maestro!",
    "endpoint": "http://192.168.3.253:5000/v3/"
  }],
  "componentPlacements": [{
    "vimID": "a4ab0bf9-188f-40da-8624-2f4a879f2257",
    "componentNodeInstanceID": "581",
    "attachmentPoints": [{
      "graphLinkNodeID": "544",
      "attachmentPointIdentifier": "6763cfb6-d7ab-43d1-bfac-c997b4685ad2"
    }]
  }, {
    "vimID": "a4ab0bf9-188f-40da-8624-2f4a879f2257",
    "componentNodeInstanceID": "587",
    "attachmentPoints": [{
      "graphLinkNodeID": "544",
      "attachmentPointIdentifier": "b66b6a90-c550-413b-b484-961ad339b2bd"
    }]
  }],
  "constraintSatisfactions": [{
    "constraintID": "591",
    "satisfied": true,
    "constraintType": "HARD"
  }, {
    "constraintID": "592",
    "satisfied": true,
    "constraintType": "HARD"
  }, {
    "constraintID": "593",
    "satisfied": true,
    "constraintType": "HARD"
  }, {
    "constraintID": "594",
    "satisfied": true,
    "constraintType": "HARD"
  }, {
    "constraintID": "595",
    "satisfied": true,
    "constraintType": "HARD"
  }, {
    "constraintID": "596",

```

```
        "satisfied": true,  
        "constraintType": "HARD"  
    }, {  
        "constraintID": "597",  
        "satisfied": true,  
        "constraintType": "HARD"  
    }, {  
        "constraintID": "598",  
        "satisfied": true,  
        "constraintType": "HARD"  
    }],  
    "dateCreated": "Jun 13, 2018 12:51:49 PM"  
}
```

8.3 *Baseline technologies*

As far as the Northbound APIs development is concerned, Spring Boot [Spring-Boot] has been used. Spring Boot made it easy to develop the Northbound APIs, as it provides a range of non-functional features such as embedded servers, security, externalized configuration, etc. On the other side, Thymeleaf [Thymeleaf] template engine along with Bootstrap [Bootstrap] are used for the user interface.

9 Conclusions

This document presents the design and implementation of the final release of the intelligent orchestration mechanisms of the vertical application orchestration, as they are developed within WP3. The set of mechanisms are provided to WP5 and have been integrated in the final release of the MATILDA integration framework. Integration has been mainly realised with the set of WP4 components for tackling the network slice creation and management per application graph, while the overall status of application graphs composition and deployment is made available in the provided Dashboard in WP2.

The monitoring solution of the MATILDA project, as described above, is responsible for the management of the metrics captured from the various infrastructure components, the management of the alerts and events, based on these metrics, and the visualization of the available data. The Data Fusion, Real-time Profiling and Analytics Toolkit is responsible for the extraction of accurate and reliable information from the monitoring data streams and the prediction of meaningful insights to the Service Provider and Communication Service Provider. The service discovery mechanisms will handle tasks such as dynamic service discovery, load balancing, publication of metrics, etc. In the meantime, the policies enforcement and CEP mechanisms facilitate on the decision-making of the execution manager, providing insights during runtime of application graphs, solving potential conflicts and suggesting actions to the execution manager. Additionally, middleware APIs for the connection with the Communication Service providers are also presented, indicating the connection of the Communication Service providers with the vertical orchestrator.

The source code supporting the current software version is included in the respective repositories in the MATILDA GitLab.

References

- [**Abadi-2016**] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. Retrieved from <http://arxiv.org/abs/1603.04467>
- [**Bootstrap**] Bootstrap website, Available Online: <https://getbootstrap.com/>
- [**Compact-Profile**] Java Compact Profile, Available Online: <http://www.oracle.com/technetwork/java/embedded/embedded-se/documentation/compact-profiles-overview-2157132.html>
- [**DL4J**] Deep Learning for Java, Available Online: <https://deeplearning4j.org/>
- [**Drools**] Drools Documentation, Available Online: https://docs.jboss.org/drools/release/7.5.0.Final/drools-docs/html_single/index.html
- [**Drools Fusion**] Drools Fusion User Guide, Available Online: https://docs.jboss.org/drools/release/5.2.0.CR1/drools-fusion-docs/html_single/
- [**Duda-2000**] Duda, R. O., Hart, P. E., & Stork, D. G. (2000). Pattern Classification (2nd Edition). New York, NY, USA: Wiley-Interscience.
- [**eBPF**] A thorough introduction to eBPF, Available Online: <https://lwn.net/Articles/740157/>
- [**Java**] Java website, Available Online: <https://www.java.com/en/>
- [**Kafka**] Apache Kafka, Available Online: <https://kafka.apache.org/>
- [**LeCun-2015**] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521, 436. Retrieved from <http://dx.doi.org/10.1038/nature14539>
- [Maaten-2008] L.J.P. van der Maaten and G.E. Hinton. Visualizing High-Dimensional Data Using t-SNE. Journal of Machine Learning Research 9 (Nov):2579-2605, 2008.
- [**MATILDA**] MATILDA Website, Available Online: <http://www.matilda-5g.eu/>
- [**MATILDA-D1.1**] MATILDA D1.1 - MATILDA Framework and Reference Architecture, MATILDA H2020 Project, Available Online: <http://www.matilda-5g.eu/index.php/outcomes>
- [**MATILDA-D1.5**] MATILDA D1.5 - Deployment and Runtime Policy metamodel, MATILDA H2020 Project, Available Online: <http://www.matilda-5g.eu/index.php/outcomes>
- [**MATILDA-D3.1**] MATILDA D3.1- Intelligent Orchestration Mechanisms – First release, MATILDA H2020 Project, Available Online: <http://www.matilda-5g.eu/index.php/outcomes>
- [**MoonGen**] MoonGen GitHub repository, Available Online: <https://github.com/emmericp/MoonGen>
- [**Netdata**] Netdata website, Available Online: <https://my-netdata.io/>
- [**NFVEfficiency**] NFV Workload Efficiency Whitepaper, Available Online: <https://tl9000.org/resources/documents/NFV%20Workload%20Efficiency%20Whitepaper.pdf>
- [OSM] OSM website, Available Online: <https://osm.etsi.org>
- [**Prometheus**] Prometheus website, Available Online: <https://prometheus.io/>
- [**qMON**] Internet Institute website, Available Online: <http://www.qmon.eu/>

[React-JS] React JS website, Available Online: <https://reactjs.org/>

[Spark-MLib] Apache MLlib, Available Online:
<https://spark.apache.org/docs/latest/mllib-guide.html>

[Spring] Spring framework, Available Online: <https://www.spring.io>

[Spring-Boot] Spring-boot website, Available Online: <https://spring.io/projects/spring-boot>

[Symvoulidis-2019] C. Symvoulidis, I. Tsoumas, and D. Kyriazis, "Towards the identification of context in 5G infrastructures", Computing Conference 2019, London, UK, 2019. In press.

[Thymeleaf] Thymeleaf website, Available Online: <https://www.thymeleaf.org/>

[wrk] wrk GitHub repository, Available Online: <https://github.com/wg/wrk>