TNOVA

NETWORK FUNCTIONS AS-A-SERVICE
OVER VIRTUALISED INFRASTRUCTURES

GRANT AGREEMENT NO. 619520

Deliverable D3.01

Interim Report on Orchestrator Platform Implementation

Editor José Bonnet (PTInS)

Contributors J. Bonnet, P. Neves (PTInS), M. McGrath, G. Petralia, V. Riccobene (INTEL), P. Paglierani (ITALTEL), F. Delli Priscoli, A. Pietrabissa, F. Liberati, R. Gambuti, V. Suraci, L. Zuccaro, F. Cimorelli (CRAT), A. Ceselli, G. Grossi, F. Pedersini, M. Trubian (UNIMI), J. Ferrer Riera, J. Batallé (i2CAT), M. Di Girolamo, P. Magli, L. Galluppi, G. Coffano (HP), A. Lopez-Ramos (ATOS), D. Dietrich (LUH)

Version 1.0

Date 22nd December, 2014

Distribution PUBLIC (PU)

Executive Summary

This deliverable presents the interim results of the four tasks in Work Package 3 of the T-NOVA project, the Orchestrator Platform Implementation.

Section 2 analyses the interfaces the Orchestrator Platform has with external systems, which is the focus of Task 3.1. These interfaces have two different functionalities: the interface with the Network Function Store and the Marketplace has to support a flexible form of defining new Network Functions and Network Services, while the other, with the Virtual Network Functions and the Virtual Infrastructure Manager, needs to support potentially very high data exchange rates. Some of the available technologies that appear to support these needs have been analysed, but further experimentation is needed before a definite conclusion can be reached.

Section 3 describes the activities of Task 3.2, Infrastructure Repositories, which is focused on exposing to the other subsystems of the T-NOVA Orchestrator infrastructure information available from the Infrastructure Virtualisation Management (IVM) layer. This information is crucial to the optimal allocation of resources (see Section 4). OpenStack and OpenDaylight provide important foundational technologies for the implementation of Cloud Computing and SDN Controller capabilities within T-NOVA. However, these technologies need to be appropriately integrated in the overall T-NOVA system by exposing the infrastructural components they manage and control. This is a combination of utilising default infrastructural information and further augmenting this information as necessary (e.g. Enhanced Platform Awareness). Finally the infrastructure information is exposed by using existing and new interfaces to provide a common view of the infrastructure landscape to the Orchestrator's subsystems. An initial prototypical solution has been designed and is being evaluated with a view to enhancing the solution to meet the needs and requirements of the various dependent tasks.

Section 4 outlines the activities Task 3.3, Service Mapping, has been focused on. There are different kinds of algorithms that allow the optimal allocation of resources to a given Network Service instance, which have been presented and compared. Additional work remains before choosing and implementing one of those algorithms (or a small set of) and integrate it (them) into the overall Orchestrator Platform.

Section 5 outlines the work carried out to date in Task 3.4, Service Provisioning, Monitoring and Management. The core of the Orchestrator Platform will be designed and implemented in this task, with new services being defined (with the agreed Service Level Agreements, SLA) and new instances, requested by a Customer at the Marketplace, through the defined interfaces (Section 2), being provisioned on the infrastructure described in the repository (Section 3) and according to an optimal mapping (Section 4). While running, every service instance is monitored, and eventually scaled or migrated, so that the agreed SLA is not breached, until the limit date is reached or the customer that requested decided to stop it. The work presented includes the initial network service descriptor, as the base data model for service provisioning, monitoring, and management, together with the high-level functional architecture to be implemented.

Finally, Section 6 presents the conclusions achieved from the work completed so far.

Table of Contents

1. INTRODUCTION

An Orchestrator Platform is a central technology component in enablement of Network Function Virtualisation (NFV) and Software Defined Networks (SDN) in carrier grade networks. The Orchestrator plays a key role in enabling performance, scalability, availability and openness. The adoption and roll out of Virtual Network Functions (VNFs) by operators has significantly lowered barriers for network Function Providers (FPs) to enter the telecommunications market, so Telecom Operators (Telcos) have no alternative but to open their infrastructures to these FPs. All these changes will have to occur while Telcos still have to reduce their capital expenditures in the face of an the exponential growth in data traffic while at the same revenues per megabyte continue to contract. The consequence of this business reality is the urgent need for infrastructures that are able to support these externally provided VNFs without jeopardizing the quality and security of current services. It is the Orchestrator's role to map new services' requests onto the existing infrastructure in an automatic, secure and efficient way, without ever being a business or operational bottleneck.

Work Package 3 is focused on the implementation, integration and testing of an Orchestrator Platform. In particular, this interim deliverable outlines the key activities and findings of the first six months of work towards this goal. The main features of the T-NOVA Orchestrator Platform that is currently being developed are as follows:

- Handles new or updated VNFs from the Network Function Store (NF Store), validate them and notifies the Marketplace of their existence and functional characteristics;
- Has the ability to receive new or updated Network Services (NSs), composed at the Marketplace level by the Service Provider (SP), from the available VNFs (including Service Level Agreements (SLA)) in the Marketplace catalogue;
- Receives NS instantiation requests from the Marketplace when the SP's customers 'buy' that NS;
- Determines the required resources for a NS instance from the composing VNFs' descriptors and the available infrastructure;
- Provisions, monitors and manages running NS instances, escalating or migrating them as required in order to maintain an associated SLA.

These features can only be accomplished by working collaboratively with the other platform implementation related Work Packages of the T-NOVA project:

- Work Package 4, Infrastructure Virtualisation and Management, for the allocation of the virtualized infrastructure needed for each of the NS instances and collection of the dynamic metrics of the running instances;
- Work Package 5, Network Functions, for the NF Store and the VNFs supporting the proposed use cases;
- Work Package 6, T-NOVA Marketplace, for the NS composition (based on the existing VNFs) and commercialization.

To support the implementation of these features, the work in the Work Package has been split into the following Tasks:

- Task 3.1, Orchestrator Interfaces, is focused in designing, implementing, testing and documenting the interfaces of the Orchestrator Platform;
- Task 3.2, Infrastructure Repository, gathers data provided by the infrastructure component controllers i.e. cloud compute and network resources, and exposes this information via a common set of interfaces to the Orchestrator;
- Task 3.3, Service Mapping, determines the best mapping between the requested service instance and the available infrastructure both locally and across the available NFVI-PoP's;
- Task 3.4, Service Provisioning, Monitoring and Management, is focused in designing, implementing, testing and documenting the core features of the Orchestrator Platform that perform the remaining features described above.

This deliverable is organized in alignment with the tasks, describing in Sections 2 to 5. The key findings and work carried for each one of them is described. Section 6 presents the key conclusions from the work to date.

2. ORCHESTRATOR INTERFACES

This sub-section summarizes the work carried out to date in **Task 3.1, Orchestrator Interfaces**.

The initial step was to define the problem statement that Task 3.1 needs to address. A variety of technologies that may provide a viable a solution to the problem statement were then investigated. The criteria used to evaluate these technologies in order to select the most appropriate ones are also presented. Finally, the initial conclusions identified by the task are presented.

2.1. Problem Statement

Due to its pivotal role in the T-NOVA architecture, the Orchestrator implements appropriate interfaces to manage the interaction with the layers above and below it. Specifically, the Orchestrator provides:

1. A Northbound interface to the **Marketplace** and the **Network Function Store**;
2. A Southbound interface to the IVM. This interface will support the exchange of metrics data generated both at the infrastructure level and at the VNF/NS level. These metrics have to be collected (and transposed) and communicated to the Orchestrator in order for the Orchestrator to identify and inform the IVM what actions are required to be taken so that the NS SLA is maintained.

Flexibility is a key feature in the Northbound Interface due to the need to define new Network Services (NSs), from existing VNFs, while the Southbound interface needs to consider carrier grade requirements, which are required to deal with large amounts of infrastructure data and infrastructure failures.

2.1.1. Orchestrator Interfaces Basic Features

These characteristics can be translated into the following basic set of features [1][2]:

- Significant **flexibility**, so that new sets of metrics can be defined for monitoring new NSs and associated SLAs;
- Low **latency**, to minimize the response time once an error condition is detected or a threshold condition is exceeded;
- High **scalability**, in order to accommodate different scenarios for the VNFs provided;
- High **resiliency** to (infrastructure) failure or performance degradation (due to failure or overload), in order that the correct action is still taken even if not all the information is always available.

These features are present in many systems and are generically known as **Streaming Data Processing Systems** (see Figure 2-1 below). Systems with this capability have been designed and implemented as real-time alternatives to Hadoop [3], the open-source and batch processing [4] based on the implementation of Google's MapReduce [5] algorithm. For these systems, **message processing** is a fundamental

paradigm for real-time computation [4], although managing the associated queues and workers (this is the specific code to process each message) in large scale and fault tolerant scenarios is very complex.

2.1.2. Streaming Data Processing Systems' Architecture

Figure 2-1 shows the typical architecture of a Streaming Data Processing System.

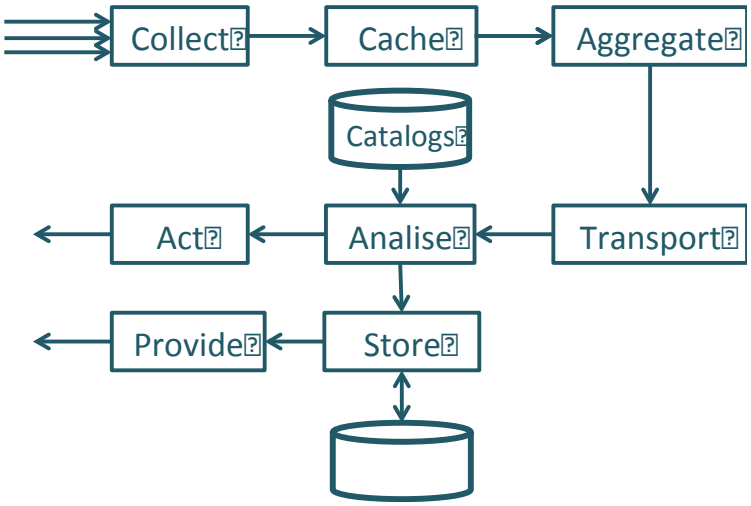

Figure 2-1: Typical architecture of a Streaming Data Processing System.

The key functional blocks of the architecture are as follows:

- **Collect:** the point-of-entry module where streaming data is inputted;
- **Cache:** where different metrics with distinct generation rates are stored;
- **Aggregate:** where different collected values may be aggregated or enriched. The data may also be processed in some manner such as the calculation of running averages;
- **Transport:** contains the necessary logic regarding the transportation of the distributed data (depending on different implementation options, data transport may also occur between other blocks of this architecture);
- **Analyse:** Data stored in **Catalogues** (e.g., SLAs) can be compared to received data;
- **Store:** in memory or on disk, where data is fetched from;
- **Act:** Module responsible for the (request for) execution of all actions (scale, migrate, etc.);
- **Provide:** this module contains the logic required to push stored data to other consumer systems.

2.1.3. Orchestrator Southbound Interfaces Requirements and Architecture

The Orchestrator Southbound Interfaces can be split in two key functions:

- The interface with the **VIM;**

- The interface with the **VNFs**.

The T-NOVA Orchestrator comprises a real Orchestrator as well as a VNF Manager. Some VNFs, because of their proprietary nature, performance needs, etc., may have their own VNF Manager (to be provided by the Function Provider in parallel to the uploading the VNF to the NF Store), so the interface between the real Orchestrator and the VNF Manager will be designed accordingly later in the project.

Both of these interfaces have their own specific requirements, which are follows.

2.1.3.1. Requirements Analysis

The requirements for these interfaces are as follows (see [6]):

- **VIM**:
  1. Reserve or release the required infrastructure needed for a VNF;
  2. Allocate, update or release the infrastructure needed for a VNF;
  3. Add, update or delete a SW image (usually for a VNF Component);
  4. Collect infrastructure utilization data (network, compute and storage;
  5. Request infrastructure's metadata from the VIM;
  6. Manage the VMs allocated to a given VNF;
  7. All the interfaces between the Orchestrator and the VIM SHALL be secure.
- **VNFs**:
  1. All the interfaces between the VNFM and the VNF SHALL be secure;
  2. Instantiate a new VNF or terminate one that has already been instantiated;
  3. Retrieve the VNF instance run-time information (including performance metrics);
  4. (Re-)Configure a VNF instance;
  5. Collect the current state or request a change in the state of a given VNF (e.g. start, stop, etc.);
  6. Request the appropriate scaling (in/out/up/down) metadata.

These requirements are represented in

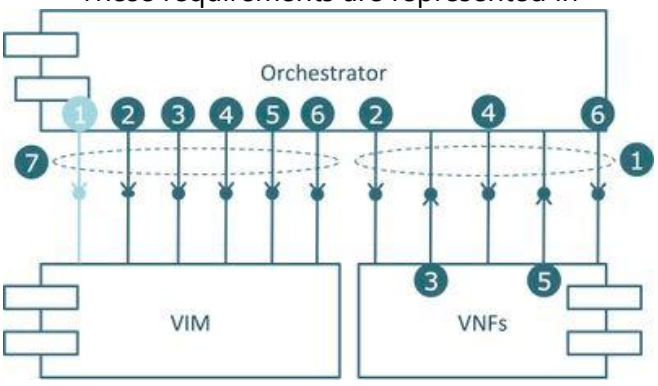

Figure 2-2.

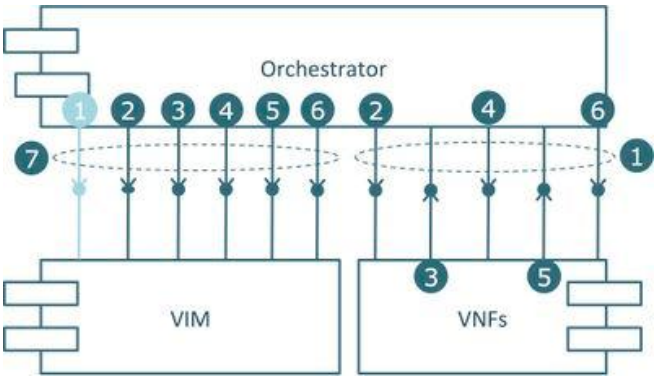

Figure 2-2: Requirements for the Interfaces between the Orchestrator and the VIM and VNFs.

Analysis of the requirements for the interface with the VIM identified the following observations. Resource reservation and release (**#1**) has not yet been implemented within the OpenStack framework, and is therefore not considered for the initial version. Allocating, updating and releasing the infrastructures required for a VNF (**#2**) will have to be designed, since the current default behaviour of OpenStack is not suitable for VNF/NS deployment. OpenStack's **Glance API** [7] will be used for the VM images management (requirement **#3**). Infrastructure information will be available in the Infrastructure Repository (see Section 3), which will expose APIs to collect and request infrastructure metadata (Requirements **#4** and **#5**). The VMs will be managed using the OpenStack's Compute API v2 [8] (Requirement **#6)**. Finally, as to securing all the interfaces (**#7**), that issue will be addressed when the output from the other work packages have developed to the point where there is sufficient visibility on any necessary requirements.

As to the **VNFs** requirements, work on securing all the interfaces (**#1**) will start when the work with the other Work Packages is more mature and needs and potential impacts can be better understood. Instantiating and terminating a VNF (**#2**) implies having the optimal Data Centre and Networking location for all its components (see Section 4, Service Mapping The VNF Manager requests the required infrastructure from the VIM as well as starting the VNF, or terminating it and requesting the VIM to release unused infrastructure as it becomes available. Retrieving the VNF instance run-time metrics (requirement **#3**) will probably be transformed into a 'from the **VNF** to the **VNF Manager'** interaction. Reconfiguring a VNF (requirement **#4**) depends on an appropriate solution being identified for the interface with VNFs (see Work Package 5's work). Collecting the current state or request a change in the state of a given VNF (e.g. start, stop, etc., requirement **#5**) is a bi-directional interface: the VNFM may want to request the VNF to change its current state (e.g., from '**running'** to '**stopped'**), however the VNF itself may want to communicate its change of state directly to the VNFM. As with requirement **#4**, its design also depends on the overall design approach for the interface with the VNF.

See Section 9, Annex A, for further details on this interface.

2.1.3.2. Proposed Supporting Architecture

As the requirements outlined in section 2.1.3.1 include the need to support event stream analysis in order to initiate appropriate actions, the project believes that Streaming Data Processing Systems, with an architecture such as the one shown in Figure 2-1, are a potential solution to the Orchestrator's problem.

Some blocks from the high level architecture may be included in an interface layer, serving the other blocks, more at the core level of the Orchestrator. A possible separation of those blocks is shown in Figure 2.3.

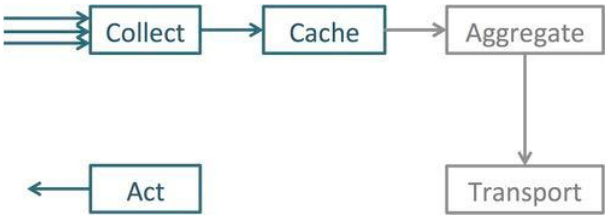

Figure 2-3: Blocks of a Streaming Data Processing System architecture to be considered as part of the interface layer.

The Aggregation and Transport blocks may or may not be considered to be part of the Orchestrator Interfaces module. The reasoning behind this division is as follows: **collection** and **caching** of data must be carried out in close proximity to data source(s), data **aggregation** (without any **enrichment**) may be carried out at this level if necessary, as well as **transportation** (this strongly depends on the level of distribution chosen for the implementation). Since **enriching** the data, implies adding dimensions like 'Network Service ID' to the data stream, doing it at this level may require implementation of too much logic at this layer. It is therefore proposed to include it only at the core layer.

Acting, depending on the technological details, may or not have limited functionality. The types of potential functionality include adapting between two different technologies (undesirable, but possible), or providing authentication/authorisation services.

2.1.4. Orchestrator Northbound Interfaces Requirements and Architecture

The Northbound Orchestrator Interfaces pose a different set of challenges, in comparison to the Southbound Orchestrator Interfaces described previously.

There are two functional entities 'north' of the Orchestrator, which it needs to interface with:

- The **Network Function Store** (NF Store):
  1. Notifies the Orchestrator about new, updated or deleted VNFs available in the NF Store;
  2. Provides VNF Descriptor (VNFD) upon request;
  3. Provides VNF Images upon request;

- The **Marketplace**:
  1. Notified about new, updated or deleted VNFs available in the NF Store;
  2. Notified about (at least part of) the VNFDs of the available VNFs;
  3. Notifies the orchestrator about new, updated or deleted Network Services (NSs);
  4. Notifies the orchestrator to instantiate and deploy an existing NS;
  5. Notifies the orchestrator about new configuration parameters for an already deployed NS;
  6. Inquiries from the orchestrator about the state of a given NS instance;
  7. Notified about changes in state of currently deployed NSs;
  8. Notified with currently running NS metrics;
  9. Notifies the orchestrator to stop a given NS instance;

These requirements are shown in Figure 2.4.

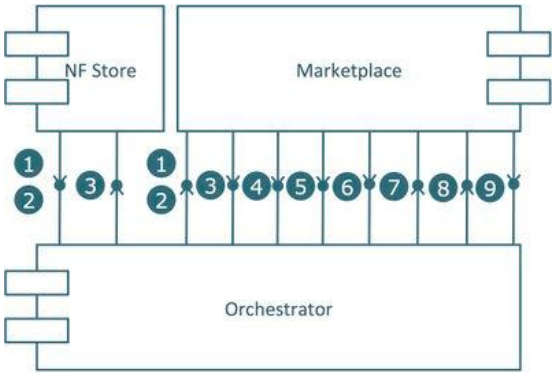

Figure 2-4: Requirements for the Interfaces between the Orchestrator and the NF Store and the Marketplace.

These requirements have been previously defined in D2.31 [6]. Simplification of some requirements is possible, for instance, having the NF Store send the VNFD to the Orchestrator (requirement #2) together with the notification to the Orchestrator about a new or updated VNF (requirement #1). The same simplification can be applied to the interface between the Orchestrator and the Marketplace (also requirements #1 and #2): the Orchestrator will pass the VNFD (or a sub-set/super-set of it, according to further analysis that still has to be done) to the Marketplace when notifying it about new or updated VNFs. Please note that the deletion of a VNF may not only imply its removal from the catalogue of available VNFs but if it is part of a NS running instance, its removal when that instance is stopped.

For the **Marketplace** requirement #3, a good starting point is the **ETSI MANO's NSD**. This NSD should include all the metrics involved in the SLA, which will be interpreted by the Orchestrator (see Section 5).

From the list above, #6 and #7 can be merged, if it is assumed that the Orchestrator will always push changes in NS status at a higher frequency than is required by the Marketplace.

There is also another issue with this interface that requires attention: if the VNF images are too large, it may be necessary to consider storing and retrieving VNF Component images instead. This makes the Orchestrator more complex, however this maybe the only suitable solution to address this problem.

See Section 9, Annex A, for further details on this interface.

2.1.5. Orchestrator Interfaces Sub-Problems

The Orchestrator Interfaces' problem can be divided into the following sub-problems:

1. **Interface definition:** what values to provide or retrieve and to/from where;
2. '**Collect', cache and aggregate/enrich large volumes of data:** even having data **pushed** into the Orchestrator from the producing systems ('keep data moving' requirement mentioned in [1]), it must accommodate different data generation rates and failures, and also aggregations and enrichments of the collected data, like adding the NSs dimension to the collected metrics, in order to pass it to the Orchestrator;
3. **Store and provide large volumes of data:** store the enriched data, for the Marketplace to consume it;

In the next sub-sections, the possible solutions for each one of these sub-problems are analysed.

2.2. Candidate Solutions

In this section, for each of the sub-problems identified in the previous section, candidate solutions are analysed and compared.

2.2.1. Interface Definition

For **inbound** requests (#1 and #2 from the NF Store and #3 to #6 and #9 from the Marketplace, above) a common **request router** receives all inbound requests and passes them to the most adequate component to process them as shown in Figure 2.5.

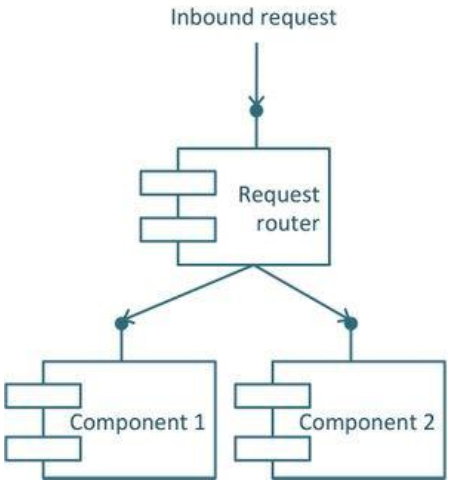

Figure 2-5: Request router proposed architecture.

The interface for the **outbound** requests will be designed with cooperation with the other sub-system implementers.

Following a **REST** [7] approach, resources are accessed as follows:

```
http(s)://.../virtual-network-functions
http(s)://.../network-services
```

With this approach, requirements such as (the Marketplace) notifying the Orchestrator about new, updated or deleted Network Services (requirement #3) could be simplified to a REST POST call for creating a new NS (with the data being defined in the body of the request, not shown):

```
http(s)://.../network-services/
```

For the same requirement, updates would be carried out using a REST PUT call (with the data being defined in the body of the request, not shown), like:

```
http(s)://.../network-services/<ns-id>
```

Deleting is a REST DELETE call, in the form of:

```
http(s)://.../network-services/<ns-id>
```

The required state (or further information for this operation is needed, e.g., the number of milliseconds until the execution of the operation) is inserted into the body of the request.

Opting for a HTTP REST API will also allow the selection of fields to be returned from a given resource. For example, for the Marketplace to inquire the Orchestrator about the state of a given NS instance (requirement **#6**), a REST GET call, selecting the **status** attribute (see [10]) can be specified as:

```
http(s)://.../network-services/<ns-id>/?fields=status
```

Standards [11] and best practices in writing JSON APIs [10] will also be taken into account, thus simplifying future reuse of the specified APIs.

2.2.2. Data Streaming

For data streaming, **Apache Storm** [12] and **Apache Spark Streaming** [13] with their batch-oriented design, and **Apache Samza** [14], with a message-by-message streaming processor, offer potential solutions.

2.2.2.1. Apache Storm

Apache Storm is the **Hadoop** (batch processing) for real-time computing systems, keeping the former's distributed features, as well as fault-tolerance, scalability, robustness, etc. It can be used with any programming language as long as **Thrift** [15], a software framework that combines a software stack with a code generation engine to build services that work efficiently and seamlessly between different languages (C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages), is used to define the interface.

Storm uses its own terminology and concepts, such as:

- **Spouts:** a source of streams, such as the dynamic metrics coming from a VM, typically reads from a queuing broker (e.g., RabbitMQ [16], Kafka [17], etc.)

but can also generate its own streams or reads from third party streaming API's such as Twitter's [18][19];

- **Bolts:** processes any number of input streams and produces any number of new output streams. Most of the logic of a computation goes into bolts, such as functions, filters, streaming joins, streaming aggregations, talking to databases, etc.;
- **Tasks:** an instance of a **bolt** or a **spout**;
- **Topologies:** a network of spouts and bolts, with each edge in the network representing a bolt subscribing to the output stream of some other spout or bolt.

One drawback of Storm is that the **topology has to be predefined in code**. If the topology needs to be created dynamically, for instance, to process a newly defined NS metric, this may be an issue. Dynamically generating the code needed for the necessary topology would be required. However implementation of this capability may introduce another problem, which is the need to interrupt the service in order to rebuild the topology. A solution to this latter problem would therefore have to be designed as well: load balancing the old-topology version and the new approach. Further research is required on this topic.

Another drawback is **latency** [20]: *...sub-second latency...* responses to metrics exceeding threshold values may be difficult for the Orchestrator to achieve.

When using Storm, messages can sometimes be **duplicated**: which might be a problem, especially when there is a need to maintain **state** (e.g., when calculating **moving averages**).

Storm has been used, as the processing engine, in a recent project called Monasca [21], a monitoring-as-a-service, multi-tenant, REST API based framework, which will be integrated into OpenStack. Monasca is designed with most of the features required by the T-NOVA Orchestrator interfaces (real time processing, scalability, fault tolerance, big data retention); therefore Storm emerges as one of the most viable choices, in spite of the drawbacks identified above. It includes Kafka as the message queue middleware and is based on a micro-service message bus for module interconnections. Monasca REST API provides metric management, alarm definition templates and notification mechanisms. Monasca implements real time anomaly detection, performing up to 150K metrics/sec for a three-node cluster with load balancing Virtual IP [21].

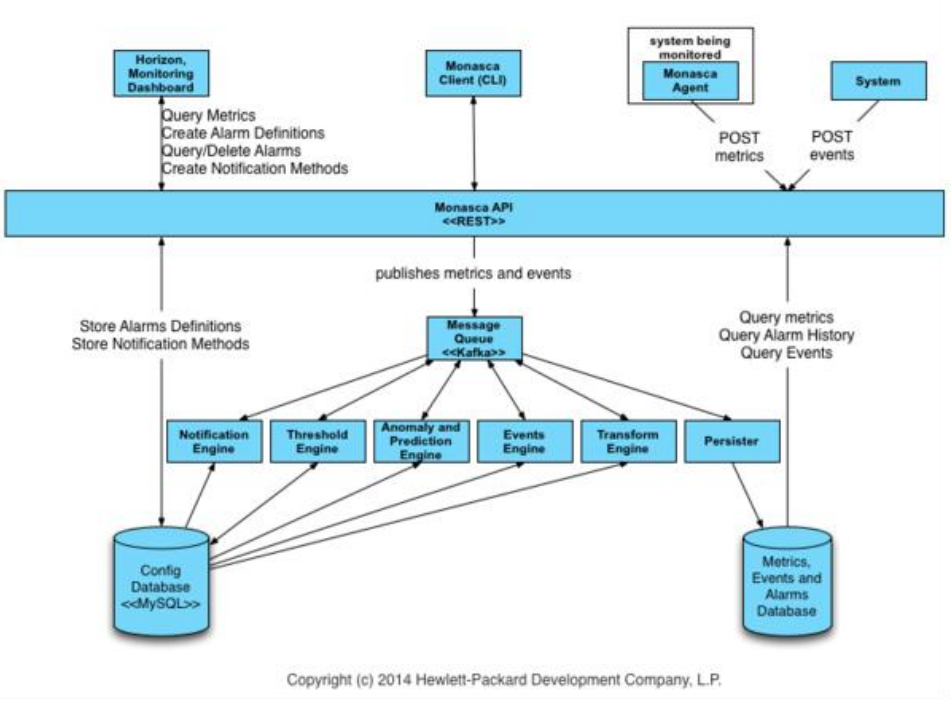

Figure 2-6: The Monasca high-level architecture.

An interesting point with respect to Monasca is that it selected Storm to implement its real time monitoring and alarm detection, therefore its latency figures were not considered to be a major limitation. Also, considering Monasca's tight integration with OpenStack, makes the Storm option far more appealing, and probably worth considering as the first choice for the Orchestrator's API implementation.

2.2.2.2. Apache Spark Streaming

Spark Streaming uses the core of **Apache Spark** API, which is especially important if there is a need to store large volumes of data as well as processing them. This is a key requirement for Orchestrator especially if metrics are to be processed and provided to the **Marketplace.**

Spark Streaming's streams are groups of batches with a fixed duration (e.g., 1 second). Each time limited batch is called a **Resilient Distributed Dataset** (RDD), which is called a **Discretized Stream** (DStream) when is part of a repetitive sequence. Received data (not yet a RDD) is stored in **Spark**, and is later transferred into a DStream, where it is either transformed or outputted.

Spark Streaming's deployment uses a Streaming Context object in the driver program to talk to a Resource Cluster Manager (such as Apache YARN [22] or Mesos [23]) and allocate **executors**. These executors run the data receiving or the data processing tasks previously outlined.

Latency when using **Spark Streaming** is even higher than **Storm**'s single digit seconds performance [20]. This is due to its batch oriented design.

2.2.2.3. Apache Samza

LinkedIn [24] released **Apache Samza**, another stream-processing framework, into open source community, through the Apache Foundation. The most significant difference from the previously described frameworks is Samza's ability to process messages on serial temporal basis (not in batches), " an immutable unbounded collection of messages of the same type" [25] called a **stream**. This feature allows "low millisecond" latency values [26], which satisfies the needs of the T-NOVA Orchestrator.

Messages in each of these streams can be read by many **jobs**, a logical collection of processing units, or **tasks**, which can produce other messages into output streams. Each job defines its **source** and **destination** streams (the **topology**). Scalability is achieved by partitioning a stream into sub-streams, "a totally ordered sequence of messages" [25], each stream processed by one of the tasks running in parallel. Tasks "can consume multiple partitions from different streams" [25].

Samza's tasks run on a cluster managed by Apache YARN, which consume and produce Apache Kafka streams (or **topics**). These streams are always persisted in a distributed way by using Kafka's topic partitioning feature: messages with the same key will always belong to the same partition. By default Kafka stores all messages in the file system and only deletes them after a pre-configured amount of time, which allows consuming tasks to consume messages at arbitrary points along the stream if they need to. Mirroring Kafka to HDFS is simple [26], but optional: in real scenarios this could be switched on, but left off for simpler scenarios (such as a demo), thus saving on required infrastructures. Kafka itself is not consensual [27, 28], especially at Operations: it is seen as difficult to manage and introduces extra precious (latency) time when allocating new workers.

Samza's tasks are written using the **Java** programming language [29]. This code specifies how to process the messages. The main task is configured in a comprehensive properties file. Property files along with the compiled code are submitted to the Yarn Samza cluster.

2.2.2.4. Bespoke Data Streaming Implementation

How would the problem of data streaming be addressed if none of the frameworks described in the previous sections existed, or if there was a requirement to allocated limited resources to the solution design?

This section describes the eventual solution that could be built by this project.

The bespoke or 'built on purpose' approach has been described by Stonebraker et al. as [1]:

...manually build a network of queues and workers to do real-time processing. Workers would process messages off a queue, update databases, and send new messages to other queues for further processing.

But that same reference also highlights the disadvantages of such a solution:

- It is **tedious:** most of the development time is spent configuring where to send messages, deploying workers, and deploying intermediate queues, leaving a relatively small percentage of time to design, implement and support the real-time processing logic;
- It is **brittle:** fault-tolerance has to be designed and implemented to keep each worker and queue up;
- It is not **easily scalable:** to increase throughput data has to be spread around new workers and queues, and the existing workers have to be reconfigured to know the new queues where to send messages.

Other alternatives, such as the ones described in [27] or [28] will also be further analysed.

2.2.2.5. Other Approaches

One new approach to the problem of real-time stream processing is called the **Lambda Architecture** [30]. In this architecture two paths for data are considered: one more **batch-oriented**, having more time to be calculated from base structures like Hadoop, and a parallel one, which is more **real-time** oriented, calculating the results of the queries based only on the most recently available data. This has the enormous advantage of reusing knowledge and tools from the Hadoop world, but needs a duplication of the logic in the two data paths. Depending on the specific use case, this may or may not be a problem.

A recent entrant into this field is **Google Cloud Dataflow** [31], which is still too immature at this stage to be included however it will considered as the T-NOVA project develops.

2.3. Solution Rationale

The following sections described the solutions for the T-NOVA Orchestrator interfaces and the supporting rational.

2.3.1. Interfaces Definition

As outlined in Section 2, interfaces between the different T-NOVA sub-systems require both **flexibility** in their definition, to support, e.g., new NSs composed of at least one VNF, and **efficiency** in terms of resource usage, due to the expected high volume of traffic (especially in the **Southbound Interfaces**). The current trend [32] is to use a **REST** architectural style with **JSON** over **HTTP**, instead of **WS-*** [33], a more **Remote Procedure Call**-based architectural style, using **SOAP** and **XML** also over **HTTP**.

Other approaches like designing a proprietary solution from the ground-up would take the project too much time and resources, and will not be pursued at this stage. Using lower level solutions (e.g., Protocol Buffers [34] or Message Pack [35]) might be needed if performance is not at the required level.

2.3.2. Data Streaming

Table 2-1 summarises the potential solutions to the T-NOVA interface implementation

Table 2-1: Summary of potential solutions for supporting a Streaming Data Processing System architecture.

	Storm	Spark Streaming	Samza	Built	Description
Origin	Twitter [36]	UC Berkeley [37]	LinkedIn [24]	T-NOVA	The entity responsible for the solution
Tech. Stack	JVM [38]/Clojure [39]	JVM/Scala [40]	JVM/Scala	TBD	Technological stack/programming language used in the framework's implementation
API Language	Java	Scala, Java	Java	TBD	Technological stack/programming language must be used
Batch Framework	N/A	Spark	N/A	N/A	Permanent message store. In the T-NOVA case the Orchestrator must provide metrics to the Marketplace
Processing model	One record at a time	Mini-batches	One record at a time	TBD	Form of consumption of the available information
Latency	Sub-second [20]	Few seconds [20]	Low milliseconds [25]	TBD	Average delay between event occurrence and event availability for analysis. As the Orchestrator requires a fast response time **low latency** is required.
Fault tolerance (every record is processed...)	At least once (may have duplicates)	Exactly once		TBD	Options for addressing failures
Form of	Task	Data	Task	TBD, could	Scalability, in order

parallelism	parallelism [41]	parallelism [42]	parallelism	mix both approaches	to process more input

2.4. Recommendation

The analysis outlined previously (see Section 2.2) highlights the existence of a dual level of functional performance to be guaranteed at the Northbound and Southbound **Orchestrator** Interfaces. The northbound interface, interacting with the **Marketplace** and the **NF Store**, must handle lower data throughputs, and less stringent requirements of real-time response. The southbound interface, on the other hand, needs to interact with the underlying infrastructure, which implies that:

- The data exchanged over the interface can be characterised as real-time streaming;
- Data volume and throughput are higher (more fine grained components, larger amount of generated data);
- Temporal responsiveness is more critical; on one hand to guarantee that no data are lost, on the other hand to ensure prompt response in cases where failure is detected or a security breach identified.

The Orchestrator adds a data mediation and enhancement dimensionality, reconciling the raw data pushed by the southbound interface (metrics measures) with the service to which the measured data are associated with, a logical connection not captured at the VNF level. The selected technical solution must thus take into account these different requirements. It is not guaranteed that a unique platform can meet the demand of both the interfaces. The southbound interface poses the most challenging requirements, and accordingly a more challenging choice.

In the Southbound Interfaces, data volume, velocity and variance are part of the problem characterisation, hence it is natural to look at Big Data platforms, narrowing the scope to the ones providing real-time processing capabilities as well as low latency data passing mechanisms, in line with the Streaming Data Processing Systems reference model. The previous sections discussed three options emerging from the Apache community, where Samza is the one natively streaming oriented, Spark Streaming and Storm have their origins in a batch processing oriented design. All these frameworks are written in Java: hence a RESTful API design using JSON over HTTP is an appropriate option.

Immediate considerations would lead to the selection of Samza as the first choice. The most important factor relates to latency: according to the available specifications, Samza is the only solution able to achieve low-millisecond single message processing performance. Its ability to run with Yarn and Kafka gives flexible file system integration, including HDFS mirroring, and accordingly good offline data processing options.

Nonetheless, Storm being a flavour of Hadoop for real-time oriented systems, it retains many features from Hadoop (distributed/parallelized processing model, intrinsic fault tolerance and scalability). The key features of Storm are **static topology** and (relatively) high **latency**, which may force the project to:

- Design a solution to support more flexibility in defining new topologies (e.g., to provide the Marketplace with a new NS metric that has to be composed from the available metrics on the VNFs that are part of that NS);
- Design a solution that will have to anticipate actions further ahead in time, to circumvent the delay from the higher latency.

But since Monasca is is using Storm for a similar problem, the project will have to further investigate these issues before taking a final decision on using it or not.

Other options had been outlined in the previous sections. Spark Streaming is appealing due to its ability to concurrently manage intra-orchestrator metric processing and data storage. However, its native batch-oriented design makes its latency significantly inferior to the one achieved by Samza or Storm, and potentially rules it out as a viable solution option.

The lambda architecture is interesting due to its reuse of significant Hadoop functionality, but it may pose some risk due to a more complex, double-path design. It's not evident if this design complexity outweighs the advantages of Hadoop technology reuse. This could be a possible contingency solution for the project's second iteration, if the results of the first trials demonstrate serious shortcomings in the selected option.

A new custom queue-worker solution can be tailored optimally to the T-NOVA functional requirements. However, rebuilding from scratch the whole framework is quite complex, error-prone, and, to be fully implemented, it will likely require time and resources beyond the current T-NOVA scope. Considering that T-NOVA is a research project, first and foremost aimed at proving the effectiveness of its concept, it makes sense to seek a first-step solution based on existing frameworks. A custom solution could be an interesting direction to follow as future work in a project follow-up.

After a careful analysis of the available data, the project hasn't made a clear decision on which Streaming Data Processing platform is the most appropriate choice. The Monasca project shows how Storm can be used to address a very similar problem. A final decision on this subject will require experimental evaluation of a number of candidate options before selecting the most appropriate one.

2.5. Relationship and Inter Task Dependencies

The dependencies of this Task3.1, Orchestrator Interfaces, towards other tasks are listed in Table 2-2.

Table 2-2: Inter-tasks dependencies from Task3.1, Orchestrator's Interfaces.

Task	Dependency Description
Task 3.2: Infrastructure repository	This task will enable an understanding of the static infrastructural metrics that will be available and how these metrics will be made available to the Orchestrator.
Task 3.3: Service mapping	This task will enable an understanding of

	how the Orchestrator can call the Service Mapping implementation and with which parameters.
Task 3.4: Service Provisioning, Monitoring and Management	Know how the Orchestrator's core components could be called for inbound requests and call the outbound requests.
Task 4.4. Monitoring and Maintenance	This task will provide the Orchestrator with dynamic infrastructure related metrics based on a push approach.

2.6. Conclusions and Future Work

The design of the Orchestrator's interfaces has started, taking into account the specific needs of each one of those interfaces.

The definition of the Northbound Interfaces is reaching a degree of maturity at the current stage of Task 3.1 activities. However, with respect to the Southbound Interfaces, further work, both within the scope of Task 3.1, Orchestrator Interfaces, and within the other tasks of the work package, is required to clarify and further elucidate some of the requirements. These clarifications, together with further experimentation of some of the possible solutions mentioned above will lead to clearer choices.

3. INFRASTRUCTURE REPOSITORY

Task 3.2 is focused on the implementation of a resource discovery and repository subsystem for the T-NOVA Orchestrator. This subsystem comprises of a number of key elements and capabilities including (i) an information model; (ii) resource information repositories; (iii) access mechanisms to the information repositories; (iv) enhancement of the information repositories provided by cloud and SDN environments and (v) a resource discovery mechanism. In addition the task is investigating the implementation of a network topology visualisation capability for the T-NOVA Orchestrator. Collectively these elements will provide detailed information on the resources and their characteristics to the Orchestrator. The Orchestrator utilises this information to reason over what collection of resource types need to be provisioned by the cloud environment for different types of VNFs within the T-NOVA system. The Orchestrator sends requests to the T-NOVA IVM to provision the required VM resources.

In order for the VNFs to approach a performance close or similar to the one of the counterpart hardware implementations, appropriate exploitation of platform features, in terms of both hardware and software, within the NFVI environment is critical. However the NFVI environment needs to be aware of such features and attributes by first discovering them and then scheduling their usage during VM instantiation. For example some VNFs can be characterised by intense I/O requirements and could benefit from the ability to access high performance packet processing capabilities such as Data Plane Development Kit (**DPDK**) software libraries and DPDK/Single Root I/O Virtualization (**SR-IOV**) compatible network interface cards.

Task 3.2 has a strong inter-relationship with Task 4.1 in terms of the technology choices that will be investigated for the implementation of the T-NOVA IVM. Within Task 4.1, OpenStack has been selected as the candidate technology for the cloud controller platform, while OpenDaylight has been selected as the SDN network controller.

It is worth noting that the current OpenStack API and Scheduler only supports limited enhanced platform features e.g. CPU flags [43]. Therefore, a mechanism is required to appropriately control VM placement among the available hosts within the T-NOVA infrastructure to increase the Orchestration capabilities for an intelligent placement of VNFs on appropriate target compute platforms (within the same Point-of-Presence). Identification of appropriate platform features to expose together with investigation of potential exposure and utilisation mechanisms is being carried in Task 4.1. Whereas Task 3.2 is focusing on the implementation aspects allowing the cloud scheduler to effectively use platform information beyond what is available in the OpenStack Icehouse release which was utilised in the preparation of this deliverable. Task 3.2 will continue to monitor and utilise new platform information as they are made in the OpenStack releases that will occur during the duration of this task, namely Juno and Kilo.

Initial work has been focused on determining what infrastructure information is available from the candidate technologies (OpenStack and OpenDaylight), what are the available mechanisms to access the information and what are the current resource information gaps, with a specific focus on platform features (e.g., the

availability of PCIe devices). In addition focus has also been given to identifying possible approaches to exploiting this information within OpenStack.

Additional work is being carrying out in cooperation with Task 3.3. This task will utilise the platform features information as an input in order to make decisions about the optimal deployment of VMs. A number of activities are on-going to align the available platform information with the input data needed by the resource mapping algorithm, identify information gaps and to determine potential approaches for obtaining the missing information.

3.1. Relevant Initiatives for Infrastructure Data Modelling

During the design and development phases of the T-NOVA Infrastructure Repository, some relevant industry and standards initiatives have been identified and analysed in order to capture the current state of the art in the Infrastructure Data Modelling. Four key initiatives have been identified and are briefly discussed.

3.1.1. Redfish

"Redfish is a modern intelligent manageability interface and lightweight data model specification that is scalable, discoverable and extensible."

Redfish [44] is a specification under development by Intel, HP, Emerson and Dell for Data Centre (DC) and system management that is focused on the achievement of improved performance, functionality, scalability and security. Redfish provides a rich set of information in human-readable format that can be easily used by DC's administrators in their remote management scripts. The specification is designed to improve scalability and expand data access and analysis and enable feature-rich remote management while protecting data at a high level supporting secure HTTP communications. Different Redfish communications can be executed in parallel. Redfish encompasses efficient cross-platform connections among various types of servers, operating systems, networks and storage. Some common aspects with the T-NOVA Data Model requirements can be identified, as Redfish is focused on the modelling of key of Data Centre physical resources features. In particular Redfish's resources categorisation has relevance for the T-NOVA data model design.

3.1.2. IPMI

The "Intelligent Platform Management Interface" (IPMI, [45]) is a series of specifications that defines a set of standardised interfaces to provide management and monitoring capabilities to servers, independently from their hardware characteristics (CPU, firmware and operating systems). Using IPMI the system administrator can monitor servers, before the OS has booted, when the system is powered down or after a system failure, sending IPMI messaging to the platform in order to obtain information produced by host sensors (temperature, voltage, fans, power supplies). IPMI provides only the specification for the interfaces, while there

are many different implementations. DCMI is an extension of IPMI specifically designed for DC management. IPMI provides a standard definition of low-level infrastructure information as sensor data and events log.

3.1.3. Desktop Management Interface

The Desktop Management Interface (DMI) is an industry framework for managing and monitoring hardware and software components in a system of personal computers from a central location [46]. The standard was created by the Distributed Management Task Force (DMTF) to automate system management. Each computer is described through a MIF text file (Management Information Format). The MIF includes both hardware and software information and contains information such as product name, version, serial number and timestamps. Manufactures can create their own MIFs specific to a component. The definition of resource characteristics through a text file may be a requirement for some platform features in OpenStack.

3.1.4. Cloud Infrastructure Management Interface

The Cloud Infrastructure Management Interface (CIMI) is a cloud management standard created by the Distributed Management Task Force [47]. It defines a logical model for the management of resources within an infrastructure environment as a Service domain and proposes an interface based on HTTP REST calls and JSON/XML messages. It also defines a model for the resources in the cloud such as computing, storage or networking resources. The model contains a Cloud Entry Point that is essentially the list of resources in the cloud (machines, volumes, networks, and network ports). For each resource it also provides metadata in order to extend the model with provider specific information. The CIMI Data Model is complementary to the Redfish Data Model, focusing on the requirements of virtual resources in the cloud. T-NOVA draws on its specification in describing virtual resources that are relevant for the Orchestrator. The HTTP-based protocol proposed by CIMI specification is a relevant consideration in the definition of OpenStack API extension.

3.2. Requirements

Requirements for the T-NOVA Orchestration and Infrastructure Virtualization Management (IVM) have previously been documented in D2.31 [6]. Analysis of these requirements was carried out and a mapping between the requirements and the infrastructure repository functionalities is presented in Table 3-1. In particular, this table provides a mapping between the requirements and how the repository will specifically address them. The table provides a useful reference for interrogating and validating the functionalities to be implemented within the repository.

Table 3-1: Infrastructure Repository Requirements.

Requirement	How repository satisfies it.
NFVO.20 Resources Inventory	The repository will provide specific fields for

tracking	tracking the resource allocation, relying on existing fields in OpenStack API (referring to CPU, disks, RAM usage, etc.). Additionally, the repository will provide a mechanism to track resources currently not tracked by OpenStack (as GPUs, NICs, DPDK libraries, etc.). Network information will be provided using OpenDaylight API.
NFVO.17 Mapping Resources	The repository will collect information relevant for resource mapping, information on host hardware capabilities and network topology and capabilities.
Or-Vi.04 Retrieve infrastructure usage data	Data related to infrastructure utilisation by VM instances will be stored into the infrastructure repository as information regarding network usage information although the latter topic needs further investigation.
Or-Vi.05 Retrieve infrastructure resources metadata	Infrastructure metadata will be stored in the infrastructure repository.
VIM.1 Ability to handle heterogeneous physical resources	The VIM will retrieve infrastructure information from the infrastructure repository (see Section 1.7).
VIM.4 Resource abstraction	The infrastructure repository will contain details of VMs and their allocated virtual resources.
VIM.7 Translation of references between logical and physical resource identifiers	The infrastructure repository will contain the IDs to identify virtual resources.
VIM.9 Control and Monitoring	Some information regarding history reports will be available in the infrastructure repository as they are associated with the history of each VM.
VIM.20 Query API and Monitoring	Hypervisor information will be collected and persisted in the infrastructure repository.
VIM.23 Hardware Information Collection	Hardware information will be collected and persisted in the infrastructure repository.
C.7 Compute Domain Metrics	Information regarding capacity, capability and utilization of hardware resources and network resources will be persisted into the infrastructure repository.
H.7 Platform Features Awareness/Exposure	Hardware-specific features will be available in the infrastructure repository.

3.3. Infrastructure Data Access Approaches

One of the key goals of the infrastructure repository is to provide information about the current infrastructure resources to the Orchestrator. Utilising the information outlined in Sections 3.4 and 3.5 the design of the overall infrastructure repository and

potential access options that can be used are presented in Figure 3-1. The respective pros and cons of each potential approach identified are also presented.

Analysis of the available infrastructure information resources from the candidate controller technologies has identified three potential approaches to making this information available to the Orchestrator as shown in Figure 3-1.

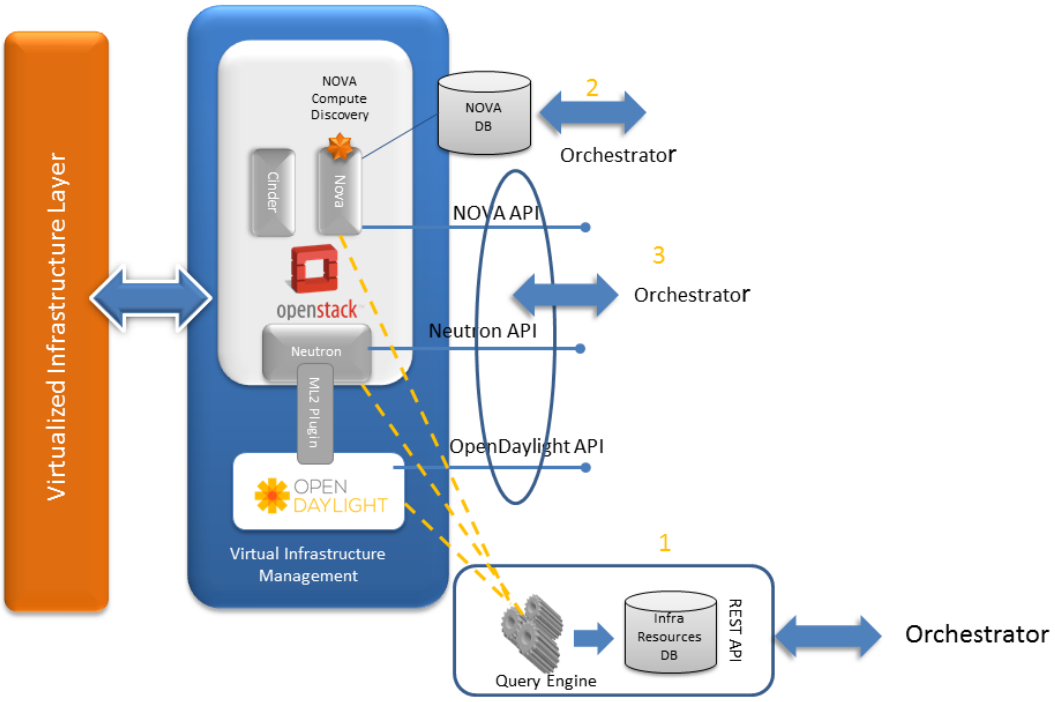

Figure 3-1: Candidate Infrastructure Data Access Approaches.

Option 1

This option is based on the implementation of a standalone database (DB). The DB is populated using a query engine to interrogate existing the NOVA, Neutron and/or OpenDaylight DBs using a mixture of existing APIs and custom queries. The Orchestrator would access the information in DB via a REST interface. The respective pros and cons of this approach are as follows:

Pros	Cons
• Data structured per Orchestrator requirements • Potentially faster response time to Orchestrator queries • More flexibility in data queries	• Adds layer of complexity and source of failure • Synchronisation and consistency challenges with the OpenStack and OpenDaylight DBs • Additional overhead on existing DBs • DB query engine tightly coupled with NOVA/Neutron DBs' structures. Potential modifications required with future OpenStack releases.

Options 2

This option is based on querying the NOVA and Neutron (and/or OpenDaylight) DBs directly using SQL queries from the Orchestration layer. The respective pros and cons of this approach are as follows:

Pros	Cons
• Up to date information always available • Queries customised to Orchestrator needs • Potentially less complex implementation	• Additional overhead on NOVA/Neutron DB's • Queries tightly coupled with NOVA/Neutron DB structures. Potential modifications required with future OpenStack releases

Options 3

Orchestrator uses existing NOVA, Neutron and OpenDaylight APIs and queries required information

Pros	Cons
• Aligns with OpenStack/OpenDaylight releases • Simplest implementation	• Less flexibility • Additional complexity at Orchestration layer to parse and structure responses from GET calls. • Multiple API's calls maybe required to retrieve data of interest

Based on analysis and discussions of these options the decision within the WP was to explore Option 3, in further detail, to determine if the information available from the APIs is sufficient to meet the Orchestration requirements. This activity is on-going with input from the dependent tasks.

3.4. OpenStack Infrastructure Data

This section outlines the main sources of infrastructure information that are currently available in the databases of the different modules of the Icehouse release of OpenStack. Since OpenStack is a modular platform, each module has a database to manage the resources and information relevant to functions of that module. In the context of the T-NOVA infrastructure repository, the databases of interest are the Nova and Neutron DBs.

3.4.1. Nova DB

The NOVA database is relative complex, containing more than 100 tables. The initial activity within Task 3.2 was to investigate the table structures in order to identify the tables containing interesting infrastructure data, potentially useful to the Orchestrator. Figure 2 shows the inter-relationships between the tables of the Nova

DB containing physical resources within the cloud environment. In particular, the table "compute_nodes" contains useful information about the physical hosts including information on the hypervisor, the number of virtual CPUs, available/used main memory (RAM), available/used disk space, CPU details (such as vendor, model, architecture, CPU flags, the number of cores, etc.). Figure 3-2 only contains a subset of the tables relating to "compute_nodes" for illustrative purposes only due to resolution constraints.

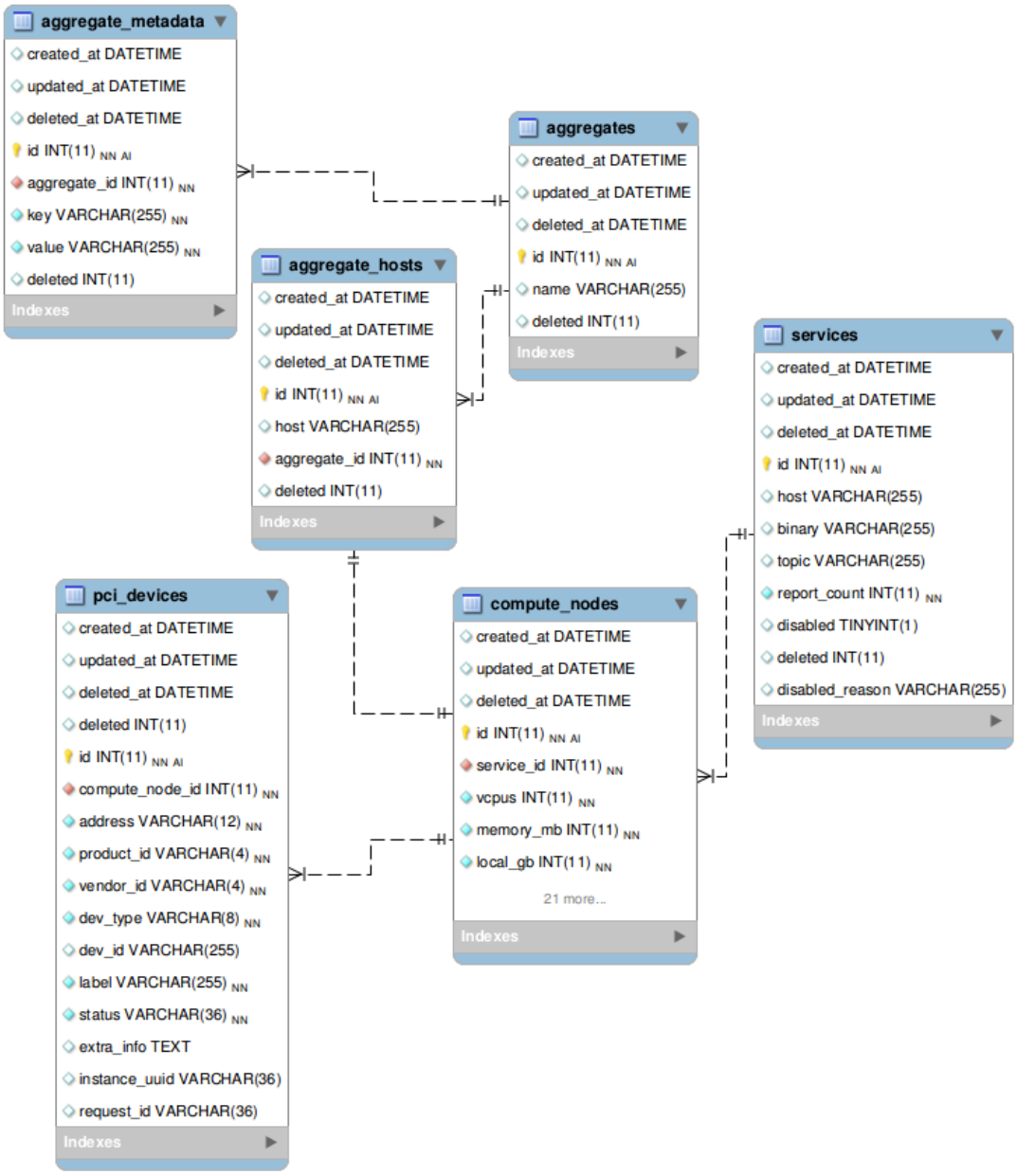

Figure 3-2 Physical resources in Nova DB.

The other tables in Figure 3-2 are referred to as the Host Aggregates mechanism. It allows OpenStack Nova to divide the hosts into subsets of nodes within the same availability zone. Host Aggregates provide a mechanism to allow administrators to assign key-value pairs to groups of machines. Each node can have multiple aggregates, each aggregate can have multiple key-value pairs, and the same key-value pair can be assigned to multiple aggregates. This information can be used in

the scheduler to enable advanced scheduling, to establish hypervisor resource pools or to define logical groups for migration. From a T-NOVA perspective the Host Aggregates offers an opportunity to use the aggregates metadata as a mechanism to influence VM placement by added platform features into the node selection process for VM placement.

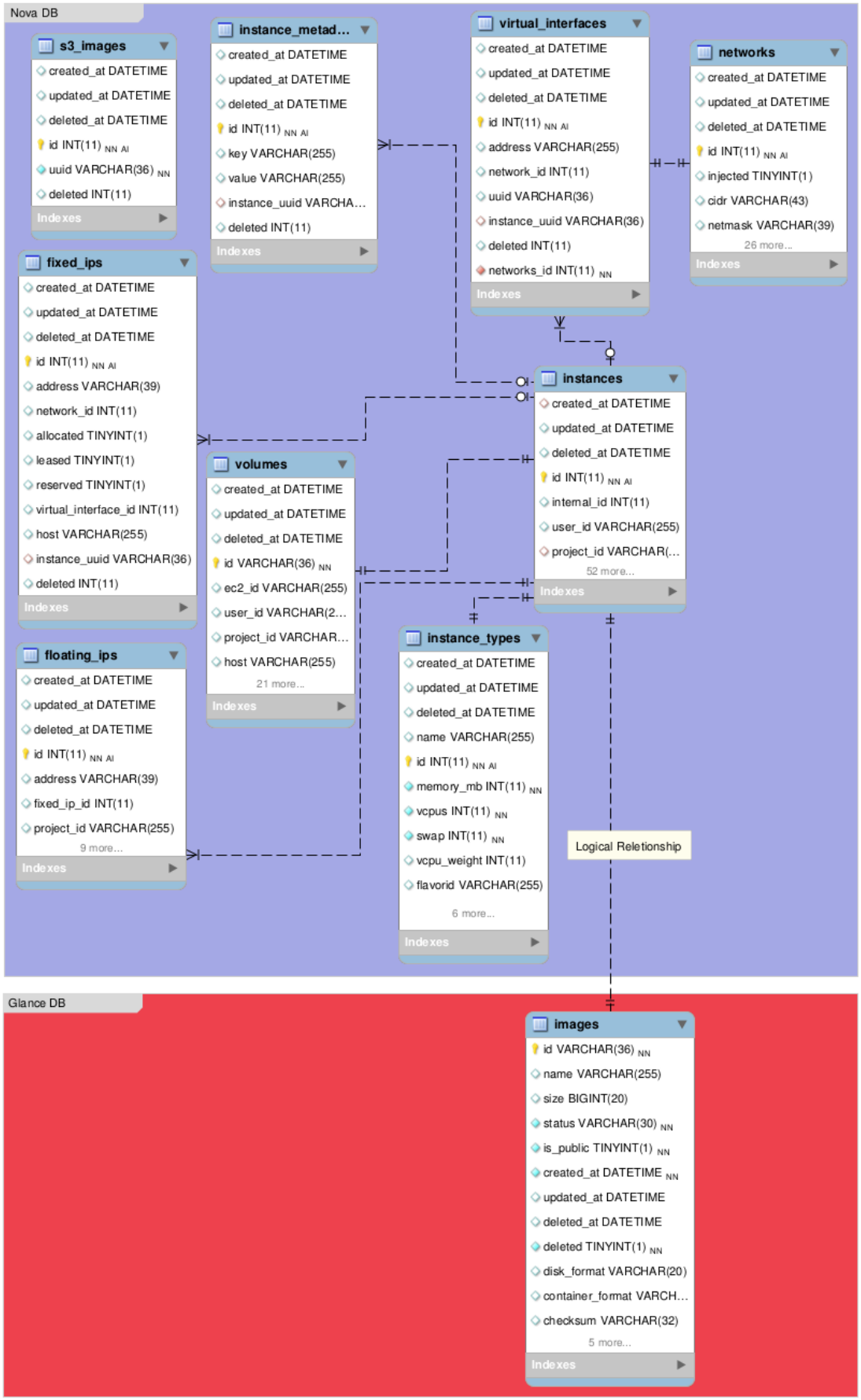

Figure 3-3 Virtual resources in Nova and Glance DBs.

The tables within the Nova and Glance DBs, which store information related to virtual resources, are shown in Figure 3-3. The primary table, where information relating to VMs is stored, is called "instances". An instance can have fixed IPs, floating IPs, volumes, virtual interfaces that give it the access to many networks, an instance type, and an image (from Glance DB). Also an instance or an instance type or an image can have metadata that could be used as part of the scheduling process providing additional information that could be utilised by the scheduler and filters. However that capability is not available in the current release of OpenStack and, for that reason, the implementation will require a standalone database as an extension of OpenStack to investigate the proof of principle.

3.4.2. Neutron DB

Neutron provides "Networking as a Service" for OpenStack resources.

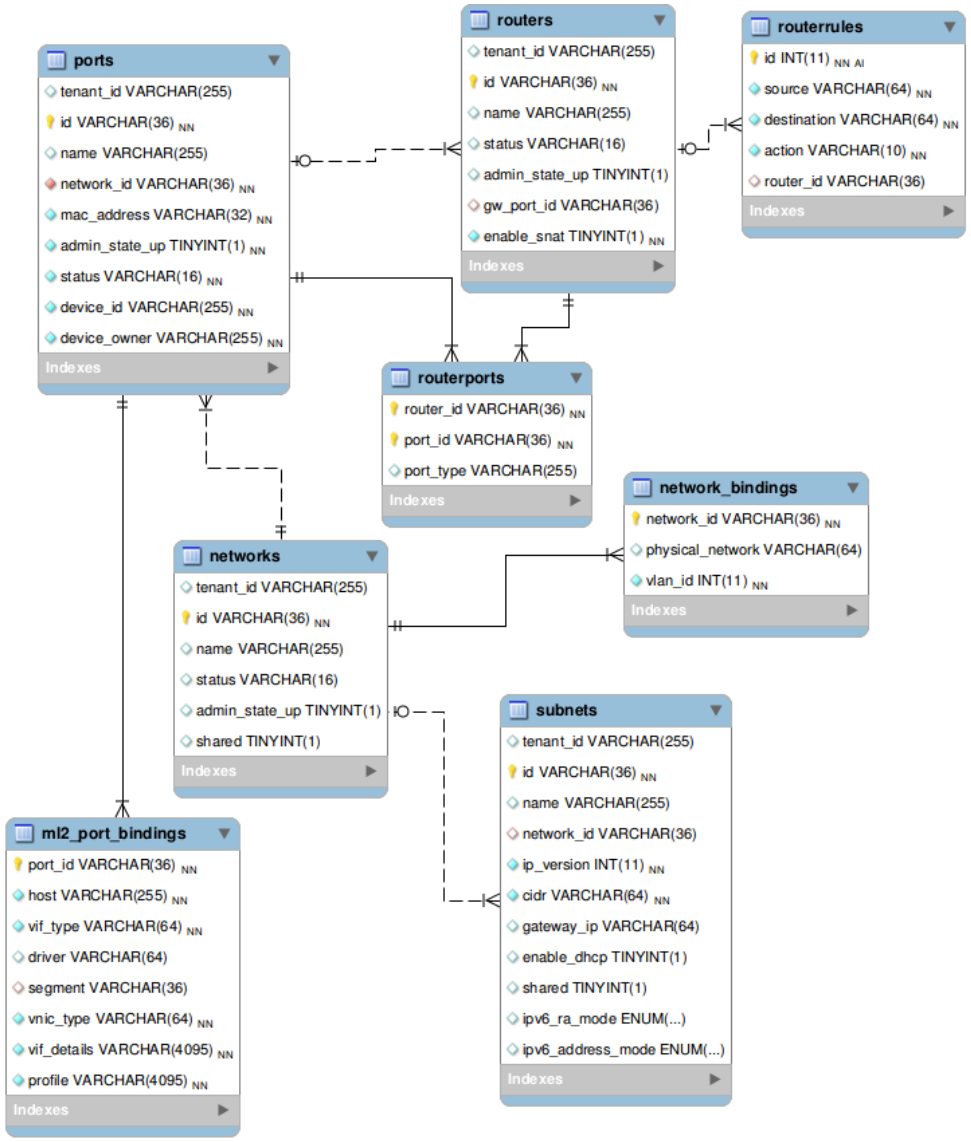

Figure 3-4 Neutron DB portion.

The service is based on a model of virtual networks, subnets and port abstractions to describe the networking resources. A network is an isolated layer-2-segment and corresponds to a VLAN in the physical networking world. The network is the primary object for the Neutron API. Ports and subnets are assigned to a specific network as shown in Figure 3-4. A subnet is a block of IP addresses that can be assigned to the VMs. A port is a virtual switch connection point. Each VM can attach its virtual Network Interface Controller (vNIC) to a network through a port. A port has a fixed IP address from those of the relative subnet. Routers are local entities that work at Layer-3 networking enabling packets routing between subnets, packets forwarding from internal to external networking, providing Network Address Translation (NAT) services and providing access instances from external networks through floating IPs.

3.5. Infrastructure Information Retrieval

As previously outlined, existing infrastructure information is stored in the Nova and Neutron DBs. Other information regarding network topology can be retrieved using OpenDaylight API. Both OpenStack and OpenDaylight offer a set of REST APIs.

3.5.1. Nova API

Information within Nova DB can be accessed externally using the Nova REST API [48]. The API currently includes more than 100 REST calls, which can be used to query and extract information from the NOVA database. Table 3-2 outlines the REST GET calls that return physical infrastructural information that is of interest at the T-NOVA Orchestration layer

Table 3-2: Nova Compute API Calls regarding Compute Nodes information.

Description	Calls (GETs)
List of hosts	/v2/{tenant_id}/os-hosts
Host's detail	/v2/{tenant_id}/os-hosts/{host_name}
List of Hypervisors	/v2/{tenant_id}/os-hypervisors
Hypervisor's details (resources' usage)	/v2/{tenant_id}/os-hypervisors/detail
Hypervisors Statistics over all compute nodes	/v2/{tenant_id}/os-hypervisors/statistics
List instances that belong to specific hypervisor	/v2/{tenant_id}/os-hypervisors/{hypervisor_id}/servers

As outlined in Sub-section 3.4.1, key-value pairs can be associated to groups of machines based on the availability of similar attributes, using the Host Aggregates. The NOVA API provides a set of GET and POST calls as outline in Table 3-2 that can be used for both information retrieval and to persist metadata, which contains key-value pairs, relating to an aggregate.

The Nova API also supports of retrieval of information relating to the virtual resources that are currently running on the cloud infrastructure. The relevant GET calls to the Orchestrator are outlined in Table 3-3.

3.5.2. Neutron API

The GET REST API [48] calls currently available in the Icehouse release of OpenStack are shown in Table 3-3. These GET calls are available to external services to retrieve network infrastructural information stored in the Neutron DB. The information is a key input for example into the Resource Mapping algorithm being developed by Task 3.3

Table 3-3: Neutron API Calls.

Description	Calls (GETs)
List of networks	/v2.0/networks
Single network	/v2.0/networks/{network_id}
List of subnets	/v2.0/subnets
Single subnet	/v2.0/subnets/{subnet_id}
List of ports	/v2.0/ports
Single port	/v2.0/ports/{port_id}
List of routers	/v2.0/routers
Single router	/v2.0/routers/{router_id}
List of floating IPs	/v2.0/floatingips
Single floating IPs	/v2.0/floatingips/{floating_ip}

3.5.3. OpenDaylight API

OpenDaylight [49] is the candidate SDN controller selected for the T-NOVA IVM as outlined in D4.0.1 [57]. OpenDaylight is integrated with OpenStack through the Modular Layer (ML) 2 plugin that exposes the Neutron API. OpenDaylight provides an SDN controller that comes with a Flow Programmer service that helps application programming flows by using a REST interface. The basic job of the Flow Programmer is to query and change state of switches by returning, adding or deleting flows. The state of a resource is represented by an XML or a JSON object. The complete collection of API calls provided by OpenDaylight that supports external interaction including network information retrieval are outlined in Table 10-3. From a T-NOVA perspective the most important REST GET API calls are described in Table 10-4.

3.6. T-NOVA Specific Data Model

3.6.1.1. Gaps Identified

Analysis of the Nova DB reveals that the infrastructure related information is relatively limited apart from CPU flags. The Icehouse version of the OpenStack Nova DB

contains a table for tracking PCI/PCIe devices [50] installed within hosts. However the table is currently not populated and there are no API calls available to interact with the table. This is significant gap, in particular for VNFs, since many of them have dependencies on the specific characteristics of the device (e.g. NIC with DPDK/SR-IOV support). Moreover, even if the information was populated into the Nova DB, there are no mechanisms currently available for the Nova Scheduler or filters to utilise the information. Therefore a mechanism to identify additional platform features and attributes is currently being developed and is described in the Enhanced Platform Awareness (EPA) Discovery Agent section 3.7.1.

In order to expose and utilise additional platform features and attributes beyond what is currently available in the NOVA DB, a specific filter needs to be implemented. This filter will utilise the set of platform features which are stored across the Nova and T-NOVA enhanced platform awareness DB's in conjunction with Nova filter chain and scheduler. The T-NOVA filtering implementation will select the physical node and network connection with the required feature set to support a given VNF. Implementation options for this requirement are being explored in Task 4.1. It is expected that the T-NOVA Orchestrator will pass the platform requirements for a given a VNF in the form of metadata in a REST API call to the VIM (specifically to the OpenStack Controller). The specifics of the required API calls are being investigated by Task 4.1 in conjunction with Task 3.1.

3.6.1.2. Data Model

An initial data model description has been developed based on the available infrastructural information resources. This initial proposal is shown in Figure 3-5.

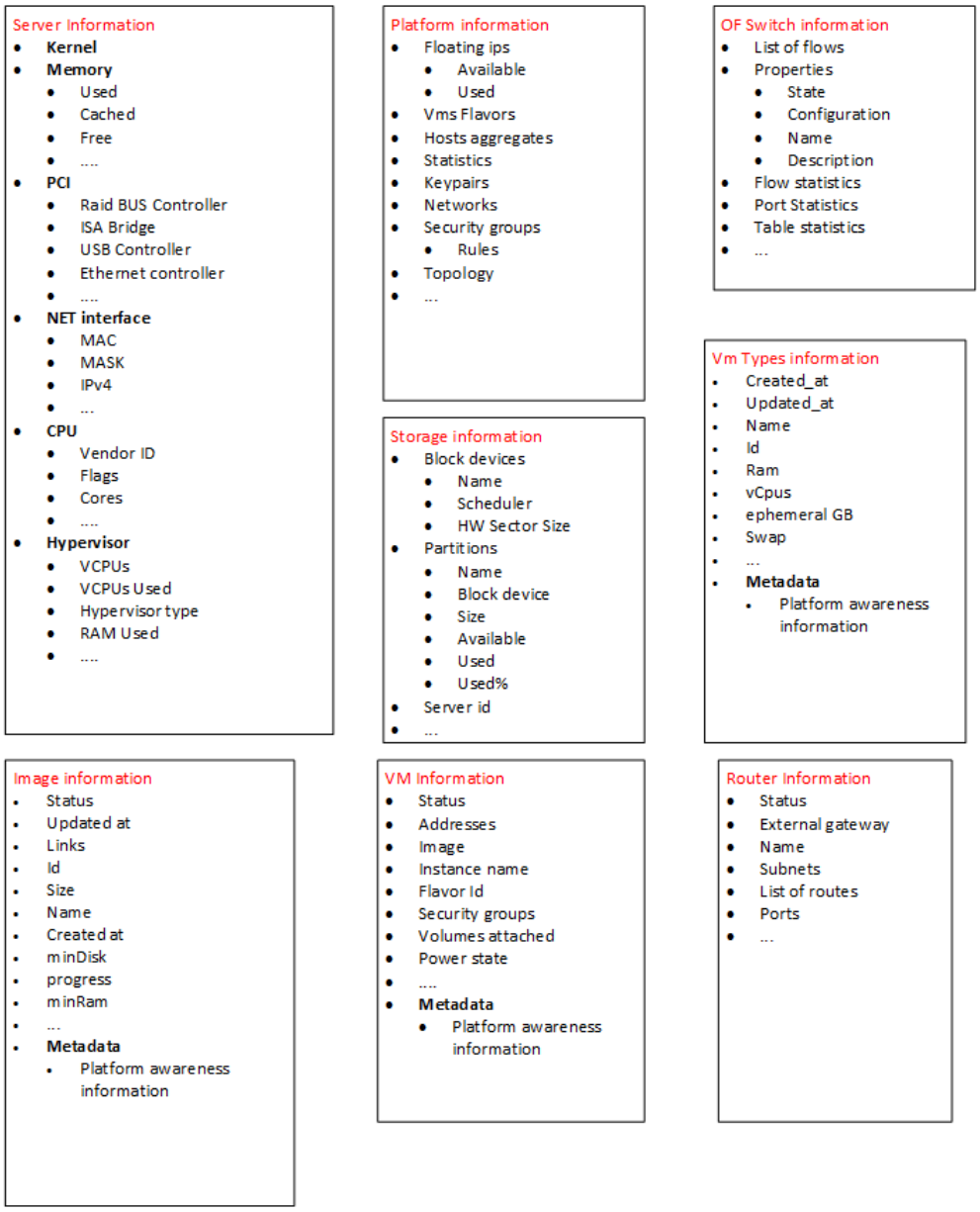

Figure 3-5: Data Model overview.

The Orchestrator retrieves information about each resource in the platform. As previously outlined, existing information will be available to the Orchestrator through OpenStack/OpenDaylight APIs. In addition, a standalone DB containing EPA information regarding physical hosts, peripheral devices, such as NICs, is being implemented and will be accessible via a REST API to the Orchestrator and Nova scheduler/filter mechanism. Based on input from Task 3.3 a significant gap in the current model has been identified in relation to physical network topology information. This gap is currently being investigated further in order to identify an appropriate solution to the issue. The relationship between the data model and T-NOVA architecture is shown in Figure 11-1 in the Annexes.

3.7. Proposed Implementation Plan

As outlined in Section 3.5 the current implementation plan is focused around the use of the existing APIs to expose infrastructural information to the Orchestrator. However exposure of additional infrastructural information is required to support more intelligent placement decisions for VNFs. The information will be collected via the implementation of a Python based agent that can collect information relating to platform features and capabilities from the physical servers. Figure 3-6 shows the high level proposed implementation based on the combination of option 3 previous described together with the EPA agent approach.

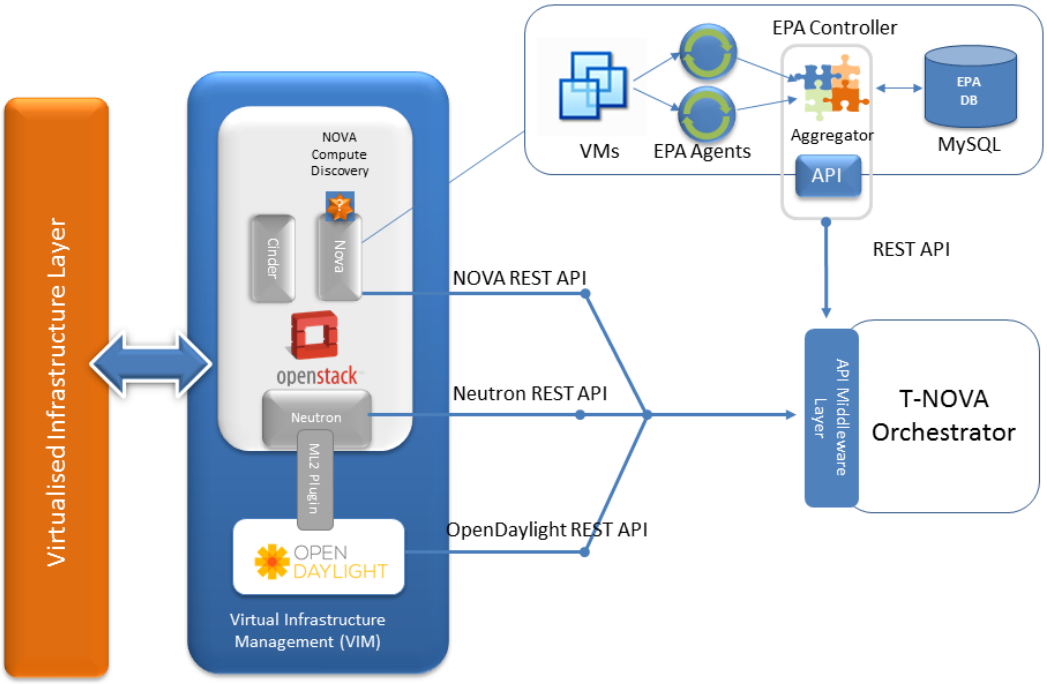

Figure 3-6 High level proposed architecture for T-NOVA IVM Infrastructure Repository.

The specifics of the EPA implementation are based on the deployment of an application called EPA collector on the OpenStack controller as shown in Figure 3-7. This application is responsible for aggregating information sent by the EPA agents running on the physical server nodes. The collector stores the platform information into a relational database and also provides a REST API that can be used by the Orchestrator to access the information as necessary.

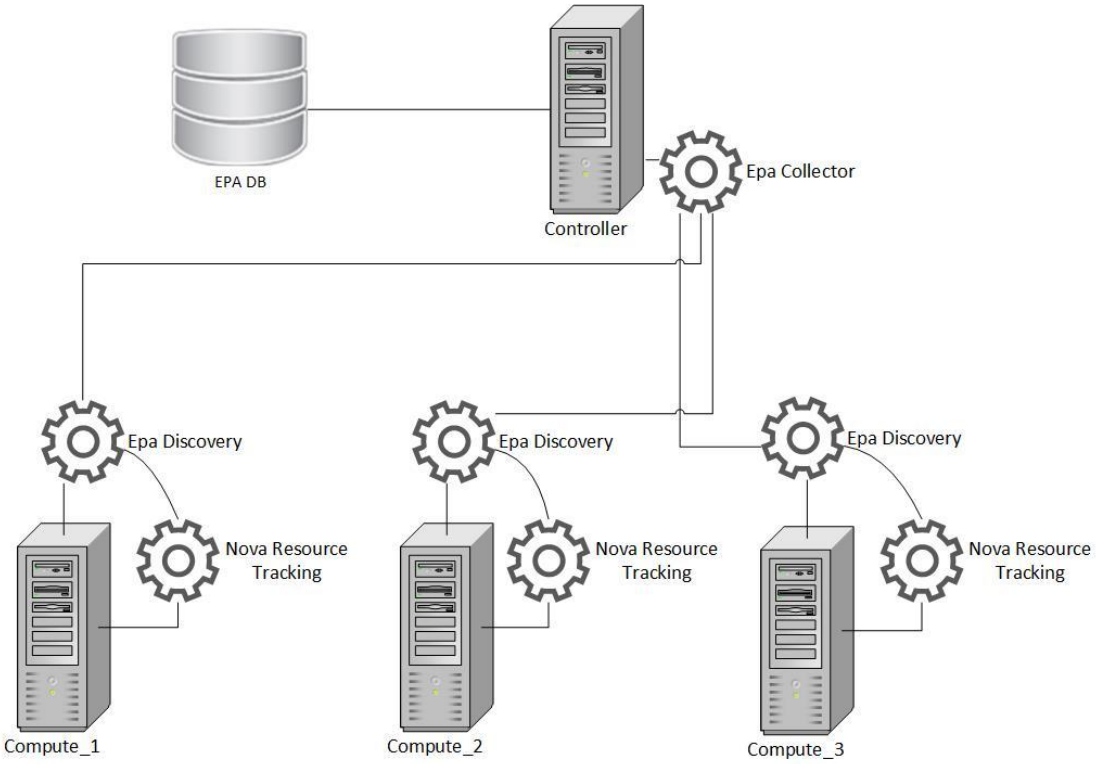

Figure 3-7. Proposed EPA Architecture for T-NOVA IVM Infrastructure Repository.

Platform features to be collected by the EPA agents include:

1. Features that are related to a specific capability of the physical host, but are independent of its utilisation. For example with DPDK it is required to know if the specific host has these set of libraries and drivers installed for fast packet processing. This type of resources is referred to as “non-enumerated”;
2. Features that are related to the specific instance usage/consumption of a resource e.g. numbers of unassigned GPUs or number of SR-IOV channels. It is important to know how many are available, how many are used and so on. This type of resource is referred to as “enumerated”.

Details relating to the two different resources types are outlined in Table 3-4.

Table 3-4. Infrastructure Repository Resources

Enumerated Resource	Non-enumerated Resource
• name - used to identify this resource • type - a resource type name for human viewing • description - a short description for human viewing • resource total capacity (configured or discovered - total amount of resource • resource used capacity (tracked - amount of resource currently used • resource limit capacity (used to implement under committing policies of the resources)	• name - used to identify this resource • type - a resource type name for human viewing • description - a short description for human viewing • enable – Boolean

Non-enumerated resources typically have a long lifetime and are updated infrequently. Enumerated resources require event driven updates, specifically every time an event related to the VMs' management occurs.

Therefore Nova Compute Resource Tracking needs to be extended to track the usage of the additional resource types and features. For each additional resource type the implementation of specific scripts will be required in order to obtain information about them.

A standalone MySQL[1] DB is deployed in order to contain information that cannot be stored into the Nova DB through current API. In particular non-enumerated resources regarding network topology could be stored into the metadata of Host aggregates while other non-enumerated resources and enumerated resources into the external DB. Using this solution additional information will be accessible through specific API that will retrieve information out of the standalone DB.

To use this information in the scheduling process a specific filter will need to be deployed that communicates with the external DB as shown in Figure 3-8.

[1] The initial implementation will utilise MySQL however alternatively solutions will also be investigated such as MongoDB to determine the most appropriate long term solution.

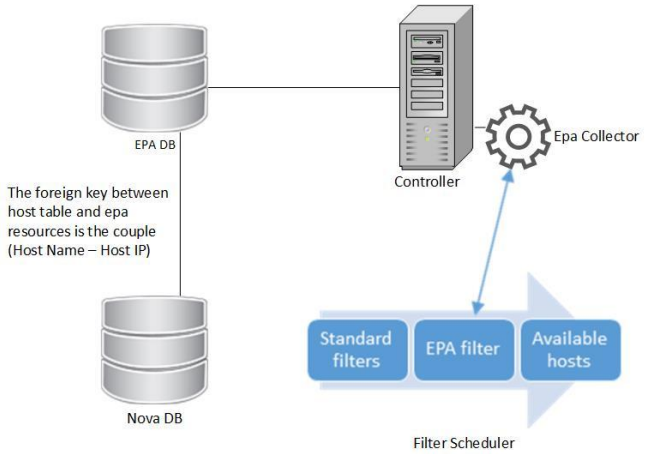

Figure 3-8. Proposed Relationship between EPA and NOVA DB's.

3.7.1. EPA Discovery Agent

The EPA discovery agent runs on each node belonging to the cloud compute cluster. It is responsible for discovering hardware capabilities of the physical server. The Python based EPA agent developed to date extracts information from each host and formats into a JSON object. This JSON object is then passed to the EPA Collector ready to be imported into the EPA DB. Details of the platform information contained JSON object are available in Section 12 (Annex D). The information available is much richer than that currently stored in the Nova DB. For illustration purposes the output of two of the most relevant infrastructure related API calls are presented in Table 12-1 on that Annex.

3.7.2. EPA Rest Interface

An initial prototype of a REST interface for the EPA DB has been implemented. It adopts the same structure as the existing OpenStack API calls and from a user perspective they appear as a simple extension of them. An example of a REST API call is shown in Figure 3-9, which returns a list of the PCIe devices available from a specified host. The call takes the form of:

```
GET /epa/v1/hosts/{host_id}/pci_devices
```

```
iolie@IRILD019:~$ curl -X GET http://epa.t-nova.eu/epa/v1/hosts/2/pci_devices
{
  "pci_devices": [
    {
      "device_type": "Ethernet controller",
      "dpdk": true,
      "dpdk_features": "unused:ixgbe drv:igb_uio ",
      "host_id": 2,
      "id": 17,
      "name": "Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection (rev 01)",
      "slot": "01:00.0",
      "sriov_channels": 63
    },
    {
      "device_type": "Ethernet controller",
      "dpdk": true,
      "dpdk_features": "unused:ixgbe drv:igb_uio ",
      "host_id": 2,
      "id": 18,
      "name": "Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection (rev 01)",
      "slot": "01:00.1",
      "sriov_channels": 63
    },
    {
      "device_type": "Ethernet controller",
      "dpdk": false,
      "dpdk_features": null,
      "host_id": 2,
      "id": 19,
      "name": "Intel Corporation Ethernet Connection I217-LM (rev 05)",
      "slot": "00:19.0",
      "sriov_channels": -1
    },
    {
      "device_type": "Host bridge",
      "dpdk": false,
      "dpdk_features": null,
      "host_id": 2,
      "id": 20,
      "name": "Intel Corporation 4th Gen Core Processor DRAM Controller (rev 06)",
      "slot": "00:00.0",
      "sriov_channels": -1
    },
    {
      "device_type": "Audio device",
      "dpdk": false,
      "dpdk_features": null,
      "host_id": 2,
      "id": 21,
      "name": "Intel Corporation Xeon E3-1200 v3/4th Gen Core Processor HD Audio Controller (rev 06)",
      "slot": "00:03.0",
      "sriov_channels": -1
    },
    {
      "device_type": "SMBus",
      "dpdk": false,
      "dpdk_features": null,
      "host_id": 2,
      "id": 22,
      "name": "Intel Corporation 8 Series/C220 Series Chipset Family SMBus Controller (rev 05)",
      "slot": "00:1f.3",
      "sriov_channels": -1
    },
```

Figure 3-9: PCI Devices of a specified host.

Additionally the EPA agent checks each Ethernet Controller to determine if it uses DPDK and stores this information as Boolean flag in the EPA DB.

For devices supporting DPDK, the SR-IOV channel support count is also identified. Querying a lookup table, the EPA agent determines the number of channels the

device can support and stores that information together with the number of channels that have assigned.

A web interface has been implemented to show the available EPA information for a given host, based on the amalgamated information from the EPA and Nova DBs. An example of the web interface is shown in Figure 3-10 and Figure 3-11.

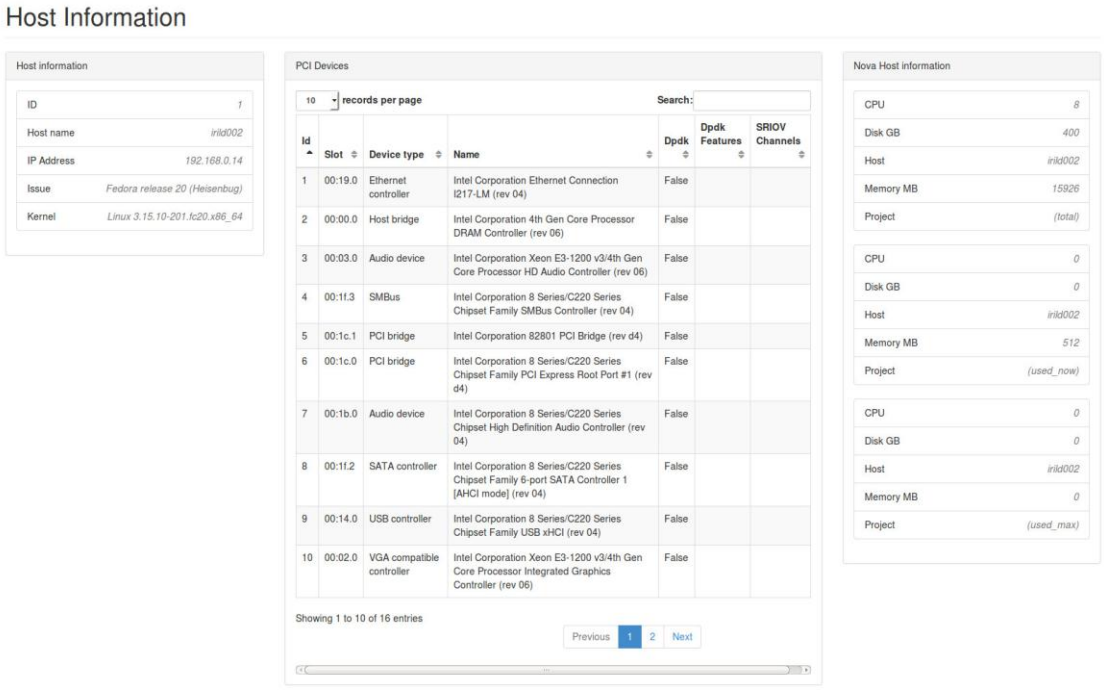

Figure 3-10: Screenshot of the EPA Server web interface.

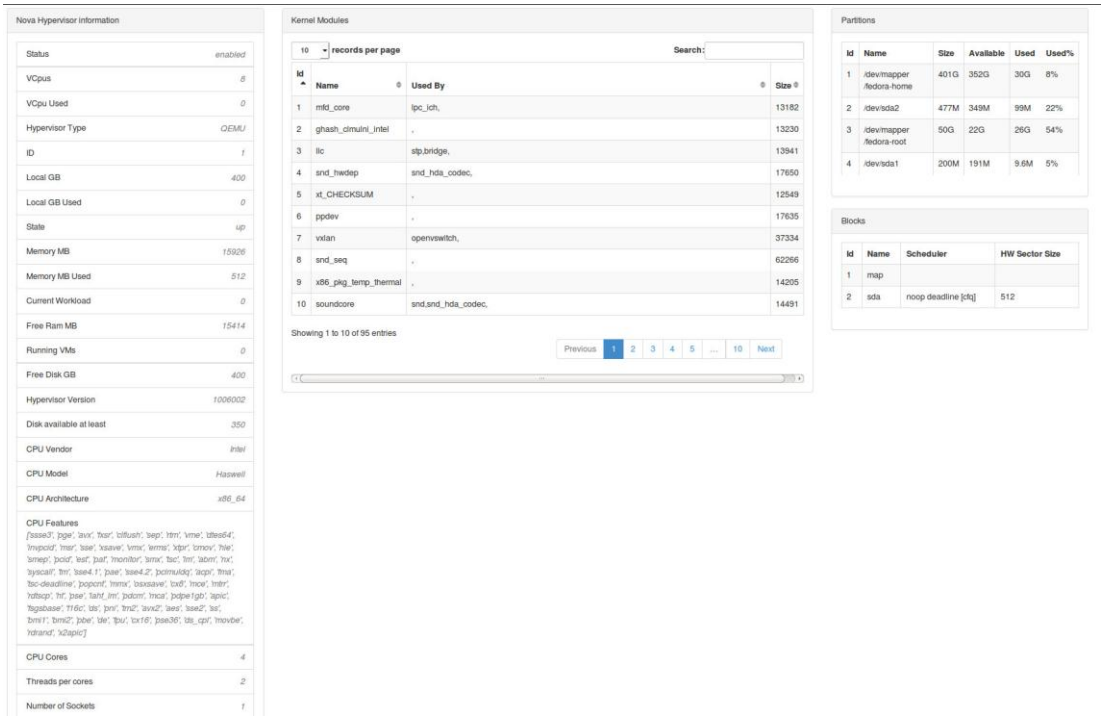

Figure 3-11: Screenshot of the EPA Server web interface.

3.8. Network Topology Visualisation

In addition to the development of the Resource Repository as outlined in the previous sections, Task 3.2 is also investigating the implementation of a "Service Visualisation" module. The purpose of this module is to provide the Orchestrator with actionable insights with respect to the virtual network builds and monitoring operations across all segments of functional virtual network within the T-NOVA system. It will function as a common inventory (including ordering and monitoring functions) by showing the status of the network builds (in case of Carrier Ethernet, ENNIs and EVCs) per access vendor data.

Additionally, through the visualisation module, the user (Service Provider and Customer) will be able to highlight key faults and alarms, with the ability to drill down and sectionalize faults to a specific segment of the network (e.g. location, vNet segment, etc). This module needs to talk to most of OpenStack APIs in order to acquire this information and also request this information from the TNM (even in a semi static way, i.e the end Transport network routing/trunking information will not change very often).

The main features of the module are:

- "Drill" down capabilities from maps to topology layers, aggregation sites and domains;
- Extensive network path visualization for rapid fault isolation and segmentation of the network, in areas/sections;
- End-to-End path from core or aggregation site to cell site with major demarcation points and segments highlighted;
- Ability to drill down to detailed attributes of each object in the path based on component/network attributes (currently support Carrier Ethernet, based on MEF attributes);
- Showcases cross-path attribute visualization across multiple views including bandwidth, VLAN, demarcation;
- View Status of network builds based on state (pending, live);
- Customizable views are provided, based on user rights/role, Service Provider, Network Operator, Customer, which are fully configurable.

This module runs as a standalone service, providing a common northbound RESTFul API for ease of integration into Service/Network Operator's systems, and a pluggable mechanism for the southbound interface to support different network inventories (Carrier Ethernet, Sonet, etc.).

3.9. Relationship and Inter Task Dependencies

Task 3.2 has a number of inter dependencies with other tasks in WP3 and WP4. The key dependencies are in Table 3-5.

Table 3-5: Inter-tasks dependencies from Task3.2, Infrastructure Repository.

Dependent Task	Dependency

Task 4.1: Resource Virtualisation	Task 4.1 will identify appropriate platform features that should be collected and stored in the Infrastructure Repository. Task 4.1 will determine the most appropriate mechanism for the use of EPA features within OpenStack scheduling and filtering processes.
Task 3.3: Service Mapping	Task 3.3 will use the information resources available within the infrastructure repository as inputs into the definition and development of the service mapping algorithm.
Task 3.1: Orchestrator Interfaces	Task 3.1 will provide input into the definition of the Orchestrators interfaces by identifying the OpenStack Nova, Neutron and OpenDaylight REST API calls that should be used the Orchestrator. Task 3.2 will implement an interface to the EPA database.
Task 3.4: Service Provisioning, Management and Monitoring	Task 3.4 will coordinate interactions of the service mapping module and the infrastructure repository in order to instantiate NS. It will also investigate how the network visualisation tool can be combined or integrated with the management UI of the orchestrator.

3.10. Conclusions and Future Work

Task 3.2 has conducted an analysis of the infrastructural informational resources currently available based on the technologies that have been selected for the initial implementation of the T-NOVA IVM. Different options have been identified and one has been selected for implementation evaluation that is based on the use of existing APIs in OpenStack and OpenDaylight. However limitations in terms of the available infrastructural information have been identified. Therefore a solution is being developed in the form of an enhanced platform awareness agent that can collected additional platform regarding attributes and features and persist that information to a relational database for use by the Orchestrator or the NOVA Scheduler/filtering mechanism. An initial data model for the IVM has been developed based on the information resources currently available and those that will be made available through the EPA implementation.

4. SERVICE MAPPING

This section presents the work carried out to date in Task 3.3, Service Mapping. A definition of the Service Mapping problem is initially outlined in Section 4.1; then different approaches proposed and investigated to date are presented in Section 4.2. Section 4.3 reports details on OpenStack's virtual machine deployment mechanisms, while an initial comparison of the proposed service mapping approaches is presented in Section 4.4. In Section 4.5 interdependences with the other tasks are outlined. Finally, the task's conclusions are presented in Section 4.6.

4.1. Problem Definition

The Service Mapping (SM) problem addressed in T-NOVA focuses on the optimal assignment of Network Service (NS) chains to servers hosted in interconnected Data Centres (DCs) that are operated by one Network Service Provider (see Figure 4-1(a)).

The optimality concept can be defined with regard to different objectives: economical profit, Quality of Service (QoS), energy-efficiency and others.

The SM is an online problem. That is, the requests for NSs will not be known in advance. Instead, they arrive to the system dynamically and, if they are accomplished, they can stay in the network for an arbitrary amount of time. Algorithms for the SM problem have to handle service requests as they arrive.

According to ETSI's NFV Architectural Framework [51] a NS is represented by one Forwarding Graph in which each vertex is a Virtual Network Function (VNF). Hence in T-NOVA a NS is defined as a directed graph $G(NS) = (V, A)$ in which each vertex, say h, in the set V represents a VNF, and each arc, say (h, k), in A represents a link connecting two VNFs required for the correct implementation of the service (e.g. a chain in a web server tier composed by firewall, NAT and load balancer).

The Network Infrastructure (NI) on which we want to run the NS can be described as a directed graph $G(NI) = (V^I, A^I)$ in which each vertex, say p, in the set V^I represents a DC, and each arc, say (p, q), in A^I represents the network connection established by the network provider among the DCs.

Hence, the first problem arises when a new NS instance request arrives to the Orchestrator and the SM is asked to assign each VNF in the required service to a DC within the available network infrastructure (note that it is possible that all the involved VNFs are eventually assigned to the same DC). More formally, this "first level problem" can be stated as follows.

First level problem: Given a NS and a NI, solving the SM problem requires to assign each VNF in the service, to a DC in the network (i.e. each vertex in *V* to a vertex in V^I) and each arc (*h*, *k*) in *A*, to an oriented path in G(NI) from the DC to which the vertex *h* has been assigned, to the DC to which the vertex *k* has been assigned.

Figure 4-1 (a) reports a NS composed by two VNFs, a NI composed by four interconnected DCs and their corresponding graphs.

Figure 4-1 (b) reports a solution of the first level problem involving the graphs of Figure 4-1 (a). VNF_1 has been assigned to DC_1, VNF_2 has been assigned to DC_4 and the arc connecting VNF_1 and VNF_2 has been assigned to the blue path from DC_1 to DC_4, through DC_3.

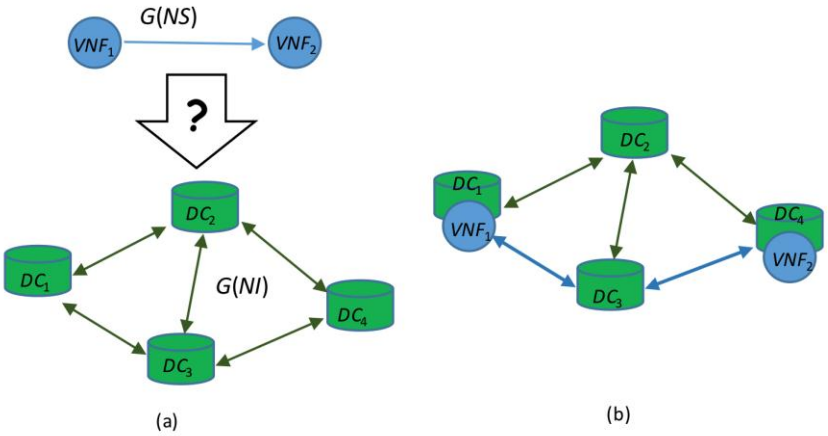

Figure 4-1: Example of a first level SM problem (a) and its solution (b).

Moreover, each VNF can have a complex structure, i.e., it can be decomposed in elementary interconnected components, each one executable on a Virtual Machine (VM). At the same time, each DC is composed by hundreds (or thousands) of interconnected servers.

Hence, once a VNF has been assigned to a DC, a second problem (referred to as a "second level problem") arises by asking to instantiate each VM composing the VNF on a server hosted in the DC.

More formally, each VNF can be described as a directed graph $G(VNF) = (V^F, V^F)$ in which each vertex, say i, in the set V^F represents a Virtual Network Function Component (VNFc), and each arc, say (i, j), in A^F represents a link between components of the VNF.

In turn, each DC can be described as a directed graph $G(DC) = (V^D, A^D)$ in which each vertex in the set V^D represents a hardware apparatus, either a server or a network switch, and each arc in A^D represents the network connection established by the DC owner between hardware apparatuses.

Figure 4-2 displays, on the left side, a VNF composed by four interconnected components, and, on the right side, the internal structure of a DC model with its interconnected apparatuses.

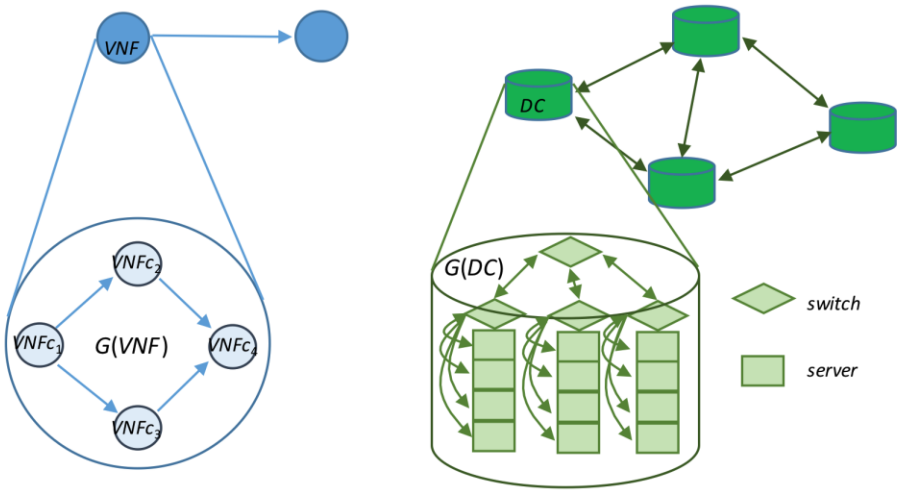

Figure 4-2: Example of a VNF composed by four VNFcs (on the left) and of the internal structure of a DC model (on the right).

Second level problem: Given a VNF and a DC, solving the SM problem requires also to assign each VNFc in the VNF to a server in the DC (i.e. each vertex in V^F to a vertex representing a server in V^D) and each arc (i, j), in A^F, to an oriented path in G(DC) from the hardware apparatus hosting the VNFc i to the hardware apparatus hosting the VNFc j.

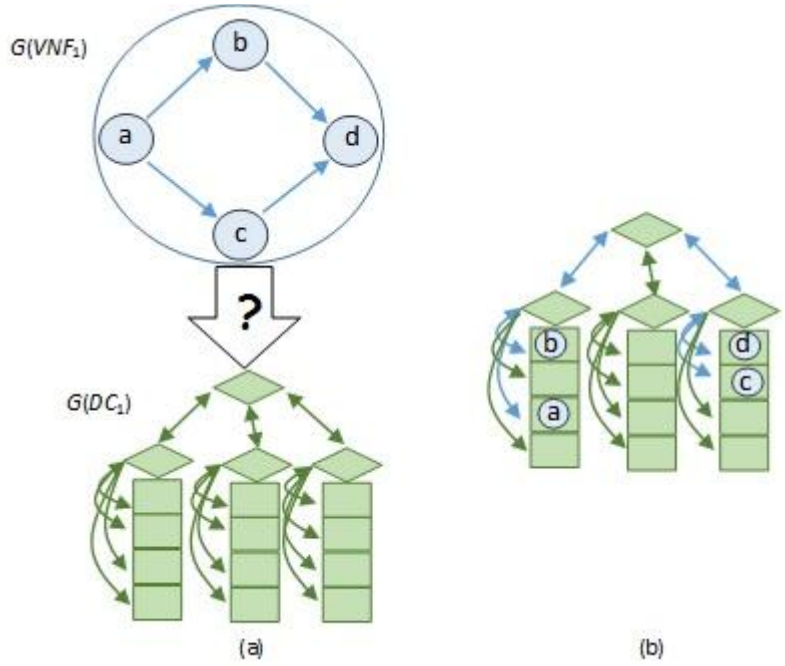

Figure 4-3: Example of second level SM problem (a) and its solution (b).

Figure 4-3 (a) shows an instance of the second level problem, in which we need to assign the components of the VNF_1 to the servers of the DC_1. Figure 4-3 (b) shows a solution of the second level problem, where each component has been assigned to a

(suitable) server and the links connecting the components have been mapped to the blue paths involving switches and servers.

Each NS is composed by one or more VNFs and is represented as a forwarding graph of those VNFs, each one of these being represented by its own graph whose vertices are Virtual Network Function Components. Hence each NS can be directly represented as a graph of components. At the same time, the NI is a composition of DC graphs and it also can be directly represented as a graph of hardware apparatuses. If we use the two representations outlined, one for the NS and the other for the NI, then the Service Mapping problem reduces to the Virtual Network Embedding (VNE) problem (see [52] for a comprehensive survey), i.e. the problem of embedding a virtual network, represented by an oriented graph, into the platform of a substrate network, represented by another oriented graph. More formally, this problem can be stated as follows.

The flat problem: If we explode each node-graph G(VNF) contained in graph $G(NS) = (V, A)$ we obtain a new *expanded* directed graph, say EG(NS) = (EV^F, $A' \cup EA^F$) which we call the *expanded representation* of the NS. Let V_h^F denote the set of components associated to the VNF corresponding to the vertex h in V. The vertex set in EG(NS) is given by all those components, i.e. $EV^F = \cup_{h \in V} V_h^F$. Similarly, let A_h^F denote the set of arcs associated to the VNF corresponding to the vertex h in V. EA^F is given by all the internal arcs, i.e. $EA^F = \cup_{h \in V} A_h^F$. At last, each arc (*i*, *j*) in A' replaces a corresponding arc (h, k) in A by connecting two suitable components, *i* and *j*, where component *i* belongs to the VNF *h*, and component *j* belongs to the VNF *k*.

In the same way, if we explode each node-graph G(DC) contained in graph G(NI) we obtain a new *expanded* directed graph, say EG(NI) = (EV^D, $A^{I'} \cup EA^D$) which we call the *expanded representation* of the network infrastructure. Let V_h^D denote the set of hardware apparatuses associated to the DC corresponding to the vertex *h* in V^I. The vertex set in EG(NI) is given by all those apparatuses, i.e. $EV^D = \cup_{h \in VI} V_h^D$. Similarly, let A_h^D denote the set of arcs associated to the DC corresponding to the vertex h in V^I. EA^D is given by all the internal arcs, i.e. $EA^D = \cup_{h \in VI} A_h^D$. At last, each arc (*i*, *j*) in $A^{I'}$ replaces a corresponding arc (*h*, *k*) in A^I by connecting two suitable switches, *i* and *j*, where the switch *i* belongs to DC *h*, and the switch *j* belongs to DC *k*.

On the left side of

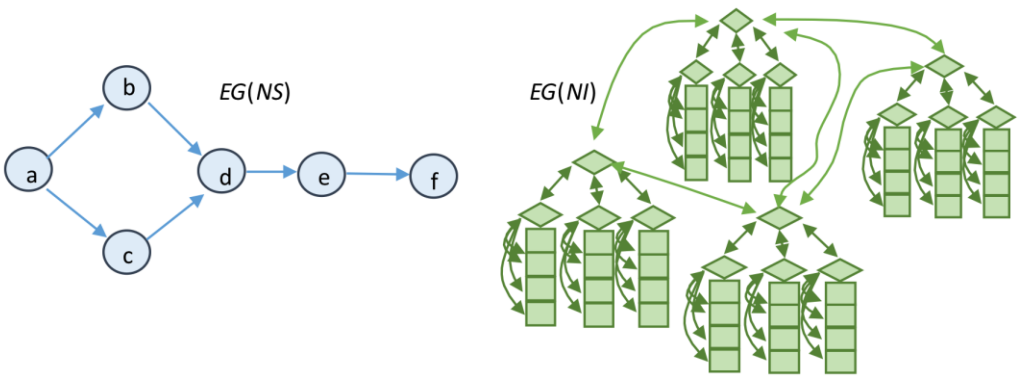

Figure 4-4, the two VNFs of NS in Figure 4-1(a) have been expanded and the corresponding expanded direct graph is presented. On the right side of the figure,

each of the four DCs in the NI presented in Figure 4-1(a) has been expanded and the direct graph modelling all the interconnected hardware apparatuses involved is presented.

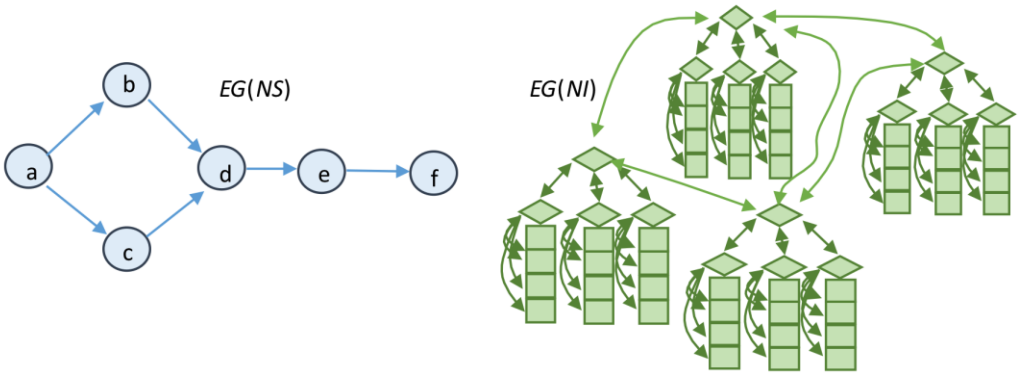

Figure 4-4: Example of a flat SM problem.

The T-NOVA Service Mapping problem requires the solution, in a feasible way, of both the first and second level problems, or the solution, in a practical way, of the flat problem.

Since the Virtual Network Embedding problem is NP-hard [52] even solving the SM problem is NP-hard and, apart for instances of small size, only heuristic approaches can be considered.

4.1.1. Assignment Feasibility

The candidate hardware apparatus for a mapping have to be able to support the performance requirements of the virtual components.

- For example, a 1000 MBit/s virtual link cannot be mapped to a path containing a 100 MBit/s substrate link.
- Likewise, the CPU computation capability requested by a virtual node has to be less than (or equal to) the CPU computation capability actually provided by a server.
- When redundancy is required, e.g. if we require that each functional link in the G(VNF) has to be protected against failures by allocating a spare companion link, both functional and spare links in the G(VNF) need to be assigned to link (or node) disjoint paths in the G(DC) of the DC to which the VNF has been mapped. In this way a single link (or node) failure in a physical apparatus does not compromise the virtualized service.
- Depending on the service and, in turn, on the VNF it belongs to, the VNFc can be characterised by specific requirements and could benefit from the ability to access to high performance computing platform features (like hardware and software accelerators, GPUs, etc.).

Since we are facing an online problem, the amount of physical resources available at any instance in time is the infrastructure hardware apparatuses in the DCs minus that allocated to VMs currently running on their servers in response to satisfying NS requests. Only when a service is terminated does it allocated resources

(computational and bandwidth demands) become available, and can be assigned, to other incoming service requests.

Resource requirements are modelled by annotating a NS with the computational demand for each node and bandwidth link associated with each VNFc involved. Likewise, the Network Infrastructure is annotated with node and link resources for each hardware apparatus. Demands and resources have to be matched in order to achieve a feasible mapping. This means that virtual demands are first mapped to the candidate hardware resources, and then only when all the virtual demands are mapped, the entire service can be allocated and hardware resources actually spent. Despite the claimed elasticity of the cloud, there will be cases (due to the dimension of the deployed physical infrastructure) where the infrastructure will not accept the allocation request.

4.1.2. Objective Functions Definition

Independently of the solution approach (flat, top-down or bottom-up, described in Section 4.2), top-level decisions and bottom level decisions may involve different objectives.

At the bottom level, that is the assignment of single VNF components to servers inside a DC, the main objective is load balancing. However, a more detailed descriptive modelling of the DC may yield different objectives. An appealing option includes inter-rack traffic as a term since it can lead to some reduction of energy consumption even if at the expense of load balancing.

At the top level (i.e. the assignment of VNF chains to DCs) user-value and economics oriented measures are more suitable. Top-level objective functions might reflect the value that the Marketplace gives to the service directly in terms of price.

The decision of which DC is most suitable for the deployment of the VNF chains could be based on several business criteria such as:

- A difference in the price of each DC for the customer. However, this would mean making more explicit to the customer the underlying infrastructure, which is not the focus of Network Functions Virtualisation (NFV) scheme. The Service Provider (SP, which in T-NOVA owns its own infrastructure) would choose the infrastructure according to his own criteria and the customer does not care about this (while the SP could decrease costs, energy consumption, etc.).
- Type of customer. This could be a distinction of several customer profiles, each pre-assigned to one DC. For example, VIP customers all grouped in the DC having the highest SLA scores). However, this option may lead to waste of resources.
- The SLA parameters and the capacity of the DCs. The Service Provider, if it had enough knowledge about it, could choose which of the DCs is able to cope with the overall demand and meet that particular SLA.

With respect to the criteria above and considering the T-NOVA customer point of view the most suitable approach is based on the SLA agreed in the Marketplace, which in turn will match with the price agreed in the Marketplace for that level of

service. The customer only requires that the service received matches what was contracted.

For instance, if a given price is assigned to a service depending on the requested Quality of Service (QoS), and that QoS is measured in terms of latency, additional computing flexibility and backup storage would be needed; then a linear combination of these measures can be used as an objective function of the top level assignment process. This information must come from the Marketplace (to be further researched later on T-NOVA).

4.1.3. Reconfiguration Issues

Since the SM is an online problem and the SM algorithms have to handle each service requirement as it arrives, reconfiguration issues arise.
We will study the feasibility of dynamic approaches, which will try to reconfigure VNFcs already mapped, without invalidating the original SLA, in order to both reorganize the resource allocation and to optimize DC resources utilization. This can be due to:

- Fragmentation of physical resources: as new services are embedded and others expire and release their resources from the NI, the embedding services become fragmented and the number of accepted services diminishes, resulting in a long-term revenue reduction.
- Changes in the service: a service may change in terms of topology, size and resources due to new requirements demanded by its users.
- Changes in the DC: network providers can update their networking infrastructure to cope with scalability issues and, hence, some DC increases its size and current virtual components can find different and more efficient allocations.
- Fault occurrences: in case of server/apparatus failures all assigned VNFcs need to be reallocated on the fly with the minimum possible impact on the agreed QoS.
- VM migration: the process of virtual machine migration between different physical servers without disconnecting the running application.

4.2. Proposed Approaches

The approaches for solving the SM problem in T-NOVA project can be grouped into three categories.

4.2.1. Flat Approaches

These approaches aim at solving the flat problem and, for this reason, are mainly viable for small dimension instances. In real life scenarios each DC can be composed by thousands of servers and the dimension of the EG(NI) can accordingly be very large. Nevertheless, ad hoc algorithms exploiting suitable clustered representations of DC can be used.

4.2.2. Top Down Approaches

These approaches aim at solving the first level problem, identifying a minimum cost mapping of each VNF in the service to a DC in the NI.

Then for each matched couple (VNF, DC), identified by solving the first level problem, top down approaches try to solve the second level problem.

Let us observe that when two or more VNFs are assigned to the same DC (which is always a possibility to be considered) the order in which the two corresponding second level problems are solved becomes relevant. Alternatively, the second level problem could consider as input into the union of the disjoint graphs associated to the two or more VNFs that tries to identify an overall assignment of all involved virtual components to the DC.

4.2.3. Bottom Up Approaches

These approaches try to solve (possibly in parallel) the second level problem for each possible couple (VNF, DC), saving an "overall mapping cost" for each couple.

Then they try to solve the first level problem by using the costs computed in the first step, to identify a final mapping of each VNF in the service to a DC in the NI.

In the following, we list the possible approaches that the partners involved in this task are investigating for solving the SM problem.

4.2.4. Multi-stage Network Service Embedding

This approach has been proposed by the **Gottfried Wilhelm Leibniz Universitaet Hannover** (LUH).

4.2.4.1. Main assumptions

- Each NF Service Provider advertises a PoP-level graph with link costs, and the NF costs, i.e., CPU cost at the DC.
- Two-level hierarchical DC topologies (i.e., fat trees).

4.2.4.2. Iterative Algorithm

1. Identify location-dependent VNFs (e.g., proxies; resources should be in proximity to the client's network).
2. Identify candidate DCs for each VNF in the service chain.
3. If there is no DC satisfying all VNF requirements and constraints, partition the service chain among DCs:
    - Formulation as (Integer) Linear Program.
    - Different objectives depending on the service and NF providers preference, for example:
        - Minimizing the client's expenditure.

- Maximizing load balancing across the DCs by considering (i.e., minimisation of) weight values that express NF Service Providers' preferences.

4. Upon partitioning, assign the VNFs to servers within the selected DCs:
    - Formulation as (Integer) Linear Program.
        - Objectives: Minimize inter-rack traffic and the number of used servers.
    - Alternative solution: Heuristic algorithm that aims at assigning the VNFs to the smallest number of racks and servers, while CPU load and bandwidth are balanced across the racks and servers.
5. Stitch together the VNF service chain segments (mapped to different DCs) with the assignment of virtual links connecting the DCs:
    - Objectives: To find the shortest path between a pair of DCs that offers the required amount of bandwidth.
    - Multi-commodity flow problem formulation.

Previous work/information relevant to this approach can be found in particular in [4].

4.2.5. VNF Scheduling over an NFV Infrastructure

This approach has been proposed by **Internet e Innovació Digital a Catalunya** (i2CAT), and it adds a temporal dimension to the problem, considering that the set of virtual network functions composing the different services need to be scheduled over the NFV infrastructure in order to optimise the service execution. It is complementary to the other approaches, since this is mainly focused on the scheduling of the different VNFs instead on the specific mapping on the physical infrastructure.

In a typical scheduling problem [53, 54] one has to find time slots in which activities should be processed under given constraints, such as resource constraints and precedence constraints between those activities (i.e. we need to find the corresponding time slots for the virtual network functions composing different network services to be executed over a given set of machines – or servers – considering that each service consists of a set of ordered virtual network functions).

4.2.5.1. Model

This problem can be formulated as a Resource Constrained Project Scheduling Problem (RCSP), in detail, a flexible job-shop problem.

4.2.5.2. Main assumptions

We have a set of N network services $NS_1,...,NS_N$, where each network service is composed of a set of virtual network functions, i.e. each NS_j consists of n_j virtual network functions F_{ij} with $(i=1,...,n_j)$, which have to be processed in the corresponding order $F_{1j} \rightarrow F_{2j} \rightarrow ... \rightarrow F_{nj}$, satisfying the precedence constraints

between the different virtual network functions that compose the corresponding service. Additionally, we have a set of *m* multi-purpose servers or machines[2] $M_1,...,M_m$ that can process the different virtual network functions. Each virtual network function F_{ij} must be processed for $p_{ij} > 0$ time units without pre-emption on a dedicated machine $u_{ij} \in \{M_1,...,M_m\}$. Each multi-purpose server can process only one virtual network function at a time. Furthermore, let us assume that there is sufficient buffer space between the different servers to store a network service if it finishes on one server and next server is still occupied by another function. Let us now define a schedule $S = (S_{ij})$, which is defined by the starting time of all network functions F_{ij}. We say a schedule is feasible if

- $S_{ij} + p_{ij} \leq S_{i+1,j}$ for all network services $j = 1,...,N$ and $i = 1,...,n_j - 1$, i.e. the precedences $F_{ij} \rightarrow F_{i+1,j}$ are respected, and
- $S_{ij} + p_{ij} \leq S_{uv}$ or $S_{uv} + p_{uv} \leq S_{ij}$ for all pairs F_{ij}, F_{uv} of functions with $u_{ij} = u_{uv}$, i.e. each server processes only one job at a time.

The objective is to determine a feasible schedule S with minimal makespan $C_{\max} = \max_{j=1}^{N}\{C_j\}$, where $C_j = S_{n_j,j} + p_{n_j,j}$ is the completion time of the Network Service NS_j. SLA parameters are not included in the model; it is assumed that a server capable of serving a given VNF will fulfil all the requirements of the specific VNF.

4.2.5.3. Solution

The problem may be solved with a two-stage approach, in the first, servers are assigned to the corresponding virtual network functions and in the second the resulting classical job-shop problem is solved, following guidelines proposed in [55].

4.2.5.4. Next Steps

Even if the scheduling problem is only considered from a theoretical perspective within the Orchestrator (and not implemented), we will propose different approaches following the proposed solution. The approaches will follow the same two-stage approach aiming at optimising different targets.

Furthermore, it is envisaged future work on the model in order to include assumptions and constraints coming from the implementation activities.

[2] In this subsection, the term "server" is utilised from the theoretical perspective used in literature of scheduling algorithms, to mean a minimal logical entity able to serve a virtual network function (and not the actual meaning of a physical server hosting VMs, etc.). In T-NOVA, the term server used in this section therefore indicates one CPU core, or one computing unit in general.

4.2.6. Reinforcement Learning Based Approach

This approach has been proposed by **Consorzio Per La Ricerca Nell' Automatica E Nelle Telecomunicazioni** (CRAT).

4.2.6.1. Main Goal

In order to provide a scalable and robust solution to the SM problem introduced in Section 4.2, a novel inter-DCs resource allocation algorithm, based on a Markov Decision Process (MDP) [56], is proposed. The main goal of this approach is to dynamically assign the service requests to a set of IT resources, with the aim of maximizing the expected revenue over time, satisfying the requirements requested by the users and taking into account also constraints of the use/availability of resources. In this scenario, the revenue has to be interpreted in the most general way. In fact the revenue could represent the profit associated to each NS assignment, but also the convenience from a load-balancing point of view.

4.2.6.2. Proposed Approach

Starting from the information provided by the other T-NOVA modules, (i.e. the NS required by the user (i.e. the Marketplace), the resources required to provide a service (i.e. the Network Service Descriptor), the resources available in each DCs (i.e. the Infrastructure Repository) and the revenue obtained by a certain assignment, we model the problem as a MDP. Thereafter, to find the optimal policy, this approach uses the Reinforcement Learning techniques (e.g. Value iteration, Policy Iteration, Q-Learning, etc.), in order to make the algorithm learn and to adapt its strategy to change. The main advantages of the MDP-based approach are the possibility to provide a solution that takes into account also the requests that could arrive in the near future and, furthermore, the possibility to run "offline" the learning phase and use the optimal policy to map "online" the NS requests.

To reduce the execution time during the learning phase, this approach uses a geographical partitioning of the DCs. In Figure 4-5 an example of geographical partitioning is provided, where DCs having the same colour belong to the same cluster. This assumption allows both to guarantee geographical constraints expressed by the user and to reduce the state space of the problem.

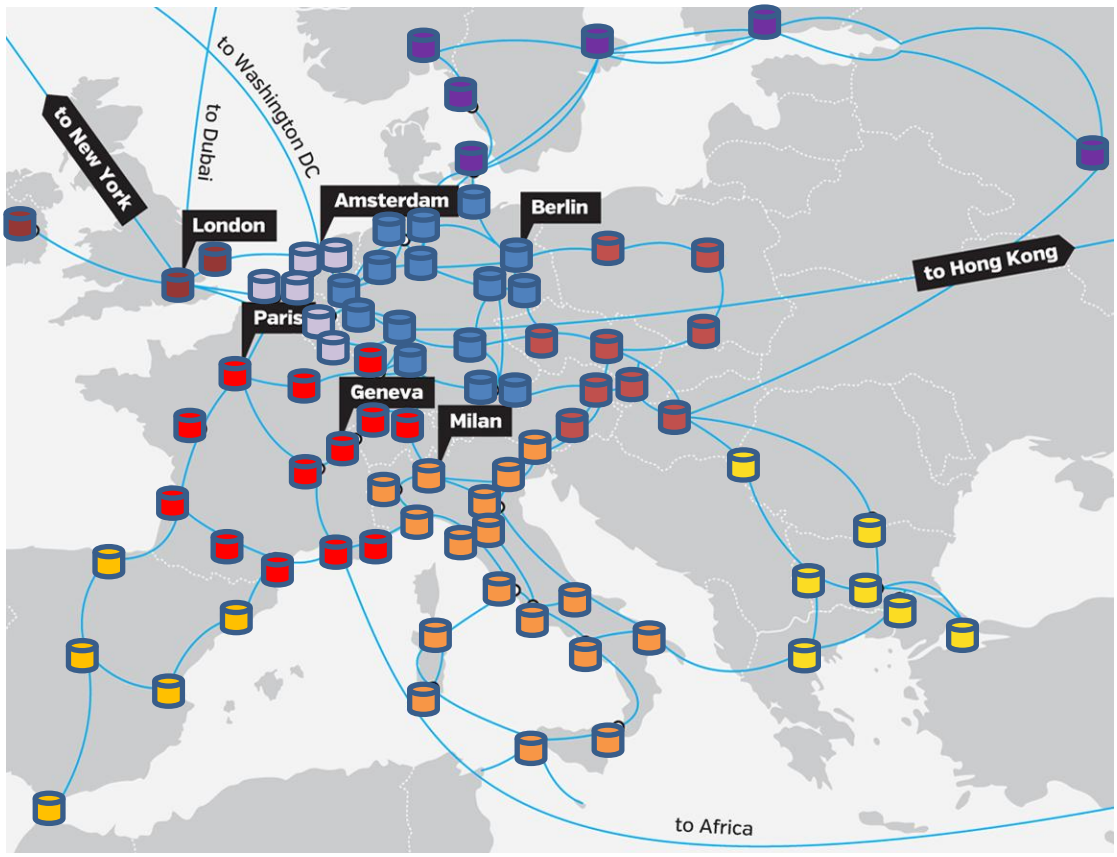

Figure 4-5: Example of geographical partitioning of DCs (source: http://www.interoute.com/sites/default/files/images/Geographic_map_europe_VDC_240214.png).

According to the classification introduced in Section 4.2, the proposed approach has to be considered basically as a bottom-up one. In fact, by performing the learning phase, the optimal policy takes into account all the information that arises from the compute node level. Even if the geographical partitioning of the DCs follows a top-down schema, the main logic of this approach is bottom-up.

4.2.7. Topology Aware Algorithms

This approach has been proposed by **Universita Degli Studi Di Milano** (UNIMI). The aim is to compare the top-down, bottom-up and flat approaches. The general technique used for solving the underline VNE problem will be local search based meta heuristics for the node-to-node mapping, and constrained network flow Integer Linear Programming (ILP) models for the link-to-path mapping.

Main assumptions:

- An annotated representation of nodes and links of the NI of the SPs and an annotated representation of nodes and links of each DC in the NI are stored in the project DBs.
- An annotated representation of nodes and links of the NS required and an annotated representation of nodes and links of each VNF in the NS are stored in the project DBs.

Besides node CPU capacity, or link bandwidth, the topological attributes of nodes have significant impact on the success and efficiency of mapping outcomes. Hence, we will try to use at the same time node specific features, resources and topological attributes. We plan to measure the topology-aware resource ranking of a node, in order to reflect the resource and quality of its connections. This approach would enhance the performances of the node-to-node mapping phase. The general structure will be:

1. Starting from their annotated representations, build the G(NI) and the G(NS) graphs. For each VNF in the NS build the G(VNF) graphs. For each DC in the NI estimate by means of topology aware techniques G(DC) or suitable sub graphs of DC.
2. Depending on the top-down, bottom-up or flat approach, for each VNE problem which requires to map G(V,A) into G(V',A') solve the constrained node-to-node and the link-to-path mapping problem.

Depending on the different VNE problem considered and on the information/cost coming from Marketplace we will adopt different objective functions to identify (sub) optimal solutions: client's expenditure, latency, inter-rack traffic, path length, and so on.

4.3. OpenStack VM Deployment Mechanisms

The second level problem of Intra DC VM allocation is strictly related to the logic of the Cloud Controller running within the DC it-self. This allocation problem, in fact, is about the choice of the specific compute node that has to host the VMs inside the specific DC. As delineated by Task 4.1 activities, and also in Deliverable 4.01 [57], the candidate technology selected for the Cloud Computing environment in T-NOVA is OpenStack.

OpenStack's scheduling and filtering mechanisms are used to select the compute node on which run a new VM is based on two main elements:

1. Host grouping.
2. Compute node Scheduling.

4.3.1. Host Grouping

Grouping the hosts within a DC can help to reduce the size of the problem, allowing the Orchestrator to select a subset of nodes where a VM has to be deployed and leave to OpenStack to specify the selection of the compute node within the identified subset.

Within OpenStack there are mainly two mechanisms to partition the DC: availability zones and host aggregates. Both those mechanisms can be useful and are under investigation for the task activities. Even if availability zones are implemented and configured in a similar way to host aggregates, they are usually used for different reasons and have different features.

An availability zone arranges OpenStack compute hosts into logical groups and provides a form of physical isolation and redundancy from other availability zones, (such as by using separate power supply or network segments). Availability zones could also help to separate different classes of hardware. The availability zones are visible at the API level (e.g. the Orchestrator would be able to see the availability zones configured in a DC and select one of them for the deployment of VMs). Another feature is that they are exclusive (e.g. a physical host can belong to one and only one availability zone) and they have to be defined at the server start-up time, so that it is not possible to change the availability zone of a host at runtime.

The host aggregates enable the administrator to partition OpenStack Compute deployments into logical groups for load balancing and instance distribution. Host Aggregates can be used to further partition an availability zone collecting hosts that either share common resources, such as storage and network, or have special properties, (such as trusted computing hardware, specific software features, or others). The host aggregates are mainly used for internal OpenStack scheduling purposes (for use with the Nova Scheduler, discussed later in this section). Moreover the host aggregates can be defined at runtime and a host can be included in more than one aggregate (e.g. they are not exclusive).

The selection of the specific techniques to cope with the second level allocation will be further investigated.

4.3.2. Nova Scheduler

Once a group of host has been selected, the specific compute node still needs to be identified. This is done within OpenStack by the Nova Scheduler. The built-in Nova Scheduler is based on a Filter mechanism: there are different built-in filters provided by OpenStack that can be used by the Scheduler to determine how to dispatch compute (and volume) requests. In fact, the nova-scheduler service determines which host a VM should launch on. The scheduler works in a two-phase approach, which are:

1. The Filtering phase.
2. The Weighting phase.

4.3.3. Nova Filters

Filtering a list a list of acceptable hosts. These hosts are the ones that satisfy specific conditions verified by the filters. There are a number of different ways to configure the Nova Filter:

- ***Using the OpenStack Filters***: there is a long list of filters that can be currently applied and that are available to filter the hosts according to specific criteria (available RAM on the host, specific computing capabilities, and so on). A complete list of the built-in filters is provided in [58]. There is a specific family of filters called Affinity Filters, which the "SameHostFilter" and "DifferentHostFitler" belong to: they are useful for scheduling the deployment of two VMs onto the same physical host or onto different hosts respectively. Again, more details can be found in [58].

- ***Implementing a Custom Filter***: a custom filter can be implemented using Python, extending the Filter standard Interface provided by Nova. This filter can implement a specific criterion according to the necessary orchestration strategy.
- ***Configuring a Filter Chain***: the filters (both built-in and custom) can be used as a chain: which means that the second filter will receive the output from the first, and so on. This will be useful to select the node using different criteria at the same time.

Each time the scheduler selects a host, it virtually consumes resources on it, and subsequent selections are adjusted accordingly.

4.3.4. Nova Weights

If the filtering phase provides more than one node, it is necessary to choose just one of them to host the new VM. In that case the weighting process will be applied. It is basically a way to associate a specific weight to each node, in order to find the best one suitable to the execution of that VM. This mechanism is based on objects called weighers, through which it is possible to assign a weight to each node. In order to prioritize one weigher against another, all the weighers have to define a multiplier that will be applied before computing the weight for a node. All the weights are normalized before being applied. Therefore the final weight for the object will be:

*weight = w1_multiplier * norm(w1) + w2_multiplier * norm(w2) + ...*

The default behaviour of the Filter Scheduler is to weigh hosts based on the following weighers:

- ***RAM Weigher***: hosts are weighted and sorted with the largest weight winning. If the multiplier is negative, the host with less RAM available will win (useful, for example, to implement a consolidation approach) or on the contrary if the multiplier is positive, the host with more RAM available will win (useful to implement a load balancing approach).
- ***Metrics Weigher***: this weigher can compute the weight based on various metrics of the compute node, those metrics has to be specified in the configuration file of each compute node and can have names chosen by the administrator.
- ***Io Ops Weigher***: the weigher can compute the weight based on the compute node workload. The default is to preferably choose light workload compute hosts. If the multiplier is positive, the weigher prefers choosing heavy workload compute hosts, the weighing has the opposite effect of the default.

It is also possible to develop custom weighers as a plugin to the existing framework.

4.4. Approach Comparison

Table 4-1 reports a comparison among the approaches proposed above.

Table 4-1: Comparison among SM approaches.

	Multi-stage Network Service Embedding	VNF Scheduling over an NFV Infrastructure	Reinforcement Learning based approach	Topology aware algorithms
Objectives	Cost minimization or load balancing	Cost minimization (minimal makespan)	Revenue maximization	Cost minimization or load balancing
Classification[3]	Top-down	N/A	Bottom-up	Bottom-up and Top-down (for comparison)
	Two-stage	Single stage	Two-stage	Two-stage
Optimization Stages	1st stage: exact 2nd stage: exact or heuristic	Exact or Heuristic	1st stage: approximate 2nd stage: approximate	1st level: exact or heuristic 2nd level: heuristic
Online/Offline	Online	Online	Hybrid Offline (Learning) and Online (Mapping)	Online
Resource constraints	CPU, bandwidth, location	Infrastructure resources available	SLA, location, node resources, link resources	SLA, node resources, link resources
Node/link mapping	Coordinated[4]	-	Coordinated	Coordinated
Dependencies	Optimizer, e.g. CPLEX[5]	N/A	No dependencies (algorithm developed in Matlab)	Optimizer, e.g. GLPK[6] or CPLEX

[3] Classification into top-down, bottom-up, or flat approaches, according to Section 4.2.

[4] Node mapping and link mapping takes place simultaneously, i.e., we can rethink node mapping if the overall cost is lower. In contrast, uncoordinated means, the node mapping is fixed before we start mapping the links, even at the risk of rejection.

[5] CPLEX, developed by IBM, is one of the most efficient commercial tools available for solving mathematical optimization problems (see http://www.ibm.com/software/commerce/optimization/cplex-optimizer/).

[6] GLPK is an open source tool for solving linear mathematical optimization problems (see https://www.gnu.org/software/glpk/).

4.5. Relationship and Inter Task Dependencies

This sub-section lists dependencies of this task from other tasks.

The SM algorithm needs to know:

- The data representing the Network Function Service with all its VNFs, to be mapped.
- The data representing the Network Infrastructure on which to install the service.
- Files with parameters for costs definitions and for constraints on feasible mapping solutions.

Table 4-2: Inter-tasks dependencies from Task3.3, Service Mapping.

Dependent Task	Dependency
Task 3.1: Orchestrator Interfaces	The information required for building the graph representing the NS and the graphs representing all the VNFs involved by the service. This involves not only the list of all nodes and links in the graphs but also their annotation, i.e. parameter values for CPU, bandwidth, maximum delays and so on, as well as the SLA thresholds for different parameters coming from the Marketplace, as part of the NSD. The information representing constraints on the mapping solution different from that represented in the annotation of nodes and arcs: a. E.g. information about all link to paths assignments that require node disjoint paths. b. All cost information that allows the SM algorithm to rank two different feasible mappings.
Task 3.2: Infrastructure repository	The information required for building the graph representing the NI and the graphs representing all the DCs composing the NI. In particular, Figure 11-1 contains all data required for the nodes representing hardware apparatuses, while studies are on-going in particular regarding the available information on internal DC links and the most efficient way to represent the information on NI and DC topology
Task 3.4: Service Provisioning, Management and Monitoring	The basic idea is not to use run-time information on the current use of resources. We need to know the resource required by all the implemented and running services, not their current use (the actual use will be less or equal to the required one). • If this information is stored and maintained in the Infrastructure repository of task 3.2 we do not need anything else. Finally, dealing with the output of the SM, the best feasible

	solution (if any) found by the SM algorithm will be written on a DB after an agreement on the format.
Task 4.4: Monitoring and Maintenance	The same as from Task 3.4 (above)

4.6. Conclusion and Future Work

The Service Mapping (SM) problem has been introduced in this Section and a mathematical formalization of the problem has been given. Different approaches under investigation by the partners have been proposed and compared, to the possible extent. Current and future work is devoted to further specifying and developing the SM algorithms (also taking into account the interdependencies with other tasks, as outlined in Section 4.5), their properties and their effectiveness/efficiency in solving the SM problem.

5. Service Provisioning, Management and Monitoring

The T-NOVA orchestrator, as defined in previous deliverables [59], is a core component of the T-NOVA architecture framework. Its primary role is to manage all network services and virtual network functions lifecycle, over distributed and virtualised network/IT infrastructures. The T-NOVA orchestrator is required to deploy and monitor T-NOVA network services by jointly managing network and cloud resources [6]. This section contains a basic description of the Orchestrator operations in order to ensure the automated lifecycle management of the orchestrator-related elements (i.e. network services and virtual network functions), as well as the detailed dependencies of the core functionalities with the rest of the task.

This section is structured as follows. First, a basic network service definition, including the abstract data model of the Network Service Descriptor (NSD) complemented with platform-awareness components from the infrastructure repository is introduced. Then, a detailed functional architecture of the orchestrator core, mainly containing the components in order to guarantee service management, provisioning, and monitoring operations at the orchestrator level is presented. Finally, the different implementation possibilities to be analysed in the next stage of the work within the context of the corresponding task are outlined.

5.1. Service Definition and Basic Descriptor

From a T-NOVA perspective, a network service is defined as a composition (graph) of different network functions. Following ETSI NFV definitions [60], the network service is defined by its functional and behavioural specification. On the one hand, the behaviour of the end-to-end service is the result of the combination of the individual network function behaviours as well as the chaining mechanisms. Thus, it can be said that, from a deployment perspective, a service is a concatenation of virtual network functions to be deployed on the corresponding NFV-Infrastructure. On the other hand, the operational specification of the service is provided in the Network Service Descriptor (NSD). ETSI defines the NSD as a deployment template for a network service referencing all other descriptors, which describe components that are part of the network service [44]. The NSD contains the set of static information elements used to instantiate and manage a network service over an NFV-enabled infrastructure. The NSD represents the reference data model to be considered within the orchestrator.

Figure 5-1 contains the basic network service descriptor included within the orchestrator. It is compatible with the NSD defined by the ETSI NFV standardisation group (i.e. it is ETSI compliant), but it includes some enhancements tied to specific T-NOVA requirements (e.g. platform-awareness).

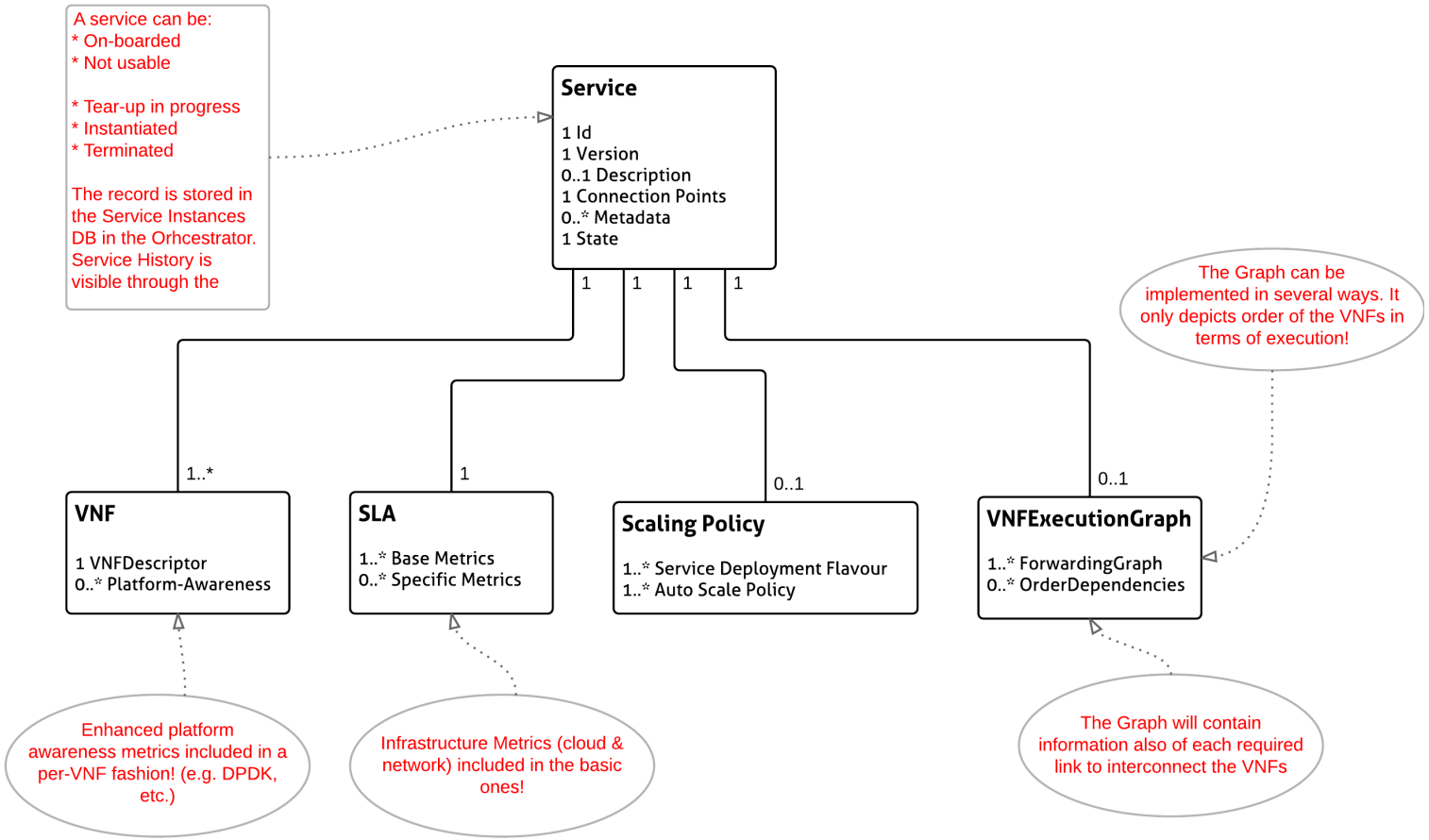

Figure 5-1: NSD considered at the orchestrator level

Table 5-1 to Table 5-5 contain the description of the attributes considered in the NSD and its details.

Table 5-1: NSD detailed attributes description.

Attribute	Description	Cardinality
Id	Identifier of the service descriptor	Mandatory
Version	Version for the service descriptor	Mandatory
Description	Description of the service	Optional (0..1)
Connection Points	Ingress and egress point of the service (e.g. virtual port, virtual NIC, physical NIC address, etc.)	Mandatory
Metadata	Metadata associated to the service	Optional (0..1)
State	Description of the state of the NS. It can take different values, depending on whether the NSD refers to a service in the NS catalogue; or a service instance in the NS Instances repository. In the NS Catalogue, the service state can be: on-boarded (i.e. the NS is ready to be instantiated), and not usable (i.e. is there any technical or business reason that prevents instantiation of the service). In the NS instances repository, the state can be: tear-up, in progress, instantiated, or terminated. For the second case, the service instances the state determines the state of the instantiated service. The state can be tear-up, in progress, instantiated, or terminated. The record of the service instances is stored in the repository.	Mandatory
VNF	The set of VNFs composing the network service. There is at least one VNF composing the service.	Mandatory (1..*)
SLA	The associated service level agreement, which needs to be enforced at the orchestrator level. There is only one SLA per service, with an associated set of metrics that should be monitored.	Mandatory
Scaling Policy	Automated scaling policies mechanisms	Optional (0..*)
VNF Execution Graph	The specific execution order and connectivity constraints for the VNF composing the services	Optional (0..1)

Table 5-2: Virtual Network Function.

Attribute	Description	Cardinality

VNF Descriptor	The descriptor of the corresponding VNF. Each VNF has one and only one VNF Descriptor. Details are provided in the corresponding WP5 deliverable [61]	Mandatory
Platform – Awareness	Includes platform awareness features 1. Features that are related to a capability of the physical host but are independent of its utilization (e.g. DPDK) 2. Features that are related to the specific instance usage/consumption (e.g. number of available GPUs)	Optional (0..*)

Table 5-3: Service Level Agreement.

Attribute	Description	Cardinality
Base Metrics	The generic set metrics common to all the network services that will be monitored (e.g. infrastructure metrics, network throughput metrics). The basic metrics will have an associated threshold for action initiation. The structure will be <Value, Threshold>, where Threshold will be in the form of Range <min, max>, so the SLA Enforcement can be performed within the Orchestrator.	Mandatory (1..*)
Specific Metrics	The set of specific metrics that can be defined in a per-service level fashion. This optional metrics may be different for different services. The structure of the specific metrics will be the same as the base metrics.	Optional (1..*)

Table 5-4: Scaling Policy.

Attribute	Description	Cardinality
Service Deployment Flavour	This field expresses the classes of service described by given KPIs. Those KPIs are checked by the system for each service to proceed with the auto-scale procedures.	Mandatory (1..*)
Auto Scale Policy	The specific scaling action that will be taken in case the condition is accomplished. <action, condition> are considered as a linked pair. The condition refers to a KPI contained in the deployment flavour field.	Mandatory (1..*)

Table 5-5: VNF Execution Graph.

Attribute	Description	Cardinality
Forwarding Graph (vnffg)	The forwarding graph for a given set of virtual network functions within the service. There may be different forwarding graphs for different types of	Mandatory (1..*)

	traffic (signalling, routing) as defined by ETSI. The graph can be implemented in a number of ways.	
Order Dependencies	Represents the strict dependencies in terms of execution order for a given set of VNFs composing the service (e.g. defines source and target VNFs, where the source is required to be executed before the target). This field is used to define the sequence in which various VNFs must be executed.	Optional (0..*)

5.1.1. ETSI NFV MANO Compliance

The following table contains the direct link between the T-NOVA NSD fields and the ETSI MANO fields, in order to provide the reader with a view of T-NOVA NSD compliance with ETSI standards.

Table 5-6: T-NOVA NSD links to ETSI MANO NSD.

T-NOVA Attribute	ETSI MANO NSD Attribute
Id	Id
Version	version
Description	description
Connection Points	connection_point
Metadata	-
State	-
VNF: EPA	-
VNF: VNF Descriptor	vnfd
SLA: Base Metrics	monitoring_parameter
SLA: Specific Metrics	-
Scaling Policy: Service Deployment Flavour	service_deployment_flavour
Scaling Policy: Auto Scale Policy	auto_scale_policy
VNFExecutionGraph: Forwarding Graph	vnffg
VNFExecutionGraph: Order Dependencies	vnf_dependency

5.1.2. Beyond ETSI NFV MANO

The standardisation group defines a basic network service descriptor in [62], which they describe as not a complete list of information elements constituting the NS but a minimum sub-set needed to on-board the network service.

As a consequence, the T-NOVA NSD goes beyond the basic NSD defined in the standardisation group. The novel concepts included in the NSD are as follows:

- **Platform-awareness**: there may be specific platform-constraints when deploying virtual network functions that are not considered at present (e.g. intense I/O requirements, ability to access high-performance instructions, or even direct access to other hardware specific features such as GPUs). This field of the NSD is based in the Enhanced Platform Awareness mechanism for OpenStack proposed within the infrastructure repository (Task 3.2). Therefore, by including this concept in the service descriptor the Orchestrator can benefit from the intelligent placement of VNFs developed to improve workload performance. Additionally, ETSI MANO contains the Virtual Deployment Unit (VDU). It is defined as a construct that can be used in an information model, supporting the description of the deployment and operational behaviour of a VNF, or the entire VNF if it was not componentised in subsets. The VDU is not as specific as the platform-awareness component in terms of platform hardware details;
- **Service Level Agreement:** the network service contains one associated SLA. The SLA comes from the Marketplace, where business agreements are managed. The SLA will contain two types of metrics. A common base set that will be common to all the network services considered within T-NOVA. Additionally VNF/NS specific metrics, which can be utilized or not depending on customer requirements. SLA enforcement at the orchestrator level will be performed as a function of the metrics included within the SLA field of the NSD. The SLA enforcement will be performed as a function of both the base and the specific metrics. SLA's at the marketplace level will also include business and commercial clauses (e.g. penalties, rewards), which are not considered within the Orchestrator. Although SLAs are not included, ETSI MANO NSD contains the monitoring parameter field, which enables the orchestrator to monitor some service parameters;
- **Scaling Policy**: the T-NOVA NSD includes a definition of the automated scaling policies at the service level. Even if manually triggered scaling operations are enabled through the management interfaces at the service (or even VNF) level, the proposed NSD includes the definition of automated scaling options for a given service (e.g. once a given threshold is reached for a given monitored metric, perform a scaling in action), based on the specific KPIs contained in the service deployment flavour. The mechanism included in the NSD is built using the <condition, action> pair, which defines the action to be taken once the KPI condition is accomplished. Scaling at the service level will contain conditions for modifying structures affecting the whole service (e.g. connections between VNFs). Specific VNF scaling policies will be declared in the corresponding VNF Descriptors, and the NSD will inherit them. The latest ETSI MANO reference document includes an initial field *auto-scaling* field for the service.

5.2. Orchestrator Overall Architecture

This section contains the initial description of the core functional architecture of the orchestrator fundamental blocks, responsible for the service-related functionalities (i.e. management, provisioning, and monitoring). The architecture has started from the overall identification of the building blocks for the orchestrator completed within deliverable D2.31 [6], jointly with the requirements specification and the functionalities to be implemented at the T-NOVA orchestrator level.

The next step in the orchestrator definition is the identification of the detailed functional components. It is assumed that each of the detailed functional components is dedicated to executing one specific task within the different T-NOVA workflows and use cases.

Four major components within the functional architecture have been identified:

- **Network Service Lifecycle Management**: includes all the components devoted to the execution of any task related to service-level lifecycle management (e.g. service monitoring, service scaling, service provisioning amongst others). This block also includes the NS Catalogue, the infrastructure repository, and the NS Instances repository;
- **Virtual Network Function Lifecycle Management**: includes all the components devoted to the execution of any task related to virtual network function lifecycle management (e.g. VNF deployment, VNF monitoring). This block also includes the VNF Catalogue and the VNF Instances repository;
- **External Interfaces:** includes all the external interfaces required to interact with the other T-NOVA architecture components. A new external interface has been included: the Orchestrator management interface, which will be used as the management and configuration entry point for the orchestrator itself;
- **Orchestrator Management and Configuration:** this functional group was not considered within the initial functional architecture. It is devoted to basic management and configuration operations for the orchestrator (e.g. number of mapping algorithms present, internal metrics to be monitored, configuration of the service monitoring type, or even user management).

The basic functionalities covered by each one of the major groups have been already identified and defined in a previous deliverable [6]. For the next iteration, we have identified functional blocks within each one of the groups, associated to a given function. The detailed T-NOVA Orchestrator architecture is depicted in Figure 5-2. The NFStore is also included in that figure, although it is not direct part of the Orchestrator, to increase readability.

The different components are not stand-alone components. They interact between them in different manners. The general and high-level workflows for the targeted use cases have been defined in the corresponding WP2 deliverable [59], and are not to be included within this deliverable to avoid unnecessary duplicity.

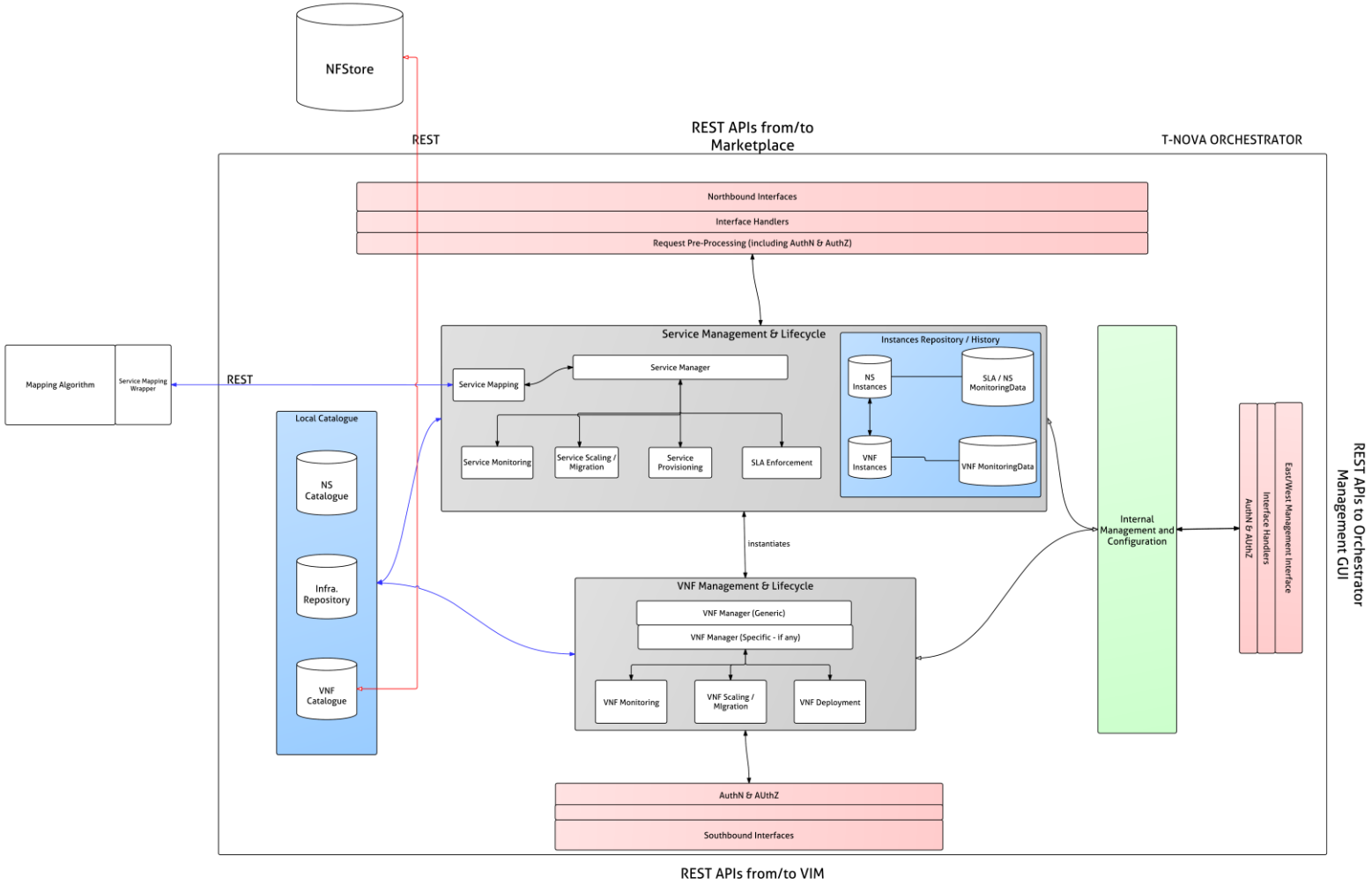

Figure 5-2: Overall Orchestrator Architecture.

5.2.1. Service Lifecycle Management

This block contains the functional components to support NS lifecycle management, and its service-level related operations.

The service lifecycle management functional group is composed of a number of different functionalities, based on the initial description provided in deliverable D2.31 [6]:

- **Service Manager:** This module manages all the service-related operations. It is the component responsible for coordinating interactions between the different functional components within the group. All the requests coming from the northbound interface (i.e. from the Marketplace) are processed by this component, which then starts the workflow execution for each request. It is also responsible for instantiating the VNF Manager(s) when provisioning a given service;
- **Service Mapping**: this module is responsible for providing the service mapping (i.e. the specific VNF placement in order to provision the service). The service mapping functional block is defined as a black box, with a set of pre-defined entries and the expected output. The internal details of the service mapping module are included in Section 4;
- **Service Monitoring**: this module is responsible for monitoring service-related metrics and coordinating the different VNF monitoring components, which are the components responsible for receiving information from the specific VNF agents. The Service Monitoring module will coordinate data collection from the specific VNF monitoring modules and will then integrate the data in order to obtain service-level metrics, which can be directly used by the SLA Enforcement module. The initial basic metrics included at the VNF level are explained later, although they do not represent the definitive set of metrics that will be collected (but the initial one which will be made available from the VIM to the Orchestrator). Service monitoring will offer an interface to the VNF Manager (the VNF Monitoring module), which will be used to post monitoring data by the VNF required to build service-level metrics. The service monitoring component will pre-process data received from the VNF monitoring modules (if necessary) in order to build and then store service-level metrics. Annex 10 contains the whole monitoring chain envisaged for the orchestrator. Furthermore, the Service Monitoring module will be responsible for getting information from the network connections between the different VNFs in situations where the NSD contains a **NFV Forwarding Graph**. This includes both the intra-DC network connections and the inter-DC network connections, managed by the Transport Network Manager. Additionally it will collect generic infrastructure data utilization and infrastructure metadata from VIM (from a generic perspective, not specific to any VNF). Service monitoring will be also responsible for filling the corresponding repositories with all the information at the service-level (i.e. the NS Monitoring Data and the NS State). The VNF repositories will be filled by the corresponding VNF Monitoring components;

- **Service Provisioning**: this module is responsible for instantiating a given NS over the NFV Infrastructure. This implies the following major actions within the Orchestrator at the service management level:
    - Instantiating the corresponding VNF Manager for each of the VNFs composing the service. There may be a different VNF Manager for each type of VNF or one common VNF Manager; this is a VNF Developer decision;
    - Configuring the corresponding service management instances (i.e. configuring service monitoring to be ready to coordinate VNF Monitoring elements, configure SLA enforcement with the specific metrics of the NS);
    - Requesting specific network connections to deploy the NFV Forwarding Graph defined in the NSD. This may encompass interaction with the Transport Network Manager interface in order to request specific network connections. Actions included in the T-Or-Tm interface which are utilized by the service for initiating provisioning are: (i) create network connection; (ii) remove network connection; (iii) update network connection; and (iv) get network connection information;
    - Request for the required infrastructure at the VIM level to deploy the VNFs considered within a NS. This includes releasing or updating the required infrastructure as well as it might include network connections between different VMs hosting the same VNF within one single DC infrastructure (i.e. not involving the Transport Network Manager connectivity services).
- **Service Scaling**: this module is responsible for coordinating the scaling actions at the service-level. These actions may be manually (request coming from the Marketplace) or automatically triggered. The service scaling will not interact with any other module outside of the service lifecycle management ones. It will only coordinate the scaling actions by means of utilizing the functionalities of the other modules. This module will be responsible for checking the automated scaling policies defined within the NSD. In scenarios where some of the conditions that are defined are reached, it will trigger the corresponding action by means of communicating with the service manager, who will then start triggering the actions which will modify the resources as expected. For manually triggered actions, which will come directly through the Northbound interface (i.e. update instantiated NS), the Service Manager will be the coordinating module;
- **SLA Enforcement:** this module is responsible for enforcing SLA accomplishment within the orchestrator. The module will be constantly retrieving the monitored data from the corresponding repository and checking the service status is compliant with the negotiated SLA. The metrics included in the NSD field (i.e. SLA based metrics, and SLA specific metrics) are monitored and checked for compliance. The specific metrics may change during service lifecycle (e.g. NS update request received during execution time from the Marketplace), in this case the SLA Enforcement needs to be notified and re-configured by the Service Manager.

Besides these functional component descriptions, the Service Manager deals with the following requests coming from the Marketplace through the Northbound interface. For further technical details on the interfaces (e.g. rationale, or REST/JSON model amongst others) please refer to Section 2.

Table 5-7: Service Manager method processing.

Method	Description	Modules	Actions
Create NS (T-Da-Or)	The marketplace notifies the orchestrator about a new NS (or an updated one)	Service Manager, NS Catalogue	The service manager will process the NSD and will add a new entry in the NS Catalogue accordingly. The status of the service will be created according to the NSD if the process ends without any problem.
Instantiate NS (T-Da-Or)	The marketplace requests the orchestrator to instantiate and deploy an existing NS	Service Manager, Service Mapping, Service Provisioning (execution time: Service Monitoring, SLA Enforcement)	The service manager will process the request. It will retrieve the NSD information from the NS repository. The Service Mapping will be responsible for calculating the specific allocation of any given service over the NFVI (i.e. decide which physical resources will be utilized to host the NS). The Service Provisioning module will be responsible for (i) instantiating the corresponding VNF Manager(s) and triggering the VNF deployment and the Transport Network connections (if required). The Service Manager after provisioning has been successfully instantiates and configures SLA Enforcement, Service Monitoring, and Service Scaling components.
Update a NS (T-Da-Or)	The marketplace requests to update a deployed NS	Service Manager, Service Mapping, Service Scaling, Service Provisioning	The Service Manager processes the request and triggers service scaling. This may or may not imply a new execution of the Service Mapping algorithm. The Service Scaling module will be responsible for coordinating the required actions before triggering the Service Provisioning component, which will directly communicate with the VNF Manager to proceed with the update of the required VNFs composing the service. The specific actions for updating the service (scale in/out, up/down) will come directly associated in the request. Updates to the SLA and Monitoring

			components will be executed if necessary.
Get a NS State (T-Da-Or)	The marketplace requests information of a deployed NS	Service Manager	The Service Manager will obtain information on the deployed NS directory from the NS Instances Repository, whose state will be updated within reasonable timeframes.

5.2.2. NS Instances Repository

This repository contains the basic information on the instantiated services (not the ones that are only created in the NS Catalogue). NS information is based on the NSD. Each service in this repository is linked to its corresponding NS Monitoring Data.

5.2.3. NS Monitoring Data Repository

The NS Monitoring Data repository will contain all the monitored data for each NS. The data may be different for each NS depending on the optional metrics defined in the corresponding SLA field of the NSD. The NS Monitoring repository is populated by the corresponding service monitoring component, and may be accessed by different components depending on the request or internal process executed (e.g. service scaling, SLA enforcement).

5.2.4. NS Catalogue

The NS catalogue will contain the set of on-boarded NSs. This implies that the NS catalogue will store all the NSDs, utilizing the aforementioned data model, but without including the VNF Descriptor, which is stored in the VNF Catalogue.

5.2.5. Implementation Possibilities for the Catalogues

There are various implementation options for the different catalogues and repositories that will be assessed in order to make the appropriate technology decision. There are various DBs approaches including relational DBs (e.g. PostgreSQL, or MySQL) and non-relational DBs (e.g. MongoDB, Apache Cassandra, or even Riak amongst others). A **micro-services** [43, 44] architectural pattern may also be applied at the orchestrator level, which may have an impact on the different repositories. This option will be analysed in detail during the next stage of the orchestrator implementation.

Several constraints need to be considered in order to take the technology decision for the catalogues and repositories (e.g. service inter-arrival time, service-related NSD data size, access frequency, update frequency, or even physical location of the repositories).

The repositories will be analysed on an individual basis in order to take the final technology implementation decision.

5.2.6. Infrastructure Repository

Please refer to Section 3 for further details on how the infrastructure repository will be implemented.

5.2.7. VNF Lifecycle Management

This component is devoted to the management of the VNF lifecycle. There will be one generic VNF Manager; in addition some VNFs may provide their own specific VNF Manager. In this scenario the T-NOVA Orchestrator (through the Service Provisioning module) will instantiate the specified VNF Manager, instead of instantiating the default one.

This section contains the description of the default VNF Manager and its associated functionalities, which are depicted in Figure 6.2. Where specific VNF Managers are implemented for VNFs, the functionalities to be covered are still the same, although the implementation may differ from the implementation of the default VNF Manager.

The VNF Manager is the component that will coordinate all the VNF-related operations within the orchestrator. The VNF Manager in turn will be coordinated by the Service Manager in order to ensure service continuity in terms of both management and monitoring.

5.2.7.1. VNF Deployment

The VNF deployment will be responsible for the instantiation and/or termination of different VNFs, following the generic instructions of the VNF Manager, which is coordinated by the Service Manager component. Once the VNF Deployment instantiates (through the Vnfm-Vnf interface) a given VNF, the VNF Manager will coordinate and configure the VNF Monitor and VNF Scaling components accordingly to enable all the operations over the VNFs in terms of management, and monitoring.

5.2.7.2. VNF Monitoring

This is the module responsible for receiving all the available VNF information. The information from the VNF will be obtained through the Vnfm-Vnf interface. The metrics to be collected per each VNF will be described in the corresponding VNF Descriptor.

The VNF Monitor will offer a REST interface that the corresponding VNF Monitoring Agent will utilize in order to post information associated to the metrics. The VNF Monitor will then populate the corresponding VNF Repository Data, at the same time it is coordinated by the corresponding service monitoring module, which will process (if required) the received information in order to build service-level information.

For the moment, initial metrics that can be exposed by the VIM Monitoring Manager as defined in WP4 follow:

In the VM/VNF Domain:

- CPU Utilisation
- No. of VCPUs

- RAM allocated
- RAM available
- Disk read/write rate
- Network interface in/out bit rate
- Network interface in/out packet rate
- No. of processes

In the Compute Node:

- CPU utilisation
- Available RAM
- Disk read/write rate
- Network i/f in/out rate

In the Storage:

- Read/Write rate
- Free Space

In the Network:

- Port in/out bit rate
- Port in/out packet rate
- Port in/out drops

Further information on these metrics is provided in the corresponding WP4 deliverable [57]. VNF Monitoring will also include an option to push metadata information for the given VNF.

5.2.7.3. VNF Scaling

The VNF Scaling component will be responsible for requesting appropriate scaling (i.e. in/out up/down) by the VIM. VNF scaling will utilise the VNF Monitoring Data in the repository in order to check the necessity of automatically scaling a given VNF. The VNF Descriptor will include all the scaling conditions for each VNF. These conditions are the ones that will be evaluated by the VNF Scaling.

5.2.7.4. VNF Catalogue

This catalogue is directly populated from the NF Store through the proposed REST/JSON interface (Please refer to Section 2 for further details of the interface). The Service Manager uses this catalogue to instantiate NSs, so the corresponding information from the VNFs composing the service can be used.

The VNF catalogue will store the available VNFs in the NF Store. It will contain an identifier for the VNF, the name, a link to the VNF image, and the VNF Descriptor, which completely details the VNF in terms of requirements and deployment dependencies. It may contain a link to the specific VNF Manager (optional) of the VNF.

5.2.7.5. VNF Instances Repository

Equivalent to the NS Instances Repository, this VNF repository will contain the basic information, based on the VNF Descriptor, of the instantiated VNFs. This repository will be linked to the NS Instances repository. Each VNF instance in this repository is linked to the VNF Monitoring Data.

5.2.7.6. VNF Monitoring Data

This repository contains the monitored data associated with the instantiated VNFs. The metrics to be monitored by the corresponding VNF agents are defined in the VNF Descriptor. The data may be different for each VNF, depending on the specific descriptor of the service.

The VNF Monitor module receives the data from the VNF Agent, and then fills the VNF Monitoring Data repository, where all the VNF-related data is stored. The repository may be accessed at any time by different components within the orchestrator.

5.2.8. External Interfaces

Please refer to Section 2 for further details on the external interfaces (northbound and southbound).

The management interface will primarily be used for the graphical user interface that will act as the internal system manager for the orchestrator. Details of the operations, monitoring options, and configuration envisaged are provided in the next sub-section.

The Web-based User Interface will be used as the management and configuration point of entry for the orchestrator. The GUI will enable mainly two major actions: (i) to visualize all the information stored in the different catalogues and repositories in a centralized manner, as well as to monitor all orchestrator system metrics (see next section); and (ii) to configure specific options for the orchestrator software system itself (e.g. logging levels).

The UI will be a stand-alone service, deployed independently from the orchestrator. It will run separately in a single web server (e.g. Tomcat), and will mainly use the different external interfaces of the orchestrator to retrieve the required information (or to send different configuration actions). For the sake of simplicity, the UI will primarily connect to the orchestrator through the management interface, avoiding the use the Northbound and Southbound interfaces. The basic functionalities of the UI will be provided through the REST management interface and will be directly handled by the orchestrator management and configuration component of the architecture.

The initial technology analysis for the graphical user interface is currently being performed. Tools and Frameworks like **Ruby-on-Rails**, **Spring Framework**, **JQuery**, and **D3.js** are some of the candidates that could be used to build the visual tool.

5.2.9. Internal Management and Configuration

This module is responsible for all the internal management and feasible configuration of the orchestrator itself. Being a software system, it requires some components for internal functional monitoring. Furthermore a web-based graphical interface tool will enable interaction with these different features through the east/west management interface of the orchestrator that will be specified and implemented in the next stage of the Task 3.4 execution plan.

5.2.9.1. Internal Monitoring

Generic Orchestrator information will include the following business metrics, which will contain mainly information relating to the internal components of the orchestrator as well as information on the relationships between the orchestrator and the other T-NOVA components. This information will be stored in the corresponding repositories within the internal management component. The component may interact directly with other components of the orchestrator in order to retrieve further information.

Table 5-8: Internal Business Metrics Monitoring.

Name	Description	Unit
#Create NS requests	Provides information on the number of created based on NS requests received from the Marketplace and their result (HTTP code only)	Integer
#Instantiate NS requests	Provides information on the number of NS instantiation requests received from the Marketplace and their result (HTTP code only)	Integer
#Terminate NS requests	Provides information on the number of NS termination requests received from the Marketplace and their result (HTTP code only)	Integer
#Update NS requests	Provides information on the number of NS update requests received from the Marketplace and their result (HTTP code only)	Integer
#Create VNF request	Provides information on the number of VNF creation requests received from the NF Store at the orchestrator and their result (HTTP code only)	Integer
#NS Scaling Requests	Provides information on the number of manually triggered NS Scaling requests have been received	Integer
#NS Scaling Actions	Provides information on the number of automatically triggered NS Scaling requests within the orchestrator	Integer
#SLA breaches	Provides information on the number of SLA breaches measured by the SLA enforcement module of the orchestrator	Integer
#SM Request	Number of service mapping requests performed by the algorithm	Integer

%SM Rate	Percentage of the successful mapping requests completed	Percentage
SM Execution Time	The average execution time of a mapping request	ms
#SM Requests	The number of service mapping requests	Integer
%SM Failed Requests[7]	The percentage of failed service mapping requests	%

This information will be exposed via a REST API and a web based GUI that can be viewed by the Orchestrator's administrator. Furthermore, this component will also be capable of exposing the size of the different repositories, mainly focusing on the monitored data of the instantiated NSs and VNFs.

In circumstances where the manager considers there is an issue with the orchestrator, the internal management system will enable temporal monitoring of some system-related metrics. This monitoring will be manually configurable from the REST interface.

The system-level monitoring will enable measurement of:

- Latency and throughput of the Orchestrator's external interfaces
- CPU and Disk usage by the Orchestrator's execution environment

It will also be possible to monitor application-level metrics in a holistic manner, enabling the identification of problems. This application monitoring will be disabled by default due to resources constraints within the orchestrator. However, if required, it can be enabled through the REST API.

5.2.9.2. Internal Configuration

Besides the internal monitoring metrics, this module will also be capable of configuring the orchestrator software platform. Initial configurations possible are related to:

- Service mapping algorithm priority list, in case there is more than one implementation of the mapping algorithm;
- Repositories configurations (enabled by the technology selected);
- Interfaces configuration;
- Start-up information required for the orchestrator booting system.

5.2.9.3. User Management

Furthermore, the internal management module will contain basic user management features. User management features will be the base for the Authentication and Authorization features to be included in the external interfaces of the orchestrator.

[7] A high value of failed Service Mapping requests could have two different reasons: either the algorithm is not correct or there is a lack of resources. Therefore, the reason why a certain request is considered not feasible should also be stored (and available for reading).

T-NOVA will not implement novel Authentication and Authorization concepts; instead it will rely on existing ones. The User Management will be performed through the web-based user interface of the orchestrator.

5.3. Relationship and Inter Task Dependencies

Task 3.4, Service Provisioning, Management and Monitoring is devoted to implement the T-NOVA orchestrator, including components from the other three tasks of the work package. The NSD as the initial and basic data model to be considered at the orchestrator level has been detailed, as well as the functional architecture to be implemented during the next stages of the project lifecycle. However, this requires interactions in terms of the outputs of supporting tasks and providing input into other tasks to ensure appropriate alignment with architecture and functionalities of the Orchestrator.

The dependencies of this Task3.4 other T-NOVA tasks are outlined in Table 5-9.

Table 5-9: Inter-tasks dependencies from Task3.4, Service Provisioning, Management and Monitoring.

Task	Dependency Description
Task 3.1: Orchestrator Interfaces	All the interfaces specification (e.g. REST structure and hierarchy of the northbound and southbound interfaces, expected HTTP results and behaviour, JSON objects, or even the syntactic pre-processing) will come from this task. Task 3.4 will integrate this into the different service management workflows
Task 3.2: Infrastructure repository	This task will be used mainly for: (i) including available infrastructure-related (i.e. static/basic metrics) information in the SLA metrics to be monitored; and (ii) enabling the orchestrator to retrieve any kind of information regarding the NFV infrastructure. Dynamic metrics will be provided by Task 4.4
Task 3.3: Service mapping	This task will implement the specific mapping algorithms, which will be included within the orchestrator (i.e. in the corresponding service lifecycle management internal module). The execution of the mapping algorithm can take place outside of the orchestrator; a single wrapper will use as the interface to include accordingly the results of the algorithm execution.

5.4. Conclusions and Future Work

This section contains the detailed functional architecture of the Orchestrator, built utilising the functional descriptions and requirements identified in the previous work package two deliverables. The architecture is composed of three major blocks, each one of them focused on a different functionality of the orchestrator (e.g. service-level management, or VNF-level management). The functional modules at the service-level have been related to the different interface actions defined in sub-section 5.2.

Furthermore, the Network Service Descriptor that will be considered within the orchestrator is presented, with the specification of all the parameters required from the T-NOVA perspective, extending the minimum set of fields to be present as defined by ETSI NFV in the basic NSD.

The relationship of this task with the other tasks of the work package is also included, defining the expected interactions between the different components of the whole system.

The next logical steps for the service management, provisioning, and monitoring tasks include two major milestones (and decision points):

- Technology selection for the implementation of the core internal components. There are different options for the management components of the Orchestrator, which need to be completely integrated with the rest of the T-NOVA components.
- Drafting of the software development plan for the second year of the project. Task 3.4 will develop the prototype for the functional architecture of the T-NOVA orchestrator, which has been presented in this manuscript. The software development task will be responsible for building the prototype based on the functional architecture, and the interfaces specification.

6. CONCLUSIONS

Two distinct kinds of interfaces have been identified in the T-NOVA Orchestrator platform:

- The first interface category is commonly found in Information Systems is a Northbound interface that is utilised to provide connectivity with the NF Store and the Marketplace;
- The second category of interface which is Southbound orientated is used to provide connectivity, with the VIM and the VNFs This interface will handle very high rates of metrics related data generated by the NFVI and the VNF's/NS's it is hosting.

The requirement for high data bandwidth support lead Task 3.1 to study tools and frameworks in the Streaming Data Processing area in addition to the common forms of interfacing two systems namely RESTful API's with JSON data object exchanged over HTTP. Supplementary work and experimentation with these tools and frameworks is still required in order to identify the mostly appropriate candidate solutions. More detailed specifications are under development for all expected system the operations, together with the other interfacing sub-systems: the NF Store, the Marketplace, the VIM and the VNFs.

Task 3.2 has conducted an analysis based on the candidate technologies that have been selected for the initial implementation of the T-NOVA IVM. Different options to retrieve infrastructural information have been identified and one has been elected for implementation evaluation. This solution extends the currently REST API implementation of OpenStack and OpenDaylight with a standalone repository of infrastructure information with new REST API's. This implementation is being designed in a manner to maintain as much compatibility with the current releases of the technologies while addressing the information deficits required to provide Enhanced Platform Awareness (EPA). This feature will play a pivotal role in supporting intelligent orchestration of VNF instantiation on virtualised cloud and compute infrastructures. The next step for Task 3.2 is to extend OpenStack beyond the current scheduling and filtering implementation in order to support the utilisation EPA data for the scheduling VNF specific resource e.g. SR-IOV capable NIC's etc.

This Service Mapping problem, i.e., the automatic determination of which resources to use in which Data Centres, has been clearly defined, and its objectives discussed, by Task 3.3, which led to the definition and discussion of various possible approaches to solve the problem. Further work is still needed to evaluate and implement each of the proposed approaches.

The initial functional architecture for the Orchestrator has been developed by Task 3.4. The architecture is composed of five major blocks, each one focusing in a group of functionalities: Service-level Management, VNF-level management, data storage, External Interfaces, and internal Management and Configuration. One of the most crucial and complex data structures to be exchanged between the Orchestrator and the Marketplace, the ETSI's Network Service Descriptor, has been extended with all the parameters required from the T-NOVA system defined. The next steps for Task

3.4 are to select the implementation technology stack for the core internal components and to draft a software development plan for the prototype of the T-NOVA Orchestrator, capable of supporting the above described features and using the selected frameworks.

7. REFERENCES

[1] *The 8 Requirements of Real-Time Stream Processing*, Stonebraker, M., Çetintemel, U. and Zdonik, S. (http://cs.brown.edu/%7Eugur/8rulesSigRec.pdf)

[2] *Apache Samza: LinkedIn's Real-time Stream Processing Framework*, by Riccomini, C. (https://engineering.linkedin.com/data-streams/apache-samza-linkedins-real-time-stream-processing-framework)

[3] *What is Hadoop?* (http://www-01.ibm.com/software/data/infosphere/hadoop/)

[4] Storm Project (http://storm-project.net/)

[5] *MapReduce: Simplified Data Processing on Large Clusters*, Dean, J. and Ghemawat, S. (http://research.google.com/archive/mapreduce.html)

[6] *T-NOVA Deliverable D2.31: Specification of the Infrastructure Virtualisation, Management, and Orchestration – Interim*, Gamelas, A. et al.

[7] OpenStack's Glance API (http://docs.openstack.org/developer/glance/glanceapi.html)

[8] OpenStack's Compute API (http://developer.openstack.org/api-ref-compute-v2-ext.html)

[9] *Architectural Styles and the Design of Network-based Software Architectures*, Fielding, R. (http://www.ics.uci.edu/%7Efielding/pubs/dissertation/top.htm)

[10] *10 Best practices for better REST-full API*, Jauker, S., 2014, (http://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/)

[11] {json:api} (http://jsonapi.org/)

[12] Apache Storm (http://storm.incubator.apache.org/)

[13] Apache Spark Streaming (https://spark.apache.org/streaming/)

[14] Apache Samza (http://samza.incubator.apache.org)

[15] Apache Thrift (https://thrift.apache.org)

[16] RabitMQ (http://www.rabbitmq.com/)

[17] Apache Kafka (http://kafka.apache.org/)

[18] Twitter (http://twitter.com)

[19] The (Twitter's) Streaming APIs (https://dev.twitter.com/streaming/overview)

[20] Storm vs. Spark Streaming: Side-by-side comparison, Huynh, X. (http://xinhstechblog.blogspot.pt/2014/06/storm-vs-spark-streaming-side-by-side.html)

[21] Monasca framework (https://www.openstack.org/assets/presentation-media/Monasca-Deep-Dive-Paris-Summit.pdf/)

[22] Apache Hadoop NextGen MapReduce (YARN) (http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html)

[23] Apache Mesos (http://mesos.apache.org/)

[24] LinkedIn (http://linkedin.com)

[25] *Survey of Distributed Stream Processing for Large Stream Sources*, Kamburugamuve, S. (http://grids.ucs.indiana.edu/ptliupages/publications/survey_stream_processing.pdf)

[26] Samza vs. Spark Streaming (http://samza.incubator.apache.org/learn/documentation/0.7.0/comparisons/spark-streaming.html)

[27] *In-Stream Big Data Processing* (https://highlyscalable.wordpress.com/2013/08/20/in-stream-big-data-processing/)

[28] *Why We Didn't Use Kafka for a Very Kafka-Shaped Problem* (http://engineering.onlive.com/2013/12/12/didnt-use-kafka/)

[29] Java programming language (https://www.oracle.com/java/)

[30] *Questioning the Lambda Architecture*, Kreps, J. (http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html)

[31] *Sneak peek: Google Cloud Dataflow, a Cloud-native data processing service* (http://googlecloudplatform.blogspot.pt/2014/06/sneak-peek-google-cloud-dataflow-a-cloud-native-data-processing-service.html)

[32] *REST vs. SOAP: How to choose the best Web service*, Dhingra, S. (http://searchsoa.techtarget.com/tip/REST-vs-SOAP-How-to-choose-the-best-Web-service)

[33] W3C Web-Services Architectural Group (http://www.w3.org/2002/ws/arch/)

[34] Protocol Buffers Overview (https://developers.google.com/protocol-buffers/docs/overview)

[35] Message Pack (http://msgpack.org/)

[36] *Twitter Will Open-Source Storm, BackType's "Hadoop of Real-Time Processing*, Finley, K. (http://readwrite.com/2011/08/05/twitter-will-open-source-storm)

[37] Berkeley University of California (http://berkeley.edu/)

[38] Java Virtual Machine Specification (https://docs.oracle.com/javase/specs/jvms/se8/html/)

[39] Clojure programming language (http://clojure.org/)

[40] Scala programming language (http://www.scala-lang.org)

[41] Task-parallelism (http://en.wikipedia.org/wiki/Task_parallelism)

[42] Data-parallelism (http://en.wikipedia.org/wiki/Data_parallelism)

[43] OpenStack Icehouse Release Notes (https://wiki.openstack.org/wiki/ReleaseNotes/Icehouse)

[44] Redfish Specification (http://www.redfishspecification.org/)

[45] Intel IPMI (http://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html)

[46] DMTF Desktop Management Interface (http://www.dmtf.org/standards/dmi)

[47] DMTF Cloud Management Initiative (http://dmtf.org/standards/cloud)

[48] OpenStack REST API (http://developer.openstack.org/api-ref.html)

[49] OpenDayLight API (https://wiki.opendaylight.org/view/OpenDaylight_Controller:REST_Reference_and_Authentication)

[50] OpenStack PCI-API Support (https://wiki.openstack.org/wiki/Pci-api-support)

[51] ETSI GS NFV 002 (http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf)

[52] *Virtual Network Embedding: A Survey*, Fischer A., Botero J.F., Beck M.T., de Meer H. and Hesselbach X., IEEE Communications Surveys & Tutorials, v. 15, n. 4, Fourth Quarter 2013

[53] *On the complex scheduling formulation of virtual network functions over optical networks*, Ferrer Riera, J., Hesselbach, X. et al, ICTON 2014 (Invited)

[54] *Virtual Network Function Scheduling: Concept and Challenges*, Ferrer Riera, J., Batalle, J., et al, SACONET 2014 (Invited)

[55] *Complex Scheduling*, Brucker, P., Knust, S., Springer Berlin-Heidelberg. ISBN-10 3-540-29545-3

[56] *A resource allocation algorithm of multi-cloud resources based on Markov Decision Process*, Oddi, G., Panfili, M., Pietrabissa, A., Suraci, V., Zuccaro, L., 5th IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom 2013), 2-5 December 2013, Bristol, UK

[57] *T-NOVA Deliverable D4.01: Interim Report on Infrastructure Virtualisation and Management*, McGrath, M., et al.

[58] OpenStack Icehouse Nova Scheduling Configuration Guide, (http://docs.openstack.org/icehouse/config-reference/content/section_compute-scheduler.html)

[59] *T-NOVA Deliverable D2.21: Overall System Architecture and Interfaces*, Xilouris, G., et al.

[60] *ETSI ISG NFV: GS-NFV-003 Network Functions Virtualisation (NFV); Terminlogy for main concepts in NFV. 2013-10*

[61] *T-NOVA Deliverable D5.01:Interim Report on Network Functions and associated Framework*, Comi, P. et al.

[62] *ETIS ISG NFV: GS-MAN-001 Network Function Virtualization (NFV) Management and Orchestration. 2014-11* (http://docbox.etsi.org/ISG/NFV/Open/Latest_Drafts/nfv-man001v081 management and orchestration.pdf)

[63] Micro-Services Architecture (http://microservices.io/)

[64] Micro-Services Resources (http://blog.arkency.com/2014/07/microservices-72-resources/)

[65] GitHub Developer API Overview (https://developer.github.com/v3)

[66] HTTP API design (https://github.com/interagent/http-api-design)

[67] GitHub (http://github.com)

[68] Heroku (http://heroku.com)

[69] Web Linking (http://tools.ietf.org/html/rfc5988)

[70] ISO 8601 (https://www.dmoz.org/Science/Reference/Standards/Individual_Standards/ISO_8601)

8. LIST OF ACRONYMS

Acronym	Explanation
API	Application Programming Interface
CIMI	Cloud Infrastructure Management Interface
DC	Data Centre
DMI	Desktop Management Interface
DMTF	Distributed Management Task Force
DPDK	Data Plane Development Kit
EPA	Enhanced Platform Awareness
ETSI	European Telecommunications Standards Institute
HDFS	Highly Distributed File System
HTTP	Hyper-Text Transfer Protocol
ILP	Integer Linear Programming
IPMI	Intelligent Platform Management Interface
JSON	JavaScript Object Notation
MANO	(ETSI NFV) Management and Orchestration
MDP	Markov Decision Problem
MIF	Management Information Format
ML	Modular Layer
NAT	Network Address Translator
NFS, NF Store	Network Function Store
NFV	Network Functions Virtualization
NI	Network Infrastructure
NS	Network Service
NSD	Network Service Descriptor
Or-Vi	Interface between the Orchestrator and the VIM
PoP	Point of Presence
QoS	Quality of Service
RCSP	Resource Constrained Project Scheduling Problem
SLA	Service Level Agreement
SM	Service Mapping

SP	Service Provider
SR-IOV	Single Root I/O Virtualization
T-Ac-Or	Interface between T-NOVA Accounting (Marketplace module) and the Orchestrator
T-Br-Or	Interface between T-NOVA Brokerage (Marketplace module) and the Orchestrator
T-Da-Or	Interface between T-NOVA Dashboard (Marketplace module) and the Orchestrator
T-Sla-Or	Interface between T-NOVA Servile Level Agreement (Marketplace module) and the Orchestrator
VCPU	Virtual CPU
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNE	Virtual Network Embedding
VNF	Virtual Network Function
VNFc	Virtual Network Function component
VNFD	Virtual Network Function Descriptor
VNFM	Virtual Network Function Manager
Vnfm-Vnf	Interface between the VNF Manager and VNFs
vNIC	virtual Network Interface Controller

Annexes

9. ANNEX A: THE ORCHESTRATOR API

This Annex lists the options and standards supporting the Orchestrator's APIs, as well as the APIs them selfs.

9.1. Base URL

As a base URL, propose something like

```
http(s)://apis.t-nova.eu/v1
```

this base URL will be referred to below as

```
<base-url>
```

9.2. Formats and conventions

For formats and conventions the GitHub Developer API is followed [65] and [66]. These guides describe a set of HTTP+JSON API design practices, that were originally extracted from the work of both GitHub [67] and Heroku [68] while designing their platform's API. We do not intend to establish **the** way to design such kind of APIs (in fact, these two references have some inconsistencies between them), but instead look for a good and consistent way to design the APIs.

9.2.1. Authentication and Authorization

At this early stage **authentication** or **authorization** are currently not in scope, since additional work with the interfacing systems is required.

9.2.2. Pagination

Requests that return multiple items will be paginated to **20 items** by default. Additional pages can be requested with the `?offset` parameter. For some resources, a **custom page size up to 100** with the `?limit` parameter can be set.

An example of this would be:

```
$ curl '<base-url>/vnfs/?offset=2&limit=100'
```

Note that page numbering is 1-based and that omitting the `?offset` parameter will return the first page. The pagination information is included in the Link header [69], and it is considered a good practice to follow these link header values (instead of constructing the URLs by hand). This link data looks something like:

```
Link: <<base-url>/vnfs/?offset=3&limit=100>;
rel="next", <<base-url>/vnfs/?offset=50&limit=100>;
rel="last"
```

The possible `rel` values are shown in Table 9-1.

Table 9-1: Possible values for the rel parameter in linking web pages.

Name	Shows the URL of the

`next`	Immediate next page of results
`last`	Last page of results
`first`	First page of results
`prev`	Immediate previous page of results

9.2.3. Querying, Sorting and Filtering

Fields who can be queried, sorted or filtered, for performance reasons as described in the following sections.

9.2.4. Timestamps format

All timestamps are returned in **ISO 8601** [70] format:

```
YYYY-MM-DDTHH:MM:SSZ
```

An example of this is:

```
"2014-11-21T10:18:23Z"
```

9.3. Standard Return Codes and Errors

The project will use standard HTTP API return codes and errors shown in Table 9-2.

Table 9-2: Standard HTTP return codes and errors to be used.

Code	Description
200	**OK:** Everything is working
201	**OK:** New resource has been created
204	**OK:** The resource was successfully deleted
304	**Not Modified:** The client can use cached data
400	**Bad Request:** The request was invalid or cannot be served. The exact error should be explained in the error payload. E.g. „The JSON is not valid“
401	**Unauthorized:** The request requires an user authentication.
403	**Forbidden:** The server understood the request, but is refusing it or the access is not allowed.
404	**Not found:** There is no resource behind the URI.
422	**Unprocessable Entity:** Should be used if the server cannot process the entity, e.g. if an image cannot be formatted or mandatory fields are missing in the payload.
500	**Internal Server Error:** API developers should avoid this error. If an error

	occurs in the global catch blog, the stack trace should be logged and not returned as response.

9.4. Proposed interfaces

This sub-section specifies some of the Orchestrator's external interfaces. Further work is still needed with the other Work Packages (WP4, WP5 and WP6) in order achieve an optimal solution for the T-NOVA architecture. Definition of the interfaces that can be called from the interfacing systems, and will work with them to specify the interfaces the Orchestrator will call is still in progress.

As outlined above, a REST abstraction of the interface architecture will be used with the JSON data-interchange format over HTTP. Possible errors for each operation are defined in section 9.3.

9.4.1. Orchestrator and NFStore Interactions

The **NFStore** calls the Orchestrator to announce new or updated VNFs, to monitor VNF usage or to delete an unused VNF.

9.4.1.1. Create a VNF

Adds a new VNF to the VNF Catalogue.

The decision about whether this method should respond synchronously or asynchronously has not yet been taken.

Method and Endpoint:	POST /orchestrator/vnfs
Parameters:	• name (string). **Required.** Name of the VNF to be created; • `vnf-image (string)`. **Required.** URL of the VNF image, to be used when provisioning the VNF as part of a NS. • `vnf-manager (string)`. URL for the manager specific to the VNF

Sample request:	`$ curl -X POST <base-url>+'/orchestrator/vnfs' \`
	`-H 'Content-Type: application/json' \`
	`-d \`
	`'{`
	`"name": "vnf-one",`
	`"vnf-image": "https://api.t-nova.eu/v1/nfstore/vnfs/123/image"`
	`}'`

Sample response:	`Status: 201 OK`
	`Location: https://api.t-nova.eu/v1/orchestrator/vnfs/123`
	`{`
	`"id":"123",`
	`"name":"vnf-one",`
	`"vnf-image": "https://api.t-nova.eu/v1/nfstore/vnf/123/image",`
	`"vnf-manager": "",`
	`"created_at":"2014-11-21T14:18:09Z",`
	`"updated_at":"2014-11-21T14:18:09Z"`
	`}`

9.4.1.2. Update a VNF

Updates an existing VNF.

The decision about whether this method should respond synchronously or asynchronously has not yet been taken. Also not taken is the decision on wheter an update should use PUT or PATCH: a PATCH should be preferred to a PUT whenever updating only part of a resource, but using the former implies careful design to ensure atomicity (i.e., any GET on the same resource must be blocked and wait for the PATCH is complete), while the later is atomic but may imply a significant overhead when updating complex resources.

Method and Endpoint:	PUT /orchestrator/vnfs/<vnf_id>
Parameters:	• name (string). **Required.** Name of the VNF to be created; • `vnf-image (string)`. **Required.** URL of the VNF image, to be used when provisioning the VNF as part of a NS; • `vnf-manager (string)`. URL for the manager specific to the VNF.

Sample request:	`$ curl -X PUT <base-url>+'/orchestrator/vnfs/123' \`
	`-H 'Content-Type: application/json' \`
	`-d \`
	`'{`
	`"name": "new-vnf-one-name ",`
	`"vnf-image": "https://api.t-nova.eu/v1/nfstore/vnfs/123/image"`
	`}'`

Sample response:	`Status: 201 OK`
	`Location: https://api.t-nova.eu/v1/orchestrator/vnfs/123`
	`{`
	`"id":"123",`
	`"name":"new-vnf-one-name",`
	`"vnf-image": "https://api.t-nova.eu/v1/nfstore/vnfs/123/image",`
	`"vnf-manager": "",`
	`"created_at":"2014-11-21T14:18:09Z",`
	`"updated_at":"2014-12-03T10:38:53Z"`
	`}`

9.4.1.3. Delete a VNF

Deletes an existing VNF.

Method and Endpoint:	DELETE /orchestrator/vnfs/<vnf_id>
Parameters:	(none)
Sample request:	`$ curl -X POST <base-url>+'/orchestrator/vnfs/123' \` `-H 'Content-Type: application/json'`

Sample response:	`Status: 204 OK`
	`Location: https://api.t-nova.eu/v1/orchestrator/vnfs/123`
	`{`
	`"id":"123",`
	`"name":"new-vnf-one-name",`

	"vnf-image": "https://api.t-nova.eu/v1/nfstore/vnfs/123/image", "vnf-manager": "", "created_at":"2014-11-21T14:18:09Z", "updated at":"2014-12-03T10:38:53Z" }

9.4.1.4. Show a VNF

Returns a specific VNF's data.

Method and Endpoint:	GET /orchestrator/vnfs/<vnf_id>
Parameters:	None
Sample request:	`$ curl <base-url>+'/orchestrator/vnfs/123' \` `  -H 'Content-Type: application/json'`

Sample response:	Status: 200 OK
	Location: **https://api.t-nova.eu/v1/orchestrator/vnfs/123**
	{
	"id":"123",
	"name":"new-vnf-one-name",
	"vnf-image": "https://api.t-nova.eu/v1/nfstore/vnfs/123/image",
	"vnf-manager": "",
	"created_at":"2014-11-21T14:18:09Z",
	"updated at":"2014-12-03T10:38:53Z"
	}

9.4.1.5. List VNFs

Returns a list of VNFs already provided. Due to the possible large number of stored VNFs, the list returned may have to be paginated [10]. Querying, sorting and filtering parameters can also be used, as described above.

Method and Endpoint:	GET /orchestrator/vnfs
Parameters:	none
Sample request:	`$ curl <base-url>+'/orchestrator/vnfs' \` `  -H 'Content-Type: application/json'`
Sample response:	Status: 200 OK Location: **https://api.t-nova.eu/v1/orchestrator/vnfs** { [{ "id":"123", "name":"new-vnf-one-name", "vnf-image": "https://api.t-nova.eu/v1/nfstore/vnfs/123/image", "vnf-manager": "", "created at":"2014-11-21T14:18:09Z", "updated at":"2014-12-03T10:38:53Z" }, { "id":"456", "name":" vnf-two", "vnf-image": "https://api.t-nova.eu/v1/nfstore/vnfs/456/image",

```
    "vnf-manager": "",
    "created_at":"2014-12-03T10:52:12Z",
    "updated_at":"2014-12-03T10:52:12Z"
    }
  ]
}
```


9.4.2. Orchestrator called by the Marketplace

The **Marketplace** calls the Orchestrator in a support of a number of requirements as outlined in Table 9-3.

Table 9-3: Requirements for the Interface between the Marketplace and the Orchestrator.

Number	Requirement	Interface
1	The Marketplace is notified about new, updated or deleted VNFs available in the NF Store	The real need and purpose of this interface is still under discussion: we might have the NFStore directly connecting with the Marketplace or through the Orchestrator.
2	The Marketplace is notified about (at least part of) the VNFDs of the available VNFs	T-Br-Or
3	The Marketplace notifies the orchestrator about new, updated or deleted Network Services (NSs)	Create a NS, Update a NS, Delete a NS (T-Da-Or)
4	The Marketplace notifies the orchestrator to instantiate and deploy an existing NS	Instantiate a NS, Deploy a NS instance (T-Da-Or)
5	The Marketplace notifies the orchestrator about new configuration parameters for an already deployed NS	Update a NS (T-Da-Or)
6	The Marketplace inquires the orchestrator about the state of a given NS instance	Show a NS (T-Da-Or)
7	The Marketplace is notified about changes in state of currently deployed NSs	(T-Ac-Or)
8	The Marketplace is notified with currently running NS metrics	(T-Sla-Or)
9	The Marketplace notifies the orchestrator to stop a given NS instance	Stop a NS instance (T-Da-Or)

While analyzing this list of requirements, a need for a generic requirement for managing a NS instance's state, like `new`, `running` and `stopped` has been detected.

9.4.2.1. Create a NS

The call adds a new NS to the NS Catalogue. Further work is needed to decide between a synchronous operation and an asynchronous one that notifies its caller later with eventual errors.

Method and Endpoint:	POST /orchestrator/network-services
Parameters:	• name (string). **Required.** Name of the NS to be created; • vnfs (array). **Required**. The list of VNF ids composing the service (NS creation will in the near future have many more parameters)

Sample request:	`$ curl -X POST <base-url>+'/orchestrator/network-services' \`
	`  -H 'Content-Type: application/json' \`
	`  -d \`
	`  '{`
	`  "name": "ns-one",`
	`  "vnfs": [ 123, 456]`
	`  }'`

Sample response:	`Status: 201 OK`
	`Location: https://api.t-nova.eu/v1/orchestrator/network-services/987`
	`{`
	`"id":"987",`
	`"name":"ns-one",`
	`"vnfs": [`
	`  { "id": "123",`
	`  "link":"https://api.t-nova.eu/v1/orchestrator/vnfs/123"`
	`  },`
	`  { "id": "456",`
	`  "link":"https://api.t-nova.eu/v1/orchestrator/vnfs/456"`
	`  }`
	`],`
	`"created at":"2014-11-21T14:18:09Z",`
	`"updated at":"2014-11-21T14:18:09Z"`
	`}`

9.4.2.2. Update a NS

Updates an existing NS. Further work is needed to decide between a synchronous operation and an asynchronous one that notifies its caller later with eventual errors. T-NOVA will also work on the dichotomy between using PUT or PATCH to update a resource: while PUT is an atomic operation, PATCH must be made atomic (i.e., no GET operation on the same resource should be answered before the PATCH is complete).

Method and Endpoint:	PUT /orchestrator/network-services/<network_service_id>
Parameters:	• name (string). **Required.** Name of the NS to be updated; • vnfs (array). **Required**. The list of VNF ids composing the service (NS updating will in the near future have many more parameters)

Sample request:	`$ curl -X PUT <base-url>+'/orchestrator/network-services/987' \`
	`  -H 'Content-Type: application/json' \`
	`  -d \`

	'{ "name": "ns-one-new-name", "vnfs": [123, 456] }'

Sample response:	Status: 201 OK
	Location: **https://api.t-nova.eu/v1/orchestrator/network-services/987**
	{
	"id":"987",
	"name":"ns-one-new-name",
	"vnfs": [
	{ "id": "123",
	"link":"https://api.t-nova.eu/v1/orchestrator/vnfs/123"
	},
	{ "id": "456",
	"link":"https://api.t-nova.eu/v1/orchestrator/vnfs/456"
	}
	],
	"created at":"2014-11-21T14:18:09Z",
	"updated_at":"2014-12-03T13:48:23Z"
	}

9.4.2.3. Delete a NS

Deletes an existing NS.

Method and Endpoint:	DELETE /orchestrator/network-services/<network_service_id>
Parameters:	(none)
Sample request:	$ curl -X DELETE <base-url>+'/orchestrator/network-services/987' \ -H 'Content-Type: application/json'

Sample response:	Status: 200 OK
	Location: **https://api.t-nova.eu/v1/orchestrator/ network-services/987**
	{
	"id":"987",
	"name":"ns-one-new-name",
	"vnfs": [
	{ "id": "123",
	"link":"https://api.t-nova.eu/v1/orchestrator/vnfs/123"
	},
	{ "id": "456",
	"link":"https://api.t-nova.eu/v1/orchestrator/vnfs/456"
	}
	],
	"created_at":"2014-11-21T14:18:09Z",
	"updated at":"2014-12-03T13:48:23Z"
	}

9.4.2.4. Show a NS

Returns all data concerning a single service, including NS instances and their status.

Method and Endpoint:	GET /orchestrator/network-services/<network_service_id>
Parameters:	(none)

Sample request:	`$ curl <base-url>+'/orchestrator/network-services/987' \ -H 'Content-Type: application/json'`

Sample response:	`Status: 200 OK`
	`Location: https://api.t-nova.eu/v1/orchestrator/ network-services/987`
	`{`
	`"id":"987",`
	`"name":"ns-one-new-name",`
	`"vnfs": [`
	`{ "id": "123",`
	`"link":"https://api.t-nova.eu/v1/orchestrator/vnfs/123"`
	`},`
	`{ "id": "456",`
	`"link":"https://api.t-nova.eu/v1/orchestrator/vnfs/456"`
	`}`
	`],`
	`"instances":[`
	`{`
	`"id":"456",`
	`"ns-id":"987",`
	`"status":"stopped",`
	`"created at":"2014-11-24T16:42:21Z",`
	`"updated_at":"2014-11-24T16:42:21Z"`
	`}`
	`],`
	`"created at":"2014-11-21T14:18:09Z",`
	`"updated at":"2014-12-03T13:48:23Z"`
	`}`

9.4.2.5. List NSs

Returns a list of NSs already provisioned. Due to the possible large number of stored NSs, the list returned might have to be paginated (see Section 9.2.2, above). Querying, sorting and filtering parameters can also be used.

Method and Endpoint:	GET /orchestrator/network-services
Parameters:	(none)
Sample request:	`$ curl <base-url>+'/orchestrator/network-services' \ -H 'Content-Type: application/json'`

Sample response:	`Status: 200 OK`
	`Location: https://api.t-nova.eu/v1/orchestrator/ network-services`
	`{`
	`"id":"987",`
	`"name":"ns-one-new-name",`
	`"vnfs": [`
	`{ "id": "123",`
	`"link":"https://api.t-nova.eu/v1/orchestrator/vnfs/123"`
	`},`
	`{ "id": "456",`
	`"link":"https://api.t-nova.eu/v1/orchestrator/vnfs/456"`
	`}`
	`],`
	`"instances":[`
	`{`
	`"id":"456",`
	`"ns-id":"987",`
	`"status":"stopped",`
	`"created at":"2014-11-21T14:18:09Z",`
	`"updated at":"2014-11-25T10:01:52Z"`
	`},`

	{
	"id":"456",
	"ns-id":"987",
	"status":"stopped",
	"created at":"2014-11-21T14:18:09Z",
	"updated at":"2014-11-25T10:01:52Z"
	}
	],
	"created at":"2014-11-21T14:18:09Z",
	"updated at":"2014-12-03T13:48:23Z"
	}

9.4.2.6. Instantiate a NS

Requests the instantiation of an already created NS.

Method - Endpoint:	POST /orchestrator/ns-instances
Parameters:	• ns-id (string). **Required.** Id of the NS to be instantiated; (NS instance creation will in the near future have many more parameters)

Sample request:	`$ curl -X POST <base-url>+'/orchestrator/ns-instances' \`
	`-H 'Content-Type: application/json' \`
	`-d \`
	`'{`
	`"ns-id": "987"`
	`}'`

Sample response:	`Status: 201 OK`
	`Location: https://api.t-nova.eu/v1/orchestrator/ns-instances/456`
	`{`
	`"id":"456",`
	`"ns-id":"987",`
	`"status":"new",`
	`"created_at":"2014-11-21T14:18:09Z",`
	`"updated_at":"2014-11-21T14:18:09Z"`
	`}`

9.4.2.7. Deploy a NS Instance

Requests the deployment of an already instantiated NS.

Method and Endpoint:	PUT /orchestrator/ns-instances/<ns_instance_id>
Parameters:	• status (string). **Required**. Status the instance is required to go to. Are there other status than `deployed`, `undeployed` and `new`? E.g., `running` or `stoped`? (The issue of which states should a NS Instance go through is still opened)

Sample request:	`$ curl -X PUT <base-url>+'/orchestrator/ns-instances/456' \`
	`-H 'Content-Type: application/json' \`
	`-d \`
	`'{`
	`"status": "deployed"`
	`}'`

Sample response:	Status: 201 OK
	Location: **https://api.t-nova.eu/v1/orchestrator/ns-instances/456**
	{
	"id":"456",
	"ns-id":"987",
	"status":"deployed",
	"created_at":"2014-11-21T14:18:09Z",
	"updated_at":"2014-11-25T09:46:17Z"
	}

9.4.2.8. Stop a NS Instance

Requests the stopping of an already deployed NS instance.

Method and Endpoint:	PUT /orchestrator/ns-instances/<ns_instance_id>
Parameters:	• status (string). **Required**. Status the instance is required to go to. Are there other status than `deployed`, `undeployed` and `new`? E.g., `running` or `stoped`? (The issue of which states should a NS Instance go through is still opened)

Sample request:	$ curl -X PUT <base-url>+'/orchestrator/ns-instances/456' \
	-H 'Content-Type: application/json' \
	-d \
	'{
	"status": "stopped"
	}'

Sample response:	Status: 201 OK
	Location: **https://api.t-nova.eu/v1/orchestrator/ns-instances/456**
	{
	"id":"456",
	"ns-id":"987",
	"status":"stopped",
	"created_at":"2014-11-21T14:18:09Z",
	"updated_at":"2014-11-25T09:46:17Z"
	}

9.4.3. Orchestrator- VIM Interactions

The exact operations of this interface are still being designed. The requirements these operations will have to support are listed in Table 9-4.

Table 9-4: Requirements for the interface between the Orchestrator and the VIM.

Number	Requirement	Interface
1	Request the VIM to reserve or release the entire required infrastructure needed for a given VNF	Or-Vi
2	Request the VIM to allocate, update or release the required infrastructure needed for a given VNF	Or-Vi
3	Add, update or delete a SW image (usually for a VNF Component)	Or-Vi
4	Collect infrastructure utilization data (network, compute and storage)	Or-Vi

	from the VIM	
5	Request infrastructure's metadata from the VIM	Or-Vi
6	Request the VIM to manage the VMs allocated to a given VNF	Or-Vi
7	The interfaces between the Orchestrator and the VIM SHALL be secure, in order to avoid eavesdropping (and other security threats)	Or-Vi

9.4.4. Orchestrator called by the VNF

VNFs composing a given NS may bring their own VNF Manager. This variability brings new challenges that still have to be understood, and a good solution be designed to address them. We therefore just mention the mapping between the requirements and the interfaces here as a placeholder. One of these interfaces is also drafted.

Table 9-5: Requirements for the Interface between the VNFs and the Orchestrator.

Number	Requirement	Interface
1	All the interfaces between the VNFM and the VNF SHALL be secure, in order to avoid eavesdropping (and other security threats)	Vnfm-Vnf
2	Instantiate a new VNF or terminate one that has already been instantiated	Vnfm-Vnf
3	Retrieve the VNF instance run-time information (including performance metrics)	Create metric readings (Vnfm-Vnf)
4	(Re-)Configure a VNF instance	Vnfm-Vnf
5	Collect/request from the NFS the state/change of a given VNF (e.g. start, stop, etc.)	Vnfm-Vnf
6	Request the appropriate scaling (in/out/up/down) metadata to the VNF	Vnfm-Vnf

9.4.4.1. Create Metric Readings

Adds a new metric reading. VNF provided metrics are defined in the specific VNF Descriptor and created when the VNF instance is created.

Method and Endpoint:	POST /orchestrator/vnf-instances/<vnf_instance_id>/metrics/<metric_id>
Parameters:	• value (string). **Required.** Value of the reading to be created.
Sample request:	`$ curl -X POST <base-url>+'/orchestrator/vnf-instances/123456/metrics/12' \` `-H 'Content-Type: application/json' \`

	`-d \`
	`'{`
	`"value": "156"`
	`}'`

Sample response:	`Status: 201 OK`
	`Location:` **`https://api.t-nova.eu/v1/orchestrator/`**`vnf-instances/123456/metrics/12`
	`{`
	`"id":"987654",`
	`"name":"metric-name",`
	`"value":"156",`
	`"created at":"2014-11-28T10:29:38Z",`
	`"updated at":"2014-11-28T10:29:38Z"`
	`}`

10. ANNEX B

Table 10-1. Nova Compute API Calls regarding Host Aggregates

HTTP verb	Action	Calls
GET	List of host aggregates	/v2/tenant_id}/os-aggregates
	Get details of a specific Host Aggregates	/v2/{tenant_id}/os-aggregates/{aggregate_id}
POST	Create aggregate	/v2/tenant_id}/os-aggregates
	Add host to aggregate	/v2/tenant_id}/os-aggregates/{aggregate_id}/action
	Set aggregate metadata	/v2/tenant_id}/os-aggregates/{aggregate_id}/action

Table 10-2: Nova Compute API Calls regarding virtual resources

Description	Calls (GETs)
List of instances	/v2/{tenant_id}/servers
Detailed list of instances	/v2/{tenant_id}/servers/detail
Details for a specified instance	/v2/{tenant_id}/servers/{server_id}
Usage data for a specified instance	/v2/{tenant_id}/servers/{server_id}/diagnostics
Instance metadata	/v2/{tenant_id}/servers/{server_id}/metadata
Instance Ips	/v2/{tenant_id}/servers/{server_id}/ips
Instance Ips in a specified network	/v2/{tenant_id}/servers/{server_id}/ips/{network_label}
List of Instance types	/v2/{tenant_id}/flavors
Details for a specified flavor	/v2/{tenant_id}/flavors/{flavor_id}
Detailed list of instance types	/v2/{tenant_id}/flavors/detail
Instance type metadata	/v2.1/{tenant_id}/flavors/{flavor_id}/flavor-extra_specs
List of images	/v2/{tenant_id}/images
Detailed list of images	/v2/{tenant_id}/images/detail
Details for a specified image	/v2/{tenant_id}/images/{image_id}
Image metadata	/v2/{tenant_id}/images/{image_id}/metadata
List of volumes	/v1.1/{tenant_id}/os-volumes
Detailed list of volumes	/v1.1/{tenant_id}/os-volumes/detail

Table 10-3: OpenDayLight APIs

Northbound API	Description
Host tracker REST APIs	Tracking host locations in a network, described through a node connector which is a logical entity standing for a switch or a port.
Statistics REST APIs	Returns statistical information exposed by the southbound protocol plugins such as Openflow.
User Manger REST APIs	Provides primitives to manage users.
Connection Manager REST APIs	Manages nodes connected to the controller.
Container Manager REST APIs	Creating, deleting and managing tenants in the network.
Topology REST APIs	Accessing to the topology of the network maintained by the Topology Manager module of OpenDaylight.
Static Routing REST APIs	Managing L3 static routes in the network.
Subnets REST APIs	Managing L3 subnets in a given container.
Switch Manager REST APIs	Providing access to nodes, node connectors and their properties.
Flow Programmer REST APIs	Programming flows in the OpenFlow network.
Bridge Domain REST APIs	Accessing to OVSDB protocol primitives which are used to program Open vSwitch.
Neutron/Network Configuration APIs	Providing integration with OpenStack matching OpenDaylight APIs with Neutron API v2.0

Table 10-4: OpenDaylight API GET Calls

Northbound API	GET Calls	Description
Topology	/controller/nb/v2/topology/{containerName}	Retrieve the Topology
	/controller/nb/v2/topology/{containerName}/ userLinks	Retrieve the user configured links
Host Tracker	/controller/nb/v2/hosttracker/{containerName}/ hosts/active	Returns a list of all Hosts
	/controller/nb/v2/hosttracker/{containerName}/ hosts/inactive	Returns a list of Hosts that are statically configured and are connected to a NodeConnector that is down

	/controller/nb/v2/hosttracker/{containerName}/ address/{networkAddress}	GET a host that matches the IP Address
Flow Programmer	/controller/nb/v2/flowprogrammer/ {containerName}	Returns a list of Flows configured on the given container.
	/controller/nb/v2/flowprogrammer/ {containerName}/node/{nodeType}/{nodeId}	Returns a list of Flows configured on a Node in a given container.
Static Routing	/controller/nb/v2/staticroute/{containerName}/ route/{route}	Get the static route on the container
	/controller/nb/v2/staticroute/{containerName}/ routes	Get a list of static routes on the container
Statistics	/controller/nb/v2/statistics/{containerName}/flow	Get a list of Flow Statistics from all the Nodes.
	/controller/nb/v2/statistics/{containerName}/flow/node/{nodeType}/{nodeId}	Get a Flow statistic of a certain Node
	/controller/nb/v2/statistics/{containerName}/port	Get a list of the statistics of all the NodeConnectors on all the Nodes
	/controller/nb/v2/statistics/{containerName}/port/node/{nodeType}/{nodeId}	Get a list of the statistics of all the NodeConnectors on a given Node
	/controller/nb/v2/statistics/{containerName}/table	Get a list of all the Table statistics on all the Nodes
	/controller/nb/v2/statistics/{containerName}/table/node/{nodeType}/{nodeId}	Get a list of all the Table statistics on a specific Node
Subnets	/controller/nb/v2/subnetservice/{containerName}/subnet/{subnetName}	List the configuration of a subnet on a given container
	/controller/nb/v2/subnetservice/{containerName}/subnets	List all the subnets of the given container
Switch	/controller/nb/v2/switchmanager/{containerName}/node/{nodeType}/{nodeId}	Get a list of all the NodeConnectors and their properties in a given Node
	/controller/nb/v2/switchmanager/{containerName}/nodes	Retrieve a list of all the nodes and their properties in the network
Container	/controller/nb/v2/containermanager /container/{container}/flowspec/{flowspec}	Get flowspec within a given container
	/controller/nb/v2/containermanager	Get all the flowspec on the

	/container/{container}/flowspecs	container
	/controller/nb/v2/containermanager/containers	Get all the containers configured in the system
Neutron Firewall	/controller/nb/v2/neutron/fw/firewalls	Get a list of all Firewalls
	/controller/nb/v2/neutron/fw/firewalls/{firewallUUID}	Get a specific Firewall
	/controller/nb/v2/neutron/fw/firewalls_policies	Get a list of all Firewall Policies
	/controller/nb/v2/neutron/fw/firewalls_policies/{firewallPolicyUUID}	Returns a specific Firewall Policy
	/controller/nb/v2/neutron/fw/firewalls_rules	Returns a list of all Firewall Rules
	/controller/nb/v2/neutron/fw/firewalls_rules/{firewallRuleUUID}	Returns a specific Firewall Rule
Neutron Floating IPs	/controller/nb/v2/neutron/floatingips	Get a list of all floating ips
	/controller/nb/v2/neutron/floatingips/{floatingipUUID}	Get a specific floating IP
Neutron Load Balancer	/controller/nb/v2/neutron/healthmonitors	Returns a list of all LoadBalancerHealthMonitor
	/controller/nb/v2/neutron/healthmonitors/{loadBalancerHealthMonitorID}	Returns a specific LoadBalancerHealthMonitor
	/controller/nb/v2/neutron/listeners	Returns a list of all LoadBalancerListener
	/controller/nb/v2/neutron/listeners/{loadBalancerListenerID}	Returns a specific LoadBalancerListener
	/controller/nb/v2/neutron/loadbalancers	Returns a list of all LoadBlancer
	/controller/nb/v2/neutron/loadbalancers/{loadBalancerID}	Returns a specific LoadBalancer
	/controller/nb/v2/neutron/pools/{loadBalancerPoolUUID}/members	Returns a list of all LoadBalancerPoolMembers in the specified Pool
	/controller/nb/v2/neutron/pools/{loadBalancerPoolUUID}/members/{loadBalancerPoolMemberUUID}	Returns a specific LoadBalancerPoolMember
	/controller/nb/v2/neutron/pools	Returns a list of all LoadBalancerPool
	/controller/nb/v2/neutron/pools/{loadBalancerPoolID}	Return a specific LoadBalancerPool

Neutron Networks	/controller/nb/v2/neutron/networks	Returns a list of all Networks
	/controller/nb/v2/neutron/networks /{netUUID}	Returns a specific Network
Neutron Ports	/controller/nb/v2/neutron/ports	Returns a list of all Ports
	/controller/nb/v2/neutron/ports/{portUUID}	Returns a specific Port
Neutron Routers	/controller/nb/v2/neutron/routers	Returns a list of all Routers
	/controller/nb/v2/neutron/routers/{routerUUID}	Returns a specific Router
Neutron Security Groups	/controller/nb/v2/neutron/security-groups	Returns a list of all Security Groups
	/controller/nb/v2/neutron/security-groups/ {securityGroupUUID}	Returns a specific Security Group
Neutron Security rules	/controller/nb/v2/neutron/security-group-rules	Returns a list of all Security Rules
	/controller/nb/v2/neutron/security-group-rules/ {securityRuleUUID}	Returns a specific Security Rule
Neutron Subnets	/controller/nb/v2/neutron/subnets	Returns a list of all Subnets
	/controller/nb/v2/neutron/subnets/{subnetUUID}	Returns a specific Subnet

11. ANNEX C: ARCHITECTURE-DATA MODEL RELATION

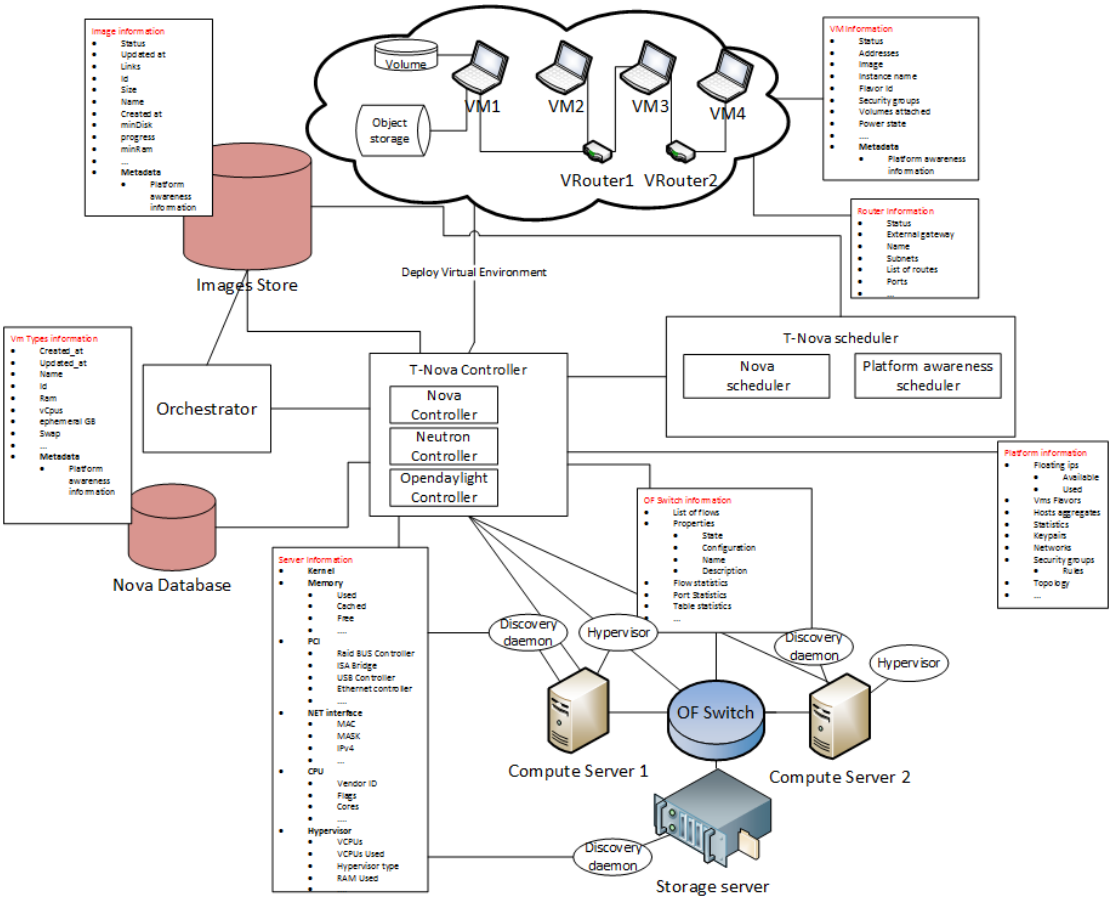

Figure 11-1: Architecture-Data Model relation

12. ANNEX D: EPA JSON OBJECT

EPA JSON Object and available information fields in current implementation


```
{ "CPU" : { "Cache" : "25600 KB",
      "Cores" : 40,
      "Flags" : [ "fpu",
          "vme", "de", "pse", "tsc", "msr", "pae", "mce", "cx8", "apic", "sep",
  "mtrr", "pge", "mca", "cmov", "pat", "pse36", "clflush", "dts", "acpi", "mmx",
  "fxsr", "sse", "sse2", "ss", "ht", "tm", "pbe", "syscall", "nx", "pdpe1gb",
  "rdtscp", "lm", "constant tsc", "arch perfmon", "pebs", "bts", "rep good", "nopl",
  "xtopology", "nonstop tsc", "aperfmperf", "eagerfpu", "pni", "pclmulqdq", "dtes64",
  "monitor", "ds_cpl", "vmx", "smx", "est", "tm2", "ssse3", "cx16", "xtpr", "pdcm",
  "pcid", "dca", "sse4_1", "sse4_2", "x2apic", "popcnt", "tsc_deadline_timer", "aes",
  "xsave", "avx", "f16c", "rdrand", "lahf_lm", "ida", "arat", "epb", "xsaveopt",
  "pln", "pts", "dtherm", "tpr shadow","vnmi", "flexpriority",  "ept", "vpid",
  "fsgsbase", "smep", "erms" ],
      "Freq" : "1254.531",
      "Model" : "Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz",
      "VendorID" : "GenuineIntel"
    },
  "Disk" : { "Blocks" : { "map" : { "hw sector size" : "", "scheduler" : "" }, "sda" :
  { "hw_sector_size" : "512", "scheduler" : "noop deadline [cfq] "}},
      "Partitions" : {
"/dev/mapper/fedora-home" : { "Available" : "374G", "Size" : "394G", "Used" : "75M",
  "Used%" : "1%"},
                      "/dev/mapper/fedora-root" : { "Available" : "374G", "Size" :
  "394G", "Used" : "75M", "Used%" : "1%"},
                      "/dev/sda1" : { "Available" : "374G", "Size" : "394G", "Used" :
  "75M", "Used%" : "1%" }
                          }
    },
  "Issue" : "Fedora release 20 (Heisenbug)\n",
  "Kernel" : "Linux 3.16.6-200.fc20.x86_64",
  "LSmod" : {  "auth_rpcgss" : { "Size" : "58761", "Used by" : [ "1",  [ "nfsd" ] ] },
               "binfmt_misc" : { "Size" : "17431", "Used by" : [ "1",  [ "" ] ] },
               "bridge" : { "Size" : "116006", "Used by" : [ "0",  [ "" ] ] },
               "coretemp" : { "Size" : "13441", "Used by" : [ "0",  [ "" ] ]  },
               "crc32_pclmul" : { "Size" : "13133", "Used by" : [ "0",  [ "" ] ] },
               "crc32c_intel" : { "Size" : "22094", "Used by" : [ "0",  [ "" ] ] },
               "crc_itu_t" : { "Size" : "12613", "Used by" : [ "1", [ "firewire_core"
  ] ] },
               "crct10dif_pclmul" : { "Size" : "14307", "Used by" : [ "0", [ "" ] ] },
               "dca" : { "Size" : "14601", "Used by" : [ "2",  [ "igb", "ioatdma" ] ]
  },
               "drm" : { "Size" : "291361", "Used by" : [ "6", [ "ttm",
  "drm_kms_helper", "nouveau" ] ] },
               "drm_kms_helper" : { "Size" : "58041", "Used by" : [ "1", [ "nouveau" ]
  ] },
               "ebtable_nat" : { "Size" : "12807", "Used by" : [ "0",  [ "" ] ] },
               "ebtables" : { "Size" : "30758", "Used by" : [ "1",  [ "ebtable_nat" ]
  ] },
```

```
                "edac_core" : { "Size" : "56654", "Used by" : [ "1",  [ "sb_edac" ] ]
  },
                "firewire core" : { "Size" : "62559",  "Used by" : [ "1",  [
  "firewire_ohci" ] ] },
                "firewire_ohci" : { "Size" : "40502", "Used by" : [ "0",  [ "" ] ]  },
                "ghash_clmulni_intel" : { "Size" : "13230", "Used by" : [ "0",  [ "" ]
  ] },
                "i2c algo bit" : { "Size" : "13257", "Used by" : [ "2",  [ "igb",
  "nouveau" ] ]  },
                "i2c_core" : { "Size" : "55486", "Used by" : [ "6",  [ "drm", "igb",
  "i2c_i801", "drm_kms_helper", "i2c_algo_bit","nouveau" ] ]},
                "i2c_i801" : { "Size" : "18146", "Used by" : [ "0",  [ "" ] ]  },
                "iTCO_vendor_support" : { "Size" : "13419", "Used by" : [ "1",  [
  "iTCO_wdt" ] ]  },
                "iTCO_wdt" : { "Size" : "13480", "Used by" : [ "0",  [ "" ]  ]  },
                "igb" : { "Size" : "192008", "Used by" : [ "0",  [ "" ] ]  },
                "ioatdma" : { "Size" : "63397", "Used by" : [ "0", [ "" ] ]  },
                "ip6 tables" : { "Size" : "26809", "Used by" : [ "1", [
  "ip6table_filter" ] ]},
                "ip6table filter" : { "Size" : "12815", "Used by" : [ "0",  [ "" ] ]
  },
                "ipmi_msghandler" : { "Size" : "43757", "Used by" : [ "1",  [ "ipmi_si"
  ] ]  },
                "ipmi_si" : { "Size" : "53386", "Used by" : [ "0",  [ "" ] ]  },
                "ipt_MASQUERADE" : { "Size" : "12880", "Used by" : [ "3",  [ "" ] ]  },
                "iptable_mangle" : { "Size" : "12695", "Used by" : [ "1",  [ "" ] ]  },
                "iptable_nat" : { "Size" : "12970", "Used by" : [ "1",  [ "" ] ]  },
                "isci" : { "Size" : "137588", "Used by" : [ "2",  [ "" ] ]  },
                "joydev" : { "Size" : "17344", "Used by" : [ "0",  [ "" ] ] },
                "kvm" : { "Size" : "452677", "Used by" : [ "1",  [ "kvm_intel" ] ]  },
                "kvm_intel" : { "Size" : "147547", "Used by" : [ "0", [ "" ] ] },
                "libsas" : { "Size" : "73498", "Used by" : [ "1",  [ "isci" ] ]  },
                "llc" : { "Size" : "13941", "Used by" : [ "2",  [ "stp", "bridge" ] ]
  },
                "lockd" : { "Size" : "93436", "Used by" : [ "1",  [ "nfsd" ] ]  },
                "lpc_ich" : { "Size" : "21093", "Used by" : [ "0",  [ "" ] ] },
                "mei" : { "Size" : "86597", "Used by" : [ "1",  [ "mei_me" ] ]  },
                "mei_me" : { "Size" : "19568", "Used by" : [ "0",  [ "" ] ]  },
                "mfd_core" : { "Size" : "13182", "Used by" : [ "1", [ "lpc_ich" ] ]  },
                "mic_host" : { "Size" : "53814", "Used by" : [ "0",  [ "" ] ] },
                "microcode" : { "Size" : "44710", "Used by" : [ "0",  [ "" ] ]  },
                "mii" : { "Size" : "13527", "Used by" : [ "1",  [ "r8169" ] ]  },
                "mxm_wmi" : { "Size" : "12865", "Used by" : [ "1",  [ "nouveau" ] ]  },
"nf_conntrack" : { "Size" : "99420", "Used by" : [ "6", [ "ipt_MASQUERADE","nf_nat",
  "nf_nat_ipv4", "xt_conntrack", "iptable_nat", "nf_conntrack_ipv4"] ] },
                "nf conntrack ipv4" : { "Size" : "14656", "Used by" : [ "2",  [ "" ] ]
  },
                "nf_defrag_ipv4" : { "Size" : "12702", "Used by" : [ "1",  [
  "nf_conntrack_ipv4" ] ]  },
                "nf nat" : { "Size" : "25178", "Used by" : [ "3", [ "ipt MASQUERADE",
  "nf_nat_ipv4", "iptable_nat" ] ]  },
```

```
                "nf_nat_ipv4" : { "Size" : "13199", "Used by" : [ "1",  [ "iptable_nat"
  ] ]  },

                "nfs_acl" : { "Size" : "12741", "Used by" : [ "1",  [ "nfsd" ] ]  },

                "nfsd" : { "Size" : "283833", "Used by" : [ "1",  [ "" ] ]  },

                "nouveau" : { "Size" : "1222531", "Used by" : [ "3",  [ "" ] ]  },

                "pps_core" : { "Size" : "19130", "Used by" : [ "1",  [ "ptp" ] ]  },

                "ptp" : { "Size" : "19140", "Used by" : [ "1",  [ "igb" ] ]  },

                "r8169" : { "Size" : "71694", "Used by" : [ "0",  [ "" ] ]  },

                "sb_edac" : { "Size" : "22272", "Used by" : [ "0",  [ "" ]  ]  },

                "scsi_transport_sas" : { "Size" : "39402", "Used by" : [ "2",  [
  "isci",  "libsas" ] ]  },

                "shpchp" : { "Size" : "37047", "Used by" : [ "0",  [ "" ] ]  },

"snd" : { "Size" : "75905", "Used by" : [ "24", [ "snd hda codec realtek",
  "snd hwdep", "snd timer", "snd hda codec hdmi", "snd pcm", "snd seq",
  "snd_hda_codec_generic", "snd_hda_codec", "snd_hda_intel", "snd_seq_device"]] },

"snd_hda_codec" : { "Size" : "131298", "Used by" : [ "5",  [ "snd_hda_codec_realtek",
  "snd_hda_codec_hdmi",   "snd_hda_codec_generic",  "snd_hda_intel",
  "snd_hda_controller"] ]  },

                "snd hda codec generic" : { "Size" : "67662", "Used by" : [ "1",  [
  "snd_hda_codec_realtek" ] ]  },

                "snd_hda_codec_hdmi" : { "Size" : "47489", "Used by" : [ "1",  [ "" ] ]
  },

                "snd_hda_codec_realtek" : { "Size" : "72791", "Used by" : [ "1",  [ ""
  ] ]  },

                "snd hda controller" : { "Size" : "30139", "Used by" : [ "1",  [
  "snd_hda_intel" ] ]  },

                "snd_hda_intel" : { "Size" : "30379", "Used by" : [ "7",  [ "" ] ]  },

                "snd_hwdep" : { "Size" : "17650",  "Used by" : [ "1",  [
  "snd_hda_codec" ] ]  },

 "snd pcm" : { "Size" : "104333", "Used by" : [ "4",  [ "snd hda codec hdmi",
  "snd_hda_codec",  "snd_hda_intel", "snd_hda_controller"] ]  },

                "snd_seq" : { "Size" : "62266", "Used by" : [ "0", [ "" ]]},

                "snd_seq_device" : { "Size" : "14136", "Used by" : [ "1",  [ "snd_seq"
  ] ]  },

                "snd timer" : { "Size" : "28778", "Used by" : [ "2",  [ "snd pcm",
  "snd_seq"] ]  },

                "soundcore" : { "Size" : "14491", "Used by" : [ "2",  [ "snd",
  "snd_hda_codec"] ] },

                "stp" : { "Size" : "12868", "Used by" : [ "1",  [ "bridge" ] ]  },

                "sunrpc" : { "Size" : "279214", "Used by" : [ "5",  [ "nfsd",
  "auth_rpcgss", "lockd", "nfs_acl"] ]  },

                "tpm" : { "Size" : "35153", "Used by" : [ "1",  [ "tpm_tis" ] ]  },

                "tpm_tis" : { "Size" : "18581", "Used by" : [ "0",  [ "" ] ]  },

                "ttm" : { "Size" : "80807", "Used by" : [ "1",  [ "nouveau" ] ]  },

                "video" : { "Size" : "19777", "Used by" : [ "1",  [ "nouveau" ] ]  },

                "vringh" : { "Size" : "20245", "Used by" : [ "1",  [ "mic_host" ] ]  },

                "wmi" : { "Size" : "18820", "Used by" : [ "2",  [ "mxm wmi", "nouveau"]
  ]  },

                "x86_pkg_temp_thermal" : { "Size" : "14205", "Used by" : [ "0",  [ "" ]
  ]  },

                "xt_CHECKSUM" : { "Size" : "12549", "Used by" : [ "1",  [ "" ] ]  },

                "xt_conntrack" : { "Size" : "12760", "Used by" : [ "1",  [ "" ] ]  } },

  "MEM" : { "Buffers" : "45056", "Cached" : "774948", "Free" : "31315304", "Shared" :
  "9996", "Total" : "32842036", "Used" : "1526732" },
```

```
  "NET" : {
"lo:" : { "Encap" : null,  "IPv4" : null,  "IPv6" : null,  "MAC" : null,  "Mask" :
  null },
       "p2p1:" : { "Encap" : null,  "IPv4" : null,  "IPv6" : null,  "MAC" : null,
  "Mask" : null },
       "virbr0:" : { "Encap" : null,  "IPv4" : null,  "IPv6" : null,  "MAC" : null,
  "Mask" : null }
    },
  "PCI" : { "Audio device" : "NVIDIA Corporation GK107 HDMI Audio Controller (rev
  a1)",
      "Co-processor" : "Intel Corporation Xeon Phi coprocessor SE10/7120 series (rev
  20)",
      "Communication controller" : "Intel Corporation C600/X79 series chipset MEI
  Controller #2 (rev 05)",
      "Ethernet controller" : "Intel Corporation I350 Gigabit Network Connection (rev
  01)",
      "FireWire (IEEE 1394)" : "Texas Instruments XIO2213A/B/XIO2221 IEEE-1394b OHCI
  Controller [Cheetah Express] (rev 01)",
      "Host bridge" : "Intel Corporation Xeon E7 v2/Xeon E5 v2/Core i7 DMI2 (rev 04)",
      "ISA bridge" : "Intel Corporation C600/X79 series chipset LPC Controller (rev
  06)",
      "PCI bridge" : "Intel Corporation Xeon E7 v2/Xeon E5 v2/Core i7 PCI Express Root
  Port 3a (rev 04)",
      "PIC" : "Intel Corporation Xeon E7 v2/Xeon E5 v2/Core i7 IOAPIC (rev 04)",
      "Performance counters" : "Intel Corporation Xeon E7 v2/Xeon E5 v2/Core i7 QPI
  Ring Performance Ring Monitoring (rev 04)",
      "SATA controller" : "Intel Corporation C600/X79 series chipset 6-Port SATA AHCI
  Controller (rev 06)",
      "SMBus" : "Intel Corporation C600/X79 series chipset SMBus Controller 0 (rev
  06)",
      "Serial Attached SCSI controller" : "Intel Corporation C602 chipset 4-Port SATA
  Storage Control Unit (rev 06)",
      "System peripheral" : "Intel Corporation Xeon E7 v2/Xeon E5 v2/Core i7 Broadcast
  Registers (rev 04)",
      "USB controller" : "Texas Instruments TUSB73x0 SuperSpeed USB 3.0 xHCI Host
  Controller (rev 02)",
      "VGA compatible controller" : "NVIDIA Corporation GK107 [GeForce GTX 650] (rev
  a1)"
    }
}
```

Table 12-1: Infrastructure information returned for HOST and Hypervisor API Call

HOST API GET	Hypervisor API GET Call
{ "host" : [{ "resource" : { "cpu" : 1, "disk_gb" : 1028, "host" : "5ca60c6792a1442f9471ff575443f94d", "memory_mb" : 8192, "project" : "(total)" }, { "resource" : { "cpu" : 0, "disk_gb" : 0, "host" : "5ca60c6792a1442f9471ff575443f94d", "memory_mb" : 512, "project" : "(used_now)" } }, { "resource" : { "cpu" : 0, "disk_gb" : 0, "host" : "5ca60c6792a1442f9471ff575443f94d", "memory_mb" : 0, "project" : "(used_max)" } }] }	{ "hypervisors" : [{ "cpu_info" : "?", "current_workload" : 0, "disk_available_least" : null, "free_disk_gb" : 1028, "free_ram_mb" : 7680, "hypervisor_hostname" : "fake-mini", "hypervisor_type" : "fake", "hypervisor_version" : 1, "id" : 1, "local_gb" : 1028, "local_gb_used" : 0, "memory_mb" : 8192, "memory_mb_used" : 512, "running_vms" : 0, "service" : { "host" : "1e0d7892083548cfb347e782d3b20342", "id" : 2 }, "vcpus" : 1, "vcpus_used" : 0 }] }

13. ANNEX E: ORCHESTRATOR'S MONITORING COMPONENTS

This appendix contains the figure describing the monitoring components of the orchestrator and how they will interact with the VIM components.

The monitoring data at the orchestrator is received at two different levels: the VNF and the NS levels. For the VNF level, the corresponding VNF Monitoring modules will receive the data from the VNF monitoring agents deployed at the VIM (WP4). All the data received is stored in the VNF repository, and part of the data is forwarded to the service monitoring component. This component, responsible for the service-level monitoring, also receives data from the VIM itself. The component processes the data (if required) in order to build service-level metrics, and send them to the corresponding NS repository.

The monitoring within T-NOVA follows the push model, so all the data is posted from the low-level component towards the upper-level component. The low-level agents (i.e. the VNF Monitoring agents in the VIM) are not allowed to directly access the repositories in the orchestrator, in order to prevent data inconsistencies, bad usages, or garbage-distribution.

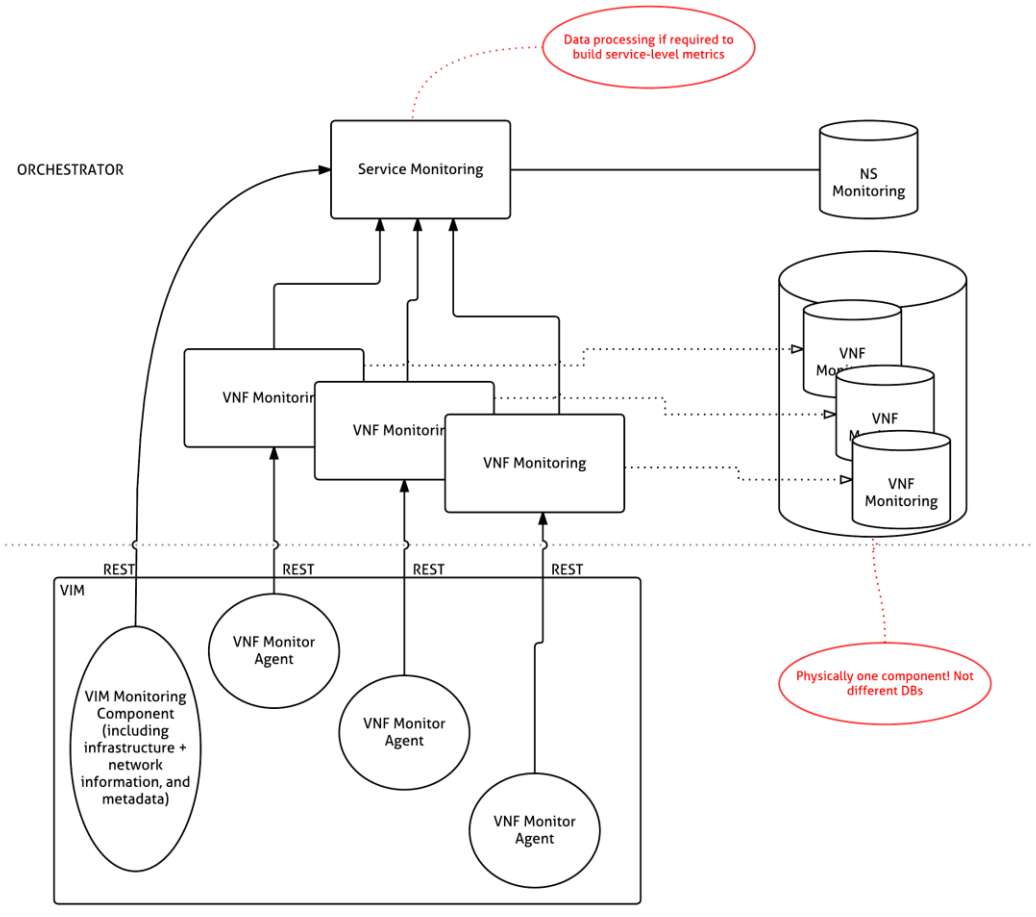

Figure 13-1: Monitoring components within the T-NOVA orchestrator.