CHALMERS
UNIVERSITY OF TECHNOLOGY

A Flexible and Scalable ML-Based Diagnosis Module for Optical Networks: A Security Use Case

Citation for the original published paper (version of record):
Natalino Da Silva, C., Gifre, L., Moreno-Muro, F. et al (2023). A Flexible and Scalable ML-Based Diagnosis Module for Optical Networks: A Security Use Case. Journal of Optical Communications and Networking, 15(8): C155-C165. http://dx.doi.org/10.1364/JOCN.482932

JOURNAL OF
Optical Communications
and Networking

Flexible and scalable ML-based diagnosis module for optical networks: a security use case

Carlos Natalino,[1,*] Lluis Gifre,[2] Francisco-Javier Moreno-Muro,[3] Sergio Gonzalez-Diaz,[3] Ricard Vilalta,[2] Raul Muñoz,[2] Paolo Monti,[1] and Marija Furdek[1]

[1]Electrical Engineering Department, Chalmers University of Technology, Gothenburg, Sweden
[2]Centre Tecnològic de Telecomunicacions de Catalunya (CTTC/CERCA), Castelldefels, Spain
[3]ATOS, Madrid, Spain
*carlos.natalino@chalmers.se

Received 6 December 2022; revised 10 February 2023; accepted 28 February 2023; published 21 June 2023

To support the pervasive digital evolution, optical network infrastructures must be able to quickly and effectively adapt to changes arising from traffic dynamicity or external factors such as faults and attacks. Network automation is crucial for enabling dynamic, scalable, resource-efficient, and trustworthy network operations. Novel telemetry solutions enable optical network management systems to obtain fine-grained monitoring data from devices and channels as the first step toward the near-real-time diagnosis of anomalies such as security threats and soft failures. However, the collection of large amounts of data creates a scalability challenge related to processing the data within the desired monitoring cycle regardless of the number of optical services being analyzed. This paper proposes a module that leverages the cloud native software deployment approach to achieve near-real-time machine learning (ML)-assisted diagnosis of optical channels. The results obtained over an emulated physical-layer security scenario demonstrate that the architecture successfully scales the necessary components according to the computational load and consistently achieves the desired monitoring cycle duration over a varying number of monitored optical channels.

https://doi.org/10.1364/JOCN.482932

1. INTRODUCTION

Optical networks are the backbone of today's information society. They support critical services and offer the necessary capacity to transfer massive amounts of traffic at low latency and relatively low energy consumption. Every new generation of networks, notably the 5G and Beyond 5G (B5G) paradigms, exacerbates network complexity and emphasizes the need for dynamic operation. These paradigms also increase traffic demands and tighten concerns about the efficiency of network resource usage. The encompassing digital evolution that they endorse requires trustworthy, intelligent networking solutions and technologies that increase the security, privacy, resilience, and performance of networked systems. Consequently, the optical network infrastructure must be able to quickly and effectively adapt to the changes stemming from either intrinsic traffic characteristics and requirements or external factors such as hardware and software component faults, environmental effects, or harmful man-made actions.

Automation of network operation is crucial for coping with increasing network complexity and efficiently addressing the myriad of the aforementioned interrelated challenges [1]. Network automation, typically comprising a loop with three main phases [2], as shown in Fig. 1, is fueled by advances in several key areas. One example is network telemetry [3,4], which enables the timely and efficient collection of optical performance monitoring (OPM) data from network devices. Another example is the proliferation of machine learning (ML) techniques that provide operators with a data-driven approach for automating their operations [1,5]. Another major enabler of network automation is the development of cloud-native software-defined networking (SDN) controllers to replace legacy, monolithic SDN software architectures. A cloud-native architecture consists of stateless micro-services, each implementing a specific set of functionalities and interacting with others to fulfill network management tasks. This enables efficient scaling of an SDN controller with the massive amount of flow management operations expected for 5G and B5G networks [6], which cannot be consistently handled by current SDN controller solutions, such as ONOS or OpenDayLight [7].

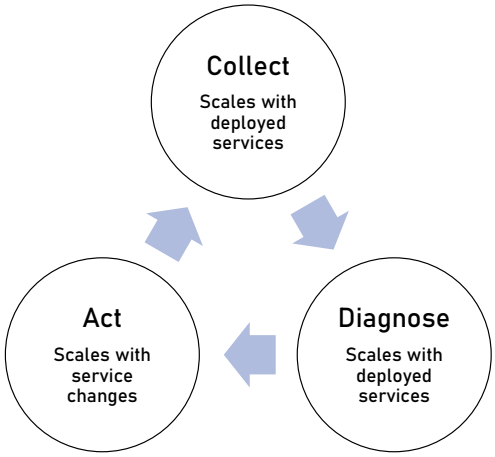


Fig. 1. Network automation loop and its scalability properties.

Figure 1 also indicates the scalability properties of network automation phases. The *Collect* phase consolidates the status/performance data from optical devices into a monitoring database. These operations are usually performed by a module within the SDN controller, which, via network telemetry, collects data from a multitude of devices in the network. The *Diagnose* phase analyzes the monitoring data to assess the performance of optical services and to identify possible anomalies (e.g., security breaches, attacks, bottlenecks, soft/hard failures). These operations are performed by a module that, depending on the specific control/management architecture, is part of or interfaces with the SDN controller to retrieve monitoring data and to deliver the results from data-processing operations. The monitoring data needs to be collected and processed periodically for each running service. As a result, the complexity of the *Collect* and the *Diagnose* phases is proportional to the number of active optical services. The *Act* phase relies on a module within the SDN controller to decide on performing changes in the network based on the feedback provided by the *Diagnose* module, called only upon detection of events that require action. The *Act* phase may also perform changes in the network triggered by external factors, e.g., the need for establishing a new optical service. Consequently, the *Collect* and *Diagnose* modules require dedicated efforts to ensure their operations scale efficiently with the number of optical services.

A large body of literature addresses the scalability and flexibility challenges related to optical network monitoring, investigating, for example, the choice of functionalities that should be implemented [8], their location [3], and scaling with the number of optical services [9–11]. However, the analogous challenges pertinent to the data diagnoses operations remain largely unaddressed. The works in this area concentrate primarily on methods to detect and classify anomalies, e.g., cognitive fault detection and management [12,13], network equipment failure prediction [14], and dynamic planning and optimization of software-defined networks [5,15]. Among the available tools, ML-based tools have been shown to achieve promising results over the past few years. For instance, ML-based anomaly detection has been shown to excel at early detection of soft failures [12,16] as well as physical-layer security threats [17,18]. These advances must be accompanied by qualifications of the design and necessary functionalities of a diagnosis module. It is also crucial to devise solutions that allow the data processing and anomaly detection functionalities to be interfaced with the remaining SDN controller operations in a scalable manner. Finally, this new design should also allow for new (ML-based) diagnosis functionalities and methods to be added/upgraded without requiring the redeployment of the entire module.

This paper addresses the challenge of designing a scalable and flexible software module to diagnose optical services. To the best of our knowledge, the works in [9,10] were the first to tackle the design of such a module. In this work, we further extend our prior work by refining the module previously proposed, detailing the specification of each component, specifying its interface and requirements toward the SDN controller, and by assessing the performance of an implementation of the module using a real-world SDN controller. The module comprises four components: manager, worker, inference, and cache. In addition to detailing their main features, the paper also explains how these four components communicate with each other and with the SDN controller used to manage the optical network. To evaluate its performance in a real setting, the proposed diagnosis module is integrated into the microservice-based SDN controller ETSI TeraFlowSDN [6] and deployed over an emulated optical data plane replaying data related to attacks at the physical layer [17]. The scalability of the proposed module is demonstrated by monitoring a few to several hundred optical services. The resource efficiency is assessed by observing how the number of resources reserved by the solution adjusts to the number of optical services in the network. The flexibility of the proposed module is demonstrated by adopting unsupervised learning (UL) and supervised learning (SL) models for anomaly detection and classification, respectively.

The remainder of the paper is organized as follows. Section 2 introduces the proposed module, detailing the communication among the components and between the module and the SDN controller. Section 3 introduces an implementation of the module integrated into the ETSI TeraFlowSDN controller. It also presents the validation experiments performed over the implementation. Section 4 concludes the paper.

2. ML-BASED DIAGNOSIS FOR OPTICAL NETWORKS

This section first describes the envisioned module and the communication among the proposed components. Then, a detailed specification of each component is provided. Finally, the interface between the module and a standard SDN controller is defined. Without loss of generality, we adopt the terminology of the core components defined for the TeraFlowSDN controller [6].

Figure 2 illustrates the envisioned ML-based diagnosis module together with its interfaces to the SDN controller. In the considered scenario, the SDN controller receives service requests from external entities through its northbound interface (NBI). The SDN controller components interact internally to fulfill the received service requests. An important step of service provisioning is to (re)configure devices through

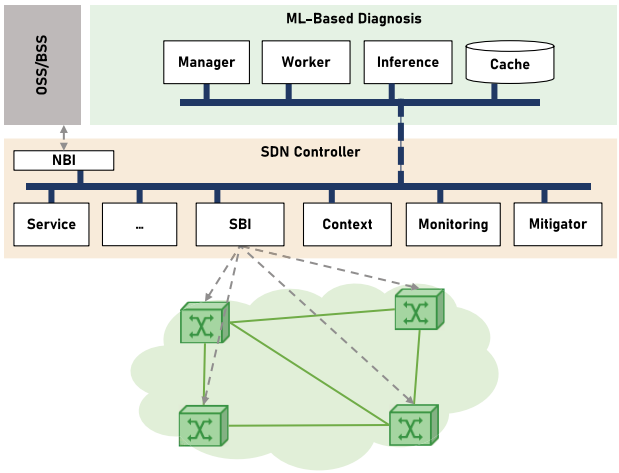


Fig. 2. Proposed diagnosis module and its integration with existing SDN controllers.

the southbound interface (SBI). Another responsibility usually assigned to the SBI is telemetry, i.e., collecting the monitoring data from optical devices and storing them in an internal monitoring database. The diagnosis module is composed of four components: *manager*, *worker*, *inference*, and *cache*. *Manager*, *worker*, and *inference* are proposed in this work and detailed in the following subsections, while the *cache* can be implemented as any off-the-shelf database (preferably an in-memory).

Figure 3 illustrates how communication takes place among the components in the proposed module. The figure highlights three stages of communication: initialization, optical service setup, and periodical diagnosis loop. During initialization, the *manager* obtains a list of optical services currently running in the network from the SDN controller. This list is maintained throughout the operation of the diagnosis module as a way to alleviate the load on the communication with the SDN controller.

During the optical service setup, the module works as follows. The SDN controller is expected to provide an interface following the *publish-subscribe* model, which allows the module to receive notifications upon any changes experienced by the services, e.g., service creation or termination. In Fig. 3, when a new service is requested, the SDN controller takes the actions required to set it up. Once the service is created, the *manager* receives a notification and adds the newly created service to its local list of services. The *manager* then requests the creation of new key performance indicator (KPI)(s) from the monitoring component of the SDN controller. The newly created KPI(s) will be used to store a time series of the results computed by the ML-based diagnosis module. A similar notification is received by the *manager* when a service is removed, followed by removing the service from its internal list.

The *manager* is responsible for periodically triggering the diagnosis loop. It iterates over the internal list of optical services, sending a request to the *worker* for each service. We leverage a combination of concurrency and parallelism to ensure this component is able to send a large number of requests. Sending an independent request per service allows the *worker* component to be replicated depending on the number

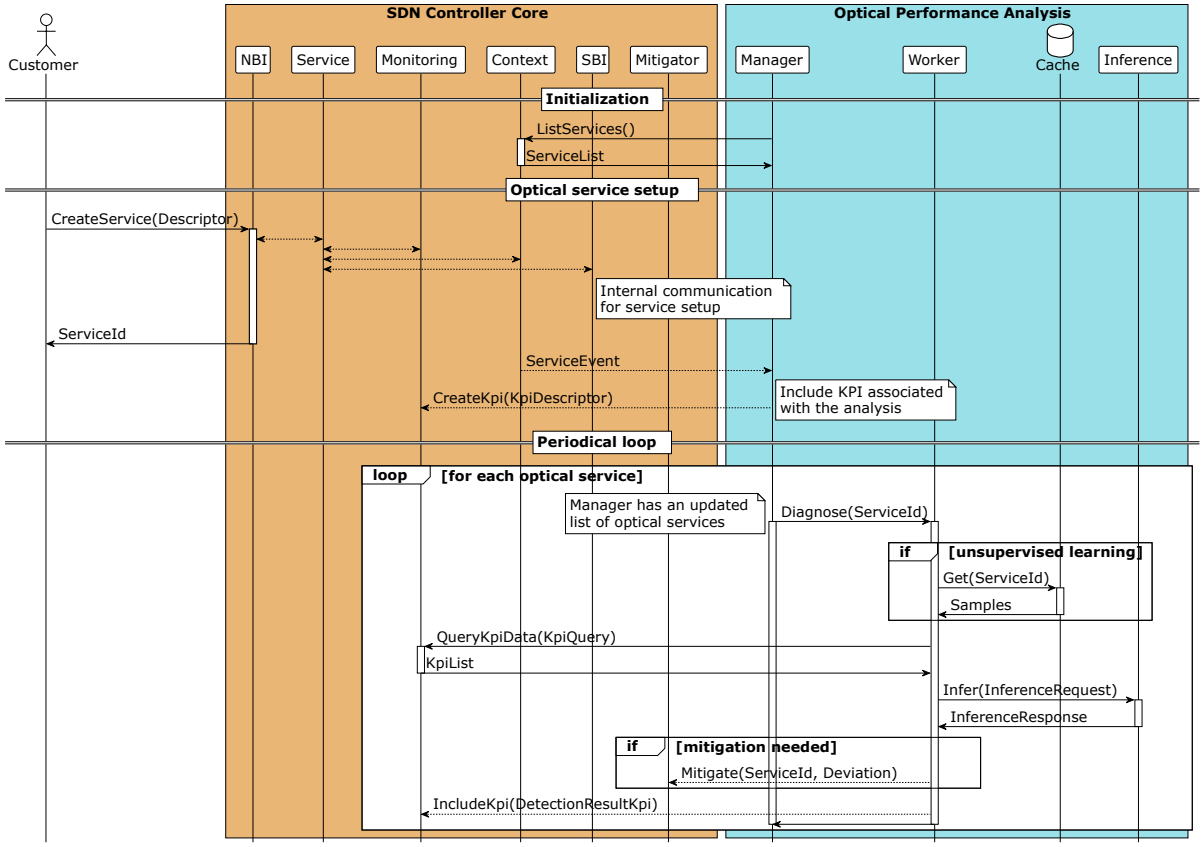


Fig. 3. Workflow of the proposed architecture.

of services to be analyzed and balancing the load among the replicas.

The *worker* runs the analysis for a particular service. To this end, it leverages the *inference* to apply various (ML-based) algorithms for processing the data of a service. Which algorithm to use and, consequently, which data to retrieve can be set flexibly on the fly, depending on the task. First, it queries the monitoring component of the SDN controller to retrieve the latest OPM sample(s) related to the service at hand. Depending on the specific model used by the *inference* component, a larger window of preceding OPM samples might be needed (e.g., to perform data clustering when unsupervised learning is applied for attack detection). In this case, the *worker* leverages the *cache* to store this data, alleviating the load on the SDN controller by only querying new data that are not yet in the *cache*. Once the necessary sample(s) for performing the inference are gathered, the *worker* invokes the *inference* component, which is responsible for executing the ML model suitable for the task. For instance, if the purpose of the diagnosis is to detect degradation caused by a previously unseen physical-layer attack, UL or semisupervised learning (SSL) models performing anomaly detection may be used. If the purpose is to identify a previously detected and known anomaly, SL models performing classification may be used.

Upon receiving the results from *inference*, the *worker* tests if the diagnosis detected a deviation in the service performance and notifies the SDN controller in the positive case. The meaning of the deviation varies with the use case but, in general, represents any assessment that differs from normal operating conditions. For instance, if the use case is physical layer security, an attack-related deviation may indicate the presence and potentially the type and location of an attack [19]. Another potential use case is the detection of soft failures, where a failure-related deviation indicates the presence of degradation in the quality of transmission of the channel and potentially the device or configuration causing the deviation [12]. Regardless of the result, the *worker* notifies the SDN controller of the inference results. By doing so, the module consolidates the result of the diagnosis in the same database where the monitoring data comes from, thus enabling the correlation and creation of integrated dashboards. At this point, the *worker* has performed all of its task and replies to the *manager*, communicating that the work for the particular service is done.

The module and its workflow are designed to allow the components to take advantage of the replication and load-balancing features available in current container orchestration platforms such as Kubernetes. Replication in these platforms can be triggered by defining thresholds for CPU or RAM usage as well as other application-specific metrics. In general, our results indicate CPU usage thresholds are enough to govern replication and achieve satisfactory results with the proposed architecture.

The communication interfaces among the components can be realized with any *web-service-like* protocol. Examples commonly used are the gRPC or representational state transfer (REST) interfaces. In our case, we decided to use the gRPC interface [20]. The gRPC protocol is an implementation of a remote procedure call (RPC) that allows a strongly typed, programming-language-agnostic definition of messages and procedures that can be used to generate language-specific code. In the following, we specify the details of the components and the interface with the SDN controller.

A. Manager

The *manager* is the component responsible for coordinating the entire diagnosis periodical loop and does so through three tasks: (i) keeping a list of active optical services in the network, (ii) maintaining a timer that triggers the periodical diagnosis loop, and (iii) delegating the individual processing of each service when executing the loop. As illustrated in Fig. 3, during initialization, the *manager* retrieves the list of active optical services in the network. This allows the component to be initialized at any time during network operation. It also grants the *manager* the ability to survive potential service restarts due to hardware or software issues. The *manager* is the only component in the module suitable for operating without replication, i.e., it runs in a single container and does not scale with the load. All other components in the architecture do scale, i.e., increase the number of replicas depending on the load. However, note that tasks (ii) and (iii) need to be executed in parallel, so each task is executed by its process within the container. Moreover, in order to send hundreds or thousands of requests per period in task (iii), the *manager* leverages a combination of concurrency and parallelism. In the following, we present the pseudocode for tasks (i) and (iii).

Algorithm 1 shows how the *manager* maintains an updated list of active optical services in the network. Note that it receives two parameters. The *services* parameter contains a shared mutable reference to the list of services used by both Algorithms 1 and 2. The *sdn* parameter represents a client to the SDN controller application programming interface (API). First, the algorithm subscribes to events by using the SDN controller API (line 2). For every received event (line 3), the *manager* first obtains (i.e., creates or retrieves existing) KPI identifier(s) that will be used to store the diagnosis result (line 4). Then, the *manager* verifies whether the event is a service creation or deletion. If it is a service creation (lines 5–6), the *manager* appends a tuple with the newly created service identifier and its associated KPI identifier(s) to the list of active services. If the event is a service deletion (lines 7–8), the *manager* removes the respective tuple from the list. Note that there is no need to delete the KPIs since this information can be useful for posterior analysis.

Algorithm 1. Subscription to Events and Maintenance of the Updated List of Services

```
1: function GetEvents(services, sdn)
2:     stream ← sdn.GetServiceEvents()
3:     for event ∈ stream do
4:         kpi_id ← sdn.SetKpi(event.service)
5:         if event.type = CREATE then
6:             services ← service_list ∪ (event.service, kpi_id)
7:         else if event.type = REMOVE then
8:             services ← services (event.service, kpi_id)
```

Algorithm 2. Periodical Loop

```
1: async fn DiagnoseServices(services, period, worker)
2:   num_threads = MIN_TH
3:   while true do
4:     tasks = ∅
5:     start = now()
6:     for service, kpi_id ∈ services do
7:       task ← worker.Diagnose(service, kpi_id)
8:       tasks ← tasks ∪ task
9:     pool ← ThreadPool(num_threads)
10:    run with pool and await all tasks
11:    elapsed_time ← now() − start
12:    report(elapsed_time)
13:    num_threads ← result from Eq. (2)
14:    if elapsed_time > period then
15:      continue
16:    sleep(elapsed_time − period)
17: end async fn
```

Algorithm 2 details the periodical diagnostic loop operations performed by the *manager*. It receives three parameters. The first is a read-only reference to the *services* list maintained in Algorithm 1. The *period* represents the time between two consecutive loops (e.g., 30 s). The *worker* is a client of the API provided by the *worker* component. The function also assumes the existence of two constants, *MIN_TH* and *MAX_TH*, representing the minimum and maximum number of threads that the function can use to send requests to the *worker*. The function starts by assigning the value of the minimum number of threads to a local variable (line 2). Then, the function runs indefinitely (line 3). For each loop, the *manager* initializes an empty list of tasks (line 4) and registers the time at which the current loop is starting (line 5). Then, for each service and KPI tuple in the *services* list, the *manager* delegates the actual diagnosis to the *worker* (line 7). It is important to note that this function is asynchronous (line 1). This means that the variable *task* (line 7) does not contain the response from the *worker* but rather a reference to the submitted task. In asynchronous programming, several tasks can be submitted for execution concurrently, and the result of a task will be provided only when the task is *awaited*. Moreover, it is possible to combine the concurrency of asynchronous programming with parallelism (e.g., multithreading). In our context, a single *manager* instance can trigger the tasks needed to diagnose all optical services without concerns for their completion while doing so. The task for the current service is appended to the list of tasks (line 8). After all tasks have been submitted, the function creates a thread pool (line 9) with *num_threads* threads. The tasks are then evenly distributed among the threads (line 10). Once the diagnosis of all services is completed, the *manager* computes the elapsed time (line 11) and reports the value to an application monitoring entity (line 12).

In order to handle a varying number of services, the *manager* dynamically changes the number of threads being used. At every loop execution, the *manager* recomputes how many threads are necessary to handle the current number of monitored services. This is done in line 13 with the help of the following two equations:

$$\text{desired\_threads} = \left\lceil \text{num\_threads} \times \frac{\text{elapsed\_time}}{\text{period}} \right\rceil, \quad \textbf{(1)}$$

$$\text{num\_threads} = \min(\text{MAX\_TH}, \max(\text{MIN\_TH}, \text{desired\_threads})). \quad \textbf{(2)}$$

Equation (1) computes the desired number of threads based on the current number of threads, the measured loop time (elapsed_time), and the *period*. Then, Eq. (2) computes the final number of threads considering the minimum and maximum allowed. This formulation is adapted from the horizontal pod autoscaling mechanism used by Kubernetes [21]. If the loop execution time exceeds a predefined value, the next loop starts immediately (lines 14–15). Otherwise, the *manager* sleeps for the remaining time (line 16).

B. Worker

The *worker* is responsible for diagnosing each service. It leverages the *inference* component to perform ML-based diagnosis and triggers relevant actions when a deviation is detected. Therefore, the *worker* needs to provide the necessary information for the *inference*. The information needed depends on the model adopted by the *inference*. For instance, if the *inference* uses a UL model, it will perform anomaly detection and needs a window of OPM samples to distinguish normal operating conditions from anomalies by, e.g., clustering. If the *inference* uses an SL or SSL model, it can perform classification or anomaly detection, respectively, and can do so with a single OPM sample [19]. It can also happen that multiple models are required, so the *inference* component will have several implementations running in parallel. For instance, both SL and UL can be used in combination to improve performance [16,18] or achieve explainability [22]. Finally, the *inference* component supports the case where an SL model is used for regression or prediction. For instance, this is suitable when performing threshold-based anomaly detection or predictive maintenance [23]. This is a key flexibility feature of this component.

Given that the *inference* component may have different interfaces, we present two versions of the *worker* implementation: one for SL/SSL and one for UL inference. In both cases, we assume that they have access to the following variables (omitted from the algorithms for clarity):

- *sdn*: a client to the SDN controller API;
- *inference*: a client to the *inference* API;
- *preprocessing*: a reference to the preprocessing algorithm commonly applied in data before using it in ML models; and
- *cache*: a client to the *cache* API.

Algorithm 3 shows the implementation of the *worker* when using an SL/SSL model. In this case, only the latest OPM sample(s) are needed, i.e., the ones that have not been analyzed so far represented by n_samples (we assume n_samples = 1 in Algorithm 3). First, the latest sample(s) are obtained from the SDN controller (line 2). The sample(s) are then preprocessed using the algorithm of choice (line 3). The preprocessed samples are then used to invoke the *inference* component that returns a list of classes, i.e., one for each sample (line 4). The results of ML inference are reported to the SDN controller (line 5). If any class in the result represents a deviation, i.e.,

Algorithm 3. Worker when Using Supervised Learning

```
1: function ANALYZE(service, kpi_id)
2:     latest ← sdn.QueryKpiData(service, n_samples)
3:     sample ← preprocessing.Process(latest)
4:     result ← inference.Classify(sample)
5:     sdn.IncludeKpi(kpi_id, result)
6:     if any value in result ≠ NORMAL then
7:         sdn.Mitigate(service, result)
```

Algorithm 4. Worker when Using Unsupervised Learning

```
1: function MONITORSERVICE(service, kpi_id)
2:     latest ← sdn.QueryKpiData(service, n_samples)
3:     samples ← preprocessing.Process(latest)
4:     cache.get(service).pop(n_samples)
5:     cache.get(service).push(samples)
6:     window ← cache.get(service)
7:     result ← inference.Detect(window)
8:     sdn.IncludeKpi(kpi_id, result)
9:     if any value in result is ≠ NORMAL then
10:        sdn.Mitigate(service, result)
```

is different from the class that represents the normal working conditions, the *worker* notifies the SDN controller about the potential need for mitigation (lines 6–7). When the model is performing regression or prediction, line 6 will check whether or not the *result* value(s) is(are) within the thresholds of what is considered normal operating conditions.

Algorithm 4 presents the algorithm using a UL model. We focus only on the differences from Algorithm 3. The first difference is that, in this case, due to the need for a relatively large number of samples for each inference, the *worker* leverages the *cache* to alleviate the load on the *monitoring* component. We assume that the cache for each service is prepared the first time the service is analyzed, and we omit this part. The *worker* first removes the oldest *n*_samples from the cache (line 4) and includes the newest *n*_samples into the cache of the service under analysis (line 5). The complete window of samples is then retrieved from the cache (line 6). The remainder of the algorithm is the same as in Algorithm 3. Note that the used number of samples will define the accuracy of the detection and the computational requirements of the operation [10,18].

C. Inference

Unlike the previous components, the *inference* component can take many different shapes depending on the model used to perform anomaly detection, classification, regression, or prediction. In this scenario, the most well-known implementation is TensorFlow Serving [24], which provides a standard interface to serve artificial neural network (ANN) models through the network. However, ANNs belong to the category of SL models. For UL, DBSCAN Serving [25] is an alternative that uses the density-based spatial clustering of applications with noise (DBSCAN) algorithm, a popular UL algorithm used for anomaly detection.

Algorithm 5 presents the interface of the *inference* component when using DBSCAN as the ML algorithm. The

Algorithm 5. Interface of the *Inference* Component when Using DBSCAN

```
1: enum DISTANCEMETRIC
2:     EUCLIDEAN = 0
3:     COSINE = 1
4:     ...
5: end enum
6: message SAMPLE
7:     repeated float features
8: end message
9: message DETECTIONREQUEST
10:    float eps
11:    int32 min_samples
12:    DistanceMetric metric
13:    repeated Sample samples
14: end message
15: message DETECTIONRESPONSE
16:    repeated int32 cluster_indices
17: end message
```

representation is inspired by the gRPC protocol buffer definition, and the messages are inspired by TensorFlow Serving. DBSCAN contains three parameters: (i) the distance function used to calculate the distance among samples, (ii) the maximum distance between two samples considered neighbors (*eps*), and (iii) the minimum number of neighboring samples necessary to form a cluster (min_samples). The *DistanceMetric* enumeration (lines 1–5) declares which distance functions are available in the implementation. The *Sample* message (lines 6–8) represents each sample to be considered, consisting of an array of features. The *DetectionRequest* message (lines 9–14) encompasses the algorithm parameters *eps*, min_samples, and *metric* as well as an array of samples to be analyzed. Finally, the *DetectionResponse* message (lines 15–17) contains an array of integers with the same number of elements as *samples*, representing the cluster to which each sample was categorized.

D. Interface with the SDN Controller

In the proposed module, we assume that the SDN controller is responsible for operating the network and offers functionalities to external modules (also known as *apps* in the context of SDN). In this section, we focus on the functionalities that the SDN controller needs to expose in the form of APIs to ensure integration of the proposed module. Note that all the functionalities required by our module are of wide use for any app integrating with the SDN controller and are usually included in the set of APIs made available by SDN controllers.

We assume that the SDN controller provides the APIs that allow for the ML-based diagnosis components to query monitoring data and perform control actions. In this way, the module can be considered an SDN *application* taking advantage of the SDN controller's APIs to expand the provided functionalities. The SDN controller is responsible for all service control and management procedures, including service monitoring (i.e., obtaining OPM data from the network devices), and mitigation of detected deviations (e.g., anomalies, degradation, or attacks). We divide the responsibilities

into three categories: (i) publish/subscribe updates, (ii) monitoring database maintenance, and (iii) anomaly/degradation mitigation. These functionalities are explained as follows.

1. Publish/Subscribe Service Updates

Our module assumes that the SDN controller provides an API that allows external entities to subscribe to internal events such as service creation and deletion. This functionality enables the *manager* to maintain an internal list of current services in operation. The *manager* can work even if the SDN controller does not provide this functionality. In that case, at the beginning of every diagnosis loop, the *manager* queries the entire list of active services from the SDN controller. Therefore, the minimum API required by our component must allow an entity (or a module in this case) to retrieve the list of active services.

2. Monitoring

Our module assumes that the SDN controller provides an API that allows external entities to (i) query monitoring data collected from optical devices and (ii) include new data associated with the services under analysis. In network monitoring and telemetry, monitoring data are usually represented as a time series. Conversely, ML models require data samples composed of features. For instance, for a single optical service, a sample represents the OPM data collected from the device at some point in time. The sample is composed of several features. In the context of optical networks, features can include metrics such as optical power received (OPR), optical signal-to-noise ratio (OSNR), and pre-FEC bit error rate (BER-FEC).

When querying the monitoring data, our module requires the API to allow for filtering the information for a specific service and defining specific KPIs to be retrieved. To avoid overloading the SDN controller, it is also desirable to define a period for the samples, e.g., the last n monitoring samples. Once the information is received, the *worker* interprets the received data and converts them into the appropriate format expected by the *inference* component.

When including new data, two interfaces are needed. The first one allows the *manager* component to create new KPI(s) upon service creation, associated with the result of the performed ML-based diagnosis. The second one allows the *worker* to assign values to the KPI(s) at the end of each diagnosis cycle.

3. Mitigation

The final API expected from the SDN controller is related to mitigating the potential anomalies, attacks, or degradation. Upon performing ML-based diagnosis, the used ML model(s) may detect potential degradation and/or disruptions that may require the SDN controller to perform a mitigation action. In this case, the ML-based diagnosis module may notify the SDN controller directly, as shown in Fig. 3. It is also possible for the SDN controller to take an indirect approach. In that case, the SDN controller requests to be notified whenever some KPI reaches a particular value or threshold. An example is soft failure detection or classification using SL where the class representing normal operating conditions takes a value equal to zero, while the value greater than zero represents a soft failure. The SDN controller may subscribe to events where the KPI representing this class takes on a value greater than zero.

In general, a mitigation strategy may begin by localizing the fault, i.e., identifying the network element that was breached or has failed and identifying the characteristics of the detected anomaly to determine the most appropriate remedy. The exact set of actions may be different depending on the attack and failure types. In the case of failures caused by component fatigue or fault, adapting the modulation format or the frequency of the affected connection can be sufficient [26], while other types of failures may require the use of backup routes to bypass the failed components.

In the case of physical-layer attacks, the breached or harmful network element (e.g., a link or a connection) can be localized, e.g., with the help of the approach based on attack syndromes from [19]. The network security operator may then decide on the short- and long-term remediation steps [18]. The first response may be to recover the affected connections by, e.g., rerouting them away from the breached element using preplanned routes that provide protection from attacks [27]. As the nature of optical-layer attacks implies fraudulent modification of the network infrastructure, it is unlikely that the attack can be permanently fixed with elementary network functions like traffic protection or rerouting. This creates the need for longer-term remediation actions that may include organizing a human repair intervention on the network infrastructure, e.g., switching off amplifiers to isolate a breached link, followed by physically removing the compromised devices in the field.

3. PROOF-OF-CONCEPT VALIDATION

This section presents a proof-of-concept implementation of the diagnosis module proposed in this work. We use TeraFlowSDN [6] as the SDN controller responsible for the optical network operations. To avoid external factors affecting our analysis, we adopted the following two measures. First, as detailed in the next subsection, we used an emulated data plane composed of software-based optical transceivers that replay data from a collected dataset. Second, we modified the provisioning procedure of TeraFlowSDN to bypass all the steps related to service establishment except for the ones related to monitoring. This means that, upon a service request, we assume to already know the path for the services; we also assume there are always enough resources available to provision that path. Since we need hundreds of optical services to stress-test the scalability of the diagnosis module, the latter assumption is necessary to establish these many services in a short time.

The components of the module were implemented using Python 3.9. All the communication among components uses gRPC. We adopted Redis [28], a fast in-memory database, as the cache solution for our module. We used the development version of ETSI TeraFlowSDN [29].

We deployed a Kubernetes node using the MicroK8s distribution. The node works as both controller and worker, i.e., the workload runs in the same machine as the Kubernetes controller node. Kubernetes is responsible for managing the

containers of the module. The machine hosting the experiments is equipped with an AMD Ryzen Threadripper 3960X 24-Core Processor with 128 GB of RAM with 3600 MHz. The components were instrumented using Prometheus [30]. The *manager* was deployed with 12 CPUs allocated to it. One of these CPUs was allocated to the maintenance of the updated list of services (Algorithm 1), while the remaining CPUs are available to the periodical loop (Algorithm 2). The minimum and maximum numbers of threads in Eq. (2) were set to 2 and 10, respectively. The *cache* was deployed with 500 mCPUs allocated to it and allowed to use up to 1 CPU. Both *worker* and *inference* were deployed with 300 mCPUs and allowed to use up to 1 CPU. We enabled horizontal pod autoscaling (HPA) for the *worker* and *inference*. We set the minimum number of replicas (*min_replicas*) to 2 and the maximum (*max_replicas*) to 10, with the target CPU usage (*target_usage*) set to 80%. Periodically, the HPA computes the (new) desired number of replicas (*des_replicas*) based on the current number of replicas (*cur_replicas*), and the current (*cur_usage*) and target CPU usage using the following formulas:

$$r = \left\lceil \text{cur\_replicas} \times \frac{\text{cur\_usage}}{\text{target\_usage}} \right\rceil, \tag{3}$$

$$\text{num\_replicas} = \min(\text{max\_replicas}, \max(\text{min\_replicas}, r)). \tag{4}$$

The Linkerd service mesh was used to balance the load among all the replicas [31]. When new replicas are added, Linkerd includes them in the pool of replicas and starts directing traffic to them. Upon the removal of replicas, Linkerd also updates its list and stops considering the removed replica in the load balancing.

To run the experiments, we developed a custom script that controls the number of optical services in the network by quickly establishing or deleting services. The experiment works as follows. The number of optical services in the network is set to {120, 240, 480, 960, 1440, 1920}. Each number of services is maintained for 30 min. We adopt a 10 min interval between different numbers of optical services to bring the system back to its idle state, as illustrated in Fig. 4. We also adopt a period of 30 s, i.e., in ideal conditions, the loop will be executed twice per minute. In the last part of the experiment, we set the period to 1 min and evaluate how the system behaves with 1920 active services.

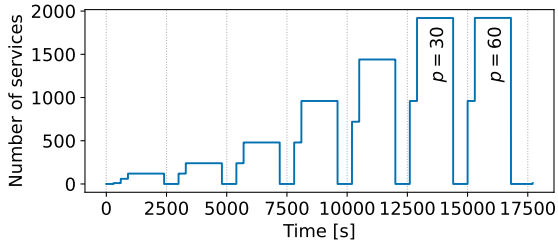


Fig. 4. Number of services over time. The last two experiments represent the case with the period (p) equal to 30 s and 60 s.

A. Use Case: Optical Physical-Layer Security Diagnosis

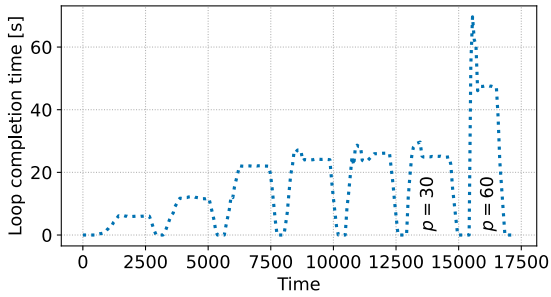
We selected the optical physical-layer security diagnosis to validate our implementation of the proposed module. We leverage the dataset reported in [17]. The dataset was collected on a real-world testbed, with two monitored optical channels under test. The monitoring system collected one sample per minute. Each sample contains 12 KPIs, including the OPR, OSNR, and BER-FEC, among others. For some features, the minimum and maximum values observed within a minute are reported in addition to the nominal value in a total of 32 features. The dataset contains seven different conditions: normal operating conditions, light in-band jamming (INBLGT), strong in-band jamming (INBSTR), light out-of-band jamming (OOBLGT), strong out-of-band jamming (OOBSTR), light polarization modulation (POLLGT), and strong polarization modulation (POLSTR). In this work, we do not benchmark the mitigation strategy. To include diversity in the dataset used and prevent any bias towards any part of the dataset, we randomly sample from any attack characterizations at each loop instance.

To scale our experiment to hundreds of optical channels, we extracted the average value and the standard deviation for each feature in each condition (normal and attacks). Then, depending on the condition desired for an optical channel, we sample a normal distribution parameterized by each feature's average and standard deviation.

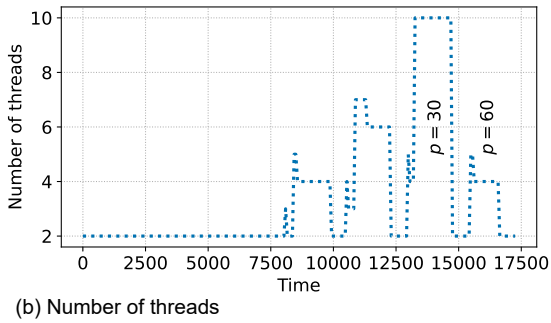
In the following experiments, we used the DBSCAN algorithm and adopted window-based attack detection (WAD) from [19] as a means to improve the UL technique performance. Namely, the relatively high false positive rate of DBSCAN can result in an excessive likelihood of false alarms, while the relatively high false negative rate may result in the omission of alarms when needed. WAD compensates for these issues by applying additional scrutiny to the outputs of the ML model rather than using them in their raw form. This is achieved by defining an observation window of size δ (i.e., the number of most recent ML outputs) and setting a threshold τ on the number of samples deemed as attacks necessary to raise an alarm. WAD has been shown to compensate for the false positive and false negative rates, alleviate the impact of ML output oscillations, and reduce the likelihood of false alarms already for moderate window size (e.g., $\delta = 10$) and relatively low thresholds (e.g., $\tau = 3$), at the expense of slightly longer attack detection time [19]. For the inference, we use 330 samples with 32 features [10]. In this case, DBSCAN serving acts as the *inference* component.

B. Performance Assessment

In this section, we assess the scalability and resource efficiency performance of the components in the diagnosis module. Figure 5 shows the statistics captured by the *manager*. Figure 5(a) shows the loop completion time. This represents the time taken to diagnose all active optical services. For reference, this time is measured in line 11 of Algorithm 2. We can observe that, while the number of services increases by 16×, i.e., from 120 to 1920, the loop completion time increases from around 5 s to below 30 s (for the 30 s period configuration). Meanwhile, Fig. 5(b) shows that the number



(a) Loop completion time measured by the *manager*



(b) Number of threads

Fig. 5. Performance measurements at the *manager* (Algorithm 2) over time.

of threads may increase from its minimum (two threads) up to the maximum (10 threads). This shows that our module and implementation can handle a large number of services while maintaining the loop completion time below the desired period (i.e., 30 s in our case). Figure 5(a) also shows for the last configuration (i.e., 1920 services and 60 s period) that, when the period is relaxed, the number of threads used is decreased substantially (from 10 to 4). Finally, we note that the loop completion time may take more than the desired period when a large number of services are added in a short time. However, the module quickly scales and stabilizes the loop completion time below the desired period.

Figure 6 shows a summary of the response time for the components when 480 optical services are active. The response time encompasses the time elapsed between receiving a request and finalizing the response to the component that made the request. The results for other numbers of optical services are similar and are omitted. The *cache* shows good performance, serving all requests within 2.5 ms. The DBSCAN algorithm used in the implementation of the *inference* can process more than 80% of the requests in under 100 ms. Given that the *worker* uses the *cache* and *inference*, it is expected that its response time is longer than that of its dependencies. In 99% of the requests, the *worker* completes the processing in under 250 ms, i.e., the diagnosis of each optical service takes less than a quarter of a second in the benchmarked implementation.

Figure 7 shows (a) the CPU utilization and (b) the number of replicas of the *worker* and *inference* components. The average CPU utilization is kept between 20% and 60% for most of the experiment, except for some peaks. These peaks in CPU utilization match the increments of the number of services, as shown in Fig. 4. The last part of the experiments, starting from

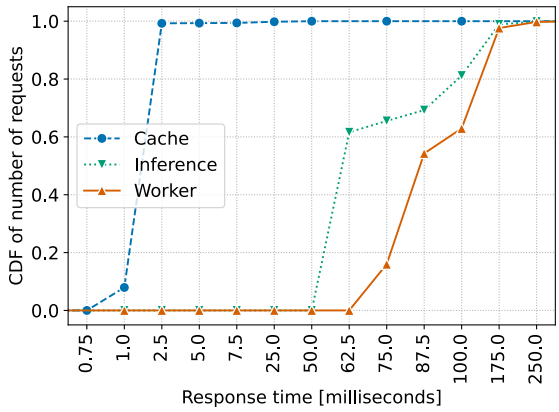
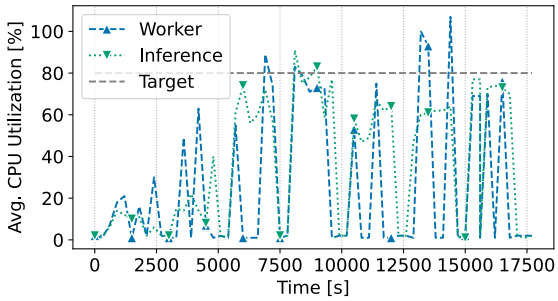
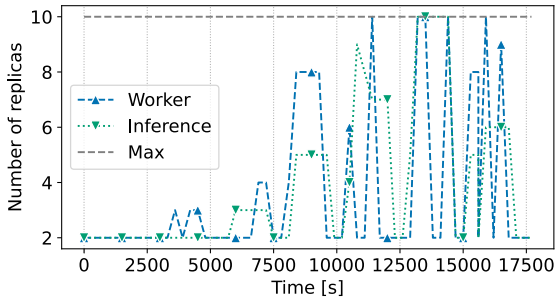


Fig. 6. Cumulative distribution function of the number of requests with respect to the response times of the scalable components for the load of 480 optical services.



(a) Avg. CPU utilization over all replicas



(b) Number of replicas per component

Fig. 7. Details of the *inference* component executing a UL algorithm.

7500 s, shows the CPU utilization of the *inference* component close to the target of 80% utilization.

The average CPU utilization is kept below 80% in almost all cases thanks to the addition of replicas, as shown in Fig. 7(b). It is worth noting that keeping CPU utilization below the threshold allows a component to maintain stable response time, as reported previously.

By dynamically adjusting the number of replicas to the needs, our module efficiently controls the number of resources reserved for the diagnosis module. Both components require only two to three replicas for diagnosing up to 240 optical services. When the number of services increases to 480, the *inference* scales to three replicas, while the *worker* scales to up to four replicas.

For 1920 optical services with a 30 s period, the *worker* reaches the maximum number of replicas, i.e., 10. However, when the period increases to 60 s, we can see that there is a drop in usage and number of replicas. Since the *manager* has more time to process all the requests, it is able to wait longer for the response of the components. This reduces the pressure on the *worker* and *inference*, which in turn can have their number of replicas reduced.

4. CONCLUSIONS

This paper introduced a new module designed for scalable and flexible diagnosis of optical services. The module divides the tasks involved in the diagnoses among four different components. The functionalities of ML models are encapsulated inside the *inference* component, which enables various types of ML models to be seamlessly integrated into the diagnosis. An implementation of the proposed module is presented, adopting microservices and a cloud-native architecture. Results obtained using a physical layer security use case demonstrate the scalability and flexibility properties of the proposed module. The completion time of the tasks is kept stable regardless of the number of services being diagnosed in the network. Moreover, we showed that the proposal is resource-efficient, i.e., it adapts to the number of resources reserved for the current needs.

The proposed module paves the way for scalable, efficient, and flexible use of ML-based optical network diagnosis in networks that scale from a few to several hundreds of services. However, some challenges are still relevant to be addressed in this area. For instance, developing and adopting confidence-aware ML models are crucial to the reliability of the ML model output. This would allow the system to trust an assessment associated with high confidence but fall back to human assessment when a low-confidence assessment is made. Moreover, more research is needed to improve the generalization capabilities of current models, so that models, once developed and trained, can be used across different networks.

Funding. Vetenskapsrådet (2019-05008); Horizon 2020 Framework Programme (101015857).

Acknowledgment. We thank M. Schiano and A. Di Giglio for their contribution to collecting the dataset used in this work. We gratefully acknowledge Infinera for providing the Groove G30 transponder.

REFERENCES

1. E. Le Rouzic, O. Renais, J. Meuric, T. Marcot, C. Betoule, G. Thouenon, A. Triki, M. Laye, N. Pelloquin, Y. Lagadec, E. Delfour, M. Ermel, J. Dost, and S. Turk, "Operator view on optical transport network automation in a multi-vendor context [Invited]," J. Opt. Commun. Netw. **14**, C11–C22 (2022).
2. T. Tanaka, A. Hirano, S. Kobayashi, T. Oda, S. Kuwabara, A. Lord, P. Gunning, O. Gonzáles de Dios, V. Lopez, A. M. Lopez de Lerma, and A. Manzalini, "Autonomous network diagnosis from the carrier perspective [Invited]," J. Opt. Commun. Netw. **12**, A9–A17 (2020).
3. F. Paolucci, A. Sgambelluri, F. Cugini, and P. Castoldi, "Network telemetry streaming services in SDN-based disaggregated optical networks," J. Lightwave Technol. **36**, 3142–3149 (2018).
4. R. Casellas, R. Martinez, R. Vilalta, R. Muñoz, A. Gonzalez-Muniz, O. G. de Dios, and J.-P. Fernandez-Palacios, "Advances in SDN control and telemetry for beyond 100G disaggregated optical networks," J. Opt. Commun. Netw. **14**, C23–C37 (2022).
5. M. Lonardi, J. Pesic, T. Zami, E. Seve, and N. Rossi, "Machine learning for quality of transmission: a picture of the benefits fairness when planning WDM networks," J. Opt. Commun. Netw. **13**, 331–346 (2021).
6. R. Vilalta, R. Muñoz, R. Casellas, R. Martínez, V. López, O. González de Dios, A. Pastor, G. P. Katsikas, F. Klaedtke, P. Monti, A. Mozo, T. Zinner, H. Øverby, S. Gonzalez-Diaz, H. Lønsethagen, J.-M. Pulido, and D. King, "TeraFlow: secured autonomic traffic management for a tera of SDN flows," in *Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)* (2021), pp. 377–382.
7. "Functional requirements for transport API," ONF TR-527 (2016).
8. M. Balanici, G. Bergk, P. Safari, B. Shariati, J. Karl, and R. Freund, "Demonstration of a real-time ML pipeline for traffic forecasting in AI-assisted F5G optical access networks," in *European Conference on Optical Communication (ECOC)* (2022), paper Tu2.5.
9. C. Natalino, C. Manso, R. Vilalta, P. Monti, R. Muñoz, and M. Furdek, "Scalable physical layer security components for microservice-based optical SDN controllers," in *European Conference on Optical Communication (ECOC)* (2021), paper We3E.2.
10. C. Natalino, C. Manso, L. Gifre, R. Muñoz, R. Vilalta, M. Furdek, and P. Monti, "Microservice-based unsupervised anomaly detection loop for optical networks," in *Optical Fiber Communication Conference (OFC)* (2022), paper Th3D.4.
11. C. Manso, R. Vilalta, R. Muñoz, N. Yoshikane, R. Casellas, R. Martínez, C. Wang, F. Balasis, T. Tsuritani, and I. Morita, "Scalability analysis of machine learning QoT estimators for a cloud-native SDN controller on a WDM over SDM network," J. Opt. Commun. Netw. **14**, 257–266 (2022).
12. A. P. Vela, M. Ruiz, F. Fresi, N. Sambo, F. Cugini, G. Meloni, L. Potì, L. Velasco, and P. Castoldi, "BER degradation detection and failure identification in elastic optical networks," J. Lightwave Technol. **35**, 4595–4604 (2017).
13. D. Wang, C. Zhang, W. Chen, H. Yang, M. Zhang, and A. P. T. Lau, "A review of machine learning-based failure management in optical networks," Sci. China Inf. Sci. **65**, 211302 (2022).
14. Z. Wang, M. Zhang, D. Wang, C. Song, M. Liu, J. Li, L. Lou, and Z. Liu, "Failure prediction using machine learning and time series in optical network," Opt. Express **25**, 18553–18565 (2017).
15. F. N. Khan, Q. Fan, C. Lu, and A. P. T. Lau, "Machine learning-assisted optical performance monitoring in fiber-optic networks," in *IEEE Photonics Society Summer Topical Meeting Series (SUM)* (2018), pp. 53–54.
16. X. Chen, B. Li, R. Proietti, Z. Zhu, and S. J. B. Yoo, "Self-taught anomaly detection with hybrid unsupervised/supervised machine learning in optical networks," J. Lightwave Technol. **37**, 1742–1749 (2019).
17. C. Natalino, M. Schiano, A. Di Giglio, L. Wosinska, and M. Furdek, "Experimental study of machine-learning-based detection and identification of physical-layer attacks in optical networks," J. Lightwave Technol. **37**, 4173–4182 (2019).
18. M. Furdek, C. Natalino, A. Di Giglio, and M. Schiano, "Optical network security management: requirements, architecture, and efficient machine learning models for detection of evolving threats [invited]," J. Opt. Commun. Netw. **13**, A144–A155 (2021).
19. M. Furdek, C. Natalino, F. Lipp, D. Hock, A. D. Giglio, and M. Schiano, "Machine learning for optical network security monitoring: a practical perspective," J. Lightwave Technol. **38**, 2860–2871 (2020).
20. gRPC, https://grpc.io.
21. F. Tonini, C. Natalino, D. A. Temesgene, Z. Ghebretensaé, L. Wosinska, and P. Monti, "Benefits of pod dimensioning with best-effort resources in bare metal cloud native deployments," IEEE Netw. Lett. **5**, 41–45 (2023).
22. C. Natalino, M. Schiano, A. D. Giglio, and M. Furdek, "Root cause analysis for autonomous optical network security management," IEEE Trans. Netw. Service Manage. **19**, 2702–2713 (2022).
23. D. Rafique and L. Velasco, "Machine learning for network automation: overview, architecture, and applications [Invited Tutorial]," J. Opt. Commun. Netw. **10**, D126–D143 (2018).
24. TensorFlow Serving, https://www.tensorflow.org/tfx/guide/serving.

25. C. Natalino, "DBSCAN Serving," GitHub (2022), https://github.com/carlosnatalino/dbscan-serving-python.
26. N. Sambo, K. Christodoulopoulos, N. Argyris, P. Giardina, C. Delezoide, D. Roccato, A. Percelsi, R. Morro, A. Sgambelluri, A. Kretsis, G. Kanakis, G. Bernini, E. Varvarigos, and P. Castoldi, "Field trial: demonstrating automatic reconfiguration of optical networks based on finite state machine," J. Lightwave Technol. **37**, 4090–4097 (2019).
27. M. Furdek, N. Skorin-Kapov, and L. Wosinska, "Attack-aware dedicated path protection in optical networks," J. Lightwave Technol. **34**, 1050–1061 (2016).
28. Redis, https://redis.io.
29. ETSI TeraFlowSDN, https://labs.etsi.org/rep/tfs/controller/-/tree/develop.
30. Prometheus, https://prometheus.io.
31. Linkerd, https://linkerd.io.