Wixor: Dynamic TDD Policy Adaptation for 5G/xG Networks

AHMAD HASSAN[*], University of Southern California, USA
SHIVANG AGGARWAL, Hewlett Packard Labs, USA
MOHAMED IBRAHIM, Hewlett Packard Labs, USA
PUNEET SHARMA, Hewlett Packard Labs, USA
FENG QIAN, University of Southern California, USA

In the era of 5G and beyond, dynamic Time Division Duplex (TDD) has become essential for supporting applications that demand high bandwidth and low latency. Emerging uplink-intensive use cases such as real-time video analytics, autonomous vehicles and augmented reality further complicate the balance between uplink and downlink resources. Despite their potential, TDD policies employed by current 5G networks remain underexplored. Our investigation reveals that existing TDD policies are static and predominantly downlink-focused, failing to adapt to fluctuating network demands. We introduce Wixor, a robust dynamic TDD policy adaptation system tailored for 5G and next-generation (xG) networks. It proactively adjusts the allocation of TDD resources between uplink and downlink, addressing various practical challenges. Prototyped on a programmable testbed, Wixor demonstrates substantial performance improvements across diverse applications, achieving up to 96.5% enhancement in Quality of Experience (QoE) compared to existing baselines.

CCS Concepts: • **Networks** → **Mobile networks**; *Cross-layer protocols*; Network measurement.

Additional Key Words and Phrases: 5G, TDD, Resource Assignment, TDD Scheduling, TDD Pattern

ACM Reference Format:
Ahmad Hassan, Shivang Aggarwal, Mohamed Ibrahim, Puneet Sharma, and Feng Qian. 2024. **Wixor**: Dynamic TDD Policy Adaptation for 5G/xG Networks. *Proc. ACM Netw.* 2, CoNEXT4, Article 38 (December 2024), 24 pages. https://doi.org/10.1145/3696395

1 Introduction

5G NR heralds a new era of connectivity, promising unprecedented speeds and ultra-low latency. These advancements are crucial for a wide range of applications, from immersive augmented reality (AR) experiences [26, 28] and autonomous vehicles [51] to critical healthcare services [75] and real-time video analytics [23, 53]. To meet the performance requirements of these *emerging* use cases, more than 80% of the 5G operators have turned to Time Division Duplex (TDD) [11], whereas previous technologies (e.g., LTE, 3G) mainly employed Frequency Division Duplex (FDD) [10]. TDD alternates uplink (UL) and downlink (DL) transmissions within the same frequency band using time slots to enable flexible spectrum utilization and dynamic UL/DL resource allocation. To accommodate different traffic patterns, 5G NR introduces *dynamic* TDD, where the base station (BS) can dynamically change the distribution of UL and DL time slots, given a TDD policy [20].

[*]Ahmad Hassan did this work as an intern at Hewlett Packard Labs.

Authors' Contact Information: Ahmad Hassan, University of Southern California, USA, ahmadhas@usc.edu; Shivang Aggarwal, Hewlett Packard Labs, California, USA, shivang.aggarwal@hpe.com; Mohamed Ibrahim, Hewlett Packard Labs, New Jersey, USA, ibrahim@hpe.com; Puneet Sharma, Hewlett Packard Labs, California, USA, puneet.sharma@hpe.com; Feng Qian, University of Southern California, USA, fengqian@usc.edu.

ACM 2834-5509/2024/12-ART38
https://doi.org/10.1145/3696395

Although the 3GPP specifications cover the mechanism for dynamic TDD, they leave the actual TDD policy implementation open for network operators.

While conceptually reasonable, it remains unclear whether dynamic TDD can fulfill its prospects in practice. To our knowledge, existing solutions focus on naive theoretical or simulation-driven analysis [66, 67, 85] and niche prototype implementation [27]. They usually ignore the impact of dynamic TDD on real-world applications' Quality of Experience (QoE). Moreover, these studies fail to consider the practical challenges in designing a dynamic TDD system capable of supporting a large number of users, diverse channel conditions, and varying application workloads.

To this end, we conduct a *timely* measurement study of TDD policies employed by public 5G networks and show their impact on the QoE of emerging applications (§3). We find that the BS traffic load fluctuates rapidly and traditional "static" TDD policies fall short in adapting to it. This leads to suboptimal performance, particularly for latency-sensitive and UL-intensive applications. Additionally, our investigation reveals that straightforward TDD policies (e.g., a reactive policy that uses the past traffic demand information to adjust UL and DL slot distribution) result in lost performance compared to a proactive approach. Lastly, our experiments demonstrate that even the arrangement of UL and DL time slots in a TDD policy has a non-trivial impact on the application QoE.

The goal of this work is to design a TDD policy that dynamically adjusts the distribution and arrangement of UL and DL time slots for application QoE improvement without any QoE feedback from the user entity (UE) or application server. This problem confronts several challenges. *First*, the flexibility in defining TDD policies requires exploring numerous UL and DL slot arrangements, making problem complexity a concern. *Second*, the rapidly fluctuating traffic load and channel conditions must be taken into consideration. *Third*, limited information about application QoE goals implies the lack of a well-defined optimization objective. *Furthermore*, frequent TDD policy adjustments can interfere with transport-layer congestion control or application-layer rate adaptation logic. *Lastly*, the inherent asymmetry between UL and DL transmission (§3.3), with UL typically experiencing higher latency and lower throughput, further complicates optimization.

To address the above challenges, we present Wixor, a practical TDD policy adjustment system for 5G (xG) radio access network (RAN). Our system performs TDD policy adjustment at the last mile BSs including private 5G deployments. Wixor reduces the problem complexity by breaking it down into two parts. *First*, it predicts the UL and DL slot distribution through a **proactive demand customization engine (§5)**. Wixor employs a learning-based approach in combination with BS-level features to handle the highly complex environment and manage the asymmetry between UL and DL transmissions. *Second,* Wixor finds the *best* arrangement of UL and DL slots given the slot distribution via a **smart policy provision framework (§6)**. Due to the lack of QoE information, our solution improves application performance indirectly by optimizing the radio protocol layer Quality of Service (QoS) metrics. Further, it leverages a "conservative" TDD policy smoothing technique to avoid abrupt policy changes, thus minimizing the interference with rate adaptation modules. Our system is particularly useful for emerging private 5G (xG) applications with stringent bandwidth and latency requirements – it offers configurable *knobs* to fine-tune UL versus DL priority, and strike a balance between network bandwidth and latency.

We prototype Wixor on a programmable testbed with an open-source 5G cellular suite [16] in total 2.3K+ lines of code. Wixor is fully 3GPP-compliant, making it readily deployable for any public or private 5G operator. We carry out comprehensive evaluations in different settings (over-the-air testbed, trace-driven simulations), with diverse channel traces (driving, walking, etc.), using real application workloads (edge video analytics, live video conferencing, etc.), and several baselines (e.g., a recently proposed dynamic TDD system [27]). We highlight experimental results as follows.
• We use a suite of six diverse apps to quantify QoE gains. Compared to baselines, Wixor improves application QoE metrics by 2.5%-96.5%, or is within 91.6% of the best scheme (§8.2 & §8.3).

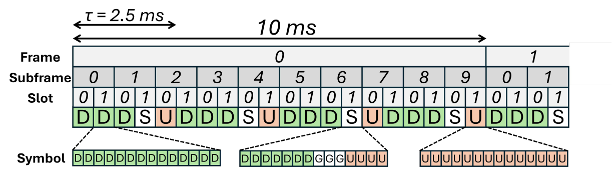

Fig. 1. 5G frame structure and TDD pattern for numerology μ=1 and transmission periodicity τ=2.5 ms. Each TDD pattern has 5 slots. S slots may contain DL, UL, and guard symbols.

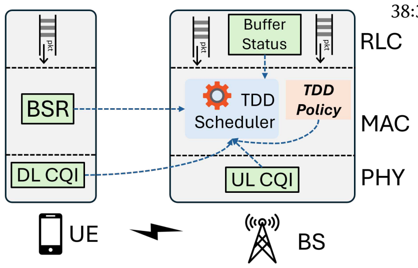

Fig. 2. Radio resource scheduling overview.

- Wixor is particularly useful under changing traffic loads and fluctuating channel conditions. For example, compared to the walking scenario, it offers 3.2% and 15.3% higher average throughput and latency improvement for driving (§8.4).
- Wixor respects the radio network's optimization objectives, is lightweight and scalable, works well under advanced BS configurations, and operates close to the optimal solution (§8.4 & §8.5).

2 Background & Related Work

2.1 A Primer on 5G

5G Frame Structure. Unlike 4G LTE, 5G New Radio (NR) has a flexible *numerology* and frame structure. In 5G, numerology ($\mu \in [0, 4]$) enables various subcarrier spacings to meet different service requirements. As illustrated in Fig. 1, the frame structure is hierarchical: a 10 ms radio frame contains 10 sub-frames (1 ms each), and sub-frames are divided into 2^{μ} slots, with each slot lasting $2^{-\mu}$ ms. Each slot typically contains 14 OFDM symbols with a normal cyclic prefix.

5G NR TDD. TDD allows UL and DL transmissions to share the same frequency band but separated in time. In the US, 5G operators use TDD in Mid-Band (1-6GHz) and mmWave (24-40GHz) frequencies [6]. Our work focuses on Mid-Band TDD, common in private 5G deployments [3], though it can be applied to mmWave as well. 5G NR defines three types of TDD scheduling: (i) static TDD with fixed UL/DL time slots; (ii) semi-static TDD, which adapts to traffic patterns using higher-layer Radio Resource Control (RRC) signaling; and (iii) *dynamic TDD*, the focus of our work, which dynamically allocates UL and DL slots based on factors such as real-time traffic load. This dynamic TDD approach is generic, encompassing the static and semi-static methods as special cases.

TDD Policy. A TDD pattern defines the allocation of time slots and symbols (S&S) for UL and DL transmissions within a radio frame, as shown in Fig. 1. Transmission periodicity τ refers to the repetition interval of a TDD pattern, typically in milliseconds (ms). Depending on μ, each TDD pattern consists of $T_s = 2^{\mu} \cdot \tau$ slots. Note that τ, and thus, T_s can change over time for a BS. A crucial element in TDD is the *guard period*, which prevents interference between UL and DL transmissions and accounts for propagation delays [29]. In 5G TDD nomenclature, a *"TDD policy"* refers to the arrangement of S&Ss within a TDD pattern. The BS uses these S&S for UL/DL (data and control) transmissions and guard periods. *This paper aims to find an arrangement of UL and DL S&Ss (TDD policy) that optimizes our objectives.* While 3GPP specifications provide signaling mechanisms to inform UEs about S&S allocation, the actual TDD policy algorithm remains open-ended.

TDD Resource Scheduling. Fig. 2 provides an overview of the TDD resource scheduling procedure. For the DL channel, UEs measure Signal-to-Interference-plus-Noise Ratio (SINR) and report Channel Quality Indicator (CQI) values to the BS, while the BS directly measures each UE's UL CQI. DL data arrives in the Radio Link Control (RLC) per-UE queues, and UEs send periodic Buffer Status Reports (BSRs) to inform the BS about the remaining UL data. Based on the TDD policy, CQI, and outstanding data in UL and DL queues, the TDD MAC scheduler allocates OFDM symbols among active UEs. Additionally, the scheduler uses CQI to determine the modulation and coding scheme (MCS) which then determines the Transport Block Size (TBS) or data rate of the UE.

Traffic load versus BS throughput. The literature defines a BS's *traffic load* in various ways, such as the number of active UEs connected to the BS [74], the utilization of available radio resources [59], or *the total volume of UL and DL data waiting to be processed* [56]. We adopt the last definition as it represents the actual data the BS needs to process. The *BS throughput*, however, refers to the amount of data being transmitted (DL) and received (UL) by the BS at any given time.

2.2 Related Work

Dynamic TDD Schemes. Several past works propose dynamic TDD schemes for 5G/LTE networks [27, 31, 32, 35, 63, 69, 72, 73, 82]. DRP [27], a deep reinforcement learning-based scheme is the closest to our work. DRP uses BSRs and DL buffer size to derive the optimal TDD pattern. However, DRP has limited practical applicability due to its lack of consideration of (i) the space-time dynamics of a wireless channel, (ii) diverse QoS/QoE requirements for different users, (iii) scalability issues as it uses per-UE features, and (iv) other practical aspects such as S&S arrangement and guard periods. Wixor, in contrast, considers all these practical concerns in its design. We experimentally compare Wixor with DRP in §8. Other works focus on specific scenarios such as high mobility HetNets [72], high density environments [73], massive IoT networks [63], small cell networks [35, 82], as well as leveraging device-to-device communication [69], whereas Wixor takes a more generic approach, i.e., public and private 5G deployments.

5G/LTE resource scheduling. As discussed in §2.1, based on the TDD policy, the MAC scheduler distributes the UL/DL resources amongst the UEs. The MAC layer scheduling problem has been studied extensively in literature [22, 33, 34, 55, 81]. RadioSaber [34] and iRSS [81] solve it in the context of network slicing. SMART [22] tackles the problem for massive MIMO networks, while ELASE [33] and UQ-vRAN [55] focus on virtual RAN-based 5G networks. Our work, however, solely focuses on the dynamic TDD adaptation problem, which is orthogonal to, and can work in conjunction with, the MAC scheduling works described above.

Application-specific optimizations in cellular networks. Recently, a large body of work has focused on improving the performance of specific applications over 5G/LTE/WiFi [25, 47, 58, 65, 71, 79]. For example, DChannel [65], Tutti [79], LRP [71], and Zhuge [58] boost the performance of latency-critical applications such as video analytics, video conferencing, and cloud gaming over 5G/WiFi networks. These works rely on obtaining QoE information directly from applications, while Wixor leverages the readily available radio protocol layer QoS metrics as *indirect* indicators of application performance. Simply put, Wixor does not communicate with applications, making it application-transparent. Furthermore, Wixor does not focus on any single application and is designed to simultaneously improve the performance of various types of applications that may have vastly different QoS objectives in terms of throughput, latency, etc.

3 Motivation & Challenges

3.1 Experiment Setup

Live 5G experiments. To *characterize* TDD polices employed by today's 5G operators, we set up a live 5G testing platform. Specifically, our setup employs the NG-Scope tool [78] to decode a BS's control channel information, e.g., TBS and MCS of all UEs connected to the BS. Simultaneously, we utilize a Samsung Galaxy S22+ smartphone, connected to the same BS, to collect lower-layer TDD configuration data with Accuver XCAL tool [19]. Overall, we collect 26 hrs+ of data, at different hours of the day, with T-Mobile (Band 41 @ 2500 MHz), Verizon (Band 77 @ 3700), and AT&T (Band 77 @ 3700). We refer to the collected dataset as DL5G in the remainder of this paper.

Over-the-air 5G testbed. We build an in-lab *end-to-end 5G* network to run tests under several controlled settings, e.g., TDD polices, applications, traffic loads, etc. The testbed comprises of 2× Google Pixel 7 devices (PX7), a BS, and a 5G Core. The BS has two components: (i) an srsRAN-based eNB/gNB stack running on a laptop equipped with Intel Core i7 @ 3.00GHz CPU, and (ii) an

Table 1. Application workloads used in our paper.

App Name	UL		DL	
	Lat. Sen.	Bwd. Int.	Lat. Sen.	Bwd. Int.
Edge Video Analytics (EVA)	✓	×	✓	×
Edge-assisted Vehicle Perception (EVP)	✓	✓	✓	×
Live Video Ingest (LVI)	×	✓	×	×
Video-on-Demand (VoD)	×	×	×	✓
Live Video Conferencing (LVC)	✓	×	✓	×
HTTP File Transfer (HFT)	×	✓	×	✓

Table 2. TDD polices employed by major public 5G operators in the US. Operators employ almost identical policy parameters, assigning 72.8%–74.2% of the S&Ss to the DL.

Operator	Band	Pattern 1 (UL/DL)		Pattern 2 (UL/DL)	
		# slots	# symbols	# slots	# symbols
T-Mobile	n41	2/3	4/4	0/4	0/0
Verizon/AT&T	n77	2/3	4/6	0/4	0/0

RF frontend based on USRP B210 [18] software defined radio (SDR) configured with Band 78 @ 3410 MHz. The Open5GS Core Network (CN) [12] runs on another laptop. To conduct experiments, we put our testbed in a 20m×12m conference room and walk randomly with PX7 in hand. All applications are hosted locally with a 15 ms delay between the CN (packet gateway) and the application server. For edge applications, the delay is 2 ms between the BS and the edge server. Both servers run on separate desktop machines that have Linux kernel 6.2 and Intel Xeon CPUs with 64 GB RAM. We use ADB scripts to automate and time-synchronize experiments. All experiments are repeated at least 5×.

Trace-driven simulator. We design a *faithful* simulator based on *ns*3 5G-Lena [5] to: *(i)* carry out experiments under reproducible channel and traffic conditions; and *(ii)* accelerate our system's model training. The setup (e.g., frequency band) is identical to the over-the-air testbed. Further, we configure our simulator to utilize channel traces collected by previous 5G studies [37, 44, 61]. Overall, the traces consists of 18 hrs+ of SINR values, collected at TTI granularity. We generate 200 traces for our *corpus*, each with a duration of 300 secs. For each individual test, we randomly select n traces for n UEs from the *corpus*. Since these traces are collected in different mobility scenarios, the heterogeneity and randomization ensures that the BS has UEs with diverse channel conditions.

Background traffic. Scaling the testbed for numerous devices generating realistic application workloads is costly. Therefore, we generate UL and DL background traffic using *real-world* traffic traces, postprocessed from NG-Scope (DL5G) TBS data to match our BS configuration. Each trace contains UL/DL data from multiple UEs. To conduct an experiment, we randomly select a traffic trace to simulate UL/DL UDP traffic between the UEs and the server. Since we only have two PX7 devices in our testbed, we use the first one to send application traffic and the second one to send/receive UDP background traffic for all UEs in the traffic trace. For the simulation setup, however, we generate background traffic for the number of UEs specified in the trace.

Applications. We develop a suite of diverse (latency-sensitive and/or bandwidth-intensive in UL/DL) apps for our over-the-air testbed and trace-driven simulator. Table 1 lists these apps, while the detailed setup is outlined in §8.1.

3.2 Need for Dynamic TDD Policy Adjustment

Existing xG networks configure *static, DL-biased* TDD slot policies, incapable of adapting to changing network loads. We leverage the 26 hrs+ long dataset DL5G to characterize how frequently the traffic load changes and how the network reacts to it. Our analysis reveals four main insights: ***(i)*** Fig. 3a shows rapid BS traffic load fluctuations, highlighting the need for a dynamic TDD policy. The empirical CDF (e-CDF) of the rolled coefficient of variation (RCV[1]) for traffic load with a 1 sec window size in Fig. 3b shows abrupt load changes for T-Mobile and Verizon, with the median load increasing/decreasing 4.7-5.2 standard deviations above/below the average. ***(ii)*** Current 5G networks employ static TDD policies which are incapable of adapting to these load fluctuations. Table 2 shows that major US network operators use similar, and more importantly *static*, TDD policy parameters, regardless of the traffic load. Note that the 3GPP standards permit two TDD

[1]RCV measures the relative variability of a time-series over a specified rolling window. It is calculated as the standard deviation divided by the mean within each window, i.e., $(\sigma_{t-\delta t:t})/(\mu_{t-\delta t:t})$ High RCV values (>1.0) indicate high relative variability within the window (δt=1s), and vice-versa.

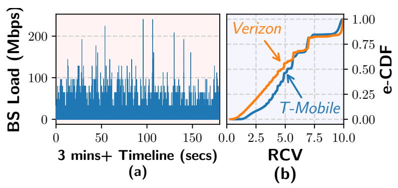

Fig. 3. The high variability in traffic load from two public 5G networks.

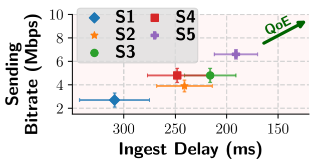

Fig. 4. Comparing the impact of static TDD policies on the live ingest application's QoE.

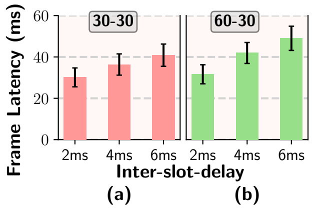

Fig. 5. Quantifying the impact of *inter-slot delay* on EVA QoE.

patterns (pattern 1 and 2) for UL and DL S&S allocation. ***(iii)*** Existing networks have *DL-biased* TDD policies. From DL5G, the median DL traffic load is 12.9× higher than UL. Networks allocate the majority of TDD S&Ss to DL, as seen with T-Mobile assigning ~72.8% to DL in Table 2. These DL-biased policies are a bottleneck for emerging xG applications that require high UL bandwidth and/or low latency. ***(iv)*** Despite predominantly DL-heavy traffic, 4.1% of instances in DL5G show higher UL traffic loads, again underscoring the need for dynamic TDD policies.

Case study: To highlight the impact of static TDD policies on application QoE, we run a live video ingest app on our over-the-air testbed. The first PX7 device runs the video ingest app, while the second one generates background traffic using 15 random UE traces from DL5G (§3.1). We test five different settings: **(S1)** UL and DL S&S assignment as per T-Mobile's configuration (22.8% UL, 72.8% DL), **(S2)** equal UL-DL S&Ss (47.8%, 47.8%), **(S3)** a "fair"[2] UL-DL assignment based on average load (67.4%, 28.2%), **(S4)** UL-dominant S&S assignment (72.8%, 22.8%), and **(S5)** our proposed dynamic TDD solution. The remaining 4.4% of S&S are reserved for the guard period in **S1-4**.

Fig. 4 shows the sending bitrate and the ingest delay for each setting. Ingest delay measures the time from frame generation to its quality variants being available for download [84]. Assigning more resources to the UL results in higher QoE (notice that **S3** > **S2** > **S1**). However, ramping up UL resources beyond a point will come at a cost of less DL resources, since the network has limited bandwidth. This will cause the DL to become a bottleneck, leading to more buffering in DL queues and longer ingest delays, as seen for **S4**. In contrast, our solution (**S5**) adapts to changing loads, achieving 37.5% higher bitrate and 11.6% lower ingest delay than the next best setting (**S3**).

A *reactive* TDD slot adjustment policy is sub-optimal. A straightforward, naive solution to the dynamic TDD policy adjustment problem is to adjust the DL and UL slot percentage based on the past traffic load. However, as shown above (Fig. 3), the traffic load changes significantly within short time periods, and TDD policy adjustment based on outdated traffic load information can potentially result in performance loss. For applications in Table 1, a reactive approach incurs significant QoE reduction compared to our proposed system (details in §8.2).

UL/DL S&S arrangement incurs additional complexity. We find that, in addition to the percentage of UL and DL S&S in TDD pattern, their arrangement can impact QoE too, especially for latency-sensitive apps. Each S&S arrangement leads to a certain average delay between two consecutive UL or DL slots (*inter-slot delay*). By keeping the background traffic rate (15 random UE traces from DL5G) and the number of UL/DL slots constant, we test the Edge Video Analytics (EVA) performance for different *inter-slot delays* (2ms, 4ms, 6ms) and traffic loads (30-30, 60-30). A 60-30 traffic load indicates a 60% UL and 30% DL traffic load. As shown in Fig. 5a, the average frame response latency is 30.1 ms for a 2 ms inter-slot delay, increasing to 40.8 ms for a 6 ms delay (a 35.5% increase). Additionally, high traffic loads cause user data to buffer, further impacting response latency (see Fig. 5b). Thus, the UL and DL S&S arrangement complicates deriving the optimal TDD policy, with possible configurations ranging from hundreds to thousands.

[2]UL needs more resources than DL to provide a same level of throughput; that is why a fair static policy will have more UL S&S than the UL traffic load ratio.

3.3 Challenges

Scalability. 5G NR enables the BS to flexibly define a TDD policy where the arrangement of S&Ss within a TDD pattern is crucial for optimizing application performance (§3.2). However, this flexibility introduces a new challenge: we must exhaustively explore all possible combinations of UL and DL S&S arrangements to identify the optimal TDD pattern, which can be computationally intensive. For example, numerology μ=1 results in over 1450 unique S&Ss arrangements.
Highly dynamic environment. Many factors render the TDD policy adjustment problem complex: **(i)** the UL and DL traffic load changes frequently and straightforward solutions (e.g., *Static* and *Reactive*) do not work well (§3.2). Additionally, traffic load is hard to predict; **(ii)** application workloads are highly diverse in a radio network ecosystem, and the BS lacks information on QoE goals and feedback from UEs or application servers; and **(iii)** channel conditions fluctuate rapidly making the problem even more complex. Channel conditions (measured through metrics such as SINR) determine the effective data transmission rate that dictates the optimal TDD policy.
Interference with rate adaptation modules. Frequently changing the TDD policy can disrupt the transport-layer congestion control or application-layer rate adaptation. To illustrate this, we use the same over-the-air testbed setup as above with a TCP sender (default Cubic congestion control [41]). The TDD policy is adjusted every second using *Oracle* (details in §8.5). Fig. 6 shows a UE's UL TCP congestion window (*cwnd*) alongside the TDD policy (% of UL TDD S&S). At the 8^{th} sec, the UL S&S % drops from 61.8% to 42.1%, causing packet buffering, a TCP timeout, and a reduction in *cwnd*. However, before the TCP sender restores *cwnd*, another drop in UL S&S leads to another reset. This creates a significant gap between the instantaneous demand and achieved throughput, as seen in the shaded region of Fig. 6, hindering rate adaptation and, ultimately, throughput.
Asymmetric UL and DL transmission. Our investigation reveals a discrepancy between UL and DL latency, in addition to throughput differences. Intuitively, one-way latency should be similar in both directions. However, over-the-air testbed experiments with equal UL and DL S&Ss show that UL is almost 40% slower than DL at low sending rates (see Fig. 7). Additionally, the UL latency inflates quickly (due to bufferbloat [39] or limited radio resources [65]) once the sending rate exceeds the link capacity. Our public 5G experiments reveal similar insights. Factors such as limited UE transmission power [36], delays from UL scheduling grants [54], lower carrier aggregation [62], and the use of SC-FDMA for power efficiency [24] contribute to less performant UL.

4 Wixor Design

To address the challenges outlined in §3.3, Wixor employs a *two-stage* approach to TDD policy adjustment. First, it predicts the UL and DL S&S distribution (percentages) based on the BS context such as traffic load and channel quality. Once the distribution is determined, Wixor finds the *best* S&S arrangement. This decomposition significantly reduces the search space compared to an exhaustive search method, evaluating only 5-25 arrangements for μ=1 (a 58-290× reduction). While this two-stage approach may incur slight performance losses if the initial prediction is inaccurate – especially when relying on fixed models that do not generalize well to complex RAN environments – Wixor mitigates this risk by employing a *learning-based* approach in combination with BS-level features. This approach effectively manages the complexity of the environment and the asymmetry between UL and DL transmission. Additionally, the system utilizes a conservative policy smoothing technique to prevent abrupt policy changes, thereby minimizing the interference with transport-layer congestion control and application-layer rate adaption logic.

The basic operation of Wixor is illustrated in Fig. 8. In the UL direction, UEs request radio resources from the BS, obtain the allocated UL resources, and transmit the data, which is then forwarded to the Internet. Conversely, in the DL direction, the incoming data arrives into the

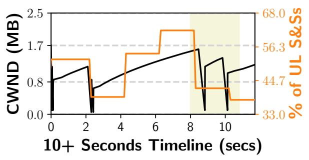

Fig. 6. Impact of frequent TDD policy updates on TCP's congestion control logic.

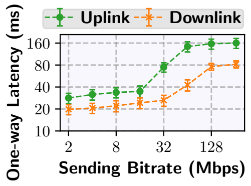

Fig. 7. Comparing one-way UL and DL latencies for testbed *(log-scale)*.

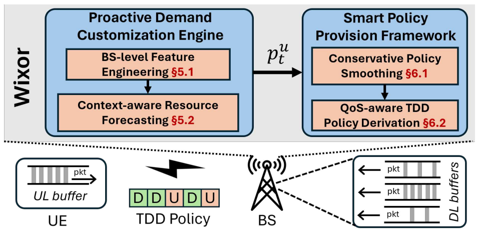
Wixor
Proactive Demand Customization Engine
BS-level Feature Engineering §5.1
Context-aware Resource Forecasting §5.2
p_t^u
Smart Policy Provision Framework
Conservative Policy Smoothing §6.1
QoS-aware TDD Policy Derivation §6.2
UL buffer
UE
D D U D U
TDD Policy
BS
DL buffers

Fig. 8. A high-level overview of Wixor.

per-UE queues, and the BS transmits it to UEs. In the above process, ensuring that the BS effectively *balances* available TDD S&Ss between UL and DL such that UEs receive *sufficient* resources *promptly* is the key to improving application QoE. Wixor operates as a lightweight service at the BS to ensure TDD policy adjustment in a timely manner.

To this end, Wixor leverages the traffic, BS load, channel quality, and QoS features (readily available at the BS) as the system inputs, and outputs the TDD pattern to guide the BS's TDD policy adjustment via two major modules: *(i)* A **proactive demand customization engine** (§5) to precisely predict future UL and DL resource demands. Specifically, it utilizes cross-layer BS-level features to capture the RAN context (§5.1). Wixor then feeds these features into the context-aware resource forecasting module (§5.2), which outputs the S&S percentage allocations for UL and DL. *(ii)* Wixor drives a **smart policy provision framework** (§6) to ensure that the TDD policy is configured reliably for application QoE improvement. It first applies conservative policy smoothing to reduce the impact of abrupt TDD policy changes on application QoE (§6.1). Then, the QoS-aware TDD policy derivation module (§6.2) computes the final arrangement of UL and DL S&S within the TDD pattern. In doing so, it judiciously balances the trade-off between the inter-slot delay (which impacts network latency) and guard period overhead (which impacts network throughput).

5 Proactive Demand Customization

Wixor first constructs BS-level features from raw BS logs (§5.1). These features are then passed to a reinforcement learning (RL) agent to forecast future traffic demand (§5.2). The RL agent employs a neural network (NN) to interpret the RAN *context*, represented by the BS-level features. Training the RL agent in a live 5G environment is impractical due to the random exploration required by RL, which would significantly impact application QoE. Therefore, we train Wixor's RL agent using a faithful simulator with real-world traffic and channel traces (§3.1). Wixor's RL agent utilizes *normalized* features to ensure seamless transferability from simulation to over-the-air setups.

5.1 Cross-layer BS-level Feature Engineering

We considered two choices for feature engineering: per-UE features and BS-level features. Per-UE features can precisely characterize the behavior of all n_t active users at time t, but they have practical issues. *First,* the dynamic nature of user presence in the RAN ecosystem causes significant variation in n_t over time. Handling this variation would require the NN to process variable-sized inputs, leading to scalability issues. *Second,* the number of NN inputs scales with n_t, affecting its learning ability (curse of dimensionality). In contrast, BS-level features aggregate per-UE features to represent overall UE behavior at the BS. They mitigate the scalability issue as the NN inputs are fixed-size regardless of n_t. However, engineering meaningful BS-level features without losing critical information is challenging. We address this by using statistical measures to aggregate per-UE features into a *comprehensive* yet practical representation of the BS state. In §7, we detail how Wixor collects raw BS logs from different radio protocol stack layers to compute these features.
(i) Traffic demand features. These features help Wixor understand the traffic demands of active users n_t. We concatenate average buffer occupancy levels B_t^u and B_t^d, maximum buffer levels M_t^u

and M_t^d, traffic arrival rates A_t^u and A_t^d, and head-of-line delays H_t^u and H_t^d at time t to create traffic demand feature vector $\overrightarrow{\mathcal{D}}_t = \{B_t^u, B_t^d, M_t^u, M_t^d, A_t^u, A_t^d, H_t^u, H_t^d\}$. For each metric, u and d represent UL and DL. B_t^u is calculated as $\sum_i b_t^{u,i}/(n_t * c)$, where $b_t^{u,i}$ is the UL buffer level for UE i and c is the RLC buffer capacity. We calculate $M_t^u = \max_i(b_t^{u,i}/c)$ as the maximum UL buffer level across all UEs. The data arrival rate A_t^u indicates how quickly data is arriving in the UL buffers. We model each UE's rate $a_t^{u,i} = \lambda_t^{u,i} * \hat{s}_t^{u,i}$ as a Poisson process, where $\lambda_t^{u,i}$ is the inter-packet arrival rate and $\hat{s}_t^{u,i}$ is the average packet size. The overall arrival rate $A_t^u = \sum_i a_t^{u,i}$ is the sum of individual UE arrival rates. We normalize A_t^u by dividing it with the maximum data arrival rate supported by the BS[3]. The head-of-line (HoL) delay $H_t^u = \sum_i h_t^{u,i}/n_t$ is the average HoL delay experienced by all UEs. The DL counterparts of these metrics in $\overrightarrow{\mathcal{D}}_t$ follow the same terminology.
(ii) BS load features. We also consider BS load features $\overrightarrow{\mathcal{L}}_t = \{T_t^u, T_t^d, R_t^u, R_t^d\}$ to capture the effect of traffic demand on BS's resources. $T_t^u = \sum_i t_t^{u,i}$ represents the UL BS throughput, where $t_t^{u,i}$ is UE i's UL throughput normalized by the maximum BS throughput. $R_t^u = \sum_i r_t^{u,i}$ is the total resource utilization, which is calculated as the sum of the normalized UL resource blocks $r_t^{u,i}$ assigned to each UE where the normalization is performed against the total number of resource blocks.
(iii) Channel quality features. Given the impact of channel conditions on network performance (§3), we incorporate channel quality indicator (CQI) information as well. However, simply averaging individual UEs' wideband CQIs $c_t^{u,i}$ does not work well in practice, as UEs encounter extremely diverse channel conditions in the real world. Therefore, to encode meaningful information about channel variation, we employ median, 25th %ile, and 75th %ile CQI values to get channel quality features $\overrightarrow{C}_t = \{\text{P-25}_i(c_t^{u,i}), \text{P-25}_i(c_t^{d,i}), \text{P-50}_i(c_t^{u,i}), \text{P-50}_i(c_t^{d,i}), \text{P-75}_i(c_t^{u,i}), \text{P-75}_i(c_t^{d,i})\}$. These CQI values are further normalized using the maximum possible CQI value in 5G (i.e., 31).
(iv) QoS features. Lastly, a buffer tolerance factor $\rho_t \in [0, 1]$ indicates BS's cumulative buffering tolerance for all UEs. A low tolerance (i.e., $\rho_t \simeq 0$) loosely represents latency-sensitive traffic.

5.2 Context-aware Resource Forecasting

We consider two options for Wixor's objectives: the application layer *QoE* metrics and the radio protocol layer *QoS* metrics. Directly optimizing QoE metrics may lead to high end-user performance, but it requires explicit QoE feedback from the applications. In contrast, although QoS metrics are generic indicators of application performance, they can be directly estimated at the BS without application support. Wixor, therefore, optimizes an objective based on three key QoS metrics: ***(O1)*** maximize the sum of UL and DL BS throughput, i.e., $\max T_t^u$ and $\max T_t^d$; ***(O2)*** minimize network latency estimated as the highest buffer occupancy level (or self-inflicted queuing delay [77]) for all UEs, i.e., $\min M_t^u$ and $\min M_t^d$; and ***(O3)*** avoid data loss approximated as the buffer overflow tendency of RLC queues, i.e., $\min 1 - M_t^u$ and $\min 1 - M_t^d$. We next describe Wixor's NN-based RL agent.
Reward: As shown in Eqn. 1, Wixor's RL agent combines ***O1***, ***O2*** and ***O3*** into a reward function r_t. ***O2*** and ***O3*** create a trade-off, hence, we leverage the buffer tolerance factor ρ_t, previously defined in §5.1, to determine the weight for each objective. $\eta \in [0, 1]$ is Wixor's UL traffic priority. On a high level, the reward increases if the BS throughput is high, and the worst buffering delay is low.

$$r_t = \eta(T_t^u + \rho_t - M_t^u) + (1 - \eta)(T_t^d + \rho_t - M_t^d) \tag{1}$$

State: At each time step t, Wixor's learning agent takes state inputs $s_t = \{\overrightarrow{\mathcal{D}}_{t-k:t}, \overrightarrow{\mathcal{L}}_{t-k:t}, \overrightarrow{C}_{t-k:t}, \rho_t\}$ for its NN. $\overrightarrow{\mathcal{D}}_{t-k:t}$, $\overrightarrow{\mathcal{L}}_{t-k:t}$, and $\overrightarrow{C}_{t-k:t}$ are traffic demand, BS load, and channel quality feature vectors, respectively, for the past k time steps.

[3]The maximum BS data arrival rate (and throughput) can either be computed empirically by saturating BS with over 100% load under optimal channel conditions, or theoretically using subcarrier spacing (μ), channel bandwidth, beamforming parameters, etc. [21]. Wixor uses the latter by default.

Action: Given s_t, the learning agent predicts the UL S&S percentage needed, i.e., $a_t = p_t^u \in [0, 1]$. Note that the sum of UL (p_t^u), DL (p_t^d) and guard period (p_t^g) S&S percentages amounts to one, i.e., $p_t^u + p_t^d + p_t^g = 1$. We explain how to compute p_t^d and p_t^g in §6.2.
RL Model Training: After applying each action, the simulated environment transitions to a new state and provides a reward to the RL agent. The primary goal of the RL agent is to maximize the expected cumulative reward, i.e., $\max \mathbb{E}[\sum_{t=0}^{\infty} r_t]$. Various algorithms can train an RL agent within the abstract RL framework described above (e.g., DQN [60], PPO [64]). We use the soft actor-critic (SAC) algorithm [42] for two primary reasons: (i) it is the state-of-the-art and has been applied successfully to numerous learning problems in networked systems [45, 46, 57]; and (ii) its asynchronous parallel training allows multiple BSs to send their experience feedback to the RL agent, leading to a shorter model convergence time as opposed to other RL algorithms. We present the details of NN architecture and training methodology in Appendix A.1. To adapt to varying traffic patterns and network policies, Wixor temporarily stores BS-level logs in a buffer and updates the model every 15 minutes with a small learning rate (1e-4). Unlike offline simulator training, the RL agent does not perform exploration during runtime.

6 Context-aware Policy Provision

After receiving the predicted UL S&S percentage, Wixor employs a conservative policy smoothing technique to reduce abrupt policy changes (§6.1). Subsequently, it balances the tradeoff between the inter-slot delay and guard period overhead to determine the *best* TDD pattern (§6.2).

6.1 Conservative Policy Smoothing

As noted in §3.3, abrupt TDD policy changes due to fluctuating load can misguide the transport-layer congestion control or application-layer rate adaptation logic. To prevent this, Wixor must tolerate traffic load noise shown in Fig. 3a, while promptly responding to long-term traffic load variations. To achieve this, Wixor applies a conservation policy smoothing technique to the action ($a_t = p_t^u$) generated by the resource forecasting module.

$$\gamma_t = \beta\gamma_{t-1} + (1-\beta)|p_t^u - p_{t-1}^u| \tag{2a}$$

$$\alpha_t = \gamma_b^t \;/\; max(\gamma_{t-t_s\,:\,t}) \tag{2b}$$

$$\hat{p}_t^u = \alpha_t p_t^u + (1-\alpha_t)\hat{p}_{t-1}^u \tag{2c}$$

Eqn. 2c represents the traditional Exponentially Weighted Moving Average (EWMA). Through large-scale simulations, we find that the proper value of weight α_t is not static. Therefore, Eqn. 2a applies another EWMA to smooth out the TDD policy variation γ_t (the weight β is found to be insensitive to the prediction result; we empirically use 0.5), and Eqn. 2b normalizes γ_b^t using a time window $[t - t_s, t]$ where t_s is a large positive multiple of system time step length Δt. Through testing various t_s values, we found that $30\Delta t$ best balances the trade-off between mitigating traffic noise and ensuring prompt responsiveness. Intuitively, a sudden large change in predicted UL S&S percentage (p_t^u) will lead to a large α_t, which makes $\hat{p}_t^u$ rely less on (potentially stale) $\hat{p}_{t-1}^u$.

6.2 QoS-aware TDD Policy Derivation

Once Wixor knows $\hat{p}_t^u$, it must determine the S&S arrangement for the TDD policy $\mathcal{P}_t$ while accounting for guard periods. However, deriving $\mathcal{P}_t$ is not straightforward due to the significant impact of *inter-slot delay* on network latency and application QoE (§3.2). The key challenge lies in balancing the tradeoff between *minimizing inter-slot delay* and *managing guard period overhead.* Lower inter-slot delays reduce network latency but increase DL→UL and UL→DL transitions, leading to higher guard period overhead and, ultimately, reduced throughput. Wixor judiciously balances this tradeoff by finding a TDD pattern with minimum normalized weight between inter-slot delay and guard period overhead.

Calculating guard period. In TDD systems, guard periods are implemented as brief intervals of no transmission between UL and DL symbols to: (i) avoid interference and account for propagation delays, and (ii) allow BS and UE radio hardware time to switch between transmission (Tx) and reception (Rx) modes. The guard period primarily depends on two factors: the maximum BS coverage radius that determines the propagation delay between the BS and the UE, and the switching delay between Rx and Tx modes for both the BS and the UE. Based on these factors, Wixor explicitly calculates the number of guard symbols required for DL→UL transitions ($g^{d,u}$) and UL→DL transitions ($g^{u,d}$). The detailed procedure to compute $g^{d,u}$ and $g^{u,d}$ is described in Appendix A.2.

Computing valid TDD policy set. Given $\hat{p}_t^u$, $g^{d,u}$, and $g^{u,d}$, Wixor can compute all possible arrangements of UL, DL, and guard S&Ss. In order to adhere to 3GPP specifications, which restrict how TDD patterns are defined [20], Wixor generates valid TDD patterns (S&S arrangements) for all suitable transmission periodicities to create a TDD policy set $\mathcal{S}_t$. $\mathcal{S}_t$ typically has a size of 5-25 valid patterns depending on $\hat{p}_t^u$, numerology (μ), and transmission periodicity (τ). For each pattern $s : s \in \mathcal{S}_t$, Wixor also computes: ***(i)*** the guard period overhead $p^{g,s}$ given by the percentage of guard S&Ss in s. Depending on the pattern and BS configuration, the guard periods can result in 0.2-3.1% of wasted bandwidth; and ***(ii)*** the total inter-slot delay d^s for DL→UL and UL→DL transitions.

Finding the best TDD policy. Wixor balances the tradeoff between the guard period overhead and the inter-slot delay. The inter-slot delay dictates the minimum amount of time network packets spend in the per-UE queues waiting to be transmitted. Therefore, the buffering tolerance factor ρ_t introduced earlier (§5.1) can be leveraged to encode our preference for network latency. Wixor then employs ρ_t to compute a normalized weight w^s using Eqn. 3. Lastly, Wixor finds the pattern with minimum normalized weight to get the *best* TDD policy $\mathcal{P}_t$ from $\mathcal{S}_t$ as $\arg\min_{s \in \mathcal{S}_t}(w^s)$.

$$w^s = \rho_t \frac{d^s}{\sum_{s \in \mathcal{S}_t} d^s} + (1 - \rho_t) \frac{p^{g,s}}{\sum_{s \in \mathcal{S}_t} p^{g,s}} \tag{3}$$

Note on design. The TDD policy derivation approach described here is not an optimal choice derived through formal analysis - it is a heuristic, particularly the calculation of buffering tolerance factor ρ_t, in part due to the lack of application-level support or QoE feedback. However: *(i)* it performs well in realistic settings (§8.5), *(ii)* it provides a configurable knob to fine-tune the latency versus throughput tradeoff, and *(iii)* Wixor supports arbitrary ρ_t derivation methods. We implement two approaches: (i) a fixed approach where ρ_t does not change over time, and (ii) a *default* approach where ρ_t is calculated based on the number of latency-sensitive flows in the BS (details in §7).

7 Implementation

Wixor is built on top of srsRAN [13, 16] in over 2.3K lines of C/C++ code. Apart from that, we add support for runtime TDD policy adaptation and multi-layer (MAC, RLC, and PDCP) data logging in srsRAN. We also develop a faithful 5G network simulator based on the *ns*3 5G Lena [5] codebase for large-scale trace-driven simulations. The implementation details are in Appendix §B.

8 Evaluation

Due to the lack of operator-side support and the cost of deploying a commercial BS, most of our experiments are carried out using an in-lab *over-the-air* testbed and a large-scale *ns*3 simulator driven by real-world *traces*, previously described in §3.1. We summarize our main findings here.

- We use a suite of six diverse apps to quantify QoE gains. Compared to baselines, Wixor improves application QoE metrics by 2.5%-96.5%, or is within 91.6% of the best scheme (§8.2 & §8.3).

• Wixor is particularly useful under changing traffic loads and fluctuating channel conditions. For example, compared to the walking scenario, it offers 3.2% and 15.3% higher average throughput and latency improvement for driving (§8.4).
• Wixor respects the radio network's optimization objectives, is scalable, works well under advanced BS configurations, and operates close to the optimal solution (§8.4 & §8.5).

8.1 Experiment Setup

Methodology. Our experiments use 5G numerology μ=1 (30 KHz subcarrier spacing) and the proportional fair MAC scheduler, unless otherwise mentioned. The BS operates with Band 78 @ 3410 MHz and 20 MHz channel bandwidth. The channel quality indicator (CQI) reporting interval is set to 40 ms. We train Wixor with 60% of the (channel and traffic) traces and use the rest for evaluation. The UL and DL priority for Wixor is equal (i.e., η=0.5), and the buffering tolerance factor (ρ_t) is adjusted according to the 5QI method discussed in §7.

Baselines. ***(i)*** *Default:* the default static, DL-heavy TDD policy employed by current 5G networks (i.e., 22.8% UL and 72.8% DL S&Ss); ***(ii)*** *SFair:* a static TDD policy that fairly distributes S&Ss among UL and DL based on the average traffic load of an experiment; ***(iii)*** *Reactive:* a TDD policy that configures S&Ss at time step t according to the previous time step's traffic load; ***(iv)*** *DRP [27]:* a recent RL-based algorithm that derives UL and DL S&S percentage to minimize buffer overflows. To the best our ability, we train *DRP*'s RL agent using BS-level features and parameters described in the paper. Further, we use the same UL and DL priority as Wixor.

Applications and metrics. Apart from the trace-generated background traffic (§3.1), we use six diverse application workloads to generate traffic. ***(i)*** *Edge Video Analytics (EVA)*: We select a popular EVA task, i.e., Object Detection. We use the frame response latency and perceptive accuracy as performance metrics. The perceptive accuracy [40, 49] captures mean average precision for sending frames, and replaces a frame's inference with the last feedback if a response is not received within 40 ms. ***(ii)*** *Edge-assisted Vehicle Perception (EAVP)*: Autonomous vehicles rely on object tracking to ensure safe and robust driving performance. We compute object tracking's frame response latency and the mean Intersection over Union (IoU) metric. IoU measures the overlap between the predicted bounding box of the object and the ground truth bounding box across consecutive frames. ***(iii)*** *Live Video Ingest (LVI)*: We measure the performance in terms of sending bitrate and ingest delay for published video streams. Ingest delay [84] calculates the time elapsed from the video frame generation to the reception of the corresponding segment at the video server for clients to download. ***(iv)*** *Video-on-Demand (VoD)*: We normalize the video bitrate using the maximum bitrate while the stall percentage is the proportion of time the video is stalled during a video playback session. ***(v)*** *Live Video Conferencing (LVC)*: We calculate the average Structural Similarity Index (SSIM) to quantify similarity between sent and received frames for both clients. Likewise, video delay is the average duration between frame reception and transmission times for both clients. ***(vi)*** *HTTP File Transfer (HFT)*: The UE repeatedly uploads/downloads a 128 MB file to/from the application server. We log the total file upload/download time to show results. The detailed setup for each of the six applications is described in Appendix C.1.

8.2 Overall Benefit for the Applications

We conduct extensive simulations to evaluate the performance of Wixor. Our evaluation only utilizes the six applications described earlier (§8.1) to generate user traffic; we do not generate any background traffic for this experiment. Each app has up to 10 instances running concurrently, with each instance running for a maximum of 300 secs. To distribute the traffic temporally, we generate app instance start times using a Poisson random process. Specifically, we determine inter-arrival times for app instances with an arrival rate $\lambda = 10/300 = 0.033$ and convert these to start times.

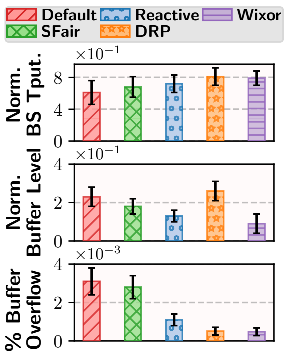

Fig. 9. BS QoS metrics for the simulation experiments in Table 3.

Table 3. Overall QoE improvement for six application workloads and 6 hrs+ of channel traces. The numbers represent the mean and standard deviation. Wixor *outperforms* baselines for most of the metrics. The up (↑) and down (↓) arrows indicate the direction of QoE improvement. The gray shaded region highlights the best performing scheme.

Application	Metric	Default	SFair	Reactive	DRP	Wixor
Edge Video Analytics (EVA)	Response Latency (ms) ↓	93.3±10.6	77.2±9.1	44.9±7.5	57.4±7.1	38.1±6.4
	Perceptive Accuracy (%) ↑	34.7±6.2	40.1±6.1	54.8±7.0	47.6±7.9	68.2±8.5
Edge-assisted Autonomous Vehicle Perception (EAVP)	Response Latency (ms) ↓	67.8±6.5	60.3±6.5	51.7±6.1	56.8±6.2	46.3±5.9
	Mean IoU ↑	0.68±0.1	0.71±0.1	0.77±0.2	0.74±0.1	0.78±0.1
Live Video Ingest (LVI)	Ingest Delay (ms) ↓	284.9±34.7	255.3±28.5	246.4±28.3	233.8±24.0	191.5±22.5
	Sending Bitrate (Mbps) ↑	3.8±1.5	5.5±1.2	5.8±1.3	6.1±1.3	6.2±1.2
Video-on-Demand (VoD) Streaming	Normalized Bitrate ↑	0.83±0.1	0.63±0.2	0.77±0.2	0.80±0.2	0.81±0.2
	Stall Percentage (%) ↓	0.63±0.3	0.11±0.1	0.18±0.1	0.25±0.2	0.12±0.2
Live Video Conferencing (LVC)	Video Quality (SSIM dB) ↑	15.3±1.1	13.1±1.4	14.1±1.5	14.6±1.1	15.7±1.4
	Video Delay (ms) ↓	48.7±5.6	65.9±5.8	48.3±5.7	58.4±6.3	43.9±5.2
HTTP File Transfer (HFT)	Upload Time (s) ↓	562.3±75.7	422.9±64.3	418.4±74.7	397.3±63.9	405.7±61.4
	Download Time (s) ↓	352.8±56.9	379.8±54.2	376.3±58.4	381.6±62.7	385.2±60.3

Each experiment ran for 30 mins, using random channel traces from our *corpus*. We repeated each experiment 5×, selecting different random traces for each run. The overall traffic load for the experiment ranged between 30% and 90%.

Overall QoS improvement. Wixor considers three BS QoS metrics in its overall objective for all applications, i.e., BS throughput, buffer level, and buffer overflows (§5.2). Fig. 9 compares these metrics across all baselines to Wixor. There are two main takeaways: *(i)* Wixor outperforms static TDD policies (*Default* and *SFair*) across all metrics. For instance, it achieves an average 16.1%-29.5% higher BS throughput compared to static schemes. While *Default* provides high DL throughput, its UL performance degrades due to the DL-heavy S&S allocation; and *(ii) DRP* performs similarly to Wixor in terms of BS throughput and buffer overflows. However, *DRP* maintains 1.9× higher average buffer level than Wixor. Since *DRP*'s reward function prioritizes a high buffer level while avoiding overflows, latency-sensitive applications will experience significant performance degradation.

QoE benefits. Table 3 showcases the overall QoE gains Wixor brings for various applications. Our results highlight three main findings: *(i)* Wixor achieves significantly higher performance than the baselines for all latency-sensitive applications (EVA, EAVP, and LVC). For EVA, it achieves an average 24.4%-96.5% higher perceptive accuracy compared to other schemes, while reducing the response latency by 15.1%-59.2%. The reduction in latency can be mainly attributed to Wixor's latency-aware optimization objective (§5.2) and policy derivation mechanism (§6.2). *(ii) Default*, having DL-heavy S&S allocation, performs slightly better than Wixor for the DL bandwidth-intensive applications (VoD, HFT download). As an example, *Default* has an average 2.5% and 3.7% higher VoD bitrate than Wixor and *DRP*, respectively. *(iii)* Wixor and *DRP* offer similar performance for the UL bandwidth-intensive applications (HFT upload, LVI). *To summarize*, Wixor primarily balances UL and DL S&S allocation to promptly provision resources for all application types - it outperforms or is within 91.6% of baselines for all metrics. Note that here we used Wixor's default parameter values (e.g., η=0.5). We later (§8.5) show that Wixor can be easily tuned to prefer certain use cases.

8.3 Over-the-air Evaluation of Wixor

We evaluate Wixor prototype with a combination of application and trace-generated background traffic. We conduct 5 hrs+ of experiments using the over-the-air setup described in §3.1. The background traffic is generated with 15 random UE traces from DL5G.

Dissecting Wixor's performance gains. We *first* test the EVA app that runs on one PX7, with background traffic running on the other. These experiments use a fixed ρ_t (i.e., 0.5). Fig. 10 (left) shows the frame response latency and perceptive accuracy. In our setup, the latency-sensitive EVA requires a frame response latency of less than 40 ms, as indicated by the dotted gray line. To

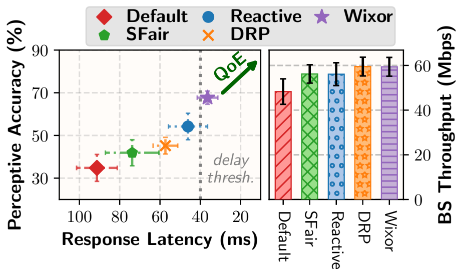

Fig. 10. Comparison of Edge Video Analytics QoE across baselines.

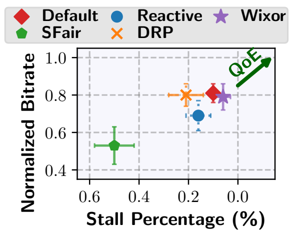

Fig. 11. Comparison of Video-on-Demand streaming QoE across baselines.

Table 4. System overhead of Wixor (65% traffic load).

Metric	*Default*	Wixor	Δ Gap
CPU Utilization (%)	52.4±6.5	60.1±6.8	7.7
Memory Utilization (%)	23.9±2.1	26.2±2.0	2.3

Table 5. Wixor neural network's inference time.

Method	Inference Time (ms)
CPU	13.1±1.9
GPU	6.8±1.3

demonstrate that Wixor maintains fairness for other users, we also plot the overall BS throughput (right) There are four key takeaways. *(i)* Wixor significantly outperforms the baselines across both metrics. For example, it offers an average 24.9%-94.5% higher perceptive accuracy and 21%-60.1% lower response latency than baselines. *(ii)* Unlike *DRP*, Wixor's incorporation of latency objectives in its RL reward (Eqn. 1) minimizes buffering delays for latency-sensitive applications. On average, *DRP* incurs 36.5% higher response latency than our approach, largely due to *DRP*'s higher buffer levels and queuing delays. Although not shown here, *DRP*'s median buffer level is 1.4× larger than Wixor's during the experiment. *(iii)* Static schemes like *Default* and *SFair* cannot adapt to changing traffic loads, leading to the lowest performance in our tests. *(iv)* Interestingly, *Reactive*, by following the UL and DL traffic patterns to reduce buffer levels, outperforms *DRP* in terms of QoE, delivering 19.6% and 19.9% better average response latency and perceptive accuracy, respectively. However, *Reactive* still falls short of Wixor performance due to its reactive nature (§3.2).

Next, we evaluate Wixor with the most ubiquitous form of Internet traffic, i.e., VoD streaming. Our results in Fig. 11 indicate that Wixor, *Default* and *DRP* offer similar throughput performance for DL bandwidth-intensive apps. For instance, Wixor offers 2.5% lower average bitrate than *Default* while reducing average stall up to 40%. *Default*'s high DL performance comes at the cost of lower UL performance as seen earlier in Table 3. *DRP*, on the other hand, performs well for bandwidth-intensive applications, but incurs QoE loss for latency-sensitive applications, as seen above.

Wixor's overhead. We record the CPU overhead and memory consumption of Wixor in Table 4. Compared to *Default*, Wixor increases the absolute CPU and memory utilization by 7.7% and 2.3%, respectively. The overhead primarily comes from Wixor's RL agent's resource forecasting (§5.2), which may slightly rise with the number of users. Although not shown here, the CPU and memory utilization only increases by 3.2% and 1.1%, respectively, when the traffic load increases from 65% to 90%. We also compute RL agent's inference time in Table 5. By default, Wixor performs inference on a CPU which takes only 13 ms on average. The inference time can be further reduced with a GPU, e.g., NVIDIA GeForce RTX 3060 Ti GPU cuts the average inference time to 7 ms (48.1% ↓).

8.4 Wixor under Diverse Settings

We use *ns*3 simulations, with trace-generated background traffic (from 30 random UE traces unless otherwise mentioned), to evaluate how Wixor performs under different settings.

Traffic load. We pick three traffic traces with varying fluctuation levels (standard deviation) from DL5G: ***T1***, ***T2***, and ***T3*** with approximately 60%±10%, 60%±20%, and 60%±30% average traffic load, respectively. Each trace is 10 mins long. Our results in Table 6 highlight that Wixor's TDD policy adapts well to the changing load, while other baselines cannot. For example, the absolute performance gap between Wixor and *Reactive* increases from 3.5% to 8.5% as load variation goes up.

Radio channel quality. Our channel trace *corpus* consists of various mobility scenarios (e.g., walking and driving). The driving scenario sees more channel fluctuations than the walking case – driving has an average SINR of 14±5 dB while walking has 17±3 dB SINR. Although not shown,

Table 6. The average BS throughput gap (Δ, in %) b/w Wixor and baselines.

Trace	Δ SFair	Δ Reactive	Δ DRP
T1	9.1	7.3	3.8
T2	12.0	10.2	5.2
T3	15.2	14.9	6.4

Table 7. Comparing Wixor performance across 5G numerologies.

μ	Reactive		Wixor	
	BS Throughput	Per-packet Latency	BS Throughput	Per-packet Latency
0	52.2±6.3	48.1±7.0	57.5±6.8	40.1±4.9
1	49.8±6.3	47.4±5.9	55.1±6.7	36.7±5.3
2	48.3±6.4	47.6±6.2	53.0±6.7	33.2±5.2

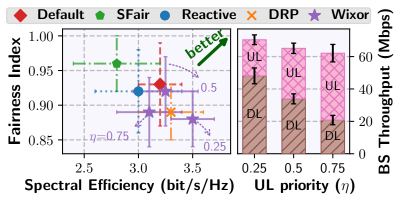

Fig. 12. Wixor's impact on RAN metrics.

Wixor offers higher performance improvement when the radio channel fluctuates frequently due to its use of channel quality features (§5.1). For example, Wixor has 1.4% higher BS throughput and 32.1% lower per-packet latency than *DRP* on average for the walking case. For driving, the average throughput and latency improvement is 4.6% (↑3.2%) and 47.4% (↑15.2%), respectively.
Advanced BS configurations. Our evaluations have used a 5G numerology μ=1 (30 KHz subcarrier spacing) so far, which is the μ used by 5G Mid-Band operators these days [37]. We test Wixor with other numerologies and summarize the results in Table 7. In general, higher μ values lead to more slots per subframe which ultimately increases the number of possible S&S arrangements. This results in lower network latency but slightly reduced BS throughput. Wixor therefore reduces network latency when μ increases (or # of S&S arrangements increases).

8.5 Wixor Deep Dive

RAN objectives. Here, our setup utilizes *ns*3 simulations with trace-generated background traffic only (from 30 random UE traces). Fig. 12 (left) plots the user fairness and spectral efficiency for different UL priority (η) values. To evaluate user fairness, we calculate the Jain's fairness index of the long-term average throughput among users. Spectral efficiency (bit/s/Hz) indicates the amount of information sent through a network using the available bandwidth. The right plot in Fig. 12 shows the distribution of UL and DL BS throughput. Our results provide two key insights: *(i)* Wixor's fairness for the default UL priority (i.e., η=0.5) is within 98.9% of the *Default*. In addition, Wixor improves average spectral efficiency by 1.6% compared to *Default*. *(ii)* The η parameter can be adjusted to fine-tune UL performance. A higher η (0.75) enhances UL performance, but the overall BS throughput and spectral efficiency decrease, as UL typically requires more S&Ss to achieve the same performance as DL (§3.3).
Scalability and application-level fairness. We evaluate Wixor with a large number of users simultaneously performing HTTP File Transfer (HFT). Each user simultaneously downloads and uploads a 128 MB file. Fig. 13 plots the average UL and DL file transfer time as the total number of users grow. When users increase, the file transfer time gradually increases due to limited bandwidth of our BS. However, the increase is almost linear, and the standard deviations are small, suggesting that Wixor also offers application-level fairness in the presence of multi-user competition.
Optimality. Next, we analyze the performance gap between Wixor and an offline optimal solution (*Oracle*). *Oracle* employs dynamic programming to compute the optimal TDD policy. Specifically, we exhaustively search all TDD patterns to find the one that offers the best performance based on Eqns. 1 & 3 (§4). Fig. 14 illustrates Wixor's BS throughput and per-packet latency for different buffering tolerance factors (ρ) and compares it with *Oracle*. Our results highlight two main findings: *(i)* On average, Wixor is within 82.2% and 88.0% of the *Oracle* for per-packet latency and BS throughput, respectively. The performance gap stems from two factors: prediction errors in the forecasting module (§5.2) and performance loss from breaking TDD policy adaptation into sequential steps instead of joint optimization (§4). We explore both factors next. *(ii)* The configurable buffering tolerance factor effectively tradeoffs latency for higher throughput and vice versa.
Prediction accuracy of demand customization engine. We utilize *ns*3 simulations (HFT with 30 users, ρ=0.9) to analyze how well Wixor predicts the UL S&S (p_t^u). Note that we use a high ρ

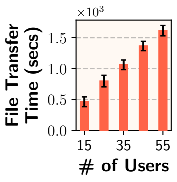

Fig. 13. System scalability under multiple users.

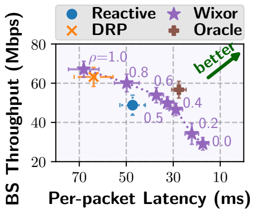

Fig. 14. Perf. gap b/w Wixor and *Oracle*.

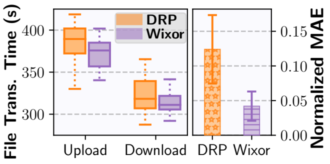

Fig. 15. Comparing the prediction accuracy of demand customization engine with *DRP*.

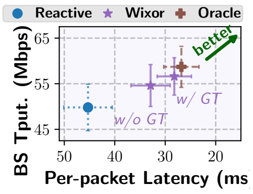

Fig. 16. Evaluating the slot derivation module.

value here to tradeoff latency for higher throughput. The left plot in Fig. 15 quantifies file download and upload times while the right plot shows the Mean Absolute Error (MAE) between p_t^u and the ground truth UL traffic load. Compared to *DRP*, Wixor reduces the median file transfer time by 2.3%-3.5%. Wixor's higher performance can be attributed to its use of cross-layer BS-level features (§5.1) and careful learning agent design (§6). Overall, Wixor leads to 66.1% lower average MAE than *DRP*.

Contribution of policy derivation module. We investigate if Wixor's policy derivation module (§6.2) effectively finds the *best* TDD policy. Again, we use *ns*3 simulations with trace-generated background traffic from 30 random UE traces in DL5G. We leverage the ground truth (GT) UL traffic load instead of the predicted UL S&S percentage p_t^u for a fair comparison (*w/ GT*). Our results in Fig. 16 depict that Wixor operates close (3.6%-7.6% gap depending on the metric) to the *Oracle* when it uses ground truth (*w/ GT*) UL S&S percentage. The gap between Wixor and *Oracle* increases slightly (7.0%-25.6%) for the *w/o GT* case due to the p_t^u prediction error.

Micro-benchmarking. Lastly, we evaluate the conservative policy smoothing (CPS) technique (§6.1), compare Wixor with other RL schemes, conduct a comprehensive parameter sweep of the NN, and profile Wixor's training time in Appendix C.2. We summarize key takeaways as follow. *(i)* CPS effectively mitigates the impact of abrupt TDD policy changes on application/transport layer rate adaptation modules. For instance, Wixor results in 5.9% lower TCP RTT and 14.7% higher throughput, on average, compared to a setting where CPS is disabled. *(ii)* Tabular RL schemes are unable to capture the complexities of the RAN environment; Wixor delivers 27.9% higher average reward compared to them. *(iii)* A NN architecture with a single hidden layer and 64 neurons and 1D-CNN filters performs the best. *(iv)* Wixor's overall training time is ~3.5 hrs.

9 Discussion & Conclusion

Limitations and future work. We acknowledge several limitations of our work. *First,* Wixor does not consider cross-link interference between BSs in its solution, an area for future exploration. *Second,* we evaluated Wixor only with 5G Mid-Band, not mmWave, although the same dynamic TDD policy adaptation principle should apply. *Fourth,* Wixor's RL agent is trained with a simulator and then transferred to the over-the-air setup; continuous online RL training remains a future goal. *In addition,* our solution requires 5QI information to compute the buffering tolerance ρ_t. While feasible for private 5G, public 5G networks could use alternatives like a fixed ρ_t (§6.2). *Lastly,* integrating Wixor with new software-defined 5G paradigms, such as network slicing [83] and OpenRAN's Radio Intelligent Controller [4], is a promising direction for future work.

Despite these limitations, we present a dynamic TDD policy adaptation solution for 5G/xG that effectively balances the UL and DL resources. We implement and evaluate Wixor through simulations and over-the-air experiments, demonstrating significant performance benefits. Furthermore, Wixor's standard-compatible design is friendly to both public and private 5G deployments.

Acknowledgments

We would like to thank our shepherd and the anonymous CoNEXT reviewers for their insightful feedback. This research was supported in part by NSF under Grants 2128489 and 2409267.

References

[1] 2022. *5G synchronization requirements and solutions.* Retrieved June 2024 from https://www.ericsson.com/en/reports-and-papers/ericsson-technology-review/articles/5g-synchronization-requirements-and-solutions

[2] 2023. *5G/NR - tdd UL/DL Common Configuration.* Retrieved June 2024 from https://www.sharetechnote.com/html/5G/5G_tdd_UL_DL_configurationCommon.html

[3] 2023. *CBRS for Private 5G.* Retrieved June 2024 from https://www.arubanetworks.com/faq/what-is-cbrs/

[4] 2023. *O-RAN Architecture.* Retrieved June 2024 from https://docs.o-ran-sc.org/en/latest/architecture/architecture.html

[5] 2024. *5G-LENA: ns-3 module to simulate 5G NR networks.* Retrieved June 2024 from https://apps.nsnam.org/app/nr/

[6] 2024. *5G NR Frequency Bands.* Retrieved June 2024 from https://en.wikipedia.org/wiki/5G_NR_frequency_bands

[7] 2024. *Ant Media: liveVideoBroadcaster.* Retrieved June 2024 from https://github.com/ant-media/LiveVideoBroadcaster

[8] 2024. *dash.js: Open Source Media Player.* Retrieved June 2024 from https://dashjs.org/

[9] 2024. *An end-to-end platform for machine learning.* Retrieved June 2024 from https://www.tensorflow.org/

[10] 2024. *FDD LTE frequency bands.* Retrieved June 2024 from https://www.4g-lte.net/about/lte-frequency-bands/fdd/.

[11] 2024. *High-Quality 5G Networks Bring the World Faster to the 5.5G Era.* Retrieved June 2024 from https://www.huawei.com/en/news/2024/2/5g-high-quality-network-5g-a#:~:text=Multi%2Dcarrier%20networks%20are%20becoming,all%20now%20multi%2Dcarrier%20capable.

[12] 2024. *Open Source 5G Implementation.* Retrieved June 2024 from https://open5gs.org/.

[13] 2024. *Open Source RAN.* Retrieved June 2024 from https://github.com/srsRAN

[14] 2024. *Real-time communication for the web.* Retrieved June 2024 from https://webrtc.org/

[15] 2024. *Serving Models.* Retrieved June 2024 from https://www.tensorflow.org/tfx/guide/serving

[16] 2024. *srsRAN: A customisable solution for Private Enterprise 5G.* Retrieved June 2024 from https://srs.io/srsran-enterprise-5g/

[17] 2024. *Understanding RTMP (Real-Time Messaging Protocol) for Seamless Streaming.* Retrieved June 2024 from https://medium.com/@usamawizard/understanding-rtmp-real-time-messaging-protocol-for-seamless-streaming-7d7d963ba0ef

[18] 2024. *USRP B210 SDR Kit.* Retrieved June 2024 from https://www.ettus.com/all-products/ub210-kit/

[19] 2024. *XCAL: PC based Advanced 5G Network Optimization Solution.* Retrieved June 2024 from https://www.accuver.com/products/network-optimization/XCAL

[20] 3GPP. 2019. *5G; NR; Physical layer procedures for control.* Technical Specification (TS) 38.306. 3rd Generation Partnership Project (3GPP). https://www.etsi.org/deliver/etsi_ts/138200_138299/138213/15.06.00_60/ts_138213v150600p.pdf Version 15.6.0.

[21] 3GPP. 2024. *NR; User Equipment (UE) radio access capabilities.* Technical Specification (TS) 38.306. 3rd Generation Partnership Project (3GPP). https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3193 Version 18.1.0.

[22] Qing An, Santiago Segarra, Chris Dick, Ashutosh Sabharwal, and Rahman Doost-Mohammady. 2023. A Deep Reinforcement Learning-Based Resource Scheduler for Massive MIMO Networks. *IEEE Transactions on Machine Learning in Communications and Networking* (2023).

[23] Guilherme H Apostolo, Pablo Bauszat, Vinod Nigade, Henri E Bal, and Lin Wang. 2022. Live video analytics as a service. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems.* 37–44.

[24] Kelvin Au, Liqing Zhang, Hosein Nikopour, Eric Yi, Alireza Bayesteh, Usa Vilaipornsawai, Jianglei Ma, and Peiying Zhu. 2014. Uplink contention based SCMA for 5G radio access. In *2014 IEEE Globecom Workshops (GC Wkshps).*

[25] Jose A. Ayala-Romero, Andres Garcia-Saavedra, Xavier Costa-Perez, and George Iosifidis. 2021. EdgeBOL: automating energy-savings for mobile edge AI. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '21).*

[26] Duin Baek, Mallesham Dasari, Samir R Das, and Jihoon Ryoo. 2021. dcSR: practical video quality enhancement using data-centric super resolution. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies.* 336–343.

[27] Miloud Bagaa, Karim Boutiba, and Adlen Ksentini. 2021. On using Deep Reinforcement Learning to dynamically derive 5G New Radio TDD pattern. In *2021 IEEE Global Communications Conference (GLOBECOM).*

[28] Giovanni Bartolomeo, Jacky Cao, Xiang Su, and Nitinder Mohan. 2023. Characterizing distributed mobile augmented reality applications at the edge. In *Companion of the 19th International Conference on emerging Networking EXperiments and Technologies.* 9–18.

[29] Gilberto Berardinelli, Klaus I. Pedersen, Frank Frederiksen, and Preben Mogensen. 2016. On the Guard Period Design in 5G TDD Wide Area. In *2016 IEEE 83rd Vehicular Technology Conference (VTC Spring).* 1–5. https://doi.org/10.1109/VTCSpring.2016.7504377

[30] Leonardo Bonati, Michele Polese, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. 2020. Open, Programmable, and Virtualized 5G Networks: State-of-the-Art and the Road Ahead. *Computer Networks* (2020).

[31] Karim Boutiba, Miloud Bagaa, and Adlen Ksentini. 2023. On enabling 5G Dynamic TDD by leveraging Deep Reinforcement Learning and O-RAN. In *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*.

[32] Karim Boutiba, Miloud Bagaa, and Adlen Ksentini. 2024. Multi-Agent Deep Reinforcement Learning to Enable Dynamic TDD in a Multi-Cell Environment. *IEEE Transactions on Mobile Computing* (2024).

[33] Yulong Chen, Junchen Guo, Yimiao Sun, Haipeng Yao, Yunhao Liu, and Yuan He. 2024. ELASE: Enabling Real-time Elastic Sensing Resource Scheduling in 5G vRAN. In *IEEE/ACM International Symposium on Quality of Service (IWQoS)*.

[34] Yongzhou Chen, Ruihao Yao, Haitham Hassanieh, and Radhika Mittal. 2023. Channel-Aware 5G RAN Slicing with Customizable Schedulers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*.

[35] Ming Ding, David López Pérez, Athanasios V. Vasilakos, and Wen Chen. 2014. Dynamic TDD transmissions in homogeneous small cell networks. In *2014 IEEE International Conference on Communications Workshops (ICC)*.

[36] Hisham Elshaer, Federico Boccardi, Mischa Dohler, and Ralf Irmer. 2014. Downlink and uplink decoupling: A disruptive architectural design for 5G networks. In *2014 IEEE global communications conference (GLOBECOM)*. IEEE.

[37] Rostand A. K. Fezeu, Jason Carpenter, Claudio Fiandrino, Eman Ramadan, Wei Ye, Joerg Widmer, Feng Qian, and Zhi-Li Zhang. 2023. Mid-Band 5G: A Measurement Study in Europe and US. arXiv:2310.11000

[38] Piotr Gawłowicz and Anatolij Zubow. 2019. ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*.

[39] Jim Gettys and Kathleen Nichols. 2011. Bufferbloat: Dark Buffers in the Internet: Networks without effective AQM may again be vulnerable to congestion collapse. *ACM Queue* (2011).

[40] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Matthew A Wright, Joseph E Gonzalez, and Ion Stoica. 2021. Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*.

[41] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review* 42, 5 (2008), 64–74.

[42] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *CoRR* abs/1801.01290 (2018). arXiv:1801.01290 http://arxiv.org/abs/1801.01290

[43] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*. PMLR.

[44] Ahmad Hassan, Arvind Narayanan, Anlan Zhang, Wei Ye, Ruiyang Zhu, Shuowei Jin, Jason Carpenter, Z. Morley Mao, Feng Qian, and Zhi-Li Zhang. 2022. Vivisecting mobility management in 5G cellular networks. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 86–100. https://doi.org/10.1145/3544216.3544217

[45] Jiyao Hu, Zhenyu Zhou, Xiaowei Yang, Jacob Malone, and Jonathan W Williams. 2020. CableMon: Improving the Reliability of Cable Broadband Networks via Proactive Network Maintenance. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 619–632. https://www.usenix.org/conference/nsdi20/presentation/hu-jiyao

[46] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 3050–3059. https://proceedings.mlr.press/v97/jay19a.html

[47] Jaehong Kim, Yunheon Lee, Hwijoon Lim, Youngmok Jung, Song Min Kim, and Dongsu Han. 2022. OutRAN: co-optimizing for flow completion time in radio access network. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '22)*.

[48] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[49] Mengtian Li, Yu-Xiong Wang, and Deva Ramanan. 2020. Towards Streaming Perception. In *Computer Vision – ECCV 2020*.

[50] Zhi Li, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali C Begen, and David Oran. 2014. Probe and adapt: Rate adaptation for HTTP video streaming at scale. *IEEE journal on selected areas in communications* 32, 4 (2014), 719–733.

[51] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. 2018. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 751–766.

[52] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 740–755.

[53] Peng Liu, Bozhao Qi, and Suman Banerjee. 2018. Edgeeye: An edge service framework for real-time intelligent video analytics. In *Proceedings of the 1st international workshop on edge systems, analytics and networking*. 1–6.

[54] M. Carmen Lucas-Estañ and J. Gozalvez. 2022. Sensing-Based Grant-Free Scheduling for Ultra Reliable Low Latency and Deterministic Beyond 5G Networks. *IEEE Transactions on Vehicular Technology* (2022).

[55] Jiamei Lv, Yi Gao, Zhi Ding, Yuxiang Lin, Xinyun You, Guang Yang, and Wei Dong. 2024. Providing UE-level QoS Support by Joint Scheduling and Orchestration for 5G vRAN. In *IEEE International Conference on Computer Communications (INFOCOM)*.

[56] Basma Mahdy, Hazem Abbas, Hossam S. Hassanein, Aboelmagd Noureldin, and Hatem Abou-zeid. 2020. A Clustering-Driven Approach to Predict the Traffic Load of Mobile Networks for the Analysis of Base Stations Deployment. *Journal of Sensor and Actuator Networks* (2020).

[57] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 197–210. https://doi.org/10.1145/3098822.3098843

[58] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. 2022. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*.

[59] Debashisha Mishra, P C Amogh, Arun Ramamurthy, A Antony Franklin, and Bheemarjuna Reddy Tamma. 2016. Load-aware dynamic RRH assignment in Cloud Radio Access Networks. In *2016 IEEE Wireless Communications and Networking Conference*.

[60] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[61] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, Feng Qian, and Zhi-Li Zhang. 2021. A variegated look at 5G in the wild: performance, power, and QoE implications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 610–625. https://doi.org/10.1145/3452296.3472923

[62] Nidhi, Albena Mihovska, and Ramjee Prasad. 2020. Overview of 5G New Radio and Carrier Aggregation: 5G and Beyond Networks. In *2020 23rd International Symposium on Wireless Personal Multimedia Communications (WPMC)*.

[63] Jaeeun Park, Joohyung Lee, Daejin Kim, and Jun Kyun Choi. 2024. Deep Reinforcement Learning Driven Joint Dynamic TDD and RRC Connection Management Scheme in Massive IoT Networks. *IEEE Access* (2024).

[64] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[65] William Sentosa, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Haitham Hassanieh, and Bruce Maggs. 2023. DChannel: Accelerating Mobile Applications With Parallel High-bandwidth and Low-latency Channels. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*.

[66] Chandan Kumar Sheemar, Leonardo Badia, and Stefano Tomasin. 2021. Game-Theoretic Mode Scheduling for Dynamic TDD in 5G Systems. *IEEE Communications Letters* 25, 7 (2021), 2425–2429. https://doi.org/10.1109/LCOMM.2021.3073908

[67] Jing Song, Qingyang Song, Ya Kang, Lei Guo, and Abbas Jamalipour. 2022. QoE-Driven Distributed Resource Optimization for Mixed Reality in Dynamic TDD Systems. *IEEE Transactions on Communications* 70, 11 (2022), 7294–7306. https://doi.org/10.1109/TCOMM.2022.3208113

[68] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. 2020. BOLA: Near-optimal bitrate adaptation for online videos. *IEEE/ACM transactions on networking* 28, 4 (2020), 1698–1711.

[69] Hongguang Sun, Matthias Wildemeersch, Min Sheng, and Tony Q. S. Quek. 2015. D2D Enhanced Heterogeneous Cellular Networks With Dynamic TDD. *IEEE Transactions on Wireless Communications* (2015).

[70] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. 2020. Scalability in Perception for Autonomous Driving: Waymo Open Dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[71] Zhaowei Tan, Jinghao Zhao, Yuanjie Li, Yifei Xu, and Songwu Lu. 2021. Device-Based LTE Latency Reduction at the Application Layer. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*.

[72] Fengxiao Tang, Yibo Zhou, and Nei Kato. 2020. Deep Reinforcement Learning for Dynamic Uplink/Downlink Resource Allocation in High Mobility 5G HetNet. *IEEE Journal on Selected Areas in Communications* (2020).

[73] Van Dat Tuong, Nhu-Ngoc Dao, Wonjong Noh, and Sungrae Cho. 2021. Deep Reinforcement Learning-Based Hierarchical Time Division Duplexing Control for Dense Wireless and Mobile Networks. *IEEE Transactions on Wireless*

Communications (2021).

[74] Kuna Venkateswararao and Pravati Swain. 2020. Traffic aware sleeping strategies for Small-Cell Base Station in the Ultra dense 5G Small Cell Networks. In *2020 IEEE REGION 10 CONFERENCE (TENCON)*.

[75] Andressa Vergutz, Guevara Noubir, and Michele Nogueira. 2020. Reliability for smart healthcare: A network slicing perspective. *IEEE Network* 34, 4 (2020), 91–97.

[76] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. 2023. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 7464–7475.

[77] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 459–471. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/winstein

[78] Yaxiong Xie and Kyle Jamieson. 2022. Ng-scope: Fine-grained telemetry for nextg cellular networks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–26.

[79] Dongzhu Xu, Anfu Zhou, Guixian Wang, Huanhuan Zhang, Xiangyu Li, Jialiang Pei, and Huadong Ma. 2022. Tutti: coupling 5G RAN and mobile edge computing for latency-critical video analytics. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking (MobiCom '22)*.

[80] Yinda Xu, Zeyu Wang, Zuoxin Li, Ye Yuan, and Gang Yu. 2020. SiamFC++: Towards Robust and Accurate Visual Tracking with Target Estimation Guidelines. arXiv:1911.06188 [cs.CV]

[81] Mu Yan, Gang Feng, Jianhong Zhou, Yao Sun, and Ying-Chang Liang. 2019. Intelligent Resource Scheduling for 5G Radio Access Network Slicing. *IEEE Transactions on Vehicular Technology* (2019).

[82] Bo Yu, Liuqing Yang, Hiroyuki Ishii, and Sayandev Mukherjee. 2015. Dynamic TDD Support in Macrocell-Assisted Small Cell Architecture. *IEEE Journal on Selected Areas in Communications* (2015).

[83] Shunliang Zhang. 2019. An Overview of Network Slicing for 5G. *IEEE Wireless Communications* (2019).

[84] Xiao Zhu, Subhabrata Sen, and Z. Morley Mao. 2021. Livelyzer: analyzing the first-mile ingest performance of live video streaming. In *Proceedings of the 12th ACM Multimedia Systems Conference (MMSys '21)*.

[85] Anna Łukowa and Venkatkumar Venkatasubramanian. 2019. Centralized UL/DL Resource Allocation for Flexible TDD Systems With Interference Cancellation. *IEEE Transactions on Vehicular Technology* 68, 3 (2019), 2443–2458. https://doi.org/10.1109/TVT.2019.2893061

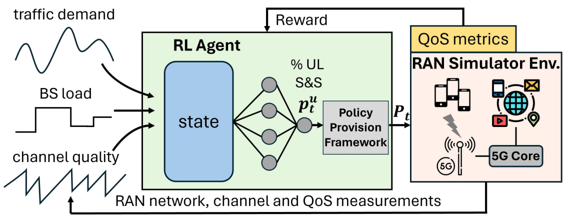

Fig. 17. An overview of how the context-aware resource forecasting module applies RL to the TDD policy adjustment problem. The simulated environment generates RAN state, which is fed to the RL agent. After executing the agent-generated action, the environment transitions to a new a state and outputs the reward.

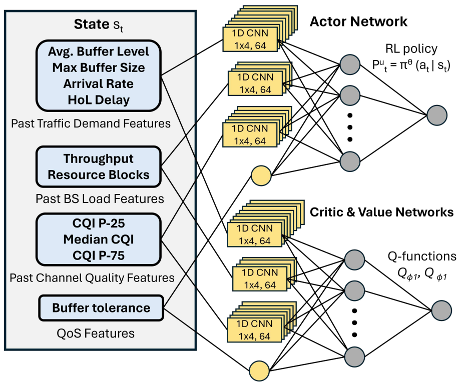

Fig. 18. The NN and Soft Actor-Critic algorithm that Wixor uses for resource forecasting.

A Design & Implementation Details

A.1 RL Agent Architecture and Training

RL Policy: Wixor's RL agent outputs action a_t based on an *RL policy*, defined as the conditional probability distribution over state π: $\pi(a_t|s_t) \in [0, 1]$. $\pi(a_t|s_t)$ is the probability of action a_t given state s_t. In practice, there are intractably many {state, action} pairs, e.g., buffer level, and throughput estimates are continuous real numbers. To address this, Wixor employs a neural network (NN) to model π with a feasible number of trainable parameters θ. The policy is thus expressed as $\pi_\theta(a_t|s_t)$. The *actor network* in Fig. 18 depicts how Wixor uses a NN to represent the RL policy.

RL algorithm: The SAC algorithm used by Wixor to train its RL policy is a *policy gradient method* [43]. The key idea in policy gradient methods is to estimate the gradient of the expected total reward by observing the trajectories of executions obtained by following the RL policy. A central feature of SAC is entropy regularization: the policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. The entropy regularization term encourages exploration, i.e., the RL agent discovers and learns about the environment by trying out different random actions. As illustrated in Fig. 18, SAC concurrently learns a policy π_θ (actor network) and two Q-functions $Q_{\phi 1}, Q_{\phi 2}$ (critic and value networks). The Q-function, denoted as $Q(s_t, a_t)$ represents the expected return (total accumulated reward) starting from state s_t, taking action a_t, and subsequently following a policy π. It is important to note that the critic and value networks merely help to train the actor network: post-training, only the actor network is required to generate actions.

Model Training: After applying each action, the simulated environment provides the learning agent with a reward r_t, as highlighted in Fig. 17. The RL agent continually performs gradient descent to improve the RL policy. To further improve and accelerate training, Wixor launches multiple RL agents to operate concurrently. By default, we employ 8 parallel agents. Each agent is set up with different input parameters (e.g., channel traces and traffic workloads). These agents continuously transmit their {state, action, reward} tuples to a central agent, which aggregates the data to create a unified model. For each received sequence of tuples, the central agent employs the SAC algorithm to compute policy gradients and perform gradient descent. Subsequently, the central agent updates the actor network and distributes the updated model to the corresponding agent that sent the tuple. This process occurs asynchronously among all agents, eliminating the need for a locking mechanism between them.

NN implementation. As shown in Fig. 18, for each input type, we use a proper embedding method to extract the underlying features. Specifically, for each traffic demand, BS load, and channel quality

Table 8. 5G QoS Identifier (5QI) for each application used in evaluation (§8).

Application	5QI	Latency-sensitive	Application	5QI	Latency-sensitive
EVA	7	Yes	LVC	80	Yes
EAVP	84	Yes	HFT	6	No
LVI	71	No	Background Traffic	6	No
VoD	6	No			

feature, we first leverage a single 1D-CNN layer with kernel=4, channels=64, stride=1 to extract corresponding features to a 64-dim layer. Meanwhile, we utilize a fully connected layer to extract useful characteristics for the QoS feature. The selected features are then passed into another fully connected layer and outputs a 64-dims vectors. Finally, the output of the actor network is a single neuron, which represents the percentage of UL S&S. We utilize *ReLU* as the active function for each feature extraction layer and leverage *sigmoid* for the last layer. We use TensorFlow [9] to construct the NN architecture and TensorFlow Serving [15] to containerize, deploy, and manage the NN models. Wixor's RL agent takes the past sequence length $k = 8$ into the NN. We set the learning rate to $1e^{-3}$ and use the Adam optimizer [48] to train the model. The batch size is 64 by default.

A.2 Guard Period Calculation

Eqns. 4a and 4b calculate the number of guard symbols required for DL⟶UL transition $g^{d,u}$ and UL⟶DL transition $g^{u,d}$, respectively. First, Wixor determines the symbol duration Δs (Eqn. 4c) according to the BS numerology μ and cyclic prefix length Δcp needed to combat inter-symbol interference. To get $g^{d,u}$, Wixor adds two terms as noted in Eqn. 4a: (i) the propagation delay given by $2 \times \mathcal{R}/c$, where $\mathcal{R}$ is the maximum coverage radius of the BS and c is the speed of light, and (ii) UE's hardware delay when switching from Rx to Tx mode $\Delta_{ue}^{Tx,Rx}$. Finally, it divides the sum of two terms by the symbol duration to get guard symbols required for DL⟶UL transition. On the other hand, the calculation of $g^{u,d}$ does not consider the propagation delay as the BS already sends the timing advance to ensure that UL transmissions from all UEs are synchronized when received by the BS [1, 2]. Therefore, Eqn. 4b simply divides BS's Rx to Tx switching delay $\Delta_{bs}^{Rx,Tx}$ with the symbol duration to get $g^{u,d}$. $\Delta_{bs}^{Rx,Tx}$, in turn, depends on BS's timing advance offset ($N_{TA,Offset} \cdot T_c$), where $N_{TA,Offset}$ is the reference point for the UE's initial transmission and T_c is simply BS's time unit (0.509 ns). For our typical BS configuration (μ=1, $\mathcal{R}$=100m, Δcp=2.34us), the calculations suggest using at least 2 guard symbols for DL⟶UL transition and 1 guard symbol for UL⟶DL transition.

$$g^{d,u} = \left\lceil (2 \times \mathcal{R}/c + \Delta_{ue}^{Rx,Tx})/\Delta s \right\rceil \tag{4a}$$

$$g^{u,d} = \left\lceil \Delta_{bs}^{Rx,Tx}/\Delta s \right\rceil = \left\lceil N_{TA,Offset} \cdot T_c/\Delta s \right\rceil \tag{4b}$$

$$\Delta s = \frac{1}{2^{\mu} \times 15KHz} + \Delta cp \tag{4c}$$

A.3 5G QoS Identifier (5QI)

Table 8 presents the 5QI values for all applications. These values are used in the calculation of buffering tolerance factor ρ_t (§7). For simulation experiments, each app sets up its data bearers with the corresponding 5QI value. The PX7 phone however lacks the ability to configure 5QI, therefore, we use a fixed value ($\rho_t = 0.5$) in our over-the-air testbed evaluations (§8.3).

B Wixor Implementation

Wixor prototype. Wixor is built on top of srsRAN [13, 16], an open-source 5G software defined radio suite. We modified the user plane protocol stack (5G Layer 2) in srsRAN to implement Wixor in over 2.3K lines of C/C++ code. First, we added support for dynamic TDD, enabling runtime TDD policy adaptation. Further, we implemented necessary logging functionality for the PDCP, RLC,

and MAC layers to support feature engineering. We developed a modular TDD policy adaptation engine atop the TDD MAC scheduler, capable of supporting any TDD policy out of the box. The engine receives BS logs at (configurable) periodic intervals and includes a callback function to change the TDD policy. Wixor is implemented as a derived class of this modular engine. It processes BS logs (to be described next) to create BS-level features (§5.1), which are then fed to Wixor's RL agent (§5.2). Deployed with TensorFlow Serving [15], the RL agent outputs the UL S&S percentage, which is post-processed using the conservative policy smoothing technique (§6.1). Wixor then derives the best TDD pattern using the UL S&S percentage and guard period information (§6.2). If the newly computed TDD pattern differs from the current one, Wixor waits until the next transmission period to execute the TDD policy via the callback. Once triggered, the modular TDD policy adaptation engine simply updates BS's objects and data structures that maintain the TDD policy information. We believe that this deployment is practical, given that cellular networks, including BSs are becoming open and programmable [30].
Faithful simulator. We developed a faithful 5G network simulator based on the *ns*3 5G Lena [5] codebase. Wixor's simulator proof-of-concept essentially mirrors the over-the-air prototype's implementation, including support for dynamic TDD and a modular TDD policy adaptation engine. Additionally, we integrated trace-driven channel simulations and implemented several application traffic workload generators (§3.1). Lastly, we used the ns3-gym toolkit [38] along with Tensorflow [9] to train the RL models. Overall, we added or modified 4.2K+ lines of C/C++ and Python code.
Data collection for feature engineering. Recall from §5.1 that Wixor uses several features to understand the RAN context. Specifically, it obtains the DL buffer occupancy $b_t^{d,i}$ for UE i from the RLC per-UE queues and $b_t^{u,i}$ from quantized buffer status reports (BSRs) via the MAC control element. Inter-arrival times $\lambda_t^{u,i}$ and $\lambda_t^{d,i}$ are calculated using RLC service data units (SDUs) arriving in the UL and DL RLC queues, respectively. Wixor also computes $\hat{s}_t^{d,i}$ and $\hat{s}_t^{u,i}$ as the average SDU packet sizes from these queues. The DL HoL delay $h_t^{d,i}$ represents the time spent by the first SDU packet in the DL RLC queue. Besides, $h_t^{u,i}$ is estimated as $(b_{t-1}^{u,i} - t_t^{u,i} \cdot \Delta t)/\hat{s}_t^{u,i}$, where $t_t^{u,i}$ ($t_t^{d,i}$) are UE's UL (DL) throughput from the PDCP layer, and Δt is the system time step. Resource utilizations $r_t^{d,i}$ and $r_t^{u,i}$ are derived from the MAC scheduler layer. DL channel quality $c_t^{d,i}$ comes from CQI reports, while BS directly measures $c_t^{u,i}$ for the UL channel. In Wixor, the buffering tolerance factor ρ_t can be either fixed or computed using 5G QoS identifier (5QI), with the default being the latter. ρ_t is computed as the ratio of latency-sensitive flows in the BS. Appendix A.3 details our use of 5QI to identify latency-sensitive flows for tested applications (§3.1).

C Evaluation Details & Supplementary Results

C.1 Application Setup

(i) *Edge Video Analytics (EVA)*: We select a popular EVA task, i.e., Object Detection. The EVA app uses a state-of-the-art video analytics model (i.e., YOLOv7 [76]) deployed on the edge server (§3.1). Instead of sending camera feeds, a UE streams video frames from the COCO dataset [52] at 30 FPS. ***(ii)*** *Edge-assisted Vehicle Perception (EAVP)*: Autonomous vehicles rely on object tracking to ensure safe and robust driving performance. Using siamFC++ model [80], we set up an EAVP app on the edge server for multiple object tracking. The UE transmits five camera feeds (front and sides) at 30 FPS using the Waymo Open Dataset [70]. ***(iii)*** *Live Video Ingest (LVI)*: We re-purpose Ant-Media's LiveVideoBroadcaster [7] to publish a pre-recorded video stream (1080p @ 30 FPS with 6.5 Mbps average bitrate). The UEs send adaptive RTMP feeds [17] to an Ant Media server [7] deployed on the application server (§3.1). ***(iv)*** *Video-on-Demand (VoD)*: Our VoD streaming experiments use a dash.js [8] player to stream a 4 min video. We mainly test buffer-based BOLA [68] and rate-based [50]

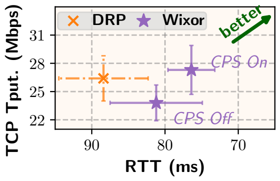

Fig. 19. Evaluating conservative policy smoothing.

Table 9. Comparing different RL schemes.

RL Scheme	Average test reward r_t
Wixor's SAC	0.93±0.22
DQN	0.67±0.28
PPO	0.90±0.21

Table 10. Varying number of neurons.

# of neurons and filters for each 1D-CNN unit	Average test reward r_t
32	0.82±0.28
64	0.93±0.22
128	0.94±0.18
256	0.94±0.14

Table 11. Varying number of hidden layers.

# of hidden layers	Average test reward r_t
1	0.93±0.22
2	0.93±0.28
3	0.89±0.26
6	0.81±0.34

adaptive bitrate (ABR) algorithms due to their popularity. The video is encoded at 6 unique quality levels with average bitrates ranging from 0.8 Mbps to 6 Mbps. ***(v)** Live Video Conferencing (LVC)*: We implement a peer-to-peer LVC app based on WebRTC [14], a real-time video communication framework. Instead of using the video camera, the LVC app streams a 1280 × 780 pre-recorded meeting video at 30 FPS. ***(vi)** HTTP File Transfer (HFT)*: The UE repeatedly uploads/downloads a 128 MB file to/from the application server. We log the total file download/upload time to show results.

C.2 Micro-benchmarking

Benefit of conservation policy smoothing. Recall from §3.2 that abrupt TDD policy changes can mislead rate adaptation modules and result in lost performance. We repeat the same TCP experiment (Fig. 6) to see how well the conservative policy smoothing (CPS) module addresses the issue. Fig. 19 compares the TCP throughput and round-trip-time (RTT) for two cases: CPS *enabled* (on) and *disabled* (off). Compared to the case when CPS is *disabled*, the CPS *enabled* setting results in 5.9% lower average RTT and 14.7% higher average throughput. In addition, the CPS *enabled* setting reduces the RTT variance caused by TDD policy changes.

Comparison with RL schemes. While we employ the Soft-Actor-Critic (SAC) algorithm to train Wixor's RL agent, a variety of algorithms can be utilized within the abstract RL framework described in §5.2. Here, we compare Wixor with Deep Q-Network (DQN [60]) and Proximal Policy Optimization (PPO [64]). DQN is a "tabular" q-learning method that represents the RL policy as a table with discrete entries for all state-action pairs, whereas PPO is a recent policy gradient method. We train PPO in the same way as Wixor while DQN uses fine-grained state and action space quantization. Table 9 presents the average QoS reward r_t (Eqn. 1) attained by each method on the test traces. The results indicate a substantial performance disparity (27.9%) between the tabular scheme and Wixor, underscoring the inadequacy of tabular RL schemes in capturing the complexities of the RAN environment. Conversely, PPO demonstrates performance comparable to Wixor's SAC method, with only a 3.2% gap.

NN architecture. We conduct a comprehensive parameter sweep to evaluate the impact of various NN parameters on r_t. Tables 10 and 11 present the average test reward corresponding to different numbers of neurons and hidden layers, respectively. Our findings indicate that performance plateaus once the number of filters in the 1D-CNN and the number of neurons each exceed 64. Additionally, the results reveal that the NN with a single hidden layer yields the best performance.

Training time. We quantify the overhead associated with training Wixor's RL agent. The training process encompassed approximately 300,000 iterations, equivalent to 3.5 hours of runtime. Each iteration required 42 milliseconds and involved concurrent parameter updates for 8 agents. It is important to note that this overhead represents a one-time, offline computational cost.

Received June 2024; revised September 2024; accepted October 2024