Department of Information Engineering and
Computer Science

Master's degree in Computer Science

Softwarized and Virtualized mobile networks

Network slice setup optimization

Authors	Student ID
Dennis Cattoni	256139
Marco Lasagna	256134
Andrea Eugenio Cesaretti	258431

Academic year 2025/2026

Contents

1 Introduction

This project explores the design and implementation of a **Software Defined Networking** (SDN) [1] based system for QoS-aware service slicing. The main objective is to dynamically optimize network behavior according to heterogeneous service requirements, such as low latency and high throughput, by leveraging a centralized SDN controller with a global view of the network [1].

The proposed architecture combines SDN with principles inspired by **Network Function Virtualization** (NFV) [2] and **Multi-access Edge Computing** (MEC) [3]. In particular, network services are deployed as virtualized functions running on multiple backend nodes, while the SDN controller dynamically directs traffic toward the most appropriate service instance based on current network conditions and **Quality of Services** (QoS) [4] constraints.

By decoupling the control plane from the data plane, the SDN controller is able to perform centralized monitoring, path computation and traffic engineering, enforcing slice-specific policies in real time [1]. The service placement and migration decisions are handled through a management plane by a dedicated Service Orchestrator [5], enabling dynamic relocation of services between edge and remote nodes in response to network congestion or failures.

The system demonstrates how SDN-based traffic control, combined with virtualized services at the network edge, can provide adaptive QoS guarantees. Although implemented in a controlled emulation environment, the proposed design reflects realistic scenarios in modern programmable networks, such as edge-assisted content delivery, adaptive service chaining and differentiated service slicing.

1.1 Experimental Environment

The experimental setup is based on **Mininet** [6] and **Comnetsemu** [7], which allow the emulation of programmable networks and containerized services. These tools enable controlled experimentation with network topologies, traffic patterns and service placement.

The data plane of the emulated network is implemented using **Open vSwitch** (OVS) [8], which acts as the software switch for all network nodes. OVS supports the OpenFlow protocol and is responsible for enforcing forwarding decisions, queue management and traffic isolation as instructed by the controller.

Communication between the controller and the switches relies on *OpenFlow v1.3* [9]. This version is selected due to its widespread adoption and its support for advanced features such as multiple flow tables, flow cookies and detailed port statistics. These capabilities are essential for implementing slice-specific forwarding rules, monitoring link utilization and reacting to network events such as **congestion** and **link failures**.

By combining Mininet, Comnetsemu, Open vSwitch, and OpenFlow 1.3, the environment provides a realistic and flexible platform for prototyping and evaluating SDN-based network slicing and QoS optimization strategies.

1.2 Workspace Organization

The project workspace is structured according to a clear separation between **control plane**, **data plane**, and **management plane** functionalities. This organization reflects the architectural principles of SDN-based systems and facilitates modular development, experimentation and debugging.

The `controller/` directory contains the core logic of the SDN control plane. It hosts the main controller application responsible for enforcing QoS policies, performing slice-specific path computation, monitoring network conditions, and reacting to topology changes. Configuration files define slice parameters and static link properties, while dedicated modules maintain the network state and implement routing and optimization algorithms. Utility components support the construction and installation of OpenFlow rules, enabling a modular and extensible controller design.

The `app/` directory contains higher-level application and management components that operate above the SDN controller. This includes the implementation of the virtualized network service, client-side traffic generators, TLS certificate generation scripts, and a lightweight REST-based service manager used to control service placement across backend nodes.

The `emulation/` directory includes scripts and definitions used to instantiate the emulated network environment. It provides the network topology description and startup procedures for the SDN controller, the virtualized hosts, and the management services. Runtime logs generated during execution are collected in the `logs/` directory, which stores controller logs and traffic measurement outputs produced during experiments and demonstrations.

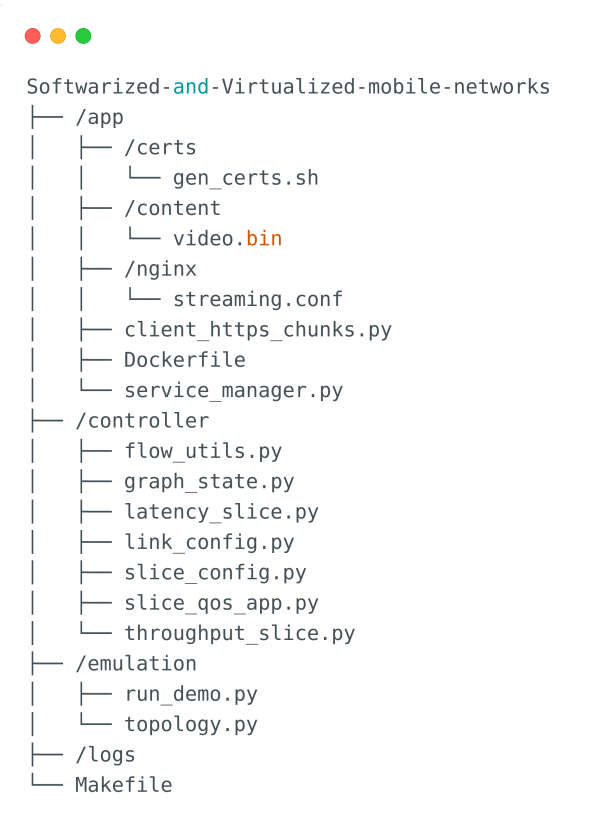

Figure 1: Project file structure

1.2.1 Northbound Components

Northbound functionalities are implemented through configuration modules that define service behavior and QoS objectives at a high level. In particular, `slice_config.py` specifies slice parameters and service requirements, such as latency bounds and minimum throughput constraints, without exposing any device-specific or protocol-dependent logic. These definitions represent the abstract service intent that the network is expected to satisfy.

Slice definitions include traffic identification rules, optimization objectives, QoS thresholds, ingress and egress points, and runtime state variables updated by the controller. This design allows the control logic to remain generic while supporting multiple heterogeneous slices with different performance goals.

1.2.2 Management Plane and Application Components

Management plane functionalities are implemented in the `app/` directory and support service lifecycle control independently of the SDN control logic. These components extend the northbound interaction model by enabling high-level service management operations without exposing any data plane or switch-level details.

The file `service_manager.py` implements a lightweight REST API that allows starting, stopping, and migrating a virtualized HTTPS service between backend nodes. This component operates outside the SDN controller and represents a simplified NFV management plane, which can be triggered by the controller as part of its decision-making process but remains logically decoupled from the control plane.

The virtualized service is implemented using a containerized web server configured for HTTPS streaming, with static on-demand content baked into the container image at build time. TLS certificates are generated using the `gen_certs.sh` script and embedded into the service image, ensuring secure communication independently of runtime configuration. Client-side traffic generation and performance evaluation are handled by `client_https_chunks.py`, which issues HTTPS requests toward a virtual IP managed by the SDN controller.

The logical separation between control plane, data plane and management plane enables experimentation with service migration, QoS enforcement and failure recovery, while keeping each subsystem conceptually isolated. Moreover, this modular design has the goal to improve clarity, maintainability and alignment with real world SDN/NFV architectures.

1.2.3 Control Plane

The core control logic of the system is implemented within the SDN controller and is responsible for translating high-level service intent into concrete forwarding decisions. The file `slice_qos_app.py` acts as the main controller application and is the only component loaded by the SDN controller runtime. It integrates slice definitions with runtime network information to perform path computation, QoS monitoring, and reconfiguration.

The controller maintains a global view of the network through the `graph_state.py` module, which represents the topology and tracks per-link performance metrics using raw measurements and exponentially weighted moving averages. A dual-tier monitoring

mechanism is implemented to balance reactivity and scalability.

Slice-specific control logic is contained in dedicated modules, namely `latency_slice.py` and `throughput_slice.py`. These components implement independent decision processes for low-latency and high-throughput slices, respectively, while sharing the same underlying network state. This separation allows different optimization criteria and recovery strategies to coexist within a single controller and shared network resources.

1.2.4 Southbound Components

Southbound functionalities are realized by modules responsible for enforcing control decisions on the network infrastructure and interacting directly with OpenFlow-enabled switches. The `flow_utils.py` module provides helper functions for constructing and installing OpenFlow rules, including priority handling and timeout configuration.

Static topology information and port-to-neighbor mappings are defined in `link_config.py`, enabling the controller to translate logical paths into concrete forwarding actions without relying on dynamic topology discovery protocols. Through these components, the controller installs forwarding rules, manages queue selection, and collects runtime statistics from the data plane using OpenFlow port statistics messages.

2 Network Topology Configuration

2.1 Topology Description

As shown in Figure 2, the network topology includes client hosts `C1` and `C2`, server hosts `srv1` and `srv2`, and a set of interconnected switches `S1`–`S5`.

The topology is statically defined and known *a priori* by the controller, allowing control decisions to be computed based on a complete and consistent view of the network.

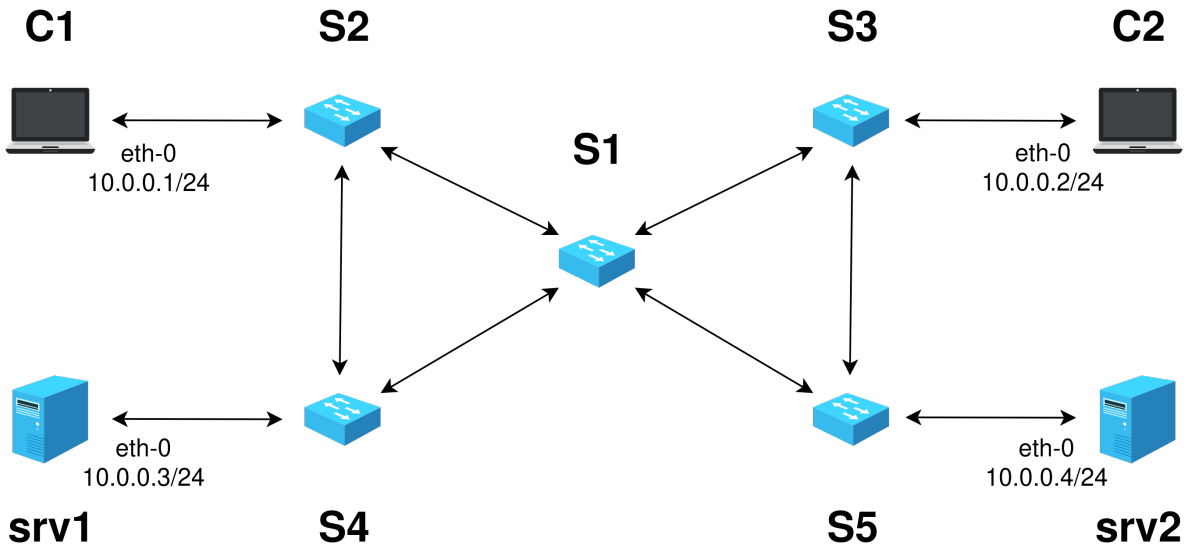

Figure 2: Network topology

Although the topology adopts a relatively simple and controlled structure, it is intentionally designed to capture key characteristics of realistic network deployments. In particular, the presence of multiple paths and routing loops introduces redundancy and alternative forwarding options, enabling the evaluation of control strategies under non-trivial connectivity conditions. These design choices also allow the inclusion of border cases, such as congestion points and path selection trade-offs, which are essential for assessing the robustness of QoS-aware optimization mechanisms.

At the center of the topology, switch `S1` acts as a critical aggregation point, concentrating traffic flows from different segments of the network. This central role makes `S1` a natural bottleneck and a strategic control point for enforcing slicing policies and evaluating the impact of resource allocation decisions across multiple services.

Overall, although the topology is intentionally simple in size, it is designed to support the simulation and evaluation of slice-specific optimization strategies. The combination of static topology knowledge, limited but meaningful path diversity and traffic convergence at central points, allows the controller behavior to be observed under different QoS requirements and network's state. This setup provides a controlled environment in which the effectiveness of the SDN controller in enforcing QoS policies across multiple slices can be analyzed.

2.2 Emulated Topology in Mininet

The network topology is emulated using *Mininet* and is designed to reproduce a multi-path core network with edge-connected clients and service backends. The topology includes end hosts, OpenFlow switches, and containerized service nodes, interconnected through links with explicitly configured bandwidth, propagation delay and packet loss.

Client hosts `c1` and `c2`, as well as backend servers `srv1` and `srv2`, are connected to the network via dedicated edge switches defined as *Host-to-switch*. The corresponding access links are left unconstrained in terms of bandwidth, delay and packet loss because their purpose is to provide transparent access to the core network. This choice ensures that any performance bottleneck is introduced exclusively within the controlled core topology rather than at the network edge.

As discussed in Section 2.1, the core of the topology is formed by a central aggregation switch connected to multiple edge switches through links with heterogeneous characteristics. These links differ in terms of capacity, latency and loss probability, reflecting realistic network conditions and enabling the controller to make non-trivial routing decisions. Additional *cross-links* between edge switches provide alternative paths, allowing the evaluation of path diversity, failure recovery and QoS-driven rerouting.

Table 1 summarizes the parameters assigned to each link in the topology, where *Host-to-switch* links are marked with "–" to indicate that no traffic control constraints are applied.

	Link ID	Endpoints	Bandwidth	Prop. delay	Packet loss
Host-to-switch	L1	c1 – s2	–	–	–
	L2	c2 – s3	–	–	–
	L3	srv1 – s4	–	–	–
	L4	srv2 – s5	–	–	–
Core-links	L5	s2 – s1	12 Mbps	5 ms	0.01 %
	L6	s3 – s1	12 Mbps	5 ms	0.01 %
	L7	s4 – s1	4 Mbps	3 ms	0.05 %
	L8	s5 – s1	7 Mbps	10 ms	0.01 %
Cross-links	L9	s2 – s4	10 Mbps	5 ms	0.02 %
	L10	s3 – s5	7 Mbps	5 ms	0.02 %

Table 1: Link characteristics of Mininet topology

The different link capacities and loss rates are intentionally chosen to create distinct trade-offs between latency and throughput. In particular, links connected to `s4` exhibit lower capacity and higher loss, making them less suitable for high-throughput services but potentially attractive for low-latency paths under certain conditions. Conversely, alternative paths through `s2` and `s3` provide higher bandwidth but may incur additional delay.

This topology design allows the SDN controller to demonstrate differentiated slice behavior, QoS monitoring and the dynamic adaptation to congestion and failure events.

3 Technologies Used and Services

This section describes the technologies adopted for traffic generation, service delivery and secure communication within the proposed SDN-based system. Particular attention is given to the role of client applications, the characteristics of the generated traffic and the mechanisms used to ensure secure service access.

3.1 Client Applications and Traffic Models

Two distinct client applications are employed to generate traffic within the emulated network, each associated with a specific network slice and Quality of Service objective. The clients differ both in the nature of the traffic they generate and the application-layer technologies they use, enabling the evaluation of heterogeneous service requirements over a shared SDN-controlled infrastructure:

- **Low-latency client:** the client host `c2` is associated with the low-latency slice and is used to generate delay-sensitive traffic. This slice models applications where responsiveness is the primary requirement rather than sustained throughput. Traffic generation is implemented using *iperf3* [10], with `srv1` acting as the corresponding server endpoint. The use of *iperf3* allows precise control over traffic patterns and facilitates the measurement of latency-related performance metrics, while keeping application-layer complexity to a minimum. In this slice, the focus is placed exclusively on SDN-based path selection and forwarding decisions that minimize end-to-end delay.
- **High-throughput client:** the client host `c1` is associated with the high-throughput slice and models an application-oriented traffic pattern based on secure content delivery. Unlike the low-latency slice, this client interacts with a virtualized service [2] exposed through a stable Virtual IP and is therefore tightly coupled with the SDN-based NAT and service migration mechanisms. The custom client application `client_https_chunks.py` performs HTTPS requests toward the service endpoint and retrieves an on-demand resource using chunked transfers. This choice aims to emulate realistic application behavior, where data is transferred over long-lived secure connections and throughput is the dominant performance metric.

3.2 Secure Communication and TLS Configuration

Secure communication between the high-throughput client and the backend service is ensured through the use of HTTPS [11]. **Transport Layer Security** (TLS) [12] is an essential component of the system, as it reflects realistic deployment scenarios where application traffic is encrypted end-to-end and network-level optimization must operate beneath the security layer.

The system adopts *TLS 1.3* as the transport security protocol for all HTTPS communications, implemented using the *OpenSSL* library [13]. *TLS 1.3* was selected for its improved security guarantees compared to earlier protocol versions. Asymmetric cryptography is used during the authentication and key exchange phase of the TLS handshake, while symmetric encryption is employed for protecting application data during the session.

In this project, server authentication is implemented using the *Ed25519* elliptic-curve signature scheme [14]. *Ed25519* provides strong security guarantees with significantly lower computational overhead compared to traditional RSA-based approaches, making it well suited for environments where performance and efficiency are critical. Once the secure session is established, *TLS 1.3* negotiates an authenticated encryption cipher suite, based on *AES-256 in Galois/Counter Mode* (AES-256-GCM) [15], ensuring confidentiality and integrity of the transmitted data.

TLS self-signed certificates are generated offline using a dedicated script (`gen_certs.sh`) and embedded into the service container image at build time. The certificates include Subject Alternative Name entries for the Virtual IP, ensuring correct validation regardless of the backend node currently hosting the service.

From the client perspective, all aspects of service migration and network reconfiguration remain completely invisible. The client always interacts with the same virtual endpoint using standard HTTPS semantics, without perceiving changes in service location, routing paths or underlying network conditions.

3.3 Software Defined Networking

The control plane is implemented using the **Ryu** SDN framework [16], which provides a Python platform for developing OpenFlow controllers. By maintaining a global view of the network state and by directly programming forwarding rules on the switches through OpenFlow, the SDN controller can react to runtime events such as congestion, link failures and service migration. This centralized control model is fundamental to implementing heterogeneous service requirements within a shared infrastructure.

3.4 SDN-based NAT with Virtual IP and VMAC

To decouple service identity from its physical deployment, the system implements an SDN-based **Network Address Translation** (NAT) [17] mechanism combined with **Virtual IP** (VIP) [18] and **Virtual MAC address** (VMAC) [19]. The VIP represents the stable network endpoint through which clients access the service, while the VMAC is used to anchor Layer 2 of the ISO-OSI model forwarding behavior within the SDN-controlled domain. Both identifiers remain constant regardless of the backend node currently hosting the service.

Within the data plane, the SDN controller programs OpenFlow rules that perform transparent header rewriting at the network edge. In the forward direction, incoming client traffic destined to the VIP is subjected to **Destination NAT** (DNAT) [17], where the destination IP and MAC addresses are rewritten to match the selected backend server. In the reverse direction, response traffic generated by the backend undergoes **Source NAT** (SNAT) [17], restoring the VIP and VMAC before packets are forwarded back to the client. This bidirectional translation ensures that clients perceive the service as a single, stable endpoint.

This approach allows to integrate address translation into the SDN control logic. The controller determines when and where NAT rules must be installed based on slice-specific routing decisions and service placement, allowing service migration to be performed by simply updating flow rules in the network. From the client perspective, service migration

is completely invisible: ongoing sessions continue to target the same VIP without any awareness of backend relocation. No changes are required at the client side, and no DNS updates or connection reconfiguration are needed.

The VIP and VMAC abstraction plays a central role in enabling seamless service migration and mobility-aware scenarios. When a service instance is relocated (e.g., from a remote backend to a closer edge node), the controller reconfigures the forwarding and NAT rules accordingly, preserving the service identity and minimizing disruption. This mechanism aligns with principles found in Network Function Virtualization and Multi-access Edge Computing, where service continuity and location transparency are fundamental requirements.

3.5 NetworkX for Topology Management

NetworkX [20], is a Python library designed for the creation, manipulation, and analysis of networks. In this project, NetworkX is employed to represent the SDN controlled infrastructure as a graph abstraction, where switches are modeled as nodes and network links are modeled as **directed** edges.

The graph representation enables the controller to maintain a global and consistent view of the network topology, including link attributes such as propagation delay, capacity, and estimated utilization. These attributes are dynamically updated based on runtime statistics collected from the data plane and are stored as edge weights within the graph.

NetworkX provides a set of efficient algorithms for shortest path computation and graph traversal, which are leveraged to determine optimal forwarding paths for different network slices. By associating slice-specific cost functions with graph edges, the controller can compute paths that minimize latency, maximize throughput, or satisfy other QoS objectives.

Within the project, NetworkX is not used for topology discovery, which is handled separately, but rather as a decision support tool that translates the controller's global network knowledge into concrete routing decisions. This approach allows the separation of topology modeling from control logic, improving modularity and simplifying the implementation of advanced optimization strategies.

3.6 Network Function Virtualization

Network Function Virtualization (NFV) [2] is leveraged in this project to decouple the service implementation from the underlying physical (or emulated) infrastructure. Rather than binding the application endpoint to a fixed host, the service is instantiated as a virtualized network function that can be activated on different backend nodes according to control-plane decisions and runtime conditions.

In the proposed architecture, the virtualized function corresponds to an HTTPS streaming service delivered through an NGINX web server. The service is designed to provide **on-demand content streaming**, and its placement can be dynamically shifted between multiple backend nodes. This enables controlled experimentation with service migration, availability recovery and different network conditions when combined with the SDN-based VIP/VMAC abstraction implemented in the data plane.

3.6.1 Backend containerization with Docker

Service virtualization is implemented using **Docker containers** [21], selected for their fast instantiation time, lightweight execution model and broad compatibility across different environments. In addition, container images make it straightforward to package the service together with its runtime dependencies, configuration files, and static content in a deterministic way.

Within the emulated environment, the backend nodes `srv1` and `srv2` are implemented as DockerHost instances in Mininet. Each backend is capable of running the same containerized NGINX [22] service and the streaming resource and the NGINX HTTPS configuration are baked into the container image at build time.

The system enforces the policy that **only one backend service is active at a time**. Service migration is therefore modeled as an activation switch between two candidate execution points: when the service is started on one backend, it is stopped on the other.

This NFV service lifecycle management, combined with SDN-controlled VIP/VMAC NAT, enables service relocation without changing the client's endpoint, thereby emulating a realistic scenario where network functions can be moved across the network infrastructure while preserving service continuity.

3.7 Multi-access Edge Computing

Multi-access Edge Computing (MEC) [3] is a paradigm that extends cloud computing capabilities toward the edge of the network, enabling services to be executed closer to end users. By reducing the physical and logical distance between clients and service instances, MEC aims to improve performance metrics such as latency, responsiveness and bandwidth efficiency, while also enabling context-aware and mobility-oriented applications.

In this project's purposes, MEC principles are used through the dynamic placement of a virtualized service across multiple backend nodes that are located at different positions within the network topology. The backend nodes `srv1` and `srv2` represent alternative execution points with distinct network characteristics: one can be interpreted as a **closer edge node**, while the other models a more **remote backend node**. The SDN controller continuously monitors network conditions and enforces slice-specific QoS objectives to decide which backend is more suitable for hosting the service at a given time.

Service migration between backend nodes is triggered by control-plane decisions and is executed transparently with respect to the client, thanks to the SDN-based VIP/VMAC abstraction. From the client's perspective, the service endpoint remains unchanged, while the actual execution location of the service shifts dynamically within the network. This behavior closely reflects MEC scenarios, where services can follow users or adapt to network dynamics without requiring application-level reconfiguration.

3.8 QoS Monitoring with Dual-Tier Round-Robin

QoS enforcement within the SDN controller relies on a periodic monitoring mechanism that continuously collects runtime statistics from the data plane and updates the internal network state used for routing and reconfiguration decisions. This functionality is implemented through a dedicated monitoring thread, which operates independently from

OpenFlow event handlers and executes in a cyclic manner.

Each monitoring cycle, referred to as a tick, is executed at a fixed time interval $\Delta T =$ `MONITOR_INTERVAL_S`. During each tick, the controller issues OpenFlow *PortStatsRequest* messages to a selected set of switches, uniquely identified by their datapath identifiers (DPIDs). For every polled switch, statistics are collected for all its ports, allowing the controller to update utilization estimates for all outgoing directed links associated with that DPID.

Let S denote the set of all switches in the topology and let $A_t \subseteq S$ be the set of switches that belong to at least one active slice path at monitoring tick t. Active paths may correspond to the low-latency slice, the high-throughput slice, or both. The remaining switches form the set:

$$B_t = S \setminus A_t \tag{1}$$

To balance monitoring responsiveness and control-plane overhead, the controller adopts a dual-tier polling strategy, structured as follows:

- **Tier A (active-path monitoring):** all switches whose DPIDs belong to A_t are polled at every tick. This guarantees that links currently carrying slice traffic are monitored with maximum frequency. If a switch belongs to multiple active paths, it is polled only once per tick. For each such switch, statistics from all ports are processed to update utilization metrics for the corresponding directed links.
- **Tier B (background round-robin monitoring):** a bounded number $K =$ `BACKGROUND_POLL_PER_TICK` of switches is selected from B_t according to a persistent round-robin ordering. The round-robin pointer advances across monitoring cycles, ensuring fair and systematic coverage of all non-active switches over time. As in Tier A, polling a switch outside the active paths collects statistics from all its ports.

Assuming that the set of active switches remains stable over time, the maximum number of monitoring ticks required to poll all non-active switches at least once is given by:

$$T_{\text{cover}} = \left\lceil \frac{|B|}{K} \right\rceil \tag{2}$$

which corresponds to a maximum refresh interval:

$$Max\ \Delta_{\text{cover}} = T_{\text{cover}} \cdot \Delta T \tag{3}$$

This bound ensures that utilization metrics on links outside the active paths are periodically refreshed, preventing stale values from influencing future routing decisions.

By construction, the two tiers operate on disjoint sets of switches, such that:

$$A_t \cap B_t = \varnothing \tag{4}$$

This guarantees that no DPID is polled more than once within the same tick, keeping the monitoring load bounded and avoiding redundant statistics requests.

After port statistics have been processed and link utilization estimates updated, slice-specific QoS evaluation routines are executed. The low-latency slice evaluates delay-oriented cost functions, while the high-throughput slice evaluates bottleneck residual capacity along candidate paths. In this way, the monitoring mechanism maintains a fresh and consistent view of the network state while preserving scalability as the topology size increases.

4 Network Slicing Model

4.1 Slice Definitions

The system supports multiple slices with different QoS requirements. Each slice is associated with a specific service and policy. Slices are uniquely identified within the data plane through a slice-specific identifier embedded in the flow rules, allowing traffic to be consistently classified and managed according to the corresponding QoS constraints across the network.

From the control-plane perspective, each slice is characterized by a set of qualitative QoS objectives that guide resource allocation and forwarding decisions. These objectives capture the relative sensitivity of the service to different performance metrics and are used by the controller to differentiate traffic handling across slices.

Slice ID	Service type	Latency	Throughput	Queue priority
1	Low Latency	High	–	High
2	High Throughput	–	High	Medium

Table 2: Slice-to-QoS objective mapping

Table 2 summarizes the qualitative QoS objectives associated with each slice. The values reported in the latency and throughput columns represent the **sensitivity** of each slice to the corresponding metric rather than absolute performance levels. For instance, the low-latency slice exhibits a high sensitivity to delay, while the high-throughput slice prioritizes bandwidth availability.

4.2 Low Latency Slice

The low latency slice is designed to minimize **one-way data plane delay** between a source host and a destination host within the SDN domain. Its primary objective is to ensure that latency sensitive traffic is forwarded along paths that provide the lowest possible estimated delay while respecting a predefined QoS constraint.

Traffic belonging to this slice is explicitly identified through a set of match fields based on Layer 3 and Layer 4 headers, including source and destination IPv4 addresses, transport protocol, and destination TCP port. This approach allows the controller to precisely classify latency sensitive flows and to apply slice specific forwarding policies without interfering with other traffic classes.

The QoS objective associated with this slice is defined as one-way latency minimization. A maximum acceptable end-to-end latency threshold is specified in the configuration and represents a logical bound on the estimated path cost. The controller continuously evaluates the latency of the currently installed path by aggregating per link delay contributions and dynamically monitored congestion metrics. When the estimated latency exceeds the configured threshold, the controller triggers a path recomputation in an attempt to restore compliance with the QoS requirement.

To prioritize latency sensitive traffic at the data plane level, packets belonging to this slice are mapped to a dedicated queue with **higher scheduling priority**. This mechanism reduces queueing delay under contention and helps preserve low latency behavior even in the presence of competing traffic from other slices.

The slice is defined at the switch level through explicit ingress and egress points. The ingress switch corresponds to the first SDN switch receiving traffic from the source host, while the egress switch represents the last switch before the destination host. This abstraction allows the controller to compute and enforce forwarding paths entirely within the SDN-controlled domain, independently of host specific details.

For evaluation purposes, the slice is exercised using TCP traffic generated by *iperf3*. This workload enables controlled saturation of selected links and provides a realistic scenario for observing latency variations, congestion effects, and the controller's ability to react to QoS violations. Although TCP is inherently bidirectional, the optimization focuses on minimizing the one way delay of the data path, which is the dominant contributor to perceived latency in latency-sensitive applications.

4.3 High Throughput Slice

The high throughput slice is designed to maximize the **sustained data rate** achievable between a client host and a virtualized service endpoint within the SDN domain. Unlike the low latency slice, whose objective is to minimize delay, this slice targets applications where throughput efficiency and bandwidth utilization are the dominant performance metrics.

Traffic belonging to this slice is identified through a dedicated set of match fields at the network edge, including the Virtual IP address associated with the service and the corresponding transport-layer parameters. By matching on the VIP rather than on a specific backend address, the controller is able to apply slice-specific forwarding policies independently of the physical location of the service instance, enabling transparent service migration.

The QoS objective for this slice is defined in terms of a minimum acceptable throughput. This requirement is expressed as a target bandwidth value that the selected path should be able to sustain based on current network conditions. To enforce this objective, the controller continuously monitors link utilization and estimates the residual capacity available along the active path. Path selection is formulated as a **widest-path** problem [23], where the chosen route maximizes the bottleneck residual capacity across all candidate paths between ingress and egress switches.

At runtime, the controller evaluates whether the currently installed path satisfies the QoS throughput constraint. If the estimated bottleneck capacity falls below the required threshold, the controller attempts to restore compliance by recomputing the widest avail-

able path. When no feasible path toward the currently active backend can satisfy the requirement, the controller may trigger a service migration toward an alternative and closer backend edge node, thereby combining network-level optimization with service-level relocation and new path selection.

To support differentiated handling at the data plane, traffic associated with the high throughput slice is mapped to a queue with lower priority than the low latency slice. This design choice deliberately prioritizes latency-sensitive traffic, as the low latency slice does not rely on service migration mechanisms and must therefore be protected through preferential access to network resources. This scheduling policy ensures fair coexistence between slices while allowing throughput-oriented flows to efficiently exploit available bandwidth without starving latency-sensitive traffic.

The slice is defined through explicit ingress and egress switches, where the ingress corresponds to the first SDN switch receiving client traffic and the egress is associated with the switch connected to the **currently active** backend server. This abstraction allows the controller to compute forwarding paths entirely within the SDN-controlled core while dynamically adapting both routing and service placement decisions.

For evaluation, the slice is exercised using an application-level HTTPS workload generated by a custom client. Traffic consists of large on-demand content transfers over secure connections, providing a realistic scenario for assessing throughput behavior, congestion adaptation, and the interaction between SDN-based routing and service migration.

5 Optimization Algorithms

5.1 Topology Model

The network topology is statically defined and known *a priori* by the controller, allowing control decisions to be computed based on a complete and consistent view of the network. The controller maintains an internal representation of the topology that reflects the expected network structure and link characteristics.

The topology is modeled as a **directed graph**, where each switch is represented as a node and each physical link is represented by two directed edges, one for each direction of transmission. This modeling choice is necessary to accurately capture the behavior of the underlying data plane and the information exposed by the SDN control interface.

In OpenFlow-based networks, traffic statistics and state information are collected on a per-port basis and are inherently directional. In particular, port statistics report the number of bytes and packets transmitted through a given port, corresponding to traffic flowing in a specific direction. As a result, link utilization, congestion, and queueing behavior can differ between the two directions of the same physical link. Modeling the topology as an undirected graph would therefore obscure these asymmetries and prevent an accurate representation of the network state.

Furthermore, the routing and QoS optimization performed by the controller is direction dependent. The low latency slice, for example, optimizes the one way data plane delay from an ingress switch to an egress switch. Since congestion and delay contributions may vary across directions, each directed edge is associated with its own set of attributes, including propagation delay, capacity, and dynamically estimated utilization. These at-

tributes are used to compute slice specific path costs and to select forwarding paths that best satisfy the QoS objectives.

The directed graph abstraction also aligns with the controller's interaction with the data plane. Flow rules, queue assignments and forwarding actions are installed per switch and per output port, reinforcing the notion that forwarding decisions are applied directionally. By adopting a directed topology model, the controller can consistently map high level optimization results to concrete OpenFlow rules without loss of information or semantic mismatch.

5.2 Low Latency Oriented Path Computation

The computation of low latency oriented paths is performed in two distinct phases: an initial **bootstrap** phase and a subsequent **operational** phase in which paths are dynamically adapted to network conditions. This separation allows the controller to start from a consistent baseline configuration and to react to relevant network events without introducing unnecessary instability.

During the bootstrap phase, the controller initializes its internal state and waits for all expected switches to establish a valid OpenFlow connection. Although the network topology is statically defined and known *a priori*, forwarding rules cannot be installed until the corresponding datapaths are available. The bootstrap phase is considered complete only when all configured switches are registered and the controller is able to interact with every node involved in the low latency slice. At this point, the controller computes the initial path between the ingress and egress switches of the slice. This first computation is performed in the absence of significant traffic and relies solely on static topology parameters, such as link propagation delays and nominal capacities. The resulting path is then installed in the data plane through dedicated OpenFlow rules.

After the bootstrap phase, the system enters the operational phase, where the controller continuously maintains the low latency path according to the current network state. In this phase, path recomputation is not performed periodically, but is instead triggered by specific events that may compromise the pre-defined QoS. The controller **monitors** the network through asynchronous notifications, such as link failure events, and through periodic collection of port's statistics from the switches belonging to the current **active path**.

Collected statistics are used to estimate link utilization and to update the corresponding edge attributes in the internal graph representation. Based on these updated metrics, the controller evaluates the cost of the currently installed path using a **latency-oriented cost function** that captures and models both static and dynamic characteristics of the network. If the estimated path cost exceeds the maximum acceptable latency configured for the slice, a QoS violation is detected and a new path computation is triggered.

Path recomputation is therefore activated in two main situations:

- link failure directly affects the current path;
- the estimated latency of the path violates the QoS constraint.

To ensure robustness in scenarios where network failures temporarily eliminate all feasible paths, the controller incorporates an explicit **disconnected state** for the low latency

slice. If a path recomputation attempt fails due to the absence of any viable route between the ingress and egress switches, the controller marks the slice as disconnected and removes the previously installed forwarding rules. This prevents stale rules from directing traffic toward unreachable next hops and avoids packet blackholing.

While the controller adopts a conservative policy that avoids path recomputation upon link restoration events, an exception is introduced to guarantee recovery. Specifically, when a link transitions back to the *UP* state, a new path computation is triggered only if the slice is currently in the disconnected state. This mechanism allows the system to automatically restore connectivity as soon as a feasible path becomes available, while still preventing unnecessary routing churn during transient link fluctuations.

In both cases, the controller computes a new optimal path using the updated network view. If a path satisfying the latency requirement exists, it is installed immediately. Otherwise, the controller installs the best available path while explicitly reporting that the QoS objective cannot be met under the current network conditions.

5.2.1 Latency-Oriented Cost Function

The latency-oriented cost function constitutes the core of the optimization algorithm used for the low latency slice. Its objective is to estimate the end-to-end delay experienced by data plane traffic and to guide routing decisions accordingly.

The network is modeled as a simmetric **directed graph** [24] $G = (V, E)$, where V is the set of switches and E is the set of directed links between them. Given a source node $s \in V$ and a destination node $d \in V$, the controller selects the path that minimizes the cumulative latency cost between the two nodes. Formally, the optimal path P^* is defined as:

$$P^* = \arg\min_{P: s \rightsquigarrow d} W_t(P) \tag{5}$$

where $W_t(P)$ denotes the total latency-oriented cost of path P at time t.

Given a directed path $P = (v_0, v_1, \ldots, v_n)$, where $v_0 = s$ and $v_n = d$, and where each consecutive pair (v_i, v_{i+1}) corresponds to a directed edge in E, the path cost is computed as the sum of the costs associated with each link at time t:

$$W_t(P) = \sum_{i=0}^{n-1} w_t(v_i, v_{i+1}) \tag{6}$$

Each directed edge $(u, v) \in E$ is associated with a time-dependent cost $w_t(u, v)$ that reflects both static link properties and dynamic network conditions. The edge cost function is designed to capture the components of packet's delay relevant to routing decisions and is defined as:

$$w_t(u, v) = d(u, v) + \alpha \frac{\rho_t(u, v)}{1 - \rho_t(u, v) + \varepsilon} + p \tag{7}$$

The term $d(u, v)$ represents the static **propagation delay** of the link, derived from its physical characteristics and assumed to be time invariant. The second term models the **congestion-dependent delay**, where $\rho_t(u, v) \in [0, 1)$ denotes the estimated utilization

of the link at time t. This component increases non-linearly as utilization approaches the link capacity, reflecting the growing impact of queuing delays under congestion. The constant α controls the relative influence of congestion with respect to propagation delay, while ε is a small positive constant introduced to ensure numerical stability when $\rho_t(u,v)$ approaches the value one. Finally, the constant term p accounts for **per-hop processing delays** incurred within network devices. Since OpenFlow does not provide primitives to directly monitor such delays, this term is assumed to be constant across all nodes, ensuring fairness among paths and discouraging the selection of routes with an excessive number of hops.

The link utilization $\rho_t(u,v)$ is defined as the ratio between the estimated traffic rate on the link and its nominal capacity:

$$\rho_t(u,v) = \frac{\widehat{x}_t(u,v)}{C(u,v)} \tag{8}$$

where $C(u,v)$ denotes the nominal capacity of link (u,v), and $\widehat{x}_t(u,v)$ represents the estimated traffic rate at time t. The instantaneous traffic rate $x_t(u,v)$ is computed from port statistics as:

$$x_t(u,v) = \frac{\Delta B_t(u,v) \cdot 8}{\Delta t \cdot 10^6} \tag{9}$$

where $\Delta B_t(u,v)$ is the variation in the number of transmitted bytes observed over the interval Δt, giving a rate expressed in *Mbps*. To reduce the impact of short-term fluctuations and measurement noise, the traffic rate estimate $\widehat{x}_t(u,v)$ is obtained using an **Exponentially Weighted Moving Average** (EWMA) [25]:

$$\widehat{x}_t(u,v) = \beta \cdot x_t(u,v) + (1-\beta) \cdot \widehat{x}_{t-1}(u,v) \tag{10}$$

where $\beta \in (0,1]$ controls the trade-off between responsiveness and smoothing. This filtering mechanism allows the controller to react to persistent changes in traffic conditions while avoiding unnecessary path oscillations due to transient variations.

The dynamic components of the cost function are updated using runtime statistics collected from the switches, allowing the controller to adjust routing decisions in response to changing network conditions while maintaining stability. With this formulation, the low latency slice can be treated as a time-dependent shortest path problem on a directed graph, solvable using classical graph algorithms while incorporating real-time network measurements.

5.3 Throughput Oriented Optimization and Service Migration

The high throughput slice is designed to maximize the effective data rate experienced by application traffic while guaranteeing a minimum throughput requirement. In contrast to the low latency slice, this slice explicitly integrates network-aware routing with service placement decisions, enabling dynamic service migration as part of the optimization process.

As for the low latency slice, the operation of the throughput slice is divided into an initial **bootstrap phase** and a subsequent **operational phase**. During the bootstrap phase,

the controller waits for all expected switches to establish an OpenFlow connection and for the static topology to become fully available. Once the network is ready, the controller computes an initial forwarding path toward the default service backend, referred to as the `remote backend`, and installs the corresponding forwarding and SDN-based NAT rules. At this stage, since the network has just been initiated, path computation relies primarily on static link capacities and the absence of significant traffic.

After bootstrap, the system enters the operational phase, where the controller continuously evaluates the quality of the currently installed path according to throughput-oriented QoS metrics. Monitoring focuses on switches belonging to the **active path**, from which port statistics are periodically collected. These statistics are used to estimate link utilization and residual capacity, allowing the controller to assess whether the current path can still satisfy the minimum throughput threshold required by the QoS policy of the slice.

Path recomputation in the throughput slice is triggered only under specific conditions that may compromise service performance. In particular, recomputation is initiated when:

- a link failure directly affects the current path;
- the estimated bottleneck throughput of the path falls below the minimum requirement.

When a throughput degradation is detected, the controller first attempts to recompute the best available path toward the currently active backend service. If a feasible path satisfying the throughput constraint exists, it is installed immediately without modifying service placement.

If no path toward the active backend can satisfy the throughput requirement, the controller triggers a **service migration procedure**. Service migration consists in relocating the execution of the virtualized service from the remote backend to an alternative backend, typically hosted on an edge node that is geographically closer to the client within the network. This operation is performed through a dedicated management plane interface and remains logically decoupled from the SDN control logic.

Following a successful migration, the controller recomputes the path toward the new backend and installs updated forwarding and NAT rules. The service remains exposed through the same Virtual IP and Virtual MAC, ensuring client-agnostic service migration.

When the slice is operating on the remote backend and a link failure eliminates all feasible paths, the controller immediately triggers service migration toward the closest backend in an attempt to preserve connectivity. Conversely, when the slice is already operating on the closest backend and no feasible path exists, the controller first attempts a failback to the remote backend, prioritizing service availability even if the resulting path violates the specified QoS constraints, as best-effort connectivity is preferred over service interruption. The slice transitions to the **disconnected** state only if no feasible path can be found toward either backend, indicating a complete loss of network connectivity.

Specifically, when a link transitions back to the *UP* state, a new path computation is triggered **only if** the slice is currently in the disconnected state. This recovery policy ensures that path recomputation is performed exclusively when connectivity can potentially be restored.

Through this combined optimization strategy, the high throughput slice demonstrates how SDN-based routing and NFV-style service management can be jointly orchestrated. The controller performs centralized decision making based on a global view of the network state, while monitoring and enforcement remain distributed across the data plane, enabling adaptive and efficient service delivery under dynamic network conditions.

5.3.1 Throughput-Oriented Cost Function

The throughput-oriented cost function, together with service migration, constitutes the core of the optimization algorithm used for the high throughput slice. Its objective is to maximize the achievable end-to-end throughput, ensuring that traffic is routed along paths that provide the highest sustainable throughput under current network conditions.

The network is modeled as a **directed graph** [24] $G = (V, E)$, where V is the set of switches and E is the set of directed links. Given a source node $s \in V$ and a destination node $d \in V$, the controller selects the path that maximizes the available bottleneck capacity along the path. Formally, the optimal path P^* is defined as:

$$P^* = \arg\max_{P: s \rightsquigarrow d} B_t(P) \tag{11}$$

where $B_t(P)$ denotes the bottleneck throughput of path P at time t.

Given a directed path $P = (v_0, v_1, \ldots, v_n)$, where $v_0 = s$ and $v_n = d$, the bottleneck throughput is defined as the minimum residual capacity among all links composing the path:

$$B_t(P) = \min_{i=0,\ldots,n-1} r_t(v_i, v_{i+1}) \tag{12}$$

Each directed edge $(u, v) \in E$ is associated with a time-dependent **residual capacity** $r_t(u, v)$, which represents the portion of link capacity that is still available for additional traffic. The residual capacity is computed as:

$$r_t(u, v) = C(u, v) - \widehat{x}_t(u, v) \tag{13}$$

where $C(u, v)$ denotes the nominal capacity of the link (u, v) and $\widehat{x}_t(u, v)$ is the estimated traffic rate currently observed on that link.

The instantaneous traffic rate $x_t(u, v)$ is derived from port statistics collected from the switches:

$$x_t(u, v) = \frac{\Delta B_t(u, v) \cdot 8}{\Delta t \cdot 10^6} \tag{14}$$

where $\Delta B_t(u, v)$ is the variation in transmitted bytes over the observation interval Δt, giving a rate expressed in *Mbps*. To mitigate the impact of measurement noise and transient fluctuations, the controller computes a smoothed estimate of the traffic rate using an **Exponentially Weighted Moving Average** (EWMA) [25]:

$$\widehat{x}_t(u, v) = \beta \cdot x_t(u, v) + (1 - \beta) \cdot \widehat{x}_{t-1}(u, v) \tag{15}$$

where $\beta \in (0, 1]$ controls the trade-off between responsiveness to traffic changes and stability of the estimate.

The residual capacity formulation assigns a bottleneck value to each directed edge in the graph. Path computation is then performed by solving a **widest path problem** [23], where the objective is to maximize the minimum residual capacity along the path as shown in Equation 11.

From an algorithmic perspective, this problem is addressed using a modified version of *Dijkstra's algorithm*, commonly referred to as a *max–min* variant. Unlike the classical shortest path formulation, where path costs are accumulated through summation, this variant propagates path values by iteratively taking the minimum residual capacity over the edges traversed so far and selecting the path that maximizes this minimum value:

Algorithm 1 Widest Path (*Max–Min* Dijkstra variant)

Input: Directed graph $G = (V, E)$, source s, destination d
Output: Path P^* maximizing the minimum residual capacity

Data structures:
$cap[v]$: best known bottleneck value from s to node v
$prev[v]$: predecessor of v on the current best path
Q: max-priority queue of nodes keyed by $cap[\cdot]$
$inQ[v]$: boolean flag, true iff node v is still in Q (not finalized)

for all $v \in V$ **do**
    $cap[v] \leftarrow 0$
    $prev[v] \leftarrow$ NULL
    $inQ[v] \leftarrow$ true
end for
$cap[s] \leftarrow +\infty$
Initialize Q with all nodes $v \in V$ keyed by $cap[v]$
while $Q \neq \emptyset$ **do**
    $u \leftarrow \textsc{ExtractMax}(Q)$
    $inQ[u] \leftarrow$ false
    if $u = d$ **then**
        break
    end if
    for all $v \in Adj[u]$ **do**
        if $inQ[v]$ **then**
            $hopRes \leftarrow \min\big(residual(u, v),\ residual(v, u)\big)$
            $c \leftarrow \min\big(cap[u],\ hopRes\big)$
            if $c > cap[v]$ **then**
                $cap[v] \leftarrow c$
                $prev[v] \leftarrow u$
                $\textsc{IncreaseKey}(Q, v, cap[v])$
            end if
        end if
    end for
end while
return reconstruct path from $prev[\cdot]$

In case multiple paths achieve the same maximum bottleneck value, ties are first resolved by selecting the path with the minimum number of hops. If both bottleneck value and hop count are equal, a deterministic lexicographic ordering over the node sequence is used. This policy is not explicitly represented in the pseudocode above and is introduced as a project-specific choice to guarantee a deterministic behavior.

This approach ensures that the selected path is limited as little as possible by bottleneck links and is therefore well suited to throughput-oriented optimization. Applying this algorithm to the directed graph with dynamically updated residual capacities makes it possible to efficiently identify the path that offers the highest sustainable throughput under current network conditions. The use of dynamically updated residual capacities allows the controller to adapt routing decisions to evolving traffic conditions, supporting high throughput delivery while avoiding persistent bottlenecks.

6 How to run and Use Cases

6.1 How to run

This section describes how to run the system and test its functionalities, including traffic generation, monitoring procedures, and failure injection.

The first step consists in installing and correctly setting up the *ComNetsEmu* environment. It is strongly recommended to use the Vagrant-based installation, as suggested by the official ComNetsEmu documentation itself. Running ComNetsEmu inside a virtual machine ensures compatibility and avoids dependency-related issues on the host operating system.

After installing Vagrant and VirtualBox, the project repository should be cloned into the shared folder between the host operating system and the virtual machine, located at `/comnetsemu`. From a terminal opened in this directory, the virtual machine can be started using the command `vagrant up comnetsemu`. Once the VM is running, a shell can be obtained by connecting via SSH with `vagrant ssh comnetsemu`.

Before proceeding, it is necessary to verify that the Python library `NetworkX` is correctly installed inside the virtual machine, as it is required by the SDN controller for topology modeling and path computation. If the library is missing, it can be installed by following the official NetworkX installation instructions [26].

After logging into the virtual machine, move to the project directory inside the ComNetsEmu workspace. In particular, if the previous instructions have been followed correctly, move to the directory `/comnetsemu/Softwarized-and-Virtualized-mobile-networks`. It is recommended to use Tmux [27] to manage multiple terminal sessions simultaneously. Start it by running the command `tmux` from the terminal, and then open three separate panes, which will be used as follows:

- **Terminal 1 (T1):** used to run the Ryu SDN controller and observe controller logs.
- **Terminal 2 (T2):** used to start the Mininet-based emulation environment, including switches, links, hosts, and Docker-based backends.
- **Terminal 3 (T3):** used to interact with the application layer, verify the status of

the streaming service, and manually trigger service migration through the REST API.

6.2 Use Case Scenarios

Several use case scenarios are presented to illustrate the behavior of the system under different network conditions and configuration choices, with the goal of validating the proposed QoS-aware control logic and service orchestration mechanisms.

The experimental environment can be configured prior to execution by enabling or disabling individual network slices through dedicated feature flags defined in the controller configuration files. In particular, the **low latency slice** and the **high throughput slice** can be activated independently, allowing the system to be evaluated either in isolation or under concurrent multi-slice operation. This modular configuration enables focused analysis of each slice without interference from unrelated control policies. The corresponding feature flags are `ENABLE_LATENCY_SLICE` and `ENABLE_THROUGHPUT_SLICE`, both defined in the file `slice_qos_app.py`.

In addition to slice selection, the system provides a control flag `migration_enabled`, defined in `slice_config.py`, which determines whether **automatic service migration** is enabled for the high-throughput slice. When migration is enabled, the controller is allowed to relocate the virtualized service between backend nodes in response to QoS degradation or loss of connectivity. Conversely, disabling migration forces the controller to rely exclusively on path recomputation and routing optimization, even when the slice remains active. This option is particularly useful for isolating and analyzing the behavior of the throughput-oriented path computation algorithm without introducing service-level dynamics.

Table 3 summarizes the possible configuration combinations considered for experimental evaluation:

Scenario ID	Description	`Latency`	`Throughput`	`Migration`
1	Low-latency only	True	False	–
2	High-throughput (no mig.)	False	True	False
3	High-throughput (mig.)	False	True	True
4	Dual slice (no mig.)	True	True	False
5	Dual slice (mig.)	True	True	True

Table 3: Experimental scenarios and controller configuration flags

The considered scenarios include controlled congestion injection, link failure and recovery events, and service migration triggers. By selectively combining slice activation and migration policies, the experiments highlight the interaction between monitoring, optimization, and orchestration components, as well as the trade-offs between QoS compliance, service availability, and control-plane stability.

6.3 Useful commands

The project includes a **Makefile** that encapsulates the full execution workflow and the auxiliary commands required to reproduce the experiments. For clarity, the available rules are grouped according to their purpose and to the slice they are associated with. As introduced in the previous subsection, the execution is typically carried out using three *Tmux* panes: **T1** for the SDN controller logs, **T2** for the emulation runtime and Mininet CLI, and **T3** for management-plane inspection and manual service operations:

- **SDN controller (T1):**
  - The SDN controller is started using the command `make ryu`, executed from terminal **T1**.
  - This rule runs `ryu-manager controller/slice_qos_app.py`, which is the only Ryu application loaded in the experiment.
  - Since this command is blocking and continuously produces logs (monitoring ticks, link measurements, QoS events, recomputations, and migration triggers), terminal **T1** is typically reserved exclusively for observing the controller behavior, while all interactive commands are issued from **T2** and **T3**.
- **Common setup and demo startup (T2):**
  - The default target `make` is executed from terminal **T2** and performs the complete bootstrap of the demo environment.
  - First, it ensures that the on-demand payload `app/content/video.bin` exists, generating a 20 MiB file if missing.
  - Then, it generates TLS certificates by invoking `app/certs/gen_certs.sh`.
  - Finally, it builds the Docker image `stream_vnf:latest` and starts the emulated environment through `emulation/run_demo.py`, which instantiates the Mininet topology and launches the service containers.
  - The `make clean` rule resets the environment by removing Mininet state, stopping service containers, and deleting generated artifacts such as payloads and certificates.
- **Low-latency slice rules (T2):**
  - The low-latency slice is exercised using *iperf3* traffic between the client host `c2` and the backend server `srv1`.
  - The rule `make server` starts the *iperf3* server inside the `srv1` container on TCP port 5001 and must be executed before generating any client traffic.
  - The rule `make pr` generates a sustained multi-stream workload from `c2` toward `10.0.0.3:5001`, running for 120 seconds with multiple parallel streams and redirecting output to `logs/iperf3_srv1.log`.
  - Both commands are typically launched from the Mininet CLI in terminal **T2** after initializing the topology.
- **High-throughput slice rules (T2 and T3):**

- The high-throughput slice is exercised using a dedicated application-level HTTPS client together with management-plane controls for service placement.
- The rule `make r`, executed from terminal **T2**, issues an HTTPS request from the client host `c1` toward the stable Virtual IP representing the service endpoint.
- Upon receiving the request, the backend node currently hosting the virtualized service (`srv1` or `srv2`) delivers the on-demand streaming content to `c1` using chunked transfers.
- Manual service migration and status inspection are executed from terminal **T3** and interact with the management-plane REST API exposed locally on `127.0.0.1:9090`.
- The rule `make s` queries the current service placement, while `make 1` and `make 2` force migration to `srv1` and `srv2`, respectively.
- These commands allow explicit validation of service migration behavior and enable controlled experimentation even when automatic migration is disabled at the controller level.

Some commands must be executed from within the Mininet CLI and therefore require explicitly specifying the target host in order to avoid ambiguity between the emulated nodes and the underlying virtual machine. For this reason, Table 4 summarizes the correct syntax for executing the rules defined in the `Makefile`, clearly indicating the appropriate terminal and, when required, the Mininet host from which each command must be issued:

Description	**Command**	**Terminal**
Start SDN controller (Ryu)	`make ryu`	T1
Start Mininet topology and NFV environment	`make`	T2
Start *iperf3* server for low-latency slice	`sh make server`	T2
Pervasive low-latency traffic (iperf3 client)	`c2 make pr`	T2
Request on-demand HTTPS resource	`c1 make r`	T2
Manually migrate service to `srv1`	`make 1`	T3
Manually migrate service to `srv2`	`make 2`	T3
Check current service backend status	`make s`	T3

Table 4: Execution commands and terminal usage

To properly terminate an experiment and clean up the environment, the Mininet CLI should be exited by typing `exit` in terminal **T2**. Afterwards, from the virtual machine prompt, the command `make clean` can be used to remove the Mininet topology, stop running Docker containers, and delete temporary files generated during execution.

Finally, to shut down the virtual machine managed by Vagrant, open a terminal on the host machine, navigate to the `/comnetsemu` directory, and run the command `vagrant halt comnetsemu`.

7 Results

The proposed control model optimizes Quality of Service at the **Network level** by relying exclusively on link and path-level statistics collected from the SDN data plane. In particular, control decisions are based on link utilization estimates derived from OpenFlow port counters, which provide link-level traffic measurements independently of transport- or application-layer behavior. Although these counters are implemented at the data link layer, they are abstracted by the controller as network-level observables for traffic engineering purposes. Path selection and QoS enforcement are therefore driven solely by these network-level metrics.

In contrast, from an end-host perspective, the log values reported by the client applications `c1` and `c2` represent **application-level** performance metrics, capturing the effective goodput perceived by the end host. These measurements inherently depend on transport-layer mechanisms such as: TCP congestion control, loss recovery, and round-trip time dynamics, as well as application-level buffering and timing effects. As a consequence, application-level goodput may differ from the residual capacity estimated at the network layer, even in the absence of network-level QoS violations.

This distinction highlights that the proposed model implements a **network-centric QoS** approach, which aims to guarantee sufficient network capacity for a given service, but does not directly optimize end-to-end application **Quality of Experience** (QoE) [28]. Accordingly, discrepancies between network-level guarantees and application-level observations are an expected outcome of the layered design of the system, particularly under induced congestion conditions.

7.1 Observed system behavior

This section reports the behavior observed during the experimental evaluation of the system under the different use case scenarios described in Section 6.2. For clarity, each scenario is referenced throughout the discussion by its corresponding Scenario ID reported in Table 3. The observations are performed during system executions and they are based on controller logs, application-level measurements, and runtime events.

The experimental observations are based on a set of recurring criteria that are consistently monitored across all scenarios. In particular, the following aspects are observed:

- the forwarding paths selected by the controller for each slice and their evolution over time;
- compliance with slice-specific QoS objectives, in terms of latency bounds for the low-latency slice and minimum throughput for the high-throughput slice;
- the system's reactions to injected congestion and link failure events;
- the triggering of service migration events when migration is enabled;

These criteria provide a structured view of the system behavior at both the control-plane and data-plane levels and serve as a common reference for all scenarios.

7.1.1 Single-Slice Behavior

The first set of experiments evaluates the behavior of each slice in isolation, allowing the effects of individual optimization policies to be observed without interference from other slices.

Low-Latency Slice Behavior (Scenario 1) When only the low-latency slice is active, the controller consistently selects the path with the minimum estimated latency cost between the ingress and egress switches. The selected path remains stable during periods of low load and is updated only when the monitored latency exceeds the configured threshold or when a link failure directly affects the active path.

Upon congestion injection, the controller detects the increase in estimated path latency through periodic monitoring and triggers a path recomputation. If an alternative path with lower estimated latency exists, it is installed immediately. In the presence of link failures, the controller recomputes the path if possible, or transitions the slice into a disconnected state when no feasible path exists. No service migration events occur in this scenario, as migration is not part of the low-latency slice policy.

Overall, the observed behavior is characterized by limited path changes and stable forwarding decisions, driven exclusively by latency-related QoS conditions.

High-Throughput Slice without Migration (Scenario 2) When the just the high-throughput slice is active without service migration enabled, the controller selects forwarding paths according to the widest-path, maximizing the minimum residual capacity along the path. The selected path may change in response to residual capacity or link failures, but all adaptations are performed exclusively through routing recomputation.

Under sustained load, application-level measurements show variations in the achieved throughput, while the controller continues to operate based on network-level statistics derived from port counters. In this configuration, no service relocation occurs, and the backend hosting the service remains fixed throughout the experiment.

The observed behavior highlights the reliance on path recomputation alone to react to changing network conditions, with no additional mechanisms to restore throughput when all feasible paths offer limited residual capacity.

High-Throughput Slice with Migration (Scenario 3) When service migration is enabled for the high-throughput slice, additional events are observed during execution. In particular, when the controller detects that no feasible path toward the currently active backend can satisfy the connectivity or QoS requirements, it triggers a service migration toward the alternative backend.

Compared to Scenario 2, this configuration introduces explicit migration events, observable through controller logs and management-plane interactions. After migration, the controller recomputes forwarding paths toward the new backend and resumes service delivery. The sequence of events differs from the non-migrating case, as routing adaptation is complemented by service relocation based on current the network state.

7.1.2 Multi-Slice Interaction

The final set of experiments evaluates the behavior of the system when both slices are active simultaneously, highlighting interactions between independent optimization policies.

Dual-Slice Operation without Migration (Scenario 4) When both slices are active and service migration is disabled, the controller enforces slice-specific policies concurrently. Each slice maintains its own forwarding path, selected according to its respective optimization objective. Shared links are observed in the topology, and traffic from different slices may traverse common segments of the network.

Despite resource sharing, the controller maintains separate monitoring and decision processes for each slice. Path recomputations are performed independently per slice and do not directly affect the operation of other slices, while no service migration events are triggered in this scenario. The observed behavior demonstrates the stable coexistence of multiple slices, where the controller preserves QoS by selecting and recomputing the optimal path for each slice independently based on current network conditions.

Dual-Slice Operation with Migration (Scenario 5) When both slices are active and service migration is enabled for the high-throughput slice, additional interactions are observed. Migration events may occur while the low-latency slice remains active, leading to dynamic changes in resource usage and path selection for the high-throughput traffic.

In this configuration, the controller continues to prioritize the latency constraints of the low-latency slice, while allowing the high-throughput slice to first relocate its service to preserve throughput and connectivity. The observed behavior includes migration-triggered path changes and transient shifts in link utilization, reflecting competition for shared network resources.

This scenario highlights the combined effects of multi-slice operation and service migration, illustrating how control decisions taken for one slice can indirectly influence the operating conditions of another. Given the static characteristics of the Mininet topology, when the network becomes stressed by a significant volume of traffic, the enforcement of QoS for the high-throughput slice converges toward relocating the NFV service to a closer edge node. At the same time, the low-latency slice consistently adapts its path selection to satisfy its delay constraints by choosing the lowest-cost path available under the current network conditions.

8 Limitations

8.1 Implementation Constraints

The experimental environment is based on *ComNetsEmu*, which explicitly restricts its support to a specific operating system version. At the time of development, only **Ubuntu 20.04 LTS** is officially supported. As documented by the framework maintainers, the virtual machine should not be upgraded internally, since the environment is still considered in a beta stage and is designed to prioritize dependency compatibility over software freshness.

As a consequence, several system libraries and Python dependencies are not aligned with the most recent upstream releases. This constraint directly influenced some design decisions made during the development of the project. In particular, the emulation of the high-throughput slice relies on HTTPS traffic using *HTTP/2* over *TLS 1.3*, rather than adopting more recent transport protocols such as **QUIC** [29]. While QUIC is widely used by modern on-demand streaming services due to its reduced latency and improved congestion handling, its deployment would have required a more up-to-date software stack not readily available in the target environment.

8.2 Scalability Considerations

Scalability represents an inherent challenge in Software Defined Networking architectures. In an SDN-based system, each new flow may require interaction with the controller, and the controller itself is responsible for maintaining a logically centralized view of the global network state. Even when multiple controllers are deployed in a distributed fashion, maintaining global consistency across control instances becomes increasingly complex, and the volume of control-plane traffic can grow faster than the data-plane traffic it manages.

Within this context, the proposed dual-tier round-robin monitoring mechanism is designed to strike a balance between monitoring accuracy and scalability. Continuously monitoring all switches at a high frequency would provide the freshest possible network global state, but would incur a control-plane cost that grows with the size and complexity of the topology. To mitigate this issue, the monitoring subsystem prioritizes active polling of switches that belong to the current slice paths, while applying a round-robin strategy to progressively refresh statistics from non-active switches. This approach prevents metric staleness and starvation, while keeping monitoring overhead bounded.

However, this design introduces an inherent limitation. As formalized by the worst-case update interval derived in Equation (3), the time required to refresh statistics for all non-active switches increases with the network size. Even with careful tuning of the parameter `BACKGROUND_POLL_PER_TICK`, a trade-off remains between metric freshness and control-plane cost. In massive large-scale topologies, this may result in delayed visibility of metrics developing outside the currently active paths, potentially impacting the optimality of routing decisions.

8.3 Security Considerations

The primary objective of this project is the optimization of Quality of Service for heterogeneous network slices, rather than the design of a security-hardened production system. As such, the development process did not explicitly follow a **SecDevOps** methodology [30], and security was not treated as a first-class design dimension.

Despite this, several design choices were made with realism in mind. The use of *TLS 1.3* for the high-throughput slice, combined with *Ed25519* for certificate key generation and *AES-256-GCM* for symmetric encryption, reflects cryptographic primitives that are widely adopted in real-world deployments.

Beyond the scope of this work, a comprehensive security-oriented revision of the system would be required for deployment in adversarial environments. Such a revision would include a systematic code audit, employing **Static analysis** [31], **Dynamic analysis** [32], and **Manual analysis** techniques, as well as a threat model addressing control-plane attacks, data-plane manipulation, and management-plane exposure. Even if aware of these aspects, they were intentionally left out of the current implementation, as they fall outside the objectives of the project.

9 Conclusions and Future Work

The proposed control model, as observed during the experimental evaluation across the different use case scenarios described in Subsection 7.1, exhibits behavior consistent with the expected design objectives. The Mininet topology employed in the experiments is characterized by heterogeneous link capacities and delays, which effectively capture both individual and concurrent traffic optimization dynamics across multiple slices. In particular, under multi-slice operation with service migration enabled, the static characteristics of the topology naturally lead the high-throughput slice to converge toward relocating the NFV service from a remote edge node to a closer one when the network traffic increases. Conversely, the low-latency slice consistently enforces its QoS requirements through dynamic path recomputation, selecting the lowest-cost path available according to current network conditions. These behaviors reflect the intended separation of control objectives and confirm the correctness of the proposed network-centric QoS model.

It is important to note that the presented solution represents a model driven design, developed and implemented based on the authors' networking knowledge, prior experience, and the abstractions and capabilities offered by the SDN paradigm. As discussed in Section 8, this approach inherently introduces several limitations.

Future works may include a more extensive evaluation of the proposed monitoring strategy in large scale network topologies with a higher number of switches and hosts, in order to assess scalability and control-plane overhead. Furthermore, the adoption of the QUIC protocol for on-demand streaming services could be explored, subject to future support within the *ComNetsEmu* environment, in order to more accurately capture modern transport-layer behavior.

In conclusion, the current implementation leaves ample room for a comprehensive security audit and for the integration of security mechanisms within the overall system design. Incorporating security-aware control policies and threat mitigation strategies represents a natural and valuable direction for future developing.

References

[1] en.wikipedia.org. *Software-defined networking*. https://en.wikipedia.org/wiki/Software-defined_networking.

[2] en.wikipedia.org. *Network function virtualization*. https://en.wikipedia.org/wiki/Network_function_virtualization.

[3] en.wikipedia.org. *Multi-access edge computing*. https://en.wikipedia.org/wiki/Multi-access_edge_computing.

[4] en.wikipedia.org. *Quality of service*. https://en.wikipedia.org/wiki/Quality_of_service.

[5] Leonardo. *The Orchestrator Service: The Unsung Hero of Your Overcomplicated Software*. https://leo88.medium.com/the-orchestrator-service-the-unsung-hero-of-your-overcomplicated-software-f027af77f1a7.

[6] mininet.org. *Mininet An Instant Virtual Network on your Laptop (or other PC)*. https://mininet.org/.

[7] comnetsemu. *comnetsemu*. https://git.comnets.net/public-repo/comnetsemu.

[8] Linux Foundation. *Open vSwitch*. https://www.openvswitch.org/.

[9] Open networking Foundation. *OpenFlow Switch Specification*. https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf.

[10] https://iperf.fr/. *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. https://iperf.fr/iperf-download.php.

[11] en.wikipedia.org. *HTTPS*. https://en.wikipedia.org/wiki/HTTPS.

[12] en.wikipedia.org. *Transport Layer Security*. https://en.wikipedia.org/wiki/Transport_Layer_Security.

[13] en.wikipedia.org. *OpenSSL library*. https://openssl-library.org/.

[14] en.wikipedia.org. *EdDSA*. https://en.wikipedia.org/wiki/EdDSA.

[15] en.wikipedia.org. *Advanced Encryption Standard*. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[16] Ryu SDN Framework Community. *Ryu SDN Framework*. https://ryu-sdn.org/index.html.

[17] en.wikipedia.org. *Network address translation*. https://en.wikipedia.org/wiki/Network_address_translation.

[18] en.wikipedia.org. *Virtual IP address*. https://en.wikipedia.org/wiki/Virtual_IP_address.

[19] rfwireless-world.com. *Virtual vs. Physical MAC Addresses Explained*. https://www.rfwireless-world.com/terminology/virtual-vs-physical-mac-addresses.

[20] NetworkX developers. *NetworkX Network Analysis in Python*. https://networkx.org/en/.

[21] Docker. *What is a container?* https://www.docker.com/resources/what-container/.

[22] https://nginx.org. *nginx*. https://nginx.org/en/.

[23] en.wikipedia.org. *Widest path problem*. https://en.wikipedia.org/wiki/Widest_path_problem.

[24] en.wikipedia.org. *Directed graph*. https://en.wikipedia.org/wiki/Directed_graph.

[25] en.wikipedia.org. *Exponential moving average*. https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average.

[26] NetworkX Developers. *Install*. https://networkx.org/documentation/stable/install.html.

[27] Nicholas Marriott. *Install*. https://github.com/tmux/tmux/wiki.

[28] intergence.com. *Quality of Experience vs Quality of Service*. https://www.intergence.com/blog/quality-of-experience-vs-quality-of-service.

[29] en.wikipedia.org. *QUIC*. https://en.wikipedia.org/wiki/QUIC.

[30] owasp.org. *OWASP DevSecOps Guideline*. https://owasp.org/www-project-devsecops-guideline/.

[31] owasp.org. *Static Code Analysis*. https://owasp.org/www-community/controls/Static_Code_Analysis.

[32] en.wikipedia.org. *Dynamic application security testing*. https://en.wikipedia.org/wiki/Dynamic_application_security_testing.