Aquarius - Enable Fast, Scalable, Data-Driven Service Management in the Cloud

Zhiyuan Yao, Yoann Desmouceaux, Juan-Antonio Cordero-Fuertes, Mark Townsley, Thomas Clausen

Aquarius - Enable Fast, Scalable, Data-Driven Service Management in the Cloud

Zhiyuan Yao, Yoann Desmouceaux, Juan-Antonio Cordero-Fuertes, Mark Townsley, Thomas Clausen

***Abstract*—In order to dynamically manage and update networking policies in cloud data centers, Virtual Network Functions (VNFs) use, and therefore actively collect, networking state information - and in the process, incur additional control signaling and management overhead, especially in larger data centers. In the meantime, VNFs in production prefer distributed and straightforward heuristics over advanced learning algorithms to avoid intractable additional processing latency under high-performance and low-latency networking constraints. This paper identifies the challenges of deploying learning algorithms in the context of cloud data centers, and proposes Aquarius to bridge the application of machine learning (ML) techniques on distributed systems and service management. Aquarius passively yet efficiently gathers reliable observations, and enables the use of ML techniques to collect, infer, and supply accurate networking state information – without incurring additional signaling and management overhead. It offers fine-grained and programmable visibility to distributed VNFs, and enables both open- and close-loop control over networking systems. This paper illustrates the use of Aquarius with a traffic classifier, an auto-scaling system, and a load balancer – and demonstrates the use of three different ML paradigms – unsupervised, supervised, and reinforcement learning, within Aquarius, for network state inference and service management. Testbed evaluations show that Aquarius suitably improves network state visibility and brings notable performance gains for various scenarios with low overhead.**

***Index Terms*—Service management, data-driven, high performance network, cloud, performance evaluation**

I. Introduction

Growing demands for responsive, high-available, low-latency cloud services require content providers and cloud operators to efficiently manage cloud data centers (DCs) [1], [2]. To increase network programmability, and balance the trade-off between capital expenditures and quality of service (QoS), Virtual Network Functions (VNFs) (*e.g.*, firewalls, load balancers, and VPN gateways [3], [4]) are deployed in cloud DCs to provide reliable service management and transparent operations. Running on commodity computing platforms, VNFs replace or augment dedicated hardware devices and play a significant role in large-scale DCs. To dynamically monitor and configure VNFs, software-defined networking (SDN) schemes can be applied, dissociating the routing and decision-making process (*control plane*) from the network packets forwarding process (*data plane*) [5], [6].

Z. Yao, Y. Desmouceaux and M. Townsley are with Cisco Systems Paris Innovation and Research Laboratory (PIRL), 92782 Issy-les-Moulineaux, France; emails {yzhiyuan, ydesmouc,townsley}@cisco.com.

Z. Yao, J.-A. Cordero-Fuertes and T. Clausen are with École Polytechnique, 91128 Palaiseau, France; emails {zhiyuan.yao,juan- antonio.cordero-fuertes,thomas.clausen}@polytechnique.edu.

Digital Object Identifier 10.1109/TNSM.2022.3197130

The control plane adaptively manages and updates networking policies in dynamic cloud DC environments to offer high service availability and QoS. Data-driven mechanisms based on machine learning (ML) [7], [8] and reinforcement learning (RL) algorithms [9], [10] are applied and show performance gains in various network applications. For instance, auto-scaling systems and load balancers can achieve improved QoS with reduced cost based on periodically polled resource utilisation of distributed network devices (*e.g.*, application servers) [11], [12]. Traffic classification and anomaly detection help detect security threats with increased accuracy based on network traffic characteristics extracted from offline-collected network traces [13], [14]. However, it is challenging to harness these algorithms to drive management decisions in networking systems in real-time.

ML and RL algorithms require fine-grained observations of network and system states [15]: Conventionally, periodically polling resource utilisation and system performance allows for obtaining timely and dedicated observations to make data-driven management decisions [11], [12], [16]–[19]. However, the active polling scheme incurs additional control messages and reduces system scalability, especially for large-scale distributed systems. Another way to gather a wide range of fine-grained networking features is to parse and extract from offline collected network traces or in simulated environments, which is employed for developing clustering algorithms and RL algorithms [13], [15], [20]. However, this scheme assumes a minimal gap between real-time systems and offline/simulated systems, which does not necessarily hold in networking systems [19], [21].

The data plane is constrained by low-latency and high-throughput requirements [22], which makes it challenging to apply off-the-shelf ML algorithms on networking problems: Applying advanced ML techniques alongside the data plane on the fly is computationally intractable [23], [24]. Therefore, in real-world high-performance and large-scale networking systems, heuristics – which may not be adaptive to dynamic environments – prevail over advanced learning alrogithms [16], [18], [19], [25]–[31].

A. Contribution

This paper proposes Aquarius, a fast and scalable data collection and exploitation mechanism that bridges different requirements for data planes (low-latency and high-throughput) and control planes (making informed decisions). It enables learning algorithms to make inferences and open/close-loop control decisions based on fine-grained observations, and it

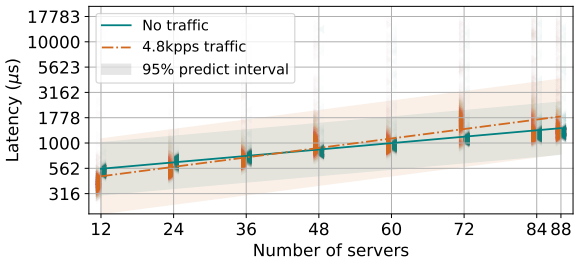


Fig. 1: Linear regression on probing latencies (with and without background network traffic) and additional response time collected on clusters of different numbers of servers.

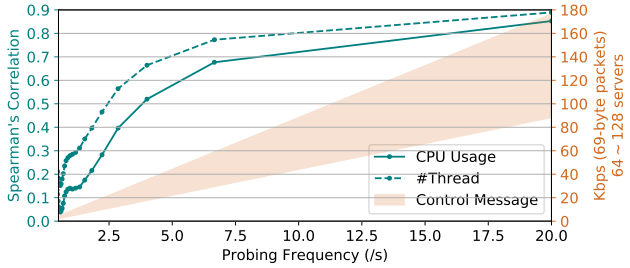


Fig. 2: Correlation (Spearman) increases when the probing frequency grows, yet, so do additional control messages.

allows the deployment of distributed and intelligent VNFs, which harness ML algorithms to make data-driven operational decisions. This paper makes the following contributions.

First, this paper identifies the challenges of gathering networking features to make valuable inference and informed operational decisions in high-performance and large-scale cloud DCs: Experimental evaluations demonstrate that traditional mechanisms for feature collection (*e.g.*, active probing [12], [17], [32]–[36] and trace capture [37]–[42]) causes substantial overhead. Using a real-world testbed, it is shown that networking features gathered by Aquarius in real time can provide valuable information for system states inference, with no additional control message and limited resource consumption.

Second, this paper proposes a fast and configurable mechanism, Aquarius, that allows collecting a wide range of fine-grained networking features in a scalable layout, which is suitable for applying various ML techniques to networking problems: Aquarius embeds programmable and flexible feature collection state machines in the data plane. These state machines are used to extract user-defined networking features. To efficiently gather observations, Aquarius collects 2 types of features – *counters* and *samples* – using *multi-buffering* [43] and *reservoir sampling* [44], respectively. Networking features are gathered separately corresponding to different network applications and types of equipment (*e.g.*, links, servers). Features are made available in a scalable layout, which offers high flexibility when aggregating and processing data under various requirements (*e.g.*, by single equipment or by groups of equipment).

Third, this paper provides an extensive performance and overhead evaluation of Aquarius with use cases experimented in a realistic testbed: Within the context of (i) an unsupervised-ML-powered network traffic classifier, (ii) a supervised-ML-powered auto-scaling system, and (iii) an RL-powered Layer-4 load balancer, this paper shows that the collected features enable:

- **unsupervised learning** + ***offline*** **data analysis**: creating benchmark datasets to gain insight into different networking problems with minimal data collection overhead;
- **supervised learning** + **VNF** ***management***: embedding ML techniques to achieve self-aware monitoring and self-adaptive orchestration in an elastic compute cloud;
- **reinforcement learning** + ***online*** **policy updates**: enabling closed-loop control, where collected networking features help optimise routing policies and improve QoS.

B. Paper Outline

The remainder of this paper is organized as follows. Section II describes the challenges of harnessing data-driven algorithms for networking problems and compares this paper with related work. Section III presents both the rationale and the design of the feature collection and exploitation mechanism of Aquarius. Section IV demonstrates the use of Aquarius in the context of 3 different VNFs on a realistic testbed. Section V concludes this paper.

II. Background

This section presents the challenges of efficient feature collection and data-driven VNFs in cloud DCs, and, with a comparison of related work, motivates the design of Aquarius.

A. Challenges

There is a rising trend of embedding intelligence and applying ML techniques in the cloud and distributed systems to dynamically monitor and adaptively configure system parameters and characteristics (*e.g.*, server configurations, forwarding rules) [13], [15], [19], [23], [55], [58]. However, this raises many challenges and trade-offs that require to be handled to efficiently collect features and make data-driven decisions.

Online Feature Collection: Datasets with high quality are essential to ML studies. However, few datasets are available and considered as a benchmark for ML applications in networking systems (*e.g.*, traffic analysis and anomaly detection) [37]–[42]. To collect a wide range of features (*e.g.*, traffic rates, packet sizes, TCP congestion window sizes), these datasets are collected based on logged network traces (*e.g.*, by TCPdump). Though log-based feature collection provides abundant information for various types of applications, it does not scale in terms of log file size [59]. Log-based feature collection also incurs performance overhead under heavy traffic, which leads to inaccurate and irrelevant measurements and makes it hard to bring ML algorithms "online" (making inference and management decisions in real-time) [23].

TABLE I: Comparison of data-driven VNF systems.

Property	[32], [33]	[45], [46]	[47]–[49]	[11], [50]–[52]	[7], [12], [15]–[18]	[9], [53], [54]	[55]	[34], [35]	[56], [57]	[24]	***Aquarius***
No Control Message	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓	✓
Distributed	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
Commodity Device	✓	✗	✓	✓	✓	✓	✓	✓	✗	✗	✓
Use Case	Generic	Management Protocol		Autoscaling	Traffic Optimisation		Traffic Classification			Generic	Generic

Scalability vs. Visibility: Active probing is another way of feature collection to monitor the system state and make informed decisions [12], [17], [32]–[36]. However, this requires modifications on each node to maintain management and communication channels. There is also a trade-off between the communication overhead via probing channels, and the visibility of VNFs over the system state. As depicted in Fig. 1, when a controller VM periodically (every 50ms) probes a cluster of servers[1] via TCP sockets, the latency overhead increases with the number of servers, which diminishes the QoS. As depicted in Fig. 2, the visibility of VNFs over the system state correlates with the probing frequency. Additional management traffic can exceed the 90-th percentile of per-destination-rack flow rate (100kbps) in production [2].

Flexibility vs. Performance: Developing, prototyping, and benchmarking ML applications on different networking problems is hard in high performance networks because of the low-latency and high-throughput expectation in the data plane. A general data processing framework has been proposed [24] to accelerate data-driven network functions on reconfigurable hardware, which provides line-rate performance. However, in dynamic and elastic networking environments where additions and removals of nodes and services happen frequently [27], hardware devices are scalable in terms of performance (*e.g.*, throughput) but not in terms of network topology (*e.g.*, number of services/nodes). Hardware programming and verification procedures can also be difficult and time-consuming for the ML community [60], which thus relies more on simulations for interdisciplinary research [10], [15], [61], [62]. Yet the flexibility offered by simulations hinders the real-world deployment of ML algorithms because simulators fail to capture the complexity of high performance networking systems [19].

B. Requirements

Based on the challenges, this paper summarises the following requirements to enable data-driven VNFs in the cloud:

Universality: the feature collection mechanism should cover a wide range of features and be application-agnostic;

Relevance: the collected features should be representative, providing useful information to address real-world applications in different circumstances;

Scalability: the feature collection and exploitation mechanism should incur minimal performance overhead and support large-scale and dynamically changing network topology;

Flexibility: the mechanism should be configurable and easy to be tailored for various use cases and learning algorithms;

Deployability: the mechanism should be plug-and-play and require no additional installation or configuration.

C. Related Work

Various mechanisms (summarised in Table I) dynamically configure and manage VNFs, making data-driven decisions.

ML benefits various networking applications, *e.g.*, congestion control [63], [64], intrusion detection systems [55], [65], traffic classification [66], [67], and task scheduling [7], [9]. It allows inferring system states from networking features. To obtain networking features, these ML applications operate at the Application Layer. However, they are not application-agnostic and do not generalise to different use cases. Acting as proxies, they also terminate networking connections, increasing processing latency [68], [69]. Aquarius collects a wide range of features at the Transport Layer and enables generic data-driven network functions with minimal overhead.

Management and Orchestration (MANO) frameworks use centralised controllers to monitor and update VNF configurations [32], [33]. Based on active monitoring, MANO helps provision computing, storage, and networking resources. Software-Defined Network (SDN) provides programmable APIs to gather per-flow or application-level features in a centralised way, to adaptively update configurations, using network equipment that supports the OpenFlow protocol [45], [46]. Other management protocols collect and send data from network equipment to a centralised controller via active probing, but with high communication overhead [47]–[49]. Aquarius passively extracts networking features from the data plane and lets VNFs make decisions in a distributed way.

Distributed VNFs also benefit from periodically polled network states (*e.g.*, packet arrival rates, CPU and memory usage), to ensure service availability, and improve QoS [12], [17], or classify networking traffic [34], [35]. Additional control messages and communication latency limit the system scalability [14], [36]. In [16], controllers are notified of the occurrence of malfunctioning nodes to avoid periodic probing. Some network functions gain more visibility via in-network telemetry (INT) [53] and covert-channels [70]. However, these require either deploying agents or modifying the protocol stack on network nodes, which reduces the deployability of data-driven mechanisms. Aquarius employs the plug-and-play design and requires no coordinated modification in the network.

Learning algorithms incur additional inference and processing latencies. To reduce latency, dedicated hardware, *e.g.*, NPU [71] and NetFPGA [57], helps improve data processing efficiency for in-network ML applications [72]. Taurus [24] enables in-network distributed data plane intelligence using a map-reduce abstraction for generic ML algorithms on a

[1] In the 69-byte control packet emitted by the server, the 24-byte payload consists of the server ID, CPU and memory usage, and the number of busy application threads.

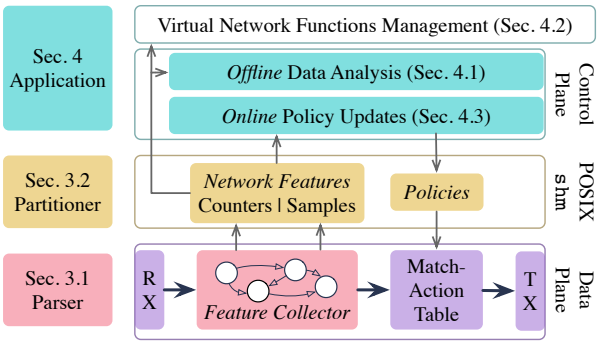


Fig. 3: Aquarius architecture overview.

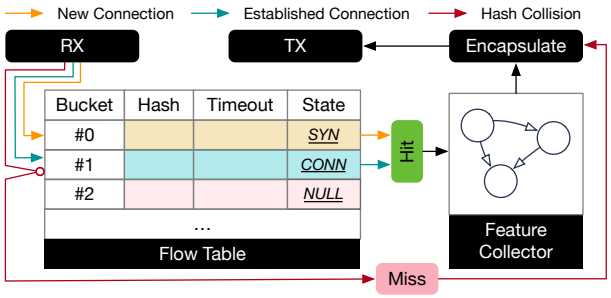


Fig. 4: Flow table data structure and workflow.

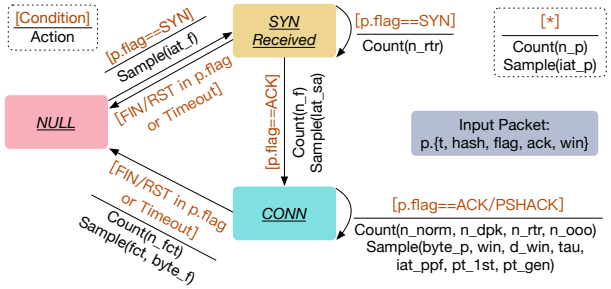


Fig. 5: A state machine of feature collector for TCP traffic.

coarse-grained reconfigurable array (CGRA) [73]. These hardware solutions boost performance, yet they lack flexibility when developing ML algorithms for different use cases in elastic networking systems. MVFST-RL [15] proposes to asynchronously update networking configurations to benefit from learning algorithms without inducing additional latency in the data plane. However, performance gains are shown only in simulators with a single use case. Aquarius can incorporate intelligence in a variety of VNFs, requiring no dedicated device, yet it is ready to be deployed in real-world systems.

III. Design

To meet the 5 requirements summarised in Section II-B, Aquarius is designed as a 3-layer architecture (Fig. 3). Aquarius embeds a feature collector at the Transport Layer in the data plane (*deployability*), to efficiently and passively extract a wide range of features (*universality*) with high quality (*relevance*), and low latency and performance overhead. It makes the features available and easily accessible via shared memory (*scalability*), for applications of ML algorithms on various use cases in the control plane (*flexibility*).

A. Parser Layer

To balance the tradeoff between scalability and visibility, networking features which indicate system states can be passively collected from the data plane to avoid active probing and additional installations and configurations. However, located within network function chains, VNFs in modern DCs may observe only 1-way traffic (*i.e.*, half-traffic of each flow) addressing to their egress equipment (*e.g.*, links and servers), to reduce additional processing latency [74]. This requires: (i) *careful design of feature collection mechanisms to offer high scalability and configurability*, and (ii) *domain knowledge to extract valuable and representative networking features and reason their correlations with system states*. This paper illustrates the design using TCP traffic, which is the most widely used protocol in the cloud [2], [75], [76]. The same workflow also applies to other network traffic (*e.g.*, UDP).

1) Stateful Feature Collection: Network traffic consists of flows that traverse different nodes (*e.g.*, edge routers, load balancers, servers) in the system, whose states can be traced and retrieved from the flows – along with traffic characteristics.

Stateless feature collection mechanisms – *e.g.*, sketching [20], [77], which is a family of streaming algorithms for networking measurement summarisations – do not track the state of network flows, yet they can gather counters as ordinal features for ML algorithms using hashing functions, with little performance overhead. However, ordinal features (counters) contain less information than quantitative features – time-related features (*e.g.*, round-trip time, inter-arrival time, flow duration) and throughput information (*e.g.*, congestion window size, flow size), which are not captured by stateless mechanisms.

Aquarius tracks flow states in bucket entries with a stateful table (Fig. 4), which can be configured to collect a wide range of features using a state machine depicted in Fig. 5. In the flow table, Aquarius stores the information of each flow into a bucket entry indexed by $hash(fid)\%M$, where fid is the flow ID[2] and M is the flow table size. An entry in the flow table can be in one of three states – `SYN`, `CONN` and `NULL` (Fig. 5). When a new flow arrives (TCP `SYN`), it is registered in a bucket entry of the flow table with its fid and state (`SYN`). On receipt of its subsequent packets, the state in the entry is retrieved and updated to `CONN` (connected) if the flow starts transmitting data[3]. On receipt of packets which terminate TCP flows (or timeout for UDP flows), the flow is evicted and the entry state returns to `NULL`, so that the bucket entry is available for new flows. In the case where the bucket entry is not available when a new flow appears, the flow is considered a "miss" and is excluded by the feature collector.

[2]TCP network flows are identified by their 5-tuples: protocol number, source and destination IP addresses and port numbers.

[3]For a TCP flow, if it is well-established (*e.g.*, after 3-way handshakes), and the first data packet is received, its state will be updated to `CONN`.

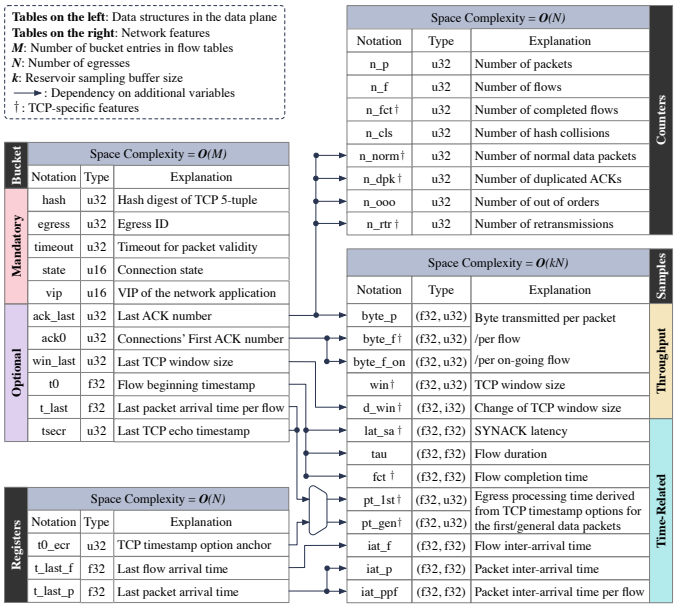


Fig. 6: Notations, categories, variable dependencies, and space complexity of all network features.

2) Network Features: Various features can gainfully benefit the decision making process for different use cases. Counting the number of ongoing flows helps track instant server load states, thus helping balance workloads distribution (Sec. IV-C). Throughput information helps classify whether the network traffic is IO-intensive (Sec. IV-A). Time-related features help understand the QoS on servers, thus helping predict resource utilisation and schedule scaling events (Sec. IV-B).

As a generic feature collection mechanism, Aquarius should be able to collect as much information as possible with minimal overhead (*e.g.*, memory space consumption). With the flow table, Aquarius allows flexible configuration of attributes, to gather the most significant features and optimise the memory usage overhead for different applications. Fig. 6 lists all configurable features that are implemented in this paper[4].

Ordinal features are collected as counters, which are incremented either as integer variables or using sketches. For simplicity, this paper uses accumulative integer variables, *e.g.*, when a flow state transits from `SYN` to `CONN`, the total number of received flows `n_f` is incremented[5]. Counters for general network protocols include:

1) *Number of packets and flows* (`n_p`, `n_f`), which quantifies the volume of network traffic addressed to each egress equipment.
2) *Number of hash collisions* (`n_cls`), which evaluates the amount of untracked connections and can be used to estimate the coverage of collected features.

[4]All features depend on the state and timeout attributes in the flow table, thus these dependencies are omitted for clarity. More attributes can be potentially added to obtain more features, *e.g.*, to track packet TTL.

[5]The counter `n_fct` is incremented only if one flow ends with a previous connection state as `CONN`. A similar DDoS mitigation mechanism based on flow tables is proposed in Prism [76], but it is out of the scope of this paper.

Algorithm 1 Reservoir sampling with no rejection

```
k ← reservoir buffer size
buf ← [(0,0), ..., (0,0)]                ▷ Size of k
for each observed sample v arriving at t do
    randomId ← rand()
    idx ← randomId%N                      ▷ randomly select one index
    buf[idx] ← (t, v)                     ▷ register sample in buffer
```

3) *Number of out-of-ordered packets (`n_ooo`)*, which indicates the multiple path existence in networks, where the packet ordering is not necessarily preserved.

For TCP traffic, additional counters can be gathered:

1) *Number of completed flows* (`n_fct`), which is incremented when flow terminates. The number of on-going connections (canonical feature) is derived as `#flows` $=$ `n_f` $-$ `n_fct`, to estimate instant queue lengths.
2) *Number of duplicated ACK packets, retransmissions* (`n_dpk`, `n_rtr`), which can be used for diagnostics, reflecting *e.g.*, the level of congestion on links.

Quantitative features are collected as samples, using reservoir sampling [78] (Algorithm 1). To capture the system dynamic, besides feature values, it is also important to trace the timestamps of different events, *e.g.*, for sequential ML algorithms. Reservoir sampling gathers a representative group of samples in fix-sized buffer from a stream with the sampling timestamps. For a Poisson stream of events with rate λ, the expectation of the amount of samples that are preserved in buffer after n steps is $E = \lambda \left(\frac{k-1}{k}\right)^{\lambda n}$, where k is the size of reservoir buffer. Based on the characteristics of different system dynamics, *e.g.*, long-term distribution shifts or short-term oscillations, the reservoir sampling mechanism can to be tuned (*e.g.*, number of buffers) to collect representative statistical distributions of the states over time, since both the sampling timestamps and exponentially-distributed numbers of samples are captured over a time window. Periodic queries to the reservoir sampling buffers can generate generic time-series data which is suitable for sequential pattern analysis.

For general network flows, the following features can be sampled in reservoir buffers:

1) *Bytes transmitted per packet (`byte_p`) and bytes transmitted per flow (`byte_f_on`)*, which help estimate overall IO occupation in the networks. Bytes transmitted per flow keep increasing as more data packets are received, until the flow ends or times out.
2) *Flows and packets inter-arrival time* (`iat_f`, `iat_p`), which reflect the arrival rates of flows thus the burst of network requests. The values of `iat_f` are updated when new flows arrive while `iat_p` requires no stateful tracking of connections.
3) *Flow duration* (`tau`), which helps characterize the type of network traffic, *e.g.*, long-lived flows or short queries. This feature is updated on receipt of each data packet of the flow.

For TCP traffic, additional features can be collected:

1) *Congestion window size* (`win`, `d_win`), which embed the congestion states of networking systems. Their val-

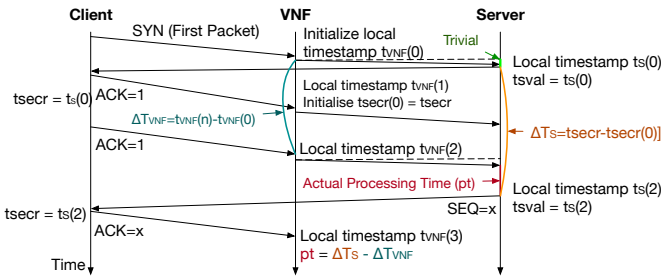


Fig. 7: Calculation of egress (*e.g.*, application server) processing time with TCP timestamp options.

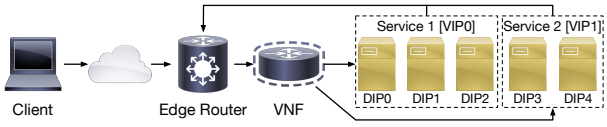


Fig. 8: Cloud service topology.

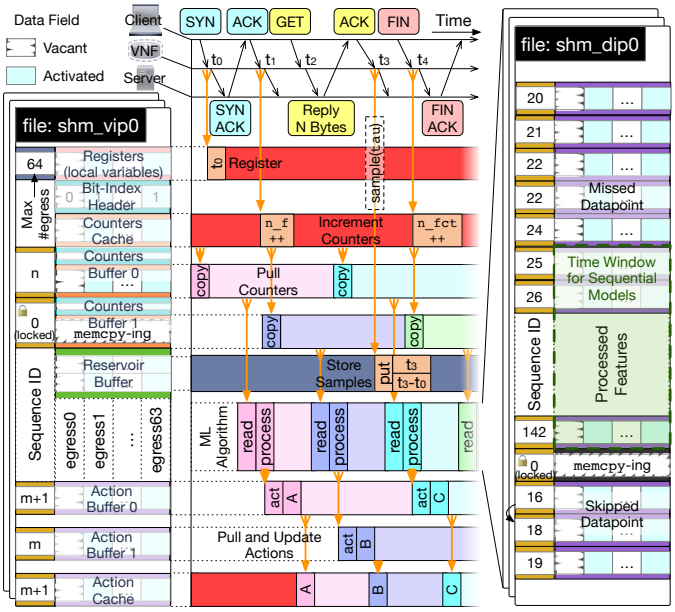


Fig. 9: Aquarius `shm` layout and data flow pipeline.

ues are updated on receipt of new ACK packets, after 3-way handshakes.

2) *Flow completion time* (`fct` and *flow byte size* (`byte_f`), which are collected when flows terminate. Their values indicate the characteristics of the managed services.
3) *SYN to first ACK latency* (`lat_sa`), which estimates the 3-way handshakes latency for TCP traffic. When a `SYN` packet is received on one host, `SYN` cookies statelessly generate a immediate `SYNACK` response. Their values help estimate and calibrate the baseline RTT between two end hosts of the connections.
4) *Data packet processing time* (`pt_1st` and `pt_gen`), which can be derived from TCP timestamp options `tsecr`. Intuitively, the time difference between the reception and the response of a data packet indicates the processing time and resource usage on the egress network equipment. However, given the constraint of observing only 1-way traffic on VNFs (*e.g.*, DSR mode for layer-4 LBs), this information is hard to obtain. Using the TCP option fields, the timestamp of the egress equipment's response (`tsval`) is recovered from the `ACK` packets sent by the client (`tsecr`). The procedure of rebuilding the processing time on the egress side is illustrated in figure 7. With respect to Web applications, the processing time is further distinguished by the first data packet (`pt_1st`) and the subsequent ones (`pt_gen`).

In-Network Telemetry (INT) features can also be collected as samples and stored in reservoir buffers [31], [79]. For simplicity, these features are omitted in this paper.

B. Partitioner Layer

Cloud services have different characteristics and they are identified by virtual IPs (VIPs) (Fig. 8), which correspond to clusters of provisioned resources – *e.g.*, servers, identified by a unique direct IP (DIP). In production, cloud DCs are subject to high traffic rates and their environments and topologies change dynamically. This requires to *organise collected features in a generic yet scalable format*, and *make features available for ML algorithms without disrupting the data plane*.

Different cloud services should be separated to (i) avoid multimodal distributions in collected features and (ii) allow dynamically adding or removing services. Based on use cases, features collected of a given service should be further partitioned – by ingress or egress equipment, *e.g.*, links, servers – to have higher granularity for learning algorithms. Even under heavy traffic and high access rates, features should be reliable and easy to access.

Aquarius organises observations of each VIP in independent POSIX shared memory (`shm`) files, to provide scalable and dynamic service management. In each `shm` file, collected features are further partitioned by egress equipment[6]. Fig. 9 exemplifies the `shm` layout and workflow.

1) Bit-Index and Masking: The first byte in the `shm` file of a VIP defines the max number of egress equipment N, which determines the number of "columns" to be reserved for feature collection. It is determined *a priori* by the scale of the cloud service, so that N equipment suffice the requirement in all circumstances. The N-bit *bit-index header* helps quickly identify activated egress and its corresponding "column" – the i-th bit is set to 1 if the i-th egress is active and 0 otherwise. With minimal memory space, this design informs ML algorithms to skip features of inactive equipment, gather features (*e.g.*, also in separated `shm` files) and update policies only for active equipment, reducing processing latency.

2) Multi-Buffering and Asynchronous I/O: While quantitative features are collected using reservoir sampling, counters are directly incremented by the data plane in the cache, and

[6]Depending on different applications, observations for each VIP can also be organised in different ways, *e.g.*, by ingress ports.

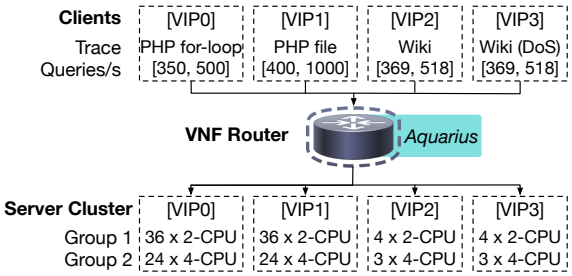


Fig. 10: Network topology for traffic classification.

then periodically drawn from cache to buffers with incremental sequence ID. The bit-index binary header is copied with the counters, to efficiently identify active equipment. When copying data between cache and buffer, the sequence ID is set to 0 to avoid I/O conflicts. ML algorithms can pull the latest observations from the buffers with no disruption in the data plane. Similarly, new network policies (*e.g.*, forwarding rules) can be updated via action buffers. This design offers an asynchronous 2-way communication interface to exchange fine-grained features extracted from data planes and data-driven decisions made by control planes with low latency.

C. Implementation

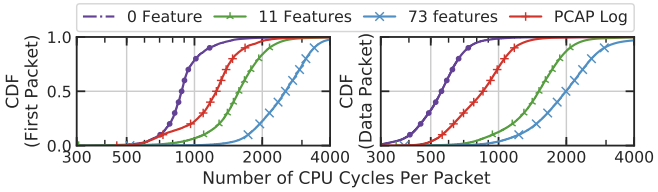
Aquarius is implemented as a plugin to the Vector Packet Processor (VPP) [80], a programmable network stack for commodity hardware. This paper sets $N = 64$ since it suffices for the typical configuration in production [81] and the 64-bit bit-index header fits in the cache line for modern computer processors. The flow table size is configured as $M = 65536$. The level of multi-buffering is set to 3 (same as in Fig. 9). The buffers draw the latest counters from the cache every 200ms (same as the active probing frequency in [12]). Each sampled network feature is a 2-tuple of a 32-bit float timestamp and a 32-bit value – fit in a single cache line. The reservoir buffer size is set to $k = 128$ for each feature per egress equipment. In these conditions, to collect all features listed in Fig. 6, the flow table takes 10.24MB of memory space. The `shm` file of each VIP consists of 6KB 3-level multi-buffering counters and 832KB reservoir sampling buffers.

IV. Applications

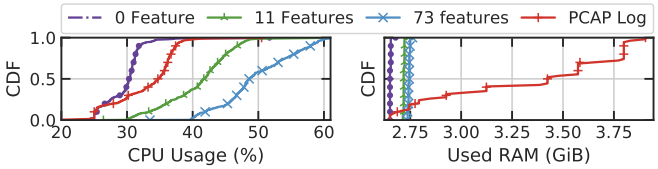
This section shows 3 applications of Aquarius in cloud DCs in the context of 3 key VNFs – traffic classification, resource prediction, and auto-scaling, and Layer-4 load balancing, along with 3 different ML paradigms.

A. Traffic Classification

As one of the key VNFs in the cloud, traffic classification allows distinguishing different types of traffic [34], [35], [55]–[57], to allocate appropriate resources and achieve service level agreements [34], [56]. It also helps detect anomalies and security threats to prevent potential damages or losses [55].



(a) Per-packet processing latency comparison.



(b) System resource consumption.

Fig. 11: Aquarius feature collection overhead.

Configuration		0 Feature	11 Features	73 Features	PCAP
First Packet	CPU Cycles	938.232	1635.838	2609.019	1295.284
	Delay (μs)	0.361	0.629	1.003	0.498
	Difference	1.000×	1.744×	2.781×	1.381×
Data Packet	CPU Cycles	576.357	1583.798	2602.684	885.041
	Delay (μs)	0.222	0.609	1.001	0.340
	Difference	1.000×	2.748×	4.516×	1.536×
CPU Usage (%)		26.687	40.858	49.716	31.480
CPU Difference		1.000×	1.376×	1.675×	1.060×
RAM Usage (GiB)		2.652	2.719	2.744	3.305
RAM Difference		1.000×	1.025×	1.034×	1.246×

TABLE II: Per-packet processing overhead (on 2.6GHz CPU) and system resource consumptions (avg.) comparison.

1) Task Description and Testbed Configuration: This section shows the capability of Aquarius to collect reliable features and conduct traffic classification with unsupervised ML algorithms. A testbed is implemented using Kernel-based Virtual Machine (KVM), where a virtual router embedded with Aquarius forwards different types of traffic to 4 VIPs (Fig. 10). In VIP0, a simple PHP `for`-loop script on each server takes requests for given number of iterations (`#iter`) and replies with proportional sizes. The flow duration (200ms on average) and number of transmitted bytes follow an exponential distribution as in [2]. In VIP1, static files of different sizes are served on each server[7] as in [17], to represent IO-bound applications. In VIP2 and VIP3, each application server is an independent replica of an Apache HTTP server [82] that serves Wikipedia databases. Two samples of 600s duration are extracted and replayed from a real-world 24-hour replay [83]. In VIP3, an additional 5000 queries per second SYN flooding traffic is applied to simulate a DoS attack. Server clusters are scaled to be able to serve all the queries under heavy traffic rates – when no attack happens – with reasonable FCT (under 400ms) as in [18].

2) Feature Engineering: The features are fetched every 250ms from counter buffers and reservoir buffers. This paper demonstrates the flexibility of feature engineering offered by samples collected in reservoir buffers, by reducing each feature channel to 5 scalars, *i.e.*, average, standard deviation, 90-

[7]The sizes of files are 100KB, 200KB, 500KB, 750KB, 1MB, 2MB, and 5MB. 50 files are generated for each size.

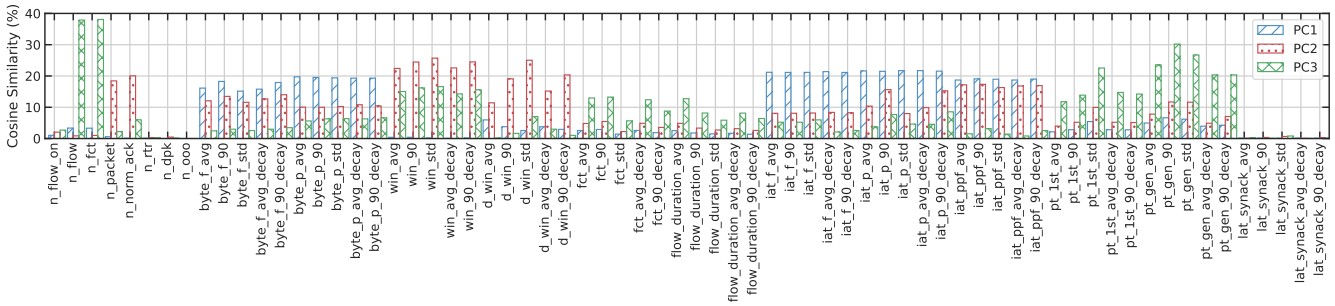


Fig. 12: Variance contribution of each feature in top-3 principal components (PCs).

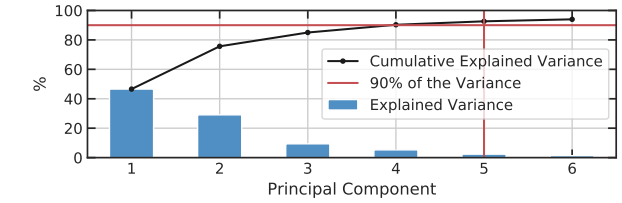
TABLE III: Comparison of (unsupervised) clustering algorithms for traffic classification.

Algorithm	KMeans	MeanShift	Spectral Clustering	Ward	Agglomerative Clustering	DBSCAN	OPTICS	BIRCH	Gaussian Mixture
Adjusted Rand Index	0.674	**0.863**	0.715	0.689	−0.002	0.696	0.757	0.742	0.687
Mutual Info Score	0.941	**1.160**	0.948	0.992	0.031	0.914	0.968	0.953	0.965
Adjusted Mutual Info Score	0.709	**0.820**	0.718	0.731	0.023	0.692	0.733	0.721	0.731
Homogeneity	0.713	**0.878**	0.718	0.752	0.024	0.692	0.733	0.722	0.731
Completeness	0.709	0.820	0.811	0.732	0.241	0.736	**0.914**	0.811	0.798
Fowlkes-Mallows Score	0.765	**0.901**	0.805	0.774	0.513	0.785	0.840	0.824	0.786
Fit Time (ms)	75.689	541.594	991.382	4806.554	2785.787	45.249	2769.734	52.959	**18.505**
Require Cluster Number	✓	✓	✓	✓	✓	✗	✗	✓	✓

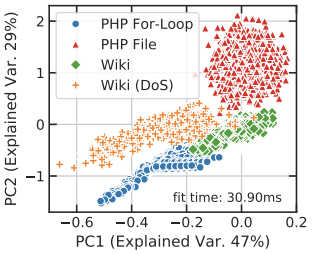
percentile, and exponential moving average (decay) of average and 90-percentile. The moving average is a sequential feature calculated, whose weight is computed as, $0.9^{t'-t}$, where t is the timestamp of each sample and t' is the moment when the reduced sample is calculated. This yields in total 8 ordinal features (counters) and 13×5 quantitative features[8].

3) Overhead Analysis: To study the feature collection overhead, Aquarius is compared with a vanilla router which collects 0 features and a router logging packet information in the memory using `pcap`. Under 500 queries/s PHP `for`-loop traffic towards a 176-CPU server cluster, when collecting 11 features[9] or collecting all 73 features[10], Aquarius incurs different overhead (Table II and Fig. 11a). On a 2.6GHz CPU, the additional per-packet processing delays are trivial compared with the typical round trip time (higher than 200μs) between network equipment [85]. The mean CPU usage of Aquarius is 1.376× and 1.675× higher than the vanilla router when collecting 11 and 73 features respectively (Fig. 11b). As expected, the log-based feature collection mechanism does not scale in terms of memory consumption[11].

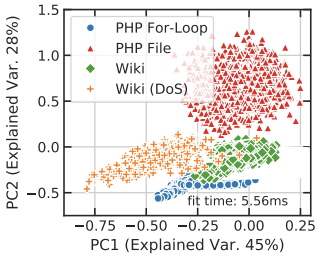
4) Feature Selection with PCA: More features give multi-dimensional observations, yet at the cost of higher computation and memory overhead. Principal Component Analysis (PCA) is thus conducted to understand the relative importance of the feature and reduce dimensionality while preserving data



(a) Percentage of explained variance.



(b) PCA clusters (all features).



(c) PCA clusters (25 features).

Fig. 13: PCA analysis and 2D visualisation.

representation. As depicted in Fig. 13a, 90% of the data variance can be explained with 4 principal components (PCs). Fig. 12 shows that multiple features share similar contributions (cosine similarity) to top-3 PCs, especially features reduced from the same reservoir buffer. Therefore, the number of features can be decreased by using only 2 (standard deviation and `decay`-ed average) out of the 5 reduced scalars. Also by removing sampled data that has low contribution to the top-4 PCs (*i.e.*, `iat_synack`), 25 features are selected out of all 73 features.

As depicted in Fig. 13b, 4 clusters for the 4 traces are

[8]The collected dataset is preprocessed and converted to have zero mean and unit standard deviation. Outlier data-points (value beyond 99th-percentile) are dropped. The data preparation procedure is done using `scikit-learn` [84] and it is the same throughout the whole paper.

[9]1 counter (n_flow_on) and 2×5 sampled features (flow duration, FCT).

[10]8 counters and 13×5 quantitative features.

[11]The results can be machine-dependent. This paper aims at showing the order of magnitudes, rather than providing a precise quantification.

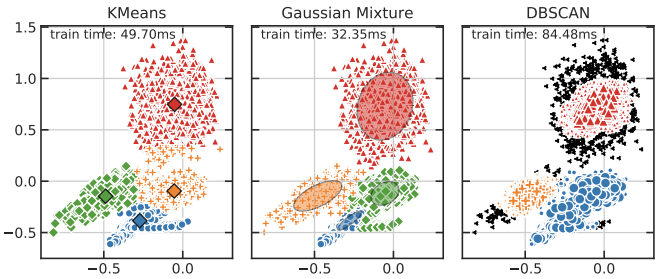


Fig. 14: Unsupervised clustering using 25 features.

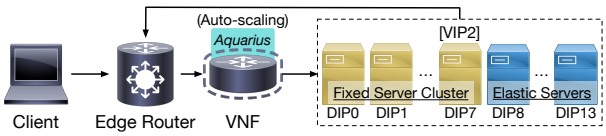


Fig. 15: Network topology for autoscaling system.

visualised in a 2D representation. Among the 4 traces, PHP `for`-loop is pure CPU-bound and PHP file is pure IO-bound. The `Wiki` trace consists of both queries for SQL database (CPU-bound) and static files (IO-bound), thus its cluster is located between the former 2 traces. The `Wiki` trace under DoS attack, however, can be clearly noticed as an independent cluster. As depicted in Fig. 13c, using the 25 selected features [12] still gives clear clustering results, yet it reduces data processing time from 30.90ms to 5.56ms.

5) Unsupervised Learning: 9 clustering algorithms are applied and compared over the obtained dataset. As in Table III, mean shift has the best overall performance, yet at the cost of relatively high fit time. As depicted in Fig. 14, when applying unsupervised learning algorithms, K-Means [86] and Gaussian Mixture [87] are able to generate clusters similar to the ground truth, while they require the number of expected clusters (4) as input. Gaussian Mixture model has the shortest fit time and can be an interesting candidate for online traffic classification. In case where the number of clusters is not known *a priori*, DBSCAN [88] can distinguish the potential security threat, based only on a predefined distance (0.1). With a training latency lower than 100ms, these algorithms can be interesting candidates for online traffic classification and anomaly detection systems. OPTICS [89] also achieves the highest completeness – all members of a given trace type are assigned to the same cluster, though with a much higher processing latency than DBSCAN.

Take-Away: Aquarius gathers fine-grained and reliable datasets, which allow feature engineering and conducting in-depth data analysis. Its fast and configurable design help achieve the right balance between visibility and performance.

[12]The input 25 networking features are: byte_f_std, byte_f_avg_decay, byte_p_std, byte_p_avg_decay, win_std, win_avg_decay, d_win_std, d_win_avg_decay, fct_std, fct_avg_decay, flow_duration_std, flow_duration_avg_decay, iat_f_std, iat_f_avg_decay, iat_p_std, iat_p_avg_decay, iat_ppf_std, iat_ppf_avg_decay, n_flow_on, n_flow, n_fct, n_packet, n_rtr, n_dpk, n_ooo.

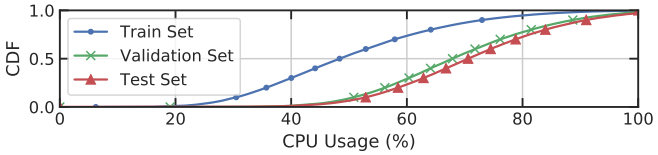


Fig. 16: Comparison of ground truth distributions.

B. Resource Prediction and Auto-Scaling

To minimize operational costs while guaranteeing QoS, cloud operators need to elastically provision server capacities. With growing interests in intelligent server capacity configurations [11], [50]–[52], this section shows the capability of Aquarius as a platform to systematically develop and adapt supervised ML algorithms to infer resource utilisation and performance with no actively signaling.

1) Task Description and Testbed Configuration: This section studies networking features and system utilisation under different levels of workloads, to avoid additional control messages in existing auto-scaling mechanisms [52]. 600s samples extracted from each hour of the real-world 24-hour Wikipedia trace are replayed on the network topology depicted in Fig. 15. Workloads are randomly distributed among running servers (2-CPU each) by way of Equal-Cost Multi-Path (ECMP). The server cluster requires $8 \sim 14$ servers to provide reasonable QoS (median FCT $\leq$ 400ms [18]). A learning task can be framed as predicting server load states (CPU usage[13]) on each server with the same set of features as in section IV-A. The predicted utilisation can be then used to plan and re-scale server clusters to guarantee QoS with reduced operational cost. This task consists of 2 steps – offline model training and online prediction. The first 23-hour samples are applied on 10 2-CPU servers to gather datasets for offline model training. The last-hour sample, which is not seen by any trained model, is synthesized to have 5 different levels of traffic rates for online prediction and real-time auto-scaling.

2) Offline Model Training: To predict the resource utilisation of server clusters using networking features, 12 widely used ML algorithms are selected to cover different families of ML algorithms, *e.g.*, sequential and non-sequential, parametric and non-parametric, linear and non-linear [19]. The dataset is pre-processed in the same way as described in section IV-A2. To adapt the dataset for sequential models, the sequence length (time steps) of input features is set as 32 and the stride as 16, which gives 50k data-points in total. These data-points are sequentially split $70 : 20 : 10$ into training, validation, and testing sets. The distribution of the ground truth CPU usages in the training set covers the other two datasets so that the prediction task is feasible, yet the ML models have not seen the datasets for evaluations (Fig. 16). Sequential models are created and trained using Keras with TensorFlow as backend [90]. Non-sequential models (built using scikit-learn)

[13]This paper uses CPU usage as the metric to evaluate and plan server cluster capacity for demonstration. The same methodology can be applied to problems using multi-variate metrics.

TABLE IV: Comparison of supervised ML algorithms for resource prediction (using selected *non-sequential* features to predict 8 steps ahead).

Algorithm	Linear Regression	Ridge Regression	Decision Tree	Random Forest	SVR (Linear)	SVR (RBF)	XGBoost	RNN	LSTM	GRU	GRU+ 1dConv	WaveNet	Active Probing
First Step MAE	9.216	9.232	12.135	8.717	9.255	9.040	8.544	7.629	7.433	7.488	7.492	7.830	**3.504**
First Step RMSE	11.779	11.763	15.558	11.125	11.876	11.395	10.871	9.591	9.403	9.478	9.481	9.770	**4.593**
Last Step MAE	10.640	10.638	15.043	11.009	10.683	11.206	10.797	9.774	9.855	9.798	10.001	**9.652**	11.892
Last Step RMSE	13.266	13.261	18.965	13.794	13.327	13.994	13.557	12.285	12.496	12.427	12.653	**12.194**	14.935
All Step Avg. MAE	10.038	10.044	14.109	10.090	10.063	10.335	9.891	8.986	9.046	9.022	9.123	9.010	**8.334**
All Step Avg. RMSE	12.575	12.575	17.866	12.742	12.616	12.963	12.512	11.331	11.505	11.464	11.594	11.390	**11.057**
Avg. Predict Time (ms)	1.429	1.375	1.765	152.558	629.212	1462.446	5.315	115.179	117.146	113.117	87.831	106.475	**0.026**
Predict Time Stdev. (ms)	0.380	0.294	0.013	0.305	0.093	2.015	0.288	1.961	4.100	3.592	1.680	1.968	**0.001**

TABLE V: Comparison of supervised ML algorithms for resource prediction (using selected *non-sequential* features to predict 16 steps ahead).

Algorithm	Linear Regression	Ridge Regression	Decision Tree	Random Forest	SVR (Linear)	SVR (RBF)	XGBoost	RNN	LSTM	GRU	GRU+ 1dConv	WaveNet	Active Probing
First Step MAE	9.186	9.206	12.134	8.697	9.232	8.999	8.486	7.577	7.379	7.458	7.496	7.527	**3.505**
First Step RMSE	11.715	11.723	15.538	10.991	11.838	11.302	10.716	9.515	9.414	9.559	9.578	9.595	**4.593**
Last Step MAE	10.858	10.861	14.976	11.129	10.897	11.322	10.964	9.794	9.935	9.579	9.786	**9.504**	12.238
Last Step RMSE	13.666	13.673	19.000	13.995	13.716	14.266	13.810	12.476	12.657	12.231	12.435	**12.130**	15.470
All Step Avg. MAE	10.399	10.404	14.587	10.555	10.424	10.776	10.362	9.342	9.478	9.165	9.374	**9.146**	10.216
All Step Avg. RMSE	13.039	13.046	18.393	13.262	13.077	13.525	13.035	11.852	12.069	11.700	11.933	**11.625**	13.335
Avg. Predict Time (ms)	2.689	2.659	3.557	304.490	1281.118	3026.451	10.445	96.102	120.756	111.489	89.297	105.831	**0.033**
Predict Time Stdev. (ms)	0.285	0.292	0.016	0.458	1.617	0.871	0.093	1.882	5.700	5.829	3.774	3.756	**0.014**

TABLE VI: Comparison of supervised ML algorithms for resource prediction (using selected *sequential* features to predict 8 steps ahead).

Algorithm	Linear Regression	Ridge Regression	Decision Tree	Random Forest	SVR (Linear)	SVR (RBF)	XGBoost	RNN	LSTM	GRU	GRU+ 1dConv	WaveNet	Active Probing
First Step MAE	9.219	9.223	12.419	8.892	9.241	8.953	8.758	8.205	7.842	7.622	7.785	8.040	**3.504**
First Step RMSE	11.543	11.553	15.745	11.328	11.582	11.288	11.141	10.370	10.012	9.716	10.005	10.207	**4.593**
Last Step MAE	10.658	10.662	15.023	10.840	10.689	10.855	10.704	9.787	9.627	**9.442**	9.566	9.550	11.892
Last Step RMSE	13.240	13.244	18.829	13.639	13.277	13.600	13.449	12.253	12.239	**11.933**	12.110	12.061	14.935
All Step Avg. MAE	10.059	10.063	14.077	10.073	10.074	10.056	9.950	9.272	9.091	8.855	8.949	9.027	**8.334**
All Step Avg. RMSE	12.528	12.534	17.772	12.743	12.552	12.647	12.585	11.646	11.564	11.225	11.379	11.433	**11.057**
Avg. Predict Time (ms)	1.394	1.390	1.727	150.036	604.762	1377.609	5.579	115.887	118.995	109.009	89.465	104.772	**0.022**
Predict Time Stdev. (ms)	0.301	0.316	0.009	0.186	0.023	0.056	0.243	3.537	2.673	4.954	3.413	2.584	**0.008**

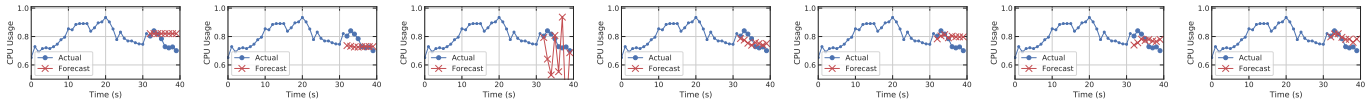
TABLE VII: Comparison of supervised ML algorithms for resource prediction (using selected *sequential* features to predict 16 steps ahead).

Algorithm	Linear Regression	Ridge Regression	Decision Tree	Random Forest	SVR (Linear)	SVR (RBF)	XGBoost	RNN	LSTM	GRU	GRU+ 1dConv	WaveNet	Active Probing
First Step MAE	9.205	9.210	12.514	8.841	9.229	8.911	8.712	8.339	7.889	7.863	7.855	8.149	**3.505**
First Step RMSE	11.527	11.537	15.799	11.158	11.567	11.162	10.964	10.375	9.990	10.115	10.014	10.353	**4.593**
Last Step MAE	10.823	10.828	15.019	11.064	10.856	11.054	10.895	9.993	9.699	**9.394**	9.564	9.447	12.238
Last Step RMSE	13.605	13.612	18.834	13.884	13.645	13.916	13.706	12.634	12.303	12.020	12.200	**12.002**	15.470
All Step Avg. MAE	10.412	10.417	14.576	10.478	10.431	10.444	10.337	9.711	9.220	**9.143**	9.316	9.148	10.216
All Step Avg. RMSE	13.003	13.010	18.328	13.177	13.029	13.103	12.998	12.181	11.661	11.668	11.852	**11.576**	13.335
Avg. Predict Time (ms)	2.547	2.522	3.464	298.758	1228.016	2832.406	10.699	95.248	114.952	111.944	89.810	105.492	**0.028**
Predict Time Stdev. (ms)	0.310	0.164	0.011	0.319	0.051	70.800	0.419	1.787	4.793	2.463	3.843	3.036	**0.007**

use the last time step features as input data. Each model is trained to predict the CPU usage multiple steps ahead.

ML Models: 6 non-sequential models are implemented using scikit-learn with their default hyperparameters, *i.e.*, linear regression, ridge regression, decision tree, random forest, SVM regression (SVR) with both linear and RBF kernel, and XGBoost. 5 sequential models are implemented using Keras, *i.e.*, RNN, LSTM, GRU, GRU with a 1-dimensional convolutional layer, and WaveNet. RNN has 2 20-hidden-unit SimpleRNN layers (first layer with return_sequence=True) and 1 output layer. LSTM replaces the SimpleRNN in the RNN model with LSTM layers and GRU replaces with GRU layers. GRU with 1d convolutional layer adds 1 1-dimentional CNN (as in textCNN) before the GRU model. Wavenet stacks 4 stacked dilated 1D convolutional layers with 1 layer of 20-hidden-unit GRU and 1 fully connected layers (output layer). As a benchmark, a naive model is implemented to simulate active probing by using the last observed CPU usage as predictions.

Feature Selection: To reduce input size, features are selected using sklearn.feature_selection.f_regression, in two different procedures, namely in a non-sequential and a sequential manner. In the non-sequential manner, the top 20-percentile features with the highest correlation with the CPU



(a) Active Probing (b) Linear Reg. (c) Decision Tree (d) Random Forest (e) LSTM (f) GRU (g) WaveNet

Fig. 17: Prediction results of 7 selected models using sequential features to predict 8 steps ahead.

Algorithm 2 Auto-scaling Rule

$n\_servers\_min, n\_servers\_max \leftarrow 8, 14$ ▷ Server number range
$\mathbb{S} \leftarrow$ Initial set of running servers
$cpu\_lo, cpu\_hi \leftarrow 0.7, 0.8$ ▷ Desired CPU usage range
for each time step **do** ▷ Δt = 250ms
  $\delta \leftarrow 0$ ▷ Initialize server state counter
  $y(\mathbb{S}) \leftarrow$ CPU usage prediction of 16 steps ahead
  $threshold \leftarrow \lceil \frac{|\mathbb{S}|}{3} \rceil$ ▷ Threshold that triggers scaling actions
  for $s \in \mathbb{S}$ **do**
    if $y(s) < cpu\_lo$ **then**
      $\delta++$ ▷ Increment δ if s is under-loaded
    else if $y(s) > cpu\_hi$ **then**
      $\delta--$ ▷ Decrement δ if s is over-loaded
  if $\delta > threshold$ **and** $|\mathbb{S}| > n\_servers\_min$ **then**
    $\mathbb{S} \leftarrow downscale(\mathbb{S})$
    skip 8 time steps ▷ Cool-down period
  else if $\delta < -threshold$ **and** $|\mathbb{S}| < n\_servers\_max$ **then**
    $\mathbb{S} \leftarrow upscale(\mathbb{S})$
    skip 8 time steps ▷ Cool-down period

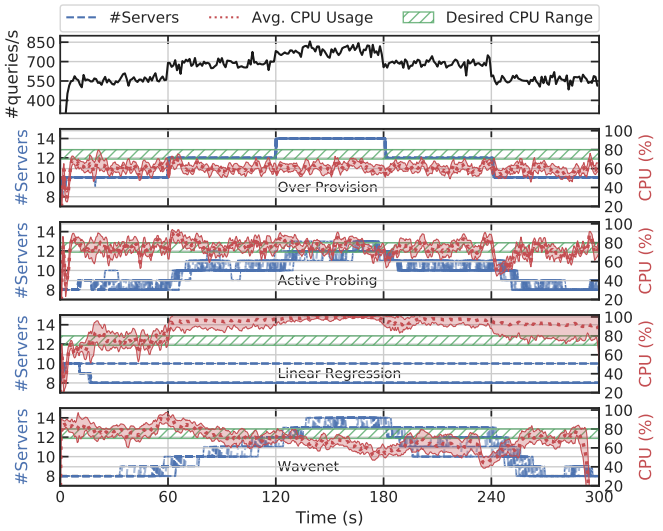


Fig. 18: Comparison of online auto-scaling performance using different algorithms. The (discrete) numbers of running servers are plotted for each run in dashed lines, while CPU usage is summarised as avg. $\pm$ stddev across 30 runs.

usage are selected[14]. In the sequential manner, networking features are first re-arranged by 32 time steps, then the features that appear more than 3 time steps in the top 20-percentile features with the highest correlation with the CPU usage, are selected[15].

Different Prediction Steps Ahead: The further in the future that one can predict, the better configuration plans can be made. Therefore, tasks are created to predict the different number of time steps ahead, namely 8 or 16 steps, to study the capabilities of predicting the future among different ML models.

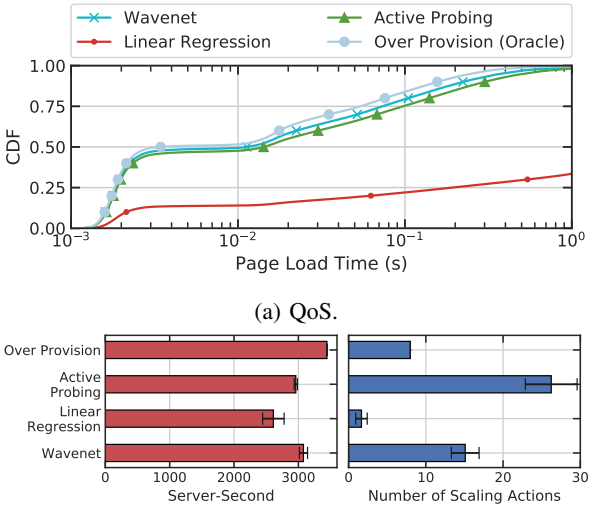
Results: Instances of the prediction results from a subset of ML models are visualised as in Fig. 17. The scores achieved by each predicting model using test set is shown in Table IV-VII. The prediction time for each model is evaluated using 256 datapoints (as predicting resource utilisation on 256 servers). The results show that sequential models achieve better performance when using sequential features as input data than using non-sequential features. Simple and non-sequential ML models perform worse that sequential models, especially when predicting 16 steps ahead as sequential models has more visibility on the history. WaveNet has the best overall performance and robustness across all 4 different tasks among all ML models, therefore it is chosen in this paper to be applied online. Linear regression, on the other hand, is the simplest ML model and has the shortest processing latency overhead when making prediction, therefore it is chosen to be applied online as well.

3) Online Auto-Scaling: To test the online performance of offline-trained ML models, a 300s Wikipedia replay trace sample of the last hour (unseen by the ML models) is synthesized to have scheduled changing traffic rates every 60s. Based on the 16-step-ahead CPU usage predictions of running servers $y(\mathbb{S})$, a simple heuristic is proposed (Algorithm 2) to keep the CPU usage of $\frac{2}{3}$ servers within the desired range ($70 \sim 80\%$). Using the same counter Δ for over- and under-loaded servers reduce the variance induced by imbalanced workload distributions. As a reference, an active probing mechanism is implemented, whose predicted CPU usage for running servers $y(\mathbb{S})$ comes from periodic polling (every 250ms, same as the prediction interval of ML methods). An "oracle" benchmark is implemented to over-provision the number of servers proportional to the scheduled traffic rates.

4) Results: As depicted in Fig. 18, active probing keeps the average CPU usage within the desired range, however, it requires frequent scaling events and leads to oscillating CPU usage with high variance. Linear regression is simple yet not robust when applied for an online auto-scaling system. Its under-estimated server load states lead to over-loaded servers.

[14]The 21 "non-sequential features" consist of: fct_90_decay, fct_avg_decay, fct_std, flow_duration_90, flow_duration_90_decay, flow_duration_avg, flow_duration_avg_decay, flow_duration_std, iat_f_avg, iat_f_avg_decay, iat_p_std, iat_ppf_90_decay, iat_ppf_avg, iat_ppf_avg_decay, iat_ppf_std, n_flow_on, pt_1st_90, pt_1st_90_decay, pt_1st_avg_decay, pt_1st_std, pt_gen_90_decay.

[15]The 15 "sequential features" consist of: n_flow, n_packet, iat_f_avg, iat_f_90, iat_f_std, iat_f_avg_decay, iat_f_90_decay, iat_p_avg, iat_p_std, iat_p_avg_decay, pt_1st_std, lat_synack_avg, lat_synack_90, lat_synack_90_decay, flow_duration_std.



(a) QoS.

(b) Operational cost and complexity.

Fig. 19: Trade-off between QoS and cost using different autoscaling mechanisms.

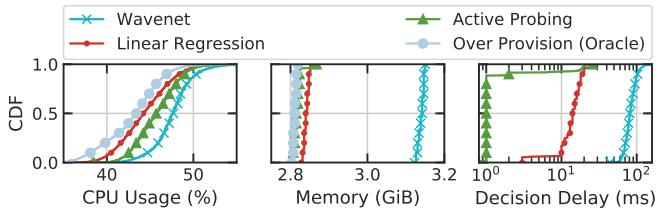


Fig. 20: Comparison of system overhead using different autoscaling mechanisms.

WaveNet takes sequential features as input and is more robust when applied online. It keeps the average CPU usage close to the desired range with less oscillations.

As depicted in Fig. 19, WaveNet is able to provide better QoS than active probing – 78.37ms less page load time (26.04%) at 90th percentile and 35.70ms less (30.45%) on average – with 3.99% additional server-second cost, and 42.44% less scaling events. When over-provisioning the server cluster, the page load time is shorter than using WaveNet by 67.13ms at 90th percentile and 28.55ms average, though it requires 11.86% more server-second operational cost than WaveNet.

5) Overhead Analysis: As depicted in Fig. 20, ML models incur additional memory usage and predicting delay. Active probing also incurs additional CPU usage. WaveNet, as a more sophisticated ML model, incurs 1.844% additional CPU usage and 322.61MiB additional memory usage than active probing. However, Aquarius parses features stored in the local shared memory with no control messages, achieving more than 94.18μs less median latency than typical VM- and container-based probing mechanisms (Fig. 21).

Take-Away: Aquarius enables agile development, offline model selection, and online deployment of learning algorithms to improve network performance. It makes features quickly accessible while saving management bandwidth for data transmission.

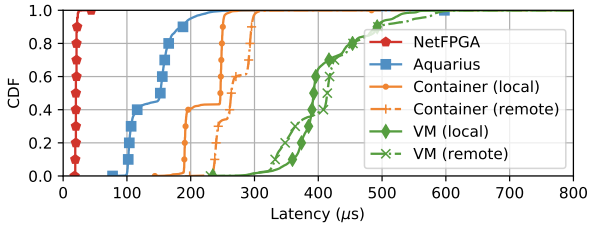


Fig. 21: Feature collection latency comparison between Aquarius and active probing techniques.

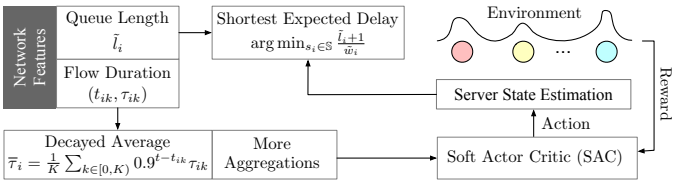


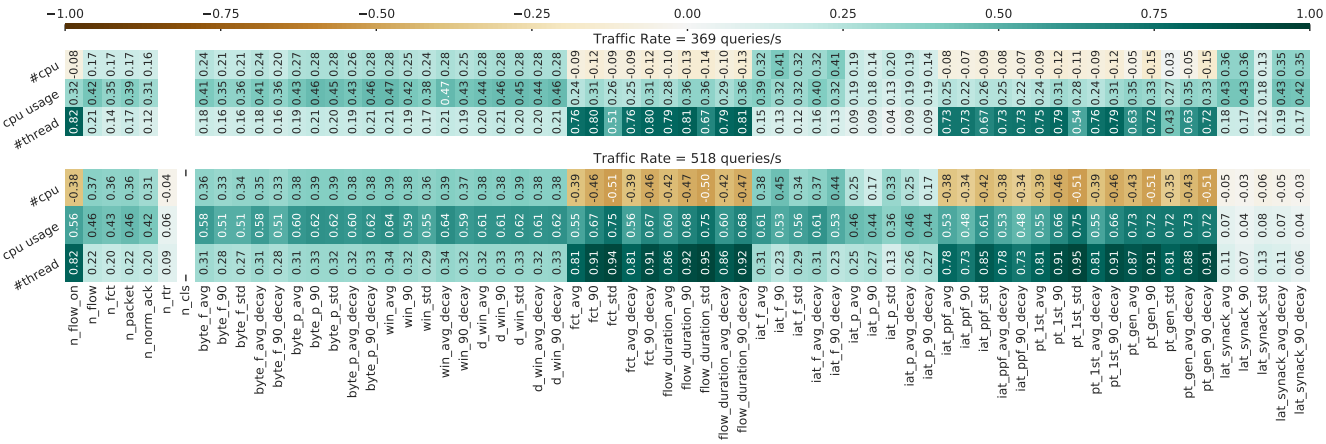
Fig. 22: Overview of the RLB algorithm [10].

C. Traffic Optimisation and Load Balancing

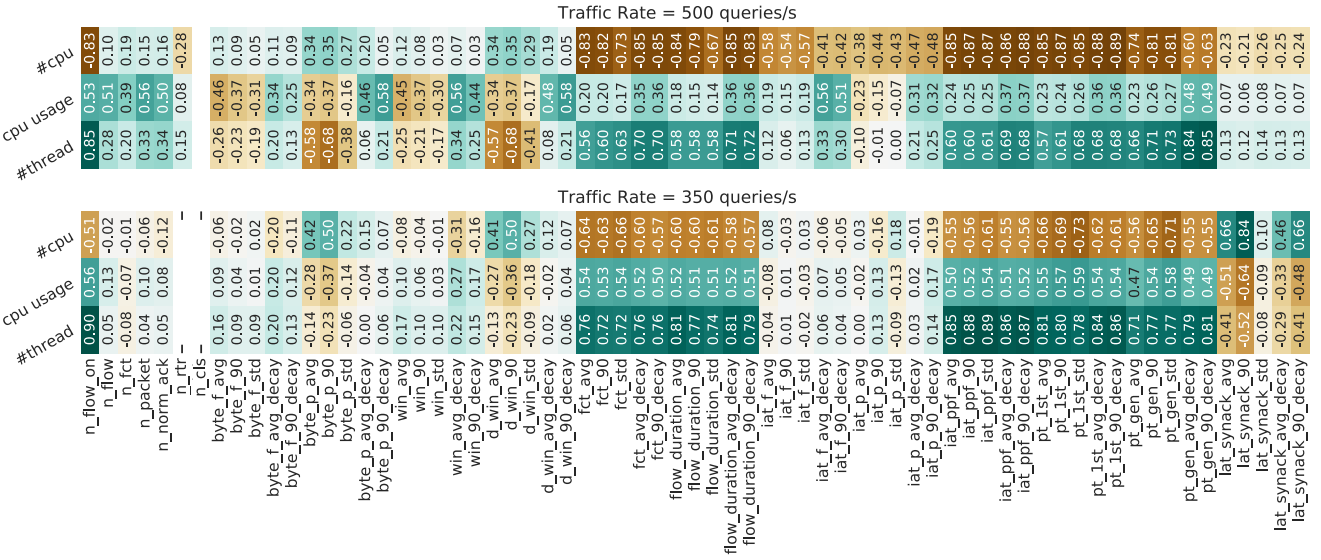
As a key component in cloud DCs, Layer-4 load balancers (LBs) distribute workloads across servers to provide scalable services. This section shows that Aquarius can apply RL algorithms to optimise load balancing performance.

1) Task Description and Testbed Configuration: In cloud DCs, servers can be virtualised on infrastructures with different processing speeds [91]. This section inherits the configuration of VIP2 (Fig. 10) – replaying the `Wiki` trace and load balancing on 2 groups of servers of different processing capacities. The task is to extract and infer server processing capacity information from networking features and make informed load balancing decisions. 3 benchmark LB algorithms are implemented – (i) ECMP [27] randomly distributes workloads regardless of server processing speed differences; (ii) WCMP [74], [92] statically assigns weights to servers based on their provisioned capacities; (iii) active WCMP [12], [16], [17] polls server job queue lengths and updates weights every 200ms based on probed utilisation information.

2) RL Algorithm: RLB [10] is an RL-based LB algorithm implemented and evaluated in simulators, similar to many RL algorithms for networking [15], [58]. In this paper, RLB is implemented and evaluated in a realistic testbed using Aquarius. As depicted in Fig. 22, with Aquarius, RLB (i) counts ongoing flows $\tilde{l}_i$ on servers and (ii) asynchronously updates (every 250ms) server weights $\tilde{w}_i$ (actions) for each application server as server load state estimations, derived from flow durations τ_i sampled in reservoir buffers as input features. The same architecture with a Soft Actor-Critic model as in [10] is implemented. However, the actor and critic networks take the batch-normalised features only based on locally observed per-server states. On receipt of new requests,



(a) VIP2 (Wikipedia trace).



(b) VIP0 (PHP for-loop).

Fig. 23: Correlation between networking features and server states.

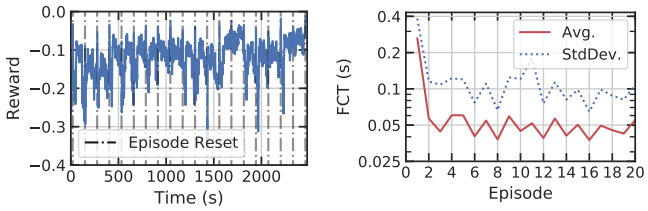
RLB assigns servers based on scores that estimate the time to finish all the workloads for each server using the shortest expected delay algorithm [93], *i.e.*, $\arg\min_i \frac{l_i+1}{\tilde{w}_i}$, which prioritizes servers with higher processing speed and shorter queue lengths. Different from [10], which uses actively probed ground truth information, this paper derives the reward from features collected by Aquarius. The reward is chosen as $\frac{\overline{\tilde{\tau}}^2}{\overline{\tilde{\tau}^2}} - 1$, where $\tilde{\tau}$ is a list of discounted average of flow duration on each server, which is also collected by Aquarius.

3) Feature Validation: RLB uses the number of ongoing connections to indicate server queue occupation and it uses flow duration as an input feature to infer server processing capacity. To verify the feature selection of RLB and study the correlation of network features with server load states in real-world networking systems, moderate and heavy network traces of both Wikipedia replay (VIP2 in Fig. 10)[16] and PHP for-loop (VIP0 in Fig. 10) are applied on the testbed. The correlation between all the features and server states under different traffic rates is depicted in Fig. 23.

As expected, one intuitive feature among the counters that helps infer server load state is the number of ongoing flows (n_flow_on). For VIP2, since the replayed trace is not IO-intensive – SQL queries with small file sizes whose average and standard deviation are both 12KiB, throughput-related features are indicative of the different provisioned server processing capacities (the number of CPUs `#cpu`). However, for both VIP0 (requests are CPU-intensive) and VIP2, latency-related features (*e.g.*, FCT, flow duration) show a higher correlation than achieved using active probing (Fig. 2), since they capture the fact that heavily loaded or less powerful servers have slow processing speeds. This effectively shows that networking features passively gathered by Aquarius are reliable and the selected input features of RLB are representative.

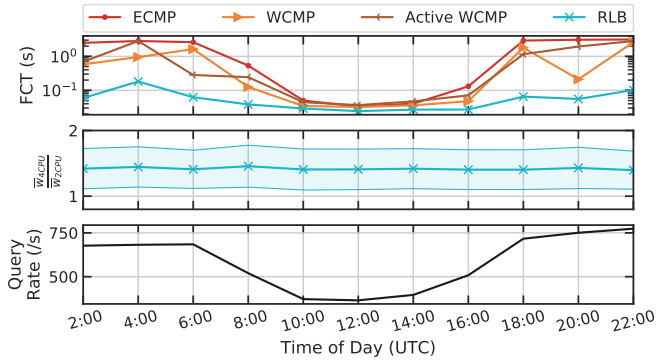
4) Results: RLB is trained using the first hour of `Wiki` trace sample for 20 runs (episodes). As depicted in Fig. 24, RLB learns server capacity differences. The rewards of RLB

[16]Using ECMP, which does not distinguish the server processing capacity difference, an average FCT of 45ms and 836ms is achieved respectively under light and heavy traffic.

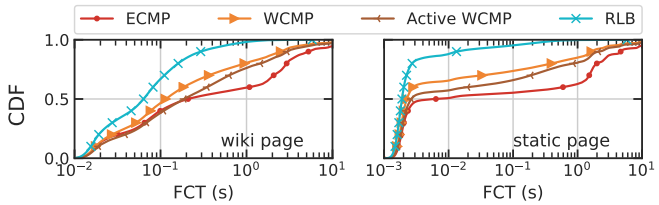


(a) RLB reward. (b) FCT.

Fig. 24: Evaluation during 20-episode training.



(a) Mean FCTs (top), the ratio between weights assigned to the 2 groups of servers by RLB (middle), and traffic rates (bottom).



(b) Peak-hour (query rate higher than 500/s) FCT distribution.

Fig. 25: Wikipedia trace replayed using different LBs.

during training grow higher and less variant, and the FCT becomes lower. The trained RLB model is then tested on unseen traffic and compared with other LB algorithms (Fig. 25a). During off-peak hours, servers are under-utilised and all algorithms show similar performances. As traffic rates grow, RLB achieves lower FCT for both static pages and Wikipedia pages when compared with other LB algorithms (Fig. 25b). RLB is trained to learn server processing speed differences and assigns higher weights, thus more queries, to more powerful servers (Fig. 26). When using RLB, 4-CPU servers handle respectively $1.258\times$ and $1.523\times$ more tasks than 2-CPU servers under 676.92 and 372.01 queries/s traffic.

5) Overhead Analysis: As depicted in Fig. 27a, throughout all test runs, RLB consumes on average 692.89 more CPU cycles (0.26μs on 2.6GHz CPU) than ECMP, as it computes and compares the server scores when making load balancing decisions. Fig. 27b depicts CPU and memory consumptions of all LBs. RLB incurs $0.22\times$ additional CPU usage and 45.99MiB memory usage on average.

6) Partial Observation: Though RLB achieves better performance than heuristic load balancing methods, it relies on

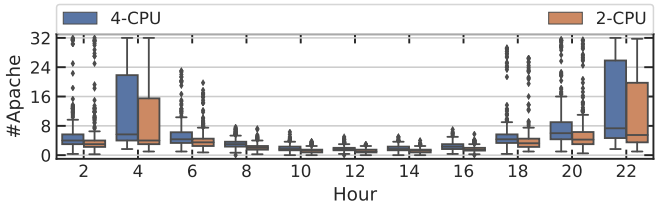
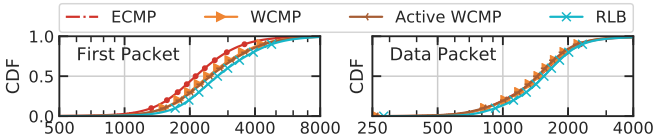
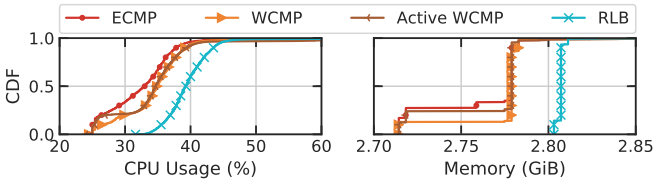


Fig. 26: Query distribution (number of busy Apache threads) on 2 groups of application servers.



(a) Per-packet processing latency comparison.



(b) System resource consumption.

Fig. 27: Overhead comparisons.

ordinal features, which, collected in a distributed system, risk reflecting only partial system states. For instance, when traffic is split across multiple load balancers, the locally counted number of ongoing flows does not reflect the actual queue length on the application server. As is depicted in Fig. 29, in presence of 2 load balancers, the ratio of the locally observed number of flows over actual queue length #thread has a standard deviation of 22.49%. However, RLB relies also on latency-related features (flow duration), which can be gainfully used to infer server load states and compensate for the impacts of partially observed ordinal features.

Take-Away: Aquarius enables closed-loop control (RL) to dynamically adapt to networking systems and optimise performance. It empowers real-world deployment and evaluation of learning algorithms developed in simulated environments.

V. Conclusion

Networking features and system state information help VNFs make informed decisions, and intelligently manage and update networking policies in cloud DCs. Actively collecting features and system state information entails substantial control signaling and management overhead, in particular in large-scale DC networks. This paper has proposed Aquarius, a framework that collects, infers, and supplies accurate networking state information with little additional processing latency, in a scalable buffer layout. By using multi-buffering and reservoir sampling, Aquarius extracts representative features from network traffic, and allows VNFs – in particular ML-based VNFs – to exploit these features. Aquarius can be deployed

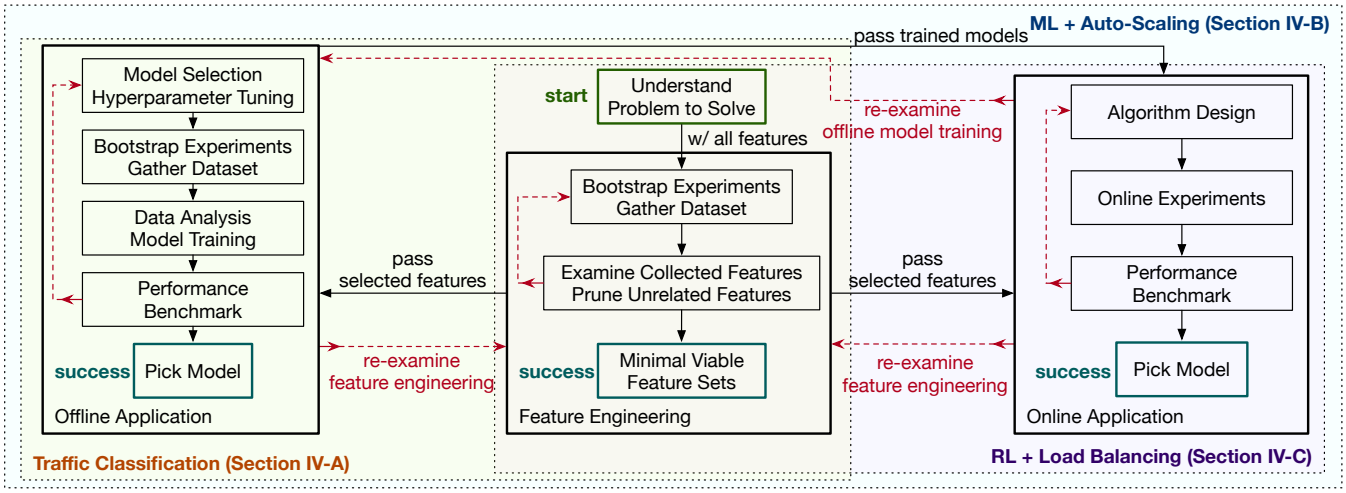


Fig. 28: Methodology blueprint. The application of ML techniques starts from understanding the problem to solve, including *e.g.*, the objectives and constraints. Aquarius provides high configurability and programmability to conduct extensive and iterative feature engineering and selection process to prune unrelated features (reduce additional feature processing overhead) and to pick a minimally viable set of features that can be gainfully used for solving the target problem. The selected features can be passed to both offline application (*e.g.*, clustering algorithms + traffic classification in Section IV-A) and online application (*e.g.*, RL + load balancing in Section IV-C). Offline trained ML models can also be brought online to evaluate their performance in real-time (*e.g.*, supervised learning + autoscaling in Secion IV-B). As a platform that helps harness reliable networking features and learning algorithms, Aquarius allows iteratively investigating networking features, developing models, and designing algorithms.

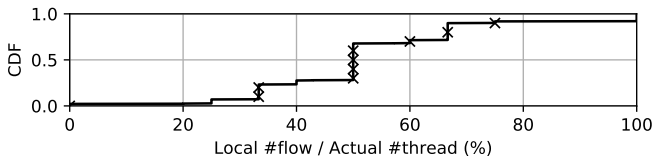


Fig. 29: Partial observations happen when traffic is split across 2 VNFs.

in the network on commodity CPU, empowering real-world learning algorithm deployments and evaluations. Following the methodology blueprint summarised in Fig. 28, this paper has illustrated the use of Aquarius for various ML-based VNFs: traffic classification (offline, unsupervised learning), autoscaling (online, supervised learning), and load-balancing (reinforcement learning) purposes, and evaluates experimentally the impact of Aquarius in the system performance. The application of ML techniques to networking problems starts from understanding the target problem to solve. Aquarius improves the visibility on the data plane and allows collecting a wide range of networking features for feature engineering, which iteratively prunes unrelated features to reduce additional feature collection processing latencies and selects the minimal set of viable features that can be gainfully used for the task. The selected features can be passed to both offline and online applications for data analysis, model training, and benchmark evaluations. Aquarius provides a reliable feature collection and experimenting platform in real-world systems that allows iteratively studying model selection, parameter tuning, and algorithm design for various use cases. Both open-loop (*e.g.*, supervised learning + autoscaling system) and close-loop (*e.g.*, RL + load balancer) control can be achieved based on Aquarius to improve resource orchestration and utilisation. Extensive evaluations show that Aquarius helps bring significant performance gains (reduced FCT, improved resource utilisation) in the three considered cases of data-driven VNFs.

REFERENCES

[1] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, and L. Mao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," p. 10, 2019.

[2] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. ACM, 2015, p. 123–137, event-place: London, United Kingdom.

[3] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[5] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *ACM Sigcomm Demo*, vol. 4, no. 5, p. 6, 2009.

[6] R. Wang, D. Butnariu, J. Rexford *et al.*, "Openflow-based server load balancing gone wild." *Hot-ICE*, vol. 11, pp. 12–12, 2011.

[7] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," *arXiv preprint arXiv:1810.01963*, 2018.

[8] S. Naseer, Y. Saleem, S. Khalid, M. K. Bashir, J. Han, M. M. Iqbal, and K. Han, "Enhanced network anomaly detection based on deep neural networks," *IEEE access*, vol. 6, pp. 48 231–48 246, 2018.

[9] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 191–205.

[10] Z. Yao, Z. Ding, and T. H. Clausen, "Reinforced workload distribution fairness," in *5th Workshop on Machine Learning for Systems at 35th Conference on Neural Information Processing Systems (NeurIPS 2021)*, 2021.

[11] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacgümüş, "Smartsla: Cost-sensitive management of virtualized resources for cpu-bound database services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1441–1451, 2015.

[12] A. Aghdai, C.-Y. Chu, Y. Xu, D. H. Dai, J. Xu, and H. J. Chao, "Spotlight: Scalable transport layer load balancing for data center networks," *arXiv preprint arXiv:1806.08455*, 2018.

[13] M. Usama, J. Qadir, A. Raza, H. Arif, K.-L. A. Yau, Y. Elkhatib, A. Hussain, and A. Al-Fuqaha, "Unsupervised machine learning for networking: Techniques, applications and research challenges," *arXiv preprint arXiv:1709.06599*, 2017.

[14] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu, "A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 393–430, 2018.

[15] V. Sivakumar, T. Rocktäschel, A. H. Miller, H. Küttler, N. Nardelli, M. Rabbat, J. Pineau, and S. Riedel, "Mvfst-rl: An asynchronous rl framework for congestion control with delayed actions," *arXiv preprint arXiv:1910.04054*, 2019.

[16] A. Aghdai, M. I.-C. Wang, Y. Xu, C. H.-P. Wen, and H. J. Chao, "In-network congestion-aware load balancing at transport layer," *arXiv preprint arXiv:1811.09731*, 2018.

[17] J. Zhang, S. Wen, J. Zhang, H. Chai, T. Pan, T. Huang, L. Zhang, Y. Liu, and F. R. Yu, "Fast switch-based load balancer considering application server states," *IEEE/ACM Transactions on Networking*, p. 1–14, 2020.

[18] Y. Desmouceaux, P. Pfister, J. Tollet, M. Townsley, and T. Clausen, "6lb: Scalable and application-aware load balancing with segment routing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 819–834, 2018.

[19] S. Fu, S. Gupta, R. Mittal, and S. Ratnasamy, "On the use of ml for blackbox system performance prediction." in *NSDI*, 2021, pp. 763–784.

[20] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.

[21] Z. Yao, Z. Ding, and T. Clausen, "Multi-agent reinforcement learning for network load balancing in data center," 2022. [Online]. Available: https://arxiv.org/abs/2201.11727

[22] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[23] M. Ahmed, A. N. Mahmood, and J. Hu, "A survey of network anomaly detection techniques," *Journal of Network and Computer Applications*, vol. 60, pp. 19–31, 2016.

[24] T. Swamy, A. Rucker, M. Shahbaz, and K. Olukotun, "Taurus: An intelligent data plane," *arXiv preprint arXiv:2002.08987*, 2020.

[25] "Amazon elastic compute cloud," https://aws.amazon.com/ec2/.

[26] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer." in *NSDI*, 2016, pp. 523–535.

[27] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. ACM, 2017, p. 15–28, event-place: Los Angeles, CA, USA.

[28] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 125–139.

[29] J. T. Araújo, L. Saino, L. Buytenhek, and R. Landa, "Balancing on the edge: Transport affinity without network state," 2018, p. 111–124.

[30] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 503–514.

[31] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "Hpcc: high precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 44–58.

[32] OPNFV, "Open Platform for NFV (OPNFV) Project Portal," https://www.opnfv.org/, 2019.

[33] OpenStack, "OpenStack Project Portal," https://www.openstack.org/, 2019.

[34] Y. Jie, Y. Lun, H. Yang, and L.-y. Chen, "Timely traffic identification on p2p streaming media," *The Journal of China Universities of Posts and Telecommunications*, vol. 19, no. 2, pp. 67–73, 2012.

[35] K. Lalitha and V. Josna, "Traffic verification for network anomaly detection in sensor networks," *Procedia Technology*, vol. 24, pp. 1400–1405, 2016.

[36] R. Cziva and D. P. Pezaros, "Container network functions: Bringing nfv to the network edge," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 24–31, 2017.

[37] K. Shafi and H. A. Abbass, "Evaluation of an adaptive genetic-based signature extraction system for network intrusion detection," *Pattern Analysis and Applications*, vol. 16, no. 4, pp. 549–566, 2013.

[38] G. Creech and J. Hu, "Generation of a new ids test dataset: Time to retire the kdd collection," in *2013 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2013, pp. 4487–4492.

[39] "Predict," https://www.predict.org/.

[40] "NSL-KDD," http://nsl.cs.unb.ca/NSL-KDD/.

[41] "CAIDA," https://www.caida.org/.

[42] "Internet Traffic Archive," http://ita.ee.lbl.gov/.

[43] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Io-lite: a unified i/o buffering and caching system," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 1, pp. 37–66, 2000.

[44] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *Proceedings of the 32nd international conference on Very large data bases*, 2006, pp. 607–618.

[45] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "Policycop: An autonomic qos policy enforcement framework for software defined networks," in *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*. IEEE, 2013, pp. 1–7.

[46] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[47] M. Hutter, A. Szekely, and J. Wolkerstorfer, "Embedded system management using wbem," in *2009 IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, 2009, pp. 390–397.

[48] J. Yu, H. Lee, M.-S. Kim, and D. Park, "Traffic flooding attack detection with snmp mib using svm," *Computer Communications*, vol. 31, no. 17, pp. 4212–4219, 2008.

[49] P. Gonçalves, J. L. Oliveira, and R. L. Aguiar, "An evaluation of network management protocols," in *2009 IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, 2009, pp. 537–544.

[50] D. Minarolli and B. Freisleben, "Distributed resource allocation to virtual machines via artificial neural networks," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2014, pp. 490–499.

[51] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, pp. 3–14.

[52] T. Chen, R. Bahsoon, and X. Yao, "A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–40, 2018.

[53] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford, "Clove: Congestion-aware load balancing at the virtual edge," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, 2017, pp. 323–335.

[54] Z. Yao, Y. Desmouceaux, M. Townsley, and T. H. Clausen, "Towards intelligent load balancing in data centers," in *Machine Learning for Systems at 35th Conference on Neural Information Processing Systems (NeurIPS 2021)*, 2021.

[55] A. Tuor, S. Kaplan, B. Hutchinson, N. Nichols, and S. Robinson, "Deep learning for unsupervised insider threat detection in structured cybersecurity data streams," *arXiv preprint arXiv:1710.00811*, 2017.

[56] A. Rizzi, A. Iacovazzi, A. Baiocchi, and S. Colabrese, "A low complexity real-time internet traffic flows neuro-fuzzy classifier," *Computer Networks*, vol. 91, pp. 752–771, 2015.

[57] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *Proceedings of the 18th ACM workshop on hot topics in networks*, 2019, pp. 25–33.

[58] Y. Xu, W. Xu, Z. Wang, J. Lin, and S. Cui, "Load balancing for ultra-dense networks: A deep reinforcement learning based approach," *IEEE Internet of Things Journal*, vol. 6, no. 6, p. 9399–9412, Dec 2019, arXiv: 1906.00767.

[59] D. Zhou, Z. Yan, Y. Fu, and Z. Yao, "A survey on network data collection," *Journal of Network and Computer Applications*, vol. 116, pp. 9–23, 2018.

[60] N. Schottelius, "High speed nat64 with p4," Master's thesis, ETH Zurich, 2019.

[61] F. Ruffy, M. Przystupa, and I. Beschastnikh, "Iroko: A framework to prototype reinforcement learning for data center traffic control," *arXiv preprint arXiv:1812.09975*, 2018.

[62] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *International Conference on Machine Learning*. PMLR, 2019, pp. 3050–3059.

[63] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 123–134, 2013.

[64] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "Pantheon: the training ground for internet congestion-control research," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 731–743.

[65] M. K. Putchala, "Deep learning approach for intrusion detection system (ids) in the internet of things (iot) network using gated recurrent neural networks (gru)," Ph.D. dissertation, Wright State University, 2017.

[66] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE communications surveys & tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[67] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar, "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1988–2014, 2018.

[68] M. A. Qadeer, A. Iqbal, M. Zahid, and M. R. Siddiqui, "Network traffic analysis and intrusion detection using packet sniffer," in *2010 Second International Conference on Communication Software and Networks*. IEEE, 2010, pp. 313–317.

[69] A. Papadogiannakis, D. Antoniades, M. Polychronakis, and E. P. Markatos, "Improving the performance of passive network monitoring applications using locality buffering," in *2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2007, pp. 151–157.

[70] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, M. Townsley, and T. Clausen, "Stateless load-aware load balancing in p4," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Sep 2018, p. 418–423.

[71] J. Papamichael, T. M. M. Liu, D. Haselman, L. A. M. Ghandi, S. Sapek, and G. W. L. Woods, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture, ser. ISCA*, vol. 18, 2018.

[72] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," *arXiv preprint arXiv:1903.06701*, 2019.

[73] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[74] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 207–218, 2013.

[75] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, "MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking," in *ACM CoNEXT '10*, Philadelphia, PA, December 2010.

[76] R. Cohen, M. Kadosh, A. Lo, and Q. Sayah, "Lb scalability: Achieving the right balance between being stateful and stateless," *arXiv:2010.13385 [cs]*, Oct 2020.

[77] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[78] J. E. Eckel, "Reservoir sampling," Jan. 7 1958, uS Patent 2,819,038.

[79] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, "In-band network telemetry: A survey," *Computer Networks*, vol. 186, p. 107763, 2021.

[80] The Fast Data Project (fd.io), "Vector Packet Processing (VPP)," https://wiki.fd.io/view/VPP.

[81] Facebook Engineering, "Reinventing Facebook's data center network," https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/, Mar 2019.

[82] "The Apache HTTP server project," http://www.apache.org.

[83] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009.

[84] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[85] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 139–152.

[86] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," Stanford, Tech. Rep., 2006.

[87] D. A. Reynolds, "Gaussian mixture models." *Encyclopedia of biometrics*, vol. 741, no. 659-663, 2009.

[88] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: why and how you should (still) use dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017.

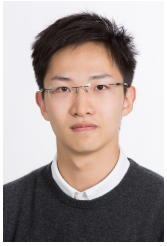
[89] E. Schubert and M. Gertz, "Improving the cluster structure extracted from optics plots," in *LWDA*, 2018.

[90] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," https://www.tensorflow.org/, 2015, software available from tensorflow.org.

[91] A. Kumar, I. Narayanan, T. Zhu, and A. Sivasubramaniam, "The fast and the frugal: Tail latency aware provisioning for coping with load variations," in *Proceedings of The Web Conference 2020*, 2020, pp. 314–326.

[92] "Katran," https://github.com/facebookincubator/katran, Apr 2020.

[93] W. Zhang *et al.*, "Linux virtual server for scalable network services," in *Ottawa Linux Symposium*, vol. 2000, 2000.



Zhiyuan Yao received the M.Sc.T from École Polytechnique, Palaiseau, France, in 2019. He is currently pursuing an industrial Ph.D. jointly between École Polytechnique's networking team and Cisco Systems PIRL, under supervision of Mark Townsley (Cisco Systems) and Thomas Clausen (École Polytechnique). His research interests include high-performance networking, load-balancing, data-center optimization algorithms, and machine learning for systems.

Yoann Desmouceaux received the Diplôme d'Ingénieur from École Polytechnique, Palaiseau, France, in 2014, the M.Sc. degree in Advanced Computing from Imperial College, London, U.K., in 2015, and the Ph.D. degree in computer networking from Université Paris-Saclay, France, in 2019. He is currently working as a Software Engineer with Cisco Systems. His research interests include high-performance networking, IPv6-centric protocols, load-balancing, reliable multicast, and data-center optimization algorithms.

Juan-Antonio Cordero-Fuertes is an associate professor at École polytechnique. He graduated in Mathematics ("Licenciatura", M.Sc) and Telecommunication Engineering (B.Sc+M.Sc, "Ingeniería Superior") at the Technical University of Catalonia (UPC, Spain) in 2006 and 2007, respectively. He got his Ph.D. at École polytechnique in 2011, with a dissertation on the optimization of link-state routing protocols for operation in MANETs and compound (wired/wireless) Autonomous Systems. He was a postdoctoral researcher at the Université catholique de Louvain (UCL, Belgium) and the Hong Kong Polytechnic University (Hong Kong SAR, PRC), before joining faculty at École polytechnique, in 2016. His research and scientific interests include routing protocols and information dissemination algorithms, and the modeling, analysis and optimization of distributed, adaptive systems in dynamic, heterogeneous networking scenarios.

Mark Townsley is a Cisco Fellow, Professor Chargé de Cours at École Polytechnique, and co-founder of the Paris Innovation and Research Laboratory (PIRL). Before Joining Cisco in 1997, he held positions at IBM, the Institute for Systems Research (ISR) and the Center for Satellite and Hybrid Communications Networks (CSHCN) at the University of Maryland. Mark served as IETF Internet Area Director from 2005-2009, IETF L2TP Working Group Chair from 1999-2005, IESG Liaison to the Internet Architecture Board (IAB), and IETF Pseudowire WG Technical Advisor. Mark was the lead developer of the original implementation of L2TP in Cisco IOS as well as lead author of IETF L2TP protocol specification (RFC 2661). One of the original architects of the World IPv6 Day and Launch, Mark contributed significantly to the deployment of IPv6 on the internet, including lead author of RFC 5969, IPv6 Rapid Deployment (6RD). In 2011, Mark co-founded the IETF Homenet Working Group, and served as chair until 2017. In addition to his Faculty appointment at École Polytechnique, Mark lectures on Future Internet Architectures at Telecom Paris Tech (TPT), and serves on the steering committee for the joint TPT-Polytechnique Advanced Computer Networking master's degree. Mark holds a Bachelor of Science (summa cum laude) degree in Electrical Engineering from Auburn University and a Master's degree in Computer Science (magna cum laude) from the Johns Hopkins University Applied Physics Laboratory.

Thomas Clausen is a graduate of Aalborg University, Denmark (M.Sc., PhD – civilingeniør, cand.polyt), and has, since 2004 been on faculty at Ecole Polytechnique, France's premiere technical and scientific university, where as a professor, he holds the Cisco endowed "Internet of Everything" academic chaire. At Ecole Polytechnique, Thomas leads the computer networking research group. He has developed, and coordinates, the computer networking curriculum, and coordinates the M.Sc.T programme "IoT: Innovation and Management". He has published more than 100 peer-reviewed academic publications, and has authored and edited 24 IETF Standards. Thomas has also consulted for the development of IEEE 802.11s, and has contributed the routing portions of the ITU-T G.9903 standard for G3-PLC networks – upon which, e.g., the current SmartGrid & ConnectedEnergy initiatives are built. Thomas is a senior member of the IEEE, and was named an "IEEE Computer Society Distinguished Contributor", as part of the 2021 inaugural class.