ENS DE LYON

THESE

pour l'obtention du grade de Docteur, délivré par

l'ECOLE NORMALE SUPERIEURE DE LYON

Ecole Doctorale N°512
InfoMaths - Informatique et Mathématiques de Lyon

Discipline : INFORMATIQUE (Informatique)

Soutenue publiquement le 15 décembre 2025, par :

Johann HUGON

Pipelines d'extraction de métriques pour la supervision du trafic réseau sous contraintes système

System-Constrained Feature Extraction Pipelines for Network Traffic Monitoring

---

Après avis de :

Chadi BARAKAT, Directeur de recherche, Centre Inria Sophia Antipolis - Méditerranée	Rapporteur
Kandaraj PIAMRAT, Maîtresse de conférences HDR, Nantes University, LS2N, INRIA	Rapporteure

Devant le jury composé de :

Chadi BARAKAT, Directeur de recherche, Centre Inria Sophia Antipolis - Méditerranée	Rapporteur
Kandaraj PIAMRAT, Maîtresse de conférences HDR, Nantes University, LS2N, INRIA	Rapporteure
Anthony BUSSON, Professeur des universités, Université Lyon 1	Examinateur
Cristel PELSSER, Professeure des universités, Université catholique de Louvain	Examinatrice
Kevin VERMEULEN, Chargé de recherche, LIX, Ecole Polytechnique	Examinateur
Francesco BRONZINO, Maître de conférences HDR, ENS de Lyon	Directeur de thèse

À la mémoire de ma mère, Dominique Basset.

ABSTRACT

Machine Learning (ML) techniques are widely used in academic literature and often demonstrate excellent performance on paper. However, many of these proposed methods fail when deployed in real-world environments. In networking, this gap is primarily due to the common practice of overlooking the data collection and feature extraction pipeline during evaluation. While ML models perform well on ideal datasets, their accuracy deteriorates when features are mangled or incomplete, often caused by packet losses or limitations in the feature extraction systems that cannot process all traffic in real time. In practical network monitoring, operators and researchers require tools capable of delivering real-time insights. Such tools must balance computational complexity by limiting the types of features they extract to cope with line rate. They also face constraints in terms of available data at any given moment, which can be restricted by memory or system capabilities. These real-world limitations significantly impact the reliability and scalability of ML-based network monitoring solutions.

In this thesis, we address the challenge of deploying ML-based network monitoring tools in real-world environments by designing feature extraction methods that are both system-aware and scalable. We focus on making feature extraction compatible with high-speed traffic processing, taking into account packet loss and limited computational resources.

RÉSUMÉ

Le Machine Learning (ML) est largement utilisé dans la littérature académique et présente souvent d'excellentes performances sur le papier. Cependant, de nombreuses méthodes sous-performent lors de leur déploiement dans le monde réel. Dans le monde des réseaux, cela est principalement dû à la pratique courante d'ignorer la phase de récolte de données et de l'extraction des métriques de l'évaluation des modèles. Les modèles de ML sont donc évalués sur des jeux de données parfaits, ce qui entraine une détérioration de leurs performances (précision) lorsque les métriques sont altérées ou incomplètes. Cette dégradation est entrainée par la perte de paquets ou une limitation des ressources de calcul causées par l'incapacité du système d'extraire les métriques en temps réel, à la vitesse du lien. Dans le cas concret de la supervision du trafic réseau, opérateur et chercheur ont besoin d'outils capables de leur fournir des aperçus de l'état du réseau en temps réel. Ces outils doivent trouver le point d'équilibre de la complexité des calculs réalisés pour pouvoir supporter la vitesse du lien. Ils font également face à de fortes contraintes en termes de disponibilité des données à un instant t, ce qui peut être fortement limité par la mémoire ou les capacités de calcul du système. Ces limitations liées au monde réel ont un impact important sur la fiabilité et le passage à l'échelle des solutions de supervision utilisant du ML.

Dans cette thèse, nous approchons le problème du déploiement de solutions de supervision de trafic réseau utilisant du ML dans des environnements réalistes en concevant des méthodes d'extraction de métriques respectant les contraintes systèmes et capables de passer à l'échelle. Nous nous intéressons particulièrement à rendre l'extraction de métriques compatible avec les réseaux à haute vitesse en prenant en compte la perte de paquets et les ressources de calcul limitées.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

[1] Johann Hugon et al. "Towards Adaptive ML Traffic Processing Systems." In: *Proceedings of the on CoNEXT Student Workshop 2023*. CoNEXT-SW '23. Paris, France: Association for Computing Machinery, 2023, pp. 11–12. ISBN: 9798400704529.

[2] Johann Hugon et al. *Cruise Control: Dynamic Model Selection for ML-Based Network Traffic Analysis*. 2024. arXiv: 2412.15146 [cs.NI].

[3] Johann Hugon et al. "The Cost of Packet Loss on ML-Based Traffic Analysis." In: *2025 IEEE 31th International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 2025.

My designs were so deceptively simple that it was easy for people to assume I just had easy problems, whereas others, who made super-complicated designs (that were technically unsound and never worked) and were able to talk about them in ways that nobody understood, were considered geniuses.

Radia Perlman —

REMERCIEMENTS

Cette thèse, au-delà d'être une centaine de pages de texte formaté, est le fruit et la conclusion de trois années (et bien plus d'années d'études) qui m'ont fait grandir et évoluer, et cela sûrement plus que je ne veux l'admettre.

Pour cela, je souhaite tout d'abord remercier Francesco Bronzino et Anthony Busson pour m'avoir encadré et accompagné tout au long de ce périple. J'espère que vous avez autant apprécié que moi ces années en ma compagnie. Dans la même veine, je souhaite remercier Paul Schmitt et Nick Feamster pour leur temps et les discussions qui ont contribué au bon déroulement de ma thèse et m'ont permis d'ouvrir ma vision de la recherche. Je souhaite par ailleurs remercier tous les membres de mon jury, Cristel Pelsser, Kevin Vermeulen et plus particulièrement les rapporteurs Chadi Bakarat et Kandaraj Piamrat. J'en profite pour remercier Russ et Marcello pour leur suivi et leur précieux retour lors de mes comités de suivis.

Bien que ces remerciements ne puissent pas être exhaustifs, je vais quand même m'essayer à l'exercice. Je souhaite remercier toutes les personnes que j'ai pu rencontrer au cours de ces années au sein de l'équipe HoWNet et avec qui j'ai eu la chance de partager une discussion, un café et/ou un repas : Anthony, Augustin, Esther, Francescomaria, Isabelle, Loic C, Loic D, Meriem, Samir, Théotime, Théophile, Thierry, Thomas, Youssouph. Je souhaite également remercier toutes les personnes, en dehors du cercle de notre équipe, avec qui j'ai eu le plaisir de discuter au sein du LIP ou plus largement de la FIL, en particulier Joël, Antoine, les MALIP et les MILIP. Merci également l'équipe de l'UChicago qui m'a accueillis pendant un semestre : Ajun, Andrew, Anna, Chase, Cory, Jonatas, Kyle, Shinan, Synthia, Taveesh et Van. J'espère que nos chemins se recroiseront à l'avenir !

J'ai une pensée particulière pour mes amis qui m'ont servi de soupape de décompression quand rien n'allait ou m'ont canalisé lorsque je m'emballais un peu trop. Merci à David pour le soutien moral mutuel durant ces cinq dernières années et merci aussi de m'avoir si souvent servi de bloc note. Mehdi et Paul d'avoir toujours été volontaire pour m'écouter déblatérer tout et n'importe quoi. Clément pour avoir supporté mes horaires de travail improbables. Merci à Anissa, Valentin et Maud pour m'avoir permis de râler et de discuter du monde académique aussi bien autour d'un verre que lors d'une randonnée. Nathan et Maëlle, pour votre bonne humeur, les vacances en votre compagnie sont toujours une incroyable aventure. Chloé-Lyne, Damien, Edward, Enora, Harry, Joslin, Julien et Tanguy pour les innombrables moments partagés et les heures passées à essayer de vous convaincre que le réseau c'est passionnant. Manon pour m'avoir supporté pendant la rédaction de ce manuscrit et de m'avoir souvent accompagné à l'aventure. Merci à tous mes amis d'enfance de m'avoir supporté, pour certains courageux, depuis plus de 20 ans, et m'avoir permis de me déconnecter le temps d'un week-end, d'un repas ou d'une soirée : Alexandre, Alizé, Aurélie, Guillaume, Chloé, Erwan, Hugo, Jerem, Mathilde, Max, Nico, Simon, Thomas. J'ai également une pensée émue pour ceux qui n'auront pas eu le temps de me voir finir ce long périple, Arno et Arthur.

Je souhaite finalement remercier ma famille, qui n'a jamais rien compris a ce que je faisais, mais qui m'a toujours soutenu. Particulièrement, mon père et mon frère, merci à vous deux d'avoir cru en moi tout au long de toutes ces années d'études.

À tous ceux qui liront ces lignes, outré que je les ai oubliés, je m'en excuse d'avance et vous remercie du fond du cœur.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AIAD	Additive Increase/Additive Decrease
AIMD	Additive Increase/Multiplicative Decrease
API	Application Programming Interface
CDN	Content Delivery Network
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
DDIO	Data Direct I/O Technology
DL	Deep Learning
DMA	Direct Memory Access
DNS	Domain Name System
DPDK	Data Plane Development Kit
DPI	Deep Packet Inspection
E2E	End-to-End
ECH	Encryption Client Hello
FPGA	Field-Programmable Gate Array

GPU	Graphical Processing Unit
GRO	Generic Receive Offload
GSO	Generic Segmentation Offload
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IPFIX	IP Flow Information eXport
IPS	Intrusion Prevention System
IQR	InterQuartile Range
ISP	Internet Service Provider
ISP	Internet Service Provider
LRU	Least Recently Used
LSTM	Long Short-Term Memory
ML	Machine Learning
MOS	Mean Opinion Score
MPPS	Million Packets per Second
NIC	Network Interface Card
NN	Nearest Neighbors
ONNX	Open Neural Network Exchange
PCAPNG	PCAP Next Generation
PCAP	Packet Capture
QoE	Quality of Experience
QoS	Quality of Service
REGEX	Regular Expression
RSS	Receive Side Scaling
SNI	Server Name Indication
SVM	Support Vector Machine
TLS	Transport Layer Security
TSO	TCP Segmentation Offload
TTD	Time To Decision
VoIP	Voice over IP

1 INTRODUCTION

1.1 Motivation

1.1.1 *Machine learning for Network Traffic Monitoring*

Networks are becoming increasingly complex nowadays. Traffic is faster, more encrypted, and more encapsulated. At the same time, operators and researchers are seeking to answer more sophisticated questions and monitor increasingly specific behaviors in order to achieve fine-grained observability and, ultimately, ensure the best possible QoE for end-users.

As network link speeds continue to rise, especially for Internet Service Provider (ISP), the number of packets processed per second increases accordingly, creating additional pressure on traffic analysis systems. This surge in data throughput demands more efficient and scalable solutions to handle the growing volume of network traffic. At the same time, the widespread adoption of encryption has significantly improved user privacy but introduced new challenges for network operators. Since most traffic is now encrypted, traditional techniques, such as Deep Packet Inspection (DPI) or rule-based systems, are becoming less effective [96]. Although some works relying on Domain Name System (DNS) or Transport Layer Security (TLS) Server Name Indication (SNI) can still be used for traffic classification, they are now considered to be obsolete, as countermeasures have already been standardized [32, 56, 59, 62, 63].

In this context, ML has emerged as a promising solution. It enables fast and complex inference over large volumes of data that are extracted or derived from network traffic. This derived data, known as *features*, is critical to the performance of any ML model, as it forms the input upon which inference relies. Importantly, ML is particularly suited for working with encrypted traffic, thanks to the use of statistical features such as time series, inter-packet timing, packet counts, and packet direction. These metadata-driven indicators carry meaningful patterns even when the payload is not accessible. Moreover, ML is naturally well-suited to networking environments, where data is abundant, highly structured, and often presented as time-series logs or metrics. Its ability to reveal hidden patterns makes it a powerful tool for a wide range of use cases. As a result, there is extensive literature on the application of ML in networking, addressing problems such as QoE inference, traffic classification, intrusion detection, and malware detection. However, despite the diversity and promise of these works, most of them require additional steps before they can be realistically deployed in operational environments.

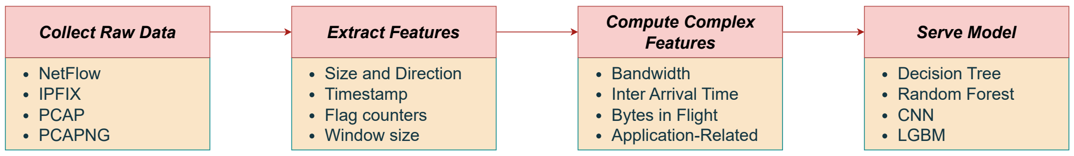


Figure 1.1: Machine Learning E2E pipeline

1.1.2 *Machine Learning End-to-End pipeline*

In the context of networking, ML involves much more than just models and features. As illustrated in Figure 1.1, it begins with the extraction of information from network data. This extraction can be performed using a wide variety of formats and tools, each with distinct characteristics and trade-offs. We can distinguish two major data sources:

- **Flow-based data**, such as NetFlow [24], or IP Flow Information eXport (IPFIX) [128] provides a high-level summary of traffic at relatively low computational and memory cost. These formats are particularly suited for coarse-grained monitoring and long-term trend analysis. However, they lack fine-grained details and significantly constrain the types of features that can be computed. As a result, it is often difficult, if not impossible, to retrospectively analyze specific behaviors once the data has been aggregated.
- **Raw packet capture**, typically in Packet Capture (PCAP) or PCAP Next Generation (PCAPNG) format, offers a near-complete snapshot of the network state, preserving full packet-level information. This makes it ideal for feature extraction and detailed behavioral analysis. However, raw captures come with substantial storage and processing costs, especially in high-speed network environments. For instance, a one-hour CAIDA trace[1] at an average rate of 1 Million Packets per Second (MPPS) can reach nearly 300GB.

[1] *Trace: CAIDA Equinix Chicago, 2016-01-21, Bidirectional*

After data collection, feature extraction converts raw network data into meaningful inputs for ML models. The complexity of this step depends on the data source and the chosen features. Simple features, readable from packet headers, are fast to compute but limited in detail, while richer features, such as time-based or statistical features, require more processing power and additional computation. These features necessitate a second step in which information is aggregated to provide more context. In practice, PCAP-based datasets remain the most convenient approach for academic ML research on network traffic, due to their richness and flexibility for feature design. This flexibility is especially convenient during the development of new ML models, where exploratory analysis and feature experimentation are crucial. However, the reliance on PCAP introduces scalability and deployment challenges that are often ignored in academic literature.

1.1.3 *Real-world constraints*

While PCAP offers a highly convenient format for model development and feature exploration, it typically captures only a partial view of network activity, often missing the full spectrum of behaviors observed in operational environments. This is precisely where a significant problem lies. Models trained and evaluated using snapshots may perform well on them, yet struggle to generalize in real-world, continuous traffic. Live network traffic is bursty, noisy, and lossy. It exhibits timing variations, anomalies, and protocol irregularities that are often underrepresented and difficult to capture in offline data.

This gap becomes particularly problematic at deployment time. Many existing models assume access to complete traffic flows or extended time windows, which enables accurate offline evaluation and model development. However, such assumptions are rarely met in operational environments due to resource constraints, packet loss, and system delays. While these models are often designed to support network monitoring, they tend to overlook the critical real-time requirements of operational settings. In practice, network operators of ISP need monitoring tools that can process data and make decisions at line rate to preserve end-user QoE. This creates a fundamental challenge: the full ML pipeline shown in Figure 1.1 must operate at line rate under realistic system constraints. This involves collecting, processing, and inferring from network data in real time while respecting tight limits on processing resources, memory, and latency.

Although system-level concerns are often considered secondary engineering details, they are nonetheless crucial for the practical deployment of ML in high-speed networks. Failing to consider system-level constraints can result in models that, while accurate on paper, will not work in practice. Therefore, any practical ML-based monitoring solution must be designed with a deep awareness of both network and system constraints. This requires operators and researchers to reconsider prevailing assumptions about input fidelity, pipeline architecture, and system issues that are not widely explored in existing research.

1.2 Problem Statement

Although ML-based methods have shown strong performance in controlled academic environments, they often fail when deployed in real-world [27]. In networking, this gap is largely due to the common practice of evaluating models in isolation from the data collection and feature extraction pipeline [15]. Such pipelines are often idealized, overlooking practical challenges such as packet loss, resource limitations, and high-throughput demands.

In operational network monitoring, tools must provide real-time insights while operating under stringent system constraints, including limited computational resources and memory. Static approaches are particularly problematic in this context, as they are unable to adapt to dynamic network conditions, leading to performance degradation or missed information.

This thesis focuses on two fundamental questions:

- How can adaptive feature extraction strategies mitigate packet loss in dynamic network environments compared to static configurations?
- How can ML-based classification systems be effectively deployed "off the shelf" for packet filtering at line-rate?

1.3 Thesis contributions

This thesis makes three main contributions, each addressing a different aspect of the challenges involved in deploying ML-based network monitoring tools in modern network environments.

First, we conducted an empirical study on the impact of packet loss on ML performance. We analyze the effects of bursty and sporadic packet losses on features at the packet level and on aggregated metrics. Then, we apply our findings to two classic ML tasks. This study provides insight into how packet loss affect model accuracy. This insight forms the foundation of the following contributions.

Second, we develop a feature extraction pipeline, called `Cruise Control`, that dynamically adapts the features collected based on the instantaneous state of the network and system. By monitoring low-cost local system metrics, the pipeline adjusts feature complexity to reduce resource consumption while reducing packet losses.

While the previous contributions focused on already filtered traffic, this final contribution aims to enable feature extraction systems, such as `Cruise Control`, to process more complex features by reducing the traffic load, using ML-based classification as a preliminary filter. This strategy reduces the overall data volume, but in doing so, we uncovered a bottleneck: directly applying off-the-shelf ML models in operational networks is resource-intensive. To address this challenge, we propose a novel hybrid approach that leverages simple, interpretable network-level rules derived from cached

ML decisions. This approach achieves a favorable balance between scalability and accuracy, enabling real-time deployment in resource-constrained environments.

Together, these contributions further the practical deployment of ML in network monitoring by bridging the gap between ideal academic models and the limitations of real-world systems.

1.4 Organization

This thesis is organized as follows:

- Chapter 2 provides a general overview of the state of the art in applying machine learning to networking, highlighting several key works that form the foundation for subsequent chapters.
- Chapter 3 investigates the impact of packet loss during feature collection and extraction. This study simulates both bursty and sporadic loss patterns while evaluating two types of features.
- Chapter 4 introduces a new system, called `Cruise Control`, that considers a pool of models rather than relying on a single one. The system dynamically selects the most appropriate model based on the current network and system conditions.
- Chapter 5 presents a concrete use case involving a machine learning task, specifically traffic classification, and explores its deployment in a practical setting, such as a traffic filter. The focus is on identifying the trade-offs required for deployment and proposing alternatives to overcome associated challenges.
- Chapter 6 concludes the thesis by summarizing key insights and outlining potential directions for future research.

Although each chapter is self-contained and can be read independently, they are designed to build upon one another. Therefore, a sequential reading is recommended, as questions raised in one chapter are often explored in the next.

2 BACKGROUND AND RELATED WORK

In this section, we provide an overview of the current state of the art in Network Traffic Analysis techniques, ML-based network analysis solutions, and ML-based approaches designed with system constraints in mind. This overview is intended to give readers the necessary context to better understand the subsequent chapters. This chapter introduces broad concepts and key ideas. More exhaustive related work, specific to individual chapters, will be presented later in their relevant contexts.

2.1 Network Traffic Monitoring

2.1.1 *Overview of Network Traffic Monitoring*

Networking fundamentally relies on packets that carry information from a source to one or more destinations. This is the responsibility of the network operator. They are responsible for making their isolated networks interoperable with others, thus enabling their customers to communicate with each other and with the rest of the Internet. Network operators are also bound by performance and security guarantees. Traffic monitoring is the key to ensuring these requirements. This goes beyond maintaining connectivity; it allows debugging, and troubleshooting traffic in real time to preserve availability, resilience, and QoE.

Monitoring consists of extracting information from the network using techniques such as port-based analysis or DPI, and inferring behavior, such as congestion, attacks, or malfunctions. Ultimately, it involves taking action, if needed, to ensure the network is functioning properly. The type of information monitored depends on the level of granularity: At the packet level, metrics capture properties such as size, timestamp, protocols and header fields. This offers insight into network events and supports traffic identification methods such as port-based analysis or DPI. At the *flow* [1] level, packet metrics are aggregated across multiple packets, enabling observation of higher-level behaviors, such as traffic volume patterns, inter-arrival times, and protocol dynamics. Together, packet and flow-level monitoring provide a comprehensive view of network behavior. Although there are many ways to monitor a network, this chapter presents

1 Several overlapping terms are used to describe a series of packet exchanges between a source and a destination. These terms include *flow*, *connection*, *session*, *5-tuple*, and *biflow*, among others. Although each term emphasizes different aspects, they generally refer to the same concept. In the following, we will use the term *flow*.

the two main use cases that will be used in the following chapters: traffic classification and QoE monitoring.

2.1.2 *Traditional Traffic Classification Techniques*

Traffic classification is the process of categorizing network packets into different classes, which can represent various levels of granularity such as protocols (TCP, UDP, ICMP), type of services (online gaming, audio, video streaming), or applications (Netflix, Facebook, Spotify). Traffic classification has always been considered a critical task by operators and researchers. It consists of identifying the service or application associated with a flow, which enables a variety of network management tasks. For example, in a saturated environment, Voice over IP (VoIP) packets requiring low latency can be prioritized to ensure QoE, while emails, which do not require latency guarantees, can be given lower priority. Beyond prioritization, traffic classification also supports intrusion and anomaly detection, enforcement of security and access-control policies, traffic engineering, load balancing and the monitoring of service-specific performance. At the same time, traffic classification can also be employed for more controversial purposes, such as censoring or limiting access to particular applications. This often leads to a cat-and-mouse dynamic between operators, who aim to ensure performances and QoE, and the security community, which seeks to protect privacy. This dynamic will be explored in more detail in the following sections, where we discuss measures and countermeasures.

PORT-BASED In 1992, RFC 1340 [105] attempted to establish a port space in which each application would have its own protocol and, thus, its own port. However, due to the increasing number of applications, this approach has become difficult to maintain. Furthermore, some applications obfuscate their ports by using other applications' ports or unreserved ports [28] to avoid being blocked or limited. Additionally, many applications are tunneled within HTTP [93], causing connections to use ports 80, 8080, or 443. Despite these limitations, port-based classification remains the fastest and simplest method. It is particularly useful for legacy applications or protocols and in scenarios where accuracy is less critical [75], as it requires reading only a single field.

PAYLOAD-BASED As port-based approaches became less reliable, payload-based methods, also called DPI, gained popularity [46, 84]. These methods perform pattern matching with signatures or Regular Expression (REGEX) directly on packet payloads. While highly accurate, they require significant computational resources, scale poorly, and raise ethical and legal privacy concerns [75]. Moreover, they are easily circumvented when payloads are encrypted [140].

IP-BASED The IP address is another valuable piece of information, as it is typically associated with a service. Identifying the IP can help determine the service. This technique, similar to port-based methods, relies on simple rules and is thus faster than DPI, which requires more complex comparisons. However, it suffers from scalability issues because it requires knowing all IP-service mappings in advance [119], or implementing methods to dynamically populate the IP table. Shared IP addresses and the widespread use of Content Delivery Network (CDN)s further complicate this approach. Nonetheless, combining IP addresses, ports, and protocols into the well-known 5-tuple remains the standard for flow identification.

DNS-BASED To overcome the limitations of static IP filtering, researchers developed techniques leveraging information orthogonal to the flow, such as DNS exchanges, which remain in clear text before HTTPS connections. By analyzing client DNS requests and responses, one can populate a dynamic IP table [34, 119]. However, this method is being rendered obsolete by the deployment of DNS over Encryption [83]. The Internet Engineering Task Force (IETF) has proposed various implementations, including DNS over TLS, HTTPS, and QUIC [56, 59, 63].

TLS-BASED For TLS connections, operators can use the TLS SNI field to identify end services, which is particularly useful for CDNs sharing IP addresses. Since the SNI is in clear text, operators can easily determine the service [119], although they must wait for the TLS Client Hello to complete. Countermeasures such as Encryption Client Hello (ECH) are being standardized to encrypt the entire Client Hello and optionally route traffic through a public proxy to anonymize end services. While deployed in web browsers such as Firefox (since version 85) [40], server-side adoption remains limited.

2.1.3 QoE *Monitoring*

QoE is a user-centric metric that captures the perceived quality of a service, such as web browsing, video streaming, or real-time communication. Beyond identifying services and applications, network operators are increasingly concerned with quantifying the actual QoE experienced by their subscribers [70, 124]. While traditional traffic classification provides visibility into the types of flows traversing the network, it does not directly reveal how well these services are performing or whether users are satisfied. QoE monitoring aims to bridge this gap by capturing user experience from measurable indicators.

Prior to the emergence of QoE, monitoring efforts primarily focused on Quality of Service (QoS) metrics [114]. Formally, RFC 2386 [26] defines QoS as “a set of service requirements to be met by the network while transporting a flow.” In practice, operators track metrics such as throughput, latency, and jitter, which reflect the network’s operational performance but remain agnostic to user perception. As a result, they offer only a partial view: while operators may measure uptime ratios, users care above all about seamless access to the service whenever needed [132]. This discrepancy highlights the need to complement QoS with user-centric measures, which QoE provides.

NETWORK-LEVEL METRICS Network-level QoE indicators are derived from traditional QoS measurements and aim to establish their relevance to user experience. Such indicators typically involve aggregating transport-level metrics (e.g., throughput, latency, packet loss) and identifying those most strongly correlated with perceived quality. Since they rely on observable network characteristics, they remain comparatively robust to traffic encryption. Nevertheless, when examined in isolation, network-level indicators often lack the granularity required to capture application-specific aspects of user satisfaction.

APPLICATION-LEVEL METRICS A substantial body of research on QoE monitoring has concentrated on video streaming, which is projected to account for more than half of global internet traffic by 2025 [110]. To ensure performance, major content providers such as Netflix deploy CDNs within or near operator networks, thereby reducing latency and improving responsiveness [11]. This architectural proximity also supports troubleshooting efforts: if multiple users simultaneously experience buffering while accessing Netflix, the underlying cause is more likely to reside within the operator's infrastructure than in the remote service itself. Consequently, video-specific QoE metrics, such as resolution, startup delay, or buffering frequency, have become particularly valuable for operational monitoring and troubleshooting. Nonetheless, obtaining such metrics frequently relies on DPI techniques [87, 129].

USER-LEVEL METRICS At the highest level, QoE can be measured directly from users, offering the most accurate assessment of perceived service quality. This is typically achieved through surveys and standardized instruments such as the Mean Opinion Score (MOS), which quantify user perception in a reproducible manner. For instance, ITU-T P.1203 specifies a standardized methodology for computing MOS values in video streaming scenarios [141]. Although these methods yield reliable ground truth, they are often costly, time-consuming, and challenging to scale, as they require extensive user participation across diverse content types and services [100].

Many traditional methods for QoE monitoring rely on application-level metrics, which are typically extracted using DPI. However, the increasing prevalence of encrypted traffic severely restricts the accessibility of payload-based indicators, thereby reducing the effectiveness of these methods. This limitation underscores the need for novel approaches capable of inferring QoE without direct reliance on payload inspection or large-scale user studies.

2.2 ML-based Network Monitoring

Encryption is now ubiquitous, according to Google's Transparency Report [50]. While vital for user privacy, it reduces the effectiveness of many traditional traffic analysis methods, a challenge noted as early as the 2000s [73]. Furthermore, recent networks carry tremendous traffic volumes at ever-increasing speeds, creating scalability challenges. Techniques like DPI struggle to keep up, and high-speed traffic introduces system-level constraints, such as limitations in the Linux kernel's networking stack [18].

This shift has motivated the exploration of alternative, scalable approaches that can operate without access to payload data. Among these, ML has emerged as a particularly promising and widely adopted solution. Unlike DPI, ML models can exploit features that remain observable in encrypted traffic, such as header information, statistical flow descriptors, and packet timing patterns, allowing them to operate even when payloads are inaccessible.

The application of ML to networking is not new. Early work such as BLINC [73] in 2005 demonstrated the feasibility of traffic analysis using behavioral patterns rather than packet content, pioneering behavior-based traffic analysis on encrypted flows and influencing later ML-based methods for traffic classification. One survey from the late 2000s [99] already documented a variety of statistical and machine learning techniques for classification. Since then, the field has evolved substantially: initial approaches generally combined classical algorithms (e.g., decision trees, Support Vector Machine (SVM)s, k-Nearest Neighbors (NN)) with hand-crafted statistical features [12, 72], whereas more recent methods increasingly leverage deep learning architectures capable of learning directly from raw packet sequences, time-series representations, or graph-based flow structures [1]. This evolution has broadened the applicability of ML to a wide range of networking tasks, from traffic analysis and intrusion detection to anomaly detection, malware detection, and service quality monitoring.

Given the vast diversity of ML applications in networking, this section focuses on two tasks presented in the last section, Traffic Classification and QoE inference.

2.2.1 ML-*based Traffic Classification*

ML offers an alternative to traditional classification methods by leveraging flow-level and header-based features that remain accessible despite encryption. These features typically include packet sizes, inter-arrival times, directionality, burst patterns, and statistical aggregates. By modeling these characteristics, ML-based approaches can distinguish between services and applications without examining packet payloads [122].

ML-based traffic classification spans from coarse-grained service identification (e.g., web browsing, streaming, video games, or file transfer) to fine-grained recognition of specific applications (e.g., Netflix, YouTube, or Spotify). Research has also explored early classification, which seeks to identify the service after observing only a limited number of packets [10, 51], a critical capability for real-time network management where early decisions can prevent congestion or QoE degradation. Approaches differ in their choice of features and models. While some adopt simple flow statistics and classical ML algorithms to prioritize computational efficiency[14, 122], others leverage deep neural networks to capture complex patterns in raw packet [106, 136], and more recently representations learning[146].

However, the reliability of these methods is inherently tied to the quality, completeness, and consistency of the collected data. Performance can deteriorate under dynamic network conditions or partial flow observations, highlighting a critical dependency on the underlying measurement infrastructure and exposing a key area for further research in system-aware ML-based traffic analysis.

2.2.2 ML-*based* QoE *Inference*

While traffic classification identifies what the traffic is, QoE inference seeks to estimate how well that traffic is performing for the end user. As encryption increasingly hides application-layer details, directly measuring QoE becomes challenging [76]. In this context, ML-based inference offers a way to estimate video quality metrics, startup delay, call quality, or other user experience indicators using only network-side observations.

Research in this area has focused primarily on video streaming services, due to their dominant share of modern Internet traffic and their sensitivity to performance fluctuations. ML models can be trained to infer application-level QoE proxies from flow-level features, such as burst size sequences, idle periods, or retransmissions [14, 55]. This approach enables operators to monitor service quality in real time without accessing or decrypting payload data, offering a practical solution for encrypted traffic while highlighting the critical role of robust and accurate feature extraction in ensuring reliable QoE estimation. Various modeling approaches have been explored over time, ranging from classical ML algorithms [33, 90] to deep learning models [121], and more recently, representation learning techniques [137].

Nevertheless, the effectiveness of ML-based QoE inference heavily depends on the quality and completeness of extracted features. Limitations in the feature extraction process, such as incomplete flows or coarse-grained measurements, can degrade prediction accuracy. The field has made incredible progress over the past two decades and remains extremely active. However, real-world deployment still faces constraints in feature extraction, robustness under diverse conditions, and scalability to high-speed networks. Addressing these limitations is central to the contributions of this thesis.

2.3 Monitoring with System Constraints

As presented in the previous section, ML is highly effective for addressing modern networking challenges. However, as noted in Chapter 1, system-level constraints, such as processing capabilities, memory limits, and thus packet loss, are frequently under-examined in the literature. Many models, developed and validated on idealized datasets, fail when deployed in real-world environments [7, 27]. Research often focuses on the model itself rather than the full pipeline for extracting features from network traffic. However, if the pipeline fails, the performance of the entire ML model can be severely compromised. In this section, we discuss several bottlenecks and corresponding strategies from the literature. We start with modifications to ML models themselves, then review feature extraction optimizations, system-level packet processing improvements, and finally dedicated hardware solutions.

2.3.1 *Model Pruning*

Model pruning involves simplifying models to reduce their computational cost, improve execution speed, and lower memory usage. This approach focuses on optimizing the model itself rather than the data pipeline. Advantages include making models deployable in resource-constrained environments [133], reducing execution latency, and decreasing overall model size [23]. However, pruning can lead to overfitting, accuracy degradation, and requires extensive engineering effort for fine-tuning. Furthermore, recent studies suggest that model execution often represents only a fraction of the pipeline's total processing time [81], making pruning a suboptimal focus in some cases.

2.3.2 *Feature Engineering*

Feature computation is a critical step in ML pipelines, and inefficient design can lead to packet loss. Feature engineering aims to reduce the number of features or packets processed to essential metrics only. For high-speed environments, several works target optimization. For instance, Retina [134] selectively processes only a subset of packets through optimized filtering, achieving high throughput at the expense of full-traffic visibility. However, this approach is less flexible in dynamic networks, requiring system downtime and recompilation for configuration changes.

Other approaches aim to standardize feature representation for broader applicability. NPrint [58] extracts features from packet headers at the bit level, while FlowPic [116] represents flows in a structured manner. These methods reduce dependence on protocol-specific details but can introduce significant computational and memory overhead, particularly when processing irrelevant or small packets. Overall, feature engineering often results in restricted features or limited packet coverage, primarily due to the dynamic nature of network traffic.

2.3.3 *Optimized Software*

High-speed networking introduces additional bottlenecks, even within the Linux kernel's networking stack. Software optimizations have been developed to mitigate these issues. A widely adopted solution is Data Plane Development Kit (DPDK)[29], which allows applications to bypass the kernel stack and interact directly with the Network Interface Card (NIC). By using techniques like continuous polling, DPDK achieves high throughput and low latency, but developers must implement packet-processing logic themselves.

Other optimizations work within the kernel and NIC, leveraging features such as segmentation offload [97], TCP Segmentation Offload (TSO), and Generic Receive Offload (GRO)/Generic Segmentation Offload (GSO), which reduce per-packet processing overhead. Recent work explores redesigning NIC–kernel interactions [111] or optimizing queueing mechanisms [69] to further improve efficiency.

2.3.4 *Dedicated Hardware*

Dedicated hardware has long been used in networking, but programmable devices such as P4 switches[139], FPGAs [39], and SmartNICs [133] now allow custom ML pipelines to run at line rate. Hardware acceleration can offload parts of the processing pipeline, providing substantial computational gains. However, these approaches require specialized engineering skills. Moreover, the number of native operations is often limited by hardware design, imposing constraints similar to those seen in model pruning.

2.4 **Limitation of Current Approaches**

While existing technologies provide a wide range of tools for monitoring and analyzing network traffic, they overlook a crucial point. Network traffic is inherently dynamic, its patterns, volumes, and behaviors fluctuate over time. These fluctuations can occur at multiple timescales: abrupt spikes may result from sudden events, or bursts of specific applications, while gradual variations reflect daily cycles, weekly patterns. Applying a static solution, no matter how optimized, is fundamentally incompatible with this dynamic nature and inevitably leads to packet loss or reduced analysis accuracy. This observation leads to the core argument of this thesis: solutions must be dynamic and adaptive, following changes in network context rather than being limited by static approaches.

To overcome these limitations, we, firstly, study the actual impact of packet loss on ML-based network analysis (this issue is examined in Chapter 3). Secondly, based on the observation that relying on a single static configuration in dynamic network environments is suboptimal. We addressed this limitation by introducing adaptive feature extraction strategies that adjust to changing conditions (in Chapter 4). Allowing the system to be able to follow changes in network context during days and also quickly react in case of bursts. Finally, we investigate a real-world deployment of an ML-based classification system used as a packet filter. We analyze the shortcomings of deploying such systems “off the shelf” and propose strategies to overcome these limitations (presented in chapter 5).

3 THE COST OF PACKET LOSS ON ML-BASED TRAFFIC ANALYSIS

Johann Hugon et al. "The Cost of Packet Loss on ML-Based Traffic Analysis." In: 2025 IEEE 31th International Symposium on Local and Metropolitan Area Networks (LANMAN). *2025[61]*

In this chapter, we investigate the impact of packet loss on the performance of ML-based traffic analysis systems. As losses introduce bias in the final features set provided to the machine learning model, we hypothesize that they will negatively impact model performance. We evaluate this hypothesis by analyzing the performance of two different ML tasks presented above, service classification and QoE analysis (video startup delay). Both are trained on a dataset of video flows, and we measure the impact of two different packet loss models: probabilistic and bursty losses. Our results show that sporadic packet loss has little impact on performance. Conversely, bursty losses, which are more common for packet processing systems, can lead to a significant negative impact.

3.1 Introduction

Deploying ML systems for traffic analysis on live, high-speed links requires a preprocessing pipeline that is able to operate within strict system constraints [81, 126, 135]. Preprocessing pipelines typically consist of several critical stages: packet extraction from the link, protocol parsing, filtering, and feature computation. Yet, the systems responsible for these actions face fundamental operational constraints that can lead to packet loss, which has the potential of compromising the quality, or even correctness, of features provided to downstream ML models.

Various approaches to network traffic analysis face different constraints. Traditional tools like Tcpdump [127] allow capturing network traffic, but often struggle with high-speed links due to kernel network stack bottlenecks [18]. While storing traces for offline usage might reduce processing and memory constraints, the cost of uploading large data volumes to a remote location becomes critical during high traffic times. More modern techniques [22], such as zero-copy packet processing through technologies like DPDK [29] or XDP [57], can prevent memory and storage exhaustion by reading packets directly from NIC memory or cache, but require the entire pipeline to process packets at line-rate—when processing capacity cannot match network throughput, packets are inevitably dropped, resulting in information loss that propagates to ML model features. In-network solutions like in-switch ML [78, 102] offer an alternative approach but provide very limited resources due to the need to perform routing in parallel with analysis tasks.

Of course, the dynamic nature of network traffic makes it nearly impossible to design measurement systems that operate without packet loss under all conditions, if not at the cost of reduced model accuracy. Even pipelines that perform well in controlled environments often fail when confronted with real-world traffic pattern[7, 27]. This raises a fundamental, yet under-explored question: *How does packet loss affect the performance of* ML*-based traffic analysis systems?* Recent work [8] has shown that packet loss can have significant effects on the performance of ML models. However, while this work proposes solutions to mitigate such degradation, it fails to provide a comprehensive understanding of the underlying impact that packet loss has on the performance of ML models. In this chapter, we aim to fill this gap by answering two key questions:

Q1. Does packet loss affect model performance independently of its distribution? Recent work by Babaria et al. [8] has demonstrated that packet loss can heavily impact accuracy for service identification models. However, they solely used a simplistic probabilistic model for loss, where every packet is considered independently. In contrast, in operational environments, packet loss typically occurs in bursts during periods of system stress [15]. Rather than isolated drops, these bursts—ranging from a few packets to thousands—disrupt the temporal continuity of features. We hypothesize that this pattern of loss more accurately reflects real-world conditions and has a disproportionately negative impact on model performance compared to random individual packet drops.

Q2. In the presence of loss, does accuracy degradation depend on ML *model input features?* In their paper, Babaria et al. [8] discuss how lost packets lead to missing or distorted features that may compromise model accuracy for various service identification solutions. Without complex (i.e., computationally expensive) stateful connection tracking, these inconsistencies become difficult to identify. For example, in DPDK-based systems, incoming packets can overwrite older, still unread packets in the buffer before processing completes, creating undetected gaps in the data. In such cases, the model may receive features that are inconsistent with the true state of the network. However, while these distortions can be evident in the case of service identification, where only a handful of packets are used to build features, they might be less evident for scenarios like video startup time inference where features are mostly aggregates collected across many (i.e., hundreds if not thousands) of packets.

To answer these questions, we focus our work on two distinct ML-based traffic analysis tasks: (1) service identification, a classification problem; and (2) video startup delay inference, a regression problem. For these use cases, we study how packet loss affects performance metrics for these by applying both a probabilistic model that randomly drops individual packets, as well as a two-state Markov chain model that simulates realistic bursts of packet drops. We find that the impact of sporadic drops is noticeable, but negligible below low thresholds. However, bursty drops can have a large impact on model performance, even with a low probability of occurrence.

The remainder of this chapter is organized as follows: Section 5.2 discusses relevant prior work; Section 3.3 details our experimental methodology; Section 3.4 presents results across both use cases; finally, Section 5.5 summarizes our findings and their implications. All analysis code and results are made available as a Jupyter notebook[1] to ensure reproducibility.

3.2 Related Work

The quality of datasets is widely recognized as a crucial factor affecting the overall performance of ML models [2, 49]. While prior research (such as Mauri et al. [89]) has investigated the impact of data poisoning, where adversaries deliberately modify training data, our work specifically addresses data quality issues at inference time rather than during model training.

Foroni et al. [47] generated variations of the same dataset with different noise levels to measure the impact on task performance. Our approach is similar in that we introduce controlled noise—specifically packet loss—at predetermined rates to evaluate performance degradation. However, our work is distinguished by its application to network systems and consideration of domain-specific constraints such as bursty packet losses.

More recently, Cavitt et al. [21] investigated the negative effects of packet drops in power systems. They employed machine learning models to detect losses and implemented various replacement policies to mitigate the impact. Their research demonstrated that performance degradation could be minimized through appropriate data replacement strategies. While their work focused on spoof detection in power systems, our study examines detectable but untraceable losses in network environments.

Yang et al. [143] addressed packet loss in encrypted traffic classification by developing an Anti-Packet-Loss method based on a Masked Autoencoder. Their approach intentionally masks portions of training traffic data to enhance the encoder's ability to reconstruct missing information. Their evaluation showed 90% classification accuracy even with 15% packet loss, significantly outperforming conventional deep learning methods. Although their research targets the same problem of packet loss during inference, our work differs by evaluating the impact on machine learning models that are not specifically designed to handle packet drops.

Most recently, Babaria et al. [8] proposed FastFlow a solution to mitigate the effects of packet loss on ML models for service identification. FastFlow employs a sequential decision-based classification model that leverages a collection of Long Short-Term Memory (LSTM) models trained with reinforcement learning to infer traffic classes using the first few packets of a flow. In the evaluation, the authors show that FastFlow is

1 https://github.com/ENSL-NS/Cost-of-Packet-Loss

Service Identification	**Video startup delay inference**
Packets size, count	Packets size, number
Packets inter arrival time	Throughput
Bytes per packets	Segments size, count
TCP Flags, Window, RTT	Segments duration
Bytes per packet	Inter segments time
Bytes in flight	
Retransmissions	

Table 3.1: Features Sets

more resilient to packet loss than other state-of-the-art ML models. However, their work solely applies per-packet probabilistic losses and does not provide a comprehensive understanding of the underlying impact that packet loss has on the performance of ML models. To the best of our knowledge, our work is the first to systematically study the impact of packet loss on ML-based traffic analysis systems, using both probabilistic and bursty loss models and on multiple traffic analysis tasks.

3.3 **Methodology**

In this section, we discuss the methodology used to evaluate the impact of packet loss on ML-based traffic analysis systems. We begin by describing the two use cases we selected for our analysis: service identification and video startup time. We then discuss the dataset used for our analysis, including the packet loss models applied.

3.3.1 *Traffic analysis tasks*

We focus our analysis on two well-studied tasks in the network community: Service identification [58] and ideo startup time inference [14].

SERVICE IDENTIFICATION. Service identification is a typical traffic classification task [38, 115]. The task involves classifying network traffic at the flow level into corresponding applications, such as YouTube and Netflix. We focus particularly on early application identification, which consists of using the first few packets—typically ten or fewer—to identify the application [58]. In our scenario, we utilize the first ten packets and derive the set of features described in Table 3.1. These features are based on network parameters such as byte quantity in each direction, packet inter-arrival time, and transport metrics like TCP windows or bytes in flight. All features listed in

Table 3.1 are divided by direction: client-to-server and server-to-client. Where possible, they are further subdivided into various statistical metrics, including standard deviation, average, maximum, minimum, median, kurtosis, and skewness. This approach aims to capture hidden patterns that raw features may not effectively reveal. While some existing works use fewer packets (e.g., only the first four), we elected to use ten packets as a best-case scenario, recognizing that using fewer packets would render the analysis even more susceptible to packet loss and distortion of input features.

VIDEO STARTUP TIME INFERENCE. Video startup time inference is a fundamental task in QoE analysis that involves predicting the duration between a user's request to play a video and the moment playback actually begins. This metric is widely recognized as a critical factor affecting user satisfaction and engagement with video streaming services. In our scenario, we leverage network traffic features collected during the initial connection and buffering phases to infer the startup delay experienced by users. Unlike service identification, which relies on the first few packets, video QoE inference typically utilizes larger temporal windows of aggregated data—generally spanning multiple seconds, such as ten-second intervals. We specifically focus on startup time prediction for two strategic reasons: first, it enables us to validate the hypothesis that larger windows of aggregated data might exhibit greater resilience to packet loss; and second, it presents a regression problem rather than classification, thereby allowing us to evaluate the impact of packet loss on a different class of ML tasks with continuous output variables. For this use case, we do not consider features collected from the transport layer, as they have been demonstrated to be less effective for this task [14]. Instead, we focus on features related to the video segments, such as the number of segments downloaded, the size of each segment, and the time taken to download each segment. We also include features related to the inter-segment time, which is the time between the end of one segment and the start of the next. This approach allows us to capture the dynamics of video streaming and how they relate to startup delays. Finally, as listed in Table 3.1, we extract features about traffic volumes.

3.3.2 *Dataset and Model Training*

DATASET. Our analysis is based on a subset of the dataset collected by Bronzino et al. [14] in their work on video quality inference. The dataset consists of 9,213 labeled traces from four major video streaming providers (Netflix, YouTube, Amazon Prime Video, and Twitch), which we split into two parts: a training set comprising 7,390 traces (80.21%) and a testing set containing 1,823 traces (19.79%), with some traces excluded due to labeling issues. Subsequently, each trace was divided into separate traces for individual flows, and the data was filtered to include only video-related flows. This process yielded a training dataset of 2,535,163 flows and a testing dataset of 627,524 flows, maintaining a similar split ratio of 19.84%. The slight variation in percentages occurs because some traces contain more flows than others.

MODEL TRAINING. For both tasks, we selected Random Forest [13] as classification and regression algorithm, as previous research has demonstrated its superior precision and recall with lower false positive rates for our use cases [14, 58]. For model training, we employ AutoGluon [41], a widely adopted AutoML library. AutoGluon automatically explores effective combinations of input features and hyperparameters to generate the most effective model. Note that the goal of this paper is not to achieve the best possible model performance, but rather to evaluate the impact of packet loss on the performance of ML models. Therefore, we did not perform any hyperparameter tuning or feature selection beyond what AutoGluon provided. To mitigate overfitting, we constrained the minimum sample split to 10 and limited the maximum depth of the trees to 10. These constraints reduce the likelihood of capturing insignificant noise patterns and enhance the model's generalization capabilities. The model was trained on 80% of the dataset without any application of packet loss, ensuring that it learned from clean, uncompromised data. Packet loss was introduced exclusively during testing to simulate real-world inference conditions, based on the assumption that the model had been trained on a lossless dataset. We evaluated the impact of varying packet loss rates on model performance using the weighted F1 score as our primary metric for classification. This weighted scoring approach was chosen to prevent misinterpretation of results due to class imbalance in the dataset, while the F1 score itself provides a balanced measure that accounts for both precision and recall. For the regression task, we used the error distributions to observe a detailed analysis of the model's performance.

3.3.3 *Loss Models*

We aim to evaluate the impact of packet drop on model performance by implementing two distinct loss models: a probabilistic model and a bursty model.

PROBABILISTIC LOSS MODEL. In this initial model, we establish a probability p of packet drop. This probability is applied independently to each packet within a flow and to each flow within the dataset. This approach yields an aggregate loss measure for the dataset, expressed as a percentage of total packets lost. By increasing the value of p, we systematically increase the overall percentage of loss in the dataset. While percentage of packet loss is a common evaluation metric in network systems, this approach has certain limitations, particularly in its inability to adequately capture the nuanced behavior of packet loss within real-world network pipelines.

BURSTY LOSS MODEL. The second approach aims to more effectively characterize burst drops by representing them as a two-state Markov chain. Markov chains have

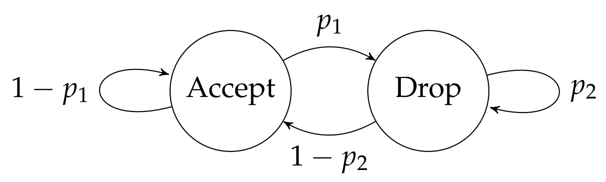

$$\begin{tikzcd}
\text{Accept} \arrow[loop left, "1-p_1"] \arrow[r, bend left, "p_1"] & \text{Drop} \arrow[loop right, "p_2"] \arrow[l, bend left, "1-p_2"]
\end{tikzcd}$$

Figure 3.1: State Transition Diagram of the Burst Models

been widely employed to model packet drops for decades [37, 48]. They are particularly effective for describing bursts of drops by defining a "good" state and a "bad" state, along with the various probabilities of transitioning between them or remaining in the current state. In our implementation, these two states correspond to the Accept state and the Drop state, as illustrated in Figure 3.1. As shown in the figure, p_1 represents the probability of transitioning from the Accept state to the Drop state, while p_2 denotes the probability of remaining in the Drop state. By calibrating these transition probabilities, we can simulate different frequencies and intensities of burst drops. A high value of p_1 increases the likelihood of burst initiation, representing the probability of network degradation in a system. Meanwhile, p_2 represents the probability of continuing the burst of dropping packets, which leads us to interpret $1 - p_2$ as the probability of resolving this drop condition. The lower the value of p_2, the more rapidly the pipeline will respond to the burst and recover from the drop condition.

APPLICATION OF LOSS MODELS TO THE DATASET. We apply our two loss models to the test portion of the dataset, which comprises 627,524 flows containing a total of 201,164,351 packets. The resulting packet drop statistics are presented in Tables 3.2 and 3.3. In Table 3.2, the first column represents the probability p of dropping a packet, the second column shows the total number of remaining packets after applying the losses, the third column displays the number of packets dropped, the fourth column indicates the resulting percentage of packet loss, and the final column the quantity of missing flows. Table 3.3 follows a similar structure, with the distinction that the first column p from Table 3.2 is replaced by two columns, p_1 and p_2, representing the transition probabilities of our two-state Markov chain. The data demonstrate that the probabilistic drop closely aligns with the overall percentage of packet loss observed in the trace. Notably, even a relatively low probability of burst initiation (p_1) can result in substantial packet loss, exceeding 10% of the original trace.

It is important to note that the loss models are applied before extracting the first N packets of a flow. Consequently, in scenarios where ML models require data from the first ten packets and packets between positions five and nine are dropped, features are subsequently extracted from packets zero to four and from ten to fourteen. This approach ensures that the ML model consistently receives the same quantity of packets for each flow, maintaining consistent input dimensionality despite varying packet loss

p	**# Packets**	**Diff**	**%**	**# Missing Flows**
0.0	200,164,351	0	0.00	0
0.005	199,164,584	999,767	0.50	145
0.01	198,162,375	2,001,976	1.00	288
0.02	196,158,686	4,005,665	2.00	446
0.05	190,155,893	10,008,458	5.00	2,680
0.1	180,148,648	20,015,703	10.00	9,946

Table 3.2: Traces after application of probabilistic loss

p_1	$1 - p_2$	**# Packets**	**Diff**	**%**	**# Missing Flows**
0.0	0.0	200,164,351	0	0.00	0
0.005	0.1	190,644,830	9,519,521	4.76	10,800
	0.01	133,593,296	66,571,055	33.26	168,647
	0.001	33,560,174	166,604,177	83.23	399,910
	0.0001	4,285,699	195,878,652	97.86	506,266
0.01	0.1	181,957,332	18,207,019	9.10	25,673
	0.01	100,084,829	100,079,522	50.00	271,475
	0.001	18,320,521	181,843,830	90.85	474,142
	0.0001	2,186,921	197,977,430	98.91	551,860

Table 3.3: Traces after application of burst loss

patterns. However, as mentioned in 3.2 and 3.3, a large drop can lead to dropping the entire connection. Thus, it is mandatory to include these missing flows when evaluating the performance of the models, as forgetting them can lead to high accuracy despite a high drop rate.

3.4 Analysis

In this section, we describe our analysis on the two use-cases: service identification and video quality inference. For both use-cases, we first present results on probabilistic losses, followed by results on bursty losses.

3.4.1 *Service Identification*

PROBABILISTIC. We apply the probabilistic loss model to the test partition of the dataset, with results presented in Table 3.4. In this table, $p = 0.0$ represents the baseline scenario with no packet drops, where data is served to the model without alteration. This serves as our benchmark for comparison; the F1 score reaches 0.971 under these optimal conditions. We note that accuracy and F1 score exhibit similar values, indicating both strong precision and recall across most classes, even without any packet drops. We also highlight that, for results with packet drops, we treat the missing connections as misclassified to avoid artificially inflating performance metrics.

At $p = 0.005$, we observe only a minor reduction in F1 score, which remains robust at 0.967. A slightly higher loss rate of $p = 0.01$ further decreases the F1 score to 0.963, though the impact remains relatively small. However, when the loss rate increases to $p = 0.02$, the F1 score drops more noticeably to 0.954, indicating the beginning of performance degradation. Beyond this threshold, the decline becomes more pronounced, with the F1 score decreasing to 0.925 at $p = 0.05$ and further to 0.879 at $p = 0.1$. These results demonstrate that while low levels of packet loss ($p < 0.02$) have minimal effect on classification performance, significant degradation occurs once packet loss exceeds $p = 0.02$, with substantial performance decline at higher loss rates. However, while the impact is noticeable, we observe that results remain acceptable even at high loss rates, with F1 scores close or above to 0.8.

BURST. As illustrated in Table 3.5, we apply the burst model to the test dataset. First, we notice that even when $1 - p_2$ is low, the F1 score remains more affected than with probabilistic losses, at equivalent percentages of drops. This can be explained by examining Table 3.3, where drops for $p_1 = 0.005$ and $p_2 = 0.1$ eliminate only 4.76% of the overall packets while missing $10,800$ flows, whereas its equivalent in the probabilistic model at $p = 0.05$ drops 5% of the traffic for merely $2,680$ flows. This demonstrates the potential context loss caused by burst patterns. When burst losses occur, the model may miss entire connections or long portions of them, leading to a more significant impact on performance.

However, as $1 - p_2$ reaches 0.01, the F1 score greatly drops, falling below 0.6. This suggests that burst loss, even at moderate levels, has a more detrimental effect on performance compared to the probabilistic loss model. As $1 - p_2$ increases further to 0.001, where a large portion of the traffic has been dropped, the decline in performance becomes more pronounced. This indicates that burst loss patterns have a significant negative impact on the model's ability to maintain performance, even with a slightly lower probability of occurrence.

p	**Accuracy**	**F1 Score**
0.0	0.972	0.972
0.005	0.967	0.967
0.01	0.964	0.963
0.02	0.954	0.954
0.05	0.927	0.925
0.1	0.885	0.879

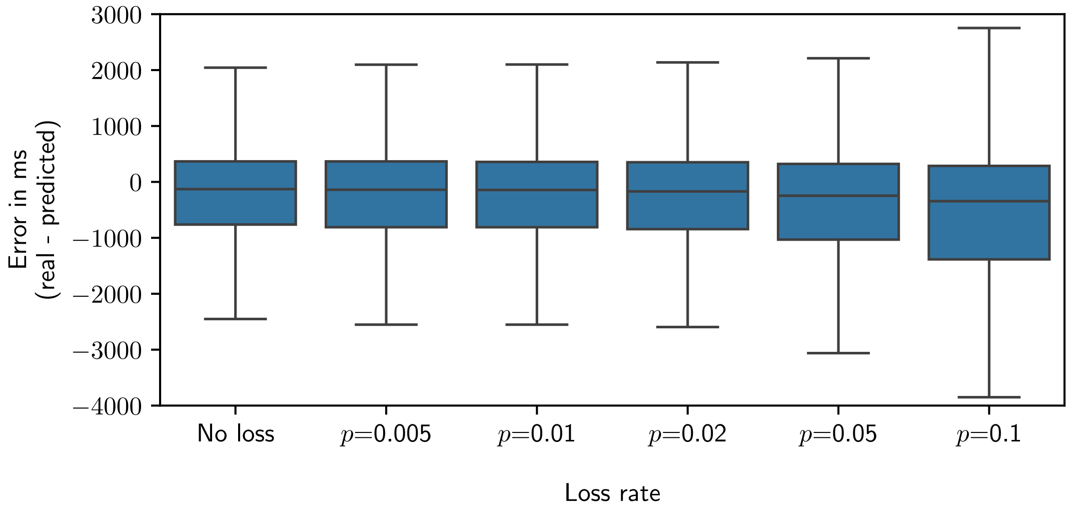
Table 3.4: Impact of probabilistic losses on Service Identification use case

p_1	$1 - p_2$	**Accuracy**	**F1 Score**
0.0	0.0	0.972	0.972
0.005	0.1	0.926	0.918
	0.01	0.618	0.551
	0.001	0.265	0.208
	0.0001	0.152	0.108
0.01	0.1	0.878	0.862
	0.01	0.432	0.363
	0.001	0.159	0.117
	0.0001	0.087	0.061

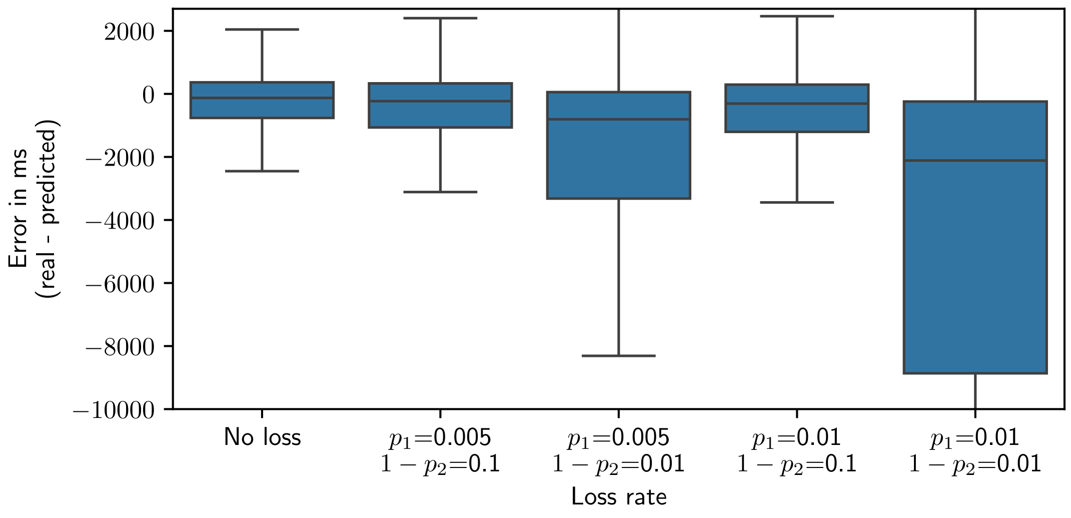
Table 3.5: Impact of burst losses on Service Identification use case

3.4.2 *Video Startup Delay Inference*

PROBABILISTIC. Similarly to the previous use case, we apply the probabilistic loss model to the test partition of the dataset, with results presented in Figure 3.2a. In the figure, we show error distributions for the regression task, as the difference between the real startup time collected from the video player and the predicted startup time from the model. The box plots show both median and interquartile ranges (InterQuartile Range (IQR)). We observe a slight impact on the median error as p increases. Starting at -129 for $p = 0$, we observe a phase of stagnation with a median at -140 for $p = 0.005$; -144 for $p = 0.01$; it begins to drop at $p = 0.02$ with a median of -170; it further decreases at $p = 0.05$ to -246 before ultimately reaching -344 at $p = 0.1$. Overall, while the error increses, the median remains relatively stable until $p = 0.05$. This indicates that the model remains relatively consistent while suffering from packet drops. While



(a) Probabilistic loss



(b) Burst loss

Figure 3.2: Impact of loss on the ideo Startup Delay Inference use case

we observe greater variation for IQR values, symptomatic of the model predicting less consistent values, the overall performance remains relatively stable. This confirms the intuition that using aggregates over many packets makes the model less sensitive to sporadic packet loss.

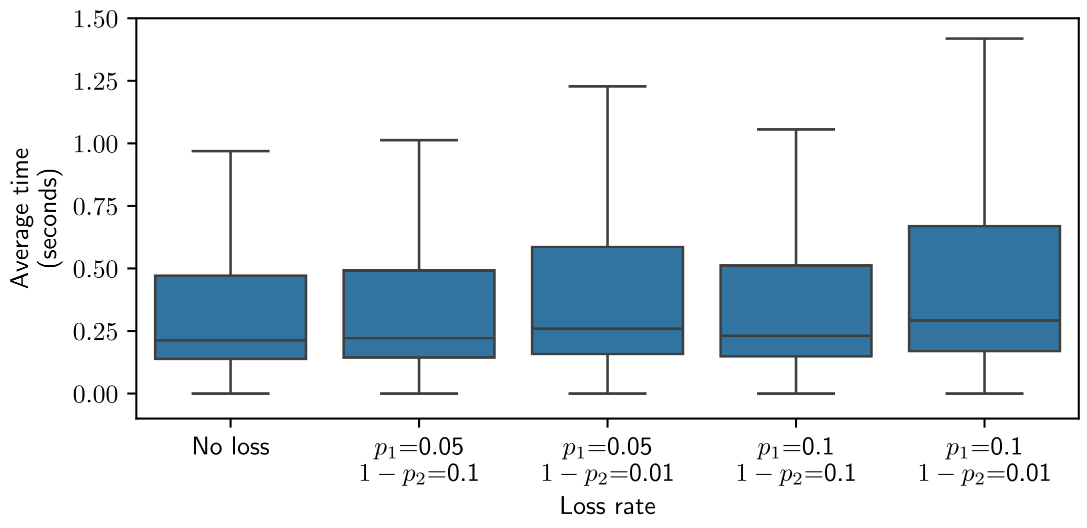


Figure 3.3: Average time to download a video segment

BURST. In contrast to probabilistic losses, for the burst loss we observe a much greater impact as soon as we reach $1 - p_2 = 0.1$ (as illustrated in Figure 3.2b), with the median dropping to −230 and −308. We do not show in the figure the results for higher burst lengths for space saving reasons. However, we observe that the median drops heavily to −806 for $p_1 = 0.005$ and −2114 for $p_1 = 0.01$. This indicates that the model is significantly affected by burst losses, even at low probabilities of occurrence. The IQR also increases significantly, indicating a wider range of errors in the predictions.

To understand the reason behind this behavior, we analyze the features used by the model. We first study the feature importance obtained at training time using the mean and standard deviation of accumulated impurity decrease within each tree of the Random Forest model. We observe that the most important features all relate to the download of video segments during the initial streaming phase. This is reasonable, as the model attempts to capture how quickly the video player is capable of downloading the necessary video buffer required to start reproduction. In particular, we observe that the most important feature is the average time to download a video segment, which effectively represents this dynamic (i.e., smaller values suggest higher download rates). We then study the distribution of the values of this feature across different test datasets (shown in Figure 3.3). We observe that the distribution of the feature is significantly affected by the burst loss model, with median values increasing as drops increase. This result suggests that key packets, used for the segment identification technique [14], are being lost, leading to a significant increase in the detected time to download video segments.

Overall, this result shows that, contrary to expectations, even a model that relies on aggregates over many packets can be significantly affected by burst losses.

3.5 Conclusion

In conclusion, this chapter provides an overview of how packet loss in the feature extraction pipeline may affect the performance ML model performance. We present comprehensive results for two use cases, evaluated under both probabilistic and burst loss models. Our findings demonstrate that while probabilistic losses with low p values ($p < 0.02$) have minimal impact on model performance, burst losses cause significantly more degradation even at equivalent drop rates. This holds true for both service identification and video startup delay inference tasks, counter-intuitively to the notion that models that use aggregate statistics as input features are less affected by packet loss. To enhance transparency and reproducibility, we provide a Jupyter notebook[2] containing all code, data, analyses, and visualizations to enable replication of our findings. In the next chapter we will see strategies to avoid and solve losses as soon as possible to avoid deteriorating too much model performance.

2 https://github.com/ENSL-NS/Cost-of-Packet-Loss

4 CRUISE CONTROL: DYNAMIC MODEL SELECTION FOR ML-BASED NETWORK TRAFFIC ANALYSIS

Johann Hugon et al. Cruise Control: Dynamic Model Selection for ML-Based Network Traffic Analysis. *2024. arXiv:* `2412.15146 [cs.NI]`[60]

In this chapter, we present a system-driven approach to dynamically select ML models based on their computational complexity and available system capacity—in real-time. Our proof-of-concept implementation, `Cruise Control`, supports multiple parallel ML tasks with efficient resource allocation across multiple analytical functions. Experimental results using two real-world traffic analysis tasks show `Cruise Control` improves median task accuracy by 2.78% while reducing packet loss by a factor of four compared to statically-selected models. In addition, `Cruise Control` offers an easy-to-configure solution for extracting multiple tasks, while effectively combining them and selectively collecting features.

4.1 Introduction

ML has rapidly become an indispensable tool for network traffic analysis tasks, thanks to ML models' excellence at discovering complex relationships between network traffic and critical events occurring across different network layers. Consequently, the networking community has increasingly developed sophisticated ML-based solutions to assist with essential tasks such as traffic classification [10, 68, 104, 106, 116], QoE inference [14, 86, 88, 117], intrusion detection [3, 74, 79], and numerous other critical network analysis functions [12, 98, 123]. However, deployment of ML models in operational networks remains a considerable challenge due to the dynamic nature of network traffic and the system requirements imposed by the need to process large volumes of data in real-time.

ML-based traffic analysis deployments typically employ a multi-stage pipeline that operates under strict real-time constraints [15, 135]. This pipeline consists of three critical stages: (1) traffic collection at a network vantage point, (2) transformation of raw packets into feature representations suitable for ML models, and (3) model execution to generate analytical outputs. Each stage must operate within tight processing budgets to prevent packet loss, which can severely inhibit model performance (i. e., accuracy). For example, this performance degradation is particularly severe for inference tasks where discriminative features often appear in the initial packet exchanges of a network flow [8]. Missing these early packets can drastically reduce classification accuracy, rendering the entire analytical pipeline ineffective.

Prior research has addressed this challenge through two main approaches. First, by designing specialized models that minimize packet requirements [8, 104]. However, models using this approach remain vulnerable to packet loss when processing budgets are exceeded and fail to scale to modern network speeds of tens or hundreds of gigabit per second. The second approach focuses on developing frameworks that help operators balance tradeoffs between model performance and system constraints such as latency [135], CPU utilization [15], and memory consumption [68]. In particular, these works have targeted the first two pipeline stages—traffic collection and traffic transformations—demonstrating that these stages often create bottlenecks in processing pipelines, thus requiring optimization to support downstream models. However, these solutions are limited to targeting static deployment of a single model during runtime.

The static model approaches face fundamental limitations: network deployment environments exhibit significant variability in topology, traffic patterns, and resource constraints–making it impossible to select a universally optimal model. Given this variability, operators typically resort to deploying models designed for worst-case traffic scenarios to avoid packet loss during peak periods. This conservative approach introduces a clear systematic inefficiency: during normal (e. g., non-peak) conditions, the worst-case model results in lower accuracy than that which could be achieved by a more complex model, given system resources and load [109].

In this chapter, we develop `Cruise Control`, a complementary, system-driven solution to adaptively select the best-fit target models for one or more feature extraction pipelines in parallel, removing the need for operators to select the best performing models for any given environment *a priori*. By leveraging real-time insights into the current system capabilities and traffic characteristics, `Cruise Control` dynamically selects the ideal target model from a pool of candidates for each task, ensuring that the system can effectively characterize traffic as network conditions evolve while avoiding packet loss. Of course, shifting to such approach requires tackling new system challenges. First, the system must be able to adapt to changing network conditions without incurring packet loss due to the overhead of pipeline changes. Second, the system must be able to monitor the existing processing pipeline and determine whether it is best suited for the ongoing conditions. Finally, the system must support running multiple heterogeneous ML tasks (each with different accuracy-cost tradeoffs) in parallel and balance model accuracy and efficiency across them to ensure effective traffic characterization under dynamic network conditions.

We address these challenges through the following contributions: first, `Cruise Control` minimizes the burden of model deployment by solely requiring a user to provide a set of models with different accuracy-cost tradeoffs. Second, `Cruise Control` implements a selection pipeline that enables parallel feature collection for multiple models, minimizing overhead. Third, `Cruise Control` monitors lightweight signals (i. e., packet loss) to evaluate its current processing capacity and dynamically selects models based on the system's current capabilities and the observed traffic. This approach allows the system to adapt to changing conditions, ensuring that the selected model remains the best-fit under varying traffic loads.

We implement `Cruise Control` and evaluate its performance for two real world traffic analysis tasks: video quality inference and traffic classification. Our results show that `Cruise Control` effectively adapts to changing network conditions, selecting the most appropriate model for the observed traffic and system capabilities. Compared to existing approaches that aim to select an optimal candidate model for each task based on offline information, `Cruise Control` reduces packet loss by a factor of four while achieving equivalent or better median accuracy. Further, `Cruise Control` supports the parallel execution of multiple network analysis tasks, allowing for efficient resource allocation across different analytical functions without sacrificing performance. This parallelization ensures ease of use and reduces operational complexity, allowing operators to perform complex tasks effortlessly, avoiding the need for multiple processing servers.

4.2 **Background and Motivation**

Network operators rely on the ability to reason about their networks, from understanding the traffic that traverses them to whether they are functioning correctly. However, answering such questions is increasingly difficult due to multiple factors, including traffic volume (i. e., relentlessly increasing network speeds)[147] and opacity (i. e., widespread adoption of encryption)[4, 101]. As such, conventional monitoring approaches are becoming inadequate [101, 147, 148] as they struggle to cope with modern Internet traffic characteristics.

To address these challenges, network operators have turned to ML-based solutions [1, 12, 122] for various network monitoring tasks, from traffic classification to quality of service inference [94]. For example, while encryption makes direct measurement of application layer performance such as video streaming quality metrics impossible, ML models are able to accurately infer these metrics from encrypted network flows, providing crucial insights into user experience [14, 55, 86, 90, 95]. Unfortunately, ML-based approaches that use static configurations fall short as they fail to capture the inherent variability of network traffic. In this section we discuss current approaches to network monitoring using ML and identify challenges that operational deployments currently face, particularly in the face of packet loss.

4.2.1 ML-*Based Traffic Analysis*

The typical ML-based traffic analysis pipeline follows a structured approach: First, raw network traffic is captured and undergoes preliminary processing, including operations such as header parsing, flow tracking, and data reassembly. The second stage focuses on feature extraction, where statistical computations and information encoding prepare

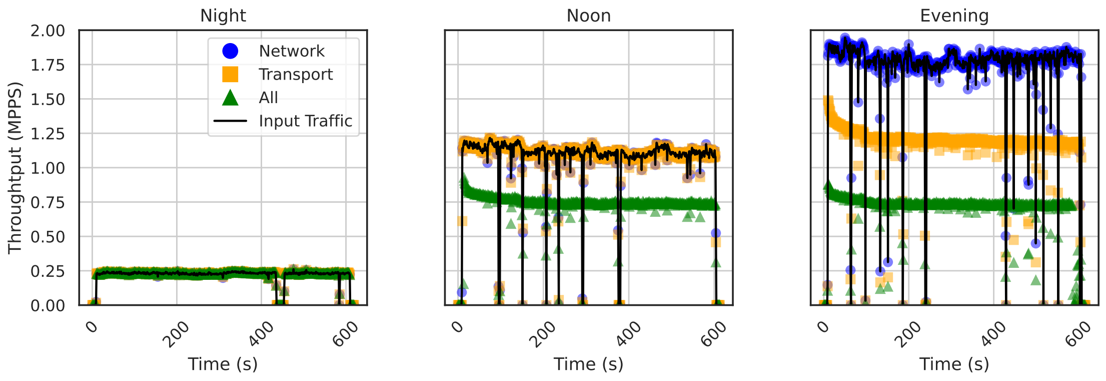


Figure 4.1: Comparison of the impact of three different video quality inference models across different times of day.

the data for model input. Finally, the processed features are fed into an ML model to perform the target inference task. Traditionally, the pipeline is evaluated based on inference performance (e. g., accuracy or F1 score). However, ML network deployments must operate in real-time and are thus subject to systems-related constraints such as the ability to extract packets from the network, compute features, and make inferences at line rates. Failing to do so can lead to packet loss, which compromises model performance [8].

A naïve solution is to create and deploy models using features with low computational complexity, leaving the system with processing headroom to accommodate unexpected traffic spikes [109]. However, this approach can impose an unnecessary ceiling on model accuracy [15] (e. g., a model with higher accuracy could be deployed when there is less traffic, and a model with lower accuracy when there is more).

To address this challenge, recent works [8, 15, 68, 104, 135] have proposed approaches that holistically consider both model accuracy and system performance. For example, CATO [135] highlighted that the choice of which features to compute—potentially even more than the selection of the model itself—can significantly affect a measurement system's ability to gather the necessary information for effective model execution. Consequently, CATO proposed a method to automatically generate Pareto-optimal configurations that maximize accuracy while minimizing system resource usage for a given network environment. However, these configurations, while statically optimal, only represent a single network environment. Further, network traffic loads are inherently dynamic, leading to significant load variance over time. The outcome of these factors is that models that perform well in offline training and testing environments may become unusable due to packet loss in real-world deployments.

4.2.2 *Downsides of Static Model Selection*

To illustrate the inefficiencies that can manifest using static model deployment, we perform a small experiment using a typical ML-based traffic analysis task: video resolution inference from encrypted traffic [14, 86, 88, 117]. We base our experiment on models developed in previous work by Bronzino *et al.* [14], where the authors evaluate several feature sets and models to infer video quality. These feature sets map to different layers of the network stack including: *Network*: basic network flow features (e. g., throughput in/out, packet counts in/out), *Transport*: end-to-end latency and packet retransmission information (e. g., RTT, retransmission in/out), and *All*: combined features from network, transport, and application layers (e. g., video segment sizes, time between video segments). These different feature sets result in three models with varying accuracy for the same task.

We evaluate the ability of an ML-based measurement system to process traffic for these three models. As in the rest of the chapter, we measure the ability of an ML system to support a given model by analyzing whether the system can successfully compute the features necessary for the model execution (i. e., the feature set) without packet loss. For this experiment, we implement the different feature sets using Retina [134], a state-of-the-art feature extraction system and the same system that CATO uses for finding Pareto-optimal configurations. Retina enables users to efficiently compute features for subsets of parsed flows. However, changing the feature set in Retina requires a full system reload, a process that can take several seconds. We deploy the system on a server equipped with a 100 Gbps network interface and evaluate its performance using real-world traffic traces. Specifically, we use three 10-minute traces derived from a one-hour trace collected at Equinix Chicago [20]. These traces are scaled to represent different traffic regimes throughout a typical day, using ratios inspired by Feldmann et al.[44]. We use TRex[131] to adjust the traces to reflect night ($\times 0.2$), noon ($\times 1.0$), and evening ($\times 1.6$) traffic volumes. This is achieved by modifying the inter-packet time in the original trace, thereby scaling the number of packets and flows. Additional details about the testbed setup are provided in Section 4.5.

Figure 4.1 shows the throughput for three periods of the day in packets processed per second by the system. Throughputs for the feature sets are represented by blue circles, orange squares, and green triangles respectively, while the black line represents the incoming load. Packet loss can inferred from the difference between the input traffic and throughput. We observe for the Night scenario, all throughputs align with the input traffic load, indicating successful processing without packet loss for all feature sets. Conversely, for Noon, we observe that the *All* feature set results in loss, while the other two are able to process traffic without loss. Finally, the figure shows that in the evening the system can process traffic without loss only for the *Network* feature set, the set that is least computationally expensive (and least accurate).

p_1	$1 - p_2$	**Network (MAE)**	**Transport (MAE)**
0.0	0.0	821.8	560.6
0.005	0.1	1084.9	667.5
	0.01	1334.5	1429.9
	0.001	1785.4	6279.5
0.01	0.1	1301.4	714.2
	0.01	2322.9	2996.1
	0.001	3596.8	10454.0

Table 4.1: Impact of bursty losses on video startup time inference as Median Absolute Error (MAE) in ms.

4.2.3 *The Accuracy Costs of Packet Loss*

The results of the experiment show that static model selection introduces inherent compromises in system performance. Two solutions to this tradeoff exist: either the operator selects a model that is guaranteed to work under all conditions, or the operator select a model that is guaranteed to work under most conditions, coping with possible packet loss. However, the former approach leads to suboptimal model performance during normal operations, while the latter approach can lead to data loss during peak periods.

To illustrate this tradeoff, we reproduce the experiment from Chapter 3. Specifically, we consider one of the two inference tasks presented by Bronzino *et al.* [14]: video startup time inference. For this task, the first 10 seconds of features collected are used to infer the delay between the start of the session and instant the video player actually starts playing the video. We train three models (corresponding to the previous feature sets) using the dataset presented in the paper. However, we test it both against an unaltered test set, as well as against test sets where network traffic has been modified via the introduction of packet loss. In particular, we introduce bursty loss which is typical for network systems. Our bursty loss model is based on a simple Markovian model that uses two loss states where p_1 represents the probability of transitioning from a no-loss state to a state of bursty loss and p_2 denotes the probability of remaining in the loss state. Results are shown in Table 4.1.

Two main observations can be made from the table. First, the *Network* model show, under no losses conditions, a 146.59% higher median absolute error compared to a model using *Transport* layer features (we omit the *All* model as presented trends are just worsened), rendering it sub-optimal. Second, bursty loss impacts faster the performance of the *Transport* model. While recent work [8] has shown that loss can negatively impact

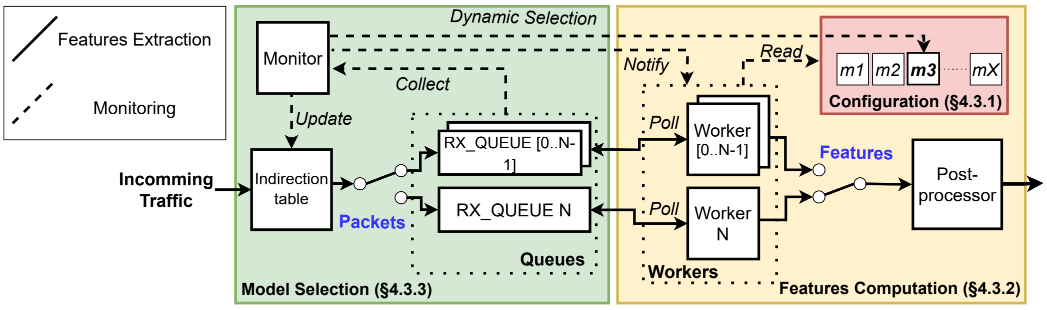


Figure 4.2: `Cruise Control` system overview.

other tasks that use single packets for inference, this confirms that even models that use statistics over multiple packet aggregates can be affected in the presence of bursty loss. Overall, this experiment shows the inherent inefficiencies in model accuracy and system performance due to static model selection and the need to avoid packet losses. We conclude that, to achieve efficiency gains in model accuracy and system performance, a dynamic system adaptation to network load is required.

4.3 Cruise Control

In this section, we present `Cruise Control`, a system for self-adaptive model deployment for ML-based network traffic analysis. `Cruise Control` is built on two key design principles. First, it aims to simplify deployment by minimizing the overhead of offline configuration and model selection. Second, it dynamically selects the best-fit model based on current system capacity and traffic conditions, maximizing the accuracy of analytical tasks while minimizing packet loss.

To realize these principles, `Cruise Control` implements three main design choices, shown in Figure 4.2:❶ It simplifies deployment by requiring only a ranked list of models per task, eliminating complex setup overheads (4.3.1). ❷ It achieves line-rate feature extraction and seamless model switching through a coordinated architecture of workers and a post-processor core; additionally, it minimizes multi-task processing overhead by merging common features and eliminating redundancy (4.3.2). ❸ It dynamically selects the most appropriate model in real-time using lightweight monitoring signals, adapting to changing network conditions (4.3.3). The rest of this section details the modules implementing these design principles, as shown in Figure 4.2.

4.3.1 *System Configuration*

Supporting dynamic model switching for traffic analysis requires understanding the trade-offs between feature collection costs, model accuracy, and system capacity under varying network conditions. `Cruise Control` minimizes deployment hurdles by elim-

Model #	Cost	Accuracy
m_9	2272	0.935
m_8	1696	0.934
m_7	1248	0.933
m_6	960	0.932
m_5	736	0.931
m_4	704	0.926
m_3	480	0.924
m_2	320	0.900
m_1	256	0.799

Table 4.2: Video quality inference models.

inating the need for manual fine-tuning of candidate models for specific workloads, requiring users to solely provide a ranked list of models and their associated feature extraction sets as input. While selecting optimal models without prior knowledge of the deployment environment is challenging, recent research [15, 135] has developed methods to quantify the accuracy-cost tradeoff. We leverage CATO [135] to build offline input configurations which, when coupled with `Cruise Control`'s dynamic model switching capabilities, significantly reduces deployment overhead.

Table 4.2 shows the resulting configuration used as system input. It consists of models labeled m_X, where X represents the accuracy-ordered index (m_1 being the least accurate and least computationally expensive). To create this configuration, we applied CATO's methodology to identify the Pareto front of candidate models illustrated in Figure 4.3. We started with 10 unitary features from Bronzino *et al.* [14], representing data collected across three protocol stack layers. For clarity and conciseness, we aggregate these into a reduced set of representative features in our analysis. A comprehensive enumeration of the original features can be found in Bronzino *et al.* [14], and we therefore do not reproduce it here. While the original paper contains more features, we aggregated those computed from the same information (e. g., different statistical representations of the same feature). Through CATO's approach, we reduced the possible feature combinations from 1023 to just nine optimal configurations, with those closer to the top left of the Pareto front being preferred. Although `Cruise Control` leverages CATO's efficient computation method, it is not tied to this specific algorithm—alternatives could be used, requiring only that feature sets be ordered by accuracy and cost.

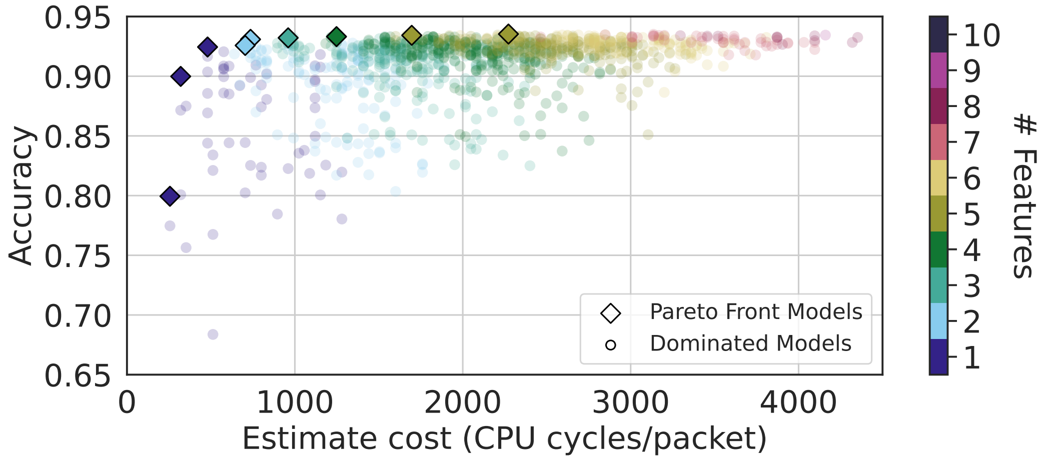


Figure 4.3: Video quality inference Pareto front.

4.3.2 *Dynamic Feature Computation*

The online phase of `Cruise Control` consists of two key components, illustrated in Figure 4.2: the *Feature Computation* module and *Model Selection* module. The Feature Computation module handles packet processing, connection reassembly, feature extraction, and the computation of statistical features required for ML model execution. The Model Selection module gathers metrics to monitor system performance and determines the set of features to be collected for the target model. We describe the Model Selection module in Section 4.3.3.

PACKET PROCESSING PARALLELIZATION. Once incoming packets are received by the NIC, they are processed by the Feature Computation module, which extracts the features required by the selected model, aggregates them, and prepares them for consumption by the model. The Feature Computation module consists of two primary components: Workers and a Post-Processor. The Workers handle feature extraction directly from incoming packets based on the feature set selected by the Model Selection module. Periodically, the Post-Processor aggregates these features and exports them for use by the model. Workers operate on dedicated CPU cores and perform feature extraction tasks, including packet filtering, pre-processing (e. g., header parsing), and basic feature computations (e. g., calculating flow throughput or packet counters). `Cruise Control` leverages multi-core architectures to parallelize packet processing across two dimensions. First, each Worker processes a subset of the traffic. To account for features that depend on entire connections rather than individual packets, all packets belonging to a given network flow are processed by the same Worker. This is done via Receive Side Scaling (RSS). Second, a backup Worker is instantiated to handle load while one of the Workers exports features to the Post-Processor.

The mapping between connections and Workers is managed through an indirection table (illustrated in Figure 4.2), which routes packets to the appropriate Worker. Each Worker maintains a dedicated hashmap to store the necessary data for monitored network flows. Each hashmap entry contains the set of features computed for a specific connection, along with the required flow state information.

When a new packet enters the pipeline, it is directed to the corresponding Worker. If the packet belongs to a new flow, the Worker first determines which features need to be computed. It consults the shared configuration, which is continuously updated by the Model Selection module (see Figure 4.2). This configuration dynamically specifies the required features. Based on this information, the Worker initializes and allocates the necessary data structures, which are then cached for future packets from the same connection. For packets from known connections, the Worker accesses the pre-existing structures associated with the connection, updates relevant counters, extracts header information, and stores the data required for the previously requested features. Since the selected features are determined during the processing of the connection's first packet, the Worker does not need to repeatedly access the configuration. This design choice ensures consistency in feature computation for the duration of a network flow. However, this approach comes with a trade-off: it limits the system's ability to quickly adapt to changes in the selected features. Nevertheless, this trade-off is necessary to maintain consistency, as certain features rely on historical data to be computed (e. g., packet interarrival time distributions or video segment size distributions).

MULTI-TASK SUPPORT. Real-world scenarios typically require simultaneous execution of multiple traffic analysis tasks [133]. Dedicating separate servers to each task is both inefficient and costly. `Cruise Control` addresses this challenge by efficiently handling feature extraction for multiple ML tasks concurrently while maintaining its key principles. When a new connection appears, `Cruise Control` combines features set from each task to avoid duplicate feature extraction and leverage intersection between required features. This combination is enabled by designing required features as composite sets of atomic features. Internally, each feature set receives a unique identifier that can be decomposed into an N-bit representation, with each bit corresponding to an atomic feature. During processing, bits with value 1 trigger extraction of corresponding features, while bits with value 0 are skipped. This approach allows straightforward combination of feature sets from multiple tasks using a bitwise OR operation, producing the final extraction set for incoming connections without additional overhead or packet duplication. Additionally, since tasks may require different packet quantities, the system extracts the higher number of packets requested.

FEATURE EXPORT. Periodically, `Cruise Control` exports the computed features from the Workers to the Post-Processor. The Post-Processor, itself running on a dedicated core, collects the features extracted by the Workers, computes statistical features, and formats the output for the machine learning model. The interval between two exports is referred to as the *export_window*. The act of exporting collected features is often overlooked in existing work that emphasizes feature extraction [68, 135]. However, this operation can be resource-intensive, incurring significant CPU overhead and memory transfers, yet it is essential for passing data to the ML model.

To perform the export, incoming traffic processing on a Worker must be temporarily halted to clear the hashmap storing the features. During this time, Workers are unable to process incoming traffic, potentially leading to significant packet drops if precautions are not taken. In our experiments, we observed that this export process can take several seconds to complete. To mitigate this, we exploit concurrent redundancy to implement a solution in `Cruise Control`. When `Cruise Control` starts, a backup Worker is created, as shown in Figure 4.2. Then when *export_window* is reached, the monitor swaps one of the current Workers and the backup Worker in the indirection table. This will cause traffic to be redirected to the new Worker, then the Monitor will notify the selected exporting Worker that it has been swapped. It can then export its data to the Post-Processor without packet loss.

Once the export is complete, the Worker is designated as a backup, ready to handle traffic during the next export cycle. At each *export_window*, the Worker clears its hashmap and sends the collected data to the Post-Processor. The *export_window* interval depends on the specific use case and the hardware capabilities, particularly the available memory. A shorter *export_window* requires less memory but introduces significant overhead. Conversely, a longer *export_window* reduces overhead but risks memory saturation and may be unsuitable if features must be delivered to ML models at shorter intervals.

4.3.3 *Adaptive Model Selection*

The Model Selection module takes the pool of feature sets listed in the input configuration and determines which model to use based on the ongoing load experienced by the system. Many challenges lie behind this task. First, monitoring the system's state cannot rely on heavyweight profiling that might itself cause increased system load and lead to packet loss. Second, the system must implement an intelligent algorithm capable of both detecting whether the system is overloaded, thus triggering selection of a more lightweight feature set, as well as whether the system has resources to spare, thus triggering a switch to a more complex feature set. We solve the first challenge by relying on a lightweight signal, i. e., packet loss, to monitor the system's state. This is done by a dedicated Monitor that tracks the state of the `rx_queues` used to receive

Algorithm 1 Feature set selection algorithm

```
1: mon_window: Time window for monitoring metrics
2: m_i : Index of the current candidate features set
3: n_drops: Dropped packets since the last cycle
4: dec_factor: Decrease factor used when a drop occurs
5:
6: loop
7:     if $n\_drops > 0$ then
8:         $m_i \leftarrow \lfloor m_i * dec\_factor \rfloor$
9:     else
10:        if $t\_since\_last\_update \geq mon\_window$ then
11:            $m_i \leftarrow m_i + 1$
12:            $t\_since\_last\_update \leftarrow 0$
13:        end if
14:    end if
15: end loop
```

packets from the NIC. We monitor the `rx_miss` counter, which counts the number of packets dropped by the hardware due to the queues being full.

METHODOLOGY. We tackle the second challenge by designing an algorithm that leverages detected losses—or their absence—to determine whether to switch between models. We propose the algorithm outlined in Algorithm 1 to select a feature set based on the system's state. Note that `Cruise Control` is not tied to this specific algorithm, and alternatives could be employed. The candidate feature sets are indexed in increasing order of complexity, with the currently selected set represented by the variable m_i. To prevent system saturation, the algorithm continuously monitors for packet loss (line 7). When loss is detected, m_i is adjusted using a multiplicative decrease (dec_factor), selecting a feature set with lower CPU cost. Conversely, the algorithm additively increases m_i to explore more complex feature sets, doing so every mon_window seconds (line 10). This Additive Increase/Multiplicative Decrease (AIMD) strategy is analogous to congestion control mechanisms used in TCP algorithms that aggressively downgrade transmission rate upon detecting a packet loss. Similarly, `Cruise Control` aggressively downgrades to a simpler model when a loss is detected and cautiously explores more complex models when no losses occur. The intuition behind this approach is that a production deployment of `Cruise Control` could likely run multiple models in parallel, each with different accuracy and CPU cost. As such, we seek convergence to a balance of ideal feature sets that do not favor one model over another. We evaluate our approach and compare it with alternatives in Section 4.5.4.

For `Cruise Control` to work effectively, the mon_window and dec_factor parameters must be carefully calibrated based on model accuracy requirements and specific use cases. A higher mon_window value creates a more stable system by causing more

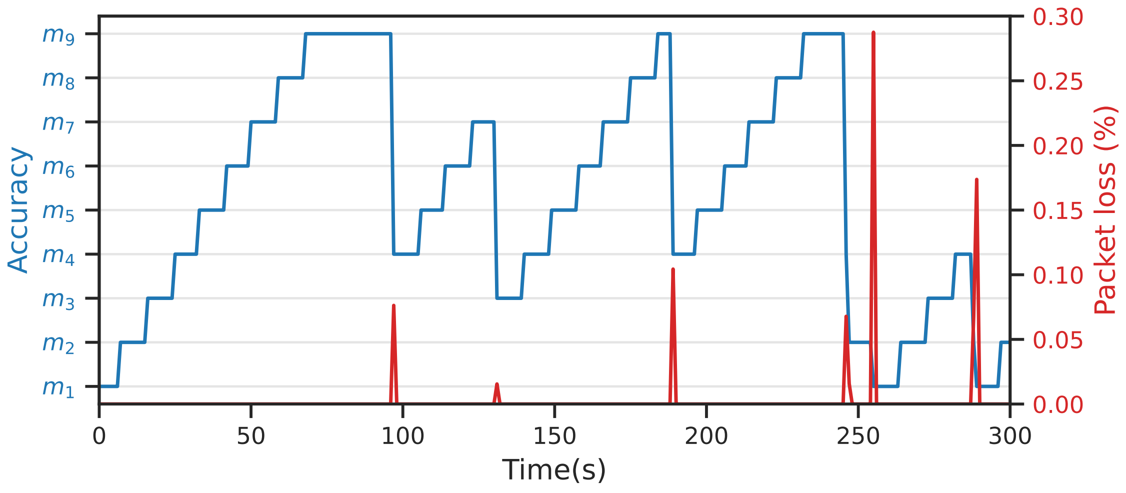


Figure 4.4: Timeseries of `Cruise Control` model selection algorithm. The blue line represents the selected feature set, while the red line shows the number of dropped packets over time.

gradual transitions to complex feature sets. Conversely, a lower *mon_window* enables quicker feature set upgrades but increases the risk of system saturation and packet loss. The *dec_factor* determines how many feature sets are skipped when drops are detected. A higher *dec_factor* reduces the likelihood of subsequent losses by selecting significantly simpler feature sets, though potentially sacrificing accuracy. After each change, the monitor updates the Workers' feature sets as shown in Figure 4.2. Finally, note that for supporting multiple parallel tasks, the algorithm requires adaptation to accommodate multiple *mon_window* values. This modification updates line 11 in Algorithm 1 by implementing a loop to test each *mon_window* rather than just one. However, when drops are detected (line 7), the model complexity (m) must be reduced for all tasks simultaneously.

EXAMPLE. To demonstrate the behavior of this algorithm in a practical scenario, we run `Cruise Control` using a sample traffic trace: the first five minutes of the CAIDA dataset previously utilized [20]. The results are presented in Figure 4.4. Each horizontal line in the figure represents a feature set shown in Table 4.2. The accuracy of each corresponding model is shown on the left y-axis, with candidate feature sets spanning accuracy values from 0.7994 (m_1) to 0.9353 (m_9). The blue line illustrates the feature set selected by `Cruise Control` over time, while the red line (right y-axis) represents the percent of dropped packets observed every second.

As shown, we observe an initial phase where `Cruise Control` iteratively increases the complexity of the computed feature sets, transitioning to more accurate models. 98 seconds into the experiment the system detects packet losses, prompting the algorithm to rapidly downgrade to a less complex feature set. In this example, with dec_factor set to 0.5, the algorithm switches from feature set m_9 to m_4. When a second drop occurs at 132 seconds, the system further reduces complexity, falling back to m_3. An interesting behavior appears at 247 seconds where a transition to a less complex feature set occurs (m_9 to m_4), but the system continues experiencing packet drops, immediately triggering a second transition down to m_2. Finally, at 256 seconds, loss triggers a drop down to m_1. It is important to note that these changes occur immediately upon detection of packet drops by the monitor, rather than at each mon_window interval. The mon_window value is used exclusively for incrementing to more complex feature sets.

4.4 Prototype Implementation

In this section, we present the proof-of-concept implementation of `Cruise Control`, as well as the two use case traffic analysis tasks currently implemented in the system release.

4.4.1 *Software prototype*

We implement `Cruise Control` in Rust to leverage its features for ensuring memory and thread safety while utilizing modern packet processing frameworks to maximize packet processing efficiency. To handle high speed traffic, `Cruise Control` bypasses the kernel to avoid bottlenecks caused by the network stack [19, 138] using the Intel DPDK [29]. DPDK allows NIC to offload packets to a dedicated CPU core's memory space without interruption by using Direct Memory Access (DMA). Further, DPDK implements Receive Side Scaling (RSS) to distribute the connection across multiple CPU cores. We leverage this capability to share the connections between multiple CPU cores and to swap traffic from one core to another when a worker needs to export its hashmap to the post-processor.

Our prototype builds on a customized version of Retina[1], extending its monitoring core to support more complex tasks such as dynamic worker reconfiguration and periodic feature export via updates to the Indirection Table. We leverage Retina[134] primarily for its convenient and extensible Rust Application Programming Interface (API) for DPDK. However, we make minimal use of Retina's higher-level features—relying mainly on basic packet filtering and the DPDK runtime environment it provides. As a result, our system interacts closely with core DPDK functionalities via Retina, which we believe could make it straightforward to adapt to other DPDK-based environments. We implement a worker that extracts monitor-specified features at line rate and exports them periodically, and a post-processor that compute and formats these features for machine learning models. The use of a dynamic trait ensures extensibility and provides a

1 Retina v0.1 from the original chapter

Model #	Cost	Accuracy
m_3	1184	0.970
m_2	1056	0.898
m_1	704	0.824

Table 4.3: Service recognition models.

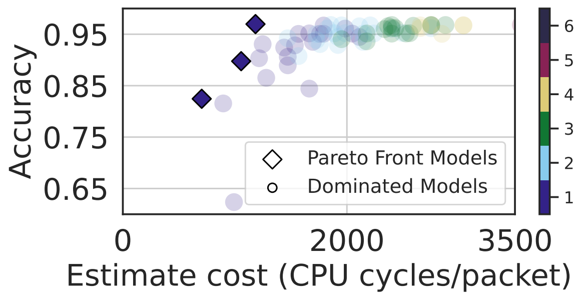


Figure 4.5: Service recognition Pareto front.

shared interface that enables seamless coordination between the worker, post-processor, and the feature set. In total, our prototype implementation was ~3300 lines of Rust code.

4.4.2 *Use Cases*

We implement two typical traffic analysis tasks, presented in Chapter 2: video streaming quality inference from encrypted traffic and service recognition.

VIDEO QUALITY INFERENCE. The first use case for `Cruise Control` focuses on inferring streaming video quality. We implement the feature sets from Bronzino *et al.* [14], prioritizing those identified by the Pareto front analysis in Section 4.3.1. These include basic network flow metrics (throughput in/out, packet counts), end-to-end performance indicators (RTT, retransmission statistics), and application-layer information derived from traffic patterns (video segment sizes, inter-segment timing). Based on our ranked list, we configure `Cruise Control` to operate with nine target models optimized for this use case. This use case is particularly well-suited for our system, as it requires continuous feature extraction throughout a flow's entire lifetime, making performance heavily dependent on the total number of packets processed.

SERVICE RECOGNITION. For the second use case, we focus on service recognition, one of the most studied traffic analysis tasks [4, 142]. We focus on early flow detection, now a standard identification approach [81, 104, 116], utilizing just the first 10 packets of each connection. Unlike the video quality inference use case, this task's collection cost is primarily determined by the number of flows rather than packets, with the first packet carrying particular significance due to memory structure initialization costs.

We apply the same Pareto front methodology as in the previous use case. We utilize six features from the video quality analysis, excluding the four video segment identification features. Additionally, we consider raw headers (IP/TCP) as a feature—a representation recently combined with deep learning models for more accurate traffic identification [58, 116]. Raw headers require minimal computational complexity since no calculations are needed for extraction, though they are more memory-intensive. This memory cost is not reflected in our current CPU cycle-based metric[2]. Consequently, raw headers offer high precision at low measured cost, causing the Pareto front to be composed by only three target models. The resulting Pareto front appears in Figure 4.5 and Table 4.3, with three feature sets selected from the original 63. The raw headers feature set is represented as model m_3.

4.5 Evaluation

We evaluate `Cruise Control`'s performance using several scenarios that allow us to compare its performance in terms of accuracy and loss rates versus static models, as well as to illustrate the effects of tunable parameters in `Cruise Control`. In all experiments, `Cruise Control` uses two Workers with only one active at a time, as presented in Section 4.3, except in the multicore experiments where multiple Workers run concurrently. This setup simplifies the following analysis.

HARDWARE ENVIRONMENT. Our testbed consists of two identical servers, each equipped with dual 16-core AMD EPYC 7343 processors and a 100GbE Intel E810 NIC with 384GiB of DDR4 split accros two NUMA. Both servers are connected to a shared 100GbE switch. One server is dedicated to traffic generation using Cisco's TRex [131], which replays real network traffic from capture files. This traffic is sent to the switch, where it is mirrored to the second server running `Cruise Control`. The testbed simulates a realistic environment in which a network operator monitors traffic on a specific span port. It is worth noting that the use of AMD CPUs precludes leveraging Intel's Data Direct I/O Technology (DDIO), which facilitates direct transfers from the NIC to the CPU cache.

NETWORK TRAFFIC. To mimic realistic traffic conditions, we use a one-hour trace collected at Equinix Chicago in 2016 [20]. The selected trace is a traffic capture available upon request from the CAIDA's website (20160121). Throughout the trace, the throughput remains relatively constant, with an average rate of approximately 10 Gbps. The trace's rate fluctuates between 1 MPPS and 1.2 MPPS. To simulate different traffic rates in our experiments, we adjust the inter-packet time using TRex. As with the experiment discussed in Section 4.2, we use three different traffic profiles: noon, evening, and night. The noon profile is used as the base profile, with the evening profile scaled by a factor of 1.6 and the night profile scaled by a factor of 0.2. Note that the trace contains intermittent gaps of approximately one second. Although the exact origin of these gaps is unknown, they may be caused by artifacts introduced during data capture or

2 A cost metric based on memory usage could also be used. We leave this for future exploration.

anonymization. They were retained in our dataset. This decision was made to reflect realistic and potentially adverse conditions, under which `Cruise Control` continues to perform robustly.

EVALUATION OBJECTIVES. We design our experiments to answer the following questions:

❶ *How does `Cruise Control` perform under varying workloads?* We demonstrate that under realistic traffic conditions typical of production deployments, `Cruise Control` improves median accuracy by 2.78% while reducing packet loss by a factor of four compared to statically-selected models, the current state-of-the-art approach (4.5.1).

❷ *What is `Cruise Control`'s overhead?* We show that `Cruise Control` outperforms static configurations for both long-duration experiments and short experiments that do not require feature exports (4.5.2).

❸ *How scalable is `Cruise Control`?* We demonstrate that `Cruise Control` effectively scales on multi-core architectures without incurring excessive synchronization overhead (4.5.3).

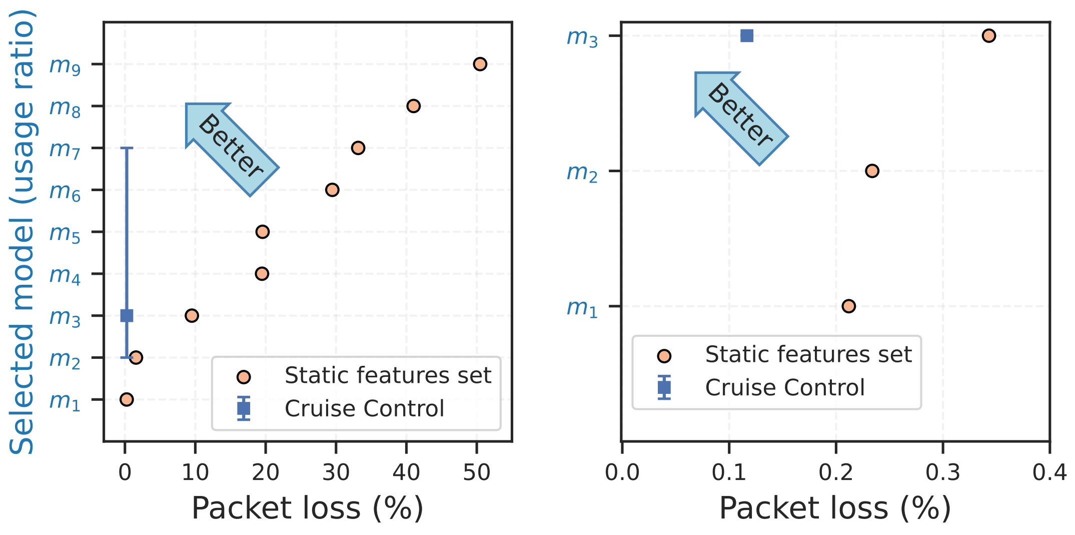
❹ *Does `Cruise Control` ease the burden of parallel feature collection?* We show that `Cruise Control` can be easily configured with multiple parallel analysis tasks without significantly affecting model accuracy or packet loss (4.5.4).

❺ *How do tunable parameters affect `Cruise Control`'s performance?* We evaluate `Cruise Control`'s performance under different *mon_window* values, selecting eight seconds as the optimal value for all other experiments based on our findings (4.5.5).

4.5.1 *Performance Under Varying Workloads*

We evaluate `Cruise Control`'s ability to adapt to evolving traffic patterns typical in real-world network deployments. Using three distinct traffic profiles (noon, evening, and night) described earlier, we implement abrupt transitions between profiles to assess the system's responsiveness to sudden load changes. We compare `Cruise Control` against state-of-the-art systems like Retina [134] and CATO [135], which require pre-selecting features before deployment and need substantial downtime to reconfigure. For our experiments, we use a modified version of Retina with feature export capabilities—a necessary adaptation to prevent memory exhaustion during extended tests (detailed in Section 4.5.2). We test Retina using all optimal configurations identified from our Pareto front analysis and label results according to the model used. We perform experiments using a single Worker core, as our goal is to compare `Cruise Control`'s performance against static configurations. We evaluate multi-core scalability in Section 4.5.3.

`CRUISE CONTROL` IMPROVES BOTH ACCURACY AND PACKET LOSS. Figure 4.6a illustrates the performance comparison, plotting selected models against packet loss



(a) Video quality inference

(b) Service Recognition

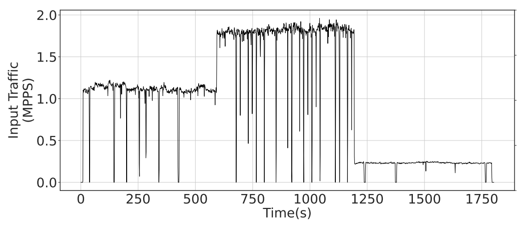
Figure 4.6: Different time of day workload

rates (x-axis). Orange circles represent static configurations (Retina), while the blue square denotes `Cruise Control`. Note that, since CAIDA traffic is unlabeled, we cannot directly evaluate ML model performance—which is beyond the scope of this chapter. For `Cruise Control`, which dynamically employs multiple feature sets, we show the usage ratios of different models (ordered by their offline performance as shown in Section 4.3.1). In particular, we show the median model selected along with the first and third quartiles. The optimal would appear in the top-left corner, representing the configuration with the highest accuracy and zero packet loss.

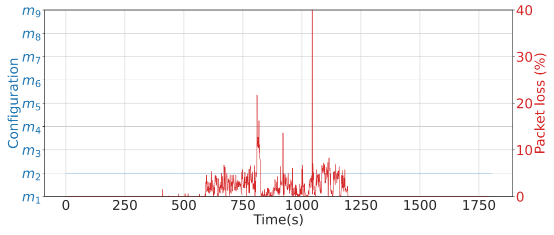
For the video quality inference use case (Figure 4.6a), all static feature sets beyond m_2 incur excessive packet loss rates (>9%), making them impractical for deployment. In contrast, `Cruise Control` achieves just 0.264% loss hile operating on m_2 or better models 75% of the time. Even m_2 itself, which might be selected in static deployments to avoid unacceptable losses, performs worse than `Cruise Control`, with a higher packet loss rate of 1.57%.

For the service recognition use case (Figure 4.6b), packet loss rates are generally lower than in the first use case, due to the reduced analysis complexity (processing only the first 10 packets per connection). Nevertheless, `Cruise Control` consistently outperforms static configurations, remaining on top-performing models 80% of the time while also delivering significant reductions in packet loss.

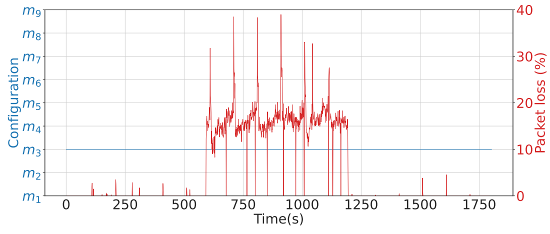
`cruise control` SELF-ADAPTS TO SUDDEN WORKLOAD CHANGES TO PREVENT PACKET LOSS. We further investigate these results by examining the causes of



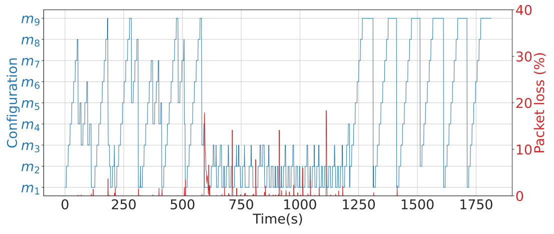
(a) Injected traffic



(b) m_2



(c) m_3



(d) `Cruise Control`

Figure 4.7: Timeseries for static feature sets and `Cruise Control` for video quality inference

packet loss, using the first use case as a representative example. Figure 4.7 presents a time series of the traffic profile (Figure 4.7a) alongside system behavior data for both `Cruise Control` (Figure 4.7d) and two representative static configurations (Figures 4.7b and 4.7c). These graphs display model accuracy (blue/left y-axis) and packet losses (red/right y-axis) throughout the experiment. With the m_3 static feature set, minimal drops occur during the 'noon' traffic profile (0-600 seconds), primarily coinciding with export events. During the 'evening' profile (600-1200 seconds), the system cannot compute m_3 for all packets, resulting in significant losses. The 'night' profile (final 10 minutes) processes without loss. We plot the average accuracies of m_2 and m_3 as horizontal blue lines at 0.899 and 0.9245, respectively. However, static configuration accuracy becomes unpredictable during packet loss due to randomness, particularly evident during peak traffic (600-1200 seconds) where m_2 experiences approximately 5% loss and m_3 reaches 15%.

In contrast, `Cruise Control` adapts dynamically, as shown in Figure 4.7d. During the first 10 minutes, it progresses through the Pareto front models to reach the most accurate ones, stepping down to less complex models when losses occur. At the 600-second mark, when traffic increases, a burst of packet loss triggers `Cruise Control` to adjust to the least accurate (but least costly) model m_1, which successfully processes traffic without loss, thus performing even better than m_2. When traffic decreases at 1200 seconds, `Cruise Control` reverts to the most accurate model. This experiment demonstrates `Cruise Control`'s ability to adapt to changing loads while minimizing losses compared to static configurations. Though `Cruise Control` introduces some accuracy variability, it effectively prevents and responds to packet loss while achieving higher accuracy during periods of excess processing capacity.

Video quality inference						
Dataset		m_3	m_4	m_5	m_6	`Cruise Control`
CAIDA	**Pkt loss (%)**	0.047	0.357	0.490	13.041	**0.033**
	Accuracy	0.924	0.926	0.931	0.932	**0.931**
No Export	**Pkt loss (%)**	0	0.064	0.062	11.393	**0.043**
	Accuracy	0.924	0.926	0.931	0.932	**0.931**

Service recognition					
Dataset		m_1	m_2	m_3	`Cruise Control`
CAIDA	**Packet loss (%)**	0.212	0.234	0.343	**0.117**
	Accuracy	0.824	0.898	0.970	**0.970**
No Export	**Packet loss (%)**	0	0.001	0.013	**0**
	Accuracy	0.824	0.898	0.970	**0.970**

Table 4.4: Performance comparison between static model and `Cruise Control` for video quality inference and service recognition

4.5.2 *System Overhead*

Our previous experiment demonstrates `Cruise Control`'s advantages under rapidly changing traffic conditions. Here, we show that `Cruise Control` also outperforms static configurations for both longer, steadier workloads and shorter workloads that do not require feature exports.

LONG WORKLOADS. We evaluate our system using a complete 1-hour CAIDA trace with a steady traffic rate of approximately 1.1 MPPS (4.2×10^9 packets total). Table 4.4, section video quality inference, CAIDA's dataset shows that `Cruise Control` performs well in this realistic scenario. Overall, it experiences minimal packet loss—lower than the static feature set m_3 (by about 0.05%)—while primarily utilizing m_5, which loses 0.49% of packets. For the second use case (Table 4.4, section Service recognition, CAIDA's dataset), `Cruise Control` primarily uses m_3 while providing almost $3\times$ lower packet loss. As noted earlier, some minimal losses during exports are inevitable—the necessary trade-off for reducing end-to-end latency and enabling longer experiments without memory exhaustion. However, this experiment demonstrates that `Cruise Control` effectively handles longer, steadier workloads while maintaining low packet loss rates and high accuracy.

SHORT WORKLOADS. To verify that our results are not artifacts created by `Cruise Control`'s export mechanism, we conduct a scaled-down experiment limited by available

RAM. This test includes only five minutes of traffic with all features kept in memory, eliminating any overheads related to hashmap transfers to the post-processor. Results appear in Tables 4.4 (No Export dataset). `Cruise Control`'s behavior remains consistent with previous experiments. In this scenario, `Cruise Control` experiences slightly more packet loss (0.043%) than m_3 (0%), but performs better than m_4 and m_5 (0.062% and 0.064% loss, respectively) while achieving median accuracy equal to m_5. This again highlights `Cruise Control`'s advantage over static configurations—significant accuracy improvements with minimal packet loss penalties. For the second use case (Table 4.4), we observe insignificant packet loss across (less than 0.001) all models except m_3, which experiences drops at startup due to processing only the first 10 packets of each flow.

4.5.3 *Multi-core Scalability*

Our previous experiments focused on single-core scenarios to demonstrate `Cruise Control`'s adaptability to changing workloads. We now assess system scalability through a dedicated multi-worker (i. e., multi-core) benchmark. We focus on the worst-case scenario: new connection arrivals. In this test, each packet represents a new connection, forcing the system to perform high-cost operations including configuration checks, data structure creation, and storage in thread-local hash maps. The following results are presented in Millions of connections per second (Mcps).

Table 4.5 shows that the system scales effectively with increasing worker counts. Moving from one to two workers, throughput doubles from 1 to 2 million connections per second (Mcps), indicating near-perfect scaling. At four workers, throughput reaches 2.5 Mcps, indicating diminishing returns relative to ideal linear scaling. This behavior could potentially be attributed to unfairness of RSS[9], we leave exploration of this to future work. Notably, with eight workers, the system reaches 3 Mcps, significantly exceeding real-world traffic levels, such as those observed in CAIDA traces, which report only 23.92 thousand new connections per second. As these tests represent worst-case scenarios where each packet establishes a new connection, we anticipate superior performance in realistic environments.

It is worth noting that experiments at 6 Mcps and 8 Mcps on eight workers, and 8 Mcps on four workers, terminated prematurely due to memory exhaustion before the allocated 10-minute duration. This highlights the importance of careful memory management when utilizing multiple workers. The primary challenge involves thread-local hashmaps—while providing processing advantages through worker-specific storage, improperly dimensioned hashmaps require runtime resizing, resulting in packet processing failures and consequent losses. Additionally, in the experiments referenced in Table 4.5, we apply identical parameters across different worker counts. Since only one worker exports at a time (determined by round-robin rotation), each of the N workers exports after a total duration of $N \times export\_window$. Consequently, with more workers, each remains active longer before exporting, necessitating larger hashmaps and increas-

# Workers Parallel	Throughput [Mcps]								
	0.5	1.0	1.5	2.0	2.5	3.0	4.0	6.0	8.0
1	0.00	**0.00**	0.79	23.87	30.39	40.13	55.73	70.48	77.62
2	0.00	0.00	0.00	0.00	**0.00**	2.90	23.66	56.50	67.24
4	0.00	0.00	0.00	0.00	**0.00**	5.64	17.50	26.74	29.45*
8	0.00	0.00	0.00	0.00	0.00	**0.00**	1.21	0.00*	5.88*

Table 4.5: Median packet loss (%) evaluation on multi-core during 10 minutes experiment. *Memory exhausted

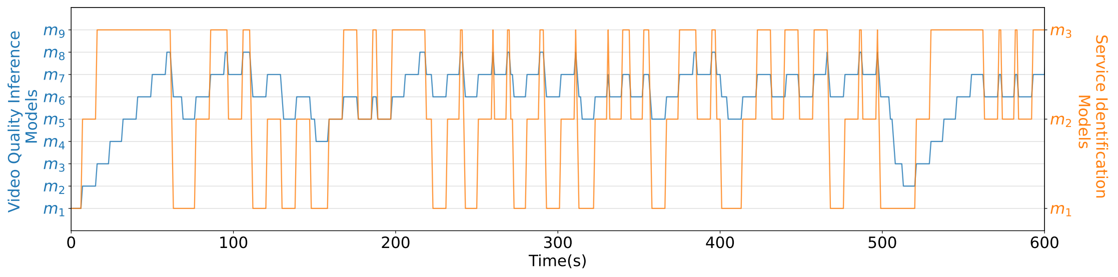
ing the load on the worker-to-post-processor channel. Neglecting this consideration leads to rapid memory exhaustion, while excessive switching increases overhead costs.

4.5.4 *Multitask Support*

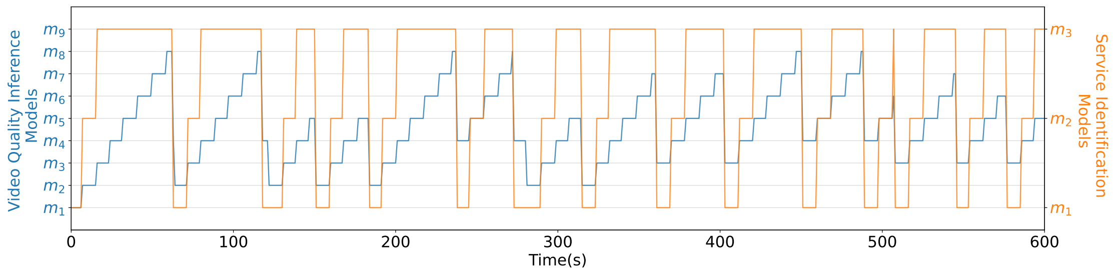
While extracting features for a single ML task demonstrates our system's basic capabilities, `Cruise Control` is designed to support feature extraction for multiple concurrent tasks. In this experiment, we demonstrate this capability by running both use cases in parallel on the first 10 minutes of the previously used CAIDA trace. In the following, $UC1$ stands for video quality inference and $UC2$ for service recognition. Our objectives are twofold: to show that the system maintains high model accuracy and low packet loss while supporting multiple tasks, and to establish that AIMD is the most appropriate control algorithm for this scenario. We accomplish this by conducting four experiments comparing AIMD against an Additive Increase/Additive Decrease (AIAD) approach that decrement by one m_i when drop occurs, unlike AIMD's multiplicative decrease. Note that, although the algorithm supports task-specific mon_window values as described in Section 4.3, we apply the same value to both tasks based on our finding that this configuration delivers optimal performance for both use cases (detailed in the following section).

Table 4.6 shows a summary of the experiment for the four combinations. We observe that using AIAD tends to increase overall accuracy but also results in a higher packet drop rate. Both outcomes can be explained by AIAD's slower reaction to packet drops, causing longer periods of packet loss but smaller decreases in overall performance. The (AIMD, AIAD) combination achieves higher accuracy with only a limited packet drop penalty. This is probably because $UC2$'s Pareto front includes only three models, making each step more impactful. As a result, AIAD appears sufficient. Nevertheless, the (AIMD, AIMD) combination demonstrates the most favorable results in minimizing packet drops.

To better understand the reasons behind these results, we plot in Figure 4.8a and 4.8b the two opposite scenarios, i. e., using AIAD in both versus using AIMD in both.



(a) (AIAD,AIAD)



(b) (AIMD,AIMD)

Figure 4.8: Service recognition features extraction across three different network load

UC1	UC2	Acc UC1	Acc UC2	Total Drop (%)
AIAD	AIAD	0.932	0.824	0.0263
	AIMD	0.932	0.824	0.0251
AIMD	AIAD	0.931	0.970	0.0089
	AIMD	**0.926**	**0.970**	**0.0082**

Table 4.6: Median Accuracy and Total Drop(%) of different combinations of control algorithms

The blue line represent the selected configuration for $UC1$ while the red line the configuration selected for the $UC2$. We add AIAD to the AIMD to test multiple combinations. We observe that (AIAD, AIAD) results in weaker synchronization between the two tasks, with $UC1$ leveraging its broader set of models on the Pareto front to benefit from this imbalance. In contrast, (AIMD, AIMD) shows a more regular sawtooth pattern, where both tasks synchronize more effectively.

Overall, these results confirm the choice of AIMD as the more effective control algorithm for exploration to limit the packet loss, even complex scenarios such as multitasking.

4.5.5 *Sensitivity to Parameters*

Finally, we evaluate how the *mon_window* parameter impacts system performance. This key parameter controls how quickly `Cruise Control` switches to more complex configurations when no packet loss is detected, serving as a performance tuning lever that operators can adjust based on specific deployment goals (e. g., maximizing accuracy or minimizing packet loss). We conduct ten experiments using the 1-hour CAIDA traffic trace. Table 4.7 summarizes the packet loss percentages and median accuracy results across various *mon_window* values. Both metrics decrease with increasing *mon_window* as the system remains longer on each model/feature set before switching. For this particular use case and trace, an eight-second *mon_window* provides the optimal balance between accuracy and packet loss. Similar experiments with our second use case show consistent behavior—both accuracy and packet loss decrease with larger *mon_window* values. The optimal value remains eight seconds (yielding 0.147% packet loss and 0.97 accuracy). We recommend conducting comparable experiments for any use case to determine the optimal *mon_window* value according to specific operator requirements.

mon_window	**1**	**2**	**4**	**8**	**10**
Packet loss (%)	0.107	0.074	0.048	**0.033**	0.031
Accuracy	0.932	0.931	0.931	**0.931**	0.926

Table 4.7: Impact of *mon_window* on `Cruise Control`

4.6 **Related work**

Many studies have investigated similar individual subcomponents integral to `Cruise Control`. Here we examine the related work and discuss key differences with `Cruise Control`'s design.

SYSTEM FOR EXTRACTING FEATURES FROM NETWORK TRAFFIC. Network feature extraction has been a dynamic research area for decades, with systems designed to derive meaningful insights from traffic. Recent advancements focus on high-speed network traffic processing. For example, PacketMill [42] optimizes software for 100Gb/s throughput, Enso [112] introduces a streaming abstraction for improved efficiency, and Retina [134] uses multilayered filtering and streamlined feature extraction for relevant network flows. However, these solutions rely on static configurations that can lead to suboptimal performance under changing network conditions. In contrast, `Cruise Control` introduces a dynamic configuration framework that adapts to varying network loads in real time, offering a more flexible approach.

COST-AWARE ML MODEL CREATION. Neglecting system constraints during model training can significantly impact inference time, causing packet loss that affects ML model performance. To address this, Traffic Refinery [15] proposes a methodology to explore and mitigate data representation technical debt, but it requires manual intervention. In contrast, CATO [135] performs automated, end-to-end optimization of the traffic analysis pipeline, but it's offline and requires model selection online. Liu *et al.* [81] underscore the impact of inter-packet time on the feature extraction process, proposing an approach using three parallel models with varying packet requirements. However, they solely focus ib early application identification and using static models. Other work explores in-network inference within programmable devices such as switches [5, 102, 139] or Field-Programmable Gate Array (FPGA)s [39, 133] to exploit high-speed processing capabilities. However, these approaches face limitations due to restricted command and extraction capabilities of in-network fabric, constraining model features and sophistication.

DYNAMIC MODEL SELECTION. Few studies explore dynamic model selection for traffic analysis. pForest [17] dynamically switches between multiple ML models based on flow packet count, while Jiang *et al.* [68] investigate selecting from classifiers with varying feature requirements to balance classification speed and memory consumption, focusing primarily on memory usage costs. In contrast, `Cruise Control` emphasizes CPU cycles and tailors model selection to optimize computational performance. Vicenzi et al. [133] propose an adaptive framework that switches between pruned Convolutional Neural Networks (CNN) variants based on accuracy and throughput. They use a fixed set of features. Our approach, on the other hand, adapts the feature set itself. Additionally, their system focus on FPGAs, while ours targets commodity hardware. Beyond traffic analysis, adaptive model selection has been explored in general ML serving systems. Clipper [25] and INFaaS [109] dynamically optimize model selection based on performance requirements. Zhang *et al.* [144] pioneered dynamic model serving based on load and pre-characterized performance, inspiring recent studies[66, 113] on machine learning as a service. These works focus on avoiding service-level objective violations with financial penalties, whereas `Cruise Control` focuses on system-level computational costs for processing features for network traffic analysis.

4.7 Conclusion

In this chapter, we introduce `Cruise Control`, a system that dynamically selects target ML models for traffic analysis tasks at runtime, without requiring user intervention. `Cruise Control` leverages lightweight signals to adapt to changing network conditions and the system's available resources. We detail the design of `Cruise Control` and evaluate it across two use cases, demonstrating its advantages under varying traffic configurations.

`Cruise Control` opens several promising directions for future research. One avenue is improving model cost estimation by incorporating additional performance and resource metrics. Another is identifying new system-level metrics to enable finer-grained model selection and improve performance.

In the next chapter, we will see how we can leverage ML to reduce the quantity of flows as an input of a system such as `Cruise Control`.

5 LO-FI: LOW-COST EARLY APPLICATION FILTER BASED ON CACHED ML DECISIONS

This chapter introduces `Lo-Fi`, a hybrid early application filter that aims to combine the strengths of traditional filtering techniques and ML-based approaches. `Lo-Fi` cuts packet loss from 38.33% (ML-only) to 1.17% on CAIDA traces. However, this comes with moderate overhead, peaking at 6.38% of campus network traffic.

5.1 Introduction

Modern traffic analysis pipelines are tasked with extracting diverse information from network flows to support critical operational functions including network telemetry [68, 81, 134, 135], QoE assessment [67, 86, 117], and intrusion detection [54, 74, 80, 145]. However, as traffic volumes increase (often tens to hundreds of gigabits per second [65, 81]) and the complexity of these analysis tasks grows, it becomes paramount to filter out irrelevant traffic to maintain system efficiency and analysis accuracy. In this context, a *filter* is a stateful mechanism that selectively admits or rejects network flows based on predetermined criteria related to their originating applications, operating as a gatekeeper within traffic analysis pipelines. However, despite this fundamental importance, modern traffic analysis solutions have largely neglected the development of robust application-aware filtering mechanisms, an oversight that is particularly problematic as contemporary networks carry increasingly complex and encrypted traffic, making accurate filtering an even more fundamental component of traffic analysis.

Over time, filtering techniques have evolved from pattern-matching methods to learning-based approaches [68, 81, 135]. Traditional static filters rely on header information such as port numbers or IP addresses, allowing for high-speed, low-overhead processing. However, these filters can be brittle as modern services operate over HTTPS [45] and frequently change IP assignments through CDNs [52]. Dynamic filtering approaches leverage DNS queries and TLS metadata, such as inspecting the SNI during TLS handshakes [118, 119], to identify applications. However, pervasive encryption undermines DPI [16, 96], while the ongoing adoption of ECH [62] threatens to eliminate remaining plaintext indicators, underscoring the unsustainability of approaches that depend on plaintext protocol information.

In response, ML and Deep Learning (DL) models have emerged as powerful alternatives, demonstrating remarkable ability to classify encrypted traffic by analyzing statistical properties like packet sizes and timings [80, 117, 135], or raw packet headers [58, 81]. However, applying these approaches to every flow in real time in a high-speed network context introduces significant computational overhead [65, 68, 81, 135, 145]. Our findings show that ML-only approaches can lead to packet loss rates as high as 38.33% on representative network traces due to processing bottlenecks. Furthermore, many academic ML solutions are developed in offline environments [58, 80, 104], often lacking the optimizations and practical considerations needed for real-world, line-rate deployment.

The limitations of existing methods—the inflexibility and diminishing visibility of traditional techniques, and the prohibitive cost of universal ML deployment—motivate the need for a new strategy. Our work introduces Lo-Fi, a hybrid application filter designed to effectively navigate the trade-off between the efficiency of traditional filtering and the accuracy of ML based classification. Lo-Fi's core design combines the low overhead of pattern-matching approaches with the flexibility of ML-based approaches, operating on the principle of judicious ML utilization. Central to its architecture is a fast "short-circuit" path that processes the bulk of recognized flows relying on cached, ML dynamically updated lists of IP destinations.

This pattern-matching is highly efficient, demonstrating significantly lower processing times compared to direct ML inference or even traditional TLS SNI inspection. The more resource-intensive ML pipeline is selectively invoked only for unclassified flows, serving both to classify this unrecognized traffic and to continually enrich the decision cache. To manage the cache effectively, particularly in the context of CDNs and dynamic IP reassignments, Lo-Fi incorporates mechanisms such as Least Recently Used (LRU) eviction, time-based expiration of entries, and thread-safe inter-core sharing for its IP cache. This architecture allows Lo-Fi to achieve substantial performance gains; for instance, on CAIDA traces, its short-circuit reduced packet loss from 38.33% (in an ML-only configuration) to just 1.17%.

5.2 Related Work

In this section, we discuss different approaches in the field of traffic filters, focusing on accuracy and efficiency tradeoffs. We show that existing solutions either rely on outdated techniques or are not designed to be used in modern networking environments, justifying the need for a new approach. We provide a comparison of filtering techniques in Table 5.1.

STATIC FILTERS. Traditional pattern-matching filter techniques that rely on heuristics such as port numbers or IP addresses are computationally efficient (i. e., capable of line-rate performance) but are increasingly inadequate for precise application identification. One primary reason is that the rules underpinning these heuristics—specific ports, static IP lists, or predefined signatures—are often ill-equipped for the dynamic nature of modern internet services. For instance, attempting to identify a specific streaming service by its port number becomes futile when most services utilize the generic HTTPS port (443). Similarly, relying on static IP address lists offers limited utility. The widespread adoption of CDNs and cloud services means that a single IP address can host numerous applications [52], and these IP assignments can change frequently without notice, rendering static IP-based rules quickly obsolete. For example, Luxemburk *et al.* [82] compare two ML approaches to an IP-based classification over QUIC traffic. Their results underscore the failure of IP filtering over time, with accuracy dropping from 100% to 0% within weeks. Our findings confirm this trend, underscoring the need to constantly update IP lists due to the dynamic nature of today's Internet.

Further, the increasing obfuscation of distinguishing information due to pervasive encryption and the adoption of more secure protocols significantly curtails the effectiveness of these traditional methods. While DPI was once a robust method for understanding plaintext traffic [16], its efficacy is fundamentally undermined by widespread network data encryption [96].

Trevisan *et al.* [130] investigated the possibility of performing classification based solely on domain names and their associated IP addresses. Their results show that up to 55% of web traffic could be identified by relying solely on domains. However, they also observed that IP addresses are not stable over time. Their monthly analysis reveals that IP addresses frequently change, posing challenges for consistent classification. Furthermore, they advocate for IP-based classification as a privacy-preserving alternative. Our work takes a similar approach, focusing on IP-based classification. However, we overcome the limitations of their method by introducing a mechanism that can dynamically classify unknown network flows.

DYNAMIC FILTERS. Static filters often lack the adaptability needed to function effectively in modern network environments, which are dominated by CDNs that commonly share IP addresses across multiple services. To overcome this challenge, many filtering systems now use dynamic information to more accurately track and identify relevant IP addresses.

In their work Maghsoudlou *et al.* [85] try to identify services in realtime by inspecting DNS answer records and associating them with NetFlow records. While the findings were positive, this approach is nullified by the increasing use of DNS over encrypted transport [32, 56, 59, 63]. In contrast, we focus solely on available information, such as IP destination, and use an ML approach to reason about encrypted information.

TLS SNI-based filtering was investigated by Shbair *et al.* [119], who examined filtering methods relying on the TLS SNI extension and highlighted two key weaknesses. In their work, they described a firewall-based filtering method where connections to unauthorized servers are reset to block access. They also developed a tool to bypass this type of filtering. In a subsequent study, Shbair *et al.* [118] proposed a new approach that combines TLS SNI monitoring with DNS-based verification. To counter forged SNI values, they required the firewall to resolve the domain via a trusted DNS and compare the resulting IP address with the actual destination IP, ensuring consistency.

Filter techniques	**Low latency**	**Dynamic**	**Payload encryption**	**ECH**	**DNS encryption**
Pattern-matching (IP, Port...)	✓		✓	✓	✓
DPI		✓		✓	✓
DNS	✓	✓	✓	✓	
TLS SNI		✓	✓		✓
ML		✓	✓	✓	✓
Lo-Fi	✓	✓	✓	✓	✓

Table 5.1: Comparison of existing filtering and their limitations

These works highlight that TLS SNI had several known weaknesses even before the introduction of ECH, reinforcing the need for alternative solutions.

ML-BASED FILTERS. ML-based filters are a promising solution for overcoming many of the challenges encountered in network service identification [12]. These filters are particularly effective when dealing with encrypted traffic and are resilient to emerging techniques, such as ECH and DNS encryption. For instance, Bernail *et al.* [10] explored early application identification based on clustering methods. More recently, with the widespread adoption of encryption, ML-based approaches proved to be useable [101]. In their work, Shbair *et al.* [120] applied ML methods for early service identification over HTTPS. Babaria *et al.* [8] introduce FastFlow a flow classifier that use time-series on minimal quantities of packets. Liu *et al.* [81] introduce ServeFlow a Fast-Slow architecture, which performs service classification utilizing early flow characteristics. However, these solutions are limited in their use for filtering large quantities of flows in real time. Designed for efficient traffic classification, ServeFlow requires a dedicated 16-core machine to classify less than 10 Gbps of network traffic. Meanwhile, FastFlow uses eight cores dedicated to classifying 50,000 concurrent flows. Furthermore, they proposes using Graphical Processing Unit (GPU)s to improve classifier performance. However, in our work, we attempt to restrict `Lo-Fi` to a few cores.

Although there has been extensive research on improving the accuracy of network service identification, only few studies consider the constraints imposed by system and network resources. One line of research focuses on feature engineering and selecting the optimal features [15, 68, 135]. However, existing inference methods depend on statistical features from multiple packets, which can cause latency, and still require processing of all flows through ML inference which results taxing for a traffic analysis system. We show in Section 5.4 ML-only configurations incur in 38.33% packet loss when processing common traffic loads. Our work differs in that we look to use ML selectively (i. e., only when necessary), with the goal of maximizing system efficiency.

HYBRID FILTERS. The presented limitations demonstrate the need for a new approach to traffic filtering, aiming to strike the right tradeoff between processing efficiency of static filters with the accuracy of ML-models. However, as far as we know, the most closely related work is SnortML, the detection engine that was introduced in Snort, the well-known open-source Intrusion Detection System (IDS), version 3.1.82.0 [108]. SnortML uses ML binary classifier to detect zero-day exploits and trigger alerts. As an IDS/Intrusion Prevention System (IPS), Snort can use an access control engine to block or permit network traffic in reaction to alerts, though this is not mandatory. Our approach is similar in that we also use binary classifiers to accept or reject traffic. However, our system differs significantly in that we aim to build an application filter based on service identification, whereas SnortML is built to detect zero-day exploits. Further, we support all models that support the Open Neural Network Exchange (ONNX) format, while SnortML is limited to TensorFlow. Finally, to the best of our knowledge, Snort3 does not explicitly aim to minimize the use of ML to enhance performance, unlike `Lo-Fi`, which does so by design. We present `Lo-Fi` in the next Section.

5.3 **`Lo-Fi`**

In this section, we present `Lo-Fi`, a proof of a concept application filter that leverages the speed of pattern-matching rules and the versatility of ML classification. We begin by presenting metrics and concept underlying `Lo-Fi`. Next, we describe the various steps a packet can take through the system. Finally, we discuss how the system uses ML decisions to improve performance.

5.3.1 *Low-cost early application filter*

In this section, we first present the key metrics used by `Lo-Fi` to achieve early service identification. Then, we detail how ML models are employed as a fallback mechanism when these metrics alone are insufficient. Lastly, we discuss the model-agnostic architecture of `Lo-Fi` and the deployment considerations for integrating various ML models.

EARLY INDENTIFICATION. We rely on two distinct set of metrics to enable early and low-cost service identification. The first set of metrics is used to determine whether a packet belongs to a previously seen flow. To this end, we use a flow cache to store accepted and rejected flows. These flows are indexed by the five-tuple, which includes the source and destination IP addresses, the source and destination ports, and the transport protocol. The second set of metrics supports service detection. Here, we maintain a cache of destination IP addresses that have been associated with the targeted service. By comparing new flows against this cache, we can infer the likely service destination. If either condition is met, we can accept the flow early without performing more costly analyses.

IDENTIFYING SERVICES. Although the cache is essential for quickly identifying services, it must be continuously updated with reliable information to maintain its accuracy. Due to the dynamic nature of IP assignments, especially in environments involving CDNs [52], manually updating the cache is impractical and error-prone. To address this issue, `Lo-Fi` leverages ML classification to supplement the cache. When packet-level metrics and existing cached data are insufficient for identifying services, ML inference is used to classify flows and update the cache with new IP. This process ensures that the cache adapts to changes in network and follows IP assignments. Additionally, a time-based expiration mechanism removes outdated entries to prevent inaccurate classification due to outdated data. By applying ML parsimoniously, only when simpler methods fail, `Lo-Fi` maintains efficiency without sacrificing accuracy.

MODEL AGNOSTIC. Due to the rapid evolution of ML, systems tied to a single model architecture may become obsolete before they are even deployed. Moreover, many models remain as research prototypes, overlooking the practical challenges encountered during real-world deployment. This results in models that show high accuracy in controlled settings but underperform in live environments [7]. To address this issue, `Lo-Fi` is designed to facilitate the seamless deployment and evaluation of off-the-shelf ML models within real-world network environments. Serving as a flexible testbed for models, `Lo-Fi` is designed to be model-agnostic, allowing network operators and researchers to test diverse models and compare them under realistic conditions with minimal integration effort.

To ensure compatibility and interoperability across models and frameworks, we enforce three key constraints:

1. **ONNX.** ONNX runtime[30] is a framework that supports a wide variety of machine learning libraries. By adopting ONNX, `Lo-Fi` becomes agnostic to the training framework. This allows developers to use their preferred tools (Such as TensorFlow[31], PyTorch[64], Scikit Learn[77]) while ensuring consistent deployment. ONNX serves as a standardized, plug-and-play interface between trained models and the system, thus simplifying integration and execution.

2. **Binary Classification.** Each model must perform binary classification over network flows, producing a decision interpreted as accepted or refused. This abstraction enables uniform handling of inference results across different model architectures. Multi-class classification could probably be used, too, but we will save that exploration for future work.

3. **Subscription.** Models must specify the type of flow data they require, referred to as a Subscription, as well as the number of packets needed for inference (see the next paragraph for details). These specifications ensure that the system can accurately match each model's requirements and maintain compatibility across different models.

By respecting to these constraints, `Lo-Fi` can seamlessly integrate a wide variety of ML models, supporting reproducible benchmarking and operational testing at scale.

SUBSCRIPTION. In `Lo-Fi`, a Subscription defines the specific type of input data that the system can provide to a ML model. Each subscription represents a set of features derived from network flows, enabling the system to tailor the data collection and processing pipeline according to the needs of the model.

In the proof of concept, `Lo-Fi` natively provide three different Subscriptions:

- **Stats.** A feature set based on coarse statistical features, inspired by the work of Fauvel *et al.* [43]. This represent the direction and the size of the packets of a flow.
- **Raw.** A byte-level representation of packets, where each packet is encoded as a fixed-length array of 1518 uint8 values. Each value corresponds to a byte from the original packet. Raw features are oftenly use by Deep Learning-oriented models[116].
- **NPrint.**[58]. A generic representation of packet data, designed for ML-based network traffic analysis. In this representation, every bit of the header is either represented by its value as 0 or 1, or is absence as -1, according to a set of protocol statics for each flow. In our implementation, we natively extract Nprint for IPv4/TCP/UDP, with each protocol including all its options.

EXTENSIBILILITY. However, `Lo-Fi` is designed to be extensible. New types of Subscriptions can be implemented, they only need to implement the dynamic Subscription trait. This trait enables the system to handle arbitrary feature extraction mechanisms uniformly. All Subscription implementations must define five core functions to create, add packets, process features, and get metadata such as the number of packets and the number of features. Once the number of packets required by the model is reached, the corresponding feature set is exported in tensor format[1], dimensioned by the number of features and the number of packets. Then it is forward to the ML model for inference, after this inference the decision and the flow is transfer to the next step.

5.3.2 *System Workflow*

The system is built in a three key steps, illustrated in Figure 5.1.

First, pattern-matching attempts to leverage previously cached ML decisions that can be applied to new incoming packets, aiming to avoid invoking ML whenever possible. If no cached decision matches the incoming packet, the system falls back to ML classification. Although `Lo-Fi` tries to minimize ML usage for efficiency, ML remains the core mechanism for handling uncertain or novel flows. Unlike traditional

1 Multidimensionnal array used by ML models

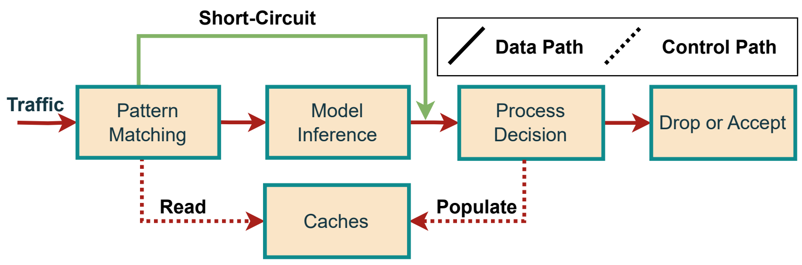


Figure 5.1: Diagram of `Lo-Fi`

filters, which often ignore unmatched packets [91], `Lo-Fi` redirects these packets to the ML module, which is designed to classify unknown traffic, including encrypted flows by analyzing statistical features [122]. To this end, packets are gathered in a dedicated cache and sorted by flow. This allows the system to gather sufficient context for analysis, as ML models typically rely on features extracted from multiple packets rather than individual ones[15, 135]. Once there are enough packets to meet the model's requirements, features are extracted and sent to the ML model for inference. The model then performs a binary classification to determine the relevance of the flow. Finally, the Process Decision component applies the classification outcome to the flow and updates the caches accordingly. This step extracts all available knowledge from the decision and stores it in the correct cache. This enables `Lo-Fi` to judiciously leverage ML while capitalizing on previous decisions to reduce unnecessary computational costs.

5.3.3 *Capitalize on past decisions*

The cornerstone of `Lo-Fi`'s performance is the parsimonious use of ML, preferring to capitalize on stored past decisions. In this section, we will see how the system populates these caches based on the filter outcome and how it manages them.

CACHE POPULATION Actions realized by the Process Decision component is highly related on the decision took by the filter. We will see here the different cases, and how it handles them.

Firstly, when a flow is identified by the ML model as irrelevant, the system record the 5-tuple in the Flow cache as a refused flow and drop it. Conversely, when the ML model identifies a flow as relevant, the system records the five-tuple in the flow cache as an accepted flow and adds the destination IP address to the IP cache. If the incoming flow has matches an entry of the IP cache, we jump directly to adding the five-tuple to the flow cache as accepted. Lastly, to prevent cache pollution and eliminate orphaned entries, the system purges related entries from various caches.

CACHE MANAGEMENT As described above, `Lo-Fi` maintains several caches throughout its decision pipeline to efficiently store and reuse knowledge. Without proper management, the caching mechanism may exhaust available memory under high traffic loads, potentially resulting in packet loss or even system crashes. To address this issue, we opted for a cache, which is bounded by a maximum size by design, rather than a resizable HashMap. Each cache has a fixed maximum number of entries and uses a predefined eviction policy when it reaches its maximum capacity. We use the LRU policy, which discards the entries that have been accessed least recently. This approach effectively removes inactive or outdated flows, including unfinished connections, which are a common occurrence in real-world traffic due to widespread TCP SYN scanning [35, 36] and incomplete handshakes [71].

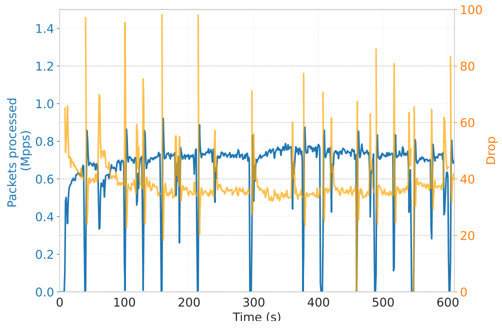
Finally, since the caches are shared across multiple CPU cores, they must be implemented as thread-safe structures in order to support concurrent access without compromising accuracy or performance. This intercore sharing depends from how network flows are distributed across cores. Many high-performance networking stacks employ consistent hashing mechanisms (e.g., based on packet five-tuples) to ensure that packets belonging to the same connection are routed to the same processing core. This behavior is sufficient to maintain consistency in the flow level caches, as all packets of a given flow are handled by a single core.

However, the IP cache presents a more complex scenario. Two distinct flows with the same destination IP address may be processed by different cores. In such cases, one core may have cached a trusted IP, while another core, unaware of this, unnecessarily invokes the ML pipeline. To prevent this inefficiency, the IP cache is implemented as a shared structure accessible across all cores. Maintaining a globally accessible, thread-safe IP cache ensures that knowledge about known IP is leveraged system-wide, reducing redundant computation and improving overall throughput. Furthermore, the IP cache uses a time-based eviction policy to remove outdated information. This helps prevent inaccurate classifications caused by outdated data(see Section 5.4.4).

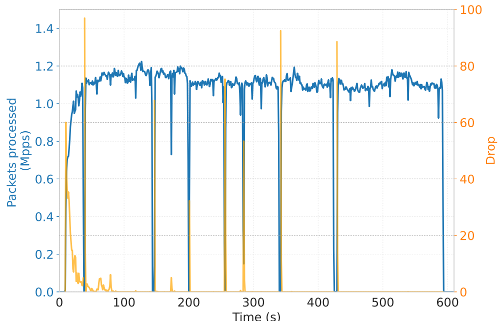
5.4 Evaluation

We evaluate `Lo-Fi` on multiple scenarios using CAIDA traces[20], network traces from a tap in our campus network[2], and purely synthetic traffic for micro benchmarks. We use the same testbed environment presented in Chapter 4.

2 This tap was configured in collaboration with our campus networking and security teams to obtain a copy of network traffic.All captured traffic was anonymized to protect privacy and mitigate ethical concerns related to data collection. Importantly, the study did not involve any analysis of individual user behavior; our focus remained exclusively on the performance and characteristics of services.



(a) Applying ML to each connections



(b) LoFi

Figure 5.2: Number of processed and dropped packets (in Mpps) over 10 minutes of CAIDA traffic using 2 CPU cores

5.4.1 *Prototype Implementation*

The proof-of-concept prototype was developed in Rust to leverage its memory safety and concurrency guarantees. During the development process, a particular effort was made to design the filter as a library, which will be released as an open-source project upon acceptation. This will enable operators and researchers to easily deploy various models. We strongly encourage the community to adapt and customize the library according to their needs. This could mean deploying and testing models using the provided Subscription type, or implementing a new one.

We use the Moka[92] library to handle the caches presented in Section 5.3. Moka's strong performance guarantees come from its internal partitioning into multiple, independently locked segments. This locks only the relevant segment during access or modification, enhancing concurrency. Multiple threads can operate on the cache simultaneously without blocking each other.

The filter is designed to be agnostic to the underlying packet delivery mechanism. For this proof of concept, we chose DPDK[29] due to its high performance and ability to process packets with low latency and high throughput via kernel bypass. We therefore encourage the use of DPDK when deploying the system to fully leverage its performance capabilities.

5.4.2 *Overall performances*

To evaluate the overall performance of the system under realistic conditions, we present an experiment, illustrated in figure 5.2, using the CAIDA trace from Equinix Chicago, captured on 2016-01-21 [20]. The trace is used in its original form, with the only modification being the merging of both traffic directions to ensure complete connections are preserved. We consider this trace to represent realistic traffic, as it originates directly from an ISP network.

We first run `Lo-Fi` without the Pattern-matching component, which consists to run the ML pipeline on each connection. This approaches try to replicate state-of-the-art service classification [80, 104], where model are typically evaluate offline. Such non-real-time analyses often process the entire traffic dataset without properly considering system constraints or deployment limitations.

We ran it with two CPU cores for the first ten minutes and with the NPrint Subscription for the first ten packets. Restricting the number of CPU cores is mandatory because we expect to add more computation and not just filter traffic on the same server. As Figure 5.2a shows, the system can not handle the load, which results in a lot of traffic dropped. At the end of the ten minutes, 38.33% of the input traffic has been dropped. Next, we run the same traffic with the Pattern-matching component. As Figure 5.2b shows, we observe a brief warm-up period during which the system achieves nearly zero packet loss with some spikes in loss. Spikes down in the traffic are naturally present in the traces, likely caused by artifacts during capture or anonymization. These irregularities allow us to observe how `Lo-Fi` behaves when faced with imperfect, real-world traffic, providing insights into its practical performance. After ten minutes, 11% of connections were rejected, 31% were accepted, and 57% were still in the cache, waiting for enough packets, resulting in only 1.17% packet loss.

This experiment demonstrates the efficiency benefits of `Lo-Fi` compared to state-of-the-art approaches under realistic workloads.

5.4.3 *ML performances*

In the following section, we will break down the system to compare `Lo-Fi` to other approaches.

COST ESTIMATION We first run micro benchmark performances of both Pattern-matching and ML subscriptions. For this experiment we send dummy traffic generated with T-Rex and manually aimed particular path within the system, illustrated in Figure 5.3. To evaluate the Pattern-matching (in red), we generate packets targeting a pre-cached IP address, enabling focused microbenchmarking of this specific system component. To evaluate the ML's subscriptions (in green, yellow, and cyan), we send the required number of packets with a random five-tuple and remove them from the cache after ML processing to avoid falling back on the Short-circuit path. Finally, to establish a baseline and highlight its performance characteristics, we implemented TLS SNI processing in purple. This shows that 4 packets are required because the Client Hello is the fourth packet of the connection, though this fourth packet can be split into multiple TCP packets, thus requiring more packets.

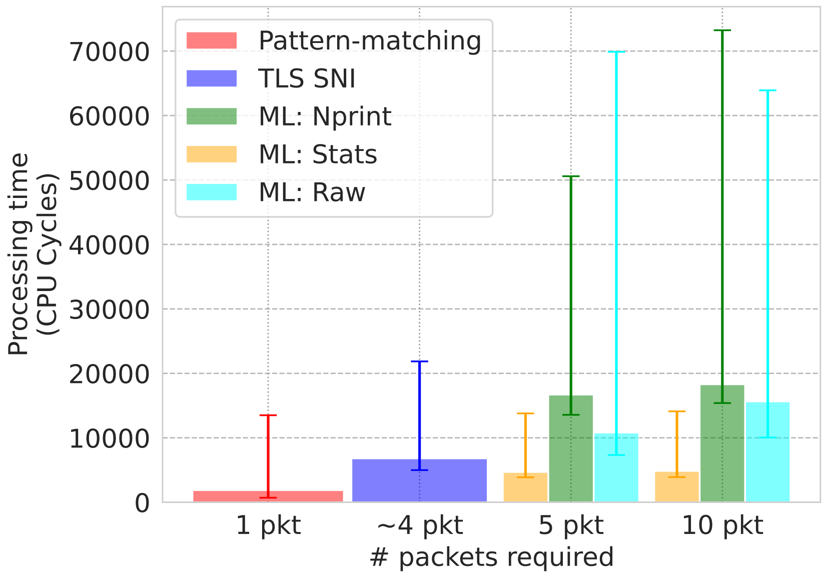


Figure 5.3: Comparison of cost estimates (1st and 99th percentiles): ML Subscriptions vs. Short-Circuit method (TLS SNI) as the baseline

To avoid skewing the results based on the performance of the models or the network, we did not consider the inference time of the machine learning or the time between packets, as these may vary according to use cases. This is motivated by the inherent design of `Lo-Fi` and recent work showing the network latency is the higher and yet incompressible cost in packet processing [81]. Thus, the evaluation focuses primarily on the costs associated with Pattern-matching and Subscription types.

We measure processing time by recording the number of CPU cycles elapsed between the first line of code representing the packet's entry into the system and the last line of code where the filter makes its decision. To do so, we read the *TSC* register via the *RDTSC* instruction.This provides us a low-overhead method for estimating costs.

Figure 5.3 shows the median processing times and the 1st–99th percentile range for each method. Pattern-matching is the best, with a median of 1,888 CPU cycles. Stats Subscriptions are next, with similar medians for 5 and 10 packets (4,704 and 4,864 CPU cycles), due to their computational simplicity, even lower than TLS SNI (6,816 CPU cycles). Raw Subscriptions are more costly, increasing from 10,816 to 15,648 CPU cycles. Nprint has the highest cost: 16,704 CPU cycles for 5 packets and 18,304 for 10. All methods are right-skewed, with more variability in complex cases, motivating the use of the median.

These results highlight the significantly lower cost of Pattern-matching compared to both the ML path and TLS SNI. This emphasizes the importance of incorporating a Pattern-matching in `Lo-Fi`, as it plays a critical role in improving overall system performance.

5.4.4 *Short-Circuit performance*

In this section, we focus on the performances of the Pattern-matching based on the cached IP. As illustrated in Section 5.4.3, an IP-based filter is competitive in terms of computational overhead. We will focus on the overhead in terms of flow noises induced by using this coarse filter. We collected traces from a network tap for eight days and extracted the available SNI and DNS records.

We removed the internal services and extracted the ten most used services. Next, we added five other popular services that were not on the list: Threads, YouTube, Facebook, Apple, and Netflix.

To emulate `Lo-Fi`, we identified IP destinations from target-labeled flows, then extracted all flows to those IPs to measure overhead. This simulates a perfect case where all the relevant IPs are already in cache, extracting flows without requiring new inference.

The first column is the service we try to extract and its flows in % of the all flows of the eight days. The Target column represents the quantity of flows that represent the perfect score, where the filter extracts solely the flows where ground truth is known as the targeted services.

In the left part of Table 5.2, we replicate our system behavior with an eviction policy based on days. As expected, the system can extract all the required flow, but also an extraneous quantity of flow. The overhead (Extraneous) can be understood as the difference between the quantity of flow identified as the service (Target) and the quantity of flows extracted (Extracted). Extraneous flow is then split into two categories: the one Miss Classified (MC, in red) identified by SNI or DNS as another service than the one targeted, and the unknown (UK, in blue) where no ground truth is available. This last category can originate either from the target services or from others. Note that, while this type of traffic is typically dropped by classical filters [91, 134], but our approach retains it.

We can see that the overhead varies depending on the service. For example, YouTube has 2.889% extraneous flows and 1.859% of its flows lack ground truth. Notably, only 0.264% of these extraneous flows are directed to non-YouTube domains, indicating relatively accurate targeting despite some ambiguity. Other domains, like Hypotheses (French research notebook in Humanities and Social Sciences), show a more moderate split, with only 0.006% misclassified and 0.007% unknown. These cases highlight how the scale and nature of a service can influence the precision of IP filtering.

With a maximum observed overhead of 6.534%, the results remain within acceptable limits, especially considering the inherent limitations of IP filtering in today's CDN-driven networks. In comparison, services like Apple show a more noticeable discrepancy, with a target volume of 0.515% and an extracted volume of 2.736%, suggesting significant IP address sharing. These results confirm that IP filtering effectively reduces overall traffic volume, despite inducing a moderate overhead.

Service	**Target (%)**	**Daily**		**Hourly**	
		Extracted(%)	Extraneous MC / UK	Extracted(%)	Extraneous MC / UK
google	1.547	6.534	0.264/4.724	4.868	0.196/3.126
icloud	0.332	1.352	0.039/0.980	0.950	0.022/0.597
office	0.434	1.324	0.480/0.410	0.850	0.209/0.207
hypotheses	0.169	0.182	0.006/0.007	0.171	0.001/0.001
gmail	0.119	0.327	0.036/0.172	0.166	0.003/0.043
microsoft	0.989	2.354	0.438/0.927	1.927	0.241/0.697
spotify	0.236	0.389	0.071/0.082	0.320	0.040/0.045
hubiconnect	0.166	0.169	0.000/0.003	0.167	0.000/0.000
github	0.189	0.231	0.008/0.034	0.199	0.002/0.008
adobe	0.077	0.507	0.259/0.171	0.240	0.084/0.078
threads	0.013	0.056	0.003/0.039	0.014	0.000/0.000
youtube	0.072	2.961	1.030/1.859	1.465	0.472/0.922
facebook	0.073	0.652	0.055/0.523	0.549	0.049/0.427
apple	0.515	2.736	0.972/1.249	1.384	0.452/0.417
nflx	0.015	0.045	0.002/0.028	0.016	0.000/0.001

Table 5.2: The percentage (rounded) of the targeted flow comparison of the extracted flow and a breakdown of the extraneous flow split into two categories: misclassified (MC) and unknown (UK).

Table 5.2 examines noise reduction and applies an hourly IP cache eviction policy, which results in a significant drop in all extracted volumes. In particular, services like HubiConnect and Netflix see extracted flows drop to nearly 0%. For others—such as Gmail, Adobe, YouTube, and Apple—the overhead is reduced by half. Although the reduction is less significant for some services, measurable gains are still observed. Shortening the IP cache expiration time highlights the necessity of time-based eviction to reduce noise and confirms that frequent invalidation is an effective tuning strategy.

5.5 Conclusion

In this chapter, we introduce `Lo-Fi`, a low-cost application filter based on an IP filter enhanced by ML models to leverage the best of both worlds. Through several examples, we showed that the coarse granularity of the IP filter is offset by its efficiency, making an ideal fit to a more complex method. Severals keys problems remains, such as the low-cost discrimination of services behind CDNs, the enhancement of filtering with multiclass or ensemble classifiers, and the exploration of hardware-based rule deployment for further performance gains.

6 CONCLUSIONS

This thesis explores the limitations induced by deploying ML to a real-world operator network. In what follows, we summarize the key contributions of this work and outline several promising directions for future research.

6.1 Contributions

In this thesis, we have made several key contributions that advance the understanding and deployment of ML in networking. First, we conducted a study on the impact of packet losses during the feature extraction process, demonstrating how such losses can significantly degrade the performance of ML models, an aspect often overlooked in prior work. Building on these insights, we designed a novel system that addresses this challenge by dynamically adapting its configuration to the current network and system context, extrapolated from coarse but reliable low-level metrics. This approach provides a lightweight yet robust mechanism to preserve model accuracy under adverse conditions. Finally, we developed and deployed a system capable of deploying off-the-shelf ML models at line rate for traffic filtering. This system shows that higher-speed traffic analysis can be sustained by leveraging previously made decisions. Together, these contributions participate in bridging the gap between on-paper ML model performance and practical deployment in real-world, high-speed networking environments, thus paving the way for more resilient, adaptive, and efficient ML-based network analysis.

6.2 Perspectives

6.2.1 *Hardware optimization*

While the present work relies on software-based mechanisms, dedicated hardware solutions are expected to deliver substantial lower latency. Offloading computationally intensive tasks, such as filtering, data preprocessing, or even flow preassembly, to specialized hardware (e.g., FPGA or smartNIC) could significantly increase system throughput, reduce latency and improve system stability. This path appears particularly promising for achieving large-scale deployment in high-speed environments, although it also introduces important trade-offs between hardware efficiency and software flexibility. A key open question concerns how tasks should be repartitioned between hardware and software, depending on the capabilities and constraints of the hardware platform. Achieving an optimal balance is unlikely to be straightforward, as demonstrated in this thesis, no single configuration remains universally optimal for network traffic analysis. This naturally leads to the question of whether hardware itself could be reconfigured on the fly, enabling the system to adapt dynamically to evolving traffic conditions [133]. Such runtime adaptability would represent a major step forward, but it raises new research challenges related to reconfiguration latency, overhead, and stability. Exploring these issues opens a promising direction for the co-design of adaptive hardware and software pipelines in future high-speed network monitoring systems.

6.2.2 *Network traffic prediction*

This thesis focused on system-level metrics to enable timely detection and mitigation of performance degradation. A natural extension is to evolve from reactive mechanisms toward proactive strategies through network traffic prediction techniques. By anticipating resource contention, traffic bursts, and the resulting packet losses before they occur, the system could adapt in advance and substantially reduce disruptions. Integrating ML or DL models for network traffic prediction [6] opens the possibility of building systems that make decisions ahead of time, shifting from mitigation to true prevention. Such a capability could redefine the resilience of high-speed network monitoring, enabling pipelines that continuously adjust not only to present conditions but also to predicted future states. As short-term fluctuations and sporadic bursts are difficult to forecast, the next generation of resilient systems should integrate predictive models alongside adaptive pipelines. This approach makes proactivity and reactivity complementary rather than antinomic, enhancing system performance and minimizing losses through their concurrent implementation.

6.2.3 *ML-related metrics*

In this work, we focus on the feature-extraction pipeline, treating it as a dynamic subsystem that adapts its internal configuration (e.g., the selected features) in response to system-level metrics. In contrast, the ML model remains fixed and outside the adaptive loop. This separation allows us to study how an adaptive pipeline interacts with a static model, highlighting the trade-offs between feature selection and network conditions. An important extension of this study is the inclusion of ML-related metrics, such as the end-to-end latency from packet ingress to the ML decision (commonly referred to as Time To Decision (TTD) [68]). Incorporating these metrics introduces new challenges. For example, how many packets should the system process before sending them to these values will likely vary according to network speed, context, and computed features. Allowing the system to explore both feature complexity and packet quantity jointly would enable it to adapt more effectively to changing network conditions. Introducing TTD handling would align the pipeline's workload characteristics with system capabilities, while retaining the strengths of `Cruise Control`.

6.2.4 *Closing the loop*

In this thesis we intentionally left the interpretation of the ML output to the operator. Future systems could close the loop by parsing ML decisions and directly back into the network-control plane, allowing the system to automatically adjust network configuration based on the model's predictions. Such a system could, for instance, reallocate resources after an ML-based classification or reroute traffic when a drop in QoE is detected. Closing the loop in this way, combined with adaptive pipelines, would move network monitoring toward full autonomy, giving rise to intelligent architectures capable of self-optimizing and maintaining performance with minimal human intervention. However, such system capable to reprogram network component on the fly already exist in different context [53]. This encouraging us to claim that such close loop can be an interessing way to explore in the future.

BIBLIOGRAPHY

[1] Mahmoud Abbasi et al. "Deep Learning for Network Traffic Monitoring and Analysis (NTMA): A Survey." In: *Computer Communications* 170 (2021), pp. 19–41. ISSN: 0140-3664. DOI: `https://doi.org/10.1016/j.comcom.2021.01.021`. URL: `https://www.sciencedirect.com/science/article/pii/S0140366421000426` (cit. on pp. 11, 33).

[2] Ziawasch Abedjan et al. "Detecting data errors: where are we and what needs to be done?" In: *Proc. VLDB Endow.* (2016) (cit. on p. 19).

[3] Zeeshan Ahmad et al. "Network intrusion detection system: A systematic study of machine learning and deep learning approaches." In: *Transactions on Emerging Telecommunications Technologies* 32.1 (2021), e4150 (cit. on p. 31).

[4] Iman Akbari et al. "A Look Behind the Curtain: Traffic Classification in an Increasingly Encrypted Web." In: *Proc. ACM Meas. Anal. Comput. Syst.* 5.1 (Feb. 2021). DOI: `10.1145/3447382`. URL: `https://doi.org/10.1145/3447382` (cit. on pp. 33, 45).

[5] Aristide Tanyi-Jong Akem et al. "Encrypted Traffic Classification at Line Rate in Programmable Switches with Machine Learning." In: *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. 2024 (cit. on p. 55).

[6] Ons Aouedi et al. "Deep learning on network traffic prediction: Recent advances, analysis, and future directions." In: *ACM Computing Surveys* 57.6 (2025), pp. 1–37 (cit. on p. 72).

[7] Daniel Arp et al. "Dos and Don'ts of Machine Learning in Computer Security." In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3971–3988. ISBN: 978-1-939133-31-1. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/arp` (cit. on pp. 13, 18, 62).

[8] Rushi Jayeshkumar Babaria et al. "FastFlow: Early Yet Robust Network Flow Classification using the Minimal Number of Time-Series Packets." In: *arXiv preprint arXiv:2504.02174* (2025) (cit. on pp. 18, 19, 31, 32, 34, 36, 60).

[9] Tom Barbette et al. "RSS++: load and state-aware receive side scaling." In: *Conference on Emerging Network Experiment and Technology* (2019). DOI: `10.1145/3359989.3365412` (cit. on p. 51).

[10] Laurent Bernaille et al. "Early Application Identification." In: *International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. 2006 (cit. on pp. 12, 31, 60).

[11] Timm Böttger et al. "Open Connect Everywhere: A Glimpse at the Internet Ecosystem through the Lens of the Netflix CDN." In: *SIGCOMM Comput. Commun. Rev.* 48.1 (Apr. 2018), pp. 28–34. ISSN: 0146-4833 (cit. on p. 10).

[12] Raouf Boutaba et al. "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities." In: *Journal of Internet Services and Applications* 9.1 (2018), pp. 1–99 (cit. on pp. 11, 31, 33, 60).

[13] Leo Breiman. "Random Forests." In: *Mach. Learn.* (2001) (cit. on p. 22).

[14] Francesco Bronzino et al. "Inferring Streaming Video Quality from Encrypted Traffic: Practical Models and Deployment Experience." In: *Proc. ACM Meas. Anal. Comput. Syst.* 3.3 (Dec. 2019) (cit. on pp. 12, 20–22, 28, 31, 33, 35, 36, 38, 45).

[15] Francesco Bronzino et al. "Traffic refinery: Cost-aware data representation for machine learning on network traffic." In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 5.3 (2021), pp. 1–24 (cit. on pp. 4, 18, 31, 32, 34, 38, 55, 60, 64).

[16] Tomasz Bujlow et al. "Independent comparison of popular DPI tools for traffic classification." In: *Computer Networks* 76 (2015), pp. 75–89 (cit. on pp. 57, 59).

[17] Coralie Busse-Grawitz et al. *pForest: In-Network Inference with Random Forests*. 2022. arXiv: `1909.05680 [cs.NI]`. URL: `https://arxiv.org/abs/1909.05680` (cit. on p. 55).

[18] Qizhe Cai et al. "Understanding host network stack overheads." In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 65–77. ISBN: 9781450383837 (cit. on pp. 11, 17).

[19] Qizhe Cai et al. "Understanding host network stack overheads." In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 65–77. ISBN: 9781450383837. DOI: `10.1145/3452296.3472888`. URL: `https://doi.org/10.1145/3452296.3472888` (cit. on p. 44).

[20] CAIDA. *CAIDA Anonymized Internet Traces 2016 Dataset*. Access restricted to approved researchers. 2016. URL: `https://www.caida.org/data/passive/passive_2016_dataset.xml` (cit. on pp. 35, 43, 46, 65, 66).

[21] Jonathan Cavitt et al. "Detecting cyber attacks with packet loss resilience for power systems." In: *Sustainable Computing: Informatics and Systems* (2022) (cit. on p. 19).

[22] Danilo Cerović et al. "Fast Packet Processing: A Survey." In: *IEEE Communications Surveys & Tutorials* (2018) (cit. on p. 17).

[23] Tejalal Choudhary et al. “A comprehensive survey on model compression and acceleration.” In: *Artificial Intelligence Review* 53.7 (Feb. 2020), pp. 5113–5155. ISSN: 1573-7462. DOI: 10.1007/s10462-020-09816-7. URL: http://dx.doi.org/10.1007/s10462-020-09816-7 (cit. on p. 13).

[24] Benoit Claise. *Cisco systems netflow services export version 9*. Tech. rep. 2004 (cit. on p. 2).

[25] Daniel Crankshaw et al. *Clipper: A Low-Latency Online Prediction Serving System*. 2017. arXiv: 1612.03079 [cs.DC]. URL: https://arxiv.org/abs/1612.03079 (cit. on p. 55).

[26] Eric S. Crawley et al. *A Framework for QoS-based Routing in the Internet*. RFC 2386. Aug. 1998. DOI: 10.17487/RFC2386. URL: https://www.rfc-editor.org/info/rfc2386 (cit. on p. 9).

[27] Alexander D’Amour et al. “Underspecification presents challenges for credibility in modern machine learning.” In: *J. Mach. Learn. Res.* (2022) (cit. on pp. 4, 13, 18).

[28] Alberto Dainotti et al. “Issues and future directions in traffic classification.” In: *IEEE Network* 26.1 (2012), pp. 35–40 (cit. on p. 8).

[29] *Data Plane Development Kit*. https://www.dpdk.org/. 2023 (cit. on pp. 14, 17, 44, 66).

[30] ONNX Runtime developers. *ONNX Runtime*. https://onnxruntime.ai/. Version: 1.22.0. 2021 (cit. on p. 62).

[31] TensorFlow Developers. “TensorFlow.” In: *Zenodo* (2022) (cit. on p. 62).

[32] Sara Dickinson et al. *Usage Profiles for DNS over TLS and DNS over DTLS*. RFC 8310. Mar. 2018. DOI: 10.17487/RFC8310. URL: https://www.rfc-editor.org/info/rfc8310 (cit. on pp. 1, 59).

[33] Giorgos Dimopoulos et al. “Measuring video QoE from encrypted traffic.” In: *Proceedings of the 2016 Internet Measurement Conference*. 2016, pp. 513–526 (cit. on p. 12).

[34] *DNS Encryption Explained — blog.cloudflare.com*. https://blog.cloudflare.com/dns-encryption-explained/. [Accessed 12-08-2025] (cit. on p. 9).

[35] Zakir Durumeric et al. “{ZMap}: Fast internet-wide scanning and its security applications.” In: *22nd USENIX Security Symposium (USENIX Security 13)*. 2013, pp. 605–620 (cit. on p. 65).

[36] Zakir Durumeric et al. “Ten years of zmap.” In: *Proceedings of the 2024 ACM on Internet Measurement Conference*. 2024, pp. 139–148 (cit. on p. 65).

[37] E. O. Elliott. “Estimates of error rates for codes on burst-noise channels.” In: *The Bell System Technical Journal* (1963) (cit. on p. 23).

[38] Reham T. Elmaghraby et al. "Encrypted network traffic classification based on machine learning." In: *Ain Shams Engineering Journal* (2024) (cit. on p. 20).

[39] Mohammed Elnawawy et al. "FPGA-Based Network Traffic Classification Using Machine Learning." In: *IEEE Access* 8 (2020), pp. 175637–175650. DOI: 10.1109/ACCESS.2020.3026831 (cit. on pp. 14, 55).

[40] *Encrypted Client Hello: the future of ESNI in Firefox – Mozilla Security Blog — blog.mozilla.org*. https://blog.mozilla.org/security/2021/01/07/encrypted-client-hello-the-future-of-esni-in-firefox/. [Accessed 12-08-2025] (cit. on p. 9).

[41] Nick Erickson et al. *AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data*. 2020. arXiv: 2003.06505 (cit. on p. 22).

[42] Alireza Farshin et al. "PacketMill: toward per-Core 100-Gbps networking." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 1–17. ISBN: 9781450383172. DOI: 10.1145/3445814.3446724. URL: https://doi.org/10.1145/3445814.3446724 (cit. on p. 54).

[43] Kevin Fauvel et al. *A Lightweight, Efficient and Explainable-by-Design Convolutional Neural Network for Internet Traffic Classification*. 2023. arXiv: 2202.05535 [cs.LG]. URL: https://arxiv.org/abs/2202.05535 (cit. on p. 63).

[44] Anja Feldmann et al. "The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic." In: *Proceedings of the ACM Internet Measurement Conference*. IMC '20. ACM, Oct. 2020, pp. 1–18. DOI: 10.1145/3419394.3423658. URL: http://dx.doi.org/10.1145/3419394.3423658 (cit. on p. 35).

[45] Adrienne Porter Felt et al. "Measuring {HTTPS} adoption on the web." In: *26th USENIX security symposium (USENIX security 17)*. 2017, pp. 1323–1338 (cit. on p. 57).

[46] Michael Finsterbusch et al. "A Survey of Payload-Based Traffic Classification Approaches." In: *IEEE Communications Surveys & Tutorials* 16.2 (2014), pp. 1135–1156 (cit. on p. 8).

[47] Daniele Foroni et al. "Estimating the extent of the effects of Data Quality through Observations." In: *IEEE 37th International Conference on Data Engineering (ICDE)*. 2021 (cit. on p. 19).

[48] E. N. Gilbert. "Capacity of a burst-noise channel." In: *The Bell System Technical Journal* (1960) (cit. on p. 23).

[49] Youdi Gong et al. "A survey on dataset quality in machine learning." In: *Information and Software Technology* (2023) (cit. on p. 19).

[50] *Google Transparency Report — transparencyreport.google.com*. [Accessed 12-08-2025]. URL: `https://transparencyreport.google.com/https/overview?hl=en` (cit. on p. 11).

[51] Idio Guarino et al. "On the use of Machine Learning Approaches for the Early Classification in Network Intrusion Detection." In: *2022 IEEE International Symposium on Measurements & Networking (M&N)*. 2022, pp. 1–6. DOI: `10.1109/MN55117.2022.9887775` (cit. on p. 12).

[52] Run Guo et al. "CDN Judo: Breaking the CDN DoS Protection with Itself." In: *NDSS*. 2020 (cit. on pp. 57, 59, 62).

[53] Arpit Gupta et al. "Sonata: Query-driven streaming network telemetry." In: *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 2018, pp. 357–371 (cit. on p. 73).

[54] Ragini Gupta et al. "Generative active adaptation for drifting and imbalanced network intrusion detection." In: *arXiv preprint arXiv:2503.03022* (2025) (cit. on p. 57).

[55] Craig Gutterman et al. "Requet: Real-Time QoE Metric Detection for Encrypted YouTube Traffic." In: *ACM Trans. Multimedia Comput. Commun. Appl.* 16.2s (July 2020). ISSN: 1551-6857. DOI: `10.1145/3394498`. URL: `https://doi.org/10.1145/3394498` (cit. on pp. 12, 33).

[56] Paul E. Hoffman et al. *DNS Queries over HTTPS (DoH)*. RFC 8484. Oct. 2018. DOI: `10.17487/RFC8484`. URL: `https://www.rfc-editor.org/info/rfc8484` (cit. on pp. 1, 9, 59).

[57] Høiland-Jørgensen et al. "The eXpress data path: fast programmable packet processing in the operating system kernel." In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. 2018 (cit. on p. 17).

[58] Jordan Holland et al. "New Directions in Automated Traffic Analysis." In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 3366–3383. ISBN: 9781450384544. DOI: `10.1145/3460120.3484758`. URL: `https://doi.org/10.1145/3460120.3484758` (cit. on pp. 13, 20, 22, 46, 57, 58, 63).

[59] Zi Hu et al. *Specification for DNS over Transport Layer Security (TLS)*. RFC 7858. May 2016. DOI: `10.17487/RFC7858`. URL: `https://www.rfc-editor.org/info/rfc7858` (cit. on pp. 1, 9, 59).

[60] Johann Hugon et al. *Cruise Control: Dynamic Model Selection for ML-Based Network Traffic Analysis*. 2024. arXiv: `2412.15146 [cs.NI]` (cit. on p. 31).

[61] Johann Hugon et al. "The Cost of Packet Loss on ML-Based Traffic Analysis." In: *2025 IEEE 31th International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 2025 (cit. on p. 17).

[62] Christian Huitema. *Issues and Requirements for Server Name Identification (SNI) Encryption in TLS*. RFC 8744. July 2020. DOI: 10.17487/RFC8744. URL: https://www.rfc-editor.org/info/rfc8744 (cit. on pp. 1, 57).

[63] Christian Huitema et al. *DNS over Dedicated QUIC Connections*. RFC 9250. May 2022. DOI: 10.17487/RFC9250. URL: https://www.rfc-editor.org/info/rfc9250 (cit. on pp. 1, 9, 59).

[64] Sagar Imambi et al. "PyTorch." In: *Programming with TensorFlow: solution for edge computing applications* (2021), pp. 87–104 (cit. on p. 62).

[65] Syed Usman Jafri et al. "Leo: Online {ML-based} Traffic Classification at {Multi-Terabit} Line Rate." In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 1573–1591 (cit. on pp. 57, 58).

[66] Beomyeol Jeon et al. "A House United Within Itself: SLO-Awareness for On-Premises Containerized ML Inference Clusters via Faro." In: *Proceedings of the Twentieth European Conference on Computer Systems*. EuroSys '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 524–540. ISBN: 9798400711961. DOI: 10.1145/3689031.3696071. URL: https://doi.org/10.1145/3689031.3696071 (cit. on p. 55).

[67] Junchen Jiang et al. "{CFA}: A practical prediction system for video {QoE} optimization." In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 2016, pp. 137–150 (cit. on p. 57).

[68] Xi Jiang et al. *AC-DC: Adaptive Ensemble Classification for Network Traffic Identification*. 2023. arXiv: 2302.11718 [cs.NI]. URL: https://arxiv.org/abs/2302.11718 (cit. on pp. 31, 32, 34, 41, 55, 57, 58, 60, 72).

[69] Stefan Schmid Jonas Köppeler Toke Høiland-Jørgensen. "mq-cake: Scaling software rate limiting across CPU cores." In: *2025 IEEE 31th International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 2025 (cit. on p. 14).

[70] Parikshit Juluri et al. "Measurement of Quality of Experience of Video-on-Demand Services: A Survey." In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 401–418. DOI: 10.1109/COMST.2015.2401424 (cit. on p. 9).

[71] Piotr Jurkiewicz et al. "Flow length and size distributions in campus Internet traffic." In: *Computer Communications* 167 (2021), pp. 15–30 (cit. on p. 65).

[72] Marios Evangelos Kanakis et al. "Machine Learning for Computer Systems and Networking: A Survey." In: *ACM Comput. Surv.* 55.4 (Nov. 2022). ISSN: 0360-0300. DOI: 10.1145/3523057. URL: https://doi.org/10.1145/3523057 (cit. on p. 11).

[73] Thomas Karagiannis et al. “BLINC: multilevel traffic classification in the dark.” In: *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’05. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2005, pp. 229–240. ISBN: 1595930094 (cit. on p. 11).

[74] Ansam Khraisat et al. “Survey of intrusion detection systems: techniques, datasets and challenges.” In: *Cybersecurity* 2 (Dec. 2019). DOI: 10.1186/s42400-019-0038-7 (cit. on pp. 31, 57).

[75] Hyunchul Kim et al. “Internet traffic classification demystified: myths, caveats, and the best practices.” In: *Proceedings of the 2008 ACM CoNEXT Conference*. CoNEXT ’08. Madrid, Spain: Association for Computing Machinery, 2008. ISBN: 9781605582108 (cit. on p. 8).

[76] Georgios Kougioumtzidis et al. “A survey on multimedia services QoE assessment and machine learning-based prediction.” In: *Ieee Access* 10 (2022), pp. 19507–19538 (cit. on p. 12).

[77] Oliver Kramer et al. “Scikit-learn.” In: *Machine learning for evolution strategies* (2016), pp. 45–53 (cit. on p. 62).

[78] Youjie Li et al. “Accelerating distributed reinforcement learning with in-switch computing.” In: *Proceedings of the 46th International Symposium on Computer Architecture*. 2019 (cit. on p. 17).

[79] Hongyu Liu et al. “Machine Learning and Deep Learning Methods for Intrusion Detection Systems: A Survey.” en. In: *Applied Sciences* 9.20 (Jan. 2019). Number: 20 Publisher: Multidisciplinary Digital Publishing Institute, p. 4396. ISSN: 2076-3417. DOI: 10.3390/app9204396. URL: https://www.mdpi.com/2076-3417/9/20/4396 (visited on 03/22/2023) (cit. on p. 31).

[80] Shinan Liu et al. “Amir: Active multimodal interaction recognition from video and network traffic in connected environments.” In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 7.1 (2023), pp. 1–26 (cit. on pp. 57, 58, 67).

[81] Shinan Liu et al. *ServeFlow: A Fast-Slow Model Architecture for Network Traffic Analysis*. 2024. arXiv: 2402.03694 [cs.NI]. URL: https://arxiv.org/abs/2402.03694 (cit. on pp. 13, 17, 45, 55, 57, 58, 60, 68).

[82] Jan Luxemburk et al. “Encrypted traffic classification: the QUIC case.” In: *2023 7th Network Traffic Measurement and Analysis Conference (TMA)*. 2023, pp. 1–10. DOI: 10.23919/TMA58422.2023.10199052 (cit. on p. 59).

[83] Minzhao Lyu et al. “A Survey on DNS Encryption: Current Development, Malware Misuse, and Inference Techniques.” In: *ACM Comput. Surv.* 55.8 (Dec. 2022). ISSN: 0360-0300 (cit. on p. 9).

[84] Reham Taher El-Maghraby et al. "A survey on deep packet inspection." In: *2017 12th International Conference on Computer Engineering and Systems (ICCES)*. 2017, pp. 188–197 (cit. on p. 8).

[85] Aniss Maghsoudlou et al. "Flowdns: correlating netflow and dns streams at scale." In: *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies*. 2022, pp. 187–195 (cit. on p. 59).

[86] Tarun Mangla et al. "eMIMIC: Estimating HTTP-Based Video QoE Metrics from Encrypted Network Traffic." In: *Network Traffic Measurement and Analysis Conference (TMA)*. 2018 (cit. on pp. 31, 33, 35, 57).

[87] Tarun Mangla et al. "MIMIC: Using passive network measurements to estimate HTTP-based adaptive video QoE metrics." In: *2017 Network Traffic Measurement and Analysis Conference (TMA)*. 2017, pp. 1–6. DOI: `10.23919/TMA.2017.8002920` (cit. on p. 10).

[88] Tarun Mangla et al. "Using session modeling to estimate HTTP-based video QoE metrics from encrypted network traffic." In: *IEEE Transactions on Network and Service Management* 16.3 (2019), pp. 1086–1099 (cit. on pp. 31, 35).

[89] Lara Mauri et al. "Estimating Degradation of Machine Learning Data Assets." In: *J. Data and Information Quality* (2021) (cit. on p. 19).

[90] M Hammad Mazhar et al. "Real-time video quality of experience monitoring for https and quic." In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE. 2018, pp. 1331–1339 (cit. on pp. 12, 33).

[91] Steven McCanne et al. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." In: *USENIX winter*. Vol. 46. Citeseer. 1993, pp. 259–270 (cit. on pp. 64, 69).

[92] *Moka, A fast and concurrent cache library inspired by Java Caffeine*. https://crates.io/crates/moka. 2021 (cit. on p. 66).

[93] Andrew W. Moore et al. "Toward the Accurate Identification of Network Applications." In: *Passive and Active Network Measurement*. Ed. by Constantinos Dovrolis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 41–54. ISBN: 978-3-540-31966-5 (cit. on p. 8).

[94] M. Sajid Mushtaq et al. "Empirical study based on machine learning approach to assess the QoS/QoE correlation." In: *2012 17th European Conference on Networks and Optical Communications*. 2012, pp. 1–7. DOI: `10.1109/NOC.2012.6249939` (cit. on p. 33).

[95] Raza Ul Mustafa et al. "EFFECTOR: DASH QoE and QoS Evaluation Framework For EnCrypTed videO tRaffic." In: *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*. 2023, pp. 1–8. DOI: `10.1109/NOMS56928.2023.10154448` (cit. on p. 33).

[96] David Naylor et al. “The cost of the" s" in https.” In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 2014, pp. 133–140 (cit. on pp. 1, 57, 59).

[97] Netdev. *Segmentation Offloads in the Linux Networking Stack*. https://www.kernel.org/doc/Documentation/networking/segmentation-offloads.txt. [Accessed 13-08-2025] (cit. on p. 14).

[98] Thuy T.T. Nguyen et al. “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities.” In: *IEEE Communications Surveys & Tutorials*. 2008 (cit. on p. 31).

[99] Thuy T.T. Nguyen et al. “A survey of techniques for internet traffic classification using machine learning.” In: *IEEE Communications Surveys & Tutorials* 10.4 (2008), pp. 56–76 (cit. on p. 11).

[100] Irena Orsolic et al. “A machine learning approach to classifying YouTube QoE based on encrypted network traffic.” In: *Multimedia tools and applications* 76.21 (2017), pp. 22267–22301 (cit. on p. 10).

[101] Eva Papadogiannaki et al. “A Survey on Encrypted Network Traffic Analysis Applications, Techniques, and Countermeasures.” In: *ACM Comput. Surv.* 54.6 (July 2021). ISSN: 0360-0300 (cit. on pp. 33, 60).

[102] Ricardo Parizotto et al. “Offloading Machine Learning to Programmable Data Planes: A Systematic Survey.” In: *ACM Comput. Surv.* 56.1 (Aug. 2023). ISSN: 0360-0300. DOI: 10.1145/3605153. URL: https://doi.org/10.1145/3605153 (cit. on pp. 17, 55).

[103] Vern Paxson. “Bro: a system for detecting network intruders in real-time.” In: *Computer Networks* 31.23 (1999), pp. 2435–2463. ISSN: 1389-1286.

[104] Julien Piet et al. “GGFAST: Automating Generation of Flexible Network Traffic Classifiers.” In: *Proceedings of the ACM SIGCOMM 2023 Conference*. Association for Computing Machinery, 2023 (cit. on pp. 31, 32, 34, 45, 58, 67).

[105] J. Reynolds et al. *Assigned Numbers*. RFC 1340. July 1992 (cit. on p. 8).

[106] Shahbaz Rezaei et al. “Deep Learning for Encrypted Traffic Classification: An Overview.” In: *IEEE Communications Magazine* 57.5 (2019), pp. 76–81. DOI: 10.1109/MCOM.2019.1800819 (cit. on pp. 12, 31).

[107] M. A. Ridwan et al. “Applications of Machine Learning in Networking: A Survey of Current Issues and Future Challenges.” In: *IEEE Access* (2021).

[108] Martin Roesch. “Snort - Lightweight Intrusion Detection for Networks.” In: *Proceedings of the 13th USENIX Conference on System Administration*. LISA ’99. Seattle, Washington: USENIX Association, 1999, pp. 229–238 (cit. on p. 61).

[109] Francisco Romero et al. "INFaaS: Automated Model-less Inference Serving." In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 397–411. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/romero (cit. on pp. 32, 34, 55).

[110] Kurt Rosenthal. *The 2025 Global Internet Phenomena Report*. en. Tech. rep. Accessed: 2025-8-8. Mar. 2025. URL: https://www.applogicnetworks.com/blog/the-2025-global-internet-phenomena-report (cit. on p. 10).

[111] Hugo Sadok et al. "Enso: A Streaming Interface for NIC-Application Communication." In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pp. 1005–1025. ISBN: 978-1-939133-34-2. URL: https://www.usenix.org/conference/osdi23/presentation/sadok (cit. on p. 14).

[112] Hugo Sadok et al. "Enso: A Streaming Interface for NIC-Application Communication." In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pp. 1005–1025. ISBN: 978-1-939133-34-2. URL: https://www.usenix.org/conference/osdi23/presentation/sadok (cit. on p. 54).

[113] Mehran Salmani et al. "Reconciling High Accuracy, Cost-Efficiency, and Low Latency of Inference Serving Systems." In: *Proceedings of the 3rd Workshop on Machine Learning and Systems*. EuroMLSys '23. Rome, Italy: Association for Computing Machinery, 2023, pp. 78–86. ISBN: 9798400700842. DOI: 10.1145/3578356.3592578. URL: https://doi.org/10.1145/3578356.3592578 (cit. on p. 55).

[114] Raimund Schatz et al. "From Packets to People: Quality of Experience as a New Measurement Challenge." In: *Data Traffic Monitoring and Analysis: From Measurement, Classification, and Anomaly Detection to Quality of Experience*. Ed. by Ernst Biersack et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 219–263. ISBN: 978-3-642-36784-7. DOI: 10.1007/978-3-642-36784-7_10. URL: https://doi.org/10.1007/978-3-642-36784-7_10 (cit. on p. 9).

[115] Satadal Sengupta et al. "Exploiting Diversity in Android TLS Implementations for Mobile App Traffic Classification." In: *The World Wide Web Conference*. Association for Computing Machinery, 2019 (cit. on p. 20).

[116] Tal Shapira et al. "FlowPic: A Generic Representation for Encrypted Traffic Classification and Applications Identification." In: *IEEE Transactions on Network and Service Management*. 2021 (cit. on pp. 13, 31, 45, 46, 63).

[117] Taveesh Sharma et al. "Estimating WebRTC Video QoE Metrics Without Using Application Headers." In: *ACM SIGCOMM Internet Measurement Conference (IMC)*. Montreal, Canada, Oct. 2023, pp. 1–12 (cit. on pp. 31, 35, 57).

[118] Wazen M Shbair et al. "Improving sni-based https security monitoring." In: *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2016, pp. 72–77 (cit. on pp. 57, 59).

[119] Wazen M. Shbair et al. "Efficiently bypassing SNI-based HTTPS filtering." In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2015, pp. 990–995 (cit. on pp. 9, 57, 59).

[120] Wazen M. Shbair et al. *Early Identification of Services in HTTPS Traffic*. 2020. arXiv: 2008.08350 [cs.CR]. URL: https://arxiv.org/abs/2008.08350 (cit. on p. 60).

[121] Meng Shen et al. "DeepQoE: Real-time measurement of video QoE from encrypted traffic with deep learning." In: *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE. 2020, pp. 1–10 (cit. on p. 12).

[122] Meng Shen et al. "Machine Learning-Powered Encrypted Network Traffic Analysis: A Comprehensive Survey." In: *IEEE Communications Surveys & Tutorials* 25.1 (2023), pp. 791–824. DOI: 10.1109/COMST.2022.3208196 (cit. on pp. 11, 12, 33, 64).

[123] Jayveer Singh et al. "A survey on machine learning techniques for intrusion detection systems." In: *International Journal of Advanced Research in Computer and Communication Engineering*. 2013 (cit. on p. 31).

[124] Lea Skorin-Kapov et al. "A Survey of Emerging Concepts and Challenges for QoE Management of Multimedia Services." In: *ACM Trans. Multimedia Comput. Commun. Appl.* 14.2s (May 2018). ISSN: 1551-6857. DOI: 10.1145/3176648. URL: https://doi.org/10.1145/3176648 (cit. on p. 9).

[125] *Suricata, Observe. Protect. Adapt.* https://suricata.io/. 2023.

[126] Tushar Swamy et al. "Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2023 (cit. on p. 17).

[127] *Tcpdump, and libpcap*. https://www.tcpdump.org/. 2023 (cit. on p. 17).

[128] Brian Trammell et al. "An introduction to IP flow information export (IPFIX)." In: *IEEE Communications Magazine* 49.4 (2011), pp. 89–95 (cit. on p. 2).

[129] Martino Trevisan et al. "Impact of Access Speed on Adaptive Video Streaming Quality: A Passive Perspective." In: *Proceedings of the 2016 Workshop on QoE-Based Analysis and Management of Data Communication Networks*. Internet-QoE '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 7–12. ISBN: 9781450344258. DOI: 10.1145/2940136.2940139. URL: https://doi.org/10.1145/2940136.2940139 (cit. on p. 10).

[130] Martino Trevisan et al. "Towards web service classification using addresses and DNS." In: *2016 International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE. 2016, pp. 38–43 (cit. on p. 59).

[131] *TRex, Realistic Traffic Generator*. https://trex-tgn.cisco.com. 2023 (cit. on pp. 35, 46).

[132] Aad Van Moorsel. "Metrics for the internet age: Quality of experience and quality of business." In: *Fifth International Workshop on Performability Modeling of Computer and Communication Systems, Arbeitsberichte des Instituts für Informatik, Universität Erlangen-Nürnberg, Germany*. Vol. 34. 13. Citeseer. 2001, pp. 26–31 (cit. on p. 10).

[133] Julio Costella Vicenzi et al. "Adaptive Inference on Reconfigurable SmartNICs for Traffic Classification." In: *Advanced Information Networking and Applications*. Ed. by Leonard Barolli. Cham: Springer International Publishing, 2023, pp. 137–148. ISBN: 978-3-031-28451-9 (cit. on pp. 13, 14, 40, 55, 72).

[134] Gerry Wan et al. "Retina: analyzing 100GbE traffic on commodity hardware." In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM '22. Amsterdam, Netherlands: Association for Computing Machinery, 2022, pp. 530–544. ISBN: 9781450394208 (cit. on pp. 13, 35, 44, 47, 54, 57, 69).

[135] Gerry Wan et al. "CATO: End-to-End Optimization of ML-Based Traffic Analysis Pipelines." In: *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. Philadelphia, PA: USENIX Association, Apr. 2025, pp. 1523–1540. ISBN: 978-1-939133-46-5 (cit. on pp. 17, 31, 32, 34, 38, 41, 47, 55, 57, 58, 60, 64).

[136] Pan Wang et al. "A survey of techniques for mobile service encrypted traffic classification using deep learning." In: *Ieee Access* 7 (2019), pp. 54024–54033 (cit. on p. 12).

[137] Duo Wu et al. "Mansy: Generalizing neural adaptive immersive video streaming with ensemble and representation learning." In: *IEEE Transactions on Mobile Computing* (2024) (cit. on p. 12).

[138] Xiaoban Wu et al. "Network measurement for 100 GbE network links using multicore processors." In: *Future Gener. Comput. Syst.* 79.P1 (Feb. 2018), pp. 180–189. ISSN: 0167-739X. DOI: 10.1016/j.future.2017.04.038. URL: https://doi.org/10.1016/j.future.2017.04.038 (cit. on p. 44).

[139] Zhaoqi Xiong et al. "Do Switches Dream of Machine Learning? Toward In-Network Classification." In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. HotNets '19. Princeton, NJ, USA: Association for Computing Machinery, 2019, pp. 25–33. ISBN: 9781450370202. DOI: 10.1145/3365609.3365864. URL: https://doi.org/10.1145/3365609.3365864 (cit. on pp. 14, 55).

[140] Chengcheng Xu et al. “A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms.” In: *IEEE Communications Surveys & Tutorials* 18.4 (2016), pp. 2991–3029 (cit. on p. 8).

[141] Kazuhisa Yamagishi et al. “Parametric Quality-Estimation Model for Adaptive-Bitrate-Streaming Services.” In: *IEEE Transactions on Multimedia* 19.7 (2017), pp. 1545–1557. DOI: 10.1109/TMM.2017.2669859 (cit. on p. 10).

[142] Baris Yamansavascilar et al. “Application identification via network traffic classification.” In: *2017 International Conference on Computing, Networking and Communications (ICNC)*. 2017, pp. 843–848. DOI: 10.1109/ICCNC.2017.7876241 (cit. on p. 45).

[143] Chen Yang et al. “Anti-Packet-Loss Encrypted Traffic Classification via Masked Autoencoder.” In: *Wireless Artificial Intelligent Computing Systems and Applications*. 2025 (cit. on p. 19).

[144] Jeff (Jun) Zhang et al. “Model-switching: dealing with fluctuating workloads in machine-learning-as-a-service systems.” In: *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’20. USA: USENIX Association, 2020 (cit. on p. 55).

[145] Qizheng Zhang et al. “CARAVAN: practical online learning of in-network ML models with labeling agents.” In: *Proceedings of the 3rd Workshop on Practical Adoption Challenges of ML for Systems*. 2024, pp. 17–20 (cit. on pp. 57, 58).

[146] Yuqi Zhao et al. “The Sweet Danger of Sugar: Debunking Representation Learning for Encrypted Traffic Classification.” In: *arXiv preprint arXiv:2507.16438* (2025) (cit. on p. 12).

[147] Zhipeng Zhao et al. “Achieving 100Gbps Intrusion Prevention on a Single Server.” In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1083–1100. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng (cit. on p. 33).

[148] Jiuxing Zhou et al. “Challenges and Advances in Analyzing TLS 1.3-Encrypted Traffic: A Comprehensive Survey.” In: *Electronics* 13.20 (2024). ISSN: 2079-9292. DOI: 10.3390/electronics13204000. URL: https://www.mdpi.com/2079-9292/13/20/4000 (cit. on p. 33).