DRST: a Non-Intrusive Framework for Performance Analysis in Softwarized Networks

Qiong Liu, Jianke Lin, Tianzhu Zhang, Leonardo Linguaglossa

DRST: a Non-Intrusive Framework for Performance Analysis in Softwarized Networks

Qiong Liu[b], Jianke Lin[b], Tianzhu Zhang[c], Leonardo Linguaglossa[a]

[a] *LTCI, Telecom Paris, Institut Polytechnique de Paris, 91120 Palaiseau, France*
[b] *ETIS UMR 8051, CYU, CNRS, ENSEA, 95000 Cergy, France*
[c] *Nokia Bell Labs, 12 Rue Jean Bart, 91300 Massy, France*

Abstract

The last decade has witnessed the proliferation of network function virtualization (NFV) in the telco industry, thanks to its unparalleled flexibility, scalability, and cost-effectiveness. However, as the NFV infrastructure is shared by virtual network functions (VNFs), sporadic resource contentions are inevitable. Such contention makes it extremely challenging to guarantee the performance of the provisioned network services, especially in high-speed regimes (e.g., Gigabit Ethernet). Existing solutions typically rely on direct traffic analysis (e.g., packet- or flow-level measurements) to detect performance degradation and identify bottlenecks, which is not always applicable due to significant integration overhead and system-level constraints. This paper complements existing solutions with a lightweight, non-intrusive framework for online performance inference that easily adapts to *drift* (i.e., a change over time of the actual state of our system). Instead of direct data-plane collection, we reuse hardware features in the underlying NFV infrastructure, introducing negligible interference in the data-plane. Our Drift-Resilient and Self-Tuning (DRST) framework can be integrated into existing NFV systems with minimal engineering effort and operates without the need for predefined traffic models or VNF-specific customization. DRST is deployed via a lightweight MLOps pipeline that automates the adaptation under runtime drift. We show how DRST can deliver accurate performance inference or diagnose run-time bottleneck diagnose, as demonstrated through comprehensive evaluation across diverse NFV scenarios.

Keywords: Network function virtualization, performance prediction, drift detection, non-intrusive, MLOps.

1. Introduction

In recent years, Network Function Virtualization (NFV) and Software-Defined Networking (SDN) have accelerated a shift in telco industry from proprietary, monolithic hardware appliances to agile, software-based functions deployable on commodity servers. Meanwhile, high-speed I/O technologies, such as Intel Data Plane Development Kit (DPDK) [1], Mellanox Messaging Accelerator (VMA) [2], netmap [3], extended Berkeley Packet Filter (eBPF) [4], and Snabb [5], have greatly enhanced the capabilities of software packet processing. As a result, modern software stacks can now attain high performance regimes as 10/40/100 Gbps, which were previously exclusive to traditional hardware middleboxes [6]. These advances in network softwarization are vital for communication service providers and cloud data centers, where performance, scalability, and cost-efficiency underpin sustained growth and operational value [7].

Despite their popularity, softwarized networks also face some intrinsic obstacles in practice. In particular, compared to traditional hardware middleboxes that utilize dedicated circuits for packet processing, the software data plane is more susceptible to performance impairments due to the shared nature of the underlying virtual infrastructure [8] and various bottlenecks [9]. The colocated VNFs can compete for subsystem resources, which severely degrades the quality of network services. Therefore, performance prediction and analysis constitute the first fundamental step to fulfilling SLAs [10]. Existing solutions mainly combine direct feature collection with statistical reasoning [11, 12, 13] or machine learning [14, 15, 16] for fine-granular, accurate performance analysis in high-speed networks. However, these solutions are not always applicable due to: (i) the potentially substantial instrumentation overhead, especially for VNFs from heterogeneous vendors [17], (ii) the collateral interference on the high-speed data plane, which can damage both network performance and data fidelity [18], (iii) inflexible inference without trade-offs in predictivity, inference latency, and interpretability, which are crucial CSP requirements [19], (iv) the ensuing data drift and model decay as the network system evolves, making it strenuous for heuristic and data-driven methods to sustain accuracy [20].

This paper complements existing solutions with a novel framework for performance intelligence in high-speed softwarized networks. Instead of relying on direct traffic measurement measurement for data collection, we leverage the

Email addresses: qiong.liu@ensea.fr (Qiong Liu), jianke.lin@cyu.fr (Jianke Lin), tianzhu.zhang@nokia-bell-labs.com (Tianzhu Zhang), linguaglossa@telecom-paris.fr (Leonardo Linguaglossa)

low-level hardware features of different subsystems, e.g., CPU pipeline, multi-level caches, Random Access Memory (RAM), and Input/Output (I/O). These features are ubiquitous in modern COTS servers and can be acquired with standard profiling tools [21, 22, 23]. Although less relevant than the packet-/flow-level statistics, these features embody rich run-time network information and can be combined with data-driven analytics to deliver actionable insights [24, 25, 26]. We employ advanced ML algorithms for performance prediction (e.g., step-ahead KPI forecasting) and analysis (e.g., bottleneck detection). The main contributions of this work are as follows:

- *Non-intrusive data collection*: Our framework leverages the low-level hardware features for analytics, which incurs negligible overhead on the data plane.
- *Seamless integration*: Our framework can seamlessly integrate with existing NFV systems without prior domain knowledge and extra engineering overhead.
- *Model benchmarking and robust selection*: We evaluate several ML architectures across heterogeneous scenarios and retain those that balance accuracy and latency.
- *End-to-end pipeline with DRST*: We implement our solution as an end-to-end ML pipeline with Drift Resilient and Self-Tuning (DRST) capabilities, enabling continuous monitoring and adaptive retraining to sustain accuracy under data drift.

This paper is organized as follows: We first review relevant background and prior work in Section 2, then outline our motivations in Section 3. The proposed system design is described in Section 4, and experimental performance is evaluated in Section 5. Section 6 concludes the paper.

2. Background

2.1. High-speed softwarized networks

Traditionally, software-based networking solutions, such as the Click Modular Router [27], have been primarily used for fast prototyping and functional testing due to their unparalleled accessibility, flexibility, and customizability. However, specialized hardware equipment (or middleboxes) stood out in real-world deployments thanks to their far superior packet processing capabilities. In recent years, the rapid development of software acceleration techniques, such as kernel-bypass, poll-mode, batch processing, and parallel computing, has significantly narrowed the "performance gap" between software solutions running on COTS servers and specialized middleboxes [7]. Nowadays, software packet processing is an integral part of the modern telco industry [1].

However, softwarized networks still bear several inherent limitations. Co-located Virtual Network Functions (VNFs) suffer performance impairments due to the erratic

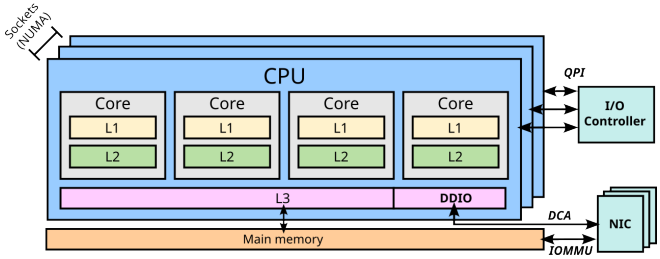


Figure 1: The architecture of a modern COTS server.

contentions in the shared NFV infrastructure [10]. Consequently, network operators are forced to spend a lot of time pinpointing and resolving performance issues [11]. Such problems are intricate to predict due to the voluminous and heterogeneous traffic therein. The growing complexity of the NFV systems and network services further compounds the situation [28]. As modern COTS servers continue to gain new functionalities, the software data plane can encompass numerous configuration knobs, including hardware options and software parameters. This vast search space makes it extremely challenging to anticipate and prevent performance contentions. In parallel, network service structures are transforming beyond the conventional linear service function chains (SFCs), and many research efforts strive for enhanced service provisioning by parallelizing VNFs as Directed Acyclic Graphs (DAGs) [17, 29, 30, 31]. As detailed in [29], 53.8% of VNF pairs in enterprise networks were parallelizable. Such flexible compositions increase the difficulty of diagnosing performance issues across service and infrastructure layers [11]. In essence, there is an urgent need for novel approaches to accurately predicting performance degradations and identifying the root causes of complex network services.

2.2. Resource contention in software data plane

Modern COTS servers rely on multi-socket NUMA architectures, where each socket has access to local memory and I/O buses to reduce latency from remote accesses. To support efficient memory usage and reduce bottlenecks, such systems adopt a hierarchical cache design. Fig. 1 shows that each CPU contains multiple cores, with its own build-in L1 and L2 caches. Cores located within the same NUMA node share the same L3 cache with a much larger size than L1/L2 caches. Incoming packets at the Network Interface Controller (NIC) may be routed directly to the main memory through the Input-Output Memory Management Unit (IOMMU) or to the L3 cache via Direct Cache Access, e.g., the Intel® Direct Data I/O (DDIO). Cores on different NUMA nodes can only communicate via specialized interconnect, such as the Intel® QuickPath Interconnect (QPI).

Given the complex layout of various components and the intricate interactions of the COTS servers, software packet I/O operations can still suffer from various contentions:

- *CPU share*: The allocated CPU share directly decides how fast packets can be processed. Despite the high frequency of modern CPUs, performance variance can still emerge due to dynamic frequency scaling (e.g., Intel® Turbo boost). Also, non-NFV workloads can be allocated to a VNF's cores due to misconfigured scheduling policies, leading to performance losses. CPU isolation mechanisms, e.g., Linux `isolcpus`, can only alleviate the issue.

- *Multi-level caches*: While cache accesses are way faster than main memory, prior works have widely deemed multi-level caches the major performance bottleneck [32]. Many NFV frameworks streamline packet processing across multiple cores, which can cause severe contention for the Last-level caches (LLCs). Cache partitioning techniques, such as Intel® Cache Allocation Technology (CAT), cannot always prevent such contentions, e.g., large incoming packets can contend for DDIO, which is referred to as the leaky Direct Memory Access (DMA) problem [33]. As the caching systems of modern CPU micro-architectures differ across generations, joint optimization of LLC and DDIO remains a daunting task, especially in cloud data centers [34].

- *Memory bandwidth*: Contention for memory bandwidth further slows the packet path. For instance, VNFs with lower LLC shares can incur high cache misses, which saturate the memory bandwidth for all the co-located VNFs and network services [35].

Other bottlenecks also exist. For instance, packet I/O across multiple NUMA nodes can be extremely slow due to the QPI contention [36], which can be avoided with a NUMA-aware design.

2.3. Related work

Existing solutions commonly employ direct, per-packet measurement for data and feature collection, as well as performance analysis. For instance, NFVPerf [37] applied packet mirroring for feature collection. PPTMon [38] employed event filtering and timestamp embedding to monitor the processing latency of VNFs. However, these works involve expensive operations in the software data plane and cannot cater to high-speed networks that handle millions of packets per second. Such tools fail to handle the huge input load in high-speed networks. For example, the throughput can reach 14.88 Mpps with the end-to-end network service latency in sub-microseconds for 64-byte synthetic packets on a 10 Gbps link. Some solutions were designed for high-speed regimes but were subject to enormous integration overhead, huge resource footprint, or operational constraints. For instance, NFV-VIPP [39] captured the execution states of DPDK-augmented VNFs but required manually attaching the per-VNF threads, a nontrivial operational exertion in production networks. Additionally, each monitoring thread must occupy one CPU core, which was costly given the limited number of cores on COTS servers [40, 41, 42]. Microscope [11] deduced performance bottlenecks via the runtime queuing states, which were not always obtainable in real networks [43]. Moreover, as VNFs can originate from heterogeneous vendors, code instrumentation and customization are necessary, which can be highly burdensome due to the diversified implementation paradigms and operational patterns [17].

To circumvent these obstacles, a line of work explored the low-level features and data-driven algorithms to derive performance insights. In particular, Dobrescu et al. [32] extrapolated the performance of software data planes using the cache features. Shelbourne et al. [25, 24] inferred throughput and packet losses for DPDK-based VNFs by exploring a larger set of low-level features. Still, they only analyzed the impact of input traffic without considering other system-level performance interferences. Antonis et al. [10] developed SLOMO, a multivariable performance prediction framework to investigate common system-level contentions and employ gradient-boosting regression to predict service throughput. Although these works delivered promising outcomes, they were primarily designed for singleton VNFs, without considering end-to-end network services (e.g., SFCs) that comprise multiple VNFs with varying topological compositions. Furthermore, these solutions were only tested in controlled environments. They did not consider the practical challenges of deploying AI/ML in real network systems with much higher scale, complexity, and dynamism [19], which makes it hard to achieve long-term, sustainable accuracy. In particular, given the data-driven nature of ML-based solutions, fulfilling performance guarantees in the presence of data and environmental drifts is non-trivial [20]. For the sake of clarity, we summarize in Table 1 the main acronyms used in the remainder of this paper.

3. Motivation

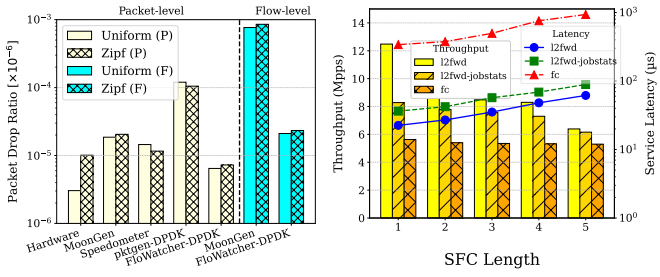
This section presents an observational study on non-intrusive data collection. We first compare the overhead of direct versus indirect measurements, and then analyze how low-level features relate to KPIs under varied traffic and service settings.

3.1. Direct vs. Indirect overhead

To evaluate the runtime overhead introduced by different monitoring approaches, we compare direct measurements (e.g., packet- and flow-level analysis) with indirect measurements via hardware performance counters. To illustrate the overhead of per-packet data collection, we consider the de facto test scenarios for network measurement: open-loop and closed-loop [44]. For the open-loop test, we generate 64B of traffic at 10 Gbps and deploy four state-of-the-art network measurement tools, i.e., MoonGen [45], Speedometer [46], pktgen-DPDK [47], and FloWatcher-DPDK [18], to measure the throughput at the receiving

Acronym	Definition
COTS	Commodity Off-The-Shelf
DAG	Directed Acyclic Graph
DDIO	Direct Data I/O
DirREC	Direct Recursive Forecasting Strategy
DMA	Direct Memory Access
DRST	Drift-Resilient and Self-Tuning
DPDK	Data Plane Development Kit
eBPF	extended Berkeley Packet Filter
KPI	Key Performance Indicator
IOMMU	Input–Output Memory Management Unit
LSTM	Long Short-Term Memory
LLC	Last-Level Cache
MANO	Management and Network Orchestration
MAPE	Mean Absolute Percentage Error
MLP	multi-layer perceptron
NIC	Network Interface Controller
NFV	Network Function Virtualization
PTP	Precision Time Protocol
PCM	Performance Counter Monitor
PMU	Performance Monitoring Units
RAM	Random Access Memory
VNF	Virtual Network Function
SDN	Software-Defined Networking
SFC	Service Function Chain
SHAP	SHapley Additive exPlanations

Table 1: List of acronyms



(a) Open-loop overhead (b) Closed-loop overhead

Figure 2: Overhead of direct vs. indirect measurements

end. MoonGen and FloWatcher-DPDK are also configured to collect per-flow statistics (e.g., flow size and inter-arrival gaps). The synthetic traffic consists of 60K flows, whose sizes are configured to follow Uniform and Zipf distributions. As illustrated in Fig. 2a, per-packet measurement incurs non-negligible overhead, even if we only access the NIC's counters (i.e., Hardware in the figure). When we proceed to perform packet- and flow-level measurements, the overhead only increases. Although the packet loss ratios seem small (10^{-5}-10^{-4}), this can already cause severe issues, given the high input rates of millions of packets per second in high-speed networks.

We deploy a sample SFC comprising identical VNFs for the closed-loop test, each performing Layer 2 packet forwarding. The VNFs are containerized with Docker and interconnected using FastClick [48]. We chose three VNFs from the DPDK example library, i.e., l2fwd, l2fwd-jobstats, and flow classification (fc). l2fwd performs simple forwarding. l2fwd-jobstats and fc further collect packet- and

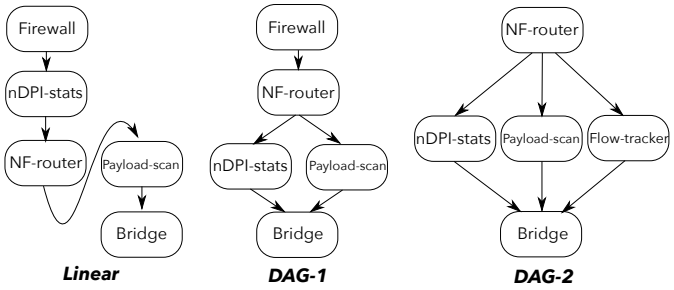


Figure 3: Three typical network service topologies

flow-level statistics, respectively. Similar to the open-loop measurement, we continuously inject 64B of synthetic traffic to the SFC at the line rate and measure the throughput. We also send Precision Time Protocol (PTP) packets to measure the end-to-end latency. The SFC length is varied from 1 to 5 [1]. As illustrated in Fig. 2b, per-packet measurement causes significant performance degradation. In particular, fc causes the throughput to drop by up to 50% while extending the latency by one order of magnitude.

To demonstrate the advantage of our approach, we repeat both experiments, but run *perf* to collect low-level features every 100 ms. Hardware counters reflect averaged or accumulated metrics over each sampling interval. Note that shorter intervals may be necessary for detecting transient phenomena such as jitter or microbursts, they are less critical for our target of sustained performance inference in NFV environments. To test different levels of interactions, we sequentially assign perf to the same worker cores as the VNFs, to different idle cores on the same NUMA node, and to cores on the other NUMA node. In all cases, the perceived throughput and latency remain the same.

Observation 1. *Indirect measurement via PMUs provides a non-intrusive and low-overhead alternative to conventional direct measurement approaches.*

3.2. Feature relevance analysis: KPIs

Perf exposes hundreds of hardware features, but many are noisy for ML. We must identify a refined subset of expressive features with strong predictive power.

To identify the relevant hardware features, we construct three typical network service topologies (or SFCs): a linear service chain and two DAGs, as shown in Fig. 3. The VNFs (Firewall, nDPI-stat, NF-router, Bridge, Payload-scan, and Flow-tracker) are implemented and open-sourced by ONVM developers. We then inject traffic at random rates and inspect the tendencies of the features. To represent typical Internet traffic, we configure MoonGen to generate IMIX traffic consisting of a variety of packet sizes in the ratio 64B: 570B: 1514B = 7: 4: 1. We collect the low-level features

[1] Note that the performance deteriorates as the SFC gets longer, mainly due to the accumulated memory copying and inter-core communication overhead [41].

Features \ VNF	Bridge	Payload-scan	NF-router	nDPI	Firewall	Average
LLC-load	0.94	0.98	0.97	0.98	0.96	0.97
Cache-reference	0.95	0.97	0.97	0.98	0.97	0.94
LLC-stores	0.97	0.96	0.96	0.97	0.97	0.97
L1-Dcache-load-misses	0.95	0.97	0.97	0.98	0.97	0.97
Instructions	0.79	0.92	0.86	0.78	0.89	0.92
Branches	0.79	0.92	0.87	0.79	0.89	0.83
Mem-stores	0.39	0.50	0.42	0.90	0.53	0.55
Cache Misses	0.32	0.18	0.35	0.65	0.36	0.38
Cycles	0.14	0.08	0.14	0.14	0.06	0.12

(a) Throughput

Features	Average
LLC-load	0.59
Cache-reference	0.58
LLC-stores	0.57
L1-Dcache-load-misses	0.55
Instructions	0.48
Branches	0.47
Mem-stores	0.11
Cache Misses	0.13
Cycles	0.02

(b) Latency

Table 2: Correlated features with throughput and latency under load stimulus

Features \ VNF	Bridge	Payload-scan	NF-router	nDPI	Firewall	Average
LLC-load	0.80	0.67	0.55	0.54	0.44	0.60
Cache-reference	0.81	0.73	0.58	0.46	0.45	0.61
LLC-stores	0.80	0.74	0.60	0.45	0.32	0.58
L1-dcache-load-misses	0.81	0.72	0.58	0.45	0.45	0.60
Cycles	0.35	0.30	0.29	0.25	0.22	0.28
Instructions	0.21	0.21	0.24	0.21	0.19	0.21
Branches	0.21	0.21	0.24	0.21	0.19	0.21
Mem-stores	0.12	0.08	0.04	0.73	0.03	0.20
Cache Misses	0.00	0.06	0.04	0.03	0.05	0.03

(a) Throughput

Features	Average
LLC-load	0.31
Cache-reference	0.23
LLC-stores	0.18
Branches	0.17
L1-Dcache-load-misses	0.15
Cycles	0.14
Instructions	0.13
Mem-stores	0.12
Cache Misses	0.02

(b) Latency

Table 3: Correlated features with throughput and latency under resource stimulus

and performance metrics for different service topologies under both load and resource stimuli tests.

We utilize Pearson's correlation and mutual information to assess the statistical dependencies between the collected features and the KPIs. We keep features whose correlation with a KPI exceeds 0.5. Table 2 and Table 3 list the correlated features with different KPIs for the linear SFC. The results are coherent with our tendency observations. There is no dominant feature that consistently achieves the highest correlation across different VNFs, suggesting the joint impact of multiple features on SFC performance. Note that similar feature correlations have also been observed with the DAG topologies; we omitted for space.

Correlation with throughput Under load stimulus, the features in Table. 2a shows consistently high correlations with the throughput across VNFs. In particular, cache-related features, especially cache-reference rate and L1-dcache-load-misses, strongly correlate with the throughput. In contrast, as shown in Table. 3a, the correlation of those same features under resource stimulus shows an ascending pattern: the correlation is small at the beginning of the chain and increases towards the end. The bridge VNF (last column) shows the highest correlation because it is memory-intensive. We observe that certain VNF-specific behaviors contribute to the peculiarities of individual features. For instance, nDPI's mem-stores feature, as shown under both stimulus tests, exhibits a very high correlation with throughput because this particular VNF requires frequent memory accesses.

Correlation with latency As shown in Tables 2b, and 3b, cache-related features show the strongest correlation with the latency. However, latency correlations are weaker overall because latency depends on high-level events such as buffer overflow and bandwidth saturation.

Observation 2. *Some hardware features trend closely with the input traffic patterns across different service topologies, making them strong candidates for intermediate variables between input traffic and output KPIs. Additionally, these features capture the unique execution characteristics of individual VNFs, providing valuable insights into their behavior.*

4. System design

We now present DRST, a framework that enables accurate performance inference, bottleneck diagnosis and drift-aware adaptive retraining. We first outline the design principles, then present the end-to-end ML pipeline. Finally, we detail the MLOps implementation that supports deployment and monitoring.

4.1. Design principles

Based on the discussion of the prior works in Sec. 2.3, our architecture should respect the following design considerations. First, it must remain *general* enough to deliver accurate and efficient performance impairment predictions for both individual VNFs and ground-up network services with limited deployment knowledge. Second, given the ever-increasing complexity of modern networks, it must be *lightweight* and easy to deploy with minimal engineering exertions. As part of the network management subsystem colocated with VNF execution, it should be *noninvasive*

and introduce a negligible impact on the software data plane's normal operations and traffic. Finally, it must be *fast* to enable real-time predictions using the available data.

Fig. 4 illustrates our approach, the workflow has two steps: (i) data collection and (ii)statistical learning.

As an alternative to direct traffic-level measurements, we extract low-level hardware features from the shared network infrastructure. Although the internals of COTS servers are extremely complex, modern systems commonly offer various toolsets for performance monitoring, namely the Performance Monitoring Units (PMUs). It exposes hundreds of micro-architectural events that we can record in production [49]. This approach has several advantages. First, PMU counters are always available and can be readily collected via standard profiling interfaces. High-level safeguard measures (e.g., encryption, private enclaves) do not hinder the collection of these low-level features. Second, they impose negligible overhead even at Gigabit Ethernet. This point is especially crucial in high-speed networks since even slight noise can cause noticeable performance losses [41]. Third, our approach does not mandate an in-depth understanding of the target NFV system internals, such as the service, the management & operation (MANO) plane, and the implementation details of (third-party) VNFs. Operators therefore avoid code instrumentation and extra integration work.

Our end-to-end workflow comprises two parallel pipelines: a **Init-training** pipeline and an **Online-serving** pipeline. The design follows two principles: (i) MLOps for reproducible and automated lifecycle management; (ii) XAI for built-in interpretability.

4.2. Init-training pipeline

The offline training pipeline builds an initial model using historical data collected under controlled yet diverse configurations. Once trained, the model is integrated into the online-serving pipeline, which receives live feature streams via Kafka to support real-time KPI inference and forecasting.

4.2.1. Data preprocessing and feature selection

- Data preprocessing: The pre-processing stage performs three actions: (1) Feature parsing & normalization: convert `perf` event tuples into a unified numeric vector and apply standardization. (2) Scenario tagging & merge: attach <topology, stimulus> labels and combine trace segments into scenario-level samples. In total, we collected 16 scenarios to emulate the real network environment. (3) Quality filtering: drop incomplete rows and align PCM timestamps with *perf* samples to ensure temporal consistency.

- Feature selection: Following the feature relevance analysis in Section 3.2, we retain system-level metrics exhibiting strong correlation with target KPIs (throughput and latency), while minimizing inter-feature redundancy.

4.2.2. Model construction

Scenario-driven KPI estimation We represent each scenario as

$$\mathcal{E} = (\mathcal{S}, \texttt{Stim}) \tag{1}$$

where $\mathcal{S} = (v, \texttt{Topo})$, and $\texttt{Stim} = (\texttt{type}, \mathbf{p})$. $v = \{v_1, \ldots, v_k\}$ is the ordered set of VNFs, `Topo` encodes their interconnection, `type` denotes the stimulus category, and $\mathbf{p}$ is a parameter vector (e.g., input-rate, CPU throttle). This abstraction underpins all evaluation scenarios (Section 5).

Given a scenario $\mathcal{E}$, the inference function $f^{\mathcal{E}}$ maps system inputs to KPI metrics such as throughput and latency. For multi-step forecasting, we denote the model output as $[\hat{y}_{t+1}, \ldots, \hat{y}_{t+H}] = f^{\mathcal{E}}(X_{t-N+1:t})$, where $X_{t-N+1:t}$ represents the input window.

Inference engine: non-sequential mapping To infer instantaneous KPIs (e.g., throughput, latency) from hardware-level features, we define the inference engine $f^{\mathcal{E}}$ as a parametric mapping:

$$y = f^{\mathcal{E}}(\boldsymbol{X}) \tag{2}$$

where $\boldsymbol{X}$ is the current observation vector, and y is the target KPI. We implement a multi-layer perceptron (MLP) as the default inference model $f^{\mathcal{E}}$, as it offers the best trade-off between resource usage and latency on our target platform. The inference component remains modular, allowing alternative backends (e.g., tree ensembles) to be integrated when deployment constraints change.

The MLP is trained via regularized log-likelihood:

$$\hat{J}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{X}_i, y_i, \boldsymbol{\theta}) + \frac{\alpha}{2}\|\boldsymbol{\theta}\|_2^2 \tag{3}$$

where α controls L2 regularization to prevent overfitting under high dynamic range, we use Adam for gradient-based minimization.

Forecasting engine: sequential prediction To enable proactive control, the forecasting task aims to predict future KPI values $[y_{t+1}, \cdots, y_{t+H}]$ from a history of observations $[\boldsymbol{X}_t, \cdots, \boldsymbol{X}_{t-N+1}]$. Our evaluation covers seven models from four major families: regression, tree-based, LSTM, and Transformer. Here, we focus on two LSTM-based designs due to their strong temporal modeling capabilities.

- Standard LSTM: This model directly maps the input window to multiple future steps in parallel:

$$[\hat{y}_{t+1}, \ldots, \hat{y}_{t+H}] = f^{\mathcal{E}}(\boldsymbol{X}_{t-N+1:t}) \tag{4}$$

- DirREC-LSTM [50] recursively feeds its predictions back as inputs to improve long-horizon accuracy:

$$\hat{y}_{t+h} = \begin{cases} f_1^{\mathcal{E}}(\boldsymbol{X}_{t-N+1:t}), & h = 1 \\ f_h^{\mathcal{E}}(\hat{y}_{t+1:t+h-1}, \boldsymbol{X}_{t-N+1:t}), & 2 \le h \le H \end{cases}$$

The recursive design enhances prediction fidelity in long-tail scenarios, but increases inference latency due to its sequential nature.

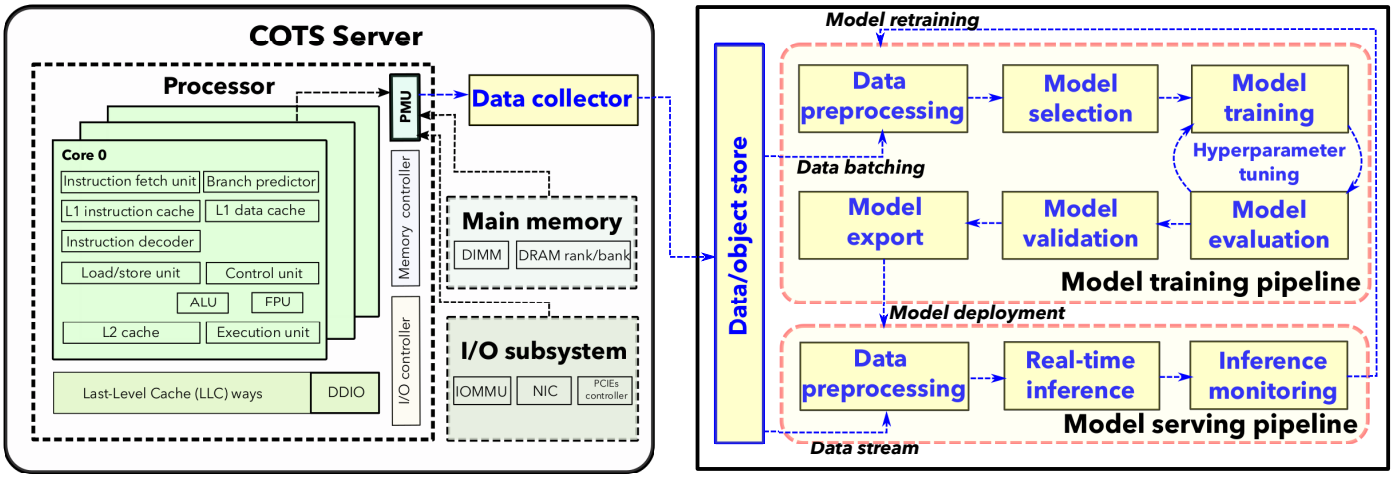


Figure 4: Overall DRST architecture for performance prediction in high-speed softwarized networks.

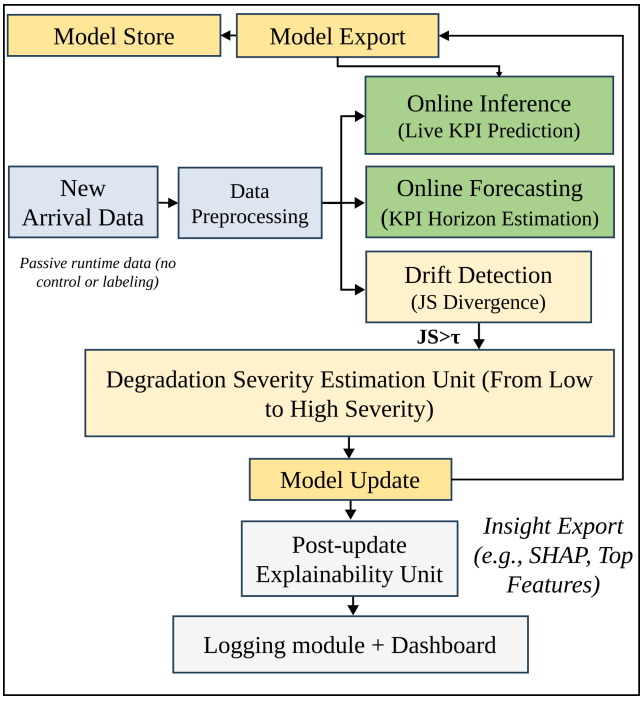


Figure 5: Model serving pipeline. All components operate on live, unlabeled data to support passive inference, drift detection, and automated model updates.

Note that DirREC-LSTM is more accurate under complex traffic. However, its runtime cost is higher, which is impractical for real-time inference on constrained hardware. We therefore adopt standard LSTM as the default model. The module remains replaceable with alternative learners.

4.2.3. Hyperparameter Search, Validation, and Export

- Hyperparameter Search: We adopt a modular tuning stage that supports interchangeable strategies such as grid search, random sampling, and Bayesian optimization, all implemented via a unified interface. In this work, we used grid search for offline training.
- Final Training: The selected model is retrained on the full dataset using the best hyperparameters to maximize generalization and statistical coverage.
- Model Validation: This stage confirms the quality of the trained model through its accuracy, F1-score, and inference latency on a held-out test set. The results serve as a reference baseline for online prediction.
- Model Export: The validated model, along with its configuration and metadata, is packaged and stored in a versioned internal registry for integration into the serving pipeline.

4.3. Model serving pipeline

The serving pipeline performs passive, real-time inference and adaptation without relying on controlled stimuli and explicit labels. It consumes live runtime features and operates continuously after deployment. As shown in Fig. 5, it comprises modular components for data handling, inference, drift analysis, model update, and system monitoring.

Data handling At runtime, monitoring streams from VNFs collect data from multiple hardware features, including memory access, PCIe patterns, and CPU consumption linked to packet I/O. Such data are time-aligned and assembled into fixed-order feature vectors matching the model input schema. Then, the resulting vectors are streamed via Kafka and dispatched in parallel to the inference, forecasting, and drift detection modules for concurrent execution, as shown in the "Data" blocks of Fig.5.

Inference and forecasting The two following blocks represent the online processes of inference and forecasting. The inference engine consumes each incoming feature vector and produces real-time KPI predictions using the most recent model generated from the training or adapted pipeline. The forecasting engine runs in parallel with inference to project short-horizon KPI trajectories based on

ALGORITHM 1: Online Inference and Drift-triggered Update Loop

Input: Monitoring features $\boldsymbol{X}_t$ arriving at time t
Output: Predicted KPI $\hat{y}_t$, possible model update
$\mathcal{W}_{\text{curr}} \leftarrow$ sliding window of recent M samples;
while system is running **do**
  Receive new feature vector $\boldsymbol{X}_t$;
  Push $\boldsymbol{X}_t$ to Kafka queue;
  Predict $\hat{y}_t \leftarrow f_{\text{current}}(\boldsymbol{X}_t)$;
  Forecast $\hat{y}_{t+1:t+H} \leftarrow f_{\text{forecast}}(\boldsymbol{X}_{t-N+1:t})$;
  Update $\mathcal{W}_{\text{curr}}$ with $\boldsymbol{X}_t$;
  if size($\mathcal{W}_{\text{curr}}$) $= M$ **then**
    Compute JS divergence D_{JS} against $\mathcal{W}_{\text{ref}}$;
    if $D_{\text{JS}} > \delta$ **then**
      severity $\leftarrow$ estimate_degradation(D_{JS});
      $f_{\text{new}} \leftarrow$ retrain_model(severity);
      $f_{\text{current}} \leftarrow f_{\text{new}}$;
      $\mathcal{W}_{\text{ref}} \leftarrow \mathcal{W}_{\text{curr}}$;
    end
  end
  Log outputs and metrics;
end

recent input windows. This enables proactive detection of emerging anomalies or performance shifts before they appear in observed KPIs.

Drift analysis To understand if a drift is occurring, we employ a drift detection module, which continuously monitors distribution changes in the input features using a sliding-window approach. At each step, it compares the current window of M (e.g., 100) recent samples against a historical reference window using Jensen-Shannon (JS) divergence. This unsupervised method quantifies input drift over time without requiring ground-truth labels, and serves as the trigger signal for downstream model update.

A second module estimates the degradation severity and decides whether to trigger an update: when the observed JS divergence exceeds a configurable threshold, the system estimates the severity of the drift and dynamically selects an update strategy. Update levels are categorized by complexity, ranging from lightweight re-optimization (e.g., learning rate tuning) to full retraining with extended search space.

Model Update and Export Once triggered, the selected retraining procedure is executed automatically, producing a new model version packaged with its configuration and metadata. The updated model replaces the active one in the serving pipeline, and all related outputs—including predictions, drift scores, and version history are logged for traceability and future analysis.

Monitoring and Explainability Runtime metrics, including per-sample latency and throughput, are collected with Prometheus and shown on a dashboard for live monitoring and alerting. Prediction results, update triggers, and SHAP-based explanations are visualized through a lightweight UI dashboard for human-in-the-loop interpretability.

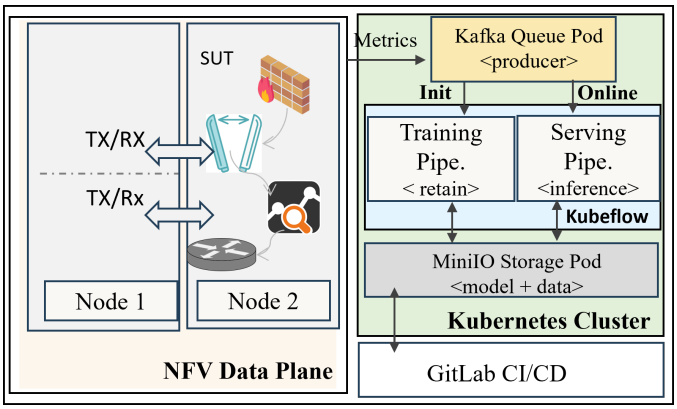


Figure 6: Infrastructure-level MLOps Implementation

To complement the component-level view, Algorithm 1 provides a runtime perspective, showing how the system performs online inference in a continuous loop.

4.4. Deployment on a Kubernetes MLOps system

The proposed system is deployed on a Kubernetes cluster and implemented via a modular MLOps stack centered on Kubeflow Pipelines. All ML components, including the training and serving pipelines, are containerized as independent services and orchestrated as DAGs.

We leverage Kafka as the real-time message bus. Feature vectors exported by VNFs are published to Kafka topics and consumed in parallel by three services: the MLP-based inference engine, the LSTM forecaster engine, and the JS divergence detector engine. These components are decoupled and horizontally scalable. The JS divergence is computed over sliding windows to quantify input drift. Once the divergence exceeds the threshold, the system launches a lightweight model update without interrupting service.

MinIO, a shared object store, serves as the internal model registry. It stores historical datasets, inference logs, model artifacts, and version metadata. Model updates are published to MinIO, and the serving engines regularly pull the latest versions. This separation of storage and execution supports rollback deployment strategies. We use a GitLab CI/CD for version control, training reproducibility and pipeline tracking.

Figure 6 summarizes the infrastructure-level implementation of the proposed MLOps architecture. On the left, the NFV data plane consists of two NUMA nodes that host the system under test (SUT), which is responsible for generating and capturing system-level metrics. These metrics are streamed into the control plane via a Kafka queue. On the right, the Kubernetes-based MLOps control

Server	Intel Xeon Processor			Memory	NUMA	Last-Level Cache		NIC	
	CPU Model	Frequency	#Cores		Sockets	Size	#Ways	Brand	Speed
Skylake	Silver 4210R	2.4 GHz	20	128 GiB	2	27.5 MiB	11	Broadcom® BCM5741X	4 × 25 Gbps
Broadwell	E5-2640 v4	2.4 GHz	20	64 GiB	1	25.6 MiB	20	Mellanox® MT27710 CX-4	1 × 40 Gbps
Haswell	E5-2667 v3	2.6 GHz	20	64 GiB	2	19.7 MiB	20	Intel® 82599ES 10-Gigabit	4 × 10 Gbps

Table 4: Testbed specifications

Category	Component	Version / Notes
System & HW	CPU / Memory	AMD EPYC 12 vCPU, 48 GB ECC
	Disk / fio	800 GiB SATA SSD, R/W ≈ 400–450 MB/s, IOPS ≈1.6–1.7k
Container Stack	Docker Engine	v28.0.1
	Kubernetes	v1.28.15 (Kind v0.27.0, kubectl v1.32.2)
	Helm / Kustomize	Helm v3.17.2, Kustomize v5.5.0
	Prometheus / Python	Prometheus 2.x, Python 3.11.11
MLOps Stack	Kubeflow Pipelines	v2.4.0, SDK v2.5.0 (DSL v2)
	Katib / KServe	Katib v0.18.0-rc.0, KServe v0.14.1
	Training Operator	v1-5170a36 (TFJob / PTJob)
	Kafka	Bitnami 3.7.1-debian-12-r4
	MinIO / MySQL	MinIO RELEASE.2019-08-14, MySQL 8.0.29

Table 5: Execution environment for containerized pipeline experiments

stack includes modular training and serving pipelines orchestrated by Kubeflow, a shared object store (MinIO), and a CI/CD layer for version control and deployment. The decoupled training and serving workflows allow independent lifecycle management, while MinIO serves as a persistent backend for both model artifacts and streaming data. The infrastructure-level implementation DAG is illustrated in Fig. A.14 (see Appendix A).

5. Experimental evaluation

5.1. Testbed environment

Hardware settings To validate the applicability of our framework, we conduct experiments on three types of COTS servers with diverse hardware components, as illustrated in Tab. 4. As indicated by the server names, the servers have processors spanning three generations of Intel CPU microarchitectures, i.e., Haswell, Broadwell, and Skylake. To minimize interference, all the cores used for our tests are isolated from the kernel scheduler (via isolcpus) with hyper-threading and turbo-boost disabled.

Software Settings We use MoonGen [45], a high-speed packet processing engine, to generate traffic and measure the end-to-end KPIs, specifically throughput and latency. At runtime, we collect the low-level hardware features of individual VNFs using *perf* [21] and *PCM* [22], two standard system profiling tools interfacing with the system PMUs at a configurable frequency (e.g., 200/500/1000 ms). The sampling frequency reflects a trade-off between temporal resolution and system overhead. Shorter intervals capture fine-grained performance fluctuations (e.g., bursty contention or latency microbursts) but incur higher profiling overhead that could interfere with VNF execution. The low-level features of each VNF can be collected via pre-assigned execution identifiers (e.g., process, thread, or function IDs). Note that other standard profiling tools [49] can also serve the same purpose.

Network service deployment

We deploy the VNFs on NUMA node 0 using OpenNetVM (ONVM) [6], which provides flexible service composition and high-speed packet steering. The instantiated VNFs operate as bare-metal processes, and can be purposely connected to form an intended network service. Each VNF runs in the busy-polling mode that monopolizes a CPU core (with 100% usage). Note that ONVM is selected because of its impressive performance and accessibility; our approach should be applicable to other prevalent NFV frameworks (e.g., ClickOS [51] or E2 [52]). The evaluation spans a range of service chains and runtime topologies, ensuring that the observed behavior is not tied to a specific deployment and that our conclusions remain topology-agnostic.

Workload generation Performance issues generally arise due to overwhelming loads and insufficient resources [53]. We consider two basic workload generation schemes to expose latent performance issues: load stimulus and resource stimulus. The former composes the input traffic with special patterns to contrive load contentions, while the latter perturbs the resource shares of individual VNFs to fabricate resource contentions. To better control the imposed contentions, we employ competitor processes to expose and analyze the impact of resource contentions. The competitors are based on stress-ng [54], a standard stress-testing tool capable of generating bogus operations at different subsystem components, including CPU pipeline, multi-level caches, and memory. For example, CPU contention can be created by pinning parasite competitors to a VNF's worker core. The cache contentions can be generated by thrashing existing lines. The memory bandwidth contention can be induced by injecting I/O requests. We can even generate multiple contentions by calculatedly assigning the competitors.

MLOps execution stack We deploy the model inference and adaptation workflow on a containerized Kubernetes cluster orchestrated with Kubeflow Pipelines and Argo. Inference is served via KServe, and hyperparameter tuning is managed by Katib. Each model component is containerized and deployed declaratively. Input data is streamed

Model	Seen trace (train/test split)			Unseen trace		Latency (ms/sample)
	R^2	MAE	Acc@5 $\%_{\log}$	MAE	Acc@5 $\%_{\log}$	
Linear	0.98	**183.52**	97.02	**163.93**	92.98	0.030
SVR	0.26	2122.68	46.00	2064.17	47.09	0.373
Decision Tree	0.96	309.46	96.78	281.43	87.92	0.030
Random Forest	0.97	286.33	97.34	258.70	88.42	0.030
Gradient Boosting	0.96	429.67	95.10	399.22	86.86	**0.001**
XGBoost	0.96	429.66	94.92	403.11	86.35	0.042
MLP	**0.99**	257.78	**98.28**	211.41	**96.81**	0.652

Table 6: Inference engine selection comparison (Best value per column in bold).

through Kafka, and model artifacts are stored in MinIO using S3-compatible access. System metrics, such as resource usage, inference latency, and update frequency, are collected via Prometheus and visualized using Grafana dashboards and Python scripts (see Table 5). [2] To ensure that storage access does not bottleneck inference or training, we benchmark the disk performance using `fio`. The measured sequential read and write throughputs are 442 MB/s and 446 MB/s, respectively. The average latency per I/O operation is approximately 300 μs on the MinIO array.

5.2. Model selection and generalization analysis

This section evaluates candidate models prior to their integration into the serving pipeline.

5.2.1. Inference engine selection

To compare accuracy and serving costs, we evaluate eight regression models, as shown in Table 6. All models are trained on a representative workload trace comprising 9788 samples (80 MB) and evaluated on an 80/20 split. A second trace with random-stage traffic is used to test generalization. Inference latency is measured as the wall-clock time per sample on a single CPU core. Accuracy is reported by the coefficient of determination (R^2), the mean-absolute error (MAE), and the log-space accuracy within 5% [3] (Acc@5 $\%_{\log}$), the latter mitigating scale imbalance in the target variable.

We have found that most models, except SVR, perform similarly well on the seen data ($R^2 \geq 0.96$). SVR struggles likely due to sensitivity to feature scaling and large target variations. In terms of generalizability, the MLP maintains the highest log-Acc@5% (96.8%) on the unseen trace, with only a modest MAE increase. Ensemble trees (RF, XGB) deliver sub-millisecond inference, while the MLP incurs a higher cost due to dense-matrix computations. Despite a higher latency (0.65 ms vs 0.05 ms), MLP remains sub-millisecond and batch-amenable within the NFV data plane system's sampling intervals.

We choose MLP as the default offline model due to its generalization ability over acceptable latency gains.

5.2.2. Forecasting engine selection

The forecasting engine periodically predicts short-horizon throughput to support proactive scaling. We benchmark several models designed to capture temporal dynamics, including *Ridge*, *XGBoost*, *standard and direct LSTM*, and *Transformer-based* models. All models are evaluated with an identical input sequence length of 10 steps and a forecast horizon of 5 steps.

We evaluate two runtime patterns: *Pattern A*, representing periodic traffic; and *Pattern B*, with stage-wise bursts with abrupt plateaus and shifts. Table 7 reports accuracy (R^2, MAE), short-horizon reliability (Acc@t+k), and per-sample latency.

Our results show that tree-based models (Random Forest, XGBoost) exhibit low latency and strong accuracy under regular workloads, but suffer notable performance drops in the presence of irregular fluctuations. DirREC-LSTM achieves better generalization across both regimes but incurs higher inference cost due to cumulative dependence across forecast steps. In contrast, the standard LSTM offers a favorable trade-off: it maintains high step-wise accuracy (Acc@t+1 to Acc@t+5) and consistent performance across both traffic patterns, while remaining computationally lightweight. Given its forecasting accuracy and operational efficiency, LSTM is adopted as the default forecasting module in our pipeline.

5.3. Online inference performance

We evaluate inference performance over three representative SFC topologies (defined in Figure 3 and described in Section 5.1), each composed of 5–6 VNFs such as Firewall, nDPI-stat, and Payload-scan.

5.3.1. Throughput inference

Accuracy across stimuli We evaluate throughput inference under realistic serving conditions by streaming time-aligned traces through Kafka and invoking per-sample inference via KServe, consistent with the production-serving pipeline. While the evaluation utilizes replayed traces for reproducibility and traceability, the inference engine operates in an online manner, processing one input at a time, within containerized serving endpoints. For each runtime

[2] We re-stream previously sampled KPI traces via Kafka with 1:1 timing replay, faithfully reproducing the original real-time sampling intervals.

[3] Due to the inherent limitations of software traffic generators [45], the rate gaps in high-speed regimes make it difficult to estimate the exact rates accurately. According to our micro-benchmarks, 5% is a realistic error.

Model	R^2		MAE		Latency (ms)		Acc@t+1		Acc@t+2		Acc@t+3		Acc@t+4		Acc@t+5	
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B
LSTM	0.95	0.61	346.6	591.7	**2.04**	**2.86**	**0.99**	0.96	0.96	**0.96**	0.93	**0.93**	0.90	0.86	0.87	0.81
DirectLSTM	0.96	**0.68**	316.2	**521.5**	324.3	424.2	**0.99**	**0.97**	0.96	**0.96**	0.94	**0.93**	**0.92**	**0.89**	**0.90**	**0.82**
Ridge	0.98	**0.73**	216.4	569.8	1.04	**0.21**	0.99	0.95	0.94	0.86	0.96	0.81	0.78	0.75	0.63	0.60
Random Forest	0.95	0.65	280.0	605.0	70.00	65.00	0.96	0.93	0.94	0.91	0.92	0.87	0.90	0.83	0.89	0.82
XGBoost	**0.99**	0.65	**94.1**	629.2	16.42	58.93	0.98	**0.94**	0.98	0.93	0.98	0.90	0.86	0.84	0.84	0.82
TransformerLight	0.90	0.54	621.0	740.2	2.11	3.43	0.96	0.89	0.96	0.87	0.93	0.84	0.93	0.83	0.88	0.81
Transformer	0.94	0.60	450.3	718.7	2.56	5.86	0.98	0.97	0.98	0.89	**0.98**	0.92	0.88	0.86	0.84	0.83

Table 7: Comparison of forecasting models under two runtime traffic patterns: A (periodic trend), B (stage-random fluctuations).

Topology	Scenario Type	Acc@5% (Mean ± Std)	MAPE (Mean ± Std)	Latency (ms)	Throughput (samples/s)
Single VNF	Load Stimulus (Regular)	98.2% ± 1.4%	2.1% ± 0.7%	0.74	1351
Linear SFC	Load Stimulus	97.4% ± 1.9%	2.5% ± 0.6%	0.76	1315
	Random Traffic Stimulus	92.8% ± 2.7%	4.1% ± 1.2%	0.78	1282
	Resource Contention	83.6% ± 3.1%	6.3% ± 1.7%	0.75	1333
	Intervention	72.4% ± 3.5%	7.9% ± 2.0%	0.77	1310
DAG1 SFC	Load Stimulus	95.1% ± 1.8%	3.1% ± 1.0%	0.73	1369
	Random Traffic Stimulus	90.2% ± 2.5%	5.2% ± 1.3%	0.74	1351
	Resource Contention	82.5% ± 2.9%	7.3% ± 1.9%	0.76	1315
DAG2 SFC	Load Stimulus	93.6% ± 2.0%	3.7% ± 1.1%	0.75	1333
	Resource Contention	79.8% ± 3.6%	8.2% ± 2.1%	0.78	1282
	Intervention	67.9% ± 3.8%	8.8% ± 2.3%	0.76	1315

Table 8: MLP-based throughput inference performance across runtime scenarios. Accuracy is reported using Acc@5% and MAPE, averaged over 10 i.i.d. test runs (170 samples each). Inference latency is measured per sample, excluding Kafka overhead. Throughput is measured via Prometheus-based monitoring.

scenario, 10 independent test runs are conducted, each consisting of 170 samples. Accuracy is reported as the mean and standard deviation of log-space Acc@5% and MAPE across 10 runs.

Under load stimulus, our model achieves impressive overall accuracy: 98% for the regular rate and 92% for the random rate, as illustrated by the examples in Fig. 7a and 7b. Under resource stimulus, throughput inference becomes more intricate due to diverse contentions from the CPU, caches, and memory buses. Still, our model's accuracy remains commendable at 83%, as in Fig. 7c.

All per-scenario results are further summarized in Table 8, which lists the inference performance across various deployment topologies and runtime stimuli.

Comparison with SoTA Prior works on NFV throughput inference primarily attribute performance degradation to memory subsystem bottlenecks. Dobrescu et al. [32] propose a linear model based on cache access rates, assuming additive contention effects across traffic flows — an assumption that proves inadequate under modern hardware conditions. SLOMO [10] improves estimation accuracy by jointly modeling cache and memory bandwidth contention using gradient boosting regression, but is evaluated only in single-VNF setups and lacks robustness under non-linear contention patterns. We compare our model against these two approaches under three scenarios of increasing complexity, as illustrated in Figure 8: **(1)** a singleton VNF (ONVM bridge) under mixed stimuli, **(2)** a linear SFC under resource contention, and **(3)** the same SFC subject to both load and resource interference. Although scenario 3 is rarely realistic, it provides rich contention samples useful for stress-testing KPI predictors.

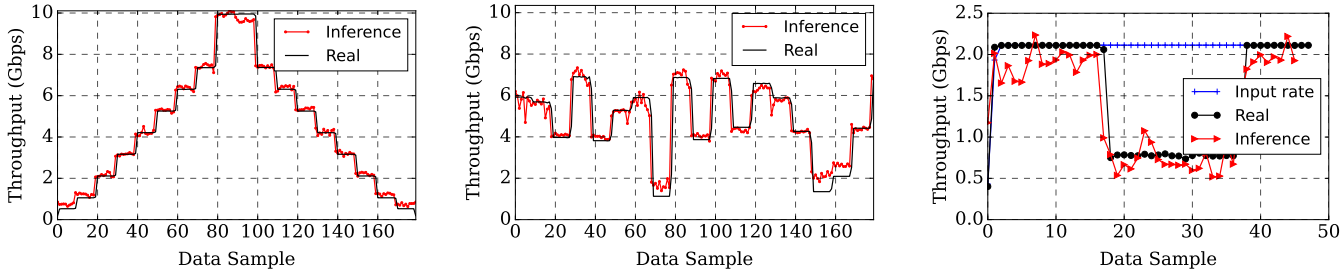
Our approach improves over these works along three key dimensions:

- *Scenario coverage.* While previous work focuses on isolated VNFs or synthetic workloads, we build and evaluate a full testbed spanning 11 real-world configurations.
- *Deployment realism.* Unlike prior offline evaluations, our inference engine runs in a live, containerized environment. Models are deployed into a Kubernetes-based pipeline, enabling real-time per-sample prediction with observability through Prometheus.
- *Accuracy comparison.* In Figure 8, we compare inference accuracy in three scenarios of increasing complexity. Even in the most stressed setting (SFC under dual interference), our model maintains 65% accuracy—compared to 41% and 56% for Dobrescu and SLOMO, respectively.

5.3.2. Latency inference

Latency inference is harder due to drastic state transitions under congestion (Sec. 3.2). Here, the latency is measured as the round-trip delay of probe packets generated by MoonGen at 1 Gbps, with a maximum wait time of 120 ms per probe. These probe packets are processed concurrently with regular service traffic and offer a realistic estimate of packet processing delay.

Under normal traffic and resource conditions, the measured latency remains stable below 50 μs and exhibits minimal variance. In such regimes, latency inference is



(a) Linear (regular rate) (b) Linear (random rate) (c) Linear (res. stim.)

Figure 7: Throughput inference under different workload generation

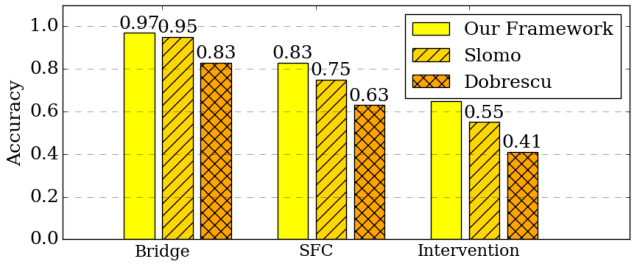
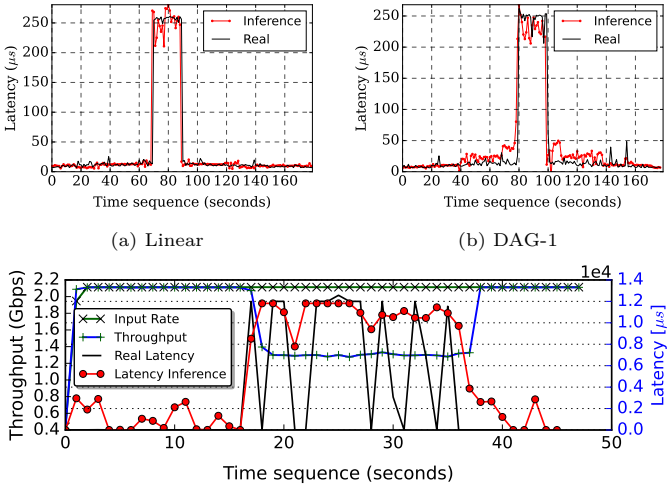


Figure 8: Throughput inference comparison



(a) Linear (b) DAG-1

(c) Resource stimulus

Figure 9: Latency inference under three scenarios

of limited operational significance. However, we design a controlled workload where the traffic rate increases linearly from 0 to the line rate (10 Gbps), and then decreases back to 0. Despite the smooth traffic pattern, the observed latency does not grow linearly. It remains low (<50 μs) for most of the duration, but exhibits sharp spikes only when approaching saturation. This indicates that latency jumps abruptly once critical thresholds are exceeded, making it challenging to predict using continuous traffic or resource metrics alone.

Prediction accuracy We train an MLP model to predict the per-sample latency using the same runtime pipeline. Fig. 9a and Fig. 9b show its performance on linear and DAG-1 topologies under increasing load. The model achieves 86% and 79% accuracy, respectively, except in the high-throughput regime (80–100s), where congestion and queue build-up cause RTT to oscillate.

Stress under resource contention Fig. 9c shows the inference results under resource stimulus. Severe contention between 15s and 35s cut throughput by up to 40%, triggering packet losses and erratic delays. During such periods, latency becomes fundamentally unpredictable, but our model can still approximate its envelope and detect high-latency transitions. Current methods are unsuitable for predicting the exact (artificial) latency under severe network congestion, as sporadic packet losses make the RTT hard to quantify. However, our model can still approximate the expected values with prior knowledge of the maximum latency and predict abnormal service latency and congestion periods.

5.3.3. *Explainability and feature sensitivity*

To facilitate interpretation, we design an XAI unit pod by applying a SHAP explainer to assess feature importance in KPI inference. To reduce the high computational complexity ($\mathcal{O}(n \cdot d^2 \cdot T_{\text{inference}})$), we first employ gradient-based sensitivity analysis to pre-select the most relevant features. This reduces the feature set size (e.g., from 42 to 10), cutting SHAP runtime from 2400 s to 170 s on a 100-sample dataset.

SHAP for throughput Fig. 10 shows the normalized SHAP values for the top features in throughput inference. We summarize three key findings: 1) As expected, cache-related features dominate. However, `*_cycles` features also show high attribution, which may indicate interference from background processes or unstable CPU frequency scaling. 2) Under **load stimulus**, `L1-dcache-load-misses` becomes the main contributor. This suggests that heavy traffic increases pressure on L1 caches, causing potential pipeline stalls. 3) Under **resource stimulus**, features like `*_cache-references` and `*_instructions` stand out. This points to last-level cache contention and possible DMA inefficiencies when resources are constrained.

To mitigate these bottlenecks, several potential opti-

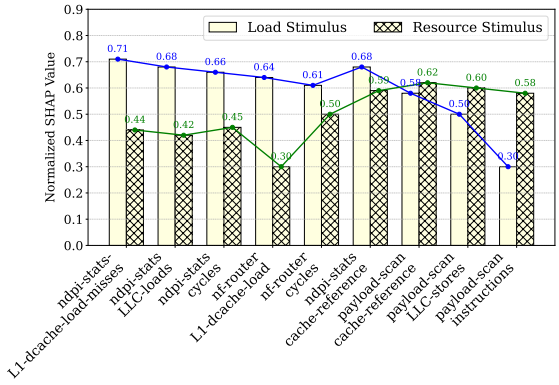


Figure 10: XAI analysis for throughput inference.

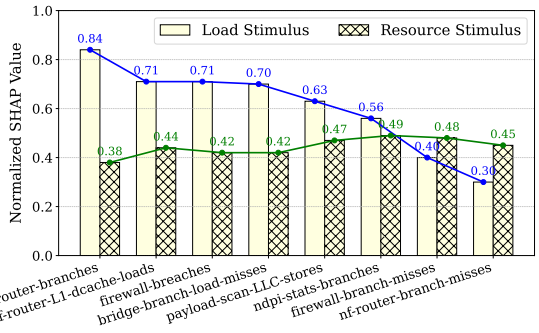


Figure 11: XAI analysis for latency inference.

mizations are recommended: (i) enabling hardware prefetching and fine-tuning CPU frequency governors to reduce latency for cache-bound operations; (ii) isolating parasite processes or dedicating cores using CPU pinning to optimize resource utilization;These measures can help network engineers improve system performance efficiency.

SHAP for latency Fig. 11 shows the top SHAP-ranked features for latency inference. We summarize three key observations: 1) Branch-related features (e.g., **_branch*, **_branch_miss*) dominate, highlighting the role of CPU branch prediction in latency behavior; 2) Under **load stimulus**, *nf-router_branches* is the top contributor, suggesting that peak traffic increases branching pressure in the router module; 3) Under **resource stimulus**, control-plane modules like *ndpi-stats-branches* and *firewall-branch-misses* become dominant, as limited resources amplify misprediction risks and delay packet processing.

5.4. Throughput forecasting

The forecasting module runs inside the real-time MLOps pipeline. Features vectors are streamed via Kafka, and predictions are computed directly on unseen inputs. While the test data follows the same topology and traffic pattern (`linear-random-stage`) as the training scenario, the specific inputs used for evaluation are entirely disjoint from the training set, thereby avoiding any data leakage. We consider a forecasting horizon of H=12 future steps. Each input is constructed from a 12-second window of performance metrics, collected at a sampling interval of 1 second using Intel PCM. The forecasting targets correspond to the aggregated throughput (in Gbps) over the next 12 seconds.

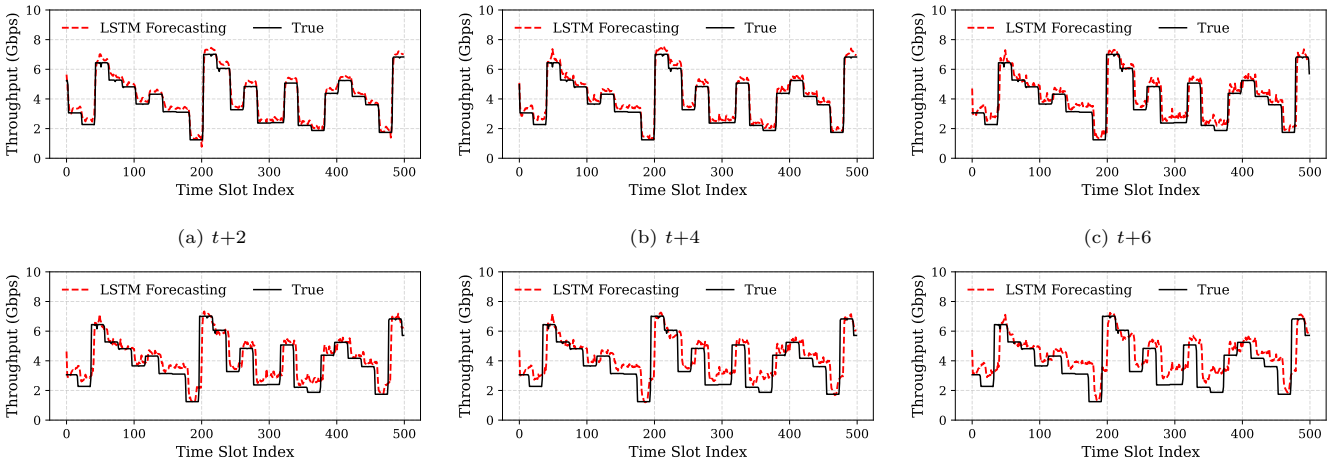
As shown in Figure 12, the predicted throughput traces from the standard LSTM model are plotted against ground truth values for selected steps from t+2 to t+12. The y-axis is fixed to a common range (0–10 Gbps) to ensure consistent visual comparison. Forecasting accuracy naturally decreases with increasing horizon length. Each subplot represents a distinct prediction slice from the same forecast vector produced in a single inference pass. Notably, the predictions are evaluated on an unseen trace, ensuring no synchronization or leak from input sources or seed alignment. We observe a natural degradation in forecasting precision as the horizon increases. Two model variants are assessed: standard LSTM and DirREC-LSTM. The DirREC variant achieves higher short-term accuracy, with absolute-relative accuracy (AbsRelAcc) exceeding 90% and log-scaled accuracy (LogAcc) above 99.7% for the first four steps. Beyond t+6, however, its accuracy declines sharply (AbsRelAcc = 52.89% at t+12). In contrast, the standard LSTM maintains more stable performance across horizons, achieving an R^2 of 0.80 and MAE of 478.43, compared to 0.73 and 604.20 for DirREC.

5.5. System runtime and adaptation efficiency

We evaluate the runtime behavior of each module within the deployed pipeline, focusing on latency, triggering frequency, and adaptation overhead. Table 9 summarizes the profiling results. Online components—including feature extraction, preprocessing, throughput inference, and short-horizon forecasting—are executed per sample or at fixed intervals (e.g., every 10 seconds). These tasks are lightweight, with module runtime ranging from 0.01s (inference engine) to 0.03s (preprocessing), ensuring real-time performance under a 1-second sampling interval.

Drift detection is triggered periodically and completes within 9 ms per window, using JS divergence between feature distributions. Upon drift detection, the system invokes adaptation routines according to a severity-aware policy. Minor drifts (Severity-1) prompt quick tuning procedures (12.7s), moderate drifts (Severity-2) adjust model structure (47.3s), while severe drifts (Severity-K) initiate full hyperparameter search (up to 141.9s). We took $K = 3$ in this experiment. Explainability modules (e.g., SHAP) are optionally enabled for in-depth analysis. Model updates are pushed online via a hot-swapping mechanism, with deployment latency under 1 second.

To complement the runtime profiling and adaptation triggers summarized in Table 9, we provide a time-series visualization of the drift detection and correction process in Fig. 13. The experiment simulates a resource-induced drift, where low-level features from a changed service topology (DAG-1) replace the initial regular load inputs (bridge). Upon detecting significant distributional shift via JS divergence, the system triggers Severity-2 retraining. Within



(a) t+2 (b) t+4 (c) t+6

(d) t+8 (e) t+10 (f) t+12

Figure 12: Online multi-step forecasting in the deployed MLOps pipeline. Each curve represents predictions for a specific forecast step (t+2 to t+12), computed from a single inference execution on unseen streamed input.

Pipeline Stage	Module	Online	Runtime (s)	Trigger Frequency	Description
Data Handling	Extraction	✓	0.01	Per sample	Stream raw feature vectors from Kafka
	Preprocessing	✓	0.03	Per sample	Normalize and transform input features
Inference	Inference Engine	✓	0.01	Per sample	Predict current throughput (MLP)
	Forecasting Engine	✓	0.13	Every 30s	Predict near-future trend
Monitoring	Drift Detection	✓	0.01	Every 10s	Compute JS divergence between sliding windows
Adaptation	Severity-1 Handler	×	12.7	On drift (low)	Lightweight tuning (learning rate, batch size)
	Severity-2 Handler	×	47.3	On drift (med)	Moderate tuning (depth, activation, optimizer)
	Severity-K Handler	×	141.9	On drift (severe)	Full grid search
	XAI Unit	×	0.8	On drift (med)	SHAP-based analysis to interpret model behavior
	Model Update	×	<1.0	Post-tuning	Push retrained model to KServe and update endpoint

Table 9: Runtime profiling of key modules in the unified inference–adaptation pipeline (see Fig. A.14). Online modules are triggered per sample or periodically. Adaptation stages are invoked based on detected drift severity.

around 44 seconds, the new model is deployed, restoring prediction accuracy on the incoming data stream.

To further characterize system responsiveness and resource efficiency, we report detailed runtime statistics of two critical components. Table 10 summarizes the monitoring performance of the JS Divergence module, showing consistently low CPU usage and memory footprint. The divergence computation completes within a few milliseconds per batch, with negligible Kafka lag, supporting high-frequency drift detection without impacting pipeline stability.

Table 11 reports the inference and deployment-time metrics measured across different system layers. The model inference latency (0.12 s) includes tensor serialization, the forward pass, and result deserialization, representing the full processing cycle within the model server. The Pod Response Latency (0.02 s) captures the round-trip time from external request to response via KServe, under warmed-up conditions. The reported inference throughput (31800samp/s) is a theoretical maximum estimated under full compute utilization, assuming minimal external bottlenecks. Finally, the cold start time (62.02s) reflects the time from pod instantiation to readiness for serving the first batch, including model loading and pipeline initialization.

6. Conclusion and future directions

As network softwarization keeps gaining momentum, there is an urgent need for stable and predictable performance in the software data plane. Existing solutions for network performance diagnostics commonly rely on per-packet measurement for data collection, which requires tremendous engineering overhead and interferes with the critical data path. We propose a novel approach that utilizes low-level hardware features for KPI prediction. Compared to per-packet data collection, our approach is easily applicable to real-world NFV systems without an in-depth understanding of their implementation details. The low-level data collection imposes a negligible impact on the software data plane. We implement tractable AI/ML model that can accurately infer throughput and latency in high-speed networks. Our model is generalizable to network services with similar topological compositions, and its predictions can be interpreted with domain-specific knowledge to identify performance bottlenecks.

Appendix A. End-to-end DAG implementation.

The diagram records the concrete Kubeflow workflow instantiated in our cluster. The upper branch depicts the

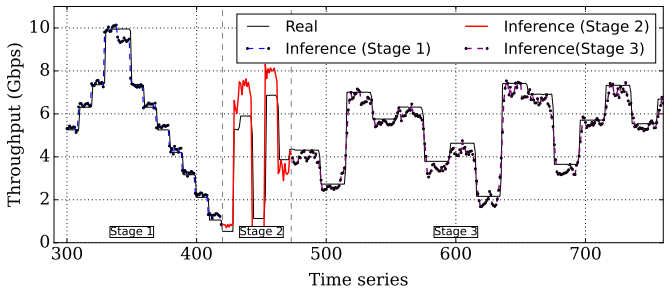


Figure 13: Drift detection and correction under load stimulus. The system transitions from Stage 1 (pre-drift) to Stage 2 (retraining), and resumes accurate inference in Stage 3 with the updated model.

Metric	Mean	Std Dev	Min	Max
JS Divergence (js_val)	0.097	0.058	0.00	0.51
CPU Usage (%)	21.97	10.19	7.20	62.70
Memory Usage (%)	3.871	0.04	38.50	38.80
I/O Wait Time (%)	0.25	0.22	0.00	1.00
Kafka Lag (msgs)	11.33	25.05	0.00	102.00

Table 10: Runtime metrics of the JS Divergence Monitor pod, collected via Prometheus. Statistics computed over one representative execution window (250 samples).

Metric	Scope	Value
Model Inference Latency[4]	Internal (MLP only)	0.12 s
Pod Response Latency	End-to-end RTT	0.02 s
Inference Throughput	Internal	3180 samp/s
Cold Start Time	Startup Time (KServelogs)	62.02 s

Table 11: End-to-end inference latency and deployment-time metrics.

init-training pipeline and final artefact storage, while the lower branch corresponds to the online serving pipeline, including inference, forecasting, drift detection, and adaptive update. Background colours indicate functional layers: blue = data ingestion & preprocessing, gray = offline model lifecycle, green = long-running online services, and orange = monitoring & self-adaptation.

References

[1] *Intel® DPDK*, `https://www.dpdk.org/`, Last accessed Dec. 2024.

[2] *Linux user space library for network socket acceleration based on RDMA compatible network adaptors*, `https://github.com/Mellanox/libvma`, Last accessed Dec. 2024.

[3] L. Rizzo, *netmap: A Novel Framework for Fast Packet I/O*, in: USENIX ATC, 2012, pp. 101–112.

[4] *Dynamically program the kernel for efficient networking, observability, tracing, and security*, `https://ebpf.io/`, Last accessed Dec. 2024.

[5] M. Paolino, et al., *SnabbSwitch user space virtual switch benchmark and performance optimization for NFV*, in: IEEE NFV-SDN, 2015, pp. 86–92.

[6] W. Zhang, et al., *OpenNetVM: A platform for high performance network service chains*, in: HotMiddlebox, 2016, pp. 26–31.

[7] T. Zhang, et al., *NFV platforms: Taxonomy, design choices and future challenges*, IEEE TNSM 18 (2020) 30–48.

[8] W. Wu, et al., *Perfsight: Performance diagnosis for software dataplanes*, in: ACM IMC, 2015, pp. 409–421.

[9] Q. Cai, et al., *Understanding host network stack overheads*, in: ACM SIGCOMM, 2021, pp. 65–77.

[10] A. Manousis, et al., *Contention-aware performance prediction for virtualized network functions*, in: ACM SIGCOMM, 2020, pp. 270–282.

[11] J. Gong, et al., *Microscope: Queue-based performance diagnosis for network functions*, in: ACM SIGCOMM, 2020, pp. 390–403.

[12] R. Haecki, et al., *How to diagnose nanosecond network latencies in rich end-host stacks*, in: USENIX NSDI, 2022, pp. 861–877.

[13] M. T. Arashloo, et al., *Formal methods for network performance analysis*, in: USENIX NSDI, 2023, pp. 645–661.

[14] Z. Yao, et al., *Aquarius—Enable Fast, Scalable, Data-Driven Service Management in the Cloud*, IEEE TNSM (2022) 4028–4044.

[15] F. Bronzino, et al., *Traffic refinery: Cost-aware data representation for machine learning on network traffic*, ACM POMACS (2021) 1–24.

[16] G. Wan, et al., *CATO: End-to-End Optimization of ML-Based Traffic Analysis Pipelines*, arXiv preprint arXiv:2402.06099 (2024).

[17] H. Li, et al., *LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed*, in: USENIX NSDI, 2023, pp. 1451–1468.

[18] T. Zhang, et al., *FloWatcher-DPDK: Lightweight line-rate flow-level monitoring in software*, IEEE TNSM (2019) 1143–1156.

[19] Q. Liu, et al., *Operationalizing AI/ML in Future Networks: A Bird's Eye View from the System Perspective*, IEEE Commun. Mag. (2024).

[20] L. Yang, et al., *Quality Monitoring and Assessment of Deployed Deep Learning Models for Network AIOps*, IEEE Network 35 (2021) 84–90.

[21] L. Foundation, *perf: Linux profiling with performance counters*, `https://perf.wiki.kernel.org/index.php/Main_Page`, Last accessed Dec. 2024.

[22] *Intel® Performance Counter Monitor - A Better Way to Measure CPU Utilization*, `https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html`, Last accessed Dec. 2024.

[23] *Intel VTune Profiler*, `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html`, Last accessed Dec. 2024.

[24] C. Shelbourne, et al., *On the learnability of software router performance via CPU measurements*, in: Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies, 2019, pp. 23–25.

[25] C. Shelbourne, et al., *Inference of virtual network functions' state via analysis of the CPU behavior*, in: International Teletraffic Congress, 2021, pp. 1–9.

[26] Q. Liu, et al., *Non-invasive performance prediction of high-speed softwarized network services with limited knowledge*, in: IEEE INFOCOM, Vancouver, Canada, 2024, pp. 2328–2337.

[27] E. Kohler, et al., *The Click modular router*, ACM Transactions on Computer Systems 18 (2000) 263–297.

[28] P. Zheng, et al., *NFV performance profiling on multi-core servers*, in: 2020 IFIP Networking Conference, IEEE, 2020, pp. 91–99.

[29] C. Sun, et al., *NFP: Enabling network function parallelism in NFV*, in: ACM SIGCOMM, 2017, pp. 43–56.

[30] Y. Zhang, et al., *Parabox: Exploiting parallelism for virtual network functions in service chaining*, in: SOSR, 2017, pp. 143–149.

[31] X. Lin, et al., *DAG-SFC: Minimize the embedding cost of SFC with parallel VNFs*, in: ACM ICPP, 2018, pp. 1–10.

[32] M. Dobrescu, et al., *Toward Predictable Performance in Software packet-processing Platforms*, in: USENIX NSDI, 2012, pp. 141–154.

[33] A. Tootoonchian, et al., *{ResQ}: Enabling {SLOs} in Network Function Virtualization*, in: USENIX NSDI, 2018, pp. 283–297.

[34] Y. Yuan, et al., *Don't forget the I/O when allocating your LLC*, in: ACM/IEEE ISCA, 2021, pp. 112–125.

[35] V. R. Chintapalli, et al., *NFVPermit: Towards Ensuring Per-*

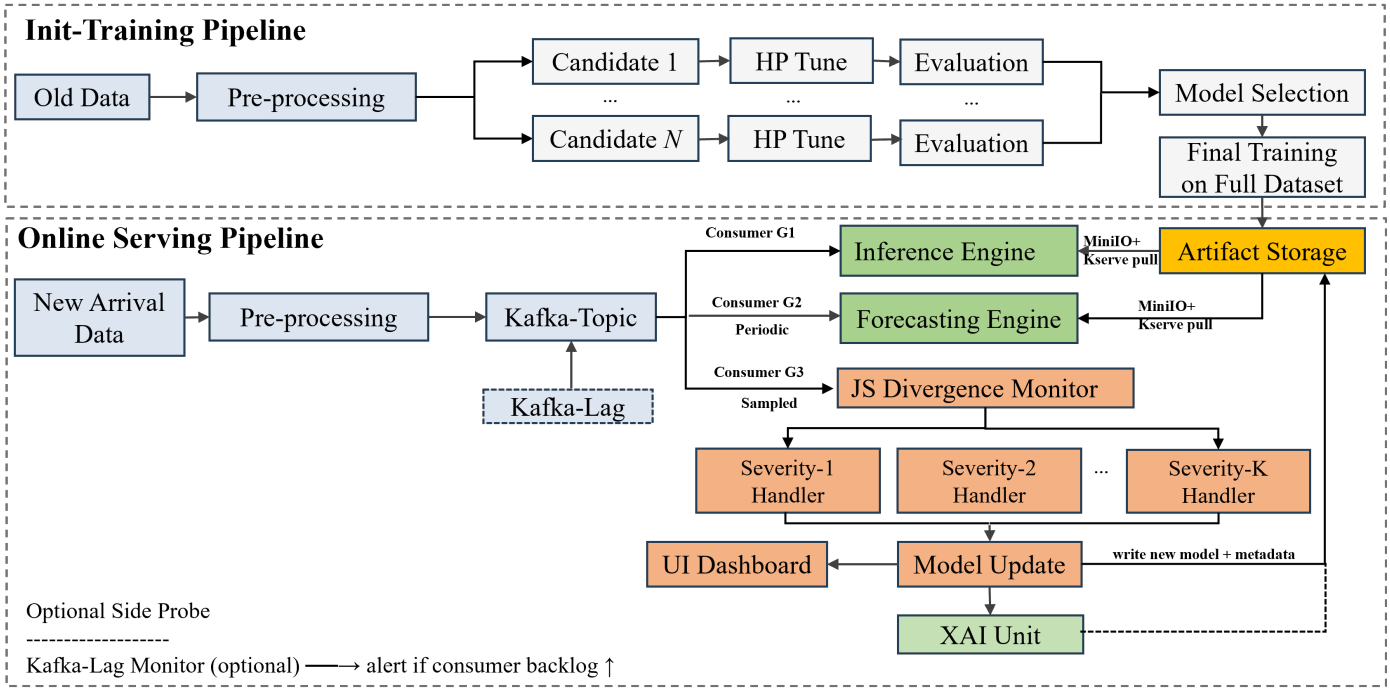


Figure A.14: Infrastructure-level MLOps implementation DAG.

formance Isolation in NFV-based Systems, IEEE TNSM (2023).

[36] Z. Niu, et al., *Unveiling performance of NFV software dataplanes*, in: ACM CAN, 2017, pp. 13–18.

[37] P. Naik, et al., *NFVPerf: Online performance monitoring and bottleneck detection for NFV*, in: IEEE NFV-SDN, 2016, pp. 154–160.

[38] N. Van Tu, et al., *PPTMon: Real-Time and Fine-Grained Packet Processing Time Monitoring in Virtual Network Functions*, IEEE TNSM (2021) 4324–4336.

[39] M. Dodare, et al., *NFV-VIPP: Catching internal figures of packet processing for accelerating development and operations of NFV-nodes*, in: CNSM, 2019, pp. 1–4.

[40] T. Zhang, et al., *A benchmarking methodology for evaluating software switch performance for NFV*, in: 2019 IEEE Conference on Network Softwarization (NetSoft), IEEE, 2019, pp. 251–253.

[41] T. Zhang, et al., *Comparing the performance of state-of-the-art software switches for NFV*, in: ACM CoNEXT, 2019, pp. 68–81.

[42] T. Zhang, et al., *Performance benchmarking of state-of-the-art software switches for NFV*, Computer Networks 188 (2021) 107861.

[43] C. Lan, et al., *Embark: Securely outsourcing middleboxes to the cloud*, in: USENIX NSDI, 2016, pp. 255–273.

[44] T. Zhang, et al., *FlowMon-DPDK: Parsimonious per-flow software monitoring at line rate*, in: 2018 Network Traffic Measurement and Analysis Conference (TMA), IEEE, 2018, pp. 1–8.

[45] P. Emmerich, et al., *MoonGen: A scriptable high-speed packet generator*, in: ACM IMC, 2015, pp. 275–287.

[46] *Speedometer (Bandwidth consumed monitor)*, `https://github.com/hpcn-uam/iDPDK-Speedometer`, Last accessed Dec. 2024.

[47] DPDK, *DPDK based packet generator*, Last accessed Dec. 2024.

[48] T. Barbette, et al., *Fast userspace packet processing*, in: ACM/IEEE ANCS, 2015, pp. 5–16.

[49] D. Bakhvalov, Performance Analysis and Tuning of Modern CPUs, Independently Published, 2024. ASIN: B0DMVQ1QDD.

[50] Z. Han, et al., *A review of deep learning models for time series prediction*, IEEE Sensors Journal 21 (2019) 7833–7848.

[51] J. Martins, et al., *ClickOS and the Art of Network Function Virtualization*, in: USENIX NSDI, 2014, pp. 459–473.

[52] S. Palkar, et al., *E2: A framework for NFV applications*, in: SOSP, 2015, pp. 121–136.

[53] A. Aghasaryan, et al., *Stimulus-based sandbox for learning resource dependencies in virtualized distributed applications*, in: IEEE ICIN, 2017, pp. 238–245.

[54] *Kernel/Reference/stress-ng - Ubuntu Wiki*, `https://wiki.ubuntu.com/Kernel/Reference/stress-ng`, Last accessed Dec. 2024.