5G Programmable Infrastructure Converging disaggregated network and compUte REsources

D5.3 Support for multi-version services

This project has received funding from the European Union's Framework Programme Horizon 2020 for research, technological development and demonstration

5G PPP Research and Validation of critical technologies and systems

Project Start Date: June 1st, 2017

Duration: 33 months

Call: H2020-ICT-2016-2

Date of delivery: 31st May 2019

Topic: ICT-07-2017

Version 1.0

Project co-funded by the European Commission
Under the H2020 programme

Dissemination Level: Public

Grant Agreement Number:	762057
Project Name:	5G Programmable Infrastructure Converging disaggregated network and compUte REsources
Project Acronym:	5G-PICTURE
Document Number:	**D5.3**
Document Title:	Support for multi-version services
Version:	1.0
Delivery Date:	31st May 2019
Responsible:	Paderborn University (**UPB**)
Editor(s):	Azahar Machwe (**ZN**), Sevil Dräxler (**UPB**), Hadi Razzaghi Kouchaksaraei (**UPB**)
Authors:	Sevil Dräxler (**UPB**), Hadi Razzaghi Kouchaksaraei (**UPB**), Kostas Choumas (**UTH**), Osama Arouk (**EUR**).
Keywords:	Network Function Virtualization, Software-Defined Networking, Heterogeneous Infrastructure.
Status:	Final
Dissemination Level	Public
Project URL:	http://www.5g-picture-project.eu/

Revision History

Rev. N	Description	Author	Date
0.0	Table of contents and document structure	UPB	18.04.2019
0.1	First draft for motivation and UPB solution	UPB	23.04.2019
0.2	First draft of scaling, placement and routing optimization	UPB	30.04.2019
0.3	First draft on multi-version and future work	EUR	02.05.2019
0.4	Update UPB solution	UPB	03.05.2019
0.4.1	Section 3.2	UTH	13.05.2019
0.5	Update section 2,5 and 6	UPB	14.05.2019
0.5.1	Update introduction and future work	EUR	15.05.2019
0.6	Update section 2	UPB	15.05.2019
0.7	Updated related work, executive summary, introduction, conclusion, and future work	UPB	16.05.2019
0.7.1	Updated introduction	EUR	17.05.2019
0.7.2	ZN review and addressing ZN's review comments	ZN, UTH, EUR	23.05.2019
0.8	Addressing ZN's review comments and integrating	UPB	24.05.2019
0.9	Addressing ZN's comments Final revision and comments' review	UPB EUR	27.05.2019
1.0	Final revision and submission to the EC	IHP	31.05.2019

Table of Contents

List of Figures

List of Tables

Executive Summary

This document is the deliverable that corresponds to Task 5.3: “Multi-version service chains supporting functional splits in heterogeneous environments” of the 5G-PICTURE project. In this document we show the practicability and the advantages of defining heterogeneous services that consist of multi-version Virtual Network Functions (VNFs). These services are defined in a flexible way with components that can be deployed using different types of resources, offered within the heterogeneous 5G-PICTURE infrastructure. For example, a VNF that can be implemented as a cost-efficient version requiring general-purposes compute resources, as well as a high-performance version that can be deployed using special-purpose hardware.

The descriptors for multi-version services and their VNFs need to include the information about the available deployment options for them. When a multi-version service descriptor is submitted to the 5G Operating System (5G OS) for deployment, the right version of each included VNF has to be selected, for example, based on the preferences of the service provider, the expected load of the service, target performance, final cost of the service, and availability of different types of resources in the infrastructure.

We have implemented a prototype of the 5G OS that can support multi-version services. With this prototype, we prove the feasibility of multi-version service management and orchestration. In this document, we describe the details of the prototype implementation. We also present optimisation and heuristic approaches for the placement, scaling, and routing decisions for multi-version services. The presented algorithms can select the right instance of each service component for deployment in the right location in the underlying infrastructure, using the right amount of resources, creating a trade-off between resource cost and performance of the service. The solutions can easily be modified to consider other objectives.

1 Introduction

To fulfil the required flexibility of 5G networks and services, e.g., on-demand service deployment, heterogeneous resource provisioning, and reducing the time-to-market for novel services, network softwarisation is adopted in 5G networks. Network softwarisation is enabled by Software-Defined Networking (SDN) and Network Function Virtualization (NFV). It is complemented by the attempts towards unifying the tools and mechanisms for controlling and orchestrating distributed infrastructures offering heterogeneous and disaggregated resources. These tools and mechanisms will facilitate the deployment and the management of the lifecycle of cloud-native services and virtual network functions (VNFs) on a variety of multi-technology multi-type compute, storage, and networking resources, in a fast and cost-efficient manner.

In such setup, *flexibly defined services* can be offered and provisioned. In contrast to fully specified services with fixed descriptors, these services consist of components that are packaged in different *versions* (i.e., using different software implementations) in order to respond to the need of running such services in different environments with different types of resources. This will offer many advantages. For example, a deep packet inspection (DPI) function can be deployed as a virtual machine (VM) at a low cost only using general-purpose hardware. Since DPI is a network- and compute-intensive function, its performance can be boosted by using special-purpose hardware support, which of course is a more expensive option. Choosing the right version of the services with the right resources depends of many factors, e.g., (i) Quality of Service/Experience (QoS/QoE), (ii) Service-Level Agreement (SLA), (iii) service-specific requirements (e.g., strict latency that can restrict the types of resources that can be used), (iv) resource availability, e.g., lack of special-purpose hardware at the edge. Therefore, supporting multi-version services can provide a good trade-off between the performance and the cost.

Due to the limited resources, especially in the radio access network (RAN) part of 5G, another dimension of network flexibility needs to be supported, which is the functional split as agreed by 3GPP [1]. The functional split is about where to run the functions (especially the RAN functions), e.g., on the network site close to the antennas, or in micro cloud (many kilometres away from the network sites) or in macro clouds (many tens of kilometres away from the network site). Depending on the dynamics of user traffic, the right functional split should be chosen in order to respect the required QoS for all the ongoing services. Therefore, multi-version and/or multi-functional split need to be supported by the management and orchestration tools of 5G.

To conform to different needs of the users (e.g., low cost vs. high performance), a 5G-PICTURE Tenant (the stakeholders are defined in Deliverable 2.2 [2]) can submit the supported versions of a multi-version service component to the 5G-PICTURE Operator that offers general-purpose as well as special-purpose hardware (like GPUs or FPGAs). Using appropriate service management and orchestration tools and algorithms, the right version of the service can be deployed in the required location(s) to serve the users. We refer to service components with different deployment options as *multi-version* service components.

In a more complex scenario, different versions of such a function may consist of different number of components. As an example, the hardware-accelerated version of a DPI might require an additional VM for post-processing its results, while the VM version of it can perform all of the required operations within the same VM. In this way, as shown in Figure 1 and Figure 2, the services that include these DPI versions, have a different *structure*. We refer to such a service as a *multi-structure* service consisting of multi-version components or a *heterogeneous service*.

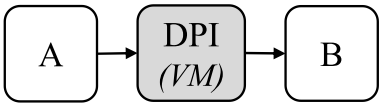

Figure 1: DPI as a VM.

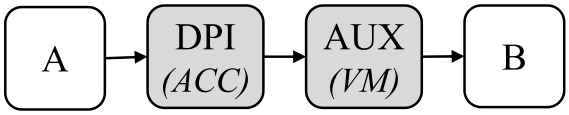

Figure 2: Accelerated DPI with an auxiliary function.

In Deliverable 5.1 [3], we have introduced the 5G Operating System (5G OS), which is designed for control, management, and orchestration of resources on a multi-domain, heterogeneous infrastructure. In this document, which is the deliverable for Task 5.3 of the 5G-PICTURE project, we describe how 5G OS can support the management and orchestration of heterogeneous services with multi-version components. We describe the current state of the prototype implementation of multi-version service orchestration and management and show a mapping of its components to the 5G OS architecture. The prototype implementation will be enhanced and complemented until the end of Task 5.3 in November 2019.

Organisation of the document

In Section 2, we describe the results of our experimental evaluation on multi-version services and dynamic service provisioning. In Section 3, we present the details of our management and orchestration solution for multi-version services, which is a prototype implementation of 5G OS. We provide a formal model for multi-version services in Section 4 and describe the optimization problem of placement, scaling, and path selection for these services based on this model. We provide a mixed integer linear program formulation for finding optimal solutions, as well as fast heuristic that gives sub-optimal results. In Section 5, we give an overview of related work, before concluding the document in Section 6. In the final section, we also describe the remaining steps towards the completion of Task 5.3.

2 Quantitative Analysis of Multi-Version Services

We have conducted an experimental evaluation to analyse multi-version services. In this evaluation, we considered virtualized Transcoder (vTC) as a case study. Transcoder makes videos suitable to be shown in end-users' devices by providing functionalities such as video encoding/decoding and resolution adjustment. We implemented and evaluated two versions of a vTC: (v1) that is implemented to run only on CPU called Commodity off-the-shelf (COTS)-based vTC and (v2) that runs on both CPU and GPU called GPU-assisted vTC. We have evaluated the CPU usage and processing time of these two versions in a range of input bitrates and video resolutions. In our test-bed, as shown in Figure 3, we used four KVM-based VMs: VM #1 hosts a packet generator that breaks down videos to Group of Picture (GOP) and embeds them into UPD packet payloads to be sent to vTCs, VM #2 runs the COTS-based vTC, VM #3 hosts the GPU-assisted vTC, VM #4 receives the transcoded videos from vTCs and display them using a video player. To implement vTC, FFmpeg1 is used which provides a wide range of transcoding functionalities.

The results, illustrated in Figure 4 and Figure 5, show that for videos with low resolutions and low bitrates, the COTS-based vTC is a better choice than GPU-assisted vTC as COTS-based vTC can process the video faster while having low CPU utilization that is an indicator of service cost. On the other hand, as the quality of incoming videos increases by having higher resolutions and bitrates, GPU-assisted vTC performs better in both resource utilization and processing time metrics. These results show a clear trade-off between using GPU-assisted and COTS-based implementations of transcoder as in the case of videos with low quality using GPU-assisted vTC is not only inefficient performance-wise but also costly as GPUs are more expensive than CPUs. These results support the idea of having multi-version services in a system and dynamically provisioning them as such a deployment approach can balance the cost and performance of services.

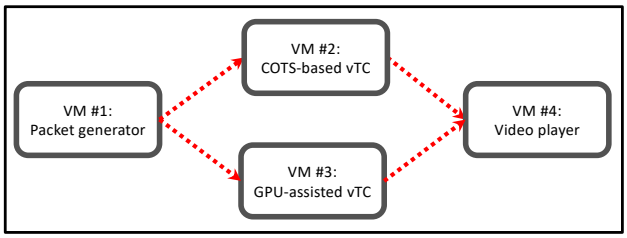

Figure 3: Test-bed set-up designed for the evaluation.

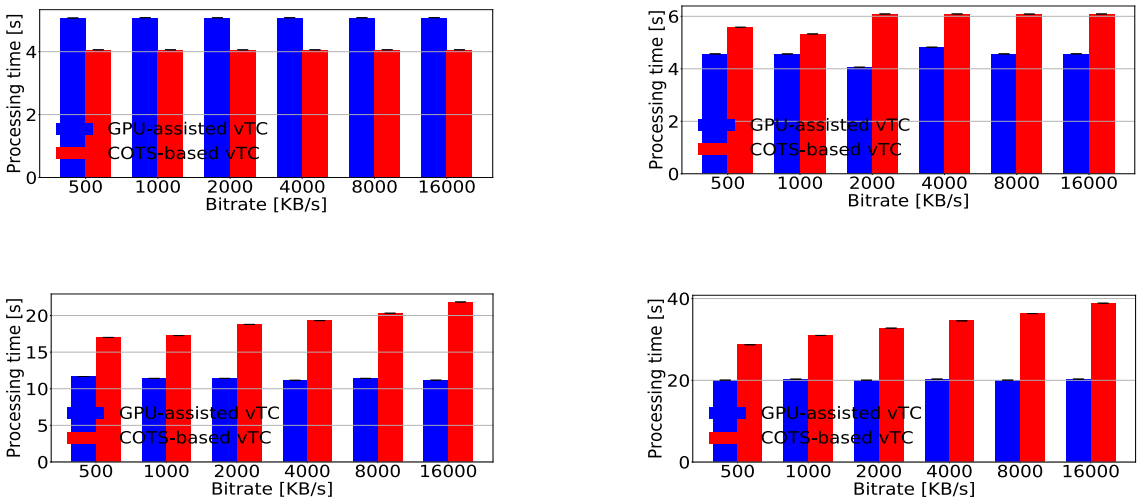

Figure 4: Processing time for videos with different resolutions (top-left 214x120, top-right 426x240, bottom-right 1280x720, bottom-left 1920x1080).

[1] https://ffmpeg.org/

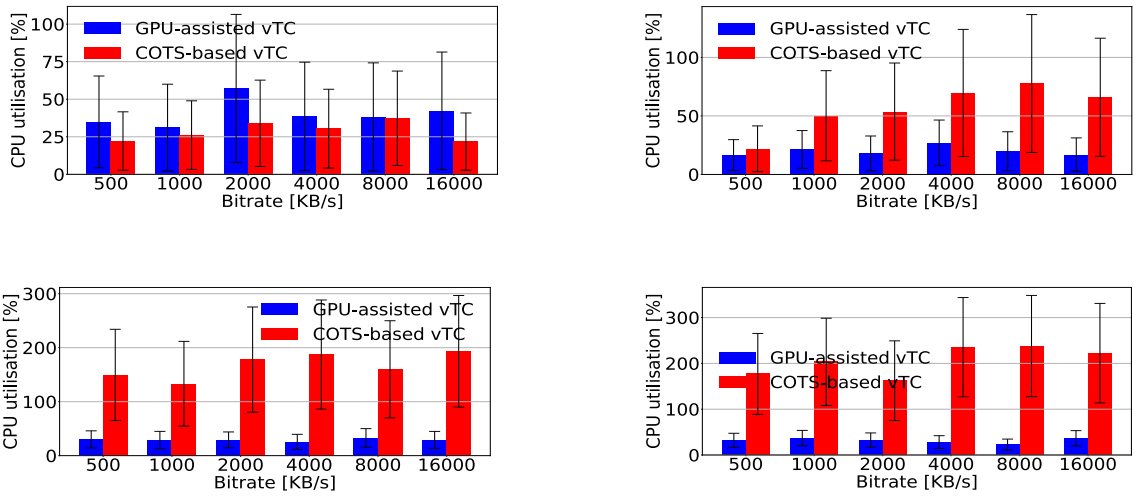

Figure 5: CPU utilisation for videos with different resolutions (top-left 214x120, top-right 426x240, bottom-right 1280x720, bottom-left 1920x1080).

We have also evaluated the deployment time of the implemented vTCs realised on VMs and containers. The service deployment time has been evaluated as it is one of the main factors in management and orchestration of multi-version service, where service deployment versions need to be switched multiple times during the lifecycle of services. For the VM-based vTC, on average, it takes more than 65 seconds to instantiate and spin up a new vTC service including the time required for both Management and Orchestration (MANO) and the Virtualized Infrastructure Manager (VIM) to deploy the service. This time excludes the image downloading time. The same process for the container-based vTC takes, on average, 2.5 seconds. This result shows that containers are a much better solution for dynamically service provisioning with respect to service deployment time as the deployment time of container-based VNFs are significantly shorter than the same for VM-based VNFs.

3 5G OS Orchestration Support for Multi-Version Services

In this section, we describe Pishahang [4], the NFV MANO framework implemented by University of Paderborn (UPB) to support orchestration of multi-version services.

3.1 Pishahang

Pishahang [2] has been implemented upon state-of-the-art NFV, SDN, and Cloud computing tools and technologies. Figure 6 shows the high-level architecture of Pishahang. In this framework, SONATA has been selected as the base MANO framework over its competitors mainly because of two reasons: (i) SONATA follows a microservice-based architecture that allows new functionalities to be added simply by creating and integrating a new microservice (i.e., this makes extending the MANO system much easier) and (ii) it also allows network services to bring their own orchestration code along with other service artefacts by the concept of Service-Specific Management (SSM) [5], which increases the flexibility of the MANO framework to meet the service requirements that are not pre-supported. To support the orchestration of multi-version services, the management of heterogeneous resources should be supported. To this end, in the Pishahang framework, we utilize OpenStack and Kubernetes as VIMs, which enables the management of VM-based VNFs along with container-based VNFs. The main reason to select Kubernetes as a VIM for container-based VNFs is that it provides orchestration not only for CPU-based containers but also for GPU-based containers as well. This, eventually, enables Pishahang to orchestrate VNFs that are realised on heterogeneous resources including VMs, containers, CPUs, and GPUs. The two OpenStack and Kubernetes domains are chained together using the SDN controller that manages the WAN domain.

To have Kubernetes in Pishahang framework, we implemented an adaptor that connects Pishahang's orchestrator to Kubernetes and allows Pishahang to deploy VNFs on Kubernetes domain. Terraform is utilized for implementing this adaptor which is a tool that sits on top of Cloud Management Systems (CMS) and provides multi-cloud functionalities [6]. This tool allows Pishahang's orchestrator to provision resources and manages the lifecycle of VNFs (instantiation, update, termination) on Kubernetes domains.

As Kubernetes is more a Cloud orchestrator and does not support NFV service chaining, we extended Kubernetes with new orchestration services. One of the main challenges here is that the IP address assigned to containers are dynamically assigned and change multiple times during the lifecycle of containers. To tackle this issue, we extended Kubernetes with MetalLB service which allows allocation of fixed IP addresses to containers. These IP addresses can also be external enabling Pishahang's orchestrator to chain VNFs running in Kubernetes domains to VNFs running in other domains.

[2] https://github.com/CN-UPB/Pishahang

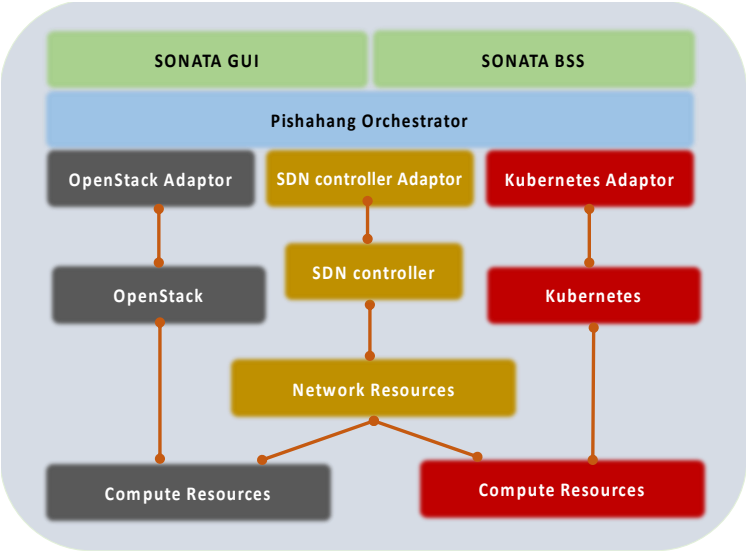

Figure 6: High-level architecture of Pishahang.

Pishahang's orchestrator on top of all domain mangers carries out the intra-domain orchestration of services that are deployed across domains. Using adaptors for OpenStack, Kubernetes and SDN controller, it communicates with underlying domains to deploy VNFs and chain them together. Pishahang's orchestrator manages the lifecycle of multi-version service in three stages: (i) based on the Key Performance Indicator (KPI) accounted for the service and available resources it decides which version should be deployed, (ii) based on the characteristics of the VNF (e.g., VM-based or Container-based), it maps VNFs to resources in corresponding domains (OpenStack or Kubernetes), and (iii) based on the monitoring information it switches to different version if the accounted KPIs are not met.

To avoid reinventing the wheel, we used SONATA's Graphical User Interface (GUI) and Business Support System (BSS) to on-board descriptors and instantiate services, respectively. These components have been slightly extended to make them compatible with Pishahang needs.

3.2 Multi-version service descriptor

Pishahang descriptors that are based on the 5G OS descriptors are designed to describe the management requirements of Multi-version services. In Pishahang, VNFs are described either as a container-based or a VM-based network function. VM-based network function descriptors are designed to describe VNFs that are intended for deployment on OpenStack. On the other hand, container-based network function descriptors are designed considering the requirements of VNFs managed by Kubernetes. We made this distinction in the descriptor as the requirements for VNFs in OpenStack is different from Kubernetes. Examples of VM-based and container-based descriptors are illustrated in Figure 7.

On a higher level, for describing multi-version services as a whole, we designed Multi-version service descriptors. In this descriptor, VNFDs of constituent VNFs in all versions are referenced. To distinguish versions from one another, we use the forwarding graph such that each version is represented by a forwarding graph that describes the constituent VNFs and the order of them in the service chain. An example of multi-version service descriptor is illustrated in Figure 8.

```yaml
descriptor_version: "vnfd-schema-01"
description: vTR
name: vTranscoder
vendor: "pishahang.vnf-descriptor"
version: "1.0"
author: "Hadi Razzaghi"
virtual_deployment_units:
- id: vTR
  description: "VM-based vTranscoder"
  vm_image: "pishahang/vtranscoder.qcow2"
  vm_image_format: qcow2
  resource_requirements:
   cpu:
    vcpus: 2
   memory:
    size: 3
    size_unit: GB
   storage:
    size: 40
    size_unit: GB
  connection_points:
  - id: eth0
    interface: ipv4
    type: external
```

```yaml
descriptor_version: "2.0"
vendor: "pishahang.cloud-service-descriptor"
name: vTR
version: "0.2"
author: "Hadi Razzaghi"
description: "CN-based virtual Transcoder"
virtual_deployment_units:
- id: vTR
  name: vTranscoder
  service_image: "pishahang/transcoder"
  service_type: LoadBalancer
  service_ports:
  - name: http
    protocol: TCP
    port: 8080
    target_port: 8080
  resource_requirements:
   memory:
    size: 128
    size_unit: Mi
  scale_in_out:
   minimum: 2
   maximum: 5
```

Figure 7: Examples of VM-based descriptor (left) and CN-based descriptor (right).

```yaml
descriptor_version: "1.0"
vendor: "pishahang.service-descriptor"
name: "ping-pong"
version: "1.0"
author: "Hadi Razzaghi"
description: "ping-pong service example."
network_functions:
- vnf_id: "vm-ping"
  vnf_vendor: "pishahang.vnf-descriptor"
  vnf_name: "ping-vm-vnf"
  vnf_version: "1.0"
cloud_services:
- service_id: "cn-ping"
  service_vendor: "pishahang.cloud-service-descriptor"
  service_name: "ping-cn-vnf"
  service_version: "1.0"
forwarding_graphs:
- fg_id: "ns:fg01"
  number_of_endpoints: 2
  number_of_virtual_links: 3
  constituent_vnfs:
  - "ping-vm-vnf"
  - "ping-cn-vnf"
  network_forwarding_paths:
  - fp_id: "ns:fg01:fp01"
    policy: none
    connection_points:
    - connection_point_ref: input
      position: 1
    - connection_point_ref: "ping-vm-vnf:eth0"
      position: 2
    - connection_point_ref: "ping-cn-vnf:eth0"
      position: 3
    - connection_point_ref: output
      position: 4
```

Figure 8: Example of Multi-version service descriptor.

3.3 Pishahang in the 5G OS context:

Figure 9 shows the mapping of Pishahang components to 5G OS components. As illustrated in the figure, the GUI and the BSS provide the service management functionalities of 5G OS. Pishahang orchestrator works as the multi-domain orchestrate and provide orchestration across VM, container, and networking domains. The adaptors carry out domain orchestrator functionalities and manage deployment of service on particular domain. On the lower layer, SDN controller will work as 5G OS domain controller and finally Kubernetes and OpenStack play the role of NFV orchestrator in the 5G OS context.

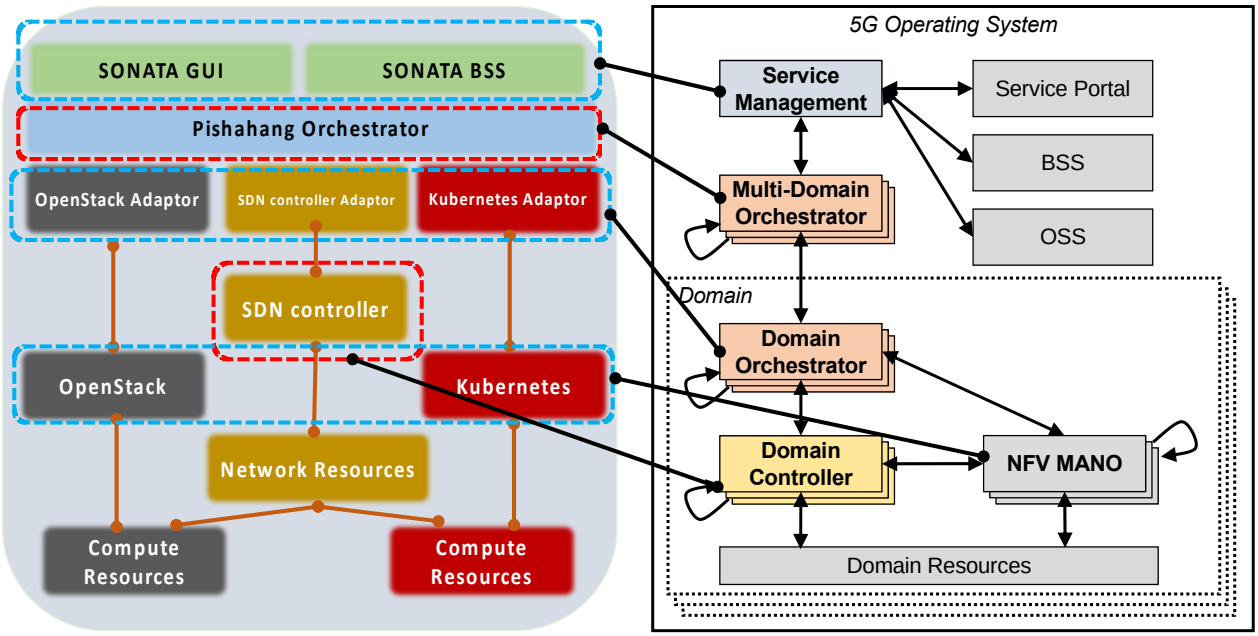

Figure 9: Mapping Pishahang components to 5G OS components.

3.4 Cross-cutting concerns

Beside the described functionalities, there are a number of cross-cutting concerns, also described in Deliverable 5.1 [3], which are commonly required for all concrete implementations of the 5G OS architectural framework. In this section, we briefly describe the requirements for the management and orchestration of multi-version services and how these requirements can be fulfilled.

3.4.1 State management

For switching among different versions of a complex service, state management approaches are required for transferring state among the versions in a consistent and timely manner. There are several state management frameworks that target state transfers during scaling operations of network functions. Our goal is to enable the management of generic functions, selecting the right version at the right time by collecting the appropriate information. When packet processing is being collectively handled by multiple instances of a VNF, the VNF deployment as a whole must typically meet three important goals: (1) satisfy tight VNF SLAs on performance or availability, (2) accurately monitor and manipulate network traffic and (3) operate with minimal cost. One option for 5G-PICTURE was to extend and exploit existing solution, for example the control plane of OpenNF [7], meeting these three goals. OpenNF is a control plane that enables loss-free and order-preserving flow state migration across multiple versions of a VNF, in scenarios where flow packets are distributed across a collection of VNF versions for processing. Our extensions enable the control plane to be more scalable and distributed, relying on a hierarchy of controllers and complementing the SDN control hierarchy within 5G OS.

One of the challenges of an efficient state migration system that we have to deal with, is to reduce the state migration overhead. 5G-PICTURE takes advantage of the fact that most flows are short-lived mouse flows (flows with size less than a given threshold), and in many cases their processing states will expire before the state migration finishes. Rather than blindly moving states of all the flows, 5G-PICTURE uses an approach that keeps the states of active mouse flows on the original VNF instance, and only migrates elephant flow states. It detects the elephant flows by monitoring all flows and keeping in a table a map between the flow ID and the corresponding flow size and the timestamp of the last packet of this flow. Every 100 ms, it checks the activeness of the flows and which of them have a size that is above a certain threshold, equal to 3MB, and then these flows are characterized as elephant ones. By limiting the flow states to be migrated only to the elephant flows, the used approach greatly reduces the migration delay and its performance penalty. This approach succeeds to reduce the average migration time by up to 87% and the latency to mice flows by up to 94% compared to OpenNF, as it is presented in [8].

Another major concern about NFV is the capability for recovery of stateful VNFs, which seems to be a complex process, concerning the difficulties of handling non-determinism such as multi-threaded processing, time dependence, and randomness. In 5G-PICTURE, a novel approach for VNF recovery with very low overhead in failure-free time is considered, concerning non-determinism support, which is presented in [9]. This is in contrast to previous suggestions to take snapshots of the VNF state at certain checkpoints or to store the VNF state externally. According to the used approach, there are two instances for each VNF, one in the master mode and one in the slave mode. The master instance provides input to a Determinant and Progress Logger for all updates happened to its state, and the slave instance gets informed by the logger for these updates. In this way, whenever master instance fails, the slave instance is ready to replace it, without suffering from the time-consuming process of reloading the last saved instance.

3.4.2 Monitoring

An efficient monitoring approach has been developed within the 5G-PPP project SONATA [10] that incorporates a monitoring framework for providing status and usage data from infrastructure nodes, VMs, based on the open-source Prometheus monitoring framework. In 5G-PICTURE, we will extend this monitoring mechanisms by adding support for container-based VNFs. A prototype implementation of this mechanism will be integrated into the Pishahang multi-version solution by the end of the project.

4 Scaling, Placement, and Routing Optimization for Multi-Version Services

In this section, we tackle the problem of **joint scaling, placement, and routing** for heterogeneous services. We describe the services using abstract *templates* that include the required components for creating the service as well the desired interconnections among them. The exact number of instances for each component is not a part of the service template; it is determined upon embedding the template into the network, depending on the source and data rate of the service flows. The required *time in system* for requests and the resource demands of each component (for each supported deployment version of it) is defined as *a function of the input data rate* in the template. The actual time in system for requests and resource demands of the instances are then determined while embedding the template into the network. The information required for the templates (described in the rest of this section) can be extracted from network service and VNF descriptors. The functions describing the resource demands based on load can be generated based on profiling results for different VNFs.

More details on the models and approaches presented in this section are available in [11].

4.1 Model and Problem Formulation

In this section, we describe our model and assumptions and, based on them, formalize the problem we are tackling.

4.1.1 Substrate network

The substrate network is a connected, directed graph, $G_{\text{sub}} = (V, L)$. Each network *node* $v \in V$ has a limited capacity of general-purpose processing resources $\text{cap}_{\text{cpu}}(v) \geq 0$ and special-purpose processing (e.g., GPU) resources $\text{cap}_{\text{gpu}}(v) \geq 0$. This can be extended to other types of resources, e.g., memory, FPGAs, etc. These resources are available at a predefined cost on each node. We denote the cost of using a unit of CPU and GPU resources for a single time unit on node v by $\text{cost}_{\text{cpu}}(v)$ and $\text{cost}_{\text{gpu}}(v)$, respectively. If a certain resource type is not available on a node, we assume the cost of using is infinitely large. Each network *link* $l \in L$ supports a maximum data rate of $\text{cap}(l)$ and has a given delay of $d(l)$.

4.1.2 Templates

Service deployment requests are given as templates that describe the general structure of a service. Each service template is a connected, directed, acyclic graph $G_T = (C_T, A_T)$, e.g., as shown in Figure 10. Each *component* $c \in C_T$ in the template represents a VNF, cloud service component, etc. We describe the details of components in Section 4.1.3.

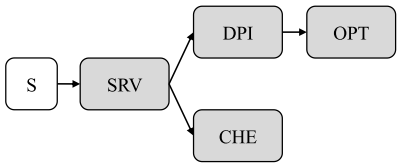

Figure 10: Example service template including a source, a server (SRV), a deep packet inspector (DPI), a video optimizer (OPT), and a cache (CHE).

Each *arc* $a \in A_T$ of the template represents the connectivity among two components. It has a maximum tolerable delay $d_a^{\max}$, which specifies the upper bound for the total delay of the path in the substrate network to which a is mapped. An arc a may be described using additional details regarding the connectivity type and the underlying networking technology. They impose additional constraints to the links that can be used for realizing the connection between the two endpoints of it, i.e., the components src_a, dst_a. Adding these details to the model is straightforward but for simplicity, we do not consider these aspects in this model.

4.1.3 Components and Deployment Versions

Each component c has a given number of inputs n_c^{in} and outputs n_c^{out}, representing the number of ingoing/outgoing connection points. The outgoing data rate of a component depends on the data rate on all its inputs. This is calculated using a given function $\text{fout}_c(\Lambda)$. Λ is a vector of length n_c^{in}. The value of $\text{fout}_c(\Lambda)$ is a vector of length n_c^{out}. This function can be obtained, e.g., by referring to historical usage data or by testing and profiling the component.

Each component may optionally be deployable using different resource types, e.g., as a VM version that can only use CPUs, or an accelerated version that needs special-purpose processing resources (in our model, GPU) in addition to CPUs. Each such *deployment version* of a component requires a specific software version, which is available by the tenant. Each component description must include at least one deployment version. We refer to components with more than one deployment version as *multi-version components*.

Each deployment version may consume different types and different number of resources and can result in a different time in system for requests and different costs. We assume a resource cost model based on the usage duration of a unit of a resource in the substrate network. We describe the specification of required resource units in the rest of this section. Without loss of generality, we assume three possible deployment versions for a component, expressed as the set VER including: a virtual machine version (VM) or a container version (CON) that only need CPUs as processing units, and an accelerated version (ACC) that needs CPUs and GPUs.

Each deployment version $\mathrm{ver} \in \mathrm{VER}$ of component c includes a function $\mathrm{fpt}_c(\mathrm{ver}, \lambda)$ that shows the *maximum* time in system for a total input data rate of λ. This description is only meaningful if accompanied by the specification of the attributes of the processing unit that has been used to profile the component. Different deployment versions can be defined for different processing unit architectures, resulting in different time in system for a given load. Figure 11 shows the DPI component from the example template of Figure 10, defined with two different deployment versions: a VM version and an accelerated version. For this DPI, if the input data rate is λ_1, the outgoing data rate from its only output is expected to be at most $0.9 \cdot \lambda_1$, for example because the tenant expects 10% of video streaming requests to be unauthorized. The VM version results in a maximum of 2.5 times more time in system than the accelerated version.

CPU and GPU demand of components rarely have a linear relationship with the input load. To overcome computational difficulties in the problem formulation (e.g., in the MIP described in Section 4), we assume the CPU and GPU demands are given as piecewise linear functions that approximate non-linear dependencies of the resource demands on the load. Figure 12 shows how such a piecewise linear function could represent the CPU demands of an example component.

In case a resource type like GPU cannot be shared among different processes, piecewise constant functions can be used, representing the step-wise increase of the number of required resource units by increasing load, e.g., as shown in Figure 13. The piecewise linear functions, $\mathrm{fcpu}_c(\mathrm{ver}, \Lambda)$, $\mathrm{fgpu}_c(\mathrm{ver}, \Lambda)$ specify the CPU and GPU demands of the deployment version ver of component c. The demand is calculated based on the vector Λ of ingoing data rates on inputs of the component.

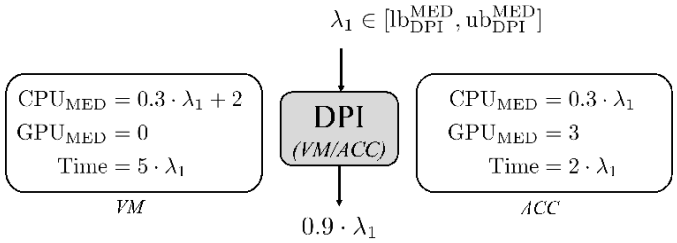

Figure 11: Example DPI as a VM and as an accelerated version (ACC).

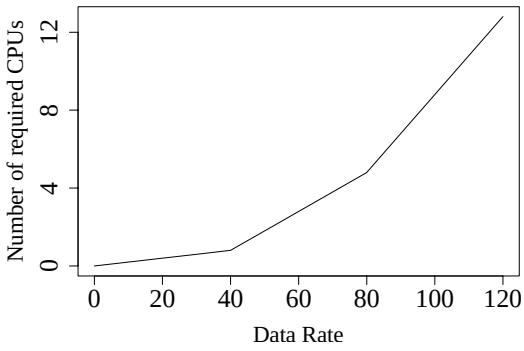

Figure 12: Example CPU demand based on incoming data rate.

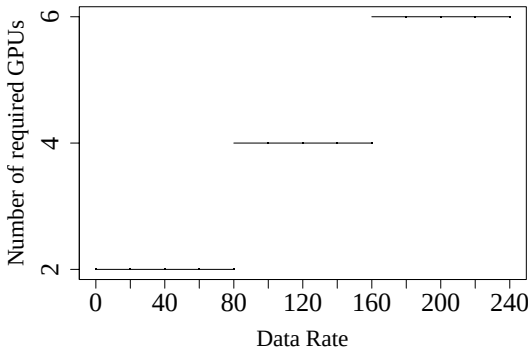

Figure 13: Example GPU demand based on incoming data rate.

The condition for selecting the right linear function to calculate the resource demand is defined based on the total load that should be handled, i.e., the sum of data rates on all inputs of the component. For example, the function shown in Figure 13 can be expressed as in Equation 1, where $\lambda = \sum_{i \in \text{in}_c} (\Lambda)_i$ is the sum of data rates on all inputs of the accelerated deployment version of component c and $(\Lambda)_i$ is the i-th element of vector Λ.

$$\mathbf{fgpu}_c(\mathbf{ACC}, \boldsymbol{\Lambda}) = \begin{cases} \mathbf{2} & \text{if } \boldsymbol{\lambda} \in (\mathbf{0}, \mathbf{80}] \\ \mathbf{4} & \text{if } \boldsymbol{\lambda} \in (\mathbf{80}, \mathbf{160}] \\ \mathbf{6} & \text{if } \boldsymbol{\lambda} \in (\mathbf{160}, \mathbf{240}] \\ \infty & \text{else} \end{cases} \quad (\mathbf{1})$$

For simplifying our notations, we assume the resource demands of all components are defined for the same *number* of load levels, i.e., the same number of non-overlapping intervals defining the domain of the function. We express this with a set LEV, consisting of four possible load levels, i.e., low (LOW), medium (MED), and high (HIG) that can be handled by a specific deployment version of a component and infinite load (INF) above those. Infinite load is any amount of total data rate, which cannot be handled efficiently by the corresponding deployment version using any reasonable (as defined by the tenant who owns the service) amount of resources in a reasonable amount of time.

We use the notations $\mathrm{lb}_c^{\mathrm{lev}}(\mathrm{ver})$, $\mathrm{ub}_c^{\mathrm{lev}}(\mathrm{ver})$ to show the lower and upper bounds of a load level lev for the deployment version ver of a component c. For example, considering an accelerated version of c, if the sum of input data rates to c is between $\mathrm{lb}_c^{\mathrm{MED}}(\mathrm{ACC})$ and $\mathrm{ub}_c^{\mathrm{MED}}(\mathrm{ACC})$, its CPU and GPU demands are calculated by the linear functions $\mathrm{fcpu}_c^{\mathrm{lev}}(\mathrm{ACC}, \Lambda)$, $\mathrm{fgpu}_c^{\mathrm{lev}}(\mathrm{ACC}, \Lambda)$.

The functions are given as a part of the template and can be obtained, e.g., using profiling methods. Figure 11 shows example functions for CPU and GPU demands of the VM and ACC versions of the DPI function, at medium load level.

Source components are special components that represent the starting point of the flows in the service, e.g., end users or content distribution servers. Source components have zero resource demands, zero delay, no input, and exactly one outgoing connection to another component. S is the source component of the template in Figure 10. Each template has exactly one source component. Several instances of each source component can be mapped to different nodes of the network where flows are initiated, e.g., to model different locations where users of a service are located.

Fixed components are also special components that are pinned to a certain location in the substrate network, e.g., modelling the endpoints (sinks) of service flows or legacy, physical network functions. They cannot be relocated or scaled. We also assume their resource demands are zero, as they are pre-defined and fixed.

4.1.4 Multi-Structure Templates

By specifying multiple deployment versions for components of a template, we can deploy services in which the best version of each component can be selected considering the trade-offs between the time in system for requests and the deployment cost.

In a more complex scenario, a certain functionality might even be realized in different variations, each including more than one component. For example, the video optimizer function in Figure 14 can either be deployed as one single container or as a chain of one accelerated component and one VM that need to work together. Selecting one version or another of such a function not only affects the resource demands of the function and the time in system for requests but may also influence the structure of the whole service graph. We refer to such templates as *multi-structure templates*.

To support this, we introduce special *ingress* components in our model. These components have at least two outputs, among which only one output can be active. Any component of the template (except the source component) can be marked as an ingress component. For this, the component must include such a load balancing and classification functionality. Otherwise, additional placeholder components with zero resource demands (and therefore, zero cost) can be defined and added to the template. The placeholder ingress components will not be a part of the actual deployment of the service.

Multi-structure templates may include multi-version components, modelling heterogeneous services. For example, the video streaming template in Figure 14 includes a multi-version DPI and can take multiple structures depending on which version of the video optimizer is selected.

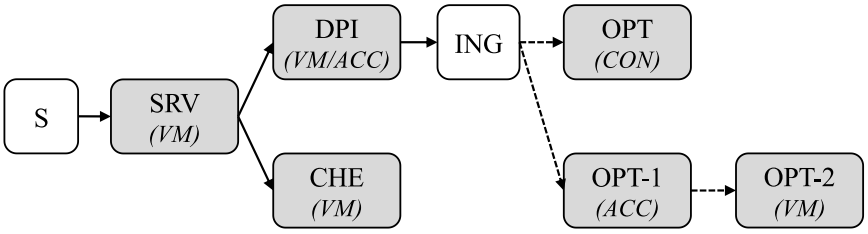

Figure 14: Example multi-structure video streaming service template including multi-version components.

4.1.5 Template Embedding

A template specifies that the corresponding service requires the included components with the specified interconnections for functioning correctly. The *template embedding* process involves deciding:

- how many instances (*horizontal scaling*),
- of which components (*structure selection*)
- using which deployment version of each component (*version selection*),
- with how many resources (*vertical scaling*),
- need to be instantiated in which locations (*placement*), and
- how the traffic should be routed among them (*routing*).

The outcome is an overlay, described in the next section.

This process can be used for the initial embedding of templates as well as for updating existing embeddings.

For template embedding, a number of inputs are required. In addition to the templates to be embedded (including the description of their components and arcs), each template T must be accompanied by a set S_T of at least one *source instance*. Source instances are given as tuples $(c, v, \lambda) \in S_T$. $c \in C_T$ refers to the source component of template T and is used to differentiate between source instances of different templates. $v \in V$ specifies the location in the substrate network where the flow initiates. λ shows the data rate of the flow.

If a template T includes fixed components, they should also be given as a part of the input. Fixed components are described as a set X_T of tuples (c, v). $c \in C_T$ shows the fixed component and $v \in V$ is the network node where it is located. Fixed components influence the embedding process of the template, e.g., because the template contains non-fixed components that can be placed at locations with a given maximum delay to the fixed component.

Another optional input is the set of previously existing instances of a template's components. This input is required if the template embedding is used for optimizing and updating an existing embedding. If a template is being embedded for the first time for a tenant, no previous embedding is required. Previous embeddings of the components of template T are given as a set P_T of tuples (c, v, ver). Such a tuple specifies that an instance of component $c \in C_T$ exists on node $v \in V$ with the deployment version ver.

4.1.6 Overlay

The template embedding process maps the abstract description of the service (template), to a concrete deployable graph, i.e., the *overlay*, embedded into the substrate network. Each overlay is a connected, directed graph, $G_{\mathrm{OL}}(T) = (I_{\mathrm{OL}}, E_{\mathrm{OL}})$. It consists of *instances* and *edges*. Each overlay has exactly one template. One template can be embedded several times, e.g., each with different identifiers, belonging to different tenants.

For each instance $i \in I_{\mathrm{OL}}$ in the overlay of template, T, there exists a component $c \in C_T$ that contains its specification. Instances are mapped to network nodes and have resources allocated to them. For each component, there can be multiple instances. If there are several deployment versions of a component, each instance of it can have a different version, if required. For simplicity, we assume only one instance of each component can be mapped to one network node, which also means, two different deployment versions of one component cannot be mapped to one network node. However, if one template is embedded multiple times, e.g., each for a different tenant, there are no limitations for embedding multiple instances of the same component from different embeddings, as they are considered different instances in our model.

Similarly, for each edge $e \in E_{\mathrm{OL}}$ in the overlay of template T, there exists an arc $a \in A_T$ that specifies its endpoints and its maximum allowed latency. Each edge e mapped to a path in the substrate network. This path is a set of network links that starts at the network node on which src_a is mapped and connects it to the node on which dst_a is mapped. Paths cannot include cycles.

4.1.7 Problem Formulation Summary

As defined in Section 4.1.5, our problem involves version and structure selection, placement, scaling, and routing decisions. Our aim is to take these decisions as a single-step template embedding process. The required inputs are:

- Substrate network.
- A template for each service.
- Location and data rate of source instances.
- Location and deployed version of previously embedded components (optional).
- Location of fixed components (optional).

The output is an overlay mapped to the substrate network, possibly modifying an existing embedding. While embedding the templates, node and link capacities cannot be exceeded. The delay bounds defined between every two component cannot be exceeded either. Our desired solution to this problem *minimizes* the values of the following metrics of interest: deployment cost, time in system, used link capacity, number of added/removed/modified instances.

4.2 Mixed Integer Program Formulation

In this section, we formalize the scaling, placement, and routing problem for heterogeneous services as a mixed-integer program (MIP). All constraints and objective functions in this formulation are linear or can be linearized. As the link and node resource demands are also specified using piecewise linear functions, we are dealing with a mixed-integer linear program.

Table 1 and Table 2 show an overview of the binary and continuous decision variables in the MIP, respectively. We define a pre-set variable $m^*_{c,v,\mathrm{ver}}$ to capture previous embeddings of components based on the input parameters; for every component c that was previously embedded into node v with version ver, represented by a tuple $(c, v, \mathrm{ver}) \in P_T$, we set $m^*_{c,v,\mathrm{ver}}$ to 1. For all other components, nodes, and versions, we set it to 0. $\mathcal{M}$ represents a constant that is sufficiently large, used in the so-called Big-M formulations. $\mathcal{C}_{\mathrm{SRC}}, \mathcal{C}_{\mathrm{FIX}}, \mathcal{C}_{\mathrm{ING}} \subset \mathcal{C}$ represent the source, fixed, and ingress components, respectively. We indicate all the normal components that are not source or fixed components with $\mathcal{C}_N$.

Table 1: Binary Decision Variables.

Variable	Definition
$x_{c,v}$	1 iff an instance of c is mapped to v.
$m_{c,v,\mathrm{ver}}$	1 iff an instance of c is mapped to v with version ver.
$\delta_{c,v}$	1 iff $m_{c,v,\mathrm{ver}} \neq m^*_{c,v,\mathrm{ver}}$, i.e., an instance of c is added, removed, or switched to another version at v.
$l^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}}$, $l^{\mathrm{gpu}}_{c,v,\mathrm{lev}}$	Helper variable for calculating the CPU, GPU demand, which indicates whether the sum of data rates on the inputs of c on v is larger than or equal to the lower bound that defines the load level lev for version ver.
$u^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}}$, $u^{\mathrm{gpu}}_{c,v,\mathrm{lev}}$	Helper variable for calculating the CPU, GPU demand, which indicates whether the sum of data rates on the inputs of c on v is smaller than or equal to the lower bound that defines the load level lev for version ver.
$b^{\mathrm{cpu}}_{c,v,\mathrm{ver},\mathrm{lev}}$, $b^{\mathrm{gpu}}_{c,v,\mathrm{lev}}$	Helper variable for calculating the CPU, GPU demand, which indicates whether the sum of data rates on the inputs of c on v is in the range that defines the load level lev for version ver.
$r_{c,v,k}$	1 iff output k of the instance of ingress component c at v is activated.
$u_{a,v,v',l}$	1 iff the link is used for the path created for arc a with source and destination on v and v'.

Table 2: Continuous Decision Variables:

Variable	Definition
$\mathrm{cpu}_{c,v}$, $\mathrm{gpu}_{c,v}$	CPU, GPU demand of the instance of c if mapped to v.
$\mathrm{time}_{c,v}$	Time in system of requests (including the waiting time and the processing time) at the instance of c if mapped to v.
$p_{c,v,\mathrm{ver}}$	Potential CPU demand of c at v for version ver.
$g_{c,v}$	Potential GPU demand of c at v, defined for its GPU-accelerated instances.
$t_{c,v,\mathrm{ver}}$	Potential time in system of requests at c at v using version ver.
$s_{c,v}$	Total CPU and GPU resource cost of c on v per time unit.
$\mathrm{in}_{c,v}$	Vector of length n_c^{in} of data rates at inputs of the instance of c at v, or an all-zero vector
$\mathrm{out}_{c,v}$	Vector of length n_c^{out} of data rates at outputs of the instance of c at v, or an all-zero vector
$o_{c,v}$	Vector of length n_c^{out} of potential data rates at outputs of the instance of ingress component c at v
$\mathrm{dr}^{e}_{a,v,v'}$	Data rate of the edge corresponding to an arc a that connects an instance of c at v to an instance of c' at v'.
$\mathrm{dr}^{l}_{a,v,v',l}$	Data rate on link l corresponding to an arc a that connects an instance of c at v to an instance of c' at v'.

4.2.1 Constraints

We assign fixed components and sources to their pre-defined locations (Constr. 2, 3). The data rate of each source is assigned to its output (Constr. 4).

$$\forall c \in \mathcal{C}_{\mathrm{SRC}}, \forall v \in V: x_{c,v} = \begin{cases} 1 & \exists (v,c,\mu) \in \mathcal{S} \\ 0 & \text{else} \end{cases} \quad (2)$$

$$\forall v \in V, \forall c \in \mathcal{C}_{\mathrm{FIX}}: \quad x_{c,v} = \begin{cases} 1 & \text{if } \exists (c,v) \in \mathcal{X} \\ 0 & \text{else} \end{cases} \quad (3)$$

$$\forall c \in \mathcal{C}_{\mathrm{SRC}}, \forall v \in V: \mathrm{out}_{c,v} = \begin{cases} \mu & \exists (v,c,\mu) \in \mathcal{S} \\ 0 & \text{else} \end{cases} \quad (4)$$

We track the added/removed/modified instances (Constr. 5). If an instance is created, the right number of inputs (Constr. 6) and outputs (Constr. 7) are created for it (we represent the k-th element of a vector w by $(w)_k$). At most one instance of each component can be mapped to a node (Constr. 8, 9).

$$\forall c \in \mathcal{C}_N, \forall v \in V: \delta_{c,v} = \begin{cases} m_{c,v,\mathrm{ver}} & \text{if } m^*_{c,v,\mathrm{ver}} = 0 \\ 1 - m_{c,v,\mathrm{ver}} & \text{if } m^*_{c,v,\mathrm{ver}} = 1 \end{cases} \quad (5)$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\mathrm{in}}]: \quad (\mathrm{in}_{c,v})_k \leq \mathcal{M} \cdot x_{c,v} \quad (6)$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\mathrm{out}}]: \quad (\mathrm{out}_{c,v})_k \leq \mathcal{M} \cdot x_{c,v} \quad (7)$$

$$\forall c \in \mathcal{C}_N, \forall v \in V: \sum_{\mathrm{ver} \in \mathrm{VER}} m_{c,v,\mathrm{ver}} \leq 1 \quad (8)$$

$$\forall c \in \mathcal{C}_N, \forall v \in V: 0 \leq |\mathrm{VER}| \cdot x_{c,v} - \sum_{\mathrm{ver} \in \mathrm{VER}} m_{c,v,\mathrm{ver}} \leq |\mathrm{VER}| - 1 \quad (9)$$

The data rate entering an instance determines its outgoing data rate (Constr. 10). The data rates are set only if the instance is mapped to a certain node. We assume all deployment versions for an instance have the same function for calculating the outgoing data rate. For ingress components, only one of the outputs can have a data rate (Constr. 11-13).

$$\forall c \in \mathcal{C}_N \backslash \mathcal{C}_{\text{ING}}, \forall v \in V:$$
$$\text{out}_{c,v} = \text{fout}_c(\text{in}_{c,v}) - (1 - x_{c,v}) \cdot \text{fout}_c(\underline{0}) \quad (10)$$
$$\forall c \in \mathcal{C}_{\text{ING}}, \forall v \in V:$$
$$o_{c,v} = \text{fout}_c(\text{in}_{c,v}) - (1 - x_{c,v}) \cdot \text{fout}_c(\underline{0}) \quad (11)$$
$$\text{out}_{c,v} = r_{c,v,k} \cdot o_{c,v} \quad (12)$$
$$\sum_{k \in [1, n_c^{\text{out}}]} r_{c,v,k} = 1 \quad (13)$$

We assign data rate to each input of the instances on each node as the sum of data rates of overlay edges that end in that input (Constr. 14). Similarly, we assign data rate to the outputs of the instances (Constr. 15).

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\text{in}}]:$$
$$(\text{in}_{c,v})_k = \sum_{\substack{a \in \mathcal{A} \text{ ends in input } k \text{ of } \text{src}_a(a), \\ v' \in V}} \text{dr}^e_{a,v',v} \quad (14)$$
$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\text{out}}]:$$
$$(\text{out}_{c,v})_k = \sum_{\substack{a \in \mathcal{A} \text{ starts at output } k \text{ of } \text{src}_a(a), \\ v' \in V}} \text{dr}^e_{a,v,v'} \quad (15)$$

The data rates of the edges are mapped to network links, ensuring flow conservation over the path(s) (Constr. 16). This ensures that the flows entering intermediate components in a service (not a source or sink) are forwarded correctly to the next component. The total delay of the used network links cannot exceed the maximum delay (Const. 17-19).

$$\forall a \in \mathcal{A}, \forall v, v_1, v_2 \in V:$$
$$\sum_{vv' \in L} \text{dr}^l_{a,v_1,v_2,vv'} - \sum_{v'v \in L} \text{dr}^l_{a,v_1,v_2,v'v} = \begin{cases} 0 & \text{if } v \neq v_1, v \neq v_2 \\ 0 & \text{if } v = v_1 = v_2 \\ \text{dr}^e_{a,v_1,v_2} & \text{if } v = v_1, v_1 \neq v_2, a \in \mathcal{A} \end{cases} \quad (16)$$
$$\forall l \in L: \text{dr}^l_{a,v_1,v_2,l} \leq \mathcal{M} \cdot u_{a,v_1,v_2,l} \quad (17)$$
$$\forall l \in L: u_{a,v_1,v_2,l} \leq \text{dr}^l_{a,v_1,v_2,l} \quad (18)$$
$$\sum_{l \in L} u_{a,v_1,v_2,l} \cdot d(l) \leq d_a^{\max} \quad (19)$$

The data rate on inputs of each instance determines the *minimum* resource demands of it. For selecting the right piece of the piecewise linear function, we determine the load level for every potential deployment version. For this, we compare the sum of all input data rates of the instance to the pre-defined upper and lower bounds for each load level (Constr. 20, 21). Exactly one load level is indicated as the right one (Constr. 22, 23). Based on the load level, we calculate the *potential* minimum CPU demand of each potential version (Constr. 24). We repeat the same process for determining the *potential* minimum GPU resource demands ($g_{c,v}$). We omit the corresponding constraints due to space limitations.

$$\forall c \in \mathcal{C}\backslash\mathcal{C}_{\mathrm{SRC}}, \forall v \in V, \forall \mathrm{ver} \in \mathrm{VER}, \forall \mathrm{lev} \in \mathrm{LEV}:$$

$$\mathrm{ub}_c^{\mathrm{lev}}(\mathrm{ver}) - \sum_{k \in [1, n_c^{\mathrm{in}}]} (\mathrm{in}_{c,v})_k \leq \mathcal{M} \cdot u_{c,v,\mathrm{ver},\mathrm{lev}}^{\mathrm{cpu}} \quad (20)$$

$$\sum_{k \in [1, n_c^{\mathrm{in}}]} (\mathrm{in}_{c,v})_k - \mathrm{lb}_c^{\mathrm{lev}}(\mathrm{ver}) \leq \mathcal{M} \cdot l_{c,v,\mathrm{ver},\mathrm{lev}}^{\mathrm{cpu}} \quad (21)$$

$$\forall c \in \mathcal{C}\backslash\mathcal{C}_{\mathrm{SRC}}, \forall v \in V, \forall \mathrm{ver} \in \mathrm{VER}:$$

$$\sum_{\mathrm{lev} \in \mathrm{LEV}} (l_{c,v,\mathrm{ver},\mathrm{lev}}^{\mathrm{cpu}} + u_{c,v,\mathrm{ver},\mathrm{lev}}^{\mathrm{cpu}}) = |\mathrm{LEV}| + 1 \quad (22)$$

$$\forall c \in \mathcal{C}\backslash\mathcal{C}_{\mathrm{SRC}}, \forall v \in V, \forall \mathrm{ver} \in \mathrm{VER}, \forall \mathrm{lev} \in \mathrm{LEV}:$$

$$0 \leq l_{c,v,\mathrm{ver},\mathrm{lev}}^{\mathrm{cpu}} + u_{c,v,\mathrm{ver},\mathrm{lev}}^{\mathrm{cpu}} - 2 \cdot b_{c,v,\mathrm{ver},\mathrm{lev}}^{\mathrm{cpu}} \leq 1 \quad (23)$$

$$p_{c,v,\mathrm{ver}} + \mathcal{M} \cdot \left(1 - b_{c,v,\mathrm{ver},\mathrm{lev}}^{\mathrm{cpu}}\right) \geq \mathrm{cpu}_c^{\mathrm{lev}}(\mathrm{ver}, \mathrm{in}_{c,v}) - \left(1 - m_{c,v,\mathrm{ver}}\right) \cdot \mathrm{ccon}_c^{\mathrm{lev}}(\mathrm{ver}) \quad (24)$$

Among the potential versions, only one version may be mapped on a potential location. The resource demands of the optimal versions of components (according to the objectives) are assigned as their final resource demand on the optimal nodes (Constr. 25, 26). Link and node resource consumption cannot be larger than the capacity (Constr. 27-29).

$$\forall c \in \mathcal{C}_N, \forall v \in V: \mathrm{cpu}_{c,v} = \sum_{\mathrm{ver} \in \mathrm{VER}} p_{c,v,\mathrm{ver}} \cdot m_{c,v,\mathrm{ver}} \quad (25)$$

$$\forall c \in \mathcal{C}_N, \forall v \in V: \mathrm{gpu}_{c,v} = g_{c,v} \cdot m_{c,v,\mathrm{ACC}} \quad (26)$$

$$\forall c \in \mathcal{C}, \forall v \in V: \sum_{c \in \mathcal{C}} \mathrm{cpu}_{c,v} \leq \mathrm{cap}_{\mathrm{cpu}}(v) \quad (27)$$

$$\forall c \in \mathcal{C}, \forall v \in V: \sum_{c \in \mathcal{C}} \mathrm{gpu}_{c,v} \leq \mathrm{cap}_{\mathrm{gpu}}(v) \quad (28)$$

$$\forall l \in L: \sum_{a \in \mathcal{A}, v, v' \in V} \mathrm{dr}_{a,v,v',l}^{l} \leq \mathrm{cap}(l) \quad (29)$$

The *potential* maximum time in system at each instance of each component is calculated using the given functions, if a version is mapped to a node (Constr. 30, 31). The time in system at the selected version is assigned as the final time in system for its requests on the target node (Constr. 32).

$$\forall c \in \mathcal{C}_N, \forall v \in V, \forall \mathrm{ver} \in \mathrm{VER}:$$

$$t_{c,v,\mathrm{ver}} = \mathrm{fpt}_c\left(\mathrm{ver}, \sum_{k \in [1, n_c^{\mathrm{in}}]} (\mathrm{in}_{c,v})_k\right) - \left(1 - m_{c,v,\mathrm{ver}}\right) \cdot \mathrm{fpt}_c\left(\mathrm{ver}, \sum_{k \in [1, n_c^{\mathrm{in}}]} (\mathrm{in}_{c,v})_k\right) \quad (30)$$

$$t_{c,v,\mathrm{ver}} \leq \mathcal{M} \cdot m_{c,v,\mathrm{ver}} \quad (31)$$

$$\forall c \in \mathcal{C}_N, \forall v \in V: \mathrm{time}_{c,v} = \sum_{\mathrm{ver} \in \mathrm{VER}} t_{c,v,\mathrm{ver}} \quad (32)$$

We calculate the total CPU and GPU resource cost of every embedded instance on their target nodes per time unit (Constr. 33). By multiplying this value and the time in system at each component, the total resource usage cost of each component on each node can be calculated as:

$$\forall c \in \mathcal{C}_N, \forall v \in V: s_{c,v} = \mathrm{cpu}_{c,v} \cdot \mathrm{cost}_{\mathrm{cpu}}(v) + \mathrm{gpu}_{c,v} \cdot \mathrm{cost}_{\mathrm{gpu}}(v) \quad (33)$$

4.2.2 Objectives

We define the following objective functions for the MIP:

- Obj_1: Minimise the total compute resource cost: $\min. \sum_{c \in \mathcal{C}, v \in V} s_{c,v}$
- Obj_2: Minimise the total time in system: $\min. \sum_{c \in \mathcal{C}, v \in V} \mathrm{time}_{c,v}$
- Obj_3: Minimise network resource consumption: $\min. \sum_{a \in \mathcal{A}, v, v' \in V, l \in L} \mathrm{dr}_{a,v,v',l}^{l}$
- Obj_4: Minimise the number of added, removed, modified instances: $\min. \sum_{c \in \mathcal{C}, v \in V} \delta_{c,v}$

To combine the benefits of using these objective functions, we define the following lexicographical combination of the objectives: min. $w_1 \cdot \text{obj}_1 + w_2 \cdot \text{obj}_2 + w_3 \cdot \text{obj}_3 + w_4 \cdot \text{obj}_4$. Ideally, the weights $w_1, \ldots, w_4$ should be selected such that the range of the values different objective can take do not overlap and each objective has a clear priority. The desired priority among these objectives depends on the use case.

4.3 Heuristic Approach

In this section, we present a scalable heuristic algorithm that finds fast solutions and can be used for large scenarios.

Figure 15 shows an overview of the main procedure of the algorithm. We describe the important steps in the rest of this section.

The list of all templates to be added (T) can be sorted beforehand, e.g., according to the total input data rate from their sources. For new templates, we create an empty overlay (lines 4-5). We then process the source and fixed instances for the template (lines 6-9).

Setting the output data rate of an instance i results in creating/updating outgoing edges from the output(s) of i as well as the inputs of the instances where these edges are destined. For this, the algorithm must decide how many instances of which versions of the subsequent component need to be created on which nodes. We describe this with an example.

In Figure 14, while setting the output data of instances of S, the algorithm needs to create at least one instance of SRV. For every deployment version of SRV (in this example, SRV has only a VM deployment version), it looks for potential nodes. As one of objectives (described in Section 4.2.2) is to minimize the number of added/removed instances, the algorithm takes a greedy decision; it tries to create an instance of SRV with the maximum possible input data rate. If the total outgoing data rate of S is higher than the upper bound of the load level HIG for SRV, it creates an instance of SRV and sets its input data rate to this highest possible value. For the remaining data rate from S, it creates additional instances of SRV in the same way, until there is no more traffic left to be forwarded to a SRV instance.

1: **if** $\exists G_{\text{OL}}(T)$ with $T \notin \mathcal{T}$ **then**
2: remove $G_{\text{OL}}(T)$
3: **for all** $T \in \mathcal{T}$ **do**
4: **if** $\nexists G_{\text{OL}}(T)$ **then**
5: create empty overlay $G_{\text{OL}}(T)$
6: **for all** $(c, v, \lambda) \in S_T$ and $(c, v) \in X_T$ **do**
7: **if** $\nexists i \in I_{\text{OL}}$ **then**
8: create instance $i \in I_{\text{OL}}$
9: set/update output data rate of i
10: **if** a source instance i with no data rate exists **then**
11: remove i
12: **for all** $i \in I_{\text{OL}}$ in topological order **do**
13: c : component corresponding to i
14: IN: sum of data rates on all inputs of i
15: **if** $n_c^{\text{in}} > 0$ and $\text{IN} = 0$ **then**
16: remove i and go to next iteration
17: compute output data rates of i
18: **for all** output k of i **do**
19: set/update output data rate of i

Figure 15: Heuristic approach:

At the same time, it selects the candidate nodes that can host the created instances. These nodes should have enough capacity and there should be a path to them from the node where S is located. The links over the path should have enough capacity and a total delay not larger than the maximum tolerable delay. Locations with an existing instance of SRV from a previous embedding are also considered. Among the candidate nodes we can now select the best option, considering the resulting time in system with the deployment version at that load level and resource usage cost on that node.

We then iterate over the instances of the overlay in a topological order (line 12). That is, each instance i is processed only after all instances that have an edge to i (as specified in the template) have already been processed. The first instances that are processed are the instances after the source instance that are created while setting the output of the source instance in line 9 (instances of SRV in the previous example).

Next, we compute and propagate the data rates from outputs of the current instance towards other components (lines 17-19). In the previous example, this is the data rate towards DPI and CHE. If the data rate needs to be increased (i.e., if there are no previous embeddings of DPI or CHE, or if the previous data rate was less than the computed data rate at this step), we proceed in the same way as described for outputs of the source instance. If the data rate needs to be decreased, a similar process is required to select the most suitable instances of the subsequent components that should get a lower data rate. To limit the range of required modifications, we select an outgoing edge that has the smallest data rate larger than or equal to the data rate that we need to decrease.

If the current instance is an instance of an ingress component, we first calculate the most suitable embedding for all of its outgoing branches. Then, comparing the total cost of each branch (calculated as the total resource usage cost during the total time in system), the cheapest branch is added to the overlay and the rest of them are removed.

4.4 Evaluation

We have implemented both the optimization and the heuristic approaches as Python programs. For solving the MIP, we have used the Gurobi Optimizer 8.0.1[3]. We have used the benchmarks for the Virtual Network Mapping Problem (VNMP)[4] to build the topology of our substrate networks. We present the results of our experiments to compare the how the heuristic algorithm solves the problem compared to the MIP approach.

To achieve results in a reasonable time, we have used a simple template (T_1) consisting of a source component and a DPI in different versions. We have set the resource demands and the time in system based on the findings of Araújo et al. [12], who have analysed DPI VNFs with and without GPU-acceleration. Based on their findings, we assume an ACC version performs 20 times faster than the VM. We assume other resources like memory, buffer, or disk space can be adjusted as needed similar to the compute resources. We have used the smallest substrate network from VNMP with 20 nodes and 44 links, with uniform capacities and resource costs over the network. We have set the GPU capacity of each node to 5 times less than its CPU capacity and the GPU usage cost per time unit on the same node to 50 times more than the CPU usage cost[5].

In these experiments, we have increased the data rate flowing from the only source instance of the template from 1 to 70. We have captured the values of different metrics for 4 cases: (i) Heuristic approach, considering both versions of DPI, (ii) MIP approach, considering both versions of DPI, (iii) MIP approach, considering only the VM version of DPI, and (iv) MIP approach, considering only the ACC version of DPI. To see the behavior of the algorithms in different load situations, we have embedded the template with different source data rates *without* considering its previous embeddings.

The heuristic approach starts selecting accelerated versions of DPI early on, because of its greedy decision process that tries to push as much of the input data rate as possible to the first instance it creates at each step. As the instances are created one by one without considering the whole template, the required instances in the next steps cannot be considered. For this reason, the heuristic creates embeddings that are more costly than

[3] http://www.gurobi.com/

[4] https://www.ac.tuwien.ac.at/files/resources/instances/vnmp

[5] These ratios *roughly* follow the Amazon EC2 On-Demand Pricing model. Concrete information about resource unit prices cannot be extracted from these models, as the pricing model is based on predefined instances with a certain group of resources reserved for them.

the ACC-only experiments with the MIP approach (Figure 16). In exchange, the created templates have a very low time in system, making the approach favourable for time-sensitive services (Figure 17).

The MIP approach creates more balanced results, favouring low-cost solutions longer. The cost of the solutions lies between those of the VM-only and ACC-only experiments. Above the source data rate of 35, the MIP approach starts creating more ACC versions of the DPI (in addition to VM versions) as the load increases, which consume GPUs (Figure 18). This results in the gradual increase in the cost (Figure 16), decrease in the time in system (Figure 17), and only minor increase in the total number of CPUs (Figure 19).

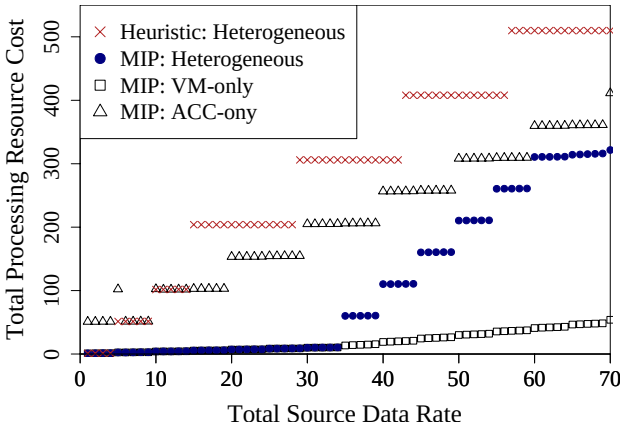

Figure 16: Resource cost based total data rate.

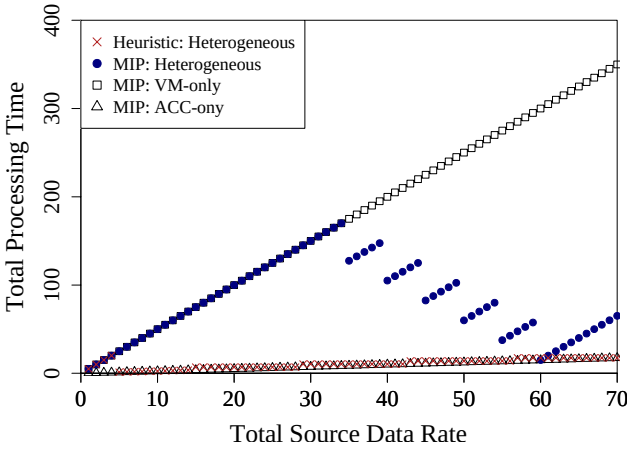

Figure 17: Time based on total data rate.

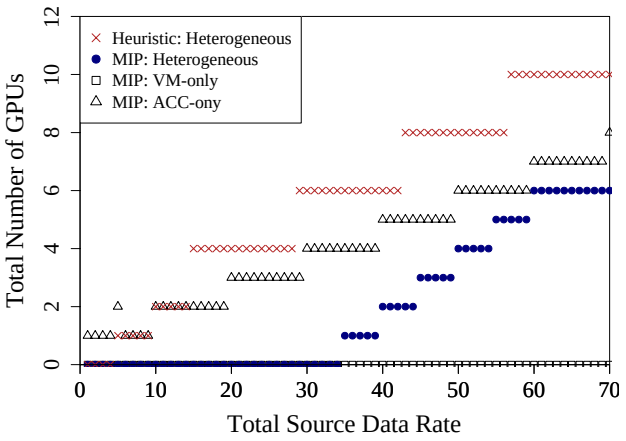

Figure 18: Total GPU demand based on total data rate.

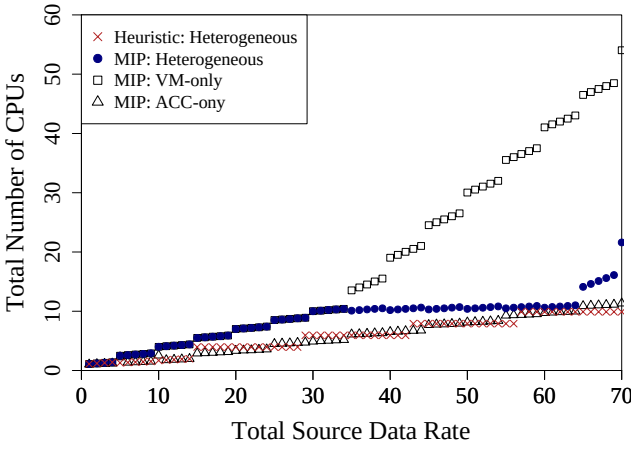

Figure 19: Total CPU demand based on total data rate.

5 Related Work

In this section, we give an overview of related work in the context of multi-version services.

5.1 Experimental evaluation of dynamically provisioning of service

There has been multiple work trying to evaluate and compare the performance of VNF deployed on heterogeneous resources which are mentioned in the following.

Nobach et al. [13] studied the performance and cost of COTS-based DPI versus FPGA-based DPI. To this end, they implemented a DPI that uses COTS resources for all its tasks and another one that offloads network-intensive tasks onto FPGA resources. In the evaluation, throughput and latency were used as metrics and packet sizes ranging from 64 to 1518 bytes were used as a parameter to compare the DPIs. Throughput evaluation results show that the FPGA-based DPI can process packets up to 8.7 Mpps while COST-DPI supports only 0.07 Mpps at maximum. The important point here is that a significant difference is seen for packet sizes equal or less than 512 bytes. For larger packets, the throughput difference is negligible. The same trend can also be seen in the latency evaluation results. The latency for the packets up to 512 bytes is less than 8 µm for the FPGA-based DPI while COTS-based DPI can take up to 8 µm for any packet sizes. This study, also, estimated the yearly cost of using FPGA and COTS resources. The cost of FPGA resources is 325 Euro; 2.4 times higher than the costs of COTS resources (135 Euro per year).This result shows a clear trade-off between performance in terms of throughput and latency and costs. Using FPGA-based DPI for packet sizes up to 512 bytes can improve the throughput up to 124 times. Also, it can improve the latency up to 6 times. However, in the case that packet sizes are bigger than 512 bytes, the performance improvement becomes negligible. With that, we can observe that the use of FPGA-based DPI is not a proper solution for large packets as it costs 2.3 times more than COTS-based DPI without have a significant performance improvement.

Araújo et al. [12], studied the use of GPUs to assist packet processing in DPIs. To this end, they implemented and evaluated a CPU-based and a GPU-assisted DPIs. Three metrics including processing time, resource utilization and overall latency were used for the evaluation. The results show that the GPU-assisted DPI can process packets up to 19 times faster than CPU-based DPI. Also, for resource utilization, GPU-assisted DPI shows a better result as it uses 4 times less CPU core and memory than CPU-based GPU. However, the overall latency shows a quite different trend as, in this metric, CPU-based DPI has up to 31 times better overall latency compared to GPU-assisted DPI. This is because, in GPU-assisted DPI, there is the extra CPU and GPU communication overhead which increases the overall latency. This result clearly shows that the use of GPU for implementing DPI is not always beneficial and that there is a trade-off between resource utilization and total latency.

Han et al. [14] proposed PacketShader, a framework that enables offloading computation and memory-intensive operations to GPUs. In this study, a softwarised GPU-assisted OpenFlow switch has been implemented and tested against a softwarised COTS-based switch to quantify the gain of using GPU as an accelerator for packet processing. For the evaluation, the throughput of switches was measured with the flow table size as parameter. The results show that a GPU-assisted OpenFlow switch can have up to 10 times better throughput than the COTS-based one. However, considering the cost of using GPUs, the use of a GPU-assisted switch is not always the best solution. The evaluation results also show that the highest throughput improvement is achieved when the number of flow entries is high (1M of exact entries and 1k of wildcard entries), but in the opposite situation, when the number of packets is low, the throughput improvement is negligible. As a result, dynamically using both COTS-based and GPU-assisted OpenFlow switch according to the number of flow entries can reduce the cost and improve the performance of softwarised OpenFlow switch.

5.2 Scaling, Placement, and Routing for Multi-Version Services

The scaling, placement, and routing problem described in this document has similarities to the virtual network embedding (VNE) problem [15] that also aims at an efficient resource allocation. Unlike the fixed structure of virtual networks in VNE, our templates can be embedded with different structures on heterogeneous resources. Among VNE studies, some also consider a heterogeneous substrate network. For example, Li et al.[16] propose a joint resource allocation and VNE solution in 5G core networks. They enable efficient physical resource sharing by optimizing the resource demands before embedding. However, the nodes of the virtual networks in their approach (the service components in our model) have a pre-defined number of instances and a fixed deployment version. Baumgartner et al. [17] consider the VNE problem in the mobile core network,

optimizing the structure of the the virtual core network service chain. The flexibility of the service structure in their solution is limited to the way a fixed number of VNFs are grouped and distributed.

Resource allocation is an important problem in the fields of distributed cloud computing [18] and network softwarisation [19]. Most of the cloud computing solutions focus on resource allocation for single-component services [20] or consider only a subset of our problem. Different solutions exist for resource allocation in network function virtualization area, each following different objectives, e.g., scalability of the approach [21], maximizing the admitted requests [22], minimizing the costs [23], [24]. We follow a multi-objective optimization approach. Similar to our approach, Mehta and Elmroth [25] study the trade-off between cost and performance in mobile edge clouds within heterogeneous 5G networks. None of the mentioned solutions consider the ability of instantiating different versions of the same service component.

6 Conclusions and Future Work

In this document, we have shown the feasibility and advantages of defining multi-version services that include components with different deployment options.

We have described the prototype implementation of a management and orchestration solution based on the 5G OS architecture, which can support deployment of VNFs on heterogeneous resources. We have also presented optimization and heuristic approaches for the joint scaling, placement, and routing decisions for these services.

Our prototype implementation as well as the theoretical approaches can create low-cost embeddings for low-load situations and with increasing load, switch to hardware-accelerated versions of service components for lower time in system. Using our approaches, different resource types in heterogeneous infrastructures can be used efficiently to get suitable resource costs and delay.

In the remaining time till the end of the project, we will focus on the following main activities.

We will extend Pishahang's orchestrator to support on-the-fly switching of service versions according to the give requirements. To this end, we will integrate the placement and scaling algorithm described in Section 4 into the Pishahang's orchestrator. We will also extend Pishahang's monitoring systems to capture monitoring data such as CPU and memory usage form containers running on Kubernetes domain.

We will implement an example multi-version service to be used for experiment and demonstration of Pishahang's copiability to support management and orchestration of multi-version services. This service will contain two versions: a COTS version and a hardware-accelerated version.

We will evaluate Pishahang's performance on managing and orchestrating multi-version services. In this evaluation we measure how much time is required to deploy multi-version service, how fast can Pishahang react to changes in service requirements, and how much does it take for Pishahang to switch to different versions of a service. The result of this evaluation will be included in deliverable D2.3 of 5G-PICTURE.

In order to support multi-version VNFs, we have also started the work to support Kubernetes in JoX[6] as a VNFM. Supporting multi-functional split VNF is done through the domain controller FlexRAN[7]. We are also planning to add support for different types of resources like GPU, in addition to what is already supported by JoX that are lxd, kvm, and physical machines.

[6] http://mosaic-5g.io/jox/
[7] http://mosaic-5g.io/flexran/

7 References

[1] 3GPP, “3GPP TR 38.801 V14.0.0 (2017-03): Study on new radio access technology: Radio access architecture and interfaces.” 3GPP, 2017.

[2] 5G-PICTURE. (2018). D2.2 System architecture and preliminary evaluations.

[3] 5G-PICTURE. (2018). D5.1: Relationships between Orchestrators, Controllers, slicing systems.

[4] Razzaghi Kouchaksaraei, H., Dierich, T., & Karl, H. (2018). Pishahang: Joint Orchestration of Network Function Chains and Distributed Cloud Applications. 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft). Montreal.

[5] Razzaghi Kouchaksaraei, H., Dräxler, S., Peuster, M., & Karl, H. (2018). Programmable and Flexible Management and Orchestration of Virtualized Network Functions. 2018 European Conference on Networks and Communications (EuCNC). Ljubljana, Slovenia.

[6] Petcu, D. (2013). Multi-Cloud: Expectations and Current Approaches. Proceedings of the international workshop on Multi-cloud applications and federated clouds.

[7] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., & Akella, A. (2014). OpenNF: Enabling Innovation in Network Function Control. SIGCOMM. In Proceedings of the 2014 ACM Conference on SIGCOMM.

[8] Liu, L., Xu, H., Niu, Z., Wang, P., & Han, D. (2016). U-HAUL: Efficient State Migration in NFV. Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems.

[9] Harchol, Y., Hay, D., & Orenstein, T. (2018). FTvNF: fault tolerant virtual network functions. Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems. Ithaca.

[10] SONATA NFV: Agile Service Development and Orchestration in 5G Virtualized Networks. (n.d.). Retrieved from http://sonata-nfv.eu/

[11] Dräxler, S., & Karl, H. (2019). SPRING: Scaling, Placement, and Routing of Heterogeneous Services with Flexible Structures. 5th IEEE International Conference on Network Softwarization (NetSoft).

[12] Araujo, I. M., Natalino, C., Santana, A. L., & Cardoso, D. L. (2018). Accelerating VNF-based Deep Packet Inspection with the use of GPUs. 20th International Conference on Transparent Optical Networks (ICTON).

[13] Nobach, L., Rudolph, B., & Hausheer, D. (2017). Benefits of conditional FPGA provisioning for virtualized network functions. 2017 International Conference on Networked Systems (NetSys). Gottingen.

[14] Sangjin, H., Keon, J., KyoungSoo, P., & Sue, M. (2011). PacketShader: A GPU-accelerated Software Router. SIGCOMM Comput. Commun. Rev., 41(0146-4833), 195--206.

[15] Fischer, A., Botero, J., Beck, M., de Meer, H., & Hesselbach, X. (2013). Virtual Network Embedding: A Survey. IEEE Communications Surveys & Tutorials, 15(4), 1888-1906.

[16] Li, J., Zhang, N., Ye, Q., Shi, W., Zhuang, W., & Shen, X. (2017). Joint Resource Allocation and Online Virtual Network Embedding for 5G Networks. IEEE Global Communications Conference (GLOBECOM).

[17] Baumgartner, A., Reddy, V. S., & Bauschert, T. (2015). Mobile Core Network Virtualization: A Model for Combined Virtual Core Network Function Placement and Topology Optimization. 1st IEEE Conference on Network Softwarization (NetSoft).

[18] Endo, P., de Almeida Palhares, A., Pereira, N., Goncalves, G., Sadok, D., Kelner, J., Melander, B, Mangs, J. (2011). Resource allocation for distributed cloud: concepts and research challenges. IEEE Network, 25(4), 42-46.

[19] Herrera, J. G., & Botero, J. F. (2016). Resource Allocation in NFV: A Comprehensive Survey. IEEE Transactions on Network and Service Management, 13(3), 518-532.

[20] Mann, Z. A. (2016). Interplay of virtual machine selection and virtual machine placement. 5th European Conference on Service-Oriented and Cloud Computing.

[21] Khebbache, S., Hadji, M., & Zeghlache, D. (2017). Virtualized Network Functions Chaining and Routing Algorithms. Computer Networks, 114, 95-110.

[22] Kuo, T. W., Liou, B. H., Lin, K. C., & Tsai, M. J. (2016). Deploying Chains of Virtual Network Functions: On the Relation between Link and Server Usage. IEEE INFOCOM.

[23] Ahvar, S., Phyu, H. P., & Glitho, R. (2017). CCVP: Cost-efficient Centrality-based VNF Placement and Chaining Algorithm for Network Service Provisioning. IEEE NetSoft.

[24] Luizelli, M. C., da Costa Cordeiro, W. L., Buriol, L. S., & Gaspary, L. P. (2017). A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining. Computer Communications, 102, 67-77.

[25] Mehta, A., & Elmroth, E. (2018). Distributed Cost-Optimized Placement for Latency-Critical Applications in Heterogeneous Environments. IEEE International Conference on Autonomic Computing (ICAC).

8 Acronyms

Acronym	Description
5G OS	5G Operating System
COTS	Commodity off-the-shelf
DPI	Deep Packet Inspector
GOP	Group of Picture
KPI	Key Performance Indicator
MANO	Management and Orchestration
MIP	Mixed-Integer Program
NFV	Network Function Virtualization
NFVO	Network Function Virtualization Orchestration
RAN	Radio Access Network
SDN	Software-Defined Networking
SLA	Service-Level Agreement
VIM	Virtualized Infrastructure Manager
VNF	Virtual Network Function
VNFM	VNF Manager
VNMP	Virtual Network Mapping Problem
vTC	Virtualized Transcoder