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

“5G-Ready Vertical Applications Orchestration”

A whitepaper written by the MATILDA Project

List of Authors

CNIT	CONSORZIO NAZIONALE INTERUNIVERSITARIO PER LE TELECOMUNICAZIONI
Roberto Bruschi, Chiara Lombardo, Franco Davoli	
ATOS	ATOS SPAIN SA
Fernando Diáz, Javier Melián, Esther Garrido Gamazo, Aurora Ramos	
ERICSSON	ERICSSON TELECOMUNICAZIONI
Orazio Toscano	
COSM	COSMOTE KINITES TILEPIKOINONIES AE
Ioanna Mesogiti	
ORO	ORANGE ROMANIA SA
Horia Stefanescu	
UBITECH	GIOUMPITEK MELETI SCHEDIASMOS YLOPOIISI KAI POLISI ERGON PLIROFORIKIS ETAIREIA PERIORISMENIS EFTHYNIS
Anastasios Zafeiropoulos, Panagiotis Gouvas, Eleni Fotopoulou, Thanos Xirofotos	
INC	INCELLIGENT IDIOTIKI KEFALAIOUCHIKI ETAIREIA
Athina Rodopi	
NCSRD	NATIONAL CENTER FOR SCIENTIFIC RESEARCH “DEMOKRITOS”
Akis Kourtis, Themis Anagnostopoulos	
AALTO	AALTO-KORKEAKOULUSAATIO
Miloud Bagaa, Ibrahim Afolabi	

Table of Contents

1 Introduction

In order to meet the increased demands from the consumer and business networking perspectives, Multi-access Edge Computing (MEC), Network Functions Virtualization (NFV) and Software Defined Networking (SDN), brought together, will lift to higher levels the computing capabilities and system-wide utility and efficiency of underlying infrastructure. When the combination of the programmability, scalability, flexibility and low latency characteristics of the aforementioned enabling technologies are leveraged, vertical network applications can easily be orchestrated while ensuring certain set Quality of Service (QoS) within the lifecycle of applications running on the network.

By leveraging these technological paradigms, the main system architecture taken as reference by the MATILDA project is a multi-layered and multi-tenant framework, fully compliant with the latest specification of 3GPP, ETSI NFV and cloud-native orchestration areas, and able to reflect the various administrative domains in a 5G network. The main peculiarity and advantage of the foundation of the MATILDA architecture consists of exposing 5G networks to application providers as a simple extension to today's Cloud Computing technologies, paradigms, and interfaces. Such an offering is exploited by Over-the-Top (OTT) players or Cloud Computing Applications providers in order to deploy and manage cloud-native applications, able to take advantage of 5G networks evolution. This flexibility and extensibility will be achieved by relying entirely on the MATILDA architecture that combines the aforementioned technologies.

In this respect, this whitepaper focuses on the design of the MATILDA vertical applications orchestrator (VAO) that is responsible for the lifecycle management of cloud-native applications based on their deployment over a 5G programmable infrastructure. The latter is managed by the MATILDA network platform that exposes an abstract view of the available resources to the VAO, while it realises the lifecycle management of network slices to serve the application needs. Following, we initially provide details for the MATILDA end to end story and the overall MATILDA reference architecture and, then, we delve into a detailed presentation of the VAO components.

2 MATILDA End to End Story

MATILDA comes up with a novel and holistic approach for tackling the overall lifecycle of applications' design, development, deployment and orchestration in a 5G ecosystem. A set of novel concepts are introduced, including the design and development of 5G-ready applications -based on cloud-native/microservices-based applications 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 applications 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, however with clear separation of concerns among them (see Figure 1).

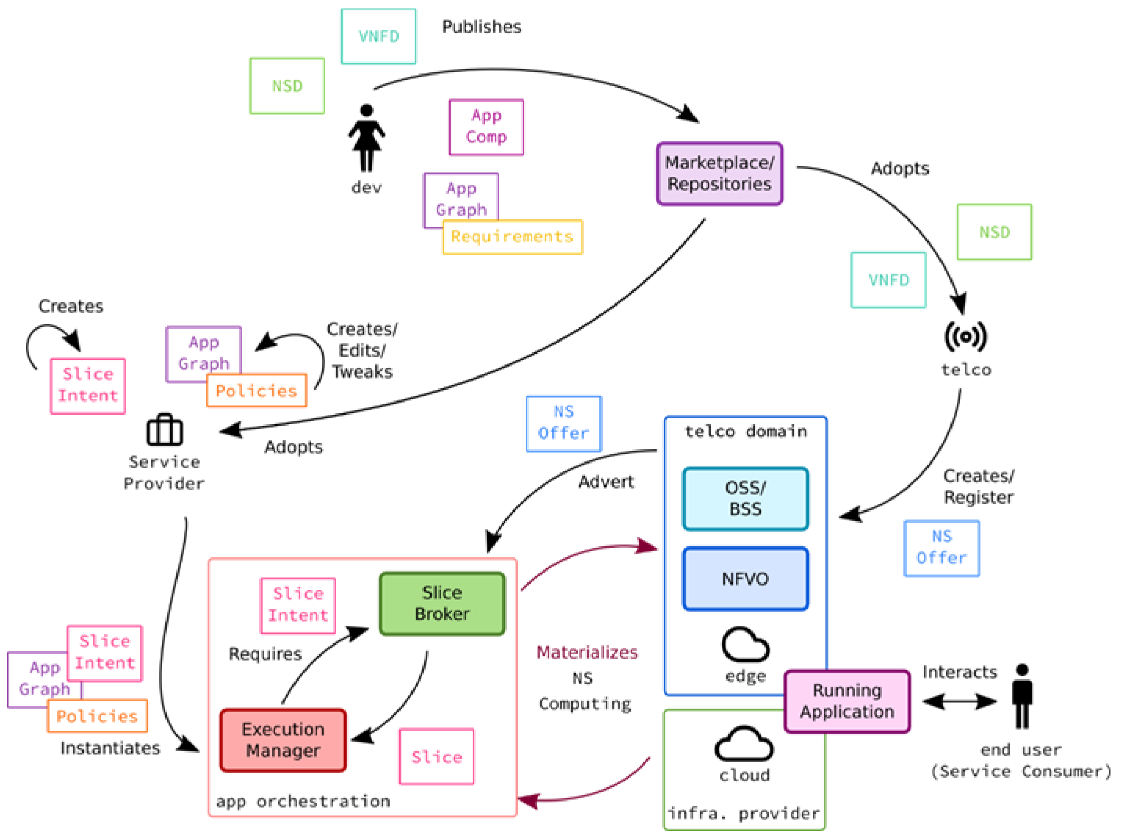


Figure 1: MATILDA Stakeholders View.

Software developers are developing applications following a microservices-based approach where each application component can be independently orchestratable. Based on the conceptualization of a metamodel (application graph metamodel), they declare information and requirements -in the form of an application descriptor- that can be exploited during the applications 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 a virtual link interconnecting two components within an application graph. The produced application is considered as 5G-ready application.

Application/Service Providers are able to adopt the developed 5G-ready applications and specify policies and configuration options for their optimal deployment and operation over programmable infrastructure. Based on the provided application descriptor, application/service providers are able to design operational policies that have to be applied as well as formulate the slice intent for the creation of the appropriate application-aware network slice. An application-aware network slice -that has to be provided by a communication service provider- is including information regarding the 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. Operational policies regard runtime adaptation of the execution mode of an application component. The deployment and runtime management of an application is realised by the Vertical Applications Orchestrator (managed by the application/service 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 communications service provider). Vertical applications orchestration is realised following a service-mesh-oriented approach.

The recently introduced service mesh concept is adopted as a software management layer for controlling and monitoring internal, service-to-service 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 application/service providers.

Communication Service Providers are getting the application deployment request by the Vertical Application Orchestrator and are able to proceed to the application-aware network slice instantiation and management during the overall lifecycle of the 5G-ready application. The concept of network slice is used for deployment and management of the required network services based on 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. Network slice instantiation and management is realised by the Network and Computing Slice Deployment Platform that 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. These actions are realized in an agnostic way to application service providers. However, through a set of open APIs, requests for adaptation of the network slice configuration may be provided by the Vertical Applications Orchestrator to the Network and Computing Slice Deployment Platform.

3 MATILDA Reference Architecture

3.1 MATILDA Architectural Approach Overview

The MATILDA reference architecture (Figure 2) 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 application/service providers and the 5G Infrastructure Slicing and Management Layer is oriented to communication service/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 main components developed in this layer regard the "Application Component" and "Application Graph" Repositories, the Programmable Infrastructure Registration Component, the Slice Intent Editor, the Runtime Policies Editor, the Application Graph Composer and the Profiler.

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. The main components developed in this layer regard the Deployment and Execution Manager, the Monitoring Engine, the Application Component Agent, the Runtime Policies Manager, the Service Discovery Mechanisms and the Data Analysis Mechanisms.

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. The main components developed in this layer regard the OSS/BSS, the NFV Orchestrator, the Wide-area Infrastructure Manager (WIM) and the Network Services Catalogue.

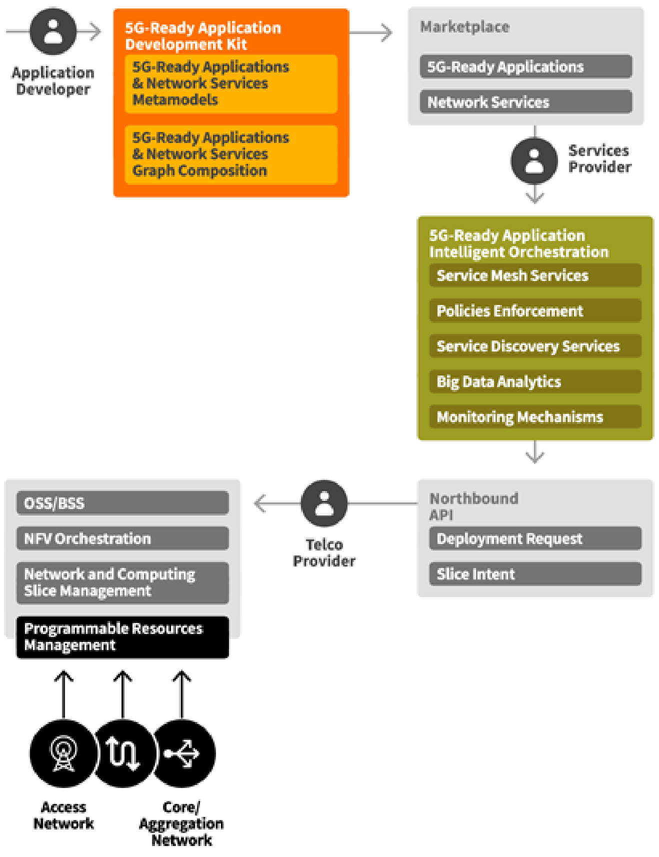

Application Developer
5G-Ready Application Development Kit
5G-Ready Applications & Network Services Metamodels
5G-Ready Applications & Network Services Graph Composition
Marketplace
5G-Ready Applications
Network Services
Services Provider
5G-Ready Application Intelligent Orchestration
Service Mesh Services
Policies Enforcement
Service Discovery Services
Big Data Analytics
Monitoring Mechanisms
Northbound API
Deployment Request
Slice Intent
Telco Provider
OSS/BSS
NFV Orchestration
Network and Computing Slice Management
Programmable Resources Management
Access Network
Core/ Aggregation Network

Figure 2: MATILDA Reference Architecture.

3.2 MATILDA Integrated Framework Overview

The aforementioned architectural approach has been adopted and instantiated in the development of the MATILDA integrated framework. This section shortly details the main integration points that have been implemented. These integration points regard a set of APIs supporting information exchange among the various components of the MATILDA architecture. The main supported interactions are as follows:

- The 5G-ready Applications Layer is interconnected with the Applications' Orchestration Layer for supporting onboarding of 5G-ready applications and application components to the MATILDA Vertical Application Orchestrator (VAO). Onboarding is realised by application developers that provide access to the developed software to application service providers. Upon the completion of the software development and validation processes, the developed software is made available into a set of Repositories and is accessible to a set of components within the VAO. Such components include the application graph composer and the policies editor. Furthermore, export of the composed application graphs in a YAML format is also supported, making the application descriptors accessible to application developers for further usage.
- The Applications' Orchestration Layer is interconnected with the Network and Computing Slice Management Layer for supporting interaction between telecom operators and application service providers during the application-aware network slice lifecycle management. The main interactions include the request for instantiation of an

application-aware network slice, the provision of information concerning the successful (or not) instantiation of the requested slice, and the provision of continuous network monitoring information related to QoS and QoE aspects.

The main interactions are depicted in Figure 3:

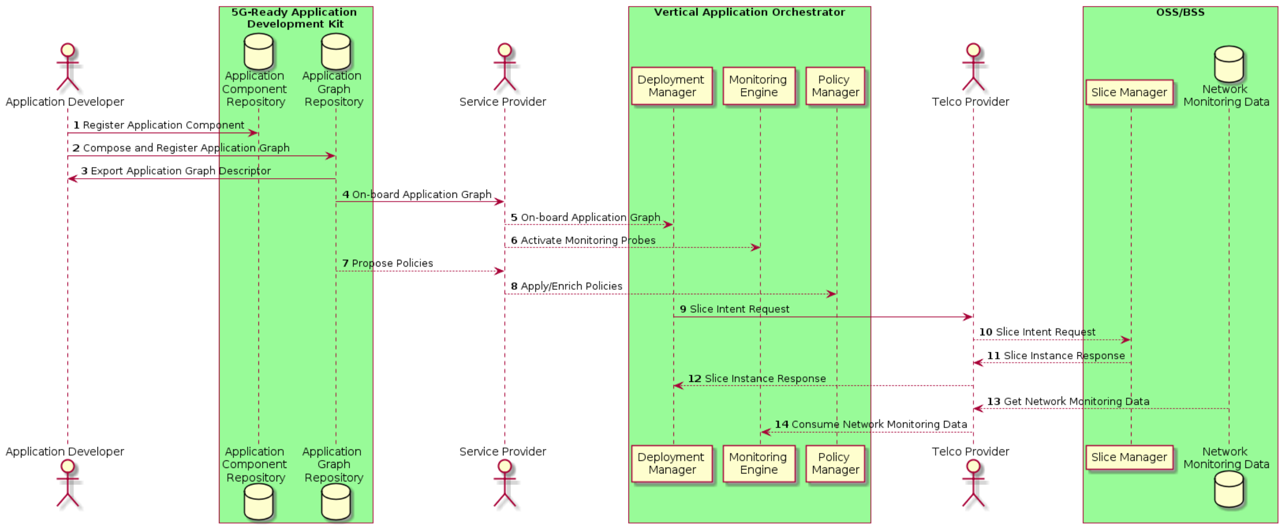

5G-Ready Application Development Kit
Vertical Application Orchestrator
OSS/BSS
Application Developer
Application Component Repository
Application Graph Repository
Service Provider
Deployment Manager
Monitoring Engine
Policy Manager
Telco Provider
Slice Manager
Network Monitoring Data
1 Register Application Component
2 Compose and Register Application Graph
3 Export Application Graph Descriptor
4 On-board Application Graph
5 On-board Application Graph
6 Activate Monitoring Probes
7 Propose Policies
8 Apply/Enrich Policies
9 Slice Intent Request
10 Slice Intent Request
11 Slice Instance Response
12 Slice Instance Response
13 Get Network Monitoring Data
14 Consume Network Monitoring Data

Figure 3: Main MATILDA Components Interactions.

4 Cloud-native Applications Orchestration

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.

4.1 *Deployment and Execution Manager*

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. Each vertical application consists of multiple components that formulate an application graph. The application graph and the components adhere to a specific metamodel [1-2]. 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 4.7. 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).

Furthermore, 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.

We have introduced the ability to support, inherently, scalability of horizontally scalable components. To achieve so, 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 4). 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. To support the development of elasticity controllers for various components, we followed a Service-Provider-Interface adapter architecture (a.k.a. SPI) and we decoupled the elasticity logic 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.

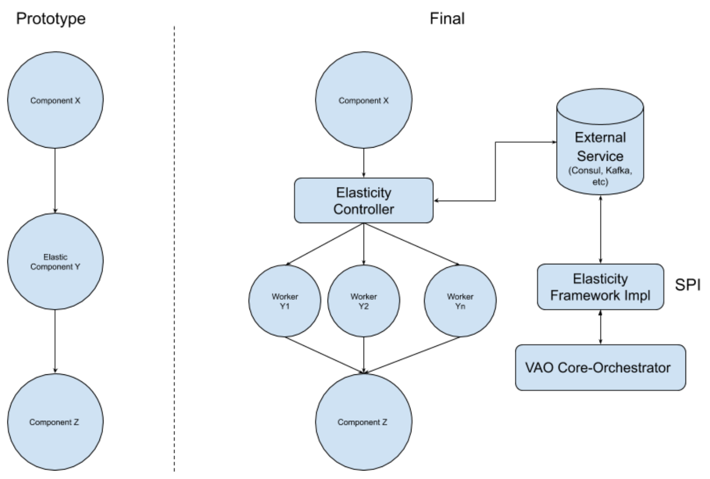


Figure 4: Elasticity Controller interaction with core orchestrator

4.2 MATILDA Agent and Service Discovery Mechanisms

The MATILDA Agent is one of the major artefacts of the first release. The main duty of the Agent is to handle the signalling (at layer 7) between the orchestrator and the core platform. However, the Agent supports also efficient security policy enforcement at the component level.

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.

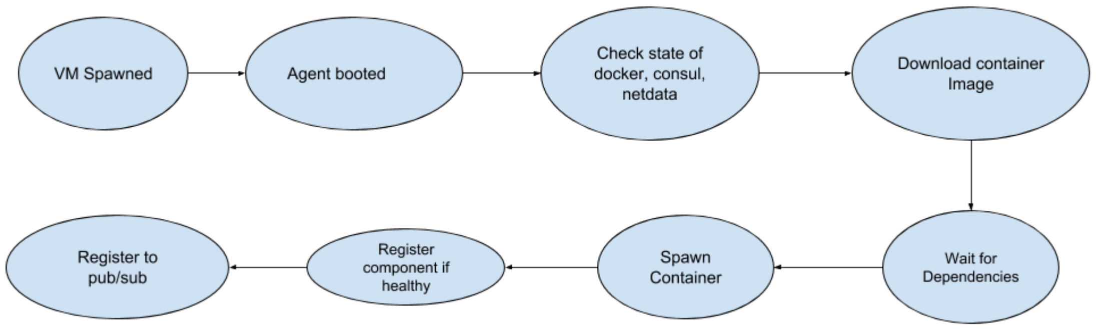


Figure 5: Lifecycle of MATILDA Agent

These steps are graphically depicted in Figure 5 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**.

As is 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.

4.3 Monitoring Mechanisms

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. 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.

One of the major challenges was the **unification of the monitoring streams** that are generated from the operation of the various components/layers. In their final form, 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.

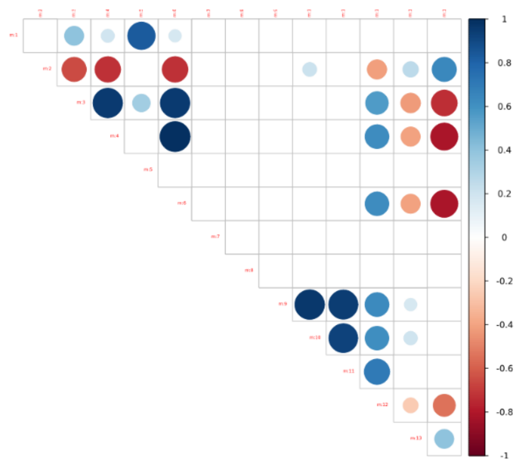
Implementation of the overall monitoring solutions is based on the deployment of Prometheus instances, able to aggregate data coming from various agents. Part of these agents are based on the QMON monitoring platform for QoS/QoE metrics as well as Netdata for VM/container-based metrics.

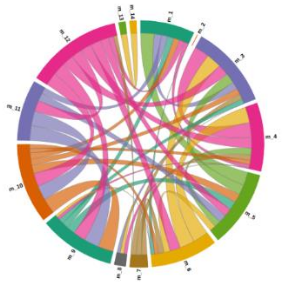
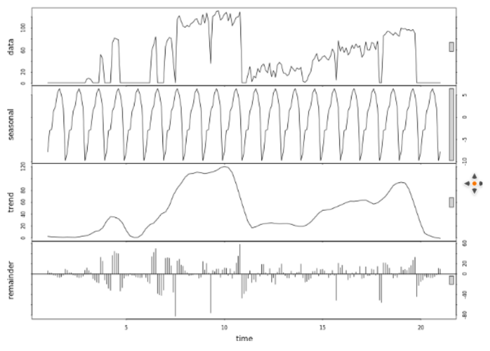
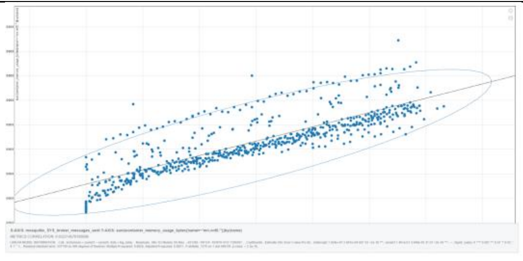
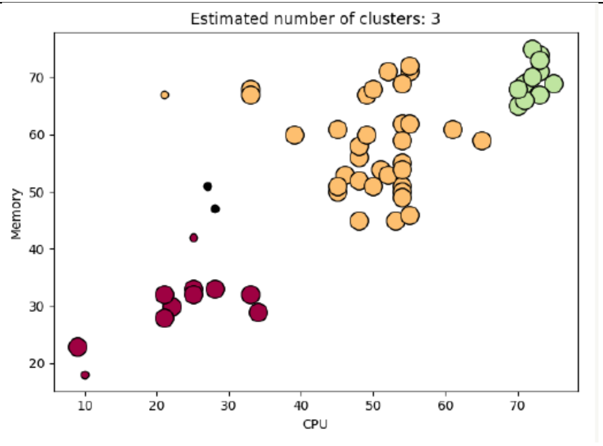
4.4 Data Analytics Toolkit

The design and implementation of the toolkit took place focusing on supporting:

- 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.

A set of analysis processes/scripts have been integrated, 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	 **Figure 6: Indicative screenshot (correlogram) from a correlation analysis**

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 7: 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 8: 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 9: Indicative screenshot from a linear regression analysis
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	 Figure 10: Indicative screenshot from a clustering analysis

operation is also realised.	
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 11**. 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.

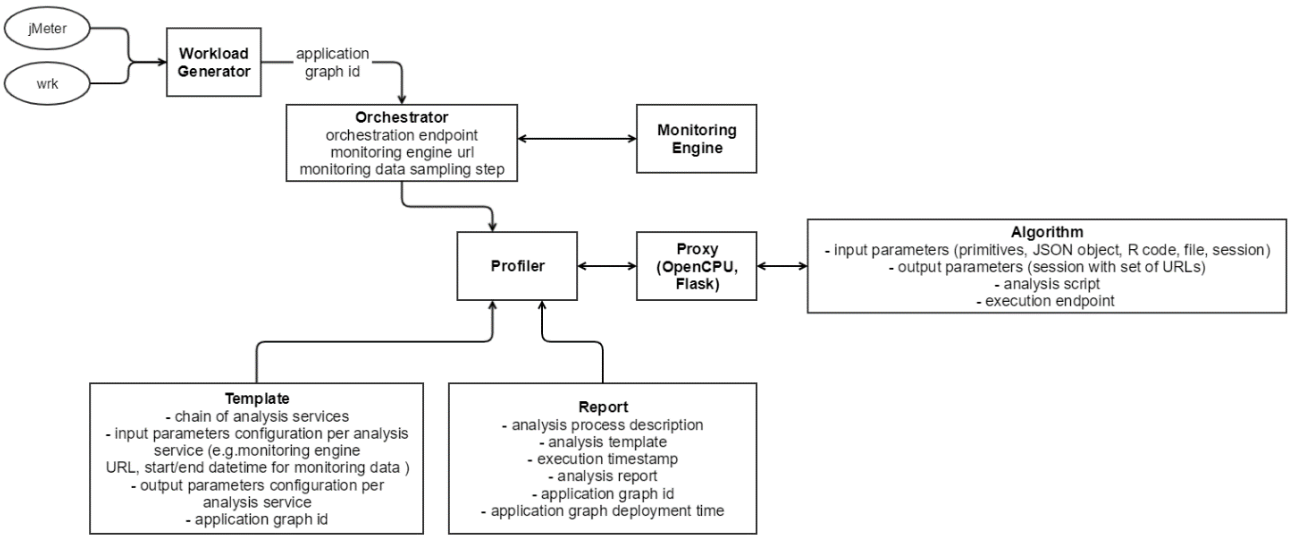

jMeter
wrk
Workload Generator
application graph id
Orchestrator
orchestration endpoint
monitoring engine url
monitoring data sampling step
Monitoring Engine
Profiler
Proxy (OpenCPU, Flask)
Algorithm
- input parameters (primitives, JSON object, R code, file, session)
- output parameters (session with set of URLs)
- analysis script
- execution endpoint
Template
- chain of analysis services
- input parameters configuration per analysis service (e.g.monitoring engine URL, start/end datetime for monitoring data)
- output parameters configuration per analysis service
- application graph id
Report
- analysis process description
- analysis template
- execution timestamp
- analysis report
- application graph id
- application graph deployment time

Figure 11: Analytics toolkit architectural approach

4.5 Runtime Policies Enforcement

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 [4].

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 12. 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.

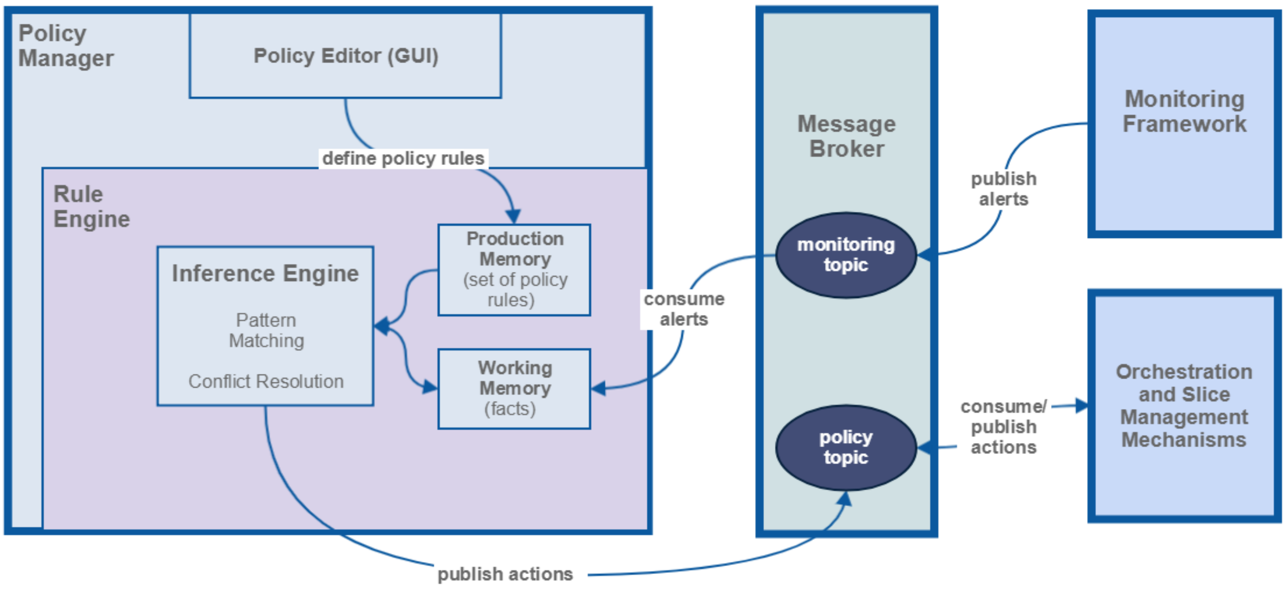


Figure 12: Policy Manager and Monitoring Mechanisms Interaction

In order to overcome potential computational problems, a distributed implementation of the policy manager has been realised, in addition to the previous monolithic approach. In the new version, horizontal scalability of the policy manager is supported, as shown in Figure 13.

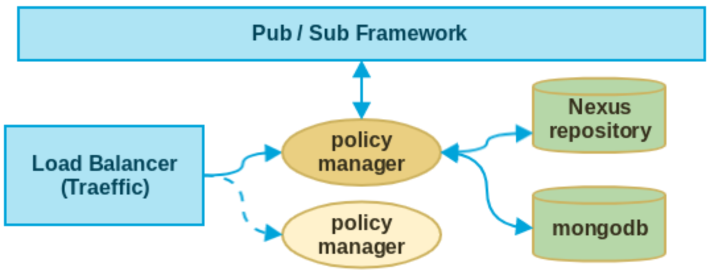


Figure 13: Horizontally Scalable Policy Manager Design

At the new approach, 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. 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 approach has as a result 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.

4.6 Complex Event Processing

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. As presented in Figure 14, the engine is divided in two major components, namely the *Complex Event Processing engine* and the *Threshold identifier*, which are described in more detail below.

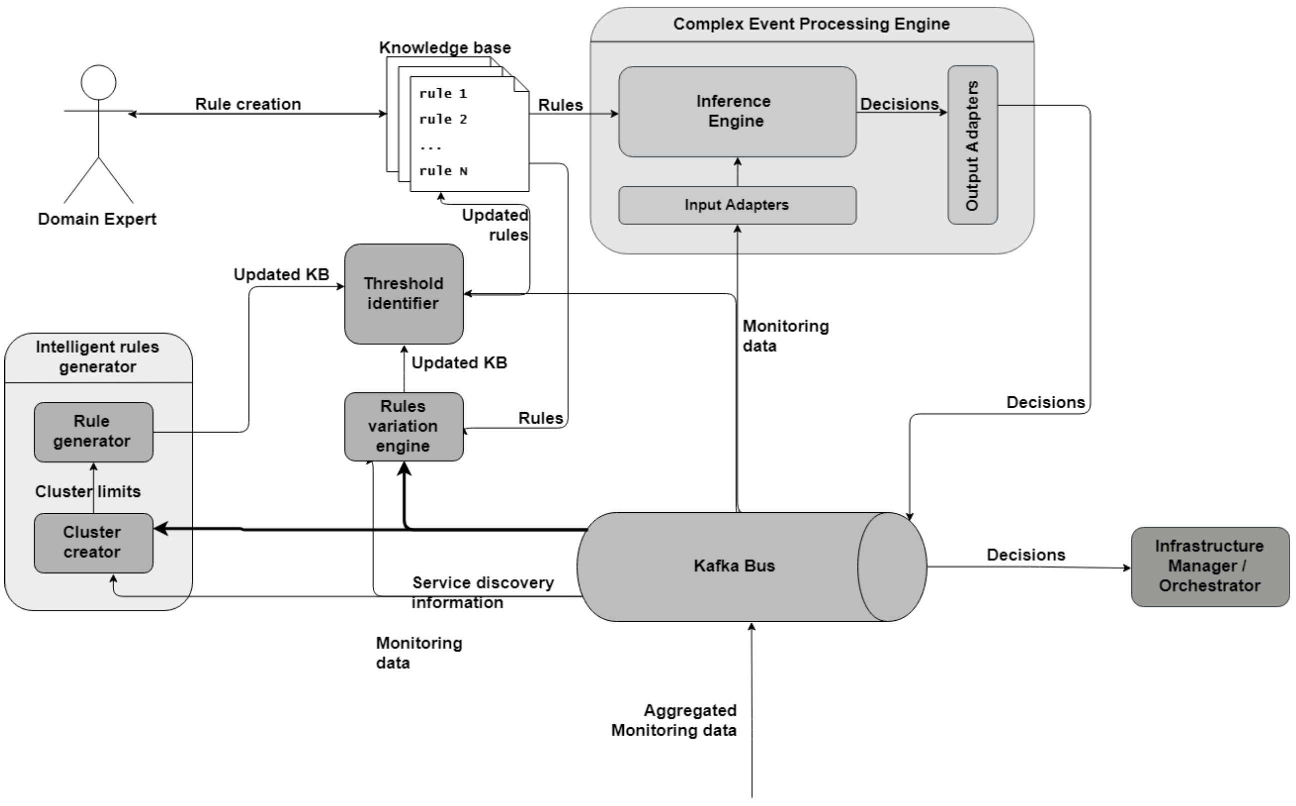


Figure 14: Dynamic Complex Event Processing mechanism – Overall architecture

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.

4.7 Northbound APIs for Communication Service Providers

Northbound APIs are provided by the Operations Support System (OSS) towards the VAO. OSS is a web application, which offers a user interface along with a RESTful API that is used by the Orchestrator. Two interfaces have been specified and partially implemented for supporting the Northbound APIs:

- Interface for accepting a Slice Intent from the Orchestrator. It 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 (see serialisation format in Table 1).
- 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 (see serialisation format in Table 2).

Table 1: 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 2: 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"
  }
```

5 Conclusions

In the current manuscript, we have provided a short overview of the MATILDA architectural approach, followed by a detailed description of the components and functionalities provided by the Vertical Applications Orchestrator (VAO). Focus is given on the presentation of the VAO components, their role in the overall architectural approach and the technological choices made for their design and development. The set of components are fully extensible and can be evolved considering evolving technologies in the area of orchestration of cloud-native applications and the need for development of interfaces for interaction with telecom operators.

References

[1] MATILDA Deliverable D1.2, "Chainable Application Component & 5G-ready Application Graph Metamodel".

[2] MATILDA Deliverable D1.2, "VNF/PNF & VNF Forwarding Graph Metamodel".

[3] MATILDA Deliverable D1.4, "Network-aware Application Graph Metamodel".

[4] MATILDA Deliverable D1.5, "Deployment and Runtime Policy Metamodel"