



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

DELIVERABLE D2.2

5G-READY APPLICATIONS AND NETWORK SERVICES DEVELOPMENT ENVIRONMENT AND MARKETPLACE

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>
Workpackage:	WP2 – 5G-Ready Applications and Network Services Development Environment and Marketplace
Type of the Deliverable:	OTHER
Dissemination level:	PU
Editors:	UPRC, UBITECH, INC, SUITE5, ININ, ATOS, NCSRD
Version:	1.0

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:

35 months

Start Date:

01/06/2017 *dd/mm/yyyy*

Project Coordinator:

Name:

Franco Davoli

Phone:

+39 010 353 2732

Fax:

+39 010 353 2154

e-mail:

franco.davoli@cnit.it

Technical Coordinator:

Name:

Panagiotis Gouvas

Phone:

+30 216 5000 503

Fax:

+30 216 5000 599

e-mail:

pgouvas@ubitech.eu

List of Authors	
NCRD	NATIONAL CENTER FOR SCIENTIFIC RESEARCH “DEMOKRITOS”
Themistoklis Anagnostopoulos, George Xylouris	
UBITECH	GIOUMPITEK MELETI SCHEDIASMOS YLOPOIISI KAI POLISI ERGON PLIROFORIKIS ETAIREIA PERIORISMENIS EFTHYNIS
Panagiotis Gouvas, Anastasios Zafeiropoulos, Eleni Fotopoulou, Thanos Xirofotos	
INC	INCELLIGENT IDIOTIKI KEFALAIOUCHIKI ETAIREIA
Nikos Stasinopoulos, Athina Ropodi, Panagiotis Demestichas	
SUITE5	SUITE5 DATA INTELLIGENCE SOLUTIONS LIMITED
Lefteris Lampathakis, Katerina Zerva	
ININ	INTERNET INSTITUTE, COMMUNICATIONS SOLUTIONS AND CONSULTING LTD
Luka Koršič, Janez Sterle	
ATOS	ATOS SPAIN SA
Aurora Ramos, Fernando Díaz, Javier Melián	
UPRC	UNIVERSITY OF PIRAEUS RESEARCH CENTER
Ilias Tsoumas, Eftychia Vorila, Dimitris Drakoulis, Marios Touloupou	
INTRA	INTRASOFT INTERNATIONAL SA
Kostas Thivaiois, Marios Logothetis	

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 © 2019 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 SCHEDIASMO S 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 FÜR 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 FIGURES	5
1 EXECUTIVE SUMMARY.....	6
2 INTRODUCTION	7
2.1 SCOPE	7
2.2 MATILDA END TO END STORY.....	7
3 MATILDA DASHBOARD	9
4 COMPONENT MANAGEMENT.....	10
5 APPLICATION GRAPH MANAGEMENT.....	15
6 RUNTIME ELASTICITY AND SECURITY POLICIES MANAGEMENT	21
7 MONITORING AND PROFILING MECHANISMS.....	24
8 NETWORK SLICE SPECIFICATION AND MANAGEMENT.....	25
9 CONCLUSIONS	31
REFERENCES	32

Table of Figures

Figure 1: MATILDA workflow highlighting the different stakeholders and metamodels.....	9
Figure 2: MATILDA Dashboard Overview.....	10
Figure 3: MATILDA Component element	11
Figure 4: Overview of Component Registration.....	12
Figure 5: Registration of Distribution Parameters	12
Figure 6: Registration of Access/Core Interfaces	13
Figure 7: Registration of Configuration Parameters	13
Figure 8: Registration of Volume and Device Mappings.....	14
Figure 9: Registration of Minimum Execution Requirements.....	14
Figure 10: Registration of Exposed Metrics.....	14
Figure 11: Component Marketplace	15
Figure 12: Application Graph element	16
Figure 13: Search Marketplace for Components.....	17
Figure 14: Introspect Component's chain-ability profile	17
Figure 15: Suggest candidate component that satisfies a specific interface.....	18
Figure 16: Materialized chain.....	19
Figure 17: Application Graph Instance and Signalling Information.....	20
Figure 18: Monitoring data regarding CPU usage of an Application Graph Component.....	20
Figure 19: Sample of a Grafana Monitoring Dashboard.....	22
Figure 20: Initiate the specification of Policies through the Application Instances view	23
Figure 21: Elasticity Policy Definition	23
Figure 22: Security Policy Definition	23
Figure 23: Analysis Configuration.....	24
Figure 24: Indicative Analysis Result (Linear Regression model)	25
Figure 25: Slice Intent - Resource Constraints element.....	25
Figure 26: Slice Intent - Graph link QoS constraints element	26
Figure 27: Application Graph that will be used for setting constraints.....	26
Figure 28: Declaring Resource Constraints.....	27
Figure 29: Declaring Link Constraints.....	28
Figure 30: Declaring Access Constraints	28
Figure 31: Materialized Slice within the OSS.....	29
Figure 32: Part of Slice Intent.....	30
Figure 33: Part of Slice Descriptor	30

1 Executive Summary

The scope of MATILDA is to deliver a Holistic, Innovative Framework for the Design, Development and Orchestration of 5G-ready Applications and Network Services over Sliced Programmable Infrastructure. As already analysed in D1.2 [2] a 5G-ready Application is represented by an application graph, consisting of chainable application components. A 5G-ready application includes in its description a set of network-oriented requirements that can be translated towards the production of a slice intent. 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.

The scope of this deliverable is to provide a short assistive documentation to the existing development environment that has been built in order to assist **a)** a developer to register his/her component in such a way which will respect the Component Metamodel; **b)** a DevOps to create an application graph in such a way which will respect the Application Graph Metamodel and **c)** a DevOps to define in a comprehensive way the constraints that can be declared which will affect the proper slice creation, as well as the set of policies that can be applied during runtime.

In order to cope with all three requirements, a dedicated web-based environment has been developed which graphically assists developers and DevOps to create and register their components without dealing with the overhead of syntactic conformance. The aim of the current document is to complement the actual deliverable, which is the environment per se.

It should be noted that the development of the MATILDA components followed a continuous and iterative process. To this end, this deliverable is a revised version of D2.1 that included the first release of the relevant mechanisms. The outcomes of this deliverable are provided to WP5 for the release of the integrated version of the MATILDA platform.

2 Introduction

2.1 Scope

This deliverable aims at describing the implementation of the final release of the MATILDA components that assist developers and DevOps register their 5G-ready applications. It regards a revised version of D2.1 that included the description of the first release of the relevant mechanisms. Minor improvements may be also realized in the upcoming period, given the feedback that is provided by the usage of the MATILDA framework within the Demonstrators.

In more detail, in this deliverable a detailed overview of the components comprising the MATILDA Development environment is presented, including the interfaces between them, as well as any interactions with external entities. Some additional information is provided per mechanism, including the baseline technologies used and the current development and integration status.

Special emphasis will be given in covering the entire logical cycle spanning from the declaration of newly developed component to its deployment in the frame of an application graph. The endmost goal of the developed environment is to make the process of registering a component, creating a graph and defining constraints that are syntactically correct.

The data models that represent the components, the application graphs and the constraints are thoroughly described in D1.2 [2], D1.3 [3], D1.4 [4] and D1.5 [5]. This deliverable assumes that the reader is familiar with the data structures and the insights of the XSD models that have been developed. In any case, for the sake of better comprehension, part of the modelling artefacts will be listed within the document.

The structure of the deliverable is as follows; Chapter 2 describes the overall MATILDA Dashboard, Chapter 3 describes the Components' registration and management functionalities, while Chapter 4 elaborates on the Application Graph creation. Chapter 5 and 6 focus on providing details on the Policies Editor, Monitoring and Profiling mechanisms. Chapter 7 elaborates the mechanisms for declaration of the set of requirements and constraints that lead to the specification of the slice intent per application graph. Finally, Chapter 8 concludes the document.

2.2 MATILDA End to End Story

As mentioned in [6], MATILDA offers a framework that 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 descriptors- that can be exploited during the deployment and operation over a 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.

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 the 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 mechanisms (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. 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 mechanisms include 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 its 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 emphasis in the first version of the Marketplace is put in the registration and management of components and application graphs of verticals. These artefacts are used by the Vertical Orchestrator in order to create a Slice Intent and perform the actual deployment upon a created slice.**

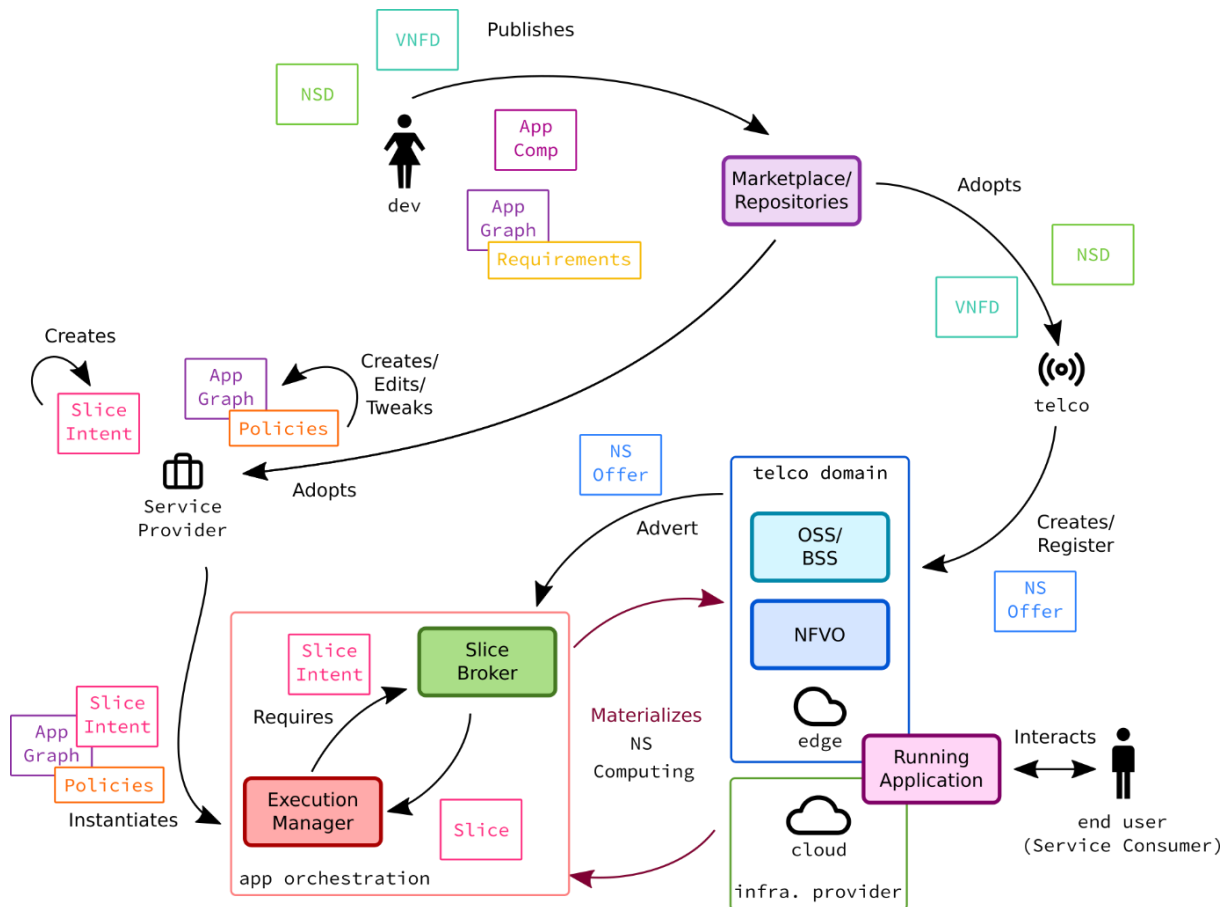


Figure 1: MATILDA workflow highlighting the different stakeholders and metamodels.

3 MATILDA Dashboard

The new integrated version of the MATILDA Dashboard comes with an ergonomic view providing access to all the supported functionalities (see Figure 2). It is the main entry point for MATILDA stakeholders, mainly application developers and service providers. Application developers are able to create and manage their applications. They can register and configure application components, while by using the graph composer they can create application graphs. A set of constraints can be declared that are considered during the deployment and runtime phase. Service providers are able to register or select the usage of 5G programmable infrastructure, design security and elasticity policies that can be applied during runtime, as well as manage the overall lifecycle of the deployment and management of the application graphs. The latter is supported by a set of developed views based on data provided by the MATILDA monitoring mechanisms.

In the frontpage of the MATILDA Dashboard, aggregated information is illustrated for the usage of the available resources, along with statistics about the operational application graph instances and the set of developed application components and graphs and information about policies enforcement aspects. The MATILDA Dashboard is designed to provide every piece of necessary information from the orchestration perspective without sacrificing the user-friendly experience.

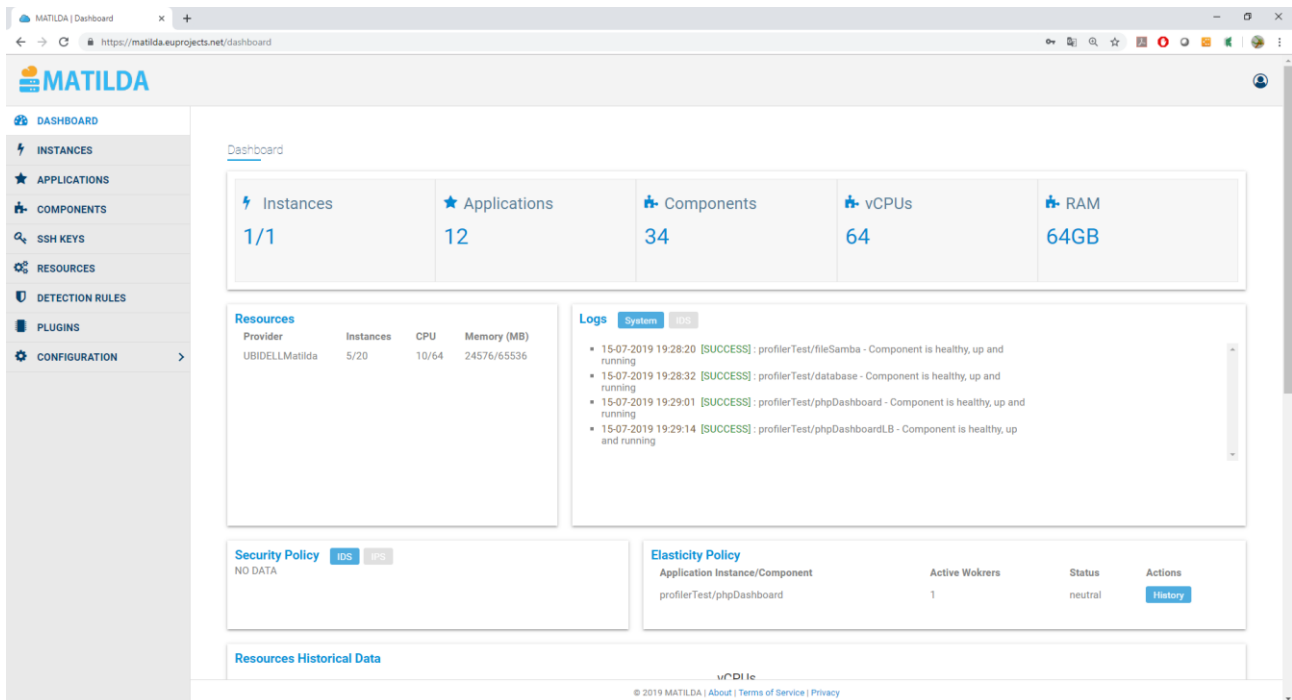


Figure 2: MATILDA Dashboard Overview.

4 Component Management

As already analyzed in D1.2 [2], the MATILDA component metamodel includes a set of fundamental complexType elements that uniquely describe each component in the entire application graph. These elements are the following – depicted in Figure 3: (i) *Distribution*, (ii) *ExposedInterface*, (iii) *Configuration*, (iv) *Volume*, (v) *MinimumExecutionRequirements*, (vi) *ExposedMetric*, (vii) *RequiredInterface*, and (viii) *Capability*.

A REACT.JS¹ user interface has been built in order to assist a developer register his/her component to the MATILDA platform. An overview of the environment is presented in Figure 4. This environment is provided to developers that are implementing 5G-ready apps.

As depicted on the XSD of Figure 3 the first complex element of the XSD is the distribution parameters. It encapsulates the information that is required for fetching an instance of a Component. It contains the information regarding the final image/container of a component and the URI where the component is located in the MATILDA repository. This information is declared and validated both from the frontend (see Figure 5) and the backend.

¹ <https://reactjs.org/>

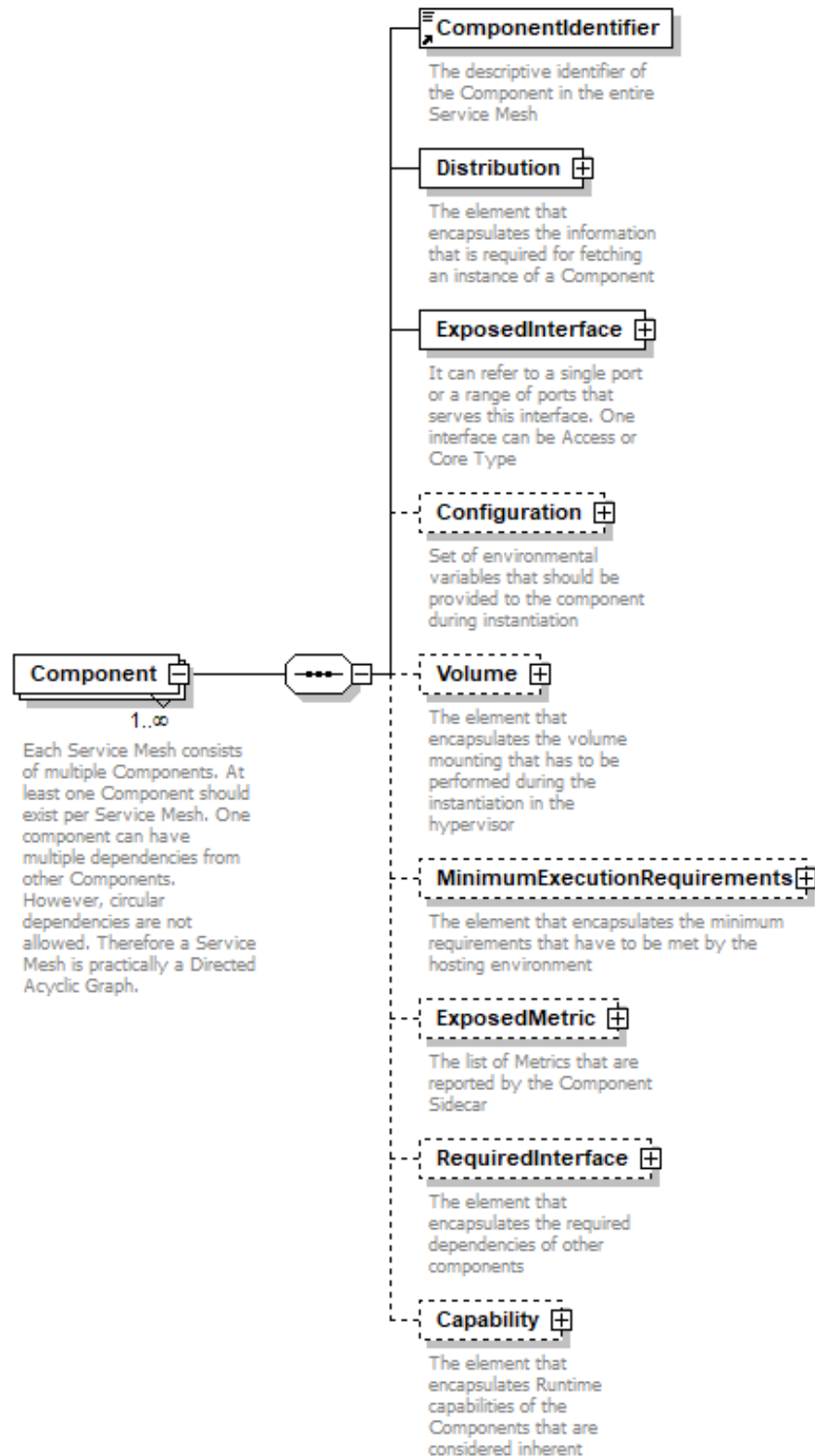
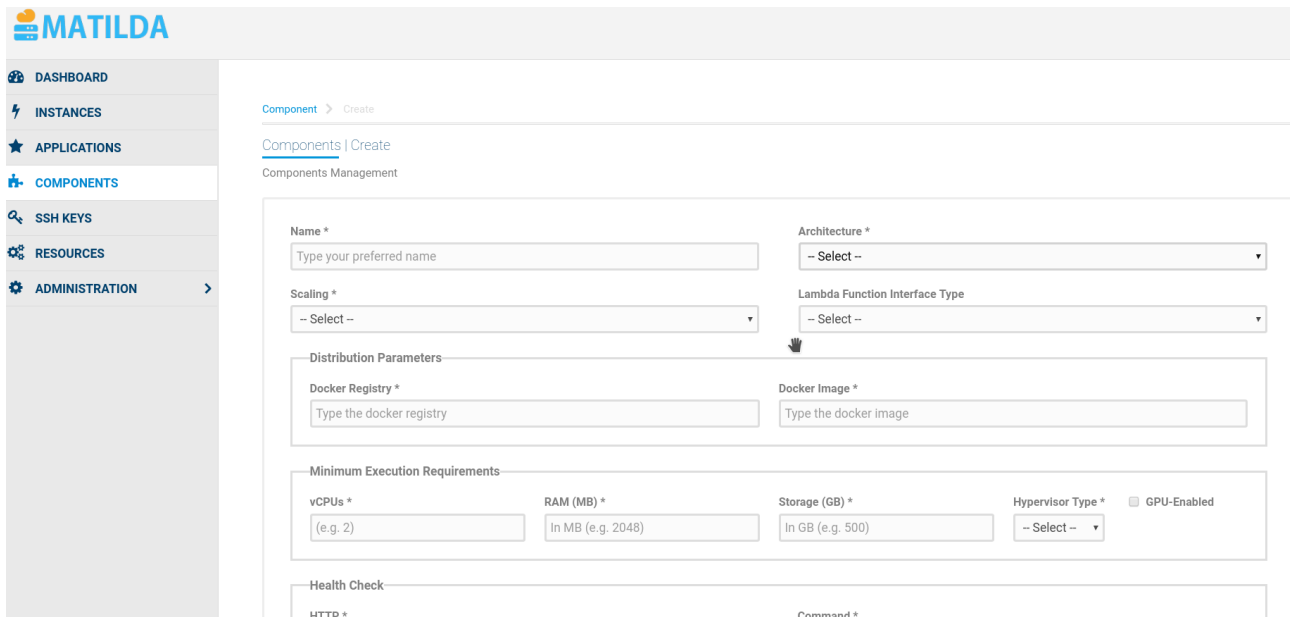
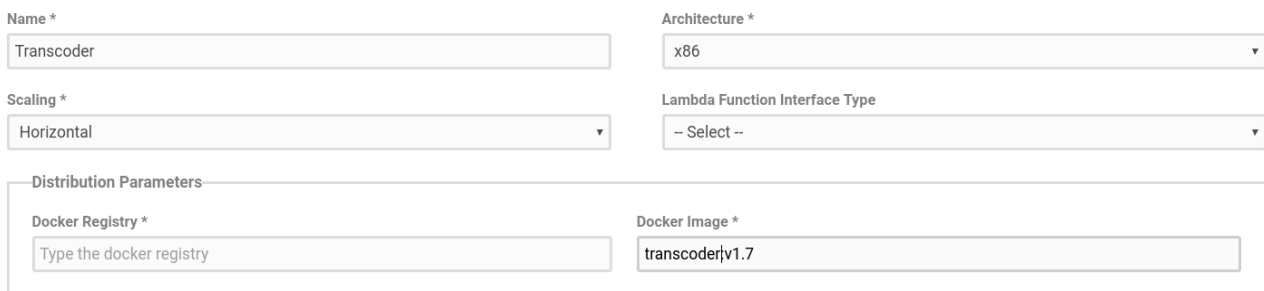


Figure 3: MATILDA Component element



The screenshot shows the MATILDA web interface with a sidebar on the left containing navigation links: DASHBOARD, INSTANCES, APPLICATIONS, COMPONENTS, SSH KEYS, RESOURCES, and ADMINISTRATION. The main content area is titled 'Component > Create' and 'Components | Create'. Below this, there's a 'Components Management' section. The form itself is divided into several sections: 'Name *' with a text input field; 'Architecture *' with a dropdown menu; 'Scaling *' with a dropdown menu; 'Distribution Parameters' with 'Docker Registry *' and 'Docker Image *' text input fields; 'Minimum Execution Requirements' with 'vCPUs *', 'RAM (MB) *', 'Storage (GB) *', and 'Hypervisor Type *' dropdowns, and a 'GPU-Enabled' checkbox; and 'Health Check' with 'HTTP *' and 'Command *' text input fields.

Figure 4: Overview of Component Registration

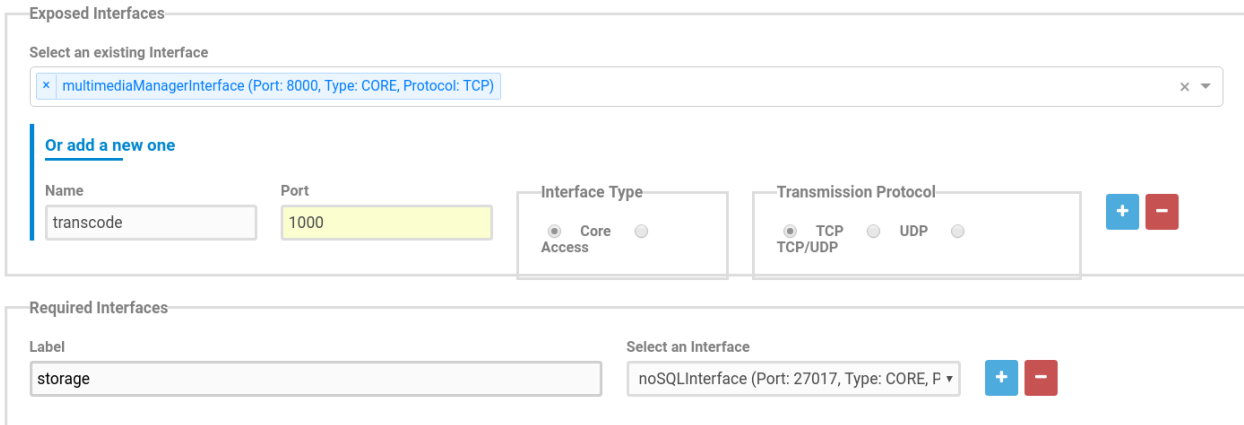


This screenshot shows a portion of the component registration form. The 'Name *' field contains 'Transcoder'. The 'Architecture *' dropdown is set to 'x86'. The 'Scaling *' dropdown is set to 'Horizontal'. The 'Distribution Parameters' section shows 'Docker Registry *' with a placeholder 'Type the docker registry' and 'Docker Image *' with the value 'transcoder|v1.7'.

Figure 5: Registration of Distribution Parameters

The second complex type element is critical since it describes the exposed interfaces. It is a “one-to-many” relation because each component may expose several interfaces. It encapsulates the descriptive identifier of the interface, which is required in order to infer the chainability of dependencies during the Service Mesh deployment. Furthermore, it contains the classification of the exposed interface based on its positioning in the 5G network. It can be ACCESS or CORE. Moreover, it contains port declaration and an optional choice for the transport layer protocol.

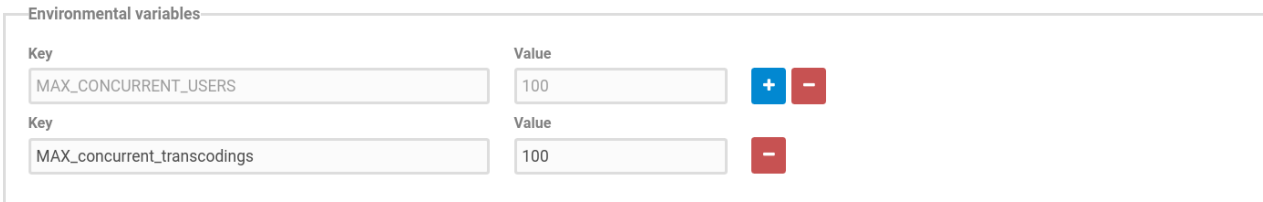
On the other hand, each component requires some inputs. Thus, the required interface section of the component encapsulates via a “one-to-many” relation the information regarding the graph link, which links the current component with another component. Specifically, it encapsulates the identifier of the component that satisfies the current component input needs and the corresponding exposed interface identifier. Of course, there is also a descriptive identifier of this logical link (“GraphLink”) between two components. The declaration of exposed and required interfaces is depicted in Figure 6.



The screenshot shows two configuration panels. The top panel, 'Exposed Interfaces', has a section 'Select an existing Interface' with a dropdown menu showing 'multimediaManagerInterface (Port: 8000, Type: CORE, Protocol: TCP)'. Below this is a link 'Or add a new one'. Underneath are input fields for 'Name' (transcode), 'Port' (1000), 'Interface Type' (radio buttons for Core and Access, with Core selected), and 'Transmission Protocol' (radio buttons for TCP, UDP, and TCP/UDP, with TCP selected). There are '+' and '-' buttons to the right. The bottom panel, 'Required Interfaces', has a 'Label' input field with 'storage'. To its right is a 'Select an Interface' dropdown showing 'noSQLInterface (Port: 27017, Type: CORE, P'. There are also '+' and '-' buttons to the right of the dropdown.

Figure 6: Registration of Access/Core Interfaces

The next element is the “*Configuration*”. Configuration represents a set of environmental variables that should be provided to the component during instantiation. Practically, it is a generic collection of key-value pairs to be exploited for deployment and instantiation. These key-value pairs are declared and validated as Figure 7 illustrates.



The screenshot shows the 'Environmental variables' panel. It contains two rows of configuration. The first row has a 'Key' input field with 'MAX_CONCURRENT_USERS' and a 'Value' input field with '100'. To the right of the value field are '+' and '-' buttons. The second row has a 'Key' input field with 'MAX_concurrent_transcodings' and a 'Value' input field with '100'. To the right of the value field is a '-' button.

Figure 7: Registration of Configuration Parameters

The application component also includes the “*Volume*” element. It is a capability of the Hypervisors to provide storage to a VM via volumes. To capture the corresponding cases, the model includes three “children” for each volume instance. The definition of the type of the volume is needed since, if it has been attached to the guest using one hypervisor type (e.g. Xen), it cannot be attached to a guest that is using another hypervisor type, for example vSphere, KVM. This is because the different hypervisors use different disk image formats. Additionally, the volume element includes sub-elements for the source and the target of each volume. Volume mapping is depicted in Figure 8.

Volumes

Name

+

-

Name

-

Devices

Key	Value	
<input type="text" value="e.g GPU"/>	<input type="text" value="100"/>	<div>+</div> <div>-</div>

Figure 8: Registration of Volume and Device Mappings

Furthermore, a crucial aspect of the component relates to its minimum requirements that have to be met by the hosting environment for the proper execution. This complex element contains the vCPUs element that refers to the minimum amount of vCPUs that should be provided by the hypervisor, the minimum RAM and Storage (through the respective elements) and an element regarding the type of the hypervisor that is preferred (i.e., Esxi, KVM, Xen). Declaration of minimum execution requirements is depicted in Figure 9.

Minimum Execution Requirements

vCPUs *	RAM (MB) *	Storage (GB) *	Hypervisor Type *	<input checked="" type="checkbox"/> GPU-Enabled
<input type="text" value="2"/>	<input type="text" value="2048"/>	<input type="text" value="500"/>	<input type="text" value="KVM"/>	

Figure 9: Registration of Minimum Execution Requirements

Besides, the application component metamodel includes a section regarding the metrics that will be reported by the proxy sidecar, the so called “*ExposedMetric*”. It is a key-value structure with the metric identifier as key and the unit of it as value. Each component may expose multiple metrics, as depicted in Figure 10.

Exposed Metrics

Name	Unit	
<input type="text" value="s"/>	<input type="text" value="%"/>	<div>+</div> <div>-</div>
Name	Unit	
<input type="text" value="transcodingtime per 100Mb"/>	<input type="text" value="%"/>	<div>-</div>

Figure 10: Registration of Exposed Metrics

Upon the registration of a component, it can be declared as a public or a private one. Public components can be used by all developers of the MATILDA ecosystem, while private ones can be used only by their developers. Each developer and DevOps user in the MATILDA ecosystem owns a private space, where components and application graphs can be registered and used. Figure 11 illustrates the search functionality of the MATILDA Marketplace as far as components are concerned.

Components

Components Management

Create new

Name

Search by Name

Filter

Reset

Name ▾	Visibility	Date Created
FaceDetector	Public	Wednesday, September 19, 2018 5:24 PM
LambdaCoreApp	Public	Wednesday, September 19, 2018 5:24 PM
LambdaCoreAppArm	Public	Wednesday, September 19, 2018 5:24 PM
LambdaProxy	Public	Wednesday, September 19, 2018 5:24 PM
MariaDB	Public	Wednesday, September 19, 2018 5:24 PM

<<

<

>

>>

Figure 11: Component Marketplace

5 Application Graph Management

Many chainable components can be combined in order to create a 5G-ready application graph. As already described, an application graph is practically a directed acyclic graph (DAG) that is implemented as a Service Mesh. As depicted in Figure 12, given the adoption of the service mesh paradigm, the form of the application metamodel is simple. It contains a “*ServiceMeshIdentifier*” for the unique identification of each 5G-ready application, a “*Name*” that includes the descriptive name of each Service Mesh. As expected, a “one-to-many” relation is used to capture the correlation between the Service Mesh and its components.

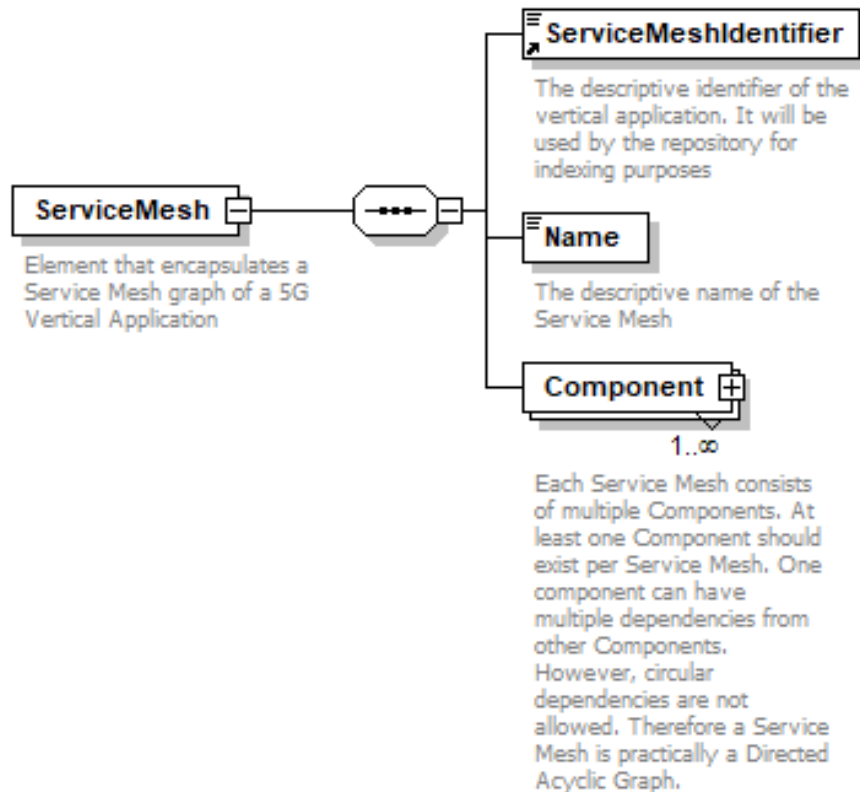


Figure 12: Application Graph element

The metamodel is fully informative. Specifically, each link is considered as an input required interface of the component which needs it. Hence it is a logical link, and each logical link has to be realized as a network link, with network and compute constraints.

In the frame of MATILDA special emphasis has been given to the development of an intuitive graph composition environment which will “guide” the user for the creation of syntactically and logically correct graphs based on the available components in the marketplace. An application graph starts by selecting one component from the available ones and dragging it to the drawing canvas, as seen in Figure 13.

Clicking on each component in the canvas, one can introspect the amount of interfaces that are exposed/required and if the required interfaces are satisfied by any other component in the canvas. This introspection is illustrated in Figure 14.

In order to create a link between two components, a developer can either try to find manually a component that satisfies the required interface or can rely on the automated component suggestion functionality that automatically identifies and presents to the user (in a ranked sorted list) the compatible components (see Figure 15).

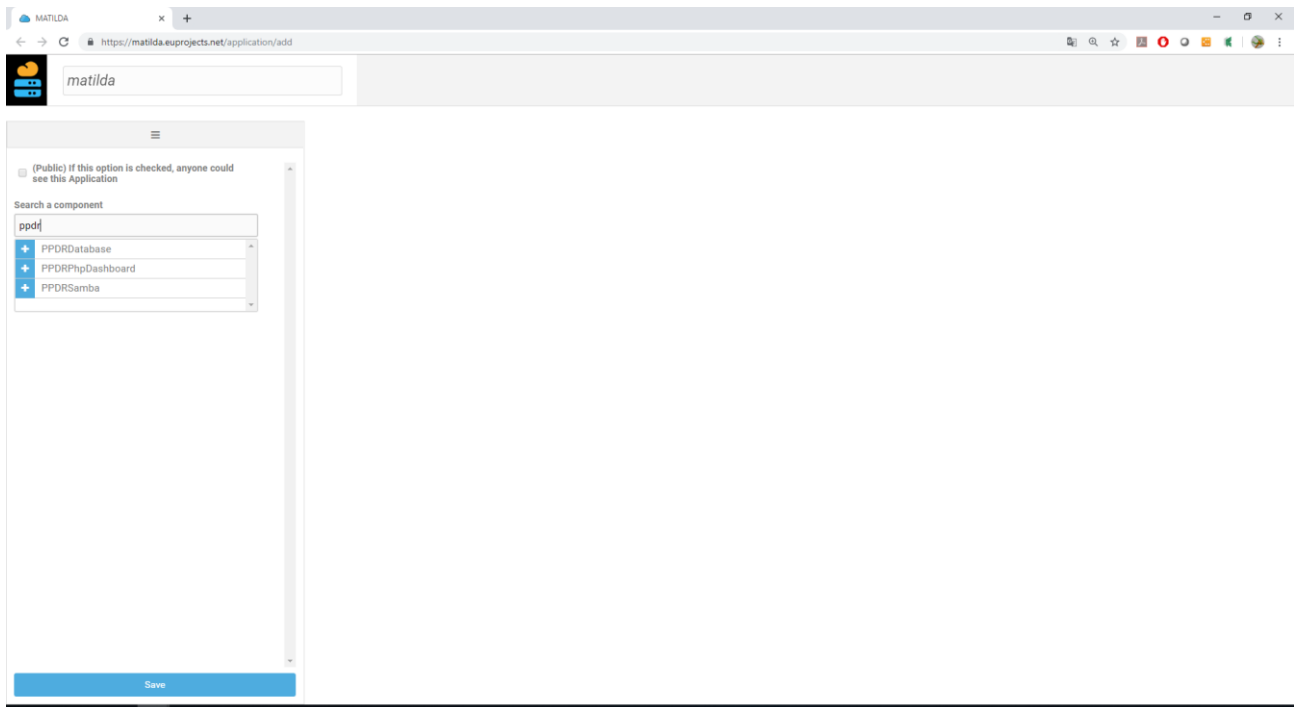


Figure 13: Search Marketplace for Components

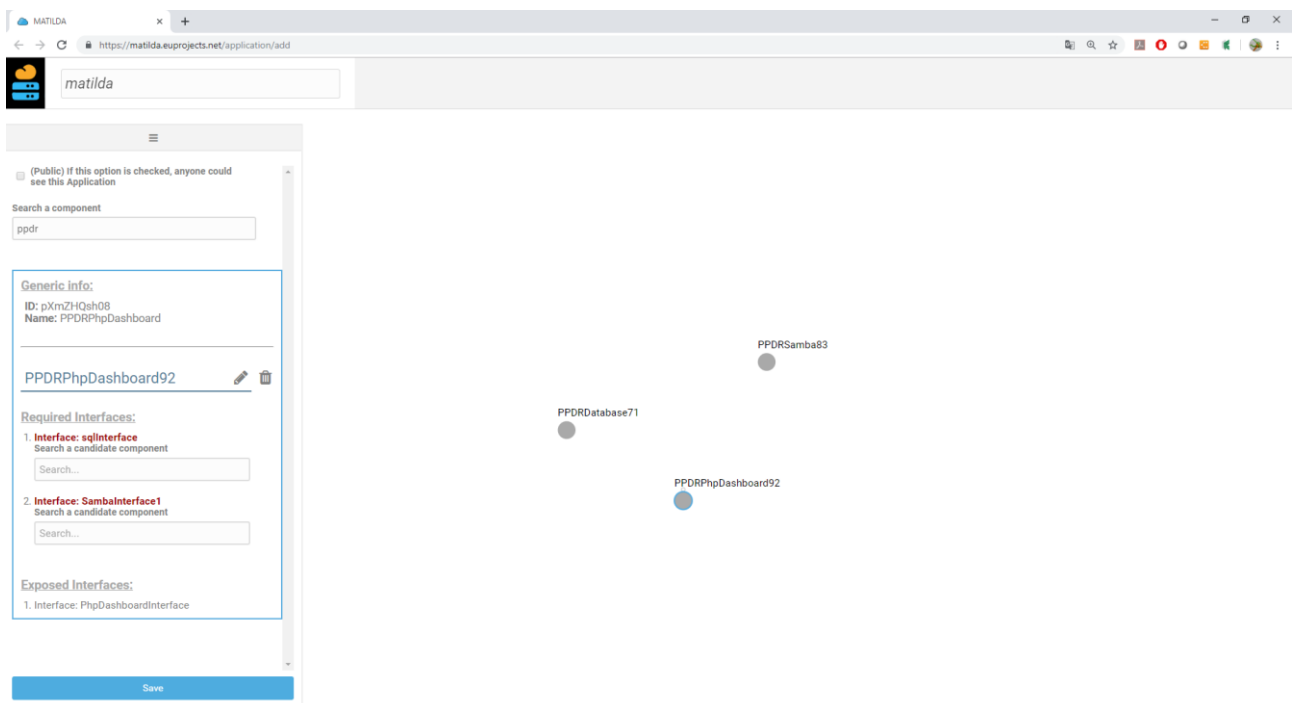


Figure 14: Introspect Component's chain-ability profile

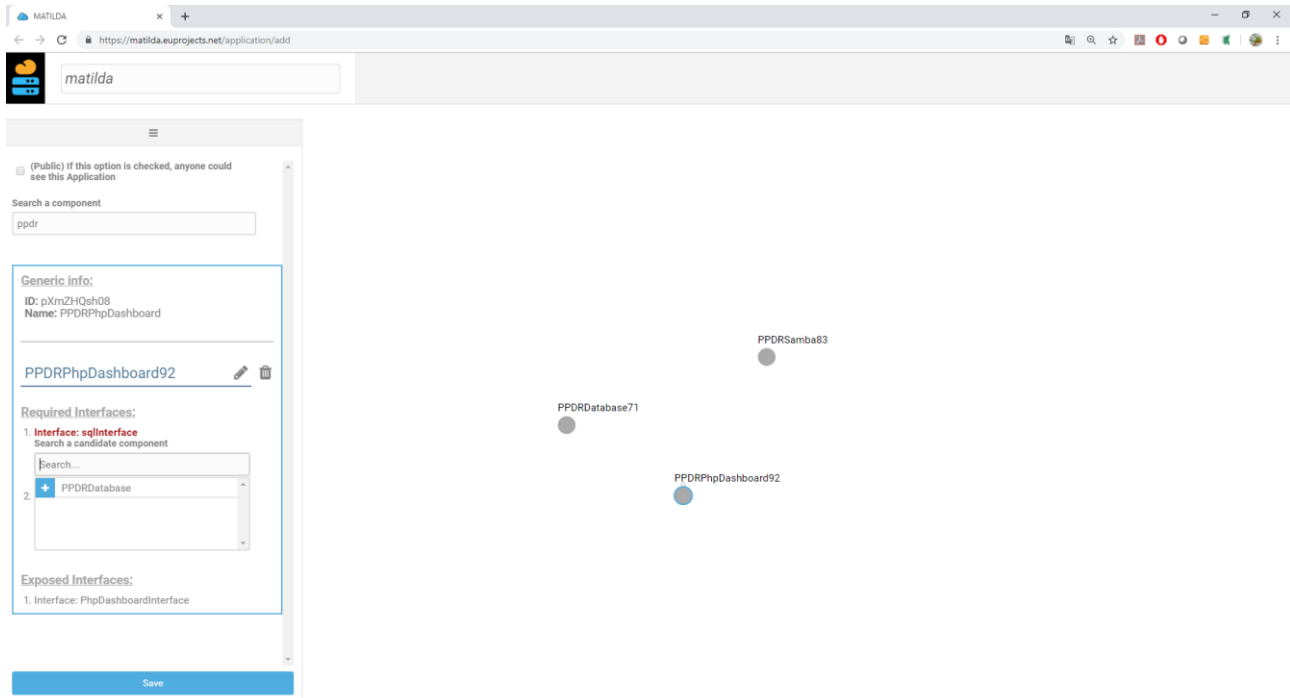


Figure 15: Suggest candidate component that satisfies a specific interface

As the number of available components is increasing, the necessity of an advanced recommender system emerges, in order to facilitate the developer.

Thus, an extra feature in the MATILDA development environment regards a novel recommender system, which is currently under development phase. We want to minimize the human factor in our solution, so techniques that rely on the human input (e.g. ratings), like collaborative filtering techniques and content-based systems, are excluded.

Hence, we use the structure of the graphs in order to determine potential relations between the components. In its simplest form, rather than using ratings that might lead to an incomplete matrix, the graph distances between the nodes are calculated, in order to identify the relations and their strength based on the length of the path and use them instead.

This way the serious cold-start problem is eliminated based on graph characteristics as the basic metric, instead of any optional and doubtful rating. When a graph is stored we then have a set of values for all the components in order to generate recommendation. Subsequently, all the graphs are aggregated, each one of them is represented by an adjacency matrix with distances annotated as values, and a proper factorization technique is applied in order to generate a first list of recommendations per component.

Then, a more advanced and dedicated step in the MATILDA context is to bias the recommendation engine outcomes, based on some critical factors, like the performance estimation of an application component, the components' communication rate and the current infrastructure condition.

Upon selection of a proper component, the user has to connect the two components in order to materialize one link of the graph (see Figure 16).

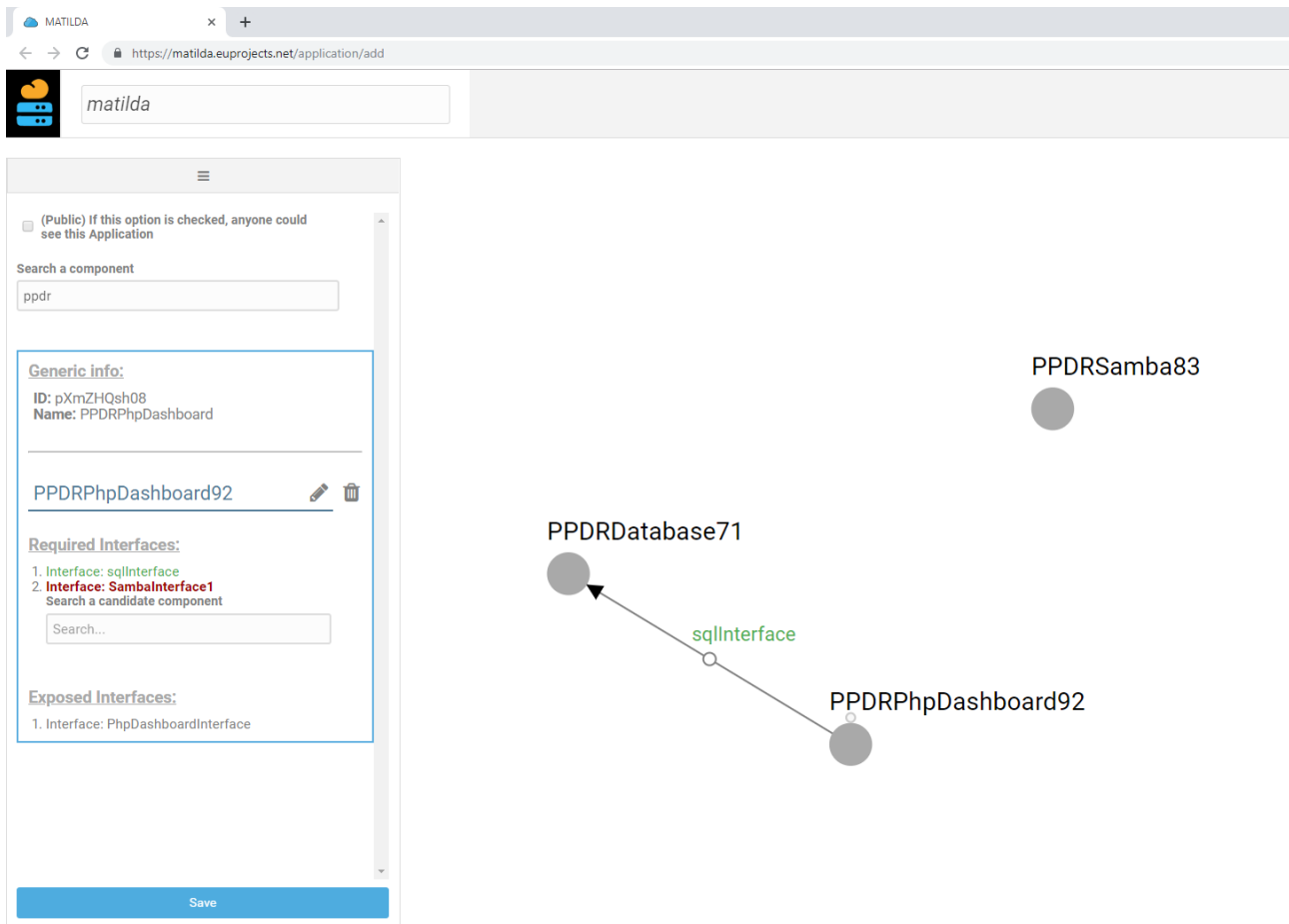


Figure 16: Materialized chain

Using the drag-and-drop functionality complemented with automated validations an always valid application graph is constructed and persisted in the database.

Subsequently, this application graph can be selected in order to initiate a deployment process. In this case, further QoS requirements that have to be guaranteed per virtual link can be declared along with the specification whether we refer to a soft or hard constraint.

The next step regards the deployment of the application graph and the creation of an application graph instance. The deployment is realised over the selected Telco Provider infrastructure. Based on the running instance, signalling and monitoring information is made available, as depicted in Figure 17 and Figure 18.

In the frame of MATILDA special emphasis has been given to the user-experience, since one of the most crucial aspects regarding the success of the platform is usability by all stakeholders i.e. Developer, DevOps and Telco Provider.

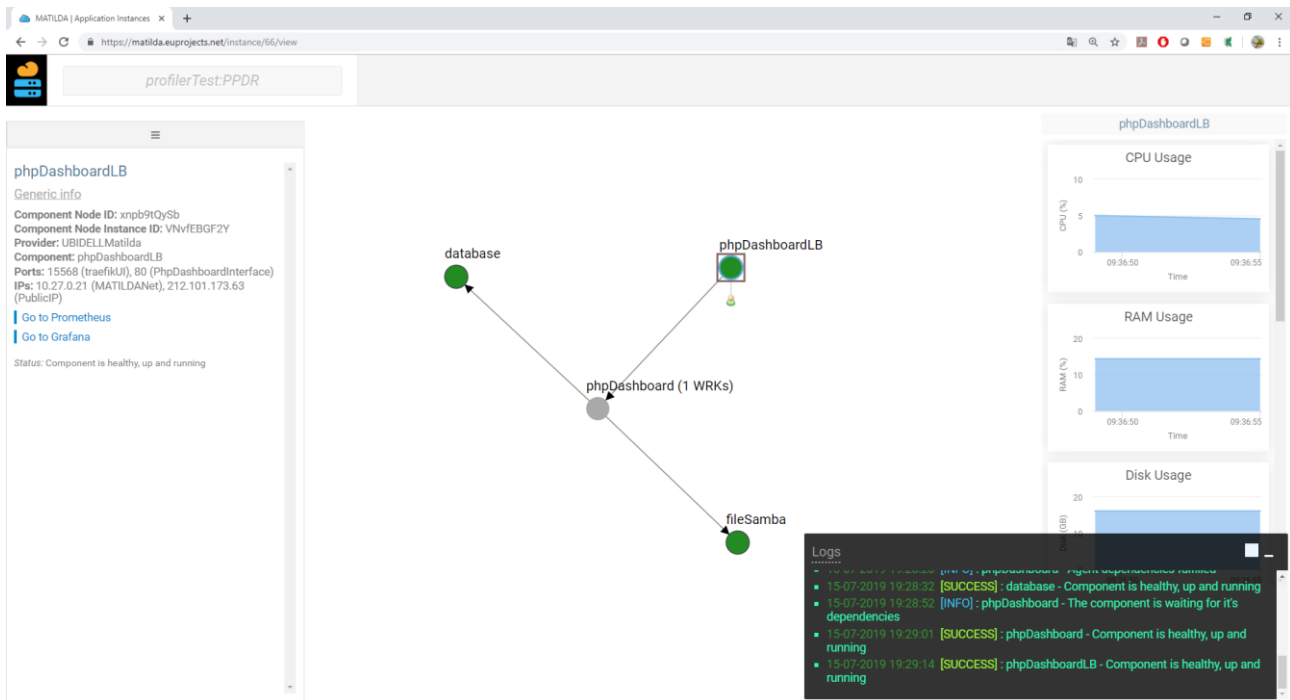


Figure 17: Application Graph Instance and Signalling Information

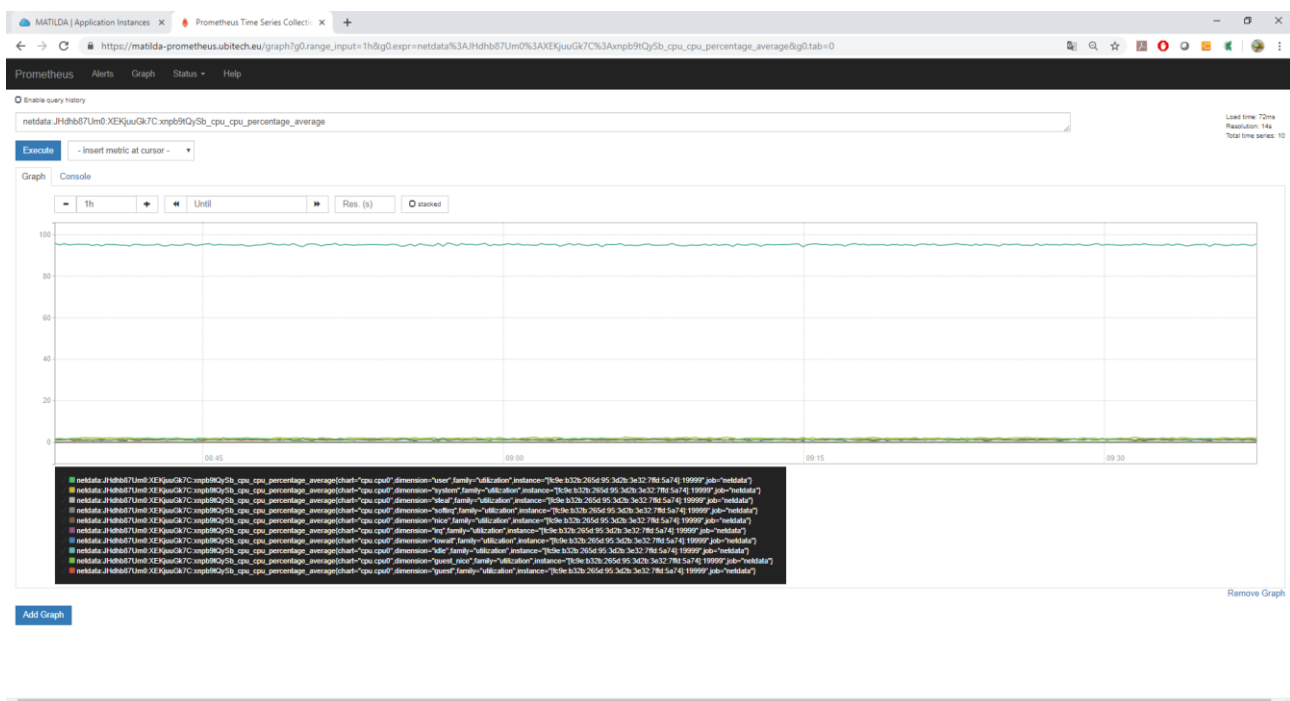


Figure 18: Monitoring data regarding CPU usage of an Application Graph Component

6 Runtime Elasticity and Security Policies Management

Application graphs in MATILDA are accompanied with different policy sets which provide the ability to developers and network engineers to specify the behaviour of graphs (and thus of their applications) both during run time (e.g. when an application is active) and at the application graph deployment stage (e.g. when the graph is to be placed in a 5G slice). The former refers to a one-off configuration which can be applied before the actual instantiation of a graph, and refers to network level policy configurations, while the latter refers to different and interchangeable configuration files, that drive the Policy Enforcement module and that can be altered at will, requiring the reboot of the application service. As such, we distinguish the following two groups of policies.

- Policies that deal with the deployment stage of a service graph and will be handled by the Deployment Manager
- Policies that relate to runtime execution of applications, titled as “5G-Ready Application Policies”, deployed to the engine’s production memory, while the working memory agent is constantly feeding the working memory with new facts.

As part of WP2 and to facilitate the subsequent work of WP3 and of the overall MATILDA technical infrastructure, a simple policy description language for MATILDA, which obeys to the MATILDA Policy Metamodel and is human-readable, has been designed. In MATILDA, policies are formulated as bundles of rules which derive as instantiations of the MATILDA Policy metamodel (described under D1.5 [5]) and are explicitly described by a metamodel language that is used to translate policies to the input requirements of the Prometheus infrastructure² that will be used for measurement extraction. These rules are linked to attributes of the aforementioned models, and each rule consists of a “passive” part that includes expressions, denoting the conditions to be met and an “active” part, which denotes the actions to be executed upon the fulfilment of the conditions. Expressions may regard metrics which can be measured by the Prometheus infrastructure, and that may originate out of different levels of the overall MATILDA stack.

A policy editor is tasked to generate a policy descriptor file, which can be extracted in the policy description language, which is in turn used as input to the Policy Enforcement module and is thus digested by the Prometheus infrastructure. The Policy Editor module is used to create the set of rules which altogether formulate a policy, allowing to declare the different expressions using a user-friendly query-by-example paradigm i.e., the left part of a rule may comprise several conditions or groups of conditions. Complex expressions can be built in order to define the triggering condition of one rule. The right part of a rule can be a list of actions that will be performed in case a condition is met. The policy editor is able to store policies to Drools rules, while it also provides an extraction mechanism in the PromQL language, in order for policies to be directly digestible by the Prometheus infrastructure.

A policy attributes’ monitoring dashboard, that is used to monitor the attributes relevant to a policy in order to allow policy designers to have a better understanding on how runtime conditions behave within the context of a policy. The overall approach takes advantage of the

² <https://prometheus.io>

Grafana analytics interface³, which is an infrastructure that can be used for monitoring purposes and provides graphical outputs of selected metrics for specific data sources. Grafana allows the generation of different dashboards using programming scripts. Furthermore, Grafana is considered one of the most prominent monitoring solutions in the Prometheus infrastructure. As such, it is essential to develop for this purpose a mechanism that allows, given a specific policy, to auto-generate the corresponding Grafana dashboard implementation scripts, so that for each policy a developer may be able to set-up a graphical monitoring interface for that specific policy.

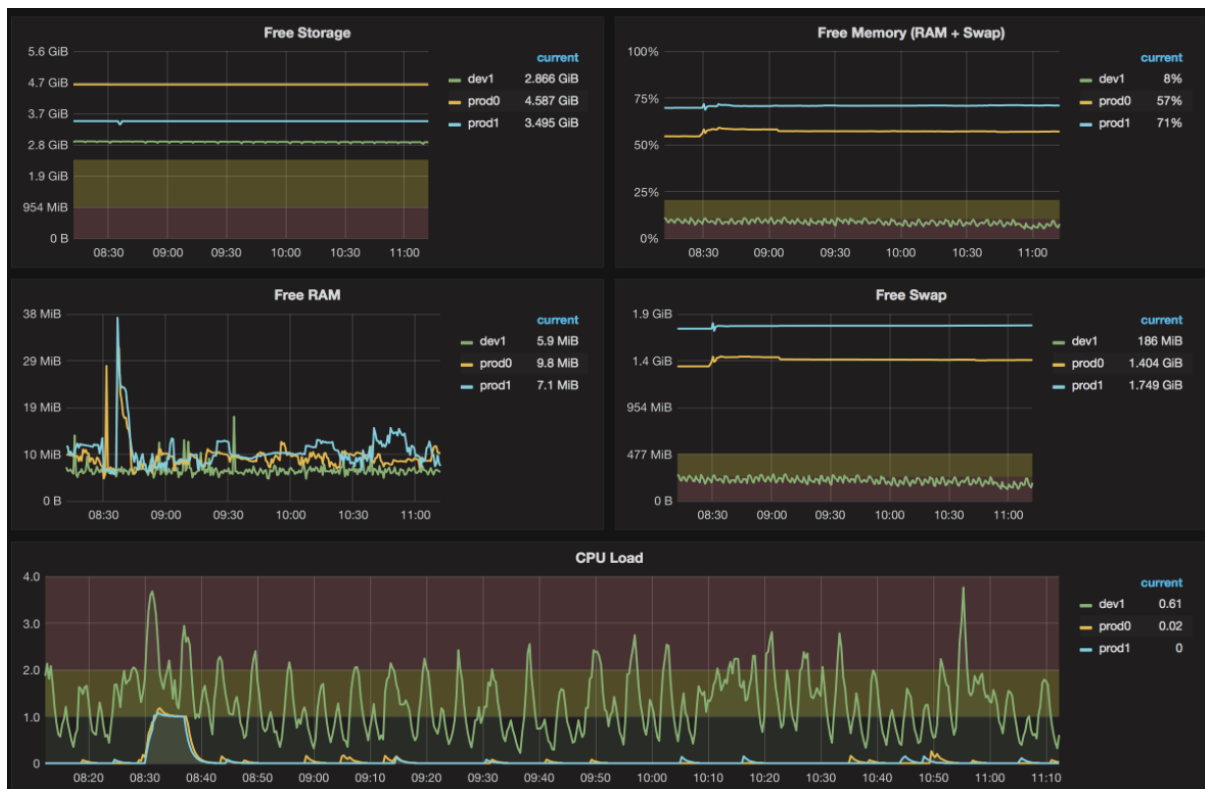


Figure 19: Sample of a Grafana Monitoring Dashboard

Two types of policies are supported; namely, elasticity and security policies. In both cases, policies can be specified upon an application graph instance by selecting the relevant option, as shown in Figure 20. Following, access to the policies editor is provided for specifying expressions, consisting of the conditions and actions part per application component. Per application component, a set of functions can be applied over the monitored data in order to trigger alerts that lead to the enforcement of a policy action. The period for examination of a policy rule, as well as the inertia period for the avoidance of oscillation effects are declared (see Figure 21 and Figure 22). Upon the activation of a policy, the runtime policies' enforcement mechanisms provided by WP3 are applied.

³ <https://grafana.com>

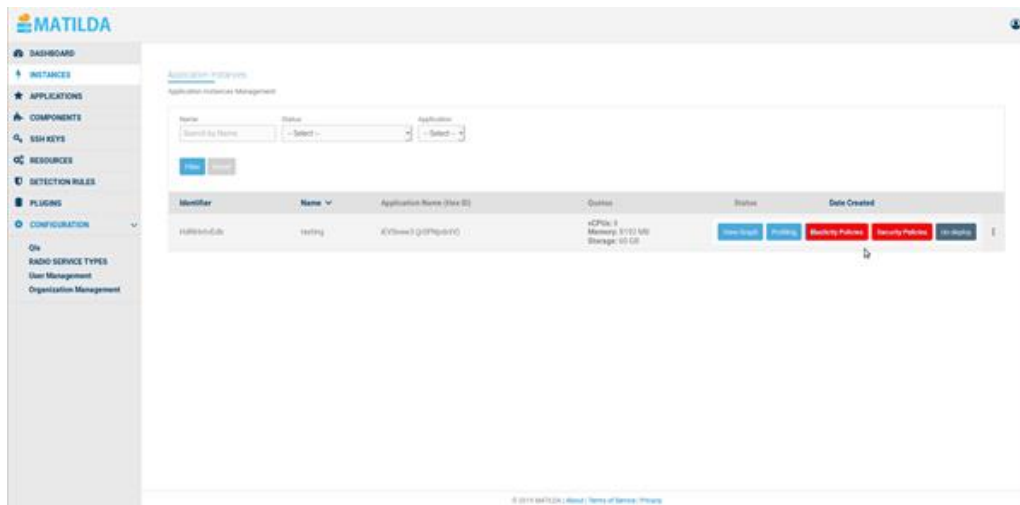


Figure 20: Initiate the specification of Policies through the Application Instances view

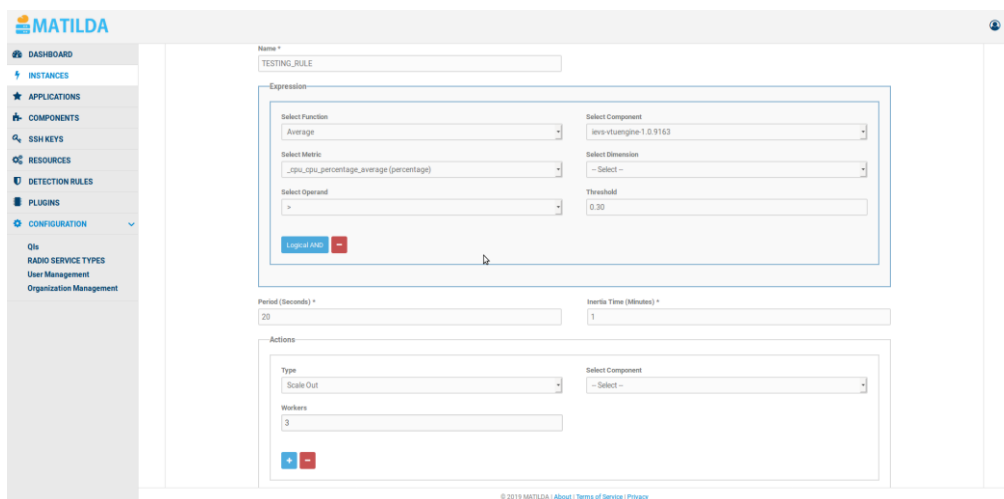


Figure 21: Elasticity Policy Definition

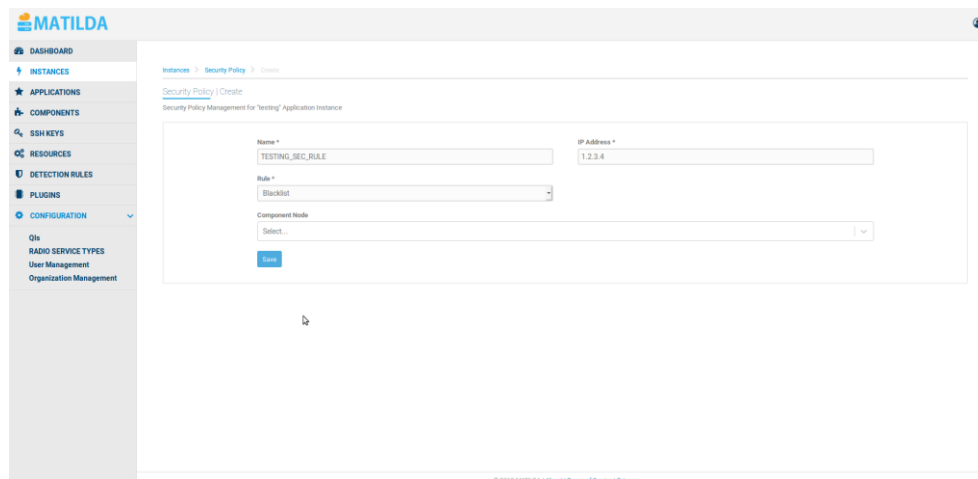


Figure 22: Security Policy Definition

7 Monitoring and Profiling Mechanisms

Monitoring in MATILDA is based on the collection of time series data in a Prometheus monitoring engine. Such data is made available through the UI provided by Prometheus, while custom reporting is also made available through Grafana. In both cases, the objective is to provide a live view of the basic resource usage and application-specific metrics to system administrators.

Moving one step further, profiling mechanisms have been developed aiming at supporting the realization of analysis over the collected time series data. The MATILDA Profiler is implemented based on the adoption of the OpenCPU framework that permits the detaching of the design and implementation of an analysis process from the execution of an analysis over selected time series data. In this way, data scientists can easily design and implement the envisaged analysis scripts and onboard them into the Profiler.

Based on the supported set of analysis scripts, the end user is able to select an algorithm, the set of monitoring metrics to be included in the analysis and the start and end time of the analysis (see Figure 23). At the current phase, the supported algorithms include linear regression, multiple linear regression, clustering, time series decomposition and correlation analysis. An indicative analysis outcome is depicted in Figure 24.

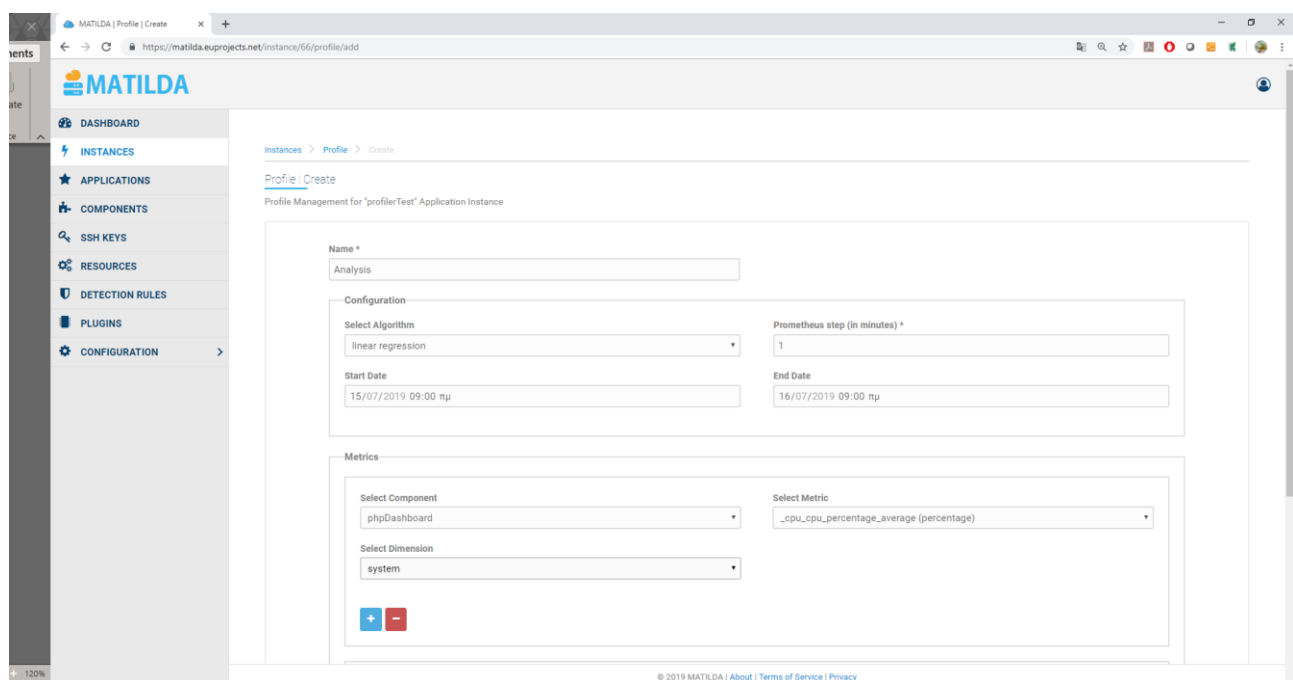


Figure 23: Analysis Configuration

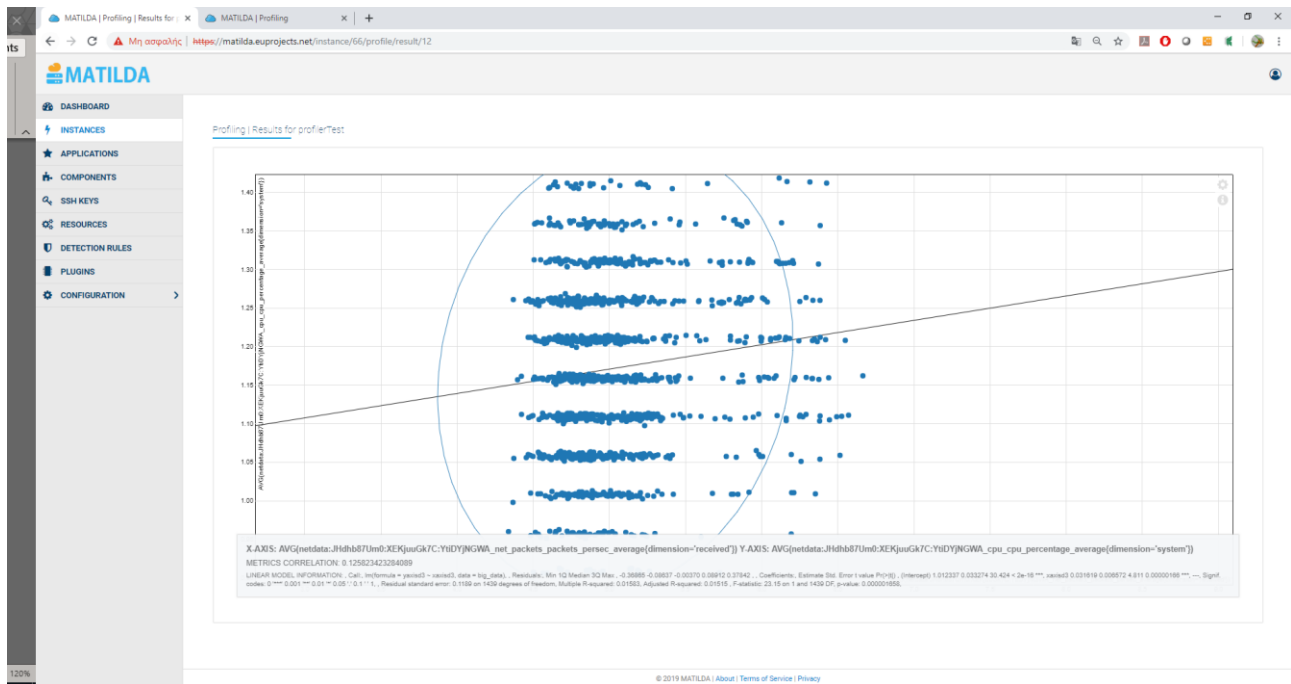


Figure 24: Indicative Analysis Result (Linear Regression model)

8 Network Slice Specification and Management

MATILDA Deliverables D1.1 [1], D1.2 [2] and D1.4 [4] elaborated on the notion of Slice Intent. The slice intent represents the full set of constraints that the MATILDA-enabled telco provider should fulfill/respect in order to create the proper slice that is needed. The aim of MATILDA is to lower the barrier of difficulty regarding the creation of the slice intent. For the sake of comprehension, Figure 25 and Figure 26 represent indicative data structures that formulate the required constraints.

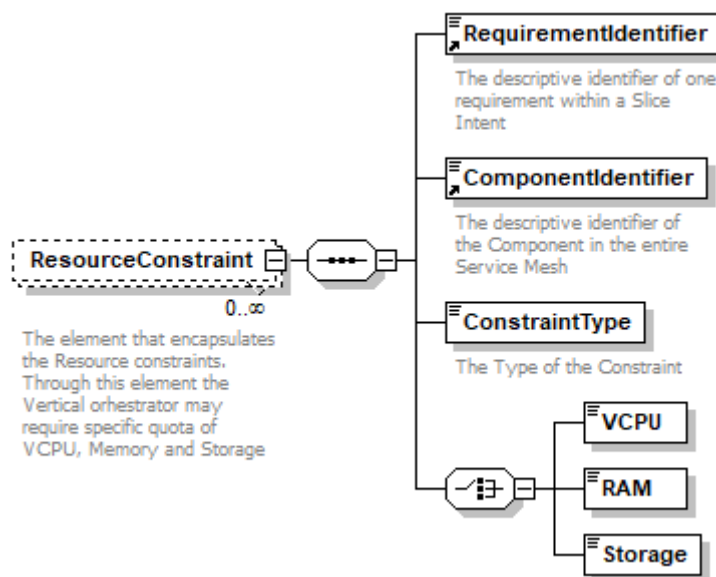


Figure 25: Slice Intent - Resource Constraints element

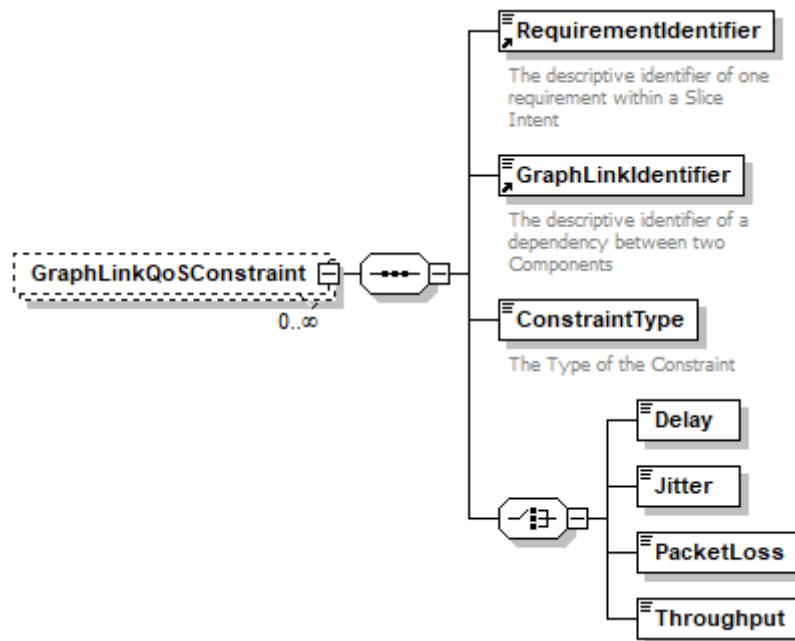


Figure 26: Slice Intent - Graph link QoS constraints element

In an abstract view, a Slice Intent consists of three types of constraints; namely a) resource constraints, b) link constraints and c) access constraints. Each of these constraints can be edited and validated through the developed UI. Let us take as an example the graph of Figure 27, which is a real application graph that relates to audio/visual analytics. It consists of four components, each of which has diverse resource requirements.

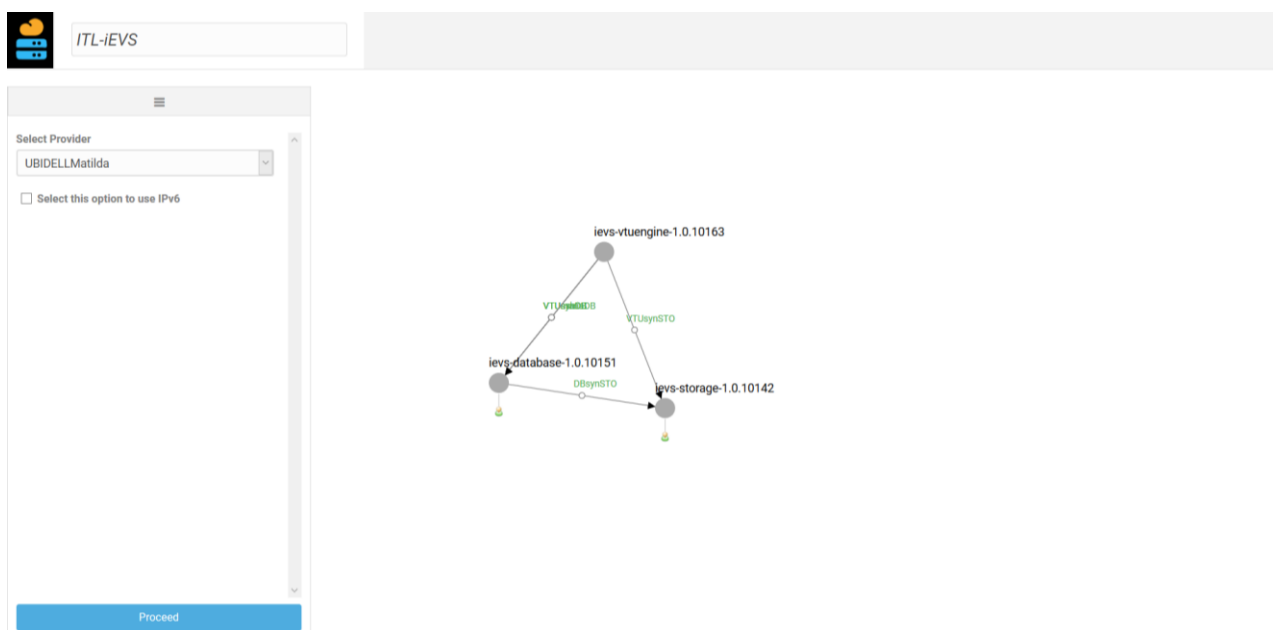
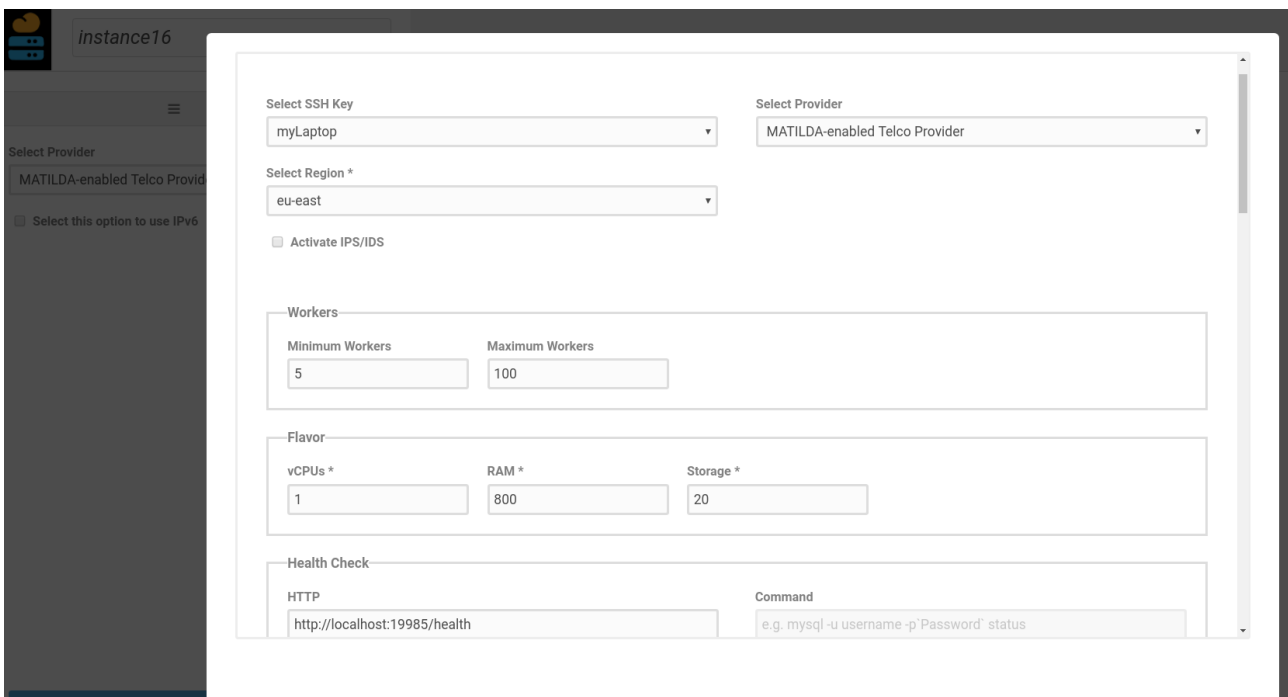


Figure 27: Application Graph that will be used for setting constraints

Prior to the deployment of a application graph to the MATILDA OSS a user can click on each component separately in order to edit the resource constraints, i.e. the min and max vCPUs, memory and storage that the telco provider will allocate to the specific graph. This is illustrated in Figure 28. It should be mentioned that the lowest bound of each component cannot override the execution requirements of the component as declared on the component profile (see Figure 9).

In analogous manner, a DevOps can click on any link of the graph in order to define the graph link constraints that must be fulfilled by the telco provider. Indicatively, Figure 29 depicts how link constraints are declared i.e. minimum throughput, maximum packet loss, maximum delay and maximum jitter. As is depicted, some constraints are considered soft and others hard. The MATILDA orchestrator will transform these declarations to a formal optimization problem that has to be solved prior to deployment.

Finally, when a user clicks on an Access interface the ability to define 5G QCI parameters is provided, as depicted in Figure 30. The user can select the type of service (NBIoT, URLLC, MB), the GPR parameters and the minimum/maximum requested bandwidth.



instance16

Select SSH Key: myLaptop

Select Provider: MATILDA-enabled Telco Provider

Select Region *: eu-east

☐ Activate IPS/IDS

Workers

Minimum Workers: 5

Maximum Workers: 100

Flavor

vCPUs *: 1

RAM *: 800

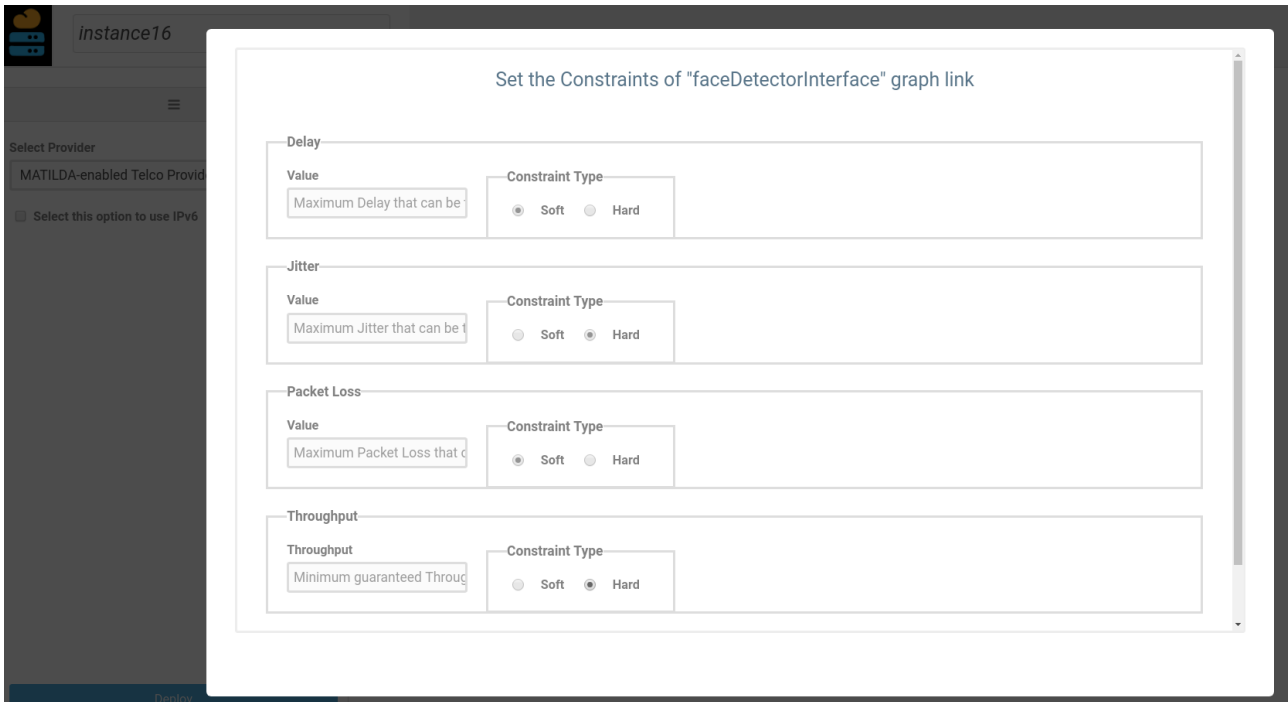
Storage *: 20

Health Check

HTTP: http://localhost:19985/health

Command: e.g. mysql -u username -p 'Password' status

Figure 28: Declaring Resource Constraints



Set the Constraints of "faceDetectorInterface" graph link

Delay

Value:

Constraint Type: ☒ Soft ☐ Hard

Jitter

Value:

Constraint Type: ☐ Soft ☒ Hard

Packet Loss

Value:

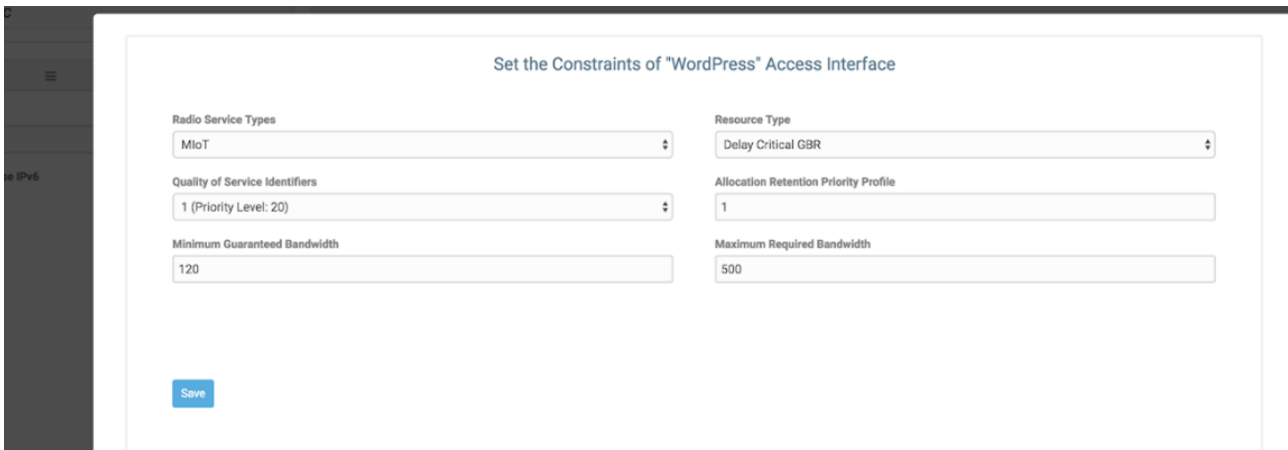
Constraint Type: ☒ Soft ☐ Hard

Throughput

Throughput:

Constraint Type: ☐ Soft ☒ Hard

Figure 29: Declaring Link Constraints



Set the Constraints of "WordPress" Access Interface

Radio Service Types:

Quality of Service Identifiers:

Minimum Guaranteed Bandwidth:

Resource Type:

Allocation Retention Priority Profile:

Maximum Required Bandwidth:

Figure 30: Declaring Access Constraints

The problem which will be formulated based on all three types of constraints will be transformed into a constraint satisfaction problem that will be loaded in a solver. The solver will generate a solution which will be communicated to the orchestrator through appropriate OSS signaling. The solution will consist of the selection of proper VIMs along with the instructions of where each component must be placed (Figure 31).

The following figures 32 and 33 present an example of a created slice intent and materialized slice.

MATILDA OSS

Slice: 5b27a0b690d7ba1063b649c2, Application Instance: Signal

[← Go back](#)

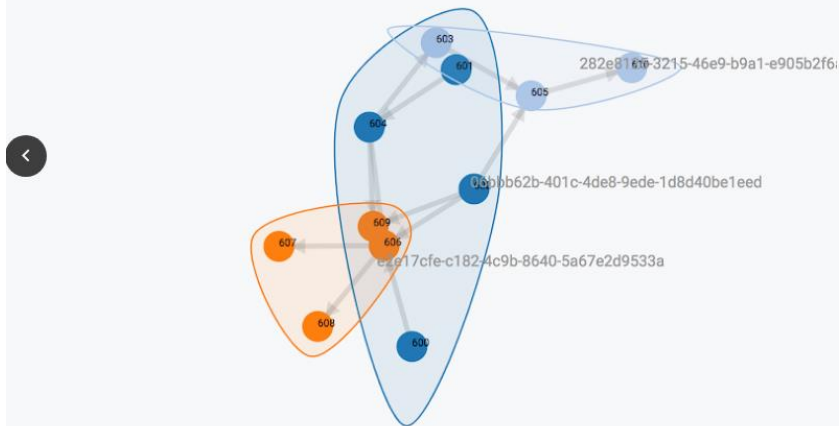


Figure 31: Materialized Slice within the OSS

```
{
  "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"
  }],
  "graphLinkNodes": [{
    "graphLinkNodeID": "544",
    "fromComponentNodeInstanceID": "587",
    "toComponentNodeInstanceID": "581",
    "type": "CORE"
  }],
  "dateCreated": "Jun 13, 2018 12:02:04 PM"
}
```

Figure 32: Part of Slice Intent

```
{
  "applicationInstanceID": "580",
  "vimDescriptors": [{
    "vimID": "115e1625-da50-4c7d-ba08-d213f6662205",
    "domain": "default",
    "project": "maestro",
    "username": "maestro",
    "password": "!maestro!",
    "endpoint": "http://192.168.3.253:5000/v3/"
  }],
  "componentPlacements": [{
    "vimID": "115e1625-da50-4c7d-ba08-d213f6662205",
    "componentNodeInstanceID": "581",
    "attachmentPoints": [{
      "graphLinkNodeID": "544",
      "attachmentPointIdentifier": "26ee5637-316d-48bd-bcb6-183dcc43444"
    }]
  }, {
    "vimID": "115e1625-da50-4c7d-ba08-d213f6662205",
    "componentNodeInstanceID": "587",
    "attachmentPoints": [{
      "graphLinkNodeID": "544",
      "attachmentPointIdentifier": "33ae68d9-4543-46eb-be00-653e1288af27"
    }]
  }],
  "constraintSatisfactions": [{
    "constraintID": "591",
    "satisfied": true,
    "constraintType": "HARD"
  }],
  "dateCreated": "Jun 13, 2018 12:02:14 PM"
}
```

Figure 33: Part of Slice Descriptor

9 Conclusions

This document presented the current implementation status of the MATILDA environment that is used by developers and DevOps users in order to manage their Components, Application Graphs and Slice Intents. The added value of such an environment is to lower the entry-barrier of MATILDA stakeholders. This barrier exists because of the modelling complexity that is needed in order to create formal (i.e. normative) data structures that represent the concepts above. In the frame of D1.2, D1.3 [3], D1.4 [4] and D1.5 [5] formal XSD models have been introduced and are ever since maintained. The models act as a cornerstone of integration among the various mechanisms that are being developed. Furthermore, they act as a reference model which will be used during standardization activities. However, these formal models introduce a relative significant overhead to the potential adopters.

Each of the modelling artefacts entails different characteristics that have to be taken under consideration during the development of an assistive environment. Indicatively, the Components include information regarding their distribution parameters, their exposed and required interfaces, their scalability profile, etc. An Application Graph must have valid declaration of links and usage of compatible components. Finally, a slice intent must include all three level of constraints, i.e. component-level, graph-link and access constraints.

Based on the specificities of each model a proper environment has been implemented using state of the art user interface technology. The actual goal was to make an **intuitive** environment which will be **less error-prone** based on the **multiple validators** that have been developed. Hence, the registration of a component in a proper way (i.e. with syntactic and logical validation), the design of a correct application graph (using automated compatibility validation and proper component suggestion) and the formulation of a constraint satisfaction problem that lies beneath the slice intent without bothering with any language semantics.

The outcome of this effort was the development of a homogeneous environment that tries to unify the perspective of all stakeholders. As was mentioned in the introduction, even if the overall environment is released in a full and mature version, the development of the environment is still an ongoing work. Several changes may be proposed based on the tests that will be conducted by the end users. The actions that are performed after the release of the first version of the development environment include the introduction of the Profiling mechanism, the integration of a set of monitoring mechanisms and the upgrade of the user interface.

References

- [1] MATILDA Deliverable D1.1 - MATILDA Framework and Reference Architecture
- [2] MATILDA Deliverable D1.2 - Chainable Application Component & 5G-ready Application Graph Metamodel
- [3] MATILDA Deliverable D1.3 - VNF/PNF & VNF Forwarding Graph Metamodel
- [4] MATILDA Deliverable D1.4 - Network-aware Application Graph Metamodel
- [5] MATILDA Deliverable D1.5 - Deployment and Runtime Policy Metamodel
- [6] MATILDA Deliverable D3.1 - Intelligent Orchestration Mechanisms – First Release
- [7] MATILDA Deliverable D2.1 - 5G-Ready Applications and Network Services Development Environment and Marketplace – First Release