



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

DELIVERABLE D1.2

CHAINABLE APPLICATION COMPONENT & 5G-READY APPLICATION GRAPH METAMODEL

Due Date of Delivery:	M9 <i>Mx</i> (28/02/2018 <i>dd/mm/yyyy</i>)
Actual Date of Delivery:	05/03/2018 <i>dd/mm/yyyy</i>
Workpackage:	WP1 – MATILDA Reference Architecture, Conceptualization and Use Cases
Type of the Deliverable:	OTHER
Dissemination level:	PU
Editors:	UPRC, UBITECH
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:

30 months

Start Date:

01/06/2017

Project Coordinator:

Name:

Franco Davoli

Phone:

+39 010 353 2732

Fax:

+39 010 353 2154

e-mail:

franco.davoli@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

ATOS	ATOS Spain SA
Aurora Ramos, Javier Melian	
UBITECH	GIOUMPITEK Meleti Schediasmos Ylopoiisi kai Polisi Ergon Pliroforikis EPE
Panagiotis Gouvas , Anastasios Zafeiropoulos	
UNIVBRIS	UNIVERSITY OF BRISTOL
Anderson Bravalheri, Dimitrios Gkounis, Reza Nejabati, Dimitra Simeonidou	
UPRC	UNIVERSITY OF PIRAEUS RESEARCH CENTER
Dimosthenis Kyriazis, Chrysostomos Symvoulidis, Ilias Tsoumas	
INC	Incelligent
Panagiotis Demestichas, Kostas Tsagkaris, Nikos Stasinopoulos, Athina Ropodi, Stavroula Vassaki, Aristotelis Margaritis, Dimitris Cardaris, Marinos Galiatsatos	

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 © 2018 the MATILDA Consortium. All rights reserved.

The MATILDA Consortium consists of:

CONSORZIO NAZIONALE INTERUNIVERSITARIO PER LE TELECOMUNICAZIONI

ATOS SPAIN SA (ATOS)

ERICSSON TELECOMUNICAZIONI (ERICSSON)

INTRASOFT INTERNATIONAL SA (INTRA)

COSMOTE KINITES TILEPIKOINONIES AE (COSM)

ORANGE ROMANIA SA (ORO)

EXXPERTSYSTEMS GMBH (EXXPERT)

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

INTERNET INSTITUTE, COMMUNICATIONS SOLUTIONS AND CONSULTING LTD (ININ)

INCELLIGENT IDIOTIKI KEFALAIOUCHIKI ETAIREIA (INC)

SUITE5 DATA INTELLIGENCE SOLUTIONS LIMITED (SUITE5)

NATIONAL CENTER FOR SCIENTIFIC RESEARCH “DEMOKRITOS” (NCSR)

UNIVERSITY OF BRISTOL (UNIVBRIS)

AALTO-KORKEAKOULUSAATIO (AALTO)

UNIVERSITY OF PIRAEUS RESEARCH CENTER (UPRC)

ITALTEL SPA (ITL)

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

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
1 EXECUTIVE SUMMARY	5
2 INTRODUCTION	6
2.1 SCOPE OF THE DELIVERABLE	6
2.2 STRUCTURE OF THE DOCUMENT	6
3 BASELINE TECHNOLOGIES	7
3.1 EXISTING SOLUTIONS	7
3.2 MODELLING LANGUAGES	9
3.3 SERVICE MESH	10
4 OVERVIEW OF THE METAMODEL	12
4.1 APPLICATION COMPONENT PART	14
4.2 APPLICATION GRAPH / SERVICE MESH PART	19
5 SUPPORT MECHANISMS SUITE	21
5.1 OVERALL ARCHITECTURE	21
5.2 SUPPORTING MECHANISMS	21
6 CONCLUSIONS	24
REFERENCES	25
APPENDIX 1: CHAINABLE COMPONENT & 5G-READY APPLICATION GRAPH METAMODEL (V1.0) DOCUMENTATION	26

1 Executive Summary

The scope of MATILDA is to deliver a Holistic, Innovative Framework for Design, Development and Orchestration of 5G-ready Applications and Network Services over Sliced Programmable Infrastructure. A 5G-ready Application consists of chainable components (i.e. micro-services). The interaction between the components can be depicted by a directed acyclic graph which refers to a 5G-ready application graph and will be implemented by exploiting the Service Mesh paradigm. In this document we present the conceptualization and description of these two fundamental entities through the corresponding models: the chainable component and the 5G-ready application graph. The key outcome is the metamodel of the two aforementioned entities which has been completed and released as planned.

The chainable application component part of the metamodel describes and verifies (with the aid of some support mechanisms which are being developed) the following: (i) the required information for the proper deployment and execution of the components (e.g. resource requirements), and (ii) the alignment with a set of rules in terms of QoS requirements, elasticity, cloud-nativeness, etc. The application graph (service mesh) part of the metamodel is the collection of the bindings among the exposed and required interfaces per chainable component. Both at component and at graph level, the denotation of a set of computing, memory, storage and of course network requirements are being supported in correlation with the corresponding slice intent metamodel.

Moreover, it should be noted that the development of the MATILDA metamodels is a continuous and iterative process. To this end, this deliverable provides the first version of the chainable component and the 5G-ready application graph metamodel. The documented and developed metamodel will evolve and will be extended based on the feedback of the early prototypes of the project. Regarding the implementation of the metamodel, the XML schema notation has been used. The updated version of the metamodels will be made available through the MATILDA web site (<http://www.matilda-5g.eu>). For the sake of completeness, the documentation of the existing normative format is provided in Appendix 1.

2 Introduction

In the MATILDA context the 5G-ready application is the highest-level entity from a top-down perspective. This is the end-point of 5G environments with which the users will interact. A 5G-ready application is reflected in an application graph and is implemented by a service mesh, which consists of several chainable components, the most granular entity of it. This metamodel describes, conceptualizes, verifies and annotates with requirements and performance estimations each component of the application and the overall application graph in order to provide the required information towards the management and orchestration mechanisms of 5G platforms.

2.1 *Scope of the Deliverable*

The aim of this deliverable is to develop and describe the chainable component and the 5G-ready application graph metamodel. This metamodel is an inseparable node in the MATILDA's metamodel chain. It is "placed" in the highest layer of the metamodel stack which come to "trigger" the fulfilment of the slice intent metamodel after the initiation of the former. Subsequently a slice intent instance-model provides the information to the telco providers regarding the compute and network materialization of the slice and consequently the realization of 5G-ready application. The key aspects documented in this deliverable are the following: (i) the research on the existing modelling solutions and tools, (ii) the identification of a proper solution to address the needs of a 5G application, (iii) the definition of the chainable component and the 5G-ready application graph metamodel, (iv) the design and specification of a set of supporting mechanisms for the efficient verification of metamodel.

2.2 *Structure of the Document*

Taking under consideration the scope of the current deliverable, this report has been structured as follows: Chapter 3 provides a brief view of various existing solutions regarding modelling of micro-services and service graph at industry and academic levels. In parallel, a comparison of key modelling languages and tools (serialization, deserialization, etc.) is presented, which concludes to the specific technologies that will be used. To this end, the final section of this chapter is devoted to the presentation of the service mesh approach, which has been adopted for the interconnection of the components and the compilation of the application graphs.

Chapter 4 focuses on the analysis of the MATILDA chainable component and the 5G-ready application graph metamodel. This chapter is divided in two sub-sections. The first sub-section describes the most granular executable unit of a 5G-ready application: the component. Several components can be combined towards the realization of a Service Mesh. The second sub-section presents the metamodel of the application graph (as a service mesh) and a part of slice intent metamodel, which is related directly with the application graph.

Chapter 5 provides an overview of the Supporting Mechanisms Suite, which has been designed for the efficient verification of the aforementioned metamodel. Specifically, in this chapter the overall architecture of the suite and a short design specification of each mechanism are presented.

For the sake of completeness, Chapter 6 summarizes the results and the innovations of this deliverable and the concrete extensions that will be provided in the frame of the next scheduled releases. The actual formal normative model is provided in Appendix 1.

3 Baseline Technologies

3.1 Existing Solutions

The scope of this sub-section is to provide a generic view of some of the current trends regarding each component/service and the entire applications/services graph modelling schemas that are going to be considered towards the definition of the MATILDA metamodels.

- **NodeRED**

It is a project of the JS Foundation. NodeRED [nodeRED] is a tool for wiring together hardware devices, APIs and online services. It tackles the problem of multi-connections in IoT. It provides a browser-based editor that makes it easy to connect together flows using the wide range of nodes in the palette that can be deployed to its runtime in a single-click. The flows created in Node-RED are stored using JSON, which can be easily shared with others. A built-in library allows one to save useful nodes and flows for reuse and an online flow library allows to share one's work with others. Focusing mostly on modelling aspects of NodeRED, it follows a very simple and light node and flow model schema. Especially, the service-model is addressed as “node” and it is described by two sets of properties:

- **core-properties:** used by the runtime/editor for the basic node functionality;

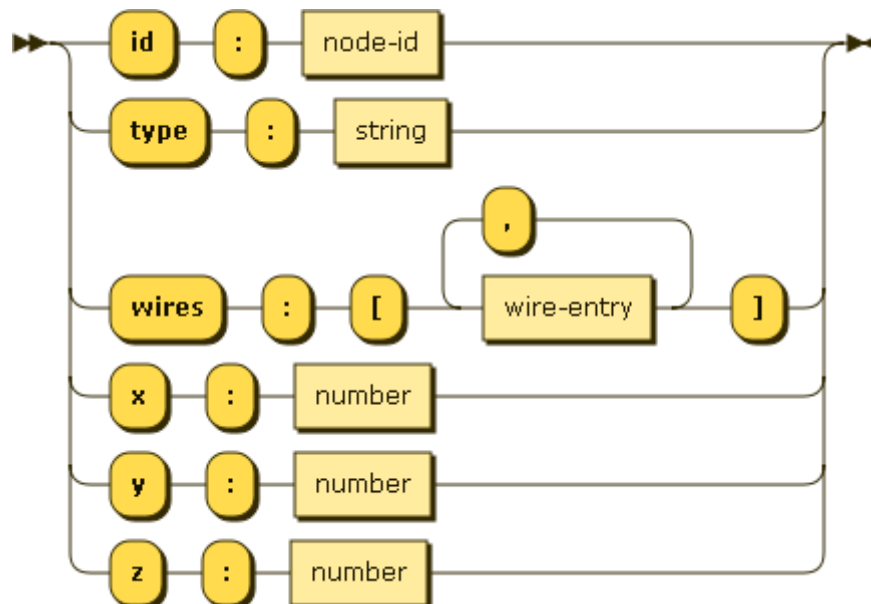


Figure 1: NodeRED core properties

- **type-properties:** created by node-developer, where each node type can contain properties that are specific to it. In addition, a custom node could declare its own

property to capture that information, and its runtime implementation knows what to do with it.

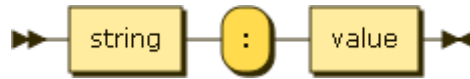


Figure 2: NodeRED type properties

To sum up, the application-graph model is addressed as “flow”, and it is represented by a Javascript array of objects. Each object is a node, with a set of core properties, and a set of type-specific properties. Furthermore, NodeRED checks the proper deployment of the flow, but it does not actually deal with nodes’ orchestration, deployment, scaling, and management. Finally, it does not allow for the specification of (network) parameters regarding links between different components.

- **Juju**

Juju [Juju], developed by Canonical, is a framework that can be used to model, manage and scale services in the cloud. It contains some interaction tools such as a command line and a graphical user interface and is a solid solution that can reduce the workload for deployment and configuration. Thus, through these tools a DevOps user can easily embed a service or a web of services on top of multiple IaaS providers (e.g. OpenStack).

The service metamodel of Juju is addressed as “charm” and contains a set of elements that are required in order, for a specific service to be composable and orchestratable. The service graph metamodel is addressed as “bundle” and it is a web of charms. Anybody can deploy a predefined charm or a bundle and use them. Both of them are described by some YAML files, and someone can moderate them with some commands called “hooks”. In JUJU documentation, a strict list of commands per charm is provided, so that anybody can use them to configure it. However, the Juju platform was not built to address more specific network quality of service requirements and constraints.

- **DOCKER Compose**

Developed by Docker, Docker Compose [Docker Compose] is a tool for defining and running complex, multi-container applications with Docker. With Compose, a multi-container application can be defined in a file and the application is deployed and executed. A Dockerfile describes units (each docker-container is considered a granular unit), in which the possible user (i.e., Devops) could define what the container needs.

- **PUPPET**

It is categorized in the middleware level and aims to model, install and deploy infrastructure’s applications and services [Puppet]. It also tracks down the dependencies between them. Puppet uses a Domain Specific Language (DSL) for the modelling. This specific language adds more complexity in Puppet, but on the other hand it becomes more consistent and offers a deeper layer of valid configurability on the spot.

- **ARCADIA Context Model**

This model is an outcome of the ARCADIA [ARCADIA-D.2.2] EU-funded research project. It deals with the modelling of services regarding highly distributed applications. With a focus on component and service-graph models, the ARCADIA Component Model represents the most

granular executable unit of an ARCADIA application. A set of interconnected components produces a service graph. Highly Distributed Applications (HDAs) are practically instantiations of a complex service graph. Each component is described through several properties and all this information is encapsulated in a XML file that is generated by a specific XML Schema Definition (XSD).

As already described, many ARCADIA Component Models can be combined in order to create one ARCADIA Service Graph Model, which is practically a directed graph. Finally, the service graph model encapsulates the description of each component, as well as a description for each virtual link, accompanied with information regarding monitoring metrics that refer to the whole service graph.

3.2 Modelling Languages

YAML

YAML (YAML Ain't Markup Language) is a human-readable language used for data serialization. It is mainly used for file configuration, but it can also be used in various applications where data is being either stored or transmitted. It uses a more minimal syntax than XML making it easier for a human to read. YAML is considered to be a superset of JSON [YAML] since it uses both Python-style indentation to indicate nesting, but also uses lists and maps. YAML has built-in variables (scalars) such as integers, floats, strings, arrays and lists, but custom data types are also allowed. YAML does not have attributes like the ones found in XML, but it does support extensible type declarations (including class types for objects).

XML

XML (Extensible Markup Language) is a markup language [XML] mainly used for the definition of rules for encoding documents in a format that is human-readable. It is broadly used in a Services Oriented Architecture (SOA) for the communication between various systems by the exchange of XML messages. The standardization of these messages is done through XSD (XML Schema). XSD's is a powerful schema, whose format is similar to XML, allowing the user to create constraints in a more detailed way on the structure of an XML document with the support of its rich data typing system.

JSON

JSON (JavaScript Object Notation) is a file format used mainly for the asynchronous communication between a browser and the server, replacing XML in some systems [JSON]. JSON uses attribute-value pairs to demonstrate the data as well as array data types. JSON Schema is used to define the structure of JSON data, to validate the data as well as for documentation purposes in a format that is similar to a JSON file. The JSON Schema is based on XML Schema (XSD) but is JSON based.

AVRO

Avro [AVRO] is a data serialization framework developed by Apache. Avro provides rich data structures and a compact, fast, binary data format supporting Remote Procedure Call (RPC). Since it is developed within Hadoop a container file to store persistent data can be used too. The integration with dynamic languages is quite simple, since code generation is not required to read or write data files nor to use or implement RPC protocols. Code generation as

an optional optimization, are only worth implementing for statically typed languages. Avro has built-in schema capabilities for the description of the documents, which is in JSON format supporting both primitive types (int, float, string, boolean, etc.) and complex types (arrays, map, records, etc.) as well.

Protocol Buffers

Protocol Buffers is Google's language-neutral extensible mechanism for serializing structured data [ProtoBuf]. It is primarily used for the communication between programs over a wire or for storing data. Protocol Buffers was designed to be a more minimal as well as faster than XML. The structure of a proto message consists of unique numbered fields, where each field has a name and a type. The types supported by Protocol Buffers are integers, floats, strings, Booleans, bytes or other proto files allowing a file to be hierarchal structured. These messages are then compiled, generating data access class for the programming language. These data access classes provide simple accessors for each field of the message (getters/setters, etc.). It does lack however a way to specify a schema within a proto file. Officially it does support an ASCII serialization format, but that way the forward- and backward-compatibility making it a bad choice for applications other than debugging.

XML (XSD) has been selected for the modelling part (to enable validations through XSD) and ProtoBuf for the internal communication between the service components. The purpose of selecting XSD for the metamodel of component and application graph is that it combines the ability to express constraints (multiplicity, optional, enumerations) with the ability to export some human-readable documentation of the metamodel. The metamodels should follow a "user-optimized" serialization format which is less GUI-dependent. Thus, the usage of Protocol Buffers which is aiming at the creation of dev-language agnostic message passing among the developed components.

3.3 Service Mesh

One of the biggest challenges in the 5G evolution lies in the way we properly introduce services and the Service Oriented Architectures (SOAs) in the 5G environments. MATILDA overcomes this issue using the new MASA – Mesh Applications and Service Architecture. The high-level difference of MASA from SOA is summed up on this *"The mesh app and service architecture (MASA) is a multichannel solution architecture that leverages cloud and serverless computing, containers and microservices as well as APIs and events to deliver modular, flexible and dynamic solutions. Solutions ultimately support multiple users in multiple roles using multiple devices and communicating over multiple networks."* [Gartner-2016]

So, in the MATILDA context we implement each 5G-ready application as a Service Mesh to exploit the 5G cutting edge technologies. More specifically, *"a service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware."* [Morgan-2017]. This infrastructure layer comes to tackle a deep issue regarding the real properties of network. Specifically, Peter Deutsch and his team at Sun describe a list of "fallacies", a set of the opposite of rules about distributed computing that people often forget based on the assumptions that they were making for the underlying network. These fallacies include the assumption that a) the **network is reliable**, b) **latency is**

zero, c) **bandwidth is infinite**, d) the **network is secure**, e) **topology does not change**, f) **there is one administrator**, g) **transport cost is zero** and h) the **network is homogeneous**. The removal of these assumptions requires the addition and the satisfaction of many hard requirements [Calçado -2017].

The Service Mesh concept is implemented in MATILDA by using a sidecar pattern, where the functionality of each of the components in the mesh is extended by a sidecar proxy. In general, a sidecar is a service that is coupled to another service (component) that does not interfere with the functionalities of the main service but extends its properties. A typical example could be a monitoring information saver, that stores (or even analyses) monitoring information issued by one or more services. In MATILDA we take advantage of the sidecar paradigm to attach a proxy to each of the components, taking over the network functionalities of it regarding interconnection with other components, abstracting the network view to the component the sidecar is attached to, in order to build a more efficient service mesh.

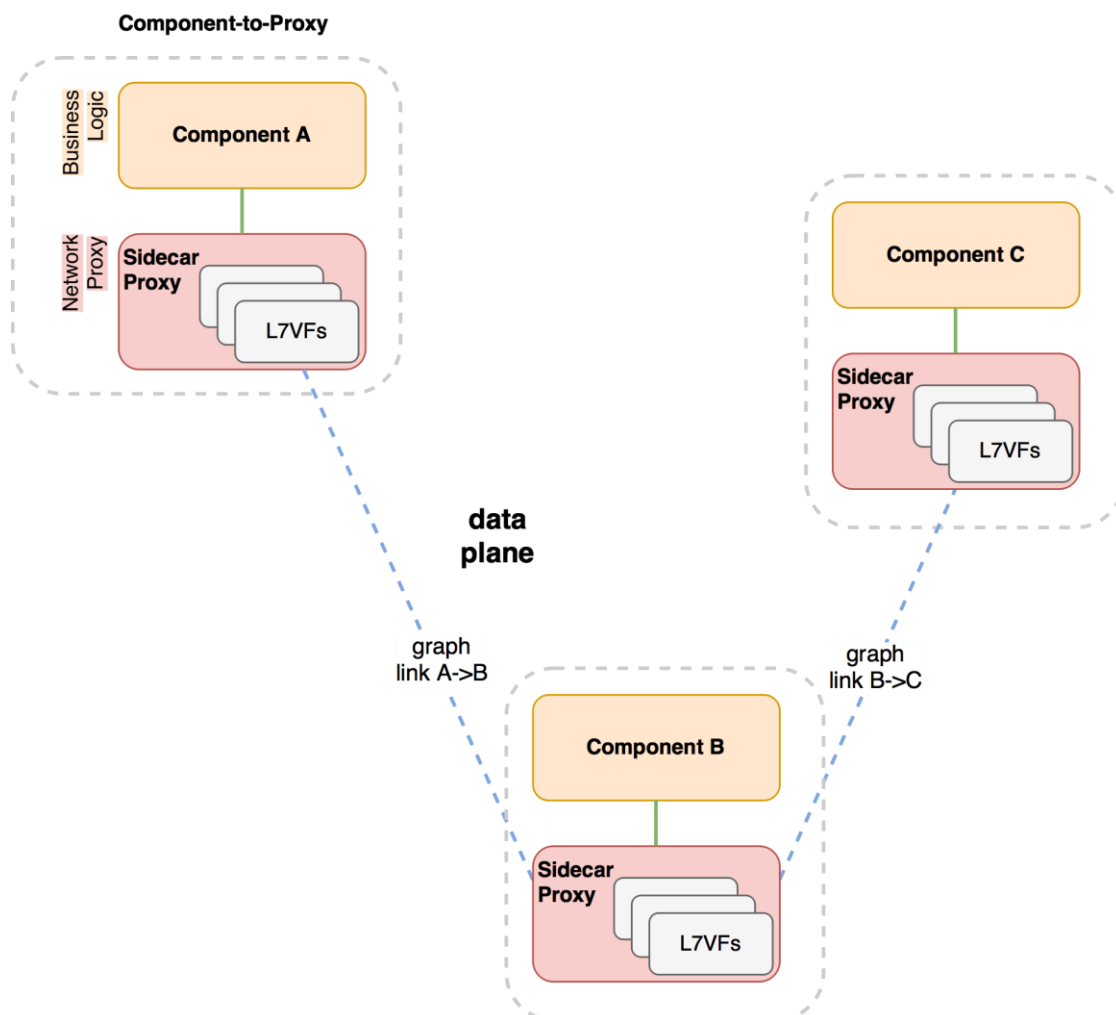


Figure 3: A component-Proxy communication example

The L7 proxy implements L7VFs (plug in functions that are dynamically loaded by the intelligent proxy) like load balancing, HTTP filter, HTTP routing, service discovery, etc. that

operate at the Application level, abstracting the network to the components connected to it, making the communication among them more efficient and easy as it is depicted in Figure 3.

As in MATILDA D1.1 there have been declared in detail the requirements of 5G-ready applications coincide with the above-mentioned requirements. Yet, they are **much more intensive**, since provisioning of infrastructure should be “instantaneous”, topology is continuously changing, delay tolerance is minimum, etc. The concept of this dedicated infrastructure layer is depicted in Figure 4.

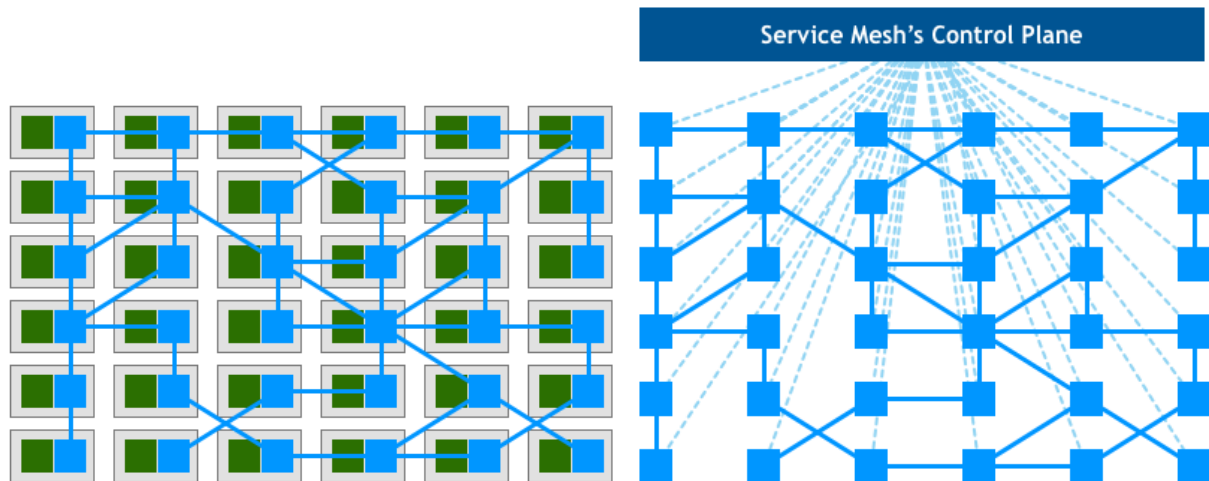


Figure 4: The concept of Service Mesh for cloud-native applications.

In conclusion, in the scope of MATILDA, a “5G-enabled application is a distributed application consisting of cloud-native components that rely on a service mesh infrastructure as a means of network abstraction. The service mesh per se has to operate on top of a programmable 5G environment”. Towards these assumptions, the MATILDA architecture relies on a solid interplay between various logical layers such as the actual data plane, the service mesh control plane, and the configured virtualized resources that are offered by the telco provider as a proper slice [MATILDA-D1.1].

4 Overview of the Metamodel

The component metamodel represents the most granular unit of MATILDA 5G framework. A collection properly selected and interacted components instances forms a DAG (directed acyclic graph), which equals to the application graph. The place of current metamodel in the MATILDA metamodel chain is depicted on Figure 5 where the basic architectural parts of the MATILDA framework and their relationship with the various models is provided.

An application graph placement flow starts with the selection of a vertical application that has to be deployed and supported by a communication service provider. Therefore, a vertical application in MATILDA consists of multiple components that can be deployed on top of programmable infrastructure.

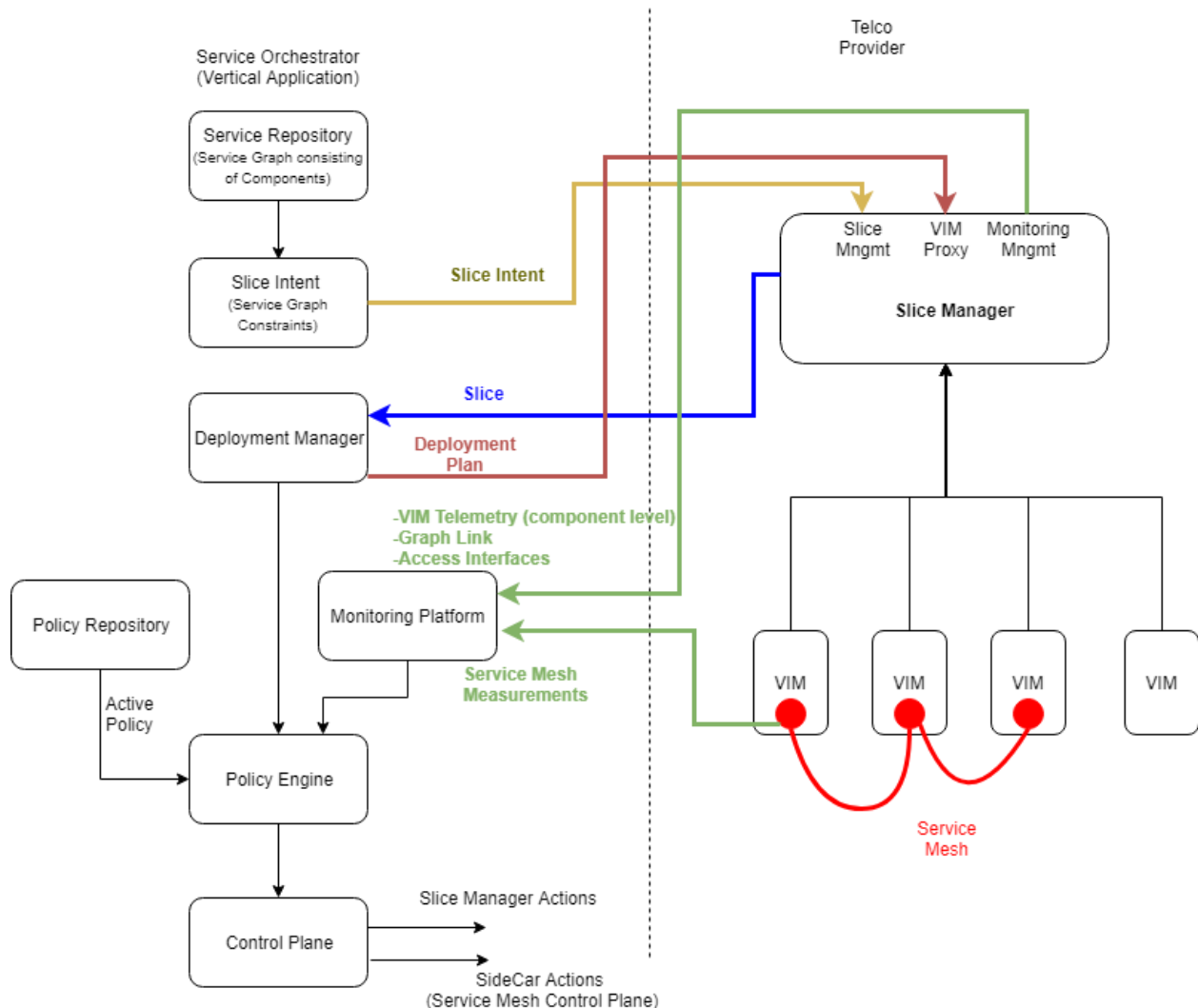


Figure 5: Usage of MATILDA Metamodels

The Service Repository contains all the instances of the application graphs that have been registered. The flow initiates by the selection of application graph by an application/service provider. As depicted in Figure 5, there are two distinct administrative zones. On the left part resides the administrative zone of the vertical application/service orchestrator while on the right part the administrative zone of the telco (communication service) provider. Hence, each administrative zone contains its own orchestration entity with clear responsibilities. The orchestration entity on the left is responsible to instantiate a vertical application that meets specific requirements on the virtualized resources that will be provided by the orchestration mechanism of the right.

Taking under consideration the scope of the two orchestrators, we can easily infer that the Application/Service Orchestrator and the Telco Orchestration mechanisms follow a request/response pattern according to which the Service Orchestrator asks for a specific “setup” that is capable to satisfy some characteristics/requirements and the telco provider responds with the details of the environment that has to be used for the appropriate setup. The first request is addressed as Slice Intent while the latter as the offered Slice. Both specifications are provided in the relevant metamodels, as detailed in D1.4 [MATILDA-D1.4].

As a third step, the telco provider receives the slice intent and tries to find/create a proper setup that will satisfy the set of denoted requirements in step 2. The solution that satisfies the constraints will be announced back to the Service Graph Orchestrator. The solution will be an instance of the Slice Metamodel. Part of the requirements request during deployment or runtime may regard the activation or configuration of network services, able to provide the requested network functionalities. Such services are provided by a NFVO, while the representation is realised based on the metamodel defined in D1.3 [MATILDA-D1.3]0.

4.1 Application Component Part

MATILDA proposes a separation between the business logic part of a component from the layer 4-7 network part of it. We implement a service mesh approach with a dedicated proxy sidecar attached per component. Thus, in this chapter we present the “core” component. For each component, a proxy sidecar will also be utilized. The sidecar will be exploited by an L7 proxy and communication bus framework, such as Envoy [Envoy]. To this end, the modelling of the proxy isn’t required.

The MATILDA component metamodel includes a set of fundamental complexType elements (except from the “ComponentIdentifier” element) that uniquely describe each component in the entire Service Mesh. These elements are the following – depicted in the next figure: (i) *Distribution*, (ii) *ExposedInterface*, (iii) *Configuration*, (iv) *Volume*, (v) *MinimumExecutionRequirements*, (vi) *ExposedMetric*, (vii) *RequiredInterface*, and (viii) *Capability*.

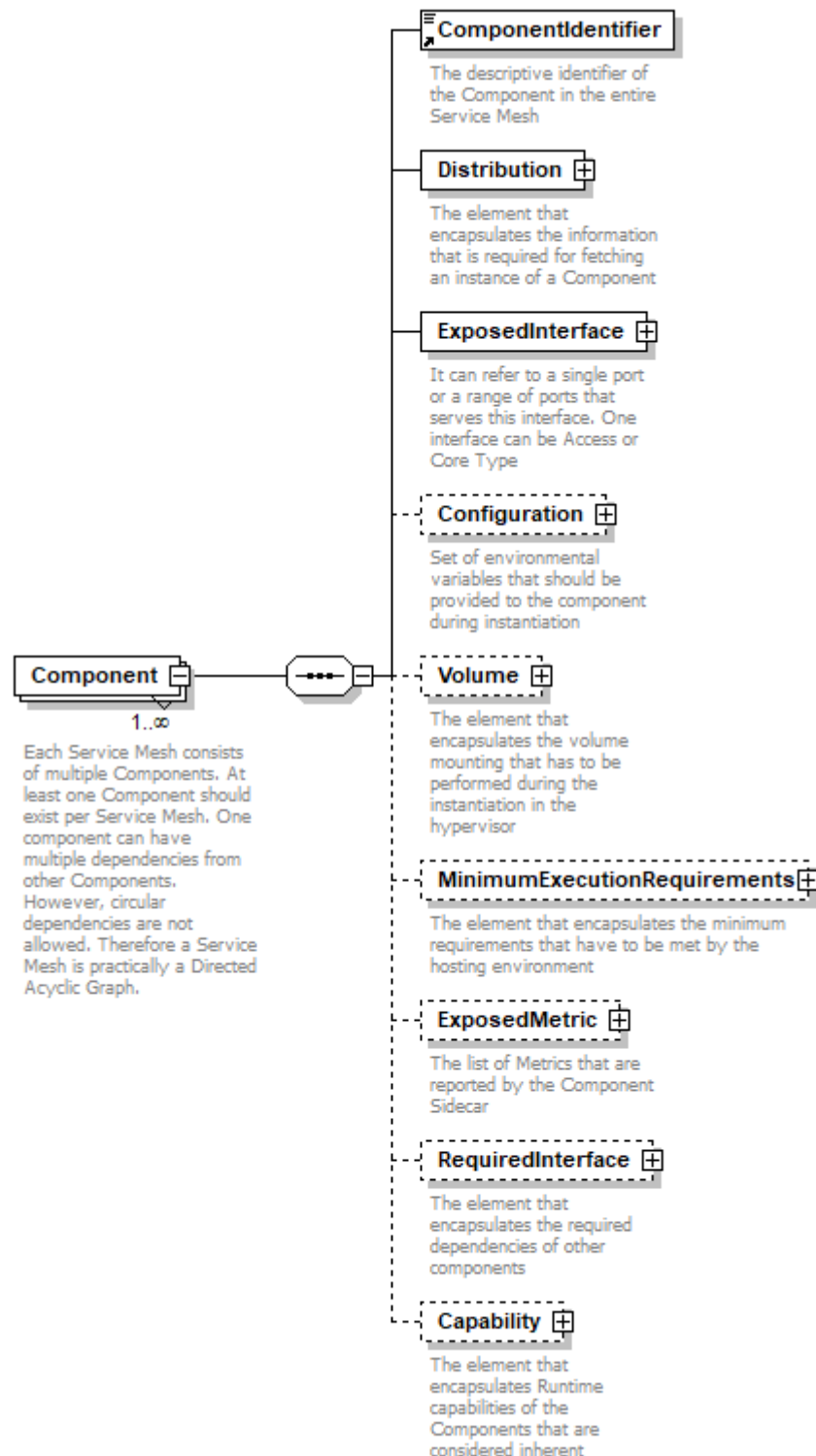


Figure 6: MATILDA Component element

The first complex type element “Distribution” 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.

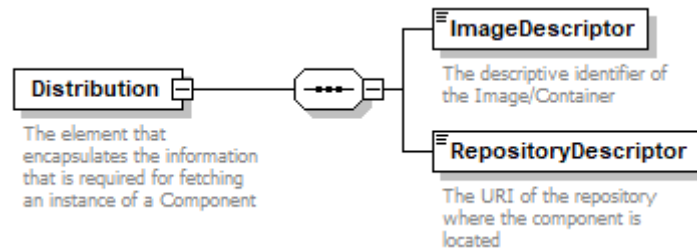


Figure 7: Distribution element

The second complex type element is critical since it describes the exposed interfaces. It is an “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. The ACCESS type refers to the UserEquipment-to-component communication and the CORE refers to the component-to-component communication. Moreover, it contains port declaration and an optional choice for the transport layer protocol.

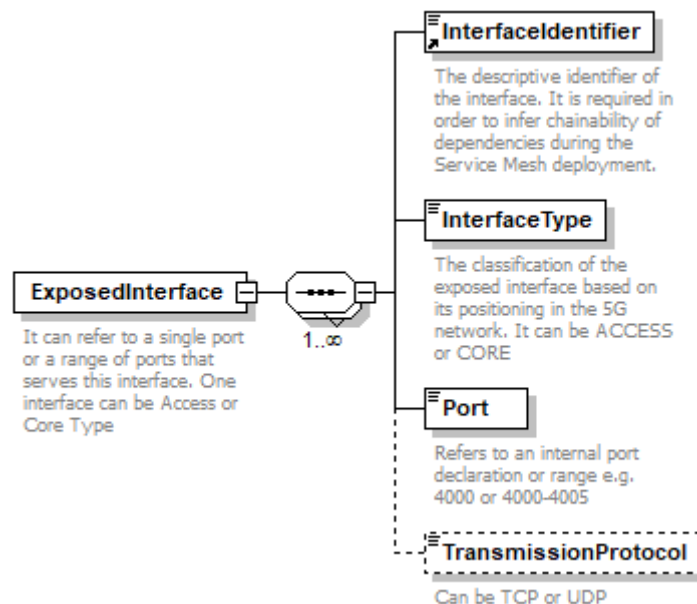


Figure 8: Exposed Interface element

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.

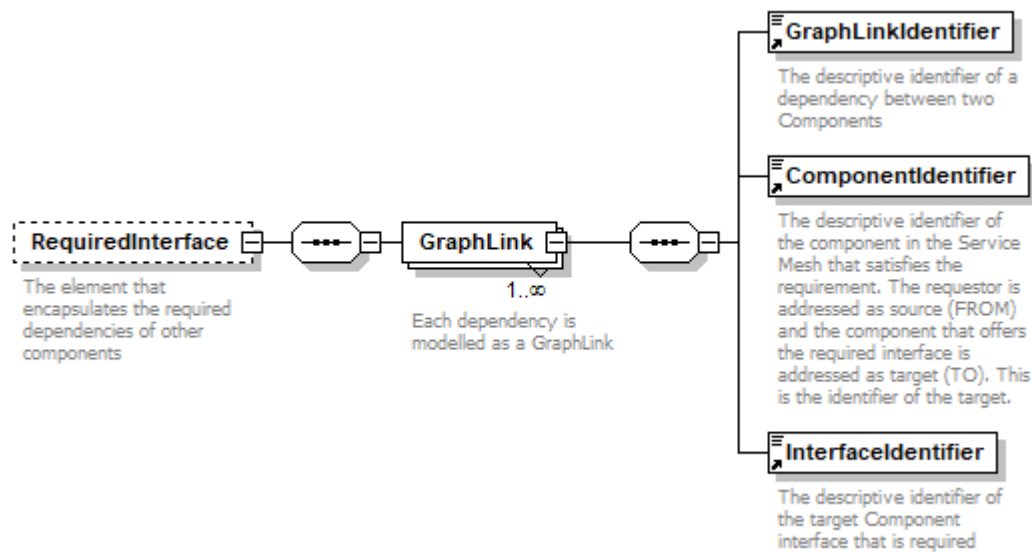


Figure 9: Required Interface element

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.

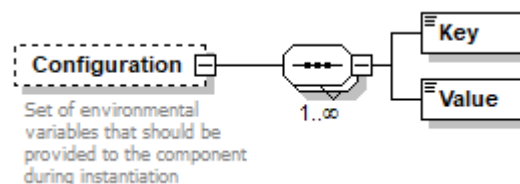


Figure 10: Configuration element

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 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.

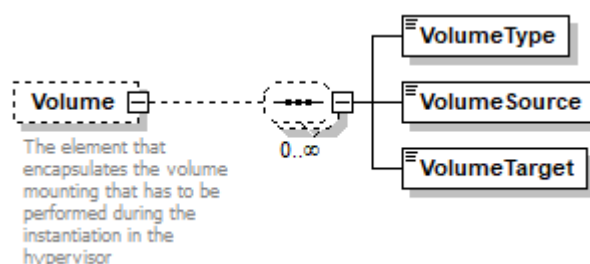


Figure 11: Volume element

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).

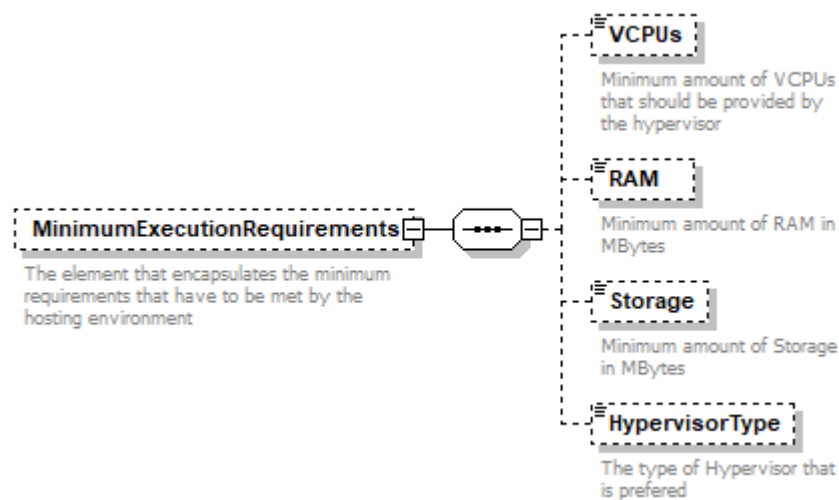


Figure 12: Minimum execution requirements element

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.

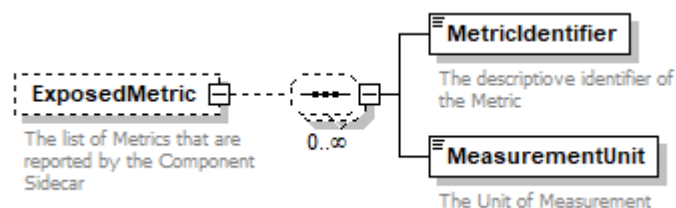


Figure 13: Exposed Metrics element

Finally, the "*Capability*" element encapsulates runtime capabilities of the components that are considered inherent. Such a capability is the scaling of the component.

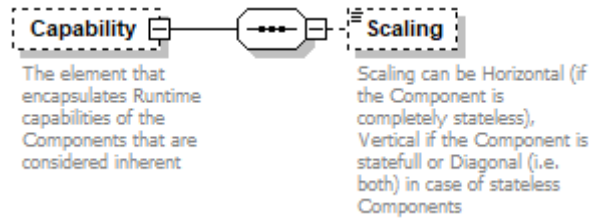


Figure 14: Capability element

A detailed analysis of the developed MATILDA chainable component metamodel is provided in Appendix 1.

4.2 Application Graph / Service Mesh Part

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 the following figure, 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.

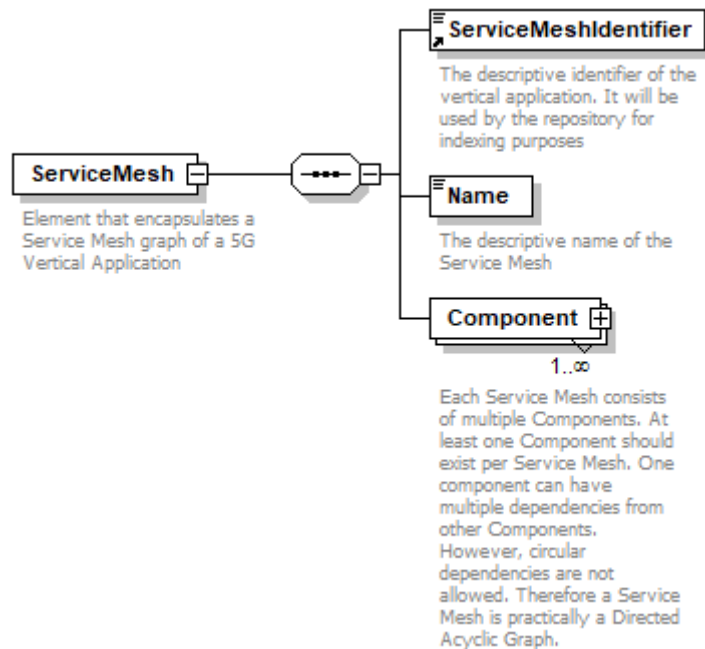


Figure 15: Service Mesh element

While this metamodel seems incomplete, it is fully informative. There is no need for the declaration of graph links and their constraints since: (i) each graph link description is encapsulated in each component as it has been analyzed in the previous subsection of this

document. Specifically, each link is considered as an input required interface of the component which needs it, and (ii) it is a logical link, and each logical link has to be realized as a network link, with network and compute constraints. These constraints are incorporated in another metamodel that is exploited by the proposed MATILDA Vertical Orchestrator [MATILDA-D.1.1]: The Slice Intent metamodel. The figures below depict the relevant part of the metamodel.

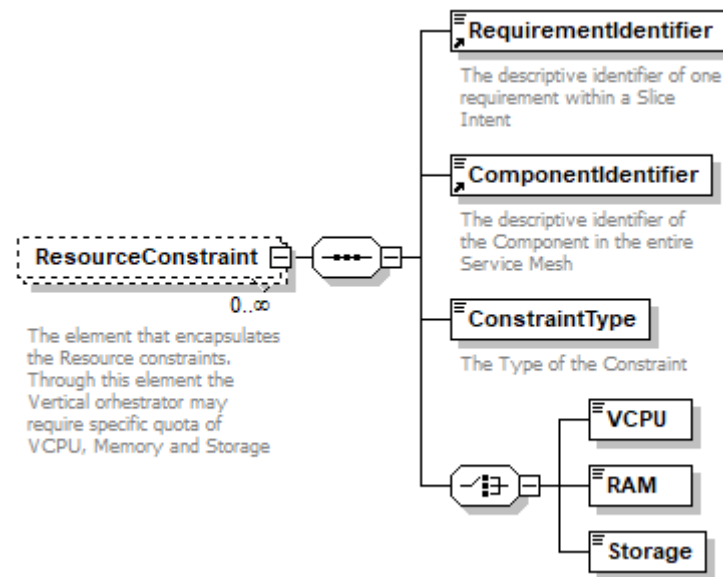


Figure 16: Slice Intent - Resource Constraints element

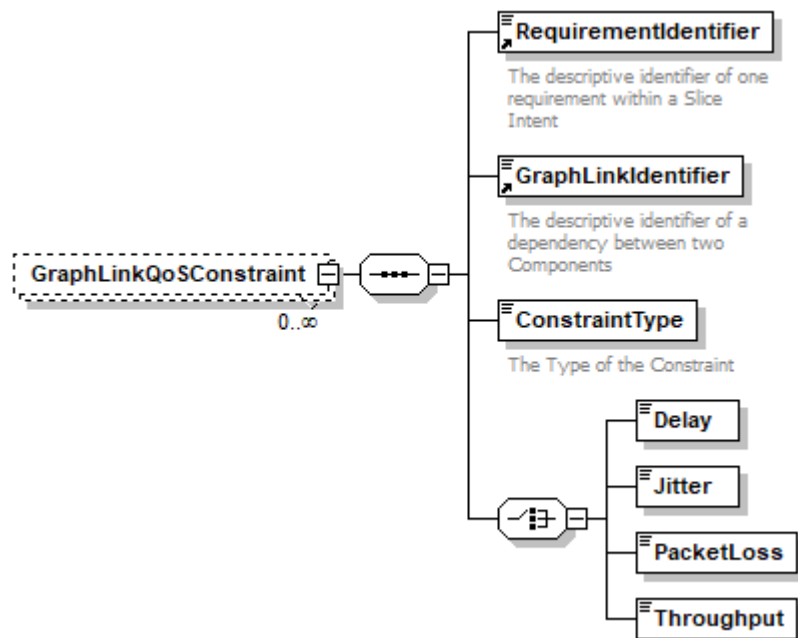


Figure 17: Slice Intent - Graph link QoS constraints element

5 Support Mechanisms Suite

5.1 Overall Architecture

The next figure provides a high-level view of the support mechanisms suite and the basic information flows between components. The main goal of the suite is to verify aspects of the application graph such as the chainability and configurability of the components, the resource needs, etc. The architecture and a brief description per component are cited in this section given that these components exploit information from the proposed 5G-ready application graph metamodel. However, these mechanisms will be further specified and developed in the scope of WP2.

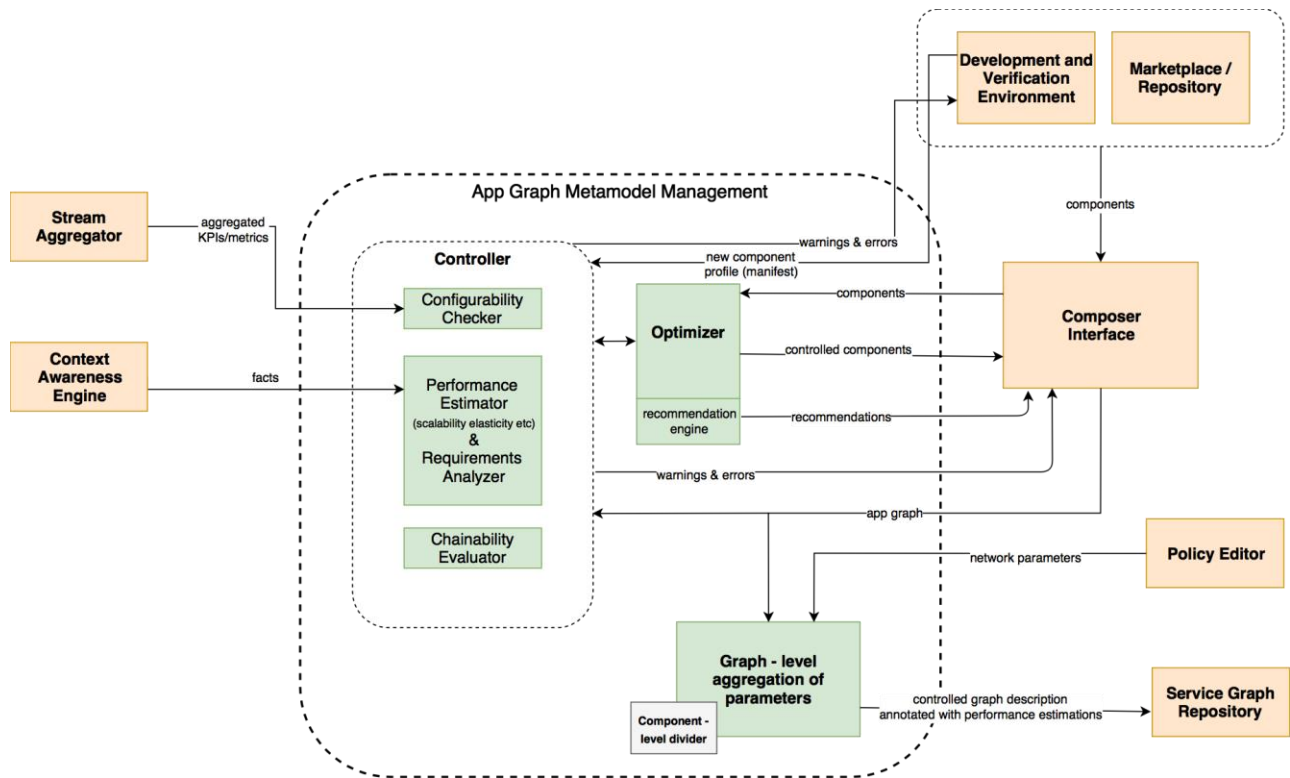


Figure 18: Application Graph metamodel management architecture

5.2 Supporting Mechanisms

A key MATILDA concept regarding 5G applications is the chainable application component. In this context, MATILDA provides a concrete metamodel which covers all characteristics that should be taken under consideration. There is a set of supporting mechanisms that will be developed for the proper support of the metamodel and of the environment in which the metamodel will be deployed and instantiated.

Configurability Checker

The first mechanism regards the evaluation of the configurability of a component. The purpose of this controller mechanism is to check whether a component exposes its configuration parameters and if it is capable of changing its configuration at runtime if necessary. For the first part, the control will be done by collecting the monitoring data and

checking whether the aforementioned parameters are exposed. For the second part, meaning the capability of a component to alter its configuration given the circumstances, there are two ways to check that. The first way is to check the monitoring data to see if any change on the configuration parameters has been made and inspect the component's behavior (e.g. if it kept running or crashed, how it adapted to the changes, etc.). The second way is to develop a test engine that manually uses a component and changes some configuration parameters during runtime to check the components behavior. This second method is crucial to be developed, since we cannot be sure about the configurability of a component based on just the first way because the configuration might not be changed.

Chainability Evaluator

The second controller regards the evaluation of the chainability of a component in an application graph. A component can be chainable only if its required interfaces match the offered interfaces of the component with which it is connected. Taking as example the figure below, the component C1 is chainable to the component C2, only if the required interfaces of the component C2 match the offered interfaces of the component C1. That might concern the data type the C1 offers and the C2 requires. For the component C2 to be chainable must not only be chainable to the component C1, but also to the components C3 and C4 as described above, meaning that the required interfaces of both the components C3 and C4 must match the offered interfaces of the component C2. To perform that kind of control, a mechanism is going to be developed, which will collect the metadata of each component and check the required/offered interfaces. Note that a component cannot be always chainable, given that this relates to the components with which it will be connected within an application graph.

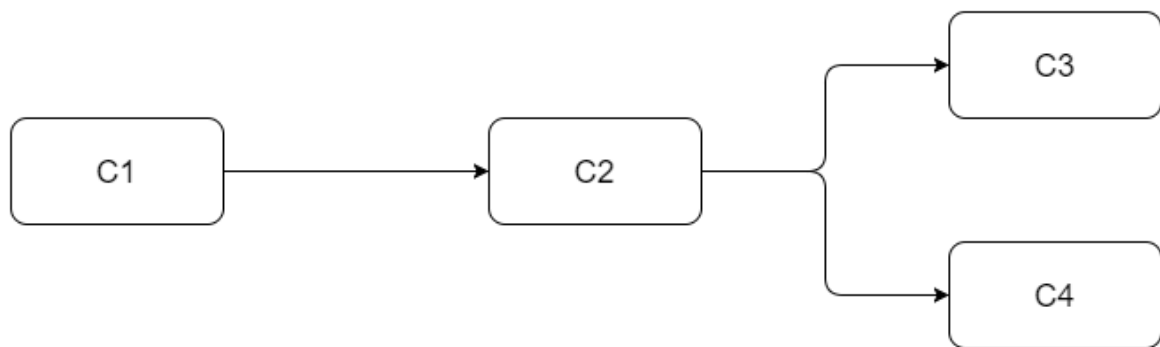


Figure 19: Chainability example

Performance Estimator

The third controller is the Performance Estimator/Profiler. This mechanism actually consists of several sub-mechanisms aiming at providing estimates regarding the required resources during runtime based on several facts (e.g. workload, migration decisions, etc). There are three main types of performance analysis: theoretical, simulation-based and test based. Within the context of MATILDA, a simulation-based approach will be used since it is a generic black-box approach (i.e. no need to analyse the source code) that is based on benchmarks prior to execution. Following the benchmarking outcomes, a training phase will take place in order to identify the relationships between different parameters (e.g. CPU and

latency) and how these evolve. As the model will be trained, during runtime the performance estimator can be utilized to provide estimates based on evolving situations and thus enable resources adaptation accordingly.

Graph Level Aggregator

The fourth mechanism regards the QoS graph-level aggregator. This mechanism is responsible for aggregating all the parameters per component in an application graph and compiling them in overall values for the complete application graph. This mechanism will gather QoS metrics as well as the performance estimations for each component and identify the aggregated requirements needed for the whole application graph, in order to facilitate the execution process. A nice-to-have feature that could be also useful though works the other way around. A sub-mechanism, called Component-level Divider, that the user of the application graph composer could use, setting end-to-end requirements for the whole graph and this mechanism would be responsible for “splitting” the given requirements per component.

Optimizer

The next mechanism is called Optimizer, more of a facilitating mechanism assisting on the faster evaluation of a component and as a result the overall graph. The Optimizer will enable to avoid any unnecessary re-control of a component. There are some controls that cannot be avoided. For instance, the chainability control cannot be avoided, since it is not entirely up to the component but has to do with the adjacent components in the application graph it is used. On the other side, controlling mechanisms like the configurability or the performance estimator can be avoided since these are totally related to the application component. Thus, the optimizer will keep historical data per controlling outcome in order to optimize the overall process.

Recommendation Engine

Another mechanism of the overall suite is the recommendation engine of the MATILDA application graph composer. It is responsible for proposing recommended components (from the ones stored in the MATILDA Marketplace) during the application composition process, thus based on the ones already included in the application graph. The proposed approach consists of a 2-way recommendation system, where an item-based collaborative filtering algorithm will be used exploiting the similarity between the components (components that have been used together in an application graph - preferably adjacent). On that step though, conflicts may arise (e.g. chainability issues). Thus, in order to overcome that kind of problems, this engine will update its recommendations based on the conflicts (e.g. check chainability between each recommended component and exclude the not applicable ones). Moreover, there could be additional features on this mechanism like the way to sort the set of the recommended components (e.g. based on the performance of the components, based on the ratings of the users, a hybrid method combined the abovementioned ways, etc.). Other solutions that have been examined include graph processing, triad models, etc. The final solution to be implemented will be finalized and scoped in the framework of WP2.

6 Conclusions

This deliverable provides the specification and the developed metamodel of the 5G-ready application graph. The metamodel addresses both the component and the overall application spaces, including the required constructs / elements to capture the corresponding information required for deployment and instantiation of the components and thus the overall application. Furthermore, the deliverable provides an initial specification of a set of supporting mechanisms that will analyse the information captured in the metamodels in order to optimize the deployment and execution of 5G applications. These will be further detailed and developed in the scope of WP2 of MATILDA project.

References

[ARCADIA-D.2.2]	ARCADIA project deliverable, D2.2 Definition of the ARCADIA context model, available online at: http://www.arcadia-framework.eu
[AVRO]	Apache Avro, available online at: https://avro.apache.org
[Calçado-2017]	P. Calçado, "Pattern: Service Mesh". Available online at: http://philcalcado.com/2017/08/03/pattern_service_mesh.html
[Docker Compose]	Docker docs, available online at: https://docs.docker.com/compose/startup-order/
[Envoy]	Envoy (an open source edge and service proxy) docs, available online at: https://www.envoyproxy.io/docs/envoy/latest/intro/intro
[Gartner-2016]	"Gartner's Top 10 Strategic Technology Trends for 2017". Available online at: https://www.gartner.com/smarterwithgartner/gartners-top-10-technology-trends-2017/?lipi=urn%3Ali%3Apage%3Ad_flagship3_pulse_read%3BMN%2B0hmkBTTuRW%2B53QE2hzw%3D%3D
[JSON]	Introducing JSON, available online at: https://www.json.org/
[Juju]	Juju Orchestrator, available online at: https://jujucharms.com/
[MATILDA-D.1.1]	D1.1 - MATILDA Framework and Reference Architecture, available online at: http://www.matilda-5g.eu/index.php/outcomes
[MATILDA-D1.3]	D1.3 – VNF/PNF & VNF Forwarding Graph Metamodel, MATILDA H2020 Project, Available Online: http://www.matilda-5g.eu/index.php/outcomes
[MATILDA-D1.4]	D1.4 – Network Slice Intent and Instance Metamodel, MATILDA H2020 Project, Available Online: http://www.matilda-5g.eu/index.php/outcomes
[Morgan-2017]	W. Morgan, "What's a service mesh? And why do I need one?". Available online at: https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/
[nodeRED]	https://nodered.org/
[ProtoBuf]	Protocol Buffers documentation, available online at: https://developers.google.com/protocol-buffers/
[Puppet]	https://puppet.com/
[XML]	Extensible Markup Language, available online at: https://www.w3.org/XML/Overview.html
[YAML]	YAML Ain't Markup Language (YAML™) Version 1.2, available online at: http://yaml.org


Appendix 1: Chainable Component & 5G-ready Application Graph Metamodel (v1.0) Documentation

This appendix provides a complete guide to the first version of the chainable component and 5G-ready application graph metamodel. The reader can click on XSD Elements and navigate to the respective part of the documentation.


Elements

[ComponentIdentifier](#)
[GraphLinkIdentifier](#)
[InterfaceIdentifier](#)
[ServiceMesh](#)
[ServiceMeshIdentifier](#)

element **ComponentIdentifier**


diagram	 <p>The descriptive identifier of the Component in the entire Service Mesh</p>
type	xs:string
properties	content simple
used by	elements ServiceMesh/Component Slice/DeploymentDescriptor/ComponentDeployment ServiceMesh/Component/RequiredInterface/GraphLink SliceIntent/Constraints/ComponentHostingConstraints/LocationConstraint SliceIntent/Constraints/ComponentHostingConstraints/ResourceConstraint
annotation	documentation The descriptive identifier of the Component in the entire Service Mesh
source	<pre> <xs:element name="ComponentIdentifier" type="xs:string"> <xs:annotation> <xs:documentation>The descriptive identifier of the Component in the entire Service Mesh</xs:documentation> </xs:annotation> </xs:element> </pre>

element **GraphLinkIdentifier**

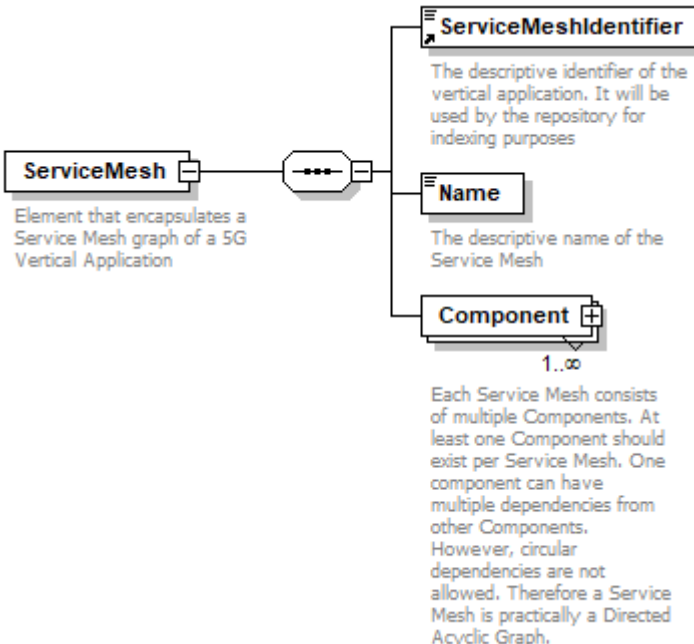
diagram	 <p>The descriptive identifier of a dependency between two Components</p>
type	xs:string
properties	content simple
used by	elements ServiceMesh/Component/RequiredInterface/GraphLink SliceIntent/Constraints/GraphLinkConstraints/GraphLinkQoSConstraint
annotation	documentation The descriptive identifier of a dependency between two Components
source	<pre> <xs:element name="GraphLinkIdentifier" type="xs:string"> </pre>

	<code><xs:annotation></code> <code><xs:documentation></code> The descriptive identifier of a dependency between two Components <code></xs:documentation></code> <code></xs:annotation></code> <code></xs:element></code>
--	---

element **InterfaceIdentifier**

diagram	 <p>It is provided as a metadata in order to infer chainability of dependencies</p>
type	xs:string
properties	content simple
used by	elements SliceIntent/Constraints/AccessConstraints/AccessConstraint ServiceMesh/Component/ExposedInterface ServiceMesh/Component/RequiredInterface/GraphLink Slice/DeploymentDescriptor/ComponentDeployment/InterfaceBinding
annotation	documentation It is provided as a metadata in order to infer chainability of dependencies
source	<code><xs:element name="InterfaceIdentifier" type="xs:string"></code> <code><xs:annotation></code> <code><xs:documentation></code> It is provided as a metadata in order to infer chainability of dependencies <code></xs:documentation></code> <code></xs:annotation></code> <code></xs:element></code>

element **ServiceMesh**

diagram	 <p>ServiceMesh Element that encapsulates a Service Mesh graph of a 5G Vertical Application</p> <p>ServiceMeshIdentifier The descriptive identifier of the vertical application. It will be used by the repository for indexing purposes</p> <p>Name The descriptive name of the Service Mesh</p> <p>Component 1..∞ Each Service Mesh consists of multiple Components. At least one Component should exist per Service Mesh. One component can have multiple dependencies from other Components. However, circular dependencies are not allowed. Therefore a Service Mesh is practically a Directed Acyclic Graph.</p>
properties	content complex
children	ServiceMeshIdentifier Name Component

annotation	documentation Element that encapsulates a Service Mesh graph of a 5G Vertical Application
source	<pre> <xs:element name="ServiceMesh"> <xs:annotation> <xs:documentation>Element that encapsulates a Service Mesh graph of a 5G Vertical Application</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element ref="ServiceMeshIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of the vertical application. It will be used by the repository for indexing purposes</xs:documentation> </xs:annotation> </xs:element> <xs:element name="Name" type="xs:string"> <xs:annotation> <xs:documentation>The descriptive name of the Service Mesh</xs:documentation> </xs:annotation> </xs:element> <xs:element name="Component" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>Each Service Mesh consists of multiple Components. At least one Component should exist per Service Mesh. One component can have multiple dependencies from other Components. However, circular dependencies are not allowed. Therefore a Service Mesh is practically a Directed Acyclic Graph.</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element ref="ComponentIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of the Component in the entire Service Mesh</xs:documentation> </xs:annotation> </xs:element> <xs:element name="Distribution"> <xs:annotation> <xs:documentation>The element that encapsulates the information that is required for fetching an instance of a Component</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="ImageDescriptor" type="xs:string"> <xs:annotation> <xs:documentation>The descriptive identifier of the Image/Container</xs:documentation> </xs:annotation> </xs:element> <xs:element name="RepositoryDescriptor" type="xs:string"> <xs:annotation> <xs:documentation>The URI of the repository where the component is located</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>

```

<xs:element name="ExposedInterface">
  <xs:annotation>
    <xs:documentation>It can refer to a single port or a range of ports that serves this
interface. One interface can be Access or Core Type</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element ref="InterfaceIdentifier">
        <xs:annotation>
          <xs:documentation>The descriptive identifier of the interface. It is required in order
to infer chainability of dependencies during the Service Mesh deployment.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="InterfaceType">
        <xs:annotation>
          <xs:documentation>The classification of the exposed interface based on its
positioning in the 5G network. It can be ACCESS or CORE</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="CORE"/>
            <xs:enumeration value="ACCESS"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="Port">
        <xs:annotation>
          <xs:documentation>Refers to an internal port declaration or range e.g. 4000 or
4000-4005</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="TransmissionProtocol" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Can be TCP or UDP</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="TCP"/>
            <xs:enumeration value="UDP"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Configuration" minOccurs="0">
  <xs:annotation>
    <xs:documentation>Set of environmental variables that should be provided to the
component during instantiation</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="Key" type="xs:string"/>
      <xs:element name="Value" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

```

<xs:element name="Volume" minOccurs="0">
  <xs:annotation>
    <xs:documentation>The element that encapsulates the volume mounting that has to be
    performed during the instantiation in the hypervisor</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="VolumeType" type="xs:string"/>
      <xs:element name="VolumeSource" type="xs:string"/>
      <xs:element name="VolumeTarget" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="MinimumExecutionRequirements" minOccurs="0">
  <xs:annotation>
    <xs:documentation>The element that encapsulates the minimum requirements that
    have to be met by the hosting environment</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="VCPU" type="xs:int" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Minimum amount of VCPUs that should be provided by the
          hypervisor</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="RAM" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Minimum amount of RAM in MBytes</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Storage" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Minimum amount of Storage in MBytes</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="HypervisorType" minOccurs="0">
        <xs:annotation>
          <xs:documentation>The type of Hypervisor that is preferred</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="ESXI"/>
            <xs:enumeration value="KVM"/>
            <xs:enumeration value="XEN"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="ExposedMetric" minOccurs="0">
  <xs:annotation>
    <xs:documentation>The list of Metrics that are reported by the Component
    Sidecar</xs:documentation>
  </xs:annotation>
  <xs:complexType>

```


```

<xs:sequence minOccurs="0" maxOccurs="unbounded">
  <xs:element name="MetricIdentifier">
    <xs:annotation>
      <xs:documentation>The descriptive identifier of the Metric</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="MeasurementUnit">
    <xs:annotation>
      <xs:documentation>The Unit of Measurement</xs:documentation>
    </xs:annotation>
  </xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="RequiredInterface" minOccurs="0">
  <xs:annotation>
    <xs:documentation>The element that encapsulates the required dependencies of other
components</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="GraphLink" maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>Each dependency is modelled as a GraphLink
</xs:documentation>
        </xs:annotation>
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="GraphLinkIdentifier">
              <xs:annotation>
                <xs:documentation>The descriptive identifier of a dependency between two
Components</xs:documentation>
              </xs:annotation>
            </xs:element>
            <xs:element ref="ComponentIdentifier">
              <xs:annotation>
                <xs:documentation>The descriptive identifier of the component in the Service
Mesh that satisfies the requirement. The requestor is addressed as source (FROM) and the
component that offers the required interface is addressed as target (TO). This is the identifier of
the target.</xs:documentation>
              </xs:annotation>
            </xs:element>
            <xs:element ref="InterfaceIdentifier">
              <xs:annotation>
                <xs:documentation>The descriptive identifier of the target Component interface
that is required</xs:documentation>
              </xs:annotation>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Capability" minOccurs="0">
  <xs:annotation>
    <xs:documentation>The element that encapsulates Runtime capabilities of the

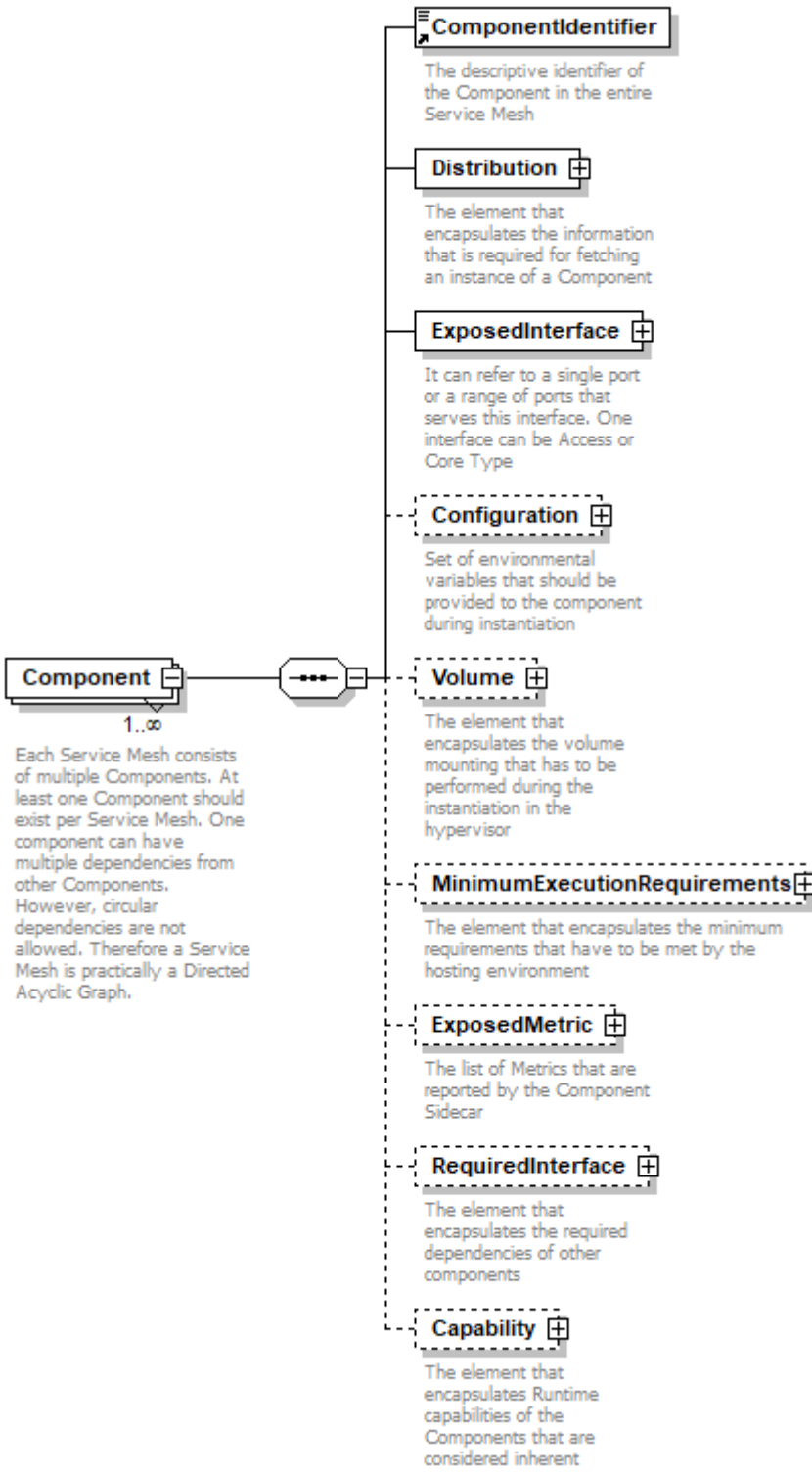
```

	<p>Components that are considered inherent</xs:documentation></p> <pre> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="Scaling" minOccurs="0"> <xs:annotation> <xs:documentation>Scaling can be Horizontal (if the Component is completely stateless), Vertical if the Component is statefull or Diagonal (i.e. both) in case of stateless Components</xs:documentation> </xs:annotation> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="HORIZONTAL"/> <xs:enumeration value="VERTICAL"/> <xs:enumeration value="DIAGONAL"/> </xs:restriction> </xs:simpleType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
--	--

element **ServiceMesh/Name**

diagram	
type	xs:string
properties	isRef 0 content simple
annotation	documentation The descriptive name of the Service Mesh
source	<pre> <xs:element name="Name" type="xs:string"> <xs:annotation> <xs:documentation>The descriptive name of the Service Mesh</xs:documentation> </xs:annotation> </xs:element> </pre>

element ServiceMesh/Component

diagram	 <p>Each Service Mesh consists of multiple Components. At least one Component should exist per Service Mesh. One component can have multiple dependencies from other Components. However, circular dependencies are not allowed. Therefore a Service Mesh is practically a Directed Acyclic Graph.</p>
properties	isRef 0 minOcc 1 maxOcc unbounded content complex
children	ComponentIdentifier Distribution ExposedInterface Configuration Volume MinimumExecutionRequirements ExposedMetric RequiredInterface Capability
annotation	documentation

	<p>Each Service Mesh consists of multiple Components. At least one Component should exist per Service Mesh. One component can have multiple dependencies from other Components. However, circular dependencies are not allowed. Therefore a Service Mesh is practically a Directed Acyclic Graph.</p>
source	<pre> <xs:element name="Component" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>Each Service Mesh consists of multiple Components. At least one Component should exist per Service Mesh. One component can have multiple dependencies from other Components. However, circular dependencies are not allowed. Therefore a Service Mesh is practically a Directed Acyclic Graph.</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element ref="ComponentIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of the Component in the entire Service Mesh</xs:documentation> </xs:annotation> </xs:element> <xs:element name="Distribution"> <xs:annotation> <xs:documentation>The element that encapsulates the information that is required for fetching an instance of a Component</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="ImageDescriptor" type="xs:string"> <xs:annotation> <xs:documentation>The descriptive identifier of the Image/Container</xs:documentation> </xs:annotation> </xs:element> <xs:element name="RepositoryDescriptor" type="xs:string"> <xs:annotation> <xs:documentation>The URI of the repository where the component is located</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="ExposedInterface"> <xs:annotation> <xs:documentation>It can refer to a single port or a range of ports that serves this interface. One interface can be Access or Core Type</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence maxOccurs="unbounded"> <xs:element ref="InterfaceIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of the interface. It is required in order to infer chainability of dependencies during the Service Mesh deployment.</xs:documentation> </xs:annotation> </xs:element> <xs:element name="InterfaceType"> <xs:annotation> <xs:documentation>The classification of the exposed interface based on its positioning in the 5G network. It can be ACCESS or CORE</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>

```

<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:enumeration value="CORE"/>
    <xs:enumeration value="ACCESS"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element name="Port">
  <xs:annotation>
    <xs:documentation>Refers to an internal port declaration or range e.g. 4000 or 4000-
4005</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="TransmissionProtocol" minOccurs="0">
  <xs:annotation>
    <xs:documentation>Can be TCP or UDP</xs:documentation>
  </xs:annotation>
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="TCP"/>
      <xs:enumeration value="UDP"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="Configuration" minOccurs="0">
  <xs:annotation>
    <xs:documentation>Set of environmental variables that should be provided to the
component during instantiation</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="Key" type="xs:string"/>
      <xs:element name="Value" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Volume" minOccurs="0">
  <xs:annotation>
    <xs:documentation>The element that encapsulates the volume mounting that has to be
performed during the instantiation in the hypervisor</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="VolumeType" type="xs:string"/>
      <xs:element name="VolumeSource" type="xs:string"/>
      <xs:element name="VolumeTarget" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="MinimumExecutionRequirements" minOccurs="0">
  <xs:annotation>
    <xs:documentation>The element that encapsulates the minimum requirements that have to
be met by the hosting environment</xs:documentation>
  </xs:annotation>

```

```

<xs:complexType>
  <xs:sequence>
    <xs:element name="VCPU" type="xs:int" minOccurs="0">
      <xs:annotation>
        <xs:documentation>Minimum amount of VCPUs that should be provided by the
hypervisor</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="RAM" minOccurs="0">
      <xs:annotation>
        <xs:documentation>Minimum amount of RAM in MBytes</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="Storage" minOccurs="0">
      <xs:annotation>
        <xs:documentation>Minimum amount of Storage in MBytes</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="HypervisorType" minOccurs="0">
      <xs:annotation>
        <xs:documentation>The type of Hypervisor that is preferred</xs:documentation>
      </xs:annotation>
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="ESXI"/>
          <xs:enumeration value="KVM"/>
          <xs:enumeration value="XEN"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="ExposedMetric" minOccurs="0">
  <xs:annotation>
    <xs:documentation>The list of Metrics that are reported by the Component
Sidecar</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="MetricIdentifier">
        <xs:annotation>
          <xs:documentation>The descriptive identifier of the Metric</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="MeasurementUnit">
        <xs:annotation>
          <xs:documentation>The Unit of Measurement</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="RequiredInterface" minOccurs="0">
  <xs:annotation>
    <xs:documentation>The element that encapsulates the required dependencies of other
components</xs:documentation>

```

```

</xs:annotation>
<xs:complexType>
  <xs:sequence>
    <xs:element name="GraphLink" maxOccurs="unbounded">
      <xs:annotation>
        <xs:documentation>Each dependency is modelled as a GraphLink
      </xs:documentation>
    </xs:annotation>
  </xs:complexType>
  <xs:sequence>
    <xs:element ref="GraphLinkIdentifier">
      <xs:annotation>
        <xs:documentation>The descriptive identifier of a dependency between two
Components</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element ref="ComponentIdentifier">
      <xs:annotation>
        <xs:documentation>The descriptive identifier of the component in the Service Mesh
that satisfies the requirement. The requestor is addressed as source (FROM) and the component
that offers the required interface is addressed as target (TO). This is the identifier of the
target.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element ref="InterfaceIdentifier">
      <xs:annotation>
        <xs:documentation>The descriptive identifier of the target Component interface that
is required</xs:documentation>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="Capability" minOccurs="0">
  <xs:annotation>
    <xs:documentation>The element that encapsulates Runtime capabilities of the
Components that are considered inherent</xs:documentation>
  </xs:annotation>
</xs:complexType>
  <xs:sequence>
    <xs:element name="Scaling" minOccurs="0">
      <xs:annotation>
        <xs:documentation>Scaling can be Horizontal (if the Component is completely
stateless), Vertical if the Component is statefull or Diagonal (i.e. both) in case of stateless
Components</xs:documentation>
      </xs:annotation>
    </xs:complexType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="HORIZONTAL"/>
      <xs:enumeration value="VERTICAL"/>
      <xs:enumeration value="DIAGONAL"/>
    </xs:restriction>
  </xs:complexType>
</xs:element>

```

	<code></xs:sequence></code> <code></xs:complexType></code> <code></xs:element></code> <code></xs:sequence></code> <code></xs:complexType></code> <code></xs:element></code>
--	--

element **ServiceMesh/Component/Distribution**


diagram	
properties	isRef 0 content complex
children	ImageDescriptor RepositoryDescriptor
annotation	documentation The element that encapsulates the information that is required for fetching an instance of a Component
source	<pre> <xs:element name="Distribution"> <xs:annotation> <xs:documentation>The element that encapsulates the information that is required for fetching an instance of a Component</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="ImageDescriptor" type="xs:string"> <xs:annotation> <xs:documentation>The descriptive identifier of the Image/Container</xs:documentation> </xs:annotation> </xs:element> <xs:element name="RepositoryDescriptor" type="xs:string"> <xs:annotation> <xs:documentation>The URI of the repository where the component is located</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>

element **ServiceMesh/Component/Distribution/ImageDescriptor**

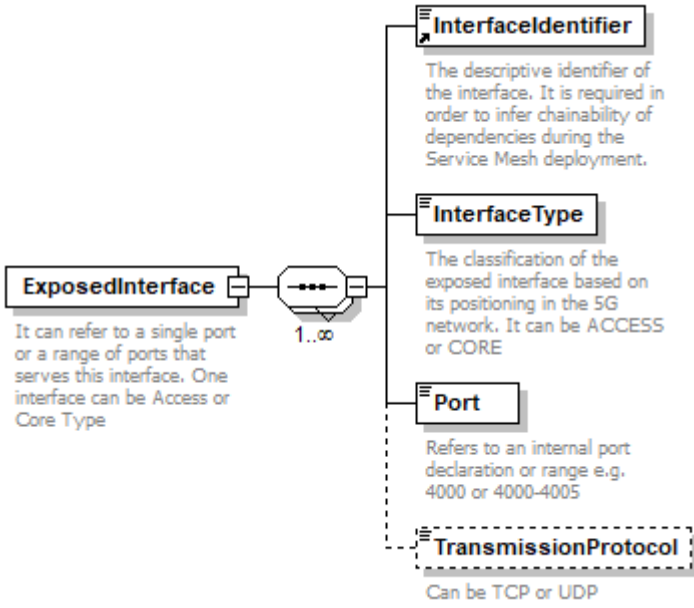
diagram	
type	xs:string
properties	isRef 0 content simple

annotation	documentation The descriptive identifier of the Image/Container
source	<pre><xs:element name="ImageDescriptor" type="xs:string"> <xs:annotation> <xs:documentation>The descriptive identifier of the Image/Container</xs:documentation> </xs:annotation> </xs:element></pre>

element **ServiceMesh/Component/Distribution/RepositoryDescriptor**

diagram	
type	xs:string
properties	isRef 0 content simple
annotation	documentation The URI of the repository where the component is located
source	<pre><xs:element name="RepositoryDescriptor" type="xs:string"> <xs:annotation> <xs:documentation>The URI of the repository where the component is located</xs:documentation> </xs:annotation> </xs:element></pre>

element **ServiceMesh/Component/ExposedInterface**

diagram	
properties	isRef 0 content complex
children	InterfaceIdentifier InterfaceType Port TransmissionProtocol
annotation	documentation It can refer to a single port or a range of ports that serves this interface. One interface can be Access or Core Type

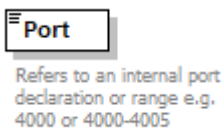
source	<pre> <xs:element name="ExposedInterface"> <xs:annotation> <xs:documentation>It can refer to a single port or a range of ports that serves this interface. One interface can be Access or Core Type</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence maxOccurs="unbounded"> <xs:element ref="InterfaceIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of the interface. It is required in order to infer chainability of dependencies during the Service Mesh deployment.</xs:documentation> </xs:annotation> </xs:element> <xs:element name="InterfaceType"> <xs:annotation> <xs:documentation>The classification of the exposed interface based on its positioning in the 5G network. It can be ACCESS or CORE</xs:documentation> </xs:annotation> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="CORE"/> <xs:enumeration value="ACCESS"/> </xs:restriction> </xs:simpleType> </xs:element> <xs:element name="Port"> <xs:annotation> <xs:documentation>Refers to an internal port declaration or range e.g. 4000 or 4000- 4005</xs:documentation> </xs:annotation> </xs:element> <xs:element name="TransmissionProtocol" minOccurs="0"> <xs:annotation> <xs:documentation>Can be TCP or UDP</xs:documentation> </xs:annotation> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="TCP"/> <xs:enumeration value="UDP"/> </xs:restriction> </xs:simpleType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>

element ServiceMesh/Component/ExposedInterface/InterfaceType


diagram	
---------	---

type	restriction of xs:string
properties	isRef 0 content simple
facets	enumeration CORE enumeration ACCESS
annotation	documentation The classification of the exposed interface based on its positioning in the 5G network. It can be ACCESS or CORE
source	<pre> <xs:element name="InterfaceType"> <xs:annotation> <xs:documentation>The classification of the exposed interface based on its positioning in the 5G network. It can be ACCESS or CORE</xs:documentation> </xs:annotation> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="CORE"/> <xs:enumeration value="ACCESS"/> </xs:restriction> </xs:simpleType> </xs:element> </pre>

element **ServiceMesh/Component/ExposedInterface/Port**

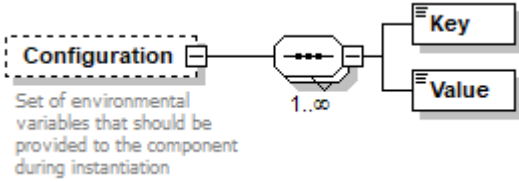
diagram	
properties	isRef 0
annotation	documentation Refers to an internal port declaration or range e.g. 4000 or 4000-4005
source	<pre> <xs:element name="Port"> <xs:annotation> <xs:documentation>Refers to an internal port declaration or range e.g. 4000 or 4000- 4005</xs:documentation> </xs:annotation> </xs:element> </pre>

element **ServiceMesh/Component/ExposedInterface/TransmissionProtocol**

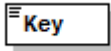
diagram	
type	restriction of xs:string
properties	isRef 0 minOcc 0 maxOcc 1 content simple
facets	enumeration TCP enumeration UDP
annotation	documentation Can be TCP or UDP
source	<pre> <xs:element name="TransmissionProtocol" minOccurs="0"> <xs:annotation> <xs:documentation>Can be TCP or UDP</xs:documentation> </xs:annotation> </pre>

	<pre> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="TCP"/> <xs:enumeration value="UDP"/> </xs:restriction> </xs:simpleType> </xs:element> </pre>
--	---

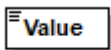
element **ServiceMesh/Component/Configuration**

diagram	
properties	isRef 0 minOcc 0 maxOcc 1 content complex
children	Key Value
annotation	documentation Set of environmental variables that should be provided to the component during instantiation
source	<pre> <xs:element name="Configuration" minOccurs="0"> <xs:annotation> <xs:documentation>Set of environmental variables that should be provided to the component during instantiation</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence maxOccurs="unbounded"> <xs:element name="Key" type="xs:string"/> <xs:element name="Value" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> </pre>

element **ServiceMesh/Component/Configuration/Key**

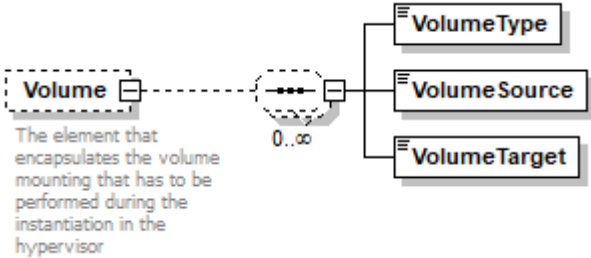
diagram	
type	xs:string
properties	isRef 0 content simple
source	<pre> <xs:element name="Key" type="xs:string"/> </pre>

element **ServiceMesh/Component/Configuration/Value**


diagram	
type	xs:string
properties	isRef 0 content simple

source	<code><xs:element name="Value" type="xs:string"/></code>
--------	--


element ServiceMesh/Component/Volume

diagram	
properties	isRef 0 minOcc 0 maxOcc 1 content complex
children	VolumeType VolumeSource VolumeTarget
annotation	documentation The element that encapsulates the volume mounting that has to be performed during the instantiation in the hypervisor
source	<pre> <xs:element name="Volume" minOccurs="0"> <xs:annotation> <xs:documentation>The element that encapsulates the volume mounting that has to be performed during the instantiation in the hypervisor</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence minOccurs="0" maxOccurs="unbounded"> <xs:element name="VolumeType" type="xs:string"/> <xs:element name="VolumeSource" type="xs:string"/> <xs:element name="VolumeTarget" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> </pre>

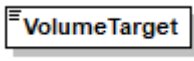
element ServiceMesh/Component/Volume/VolumeType

diagram	
type	xs:string
properties	isRef 0 content simple
source	<code><xs:element name="VolumeType" type="xs:string"/></code>

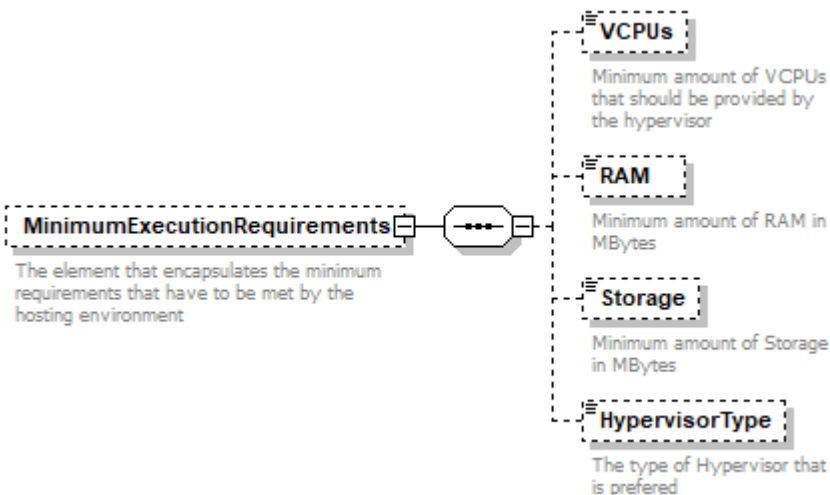
element ServiceMesh/Component/Volume/VolumeSource

diagram	
type	xs:string
properties	isRef 0 content simple
source	<code><xs:element name="VolumeSource" type="xs:string"/></code>

element ServiceMesh/Component/Volume/VolumeTarget

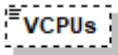
diagram	
type	xs:string
properties	isRef 0 content simple
source	<code><xs:element name="VolumeTarget" type="xs:string"/></code>

element ServiceMesh/Component/MinimumExecutionRequirements

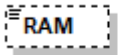
diagram	
properties	isRef 0 minOcc 0 maxOcc 1 content complex
children	VCPUs RAM Storage HypervisorType
annotation	documentation The element that encapsulates the minimum requirements that have to be met by the hosting environment
source	<pre><xs:element name="MinimumExecutionRequirements" minOccurs="0"> <xs:annotation> <xs:documentation>The element that encapsulates the minimum requirements that have to be met by the hosting environment</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="VCPUs" type="xs:int" minOccurs="0"> <xs:annotation> <xs:documentation>Minimum amount of VCPUs that should be provided by the hypervisor</xs:documentation> </xs:annotation> </xs:element> <xs:element name="RAM" minOccurs="0"> <xs:annotation> <xs:documentation>Minimum amount of RAM in MBytes</xs:documentation> </xs:annotation> </xs:element> <xs:element name="Storage" minOccurs="0"></pre>

	<pre> <xs:annotation> <xs:documentation>Minimum amount of Storage in MBytes</xs:documentation> </xs:annotation> </xs:element> <xs:element name="HypervisorType" minOccurs="0"> <xs:annotation> <xs:documentation>The type of Hypervisor that is preferred</xs:documentation> </xs:annotation> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="ESXI"/> <xs:enumeration value="KVM"/> <xs:enumeration value="XEN"/> </xs:restriction> </xs:simpleType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
--	---

element ServiceMesh/Component/MinimumExecutionRequirements/VCPUs


diagram	 <p>Minimum amount of VCPUs that should be provided by the hypervisor</p>
type	xs:int
properties	isRef 0 minOcc 0 maxOcc 1 content simple
annotation	documentation Minimum amount of VCPUs that should be provided by the hypervisor
source	<pre> <xs:element name="VCPUs" type="xs:int" minOccurs="0"> <xs:annotation> <xs:documentation>Minimum amount of VCPUs that should be provided by the hypervisor</xs:documentation> </xs:annotation> </xs:element> </pre>

element ServiceMesh/Component/MinimumExecutionRequirements/RAM

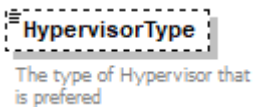
diagram	 <p>Minimum amount of RAM in MBytes</p>
properties	isRef 0 minOcc 0 maxOcc 1
annotation	documentation Minimum amount of RAM in MBytes
source	<pre> <xs:element name="RAM" minOccurs="0"> <xs:annotation> <xs:documentation>Minimum amount of RAM in MBytes</xs:documentation> </xs:annotation> </pre>

```
</xs:element>
```

element ServiceMesh/Component/MinimumExecutionRequirements/Storage

diagram	
properties	isRef 0 minOcc 0 maxOcc 1
annotation	documentation Minimum amount of Storage in MBytes
source	<pre><xs:element name="Storage" minOccurs="0"> <xs:annotation> <xs:documentation>Minimum amount of Storage in MBytes</xs:documentation> </xs:annotation> </xs:element></pre>


element ServiceMesh/Component/MinimumExecutionRequirements/HypervisorType

diagram	
type	restriction of xs:string
properties	isRef 0 minOcc 0 maxOcc 1 content simple
facets	enumeration ESXI enumeration KVM enumeration XEN
annotation	documentation The type of Hypervisor that is preferred
source	<pre><xs:element name="HypervisorType" minOccurs="0"> <xs:annotation> <xs:documentation>The type of Hypervisor that is preferred</xs:documentation> </xs:annotation> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="ESXI"/> <xs:enumeration value="KVM"/> <xs:enumeration value="XEN"/> </xs:restriction> </xs:simpleType> </xs:element></pre>

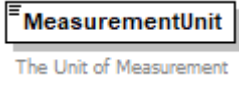
element ServiceMesh/Component/ExposedMetric

diagram	
properties	isRef 0 minOcc 0 maxOcc 1 content complex
children	MetricIdentifier MeasurementUnit
annotation	documentation The list of Metrics that are reported by the Component Sidecar
source	<pre> <xs:element name="ExposedMetric" minOccurs="0"> <xs:annotation> <xs:documentation>The list of Metrics that are reported by the Component Sidecar</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence minOccurs="0" maxOccurs="unbounded"> <xs:element name="MetricIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of the Metric</xs:documentation> </xs:annotation> </xs:element> <xs:element name="MeasurementUnit"> <xs:annotation> <xs:documentation>The Unit of Measurement</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>

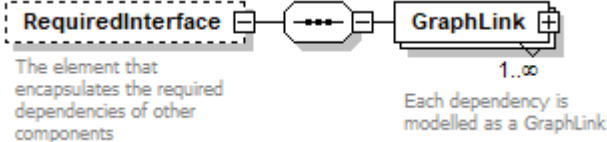
element ServiceMesh/Component/ExposedMetric/MetricIdentifier

diagram	
properties	isRef 0
annotation	documentation The descriptive identifier of the Metric
source	<pre> <xs:element name="MetricIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of the Metric</xs:documentation> </xs:annotation> </xs:element> </pre>

element ServiceMesh/Component/ExposedMetric/MeasurementUnit

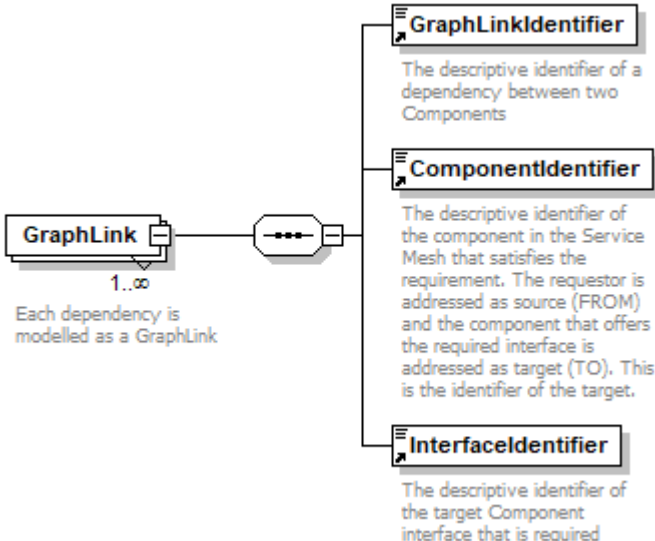
diagram	
properties	isRef 0
annotation	documentation The Unit of Measurement
source	<pre><xs:element name="MeasurementUnit"> <xs:annotation> <xs:documentation>The Unit of Measurement</xs:documentation> </xs:annotation> </xs:element></pre>

element ServiceMesh/Component/RequiredInterface

diagram	
properties	isRef 0 minOcc 0 maxOcc 1 content complex
children	GraphLink
annotation	documentation The element that encapsulates the required dependencies of other components
source	<pre><xs:element name="RequiredInterface" minOccurs="0"> <xs:annotation> <xs:documentation>The element that encapsulates the required dependencies of other components</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="GraphLink" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>Each dependency is modelled as a GraphLink </xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element ref="GraphLinkIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of a dependency between two Components</xs:documentation> </xs:annotation> </xs:element> <xs:element ref="ComponentIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of the component in the Service Mesh that satisfies the requirement. The requestor is addressed as source (FROM) and the component that offers the required interface is addressed as target (TO). This is the identifier of the target.</xs:documentation> </xs:annotation> </xs:element> <xs:element ref="InterfaceIdentifier"></pre>

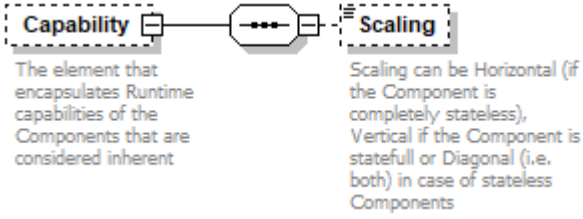
	<pre> <xs:annotation> <xs:documentation>The descriptive identifier of the target Component interface that is required</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
--	--

element ServiceMesh/Component/RequiredInterface/GraphLink

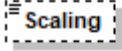
diagram	 <p>Each dependency is modelled as a GraphLink</p> <p>GraphLinkIdentifier The descriptive identifier of a dependency between two Components</p> <p>ComponentIdentifier The descriptive identifier of the component in the Service Mesh that satisfies the requirement. The requestor is addressed as source (FROM) and the component that offers the required interface is addressed as target (TO). This is the identifier of the target.</p> <p>InterfaceIdentifier The descriptive identifier of the target Component interface that is required</p>								
properties	<table> <tr><td>isRef</td><td>0</td></tr> <tr><td>minOcc</td><td>1</td></tr> <tr><td>maxOcc</td><td>unbounded</td></tr> <tr><td>content</td><td>complex</td></tr> </table>	isRef	0	minOcc	1	maxOcc	unbounded	content	complex
isRef	0								
minOcc	1								
maxOcc	unbounded								
content	complex								
children	GraphLinkIdentifier ComponentIdentifier InterfaceIdentifier								
annotation	documentation Each dependency is modelled as a GraphLink								
source	<pre> <xs:element name="GraphLink" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>Each dependency is modelled as a GraphLink </xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element ref="GraphLinkIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of a dependency between two Components</xs:documentation> </xs:annotation> </xs:element> <xs:element ref="ComponentIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of the component in the Service Mesh that satisfies the requirement. The requestor is addressed as source (FROM) and the component that offers the required interface is addressed as target (TO). This is the identifier of the </pre>								

	<pre> target.</xs:documentation> </xs:annotation> </xs:element> <xs:element ref="InterfaceIdentifier"> <xs:annotation> <xs:documentation>The descriptive identifier of the target Component interface that is required</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
--	--


element ServiceMesh/Component/Capability

diagram	
properties	<pre> isRef 0 minOcc 0 maxOcc 1 content complex </pre>
children	Scaling
annotation	<pre> documentation The element that encapsulates Runtime capabilities of the Components that are considered inherent </pre>
source	<pre> <xs:element name="Capability" minOccurs="0"> <xs:annotation> <xs:documentation>The element that encapsulates Runtime capabilities of the Components that are considered inherent</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="Scaling" minOccurs="0"> <xs:annotation> <xs:documentation>Scaling can be Horizontal (if the Component is completely stateless), Vertical if the Component is statefull or Diagonal (i.e. both) in case of stateless Components</xs:documentation> </xs:annotation> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="HORIZONTAL"/> <xs:enumeration value="VERTICAL"/> <xs:enumeration value="DIAGONAL"/> </xs:restriction> </xs:simpleType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>

element ServiceMesh/Component/Capability/Scaling

diagram	 <p>Scaling can be Horizontal (if the Component is completely stateless), Vertical if the Component is statefull or Diagonal (i.e. both) in case of stateless Components</p>
type	restriction of xs:string
properties	isRef 0 minOcc 0 maxOcc 1 content simple
facets	enumeration HORIZONTAL enumeration VERTICAL enumeration DIAGONAL
annotation	documentation Scaling can be Horizontal (if the Component is completely stateless), Vertical if the Component is statefull or Diagonal (i.e. both) in case of stateless Components
source	<pre><xs:element name="Scaling" minOccurs="0"> <xs:annotation> <xs:documentation>Scaling can be Horizontal (if the Component is completely stateless), Vertical if the Component is statefull or Diagonal (i.e. both) in case of stateless Components</xs:documentation> </xs:annotation> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="HORIZONTAL"/> <xs:enumeration value="VERTICAL"/> <xs:enumeration value="DIAGONAL"/> </xs:restriction> </xs:simpleType> </xs:element></pre>

element ServiceMeshIdentifier

diagram	 <p>The descriptive identifier of the vertical application. It will be used by the repository for indexing purposes</p>
type	xs:string
properties	content simple
used by	elements ServiceMesh SliceIntent
annotation	documentation The descriptive identifier of the vertical application. It will be used by the repository for indexing purposes
source	<pre><xs:element name="ServiceMeshIdentifier" type="xs:string"> <xs:annotation> <xs:documentation>The descriptive identifier of the vertical application. It will be used by the repository for indexing purposes</xs:documentation> </xs:annotation> </xs:element></pre>

