University of Amsterdam

Research Project 2

Design and Integration of a Smart Orchestrator for 6G Services

January 23, 2026

Students:
Muhammad Mansour
15294234

Tomás E. Agata
15767647

Supervisor:
Marios Avgeris

Course:
Research Project 2

Course code:
53842REP6Y

Abstract

6G networks demand ultra-low latency and high reliability requiring advanced orchestration frameworks over application and network infrastructure. This project presents a hierarchical orchestration framework integrating Oakestra[1], an open-source edge application orchestrator, with the Infrastructure Management Layer (IML), a network function orchestrator from the DESIRE6G project[2]. The integrated architecture extends Oakestra's deployment capabilities to enable unified orchestration of application microservices and network services. Through hierarchical delegation, Oakestra handles service deployment while delegating network function deployment to IML, thus bridging the gap between application-level and network-level orchestration.

We demonstrate the proposed orchestrator through an experimental deployment of two microservices connected via an NFRouter (network function router) deployed on-demand by IML. This demonstration shows automated, service-aware network deployment, which satisfies 6G performance requirements. The key contribution is a unified orchestration approach based on NSDs and hierarchical delegation to orchestrate application and network services in an end-to-end manner, ensuring low-latency and reliable service delivery as required in 6G[3].

1 Introduction

Modern 5G networks have significantly improved mobile connectivity, but emerging applications such as smart transportation systems, remote robotic surgery, and industrial automation require near-zero latency and ultra-high reliability, exceeding current capabilities. The transition to 6G networks aims at reaching these objectives, including hyper-reliable, ultra-low latency, and large-scale communication[3].

Realizing this vision involves great enhancements in the management and deployment of distributed services, requiring existing network architectures to evolve and scale effectively with minimal human intervention. Specifically, moving from centralized 5G deployments to the distributed, multi-site architecture anticipated in 6G presents difficult scalability issues [4]. Traditionally, the easiest way to support more applications on a single server is to buy a bigger server; this is called vertical scaling. But still there are only so many applications a single server can support, therefore if further scaling is desired a switch to horizontal scaling must be made: instead of buying bigger and better servers, multiple smaller servers can be used that distribute the load between them. By grouping servers together we obtain a cluster or site. Operating ap-

plications across many sites is far more complex than in a single cluster, as it must maintain consistency and reliability across many different resources [5].

Managing service deployment in distributed infrastructures is complex due to the diverse and dynamic nature of the environment, that becomes more complicated as the system scales. Moreover, when managing multiple sites at the same time, determining the best location for an application involves figuring out the best clusters for the task, ensuring compliance with local data regulations, orchestrating the application deployment in each of the clusters, and configuring the networking components -such as DNS or firewalls- required to allow multi-site communications. This would not be much of a problem if no errors were to arise, however, nowadays service level agreements (SLAs) require that an application is able to withstand failures and achieve almost to no downtime. As a result, the process of multi-site deployments must lean towards full automation [1].

Another major challenge lies in automatically configuring the network for deployed applications by seamlessly creating, modifying, and removing network service chains with zero or near-zero latency. When applications extend across several clusters, the network must be dynamically reconfigured (e.g. routing, DNS, firewalls, compression proxies) to connect service components across sites. Traditional static network configurations are insufficient, instead, the concept of Network Service Descriptors (NSDs) from the NFV (Network Function Virtualization) domain offers a way to describe end-to-end network service chains that meet an application's specific needs, and NSDs also specify the dependencies between the application and the underlying network. By decoupling the application logic from the underlying network setup, NSDs allow an orchestrator to customize the network based on each application's requirements. This decoupling is critical for the flexibility and adaptability demanded by 6G scenarios [6].

The main purpose of this research project is to try our hands at addressing these challenges by designing and developing an orchestrator that is able to seamlessly integrate application and network service chain deployments into one single unified platform. Building on Oakestra (an open-source hierarchical orchestrator[1]), we propose to extend its architecture with an additional module for network service orchestration called Infrastructure Management Layer (IML) module[2]. The IML will receive high-level network service descriptors from the Root Orchestrator and deploy it on worker nodes. By automating the deployment of both application components and the necessary network functions, the system aims to enable scalable, low-latency, and reliable service deployments across multiple sites, which is in line with 6G programmability requirements.

2 Connection to Previous Work

Our first research project (RP1) investigated heavy-hitter detection algorithms (Count Min, Bloom Filter, PRECISION) on programmable switches and produced a reusable benchmarking testbed with virtual P4 switches (BMv2). This previous work established a foundation for understanding and implementing heavy-hitter detection in the network data plane. While RP1 focused on data plane mechanisms for detecting heavy hitters, the current project builds on those results by moving up a layer to the control plane. In this project, we explore control-plane automation that would allow deploying such algorithms as part of a larger service chain. The Heavy-Hitter module from the first project could be deployed as an infrastructure network function that would be used to export real-time traffic statistics to the new orchestrator. Using these statistics, the orchestrator can react automatically by scaling out new application micro-services or replicas, or adding extra network functions, such as compression, and steering Heavy-Hitter flows through them. The project delivers a prototype multi-site orchestrator that intelligently manages both application containers and network services, improving automation in edge computing environments.

3 Research questions

This research is guided by the following key questions:

- How can we design a hierarchical, distributed orchestration system to support scalable, low-latency, and reliable service deployments across multiple sites?
- What architecture is suitable for integrating network service orchestration into a hierarchical multi-site orchestrator without disrupting the normal application deployment flow?
- In what ways can network service descriptions (NSDs) be used to improve flexibility and adaptability in application-to-network integration? How the orchestrator can interpret an NSD to deploy virtual network functions (routers, firewalls, etc.) across sites and dynamically reconfigure networking between application components?

4 Related Work

The growth of edge computing has increased the number of studies of orchestrators capable of managing both cloud and edge resources. A service orchestrator allows developers to specify an application's requirements (resources, placement constraints, etc.), and the system automatically handles the low-level deployment and management details. Kubernetes is a leading example, an open-source container orchestration platform originally designed for datacenter clusters. Kubernetes has become the most popular orchestration system for cloud container management (used in production by ≈59% of large organizations). However, Kubernetes assumes a single cluster with reliable high-bandwidth connectivity between nodes, which is an assumption that has limitation in in geo-distributed and edge environments. All resources must be located in one cluster and be directly reachable, meaning Kubernetes cannot easily coordinate multiple independent clusters. KubeFed (Kubernetes Federation), K3s, KubeEdge, and MicroK8s are variants and extensions developed to make Kubernetes more lightweight or to federate multiple clusters. However, these solutions still inherit many assumptions of the original Kubernetes design and they showed limitations at edge environment [1]. Early efforts to extend orchestration to the edge include platforms such as FocusStack and ParaDrop and to overcome Kubernetes' limitations for multi-site deployments, researchers have explored hierarchal orchestration frameworks such as Oakestra.

Amento et al., 2016 [7] introduced FocusStack, an orchestration approach that allows the focus of attention of the cloud control plane to be based on the location, health, and capabilities of the edge device to decide where to deploy services. As a result, it incorporates the client into scheduling. At the same time, Liu et al., 2016 [8] explored ParaDrop, a lightweight edge computing platform on WiFi routers, that uses computing and storage resources at the edge of the network (access points) to allow third-party developers to flexibly create new types of service. ParaDrop framework enables the deployment of various third-party applications on the Access Point, allowing support for various end devices, such as wireless cameras and environmental sensors. These systems established the need for decentralized orchestration but were often limited to specific scenarios and did not offer a general multi-site hierarchy.

Oakestra, proposed by Bartolomeo et al.[1], is a recent study that employs hierarchical orchestration frameworks to effectively manage resources from cloud to edge. Oakestra offers a two-tier orchestration model, Root Orchestrator and Cluster Orchestrator. The Root Orchestrator performs top-level scheduling by selecting suitable clusters for an application's microservices based on high-level constraints (latency, hardware, policies, etc.), while each Cluster Orchestrator then places those microservices to suitable worker nodes within its local cluster. Oakestra also introduces an overlay network with built-in tunneling to allow application components on different clusters to communicate with minimal developer intervention. Oakestra outperforms traditional orchestration frameworks (including standalone Kubernetes and its lightweight variants), and achieves lower CPU and memory overhead and faster deployment times. The hierarchical delegation of Oakestra accelerates scheduling and more effectively supports the unified cloud-edge framework, where different clusters might be operated by different providers but collaborate to host a service. We plan to utilize Oakestra's open-source and extend it with new capabilities for network service management.

Besides compute orchestration, the networking community has developed frameworks for orchestrating virtual network functions (VNFs) and network services. The ETSI NFV (Network

Function Virtualization) architecture defines a Management and Orchestration (MANO) layer which is responsible for managing VNFs and chaining them into full network services [9]. NFV MANO has an important concept called Network Service Descriptor (NSD), which is a specification of an end-to-end network service, including its VNFs and their interconnections [10]. Many open-source projects implement these ideas such as ETSI Open Source MANO (OSM) which is an orchestrator aligned with the ETSI NFV standards [11]. OSM offers a platform for the deployment of network services over multiple edge sites, and allows operators to provide VNF descriptors and NSDs, then automates the setup of the required virtual machines/containers and network connectivity across available infrastructure. By using NSDs, an orchestrator such as OSM can dynamically compose network services on-demand, instead of depending on static network configurations. For instance, automatically setting up a chain of a traffic load-balancer and a firewall between two microservices hosted on different clusters [11]. There has also been research on combining NFV network service orchestration with edge computing. For example, projects such as SONATA and ETSI MEC attempt to integrate service function chaining with edge application placement, even though most existing solutions consider network orchestration and application orchestration as separate domains [12] [13].

Deliverable D2.2 of the Horizon-Europe DESIRE6G project [2] defines a service-oriented 6G architecture. It couples a fully-programmable user-plane extending from the RAN to the core, a cluster-level Infrastructure Management Layer (IML) that abstracts the diversity of CPUs, SmartNICs, and FPGAs, and a set of AI control loops that modify resources to keep the system zero-touch. Our work intersects with this idea by integrating network-service orchestration into the general computing orchestration domain. Instead of initiating an application and then calling a separate MANO stack to configure the network, we ask the Oakestra orchestrator to generate the required network functions as part of the same deployment phase. This is done in this study by extending Oakestra with its own IML, which is adjusted to be suitable. In our work, the IML plays a role similarly to an NFV Orchestrator, but specialized for cluster-level and integrated with the application orchestrator. The 5G-PPP 5G-INDUCE project [14] has already shown that this combination is practical: its three-layer platform (Application Orchestrator, OSS/BSS and NFV-O) enable developers to request network capabilities through one API while separating them from the details of the infrastructure. This approach - where developers can simply state latency or bandwidth goals and let automation delver them- is now widely recognized as a fundamental component for 6G.

In light of all the above, our work stands at the intersection of distributed systems (multi-cluster orchestration) and network systems (NFV orchestration). Existing technologies provide building blocks: Kubernetes for container management, OSM for NFV management, Oakestra for hierarchical edge orchestration, and IML for Network services management. However, the integration of these elements into a coherent, developer-friendly platform is still an open research problem. Our contribution will be to design and prototype such an integrated orchestrator that will be a key element of the 6G era of distributed services and its requirements including scalability, low latency, reliability, and flexibility.

5 Methodology

To address the research questions, we followed a methodology combining design analysis, prototype development, and iterative testing. We began by studying Oakestra's open-source code and documentation [1] to understand its hierarchical orchestration flow. This involved reviewing Oakestra's architecture (Root Orchestrator, Cluster Orchestrators, Worker Nodes), how clusters register with the root, how tasks are scheduled, and how it handles application deployment requests. In parallel, we examined the Infrastructure Management Layer (IML) concept from the DESIRE6G project's Deliverable D2.2 [2], treating it as our candidate solution for network service orchestration. The IML specification was analyzed to determine how it abstracts diverse data-plane resources and manages virtual network functions (VNFs) through Network Service Descriptors (NSDs).

Based on this background, we formulated possible integration methods. One design scenario was to write a custom network orchestration module from scratch to integrate with Oakestra.

However, developing a custom solution carried the danger of duplicating existing NFV orchestration concepts and would require significant effort to reach feature equivalence with known frameworks. The selected alternative approach was to leverage the existing IML framework from DESIRE6G [2] by modifying and integrating it with Oakestra [1]. This reuse offered the potential for alignment with state-of-the-art 6G research, given IML's design that already aimed at managing VNFs. We decided that extending Oakestra with IML would best meet the project goals of unifying application and network orchestration.

After selecting the integration approach, we proceeded with system design and prototyping. We first extended Oakestra's deployment descriptor schema to include an NSD section (detailing required network functions and their connectivity). This extension was guided by NFV standards. For example, NFV's NSD concept provides a template to describe an end-to-end network service chain [15]. By using this format, we ensured our design could dynamically compose network services alongside application deployments. Next, we modified Oakestra's Root Orchestrator code to parse the extended descriptor and split the deployment plan into two parts: (1) the application components (microservices) to be handled by Oakestra's normal workflow, and (2) the network service requirements (the NSD) to be passed to the IML module.

On the IML side, we adapted the available IML prototype to function as an embedded network orchestrator within our platform. This involved writing new interfaces for the Root Orchestrator to transfer the NSD to the IML, and modifying the IML code so that it can accept any application identifiers and connection requirements without needing prior definitions of each application in its code. We instructed IML to interpret the high-level connectivity specification (who needs to be connected to who, with what constraints) and dynamically deploy the required infrastructure network functions. During development, we followed best practices of modular design: Oakestra's existing components (System Manager, Root Scheduler) were minimally changed, and the IML integration was added as a new module to keep the stability of the core orchestrator.

Finally, we established a test environment to validate the integrated orchestrator. We used instances of Oakestra's Root and Cluster Orchestrators, along with an IML agent running in the cluster. More information about the implementation is in the deployment demonstration section. This iterative methodology (analyze, design, implement, then test) allowed us to enhance the integration and ensure that the research questions were answered by the final prototype.

6 Design

6.1 Oakestra

We built our solution on Oakestra's hierarchical orchestration framework [1], as shown in Figure 1. In standard Oakestra without IML integration, a centralized *Root Orchestrator* manages the global view of the infrastructure, handling user requests to deploy applications across edge sites. It maintains a registry of clusters and their aggregated resources (CPU, memory, etc.) reported by each *Cluster Orchestrator*. When a user sends an application deployment request (via a Deployment Descriptor), the Root Orchestrator's Service Manager records the request and calls the Root Scheduler. The Root Scheduler filters and ranks clusters based on the requirements (latency constraints, hardware needs, etc.) using the summary statistics provided by clusters. After selecting the best cluster for a given microservice, the Root Orchestrator assign the deployment to that cluster's orchestrator. Each Cluster Orchestrator serves as the localized control plane for its site. The cluster orchestrator's scheduler does placement of the microservice on one of the available Worker Nodes in its cluster, based on real-time node capacities. The chosen worker then pulls the required container image and launches the microservice. Oakestra's design emphasizes delegated scheduling by dividing the scheduling problem between global (cluster selection) and local (node selection) levels. Another key feature of Oakestra is its overlay networking: each worker's network stack cooperates to provide unified connectivity between microservices even across clusters. Oakestra uses a semantic addressing and tunneling so that services can reach each other by their service IDs regardless of their physical location [1]. This allows Oakestra to abstract the network complexities from developers and it is a starting point for our 6G orchestration platform.

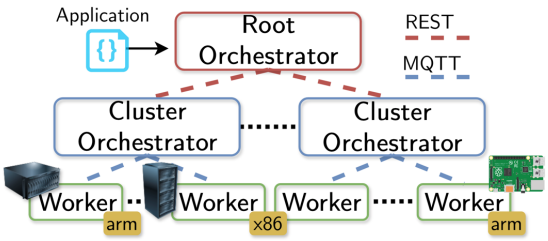

Figure 1: High-level architecture of Oakestra. Adapted from [16]

There is however, certain changes that needed to be reworked on the way that Oakestra works. Oakestra is built to directly manage deployments by connecting to the underlying container engines. However, in order to allow both applications and network service deployments, we needed a orchestration engine that could provide a common API to allow both types of deployments to coexist at the same time while preventing resource competition. As a result of this, we decided to utilize Kubernetes as the underlying orchestration engine. We chose Kubernetes because it is one of the most popular and battle-tested orchestration engines, and also because the Oakestra team provides one plugin that allows modeling all of Oakestra's cluster components as Kubernetes resources, which heavily facilitates the development process. Additionally, by only limiting the reach of Kubernetes to a single cluster, this heavily reduces the impact of the one requirement that Kubernetes requires: high-speed links between nodes.

6.2 Infrastructure Management Layer

In the DESIRE6G architecture [2], the IML is introduced as a specialized layer to manage programmable data-plane resources in a unified way. IML acts as a combination of a Virtual Infrastructure Manager (VIM) and a hardware abstraction layer. Its responsibility is to bridge the gap between the logical network functions that higher-layer want to deploy and the physical infrastructure (CPUs, SmartNICs, FPGAs, P4 switches, etc.) that actually process packets. For example, if a network service requires a firewall or router function, IML can decide whether to run that function in software on a CPU core, or on a programmable switch ASIC, or on a SmartNIC, depending on resource availability and performance needs [2]. At deployment time, IML also sets up virtual links between the deployed functions to ensure that all microservice components and VNFs in a service chain are connected in the data plane. The DESIRE6G IML design includes an embedded local NFV Orchestrator (NFVO) at each site. This local NFVO utilize a site network service description (NSD) and performs tasks like: selecting suitable hosts for each network function, initiating VNF instances (e.g. launching a container), and connecting the VNFs with each other. This distributed NFVO model enhances the hierarchical structure of Oakestra, as each site can independently handle its network functions under the policies set by a central controller. The IML is also designed to enable advanced features such as seamless scaling and heavy-hitter flow management [2]. For instance, it supports dynamic load balancing and can offload elephant flows to hardware accelerators. These capabilities align with 6G goals of high performance and zero-touch automation as the network can reconfigure itself in reaction to traffic conditions[17].

6.3 Unified Architecture

As shown in Figure 2, our Smart Orchestrator for 6G integrates Oakestra's hierarchical orchestration with IML's network service orchestration capabilities into a unified system. The overall design is a three-tier hierarchy with the following roles:

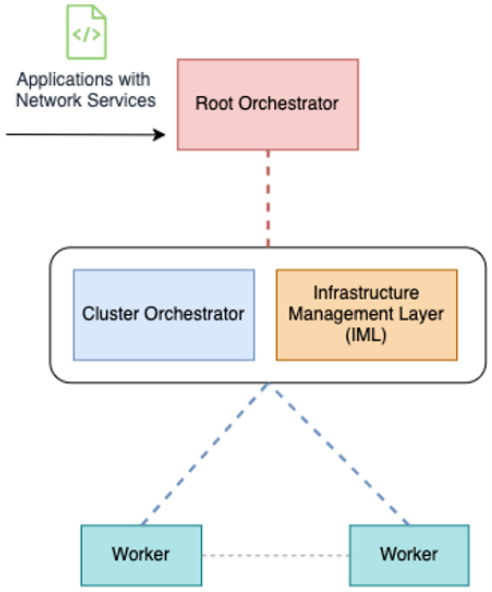

Figure 2: Unified Architecture of our Smart Orchestrator

- *Root Orchestrator* (**RO**): The global entry point that receives deployment requests. In our design, the RO now handles both aspects of a deployment:
  - the application components (microservices), and
  - the required network service chain.

  We extended the deployment descriptor to contain a section for network service requirements in addition to the applications section. When a request comes in, the RO uses its Service Manager to log the request and then invokes two parallel workflows. The first workflow is the standard Oakestra scheduling: determine which cluster will host each microservice (based on SLA and resources) using the Root Scheduler, and then forward each microservice specification to the corresponding Cluster Orchestrator [1]. The new second workflow is that the RO passes the network service descriptor to the IML layer. This means that the RO produces a local NSD file for the target site or cluster. If all microservices of the application are placed in a single cluster, the RO generate one NSD targeting that cluster's IML. If the application spans multiple clusters, the RO can generate a sub-NSD for each site's IML (describing the part of the service chain that runs locally). The RO then triggers each cluster's IML to handle the network setup. This design makes the RO an orchestrator of orchestrators: it coordinates the cluster orchestrators for computing and the IML for networking, ensuring both deployments happen simultaneously. This approach does not require the RO to manage low-level network configurations.[18]

- *Cluster Orchestrator* (**CO**): This component is part of every cluster to manage its workers and application containers. We kept the CO largely unchanged from Oakestra for deploying microservices; we only introduced a communication mechanism between the Worker nodes and the local IML to support runtime coordination. This mechanism works as follows: after the CO schedules and launches the application's microservices on its workers, a special application inside each worker node informs the IML about the deployed applications. This special application consists of a *Container Network Interface* or CNI. This CNI is a common architectural pattern in Kubernetes that is usually tasked with setting up the network interfaces of a newly deployed container. However, this version of the CNI is also tasked with registering itself to the IML; the IML, in turn, answers back with the necessary network details required to set up the interfaces. This addition is absolutely necessary because the IML needs to know where each microservice is running and how to manage or route traffic to/from it. This allows the IML to dynamically integrate new network

functions among microservices. The cluster orchestrator continues to perform monitoring and local scaling decisions for applications, while leaving specific network logic to the IML. This flexibility allows our integrated design to maintain the independence of clusters. If the Root Orchestrator is unreachable, a cluster could continue operating and adjusting its own microservice placement, and the IML at that site could continue managing local network optimizations.[19]

- *Infrastructure Management Layer Agent*: The IML agent includes the local NFVO functionality from DESIRE6G's design [2]. When it receives a network service descriptor from the Root Orchestrator, it parses it to know the list of virtual links and required VNFs and decides how to realize it on the local infrastructure. In our current implementation, the IML agent uses a simple strategy: for each required network function (in our test, a Network Function Router or NFRouter), launch it as a container on one of the available worker nodes, and then configure the routing rules based on the descriptor. Because our testbed did not include special hardware such as SmartNICs, the IML in this phase treated all VNFs as containerized software functions. The IML agent utilizes the assigned network information to microservices to configure the NFRouter's forwarding table. For instance, if the descriptor specifies that microservice A and microservice B should be linked through a router, the IML will create an NFRouter and add rules so that packets from A are forwarded to B (and vice versa) through the router. This is similar to establishing a service function chaining (SFC) path as described by the NSD. In our design, once the NFRouter (or any network function) is running, the IML monitors it just as the CO monitors application containers. The IML regularly reports status back to the Root Orchestrator (e.g., to log that the network service has been deployed and is healty).[20]

This flexible integration method kept Oakestra and IML as separate modules that exchange information, which made implementation easier and preserved modularity. An alternative, but more advanced approach could come from merging the cluster orchestrator and IML functionalities more deeply. In our design, we chose the modular method as this solution retains all existing functionality while preserving independence between the components.

Finally, our integrated orchestrator design supports the idea of developers using a single API to deploy everything. The developer simply submits a deployment descriptor containing both application requirements and a high-level network service specification while the orchestrator manages the internal complexity of scheduling, placement, and chaining. By using NSDs to represent network requirements, we ensure that adding a new network function (e.g., a firewall between two microservices) does not require manual network configuration. Instead, the orchestrator understands that descriptor and automates the necessary steps. This approach aligns with 6G management mechanism that emphasize intent-based requests and closed-loop automation.[17].

7 Deployment Demonstration

To validate the smart orchestrator, we implemented a proof-of-concept deployment including a simple application and a network service function inserted between its components. This scenario describes a common use case: Two microservices, MongoDB and NGINX must be able to communicate with each other using a network service containing a router in the middle. This demonstration is intended act as a base case that simulates a client-server application, with some network functionality in the middle. The entire deployment process consists of a series of three distinct stages or steps:

- User submits specification to the Root Orchestrator
- Cluster Orchestrator deploys microservices
- IML deploys the network service

7.1 User submits application specification to Root Orchestrator

The first phase of deployment starts with a user passing a specification of the desired applications and services in the form of a deployment descriptor to the RO. We defined the deployment using the new descriptor format that our orchestrator supports. In JSON form, the descriptor contains an applications array and a `net_services` array. Each application entry lists its microservices and resource requirements, and each network service entry describes a forwarding graph. Listing 1 is an illustrative snippet of the descriptor used in the test. In this descriptor, each microservice has an `ns_ref` tag (like `app_a` or `app_b`) that is referenced by the network service definition. The `net_services` section describes a forwarding graph named `test1`. This graph contains two directed links: `to_dst` connecting the source (`app_a` instance 1) to the destination (`app_b` instance 1), and `to_src` for the opposite direction. This specifies that traffic between the two microservices should flow through a network service chain. The orchestrator interprets this as a requirement to deploy a service function (in this case, an NFRouter) that will handle traffic in both directions. The `e2e_delay_budget` property indicates the application's tolerance for network delay.

```
{ "sla_version" : "D6G",
  "customerID" : "Admin",
  "applications" : [
    {
      "applicationID" : "",
      "application_name" : "nstest",
      "application_namespace" : "default",
      "application_desc" : "ns test",
      "microservices" : [
        {
          "microserviceID": "",
          "microservice_name": "src",
          "microservice_namespace": "default",
          "virtualization": "container",
          "vcpu": 1,
          "storage": 100,
          "code": "docker.io/library/busybox",
          "ns_ref": "app_a"
        },
        {
          "microserviceID": "",
          "microservice_name": "dst",
          "microservice_namespace": "default",
          "vcpu": 2,
          "storage": 200,
          "virtualization": "container",
          "code": "docker.io/library/busybox",
          "ns_ref": "app_b"
        }
      ]
    }
  ],
  "net_services": [
    {
      "nsID": ""
      "ns_name": "Ping demo"
      "ns_vendor": "D6G"
      "siteID": "d6g-001"
      "forwarding_graphs": [
        {
          "graph_name": "test1",
          "e2e_delay_budget": "5ms",
          "links": [
```

```
            {
              "id": "to_dst",
              "connection_points": [
                {"microservice_ref": "app_a:1"},
                {"microservice_ref": "app_b:1"}
              ]
            },
            {
              "id": "to_src",
              "connection_points": [
                {"microservice_ref": "app_b:1"},
                {"microservice_ref": "app_a:1"}
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

Listing 1: Illustrative snippet of the Deployment Descriptor used in the test

This descriptor is then processed by RO, which schedules in which of the clusters they are going to be deployed (see Figures 3 and 4).

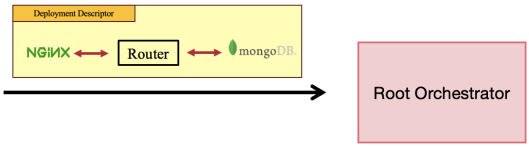

Figure 3: User hands over a deployment descriptor to the Root Orchestrator

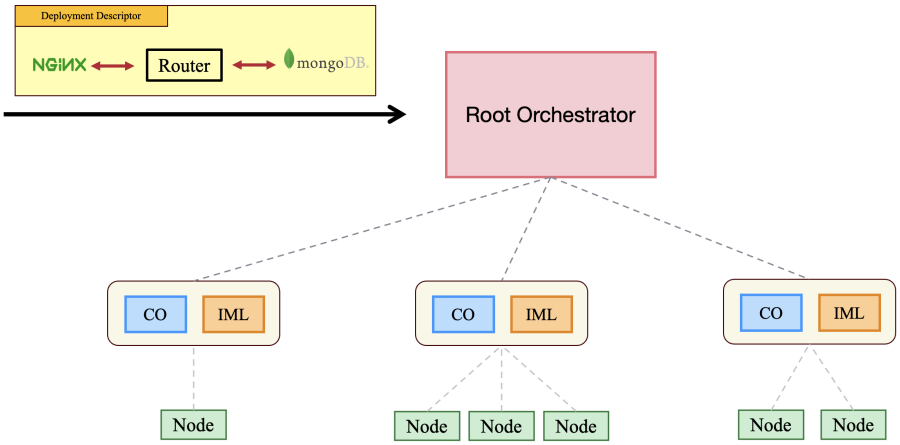

Figure 4: Root Orchestrator schedules the deployment to the best cluster

Once it decides which cluster is the best, it creates a local network service descriptor (local NSD) which is handed over to the IML component in the selected cluster. This local NSD registers the network services for the deployment.

7.2 Cluster Orchestrator deploys microservices

Once the deployment has been scheduled and the network services have been registered with the site's IML, the cluster orchestrator can initiate the process of deploying the microservices (fig. 5). In order for them to register themselves with the IML, the CO has to add an annotation to the Kubernetes deployment file to make them use IML's CNI. Kubernetes will in turn, force the deployment to use the CNI.

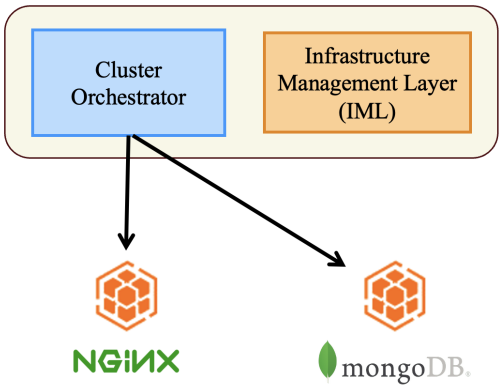

Figure 5: Cluster Orchestrator deploys microservices

7.3 IML deploys the network service

Once the required microservices have been deployed, IML will automatically start the deployment of the required VNFs in the service chain as well as the necessary connections to ensure data-plane connectivity (fig. 6). Once this last step is done, then the network service chain has been successfully set up.

8 Results

The results of our project are primarily qualitative, focusing on the functionality and architectural benefits of the integrated orchestrator. We successfully developed a unified 6G service orchestrator that can deploy both application containers and network functions based on a single high-level specification. The demonstration described above confirmed that the system operates as expected: a network service chain is created in response to the specifications of a user, and the automatically deployed end-to-end service works correctly with minimal manual intervention.

One key outcome is that the integration of IML with Oakestra did not break the deployment process. The orchestration of the network function occurred in parallel with the normal

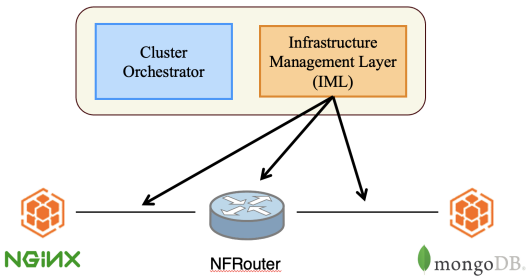

Figure 6: IML deploys network services once all microservices are registered

application scheduling, adding only a small constant overhead required for the NFRouter container to start; this overhead is similar to the startup duration of the application containers themselves. From the user's perspective there is some extra waiting time as the NFRouter deployment process is finished. However, these times can be almost completely removed by modifying the NFRouter deployment to dynamically introduce new rules during runtime operation. This change would allow to deploy the NFRouter early and avoid waiting for the deployment to finish.

Additionally, any application can be used with no changes to its own internal logic. This allows fluid integration with previous orchestration strategies. The only overhead on their deployment comes from the execution routine of the CNI. Performance-wise, this CNI must wait for a response from the IML, and as the IML is using a REST API using HTTP, the exact delay can be approximated to 3 or 4 Round-Trip-Times (RTTs) depending on the TLS version in use [21]. However, these times can be heavily reduced by introducing a local network manager in each node that acts as the IP addressing manager that registers the services and sets up the interfaces asynchronously. This would essentially limit each round-trip-time to microseconds as CNI only has to communicate locally instead of remotely.

Another added benefit from our architecture is that it scales the management plane horizontally by assigning tasks to the cluster and IML agents, which avoids creating bottlenecks at the Root Orchestrator. In a more complex scenario with many microservices and VNFs, we expect similar behavior, where cluster orchestrators and IML agents work simultaneously under the coordination of the root. This is a positive result for the scalability of the architecture, indicating it can support the distributed structure of 6G services.

Another result is the simplification for developers. By using the extended descriptor with NSDs, developers can specify their network requirements, and the orchestrator ensures those requirements are met. In our test, for example, we inserted a routing function into the service chain with less effort. Before, accomplishing the same would require manually provisioning a router VM or container, configuring its interfaces, and updating routing tables on the application nodes. Our orchestrator automated all those steps. This indicates improved flexibility: the platform can support a variety of application topologies (e.g. inserting firewalls, or proxies between components) without hard-coding those in the infrastructure beforehand. The concept of NSD proved to be useful, as it decouples the service logic from the underlying network configuration, and our integration shows that this concept can extend to edge service orchestration.

The tests performed were simple, but they reflect that our orchestrator's decisions to place the NFRouter in software on the edge cluster were reasonable. In the future, with heavier traffic or stricter SLAs, the orchestrator (via IML) could choose alternative placements (e.g. on an FPGA card), and our architecture is ready for such extensions. In addition, the system preserved Oakestra's lightweight nature: the resource overhead of running the IML agent and an extra container was small.

9 Discussion

Our integrated orchestrator addresses various challenges and provides many insights

- **Hierarchical Orchestration Effectiveness**: The project demonstrates the benefits of a hierarchical approach for scaling edge orchestration. We achieved scalability and low latency in the control plane by using a Root Orchestrator to make high-level decisions and offloading finer decisions to cluster-level orchestrators. However, one might ask: could a centralized orchestrator with an integrated network manager do the same? The answer is yes. However as the system grows, we run into the same scalability problems of a monolithic design: The problem of orchestrating applications across multiple nodes is in essence a hard mathematical problem [22]. When adding more nodes, it becomes increasingly hard to schedule deployments, making this theoretical central orchestrator a bottleneck.

  By introducing multiple layers of orchestrators, we heavily reduce the scope of each of the orchestrators, thereby reducing decision complexity while also providing fault tolerance because if one cluster is isolated, it can still operate locally. Thus, for 6G services which may

cover thousands of sites, a hierarchical control plane is not just beneficial but also necessary. We saw that our prototype handled the sample architecture effectively, and we expect the design will scale up well as the number of clusters increases.

- **Integrating Network Service Orchestration**: One of the key contributions of this project is demonstrating a practical way to combine application and network service orchestration. We achieved this by integrating two different orchestrators: a network orchestrator (IML) and an application orchestrator (Oakestra's CO), both managed by a global orchestrator. The loose connection between these two orchestrators is an important consideration. We intentionally kept clear boundaries between their roles: the root orchestrator (**RO**) provides the network service descriptor (**NSD**) to IML without interfering with application details. Similarly, the IML focuses only on network-related tasks and does not manage application containers. This separation clearly defines responsibilities and aligns with the DESIRE6G principle of separating "business logic" (application concerns) from "infrastructure management" (network concerns) [2]. This separation provided two main benefits. First, it enabled us to integrate IML into Oakestra's system with minimal changes to its core. Second, it allowed for modularity, meaning each orchestrator can be upgraded or modified independently. However, it also introduces coordination challenges. Both orchestrators (CO and IML) must consistently share information about the deployment state. In our prototype, we used a basic messaging system where the worker node CNI informs IML about applications and IML replies with the required network information. For real-world applications, a more robust synchronization mechanism, such as a shared database or an event streaming platform such as Apache Kafka would likely be necessary.
- **Network Service Descriptors (NSDs)**: Using NSDs enhanced the flexibility of our orchestrator, enabling intent-based networking. The user specifies what they need (e.g. a chain connecting microservice A and B with certain QoS) and the system decides how to implement it. Users define network requirements through NSDs, allowing the orchestrator to automatically select and deploy Virtual Network Functions (VNFs), simplifying the developer's role and making the system adaptable. For example, if the descriptor had a different topology or multiple network functions, the orchestrator would handle it with the same process, choosing suitable VNFs and linking them. This is similar to how NFV MANO systems operate with their own implementation of NSDs [9], but integrating it with application deployment is novel. As of right now, there is only so much our current NSDs can provide. However, we believe that this version should be considered more of a prototype than production-ready. For the future, a more standardized NSD with popular functionalities could be adopted.

To conclude, we would like to briefly mention that traditionally, applications and network services are deployed separately (e.g. using Kubernetes for applications and NFV orchestrators like ETSI OSM or ONAP for networks), often requiring a manual or custom integration. Our solution merges these aspects into a unified orchestrator, creating a single interface. Although extending Oakestra to include networking features required significant effort, its modular design made this feasible. In this project, we showed an example of the ability to treat "network-as-code" just like "infrastructure-as-code": writing a deployment descriptor to describe a network function and having the system implement it automatically. This means that application developers can start to assume the network is programmable and available on demand as part of their application deployment workflow.

10 Limitations

In the current implementation, we only fully integrated a single type of network function (the NFRouter). This means that if a user wanted, for example, to include a heavy-hitter detection component as a network function in the service chain, additional development would be required. The IML codebase would need extensions to handle this new VNF. The current approach is

not fully generic, but future work could introduce more flexible plugins or adopt a standard like P4 programs to define new functions. Our vision of this project is to simplify the creation and configuration of network services so that they can be easily developed and implemented with very few clicks to avoid many of the downsides of development that we confronted in our first research project where we developed a heavy-hitter detection component. Additionally, the integration between Oakestra and IML is still preliminary and the current messaging mechanism with IML lacks some robustness features. For instance, the messaging between IML and other components hasn't been tested under failure conditions. If those messages were lost or delayed, new application will have to wait until the underlying network becomes reliable again or its problems are fixed. In a production environment, we might would need to implement reliable signaling or a handshake mechanism. Scalability represents another limitation, as the unified orchestrator has not been evaluated at large scale. Oakestra has been shown to scale to many clusters and services, and the IML is theoretically designed for large deployments, but when integrated, new performance questions arise. For example, how quickly can the Root Orchestrator generate and send NSDs for 100 network functions across 50 sites at the same time. The RO might get overloaded if it had to process extremely large descriptors or coordinate many sites.

Finally, our work was done in a lab setting with virtual machines and simulated edge conditions. We did not test the orchestrator in a real 5G/6G environment (with radio base stations, real user traffic, etc.), meaning there may be unexpected limitations when applying this orchestrator in an operational network. Addressing all mentioned limitations is important to move from a prototype to a production 6G orchestration platform.

11 Conclusion

In this project, we designed and implemented a Smart Orchestrator for 6G services that unifies application deployment and network function orchestration in a hierarchal framework. By adding an Infrastructure Management Layer (IML) to the Oakestra edge orchestrator's design, we were able to automate the deployment of network service chains in addition to traditional microservice applications. Through this integration, we addressed the complexities of multi-site service management by splitting responsibilities: the Root Orchestrator performs global placement and splits the deployment into application and network components. The Cluster Orchestrators start application containers and the IML agents dynamically configure the interconnecting network functions. Our orchestrator interprets high-level Network Service Descriptors, which means that developers can specify the networking requirements for their services, and the system will provide those requirements.

The successful proof-of-concept of an end-to-end deployment, where two microservices and an NFRouter were created and connected automatically, validates our approach. It shows that it is feasible to achieve zero-touch service orchestration where both compute and network resources are managed under a common framework, a capability expected to be crucial in 6G networks. The hierarchical design ensures scalability to many edge sites, and the modular integration of IML shows how diverse (from CPUs to SmartNICs) can be abstracted for use by service developers.

Our work bridges a gap between cloud orchestration and telecom network management. This is an important step toward the 6G vision of smart, distributed networks that can be programmatically controlled to meet strict performance and reliability requirements. We built a prototype orchestrator and we demonstrated the features (hierarchical scheduling, NSD-driven NF deployment) on a small scale, which represents the groundwork for handling more complex deployments in the future. The positive outcomes answer the research questions positively: Yes, we can design a hierarchical distributed orchestrator for multi-site 6G services, and yes, we can integrate network service orchestration into it in a way that uses NSDs to enhance flexibility without interrupting normal application deployment.

12 Future Work

As previously discussed, there are multiple ways this orchestrator can be improved to migrate from the current proof-of-concept implementation to a fully-fledged production solution.

- **Dynamic rule allocation for VNFs**: By allowing dynamic rule allocations in each VNF, the IML is capable of pre-deploying to avoid waiting. Once the applications are deployed, then they can be reassigned some control-plane rules to allow for dynamic reconfiguration.
- **Support for additional network functions**: A clear next step is to expand the range of VNFs supported by the orchestrator. In particular, we plan to add the heavy-hitter detection mechanisms studied in our previous project (RP1). In this project, the architecture was built and a future work would be to develop a network function (e.g., a P4 program on a virtual switch or a SmartNIC) that can detect large "elephant" flows in real time, and to integrate it so the orchestrator can deploy it on demand. For example, if an application's traffic pattern indicates that certain flows are consuming a large amount of bandwidth, the orchestrator (through IML) could insert a heavy-hitter monitor NF and possibly a load optimizer function to redistribute or offload those flows to a different path or hardware accelerator.
- **Ability to create own network functions**: So far, the number of available VNFs that the IML can deploy are limited. By designing a way to define and configure any network function, the user can leverage full flexibility to allow any range of network service chains.
- **Local network manager in each node**: The current implementation of IML's CNI requires that this component directly communicates with the IML to obtain the required interface configurations before starting the application. As mentioned in the result section, this can generate around 3 to 4 RTTs of latency when creating a new application instance. However, by implementing a local network manager in each node, this latency can be heavily reduced as the CNI only needs to communicate with another process in the same node, which speeds up the configuration process.
- **Rewrite NFRouter to leverage eBPF acceleration**: Our current solution utilizes software containerization as the deployment method for the NFRouter. This layer of virtualization introduces a minor amount of latency to the network service chains. A better, more refined solution could come from rewriting the NFRouter in eBPF to leverage from both kernel-level speed and also hardware acceleration[23]. This change could be able to reduce the latency considerably while increasing the overall throughput significantly.
- **Multi-Cluster Deployments**: Our demonstration was within a single cluster. In future work, we aim to test the orchestrator across multiple clusters. This will involve scenarios where microservice A is in cluster X and microservice B is in cluster Y, and the NSD might require a chain spanning both. This will be an important step to show that our orchestrator can manage global networks, not just local ones.

References

[1] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott. "Oakestra white paper: An orchestrator for edge computing." arXiv: 2207.01577 [cs.DC]. (2022).

[2] G. Pongrácz and C. Papagianni, "D2.2: DESIRE6G Functional Architecture Definition," Zenodo, Tech. Rep., 2024, Deliverable for the DESIRE6G project. DOI: 10.5281/zenodo.12784579. [Online]. Available: https://doi.org/10.5281/zenodo.12784579.

[3] T. Tao, Y. Wang, D. Li, Y. Wan, P. Baracca, and A. Wang, "6g hyper reliable and low-latency communication – requirement analysis and proof of concept," in *Proceedings of the 2023 IEEE 98th Vehicular Technology Conference (VTC2023-Fall)*, Hong Kong, 2023. [Online]. Available: https://www.ieeevtc.org/vtc2023fall/DATA/2023002457.pdf.

[4] Ericsson. "6g - follow the journey to the next generation networks." Accessed: 2025-05-29. (), [Online]. Available: https://www.ericsson.com/en/6g.

[5] S. R. "Horizontal scaling vs vertical scaling: Choosing your strategy." Accessed: 2025-05-29. (Feb. 2024), [Online]. Available: https://www.digitalocean.com/resources/articles/horizontal-scaling-vs-vertical-scaling.

[6] A. Gurtov, "Dynamic service function chaining orchestration in a multi-domain: A heuristic approach based on srv6," *Sensors*, vol. 21, no. 19, p. 6563, 2021. DOI: 10.3390/s21196563. [Online]. Available: https://www.mdpi.com/1424-8220/21/19/6563.

[7] B. Amento, B. Balasubramanian, R. J. Hall, K. Joshi, G. Jung, and K. H. Purdy, "Focusstack: Orchestrating edge clouds using location-based focus of attention," eng, in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2016, pp. 179–191, ISBN: 150903322X.

[8] "Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge," eng, in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2016, pp. 1–13, ISBN: 150903322X.

[9] A. Alwakeel, A. Alnaim, and E. Fernández, "A pattern for nfv management and orchestration (mano)," Mar. 2019.

[10] K. Kaur, V. Mangat, and K. Saluja, "A review on virtualized infrastructure managers with management and orchestration features in nfv architecture," *Computer Networks*, vol. 217, p. 109 281, Aug. 2022. DOI: 10.1016/j.comnet.2022.109281.

[11] G. M. Yilma, F. Yousaf, V. Sciancalepore, and X. Costa-Pérez, "Benchmarking open-source nfv mano systems: Osm and onap," Mar. 2020. DOI: 10.48550/arXiv.1904.10697.

[12] S. Dräxler, M. Peuster, H. Karl, *et al.*, "Sonata: Service programming and orchestration for virtualized software networks," May 2016. DOI: 10.48550/arXiv.1605.05850.

[13] K. Antevski, C. Bernardos, L. Cominardi, A. de la Oliva, and A. Mourad, "On the integration of nfv and mec technologies: Architecture analysis and benefits for edge robotics," *Computer Networks*, vol. 175, p. 107 274, Apr. 2020. DOI: 10.1016/j.comnet.2020.107274.

[14] B. Sayadi, C.-Y. Chang, C. Tranoris, *et al.*, "Network Applications: Opening up 5G and beyond networks," Zenodo, Tech. Rep., 2022. DOI: 10.5281/zenodo.7123919. [Online]. Available: https://doi.org/10.5281/zenodo.7123919.

[15] OpenStack Foundation. "Network service descriptor management — tacker cli (wallaby)." Accessed June 23, 2025. (Aug. 2020), [Online]. Available: https://docs.openstack.org/tacker/wallaby/cli/cli-legacy-nsd.html.

[16] Oakestra Documentation, *High-Level Setup Overview (Architecture Diagram)*, https://www.oakestra.io/docs/getting-started/oak-environment/high-level-setup-overview/, Accessed July 2025, 2025.

[17] g. marco gramaglia, B. Ömer, X. Li, *et al.*, "Towards 6g architecture: Key concepts, challenges, and building blocks," May 2025. DOI: 10.5281/zenodo.15001377.

[18] T. Agata, *Oakestra: A container orchestrator for embedded edge systems*, https://github.com/tomasagata/oakestra, Accessed: 2025-07-04, 2024.

[19] T. Agata, *Plugin-kubernetes*, https://github.com/tomasagata/plugin-kubernetes, Accessed: 2025-07-04, 2024.

[20] DESIRE6G Consortium, *Iml-lnfvo*, https://github.com/DESIRE6G/IML-LNFVO, Accessed: 2025-07-04, 2024.

[21] A. Name, "Performance and security evaluation of tls, dtls and quic security," M.S. thesis, Politecnico di Torino, 2022.

[22] P. Liu and J. Guitart, "Multi-dimensional resource placement algorithm based on parallel optimization for cloud environments," *Future Generation Computer Systems*, 2025, Integrates island-model genetic algorithms to address the NP-hard resource placement problem. DOI: 10.1016/j.future.2025.01.023.

[23] eBPF Community, *Ebpf – introduction, tutorials & community resources*, https://ebpf.io/, Accessed: 2025-07-06, 2025.